TELESCOPE PARALLEL ACTUATOR MOUNT:

Probably needs a good subtitle

I have confidence that it will soon be supplied with one.

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

Samuel Steejans Artho-Bentz

June 16, 2017

COMMITTEE MEMBERSHIP

TITLE:	
AUTHOR:	Samuel Steejans Artho-Bentz
DATE SUBMITTED:	June 16, 2017
COMMITTEE CHAIR	John Ridgely, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	William Murray, Ph.D. Professor of Mechanical Engineering
COMMITTEE MEMBER:	Glen Thorncroft, Ph.D. Professor of Mechanical Engineering

ABSTRACT

Title

Samuel Steejans Artho-Bentz

This thesis is a continuation of the work of Mr. Garret Gudgel. It contains guidelines for the software, mechanical, and electrical systems required for the implementation of a Parallel Actuator Mount as well as establishing test procedures.

The tested telescope does not demonstrate the required precision for astronomical use but recommended improvements are laid out.

What I did, what it was based on (acknowledge Gudgel), overview of results (pick the most important, give numbers, say whether it is sufficient or not)

Keywords:

I'd like to see a significant change in the emphasis of the abstract, but it's best not to worry about that right now -- I think when the rest of the document is done, updating the abstract won't be difficult. In general, beginning with a reference to someone else's work isn't the right idea; you definitely will be mentioning his work, but that can come in the background section, and possibly in the third or fourth sentence of the abstract. The first things to mention are what you have accomplished.

ACKNOWLEDGMENTS

Placeholder text… Insert content here.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

A previous thesis at California Polytechnic State University, San Luis Obispo (Gudgel, 2015) demonstrated the feasibility of a three degree-of-freedom parallel actuator telescope mount based loosely on the six degree-of-freedom Array for Microwave Background Anisotropy (AMiBA) telescope. The load paths created by the parallel actuators are shorter than those of a similarly sized traditional telescope mount. This could contribute to a stiff, light-weight system with a high natural frequency, which is ideal for accurate pointing. This simplified mount sacrifices full sky coverage for portability, stiffness, and lower cost. The purpose of this thesis is to refine the proof-of-concept developed at Cal Poly with commonly available, lower priced components and more flexible software. By doing so, it allows the development of a design method and code that can be utilized by universities and astronomers to create their own parallel actuator telescope mounts. Further, it builds upon the partial proof-of-concept prototype in order to demonstrate the capability of the system to accurately point at and track stars. This includes improvements to the mechanical, electrical, and control systems. A focus will be on the simplification of the system, cost saving, and the use of off-the-shelf parts in order to increase the feasibility of the design for educational and hobbyist use.

2. BACKGROUND

Telescope mounts have three angles of interest which are used to describe where the axis of the telescope is pointed and the orientation of the telescope about that axis. These three angles can be defined in various ways. A straightforward definition for telescope aiming is altitude-azimuth-image rotation. This set of angles is defined for a local observer. Altitude is the angle above the horizon. Azimuth is the angle about an axis perpendicular to Earth's surface starting from some reference point, generally north or south (see Figure 1: Definition Altitude/Azimuth relative to an observer Something weird happened here?). The third angle, which is not a controlled angle in all mounting systems, is the rotation about the axis at which the telescope is pointing. This last angle can be very important in astronomy. It is critical to binary star position measurements which refer to celestial North for measurement (Ridgely). Controlling the third angle also allows for long exposure photographs to be taken by keeping the viewing angle ^with respect to... of the object constant. Although this set of angles is convenient for aiming a telescope from a specific place and time, another coordinate system, Right Ascension-Declination, is more commonly used to give celestial coordinates in books and tables of celestial bodies because it is independent of time and location. It is possible to convert from Right Ascension-Declination to Altitude-Azimuth as shown in Section 3.7.

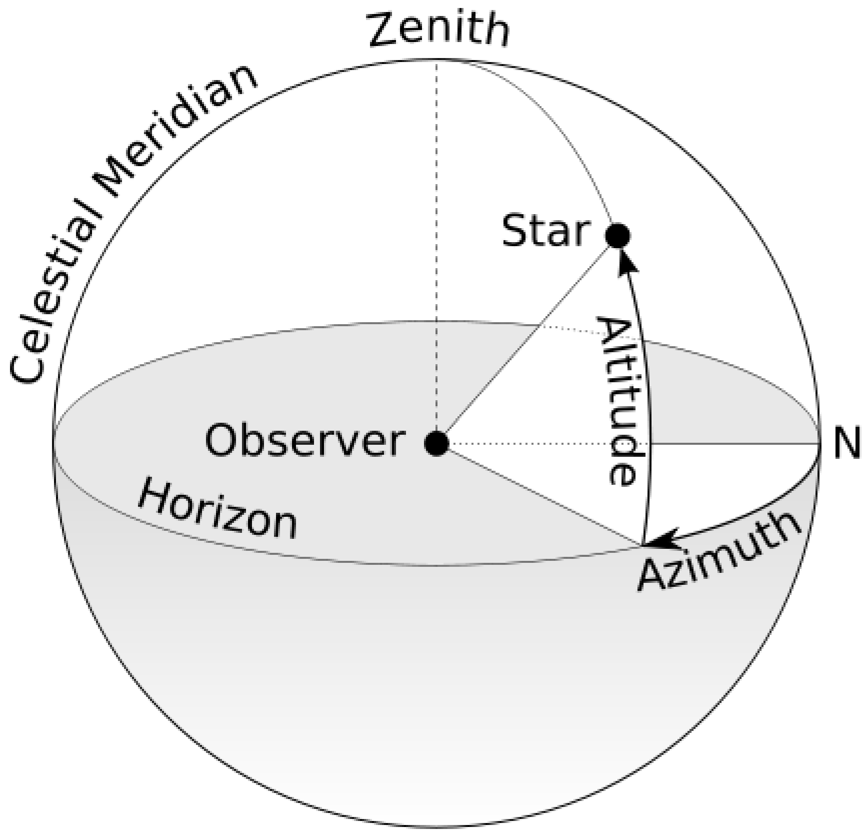


Figure 1: Definition of Altitude/Azimuth relative to an observer (TWCarlson)

2.1 Traditional Telescope Mounts

Traditional telescope mounts generally use one actuator per angle of interest. These actuators are necessarily mounted in series such that each actuator must hold the entire weight of the telescope as well as that of each actuator above it. This results in large required actuation force or torque and long load paths. This requires massive systems in order to achieve the required stiffness.

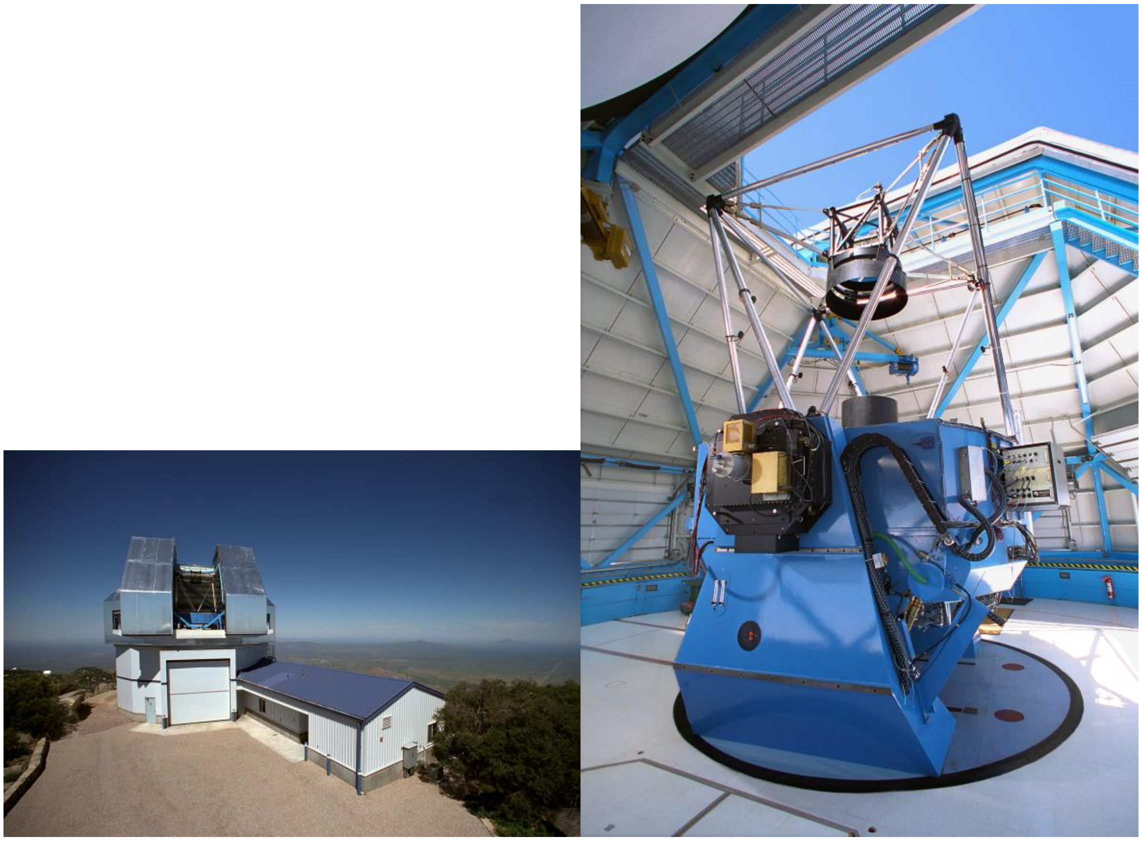


Figure 2: WIYN Altazimuth Mount 3.5m Telescope (National Optical Astronomy Observatory/Association of Universities for Research in Astronomy/National Science Foundation)

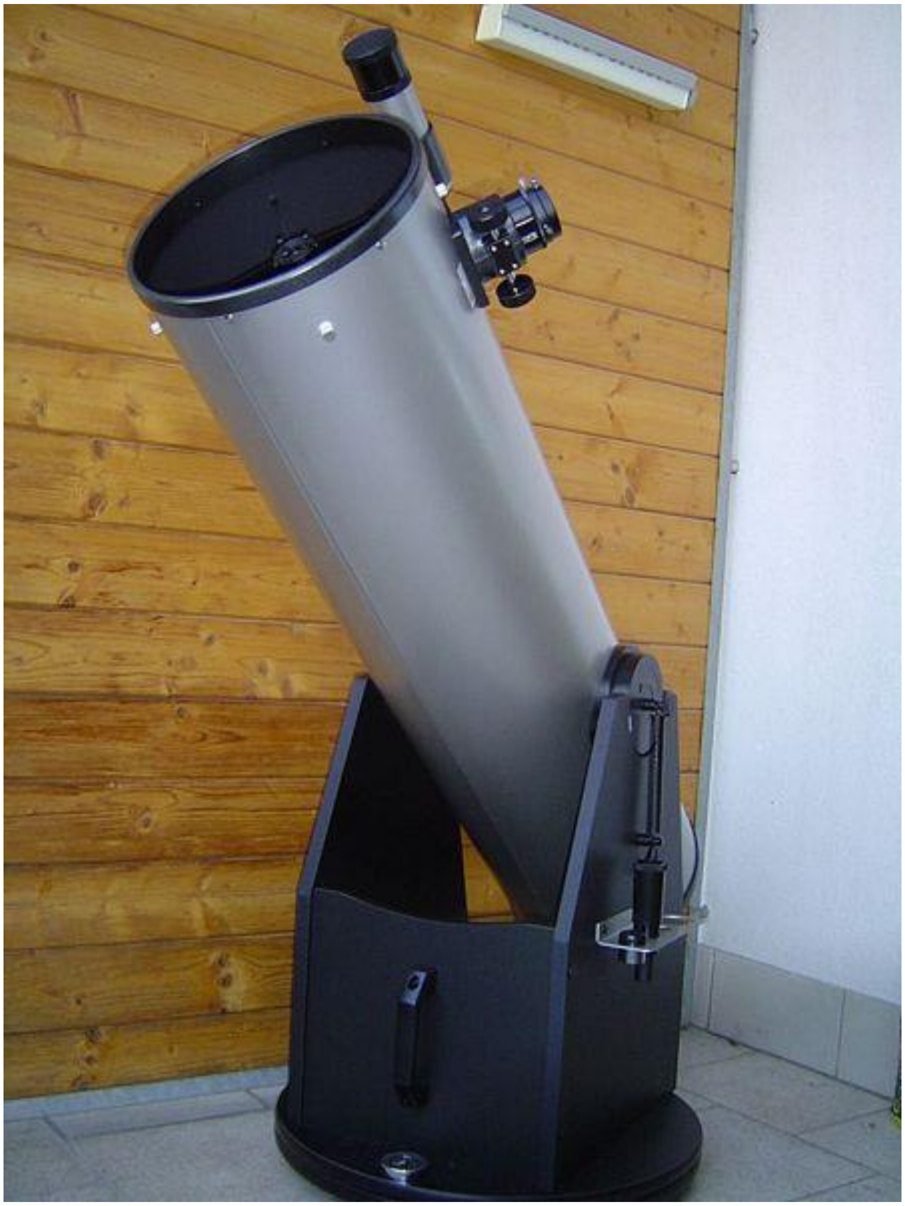


Figure 3: 10" Telescope on an Altitude Azimuth Mount (Dobson Mount, 2005)

The most basic mount, referred to as an altazimuth mount (see Figure 2 and Figure 3), has a rotational actuator which directly moves the azimuth angle. On top of that is a second actuator which controls the altitude angle. The altitude azimuth mount generally has no way to directly control the image rotation angle and requires that functionality to be built into the telescope itself through means of an image de-rotation device.

Perhaps a very brief discussion of how "de-rotators" usually rotate only the camera, etc.?

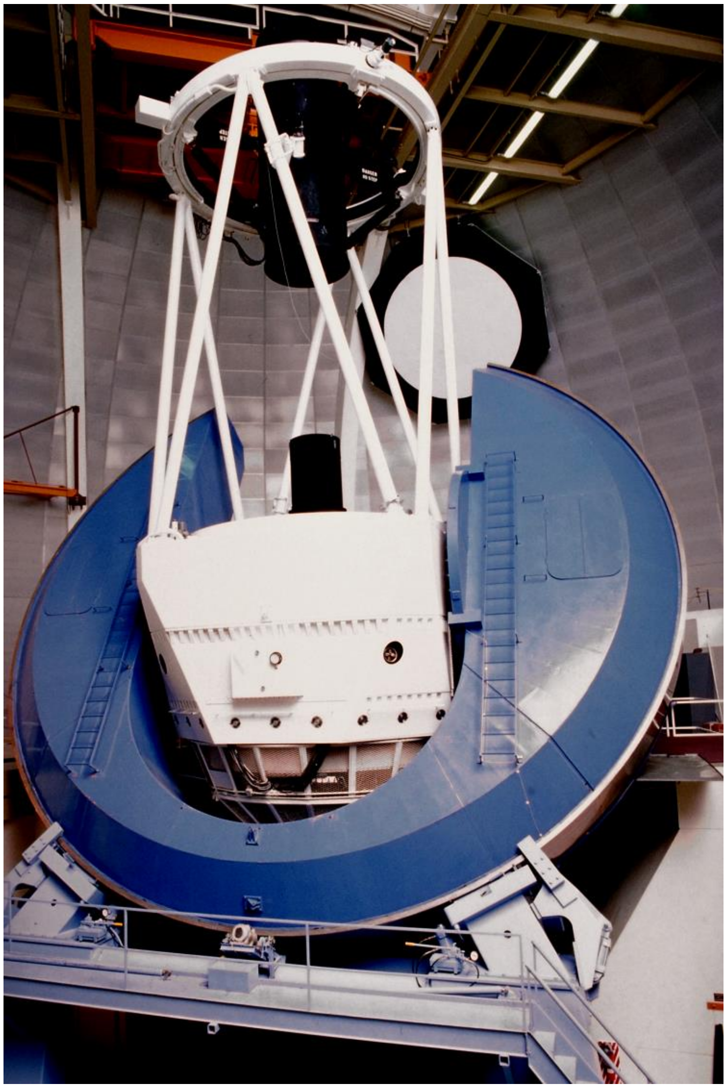


Figure 4: Mayall Equatorial Mount 4m Telescope (National Optical Astronomy Observatory/Association of Universities for Research in Astronomy/National Science Foundation)

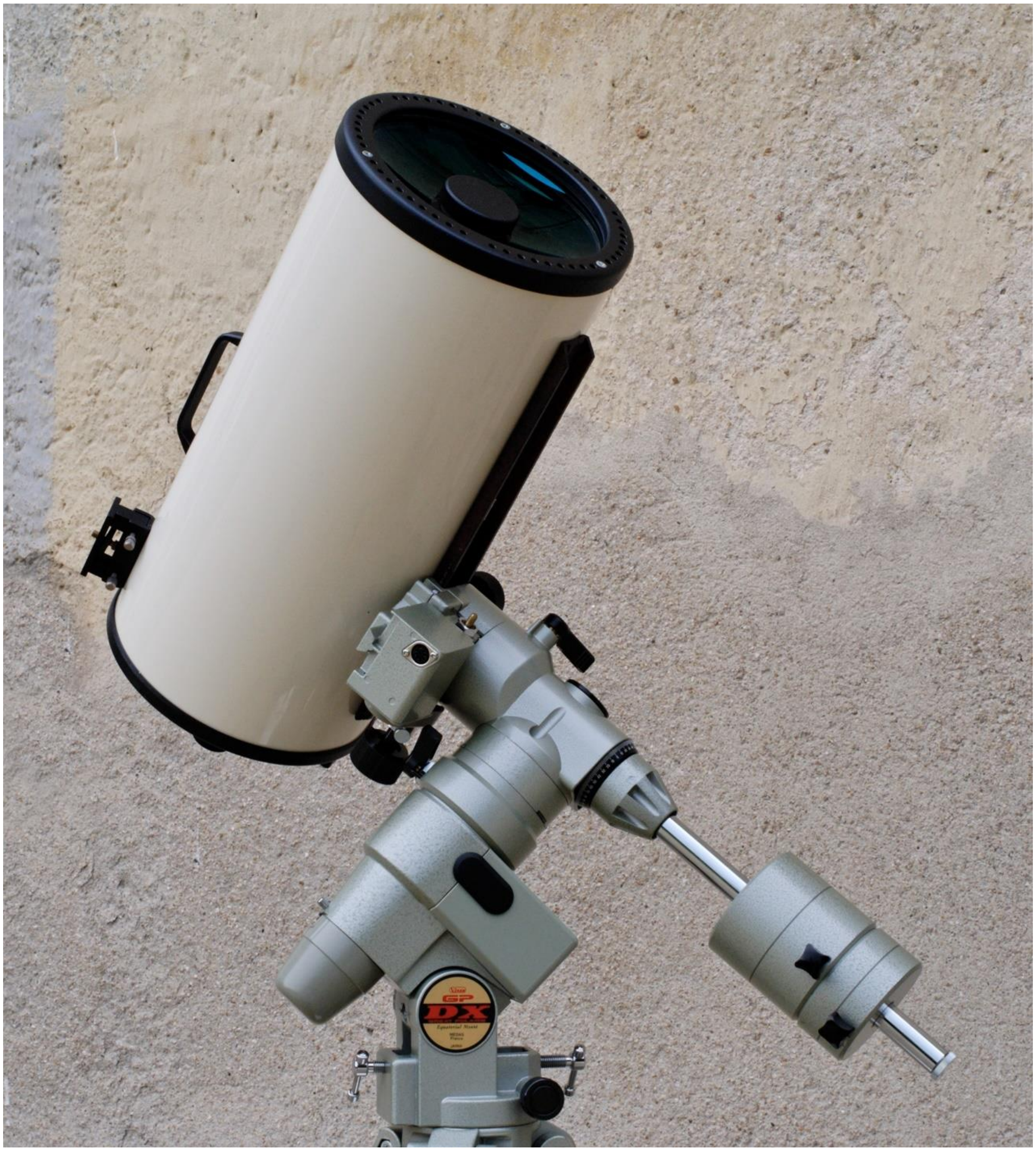


Figure 5: Vixen GP-DX German equatorial mount (Nguyen, 2007)

Should probably all be singular -- "An equatorial mount does..."

Equatorial mounts do not have an actuator to control the image rotation angle. Instead, the whole telescope is tilted to match the Earth's rotational axis, causing the image

(in particular, the axes of rotation)

rotation to remain constant (see Figure 4: Mayall Equatorial Mount 4m Telescope and Figure 5). Unfortunately, this creates an even more complicated load path than an altazimuth mount, and often necessitates large counterweight systems.

2.2 Hexapod Mount

The Stewart platform was initially conceived of as a method of simulating flight conditions for pilot training (Stewart, 1965). It is a mechanism based on six independently actuated linear legs which provide six degrees of freedom: x, y, z, pitch, roll, and yaw. Stewart platforms are used in machine tools, flight simulators, and astronomy (see Figure 6 (Koch, et al., 2009)).

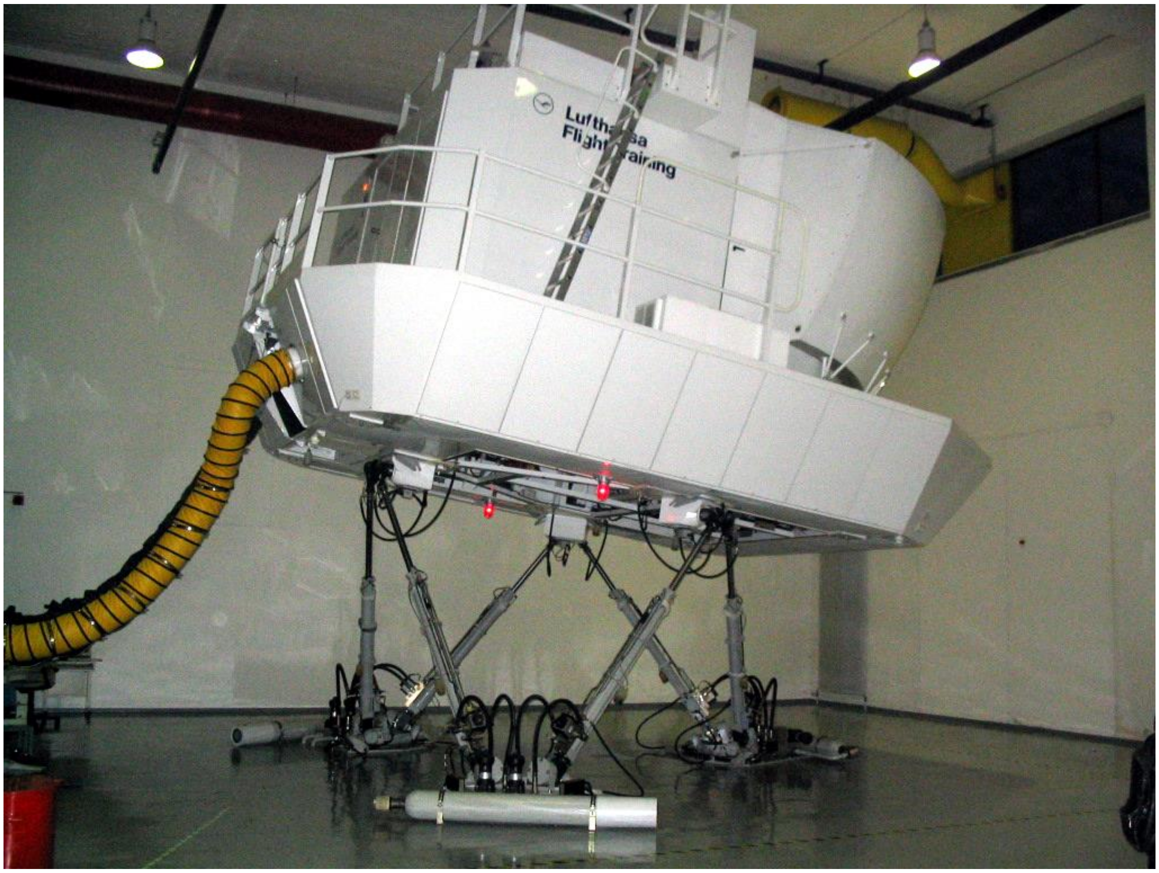


Figure 6: Stewart-Gough Platform in use as a flight training module for Lufthansa (Arnold, 2004)

In 1969, Peter Fellgett proposed the use of the Stewart platform for astronomical purposes using hydraulic actuation. In 1989, a 1.5m prototype of a hexapod telescope was funded in Germany with the intent of proving the concept for use with a 12m telescope. The mechanical system was completed and demonstrated to meet the required specifications, but the full telescope was not completed due to complications stemming from the reunification of East and West Germany (Chini, 2000). In 2006, observations began at the Array for Microwave Background Anisotropy (AMiBA) (see Figure 7). It is the largest hexapod telescope in operation. The hexapod mount was chosen for this application based on size, weight, accessibility, and portability requirements (Koch, et al., 2009).

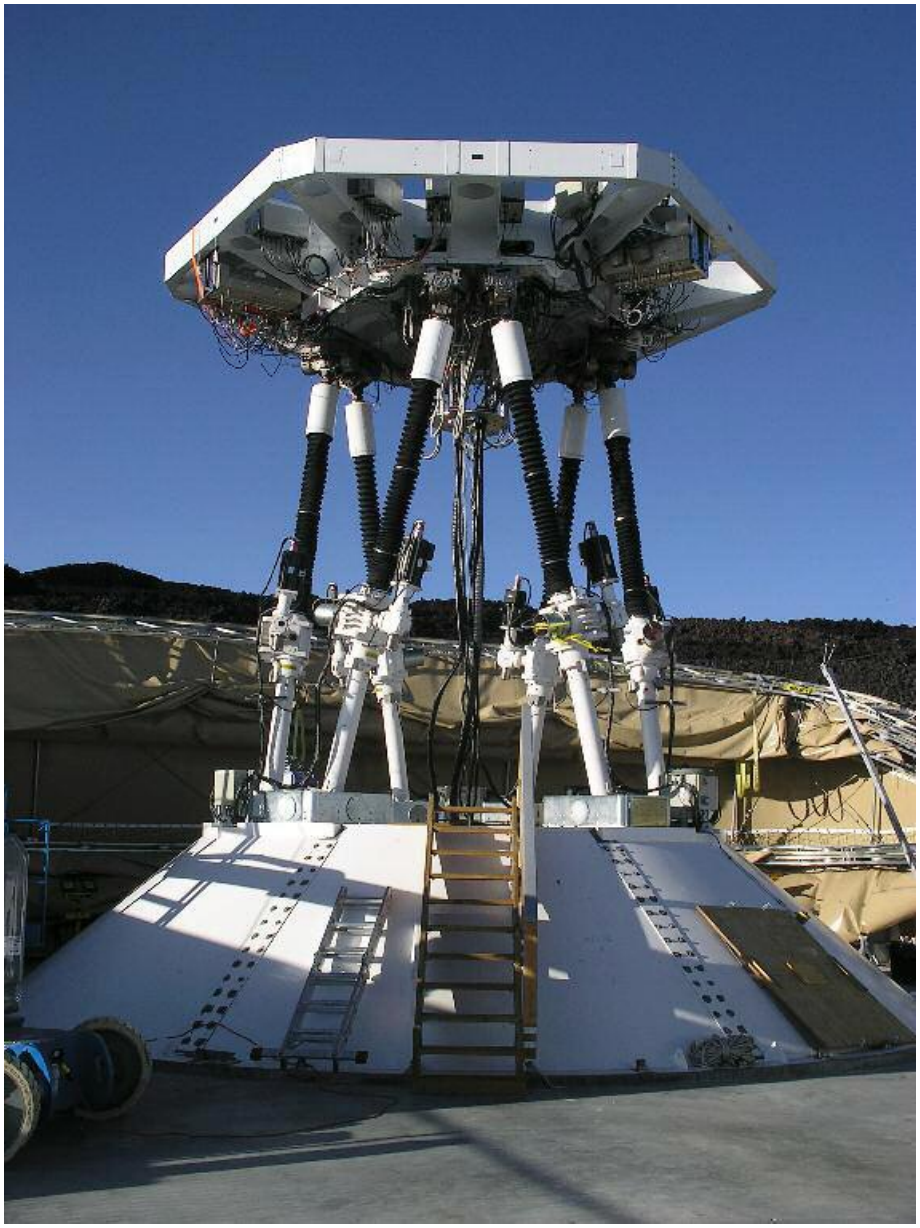


Figure 7: AMiBA in neutral position. (Koch, et al., 2009)

2.3 Parallel Actuator Mount

The Parallel Actuator Mount is a novel system conceived by Dr. John Ridgely and initially implemented by Garrett Gudgel with guidance from Dr. John Ridgely and Dr. Russell Genet (Gudgel, 2015). Mr. Gudgel approached the issue of transportability, long exposure image rotation, and excess required mass with his telescope mount. In his investigation, he found that current telescope mounting systems could be improved for the use of amateur and small-scale research purposes, which did not require full sky coverage nor full six degree of freedom capabilities. His goal was to create a system which was less massive and cheaper without sacrificing stiffness or accuracy.

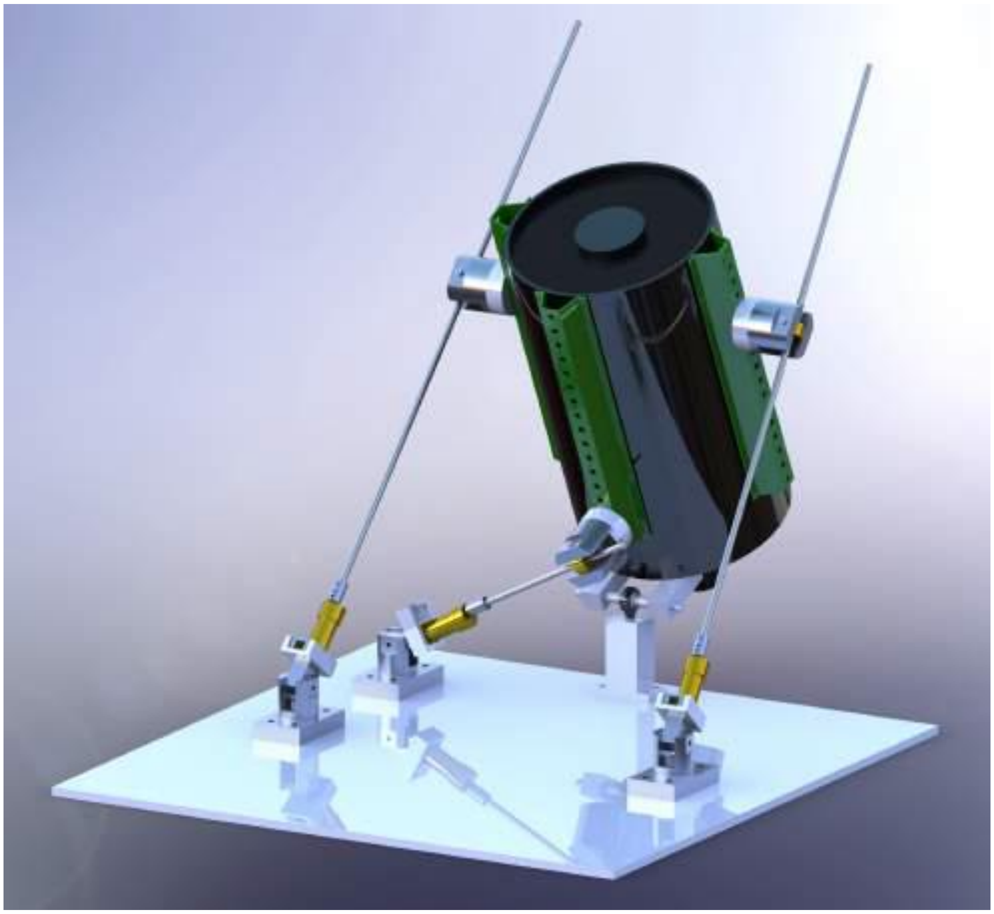


Figure 8: Solid model showing the design of the Parallel Actuator Mount prototype (Gudgel, 2015)

His solution to these issues was to design a mount system, based on the AMiBA telescope, which used linear actuators in parallel instead of rotational actuators in series. This allowed him to build image rotation into the system as well as to create shorter loading paths which lower the overall required strength, and thus mass, of each actuator. Mr. Gudgel's design is a novel modification of the hexapod mount. It is composed of three linear actuators, a stationary three degree-of-freedom ball-in-socket joint, six two degree-of-freedom joints, a baseplate, and a frame which contains and/or represents the telescope.

In this system a large portion of the load is supported by the ball-and-socket joint with the remaining load being shared between the three linear actuators.

3. THEORY

3.1 Coordinate System Definitions

The Parallel Actuator Mount uses a coordinate system transformation to determine the actuator lengths required to aim the system at the desired point. See Section 3.5 for details on this transformation. This requires the definition of two coordinate systems: One is defined with respect to the unmoving base of the system and the other is defined with respect to the telescope itself.

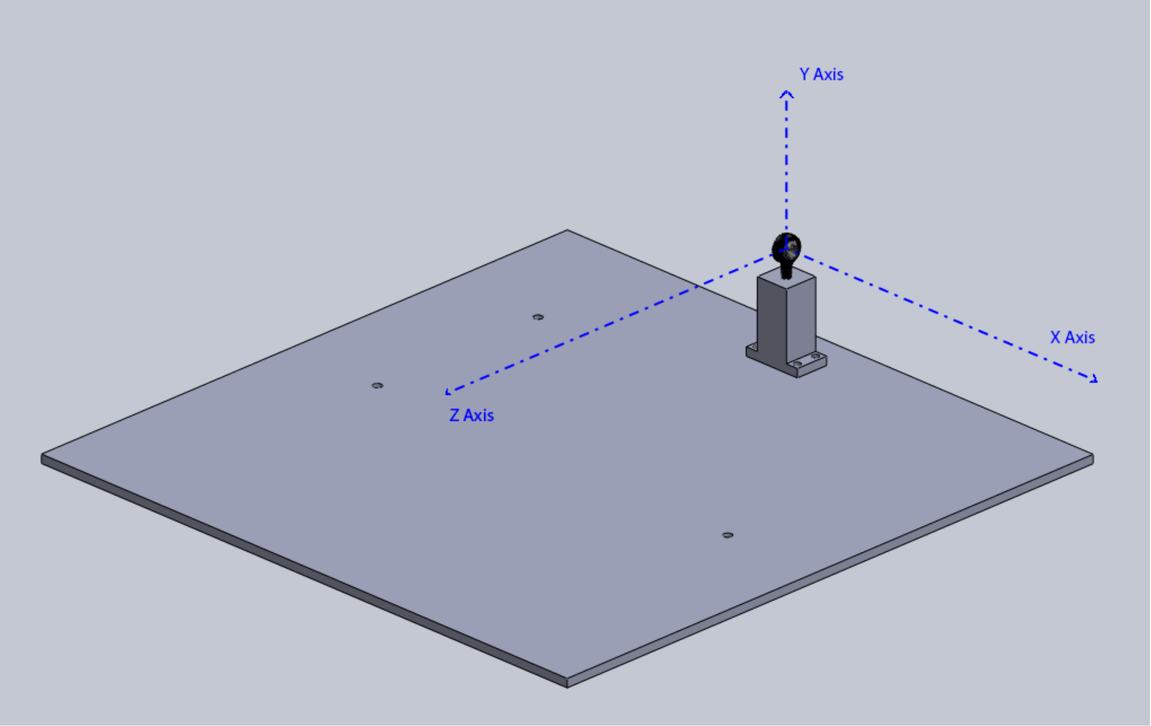


Figure 9: XYZ Coordinate system.

The XYZ coordinate system is a global, orthonormal coordinate system. Its origin is at the center of the stationary ball-in-socket joint (see Section 4.1.4) and the axes are defined as shown in Figure 9.

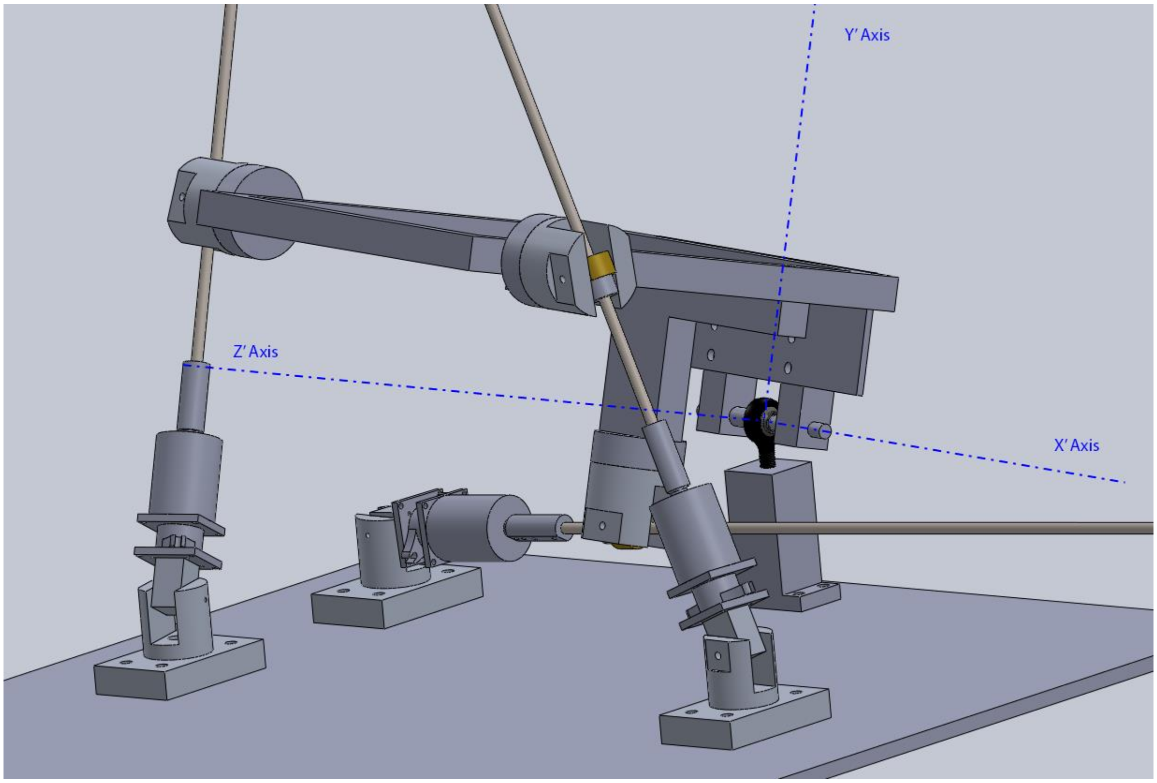


Figure 10: X'Y'Z' Coordinate System

The X’Y’Z’ coordinate system is local to the telescope itself. Figure 10 shows the definition of these axes on a plain frame rather than an actual telescope. The purpose of removing the telescope is explained in Section 4.2.1. It is an orthogonal system with its origin at the center of the stationary ball-in-socket joint. Its axes are defined as shown. The Z’ axis is along the optical axis of the telescope: The telescope points in the direction of the Z’ axis.

A major feature of these coordinate system definitions is that the origin is at the same point. This allows the transformations to be purely rotational. The translational elements of transformation matrices can be ignored.

3.2 Angles of Interest

This system uses the altitude-azimuth definition of angles of interest. These angles include azimuth, altitude, and image rotation angles which are defined as shown in Figure 11, Figure 12, and Figure 13 respectively. This common definition of the angles is chosen in order to allow the use of readily available software and calculations.

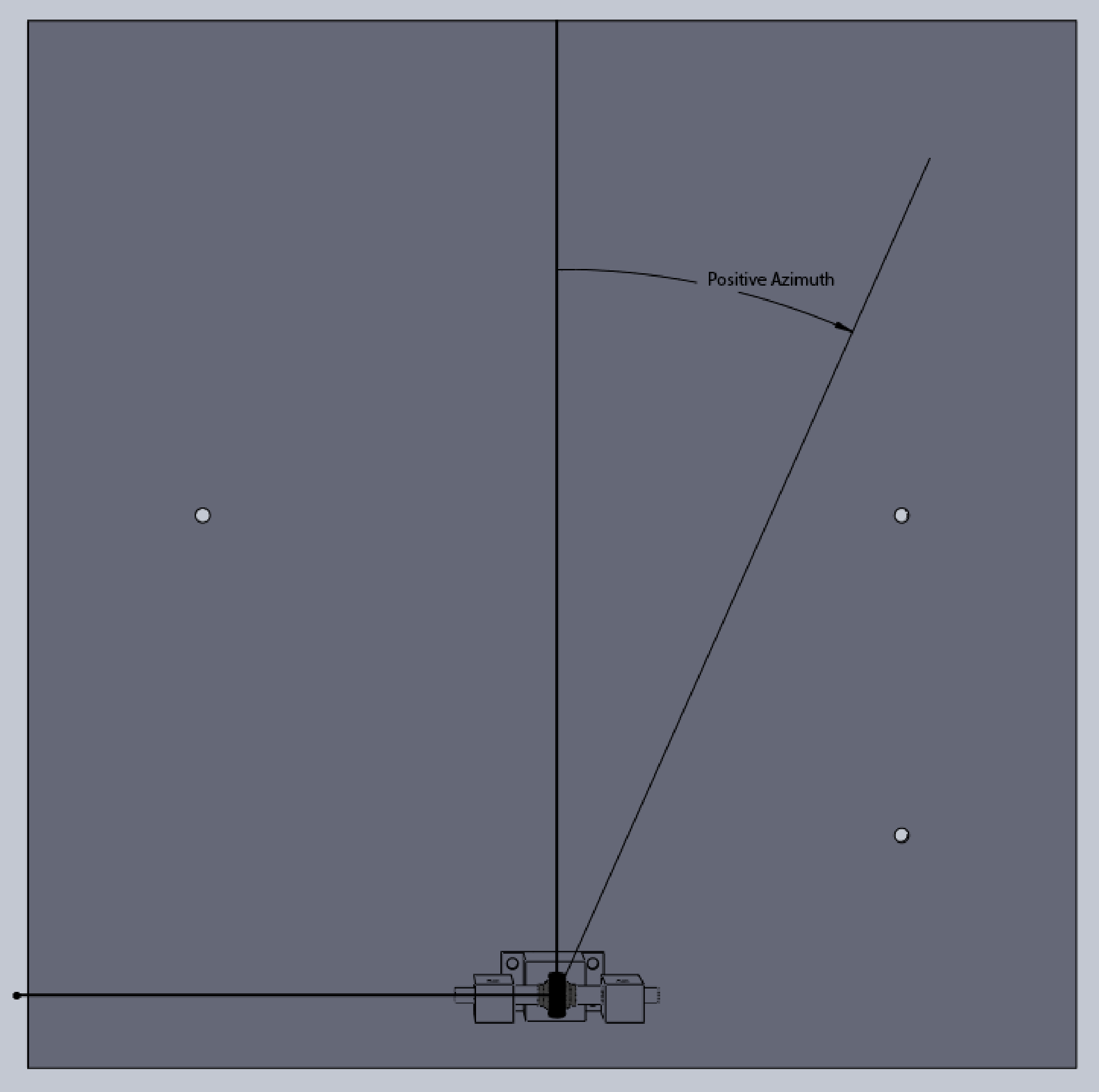


Figure 11: Definition of the Positive Azimuth Angle. View is looking down at the baseplate with the Telescope Joint Assembly visible as a reference to the origin.

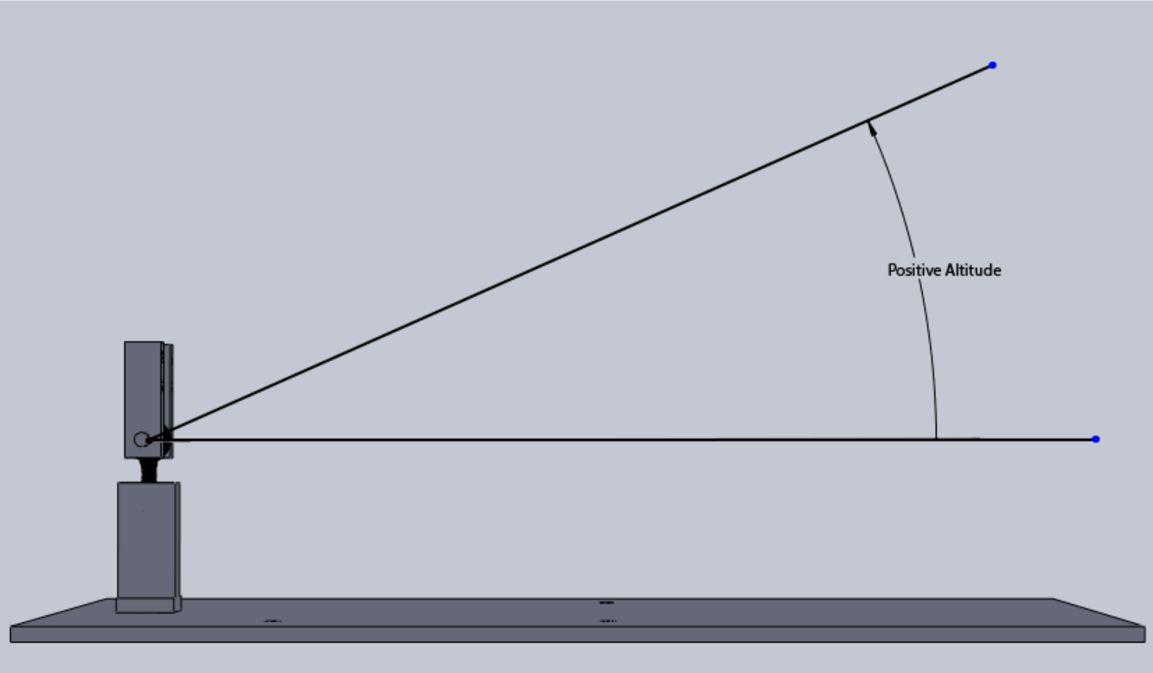


Figure 12: Definition of the Positive Altitude Angle. View is side on to the baseplate with the Telescope Joint Assembly visible as a reference to the origin.

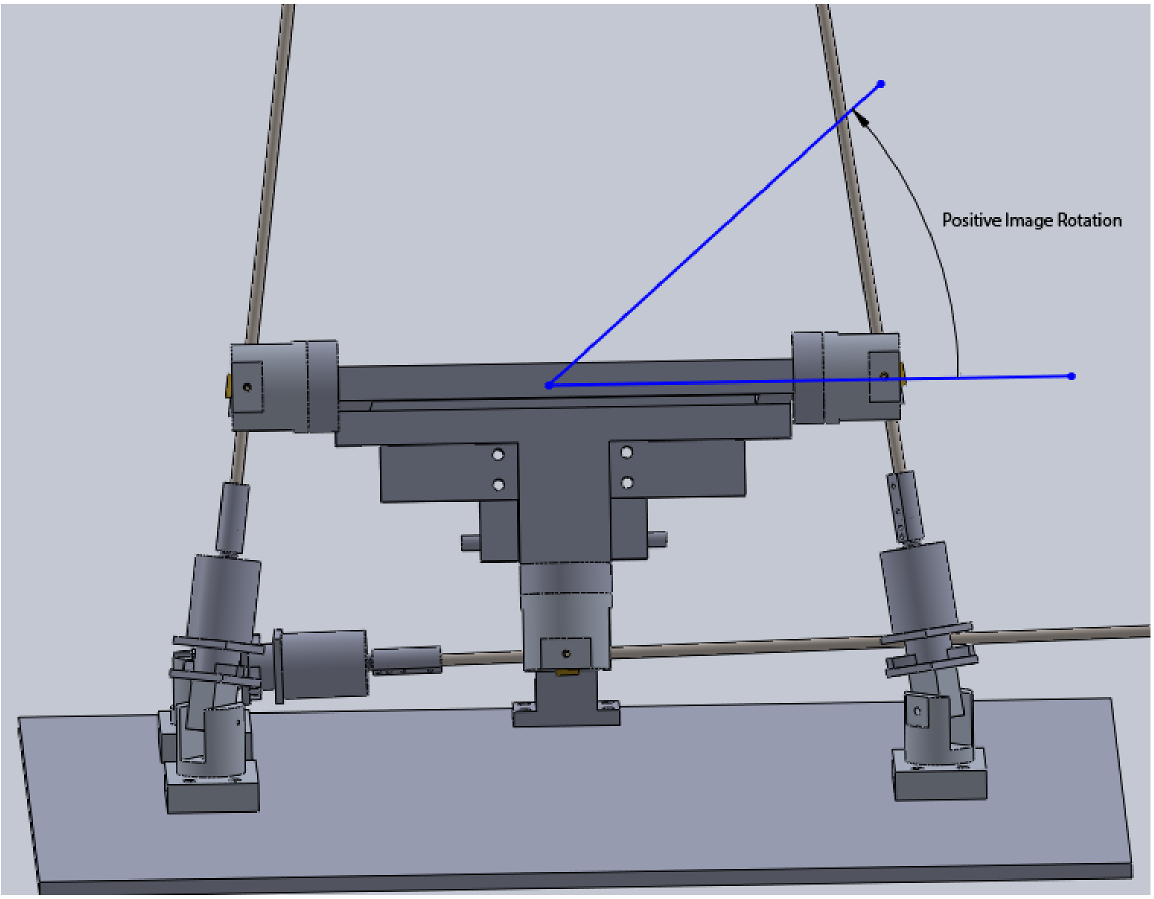
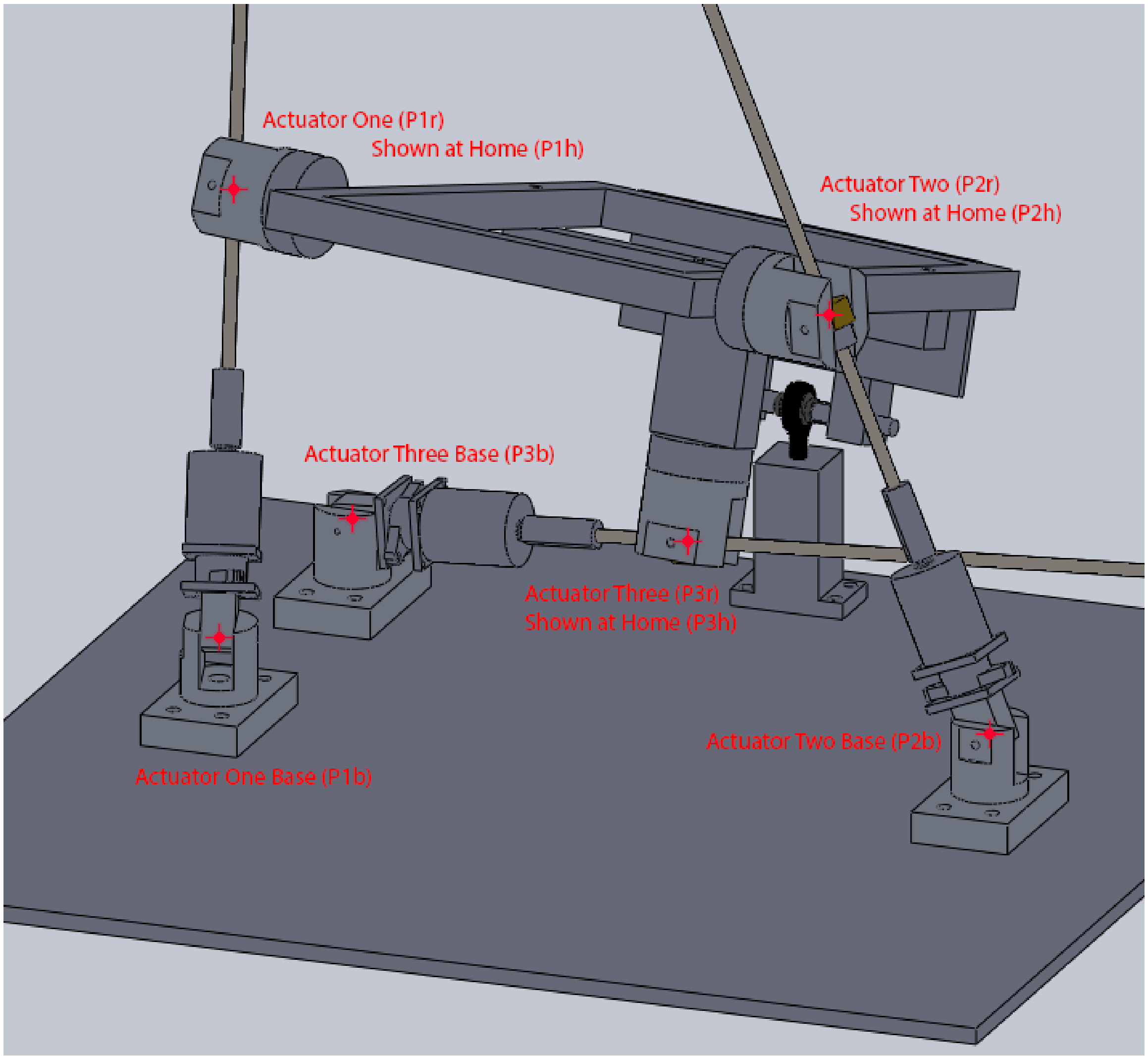


Figure 13: Definition of the Positive Image Rotation Angle. View is from the sky looking down into the telescope.

3.3 Points of Note

There are several points on the mount that are used for calculating the position of the system. These include the centers of rotation of the ground-to-actuator joint assemblies (Section 4.1.2), telescope joint assembly (Section 4.1.4), and actuator-to-telescope joint assemblies (Section 4.1.3). The ground-to-actuator centers of rotation are referred to as points P_b. The actuator-to-telescope centers of rotation are referred to as points P_r. These are shown in Figure 14. The center of the telescope joint assembly is referred to as the coordinate systems origin as shown in Figure 9 and Figure 10. The origin and P_b are stationary points that do not change no matter what position the telescope is in. P_r moves based on the location of the telescope. In Figure 14, P_r is shown in the home position P_h.



Would it be easy to put white boxes behind each bit of text? That would help make things as readable as possible. If not easy, it's OK this way.

Figure 14: Points of Note

3.4 Home Position

The Parallel Actuator Mount relies on having an accurately known position to reference. This position is referred to as the "home position" and is shown in Figure 14. It is created through the use of hard stops on the linear actuators which set a minimum length. When all actuators are at the hard stops, the system is in the home position. All position information in the system is incremental, so a homing protocol must be done at the beginning of each use in order to redefine the home position. The combination of the known home position and the known movement away from the home position results in the absolute position.

...not really redefine, but find.

Due to the geometry of the system and the definition of the angles of interest, the home position cannot be at a “clean” position like altitude = 0°, azimuth = 0°, image rotation = 0°. The home position must be set at an extreme of the desired azimuth angle range. It will also limit how close to the horizon the system can aim.

Does extreme of actuator movement require extreme azimuth?

No problem -- aiming really low to the horizon is seldom good for science anyway, as the turbulence in the atmosphere is worst there. Below 10 or 15 degrees it's awful.

3.5 Transformations

In order to find the actuator lengths required to match a particular set of altitude, azimuth, and image rotation angles, a known reference (home) position must be defined. This reference position (see Section 3.3), contains complete information of the locations of points of interest: both ends of each actuator, a point on the image axis, and the point of rotation. This reference position, along with appropriate rotation matrices, allows us to find the required lengths.

All desired angular positions are treated as rotations away from the home position. This requires three rotations, one about each of the X, Y, and Z axes.

But when you move about one axis, you move two others: Do you mean the X, Y', and Z' axes? (I'm not sure...just asking.) Also, isn't it Y, then X', then Z'?

$$RotX(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$RotZ(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$RotY(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

These transformations align with the angles of interest of the system. A rotation about the X axis is a change of altitude (θ_{alt}). A rotation about the Y axis is a change of azimuth (θ_{az}). A rotation about the Z axis is a change of image rotation (θ_{rot}).

Matrix multiplication is not commutative, so care must be given to the order in which the rotations are applied. For this system, the required order is azimuth > altitude > image rotation. This order was determined by examining traditional telescope mounts with series actuators and noting which actuators "supported" others. For example, a change in the azimuth angle moves the axes of rotation of both the altitude and image rotation angles and so must come first.

Note:

- sTb indicates a transformation from the base 'b' coordinate system to the telescope 's' coordinate system.
- $\cos\theta$ and $\sin\theta$are abbreviated to $c\theta$ and $s\theta$ respectively.
- Due to the definition of the angles, the negative of the altitude and azimuth angles are used.

$$sTb = \begin{bmatrix} c(\theta_{rot}) & -s(\theta_{rot}) & 0 \\ s(\theta_{rot}) & c(\theta_{rot}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c(-\theta_{alt}) & -s(-\theta_{alt}) \\ 0 & s(-\theta_{alt}) & c(-\theta_{alt}) \end{bmatrix} \begin{bmatrix} c(-\theta_{az}) & 0 & s(-\theta_{az}) \\ 0 & 1 & 0 \\ -s(-\theta_{az}) & 0 & c(-\theta_{az}) \end{bmatrix}$$

$$sTb = RotZ(\theta_{rot}) * RotX(-\theta_{alt}) * RotY(-\theta_{az})$$

However, this transformation only works if the home position is aligned with θ_{alt}=0°, θ_{az}=0°, and θ_{rot}=0°. This is not possible (see section 3.4) so a set of corrective rotations must also be included. These three correction angles ($\phi_{alt}, \phi_{az}, \phi_{rot}$) are simply an additional set of rotation matrices that must be applied.

$$sTb = \begin{bmatrix} c(\theta_{rot}) & -s(\theta_{rot}) & 0 \\ s(\theta_{rot}) & c(\theta_{rot}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c(\phi_{rot}) & -s(\phi_{rot}) & 0 \\ s(\phi_{rot}) & c(\phi_{rot}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c(-\theta_{alt}) & -s(-\theta_{alt}) \\ 0 & s(-\theta_{alt}) & c(-\theta_{alt}) \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & c(\phi_{alt}) & -s(\phi_{alt}) \\ 0 & s(\phi_{alt}) & c(\phi_{alt}) \end{bmatrix} \begin{bmatrix} c(-\theta_{az}) & 0 & s(-\theta_{az}) \\ 0 & 1 & 0 \\ -s(-\theta_{az}) & 0 & c(-\theta_{az}) \end{bmatrix} \begin{bmatrix} c(\phi_{az}) & 0 & s(\phi_{az}) \\ 0 & 1 & 0 \\ -s(\phi_{az}) & 0 & c(\phi_{az}) \end{bmatrix}$$

Note that when two rotations about the same axis are applied they can be done in either order.

$$\begin{aligned} RotX(\theta_1) * RotX(\theta_2) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}\,\theta_1 & -\mathrm{s}\,\theta_1 \\ 0 & \mathrm{s}\,\theta_1 & \mathrm{c}\,\theta_1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}\,\theta_2 & -\mathrm{s}\,\theta_2 \\ 0 & \mathrm{s}\,\theta_2 & \mathrm{c}\,\theta_2 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}\,\theta_1\,\mathrm{c}\,\theta_2 - \mathrm{s}\,\theta_1\,\mathrm{s}\,\theta_2 & -(\mathrm{c}\,\theta_1\,\mathrm{s}\,\theta_2 + \mathrm{s}\,\theta_1\,\mathrm{c}\,\theta_2) \\ 0 & \mathrm{c}\,\theta_2\,\mathrm{s}\,\theta_1 + \mathrm{s}\,\theta_2\,\mathrm{c}\,\theta_1 & \mathrm{c}\,\theta_2\,\mathrm{c}\,\theta_1 - \mathrm{s}\,\theta_2\,\mathrm{s}\,\theta_1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}(\theta_1 + \theta_2) & -\mathrm{s}(\theta_1 + \theta_2) \\ 0 & \mathrm{s}(\theta_1 + \theta_2) & \mathrm{c}(\theta_1 + \theta_2) \end{bmatrix} \\ &= RotX(\theta_1 + \theta_2) \\ &= RotX(\theta_2 + \theta_1) \end{aligned}$$

This allows for a simplified combined transformation:

$$sTb(\theta_{alt}, \theta_{az}, \theta_{rot}) = RotZ(\theta_{rot} + \phi_{rot}) * RotX(-\theta_{alt} + \phi_{alt}) * RotY(-\theta_{az} + \phi_{az})$$

The correction angles are constants based on the design of the system and orientation of the home position. Thus, they are not required inputs to the transformation during the run of the telescope. This combined transformation allows an expression to be formed which takes as input the three angles of interest and, along with knowledge of the home position, outputs the end positions of the linear actuators required to achieve those angles as shown below:

Let P1$_h$, P2$_h$, P3$_h$ represent the coordinates of the actuator ends in the base coordinate system while in the home position and P1$_r$, P2$_r$, P3$_r$ represent the coordinates of the actuator ends after a rotation has occurred. This results in a fundamental equation for this system:

$$\begin{bmatrix} P_r x \\ P_r y \\ P_r z \end{bmatrix} = sTb(\theta_{alt}, \theta_{az}, \theta_{rot}) * \begin{bmatrix} P_h x \\ P_h y \\ P_h z \end{bmatrix}$$

Once the rotated end positions of the linear actuators have been found, the distance formula is applied between these new end positions and the base position of the actuator in order to calculate the required length for each actuator:

$$L = \sqrt{(P_r x - P_b x)^2 + (P_r y - P_b y)^2 + (P_r z - P_b z)^2}$$

3.6 Angular Velocities

Two methods were attempted for calculating the required linear velocities to produce the desired angular velocities. The first, taking the time derivative of a transformation matrix, would be a more elegant solution but requires more information than is available in the system. The second, forward calculating what length would be required after a specified time step, requires certain assumptions to be made but is able to be implemented in the system.

3.6.1 Time Derivative of a Transformation Matrix

As discussed in Section 3.1, transformation matrices are used to create a relationship between actuator lengths and system angular position. The time derivative of those transformation matrices allows for the creation of a relationship between the actuator linear velocities and the system angular velocities.

A little bit of "the following equations are..." would be helpful here.

$$\boldsymbol{p_r} = Coordinate\ of\ the\ actuator\ end\ point\ after\ rotation$$

$$\boldsymbol{p_h} = Coordinate\ of\ the\ actuator\ end\ point\ in\ the\ home\ position$$

$$L = \|\boldsymbol{p_r} - \boldsymbol{p_h}\| Length\ of\ the\ Actuator$$

$$sTb\ = \ transformation\ as\ described\ in\ section\ 3.5$$

$$\boldsymbol{\theta} = Angular\ change\ from\ \boldsymbol{p_h}\ \boldsymbol{to}\ \boldsymbol{p_r}$$

$$\boldsymbol{\omega} = rate\ of\ change\ of\ \boldsymbol{\theta}$$

$$S(\boldsymbol{\omega}) = \begin{matrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ \omega_y & \omega_x & 0 \end{matrix} = skew\ symmetric\ of\ \boldsymbol{\omega}$$

[]? ()? {}?

$$\boldsymbol{p_r} = sTb * \boldsymbol{p_h}$$

$$\frac{d}{dt}\boldsymbol{p_r} = \frac{d}{dt}(sTb * \boldsymbol{p_h})$$

$$\dot{\boldsymbol{p_r}} = (s\dot{T}b * \boldsymbol{p_h} + sTb * \dot{\boldsymbol{p_h}})$$

$$\dot{\boldsymbol{p_h}} = 0\ (\boldsymbol{p_h}\ \text{is a constant position})$$

$$\dot{\boldsymbol{p_r}} = s\dot{T}b *\ \boldsymbol{p_h}$$

$$s\dot{T}b(\boldsymbol{\theta}) = S(\boldsymbol{\omega}) * sTb(\boldsymbol{\theta})\ (\text{Zhao}, 2016)□$$

What's this?

$$\dot{\boldsymbol{p_r}} =\ S(\boldsymbol{\omega}) * sTb(\boldsymbol{\theta}) * \boldsymbol{p_h}$$

$$L = \sqrt{(P_{rx} - P_{hx})^2 + \left(P_{ry} - P_{hy}\right)^2 +\ (P_{rz} - P_{hz})^2}$$

$$\dot{L} = \frac{\dot{\boldsymbol{p_r}}}{L} * (\boldsymbol{p_r} - \boldsymbol{p_h})$$

This shows that $\dot{\boldsymbol{p}}_{\boldsymbol{r}} = f(\boldsymbol{\omega}, \boldsymbol{\theta}, \boldsymbol{p}_{\boldsymbol{h}})$ and $\dot{L} \ = f(\boldsymbol{\omega}, \boldsymbol{\theta}, \boldsymbol{p}_{\boldsymbol{h}}, \boldsymbol{p}_{\boldsymbol{r}}, L)$**.** In this equation, $\boldsymbol{\omega}, \boldsymbol{p}_{\boldsymbol{h}}$, and L are known at all times. However, the established transformations only go from $\boldsymbol{p}_{\boldsymbol{h}}, \boldsymbol{\theta} \rightarrow \boldsymbol{p}_{\boldsymbol{r}}, \boldsymbol{L}$**,** so either $\boldsymbol{\theta}$ or $\boldsymbol{p}_{\boldsymbol{r}}$ would need to be known or calculable. The addition of an external device such as an inertial measurement unit could be sufficient to allow this method of velocity control to be implemented.

I doubt that the accuracy of a cheap IMU would be even close; in any case, there needs to be some explanation of why we're not doing this.

3.6.2 Forward Calculation of Desired Position

Angular velocities are calculated by discretizing the movement over small time steps. The first step is to determine what length each actuator should have at some specific moment in the future. Basic kinematic equations with constant angular acceleration lead to:

$$\theta_{new} = \theta_{current} + \omega * t_{step}$$

Utilizing the transformations developed in Section 3.5, in conjunction with these new angular positions, results in the required lengths of each actuator:

$$L_{new} = f(\theta_{altitude\ new}, \theta_{azimuth\ new}, \ \theta_{rotation\ new})$$

The definition of linear velocity as the rate of change of position is then used to calculate the required angular velocity based on the physical system:

$$V \approx \frac{L_{new} - L_{current}}{t_{step}}$$

$$stepsPerSec = V \left[\frac{inch}{second}\right] * \ threadPitch \ \left[\frac{rotations}{inch}\right]$$
$$* \ stepperAccurancy \left[\frac{steps}{rotation}\right]$$

Once the system has run for t_{step} time, the process starts over and a new future position is calculated.

This method assumes that the acceleration of the linear actuator is sufficient such that the velocity over the time step can be treated as constant. If a system does not accelerate fast enough, this method will result in significant positional error.

Can some numbers be put here to show that this isn't a problem? I think the numbers are so extreme (slow speeds, long(ish) time steps like a whole second, reasonable stepper acceleration) that you don't need any complex calculations.

3.7 Pointing a Telescope

In order to know where to point a telescope, the altitude and azimuth of the desired object must be known. However, altitude and azimuth vary depending on the location of observation and the time at which the observation is taken. Therefore, a different coordinate system, right ascension/declination, is used to create references of the locations of stars.

Right ascension (RA) and declination (Dec) form an Earth-centered celestial coordinate system which is independent of the viewer's current time and location. The declination of a celestial object is the angle between two lines, one from the center of the Earth to the equator and the other from the center of the Earth to the object. Right ascension is analogous to longitude. It is measured from the vernal equinox—the point in the sky where the sun crosses the celestial equator from South to North—and increases eastward (Collins, 1989). RA/Dec is important because it is the most common method of listing the coordinates of celestial objects. Converting these coordinates to Alt/Az is necessary for them to be useful to an observer.

The information required to convert from RA/Dec to Alt/Az is: Right ascension (RA) and declination (Dec) of the object, latitude (lat) and longitude (long) of the observer, and the time and date of observation.

The first step is to find the decimal number of days elapsed from the reference date (fundamental epoch) of the RA/Dec coordinates used, usually J2000. The process is:

1. Find the observation time in coordinated universal time (UTC).
2. Convert the observation time to a decimal number of days. $d_{beginning\ of\ month} = d_{of\ the\ month} + \frac{\frac{minute}{60}+hour}{24}$
3. Look up the number of days to the beginning of the month.

Month	Normal Year	Leap Year
January	0	0
February	31	31
March	59	60
April	90	91
May	120	121
June	151	152
July	181	182
August	212	213
September	243	244
October	274	274
November	304	305
December	334	335

4. Look up the number of days from J2000 to the beginning of the year.

Year	Days	Year	Days	Year	Days
2018	6573.5	2024	8764.5	2030	10956.5
2019	6938.5	2025	9130.5	2031	11321.5
2020	7303.5	2026	9495.5	2032	11686.5
2021	7669.5	2027	9860.5	2033	12052.5
2022	8034.5	2028	10225.5	2034	12417.5
2023	8399.5	2029	10591.5	2035	12782.5

5. $d_{since\ J2000} = d_{to\ beginning\ of\ year} + d_{to\ beginning\ of\ month}$

The next step is to find the local sidereal time (LST). Sidereal time references the stars instead of the Sun and thus only tracks the rotation of the Earth and not the orbit of the Earth around the sun. A sidereal day is about 4 minutes shorter than a solar day. LST can be approximated by the following formula (Burnett, 1998):

$$LST = 100.46 + 0.985647 * d + long + 15 * UTC$$

Where

$$d = \text{ days from J2000, } d_{since\ J2000} \text{ above}$$

$$UTC = \text{universal time in decimal hours}$$

$$long = \text{observation longitude in decimal degrees, East positive}$$

This results in the local sidereal time in degrees to an accuracy of ~0.3 seconds.

Next, hour angle (HA) in degrees is calculated from LST and RA to account for the rotation of the Earth (Duffett-Smith & Zwart, 2011):

$$HA = LST - RA$$

Finally, Altitude (Alt) and Azimuth (Az) for the time and location of observation can be calculated (Burnett, 1998):

$$\sin(Alt) = \sin(Dec) * \sin(lat) + \cos(Dec) * \cos(lat) * \cos(HA)$$

$$Alt = \text{asin}(Alt)$$

$$\cos(A) = \frac{\sin(Dec) - \sin(Alt) * \sin(lat)}{\cos(Alt) * \cos(lat)}$$

$$A = \text{acos}(A)$$

$$Az = \begin{cases} A, & \sin(HA) < 0 \\ 360 - A, & \sin(HA) \geq 0 \end{cases}$$

This calculation is easily programmable as a standalone program or it could be incorporated into the Parallel Actuator Mount software.

Did you use such software? If so, a link here would be nice.

4. DESIGN

4.1 State of Previous System

The system as received from Mr. Gudgel was a solid proof of concept with a few design choices that were not optimal for this continuation of the project.

4.1.1 Linear Actuators

The linear actuators for the system (Figure 15) received were made up of several parts. Linear movement was generated with an Acme threaded rod rigidly mounted to the output shaft of a DC motor with gearbox. A nut moved along the threaded rod to create the linear motion.

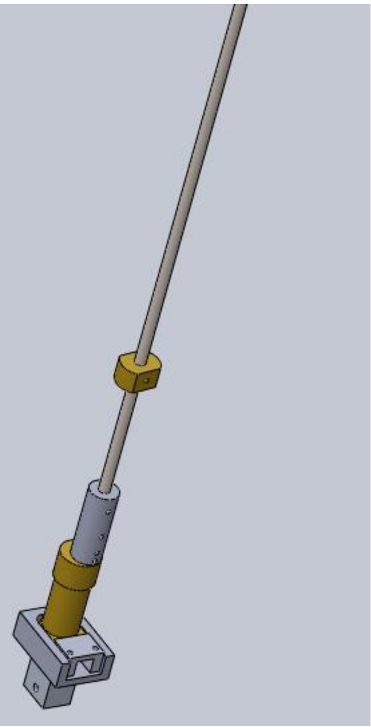


Figure 15: Original Linear Actuator

4.1.2 Ground-to-Actuator Joint Assembly

The ground-to-actuator joint assembly (Figure 16) was the connection between the base plate and the motor. It constrained the motion such that the motor could drive the threaded rod without the motor body spinning. A possible singularity position existed if the motor output shaft was in a vertical position. The geometry of the system prevented this position from being reached (see section 4.3.2.1).

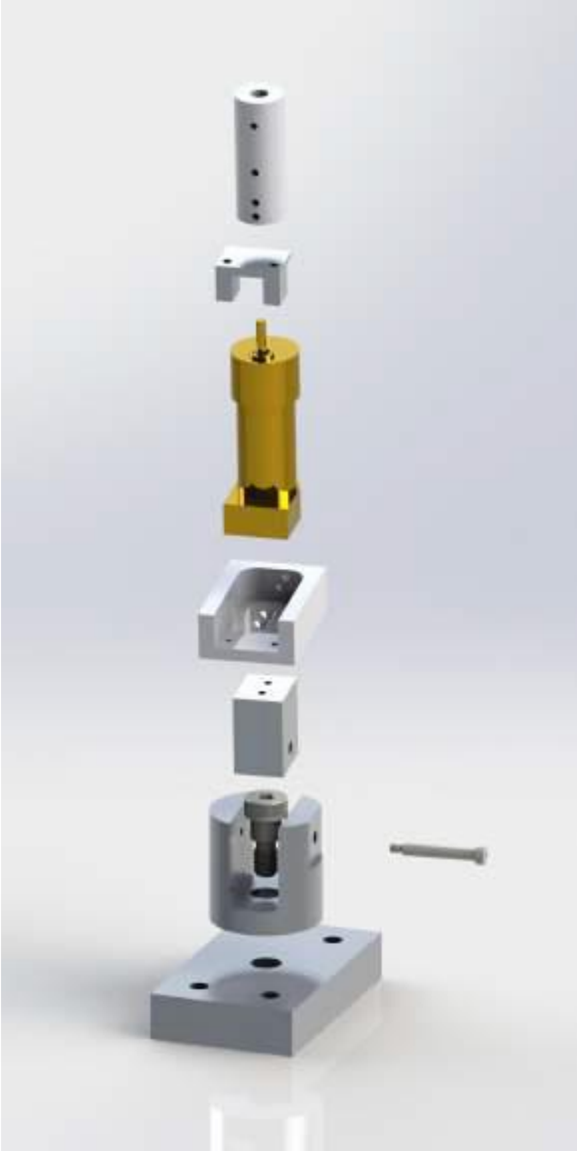


Figure 16: Ground to Actuator Joint Assembly (Gudgel, 2015)

4.1.3 Actuator-to-Telescope Joint Assembly

The actuator-to-telescope joint assembly (Figure 17) connected the nut which moved on the linear actuator to the rigid body of the telescope. The assembly prevented the nut from spinning freely and thus caused it to travel along the threaded rod as the rod spun.

Why is this past tense?

Also, consider discussing the functionality -- which degrees of freedom are restricted and which aren't.

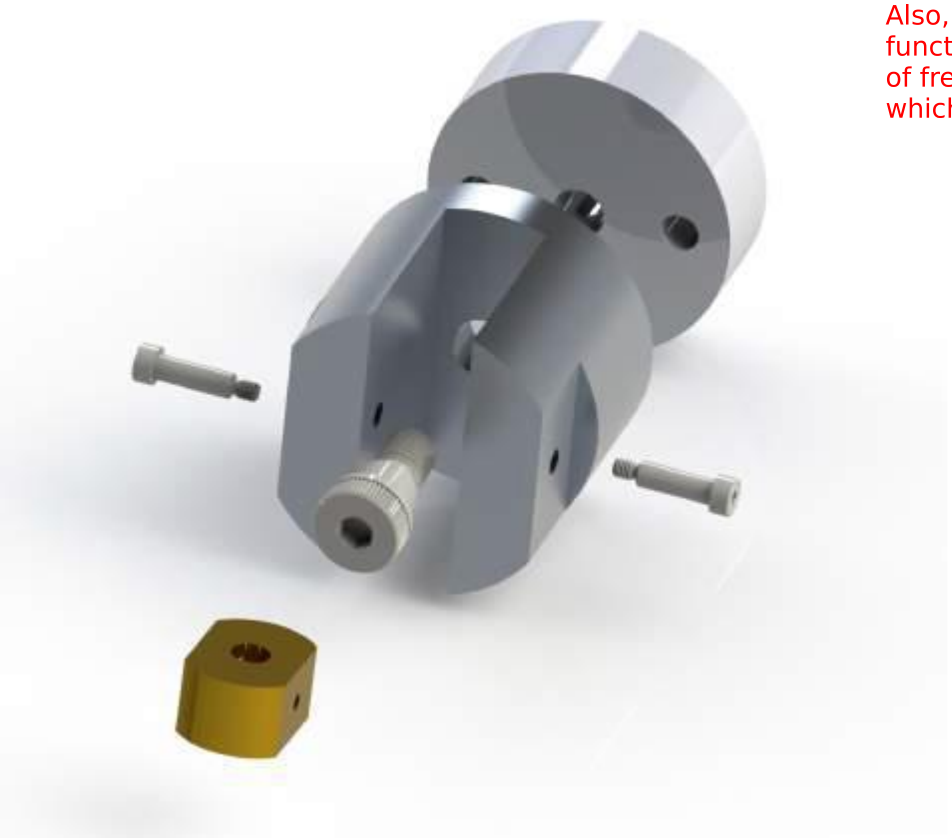


Figure 17: Actuator to Telescope Joint Assembly (Gudgel, 2015)

4.1.4 Telescope Joint Assembly

The telescope joint assembly (Figure 18) was the stationary joint about which the telescope rotated. This joint was the origin for all transformation calculations. It was built from a ball joint rod end, a shaft, and several simple machined parts.

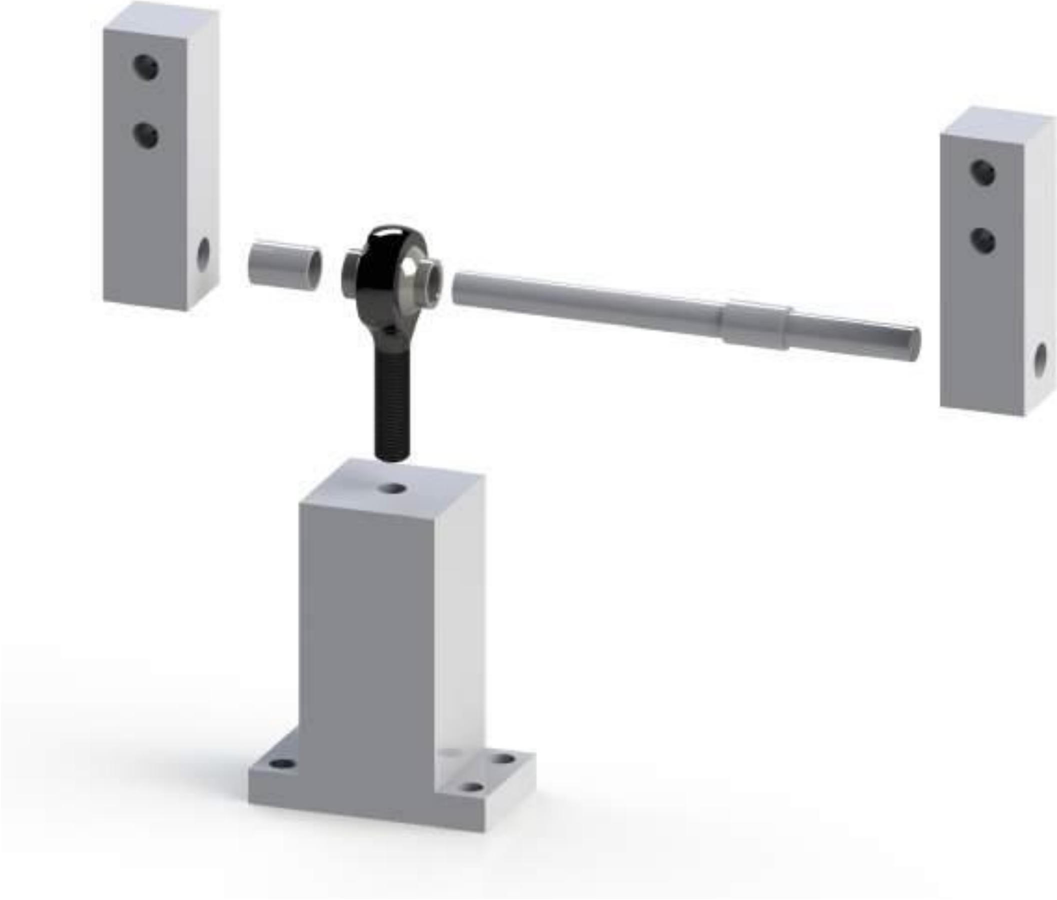


Figure 18: Telescope Joint Assembly (Gudgel, 2015)

4.1.5 Electronics and Software

The electronics provided were built around an Mbed LPC1768 Prototype board which was programmed in mbed C++. It communicated with a partner program on a computer and controlled a custom board (Figure 19) with three DC motor drivers and three encoder counter circuits.

There was also a secondary piece of electronics which controlled the focusing mirror stepper motor (Figure 20).

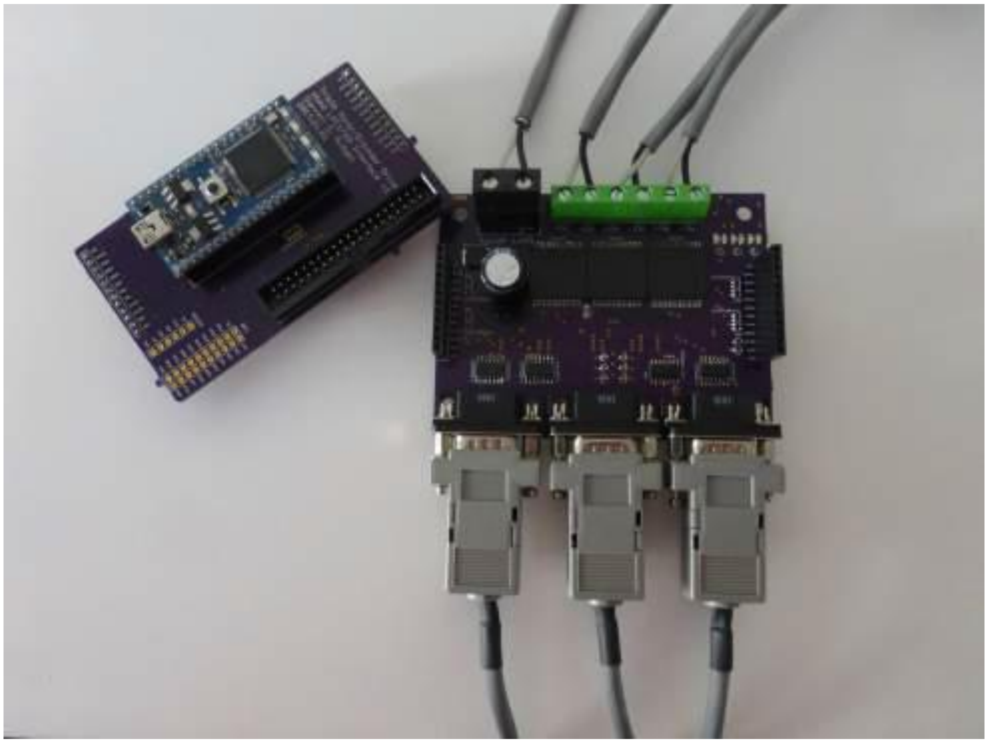


Figure 19: Mbed LPC1768 Prototype board with custom motor driver board installed (Gudgel, 2015)

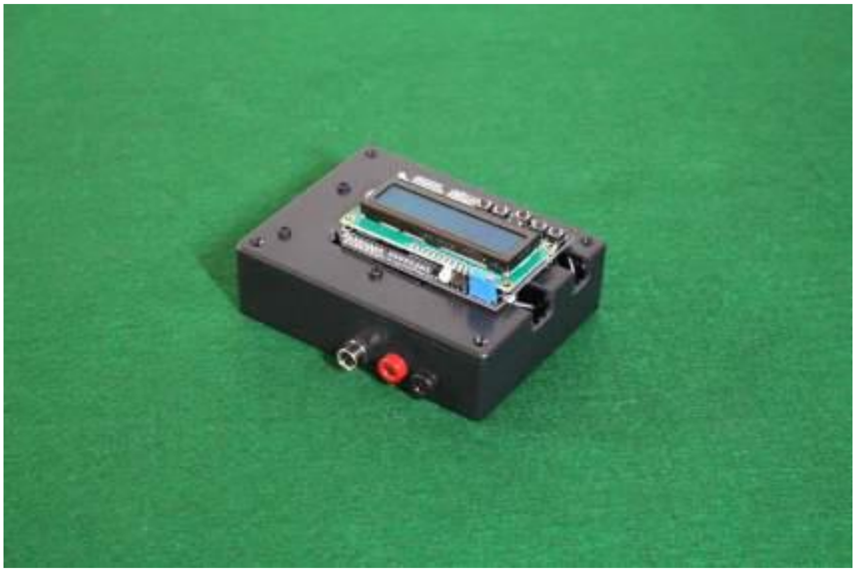


Figure 20: Secondary electronics for control of telescope's focusing mirror (Gudgel, 2015)

Small thing: Single spacing for captions is preferred by me. Not sure what the Grad Division says about it, though.

Along with the software onboard the Mbed, a Python program with user interface was written. This program supported telescope calibration, positioning, and emergency shutoff.

4.2 Implemented Design

4.2.1 Frame

The mounting rails used in the previous system were steel and added significantly to the weight of the system. It was decided to represent the telescope and rails with a T-slotted aluminum extrusion frame. This served as an adequate substitution for the purpose of this project and allowed for smaller stepper motors to be employed.

4.2.2 Linear Actuators

4.2.2.1 Motors

There are two general options for the motors of this system: DC motors with gearboxes and encoders or stepper motors. Other options exist, such as servomotors, but they are generally much more expensive or otherwise do not fit the requirements of the parallel actuator mount.

The original parallel actuator mount utilized DC motors with gearboxes. These were changed to stepper motors to lower cost and simplify the control scheme.

DC motors excel at high velocity, low torque motion. They have no inherent knowledge of their own position and require an encoder for position control. In order to make a DC motor suitable for this system, a gear box and encoder are required. These greatly increase the expense of the system. Gearboxes can also add inaccuracy to the system through the introduction of backlash.

...although the backlash problem can be solved pretty reasonably by mounting encoders on the leadscrews, not the motors. (You make a good point but it should be noted that backlash doesn't have to add inaccuracy to the system with a good encoder, decent motor, and good controller.)

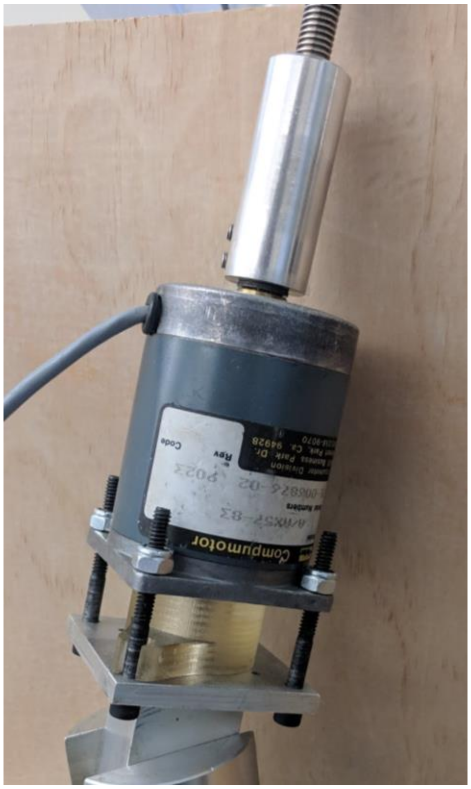


Figure 21: Stepper Motor Assembly

often (not always)

Stepper motors are the least expensive method of implementing precise angular motion. They are used in many industries including having a strong presence in the astronomy field (Anaheim Automation). Stepper motors, in conjunction with high quality drivers, are very simple to control for both position and velocity. Stepper motors do not require an encoder which creates a simpler, cheaper control system. Stepper motors excel at outputting high torque at low speeds without requiring the use of gearboxes.

The largest issue with the change to stepper motors is that it shifts closed loop control from the motor/encoder to a telescope camera and plate solver. As the camera and plate solver are beyond the scope of this project, the system must be run with an open loop control scheme and it must be assumed that a commanded position change occurs instead of being able to track the change. This is a decent assumption if the stepper motors are rated appropriately for the system requirements.

Selecting a stepper motor requires knowledge of the torque requirements, accuracy requirements, and velocity requirements of the system.

For this system, the torque requirements can be calculated based on the torque required to raise a load using a screw (Budynas & Nisbett, 2015):

This is an Acme screw in particular, right?

$$T_r = \frac{Fd_m}{2}\left(\frac{\ell + \pi * fd_m}{\pi d_m - f\ell}\right)$$

Where:

$$T_r = torque\ required\ to\ raise\ the\ load$$

$$F = axial\ load$$

$$d_m = mean\ diameter\ of\ screw$$

$$f = coefficient\ of\ friction\ between\ screw\ and\ nut$$

$$\ell = lead\ of\ screw, advancement\ per\ revolution$$

Unfortunately, accurately measuring the axial load on the actuator is very difficult. An estimate was established through use of a pull scale. The scale was attached to the actuator-to-telescope joint assembly and pulled axially in the direction of the acme rod until the nut was no longer under load. The highest reading found was approximately 11 kg which would indicate a torque requirement of 0.10 Nm. The stepper motor selected is an LA23BCK-121RB stepper motor from Eastern Air Devices. It is a 9 volt stepper with 200 steps per revolution and about 0.5 Nm of torque at speeds less than 100 steps/second (see Appendix E).

4.2.2.2 Lead Screws

Lead screws are used to translate rotational motion into linear motion. There are two primary options for lead screws: threaded and ball screws. Changing from threaded screws to ball screws was considered.

The most common thread form used for power transmission is Acme thread. Acme thread is a trapezoidal thread with a thread angle of 29°. It is a very common thread for power and motion transmission due to its high strength and simplicity (Budynas & Nisbett, 2015). Acme thread is a high friction thread and has low mechanical efficiency. Plural? They are self-locking which means an axial load will not cause the thread to turn: They can translate rotational motion into linear motion easily but in most circumstances cannot translate linear motion into rotational motion. This allows for the elimination of a brake from the system. ...or it reduces the holding torque requirements on stepping motors. Note that permanent-magnet-hybrid motors such as ours have a bit of power-off holding torque.

Ball Screws are made up of a shaft with a helical raceway and a ball bearing assembly. Ball bearings within the ball bearing assembly roll along the raceway. They have lower friction than Acme screws but tend to be noisier. Ball Screws are more efficient at transmitting power but do not self-lock (Matara UK). They don't _necessarily_ self-lock, depending on the pitch. And dirt. :)

It was decided to continue with the existing Acme threaded lead screws for their lower XX cost, ability to self-lock, and general simplicity.

4.2.3 Bearings

All of the joint assemblies have aluminum against aluminum interfaces over a relatively large surface area and steel shafts which move directly against aluminum. Both of these interfaces have large frictional coefficients (Friction and Friction Coefficients). Is this a reference? These interfaces were all greased to lower the frictional coefficient and help prevent binding. It would be ideal to redesign the system with appropriate bearings (Section 4.3.1) but that was deemed beyond the scope of this project.

4.2.4 Electrical Modifications

The electronics were replaced with off-the-shelf products as much as possible. This was accomplished using STMicroelectronics' series of microcontrollers which allowed access to a wide range of STM32 microcontrollers and allowed for the easy incorporation of a wide variety of expansion boards. These boards are designed to be programmed in many different languages.

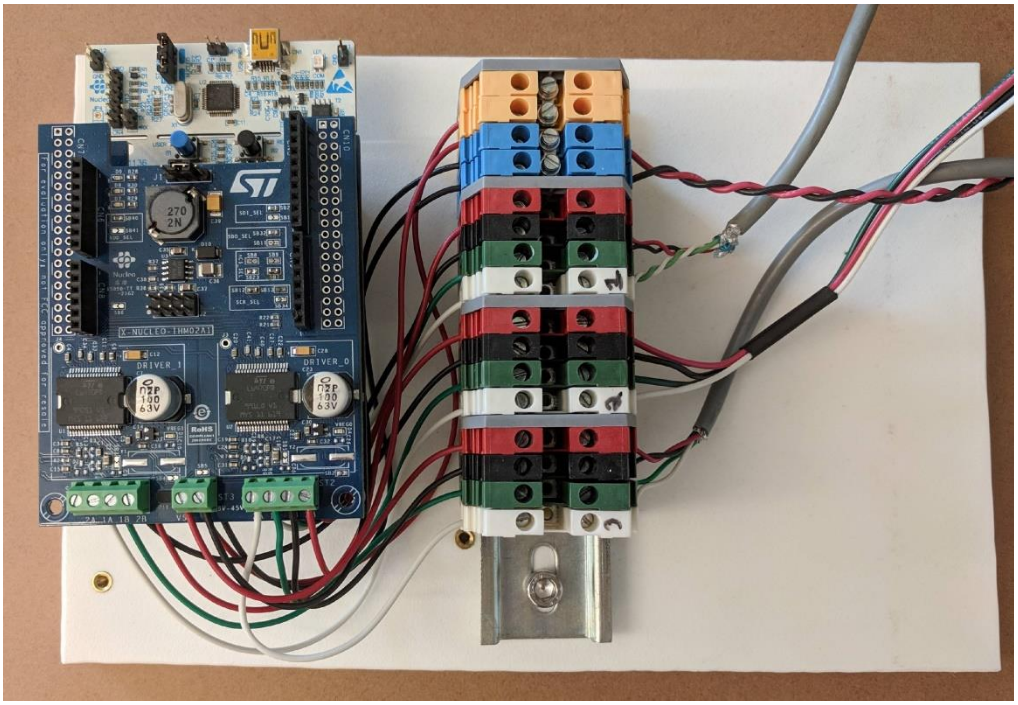


Figure 22: Fully Assembled Electronics Stack

4.2.4.1 STM32 Nucleo Development Board with STM32L476RG MCU

The STM32L476RG is an 80MHz, 32-bit ultra-low-power microcontroller with a built in floating point unit. It was selected over other models of the STM32 family due to its 1 MByte of Flash memory (STMicroelectronics).

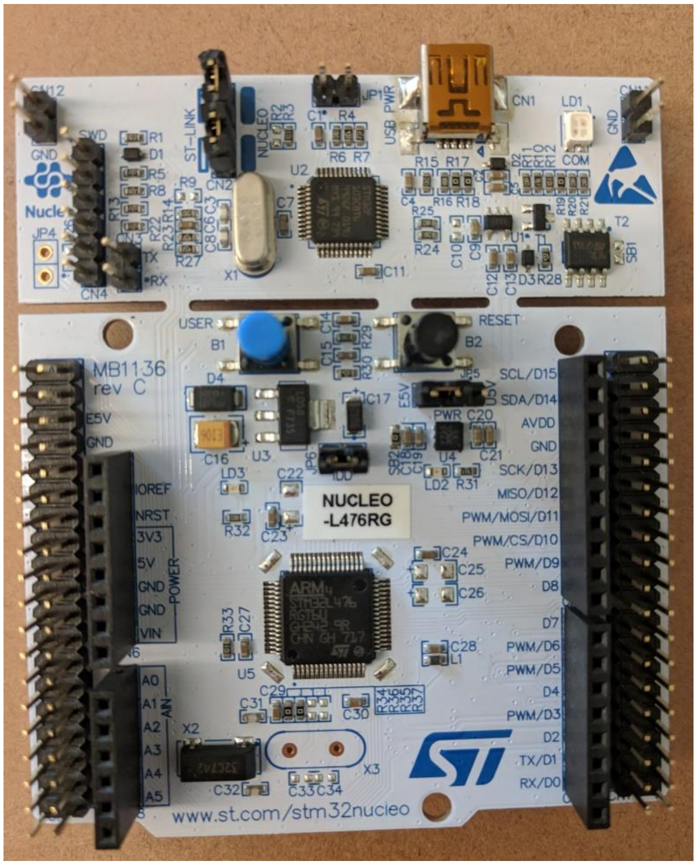


Figure 23: ST Nucleo L476RG

4.2.4.2 ST L6470 Stepper Motor Drivers

The L6470 stepper motor driver from ST is a fully integrated bipolar stepper motor driver with microstepping (STMicroelectronics). This driver is communicated with over I^2C ~~i2c~~ and is “smart” due to an onboard microprocessor that handles the stepper motor feedback and control. It allows for simple commands to be sent such as “Run”, “Move”, and “GoHome”. This greatly simplifies the control scheme for the project.

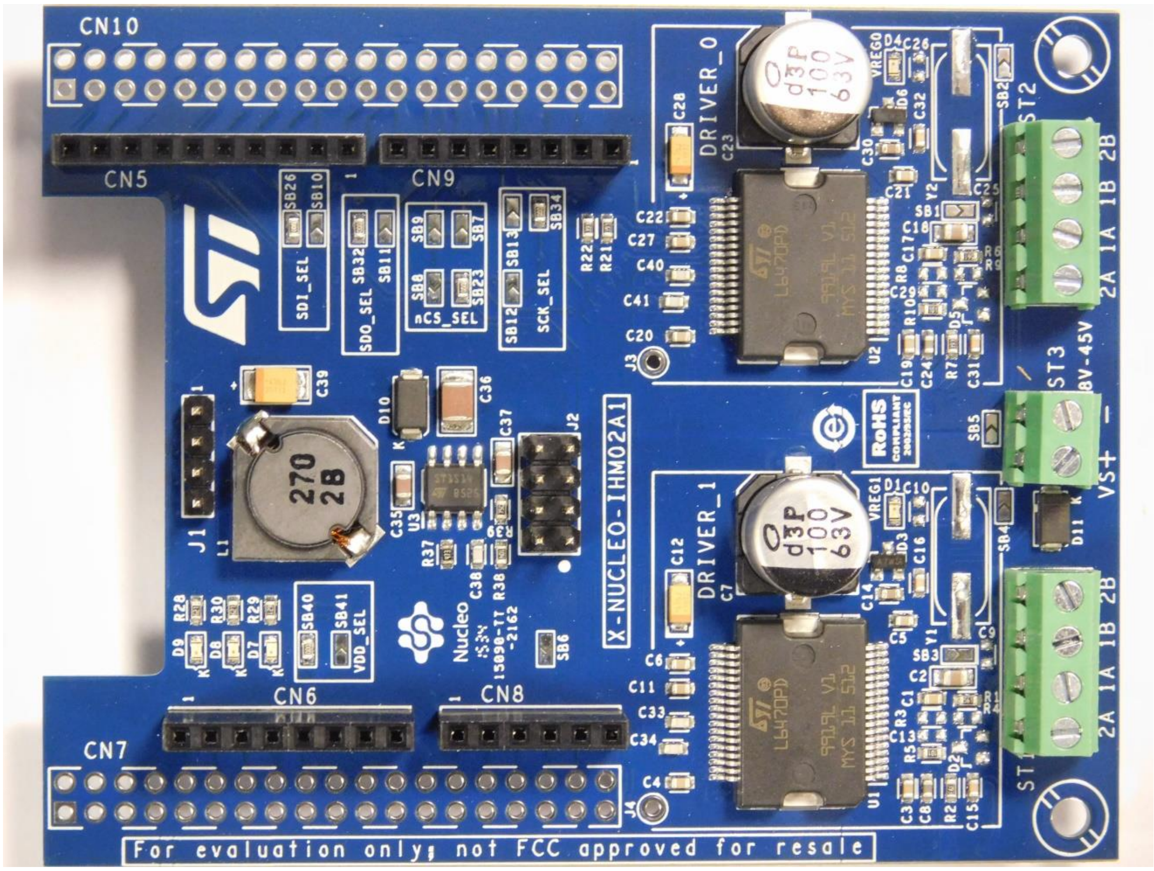


Figure 24: ST L6470 Stepper Motor Driver

4.2.4.3 Shoe of Brian

The Shoe of Brian is the only custom circuit board required for this system. It is a simple board which allows for the STM32L476RG to run Micropython instead of its default Mbed. It was designed by Dr. John Ridgely for use in his mechatronics classes (Ridgely).

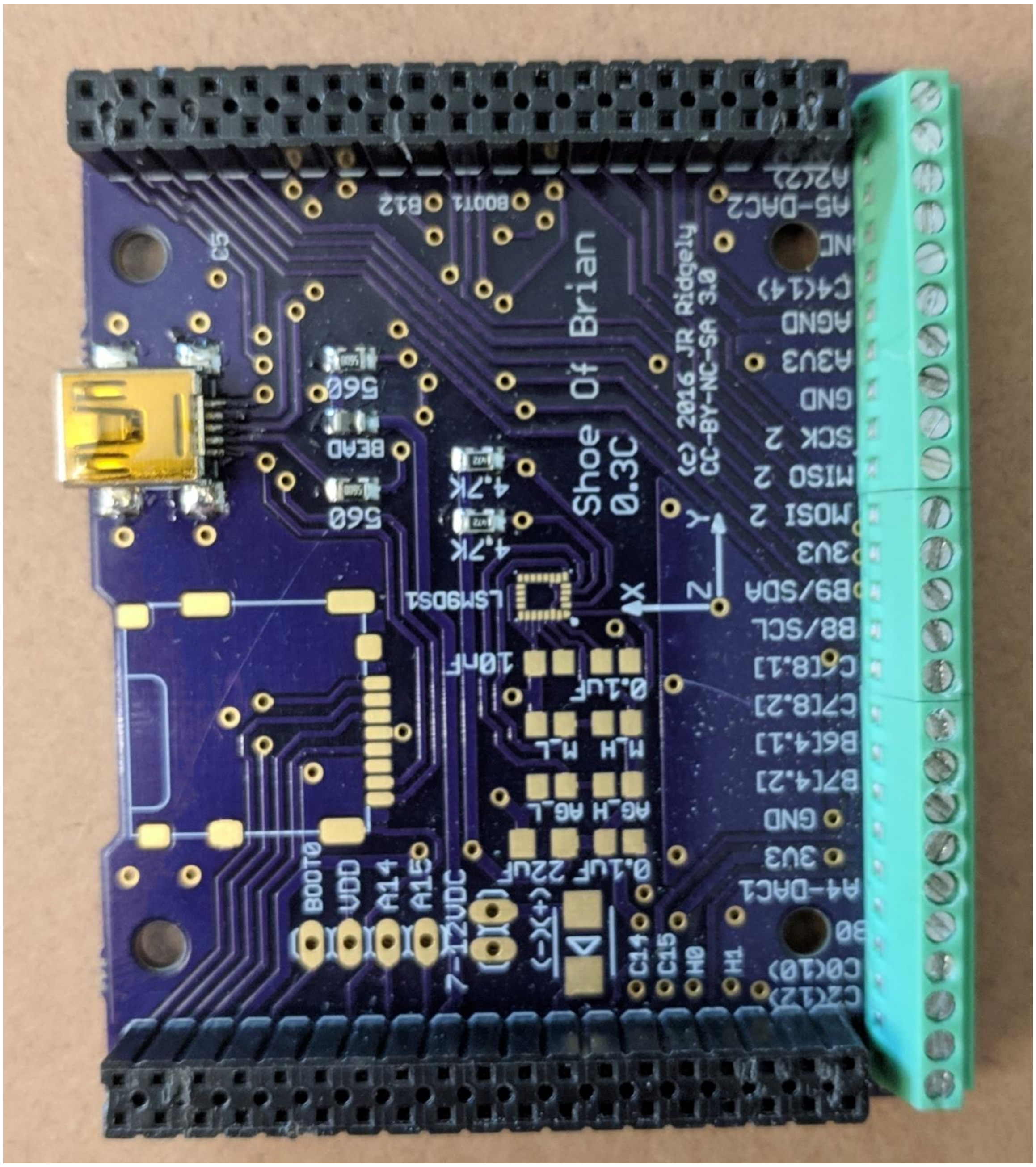


Figure 25: Shoe of Brian Custom PCB

4.2.5 Software Redesign

This project is developed in Micropython which is an implementation of Python designed to run on microcontrollers. It includes a subset of the standard Python libraries.

Micropython allows for easier development with code that is straightforward to follow and easy for other users to modify to fit their needs.

4.2.5.1 Program architecture

The control program is constructed as a series of nested objects. These objects include the L6470 driver, the actuator driver, and the telescope driver.

The lowest level object is found in l6470nucleo.py (see page 90). The Dual6470 class contains methods for communicating directly with the stepper motor drivers. The L6470 is a "smart" stepper motor driver so the software does not have to implement a closed-loop control system for the stepper motors. Instead appropriate commands must be sent to the L6470 and the on-chip firmware handles the implementation.

The Actuator class is found in ActuatorDriver.py (see page 96). Each instance of this object represents one of the linear actuators in the system. It requires an associated L6470 object and information about how the system changes the rotational motion of the stepper motor to linear motion. In particular, this class requires the ratio of stepper motor steps to distance traveled. Depending on the system, this could be as simple as:

$$stepper\ motor\ resolution * \frac{1}{leadscrew\ lead}$$

$$\left[\frac{steps}{revolution}\right] * \left[\frac{revolution}{distance\ traveled}\right] = \left[\frac{steps}{distance\ traveled}\right]$$

The Telescope class in TelescopeDriver.py (see page 99) represents one full Parallel Actuator Mount. This class requires three Actuator objects. This is where the geometry of the system is defined and where the transformations in Section 3.1 are implemented. It is also where the velocity calculations from Section 3.6 are implemented. The basic set of commands for controlling the whole system is defined at this level.

The final layer of the program is main.py (see page 103) which sets up all the above objects and allows for definition of custom commands (see 4.2.5.2).

4.2.5.2 Program Usage

A minimal set of commands have been defined for user interaction with the telescope. Basic commands for control of the telescope are at the Telescope object level. These include commands such as *goVelocity,* which commands the telescope to move at specified angular velocities, *goToAngles*, which calculates the correct lengths of the linear actuators for the input angles of interest then commands the calculated motion, and *resetPosition*, which defines the current position as the home position.

Customizable commands are contained in main.py. Several commands were setup for testing, including things such as:

- *Estop* – Stops each motor as quickly as possible. Directly interacts with the L6470 drivers.
- *Go* – Moves each stepper motor at a specified speed.
- *testRepeatability* – Runs through a series of points multiple times for the Long Run Repeatability test found in Section 5.3.3.
- *testGrid* – Runs through all the points in a grid, pausing in between each point.

Using these commands is very simple. For example, in order to run the repeatability test:

- Connect to the Nucleo
- Home the telescope
    - o Type “Telescope.goVelocity(x, x, x)” where x is a negative linear velocity appropriate to your system.

    - Once the actuators are at their hardstop, type “estop()” followed by “Telescope.resetPositions()”
- Run the repeatability test
    - Type “testRepeatability()”
        - The telescope will begin moving to the first point and pause on arrival.
        - Mark the location of the point.
        - Press any key on the keyboard to start the telescope moving to the next point.
        - Repeat for all points.
    - Once all points are marked, the telescope will begin cycling through the points.
    - Type “testRepeatability()” again and do the beginning steps from above.
        - The difference between the first set of points and the later set is the repeatability error.

4.3 Design Guidelines

The Parallel Actuator Mount has three general parts that need to be taken into account when designing a mount for a specific purpose: joint assemblies, linear actuators, and system geometry. These guidelines result from the experience of testing the Parallel Actuator Mount prototype. Their implementation can help accelerate the design process for future mounts.

4.3.1 Bearings and Coupler

The joint assemblies and linear actuators are made up of moving parts which experience loading. As such, they should be adequately designed for the loads they experience including the use of bearings/bushings as described below.

4.3.1.1 Joint Assemblies

The joint assemblies in this system are made of two revolute joints set perpendicular to each other. Each of these joints should be supported by appropriate bearings. For the ground-to-actuator joint assemblies (Figure 26), the bearings on the joint whose axis of rotation is parallel to the Y axis must support both axial and radial loads. The bearings on the joint whose axis of rotation is perpendicular to the Y axis primarily need to support radial load as the axial load should be very minimal. The actuator-to-telescope joint assemblies (Figure 27) see similar loading.

The joint assemblies are one of the biggest possible sources of misalignment and error in the system. They require tight tolerances and high quality bearings.

This is a bit too vague, I think, to be accepted by an MS committee. Specifying a type of bearings (angular contact? tapered roller? what do you think?) and a configuration (I think that having two bearings mounted to oppose each other axially, with the ability to adjust the force squeezing them together -- like bicycle hubs -- would be good, but again, your opinion is important here) would be a minimally acceptable level of detail. Having a full design would be ideal, but you can probably get away without that.

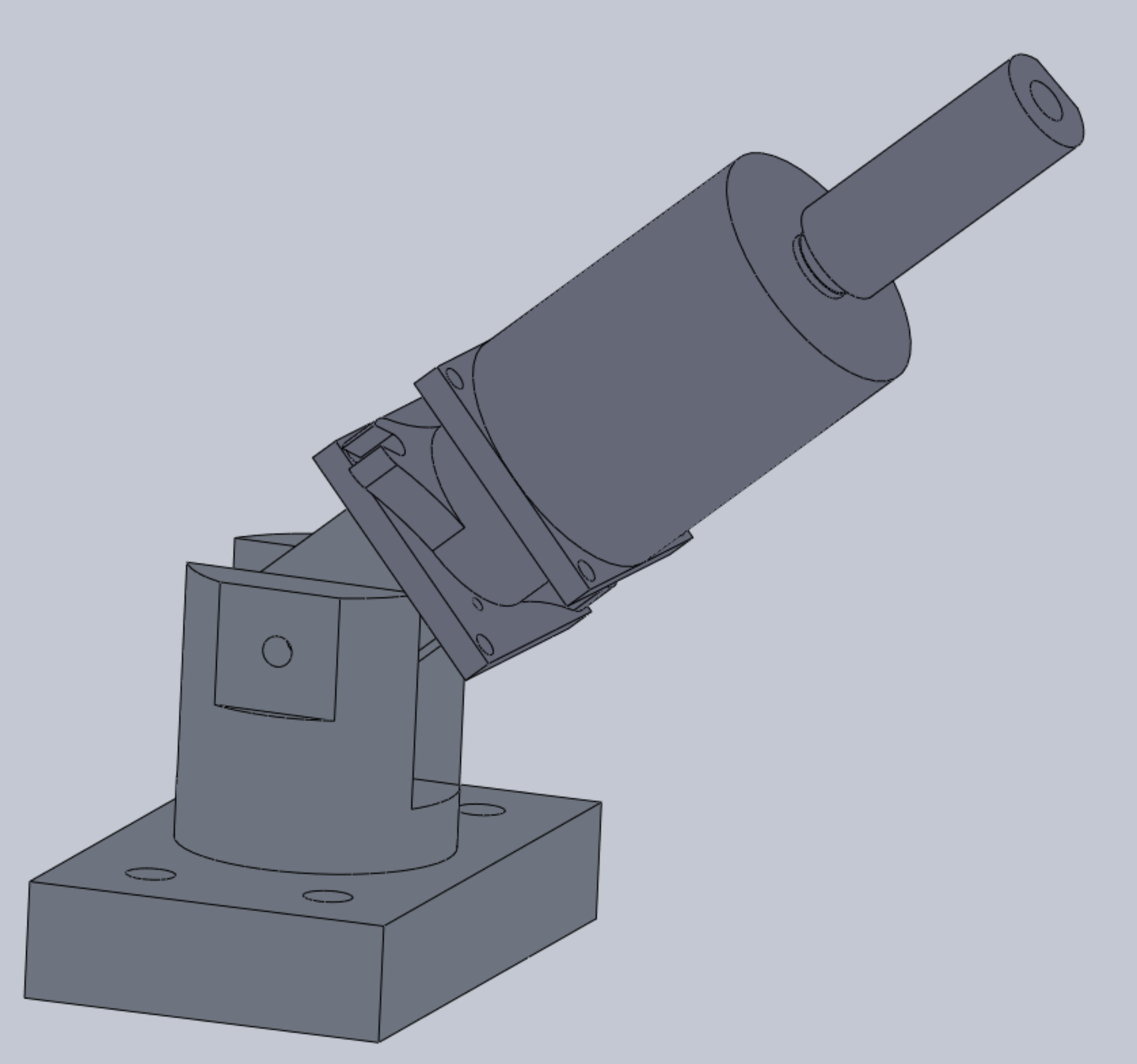


Figure 26: Ground to Actuator Joint Assembly

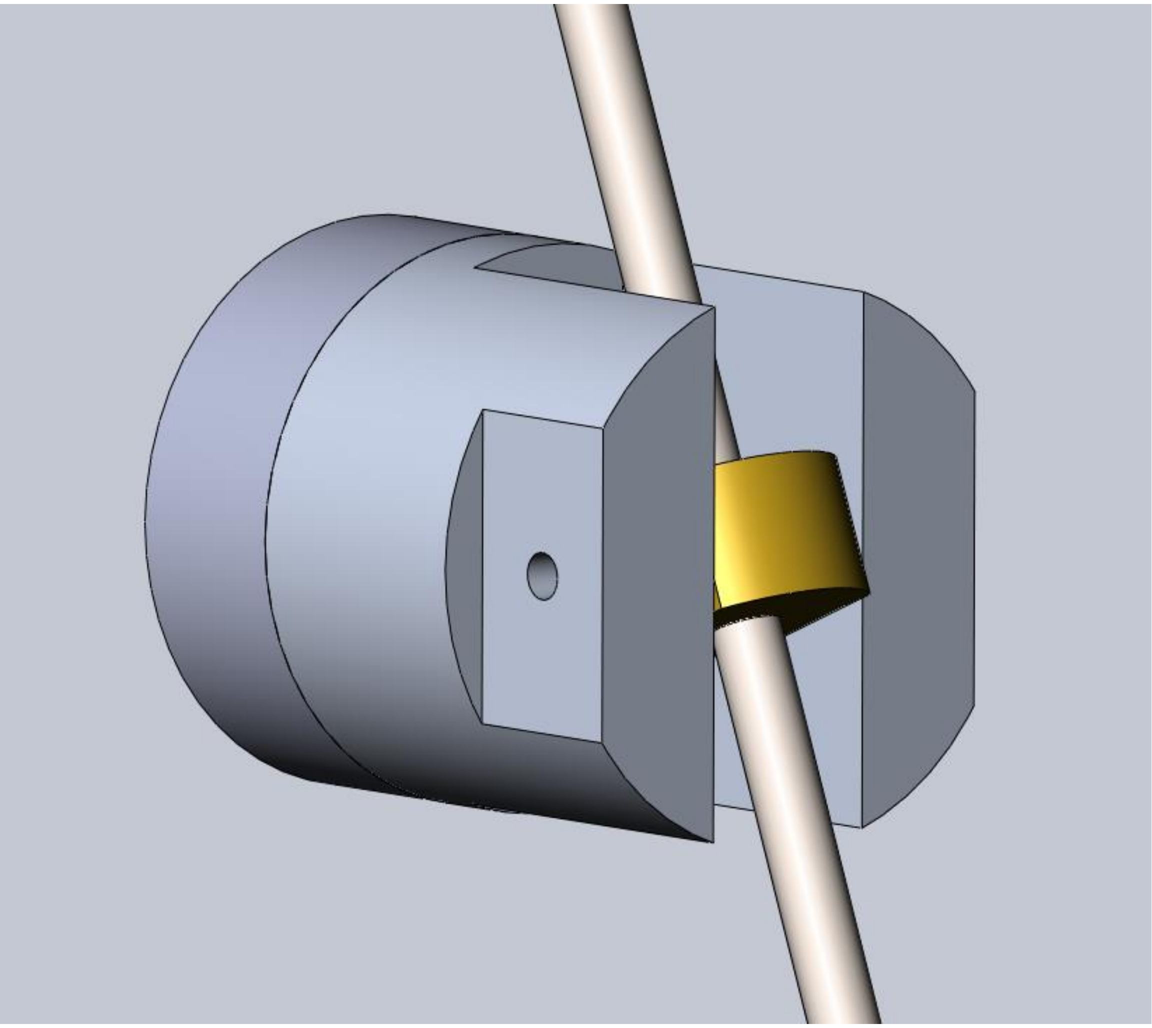


Figure 27: Actuator to Telescope Joint Assembly

4.3.1.2 Linear Actuator

Generally, motors are designed to experience minimal axial loading and bending moments. As such, bearings should be arranged which take the load on the lead screw and transmit it into the joint assembly rather than having the motor directly in the load path. To further minimize undesirable motor loading, those bearings should also prevent the motor from ~~seeing~~ having to support a bending moment. Although motors can withstand some axial loading and bending moments, it is best to design to minimize this. This can be as simple as a pair of angular contact bearings supporting the lead screw as in Figure 28.

Aha, this is the sort of clear concept specification which I think is a good level of detail to provide. Even better (of course I have to ask!) would be showing a way to hold those bearings in place...snap ring and spacer ring? Also, why are these both angular contact...do they both support an axial load?

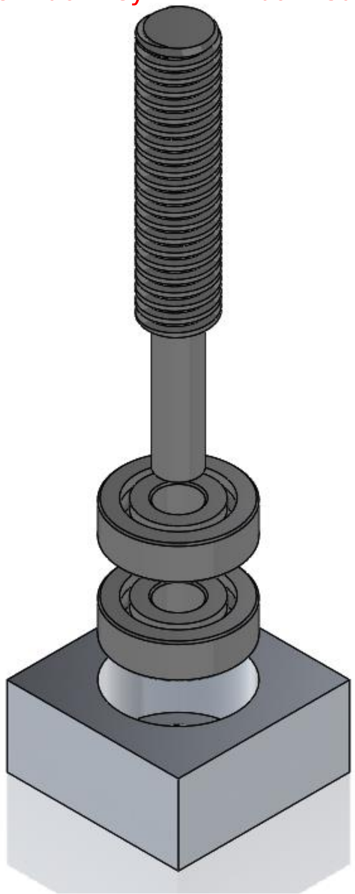


Figure 28: Simple bearing arrangement to take axial and moment load

4.3.2 Geometry

Several items should be considered when designing the geometry of the system.

4.3.2.1 Singularity points

Care should be taken to avoid including singularity points within the envelope of the system. The primary concern is for the two forward ground-to-actuator assemblies. The axis of the leadscrew ~~motor~~ must not become aligned with any axis of rotation of the ground-to-actuator assembly. The acceptable angle away from "aligned" depends largely on the machining tolerances of the ground-to-actuator assemblies. Assemblies with less slop can approach closer to alignment without difficulty.

4.3.2.2 Center of Mass

The center of mass of the telescope should remain within a triangle formed by the telescope joint assembly and the forward two ground-to-actuator joint assemblies. This guideline should be followed even when the telescope is at its most extreme positions. This is necessary to ensure that the linear actuators remain in compression. Compression is desirable for two reasons:

1) If the lead screws transition to tension, backlash becomes an issue and accuracy is lost
2) A proper bearing setup, as described in Section 4.3.1, easily protects the motor from one direction of axial load only,; changing the loading could cause the motor to again see axial loading.

spacing?

5. TESTING AND VERIFICATION

spacing?

All tests have been performed using laser diodes mounted to the front of the aluminum extrusion frame. Three lasers were required to perform all the tests. The primary laser was used for repeatability tests and relative angular motion tests. This laser past was mounted on an axis parallel to the telescope optical axis and present goes through the center of the pivot point. The alignment of this laser was not critical for repeatability tests but was vital for relative motion tests. Deviation from the described positioning can have a major impact on comparative measurements.

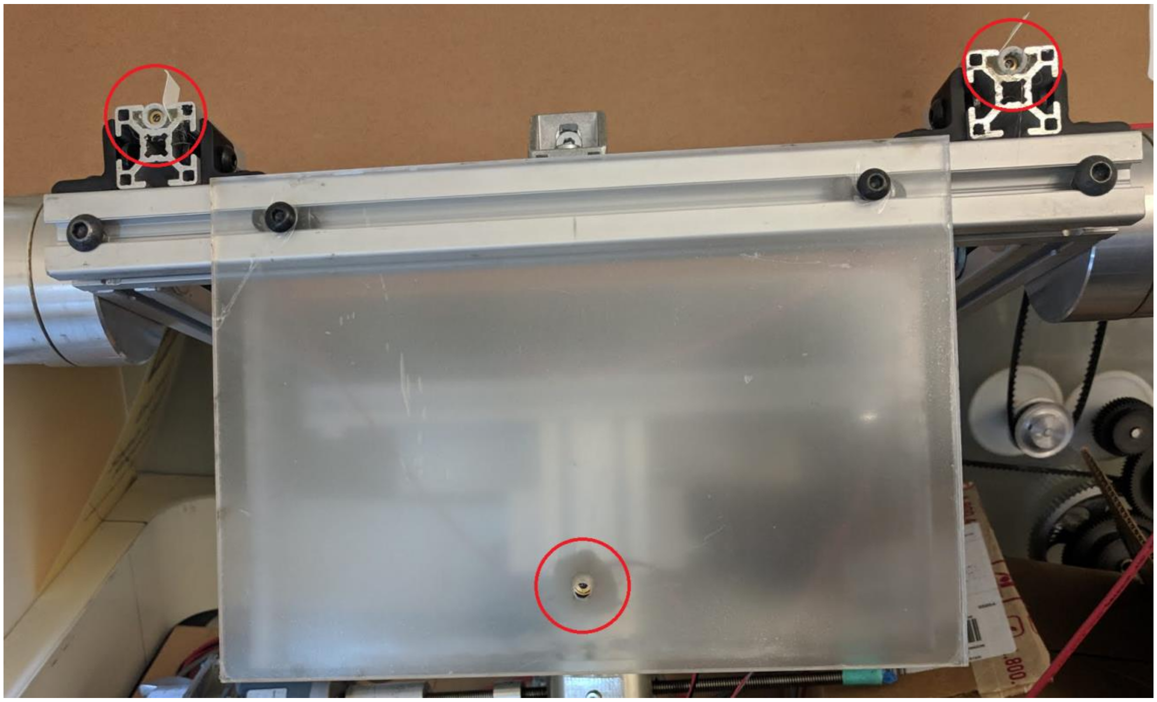


Figure 29: Laser diodes mounted for testing

The second and third lasers were used for measuring relative image rotation angle. They were also mounted parallel to the telescope optical axis but they did not need to go through the center of the pivot point. These two lasers were on the same level such that if the telescope were pointed at a wall with all angles at 0, the two marks would be horizontal.

The apparatus was oriented relative to a vertical surface (wall) with the XY plane parallel to the wall and the XZ plane perpendicular to the wall. The origin of the apparatus was as far as possible from the wall, such that the optical axis laser remained on the wall at the extremes of the desired testing area.

5.1 Relative Positioning

The current system does not have an external absolute reference for its positioning, so all position testing must be done as relative testing from the defined home position. This is sufficient for the purpose of this thesis because a future refinement would be to incorporate feedback via a plate solver.

Hmm...was plate solver defined back in the introduction?

5.1.1 Rotation

The relative rotation test is designed to test the relative accuracy of commanded image rotations. This test utilizes two laser diodes mounted on the front of the telescope. The telescope is commanded to a position with zero image rotation angle, then the two lasers are marked on the wall. This will be the reference angle. Without moving the telescope base, it is commanded to another position with the same altitude and azimuth but different image rotation. The lasers are again marked on the wall. These sets of points are connected to create two lines which should be at the commanded image rotation angles relative to the horizontal. A photograph of these lines is then taken to be analyzed.

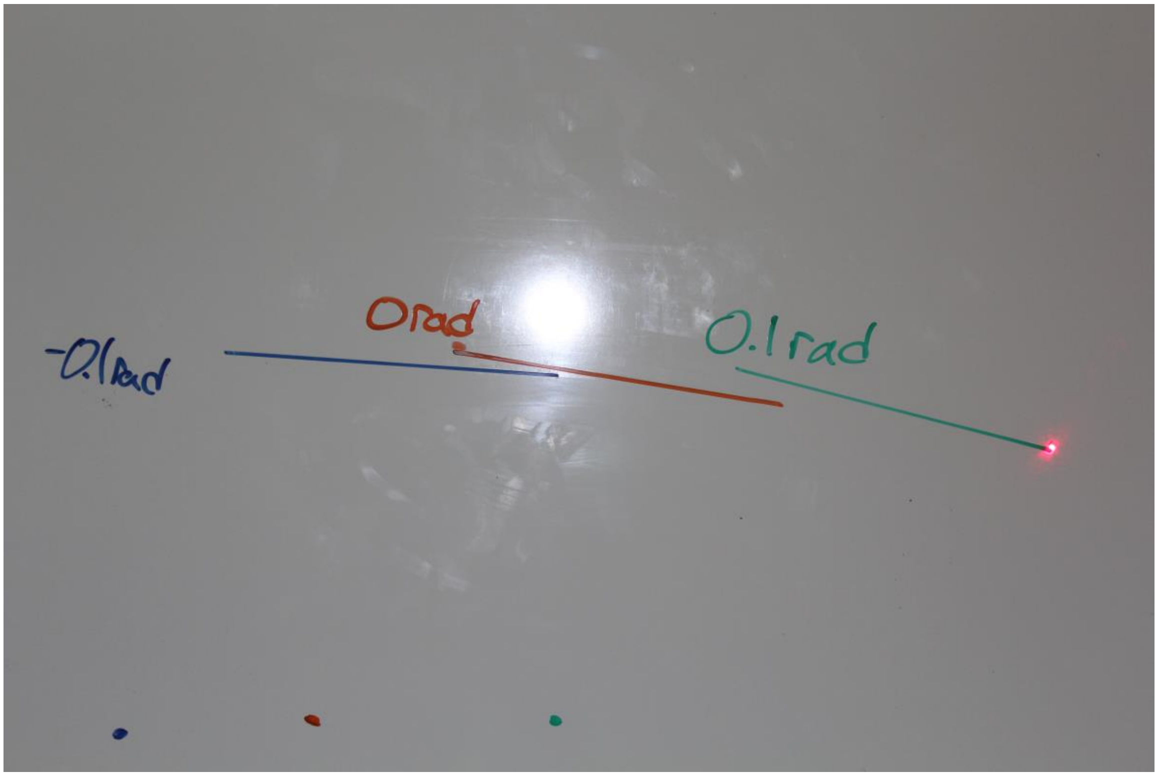


Figure 30: Relative Rotation Test

The angle of the reference line and the angle of the other lines position are measured via Matlab (see page 132). As shown in Table 1, the measured angle of the reference line can be subtracted from the angle of the other lines in order to calculate a corrected, relative angle.

Table 1: Relative Rotation Test Results

	Commanded Angle [rad]	Measured Angle [rad]	Corrected Angle [rad]	% Error
Zero Line	0	0.1655	0	
Positive Line	0.1	0.2651	0.0996	-0.40%
Negative Line	-0.1	0.0698	-0.0957	-4.30%

5.1.2 Altitude

The Altitude Relative Position Test used a series of commanded angles to measure how accurate the system is at changing altitude. The desired angles were commanded and marked on the board. The distance of these positions above the reference commanded point was measured as well as the orthogonal distance from the board to the system origin. As the system is not physically able of pointing at altitude = 0°, the reference commanded point was chosen as 17°.

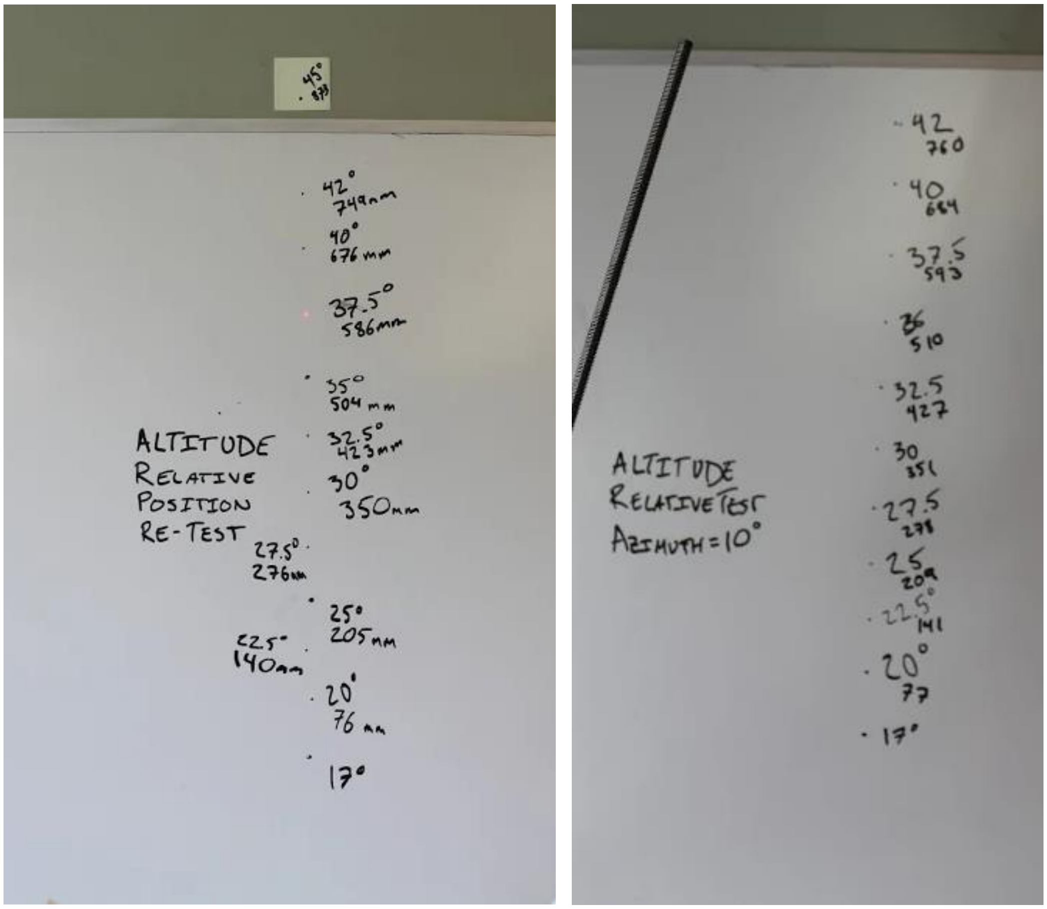


Figure 31: Altitude Relative Position Test, 0° Azimuth and 10° Azimuth. Numbers below angles are millimeters above 17°

The distance to the board along with the commanded angles were used to calculate the theoretical height the system would point above altitude = 0° (see Figure 32: Theoretical Height Above Zero):

$$heightAboveZero_{theoretical} = distanceToBoard_{orthogonal} * \frac{\tan \theta_{alt}}{\cos \theta_{az}}$$

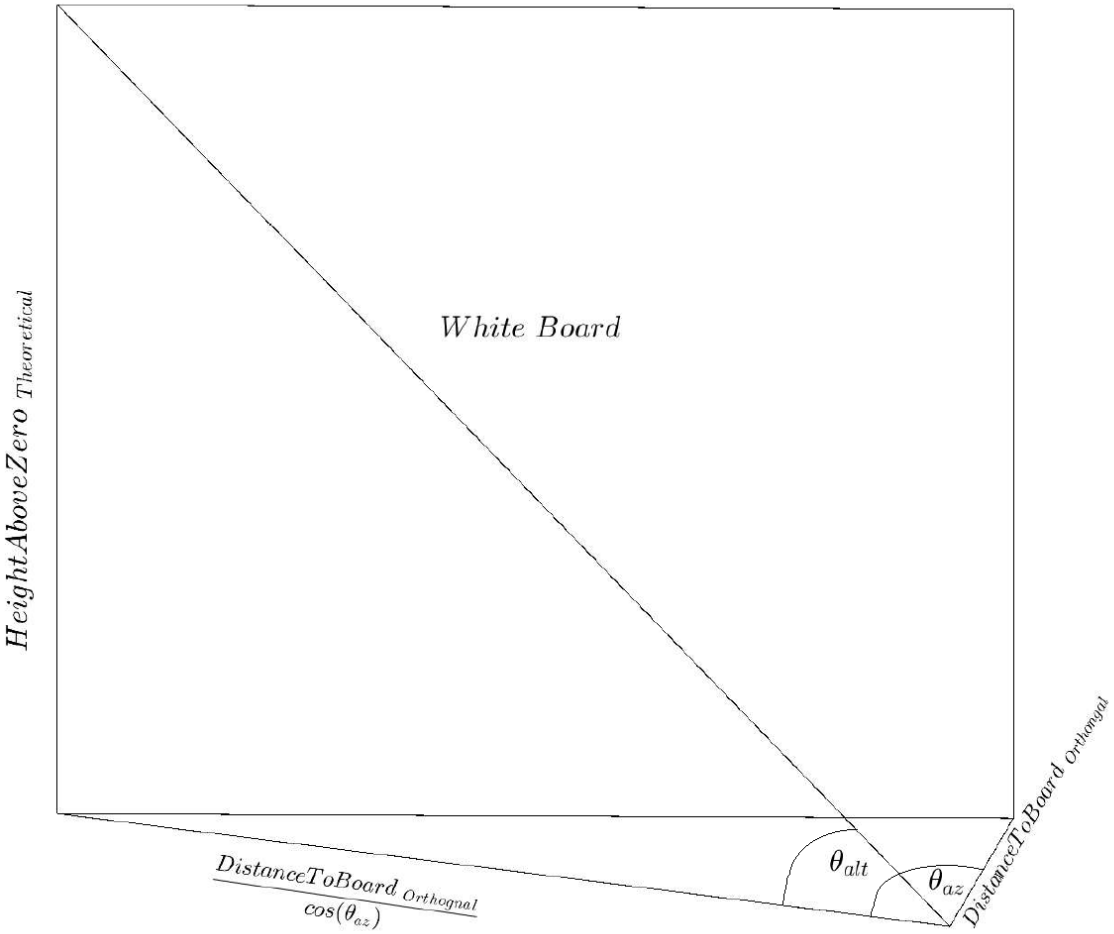

White Board
$HeightAboveZero_{Theoretical}$
$\frac{DistanceToBoard_{Orthognal}}{cos(\theta_{az})}$
θ_{alt}
θ_{az}
$DistanceToBoard_{Orthongal}$

Figure 32: Theoretical Height Above Zero

For each point, the distance measured above the reference point was subtracted from the theoretical height to find where the reference point would be assuming the current point is in the correct spot:

$$heightZeroToRef = heightAboveZero_{theoretical} - heightAboveRef_{measured}$$

When this height is close to the theoretical height of the reference point above zero, the system is performing well.

Table 2: Altitude Relative Position Test. Azimuth = 0°, DistanceToBoard = 1295mm

Command Angle [deg]	Theoretical Height Above Zero [mm]	Measured Height Above Ref [mm]	Height from Zero to Ref [mm]	Calculated Angle [deg]	Angle Error [deg]
17	395.92	0	395.92	17.0000	0.0000
20	471.34	76	395.34	20.0226	0.0226
22.5	536.41	140	396.41	22.4816	-0.0184
25	603.87	205	398.87	24.8928	-0.1072
27.5	674.13	276	398.13	27.4229	-0.0771
30	747.67	350	397.67	29.9419	-0.0581
32.5	825.01	423	402.01	32.3081	-0.1919
35	906.77	504	402.77	34.7962	-0.2038
37.5	993.69	586	407.69	37.1708	-0.3292
40	1086.63	676	410.63	39.6158	-0.3842
42	1166.02	749	417.02	41.4802	-0.5198
45	1295.00	873	422.00	44.4172	-0.5828

Maybe reducing inter-table space helps? ...Nice tables, BTW.

Table 3: Altitude Relative Position Test. Azimuth =10°, DistanceToBoard = 1295mm

Command Angle [deg]	Theoretical Height Above Zero [mm]	Measured Height Above Ref [mm]	Height from Zero to Ref [mm]	Calculated Angle [deg]	Angle Error [deg]
17	402.03	0	402.03	17.0000	0.0000
20	478.61	77	401.61	20.0616	0.0616
22.5	544.68	141	403.68	22.5194	0.0194
25	613.18	209	404.18	25.0382	0.0382
27.5	684.53	278	406.53	27.4925	-0.0075
30	759.20	351	408.20	29.9752	-0.0248
32.5	837.73	427	410.73	32.4343	-0.0657
35	920.76	510	410.76	34.9748	-0.0252
37.5	1009.02	593	416.02	37.3670	-0.1330

40	1103.40	684	419.40	39.8252	-0.1748
42	1184.01	760	424.01	41.7522	-0.2478

Is there some way to set the word processor to force tables to all be on one page? I consider a little extra use of (virtual) paper better than a broken table. Also, I hate trees. They're so smug and green.

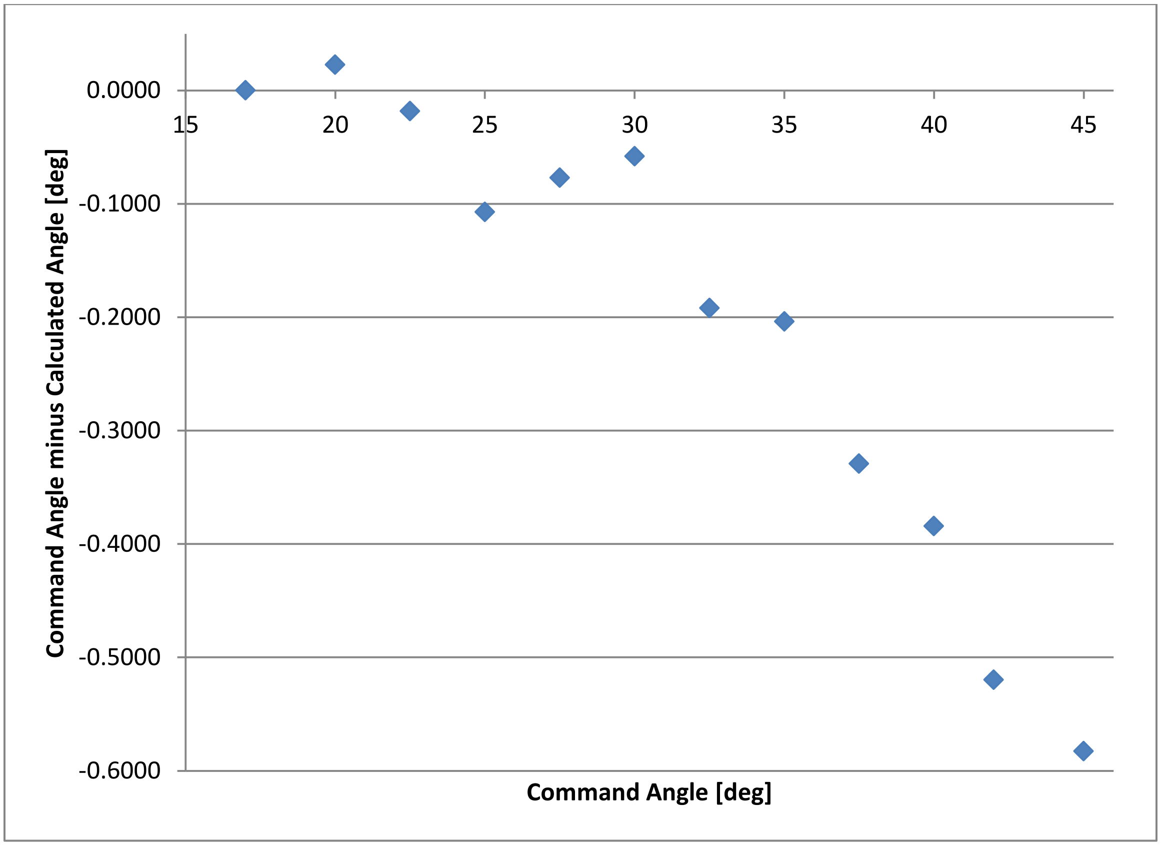


Figure 33: Altitude Relative Position Test. Azimuth = 0°, DistanceToBoard = 1295mm.

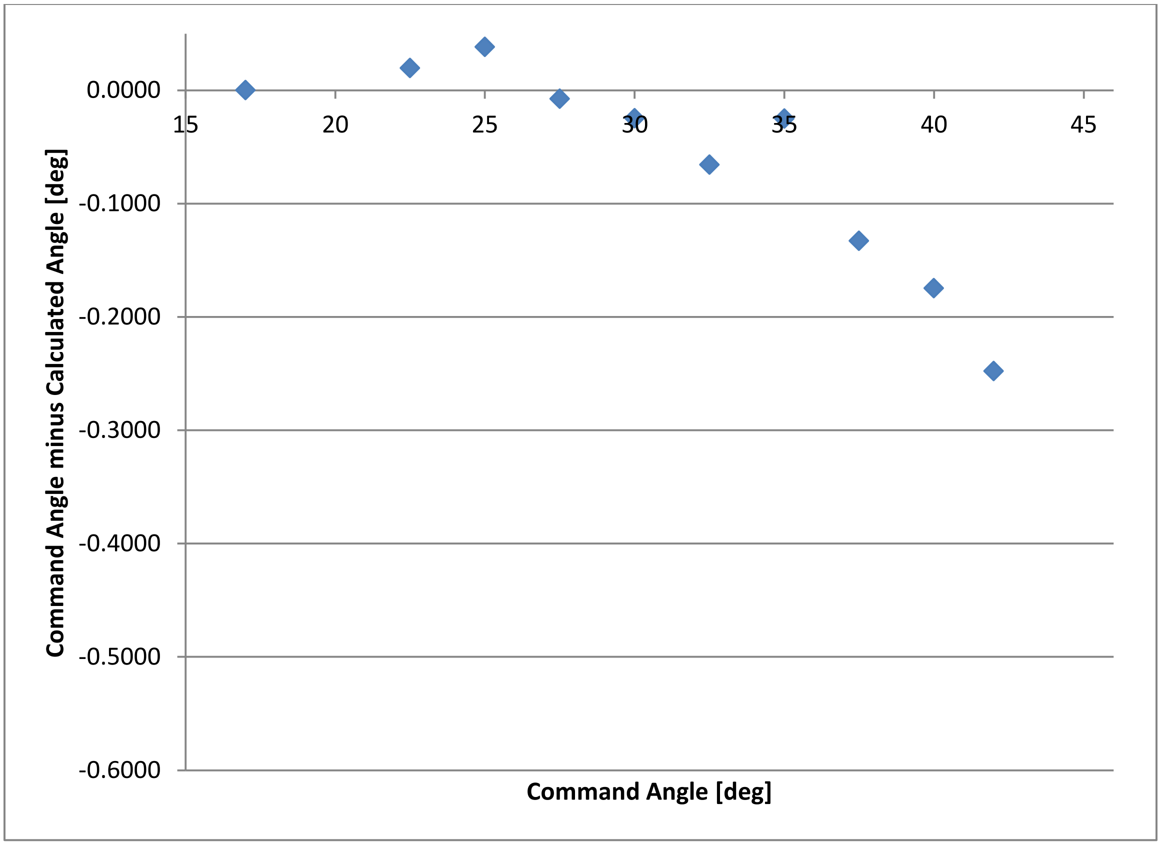


Figure 34: Altitude Relative Position Test. Azimuth = 10°, DistanceToBoard = 1295mm.

As seen in ....???

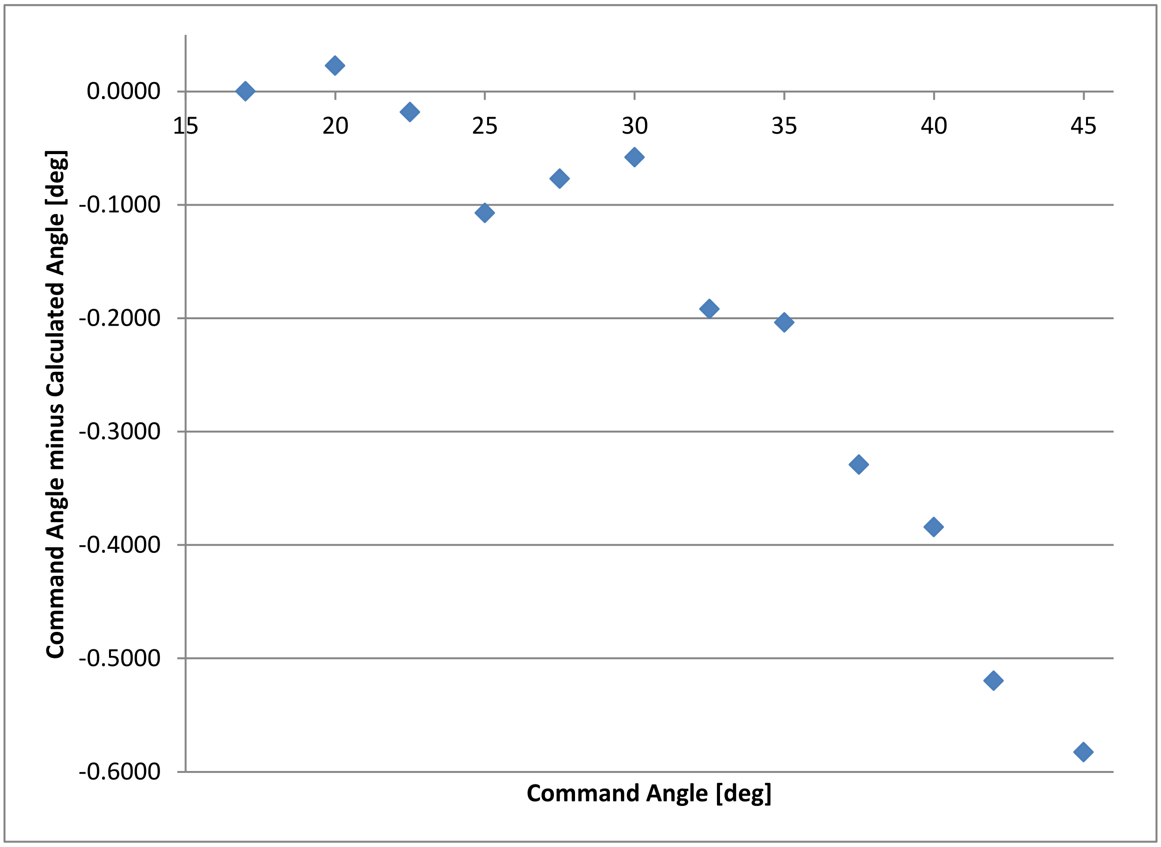


Figure 33 and Figure 34, the deviation from the expected distance increases as the altitude increases for both tests but remains small overall. This deviation is unacceptable

mechanical?

for astronomical systems but refinements to the system should significantly decrease it.

5.2 Velocity

Velocity testing was accomplished by commanding the apparatus to two points and marking them on the wall. When the apparatus was aimed at one of the points, it was commanded along the vector that would intersect with the second point. The motion was filmed, then analyzed to find the velocity.

OK, a thing that you should consider if it's possible: Estimates of the accuracy of your measurements. These don't need to be really complicated calculations unless Prof. Thorcroft insists on that...I think basic estimates based on things like how many pixels are in the width of a picture, how many frames per second are in the video, and what's a reasonable width of a laser pointer dot are good enough to be very helpful in understanding the limits of the accuracy of your tests.

The video viewing program VLC was used to find the time motion began and the time when the laser passed the second mark. This allowed for the calculation of angular velocity when combined with the angular distance between the points.

The test was run twice for a purely altitude motion and twice for a purely azimuth motion as seen in Table 4: Velocity Test Results.

Table 4: Velocity Test Results

	Command Velocity [rad/s]	Starting Position [rad]	Ending Position [rad]	Time to Completion [s]	Calculated Velocity [rad/s]	% Error
Altitude 1	0.001	0.29	0.6	317.58	9.76E-04	2.39
Altitude 2	0.002	0.29	0.6	161.162	1.92E-03	3.82
Azimuth 1	0.004	-0.2	0.2	104.852	3.81E-03	4.63
Azimuth 2	0.006	-0.2	0.2	69.564	5.75E-03	4.16

5.3 Repeatability

Repeatability was tested by moving between a series of points multiple times and seeing how much deviation there was between the first time and subsequent moves. It was done using a laser diode attached along the axis of the telescope. Setting the mount a known distance from a whiteboard, the mount was commanded to move to various altitude and azimuth positions. Image rotation was held constant at $\theta_3 = 0$. After each move, the location of the laser on the whiteboard was marked. Three types of motions were examined: altitude, combined, and long-run.

5.3.1 Altitude Repeatability

The Altitude Repeatability test moved back and forth between two azimuth positions holding altitude and image rotation constant. The test started the system aimed at the 0.3 rad point. As shown in Figure 35, the system returned to the previous position to within the marking size (about 3mm) each cycle, but the point which it returned to for 0.3 rad was not the same point at which it started. The mark is about 0.1268° wide at 1295mm.

Aha, there's the kind of accuracy estimate which would be helpful earlier on as well.

Also, the font changed. (Talk about a nitpick... but the grad office does that sometimes...)

Why are there two points at the bottom?

Figure 35: Altitude Repeatability Test. Mount traveled between points five times. Arrows indicate the laser was on top of a previous point.

5.3.2 Combined Repeatability

The Combined Repeatability test first established reference points by commanding the telescope to six positions and marking those positions on the board in blue. After this, three tests were run which moved between the points in different ways:

- The first test, labeled 2A, moved from point 6 to each other point and returned to point 6 in between.
- The second test, labeled 2B, moved from point 2 to each other point and returned to point 2 in between.
- The third test, labeled 2C, moved through the points 6>5>4>3>2>1>6.

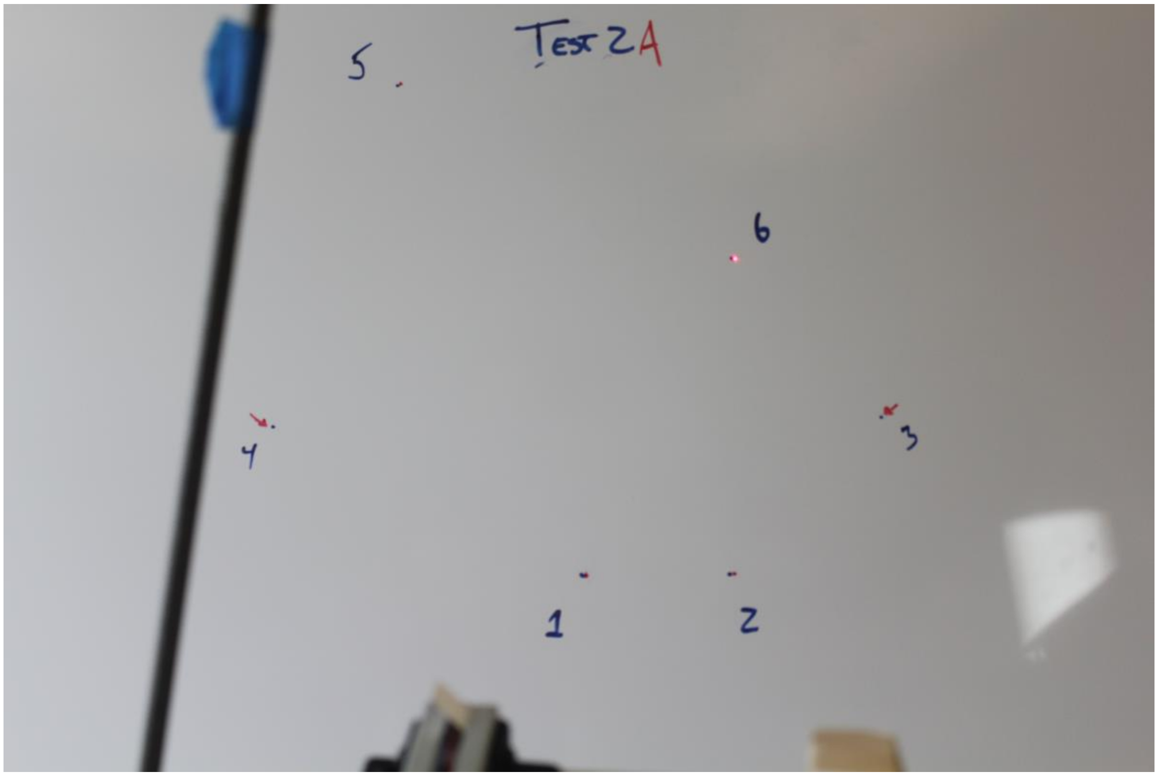


Figure 36: Combined Repeatability Test 2A

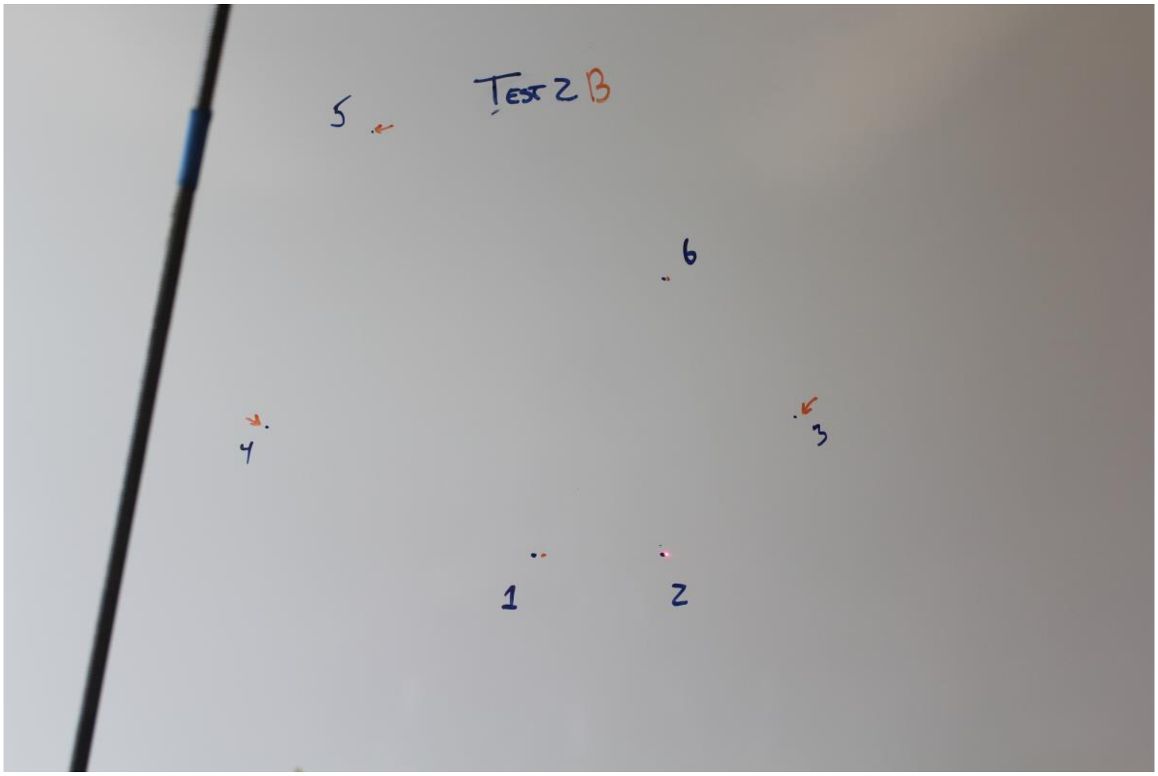


Figure 37: Combined Repeatability Test 2B

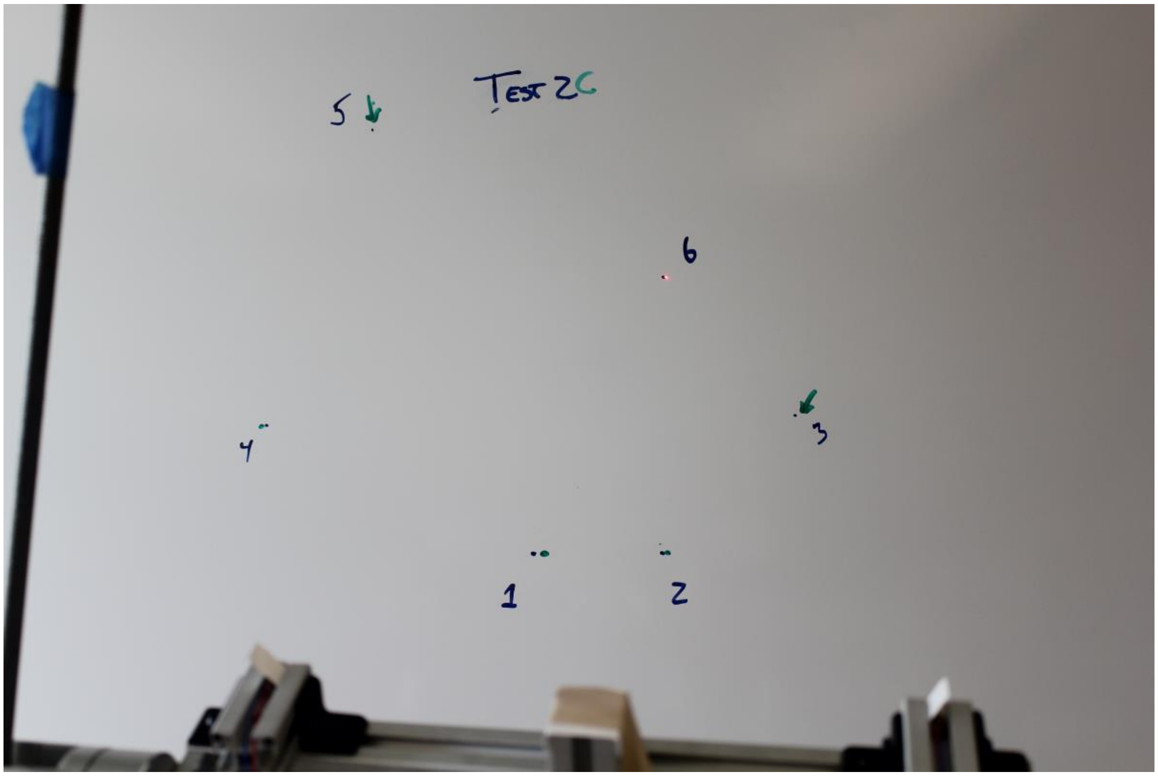


Figure 38: Combined Repeatability Test 2C

5.3.3 Long Run Repeatability

The Long Run Repeatability test cycled through seven points ten times and marked the laser at the first and last cycle.

OK, good...but where are the numerical results? Even saying that the errors were smaller than we could measure due to the size of the laser dot is helpful if that's the case. Or maybe I'm just not finding those numbers?

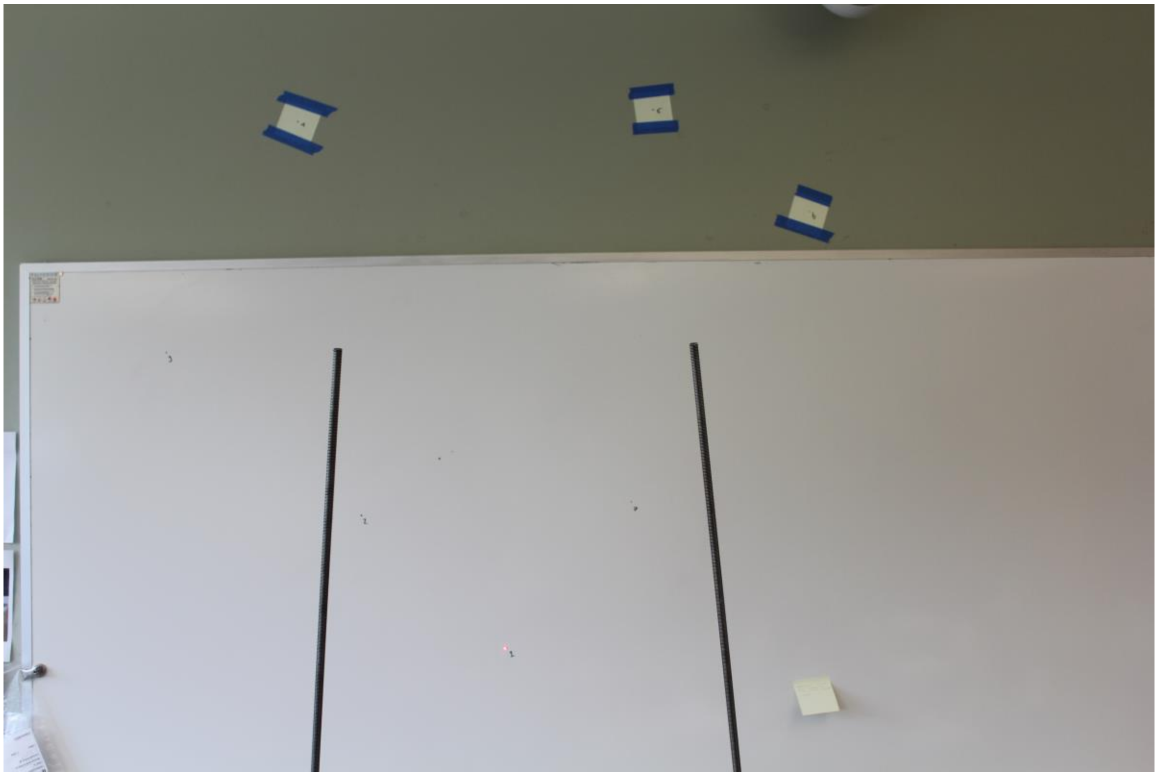


Figure 39: Long Run Repeatability Test Setup

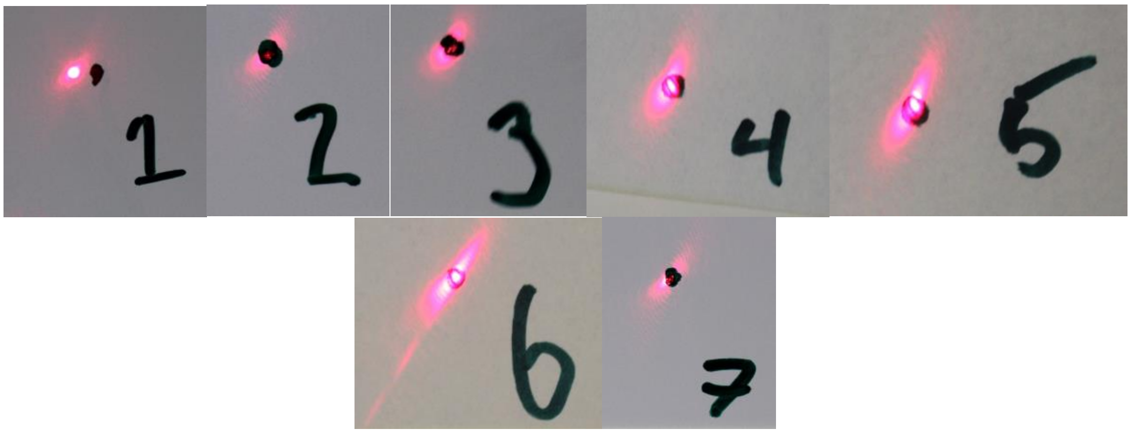


Figure 40: Long Run Repeatability Test Close View of Results

5.4 Tolerance Stack up

This test attempted to characterize the ability of the system to move in response to an external force while being commanded to hold position. It attempted to quantize the stack up of backlash and tolerance throughout the system.

The tolerance stack up test involved commanding the system to a series of points laid out in a grid. While the laser was aimed at each command location, the frame was pushed from left to right and the location of the laser was marked at its extremes. The surface this was marked on also had a set of reference points pre-printed on it which were a known distance apart (see Figure 41: Tolerance Stack Up Test markings with reference points.

Figure 41: Tolerance Stack Up Test markings with reference points

Once the full set of marks was recorded, a DSLR camera on a tripod was placed behind telescope. A high quality photo was taken centered on the grid. The Matlab image processing toolbox was used to combine the photo, the distances between the reference points, and the distance from the coordinate system origin to the marked surface into coordinates for each mark (Table 5 with reference to Figure 42. See page 133 for code).

Table 5: Tolerance Stack Up Coordinates

Cmd Azimuth	Cmd Altitude	x1	y1	x2	y2
-10	35	-12.01	36.18	-10.57	36.57
-10	30	-11.11	29.55	-10.21	29.79
-10	25	-10.46	23.57	-9.88	23.72
-10	20	-9.92	18.15	-9.23	18.37
-10	17	-9.79	15.10	-9.17	15.29
-5	35	-6.17	36.39	-5.08	36.48
-5	30	-5.69	29.68	-4.72	29.78
-5	25	-5.42	23.67	-4.34	23.80
-5	20	-5.10	18.21	-4.12	18.36
-5	17	-4.99	15.16	-3.96	15.31
0	35	-0.81	36.50	0.09	36.46
0	30	-0.69	29.79	0.21	29.76
0	25	-0.58	23.79	0.41	23.73
0	20	-0.48	18.30	0.41	18.30
0	17	-0.42	15.23	0.42	15.22
5	17	4.09	15.43	4.88	15.30
5	20	4.15	18.50	4.98	18.38
5	25	4.20	23.96	5.17	23.80
5	30	4.44	30.00	5.33	29.84
5	35	4.68	36.74	5.59	36.58
10	17	8.57	15.71	9.44	15.47
10	20	8.82	18.73	9.61	18.49
10	25	9.25	24.18	10.02	23.97
10	30	9.66	30.25	10.54	30.05
10	35	10.14	37.03	11.14	36.79

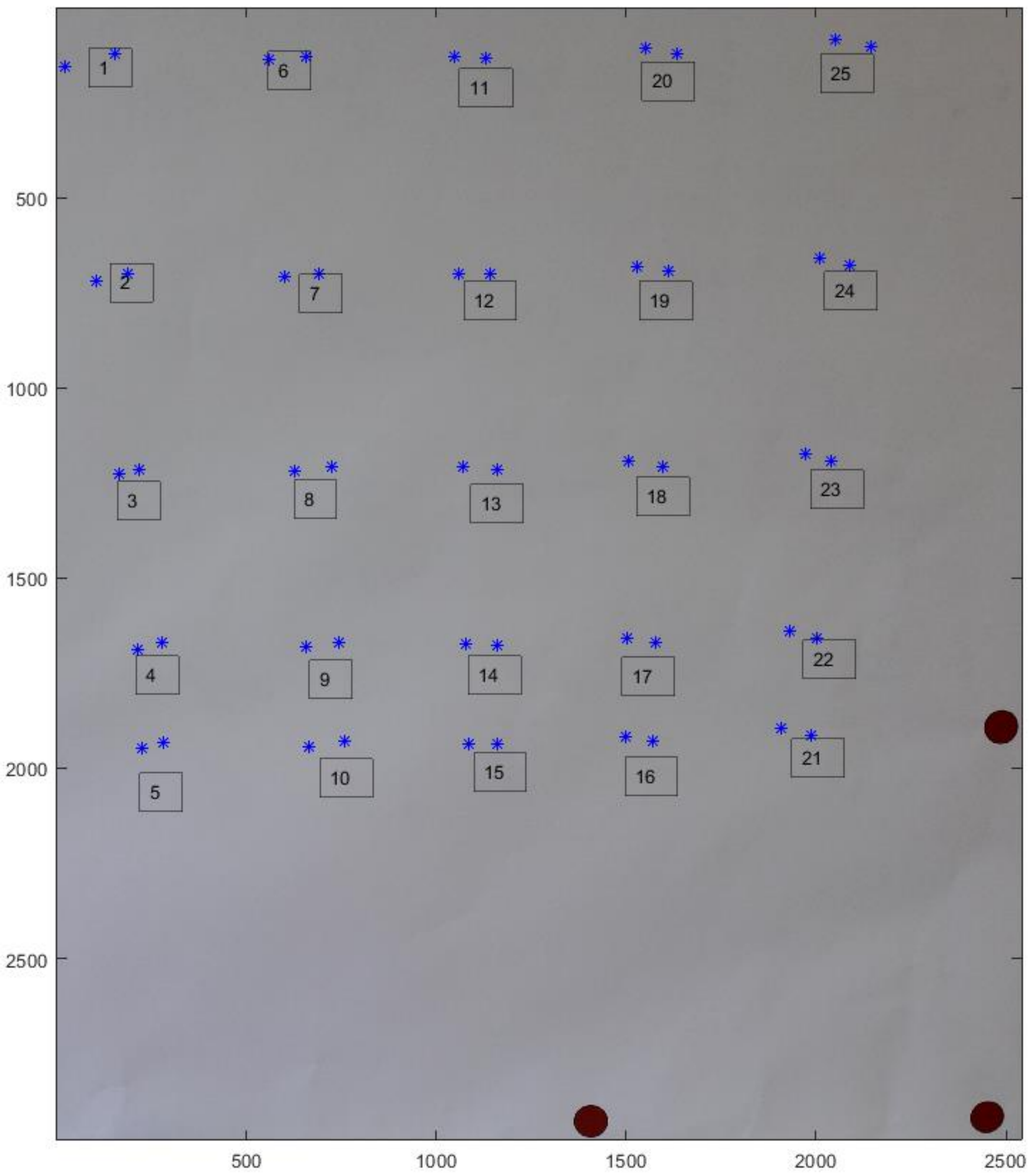


Figure 42: Tolerance Stack Up Test Coordinate Reference

Then it was calculated at what angle the telescope would need to point in order to reach the coordinates for each mark. The difference between these angles for each pair of points is the error caused by the tolerance stack up of the system as shown in Table 6.

Table 6: Error caused by tolerance stack up

Cmd Azimuth	Cmd Altitude	Az Error [deg]	Alt Error [deg]
-10	35	1.58	0.46
-10	30	0.98	0.29
-10	25	0.64	0.19
-10	20	0.77	0.26
-10	17	0.69	0.23
-5	35	1.24	0.13
-5	30	1.10	0.13
-5	25	1.24	0.17
-5	20	1.11	0.18
-5	17	1.18	0.19
0	35	1.04	0.03
0	30	1.04	0.02
0	25	1.14	0.05
0	20	1.02	0.00
0	17	0.97	0.00
5	17	0.90	0.16
5	20	0.95	0.16
5	25	1.11	0.19
5	30	1.02	0.18
5	35	1.04	0.17
10	17	0.97	0.29
10	20	0.87	0.29
10	25	0.85	0.27
10	30	0.97	0.25
10	35	1.09	0.29

5.5 Results

The test results show that the Parallel Actuator Mount is a viable design. It can move in a relatively reliable and repeatable manner though a useful range of alt-az positions. The results for this prototype are not sufficient for actual astronomical use but they are good enough that it is worthwhile to continue investigating this design. The repeatability and velocity tests are very promising. The relative position tests are more concerning.

The repeatability tests demonstrated that the system could repeatedly arrive at a specified position. This was possible while going back and forth between two points and going between various points. This test demonstrated that approaching a point from a particular direction was best for arriving at the specified point. This is not a desirable trait but improving the joint assemblies to reduce the movement seen in the tolerance stack up test would help minimize this issue.

It should be reasonable to conclude that bearing backlash was causing this problem, and using preloaded rolling contact bearings (like opposing angular contact ones) can fix this.

The velocity results were adequate to show that the method developed in section 3.6.2 is a viable approach to commanding velocity. Higher torque motors may allow the actuators to get up to the commanded velocity more quickly which would improve the results further. Smoothness of the motion was not measured in this test which could be another limiting factor in the usability of this method.

The relative altitude test also was an initial cause for concern. The test performed at azimuth = 0° results in points that line up vertically. The line of points from the test performed at azimuth = 10° clearly has a nonzero slope. This same issue is visible in the tolerance stack up results (Figure 43).

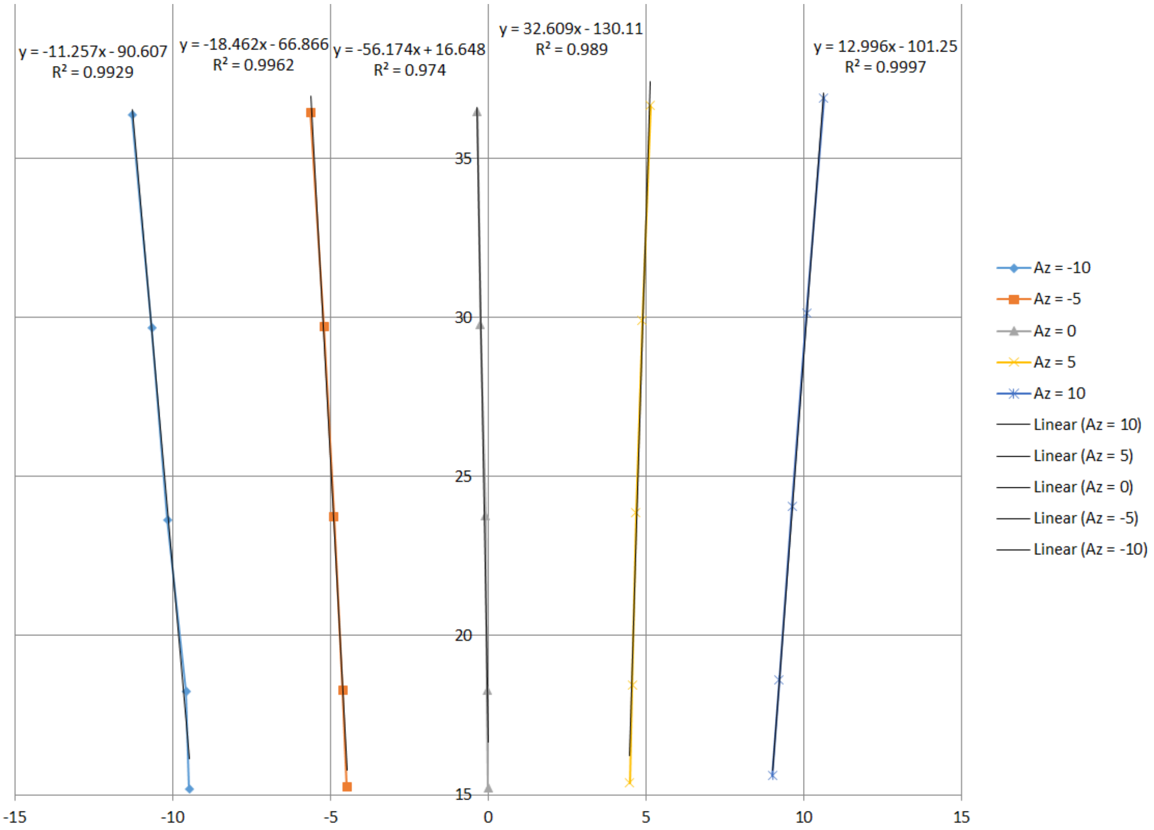


Figure 43: Tolerance stack up results plotted

This led to further testing of the model in Matlab which showed that the lines were not in fact straight but rather arcs (Figure 44).

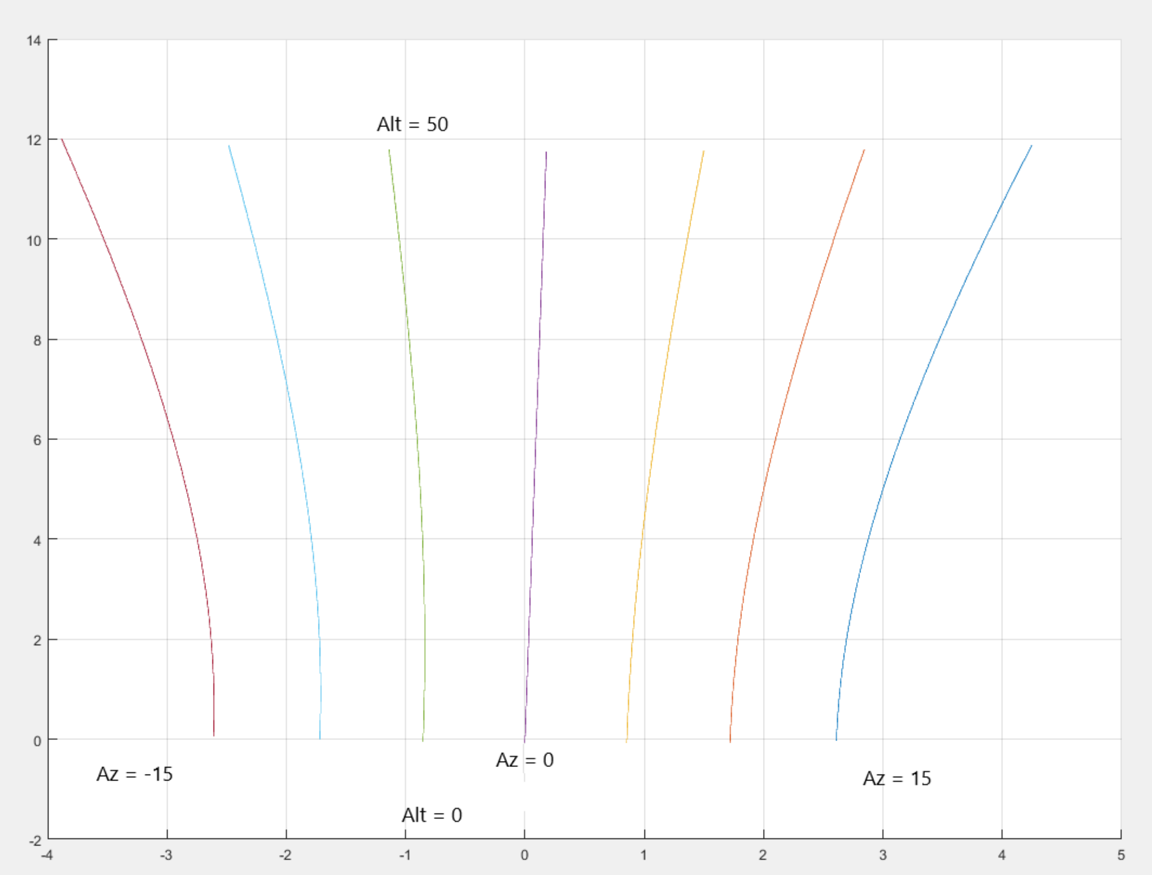


Figure 44: Telescope Optical Axis projected onto flat surface

Next the optical axis was projected onto a sphere (Figure 45) which showed the lines of constant azimuth as vertical as expected. Ideally this test would have been done with the laser projecting directly onto a spherical surface but this is impractical. Translating the points from the flat, available surface back onto a sphere via Matlab, the results can be evaluated.

What, no giant spheres around?

Figure 45 looks good, but somewhere we need to see numbers showing how much deviation there was between the measured points and where they were supposed to be. Or have I missed that? I'll go back and look at the tables again....

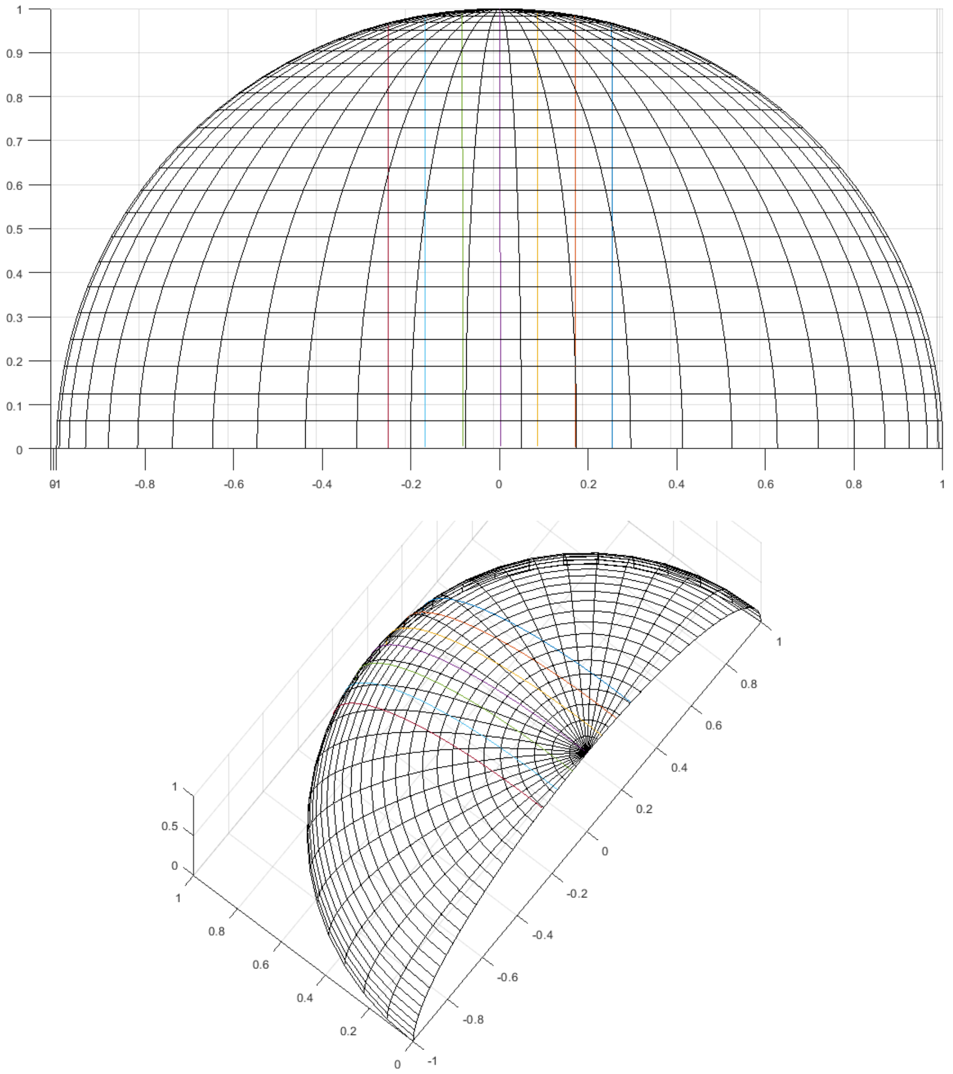


Figure 45: Telescope Optical Axis projected onto a sphere

The relative rotation test demonstrates a heavy asymmetry in the error with counter clockwise rotation having 10 times the error as clockwise. This may be due to the projection of a sphere onto a flat surface as was the case for the relative altitude test. If the lasers had been aiming perpendicular to the board (altitude of zero degrees), the alignment of the lasers for rotation testing would have been non-critical. However, the

I'm a bit confused: Where's the data for this test which demonstrates asymmetry?

altitude angle for the test was non-zero so any non-parallelism of the lasers to the optical axis may have had a major effect.

6. RECOMMENDATIONS

6.1 Software

The software is setup such that functionality can easily be added to main.py (see section 4.2.5). It is recommended that sufficient functionality be added such that the end user does not need to interact with the Actuator and Telescope objects directly.

6.2 Electrical

The use of the Nucleo microcontroller for this project has been well proved out. The L6470 stepper motor driver is easy to use and able to drive a wide range of stepper motors. The main recommendation for the electrical system is to incorporate appropriate peripheral electronics such as a dedicated power supply and voltage regulators in order to remove the need for a lab power supply. Additionally, a better system for housing the electronics and connecting the motors would be desireable.

6.3 Mechanical

The mechanical system has several necessary changes. These include redesigning the joint assemblies with proper bearings (as discussed in section 4.3) and replacing the motors for higher torque output. Small geometry changes may also greatly improve the system such as tilting the ground-to-actuator joint assemblies away from parallel with the base plate. This could help minimize the region where a singularity could occur.

7. CONCLUSION

This thesis approached the tasks of redesigning the Parallel Actuator Mount developed by Dr. John Ridgely and Mr. Garrett Gudgel. It incorporated lower priced components and developed more easily approachable software with great functionality. A method for velocity control was established and tested. Design guidelines were written which hope to allow for the easy implementation of the Parallel Actuator Mount. Developing repeatable tests was a major focus.

great?

The results of the developed testing found that the Parallel Actuator Mount is a potentially viable system for aiming a telescope when full sky coverage is not required. The tests showed too much error to fully recommend the system as built and tested but there seem to be paths to increase accuracy of the system without greatly increasing the complexity or cost. It would be interesting to investigate the inclusion of various forms of feedback including a plate solver and an inertial measurement unit.

This is a really short conclusion; I think you can increase the level of detail a bit without expending too much effort.

8. BIBLIOGRAPHY

Anaheim Automation. (n.d.). *Stepper Motor Guide*. Retrieved August 4, 2017, from Anaheim Automation: http://www.anaheimautomation.com/manuals/forms/stepper-motor-guide.php

Arnold, E. (2004, October 27). *Simulator Flight Compartment*. Retrieved November 16, 2018, from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Simulator-flight-compartment.jpeg

Budynas, R. G., & Nisbett, J. K. (2015). *Shigley's Mechanical Engineering Design* (10th ed.). New York, NY, USA: McGraw-Hill Education.

Burnett, K. (1998, May). *Converting RA and DEC to ALT and AZ.* Retrieved February 9, 2019, from Stargazing Network: http://www.stargazing.net/kepler/altaz.html

Cav. (2006). *Zeiss di Merate - pilastro sud*. Retrieved November 16, 2018, from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Zeiss_di_Merate_-_pilastro_sud.jpg

Chini, R. (2000). The Hexapod Telescope - A Never-ending Story. *Astronomische Gesellschaft: Reviews in Modern Astronomy*, 257-268.

Collins, G. W. (1989). *The Foundations of Celestial Mechanics.* Pachart Publishing House.

Dobson Mount. (2005, September). Retrieved November 16, 2018, from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Dobson-mount.jpg

Duffett-Smith, P., & Zwart, J. (2011). *Practical Astronomy with your Calculator or Spreadsheet.* Cambridge University Press.

Friction and Friction Coefficients. (n.d.). Retrieved 08 07, 2018, from The Engineering Toolbox: https://www.engineeringtoolbox.com/friction-coefficients-d_778.html

Gudgel, G. D. (2015). *Three Degree-Of-Freedom Parallel Actuator Telescope Mount.* California Polytechnic State University, Mechanical Engineering, San Luis Obispo.

Koch, P. M., Kesteven, M., Nishioka, H., Jiang, H., Lin, K.-Y., Umetsu, K., et al. (2009, April 1). The AMiBA Hexapod Telescope Mount. *The Astrophysical Journal*, 1670-1684.

Matara UK. (n.d.). *Ballscrews vs. Leadscrews*. Retrieved 07 08, 2019, from Matara UK: https://www.matara.com/ballscrews-vs-leadscrews/

National Optical Astronomy Observatory/Association of Universities for Research in Astronomy/National Science Foundation. (n.d.). *Mayall 4-meter telescope.* Retrieved November 24, 2018, from National Optical Astronomy Observatory: https://www.noao.edu/image_gallery/html/im0299.html

National Optical Astronomy Observatory/Association of Universities for Research in Astronomy/National Science Foundation. (n.d.). *WIYN telescope.* Retrieved November 24, 2018, from National Optical Astronomy Observatory: https://www.noao.edu/image_gallery/WIYN/telescopes.html

Nguyen, M.-L. (2007, July). *Maksutov-Cassegrain Intes M703 mounted*. Retrieved November 16, 2018, from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Maksutov-Cassegrain_Intes_M703_mounted.jpg

Ridgely, J. (n.d.). Conversations with.

Stewart, D. (1965). A PlatformwithSix Degrees of Freedom. *Proceedings of The Institution of Mechanical Engineers*, 371-386.

STMicroelectronics. (n.d.). *L6470.* Retrieved July 2017, from STMicroelectronics: http://www.st.com/resource/en/datasheet/l6470.pdf

STMicroelectronics. (n.d.). *STM32L476RG*. Retrieved 8 14, 2018, from STMicroelectronics: https://www.st.com/en/microcontrollers/stm32l476rg.html

TWCarlson. (n.d.). *Azimuth-Altitude Schematic*. Retrieved 11 16, 2018, from Wikimedia Commons: https://commons.wikimedia.org/w/index.php?curid=17727911

Zhao, S. (2016, September). Time Derivative of Rotation Matrices: A Tutorial. *arXiv e-prints*.

大蔵. (2009, Februrary). *Baker-Nunn Camera*. Retrieved November 16, 2018, from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Baker-Nunn_camera_001.JPG

OK, now I'm confused. The author is big....who??

APPENDIX A DOXYGEN MANUAL

Better to refer to it as a reference manual for telescope mount control software (it's not a manual for Doxygen)

File List

Here is a list of all documented files with brief descriptions:

- ActuatorDriver.py (This file contains a driver for a linear actuator for use in a 3 DOF parallel actuator telescope mount)
- BitUtilities.py (Contains function to take the twos complement)
- l6470nucleo.py (This file contains a driver for a dual L6470 stepper driver board which is part of the Nucleo package from ST Micro)
- main.py (Control file for a Three Degree of Freedom Parallel Actuator Telescope Mount)
- TelescopeDriver.py (This file contains a driver for a Three Parallel actuator telescope)

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

- ActuatorDriver.Actuator (Object which represents one linear actuator for a 3 DOF parallel actuator telescope mount)
- l6470nucleo.Dual6470 (This class implements a driver for the dual L6470 stepping motor driver chips on a Nucleo IHM02A1 board)
- TelescopeDriver.Telescope (This class creates a telescope object controlled by 3 parallel actuators)

l6470nucleo.Dual6470 Class Reference

This class implements a driver for the dual L6470 stepping motor driver chips on a Nucleo IHM02A1 board.

Public Member Functions

def **__init__** (self, spi_object, **cs_pin**, **stby_rst_pin**)

Initialize the Dual L6470 driver.

def **run** (self, motor, steps_per_sec)

Tell the motor to run at the given speed.

def **StepClock** (self, direc=None)

Switch the motor to StepClock mode.

def **Move** (self, motor, steps, direc=None)

Tell motor driver `motor` *to move* `num_steps` *in the direction* `direction`.

def **GoTo** (self, motor, pos_steps)

This command asks the specified motor to move to the specified position.

def **GoTo_DIR** (self, motor, pos_steps, direc)

NOT IMPLEMENTED

def **GoUntil** (self, motor, action, steps_per_sec, direction)

NOT IMPLEMENTED

def **ReleaseSW** (self, motor, action, direc)

NOT IMPLEMENTED

def **GoHome** (self, motor)

NOT IMPLEMENTED

def **GoMark** (self, motor)

NOT IMPLEMENTED

def **ResetPos** (self, motor)

This command sets the given motor driver's absolute position register to zero.

def **ResetDevice** (self, motor)

NOT IMPLEMENTED

def **softStop** (self, motor)

Tell a motor driver to decelerate its motor and stop wherever it ends up after the deceleration.

def **HardStop** (self, motor)

Tell the specified motor to stop immediately, not even doing the usual smooth deceleration.

def **SoftHiZ** (self, motor)

Tell the specified motor to decelerate smoothly from its motion, then put the power bridges in high-impedance mode, turning off power to the motor.

def **HardHiZ** (self, motor)

Tell the specified motor to stop and go into high-impedance mode (no current is applied to the motor coils) immediately, not even doing the usual smooth deceleration.

def **GetStatus** (self, motor, verbose=0)

Gets the value of the STATUS register for the specified `motor`.

def **getPositions** (self, motor)

Get the positions stored in the drivers for the selected motor.

def **isStalled** (self, motor, homing=False)

Checks if the specified motor is stalled.

def **Print_Status** (self, motor, status)

Formatted printing of status codes for the driver.

Public Attributes

cs_pin

The CPU pin connected to the Chip Select (AKA 'Slave Select') pin of both the L6470 drivers.

stby_rst_pin

The CPU pin connected to the STBY/RST pin of the 6470's.

spi

The SPI object, configured with parameters that work for L6470's.

Static Public Attributes

dictionary **REGISTER_DICT** = {}

Dictionary of user available registers and their addresses.

dictionary **STATUS_DICT** = {}

Dictionary for the STATUS register.

dictionary **COMMAND_DICT** = {}

Dictionary for Application Commands.

Detailed Description

This class implements a driver for the dual L6470 stepping motor driver chips on a Nucleo IHM02A1 board.

The two driver chips are connected in SPI daisy-chain mode, which makes communication a bit convoluted.

NOTE: One solder bridge needs to be moved for the IHM02A1 to work with unmodified MicroPython. Bridge SB34 must be disconnected, and bridge SB12 must be connected instead. This is because the SCK signal for which the board is shipped is not the one which MicroPython uses by default.

Parameters:

spi_object	A SPI object already initialized. used to talk to the driver chips, either 1 or 2 for most Nucleos

cs_pin	The pin which is connected to the driver chips' SPI chip select (or 'slave select') inputs, in a pyb.Pin object
stby_rst_pin	The pin which is connected to the driver chips' STBY/RST inputs, in a pyb.Pin object

Constructor & Destructor Documentation

def l6470nucleo.Dual6470.__init__ (*self*, *spi_object*, *cs_pin*, *stby_rst_pin*)

Initialize the Dual L6470 driver.

The modes of the `CS` and `STBY/RST` pins are set and the SPI port is set up correctly.

Member Function Documentation

l6470nucleo.Dual6470.getPositions (*self*, *motor*)

Get the positions stored in the drivers for the selected motor.

Each driver stores its motor's position in a 22-bit register. If only one position is needed, it's efficient to get both because the drivers are daisy-chained on the SPI bus, so we have to send two commands and read a bunch of bytes of data anyway.

Parameters:

motor	The motor whose position should be read, 1 or 2

Returns:

The current positions of the selected motor

l6470nucleo.Dual6470.GetStatus (*self*, *motor*, *verbose* = 0)

Gets the value of the STATUS register for the specified `motor` .

Resets STATUS register warning flags. Exits the system from error state. Does not reset HiZ.

WARNING: resets warning flags and error state for both steppers attached to the L6470 Nucleo.

Parameters:

motor	The motor whose status register should be read, 1 or 2
verbose	Print the status register results if TRUE

l6470nucleo.Dual6470.GoHome (*self*, *motor*)

NOT IMPLEMENTED

l6470nucleo.Dual6470.GoMark (*self*, *motor*)

NOT IMPLEMENTED

l6470nucleo.Dual6470.GoTo (*self*, *motor*, *pos_steps*)

This command asks the specified motor to move to the specified position.

Parameters:

motor	The motor to move, either 1 or 2
pos_steps	The position to which to move, in absolute steps

l6470nucleo.Dual6470.GoTo_DIR (*self*, *motor*, *pos_steps*, *direc*)

NOT IMPLEMENTED

l6470nucleo.Dual6470.GoUntil (*self*, *motor*, *action*, *steps_per_sec*, *direction*)

NOT IMPLEMENTED

l6470nucleo.Dual6470.HardHiZ (*self*, *motor*)

Tell the specified motor to stop and go into high-impedance mode (no current is applied to the motor coils) immediately, not even doing the usual smooth deceleration.

This command puts the motor into a freewheeling mode with no control of position except for the small holding torque from the magnets in a PM hybrid stepper. This will probably cause the motor to miss some steps and have an inaccurate position count.

Parameters:

motor	The motor to be turned off, 1 or 2

l6470nucleo.Dual6470.HardStop (*self*, *motor*)

Tell the specified motor to stop immediately, not even doing the usual smooth deceleration.

This command asks for nearly infinite acceleration of the motor, and will probably cause the motor to miss some steps and have an inaccurate position count.

Parameters:

motor	The motor to halt, 1 or 2

l6470nucleo.Dual6470.isStalled (*self*, *motor*, *homing* = `False`)

Checks if the specified motor is stalled.

Did not work with all motors. Should be tested heavily before use.

Parameters:

motor	The motor to check for stall
homing	If TRUE, reset position when stalled

l6470nucleo.Dual6470.Move (*self*, *motor*, *steps*, *direc* = `None`)

Tell motor driver `motor` to move `num_steps` in the direction `direction`.

Parameters:

motor	The motor to move, either 1 or 2
steps	How many steps to move, in a 20 bit number
direc	The direction in which to move, either 0 for one way or nonzero for the other; if unspecified, the sign of the number of steps will be used, positive meaning direction 0

l6470nucleo.Dual6470.Print_Status (*self*, *motor*, *status*)

Formatted printing of status codes for the driver.

Parameters:

motor	The motor which the status is representing.
status	The code returned by a GetStatus call.

l6470nucleo.Dual6470.ReleaseSW (*self*, *motor*, *action*, *direc*)

NOT IMPLEMENTED

l6470nucleo.Dual6470.ResetDevice (*self*, *motor*)

NOT IMPLEMENTED

l6470nucleo.Dual6470.ResetPos (*self*, *motor*)

This command sets the given motor driver's absolute position register to zero.

Parameters:

motor	The motor whose position is to be zeroed, either 1 or 2

l6470nucleo.Dual6470.run (*self*, *motor*, *steps_per_sec*)

Tell the motor to run at the given speed.

The speed may be positive, causing the motor to run in direction 0, or negative, causing the motor to run in direction 1.

Parameters:

motor	The motor to move, either 1 or 2
steps_per_sec	The number of steps per second at which to go, up to the maximum allowable set in `MAX_SPEED`

l6470nucleo.Dual6470.SoftHiZ (*self*, *motor*)

Tell the specified motor to decelerate smoothly from its motion, then put the power bridges in high-impedance mode, turning off power to the motor.

Parameters:

motor	The motor to be turned off, 1 or 2

l6470nucleo.Dual6470.softStop (*self*, *motor*)

Tell a motor driver to decelerate its motor and stop wherever it ends up after the deceleration.

Parameters:

motor	The motor to halt, 1 or 2

l6470nucleo.Dual6470.StepClock (*self*, *direc* = `None`)

Switch the motor to StepClock mode.

In StepClock mode the motor moves in response to the rising edge of a signal applied to the STCK pin.

NOT IMPLEMENTED

Parameters:

direc	Direction in which the motor should move on rising edge

Member Data Documentation

dictionary l6470nucleo.Dual6470.COMMAND_DICT = {}`[static]`

Dictionary for Application Commands.

See L6470 Programming manual Pg 56 for information on usage. These commands must be OR'd with the values for use.

l6470nucleo.Dual6470.cs_pin

The CPU pin connected to the Chip Select (AKA 'Slave Select') pin of both the L6470 drivers.

dictionary l6470nucleo.Dual6470.REGISTER_DICT = {}`[static]`

Dictionary of user available registers and their addresses.

l6470nucleo.Dual6470.spi

The SPI object, configured with parameters that work for L6470's.

pyb.SPI (spi_number, mode=pyb.SPI.MASTER, baudrate=2000000, polarity=1, phase=1, bits=8, firstbit=pyb.SPI.MSB)

dictionary l6470nucleo.Dual6470.STATUS_DICT = {}`[static]`

Dictionary for the STATUS register.

Contains all error flags, as well as basic motor state information.

l6470nucleo.Dual6470.stby_rst_pin

The CPU pin connected to the STBY/RST pin of the 6470's.

ActuatorDriver.Actuator Class Reference

Object which represents one linear actuator for a 3 DOF parallel actuator telescope mount.

Inheritance diagram for ActuatorDriver.Actuator:

Public Member Functions

def **__init__** (self, **stepPerLength**, motorController, **motorNumber**)

*Initialize the **Actuator** object.*

def **commandVelocity** (self, velocity)

Commands the actuator to move at a specified velocity.

def **commandLength** (self, length)

Commands the actuator to move to a specific length.

def **getCurrentLength** (self)

Reads the stepcount of the motor and converts to length.

def **findHome** (self)

commands the actuator to slowly shorten.

def **isHome** (self)

Checks if the motor is stalled.

def **resetPosition** (self)

Defines the current position as zero steps.

def **stop** (self)

Commands the actuator to come to a soft stop.

Public Attributes

stepPerLength

The number of steps the stepper motor must take in order to move a distance.

controller

The motor controller associated with the actuator.

motorNumber

The motor to be used from the controller.

currentLength

Variable to track the length of the actuator.

Detailed Description

Object which represents one linear actuator for a 3 DOF parallel actuator telescope mount.

Parameters:

stepPerLength	Total resolution of the actuator: STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS
motorControlle	Object which controls the motor: **l6470nucleo.Dual6470**

r	
motorNumber	Motor selection for the **l6470nucleo.Dual6470**: 1 or 2

Constructor & Destructor Documentation

def ActuatorDriver.Actuator.__init__ (*self*, *stepPerLength*, *motorController*, *motorNumber*)

Initialize the **Actuator** object.

Member Function Documentation

ActuatorDriver.Actuator.commandLength (*self*, *length*)

Commands the actuator to move to a specific length.

Length unit is defined by the stepPerLength parameter.

Parameters:

length	[length]

ActuatorDriver.Actuator.commandVelocity (*self*, *velocity*)

Commands the actuator to move at a specified velocity.

Length unit is defined by the stepPerLength parameter.

Parameters:

velocity	[length/sec]

ActuatorDriver.Actuator.findHome (*self*)

commands the actuator to slowly shorten.

WARNING: no automatic stop is built into this function

ActuatorDriver.Actuator.getCurrentLength (*self*)

Reads the stepcount of the motor and converts to length.

Returns:

currentLength the current length of the actuator

ActuatorDriver.Actuator.isHome (*self*)

Checks if the motor is stalled.

WARNING: results were not consistent and where very dependent on motor tuning.

ActuatorDriver.Actuator.resetPosition (*self*)

Defines the current position as zero steps.

Used for defining the home position.

ActuatorDriver.Actuator.stop (***self*****)**

Commands the actuator to come to a soft stop.

Member Data Documentation

ActuatorDriver.Actuator.controller

The motor controller associated with the actuator.

ActuatorDriver.Actuator.stepPerLength

The number of steps the stepper motor must take in order to move a distance.

Defines the units of length throughout the system.

TelescopeDriver.Telescope Class Reference

This class creates a telescope object controlled by 3 parallel actuators.
Inheritance diagram for TelescopeDriver.Telescope:

Public Member Functions

def **__init__** (self, **actuatorOne**, **actuatorTwo**, **actuatorThree**)

*Initialize the **Telescope** object.*

def **goVelocityAngular** (self, omega_alt, omega_az, omega_rot, tstep=1)

Move the telescope at specified radians/sec.

def **goToAngles** (self, O_alt, O_az, O_rot, move=False)

Function which calculates the correct lengths of the three linear actuators in order to achieve the specified angles (in degrees) then can send the scope to that location.

def **findHome** (self)

*Causes each actuator to shorten until it stalls then stops **WARNING:** Does not function!*

def **resetPositions** (self)

Zeros the absolute position of each actuator.

def **goToLengths** (self, lengthOne, lengthTwo, lengthThree)

Commands each actuator to a specified length.

def **goVelocity** (self, velOne, velTwo, velThree)

Commands each actuator to move at a specified velocity.

Public Attributes

actuatorOne

actuatorOne: Right actuator when viewed from the origin looking along the X axis

actuatorTwo

actuatorTwo: Left actuator when viewed from the origin looking along the X axis

actuatorThree

actuatorThree: Rear, horizontal actuator

P0_b

Origin with respect to the base.

P1_b

Fixed end of actuator one.

P2_b

Fixed end of actuator two.

P3_b

Fixed end of actuator three.

OA_b

Optical Axis base.

P1_h

Position of the moving end of actuator one in the home position.

P2_h

Position of the moving end of actuator two in the home position.

P3_h

Position of the moving end of actuator three in the home position.

OA_h

Position of the moving end of optical axis in the home position.

L_p1min

Minimum length of actuator one.

L_p2min

Minimum length of actuator two.

L_p3min

Minimum length of actuator three.

L_p1p2

Fixed distance from the moving end of actuator one to actuator two.

phi_az

The azimuth correction angle required to rotate from the home position to (0,0,0)

phi_alt

The altitude correction angle required to rotate from the home position to (0,0,0)

phi_rot

The image rotation correction angle required to rotate from the home position to (0,0,0)

alt_cur

The current altitude of the system.

az_cur

The current azimuth of the system.

rot_cur

The current image rotation of the system.

Detailed Description

This class creates a telescope object controlled by 3 parallel actuators.

Each actuator is driven by a stepper motor with an L6470 stepper driver.

Parameters:

actuatorOne	Right actuator when viewed from the origin looking along the X axis
actuatorTwo	Left actuator when viewed from the origin looking along the X axis
actuatorThree	Rear, horizontal actuator

Member Function Documentation

TelescopeDriver.Telescope.findHome (*self*)

Causes each actuator to shorten until it stalls then stops **WARNING:** Does not function!

Non-functioning function which is designed to implement automatic home checking

TelescopeDriver.Telescope.goToAngles (*self*, *O_alt*, *O_az*, *O_rot*, *move* = `False`)

Function which calculates the correct lengths of the three linear actuators in order to achieve the specified angles (in degrees) then can send the scope to that location.

Parameters:

O_alt	Desired Altitude Angle in radians
O_az	Desired Azimuth Angle in radians
O_rot	Desired Rotation Angle in radians
move	Boolean value which determines if the function only calculates the required lengths or also commands motion

TelescopeDriver.Telescope.goToLengths (*self*, *lengthOne*, *lengthTwo*, *lengthThree*)

Commands each actuator to a specified length.

Parameters:

lengthOne	Desired length for actuator one
lengthTwo	Desired length for actuator two
lengthThree	Desired length for actuator three

TelescopeDriver.Telescope.goVelocity (*self*, *velOne*, *velTwo*, *velThree*)

Commands each actuator to move at a specified velocity.

Parameters:

velOne	Velocity in length/sec
velTwo	Velocity in length/sec
velThree	Velocity in length/sec

TelescopeDriver.Telescope.goVelocityAngular (*self*, *omega_alt*, *omega_az*, *omega_rot*, *tstep* = `1`)

Move the telescope at specified radians/sec.

Parameters:

omega_az	Angular rate of azimuth (Rad/sec)
omega_alt	Angular rate of altitude (Rad/sec)
omega_rot	Angular rate of image rotation (Rad/sec)
tstep	time step for calculation of velocity. Default 1 sec

TelescopeDriver.Telescope.resetPositions (***self*****)**

Zeros the absolute position of each actuator.

Used for defining the home position.

main.py File Reference

Control file for a Three Degree of Freedom Parallel Actuator Telescope Mount.

Functions

def **main.estop** ()

commands all motors to halt immediately.

def **main.Go** (speed)

commands all motors to go at the specified speed.

def **main.getPos** ()

Queries the actuators for their current lengths.

def **main.getStatus** (motornumber)

Requests the status of the specified motor.

def **main.testRepeatability** (numloops=10)

Runs through a series of points `numloops` times.

def **main.testGrid** ()

Moves to a series of points and waits for user input at each one.

Variables

main.STEPPER_RESOLUTION = const (200)

Stepper Resolution [steps/rev].

int **main.MICROSTEPS** = 4

*Number of microsteps This must match `NUMBER_OF_MICROSTEPS` in **`TelescopeDriver.py`** and `STEP_SEL` in **`l6470nucleo.py`**.*

main.SCREW_RESOLUTION = const(16)

Leadscrew Resolution [rev/in].

main.stby_rst_pin1 = pyb.Pin.cpu.B5

SPI one standby/reset pin.

main.stby_rst_pin2 = pyb.Pin.cpu.B3

SPI two standyby/reset pin.

main.nCS1 = pyb.Pin.cpu.A4

SPI one chip select pin.

main.nCS2 = pyb.Pin.cpu.A10

SPI two chip select pin.

int **main.spi_number** = 1

SPI number.

main.spi_object

spi object for motor drivers

main.Driver1 = l6470nucleo.Dual6470(spi_object,nCS1, stby_rst_pin1)

l6470nucleo board object

main.Driver2 = l6470nucleo.Dual6470(spi_object,nCS2, stby_rst_pin2)

l6470nucleo board object

main.actuatorOne = **ActuatorDriver.Actuator**(STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS, Driver1, 1)

Right actuator when viewed from the origin looking along the X axis.

main.actuatorTwo = **ActuatorDriver.Actuator**(STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS, Driver1, 2)

Left actuator when viewed from the origin looking along the X axis.

main.actuatorThree = **ActuatorDriver.Actuator**(STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS, Driver2, 1)

Rear, horizontal actuator.

main.Telescope = **TelescopeDriver.Telescope**(actuatorOne, actuatorTwo, actuatorThree)

Define the telescope based on the actuators above.

Detailed Description

Control file for a Three Degree of Freedom Parallel Actuator Telescope Mount.

Author:

Samuel Steejans Artho-Bentz

Function Documentation

main.estop ()

commands all motors to halt immediately.

def main.getStatus (*motornumber*)

Requests the status of the specified motor.

Parameters:

motornumber	The motor associated with the desired actuator: 1, 2, or 3

def main.Go (*speed*)

commands all motors to go at the specified speed.

directly communicates with the stepper motors.

Parameters:

speed	steps/second

main.testGrid ()

Moves to a series of points and waits for user input at each one.

Used for testing.

def main.testRepeatability (*numloops* = `10`)

Runs through a series of points `numloops` times.

Waits for the user to hit a key before moving on to the next point. Used for testing.

Parameters:

numloops	Desired number of times the points should be gone through.

APPENDIX B MICROPYTHON CODE

L6470nucleo.py

```
# -*- coding: utf-8 -*-
#
## @file l6470nucleo.py
#  This file contains a driver for a dual L6470 stepper driver board
which
#  is part of the Nucleo package from ST Micro. It can control each of
the
#  two L5470 chips which are on the board.
#
#  In order to use a stepping motor, the constants @c MAX_SPEED, @c
ACCEL,
#  @e etc. need to be tuned for that motor using the methods shown at
#  @c http://www.st.com/resource/en/application_note/dm00061093.pdf.
#
#  @author JR Ridgely, Samuel Steejans Artho-Bentz


import pyb
import time
import BitUtilities
from math import ceil as math_ceil


## Maximum speed for the motor; default 65, or ~992 steps/s
MAX_SPEED = const (10)

## Acceleration for the motor; default 138, or 2008 steps/s^2
ACCEL = const (12)

## Deceleration for the motor; default 138, or 2008 steps/s^2
DECEL = const (12)

## The K_val constant for registers 0x09, 0x0A, 0x0B, 0x0C; default
0x29 = 41
K_VAL = const (255)   # Calculated 200??

## Intersection speed for register 0x0D; default 0x0408 = 1032
INT_SPEED = const (2307)

## Startup slope for register 0x0E; default 0x19 = 25
ST_SLP = const (25)

## Final slope for registers 0x0F, 0x10; default 0x29 = 41
FN_SLP = const (255)

## Number of Microsteps, num which are powers of 2 up to 128 are
acceptable.
STEP_SEL = 4

## Define SYNC_ENable bitmask. 0x80 for High, 0x00 for Low
```

```
SYNC_EN = const(0x00)

## SYNC_SEL modes. Datasheet pg 46 for full information
SYNC_SEL = const(0x10)

## STALL_TH. Default of 2.03A
# pg 47 of L6470 Programming Manual for details
STALL_TH = const(13)


## @class Dual6470
#  This class implements a driver for the dual L6470 stepping motor
driver
#  chips on a Nucleo IHM02A1 board. The two driver chips are connected
in
#  SPI daisy-chain mode, which makes communication a bit convoluted.
#
#  NOTE: One solder bridge needs to be moved for the IHM02A1 to work
#  with unmodified MicroPython. Bridge SB34 must be disconnected, and
#  bridge SB12 must be connected instead. This is because the SCK
signal
#  for which the board is shipped is not the one which MicroPython uses
#  by default.
#  @param spi_object A SPI object already initialized. used to talk to
the
#      driver chips, either 1 or 2 for most Nucleos
#  @param cs_pin The pin which is connected to the driver chips' SPI
#      chip select (or 'slave select') inputs, in a pyb.Pin object
#  @param stby_rst_pin The pin which is connected to the driver
#      chips' STBY/RST inputs, in a pyb.Pin object

class Dual6470:
    # === DICTIONARIES ===

    ## Dictionary of user available registers and their addresses.
    REGISTER_DICT = {} #ADDR | LEN |  DESCRIPTION     | xRESET | Write
    REGISTER_DICT['ABS_POS']=[0x01, 22] # current pos     | 000000 |   S
    REGISTER_DICT['EL_POS']=[0x02,  9] # Electrical pos  |    000 |   S
    REGISTER_DICT['MARK']=[0x03, 22] # mark position     | 000000 |   W
    REGISTER_DICT['SPEED']=[0x04, 20] # current speed    |  00000 |   R
    REGISTER_DICT['ACC']=[0x05, 12] # accel limit        |    08A |   W
    REGISTER_DICT['DEC']=[0x06, 12] # decel limit        |    08A |   W
    REGISTER_DICT['MAX_SPEED']=[0x07, 10] # maximum speed|    041 |   W
    REGISTER_DICT['MIN_SPEED']=[0x08, 13] # minimum speed|      0 |   S
    REGISTER_DICT['FS_SPD']=[0x15, 10] # full-step speed |    027 |   W
    REGISTER_DICT['KVAL_HOLD']=[0x09,  8] # holding Kval |     29 |   W
    REGISTER_DICT['KVAL_RUN']=[0x0A,  8] # const speed Kval|   29 |   W
    REGISTER_DICT['KVAL_ACC']=[0x0B,  8] # accel start Kval|   29 |   W
    REGISTER_DICT['KVAL_DEC'  ]=[0x0C,  8] # decel start Kval| 29 |   W
    REGISTER_DICT['INT_SPEED' ]=[0x0D, 14] # intersect speed|0408 |   H
    REGISTER_DICT['ST_SLP'    ]=[0x0E,  8] # start slope |     19 |   H
    REGISTER_DICT['FN_SLP_ACC']=[0x0F,  8] # accel end slope|  29 |   H
    REGISTER_DICT['FN_SLP_DEC']=[0x10,  8] # decel end slope|  29 |   H
    REGISTER_DICT['K_THERM'   ]=[0x11,  4] # therm comp factr|  0 |   H
    REGISTER_DICT['ADC_OUT'   ]=[0x12,  5] # ADC output      | XX |
    REGISTER_DICT['OCD_TH'    ]=[0x13,  4] # OCD threshold|     8 |   W
    REGISTER_DICT['STALL_TH']=[0x14,  7] # STALL threshold|    40 |   W
    REGISTER_DICT['STEP_MODE' ]=[0x16,  8] # Step mode    |      7 |   H
    REGISTER_DICT['ALARM_EN'  ]=[0x17,  8] # Alarm enable|     FF |   S
    REGISTER_DICT['CONFIG'   ]=[0x18, 16] # IC configuration|2E88 |   H
    REGISTER_DICT['STATUS'    ]=[0x19, 16] # Status       |   XXXX |
    REGISTER_DICT['RESERVED A']=[0x1A,  0] # RESERVED     |        |   X
    REGISTER_DICT['RESERVED B']=[0x1B,  0] # RESERVED     |        |   X
    # Write: X = unreadable, W = Writable (always),
    #        S = Writable (when stopped), H = Writable (when Hi-Z)

    ## Dictionary for the STATUS register.
    # Contains all error flags, as well as basic motor state
information.

    STATUS_DICT = {}   # [    NAME    | OK/DEFAULT VALUE ]
    STATUS_DICT[14] = ['STEP_LOSS_B',1] # stall detection on bridge B
    STATUS_DICT[13] = ['STEP_LOSS_A',1] # stall detection on bridge A
    STATUS_DICT[12] = ['OVERCURRENT',1] # OCD, overcurrent detection
    STATUS_DICT[11] = ['HEAT_SHUTDN',1] # TH_SD, thermal shutdown
    STATUS_DICT[10] = ['HEAT_WARN  ',1] # TH_WN, thermal warning
    STATUS_DICT[ 9] = ['UNDERVOLT  ',1] # UVLO, low drive supply
voltage
    STATUS_DICT[ 8] = ['WRONG_CMD  ',0] # Unknown command
    STATUS_DICT[ 7] = ['NOTPERF_CMD',0] # Command can't be performed
    STATUS_DICT[ 3] = ['SWITCH_EDGE',0] # SW_EVN, signals switch
falling edge
    STATUS_DICT[ 2] = ['SWITCH_FLAG',0] # switch state. 0=open,
1=grounded
    STATUS_DICT[15] = ['STEPCK_MODE',0] # 1=step-clock mode, 0=normal
    STATUS_DICT[ 4] = ['DIRECTION'  ,1] # 1=forward, 0=reverse
    STATUS_DICT[ 6] = ['MOTOR_STAT' ,0] # two bits: 00=stopped,
01=accel, 10=decel, 11=const spd
    STATUS_DICT[ 1] = ['BUSY'       ,1] # low during movement commands
    STATUS_DICT[ 0] = ['Hi-Z'       ,1] # 1=hi-Z, 0=motor active

    ## Dictionary for Application Commands.
    # See L6470 Programming manual Pg 56 for information on usage.
    # These commands must be OR'd with the values for use.

    COMMAND_DICT = {} # [    NAME    | Command Hex Code |  Action ]
    COMMAND_DICT['SetParam'] = 0x00 # Writes VALUE in PARAM register
    COMMAND_DICT['GetParam'] = 0x20 # Returns the stored value in PARAM
register
    COMMAND_DICT['Run'  ] = 0x50 # Sets the target speed and direction
    COMMAND_DICT['StepClock'] = 0x58 # Puts the device in Step-clock
mode and imposes direction
    COMMAND_DICT['Move'] = 0x40 # Moves specified number of steps in
direction
    COMMAND_DICT['GoTo'] = 0x60 # Goes to specified ABS_POS (min path)
    COMMAND_DICT['GoTo_DIR'] = 0x68 # Goes to specified ABS_POS (forced
direction)
    COMMAND_DICT['GoUntil'    ] = 0x82 #
    COMMAND_DICT['ReleaseSW'  ] = 0x92 #
    COMMAND_DICT['GoHome'     ] = 0x70 #
    COMMAND_DICT['GoMark'     ] = 0x78 #
    COMMAND_DICT['ResetPos'   ] = 0xD8 #
    COMMAND_DICT['ResetDevice'] = 0xC0 #
    COMMAND_DICT['SoftStop'   ] = 0xB0 #
    COMMAND_DICT['HardStop'   ] = 0xB8 #
    COMMAND_DICT['SoftHiZ'    ] = 0xA0 #
    COMMAND_DICT['HardHiZ'    ] = 0xA8 #
    COMMAND_DICT['GetStatus'  ] = 0xD0 #

    ## Initialize the Dual L6470 driver. The modes of the @c CS and
    #  @c STBY/RST pins are set and the SPI port is set up correctly.
    def __init__ (self, spi_object, cs_pin, stby_rst_pin):

        ## The CPU pin connected to the Chip Select (AKA 'Slave
Select') pin
        #    of both the L6470 drivers.
        self.cs_pin = cs_pin

        ## The CPU pin connected to the STBY/RST pin of the 6470's.
        self.stby_rst_pin = stby_rst_pin

        ## The SPI object, configured with parameters that work for
L6470's.
        # pyb.SPI (spi_number, mode=pyb.SPI.MASTER,
        #                    baudrate=2000000, polarity=1, phase=1,
        #                    bits=8, firstbit=pyb.SPI.MSB)
        self.spi = spi_object

        # Make sure the CS and STBY/RST pins are configured correctly
        self.cs_pin.init (pyb.Pin.OUT_PP, pull=pyb.Pin.PULL_NONE)
        self.cs_pin.high ()
        self.stby_rst_pin.init (pyb.Pin.OUT_OD, pull=pyb.Pin.PULL_NONE)

        # Reset the L6470's
        stby_rst_pin.low ()
        time.sleep (0.01)
        stby_rst_pin.high ()

        # Set the registers which need to be modified for the motor to
go
        # This value affects how hard the motor is being pushed
        self._set_par_1b ('KVAL_HOLD', K_VAL)
        self._set_par_1b ('KVAL_RUN', K_VAL)
        self._set_par_1b ('KVAL_ACC', K_VAL)
        self._set_par_1b ('KVAL_DEC', K_VAL)

        # Speed at which we transition from slow to fast V_B
compensation
        self._set_par_2b ('INT_SPEED', INT_SPEED)

        # Acceleration and deceleration back EMF compensation slopes
        self._set_par_1b ('ST_SLP', ST_SLP)
        self._set_par_1b ('FN_SLP_ACC', ST_SLP)
        self._set_par_1b ('FN_SLP_DEC', ST_SLP)

        # Set the maximum speed at which motor will run
        self._set_par_2b ('MAX_SPEED', MAX_SPEED)

        # Set the maximum acceleration and deceleration of motor
        self._set_par_2b ('ACC', ACCEL)
        self._set_par_2b ('DEC', DECEL)
        # Set the number of Microsteps to use
        self._set_MicroSteps (SYNC_EN, SYNC_SEL, STEP_SEL)

        # Set the Stall Threshold
        self._setStallThreshold(STALL_TH)

        # Set Minimum Speed = 400 steps/s and turn on Low Speed Opt (pg
35)

        #self._set_par_2b('MIN_SPEED', 0x168E)

        # Set motors in high impedence mode
        self.SoftHiZ(1)
        self.SoftHiZ(2)

    def _setStallThreshold(self, value):
        self._set_par_1b('STALL_TH', value)

    ## Set the number of Microsteps to use, the SYNC output frequency,
and the
    # SYNC ENABLE bit.
    # @param SYNC_Enable A 1-bit integer which determines the behavior
of the
    # BUSY/SYNC output.
    # @param SYNC_Select A 3-bit integer which determines SYNC output
frequency
    # @param num_STEP the integer number of microsteps, numbers which
are
    # powers of 2 up to 128 are acceptable.


    def _set_MicroSteps(self, SYNC_Enable, SYNC_Select, num_STEP):

        for stepval in range(0,8): #convert num_STEPS to 3-bit power of
2
            if num_STEP ==1:
                break
            num_STEP = num_STEP >>1
        if SYNC_Enable == 1:
            SYNC_EN_Mask = 0x80
        else:
            SYNC_EN_Mask = 0x00
        self._set_par_1b('STEP_MODE', SYNC_EN_Mask|stepval|SYNC_Select)

    ## Read a set of arguments which have been sent by the L6470's in
    #  response to a read-register command which has already been
    #  transmitted to the L6470's. The arguments are shifted into the
    #  integers supplied as parameters to this function.
    #  @param num_bytes The number of bytes to be read from each L6470


    def _read_bytes (self, num_bytes):

        data_1 = 0
        data_2 = 0
        # Each byte which comes in is put into the integer as the least
        # significant byte so far and will be shifted left to make room
for
        # the next byte
        for index in range (num_bytes):
            self.cs_pin.low ()
            data_1 <<= 8
            data_1 |= (self.spi.recv (1))[0]
            data_2 <<= 8
            data_2 |= (self.spi.recv (1))[0]
            self.cs_pin.high ()
        #print("in read bytes motor 1 is " + str(bin(data_1)))
        #print("in read bytes motor 2 is " + str(bin(data_2)))
        return ([data_1, data_2])

    ## Send one byte to each L6470 as a command and receive two bytes
    #  from each driver in response.
    #  @param command_byte The byte which is sent to both L6470's
    #  @param recv_bytes The number of bytes to receive: 1, 2, or 3


    def _get_params (self, command_byte, recv_bytes):

        # Send the command byte, probably a read-something command
        self._sndbs (command_byte, command_byte)

        # Receive the bytes from the driver chips
        [data_1, data_2] = self._read_bytes (recv_bytes)


        return([data_1, data_2])

    ## Set a parameter to both L6740 drivers, in a register which needs
    #  two bytes of data.0
    #  @param reg_name (string) The register to be set
    #  @param num The two-byte number to be put in that register

    def _set_par_2b (self, reg_name, num):
        reg = self.REGISTER_DICT[reg_name][0]
        self._sndbs (reg, reg)
        highb = (num >> 8) & 0x03
        self._sndbs (highb, highb)
        lowb = num & 0xFF
        self._sndbs (lowb, lowb)


    ## Set a parameter to both L6740 drivers, in a register which needs
    #  one byte of data.
    #  @param reg_name (string) The register to be set
    #  @param num The one-byte number to be put in that register

    def _set_par_1b (self, reg_name, num):
        reg = self.REGISTER_DICT[reg_name][0]
        self._sndbs (reg, reg)
        self._sndbs (num, num)
    ## Send one command byte to each L6470. No response is expected.
    #  @param byte_1 The byte sent first; it goes to the second chip
    #  @param byte_2 The byte sent second, to go to the first chip

    def _sndbs (self, byte_1, byte_2):

        self.cs_pin.low ()
        self.spi.send (byte_1)
        self.spi.send (byte_2)
        self.cs_pin.high ()


    ## Send a command byte only to one motor. The other motor is sent a
NOP
    #  command (all zeros).
    #  @param motor The number, 1 or 2, of the motor to receive the
command

    def _cmd_1b (self, motor, command):

        if motor == 1:
            self._sndbs (command, 0x00)
        elif motor == 2:
            self._sndbs (0x00, command)
        else:
            raise ValueError ('Invalid L6470 motor number; must be 1 or
2')


    ## Send a command byte plus three bytes of associated data to one
of
    #  the motors.
    #  @param motor The motor to receive the command, either 1 or 2
    #  @param command The one-byte command to be sent to one motor
    #  @param data The data to be sent after the command, in one 32-bit
    #    integer

    def _cmd_3b (self, motor, command, data):

        # Break the integer containing the data into three bytes
        byte_2 = (data >> 16) & 0x0F
        byte_1 = (data >> 8) & 0xFF
        byte_0 = data & 0xFF

        # Send commands to one motor, NOP (zero) bytes to the other
        if motor == 1:
            self._sndbs (command, 0x00)
            self._sndbs (byte_2, 0x00)
            self._sndbs (byte_1, 0x00)
            self._sndbs (byte_0, 0x00)
        elif motor == 2:
            self._sndbs (0x00, command)
            self._sndbs (0x00, byte_2)
            self._sndbs (0x00, byte_1)
            self._sndbs (0x00, byte_0)
        else:
            raise ValueError ('Invalid L6470 motor number; must be 1 or 
2')


#   ---------------- L6470 Built-in Functions------------------


    '''-------------------------------------------------------'''

    ## @fn run
    #  Tell the motor to run at the given speed. The speed may be
    #  positive, causing the motor to run in direction 0, or negative,
    #  causing the motor to run in direction 1.
    #  @param motor The motor to move, either 1 or 2
    #  @param steps_per_sec The number of steps per second at which to
    #      go, up to the maximum allowable set in @c MAX_SPEED

    def run (self, motor, steps_per_sec):

        # Figure out the direction from the sign of steps_per_sec
        if steps_per_sec < 0:
            direc = 0
            steps_per_sec = -steps_per_sec
        else:
            direc = 1
        if abs(steps_per_sec) > MAX_SPEED*15.259:
            print('This is beyond the current Max Speed')
        else:
            # Convert to speed register value: multiply by 
~(250ns)(2^28)
            steps_per_sec *= 67.108864
            steps_per_sec = int (steps_per_sec)
            # Have the _cmd_3b() method write the command to a driver 
chip
            self._cmd_3b (motor, self.COMMAND_DICT['Run'] | direc,
                            steps_per_sec & 0x000FFFFF)

    '''-------------------------------------------------------'''
    ## @fn StepClock
    #  Switch the motor to StepClock mode.
    #  In StepClock mode the motor moves in response to the rising edge 
of a signal applied to the STCK pin.
    #
    #  @b NOT @b IMPLEMENTED
    #  @param direc Direction in which the motor should move on rising 
edge
    def StepClock(self, direc = None):
      ''' '''

    '''-------------------------------------------------------'''
    ## @fn Move
    #  Tell motor driver @c motor to move @c num_steps in the direction
    #  @c direction.
    #  @param motor The motor to move, either 1 or 2
    #  @param steps How many steps to move, in a 20 bit number
    #  @param direc The direction in which to move, either 0 for one
    #      way or nonzero for the other; if unspecified, the sign of
the
    #      number of steps will be used, positive meaning direction 0

    def Move (self, motor, steps, direc = None):

        # Figure out the intended direction
        if direc == None:
            if steps <= 0:
                direc = 1
                steps = -steps
            else:
                direc = 0
        else:
            if direc != 0:
                direc = 1

        # Call the _cmd_3b() method to do most of the work
        self._cmd_3b (motor, self.COMMAND_DICT['Move'] | direc, steps &
0x003FFFFF)



    '''-------------------------------------------------------'''


    ## @fn GoTo
    #  This command asks the specified motor to move to the specified
    #  position.
    #  @param motor The motor to move, either 1 or 2
    #  @param pos_steps The position to which to move, in absolute
steps

    def GoTo (self, motor, pos_steps):

        # Call the _cmd_3b() method to do most of the work
        self._cmd_3b (motor, self.COMMAND_DICT['GoTo'], pos_steps &
0x003FFFFF)


    '''-------------------------------------------------------'''
    ## @fn GoTo_DIR
    #  @b NOT @b IMPLEMENTED
    def GoTo_DIR(self, motor, pos_steps, direc):
        ''' '''


    '''-------------------------------------------------------'''
    ## @fn GoUntil
    #  @b NOT @b IMPLEMENTED
    def GoUntil (self, motor, action, steps_per_sec, direction):
        ''' '''


    '''-------------------------------------------------------'''
    ## @fn ReleaseSW
    #  @b NOT @b IMPLEMENTED
    def ReleaseSW(self, motor, action, direc):
        ''' '''


    '''-------------------------------------------------------'''
    ## @fn GoHome
    #  @b NOT @b IMPLEMENTED
    def GoHome(self, motor):
        ''' '''



    '''-------------------------------------------------------'''
    ## @fn GoMark
    #  @b NOT @b IMPLEMENTED
    def GoMark(self, motor):
        ''' '''



    '''-------------------------------------------------------'''

    ## @fn ResetPos
    #  This command sets the given motor driver's absolute position
register
    #  to zero.
    #  @param motor The motor whose position is to be zeroed, either 1
or 2


    def ResetPos (self, motor):

        self._cmd_1b (motor, self.COMMAND_DICT['ResetPos'])


    '''-------------------------------------------------------'''
    ## @fn ResetDevice
    #  @b NOT @b IMPLEMENTED
    def ResetDevice(self, motor):
        ''' '''


    '''-------------------------------------------------------'''

    ## @fn softStop
    #  Tell a motor driver to decelerate its motor and stop wherever it
ends
    #  up after the deceleration.
    #  @param motor The motor to halt, 1 or 2

    def softStop (self, motor):

        self._cmd_1b (motor, self.COMMAND_DICT['SoftStop'])


    '''-------------------------------------------------------'''

    ## @fn HardStop
    #  Tell the specified motor to stop immediately, not even doing the 
usual
    #  smooth deceleration. This command asks for nearly infinite
    #  acceleration of the motor, and will probably cause the motor to
    #  miss some steps and have an inaccurate position count.
    #  @param motor The motor to halt, 1 or 2

    def HardStop (self, motor):

        self._cmd_1b (motor, self.COMMAND_DICT['HardStop'])


    '''-------------------------------------------------------'''

    ## @fn SoftHiZ
    #  Tell the specified motor to decelerate smoothly from its motion, 
then
    #  put the power bridges in high-impedance mode, turning off power 
to the
    #  motor.
    #  @param motor The motor to be turned off, 1 or 2

    def SoftHiZ (self, motor):

        self._cmd_1b (motor, self.COMMAND_DICT['SoftHiZ'])


    '''-------------------------------------------------------'''

    ## @fn HardHiZ
    #  Tell the specified motor to stop and go into high-impedance mode 
(no
    #  current is applied to the motor coils) immediately, not even 
doing the
    #  usual smooth deceleration. This command puts the motor into a
    #  freewheeling mode with no control of position except for the 
small
    #  holding torque from the magnets in a PM hybrid stepper. This 
will
    #  probably cause the motor to miss some steps and have an 
inaccurate
    #  position count.
    #  @param motor The motor to be turned off, 1 or 2

    def HardHiZ (self, motor):

        self._cmd_1b (motor, self.COMMAND_DICT['HardHiZ'])


    '''-------------------------------------------------------'''

    ## @fn GetStatus
    #  Gets the value of the STATUS register for the specified @c 
motor.
    #  Resets STATUS register warning flags.
    #  Exits the system from error state.
    #  Does not reset HiZ.
    #
    #  @b WARNING: resets warning flags and error state for both
steppers
    #  attached to the L6470 Nucleo.
    #  @param motor The motor whose status register should be read, 1
or 2
    #  @param verbose Print the status register results if TRUE
    def GetStatus (self, motor, verbose=0):
        status = self._get_params(self.COMMAND_DICT['GetStatus'], 2)
        if verbose:
            self.Print_Status(motor, status)
        return status

    '''-------------------------------------------------------'''
#   -----------------Secondary Functions-------------------

    ## @fn getPositions
    #  Get the positions stored in the drivers for the selected motor.
Each
    #  driver stores its motor's position in a 22-bit register. If only
    #  one position is needed, it's efficient to get both because the
    #  drivers are daisy-chained on the SPI bus, so we have to send two
    #  commands and read a bunch of bytes of data anyway.
    #  @param motor The motor whose position should be read, 1 or 2
    #  @return The current positions of the selected motor

    def getPositions (self, motor):

        # Read (command 0x20) register 0x01, the current position
        [data_1, data_2] = self._get_params
(self.COMMAND_DICT['GetParam']|self.REGISTER_DICT['ABS_POS'][0], 3)
        #print("motor 1 is " + str(bin(data_1)))
        #print("motor 2 is " + str(bin(data_2)))


        # Sign-extend the signed absolute position numbers to 32 bits
        '''  We Don't need to sign extend the number
        if data_1 & 0x00400000:
            data_1 |= 0xFF800000
        else:
            data_1 &= 0x001FFFFF
        if data_2 & 0x00400000:
            data_2 |= 0xFF800000
        else:
            data_2 &= 0x001FFFFF
        '''
        if motor == 1:
            data = data_1
        else:
            data = data_2
        return (BitUtilities.GetTwosComplement(data, 22))

    '''-------------------------------------------------------'''
    ## @fn isStalled
    #  Checks if the specified motor is stalled.
    #  Did not work with all motors. Should be tested heavily before
use.
```

```
    #  @param motor The motor to check for stall
    #  @param homing If TRUE, reset position when stalled
    def isStalled(self, motor, homing=False):
        status = self.GetStatus(1)
        if ((status[motor-1]&(1<<13) == 0) or (status[motor-1]&(1<<14)
== 0)):
            if homing:
                self.ResetPos(motor)
            return True
        else:
            return False


    '''-------------------------------------------------------'''
    ## @fn Print_Status
    #  Formatted printing of status codes for the driver.
    #  @param motor The motor which the status is representing.
    #  @param status The code returned by a GetStatus call.
    def Print_Status(self, motor, status):

        # check error flags
        print ('Driver ', str(motor), ' Status: ') #, bin(status))
        print(status)
        for bit_addr in range(7,15):
            print('  Flag ', self.STATUS_DICT[bit_addr][0], ': ',
end='')
            # we shift a 1 to the bit address, then shift the result
down again
            if ((status[motor-1] &
1<<bit_addr)>>bit_addr)==self.STATUS_DICT[bit_addr][1]:
                # the result should either be a 1 or 0. Which is 'ok'
depends.
                print("ok")
            else:
                print("Alert!")

        # check SCK_MOD
        if status[motor-1] & (1<<15):
            print('  Step-clock mode is on.')
        else:
            print("  Step-clock mode is off.")

        # check MOT_STATUS
        if status[motor-1] & (1<<6):
            if status[motor-1] & (1<<5):
                print("  Motor is at constant speed.")
            else:
                print("  Motor is decelerating.")
        else:
            if status[motor-1] & (1<<5):
                print("  Motor is accelerating.")
            else:
                print("  Motor is stopped.")

        # check DIR
        if status[motor-1] & (1<<4):
            print("  Motor direction is set to forward.")
        else:
            print("  Motor direction is set to reverse.")

        # check BUSY
        if not (status[motor-1] & (1<<1)):
            print("  Motor is busy with a movement command.")
        else:
            print("  Motor is ready to recieve movement commands.")

        # check HiZ
        if status[motor-1] & 1:
            print("  Bridges are in high-impedance mode (disabled).")
        else:
            print("  Bridges are in low-impedance mode (active).")

        # check SW_EVEN flag
        if status[motor-1] & (1<<3):
            print("  External switch has been clicked since last 
check.")
        else:
            print("  External switch has no activity to report.")
        # check SW_F
        if status[motor-1] & (1<<2):
            print("  External switch is closed (grounded).")
        else:
            print("  External switch is open.")
```

ActuatorDriver.py

```
# -*- coding: utf-8 -*-
#
## @file ActuatorDriver.py
#  This file contains a driver for a linear actuator for use in a 3 DOF
parallel
#  actuator telescope mount.
#
#
#  @author Samuel Steejans Artho-Bentz

## @class Actuator
#  Object which represents one linear actuator for a 3 DOF parallel
actuator telescope mount
#  @param stepPerLength Total resolution of the actuator:
STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS
#  @param motorController Object which controls the motor:
l6470nucleo.Dual6470
#  @param motorNumber Motor selection for the l6470nucleo.Dual6470: 1
or 2
class Actuator(object):

    ## Initialize the Actuator object.

    def __init__(self, stepPerLength, motorController, motorNumber):
        ## @var stepPerLength
        #  The number of steps the stepper motor must take in order to
move
        #  a distance.
        #  Defines the units of length throughout the system.
        self.stepPerLength = stepPerLength
        ## @var controller
        #  The motor controller associated with the actuator.
        self.controller = motorController
        ## @var motorNumber
        #  The motor to be used from the controller
        self.motorNumber = motorNumber
        ## @var currentLength
        #  Variable to track the length of the actuator
        self.currentLength = 0

    ## @fn commandVelocity
    #  Commands the actuator to move at a specified velocity.
    #  Length unit is defined by the stepPerLength parameter.
    #  @param velocity [length/sec]
    def commandVelocity(self, velocity):

        stepsPerSec = self.__calcSteps(velocity)
        print('Command to ' + str(stepsPerSec) + ' steps/sec')
        self.controller.run(self.motorNumber, stepsPerSec)

    ## @fn commandLength
    #  Commands the actuator to move to a specific length.
    #  Length unit is defined by the stepPerLength parameter.
    #  @param length [length]
    def commandLength(self, length):
        steps = self.__calcSteps(length)
        print(['command to ' + str(steps) + ' steps'])
        self.__commandStepMotion(steps)

    ## convert length into steps
    def __calcSteps(self, length):
        steps = round(length*self.stepPerLength)
        return steps

    ## interface with motorController to move to a specific step count
    def __commandStepMotion(self,steps):
        self.controller.GoTo(self.motorNumber, steps)

    ## @fn getCurrentLength
    #  Reads the stepcount of the motor and converts to length
    #  @return currentLength the current length of the actuator
    def getCurrentLength(self):
        self.currentLength =
self.controller.getPositions(self.motorNumber)/self.stepPerLength
        return self.currentLength

    ## @fn findHome
    #  commands the actuator to slowly shorten.
    #  @b WARNING: no automatic stop is built into this function
    def findHome(self):
        self.controller.run(self.motorNumber, -10)

    ## @fn isHome
    #  Checks if the motor is stalled.
    #  @b WARNING: results were not consistent and where very dependent
on motor tuning.
    def isHome(self):
        stalled = self.controller.isStalled(self.motorNumber, True)
        if stalled:
            self.controller.HardHiZ(self.motorNumber)
            return True
        else:
            return False

    ## @fn resetPosition
    #  Defines the current position as zero steps.
    #  Used for defining the home position.
    def resetPosition(self):
        self.controller.ResetPos(self.motorNumber)
    ## @fn stop
    #  Commands the actuator to come to a soft stop.
    def stop(self):
        self.controller.softStop(self.motorNumber)
```

TelescopeDriver.py

```
# -*- coding: utf-8 -*-
#
## @file TelescopeDriver.py
#  This file contains a driver for a Three Parallel actuator telescope.
#
#
#
#  @author Samuel Steejans Artho-Bentz

import math
import time
import pyb

## @var NUMBER_OF_MICROSTEPS
#  Must define the number of microsteps to correct for oddity of
stepper driver
NUMBER_OF_MICROSTEPS = 4

## @class Telescope
#  This class creates a telescope object controlled by 3 parallel
actuators.
#  Each actuator is driven by a stepper motor with an L6470 stepper
driver.
#  @param actuatorOne Right actuator when viewed from the origin
looking along the X axis
#  @param actuatorTwo Left actuator when viewed from the origin looking
along the X axis
#  @param actuatorThree Rear, horizontal actuator
class Telescope(object):

    ## Initialize the Telescope object

    def __init__ (self, actuatorOne, actuatorTwo, actuatorThree):

        ## actuatorOne: Right actuator when viewed from the origin
looking along the X axis
        self.actuatorOne = actuatorOne
        ## actuatorTwo: Left actuator when viewed from the origin
looking along the X axis
        self.actuatorTwo = actuatorTwo
        ## actuatorThree: Rear, horizontal actuator
        self.actuatorThree = actuatorThree


        ## Origin with respect to the base
        self.P0_b = [[0],[0],[0]]

        ## Fixed end of actuator one
        self.P1_b = [[-12],[-3.25],[16]]

        ## Fixed end of actuator two
        self.P2_b = [[12],[-3.25],[16]]

        ## Fixed end of actuator three
        self.P3_b = [[-12],[-3.25],[5]]
        ## Optical Axis base
        self.OA_b = self.P0_b;


        ## Position of the moving end of actuator one in the home
position
        self.P1_h = [[-13.55],[8.55],[10.64]]

        ## Position of the moving end of actuator two in the home
position
        self.P2_h = [[6.85],[8.87],[15.63]]

        ## Position of the moving end of actuator three in the home
position
        self.P3_h = [[-1.32],[-2.44],[5.57]]

        ## Position of the moving end of optical axis in the home
position
        self.OA_h = [[-4.0],[3.75],[16.25]]

        # Calculate Minimum Lengths
        ## Minimum length of actuator one
        self.L_p1min = rootsumsquare(self.P1_h, self.P1_b)

        ## Minimum length of actuator two
        self.L_p2min = rootsumsquare(self.P2_h, self.P2_b)

        ## Minimum length of actuator three
        self.L_p3min = rootsumsquare(self.P3_h, self.P3_b)

        ## Fixed distance from the moving end of actuator one to
actuator two
        self.L_p1p2  = rootsumsquare(self.P1_h, self.P2_h)

        #Calculate Correction Angles \phi to Rotate from home position
to alt = 0, az = 0

        ## The azimuth correction angle required to rotate from the
home position to (0,0,0)
        self.phi_az = -math.atan2(self.OA_h[0][0], self.OA_h[2][0]);
        ## The altitude correction angle required to rotate from the
home position to (0,0,0)
        self.phi_alt = math.atan2(self.OA_h[1][0], self.OA_h[2][0]);
        ## The image rotation correction angle required to rotate from
the home position to (0,0,0)
        self.phi_rot = math.atan2(self.P1_h[1][0]-self.P2_h[1][0],
self.L_p1p2)

        # Define current angles


        ## The current altitude of the system
        self.alt_cur = self.phi_alt
        ## The current azimuth of the system
        self.az_cur = self.phi_az
        ## The current image rotation of the system
        self.rot_cur = self.phi_rot
        # Print warning message
        print('Please do not interrupt any movement commands. Allow
them to finish completely')

    ## @fn goVelocityAngular
    #  Move the telescope at specified radians/sec.
    #  @param omega_az Angular rate of azimuth (Rad/sec)
    #  @param omega_alt Angular rate of altitude (Rad/sec)
    #  @param omega_rot Angular rate of image rotation (Rad/sec)
    #  @param tstep time step for calculation of velocity. Default 1
sec
    def goVelocityAngular(self, omega_alt, omega_az, omega_rot,
tstep=1):
        Walt = omega_alt
        Waz = omega_az
        Wrot = omega_rot
        print('Walt: ' +str(Walt)+ ' Waz: ' + str(Waz)+ ' Wrot: '
+str(Wrot))
        V = [0,0,0]
        # Use current position to find new target position
        vcp = pyb.USB_VCP ()
        while not vcp.any (): # causes any keypress to interrupt
            alt_new = self.alt_cur+Walt*tstep
            az_new = self.az_cur+Waz*tstep
            rot_new = self.rot_cur+Wrot*tstep
            L_new = self.goToAngles(alt_new, az_new, rot_new)
            print('New Angles are: ' + str(alt_new) + ' ' + str(az_new)
+ ' ' + str(rot_new))
            print('L_new is: ' + str(L_new))
            L1 = self.actuatorOne.getCurrentLength()+self.L_p1min
            L2 = self.actuatorTwo.getCurrentLength()+self.L_p2min
            L3 = self.actuatorThree.getCurrentLength()+self.L_p3min
            V[0] = (L_new[0]-(L1))/tstep/NUMBER_OF_MICROSTEPS
            V[1] = (L_new[1]-(L2))/tstep/NUMBER_OF_MICROSTEPS
            V[2] = (L_new[2]-(L3))/tstep/NUMBER_OF_MICROSTEPS
            print('Current Lengths are: ' + str(L1) + ' ' + str(L2) + '
' + str(L3))
            print('Velocities are: ' + str(V))
            self.actuatorOne.commandVelocity(V[0])
            self.actuatorTwo.commandVelocity(V[1])
            self.actuatorThree.commandVelocity(V[2])
            start = pyb.micros()
            self.alt_cur = alt_new
            self.az_cur = az_new
            self.rot_cur = rot_new
            while pyb.elapsed_micros(start)<int(tstep*(10**6)):
                pass
        self.actuatorOne.stop()
        self.actuatorTwo.stop()
        self.actuatorThree.stop()
        self.alt_cur = alt_new
        self.az_cur = az_new
        self.rot_cur = rot_new
        return ([self.alt_cur, self.az_cur, self.rot_cur])

    ## @fn goToAngles
    #  Function which calculates the correct lengths of the three
linear actuators in order to
    #  achieve the specified angles (in degrees) then can send the
scope to that location.
    #  @param O_alt Desired Altitude Angle in radians
    #  @param O_az Desired Azimuth Angle in radians
    #  @param O_rot Desired Rotation Angle in radians
    #  @param move Boolean value which determines if the function only
calculates the required lengths or also commands motion
    def goToAngles(self, O_alt, O_az, O_rot, move=False):

        # Assemble Translation Matrix from Base (b) reference frame to
Telescope (s)
        # Rotations Azimuth -> Altitude -> Image Rotations
        sTb = matmul(RotZ(O_rot+self.phi_rot),matmul(RotX(-
O_alt+self.phi_alt),RotY(-O_az+self.phi_az)))

        # Calculate Rotated Positions
        P1_r = matmul(sTb,self.P1_h)
        P2_r = matmul(sTb,self.P2_h)
        P3_r = matmul(sTb,self.P3_h)
        OA_r = matmul(sTb,self.OA_h)

        # Calculate Length
        #L_p1 = pow(pow((P1_r[0][0]-
self.P1_b[0][0]),2)+pow((P1_r[1][0]-
self.P1_b[1][0]),2)+pow((P1_r[2][0]-self.P1_b[2][0]),2),.5)
        #L_p2 = pow(pow((P2_r[0][0]-
self.P2_b[0][0]),2)+pow((P2_r[1][0]-
self.P2_b[1][0]),2)+pow((P2_r[2][0]-self.P2_b[2][0]),2),.5)
        #L_p3 = pow(pow((P3_r[0][0]-
self.P3_b[0][0]),2)+pow((P3_r[1][0]-
self.P3_b[1][0]),2)+pow((P3_r[2][0]-self.P3_b[2][0]),2),.5)
        L_p1 = rootsumsquare(P1_r, self.P1_b)
        L_p2 = rootsumsquare(P2_r, self.P2_b)
        L_p3 = rootsumsquare(P3_r, self.P3_b)
        # Need to check that the lengths are legitimate
        if abs(L_p1<self.L_p1min) or abs(L_p2<self.L_p2min) or
abs(L_p3<self.L_p3min):
            print('This position is beyond the minimum lengths')
        # Need to move actuators
        elif move == True:
            self.goToLengths(L_p1-self.L_p1min, L_p2-self.L_p2min,
L_p3-self.L_p3min)

            # Update current angles
            self.alt_cur = O_alt
            self.az_cur = O_az
            self.rot_cur = O_rot
        return(L_p1, L_p2, L_p3)


    ## @fn findHome
    #  Causes each actuator to shorten until it stalls then stops
    #  @b WARNING: Does not function!
    def findHome(self):
        '''
        Non-functioning function which is designed to implement
automatic home checking
        '''
        isOneHome = False
        isTwoHome = False
        isThreeHome = False
        self.actuatorOne.findHome()
        self.actuatorTwo.findHome()
        self.actuatorThree.findHome()

        while not(isOneHome and isTwoHome and isThreeHome):
            if self.actuatorOne.isHome():
                print('A1 home')
                isOneHome = True
            if self.actuatorTwo.isHome():
                isTwoHome = True
                print('A2 home')
            if self.actuatorThree.isHome():
                isThreeHome = True
                print('A3 home')


            time.sleep(.01)

        print("There's no place like home")

    ## @fn resetPositions
    #  Zeros the absolute position of each actuator.
    #  Used for defining the home position.
    def resetPositions(self):

        self.actuatorOne.resetPosition()
        self.actuatorTwo.resetPosition()
        self.actuatorThree.resetPosition()

    ## @fn goToLengths
    #  Commands each actuator to a specified length
    #  @param lengthOne Desired length for actuator one
    #  @param lengthTwo Desired length for actuator two
    #  @param lengthThree Desired length for actuator three
    def goToLengths(self, lengthOne, lengthTwo, lengthThree):

        self.actuatorOne.commandLength(lengthOne)
        self.actuatorTwo.commandLength(lengthTwo)
        self.actuatorThree.commandLength(lengthThree)

    ## @fn goVelocity
    #  Commands each actuator to move at a specified velocity
    #  @param velOne Velocity in length/sec
    #  @param velTwo Velocity in length/sec
    #  @param velThree Velocity in length/sec
    def goVelocity(self, velOne, velTwo, velThree):

        self.actuatorOne.commandVelocity(velOne)
        self.actuatorTwo.commandVelocity(velTwo)
        self.actuatorThree.commandVelocity(velThree)
```

```
## Calculates the root sum square of two vectors of size 3
#  @param a vector of size 3
#  @param b vector of size 3
#  @return The root-sum-of-squares of the difference of @c a and @c b
def rootsumsquare(a, b):
    rss = pow(pow((a[0][0]-b[0][0]),2)+pow((a[1][0]-
b[1][0]),2)+pow((a[2][0]-b[2][0]),2),.5)
    return rss

## Multiplies two matrices together @c a * @c b
#  @param a [m x n] matrix
#  @param b [n x o] matrix
def matmul(a, b):
    zip_b = zip(*b)
    zip_b = list(zip_b)
    return [[sum(ele_a*ele_b for ele_a, ele_b in zip(row_a, col_b)) for
col_b in zip_b] for row_a in a]

## Rotation matrix about the x axis
def RotX(rads):
    return [[1,0,0],[0,math.cos(rads),-
math.sin(rads)],[0,math.sin(rads),math.cos(rads)]]

## Rotation matrix about the y axis
def RotY(rads):
    return [[math.cos(rads),0,math.sin(rads)],[0,1,0],[-
math.sin(rads),0,math.cos(rads)]]

## Rotation matrix about the z axis
def RotZ(rads):
    return [[math.cos(rads),-
math.sin(rads),0],[math.sin(rads),math.cos(rads),0],[0,0,1]]
```

main.py

```
# -*- coding: utf-8 -*-
#
## @file main.py
#  Control file for a Three Degree of Freedom Parallel Actuator
Telescope Mount
#
#  @author Samuel Steejans Artho-Bentz
import pyb
import l6470nucleo
import time
import ActuatorDriver
import TelescopeDriver


## Stepper Resolution [steps/rev]
STEPPER_RESOLUTION = const (200)

## Number of microsteps
#  This must match @c NUMBER_OF_MICROSTEPS in @c TelescopeDriver.py and
#  @c STEP_SEL in @c l6470nucleo.py
MICROSTEPS = 4

## Leadscrew Resolution [rev/in]
SCREW_RESOLUTION = const(16)

# Set up pins

## SPI one standby/reset pin
stby_rst_pin1 = pyb.Pin.cpu.B5
## SPI two standyby/reset pin
stby_rst_pin2 = pyb.Pin.cpu.B3
## SPI one chip select pin
nCS1 = pyb.Pin.cpu.A4
## SPI two chip select pin
nCS2 = pyb.Pin.cpu.A10
## SPI number
spi_number = 1
print('pins initialized')

## spi object for motor drivers
spi_object = pyb.SPI (spi_number, mode=pyb.SPI.MASTER,
                           baudrate=2000000, polarity=1, phase=1,
                           bits=8, firstbit=pyb.SPI.MSB)

## l6470nucleo board object
Driver1 = l6470nucleo.Dual6470(spi_object,nCS1, stby_rst_pin1)
## l6470nucleo board object
Driver2 = l6470nucleo.Dual6470(spi_object,nCS2, stby_rst_pin2)

Driver1._get_params(0xD0,2)

# Define the 3 actuators
## Right actuator when viewed from the origin looking along the X axis
actuatorOne =
ActuatorDriver.Actuator(STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS,
Driver1, 1)
```

```
## Left actuator when viewed from the origin looking along the X axis
actuatorTwo =
ActuatorDriver.Actuator(STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS,
Driver1, 2)
## Rear, horizontal actuator
actuatorThree =
ActuatorDriver.Actuator(STEPPER_RESOLUTION*SCREW_RESOLUTION*MICROSTEPS,
Driver2, 1)

## Define the telescope based on the actuators above
Telescope = TelescopeDriver.Telescope(actuatorOne, actuatorTwo,
actuatorThree)


# Create some basic utility functions

## @fn estop
#  commands all motors to halt immediately.
def estop():
   Driver1.HardHiZ(1)
   Driver1.HardHiZ(2)
   Driver2.HardHiZ(1)

## @fn Go(speed)
#  commands all motors to go at the specified speed.
#  directly communicates with the stepper motors.
#  @param speed steps/second
def Go(speed):
   Driver1.run(1, speed)
   Driver1.run(2, speed)
   Driver2.run(1, speed)

## @fn getPos
#  Queries the actuators for their current lengths
def getPos():
   print(['Actuator One is at: ' +
str(actuatorOne.getCurrentLength()+Telescope.L_p1min)])
   print(['Actuator Two is at: ' +
str(actuatorTwo.getCurrentLength()+Telescope.L_p2min)])
   print(['Actuator Three is at: ' +
str(actuatorThree.getCurrentLength()+Telescope.L_p3min)])

## @fn getStatus(motornumber)
#  Requests the status of the specified motor
#  @param motornumber The motor associated with the desired actuator:
1, 2, or 3
def getStatus(motornumber):
   if motornumber ==1:
       actuatorOne.controller.GetStatus(1, 1)
   elif motornumber ==2:
       actuatorTwo.controller.GetStatus(2, 1)
   elif motornumber ==3:
       actuatorThree.controller.GetStatus(1, 1)

## @fn testRepeatability(numloops)
#  Runs through a series of points @c numloops times.
#  Waits for the user to hit a key before moving on to the next point.
#  Used for testing.
#  @param numloops Desired number of times the points should be gone
through.
def testRepeatability(numloops=10):

  pt1 = [.349066, 0, 0]
  pt2 = [.523599, -0.17453, 0]
  pt3 = [0.698132, -0.34907, 0]
  pt4 = [0.872665, -0.17453, 0]
  pt5 = [0.959931, 0, 0]
  pt6 = [0.872665, 0.174533, 0]
  pt7 = [0.785398, .349066, 0]
  pt8 = [0.523599, 0.174533, 0]
  Telescope.goToAngles(pt1[0], pt1[1], pt1[2], 1)
  print('point 1')
  input()
  Telescope.goToAngles(pt2[0], pt2[1], pt2[2], 1)
  print('point 2')
  input()
  Telescope.goToAngles(pt3[0], pt3[1], pt3[2], 1)
  print('point 3')
  input()
  Telescope.goToAngles(pt4[0], pt4[1], pt4[2], 1)
  print('point 4')
  input()

  Telescope.goToAngles(pt6[0], pt6[1], pt6[2], 1)
  print('point 6')
  input()
  Telescope.goToAngles(pt7[0], pt7[1], pt7[2], 1)
  print('point 7')
  input()
  Telescope.goToAngles(pt8[0], pt8[1], pt8[2], 1)
  print('point 8')
  input()
  for x in range(1,numloops):
    Telescope.goToAngles(pt1[0], pt1[1], pt1[2], 1)
    print('going to 20,0')
    time.sleep(80)
    Telescope.goToAngles(pt2[0], pt2[1], pt2[2], 1)
    print('going to 30,-10')
    time.sleep(80)
    Telescope.goToAngles(pt3[0], pt3[1], pt3[2], 1)
    print('going to 40,-20')
    time.sleep(80)
    Telescope.goToAngles(pt4[0], pt4[1], pt4[2], 1)
    print('going to 50, ,0')
    time.sleep(80)
    Telescope.goToAngles(pt6[0], pt6[1], pt6[2], 1)
    print('going to 50,10')
    time.sleep(80)
    Telescope.goToAngles(pt7[0], pt7[1], pt7[2], 1)
    print('going to 45,20')
    time.sleep(80)
    Telescope.goToAngles(pt8[0], pt8[1], pt8[2], 1)
    print('going to 30,10')
    time.sleep(100)
  input()
## @fn testGrid
#  Moves to a series of points and waits for user input at each one.
#  Used for testing.
def testGrid():
  alts = [.296705973, .34906585, .436332313, .523598776, .610865238]
  azs = [.174532925, .087266463, 0, -.087266463, -.174532925]
  for x in azs:
    for y in alts:
      Telescope.goToAngles(y, x, 0,1)
      print('Altitude: ', str(y), ' Azimuth: ', str(x))
      input()
```

APPENDIX C MATLAB PROGRAM

Relative Rotation Testing Matlab code

```
%% Line-Angle Calculation from Visual Inspection
% Sam Artho-Bentz
% 7/30/2018

%% Clean up from previous runs
clc
clear all
close all

%% Load the file
filename = 'IMG_7861.JPG';
imshow(filename)

%% Wait for user to draw a line which matches the angle of the negative
line
h1=imline; %negative line
position1 = wait(h1);
angle1 = tan((position1(1,2)-position1(2,2))/(position1(1,1)-
position1(2,1)));
close
imshow(filename)

%% Wait for user to draw a line which matches the angle of the neutral
line
h2=imline; %zero line
position2 = wait(h2);
angle2 = tan((position2(1,2)-position2(2,2))/(position2(1,1)-
position2(2,1)));
close
imshow(filename)

%% Wait for user to draw a line which matches the angle of the positive
line
h3=imline; %positive line
position3 = wait(h3);
angle3 = tan((position3(1,2)-position3(2,2))/(position3(1,1)-
position3(2,1)));
close

%% Output results
disp('corrected negative line is')
disp(angle1-angle2)
disp('corrected positive line is')
disp(angle3-angle2)
```

Tolerance Stackup Testing Matlab Code

```
%% Tolerance Stackup Image Processing
% Samuel S. Artho-Bentz


%% Clean Up
clc
clear all
close all

%% Read in Photo
myPhoto = imread('img_7849.jpg');

%% Set system param
% distance to board from system origin
% dboardMin_mm was measured at 1260. however, it is found that the
result
% is VERY dependant on small changes to this
dBoardMin_mm = 1265;
dBoardMin_in = dBoardMin_mm/25.4;
%% Find location of red reference points
% The red dots are 12 inches apart
redDots=myImgStats(myPhoto,1);

%% Calculate a Vertical Scale for pixels per inch
% Find vertical distances between points in pixels

vDist = [redDots(1,2)-redDots(3,2), redDots(2,2)-redDots(3,2),
redDots(1,2)-redDots(2,2)];

% Actual distance in inches

vDistRef=[12, 12, 0];

% Apply simple linear regression to find a conversion between pixels
and inches in the vertical direction
% Measurement[inches] = vertreg(1) * Measurement[pixels] + vertreg(2)

vertReg=myLinReg(vDist,vDistRef);

% Verify linear regression is reasonable
vDistCheck=vertReg(1)* vDist + vertReg(2);

%% Calculate a Horizontal Scale for pixels per inch
% Find Horizontal distances between points in pixels

hDist = [redDots(3,1)-redDots(1,1), redDots(2,1)-redDots(3,1),
redDots(2,1)-redDots(1,1)];

% Actual distance in inches

hDistRef=[12, 0, 12];

% Apply simple linear regression to find a conversion between pixels
and inches in the vertical direction
```

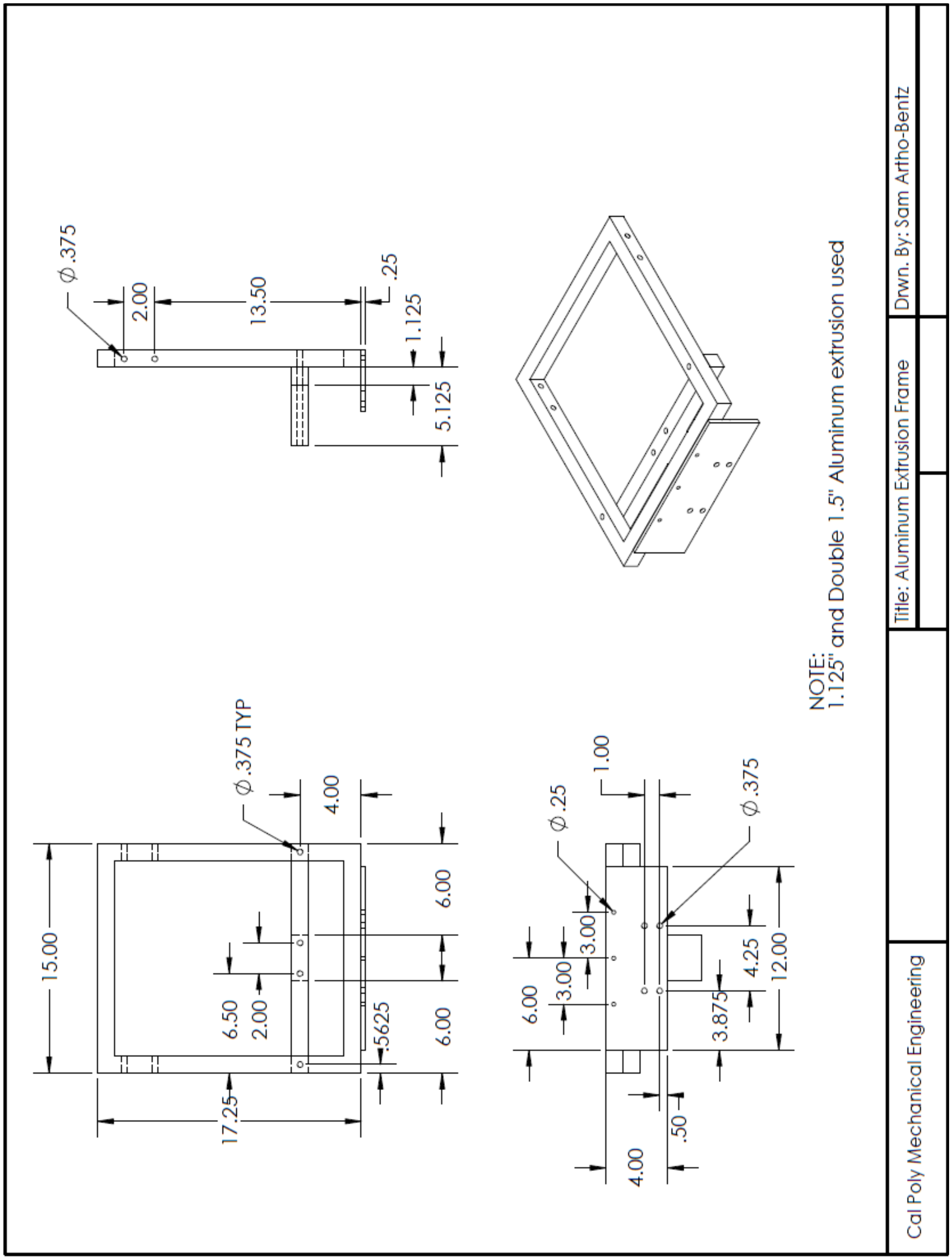
```
% Measurement[inches] = vertreg(1) * Measurement[pixels] + vertreg(2)

horReg=myLinReg(hDist,hDistRef);

% Verify linear regression is reasonable
hDistCheck=horReg(1)* hDist + horReg(2);

%% Find all black dots

myGSPhoto=rgb2gray(myPhoto);
myBWPhoto=myGSPhoto<100;
rp=regionprops(myBWPhoto,myGSPhoto,'Centroid');
centroids = cat(1, rp.Centroid);

% Remove the centroids of the red dots from the list
for i = 1:length(centroids)
    if i > length(centroids)
        break
    end
    if ((centroids(i,1)>redDots(1,1)*.9) &&
(centroids(i,1)<redDots(1,1)*1.1)) && ((centroids(i,2)>redDots(1,2)*.9)
&& (centroids(i,2)<redDots(1,2)*1.1))
        centroids(i,:)=[];
    end
    if ((centroids(i,1)>redDots(2,1)*.9) &&
(centroids(i,1)<redDots(2,1)*1.1)) && ((centroids(i,2)>redDots(2,2)*.9)
&& (centroids(i,2)<redDots(2,2)*1.1))
        centroids(i,:)=[];
    end
    if ((centroids(i,1)>redDots(3,1)*.9) &&
(centroids(i,1)<redDots(3,1)*1.1)) && ((centroids(i,2)>redDots(3,2)*.9)
&& (centroids(i,2)<redDots(3,2)*1.1))
        centroids(i,:)=[];
    end
end

%% Find the matching pairs of points by looking for those that are
within 150 pixels of each other.
pairIndex = 1;
for i = 1:length(centroids)
    for j = 1:length(centroids)
        if (i~=j)&& (j>i)
            if hypot(centroids(i,1)-centroids(j,1), centroids(i,2)-
centroids(j,2))<150
                centroidPairs(pairIndex, :) = [i,j];
                pairIndex= pairIndex +1;
            end
        end
    end
end



%% Convert centroid coordinates from pixels to inches using the scales
above
centroidsInches(:,1) = horReg(1) * centroids(:,1) + horReg(2);
centroidsInches(:,2) = vertReg(1) * centroids(:,2) + vertReg(2);
```

```
%% Create new variable pair data
for i = 1:length(centroidPairs)
    % pairedCoord = [p1x p1y p2x p2y]
    pairedCoord(i,:) = [centroidsInches(centroidPairs(i,1),1)
centroidsInches(centroidPairs(i,1),2)
centroidsInches(centroidPairs(i,2),1)
centroidsInches(centroidPairs(i,2),2)];
end

%% Find distance between each pair of points
for i = 1:length(centroidPairs)
    pairDistance(i,1) = hypot(pairedCoord(i,1)-
pairedCoord(i,3),pairedCoord(i,2)-pairedCoord(i,4));
end

%% use the distance from the board (known)
% define one dot (center bottom) as 'correct'
% use locations of all the dots, along with the commanded angles, to do
% some sort of error? +/- theoretical angle?

%% Reassociate all points with new origin
% New origin at bottom middle pair
middleBottomPair = 15; % Determined manually (az = 0, alt = 17)
middleBottomCoord = [mean([pairedCoord(middleBottomPair, 1),
pairedCoord(middleBottomPair,3)]) mean([pairedCoord(middleBottomPair,
2), pairedCoord(middleBottomPair,4)])];
middleBottomWRTZero = [0, tand(17)*dBoardMin_in];
% All points wrt 0,0
pairedCoord = [pairedCoord(:,1)-middleBottomCoord(1), -
pairedCoord(:,2)+middleBottomCoord(2)+middleBottomWRTZero(2),
pairedCoord(:,3)-middleBottomCoord(1), -
pairedCoord(:,4)+middleBottomCoord(2)+middleBottomWRTZero(2)];

%% Find average point between each pair
for i=1:length(pairedCoord)
    pairedCoord(i,5) = mean([pairedCoord(i,1),pairedCoord(i,3)]);
    pairedCoord(i,6) = mean([pairedCoord(i,2),pairedCoord(i,4)]);
end
%% Associate commanded angles to each pair
% Ideally would automatically assign commanded angles to each pair but
for
% sake of time, doing it manually. adding [az, alt] to each pair
pairedCoord(1,7:8) = [-10, 35];
pairedCoord(2,7:8) = [-10, 30];
pairedCoord(3,7:8) = [-10, 25];
pairedCoord(4,7:8) = [-10, 20];
pairedCoord(5,7:8) = [-10, 17];
pairedCoord(6,7:8) = [-5, 35];
pairedCoord(7,7:8) = [-5, 30];
pairedCoord(8,7:8) = [-5, 25];
pairedCoord(9,7:8) = [-5, 20];
pairedCoord(10,7:8) = [-5, 17];
pairedCoord(11,7:8) = [0, 35];
pairedCoord(12,7:8) = [0, 30];
pairedCoord(13,7:8) = [0, 25];
pairedCoord(14,7:8) = [0, 20];
pairedCoord(15,7:8) = [0, 17];
pairedCoord(16,7:8) = [5, 17];
pairedCoord(17,7:8) = [5, 20];
pairedCoord(18,7:8) = [5, 25];
pairedCoord(19,7:8) = [5, 30];
pairedCoord(20,7:8) = [5, 35];
pairedCoord(21,7:8) = [10, 17];
pairedCoord(22,7:8) = [10, 20];
pairedCoord(23,7:8) = [10, 25];
pairedCoord(24,7:8) = [10, 30];
pairedCoord(25,7:8) = [10, 35];

%% Associate Theoretical x,y distances with each pair
% pairedCoord = [x1, y1, x2, y2, xavg, yavg, azCmd, altCmd,
xTheoretical, yTheoretical]

%
for i=1:length(pairedCoord)
    pairedCoord(i,9) = dBoardMin_in*tand(pairedCoord(i, 7));
    pairedCoord(i,10) =
dBoardMin_in*tand(pairedCoord(i,8))/cosd(pairedCoord(i,7));
end

%% Calc Percent Error of each point from its theoretical location
% This gives poor comparison values because the range of distances
examined
% are so different. Better to back calculate from x1,y1,etc into what
the
% angle its looking at is. This will be better for doing error.
for i = 1:length(pairedCoord)
    percentErrorDistance(i,1) = abs((pairedCoord(i,1)-
pairedCoord(i,9))/pairedCoord(i,9))*100;
    percentErrorDistance(i,2) = abs((pairedCoord(i,2)-
pairedCoord(i,10))/pairedCoord(i,10))*100;
    percentErrorDistance(i,3) = abs((pairedCoord(i,3)-
pairedCoord(i,9))/pairedCoord(i,9))*100;
    percentErrorDistance(i,4) = abs((pairedCoord(i,4)-
pairedCoord(i,10))/pairedCoord(i,10))*100;
    percentErrorDistance(i,5) = abs((pairedCoord(i,5)-
pairedCoord(i,9))/pairedCoord(i,9))*100;
    percentErrorDistance(i,6) = abs((pairedCoord(i,6)-
pairedCoord(i,10))/pairedCoord(i,10))*100;
    absoluteErrorDistance(i,1) =  abs((pairedCoord(i,1)-
pairedCoord(i,9)));
    absoluteErrorDistance(i,2) =  abs((pairedCoord(i,2)-
pairedCoord(i,10)));
    absoluteErrorDistance(i,3) =  abs((pairedCoord(i,3)-
pairedCoord(i,9)));
    absoluteErrorDistance(i,4) =  abs((pairedCoord(i,4)-
pairedCoord(i,10)));
    absoluteErrorDistance(i,5) =  abs((pairedCoord(i,5)-
pairedCoord(i,9)));
    absoluteErrorDistance(i,6) =  abs((pairedCoord(i,6)-
pairedCoord(i,10)));
end
testVar = [percentErrorDistance(:,5:6) absoluteErrorDistance(:,5:6)];
```

```
%% Calc what angle each point is theoretically looking at.
% theoreticalAngle = [azCmd, altCmd, p1Az, p1Alt, p2Az, p2Alt, pavgAz,
% pavgAlt]
for i = 1:length(pairedCoord)
    angleBackCalculated(i,1:2) = pairedCoord(i, 7:8);
    angleBackCalculated(i,3) = atan2d(pairedCoord(i,1), dBoardMin_in);
    angleBackCalculated(i,4) =
atan2d(pairedCoord(i,2)*cosd(angleBackCalculated(i,3)), dBoardMin_in);
    angleBackCalculated(i,5) = atan2d(pairedCoord(i,3), dBoardMin_in);
    angleBackCalculated(i,6) =
atan2d(pairedCoord(i,4)*cosd(angleBackCalculated(i,5)) ,dBoardMin_in);
    angleBackCalculated(i,7) = atan2d(pairedCoord(i,5), dBoardMin_in);
    angleBackCalculated(i,8) =
atan2d(pairedCoord(i,6)*cosd(angleBackCalculated(i,7)), dBoardMin_in);

end

%% Calc Percent Percent Error of back calculated angle vs command angle
for i = 1:length(angleBackCalculated)
   percentErrorAngular(i,1) = abs((angleBackCalculated(i,3)-
angleBackCalculated(i,1))/ angleBackCalculated(i,1))*100;
   percentErrorAngular(i,2) = abs((angleBackCalculated(i,4)-
angleBackCalculated(i,2))/ angleBackCalculated(i,2))*100;
   percentErrorAngular(i,3) = abs((angleBackCalculated(i,5)-
angleBackCalculated(i,1))/ angleBackCalculated(i,1))*100;
   percentErrorAngular(i,4) = abs((angleBackCalculated(i,6)-
angleBackCalculated(i,2))/ angleBackCalculated(i,2))*100;
   percentErrorAngular(i,5) = abs((angleBackCalculated(i,7)-
angleBackCalculated(i,1))/ angleBackCalculated(i,1))*100;
   percentErrorAngular(i,6) = abs((angleBackCalculated(i,8)-
angleBackCalculated(i,2))/ angleBackCalculated(i,2))*100;
end
% figure();
% image(myPhoto);
% axis image;
% hold on;
% plot(centroids(:,1),centroids(:,2), 'b*')
% hold off
%


function myReg = myLinReg(x,y)
% Calculates the slope and intercept of the linear regression line
using
% the least squares method outlined in ME 236 Course Pack

% Determining number of data points
n = length(x);

% Initializing variables
xy = 0;
xSq = 0;

% Determines sum of x*y and x^2 values
for i = 1:n
    xy = xy + x(i)*y(i);
    xSq = xSq + (x(i)^2);
```

```
end

% Calculates slope and intercept of linear regression
intercept = (sum(x)*xy - xSq*sum(y))/((sum(x))^2 - n*xSq);
slope = (sum(x)*sum(y) - n*xy)/((sum(x))^2 - n*xSq);

% Output calculated slope and intercept
myReg = [slope,intercept];

end



function centroids = myImgStats(myPhoto,mask)

myGrey = rgb2gray(myPhoto);

myMask = imsubtract(myPhoto(:,:,mask),myGrey);

myFiltered = medfilt2(myMask,[3 3]);

thres = graythresh(myFiltered);

myBinary = im2bw(myFiltered,thres);

myObjects = bwareaopen(myBinary,300);

labels = bwlabel(myObjects,8);

stats = regionprops(labels,'BoundingBox','Centroid');

centroids = cat(1, stats.Centroid);

end
```

APPENDIX D PARTS

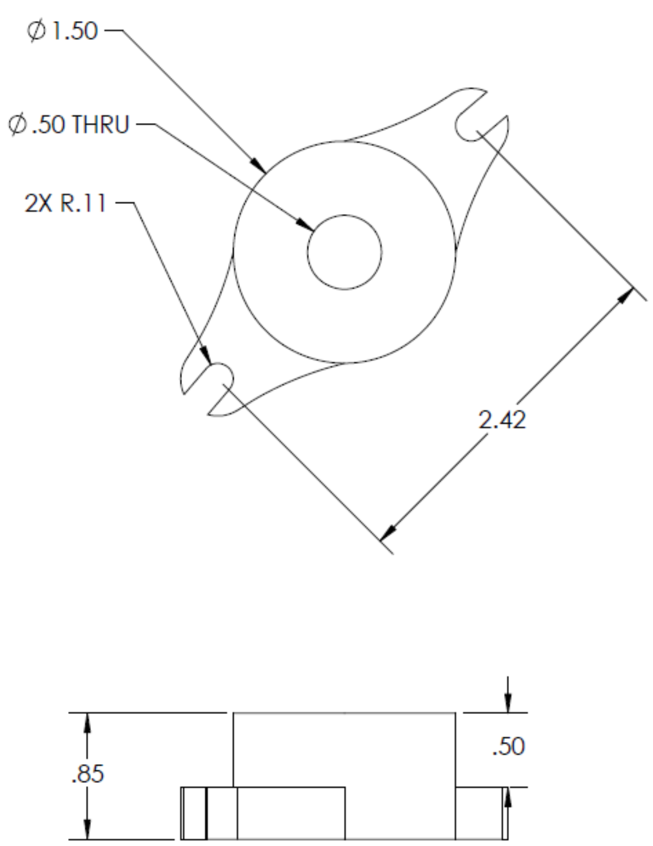


NOTE:
1.125" and Double 1.5" Aluminum extrusion used

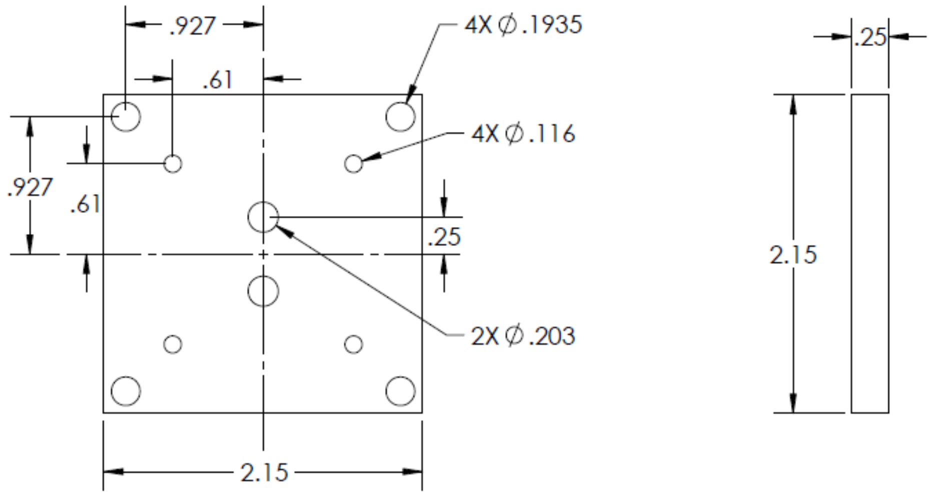
Title: Aluminum Extrusion Frame

Drwn. By: Sam Artho-Bentz

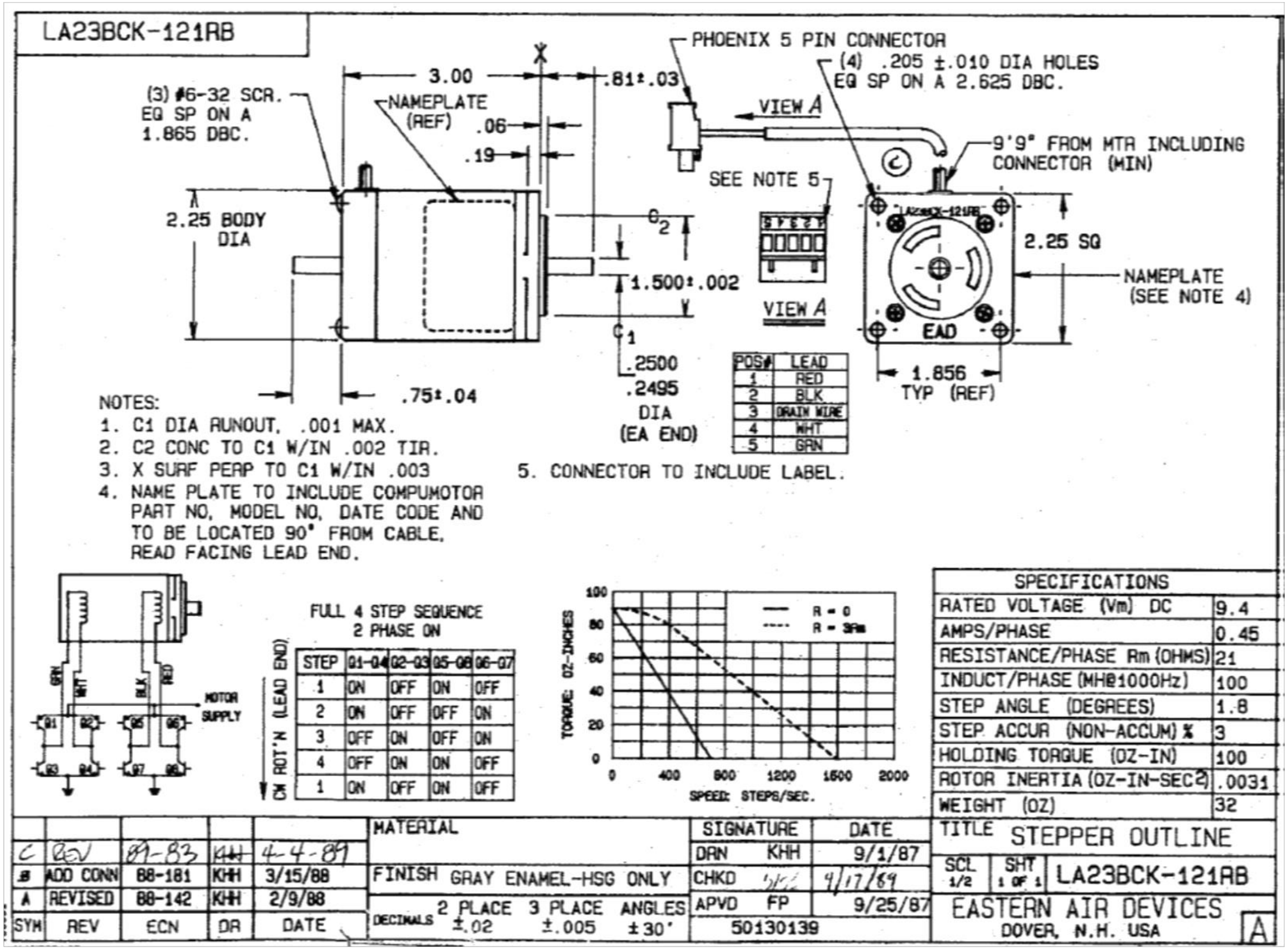
Cal Poly Mechanical Engineering



Cal Poly Mechanical Engineering		Title: Stepper Motor Spacer	Drwn. By: Sam Artho-Bentz



Cal Poly Mechanical Engineering		Title: Motor Conversion Plate	Drwn. By: Sam Artho-Bentz



NOTES:
1. C1 DIA RUNOUT, .001 MAX.
2. C2 CONC TO C1 W/IN .002 TIR.
3. X SURF PERP TO C1 W/IN .003
4. NAME PLATE TO INCLUDE COMPUMOTOR PART NO, MODEL NO, DATE CODE AND TO BE LOCATED 90° FROM CABLE, READ FACING LEAD END.
5. CONNECTOR TO INCLUDE LABEL.

POS#	LEAD
1	RED
2	BLK
3	DRAIN WIRE
4	WHT
5	GRN

FULL 4 STEP SEQUENCE
2 PHASE ON

STEP	Q1-Q4	Q2-Q3	Q5-Q8	Q6-Q7
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	ON
3	OFF	ON	OFF	ON
4	OFF	ON	ON	OFF
1	ON	OFF	ON	OFF

SPECIFICATIONS	
RATED VOLTAGE (Vm) DC	9.4
AMPS/PHASE	0.45
RESISTANCE/PHASE Rm (OHMS)	21
INDUCT/PHASE (MH@1000Hz)	100
STEP ANGLE (DEGREES)	1.8
STEP ACCUR (NON-ACCUM) %	3
HOLDING TORQUE (OZ-IN)	100
ROTOR INERTIA (OZ-IN-SEC2)	.0031
WEIGHT (OZ)	32

SYM	REV	ECN	DR	DATE
C	REV	89-83	KHH	4-4-89
B	ADD CONN	88-181	KHH	3/15/88
A	REVISED	88-142	KHH	2/9/88

MATERIAL

FINISH GRAY ENAMEL-HSG ONLY

DECIMALS 2 PLACE ±.02 3 PLACE ±.005 ANGLES ±30'

SIGNATURE		DATE
DRN	KHH	9/1/87
CHKD		4/17/89
APVD	FP	9/25/87

50130139

TITLE STEPPER OUTLINE

SCL 1/2 SHT 1 OF 1 LA23BCK-121RB

EASTERN AIR DEVICES
DOVER, N.H. USA

A