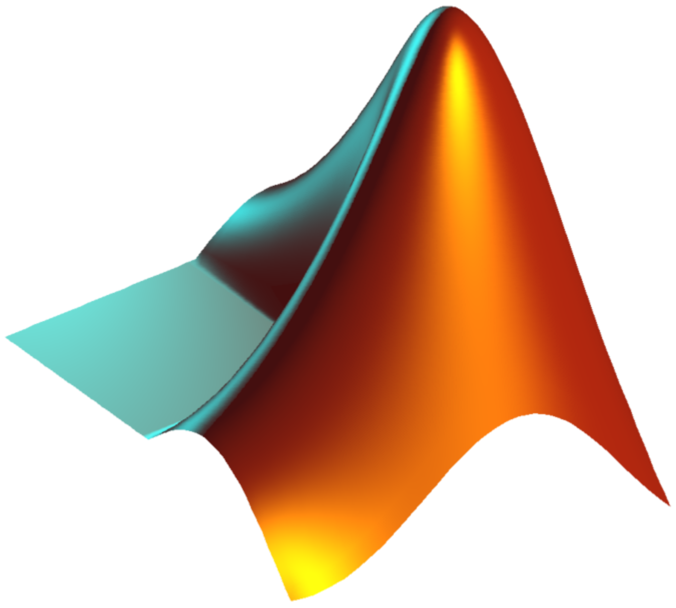
Mathematical Model of a CubeSat Attitude Determination System in MATLAB

Sergio Ribeiro, Rohith Yerrabelli

Tuesday May 3, 2022



1 Project Purpose

The purpose of this project is to provide a mathematical model for computing a CubeSat's orientation based on the response of six photo-diodes on the center of each face of the cube. The first part of this project is to provide a mathematical description of light falling on the face of each cube.

We will need to come up with three things,

1. A method to describe the orientation of the CubeSat in three dimensional space
2. A method to compute and define cube rotations
3. A method to compute the light falling on a cube face based on the cube rotation

Once these items are properly described, we can then begin to work backwards and compute the diode responses on each cube face. Here are some simplifying assumptions we are making about this problem,

1. We are assuming that the diode response is linear and proportional to the light falling on it. We do not consider the case of where the photo-diode is saturated or pushed to a non-linear region of the curve.
2. We are not considering the effect of temperature on the diode response curve. This would be proportional to the light falling on the diode as this will heat up the panel the diode rests on.
3. We are not considering the reflection of the light source falling on the other faces of the cube.
4. We assume the light source is at a nearly infinite distance, so the light flux can be assumed to be constant in the vicinity of the space of the CubeSat.

1.1 Describing the CubeSat Orientation

In Figure 1 we can see the CubeSat described by a series of six vectors. Each of the vectors are unit vectors are orthogonal to each other. Each of the unit vectors is normal to the plane of the face it describes.

We will use the following naming convention for each of the faces that are described by the unit vectors,

Face Name	Unit Vector Associated	Vector Notation
North X	$+\hat{\mathbf{x}}$	$\langle +1, 0, 0\rangle$
North Y	$+\hat{\mathbf{y}}$	$\langle 0, +1, 0\rangle$
North Z	$+\hat{\mathbf{z}}$	$\langle 0, 0, +1\rangle$
South X	$-\hat{\mathbf{x}}$	$\langle -1, 0, 0\rangle$
South Y	$-\hat{\mathbf{y}}$	$\langle 0, -1, 0\rangle$
South Z	$-\hat{\mathbf{z}}$	$\langle 0, 0, -1\rangle$

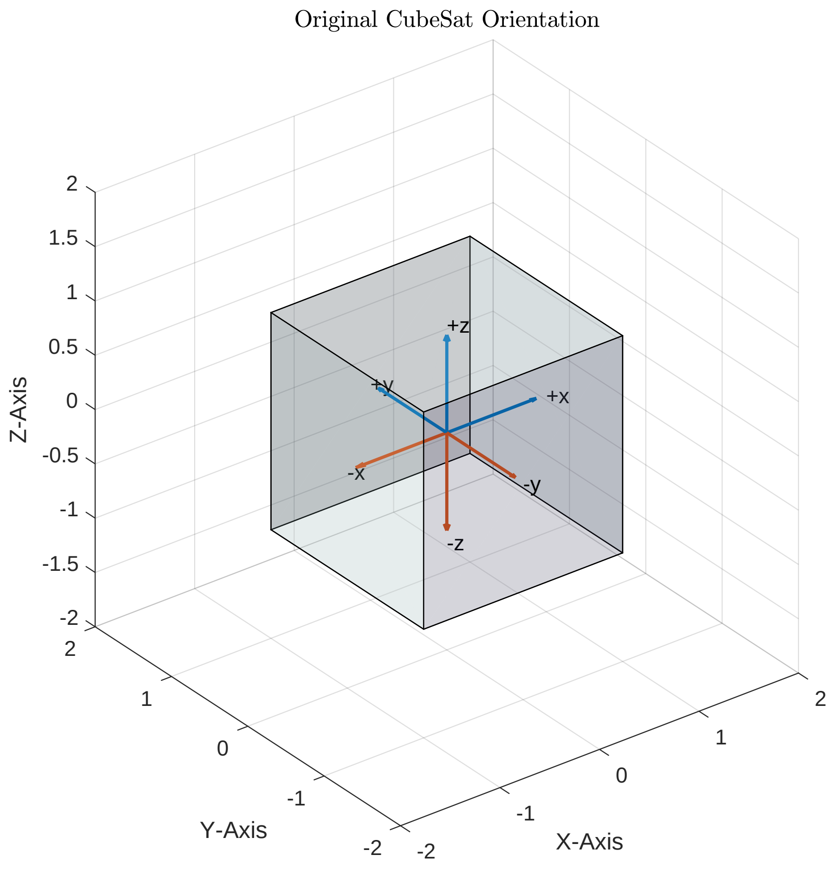


Figure 1: The CubeSat in an orientation where all the faces are aligned with the x, y and z axes.

1.2 Computing CubeSat Rotation in 3D Space

Computing roll, pitch and yaw coordinates can be achieved using three rotation matrices as defined below. The R_x matrix defines 'roll', the R_y matrix defines 'pitch' and the R_z matrix defines 'yaw'.

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

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

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

Multiplying a column vector (as shown below),

$$\begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$$

By R_x, R_y and R_z will rotate the vector by roll, pitch and yaw in the order they are multiplied. For our sake we will define our rotations in terms of roll, pitch and yaw in that order.

Therefore our overall rotation matrix will be,

$$R(\alpha, \beta, \gamma) = R_x(\alpha) \cdot R_y(\beta) \cdot R_z(\gamma)$$

Figure 2 shows how the rotation matrices rotate the entire orientation of the cube and the vectors describing each cube face are rotated with it.

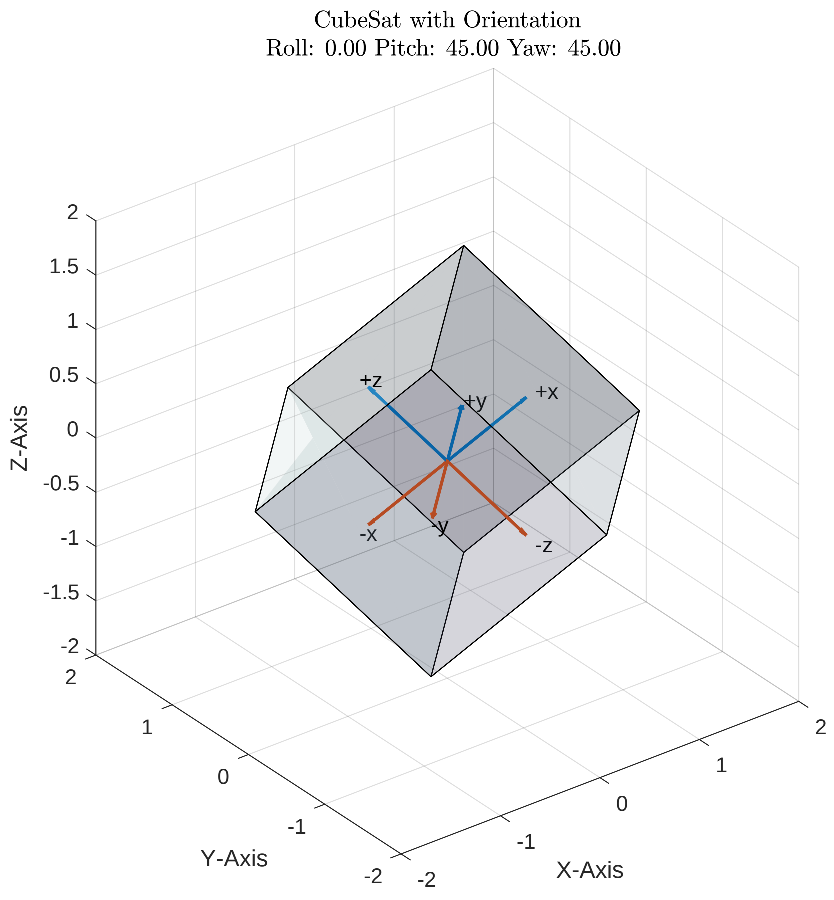


Figure 2: The CubeSat has been rotated with a roll of 0°, a pitch of 45° and a yaw of 45°.

1.3 Computing the Light Flux

Computing the light flux can be accomplished by defining what the light field looks like in the vicinity of the cube. The light flux is assumed to be from the sun and since the light source is far (infinite distance) the light flux is a constant vector field.

We will define this vector field as,

$$\begin{aligned}\mathbf{\Phi}_{\text{Sun}}(x,y,z) &= \phi_x \cdot \hat{\mathbf{x}} + \phi_y \cdot \hat{\mathbf{y}} + \phi_z \cdot \hat{\mathbf{z}} \\ &= \langle \phi_x, \phi_y, \phi_z \rangle\end{aligned}$$

This shows that it is a constant vector throughout the entire three dimensional space we are considering. Now computing the light flux through each space will be a simple task. Ordinarily we would need to compute a surface integral for each surface, however since each surface is perfectly flat and square and the vector field is constant, no calculus is required to evaluate the light flux through each face.

The light flux is a scalar quantity that can be calculated by the following equations where A is the area of the cube face,

$$\begin{aligned}\Phi_{\text{North, x}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x,y,z) \cdot \hat{\mathbf{x}}_{\text{North}} \\ \Phi_{\text{North, y}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x,y,z) \cdot \hat{\mathbf{y}}_{\text{North}} \\ \Phi_{\text{North, z}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x,y,z) \cdot \hat{\mathbf{z}}_{\text{North}} \\ \Phi_{\text{South, x}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x,y,z) \cdot \hat{\mathbf{x}}_{\text{South}} \\ \Phi_{\text{South, y}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x,y,z) \cdot \hat{\mathbf{y}}_{\text{South}} \\ \Phi_{\text{South, z}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x,y,z) \cdot \hat{\mathbf{z}}_{\text{South}}\end{aligned}$$

When using these formulas, the positive flux numbers represent cubes that are directly in the light. The negative numbers are faces that are shaded. The response of the photodiodes of the negative numbers would be zero in reality.

2 Sample Data Generated for Diodes

The sample data generated from a known sun flux vector field $\mathbf{\Phi}_{\text{Sun}}(x,y,z)$ and known cube orientation specified by roll, pitch and yaw angles are are recorded in CSV files. They have been attached to the project as raw files to look at.

The CSV files are `RollAngleChange.csv`, `PitchAngleChange.csv` and `YawAngleChange.csv`. The following plots will show the diode response curves associated with changing roll, pitch and yaw angles. The starting sun flux vector is $\langle 15, 18, 12 \rangle$. The default satellite orientation is $0°$ roll, $0°$ pitch and $0°$ yaw. Then each of these parameters is stepped from $0°$ to $90°$. The diode response curves are plotted in Figures 3, 4 and 5.

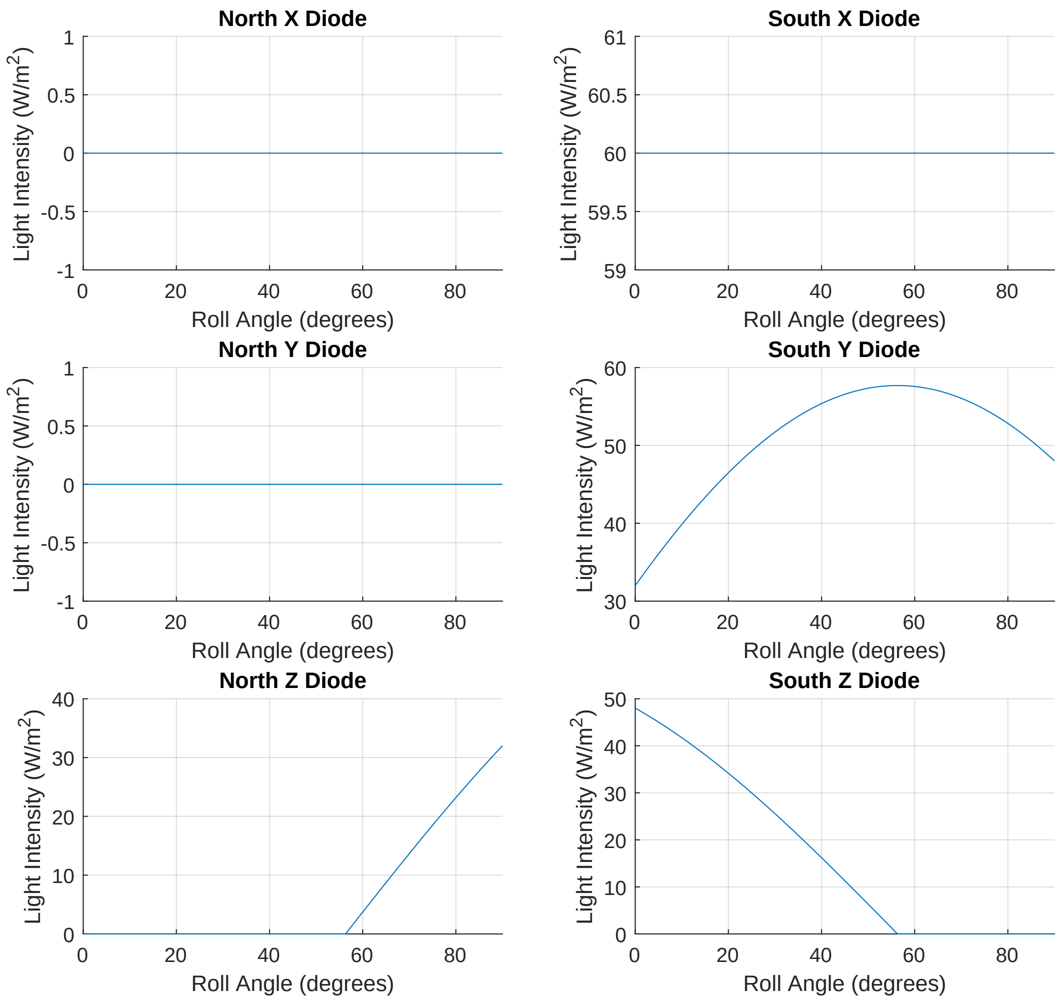


Figure 3: Roll angle is changed continuously from 0° to 90°.

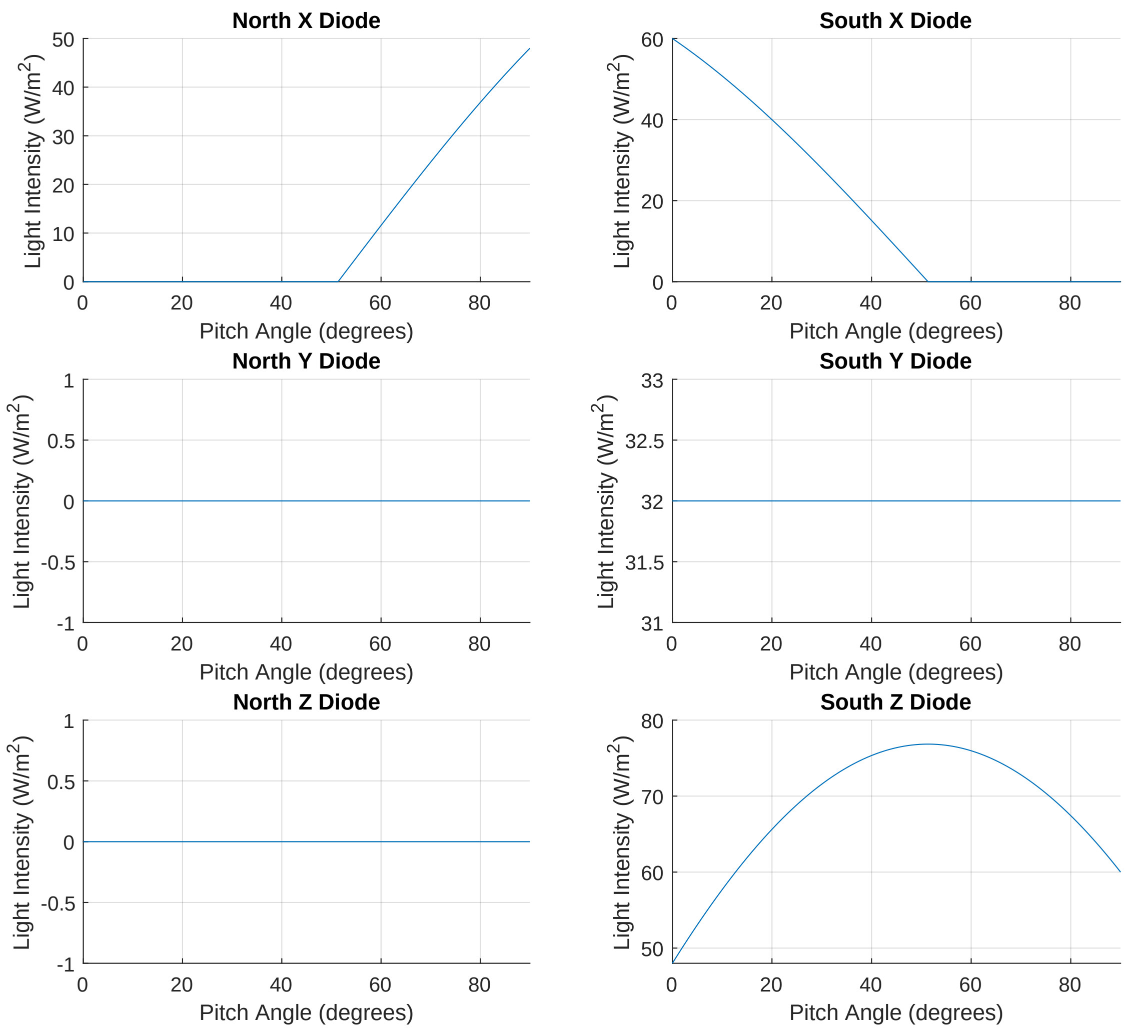


Figure 4: Pitch angle is changed continuously from 0° to 90°.

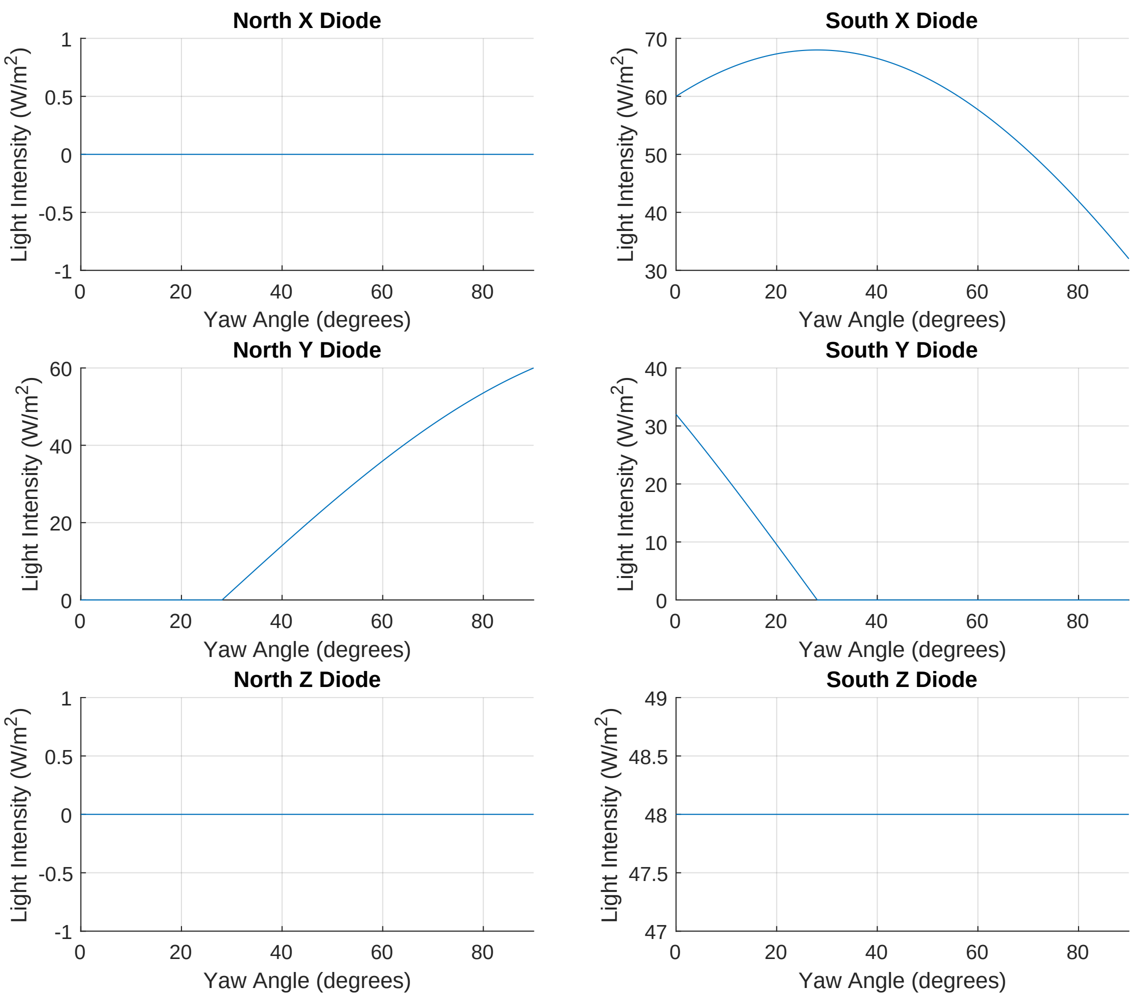


Figure 5: Yaw angle is changed continuously from 0° to 90°.

3 Determining CubeSat Orientation from Diode Response

3.1 Octant Naming Convention

From our proposed model of how the light flux shines on each face of the CubeSat, we can see that any given orientation only illuminates three cube faces at any given moment. Knowing this information allows us to at least know what octant the cube is oriented and where the light is shining. It also tells us approximately what octant the source is in. Octants will be named according to which corners of the unit cube are present in the octant. Each corner is specified by the unit vector sum associated with each cube corner.

Octant Name	Corresponding Vector	Decimal
000	$\langle -1, -1, -1 \rangle$	0
001	$\langle -1, -1, +1 \rangle$	1
010	$\langle -1, +1, -1 \rangle$	2
011	$\langle -1, +1, +1 \rangle$	3
100	$\langle +1, -1, -1 \rangle$	4
101	$\langle +1, -1, +1 \rangle$	5
110	$\langle +1, +1, -1 \rangle$	6
111	$\langle +1, +1, +1 \rangle$	7

Table 1: Octant naming convention based on vectors.

3.2 Insolubility of the Rotation Matrix by Direct Means

Attempting to solve the attitude control problem directly without making some simplifying assumptions yields a set of difficult equations to solve that we have not found any easy solutions to.

Fundamentally the equations that we need to solve are the following,

$$\begin{aligned}
\Phi_{\text{North, x}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x, y, z) \cdot R(\alpha, \beta, \gamma)\hat{\mathbf{x}} \\
\Phi_{\text{North, y}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x, y, z) \cdot R(\alpha, \beta, \gamma)\hat{\mathbf{y}} \\
\Phi_{\text{North, z}} &= -A \cdot \mathbf{\Phi}_{\text{Sun}}(x, y, z) \cdot R(\alpha, \beta, \gamma)\hat{\mathbf{z}} \\
\Phi_{\text{South, x}} &= A \cdot \mathbf{\Phi}_{\text{Sun}}(x, y, z) \cdot R(\alpha, \beta, \gamma)\hat{\mathbf{x}} \\
\Phi_{\text{South, y}} &= A \cdot \mathbf{\Phi}_{\text{Sun}}(x, y, z) \cdot R(\alpha, \beta, \gamma)\hat{\mathbf{y}} \\
\Phi_{\text{South, z}} &= A \cdot \mathbf{\Phi}_{\text{Sun}}(x, y, z) \cdot R(\alpha, \beta, \gamma)\hat{\mathbf{z}}
\end{aligned}$$

In the set of above equations the quantities of flux on each cube face is known as that is what the diodes measure, the area of each cube face (A) is known and the unit vector values $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ are known. What we are trying to solve for is R. If we treat all the quantities as matrices our equations become,

$$\begin{bmatrix} \Phi_{\text{North, x}} \\ \Phi_{\text{North, y}} \\ \Phi_{\text{North, z}} \end{bmatrix} = -A \begin{bmatrix} \Phi_{\text{Sun, x}} & \Phi_{\text{Sun, y}} & \Phi_{\text{Sun, z}} \\ \Phi_{\text{Sun, x}} & \Phi_{\text{Sun, y}} & \Phi_{\text{Sun, z}} \\ \Phi_{\text{Sun, x}} & \Phi_{\text{Sun, y}} & \Phi_{\text{Sun, z}} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and,

$$\begin{bmatrix} \Phi_{\text{South, x}} \\ \Phi_{\text{South, y}} \\ \Phi_{\text{South, z}} \end{bmatrix} = -A \begin{bmatrix} \Phi_{\text{Sun, x}} & \Phi_{\text{Sun, y}} & \Phi_{\text{Sun, z}} \\ \Phi_{\text{Sun, x}} & \Phi_{\text{Sun, y}} & \Phi_{\text{Sun, z}} \\ \Phi_{\text{Sun, x}} & \Phi_{\text{Sun, y}} & \Phi_{\text{Sun, z}} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

However this representation is unable to be solved due to the fact that the matrix on the right hand side of the equation is always singular due to the fact that the rows are all linearly dependent.

3.3 Paring Down the Solution Space

We cannot solve this matrix directly. Our first step will be to pare down the space of possible solutions to these two matrix equations. This can be determined by using the following lookup table regarding which cube faces are illuminated. A "1" means the cube face is receiving light and "0" means no visible light was detected,

North X	North Y	North Z	South X	South Y	South Z	Octant
0	0	0	1	1	1	000
0	0	1	1	1	0	001
0	1	0	1	0	1	010
0	1	1	1	0	0	011
1	0	0	0	1	1	100
1	0	1	0	1	0	101
1	1	0	0	0	1	110
1	1	1	0	0	0	111

Table 2: Face shading to octant lookup table.

3.4 Determining Angles in Relation to the Flux

This method is based off of computing the angle between two vectors based on their dot product. In Figure 6 we can see the definition of the dot product,

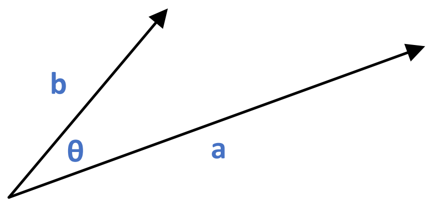


Figure 6: The dot product $\mathbf{a} \cdot \mathbf{b}$ can be calculated by the formula $|\mathbf{a}||\mathbf{b}| \cos\theta$.

From here we can easily back calculate the angle between the vectors by solving the definition,

$$\begin{aligned} |\mathbf{a}||\mathbf{b}| \cos\theta &= \mathbf{a} \cdot \mathbf{b} \\ \cos\theta &= \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} \\ \theta &= \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right) \end{aligned}$$

From this definition we can gain some more information about the CubeSat orientation in space. When the angle between the face of the cube and the flux vector is 0° then the flux is at its maximum.

$$\max\left(\Phi_{\text{North, x}}\right) = A|\mathbf{\Phi}_{\text{Sun}}(x,y,z)|$$
$$\Phi_{\text{North, x}} = -A\mathbf{\Phi}_{\text{Sun}}(x,y,z)\cdot\hat{\mathbf{n}}_{\text{North, x}}$$

Therefore an angle can be determined for each face by doing,

$$\theta = \arccos\left(\frac{\Phi_{\text{North, x}}}{\max\left(\Phi_{\text{North, x}}\right)}\right)$$

As shown in Figure 7.

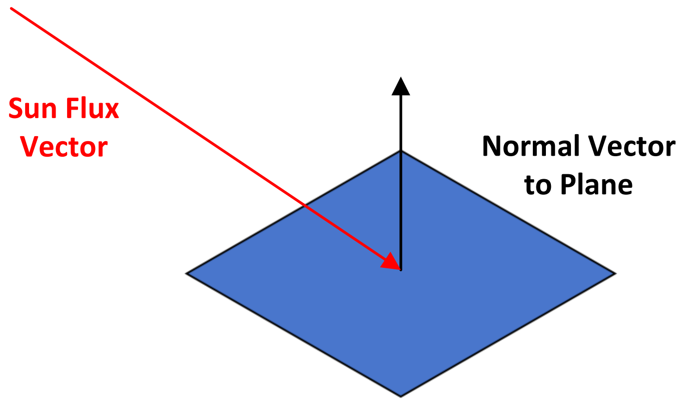


Figure 7: Sun flux vector hitting the plane of the cube face.

When computing this however we find the angle is unique to the orientation of the cube (as far as we can tell). We have not sufficiently developed the technique enough to convert the angle back to roll, pitch and yaw angles.

Here is a sample printout of our MATLAB code in the `CubeDisplay.m` code.

```
SUN FLUX VECTOR
    SunFluxX: -1
    SunFluxY: 2
    SunFluxZ: 3

CURRENT CUBE ORIENTATION
    Roll: 0
    Pitch: 0
    Yaw: 0

LIGHT FLUX RECEIVED AT EACH CUBE FACE
(NEGATIVE VALUES HERE ARE PRESENTED FOR DIAGNOSTIC PURPOSES)
    NorthX: 4
    NorthY: -8
    NorthZ: -12
    SouthX: -4
    SouthY: 8
    SouthZ: 12

COMPUTE ANGLES FROM CUBE FACE TO FLUX VECTOR
    NorthX: 74.498640433063002
    NorthY: 1.223115332374239e+02
    NorthZ: 1.433007747995101e+02
    SouthX: 1.055013595669370e+02
    SouthY: 57.688466762576155
    SouthZ: 36.699225200489877
```

And for when we perturb the roll orientation by 10 degrees we get,

```
SUN FLUX VECTOR
    SunFluxX: -1
    SunFluxY: 2
    SunFluxZ: 3

CURRENT CUBE ORIENTATION
    Roll: 10
    Pitch: 0
    Yaw: 0

LIGHT FLUX RECEIVED AT EACH CUBE FACE
(NEGATIVE VALUES HERE ARE PRESENTED FOR DIAGNOSTIC PURPOSES)
    NorthX: 4
    NorthY: -9.962240156100828
    NorthZ: -10.428507614811053
    SouthX: -4
    SouthY: 9.962240156100828
    SouthZ: 10.428507614811053

COMPUTE ANGLES FROM CUBE FACE TO FLUX VECTOR
    NorthX: 74.498640433063002
    NorthY: 1.317306884224274e+02
    NorthZ: 1.341695479077966e+02
    SouthX: 1.055013595669370e+02
    SouthY: 48.269311577572573
    SouthZ: 45.830452092203451
```

Currently due to the time constraints on this project, we have been unable to derive a fundamental relationship between the roll, pitch and yaw angles and the angles calculated by the `CubeDisplay.m` script. This will be an area of further work, however it seems plausible to find the roll, pitch and yaw given this information.

For a simpler flux vector and roll/pitch/yaw perturbation the results are much easier to interpret. An example with a flux vector of $\langle 1, 0, 0\rangle$ is shown below as well when pertubing the yaw angle by $10°$. We can clearly see the angle computed at `SouthX` corresponds to the correct yaw angle perturbation.

```
SUN FLUX VECTOR
    SunFluxX: 1
    SunFluxY: 0
    SunFluxZ: 0

CURRENT CUBE ORIENTATION
     Roll: 0
    Pitch: 0
      Yaw: 10

LIGHT FLUX RECEIVED AT EACH CUBE FACE
(NEGATIVE VALUES HERE ARE PRESENTED FOR DIAGNOSTIC PURPOSES)
    NorthX: -3.939231012048832
    NorthY: 0.694592710667721
    NorthZ: 0
    SouthX: 3.939231012048832
    SouthY: -0.694592710667721
    SouthZ: 0

COMPUTE ANGLES FROM CUBE FACE TO FLUX VECTOR
    NorthX: 1.700000000000000e+02
    NorthY: 80
    NorthZ: 90
    SouthX: 10.000000000000012
    SouthY: 9.999999999999999e+01
    SouthZ: 90
```

In reality the negative flux numbers would also be zero. This would help narrow down the possible solutions by providing us information about what octant the cube is oriented to. A more rigorous look at the angle information may yield a numerical solution to the problem of determining yaw, pitch and roll. However as of now, we have been unable to find that solution.

3.5 Code Results

Our code is fully capable of simulating the diode responses for each cube face given a known flux vector and a specified roll, pitch and yaw orientation.

A more rigorous treatment of the interpretation of the angles between the known flux vector and the cube faces however is required to properly back calculate the roll, pitch and yaw orientation. Currently this is beyond our capabilities, however this research should serve as a good starting point for future iterations of this project.

We have generated a code base for this project and posted it as a publicly available repository on GitHub. It can be found here `https://github.com/saribeiro/CubeSatOrientation.git`

4 Future Improvements

4.1 Problems with Roll, Pitch and Yaw

Regarding the representation of rotations or the linear combination of rotations, it is easy to understand and use the roll, pitch and yaw matrices. However it is known that certain issues arise with using this relatively simplistic method of representing rotations in space.

The issue of 'gimbal lock' often arises in attitude control systems that only use roll, pitch and yaw where a degree of freedom is lost due to two parameters lining up. This can result in a complete loss of orientation and the inability to regain control of the spacecraft. One such mathematical representation that helps avoid ambiguous rotations or gimbal lock from occurring is the quaternion representation for rotations.

Perhaps if we rebuilt our mathematical model using quaternion algebra to represent the rotations and to calculate the light fluxes on each cube face, it may yield solvable equations that can pinpoint orientations exactly.

4.2 Quaternions

Quaternions are a mathematical object invented by Sir William Rowan Hamilton that is an extension of the complex numbers $\mathbb{C}$. Quaternions are sometimes referred to as 'hypercomplex numbers' and are symbolized by the set $\mathbb{H}$. Quaternions have four components associated with them.

A quaternion $\mathbf{q}$ can be written as follows,

$$\mathbf{q} = q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k}$$

There are four 'unit vectors' associated with quaternions. We have the scalar value 1, and the other three components $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$. Quaternion algebra does not preserve the commutative property and thus the multiplication rules for quaternions are slightly more complicated. Figure 8 shows how the multiplication of unit quaternions works. As you can see multiplication is not commutative.

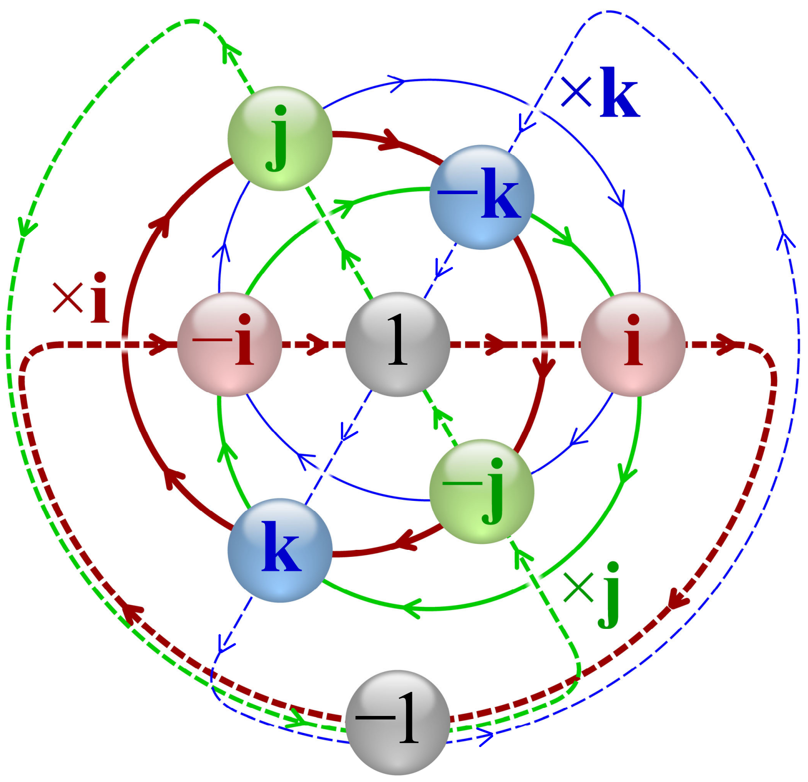


Figure 8: Descriptive chart to show how quaternion multiplication works.

×	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

William Hamilton's original motivation for developing quaternions was to extend the rotation properties of complex numbers to three dimensions. Ordinarily complex numbers are able to encode rotations on a two dimensional plane as a single complex number with a real and imaginary component.

MATLAB has a large number of quaternion capabilities built into it. We have compiled some sample code in MATLAB for working with quaternions.

4.3 Rotation Using Quaternions

Quaternion rotation can be achieved by specifying an axis about which we rotate our point(s) (described as a unit vector $\langle r_x, r_y, r_z \rangle$) and the rotation angle by which we want to rotate by.

Once we have built our rotation quaternion (symbolized by $\mathbf{q}_\theta$) we can rotate our three dimensional point described as a vector $\mathbf{r}$ by the following computation,

$$\mathbf{r}_{\text{rotated}} = \mathbf{q}\mathbf{r}\mathbf{q}^*$$

The quaternion conjugate $\mathbf{q}^*$ is identical to the idea of a complex conjugate. If the quaternion $\mathbf{q} = q_0 + q_1 \cdot \mathbf{i} + q_2 \cdot \mathbf{j} + q_3 \cdot \mathbf{k}$, then the conjugate is $\mathbf{q}^* = q_0 - q_1 \cdot \mathbf{i} - q_2 \cdot \mathbf{j} - q_3 \cdot \mathbf{k}$. The vector $\mathbf{r}$ can be thought of a quaternion where the first scalar argument is simply zero. Thankfully the associative property applies to quaternion algebra.

Calculating the rotation quaternion is actually quite trivial and it looks awfully similar to Euler's formula which represents complex number rotations.

$$\mathbf{q}_\theta = \cos\left(\frac{\theta}{2}\right) + (r_1 \cdot \mathbf{i} + r_2 \cdot \mathbf{j} + r_3 \cdot \mathbf{k}) \cdot \sin\left(\frac{\theta}{2}\right)$$
$$\mathbf{q}_\theta = \cos\left(\frac{\theta}{2}\right) + \mathbf{r} \cdot \sin\left(\frac{\theta}{2}\right)$$

Below is an example of using MATLAB's built in quaternion libraries and capabilities to rotate a cube about a defined axis. The full code for this is included in the appendix of this paper. See the results below in Figures 9 and 10. The rotation was computed by this code snippet,

```
% We could go the long way and multiply our vectors as q_rot * r *
% conj(q_rot). However MATLAB has built in routines for dealing with
% quaternions quite efficiently. It would be a waste to re-derive known and
% compiled code

for i = 1:8
    rot_verts(i, :) = quatrotate(compact(q_rot), solid_verts(i, :));
end
```

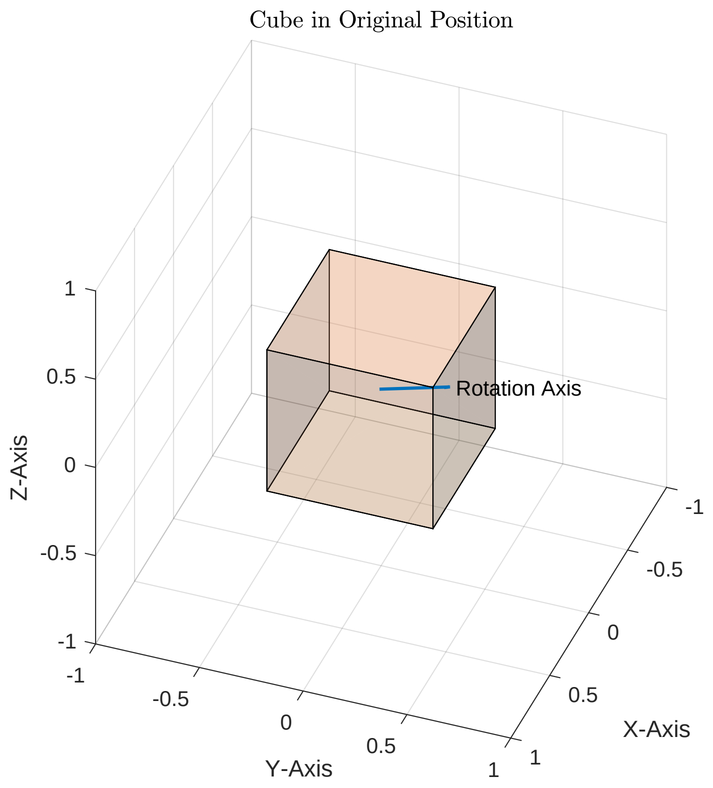


Figure 9: Original cube orientation. Rotation axis is shown as a blue unit vector along the vector $\langle 1, 1, 1 \rangle$.

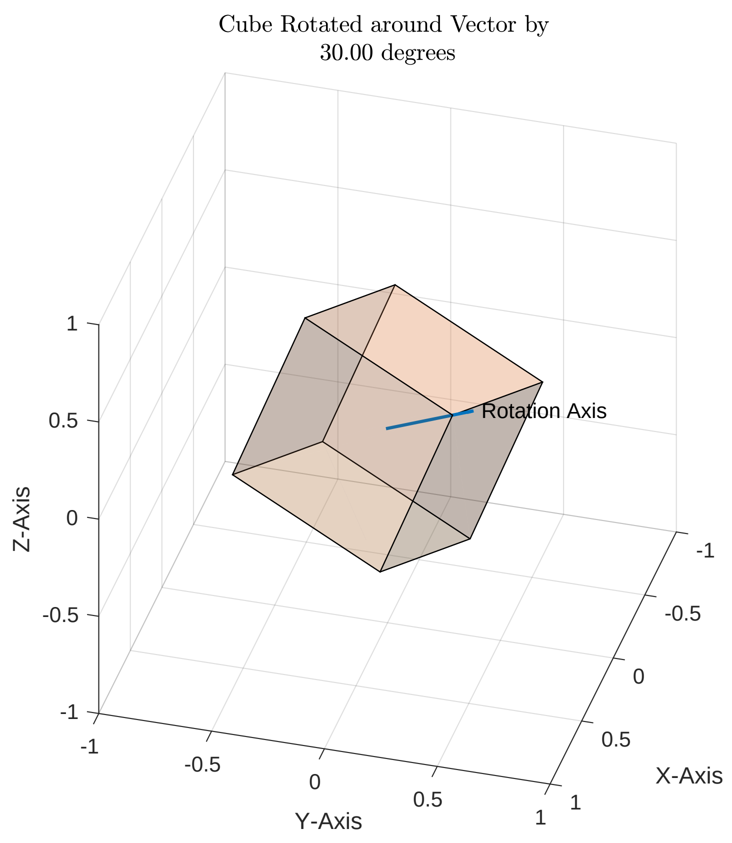


Figure 10: Cube has been rotated by 30° about the rotation axis.

5 Concluding Remarks

A more rigorous treatment of the interpretation of the angles between the known flux vector and the cube faces however is required to properly back calculate the roll, pitch and yaw orientation. Currently this is beyond our capabilities, however this research should serve as a good starting point for future iterations of this project.

We have generated a code base for this project and posted it as a publicly available repository on GitHub. It can be found here `https://github.com/saribeiro/CubeSatOrientation.git`. Future iterations of this project may find it far easier to rebuild or build upon our model using quaternion algebra and quaternion methods to describe cube rotations and cube orientations.

6 MATLAB Code

6.1 Cube Display Code

The purpose of this MATLAB code is to generate a useful visual representation of the item we are talking about and to generate useful figures and test data to spot check our assumptions.

```
%% ECE 580 Project, Mathematical Model for Cubesat Attitude Sensors
%
% ---------------------------------------------------------------------
%
% Authors:  Sergio Ribeiro, Rohith Yerrabelli
% Date:     26-APR-2022
% Class:    ECE 580 Small Satellite Design
%
% ---------------------------------------------------------------------
%
% The purpose of this project is to provide a mathematical model for a
% cubesat satellite attitude sensor system. The sensor system is a
% photodiode on each face of the cubesat. There are six sensors in all. The
% purpose of this program is to model the response of the photodiodes to
% the sunlight hitting the cubesat in different orientations. For
% simplicity we will consider a cube of size 2U x 2U x 2U. We are not using
% a unit cube because we would like our vectors for describing the cube
% orientation to be unit vectors.
%
% ---------------------------------------------------------------------
%

clearvars
clc
clf
format long

%% Sun Source Simulation
% The sun source will be simulated as a constant flux vector field. It is a
% constant vector value at all points in space around the cube. The flux
% value is expressed in W/m^2

Sun_Flux = [1; 0; 0];

SunFluxStruct = struct('SunFluxX', Sun_Flux(1), 'SunFluxY', Sun_Flux(2), ...
    'SunFluxZ', Sun_Flux(3));
disp('SUN␣FLUX␣VECTOR');
disp(SunFluxStruct);



%% Cube Setup

% Setup our origin unit vectors
origin = [0; 0; 0];
north_x = [1; 0; 0];
north_y = [0; 1; 0];
```

```matlab
north_z = [0; 0; 1];
south_x = [-1; 0; 0];
south_y = [0; -1; 0];
south_z = [0; 0; -1];

cube_verts = [1, 1, 1; -1, 1, 1; 1, 1, -1; -1, 1, -1; ...
       1, -1, 1; -1, -1, 1; 1, -1, -1; -1, -1, -1];
cube_faces = [1, 2, 4, 3; 5, 6, 8, 7; ...
    1, 5, 7, 3; 2, 6, 8, 4; ...
    1, 5, 6, 2; 3, 7, 8, 4];

% Create a figure showing the cubesat that we are considering
figure(1)
hold on
grid on
title('Original␣CubeSat␣Orientation', 'interpreter', 'latex');
quiver3(origin, origin, origin, north_x, north_y, north_z, ...
    'LineWidth', 1.5);
text(north_x, north_y, north_z, {'+x', '+y', '+z'});
quiver3(origin, origin, origin, south_x, south_y, south_z, ...
    'LineWidth', 1.5);
text(south_x, south_y, south_z, {'-x', '-y', '-z'});
axis([-2,2,-2,2,-2,2]);
xlabel('X-Axis');
ylabel('Y-Axis');
zlabel('Z-Axis');
patch('Vertices', cube_verts, 'Faces', cube_faces, ...
      'FaceVertexCData', bone(6), 'FaceColor', 'flat', ...
      'FaceAlpha', 0.2)
view(3)
axis vis3d
rotate3d

%% Pitch, Yaw and Roll Matrices
% In order to simulate the cube turning in different orientations we need
% to specify how it turns about the three axes by degrees. These changes
% are referred to as 'pitch', 'roll' and 'yaw'. Yaw is rotation about the
% z-axis in a clockwise fashion. Pitch is a rotation about the
% y-axis in a clockwise fashion. Roll is a rotation about the x-axis
% in a clockwise fashion. Here we define these standard matrices to
% compute these rotations.

%
% -----------------------------------------------------------------------
%
% DO NOT MODIFY THIS SECTION / STANDARD ROTATION MATRICES
%
% -----------------------------------------------------------------------
%
% Roll Matrix
x_rot = @(theta)([1, 0, 0; ...
    0, cosd(theta), -sind(theta); ...
    0, sind(theta), cosd(theta)]);

```

```
% Pitch Matrix
y_rot = @(theta)([cosd(theta), 0, sind(theta); ...
    0, 1, 0; ...
    -sind(theta), 0, cosd(theta)]);

% Yaw Matrix
z_rot = @(theta)([cosd(theta), -sind(theta), 0; ...
    sind(theta), cosd(theta), 0; ...
   0, 0, 1]);

% Combined Roll, Pitch, Yaw matrix
xyz_rot = @(theta_x, theta_y, theta_z)(x_rot(theta_x) * y_rot (theta_y) * ...
    z_rot(theta_z));

%
% -----------------------------------------------------------------------
%

%% Rotated Cube
% This section will test out our roll, pitch and yaw matrix to see if we
% can get a cube that is properly rotated

roll = 0; % X Rotation
pitch = 0; % Y Rotation
yaw = 10; % Z Rotation

CubeOrientation = struct('Roll', roll, 'Pitch', pitch, 'Yaw', yaw);
disp('CURRENT␣CUBE␣ORIENTATION');
disp(CubeOrientation);

rot_matrix = xyz_rot(roll, pitch, yaw);

north_x_rot = rot_matrix * north_x;
north_y_rot = rot_matrix * north_y;
north_z_rot = rot_matrix * north_z;
south_x_rot = rot_matrix * south_x;
south_y_rot = rot_matrix * south_y;
south_z_rot = rot_matrix * south_z;

cube_verts_rot = cube_verts * rot_matrix;

figure(2)
hold on
grid on
title({'CubeSat␣with␣Orientation', ...
    sprintf('Roll:␣%2.2f␣Pitch:␣%2.2f␣Yaw:␣%2.2f␣', roll, pitch, yaw)}, ...
    'interpreter', 'latex');
quiver3(origin, origin, origin, north_x_rot, north_y_rot, north_z_rot, ...
    'LineWidth', 1.5);
text(north_x_rot, north_y_rot, north_z_rot, {'+x', '+y', '+z'})
quiver3(origin, origin, origin, south_x_rot, south_y_rot, south_z_rot, ...
    'LineWidth', 1.5);
text(south_x_rot, south_y_rot, south_z_rot, {'-x', '-y', '-z'})
xlabel('X-Axis');
```

```matlab
ylabel('Y-Axis');
zlabel('Z-Axis');
axis([-2,2,-2,2,-2,2]);
patch('Vertices', cube_verts_rot, 'Faces', cube_faces, ...
      'FaceVertexCData', bone(6), 'FaceColor', 'flat', ...
      'FaceAlpha', 0.2)
view(3)
axis vis3d
rotate3d


%% Compute the Light Flux
% This section will compute the light flux through each cube face
% The faces will be named according to which unit vector is normal to the
% surface and the positive vectors will be referred to as 'North' and
% negative vectors will be referred to as 'South'

CubeArea = 4;

disp('LIGHT␣FLUX␣RECEIVED␣AT␣EACH␣CUBE␣FACE');
disp('(NEGATIVE␣VALUES␣HERE␣ARE␣PRESENTED␣FOR␣DIAGNOSTIC␣PURPOSES)');
CubeFlux = struct('NorthX', -CubeArea * dot(Sun_Flux, north_x_rot), ...
    'NorthY', -CubeArea * dot(Sun_Flux, north_y_rot), ...
    'NorthZ', -CubeArea * dot(Sun_Flux, north_z_rot), ...
    'SouthX', -CubeArea * dot(Sun_Flux, south_x_rot), ...
    'SouthY', -CubeArea * dot(Sun_Flux, south_y_rot), ...
    'SouthZ', -CubeArea * dot(Sun_Flux, south_z_rot));

disp(CubeFlux);


%% Get Angles
% This portion of the code gets the angle associated with each dot product
% in an attempt to reconstruct the roll, pitch and yaw.

Max_Flux = CubeArea * norm(Sun_Flux);

disp('COMPUTE␣ANGLES␣FROM␣CUBE␣FACE␣TO␣FLUX␣VECTOR');
CubeAngles = struct('NorthX', acosd(CubeFlux.NorthX/Max_Flux), ...
    'NorthY', acosd(CubeFlux.NorthY/Max_Flux), ...
    'NorthZ', acosd(CubeFlux.NorthZ/Max_Flux), ...
    'SouthX', acosd(CubeFlux.SouthX/Max_Flux), ...
    'SouthY', acosd(CubeFlux.SouthY/Max_Flux), ...
    'SouthZ', acosd(CubeFlux.SouthZ/Max_Flux));

disp(CubeAngles);
```

6.2 Diode Response Code

The purpose of this MATLAB code is to simulate the photo-diode responses given a sun vector, a starting orientation (roll, pitch and yaw) and stepping the degrees of one parameter (roll, pitch or yaw). The resulting data is parsed into a CSV file for later examination.

```
%% ECE 580 Project, Mathematical Model for Cubesat Attitude Sensors
%
% ------------------------------------------------------------------------
%
% Authors:  Sergio Ribeiro, Rohith Yerrabelli
% Date:     26-APR-2022
% Class:    ECE 580 Small Satellite Design
%
% ------------------------------------------------------------------------
%
% The purpose of this project is to provide a mathematical model for a
% cubesat satellite attitude sensor system. The sensor system is a
% photodiode on each face of the cubesat. There are six sensors in all. The
% purpose of this program is to model the response of the photodiodes to
% the sunlight hitting the cubesat in different orientations. For
% simplicity we will consider a cube of size 2U x 2U x 2U. We are not using
% a unit cube because we would like our vectors for describing the cube
% orientation to be unit vectors.
%
% ------------------------------------------------------------------------
%

clearvars
clc
clf
format long

%% Rotation Matrices for Roll, Pitch and Yaw
% Roll Matrix
x_rot = @(theta)([1, 0, 0; ...
    0, cosd(theta), -sind(theta); ...
    0, sind(theta), cosd(theta)]);

% Pitch Matrix
y_rot = @(theta)([cosd(theta), 0, sind(theta); ...
    0, 1, 0; ...
    -sind(theta), 0, cosd(theta)]);

% Yaw Matrix
z_rot = @(theta)([cosd(theta), -sind(theta), 0; ...
    sind(theta), cosd(theta), 0; ...
   0, 0, 1]);

% Combined Roll, Pitch, Yaw matrix
xyz_rot = @(theta_x, theta_y, theta_z)(x_rot(theta_x) * y_rot (theta_y) * ...
    z_rot(theta_z));

%% Calculate Flux Based on Spacecraft Orientation
```

```
% Describe the origin as well as the vectors describing each cubesat face
% Naming convention will be NorthX, NorthY, NorthZ, SouthX, SouthY, SouthZ

% Define the sun flux vector here
% Units are in W/m^2

Sun_Flux = [15; 8; 12];

% Describe the unit vectors corresponding to spacecraft
origin = [0; 0; 0];
north_x = [1; 0; 0];
north_y = [0; 1; 0];
north_z = [0; 0; 1];
south_x = [-1; 0; 0];
south_y = [0; -1; 0];
south_z = [0; 0; -1];


%% Loop to Calculate Light Flux on All Diodes
% Loop through various roll, pitch and yaw angles and store the values in
% an array to store the Diode responses to the varying roll pitch and yaw
% angles

angle_start = 0;
angle_step = 0.1;
angle_stop = 90;
angle_array = angle_start:angle_step:angle_stop;

Diode_NorthX = [];
Diode_NorthY = [];
Diode_NorthZ = [];
Diode_SouthX = [];
Diode_SouthY = [];
Diode_SouthZ = [];

roll_start = 0;
pitch_start = 0;
yaw_start = 0;

% Print output data results to a file
file_ID = fopen('PitchAngleChange.csv', 'w');

fprintf(file_ID, ...
    'Nx, Ny, Nz, Sx, Sy, Sz, Roll, Pitch, Yaw, SunFluxX, SunFluxY, SunFluxZ\n'
        );

for i = angle_start:angle_step:angle_stop

% Rotate these by a specified roll, pitch and yaw
rot_matrix = xyz_rot(roll_start, pitch_start + i, yaw_start);

% Calculate the rotated vectors
rot_north_x = rot_matrix * north_x;
rot_north_y = rot_matrix * north_y;
```

```
rot_north_z = rot_matrix * north_z;
rot_south_x = rot_matrix * south_x;
rot_south_y = rot_matrix * south_y;
rot_south_z = rot_matrix * south_z;

CubeArea = 4;

CubeFlux = struct('NorthX', -CubeArea * dot(Sun_Flux, rot_north_x), ...
     'NorthY', -CubeArea * dot(Sun_Flux, rot_north_y), ...
     'NorthZ', -CubeArea * dot(Sun_Flux, rot_north_z), ...
     'SouthX', -CubeArea * dot(Sun_Flux, rot_south_x), ...
     'SouthY', -CubeArea * dot(Sun_Flux, rot_south_y), ...
     'SouthZ', -CubeArea * dot(Sun_Flux, rot_south_z));

% Zero out diode responses that have negative flux
% These are calculated by multiplying the response by the boolean
% expression for what's true when the response is larger than 0

CubeFlux.NorthX = (CubeFlux.NorthX) * (CubeFlux.NorthX >= 0);
CubeFlux.NorthY = (CubeFlux.NorthY) * (CubeFlux.NorthY >= 0);
CubeFlux.NorthZ = (CubeFlux.NorthZ) * (CubeFlux.NorthZ >= 0);
CubeFlux.SouthX = (CubeFlux.SouthX) * (CubeFlux.SouthX >= 0);
CubeFlux.SouthY = (CubeFlux.SouthY) * (CubeFlux.SouthY >= 0);
CubeFlux.SouthZ = (CubeFlux.SouthZ) * (CubeFlux.SouthZ >= 0);

% Put the diode response curves in an array to be plotted later
Diode_NorthX = [Diode_NorthX, CubeFlux.NorthX];
Diode_NorthY = [Diode_NorthY, CubeFlux.NorthY];
Diode_NorthZ = [Diode_NorthZ, CubeFlux.NorthZ];
Diode_SouthX = [Diode_SouthX, CubeFlux.SouthX];
Diode_SouthY = [Diode_SouthY, CubeFlux.SouthY];
Diode_SouthZ = [Diode_SouthZ, CubeFlux.SouthZ];

fprintf(file_ID, '%5.8f,␣%5.8f,␣%5.8f,␣%5.8f,␣%5.8f,␣%5.8f,␣', ...
     CubeFlux.NorthX, CubeFlux.NorthY, CubeFlux.NorthZ, ...
     CubeFlux.SouthX, CubeFlux.SouthY, CubeFlux.SouthZ);

fprintf(file_ID, '%5.8f,␣%5.8f,␣%5.8f,␣', roll_start, ...
     pitch_start + i, yaw_start);

fprintf(file_ID, '%5.8f,␣%5.8f,␣%5.8f\n', ...
     Sun_Flux(1), Sun_Flux(2), Sun_Flux(3));

end

fclose(file_ID);

%% Diode Response Curves Plot
% This section will plot the diode response curves based on one of the
% angles of degrees of freedom rotating.

xlabel_string = 'Pitch␣Angle␣(degrees)';

figure(1)
```

```
% North X Diode Response
subplot(3,2,1)
hold on
grid on
plot(angle_array, Diode_NorthX);
title('North X Diode');
xlabel(xlabel_string);
ylabel('Light Intensity (W/m^2)');
xlim([angle_start, angle_stop]);

% North Y Diode Response
subplot(3,2,3)
hold on
grid on
plot(angle_array, Diode_NorthY);
title('North Y Diode');
xlabel(xlabel_string);
ylabel('Light Intensity (W/m^2)');
xlim([angle_start, angle_stop]);

% North Z Diode Response
subplot(3,2,5)
hold on
grid on
plot(angle_array, Diode_NorthZ);
title('North Z Diode');
xlabel(xlabel_string);
ylabel('Light Intensity (W/m^2)');
xlim([angle_start, angle_stop]);

% South X Diode Response
subplot(3,2,2)
hold on
grid on
plot(angle_array, Diode_SouthX);
title('South X Diode');
xlabel(xlabel_string);
ylabel('Light Intensity (W/m^2)');
xlim([angle_start, angle_stop]);

% South Y Diode Response
subplot(3,2,4)
hold on
grid on
plot(angle_array, Diode_SouthY);
title('South Y Diode');
xlabel(xlabel_string);
ylabel('Light Intensity (W/m^2)');
xlim([angle_start, angle_stop]);

% South Z Diode Response
subplot(3,2,6)
hold on
grid on
```

```
plot(angle_array, Diode_SouthZ);
title('South␣Z␣Diode');
xlabel(xlabel_string);
ylabel('Light␣Intensity␣(W/mˆ2)');
xlim([angle_start, angle_stop]);
```

6.3 Quaternion Code for Future Work

The purpose of this MATLAB code is to demonstrate the rotational capabilities using quaternions for future development. Perhaps in the next iteration of this project, we will build our model based solely on quaternion algebra.

```
%% ECE 580 Project, Quaternion Examples
%
% -------------------------------------------------------------------------
%
% Authors:  Sergio Ribeiro, Rohith Yerrabelli
% Date:     26-APR-2022
% Class:    ECE 580 Small Satellite Design
%
% -------------------------------------------------------------------------
%
% The purpose of this code is to test out MATLAB quaternion capabilities to
% use for future work on this project. MATLAB has built in quaternion
% capabilities specifically related to rotation problems.
%
% -------------------------------------------------------------------------
%


clearvars;
clc;
clf;
format long;

%% Commutative Property does not Apply for Quaternions
% This section demonstrates that quaternion multiplication is not
% commutative. Neither is it anti-commutative (i.e. A x B = -B x A).
% Reversing the order changes the output drastically.

q1 = quaternion(3, 5, 1, 4);
q2 = quaternion(2, 1, 5, 1);

q_12 = q1 * q2;
q_21 = q2 * q1;
disp('QUATERNIONS␣MULTIPLICATION␣IS␣NOT␣COMMUTATIVE');
disp(q_12);
disp(q_21);


%% Quaternion Rotation of Points
% We will build a quaternion rotation system. We need to define the unit
% vector which will describe the axis of rotation and an angle which will
% describe the angle by which we rotate around such a defined axis.

% Define a rotation axis and normalize it to a unit vector
rot_axis = [1, 1, 1];
rot_axis = rot_axis/norm(rot_axis);

% Define our angle of rotation in degrees
```

```matlab
rot_angle = 30;

% Compute the resulting quaternion associated with this rotation
q_rot = quaternion(cosd(rot_angle/2), ...
    rot_axis(1) * sind(rot_angle/2), ...
    rot_axis(2) * sind(rot_angle/2), ...
    rot_axis(3) * sind(rot_angle/2));


%% Define a Flat Polygon to Rotate
% In this section we will define a simple polygon shape to show how
% quaternion rotation works

solid_verts = 0.4 * [1, 1, 1; -1, 1, 1; 1, 1, -1; -1, 1, -1; ...
       1, -1, 1; -1, -1, 1; 1, -1, -1; -1, -1, -1];
solid_faces = [1, 2, 4, 3; 5, 6, 8, 7; ...
    1, 5, 7, 3; 2, 6, 8, 4; ...
    1, 5, 6, 2; 3, 7, 8, 4];



figure(1)
grid on
hold on
quiver3(0, 0, 0, rot_axis(1), rot_axis(2), rot_axis(3), 'LineWidth', 1.5);
text(rot_axis(1), rot_axis(2), rot_axis(3), 'Rotation Axis');
patch('Vertices', solid_verts, 'Faces', solid_faces, ...
      'FaceVertexCData', copper(6), 'FaceColor', 'flat', ...
      'FaceAlpha', 0.2)
axis([-1, 1, -1, 1, -1, 1]);
title('Cube in Original Position', 'interpreter', 'latex');
xlabel('X-Axis');
ylabel('Y-Axis');
zlabel('Z-Axis');
view(3)
axis vis3d
rotate3d


% Now using the rotation quaternion we defined, rotate all the solid
% figures vertices.

rot_verts = zeros(8, 3);
rot_faces = [1, 2, 4, 3; 5, 6, 8, 7; ...
    1, 5, 7, 3; 2, 6, 8, 4; ...
    1, 5, 6, 2; 3, 7, 8, 4];

% We could go the long way and multiply our vectors as q_rot * r *
% conj(q_rot). However MATLAB has built in routines for dealing with
% quaternions quite efficiently. It would be a waste to re-derive known and
% compiled code

for i = 1:8
    rot_verts(i, :) = quatrotate(compact(q_rot), solid_verts(i, :));
```

```
end

figure(2)
grid on
hold on
quiver3(0, 0, 0, rot_axis(1), rot_axis(2), rot_axis(3), 'LineWidth', 1.5);
text(rot_axis(1), rot_axis(2), rot_axis(3), 'Rotation Axis');
patch('Vertices', rot_verts, 'Faces', rot_faces, ...
      'FaceVertexCData', copper(6), 'FaceColor', 'flat', ...
      'FaceAlpha', 0.2)
axis([-1, 1, -1, 1, -1, 1]);
title({'Cube Rotated around Vector by ', ...
    sprintf('%2.2f degrees', rot_angle)}, 'interpreter', 'latex');
xlabel('X-Axis');
ylabel('Y-Axis');
zlabel('Z-Axis');
view(3)
axis vis3d
rotate3d
```