ECE 4972

Automated Satellite Tracking

John Buglione, Steven Gulotta, Jordan Ly

Department of Electrical and Computer Engineering
Villanova University, 800 Lancaster Ave, PA 19085

27 March 2014

ABSTRACT

The Automated Satellite Tracking System is a low-cost and accurate device designed and implemented by this project team. It will calculate the real time positions of various satellites using Keplerian elements downloaded from the AMSAT website. It will then feed these coordinates to a microcontroller which will orient an antenna mounted on a pan-tilt head to point in the direction of a satellite. These calculations will be repeated, thus allowing the antenna to track the satellite's path. This project has been completed and meets its major design goals

ACKNOWLEDGEMENTS

We would like to acknowledge the following people who assisted us in numerous ways, were resources of knowledge for us, and whose efforts were instrumental to the successful completion of our project.

Firstly, we express our deepest gratitude to our advisor, Dr. Stephen Konyk, who guided us through each step of the project. His insights and knowledge into the physics of satellites gave us a strong foundation upon which to base our work.

We would also like to thank Helen Cook for ordering many of the parts we needed to complete this project and dealing with our constant requests for more materials.

Finally we would like to thank Chris Townend for giving of his time and skills so freely in assisting us in the mechanics of our project. Without his help we could not have built our system successfully. He took days out of his schedule to assist us and gave us advice every step of the way.

TABLE OF CONTENTS

1. INTRODUCTION

The system outlined here uses a stand-alone MATLAB program to calculate the real time positions of low earth orbit satellites [1] and using these generated locations controls the orientation of a directional antenna that can be used to communicate with and receive data from said satellites. Most modern communication techniques in use today require terrestrial infrastructure or are very high in cost. The system proposed in this paper would be low cost and does not rely upon pre-existing terrestrial infrastructure.

The system outlined in this paper forms a method for accurately tracking satellites by pointing an antenna in the direction of the satellite. Communication with the satellite in question is outside the scope of this system. Communication would be the next step in creating a marketable system; however our focus here was creating a method for automatically tracking the satellites.

Modern communication systems such as cell phones and the internet require terrestrial infrastructure and are not a viable method of communication in remote areas or in emergency situations, where conventional means of communication are disabled. Iridium satellite phones are a commonly used alternative communication technique. They use portable radios to communicate directly with satellites, and therefore would be a viable technology for use in remote areas. However, even the most basic satellite phones can cost close to $1000, excluding monthly service costs. HAM radio is another popular option for communication in remote locations. HAM radios generally costs less than Iridium Satellite phones due to lack of service costs; however they generally do not communicate via satellite and therefore cannot be used to receive data transmitted via satellite.

A similar system to the one outlined here, filed for patent in 2011, uses motors to control an antenna in order to track a geosynchronous satellite. The system proposed in the patent also contains an estimator that calculates the reception quality and corrects the placement of the antenna in order maintain maximum signal. The system outlined here differs in that it will be used to track low earth orbit satellites, which move rapidly across the sky rather than staying relatively motionless, and will use set calculated orbitals to locate the satellite rather than signal strength based correction, a process which would increase complexity and cost. [2]

2. DESIGN OVERVIEW

This project can be divided into three regions, the user interface and calculation of satellite location, the calculation of antenna displacement and the physical movement of the antenna. The division of the overall tasks is shown in Figure 1.

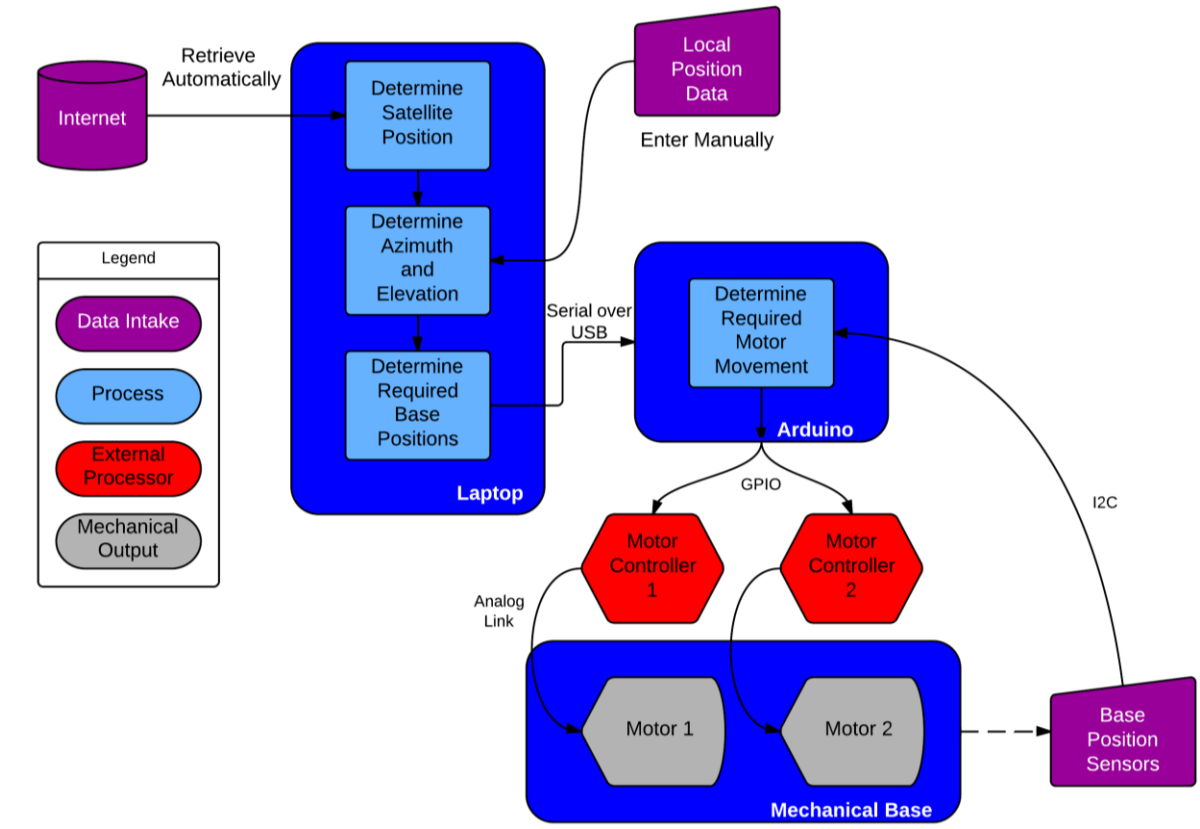


Figure 1 – System Block Diagram

3. SATELITTE CALCULATION SUBSYSTEM

This system follows a satellite by predicting its orbital path. Unlike the traditional sense of "tracking", this does not lock onto a satellite and follow its path, but instead constantly calculates a satellite's expected position and updates to follow a satellite's path across the sky.

3.1 Keplerian Elements

In order to do these calculations, unique parameters provided by AMSAT for satellites sent into space are required. These parameters are known as Keplerian elements. There are seven that are of interest to us and they are as follows: Epoch Time, Inclination, Right Ascension of Ascending Node (RAAN), Argument of Perigee, Eccentricity, Mean Motion, and Mean Anomaly. These seven elements define an ellipse, orient it about the earth, and place the satellite on the ellipse at a particular time. Epoch Time is the time at which the satellite was launched into space. Inclination is angle between the Equatorial and orbital plane. RAAN is an angle, measured at the center of the earth, from the vernal equinox to the ascending node, as seen in Figure 2. Argument of Perigee is the angle, measured at the center of the earth, from the ascending node to perigee (The point where the satellite is closest to the earth), again Figure 2. Eccentricity is how round the ellipse (orbit) is. Mean Motion is the average speed of the satellite.

Finally, Mean Anomaly is an angle measured on the orbital plane where 0 degrees is perigee and 180 degrees is apogee [3]. With these seven Keplerian elements it is possible to calculate and accurately predict the orbital path of a satellite.

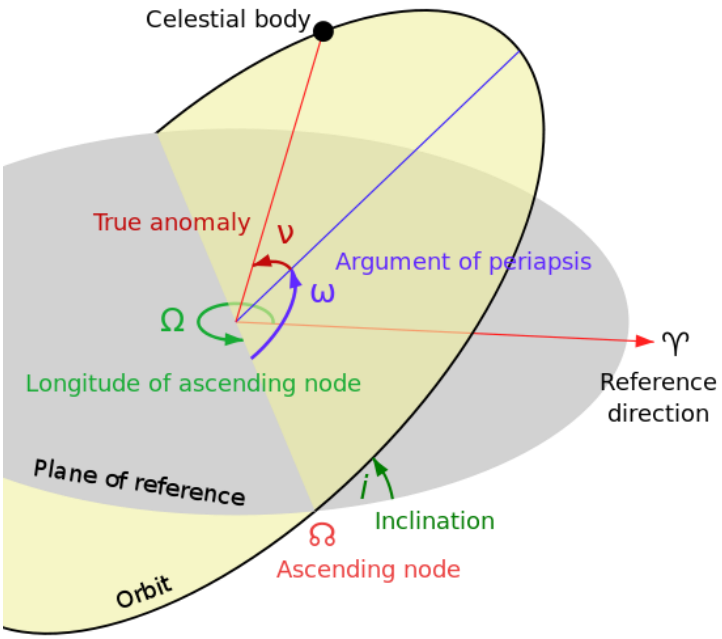


Figure 2 – Keplerian Elements

3.2 Location Calculations

There are three steps involved with using Keplerian elements to calculate the azimuth and elevation of a satellite. The first step is to calculate the satellite's position with respect to the vernal equinox, which is the point at which the sun crosses the celestial equator (equator extended into space) and used as a reference point. The next step is to calculate earth's position with respect to the vernal equinox. Now that the position of both the earth and the satellite is with respect to the same reference point, subtract the two to get the satellite's position with respect to earth. However, since this in the Geocentric Equatorial plane system. The final step is to convert that into the Topocentric Horizon system, which provides azimuth and elevation.

There is an open source Python library, called PyEphem that does astronomical calculations such as the one described above. By inputting your current latitude and longitude, it can calculate the azimuth and elevation of a satellite, as seen in Appendix A.2. Since the goal is to create single MATLAB program that controls the entire system, this library was used in to create a python script, called by MATLAB, to calculate a satellite's azimuth and elevation relative to a point on earth.

A MATLAB function, Appendix A.3, was created to load the Python script and input three variables: satellite index, user latitude, and user longitude. The script then outputs both satellite azimuth and elevation. The main program is a MATLAB graphical user interface shown in Figure 3. The code used to implement this GUI can be found in Appendix A.1. The user will enter their current latitude and longitude into the GUI and then select from a range of satellites to track. Once the program is started, the aforementioned MATLAB function will be called and the satellite azimuth and elevation will be displayed on the screen and periodically updated. It will also send this information to the Arduino microcontroller so that it can move the motors to the desired position.

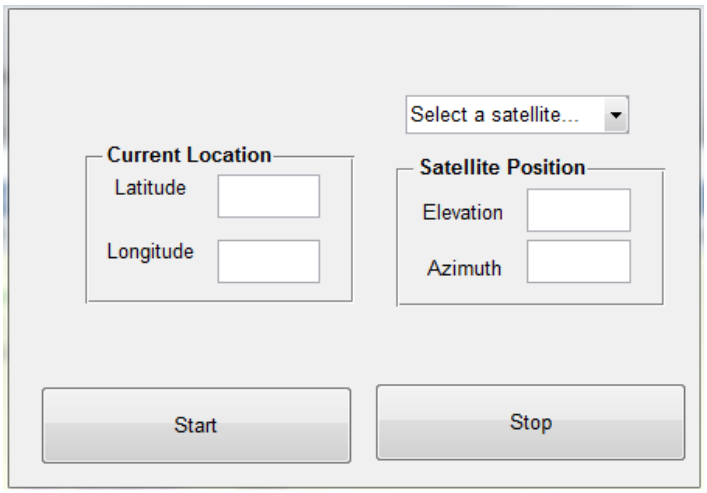


Figure 3 – MATLAB GUI

4. MOTOR DISPLACEMENT SUSBSYTEM

The Motor Displacement subsystem was used to move the antenna via stepper motors to point to the location calculated in the Satellite Calculation Subsystem.

4.1 Compass Feedback

Once the current position of our satellite of interest was calculated, the antenna needed to point to this location. This requires the use of a feedback loop incorporating a compass chip that would give the current location of the antenna and use that to calculate the displacement of the motors.

The chip that was chosen for this project was a SparkFun LSM303DLMTR tilt compensated compass. The chip was chosen for its low cost, high reliability and most importantly its tilt compensation features. These allow the chip to accurately determine magnetic north regardless of its orientation. This was important for us because our system tilts to follow the path a satellite traces across the sky. If the chip was not tilt compensated the chip would not function properly when the antenna had an elevation relative to the horizon. Of equal importance, the tilt compensated compass also contains an accelerometer. This is needed to get the current tilt of the antenna, which is directly related to the elevation vector of the satellite calculation.

The LSM303DLMTR's built in library does not come with a default method for reading the raw tilt data from the chip. In order to make use of this data, the register where this data is stored was located and the chip's library was edited to contain a method for returning the tilt of the chip in degrees with respect to the horizon [5]. This method is located in Appendix A.6.

The LSM303DLMTR chip is very sensitive to minor changes in the surrounding magnetic field as well as any small motion or vibration the chip undergoes. To ensure that a false value is not being used in our position calculation ten successive compass and accelerometer readings are taken and then averaged. The average value of these readings is then used in calculations.

The first method of calculating the motor displacement involved sending the compass data to MATLAB, calculating displacement with MATLAB, and using MATLAB to send control signals to the Arduino in order to move the motors. This was to be accomplished using MATLAB's built in Arduino Communication Library. This method is demonstrated in Figure 4. Steps completed by the Arduino are shown in blue and steps completed by MATLAB are shown in red.

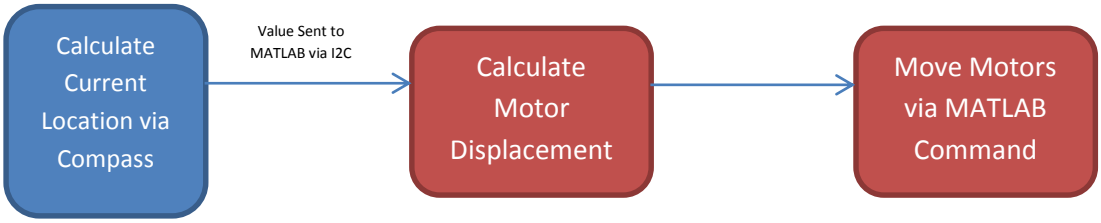


Figure 4 – System Outline

The LSM303DLMTR, however, communicates using an I2C interface [5] and can be attached directly to the Arduino Leonardo Microprocessor. The MATLAB Arduino Communication Library does not have a built in method for reading I2C from the Arduino. An attempt was made to edit the library to include I2C communication but this was determined to be more complicated than alternative solutions.

4.2 Serial Communication

After I2C communication using the built in MATLAB Arduino Communication Library was deemed too complicated an alternative method was devised to use serial communication between MATLAB and the Arduino, rather than the aforementioned Arduino communication library. The new system is outlined in Figure 5. Steps completed by the Arduino are shown in blue and steps completed by MATLAB are shown in red.

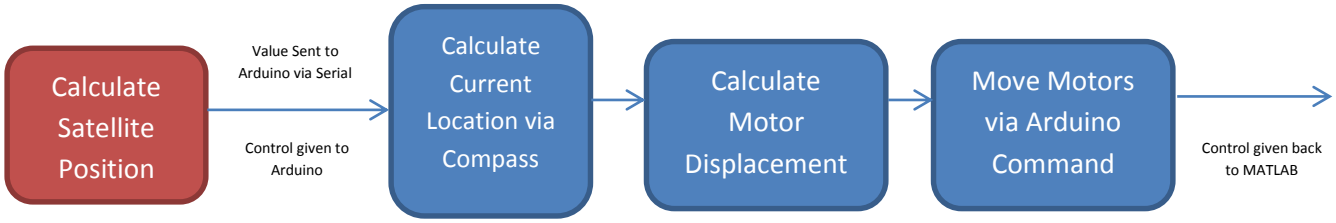


Figure 5 – New System Outline

The new system of interfacing the Arduino with MATLAB does not use the built in library, but rather communicates data by converting it to a string and sending this string via a serial link between the Arduino microcontroller and the computer. In order to successfully do this and get the timing correct for sending data back and forth, a method of handshaking needed to be devised. Between each serial communication step MATLAB will set a status value for the step and will not move to the next step until the same status value appears over the serial. Once a responding value has been acknowledged, MATLAB knows that Arduino is ready to receive that piece of data and then sends it through.

4.3 Motor Movement

Once the satellite position is received by Arduino from MATLAB, the antenna displacement is calculated by subtracting the satellite position from the antenna's current position. This displacement, in the form of degrees needed to move, is then converted to number of steps each stepper motor needs to take. The actual movement of the stepper motors is implemented through the use of the EasyDriver Stepper Motor Driver chip. The motors require a driving circuit to provide the correct pulse to move one step. Rather than designing a driving circuit from scratch, the EasyDriver was used for to its reliability and simplicity of use. The EasyDriver also breaks down each step taken by the stepper motor into 8 separate micro steps. Without using these micro steps, we would be limited to steps of 1.8^{o}. Using the EasyDriver, step size was decreased to 0.22^{o} [4].

A loop was created for each motor and driver to take that the correct number of steps as calculated by Arduino. The motors move separately, with the motor controlling azimuth moving first followed by the motor controlling elevation. Due to the slow nature of the satellites' paths we are tracking and the speed of the motors used and the wide beam of the directional antenna used; this does not create any problems with the system lagging the satellites actual movement. Once the motors have both moved to the correct positions, Arduino sends control back to MATLAB which, as long as the user has not ended the run, does another position calculation.

5. HARDWARE SUSBSYTEM

The proper interaction between hardware and software is crucial to this project. As such, a logical and effective hardware subsystem was needed in order to transform the rotation of the motors, controlled by the electronics of the system, into movement of the antenna or other pointing device. The hardware system also needed to be robust enough to survive the emergency conditions outlined in this project's objectives.

5.1 Design Decisions

Based on the necessary movements of the antenna, azimuth and elevation, seen in Figure 6 below, a few possible design architectures for the hardware subsystem were evaluated and the best design was chosen.

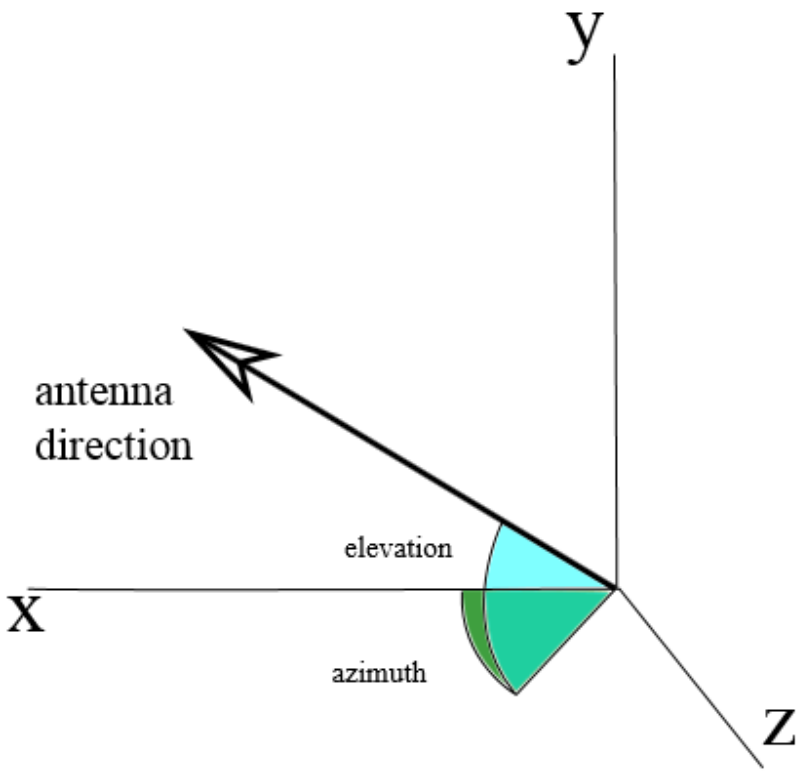


Figure 6 – Antenna Direction

Design	Pro	Con
Direct Drive	• Easy to implement • Less parts	• Large stepper motors required • High holding torque required • Least smooth (less steps per revolution of antenna)
Belt Drive	• Lower torque motor • Lighter weight	• Belt Slippage • Most cumbersome • Requires holding torque to keep antenna stationary
Worm Drive	• No holding torque required • Smoothest operation (most steps per revolution of antenna) • Compact Design	• Difficult to implement • Gears can be expensive

Table 1 – Drive Methods

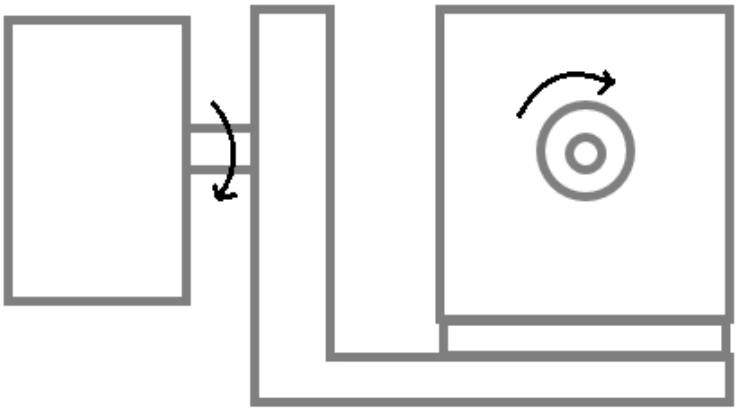


Figure 7 – Direct Drive

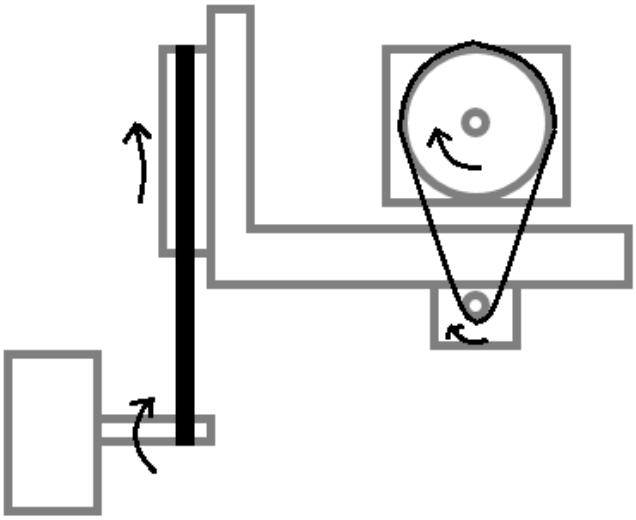


Figure 8 – Belt Drive

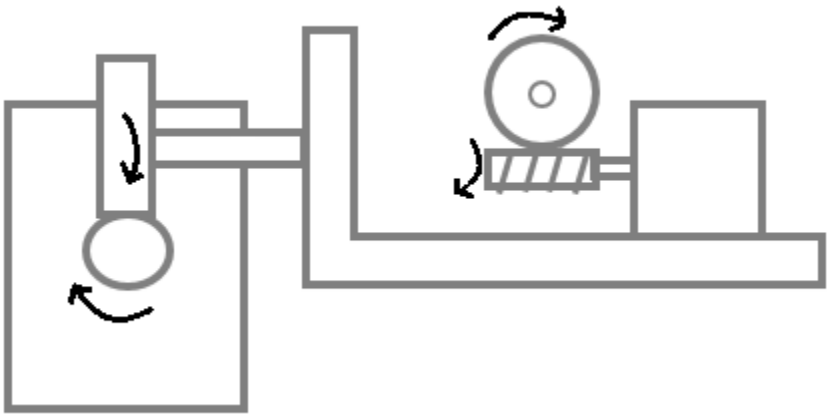


Figure 9 – Worm Drive

Based on this analysis the worm drive method was chosen as the basis of the hardware system. Although the worm system is more difficult to implement than the other systems, the advantages of the worm gear, mainly the advantage of not needing holding torque, outweigh its disadvantages.

5.2 Material Decisions

Material	**Pro**	**Con**
Lexan (polycarbonate)	• RF transparent • Very easy to machine • Durable	• Heavy
Wood	• Inexpensive • Lightweight • Easy to machine • RF transparent	• Can warp • Not durable
Aluminum	• Lightweight • Durable	• Expensive • Difficult to machine • Could interfere with RF

Table 2 - Materials

Polycarbonate was chosen as the base material for the mechanical subsystem because of its non-interference with the RF emitted and received by the antenna and its resistance to warping. Aluminum and wood were eliminated as options because of their respective interference with RF and propensity to warp, both properties deemed unacceptable in the end product.

5.3 Design and Verification

Before we moved onto physically constructing the hardware aspects of the projects, each of the pieces were designed separately in SolidWorks, a computer aided design program, and combined into a single SolidWorks assembly in order to assure that the parts would all fit together properly. It was at this point that we designed mounts for both the device and the antenna that was to attach to it. Standard ¼-20 mounts, commonly used on camera tripods, were chosen in order to provide a wide range of mountable instruments. A camera tripod was also chosen as the mount for our device due to its robustness and availability. The completed model is shown below in Figure 10 and individual SolidWorks part models can be seen in the Appendix B.

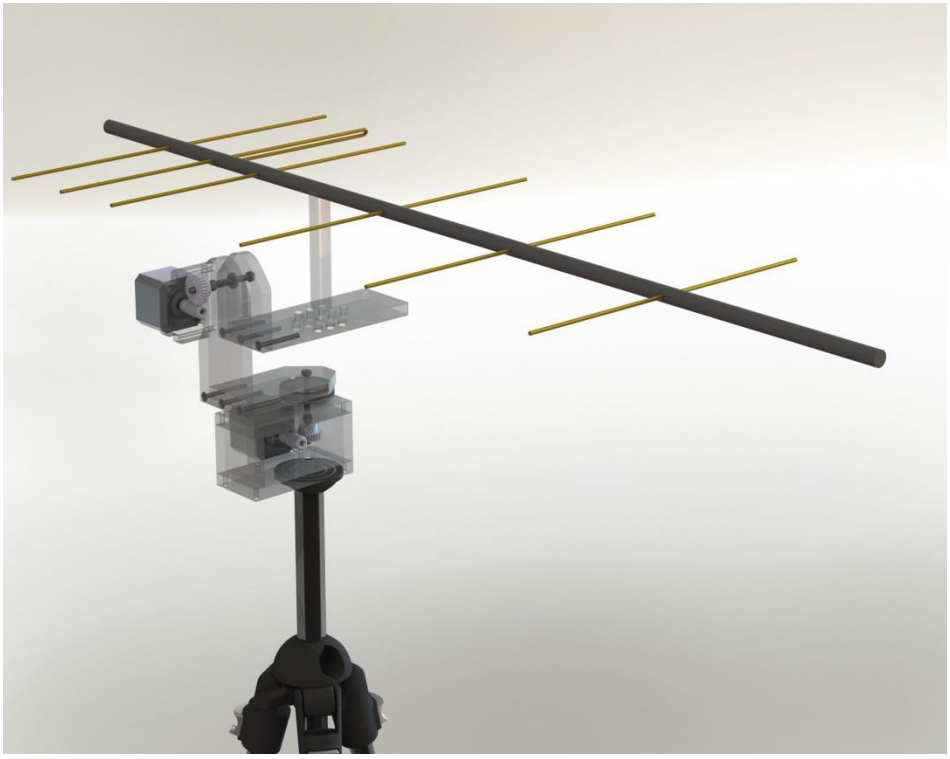


Figure 10 – SolidWorks Completed Model

5.4 Construction

The device was manufactured in the Villanova machine shop using three main tools; a band saw, end mill, and drill press. Using the band saw pieces of the polycarbonate were cut to the sizes shown in the individual parts drawings. Parts were checked for accuracy and "squared off" using the end mill. Once the parts were square, holes were drilled in the locations specified in the part drawings. Holes meant to accept screws were then threaded, and the finished product was assembled by fastening the parts together using machine screws.

6. TESTING AND EVALUATION

After the construction of the project was completed, a series of basic tests were performed in order to verify the design's operation. The first test performed was a simple range of motion and

durability test. We set the device to actuate its full range of motion, both azimuth and elevation, and then to continuously repeat this for four hours. The device performed this task without flaw, but a squeaking noise developed from the worm gears. Because of this, a lubricant was applied to the gears. This lubricant eliminated the squeaking noise.

The device was also tested to verify that the algorithm functioned properly and the device correctly pointed to astronomical objects. A new satellite named "Moon" was added to the set of track-able objects. This satellite was an approximation of the moon's orbit around the earth. The device was then set to track the moons orbit, which could be easily verified visually.

The device was able to survive the endurance testing it was subject to, and the accuracy of its tracked paths was verified by using the moon to visually confirm a proper tracking path. Although an antenna was purchased in order to use the device for satellite communication, the group did not have access to the radios necessary in order to verify that the system was tracking properly for radio communication via satellite. The group is currently in the process of getting radio access that will be used to verify the system's utility in a communications role.

7. PROJECT MANAGEMENT

7.1 Schedule

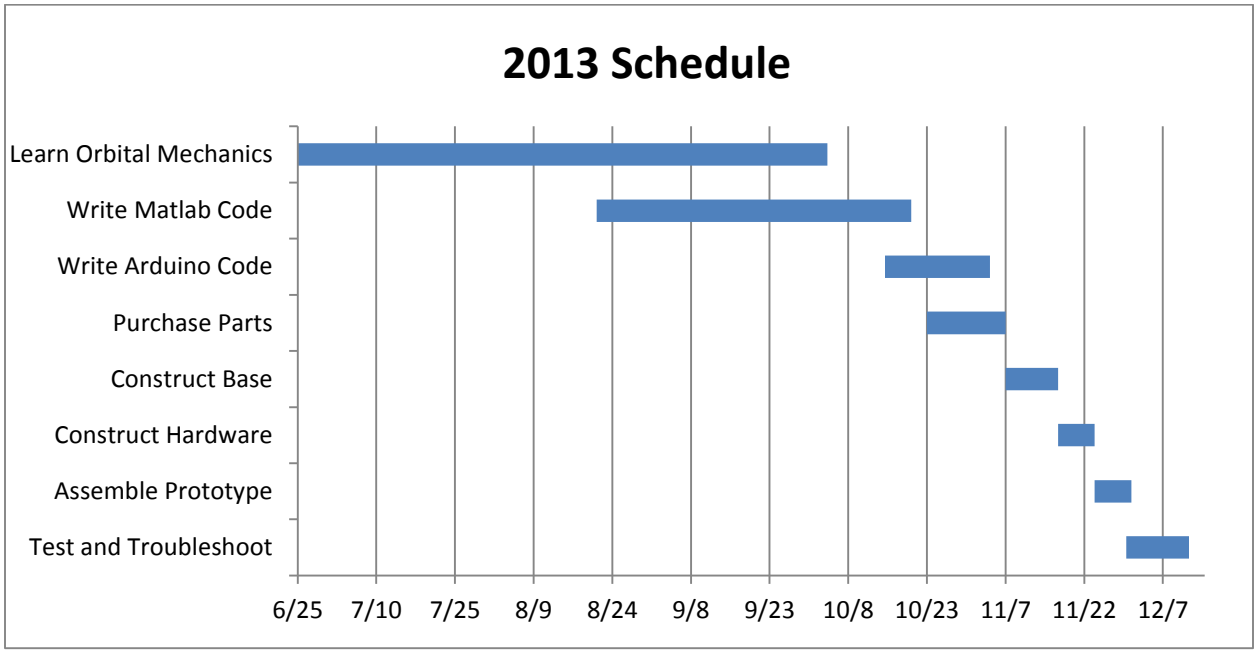


Chart 1 – Schedule

7.2 Personnel

Group Member	Assigned Focus Tasks
John Buglione	-Purchase Parts - Design and Construct Base

	- Construct Hardware -Assemble Prototyping -Testing
Steven Gulotta	-Purchase Parts -Design MATLAB/Arduino Interface -Arduino Programming -Assemble Prototyping -Testing
Jordan Ly	-Purchase Parts -Create Method for Calculating Satellite Positions -MATLAB Programming/Creation of GUI -Assembling Prototyping -Testing

Table 3 – Personnel

7.3 Physical Resources & Budget

Item	Quantity	Cost	Subtotal
6" x 12" Polycarbonate Sheet	4	$6.75	$27.00
25 pack m4 30mm screws	1	$0.77	$0.77
3" square 200lb load turntable	2	$0.53	$1.06
25 pack Low profile 6-32, 3/8" torx screws	1	$8.58	$8.58
25 pack 10-24, 1-3/8" socket head cap screws	1	$5.76	$5.76
25’ speaker cable wire	1	$0.00	$0.00
LSM30DLMTR Chip	1	$29.95	$29.95
Ravelli APLT4 Tripod	1	$22.95	$22.95
440-3 Arrow II Portable Antenna	1	$29.00	$29.00
Tetrix 20 to 1 Worm Gear Pack	2	$15.95	$31.90
Arduino Leonardo	1	$24.95	$24.95
ROB-09238 Stepper Motor	2	$14.95	$29.90
EasyDriver motor driver	2	$14.95	$29.90
			Total

$241.72

Table 4 – Material Costs

Person	Hours	Salary cost ($25/hr)
Jake Buglione	100	$2,500.00
Steven Gulotta	100	$2,500.00
Jordan Ly	100	$2,500.00
Total	300	$7,500.00
Overhead (83%)		$6,225.00
Materials Cost (Table 4)		$241.72
Grand Total		$13,966.72

Table 5 – Total Budget

8. ACHIEVEMENTS

The finished prototype meets most, if not all, of the originally proposed specifications. It is a fully functional, automated system that follows a satellite by predicting its orbital path. It is a much cheaper alternative for satellite communications coming in just under $250 for materials. It is very portable, capable of being transported and operated by a single person. It is also very accurate, within 5 degrees, due to its carefully implemented sensors and feedback loop.

9. CONCLUSIONS & RECOMMENDATIONS

This project successfully designed, built and implemented an accurate and low cost system for automated tracking of satellites. While the team was unable to get to the point where the system was used to communicate with satellites, it was shown that the system was an accurate at estimating the real time location of satellites as well as automatically adjusting the position of the antenna to keep pace with the movement of the satellite.

There are a couple of shortcomings to this project. Firstly, the manufacturer of the compass chip recommends that the chip be recalibrated for every location in which the chip is used. The recalibration of the chip is not difficult, but it involves changing values in the Arduino source code. This would require the user to have the Arduino IDE and feel comfortable in editing code and could be a shortcoming for the average user.

Another potential issue with the system is that it is currently dependent on an internet connection to get up to date Keplerian elements from AMSAT. If internet is not available, the Keplerian elements can be read from a text file, but depending upon when the text file is generated, they might not be the most up-to-date or accurate.

REFERENCES

[1] St. D. Ilcev, “Low Earth Orbits,” in the *20th Microwave and Telecommunication Technology Conference,* 2010, pp. 406-408

[2] M. Halavi, “Satellite Tracking Method and Apparatus Thereof.” U.S. Patent 8,314,735, issued November 12, 2011

[3] "Keplerian Elements Tutorial." Keplerian Elements Tutorial. N.p., n.d. Web.

[4] "Bildr » Use The EasyDriver Stepper Motor Driver + Arduino." *Bildr RSS*. N.p., n.d. Web. 27 Mar. 2014.

[5] STMicroelectronics “Ultra-compact high-performance Compass module: 3D accelerometer and 3D magnetometer” LSM303DLHC Datasheet, November 2013.

APPENDIX A: CODE USED

A.1 MATLAB GUI

```
%%//////////////////////////////////////////////////////////////////////////
% Authors: Steven Gulotta, Jordan Ly, John Buglione
%
%Program creates GUI as well as calls methods for calculating satellite
%positions and for sending data to Arduino to move motors.
%
%
%//////////////////////////////////////////////////////////////////////////
%% GUI Code
function varargout = UI2(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @UI2_OpeningFcn, ...
                   'gui_OutputFcn',  @UI2_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

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

% --- Executes just before UI2 is made visible.
function UI2_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;
guidata(hObject, handles);
```

```
% --- Outputs from this function are returned to the command line.
function varargout = UI2_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;

% --- Executes on button press in pushbutton1. This is the stop button
function pushbutton1_Callback(hObject, eventdata, handles)
global status
status = -1;
  delete(instrfind({'Port'},{'COM9'}))

% --- Executes on button press in pushbutton2. This is the Start Button
function pushbutton2_Callback(hObject, eventdata, handles)
%% Set Status for Button
global status
status = 1;
cd('C:\Users\wildcat\Documents\MATLAB\SeniorDesign')
%% Initiates Arduino
% Opens serial at specified baud
display (['Establishing Connection. Please Wait']);
s1 = serial('COM9');    % define serial port
s1.BaudRate=9600;       % define baud rate
fopen(s1);
w=fscanf(s1,'%s');      % must define the input %s

% Following loop does handshake to verify Arduino Connection
if (w=='A')
    fprintf(['Verifying Connection\n']);
    fprintf(s1,'%s\n','A');
    fprintf(['Connection Verified\n']);
end
%% User Prompted Information is read from GUI
global index
global lambda
global phi
index = get(handles.popupmenu1,'Value')-1
lambda = str2num(get(handles.latitude,'string'))
phi = str2num(get(handles.longitude,'string'))
%% Motor Movement
global El
global Az
global Azimuth_Displacement;
global Elevation_Displacement;
while status > 0
Satellite_Position_Calculator(index, lambda, phi);
set(handles.Azimuth_Print, 'string', Az)
set(handles.Elevation_Print, 'string', El)
Move_Motors_With_Arduino(s1,Az,El);
end

% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)

function popupmenu1_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function altitude_Callback(hObject, eventdata, handles)

function altitude_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function latitude_Callback(hObject, eventdata, handles)

function latitude_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function longitude_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function longitude_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Elevation_Print_Callback(hObject, eventdata, handles)

function Elevation_Print_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Azimuth_Print_Callback(hObject, eventdata, handles)

function Azimuth_Print_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

A.2 Python Calculation

```
#////////////////////////////////////////////////////////////////////////////
```

```
#
# Authors: Steven Gulotta, John Buglione, Jordan Ly
#
# Calculate azimuth and elevation of satellite given current latitude and
# longitude location
#
#////////////////////////////////////////////////////////////////////////
from urllib2 import urlopen
import ephem
import datetime

#Create observer object
me = ephem.Observer()
lat = raw_input("Please enter your latitude: ")
#Set observer latitude to user input
me.lat =lat
lon = raw_input("Please enter your longitude: ")
#Set observer longitude to user input
me.lon=lon
#Set observer date/time in UTC
me.date = datetime.datetime.utcnow()

#Load satellite information from online
BASE_URL = "http://www.amsat.org/amsat/ftp/keps/current/nasabare.txt"
html = urlopen(BASE_URL)

search = raw_input('What Sattelite: ')

for line in html:
      #Parse through data
      if line == (search + '\n'):
            l1 = line
            l2 = html.readline()
            l3 = html.readline()

else:
      sat = ephem.readtle(l1,l2,l3)
      #Compute satellite location with respect to observer
      sat.compute(me)
      print l1, sat.alt,'(altitude)\n', sat.az, '(azimuth)'
```

A.3 MATLAB Satellite Calculation

```
%%//////////////////////////////////////////////////////////////////////
%
% Authors: Steven Gulotta, John Buglione, Jordan Ly
%
% Loads Python program to calculate azimuth and elevation
%
%//////////////////////////////////////////////////////////////////////

clc
clear

% Load Python program
```

```
input = sprintf('"sat.exe" %.5f %.5f %s',lat,long,name);
[x y] = system(input);
calc = str2num(y);
az = calc(2)
el = calc(1)
```

A.4 MATLAB Motor Movement

```
%%/////////////////////////////////////////////////////////////////////////
%
% Authors: Steven Gulotta, John Buglione, Jordan Ly
%
%Code will take values calculated by satellite tracking function and send
%to Arduino over serial.
%
%/////////////////////////////////////////////////////////////////////////
%%
function move=Move_Motors_With_Arduino(s1,Az,El)
%% Send Azimuth value to Arduino and move motor
% MATLAB first converts values to string, handshakes with Arduino get
% permission to send value and then places ("prints") it on the serial
% Once string is placed onto serial, MATLAB checks to see it is there and
% prints a confirmation for the user.
Az_str = int2str(Az);
Azimuth_Status = 1
fprintf(s1,'1');
  while (Azimuth_Status == 1)
        az_hs = fscanf(s1,'%s');
        if (az_hs == '1')
            fprintf(s1,'%s\n',Az_str);
            fscanf(s1,'%s');
            fprintf('AZ Measurement sent to arduino');
            Azimuth_Status = 0;
        end
  end

%% Wait for return of control from Arduino
% Handshake before moving to next step
Control_Status = 1
while (Control_Status == '1')
    CS = fscanf(s1,'%s');
    if (CS== '9')
        Control_status = 0;
    end
end

%% Send Elevation value to Arduino and move motor
% MATLAB first converts values to string, handshakes with Arduino get
% permission to send value and then places ("prints") it on the serial
% Once string is placed onto serial, MATLAB checks to see it is there and
% prints a confirmation for the user.
El_str = int2str(El);
Elevation_Status = 1
fprintf(s1,'2');
while (Elevation_Status == 1)
```

```
    el_hs = fscanf(s1,'%s');
        if (el_hs == '2')
            fprintf(s1,'%s\n',El_str);
            fscanf(s1,'%s');
            fprintf('El Measurement sent to arduino');
            Elevation_Status = 0;
        end
end
%% Wait for return of control from Arduino
% Handshake before moving to next step
Control_Status = 1
while (Control_Status == '1')
    CS = fscanf(s1,'%s');
    if (CS== '9')
        Control_status = 0;
    end
end
```

A.5 Arduino Function

```
////////////////////////////////////////////////////////////////////
//
// Authors: Steven Gulotta, John Buglione, Jordan Ly
//
// Program to be loaded onto Arduino
//
////////////////////////////////////////////////////////////////////
#include <Wire.h>
#include <LSM303.h>

LSM303 compass;
int AZ_DIR_PIN = 7;
int AZ_STEP_PIN = 6;
int EL_DIR_PIN = 4;
int EL_STEP_PIN = 5;

int az_orientation;
int el_orientation;


void setup()
{
// initialize motors pins

  pinMode(AZ_STEP_PIN, OUTPUT);
  pinMode(AZ_DIR_PIN, OUTPUT);
  pinMode(EL_STEP_PIN, OUTPUT);
  pinMode(EL_DIR_PIN, OUTPUT);

// start serial port at 9600 bps:
   int ledPin=13;
   Serial.begin(9600);
   digitalWrite(ledPin,HIGH);
   establishContact();  // send a byte to establish contact until receiver
responds
```

```
   digitalWrite(ledPin,LOW);

 initalize compass
  Wire.begin();
  compass.init();
  compass.enableDefault();
  // Calibration values. Use the Calibrate example program to get the values
for
  // your compass.
  compass.m_min.x = -520; compass.m_min.y = -570; compass.m_min.z = -770;
  compass.m_max.x = +540; compass.m_max.y = +500; compass.m_max.z = 180;


}


void loop() {

//command

int command= ReadIn();
switch(command){
  case 1:
    {
    Serial.println('1');
    int az = ReadIn();
    AzimuthMove(az);
    Serial.println('9');
    break;
    }
    case 2:
    {
    Serial.println('2');
    int el = ReadIn();
    ElevationMove(el);
    Serial.println('9');
    break;
    }
}
}// End Loop

void establishContact() {
     while (Serial.available() <= 0) {
       Serial.println('A');   // send a capital A
       delay(300);
     }
 }

int ReadIn(){
   char buffer[7] ; // Receive up to 7 bytes
 while (!Serial.available()); // Wait for characters
 Serial.readBytesUntil('n', buffer, 7);
return atoi(buffer);
;
}
```

```
void rotateDeg(float deg, float speed, int DIR_PIN, int STEP_PIN ){
  //rotate a specific number of degrees (negitive for reverse movement)
  //speed is any number from .01 -> 1 with 1 being fastest - Slower is
stronger
  int dir = (deg < 0)? HIGH:LOW;
  digitalWrite(DIR_PIN,dir);

  int steps = abs(deg)*(20/0.225);
  float usDelay = (1/speed) * 70;

  for(int i=0; i < steps; i++){
    digitalWrite(STEP_PIN, HIGH);
    delayMicroseconds(usDelay);

    digitalWrite(STEP_PIN, LOW);
    delayMicroseconds(usDelay);
  }
}
void AzimuthMove(int az){
int az_orientation;
  for(int i=0;i<10;i++){
    compass.read();
    az_orientation= az_orientation+ compass.heading((LSM303::vector){0,-
1,0});
  }
 az_orientation= az_orientation/10;
 rotateDeg(az_displacement, 0.25, AZ_DIR_PIN, AZ_STEP_PIN );
}

void ElevationMove(int el){
int el_orientation;
for(int i=0;i<10;i++){
    compass.read();
    el_orientation = el_orientation + compass.y_tilt((LSM303::vector){0,-
1,0});
  }
  el_orientation= el_orientation/10;
 rotateDeg(el_displacement, 0.25, EL_DIR_PIN, EL_STEP_PIN );
}
```

A.6 LSM303DLMTR Code

```
#include <LSM303.h>
#include <Wire.h>
#include <math.h>

// Defines /////////////////////////////////////////////////////////////

// The Arduino two-wire interface uses a 7-bit number for the address,
// and sets the last bit correctly based on reads and writes
#define MAG_ADDRESS            (0x3C >> 1)
#define ACC_ADDRESS_SA0_A_LOW  (0x30 >> 1)
#define ACC_ADDRESS_SA0_A_HIGH (0x32 >> 1)
```

```
// Constructors
////////////////////////////////////////////////////////////////

LSM303::LSM303(void)
{
  // These are just some values for a particular unit; it is recommended that
  // a calibration be done for your particular unit.
  m_max.x = +540; m_max.y = +500; m_max.z = 180;
  m_min.x = -520; m_min.y = -570; m_min.z = -770;

  _device = LSM303_DEVICE_AUTO;
  acc_address = ACC_ADDRESS_SA0_A_LOW;

  io_timeout = 0;  // 0 = no timeout
  did_timeout = false;
}

// Public Methods
////////////////////////////////////////////////////////////////

bool LSM303::timeoutOccurred()
{
  return did_timeout;
}

void LSM303::setTimeout(unsigned int timeout)
{
  io_timeout = timeout;
}

unsigned int LSM303::getTimeout()
{
  return io_timeout;
}

void LSM303::init(byte device, byte sa0_a)
{
  _device = device;
  switch (_device)
  {
    case LSM303DLH_DEVICE:
    case LSM303DLM_DEVICE:
      if (sa0_a == LSM303_SA0_A_LOW)
        acc_address = ACC_ADDRESS_SA0_A_LOW;
      else if (sa0_a == LSM303_SA0_A_HIGH)
        acc_address = ACC_ADDRESS_SA0_A_HIGH;
      else
        acc_address = (detectSA0_A() == LSM303_SA0_A_HIGH) ?
ACC_ADDRESS_SA0_A_HIGH : ACC_ADDRESS_SA0_A_LOW;
      break;

    case LSM303DLHC_DEVICE:
      acc_address = ACC_ADDRESS_SA0_A_HIGH;
      break;
```

```
    default:
      // try to auto-detect device
      if (detectSA0_A() == LSM303_SA0_A_HIGH)
      {
        // if device responds on 0011001b (SA0_A is high), assume DLHC
        acc_address = ACC_ADDRESS_SA0_A_HIGH;
        _device = LSM303DLHC_DEVICE;
      }
      else
      {
        // otherwise, assume DLH or DLM (pulled low by default on Pololu
boards); query magnetometer WHO_AM_I to differentiate these two
        acc_address = ACC_ADDRESS_SA0_A_LOW;
        _device = (readMagReg(LSM303_WHO_AM_I_M) == 0x3C) ? LSM303DLM_DEVICE
: LSM303DLH_DEVICE;
      }
  }
}

// Turns on the LSM303's accelerometer and magnetometers and places them in
normal
// mode.
void LSM303::enableDefault(void)
{
  // Enable Accelerometer
  // 0x27 = 0b00100111
  // Normal power mode, all axes enabled
  writeAccReg(LSM303_CTRL_REG1_A, 0x27);

  if (_device == LSM303DLHC_DEVICE)
    writeAccReg(LSM303_CTRL_REG4_A, 0x08); // DLHC: enable high resolution
mode

  // Enable Magnetometer
  // 0x00 = 0b00000000
  // Continuous conversion mode
  writeMagReg(LSM303_MR_REG_M, 0x00);
}

// Writes an accelerometer register
void LSM303::writeAccReg(byte reg, byte value)
{
  Wire.beginTransmission(acc_address);
  Wire.write(reg);
  Wire.write(value);
  last_status = Wire.endTransmission();
}

// Reads an accelerometer register
byte LSM303::readAccReg(byte reg)
{
  byte value;

  Wire.beginTransmission(acc_address);
```

```
  Wire.write(reg);
  last_status = Wire.endTransmission();
  Wire.requestFrom(acc_address, (byte)1);
  value = Wire.read();
  Wire.endTransmission();

  return value;
}

// Writes a magnetometer register
void LSM303::writeMagReg(byte reg, byte value)
{
  Wire.beginTransmission(MAG_ADDRESS);
  Wire.write(reg);
  Wire.write(value);
  last_status = Wire.endTransmission();
}

// Reads a magnetometer register
byte LSM303::readMagReg(int reg)
{
  byte value;

  // if dummy register address (magnetometer Y/Z), use device type to
determine actual address
  if (reg < 0)
  {
    switch (reg)
    {
      case LSM303_OUT_Y_H_M:
        reg = (_device == LSM303DLH_DEVICE) ? LSM303DLH_OUT_Y_H_M :
LSM303DLM_OUT_Y_H_M;
        break;
      case LSM303_OUT_Y_L_M:
        reg = (_device == LSM303DLH_DEVICE) ? LSM303DLH_OUT_Y_L_M :
LSM303DLM_OUT_Y_L_M;
        break;
      case LSM303_OUT_Z_H_M:
        reg = (_device == LSM303DLH_DEVICE) ? LSM303DLH_OUT_Z_H_M :
LSM303DLM_OUT_Z_H_M;
        break;
      case LSM303_OUT_Z_L_M:
        reg = (_device == LSM303DLH_DEVICE) ? LSM303DLH_OUT_Z_L_M :
LSM303DLM_OUT_Z_L_M;
        break;
    }
  }

  Wire.beginTransmission(MAG_ADDRESS);
  Wire.write(reg);
  last_status = Wire.endTransmission();
  Wire.requestFrom(MAG_ADDRESS, 1);
  value = Wire.read();
  Wire.endTransmission();
```

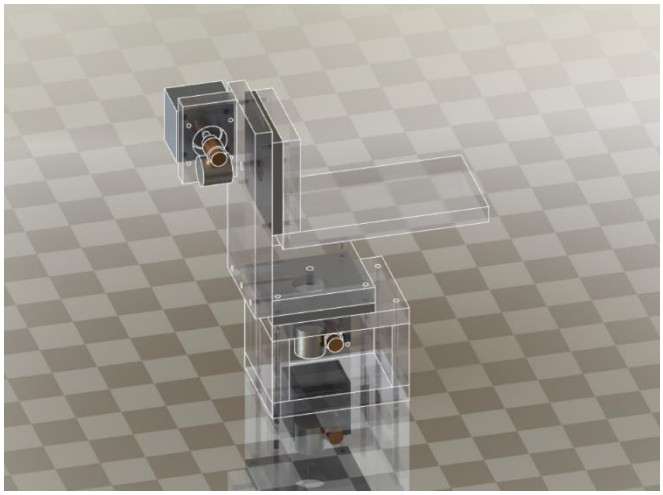
```
  return value;
}

void LSM303::setMagGain(magGain value)
{
  Wire.beginTransmission(MAG_ADDRESS);
  Wire.write(LSM303_CRB_REG_M);
  Wire.write((byte) value);
  Wire.endTransmission();
}

// Reads the 3 accelerometer channels and stores them in vector a
void LSM303::readAcc(void)
{
  Wire.beginTransmission(acc_address);
  // assert the MSB of the address to get the accelerometer
  // to do slave-transmit subaddress updating.
  Wire.write(LSM303_OUT_X_L_A | (1 << 7));
  last_status = Wire.endTransmission();
  Wire.requestFrom(acc_address, (byte)6);

  unsigned int millis_start = millis();
  did_timeout = false;
  while (Wire.available() < 6) {
    if (io_timeout > 0 && ((unsigned int)millis() - millis_start) >
io_timeout) {
      did_timeout = true;
      return;
    }
  }

  byte xla = Wire.read();
  byte xha = Wire.read();
  byte yla = Wire.read();
  byte yha = Wire.read();
  byte zla = Wire.read();
  byte zha = Wire.read();

  // combine high and low bytes, then shift right to discard lowest 4 bits
(which are meaningless)
  // GCC performs an arithmetic right shift for signed negative numbers, but
this code will not work
  // if you port it to a compiler that does a logical right shift instead.
  a.x = ((int16_t)(xha << 8 | xla)) >> 4;
  a.y = ((int16_t)(yha << 8 | yla)) >> 4;
  a.z = ((int16_t)(zha << 8 | zla)) >> 4;
}

// Reads the 3 magnetometer channels and stores them in vector m
void LSM303::readMag(void)
{
  Wire.beginTransmission(MAG_ADDRESS);
  Wire.write(LSM303_OUT_X_H_M);
  last_status = Wire.endTransmission();
  Wire.requestFrom(MAG_ADDRESS, 6);
```

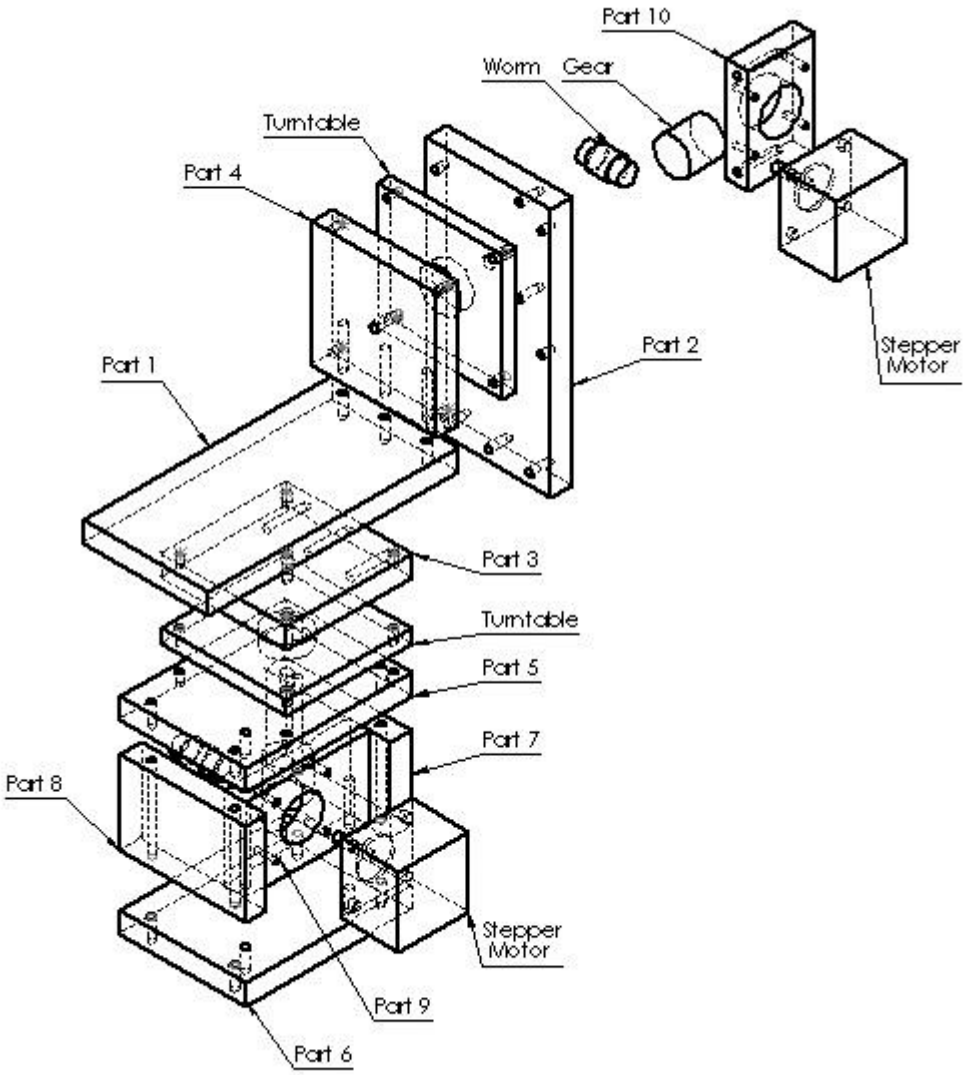
```
  unsigned int millis_start = millis();
  did_timeout = false;
  while (Wire.available() < 6) {
    if (io_timeout > 0 && ((unsigned int)millis() - millis_start) >
io_timeout) {
      did_timeout = true;
      return;
    }
  }

  byte xhm = Wire.read();
  byte xlm = Wire.read();

  byte yhm, ylm, zhm, zlm;

  if (_device == LSM303DLH_DEVICE)
  {
    // DLH: register address for Y comes before Z
    yhm = Wire.read();
    ylm = Wire.read();
    zhm = Wire.read();
    zlm = Wire.read();
  }
  else
  {
    // DLM, DLHC: register address for Z comes before Y
    zhm = Wire.read();
    zlm = Wire.read();
    yhm = Wire.read();
    ylm = Wire.read();

  }

  // combine high and low bytes
  m.x = (int16_t)(xhm << 8 | xlm);
  m.y = (int16_t)(yhm << 8 | ylm);
  m.z = (int16_t)(zhm << 8 | zlm);
}

// Reads all 6 channels of the LSM303 and stores them in the object variables
void LSM303::read(void)
{
  readAcc();
  readMag();
}

// Returns the number of degrees from the -Y axis that it
// is pointing.
int LSM303::heading(void)
{
  return heading((vector){0,-1,0});
}

// Returns the angular difference in the horizontal plane between the
```

```
// From vector and North, in degrees.
//
// Description of heading algorithm:
// Shift and scale the magnetic reading based on calibration data to
// to find the North vector. Use the acceleration readings to
// determine the Up vector (gravity is measured as an upward
// acceleration). The cross product of North and Up vectors is East.
// The vectors East and North form a basis for the horizontal plane.
// The From vector is projected into the horizontal plane and the
// angle between the projected vector and north is returned.
int LSM303::heading(vector from)
{
    // shift and scale
    m.x = (m.x - m_min.x) / (m_max.x - m_min.x) * 2 - 1.0;
    m.y = (m.y - m_min.y) / (m_max.y - m_min.y) * 2 - 1.0;
    m.z = (m.z - m_min.z) / (m_max.z - m_min.z) * 2 - 1.0;

    vector temp_a = a;
    // normalize
    vector_normalize(&temp_a);
    //vector_normalize(&m);

    // compute E and N
    vector E;
    vector N;
    vector_cross(&m, &temp_a, &E);
    vector_normalize(&E);
    vector_cross(&temp_a, &E, &N);

    // compute heading
    int heading = round(atan2(vector_dot(&E, &from), vector_dot(&N, &from)) *
180 / M_PI);
    if (heading < 0) heading += 360;
  return heading;
}
int LSM303::y_tilt(void)
{
  return y_tilt((vector){0,-1,0});
}
int LSM303::y_tilt(vector from)
{
    //compute tilt in degrees
    int y_tilt = round(atan2(a.z,a.x)*180 / M_PI);
    if (y_tilt < 0) y_tilt += 360;
  return y_tilt;
}



void LSM303::vector_cross(const vector *a,const vector *b, vector *out)
{
  out->x = a->y*b->z - a->z*b->y;
  out->y = a->z*b->x - a->x*b->z;
  out->z = a->x*b->y - a->y*b->x;
}
```

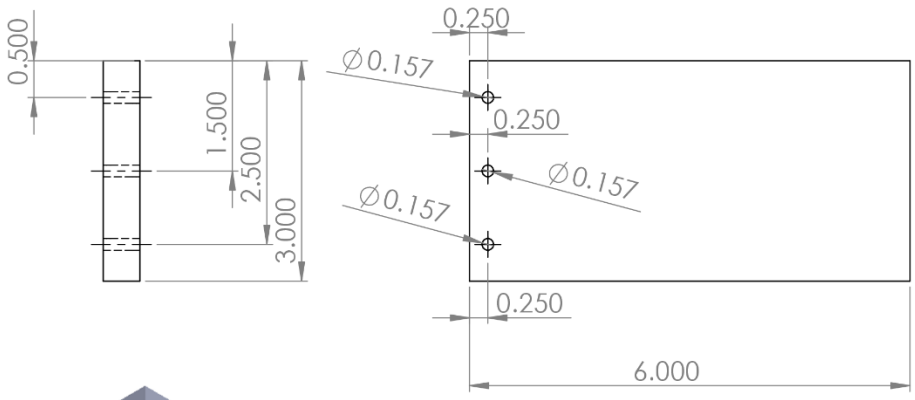
```
float LSM303::vector_dot(const vector *a,const vector *b)
{
  return a->x*b->x+a->y*b->y+a->z*b->z;
}

void LSM303::vector_normalize(vector *a)
{
  float mag = sqrt(vector_dot(a,a));
  a->x /= mag;
  a->y /= mag;
  a->z /= mag;
}

// Private Methods
/////////////////////////////////////////////////////////////

byte LSM303::detectSA0_A(void)
{
  Wire.beginTransmission(ACC_ADDRESS_SA0_A_LOW);
  Wire.write(LSM303_CTRL_REG1_A);
  last_status = Wire.endTransmission();
  Wire.requestFrom(ACC_ADDRESS_SA0_A_LOW, 1);
  if (Wire.available())
  {
    Wire.read();
    return LSM303_SA0_A_LOW;
  }
  else
    return LSM303_SA0_A_HIGH;
}
```

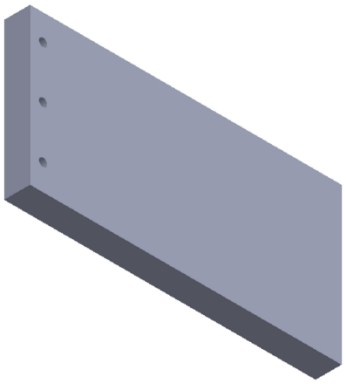
APPENDIX B: SOLIDWORKS MODELS



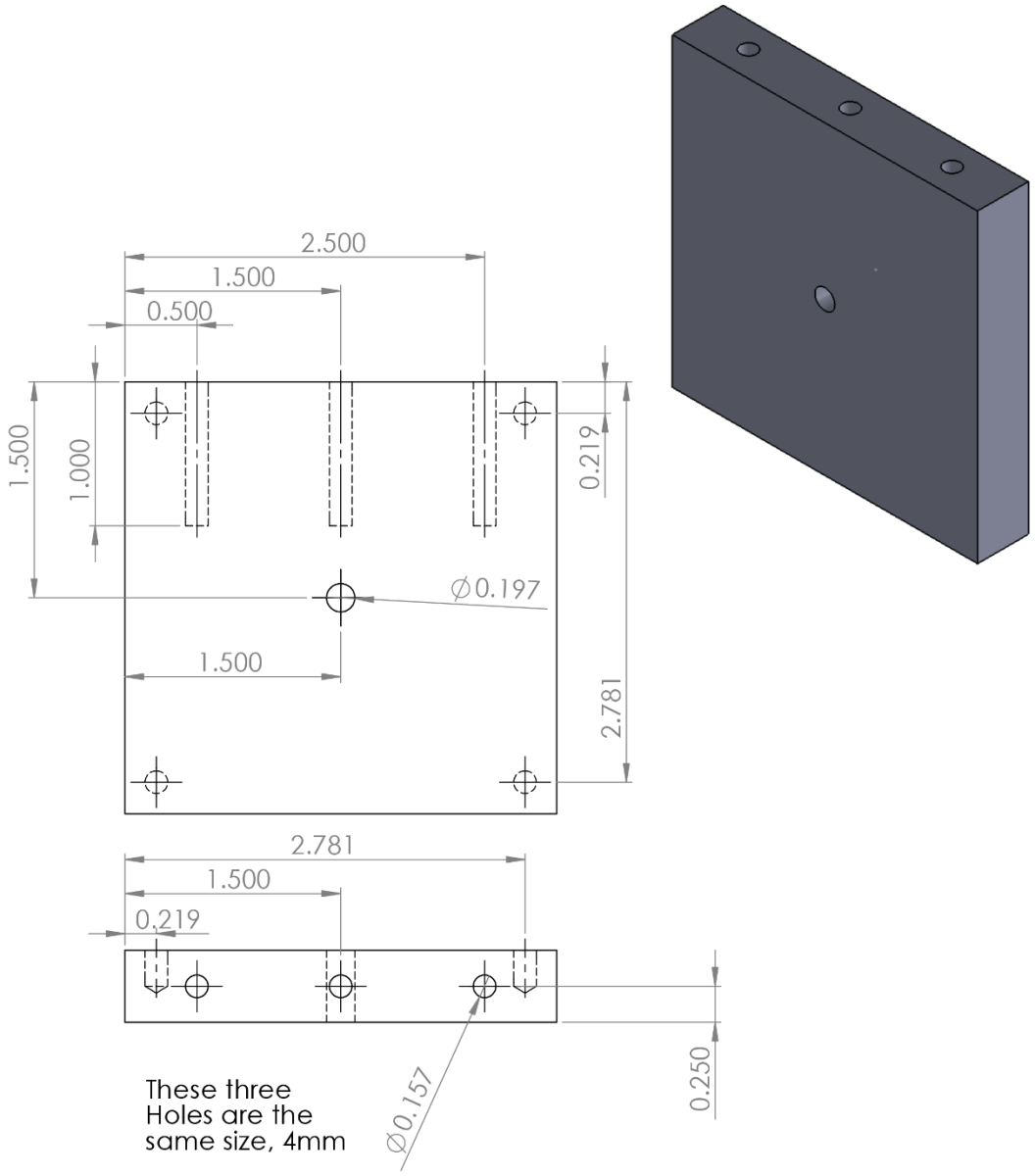

Part 10
Worm
Gear
Turntable
Part 4
Part 2
Stepper Motor
Part 1
Part 3
Turntable
Part 5
Part 7
Part 8
Stepper Motor
Part 9
Part 6

0.157 is metric 4mm

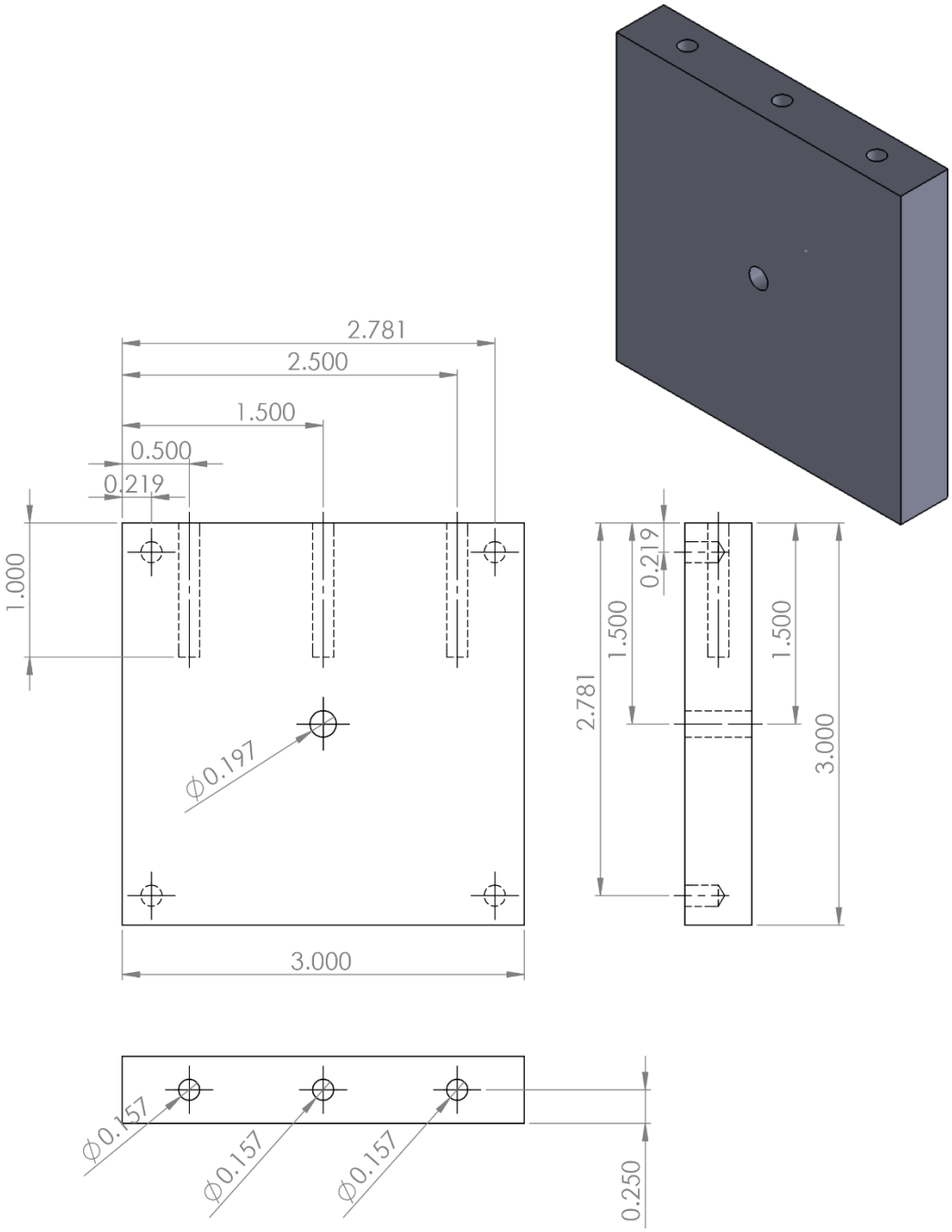




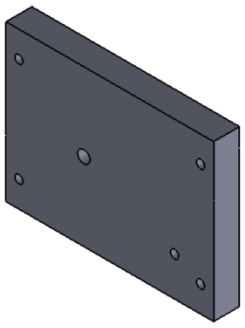
Part 1

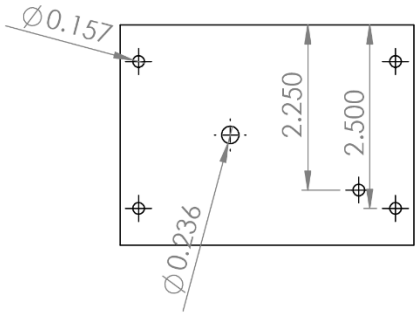

2.500
1.500
0.500
1.500
1.000
0.219
∅0.197
1.500
2.781
2.781
1.500
0.219
0.250
∅0.157
These three
Holes are the
same size, 4mm

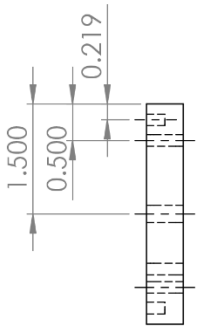
Part 3

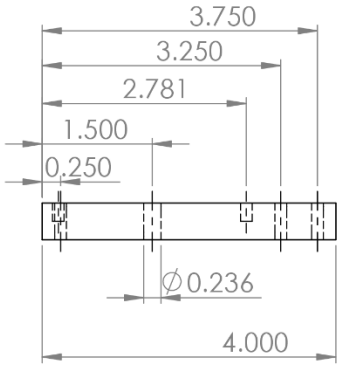

2.781
2.500
1.500
0.500
0.219
1.000
Ø0.197
3.000
0.219
1.500
1.500
2.781
3.000
Ø0.157
Ø0.157
Ø0.157
0.250

Part 4

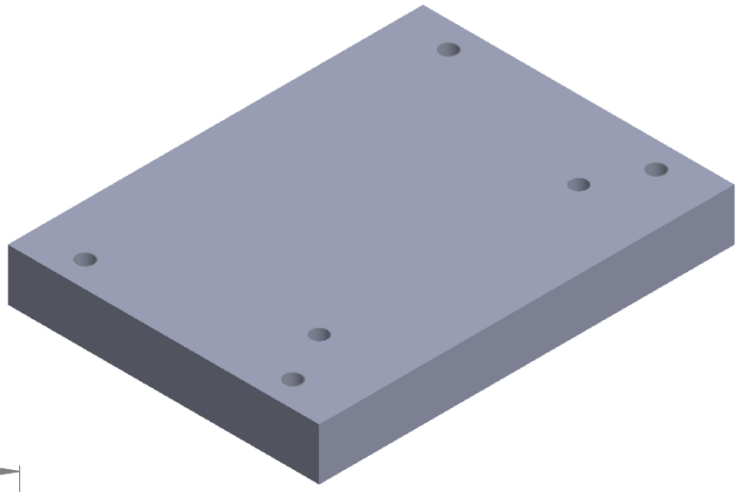


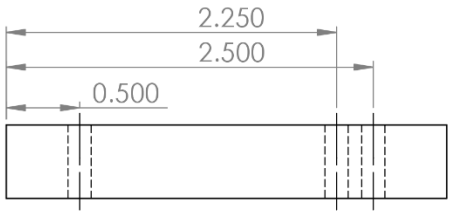

∅0.157
2.250
2.500
∅0.236

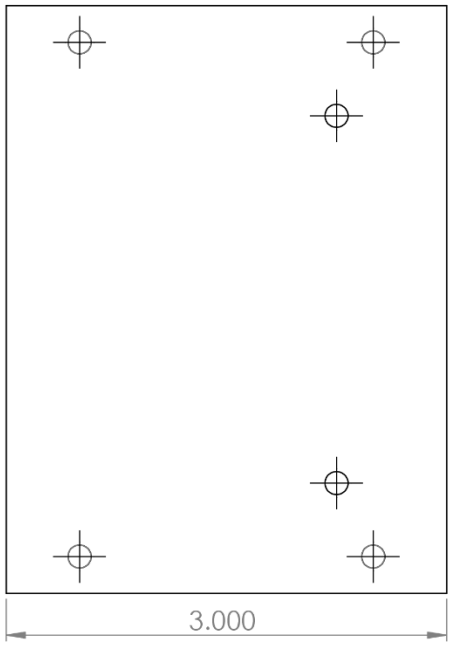

0.219
1.500
0.500

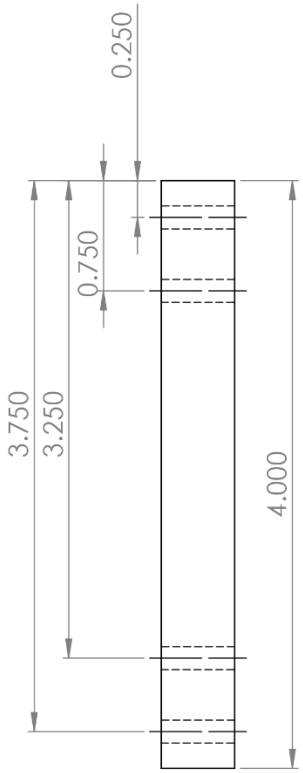

3.750
3.250
2.781
1.500
0.250
∅0.236
4.000

Part 5

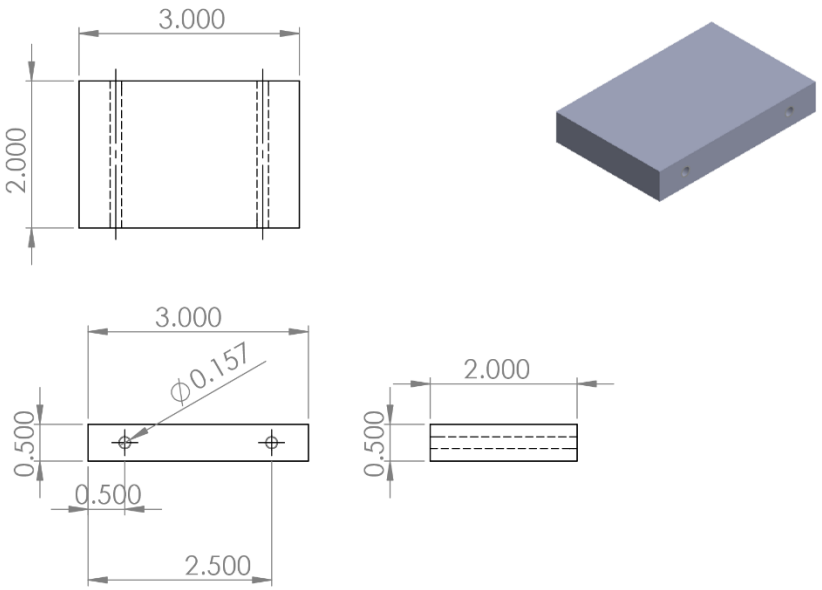




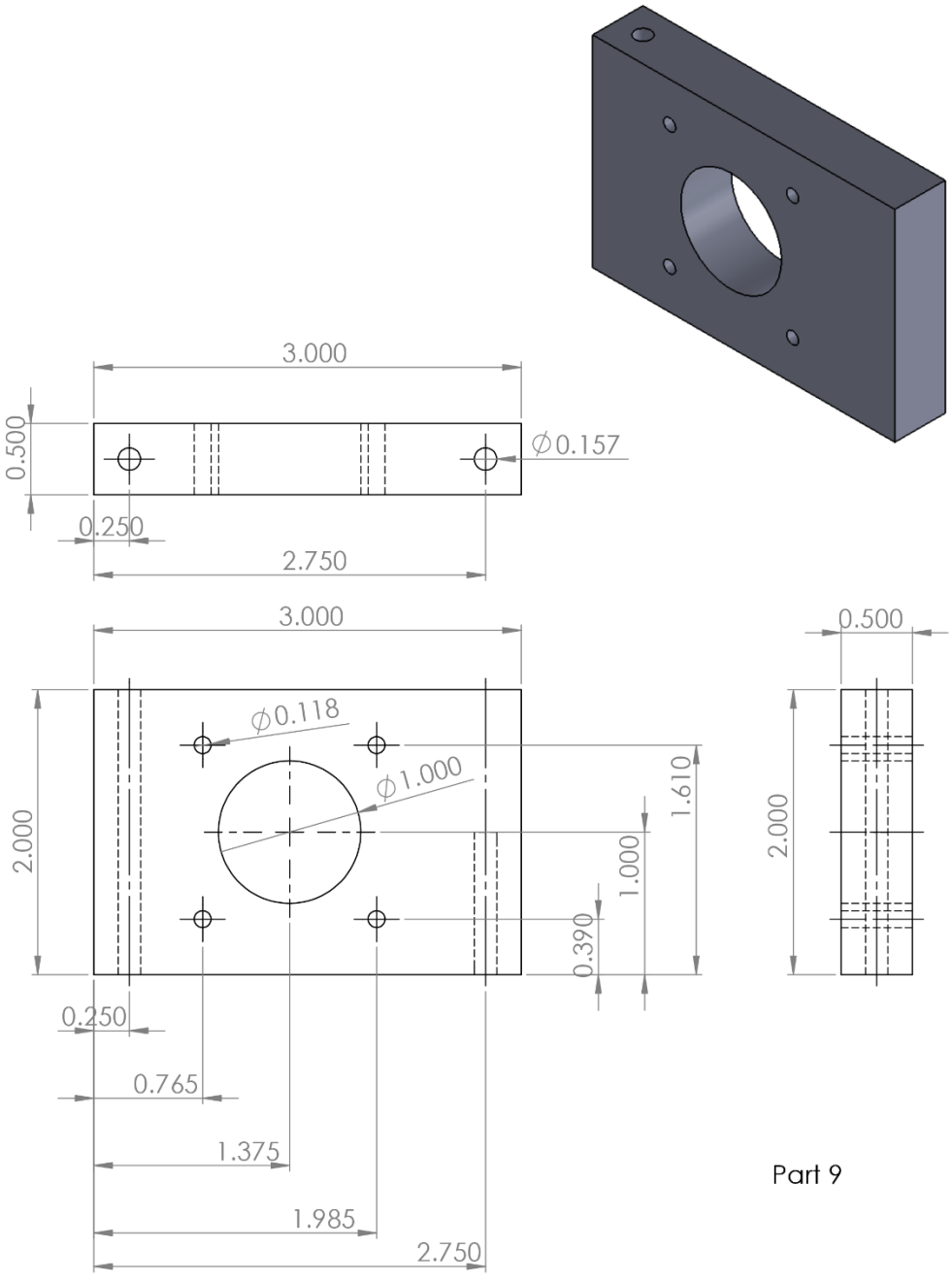




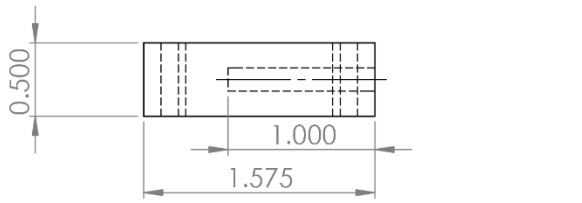
Part 6

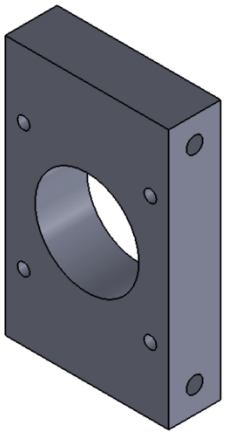


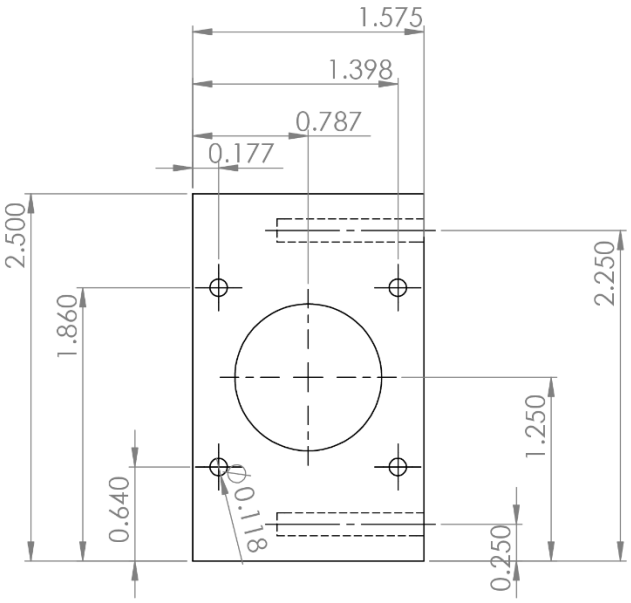
Part 7 and 8

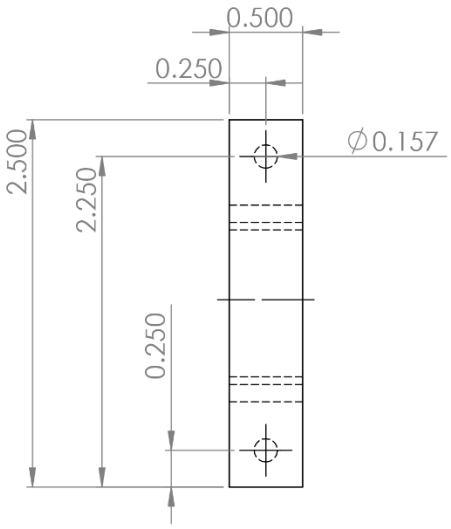

3.000
0.500
∅0.157
0.250
2.750
3.000
0.500
∅0.118
∅1.000
2.000
1.610
1.000
0.390
2.000
0.250
0.765
1.375
1.985
2.750

Part 9





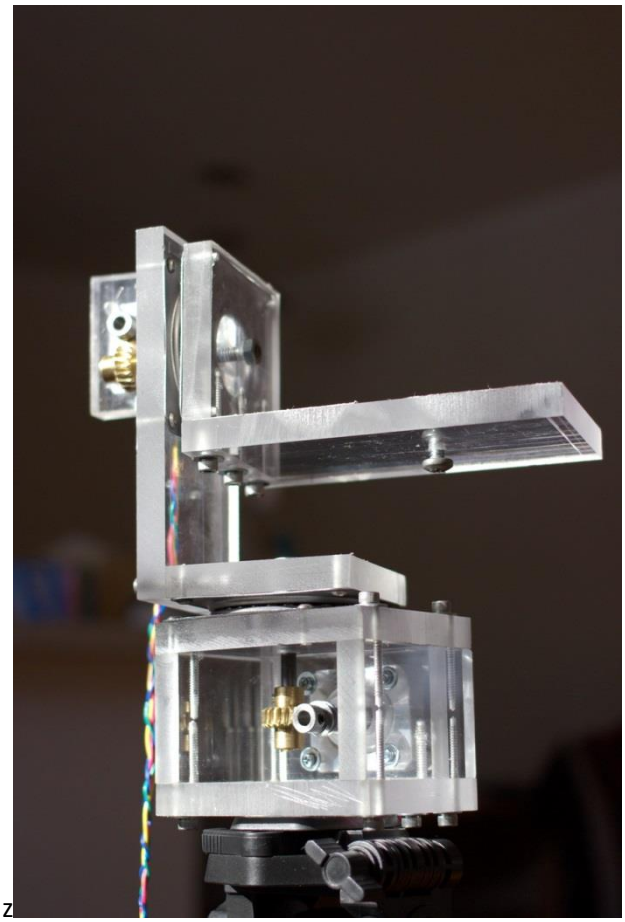


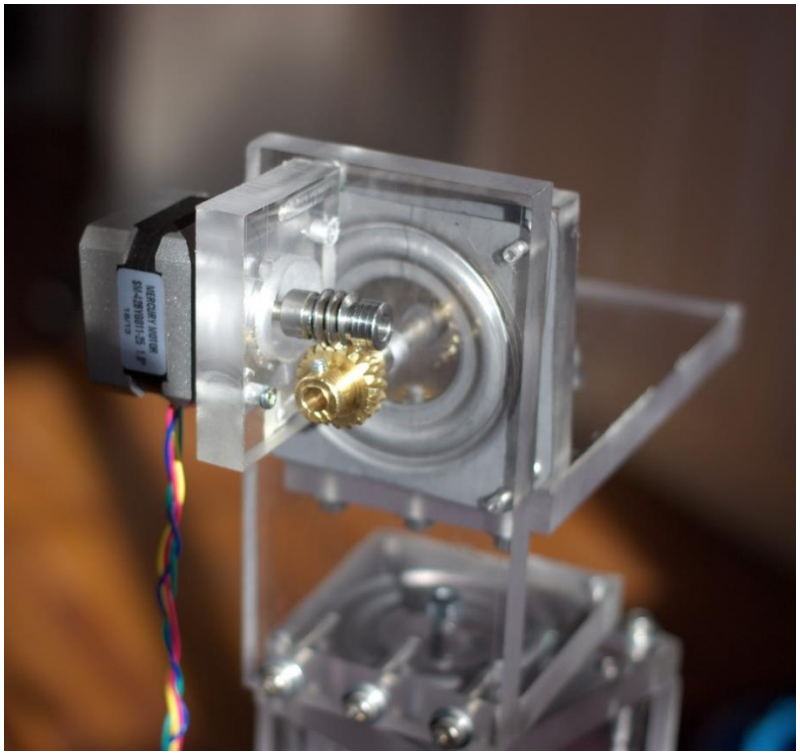


The four small holes are the same distance from the center of the large hole and the same size

Part 10

APPENDIX C: PHOTOGRAPHS

z 



APPENDIX D: SPECIFICATIONS

The system proposed in this paper should at a bare minimum be able to accurately calculate a satellite's position in real time, relate this position to the user's position, point a directional antenna in the direction of said satellite, and track it as it moves across the sky. Should the proposed system successfully accomplish these goals, the next step would be to send and receive test signals from a satellite.

Functional Specification

The proposed system should be portable enough to be carried by a single person and intuitive enough to be used by a person with only a basic knowledge of wireless communications. The system should also be reliable enough for continuous use without malfunction. In order to meet the prior portability requirements, the system must also be low power and must be able at the very least to operate off a small car battery, but ideally, the system would be powered via a USB connection.

Performance Specification

- 1-2 degree pointing accuracy
- Completes a full sweep of the horizon in less than 8 minutes
- Less than 60 lbs. total weight
- Minimum 100 hours of continuous use without downtime (assuming unlimited power)
- Will fit within a 5'x2'x1' case when disassembled
- at least 2 hours of battery life (excluding computer)
- uses less than 125 watts of power (excluding computer)
- Should be able to operate for at least one hour in moderate rain (assuming the computer is shielded from the elements)