Delft University of Technology

AE4S15 - Space Embedded Systems

Cubesat Sun Sensor Attitude Determination

Authors:
Arthur Thiam (4472586)
October 8, 2020

Instructors: Dr. A. Menicucci

1 Introduction

This report details the steps undertaken during the design of an on-board sun-sensor attitude determination computer, as part of the Space Embedded Systems course. The project largely consisted of two phases: the physical design logic and construction of the cubesat-like structure, and the data processing part. Chapter 2 will first describe the problem and system requirements. Chapters 3 and 4 describe the hardware and software side of the project respectively. Finally, Chapter 5 sheds light on the test results and system accuracay, and Chapter 6 provides a number of recommendations for further improvement of the system.

2 Problem Description and Requirements

The project consists of designing the on-board computer for an attitude determination system capable of deriving the sun-vector from sun-sensor readings. In order to do so, each side of the cube sat will be equipped with sun sensors, which provide an analog signal. These must then be converted to a sun vector through the use of a program. The main requirement for the system is the fault-tolerant characteristic: the failure of any single component cannot result in the failure of the full system.

The designed structure consists of three main elements. Firstly, sun-sensors installed throughout the satellite provide an analog reading that is dependent on the solar incidence angle. Secondly, an internal electrical circuit and arduino board handle the conversion of this analog signal to a digital signal, as well as communication with the serial port. Lastly a python code handles data-processing and converts the raw data into a sun vector. These three elements will be more elaborately explained throughout this report.

3 Electrical Circuit and Model

3.1 Electrical Circuit

The system makes use of LDR GL5528 sun sensors. These sensors are essentially variable resistors, of which the resistance changes as a function of the light they receive. Since the perpendicular component of the incident light is proportional to the angle under which the sensor is lit, a relation can be established between it's readout and the incidence angle. By combining the incidence angle of multiple satellite faces, the sun vector can be derived.

A number of things are required in order to properly read out the data from a sensor. The sensor will be powered on one side with the 5V power supply from the arduino board. However, the resulting signal can not directly be read out after the sensor. This would result in a consistent voltage drop that is not variable over different incidence angles. Instead, the sensor is first grounded over a 100K resistor. This provides the reference point needed to correctly read out the sensor. The electrical diagram for all sensors is shown in Figure 1:

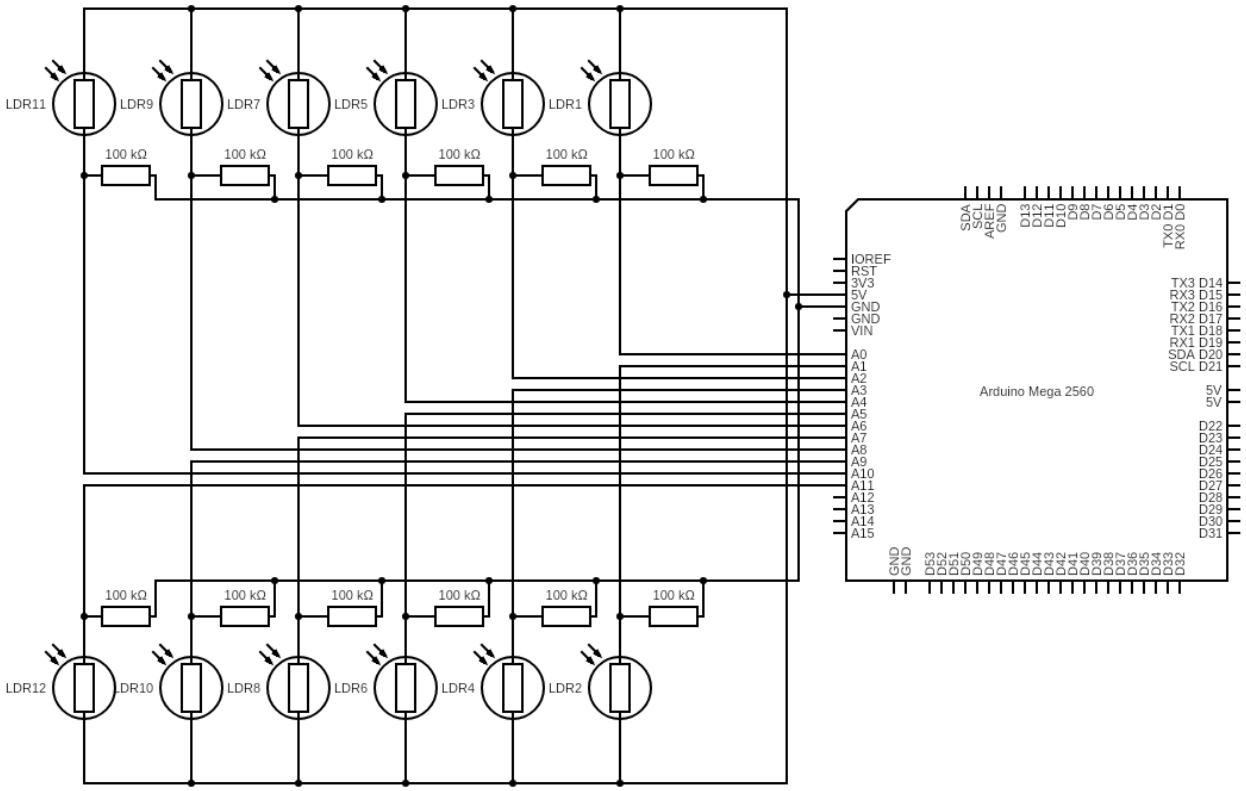


Figure 1: Sun sensor electrical diagram

As can be seen, there are twelve sensors for the 6 faces of the satellite cube. This ensures that every side gets two readings. In the event that one sensor malfunctions, the on-board computer will still receive relevant values from the second sensor. This contributes to the sytem fault-tolerance. The available equipment during completion of this project only allows for the use of one power/serial port, prompting the need for an Arduino Mega 2560. This board has sufficient analog pins to accomodate all sensors and communicate through a single serial port. In order to further increase the fault tolerance, two arduino unos could be used. This was purposefully not done given the available equipment, but the addition of a second arduino adds little to no complexity. Every satellite side would simply send one analogue system to one arduino and one to the second. In the event that an arduino breaks down, the other arduino will still receive all sensor values.

3.2 Cubesat model

The cubesat model was constructed with 6 20x20cm wooden panes (3mm thick). The arduino was secured to the base of the cubesat (z- plane) using screws and bolts. Additionally, a 3-piece breadboard was installed on the base to facilitate building the electric circuit. Figure 2 shows the open satellite:

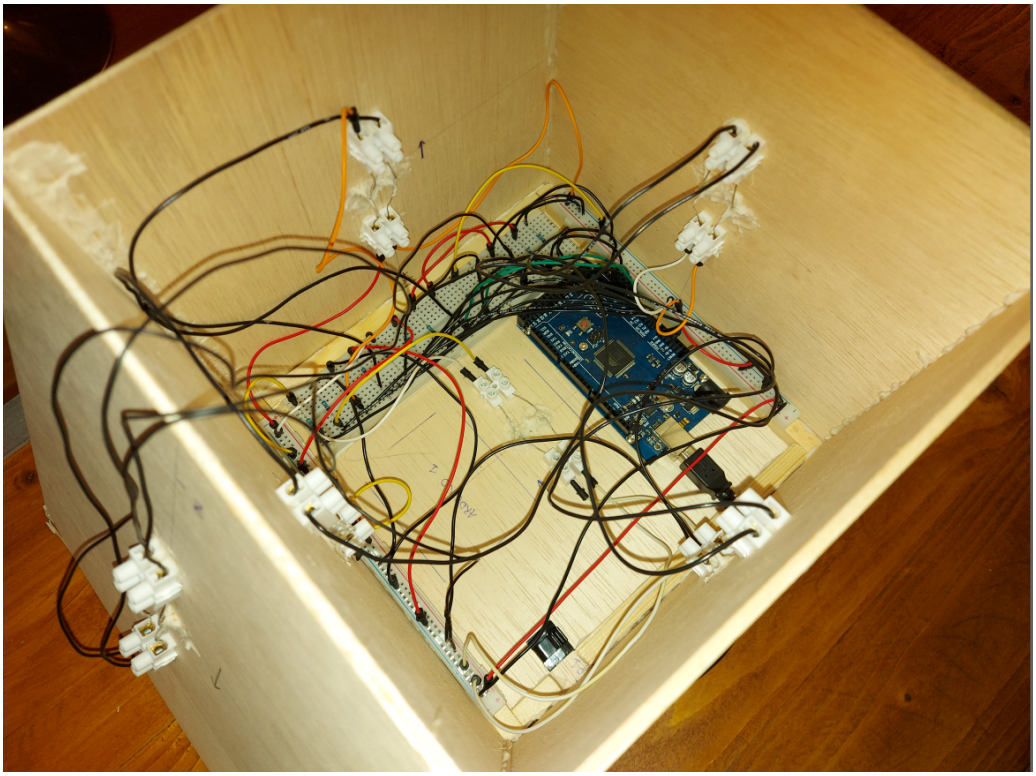
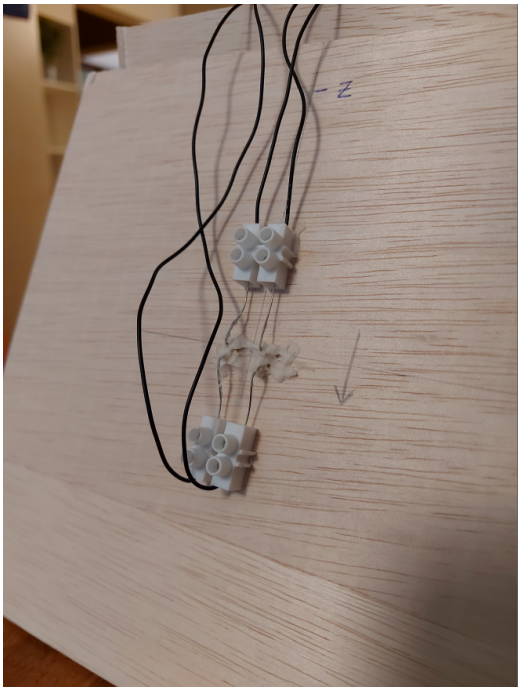


Figure 2: Inner Wiring

One opening was left in the cube for connecting the Arduino to a usb-port. 3 mm holes were drilled in the wood to secure the sensors. A small amount of glue was then added to the back side to ensure the sensors would stay in place, as well as to prevent the to ends from coming into contact and shorting the sensor. Screw terminals were also attached to the same surface and used to attach sensors to power or readout/ground. These screw terminals also aid in making the whole assembly more rigid.

(a) Sensor positioning



(b) Screw terminals

4 Signal Processing

The second element of this project is the data processing. This is ecompasses everything that happens between the board generating the analog-to-digital converted values and the output of a sun vector. The Arduino code is used only for extracting the digital values from the board and writing them to the serial port. A python code is used to detect a new set of output values. It then runs an (software) interrupt in order to determine the sun vector. These two sections of code are explained in more detail in this chapter.

4.1 On-board controller code

As mentioned above, the on-board code only handles reading data, converting it and writing it to the serial port. Before the main loop, it initializes the LDR readout values as integers and specifies the serial port parameters. Within the loop it continuously reads the values from the Analog-to-Digital converter, and writes them to the serial port. In this specific scenario, the actual sun vector is not calculated on the Arduino board itself. In reality, this python code would be ported to either a central satellite computer or to the Arduino itself. Porting python software to micro-controllers is discussed in more detail in the recommendations section.

4.2 Main Loop

The main outline of the python code consists of a nested loop that is either listening to the serial port or calucating the sun vector. The inner loop runs for as long as the software is listening for a dataset delimiter (denoted by ','). Once it is detected, listening is interrupted and the attitude is determined. The attitude is determined in three steps. First, the Data class is used to process the raw data. Secondly, the Attitude class is used to extract the unit vector. Finally, a moving average is applied in order to smooth out the result. The period of the moving average can be specified in 'calibration.ini', and is set to 3 by default. Once it is determined, the 'listening' variable is reset to True and the main loop continues[1]:

```
# While attitude determination is required
while running:

    # Listen to serial port until dataset delimiter ',' is detected
    while listening:
        output = ser.readline()
        if output == b',\r\n':

            # Dataset delimiter detected, run interrupt to determine attitude
            listening = False

    # Read all sensor values from the serial port
    A0 = int(ser.readline())
    A1 = int(ser.readline())
    A2 = int(ser.readline())
    A3 = int(ser.readline())
    A4 = int(ser.readline())
    A5 = int(ser.readline())
    A6 = int(ser.readline())
    A7 = int(ser.readline())
    A8 = int(ser.readline())
    A9 = int(ser.readline())
    A10 = int(ser.readline())
    A11 = int(ser.readline())

    # Save data into required format and instantiate Data class
    raw_data = [A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11]
    measurement = Data(raw_data)

    # calibrate, average and sort data
    sorted_data = measurement.sorted()

    # Request sun vector
```

[1] https://github.com/ArthurThiam/Cubesat-Attitude

```
attitude = Attitude(sorted_data)
sun_vector = attitude.unit_vector()

# Add data to temporary storage and move counter
counter += 1

if len(stored_data) < moving_average_period:
    stored_data.append(sun_vector)

elif len(stored_data) >= moving_average_period:
    stored_data.pop(0)              # remove oldest value
    stored_data.append(sun_vector)  # add newest value

# if counter is not yet above threshold, use singular value
if counter < moving_average_period:
    averaged_sun_vector = stored_data[-1]

# if counter is above threshold, apply moving average
elif counter >= moving_average_period:
    summed_data = [0, 0, 0]

    # Sum all of the parameters and store them in summed_data
    for i in stored_data:
        summed_data[0] += i[0]
        summed_data[1] += i[1]
        summed_data[2] += i[2]

    # Add averages to the averaged sun vector
    averaged_sun_vector = [(summed_data[0] / moving_average_period),
                           (summed_data[1] / moving_average_period),
                           (summed_data[2] / moving_average_period)
                           ]

normalized_vector = averaged_sun_vector/norm(averaged_sun_vector)
print('Normalized sun vector: ', normalized_vector)

listening = True
time.sleep(0.5)
```

4.3 Serial Port Interfacing

As mentioned above, the code knows to run an interrupt when it detects the dataset delimiter ','. This is a line that was embedded in the Arduino code[2], to be printed to the serial port each loop before the sensor values are. This means the python code knows that when it sees ',', the next line will contain the first sensor value.

In order to actually communicate with the serial port, a module called 'pyserial' was used. As can be seen in the code above, the serial port can be read using the 'ser.readline()' command. Note that the serial port to be used must be specified in the 'calibration.ini' file. For Windows based systems this will be of the format 'COM4', and for Linux systems it is of the format '/dev/ttyAMC0'. Once the raw data is obtained data processing is required to extract the relevant information, and attitude determination is required to determine the subsequent sun vector. In order to do this, a Data class and an Attitude class were built.

4.4 Data processing

A number of operations need to be performed on the raw data before it can be used to determine incidence angles and satellite attitude. The Data class does this by providing 3 methods.

[2]https://github.com/ArthurThiam/Cubesat-Attitude

4.4.1 Sensor Calibration

Firstly, not all sensors will read out the same values in the same conditions. Each sensor is slightly different, and in addition the electrical wires are not all of the same length or thickness. This means that the resistance from sensor to read-out pin is different for each sensor. In addition to this, the testing environment is not exactly similar to the final operational environment. There might be residual light left in 'full darkness', and the light source might have an intensity that does not result in a digital read-out of 1023 at 90 incidence. Each sensor is therefore individually calibrated before using the software. The 'calibrated()' method of the Data class does this by using the true sensor range (i.e. the calibration data) to project the measured sensor value onto the theoretical sensor range (i.e. 0-1023) as follows:

$$V_{projected} = \frac{V_{true} - V_{min}}{V_{max} - V_{min}} \cdot 1023 \tag{1}$$

Where V_{max} and V_{min} are the sensor specific minimum and maximum values. These are entered by the user in the 'calibration.ini' file before running the software.

4.4.2 Value averaging

Even with the calibrated values, one might find that two sensors read a slightly different value. This can be due to the inaccuracy of the calibration, or due to the fact that they are not placed at exactly the same location. The 'averaged()' method averages values of sensor pairs, and outputs only 12 sensor values. In addition, these values are attributed to axis identifiers 'x+', 'x-', etc... At this point, the sensor values are calibrated, averaged and classified by axis name.

4.4.3 Dominant Face Detection

The final operation is done by the 'sorted()' method. In order to determine the attitude of the satellite, the relevant data points must be extracted from the data set. Since the sensors are installed on a cube, a directional light source can only light up 3 faces simultaneously. The 'sorted()' method loops through the averaged data set to find the 3 highest values.

By using the Data class as follows one can therefore convert the raw pinout data into the calibrated and averaged sensor values of the three most dominantly lit surfaces:

```
sorted_data = Data([raw_data]).sorted()
```

This data is then further used to obtain the sun vector at a particular instant.

4.5 Attitude determination

The attitude determination consists of three main operations. The first operation consists of verifying if all sensor values in the processed data are above a detection threshold. Next, the incidence angles on the remaining surfaces are determined. Finally, those angles are combined with the knowledge of what surface they correspond to in order to determine the sun vector with respect to the local frame of reference.

4.5.1 Residual Light Removal

The light detection threshold is used to eliminate low level readings in the scenario where only 1 or 2 out of three faces are lit. This would happen when the light source is perpendicular to one surface, or parallel to a third. In the 'remove_low_ldr()' method, the program changes the values of data points that are below the threshold (which is specified in 'calibration.ini') to 0.

4.5.2 Incidence Angle Determination

The incidence angle is determined by comparing the sensor read-out to it's maximum value. The maximum value used here is obtained from the calibration data. The program counts the number of 0 values as set by the 'remove_low_ldr()' method, and based on that determines four scenarios:

- No surface is sufficiently lit: the satellite is in eclipse
- Only one surface is lit: The light source is shining perpendicular to that surface
- Two surfaces are lit: a 2D unit vector should be calculated
- Three surfaces are lit: a 3D unit vector should be calculated

The incidence angles are then calculated for each relevant surface using:

$$i = sin^{-1}\left(\frac{V_{sensor}}{V_{sensor-max}}\right) \tag{2}$$

The incidence angle method is the following[3]:

```
# determine incidence angles for three dominant sides
def incidence_angles(self):
    low_ldr_data = self.remove_low_ldr()

    # Check how many 0 vectors there are:
    counter = 0
    for i in low_ldr_data:
        if i[1] == 0:
            counter += 1

    if counter == 0:
        incidence_angles = [

            (low_ldr_data[0][0], asin(low_ldr_data[0][1]/1023)
                * 180 / pi),
               (low_ldr_data[1][0], asin(low_ldr_data[1][1]/1023)
                * 180 / pi),
               (low_ldr_data[2][0], asin(low_ldr_data[2][1]/1023)
                * 180 / pi)
            ]

        elif counter == 1:
            incidence_angles = [
                (low_ldr_data[0][0], asin(low_ldr_data[0][1] / 1023)
                * 180 / pi),
                (low_ldr_data[1][0], asin(low_ldr_data[1][1] / 1023)
                * 180 / pi),
                ('', 0)
            ]

        elif counter == 2:
            incidence_angles = [
                (low_ldr_data[0][0], asin(low_ldr_data[0][1] / 1023)
                * 180 / pi),
                ('', 0),
```

[3]https://github.com/ArthurThiam/Cubesat-Attitude

```
            ('', 0)
        ]

    else:
        incidence_angles = [
            ('', 0),
            ('', 0),
            ('', 0)
        ]

    return incidence_angles
```

At this point the raw data has been converted into a set of three incidence angles of the three most dominantly lit surfaces. The data also contains the name of the surface on which it is being measured. If one of these three surfaces is deemed unlit by the threshold value, it has been overridden to 0. An example of the data format after 'Attitude.incidence()' is called would be:

```
raw_data = [0, 0, 596, 640, 13, 25, 358, 684, 10, 6, 12, 49] # raw data measured from serial port
sorted_data = Data(raw_data).sorted_data() # Processed data from Data class
incidence_data = Attitude(sorted_data).incidence_angles() # incidence data from Attitude class

print(incidence_data)

incidence_data = [('y+', 55.974544572593786), ('x-', 42.066372823768674), ('', 0)]
```

The only operation remaining is to convert this to a unit vector which will correspond to the sun vector.

4.5.3 Vector Derivation

In its current format, the data already provides a good interpretation of where the sun is coming from. It is however not measured as a reference to a local reference frame. The components of any vector are dependent on the angles the unit vector makes with the relevant axis. For a 3D vector, the components can be broken down as following:

$$S = (cos\theta) \cdot i + (cos\phi) \cdot j + (cos\gamma) \cdot k \tag{3}$$

Where S is the sun vector or unit vector. The 'vector()' method derives these components by looping through the incidence angle data. if it finds the value of one data point is associated with x, y or z, it respectively assigns the results to the i, j or k components of the sun vector. If the face side is -, the component angle is subtracted from 180, and if it is + the component angle is left as is. Equation 3 uses angles a vector makes with the axis. In this case, the incidence data contains angles made with the surface, so the cosine is replaced with a sine. Since the low level LDR readings were overridden to 0, an unlit surface will automatically return a null vector component[4]:

```
# Method to derive unit vector from incidence angle data
def vector(self):
    incidence_data = self.incidence_angles()
    print('incidence_data: ', incidence_data)

    # Initialize unit vector components
    unit_component_1 = 0
    unit_component_2 = 0
    unit_component_3 = 0

    for i in incidence_data:
        if i[0] == 'x+':
            unit_component_1 = sin(i[1] * pi / 180)

        elif i[0] == 'x-':
            unit_component_1 = sin(i[1] * pi / 180 - pi)
```

[4]https://github.com/ArthurThiam/Cubesat-Attitude

```
        elif i[0] == 'y+':
            unit_component_2 = sin(i[1] * pi / 180)

        elif i[0] == 'y-':
            unit_component_2 = sin(i[1] * pi / 180 - pi)

        elif i[0] == 'z+':
            unit_component_3 = sin(i[1] * pi / 180)

        elif i[0] == 'z-':
            unit_component_3 = sin(i[1] * pi / 180 - pi)

    vector = [unit_component_1, unit_component_2, unit_component_3]
    print('vector return: ', vector)
    return vector
```

The value returned here is the sun vector. Once the moving average is applied to the returned value and the vector is normalized, the code will begin listening to the serial port again until the data set delimiter is encountered.

5 Test Results

This chapter gives a brief overview the insights testing provided with regards to the system accuracy. Chapter 6 details improvements that might improve the accuracy.

It should be noted that the largest loss in accuracy is a result of the testing conditions, and it is difficult to determine what portion of the measurement offset is actually due to the system characteristics.

The system performs very well in the scenario where only 1 surface is lit. The two low level readings are automatically discarded, and a unit vector with one non-zero component in the direction of the lit surface is returned:

```
raw data: [0, 0, 70, 63, 11, 36, 472, 729, 28, 26, 27, 79]
calibrated data: [('x-', 799.5434787194768), ('y+', 91.45474475555956), ('x+', 67.43807877421713)]
remove low ldr return: [('x-', 799.5434787194768), ('y+', 0), ('x+', 0)]
incidence_data: [('x-', 51.40431249315629), ('', 0), ('', 0)]
vector return: [-0.7815674278782765, 0, 0]
Normalized sun vector: [-1. 0. 0.]
```

When two surfaces are lit, a deviation of approximately 6 degrees from the expected reading is found. At a 45 angle between the y+ and x- axes, the following output is returned:

```
raw data: [0, 0, 596, 640, 14, 26, 359, 684, 10, 7, 13, 50]
calibrated data: [('y+', 847.8512001440533), ('x-', 686.1642608512209), ('x+', 39.67755422200561)]
remove low ldr return: [('y+', 847.8512001440533), ('x-', 686.1642608512209), ('x+', 0)]
incidence_data: [('y+', 55.974544572593786), ('x-', 42.12399572330306), ('', 0)]
vector return: [-0.670737302884869, 0.8287890519492213, 0]
Normalized sun vector: [-0.6288101 0.77755891 0.        ]
```

The "calibrated data" shows the LDR values projected onto the theoretical sensor range. The "remove low ldr" output shows that the x+ surface is not sufficiently lit to be considered. In the incidence data, we see the incidence angles on surfaces y+ and x-. The x- surfaces shows a deviation of 3 from the expected 45, and the y+ surfaces shows a 10 deviation. There are two elements to this deviation. On the one hand, the individual sensors deviate from the expected 45°. On the other, the two incidence angles do not add up to 90. This can be due to a number of reasons:

- The light source or satellite were not placed at exactly 45
- The light source is not exactly directional
- The sensor calibration was slightly offset for one of these two sensors
- The used light source is not powerful enough. With this light source, the true sensor ranges were limited to about 80% of their theoretical range. This reduces the amount of detail a sensor can capture.

Calculating the angle of the resulting unit vector, we find that these inaccuracies amount to a total deviation of 6.04 degrees:

$$\alpha = tan^{-1}\left(\frac{0.6288101}{0.77755891}\right) = 38.96 \tag{4}$$

It can be expected that a similar accuracy will be obtained for scenarios where all three surfaces are lit. With the available equipment it is however not possible to compare the output reliably to a true light source position/direction. An example of the vector output in a x-, y+, z+ lighting case can however be found below:

```
raw data: [1, 0, 600, 634, 15, 27, 496, 774, 591, 738, 36, 85]
calibrated data: [('z+', 1006.977168670818), ('y+', 846.666741694427), ('x-', 844.9771741996731)]
remove low ldr return: [('z+', 1006.977168670818), ('y+', 846.666741694427), ('x-',
    844.9771741996731)]
incidence_data: [('z+', 79.8459820523611), ('y+', 55.85617077235566), ('x-', 55.687938065754196)]
vector return: [-0.8259796424239231, 0.8276312235527146, 0.9843374082803694]
Normalized sun vector: [-0.53770376 0.54835951 0.64045024]
```

6 Recommendations

Several improvements to the design and testing procedures are suggested.

6.1 Light source characteristics

The biggest source of inaccuracy, in the context of ground testing, are the characteristics of the light source. The sun-vector derivation is based on the assumption that the light source is directional. The further away the light source is positioned, the more valid this assumption becomes. In this case however, the non-directionality of the light source will have an impact on the system accuracy.

In addition, the relatively weak light source used also reduces the effective sensory range from the theoretical 0-1023 down to an average 0-700. A lower range means there is less nuance to the output signal as a function of incident angle.

This issue can be tackled by taking into account the angle under which the light arrives at the satellite (with respect to a directional light source), as well as the distance between a sensor and the center of the satellite. It would however be preferable instead to replace the light source with a directional one, such that operational conditions are more closely replicated.

6.2 Fault Tolerance Assessment

As mentioned in Chapter 3, the system is fault tolerant in the sense that all necessary information to continue operation is still available in the event of a single sensor failure. The program could look at the difference in sensor pair values, and if the difference is above a certain threshold it will use only the information of the higher value sensor (a failed sensor will cause the board to read out ground, driving that apparent sensor value to 0).

In the case of a board failure, fault tolerance would have to be assured by a redundant board. Both boards then need access to all sensor values in order to continue operation. Due to available resources that kind of redundancy was not implemented. However, the addition of a second board would not require any additional code to be re-written.

6.3 Porting Code

The current software largely runs off of the satellite. For this system to truly be qualified as an embedded system, all code necessary to determine the sun vector would have to be run on board. There are several ways one could conceive doing so. Firstly, there are projects that allow directly compiling python code to a micro-controller like Arduino Mega (e.g. PyMite). The current code makes use of very basic python libraries, and would most likely be qualified for compiling on Arduino.

If compiling the code directly on the board is not deemed reliable or functional enough, the code could be run on a central computer that is able to run Python. This central computer could then communicate with all sensors requiring Python interaction. Of course if these solutions do not work, the current code could be re-written to C and written directly on the Arduino.

7 Conclusion

This report outlined the steps taken towards building and programming an attitude determination system based on light dependent resistor data. 12 sensor placed around the 6 surfaces of a cube sat representative structure provided the raw data, which was then converted to a unit vector in the direction of the light source in a number of steps. the accuracy of the resulting vector was analyzed for the scenario where two surfaces were lit, and was found to be within 7. The angular deviation is expected to be similar in the case of a 3D unit vector, but cannot be assessed with the available equipment. Finally, several improvements were proposed in order to further reduce system inaccuracy and improve system fault tolerance.