Sun Sensor Simulation

UCI CubeSat

November 2023

1 Introduction

This document details the Sun Sensor Simulation created for UCI CubeSat, which can be found at: *https://github.com/UCI-CubeSat/sun_sensor_py_simulator*.

2 Mathematical Context

This simulation employs three techniques for representing points in $\mathbb{R}^3$ space

- 3-Dimensional Cartesian Vectors
  These are vectors of the form $\langle x, y, z \rangle$, and represented in the file `vector_math.py` by the `namedtuple` object `cartVec3`.
  These can be converted to a Spherical Vector by the set of equations below:

$$R = \sqrt{x^2 + y^2 + z^2} \tag{1}$$

$$\theta = arctan(\frac{y}{x}) \tag{2}$$

$$\phi = arccos(\frac{z}{R}) \tag{3}$$

- 3-Dimensional Spherical Vectors
  These are vectors of the form $\langle R, \theta, \phi \rangle$ and represented in `vector_math.py` by the `namedtuple sphVec3`.
  These can be converted to a Cartesian Vector by the set of equations below:

$$x = R \cdot sin(\phi) \cdot cos(\theta) \tag{4}$$

$$y = R \cdot sin(\phi) \cdot sin(\theta) \tag{5}$$

$$z = R \cdot cos(\phi) \tag{6}$$

- Quaternions (4-Dimensional)
  Quaternions are a way to represent rotations within 3D space while avoiding the pitfalls of Spherical Vectors, such as Gimbal Lock. Quaternions are mathematically represented as:

$$cos(\frac{\sigma}{2}) + sin(\frac{\sigma}{2})\langle x, y, z \rangle \tag{7}$$

where $\langle x, y, z \rangle$ is a unit vector representing the axis that you want to rotate about, and σ represents the angle to rotate by.
Quaternions are represented by the class `Quaternion` in the `vector_math.py` file.

For a point $P = (x_0, y_0, z_0)$, the rotation about the quaternion $Q = (w, x, y, z)$ can be represented as:

$$P_{adj} = Q \cdot P \cdot Q^{-1}$$

This is represented in the code by the `__mul__`, `inverse`, and `rotate_point` methods within the Quaternion class (Sparing the space to delve into calculation details for summarizing this at a high level)

- 2D Gaussian Distribution
  This is used to simulate the "spread" of the Sun as it hits the simulated sensor. The mathematical equation is as follows:

  $$\frac{1}{2\pi\sigma^2} e^{\frac{-(x-\mu_x)^2+(y-\mu_y)^2}{2\sigma^2}} \tag{8}$$

  This is rather unnecessary to understand the code, however, if you are interested, refer to this *Desmos Simulation* of a normal distribution in 1D.
  This distribution is multiplied by the `peak_intensity` variable for solar simulation which creates a natural spread.

This concludes the mathematical concepts required to understand the sun sensor orientation.

3 Representing the Sensor

Now that we have prefaced with the mathematical context behind the problem of simulating the response of a CubeSat in 3D, lets get into the code analysis. This will primarily go over the `simulation.py` file, since `vector_math.py` is a set of helper functions for the math described above.

- Lines `1-5`
  Dependencies and Module Imports (NumPy, MatPlotLib, Vector Math, and the MatPlotLib 3D Library)

- Lines `6-12`
  Constants:

  – `sensor_width`: The Width of the Sensor Frame for NumPy `linspace` functions.

  – `num_pixels`: The number of pixels on the Quad-Photodiode Sun Sensor (4 Pixels per Photodiode was the "assumed" value)

  - `pixel_size`: The size per pixel based on the number of pixels and the width of the sensor.
  - `peak_intensity`: The comment explains my confusion with this, but I believe it represents the maximum intensity that the "Sun" can put on the sensor.

- Lines `13-17`
  `np.linspace(start, stop, step)` creates evenly spaced arrays for the simulated sensor, where the spacing is specified by the number of pixels (`num_pixels`) per axis. These are used to create a matrix representation of the sun sensor.

- Lines `18-29`
  The user is prompted to enter a "location" for where the Sun will hit the sensor (in the range $[-4, 4]$ for both x and y), and the `height` variable describes the height of the proposed "filter" above the actual photodiodes. The FOV represents the resultant FOV provided to the photodiodes by the filter. This *should* be a function of the height ($FOV = f(height)$) however more information on the filter specifics is needed to implement this.

- Lines `33-39`
  This section uses the 2D Gaussian Distribution (Equation 8), given the user-inputted `x` and `y` positions to create the spread which will be applied to the Plot for the Sun Sensor (With the FOV filter restricting this spread).

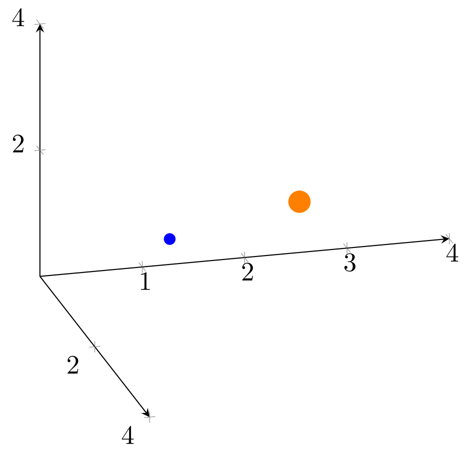
- Lines `42-43`

```
angle_with_x_axis = np.arctan2(y, x)
azimuth = np.radians((90 - np.degrees(angle_with_x_axis)) % 360)
```

  These two lines of code are used to calculate the azimuth angle, or the that represents the position of the (x, y) "strike" point of the Sun on the sensor, converting it to radians to make sure that it is in the range $[0, 2\pi]$ for future calculation. This will be important for figuring out the rotations later.

This concludes the *raw* sensor representations within Python, which are then utilized for visualization in the following section.

4 Visualization



Consider the above graph, which has a blue dot on point $P = (1, 1, 1)$ and an orange dot on point $S = (2, 2, 2)$. Assume that P represents the CubeSat, while S represents the Sun in $\mathbb{R}^3$ space.
We can calculate the vector $\vec{PS}$ as $\vec{PS} = vec(S - P) = vec((1, 1, 1)) = \langle 1, 1, 1\rangle$.

What about spherically? Spherical vectors as defined in Section 2 with the form $\langle R, \theta, \phi\rangle$ can also represent this relationship. We assume $R = 1$ since the real distance between the satellite and the sun doesn't matter in a fixed earth orbit. We already have θ as the Azimuth variable `azimuth` from lines `42-43`, and the Sun Sensor Math states that the elevation (ϕ) is equivalent to $arctan(x/y)$ for the user-inputted positions `x` and `y`.

Thus, `sat_to_sun` defined on line `47` represents the spherical vector from the Satellite to the Sun. Converting this to a Cartesian $\mathbb{R}^3$ vector results in $\langle 1, 1, 1\rangle$ as we calculated by hand earlier.

Now, since `sat_to_sun` contains a spherical vector, and `vm.sph_to_cart(sat_to_sun)` contains the cartesian version of it, we can compute the rest of the visualization. After the printing on lines `56-58`, `sat_to_sun` gets renamed to `spoint` for the rest of the program.

- Lines `61-77`

```
QuaternionX = vm.Quaternion(
    np.cos(np.radians(roll_angle/2)),
    (sin:=np.sin(np.radians(roll_angle/2)))*1,
    sin*0,
    sin*0)
```

```
QuaternionY = vm.Quaternion(
    np.cos((np.radians(90-np.degrees(spoint.phi)))/2),
    (sin:=np.sin((np.radians(90-np.degrees(spoint.phi)))/2))*0,
    sin*1,
    sin*0)
QuaternionZ = vm.Quaternion(
    np.cos(np.degrees(np.radians(spoint.theta)/2)),
    (sin:=np.sin(np.radians(np.degrees(spoint.theta))/2))*0,
    sin*0,
    sin*1)
```

This block creates 3 Quaternions for each rotation on the X, Y, and Z axes, using the Unit vectors $\langle 1,0,0\rangle, \langle 0,1,0\rangle, \langle 0,0,1\rangle$ for each respective axis, and rotating based on the azimuth, elevation, and roll (yaw, pitch, roll for "Aerospace-y" terms).

- Lines `77-145` These lines declare a cube and faces for a cube, rotate the vertices on the cube and generate the respective faces, and then create plots for each respective face. After that, the plots are displayed, and look somewhat like the figures which are on the following page.

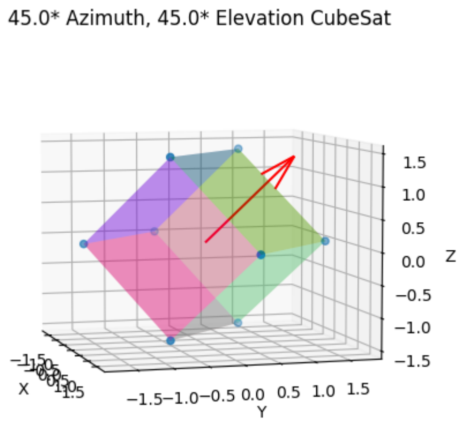


Figure 1: CubeSat Orientation at $\theta = 45°$, $\phi = 45°$ ()

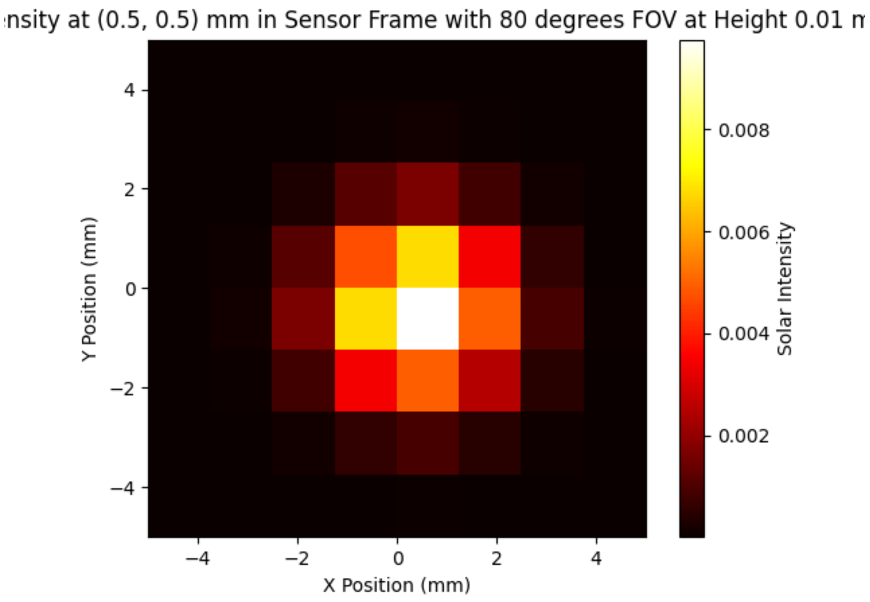


Figure 2: Corresponding Sun Sensor Output for Figure 1

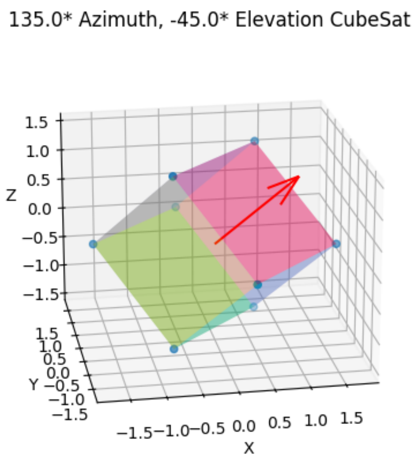


Figure 3: CubeSat Orientation at $\theta = 135°$, $\phi = 45°$ (The "-" on the Graph Title was a typo)

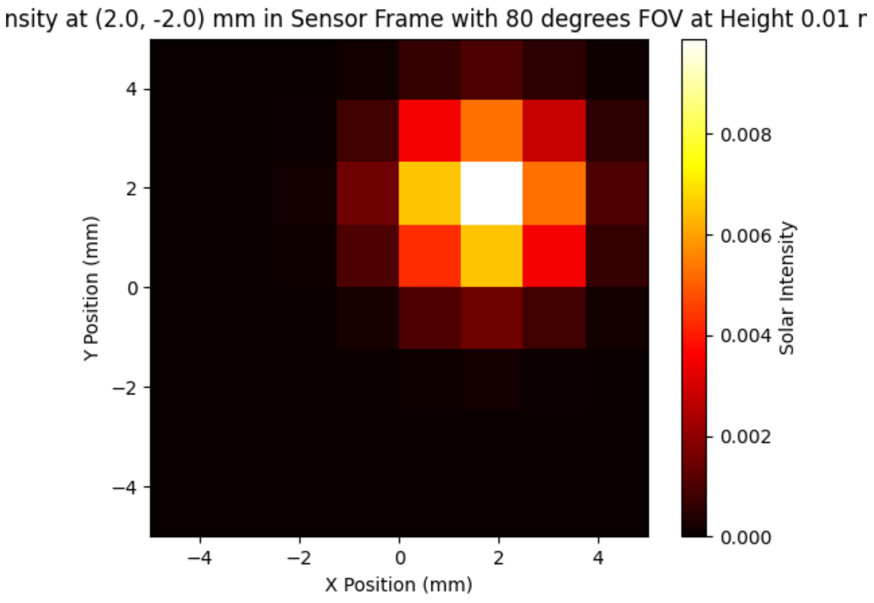


Figure 4: Corresponding Sun Sensor Output for Figure 3