GLASGOW CALEDONIAN UNIVERSITY

MEng Group Research Project

MMH723842-24-AB-GLAS

Design and Implementation of a Photodiode Array-Based Analogue 2D Sun Sensor

word count: xxx

by Zac McCaffery, Alexandru Belea,
Sebastian Alexander, William Kong, Nassor Salim,

Date: April 14, 2025

Contents

List of Figures

List of Equations

Abstract

add abstract here

1. Acknowledgements

We would like to express our sincere gratitude to our supervisors, Dr. Roberto Ramirez-Iniguez and Geraint Bevan, for their invaluable guidance and unwavering support throughout this project.

Our appreciation extends to the 3rd Floor EEE Lab Technicians and Dr. Carlos Gamio-Roffe, whose technical expertise and assistance were instrumental in the successful construction of the prototype.

We are particularly grateful to the European Project Semester RED Team members—Nikolay Ivanov Shopov, Stef Hannisse, and Samridhi Gupta—for generously allowing the use of their Renewable Energy Demonstrator as a testbench. Their contribution provided an ideal platform for positioning light sources during the testing phase of this project.

2. Introduction

2.1. Problem Statement

With the ever-increasing commercialization of the space and satellite industry there is a growing need for a cost-effective method of attitude tracking for smaller satellite missions of such as CubeSat as these missions are purpose built for very specific objectives. Whilst the larger commercial satellite missions make use of expensive digital camera systems for tracking purposes, this is not feasible for much smaller CubeSat setups. CubeSats are defined from 1 unit to 12 – where 1U is a 10x10x10 cm satellite. Consequently, there is a demand for a cost-effective and easily implementable attitude tracking system that can provide accurate measurements for CubeSat missions, such as a Position Sensitive Detector (PSD) using photodiodes.

2.2. Aim of the Project

"To investigate and develop a cost-effective and reliable sun sensing solution suitable for Low Earth Orbit (LEO) nanosatellite attitude determination."

2.3. Objectives of the Project

To investigate the design of a sun sensing system for nanosatellites, used in orientation determination, through detection of its relative position to the sun using analogue sensors located on the satellite's body. Our goal is to create a system which balances cost-effectiveness and simplicity. To achieve this, we will create a software model of the analogue sensor(s) to simulate the system's ability to track the sun from various angles in orbit. After which, we aim to build a physical prototype and use a movable light source to simulate the sun's movement, allowing comparison between the real sensor's performance against our simulations. Although the physical prototype will be built using non-space-grade materials, one of the objectives is to look at and analyse materials required for building a space-grade PCB and sensor. For this step, the Mechanical side of the team will perform Printed circuit board (PCB) and aperture device finite analysis using ANSYS to determine resilience to environmental factors such as stress and thermal simulation. The application of signal processing will be explored to provide usable data, filter out noise, and improve the system's accuracy. This approach aims to develop a cost-effective and reliable, in-house sun sensing solution specifically for nanosatellites operating in Low Earth Orbit. Major Objective points:

- **Conduct literature review:**
  - Analyse existing research on sun sensing technologies, with a focus on PSD-based analogue sensors and their applications in nanosatellites.
  - Identify current challenges, best practices, and advancements in attitude determination in Low Earth Orbit. Use these insights to guide the design and optimisation of the proposed sun sensing system.
- **Develop software model:**
  - Simulate the performance of the PSD-based analogue sun sensor in tracking the sun's position from various angles in Low Earth Orbit.
- **Design and fabrication of physical prototype:**
  - Integrate analogue sun sensor components, test and validate its performance under controlled conditions.
- **Compare simulated and experimental results:**
  - Establish evaluation methodology between simulated and experimental test results to ensure that topology evaluation is applicable.
- **Optimise sensor topology:**
  - Research and evaluate various configurations of analogue sun sensing systems to maximise sun detection accuracy and minimise blind spots.
- **Investigate environmental factors:**
  - Evaluate the material requirements of the PCB and aperture device.
- **Implement signal processing algorithms:**
  - Investigate the filtering of noise to enhance the signal-to-noise ratio and otherwise ensure the acquisition of usable data for accurate sun position determination.
  - Implement data handling which optimises scanning rates and efficiently processes the analogue signal data for real-time attitude determination.
- **Document results and overall cost-effectiveness:**
  - Develop criteria for final evaluation of sun sensing systems, on which to base the final presentation of project findings.

3. LiteratureReview

3.1. CubeSat Design

Puig-Suari, Turner and Ahlgren published an IEEE paper in 2001 with the help of their students at California Polytechnic State University exploring a need for micro satellites for use by universities in an ever-expanding space programme. They provide as a solution a standard satellite form-factor that will bring down the cost of both manufacture and deployment of satellites by smaller entities: the CubeSat. The paper identifies a key component for the success of this form factor a need for a standard CubeSat deployer mechanism which can deploy several satellites safely and develop such a platform, called Poly Picosatellite Orbital Deployer or P-POD. They point out the need and provide microsatellite size and shape of the CubeSat form factor [?]. Sai balaji et al. performed a study using MATLAB simulation of several attitude control algorithms to look at the ability to control a CubeSat of size 1U. They also simulated sensors such as sun sensors, magnetometer, and gyroscope. They concluded that it is possible to operate the satellite using a magnetorquer type actuator and an array of mathematical models and algorithms: it would take 2000 seconds for a 1U satellite to stabilize at 505km, 98° degree attitude in orbit with the methods utilized by them [1]. Incentivised by the rapidly increasing use of LEO, Lopez-Calle and Franco perform a quantitative comparative study on the catastrophic failure of CubeSats and Nanosats from radiation exposure due to the harsh environment of space versus failure due to collisions in the increasingly busy Low Earth Orbit (LEO). The authors concluded that while sustained damage and damage protection from radiation exposure used to be and currently still is the most crucial factor in protecting LEO microsatellites, increasingly the risk of debris collisions is becoming more important and will become the most important in the following 50 to 70 years. The authors conclude that microsatellite designers need to move their focus more towards defence from debris impacts as these, even if not resulting in catastrophic failure of the satellite, they will impact the attitude of the satellite [?].

3.2. PSD Enabled Sun Sensor

3.3. Mechanical Design and Analysis

3.4. Photodiode Simulation and Signal Analysis

3.5. IoT Communication Enhancement with LEO Satellites

4. Methodology

4.1. Prototype Development

4.1.1. Lifecycle

This section provides an overview of the Prototype Development Lifecycle.

Conceptualization and Requirements Definition

- The prototype must have four photodiodes in an xy pattern with respective circuitry required to output 0-5 Volts that will be read by an Arduino based Data Acquisition System (DAQ). The circuit must be able to react to light intensity changes, however the change will be at low frequency (below 1Hz) as a satellite attitude is considered to change only gradually.

- While the prototype may not have a high accuracy, it is hoped that it will be enough to measure light position changes roughly, even if at a low accuracy of $10°$ or $20°$ but this will remain to be seen.

- The prototype within the scope of this paper will show the ability to detect the position of light at normal room conditions, therefore it does not need to withstand temperature changes or radiation that a final product would require if deployed in space.

- Interface requirements: the prototype electrical output needs to be compatible with the Arduino Analog to Digital Converter (ADC) input. Therefore, the signal shall not go below 0 Volts or exeed 5 volts.

- Size and weight are not of high importance, but the device must fit in the testing equipment, which is the Renewable Energy Demonstrator arch. Preferably a height not higher than 5cm.

Theoretical Design

- Research photodiode technology options and selection criteria

- Model sun sensor geometry and aperture design

- Determine optimal photodiode placement for coverage and accuracy
- Develop mathematical models for sun vector determination
- Simulate sensor performance under various lighting conditions

Preliminary Design

- The design of the Prototype must have four photodiodes in an xy patern with respective apertures placed such that they cover opposite halfs of the photodides. This is to facilitate light location detection by the aperture shadowing the photodiode when the light is on one side but not the other.

Component Selection and Procurement

- Select appropriate photodiodes (spectral response, sensitivity)
- Choose microcontroller/processor
- Source analog-to-digital converters
- Identify appropriate materials for aperture and housing
- Procure test equipment for validation

Breadboard Testing

- Assemble basic circuit on breadboard
- Test photodiode response characteristics
- Verify analog front-end performance
- Validate signal processing approach
- Identify design weaknesses and optimization opportunities

First Prototype Development

- Design printed circuit board (PCB)
- Manufacture PCB
- Design and fabricate aperture mask
- Develop housing/enclosure
- Assemble prototype components
- Write initial firmware implementation

Initial Testing and Characterization

- Conduct functional testing
- Measure photodiode response curves
- Characterize sun angle determination accuracy
- Test temperature sensitivity
- Evaluate power consumption
- Assess signal-to-noise ratio

Design Refinement

- Analyze test results
- Modify aperture design if needed
- Optimize photodiode configuration
- Update signal processing algorithms
- Refine PCB layout
- Improve firmware algorithms

Second Prototype Development

- Implement design improvements
- Manufacture revised PCB
- Fabricate improved aperture
- Enhance housing design
- Update firmware with optimized algorithms
- Assemble refined prototype

Comprehensive Testing

- Laboratory performance testing (angular accuracy, resolution)
- Environmental testing (thermal cycling, vibration)
- Radiation testing (if applicable for space applications)

- Interface compatibility testing
- Long-term stability assessment

Validation and Calibration

- Develop calibration procedures
- Create calibration fixtures
- Perform sensor calibration
- Document calibration coefficients
- Validate sensor performance against requirements

Documentation and Production Readiness

- Create detailed technical specifications
- Document calibration procedures
- Prepare assembly instructions
- Write user manual/interface control document
- Develop acceptance test procedures

Pre-production Prototype

- Build small batch of pre-production units
- Conduct acceptance testing
- Verify production processes
- Validate consistency between units
- Finalize design for production

Technology Transfer to Production

- Document manufacturing processes
- Train production personnel
- Establish quality control procedures
- Define production testing requirements
- Prepare for volume manufacturing

4.1.2. Signal Conditioning Circuitry

A photodiode produces a certain amount of current when light hits the depletion region. Therefore, a larger depletion region is desirable, to produce more current. [2021KeiserFibreOptics]

Functional Requirements

Design Approach

Technical Specifications

Implementation Plan

Testing Strategy

Deployment Process

Evaluation

4.1.3. Enclosure Design 3D print

This section provides an overview on the design and fabrication of the enclosure for the prototype. The enclosure is a critical component that houses the electronic components and provides protection against environmental factors. The design process involves several steps, including conceptualization, modeling, and fabrication.

Functional Requirements

Design Approach

Technical Specifications

Implementation Plan

Testing Strategy

Deployment Process

Evaluation

4.2. Data Acquisition System

4.2.1. Functional Requirements

The output signal from the photodiode array amplifier is required to be converted to digital form for post-processing. This requirement is filled by designing a Digital Acquisition System (DAQ) capable of recording the signal from the four photodiode circuits simultaneously. The choice of design was conceived by analyzing the analog signal and determining some basic requirements of the Analog to Digital Converter (ADC) the DAQ must possess.

Analog Signal Characteristics

- The signal is four channel, one per photodiode, and between 0 and 5 Volts, as the TIA and post amplification was designed specifically for this output.
- Close to DC frequency, i.e., static in nature, due to light intensity remaining static under most tests. One test is performed at 0.2Hz, which is still still very low frequency, with the light completing a semicircular arc once in 130 seconds (26 positions of 5 seconds each).
- Later in testing it was found that the signal is impacted by interference of 400mVpp at a frequency fluctuating from 160kHz to 180kHz from the RED testbench power supply, as pictured in Figure 4.1.

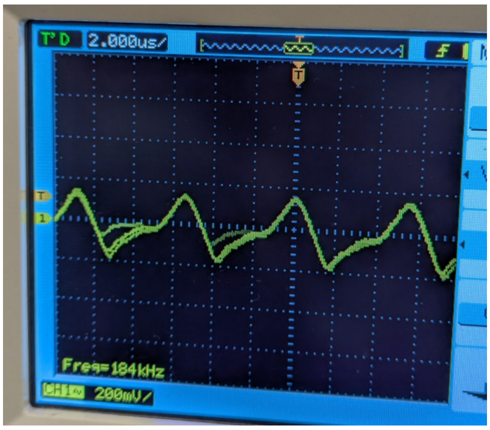

Figure 4.1.: Signal Noise Analysis, oscilloscope AC coupled

CSV Data Structure and Format Specification is as follows:

The output of the DAQ is to be saved in Comma Separated Values (CSV) file format, with columns as follows: Sample(nr.), Time(ms), A0(V), A1(V), A2(V), A3(V). This allows for easy post-processing and plotting.

4.2.2. Design Approach

The characteristics of the signal being low frequency, combined with the requirement to read all four signals simultanously and in sync, meant two things: the Sampling Rate could be quite low due to Nyquist theorem telling us that the sampling rate must be at least twice the frequency of the signal being sampled, in order to maintain the original signal without aliasing [2, p. 146]. Therefore a low performing ADC is acceptable for a signal changing at under 1Hz. And secondly, the DAQ must support sampling from at least four analog inputs. These requirements meant that a cheap Arduino based DAQ could fit perfectly the needs of the project: it is powered by the Atmega328P which has an included ADC of 15 ksps [3, p.205]. And the Arduino Nano has four analog inputs.

Arduino Programming

The Arduino-based DAQ will require both a C++ program written for the Arduino itself, as well as a program or script on the PC receiving the digitized signal, this is because the Arduino lacks both the memory requirements and capability to store the recorded digitized signal to some internal memory.

The Arduino C++ Program must be able to listen to commands from the user on the PC receiving, start a recording, and immediatly transmit to the PC over serial communication.

4.2.3. Technical Specifications

Arduino Code

The Arduino Code which uses the Arduino ADC is formed of the setup() function triggered once at the start/reset of the device and a standard contnuous loop triggered after setup comletes. Inside the loop, two if statements check for instructions from the PC script. The recording time limit is hardcoded as a global function. Figure 4.2 shows the algorithm as a Flowchart that checks for Serial data in, waits for a command to start recording, and if recording time has reached the preset limit, it stops recoring, sends the last values to the Python script on the PC and a "recording_stopped" command. A FSM diagram is also available in Figure 4.3. The pseudocode used while designing the Arduino side of the DAQ system, is available in Listing 4.1. The final code is available in Appendix A.1.1.

The Atmega328P does not have a separate ADC clock input, therefore the CPU clock is used by first dividing by a default rate of 128, this divider is changed to 16 by changing bits 2-0 to 100, as per [3, p.219]. This increases the clock speed available to the ADC for a higher sampling rate. This results in a 1MHz clock signal to the ADC (16MHz/16) which seemed needed when dealing with multiplexing four analog inputs to a single ADC. The process is as follows:

Original ADC Clock Speed (with default prescaler of 128):

$$f_{\text{ADC-default}} = \frac{f_{\text{CPU}}}{Prescaler_{\text{default}}} = \frac{16\text{ MHz}}{128} = 125\text{ kHz} \tag{1}$$

Optimized ADC Clock Speed (with modified prescaler of 16):

$$f_{\text{ADC-optimized}} = \frac{f_{\text{CPU}}}{Prescaler_{\text{optimized}}} = \frac{16 \text{ MHz}}{16} = 1 \text{ MHz} \tag{2}$$

Conversion Time Calculations: ADC requires approximately 13 clock cycles for each conversion [3, p.208] Optimized ADC Clock Speed (with modified prescaler of 16):

$$T_{\text{conversion-default}} = 13 \times \frac{1}{f_{\text{ADC-default}}} = 13 \times \frac{1}{125 \text{ kHz}} \approx 104 \ \mu\text{s} \tag{3}$$

$$T_{\text{conversion-optimized}} = 13 \times \frac{1}{f_{\text{ADC-optimized}}} = 13 \times \frac{1}{1 \text{ MHz}} \approx 13 \ \mu\text{s} \tag{4}$$

Time required to sample all 4 analog inputs:

$$T_{\text{4channels-default}} = 4 \times T_{\text{conversion-default}} = 4 \times 104 \ \mu\text{s} \approx 416 \ \mu\text{s} \tag{5}$$

$$T_{\text{4channels-optimized}} = 4 \times T_{\text{conversion-optimized}} = 4 \times 13 \ \mu\text{s} \approx 52 \ \mu\text{s} \tag{6}$$

Maximum theoretical sampling frequency for all 4 channels:

$$f_{\text{sampling-max-default}} = \frac{1}{T_{\text{4channels-default}}} = \frac{1}{416 \ \mu\text{s}} \approx 2.4 \text{ kHz} \tag{7}$$

$$f_{\text{sampling-max-optimized}} = \frac{1}{T_{\text{4channels-optimized}}} = \frac{1}{52 \ \mu\text{s}} \approx 19.2 \text{ kHz} \tag{8}$$

Actual limited sampling frequency (based on minSampleInterval = 2ms):

$$f_{\text{sampling-actual}} = \frac{1}{2 \text{ ms}} = 500 \text{ Hz per channel} \tag{9}$$

Effective data rate across all channels:

$$\text{Data Rate} = 4 \text{ channels} \times 500 \text{ Hz} = 2000 \text{ samples/second} \tag{10}$$

In real testing the actual sampling rate was closer to 330Hz for 5 second recordings or 100Hz for a 2 minute recording - after some investigation the only explanation was the relativly small size of the transmission buffer implemented by the Serial C++ library. The buffer is of only 64 bytes, and when it fills, the function Serial.write() (used by println()) will block the write untill there is space in the buffer[ref:arduino.cc/serial.write]. As our line of text is quite long "498,5000,0.059,0.054,0.073" for example has 26 characters (last line of a 5 second recording). For larger recording length, where the first and second column, Sample and Time, can get quite large, the sampling rate decreased considerably, but was kept constant (around 10ms for a 2 minute recording). Presumably due to optimization in the Arduino Serial Hardware/Software or compiler, it remains constant at 10ms. However this was not investigated further as for our near-DC signal, even 10ms was a fast enough sampling rate for our DC-like signal.

```
recordingDuration = 5000 // for how long to record in milliseconds

minSampleInterval = 2    // control how fast to sample to avoid
                         // relying on Arduino performance
// Initialize serial communication
// Initialize analog inputs
// Setup ADC

// Infrom PC listening on Serial Connection: Arduino is ready to
 record
Serial.print("Arduino_DAQ_Ready")
// enter the loop
void loop(){
  //listen for command from PC script:
    String command = Serial.read()
  //set system state
  if (command == "START"){
    // Send header of csv
    Serial.println("Sample,Time(ms),A0(V),A1(V),A2(V),A3(V)")
    // keep track of system state
    state = recording
    // keep time
    startTime = currentTime()
    // send confirmation
    Serial.println("recording in progress")
    }
  // check if recording
```

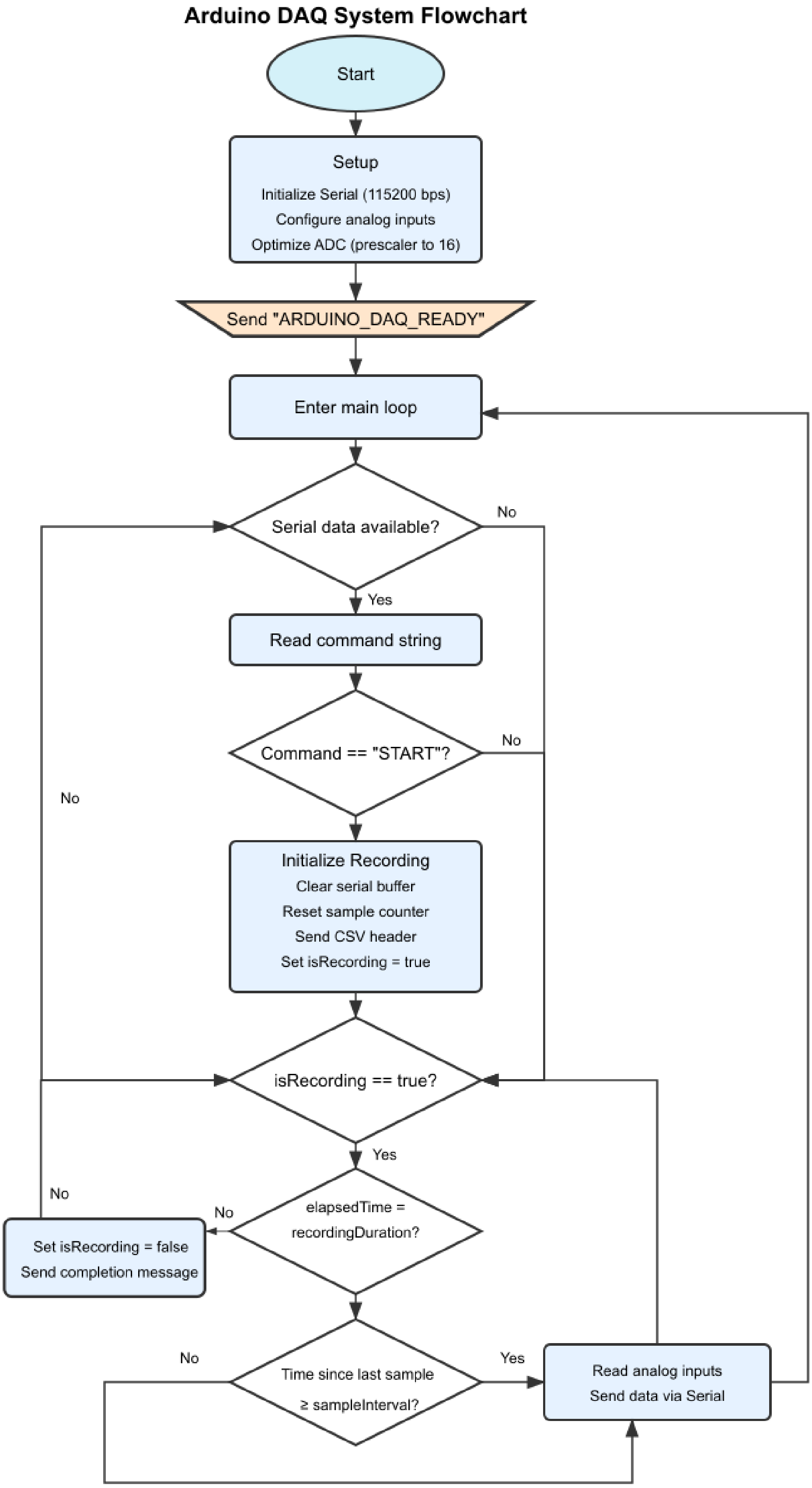

Figure 4.2.: Flowchart Arduino DAQ C++ program

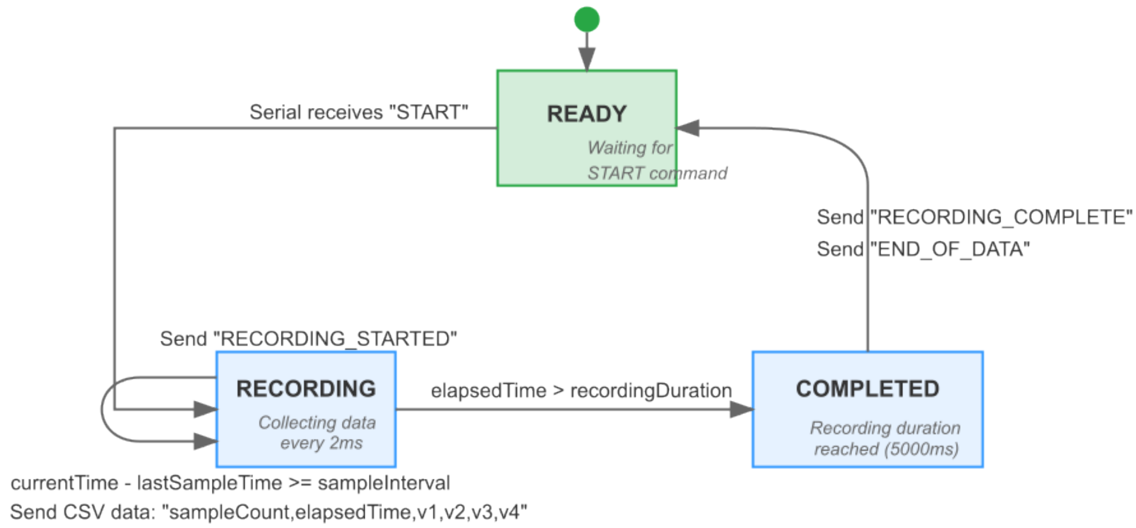

Figure 4.3.: FSM Arduino C++ LOOP

```
  if (state == recording){
    // check if within recording period
    currentTime = currentTime()
    elapsedTime = currentTime - startTime
    if(currentTime <= recordingDuration){
      // Also check not recording too fast
      if(currentTime - lastSampleTime >= minSampleInterval){
        sampleCount++;
        // Start each row with sample count and time of sample
        String currentCSVrow = String(sampleCount) , string(elapsedTime)
        // Multiplex through all analog inputs
        for (int = 0;i<4;i++){
          //read raw values
          rawValue = analogRead(analogInputs[i]);
          // compute real value
          voltage = rawValue * 5/1023
          // add value to current row to send
          currentCSVrow += String(voltage)
          }
        // Send completed row to PC
        Serial.println(currentCSVrow)
        }
      }
    }
```

```
  // end recording
  else{
    state = notRecording
    // tell PC recording finished
    Serial.println("Recording_finished")
    }
}
```

Listing 4.1: Arduino DAQ PseudoCode

Sampling Rate Details Several factors restrict the sampling rate:

Sample Interval Setting The most direct limitation is the `sampleInterval` constant set to 2ms in the code. It was meant to avoid having random sampling rates based on the number of computations required. This means samples are taken no more frequently than every 2 milliseconds (500 Hz theoretical maximum) of all four channels. The "jump" to sample the next channel is not limited in the code, but it will take 13 clock cycles, (ie. around 13µs at 1MHz ADC clock) to switch to the next channel.

ADC Prescaler Configuration The ADC prescaler is set to 16 (from the default of 128) with this line:

```
ADCSRA = (ADCSRA & 0xF8) | 0x04;
```

This increases the ADC clock to 16MHz/16 = 1MHz. With each conversion taking 13 ADC clock cycles, the theoretical maximum sampling rate is about 76.9kHz for a single channel.

Serial Transmission Overhead Each sample requires formatting and sending data over serial:

```
String dataString = String(sampleCount) + "," + String(elapsedTime);
// ... format and add voltage values ...
Serial.println(dataString);
```

This string creation and serial transmission takes some time to process as mentioned earlier.

Serial Baud Rate The code uses 115200 bps, which limits how quickly data can be transmitted. Each sample in this format might be around 30-40 bytes, which means ~3000-3800 samples/second theoretical maximum throughput.

String Operations The use of the Arduino `String` class is memory-intensive and can cause fragmentation over time, potentially causing the slowdowns noticed during testing with larger timeframes (and longer strings).

Python Script

The digitized signal must be interpreted and saved on the PC. This is done via a python script which listens to the Serial port from the arduino. The signal then also required cleaning from RF interference discovered during testing. In Figure 4.4 a flowchart is produced showing the way the script works: after initial setup that sets up the Serial Communication, the script waits for a "DAQ_READY" signal from the Arduino. Once this is received, a csv file is created and the script sends a "START" signal which the Arduino interprets and starts sampling and sending the data. The script receives each line and saves it in the new CSV, and continues to record data until the Arduino sends a "RECORDING_COMPLETE" signal - which will happen when the recordingDuration is reached. At this point the python script performs the following post-processing steps:

The filter_and_save_data() function' purpose is mainly to correctly interpret the CSV. It takes a csv with the raw voltage values, saves them into a Pandas DataFrame for easier manipulation and sends each channel to the apply_lowpass_filter() function which will be described below. filter_and_save_data() pseudocode is produced in Listing 4.2.

```
FUNCTION filter_and_save_data(filename)
// Load data from CSV file into a table structure
data_table = READ_CSV(filename)

// Streamline the data by converting text to numbers
FOR EACH column IN data_table
    CONVERT column values to numeric type
    IF conversion fails for any value
        REPLACE with NaN (Not a Number)
END FOR

// Remove any rows containing NaN values
REMOVE all rows with NaN values from data_table

// Calculate sampling frequency
time_differences = CALCULATE differences between consecutive time values
typical_time_difference = FIND median of time_differences
sampling_frequency = 1000.0 / typical_time_difference  // Convert ms to Hz

// Process each data channel
channel_list = ["A0(V)", "A1(V)", "A2(V)", "A3(V)"]
```

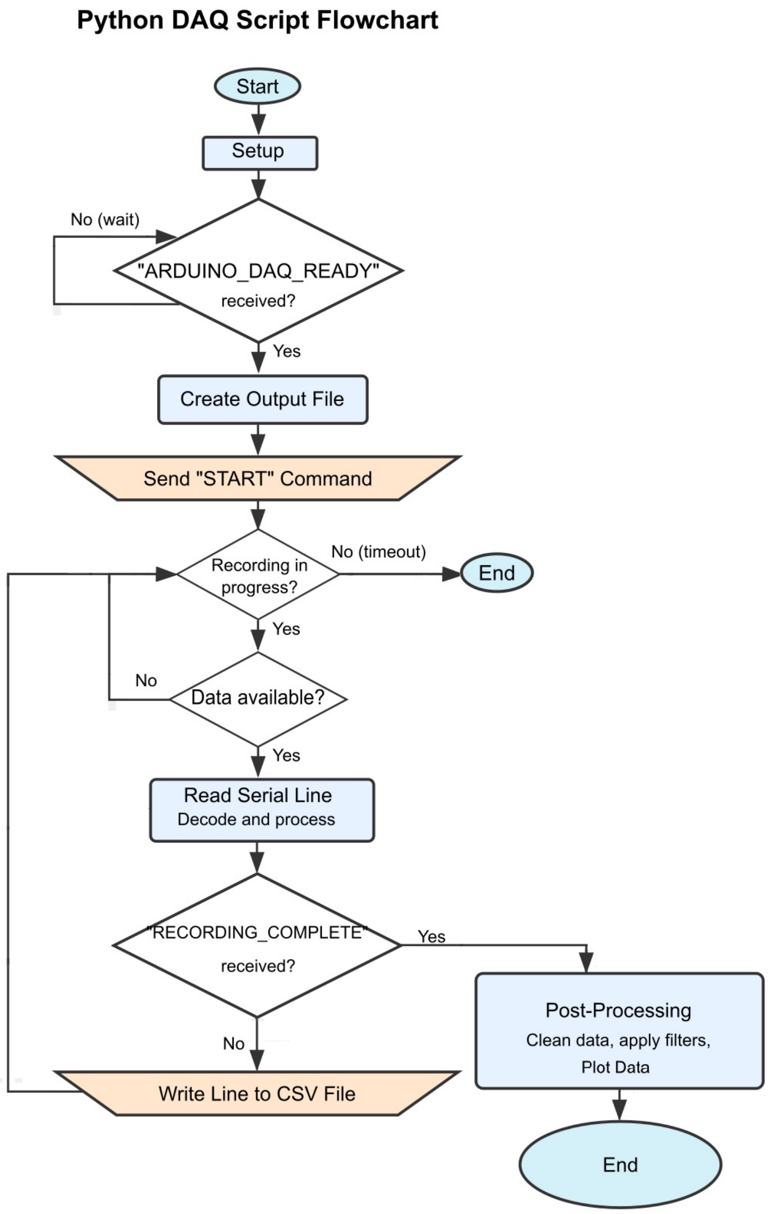

Figure 4.4.: Python Script Flowchart

```

  FOR EACH channel_name IN channel_list
      IF channel_name EXISTS in data_table
          filtered_values = APPLY_LOWPASS_FILTER(original_values,
   sampling_frequency)
          ADD new column named channel_name + "_filtered" with
   filtered_values
      END IF
  END FOR

  // Save results to new file
  new_filename = REMOVE_EXTENSION(filename) + "_filtered.csv"
  WRITE data_table TO new_filename

  RETURN new_filename
END FUNCTION
```

Listing 4.2: Python filter_and_save_data() PseudoCode

The apply_lowpass_filter() function was created and integrated into the script once it was observed that the RED testbench used was introducing noise as seen in Figure 4.1. The benefit of using an already built testbench that could place the light at exact positions repeatedly, meant it would make sense to accept the noise and just filter the data, as the signal of interest was very low frequency while the noise was around 170kHz. The filter could be relativly simple, due to the large frequency difference beteween noise and signal. A 4th order Butterworth IIR filter was deemed acceptable and a low cut-off frequency of 1Hz or 2Hz was used for the static readings. This was found to be acceptable for the test involving light location at a frequency of 0.2Hz the transition of the light from one position to another was still visibly sharp. The post-processing is not the only reason light transition would not appear instantaneous on the Voltage graph, another reason is the amplification circuit which has a 1Hz cutoff frequency via the feedback capacitor on the secondary-amplification OpAmp circuit. The pseudocode of the function that creates the low-pass digital filter and filters the data is reproduced in Listing 4.3. The butter() function from the library signal is used, from the scipy package [4] which is a free and open source library offered to the scientific comunity. Before feeding the cutoff frequency to the filter, it is normalized to the Nyquist rate, which means it is between 0 (DC) and 1 (Nyquist frequency), and therefore the filter can be applied no matter the sampling rate. As Schafer and Oppenheim explain in "Discrete-Time Signal Processing" Third Edition: "The frequency scaling or normalization in the transformation from $X_s(j\Omega)$ to $X(e^{j\omega})$ is directly a result of the time normalization in the transformation from $x_s(t)$ to $x[n]$" [2, p.171]. When creating a digital filter, this normalization becomes crucial because in the continuous-time domain, frequencies are measured in Herz and in the discrete-time domain, frequencies become relative to the sampling rate. The relationship between these two frequency domains is given by $\omega = \Omega T$, where:

- ω is the normalized digital frequency (radians/sample)
- Ω is the analog frequency (radians/second)
- T is the sampling period (seconds)

However, when implementing IIR filters like the Butterworth filter used for our purpose, the bilinear transformation is employed to convert from the continuous-time domain to the discrete-time domain. This transformation introduces frequency warping, where the relationship between the analog and digital frequencies becomes:

$$\omega = 2\arctan(\Omega T_d/2)$$

where T_d is the sampling period. This nonlinear relationship compresses the infinite analog frequency range $(-\infty, \infty)$ into the finite digital frequency range $(-\pi, \pi)$. The warping effect is more pronounced at higher frequencies, meaning that in our case it would not be noticeable [2, p.529-530].

A critical property of the bilinear transformation is stability preservation. In the analog domain, a stable system has all poles in the left half of the s-plane reproduced in Figure **??**. The bilinear transformation maps the entire left half of the s-plane to the interior of the unit circle in the z-plane. This ensures that the 4-pole Butterworth filter, which is stable in the continuous-time domain, remains stable when converted to its discrete-time equivalent.

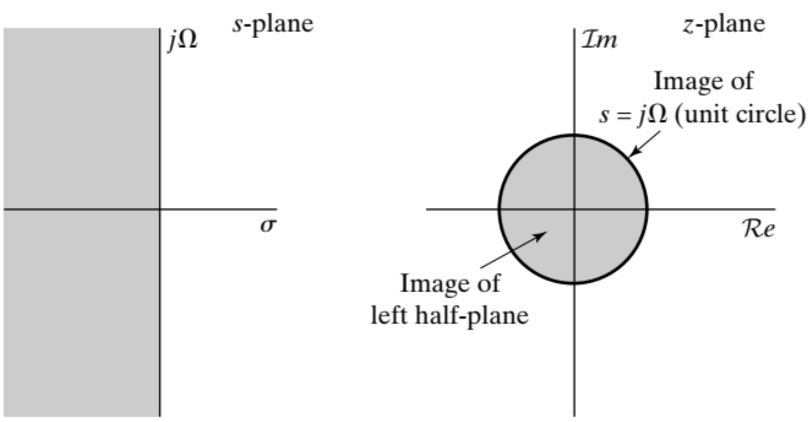

Figure 4.5.: Mapping of the s-plane onto the z -plane using the bilinear transformation [2, p.130]

The filter Frequency Response was reproduced in Figure 4.5 using the scipy.signal-.freqz() method. The actual implementation uses filtfilt() which applies the filter twice, effectively doubling the filter order [5]. This affects the transition steepness and phase response. The Discrete-Time Transfer Function is reproduced in Equation 11.

$$H(e^{j\omega}) = \frac{b[0] + b[1]e^{-j\omega} + b[2]e^{-j2\omega} + b[3]e^{-j3\omega} + b[4]e^{-j4\omega}}{a[0] + a[1]e^{-j\omega} + a[2]e^{-j2\omega} + a[3]e^{-j3\omega} + a[4]e^{-j4\omega}} \tag{11}$$

```
from scipy import signal
FUNCTION apply_lowpass_filter(data, sampling_rate)
  // set hardcoded filter values
   cutoff_freq = 2;
   filter_order = 4;
  // calculate Nyquist Frequency and Normalized cut_off
  nyquist = 0.5 * sampling_rate;
  norm_cutoff = cutoff_freq / nyquist;
  // generate numerator b and denominator a polinomials
  b, a = signal.butter(filter_order, norm_cutoff, filterType = LOW);
  // filter the data
  filtered_data = filter(b,a,data);

  return filtered_data;
```

Listing 4.3: Python apply_lowpass_filter() PseudoCode

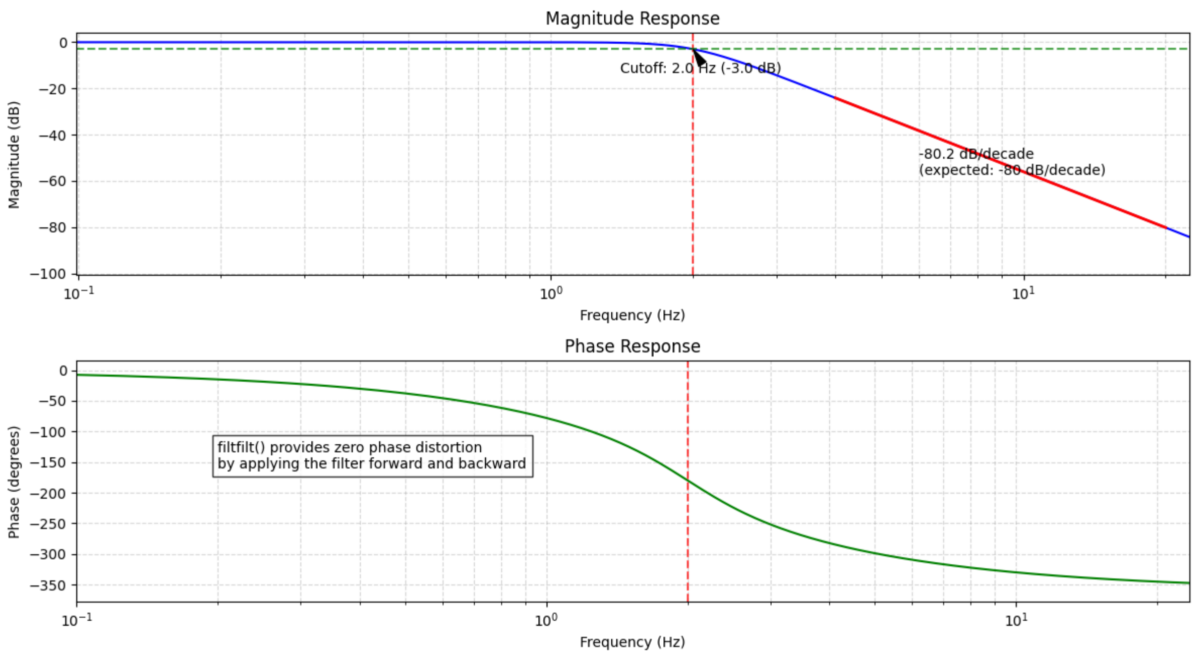

Figure 4.6.: Frequency Response of 4-pole Butterworth Low-Pass Filter (Cutoff: 2.0 Hz, Order: 4, Fs: 500 Hz)

4.2.4. Testing Strategy

The testing was done incrementally with each new version of the program, many tests were carried out as the development was dependent on both C++ code on the Arduino and the Python script on the PC to somewhat work together. At first the C++ program was tested by listening to the output in the Serial Monitor of the Arduino IDE and sending commands the same way. Debug Serial.println() lines were used to ensure the loop on the Arduino was in the correct state. Once the C++ program seemed to somewhat work as intended, the Python script was implemented to send / receive.

4.2.5. Evaluation of design

The design was tested iterartivly while making changes to the code with a signal generator at first to simulate a stead known input signal and compare readings on the Arduino IDE Serial Monitor. Once the Python script was fully working, tests were performed again for the python script's ability to read the values sent correctly. An issue was identified where sometimes the Arduino would send the column headers not at the beggining of the CSV and these lines had to be cleaned manually. This could be due to Serial buffer issues but was not investigated. Later an attempt was made to perform the cleaning programatically by creating a function in the script to do so.

4.3. Renewable Energy Demonstrator Testbench

For testing the capability of the Sun Sensor to correctly detect the location of the light source, a test bench was required that could reliably place the location of the light at a precise location repeatedly. For this purpose we used a project built by our colleagues in the European Project Semester year 2021/22 who createad just such a device intended for demonstrating renewable energy creation live [6]. Their device was able to demonstrate the energy levels created by a Photovoltaic (PV) cell by light emited at different angles. The light emission would change location based on time of day and the PV cell readings would show the difference in energy. Futher the PV cell was controllable by a joystick to point the PV Cell at the optimum angle for the highest energy capture. For our project, the arch and LED strip were used for outputing light from different angles.

Analysis of High Frequency Noise in AC-DC Power Supply

Interference structure of around 170kHz with 400mV peak-to-peak was detected on the signal being received while the RED testbench was on as shown in Figure 4.1. This noise could be generated by several factors in the AC power supply used by the RED testbench:

1. **Switching frequency harmonics** — If it's a switch-mode power supply (SMPS),

the fundamental switching frequency or its harmonics might be causing the noise. Many SMPS operate in the 50–200,kHz range.

2. **Poor filtering** — Inadequate output filtering (insufficient capacitance or poor quality capacitors) can allow switching noise to appear on the output.

3. **Improper design of magnetics** — Issues with the transformer or inductor design could cause ringing or oscillations.

4. **Resonance in the circuit** — Parasitic capacitance and inductance forming a resonant circuit at around 170,kHz.

5. **Control loop instability** — PWM controller instability can cause oscillations.

6. **Ground loops or poor PCB layout** — Improper grounding or PCB layout can create noise paths. [7]

To avoid spending time diagnosing and trying to repair the testbench, an easier solution was reached: performing digital filtering of the acquired signal in post processing. Due to the signal of interest being close to DC - frequencies much lower than 1Hz, and the noise being high frequency, around 175kHz, a simple digital Butterworth filter with a cutoff frequency at around 1-2 Herz was found to be a good solution.

The only remaining issue was that this noise would sometimes trigger the internal components of the testbench, unintentionally triggering the button press from the control interface that was changing the light position, but it happened so rare that it was not a major concern.

4.4. Software Model

4.4.1. Introduction

A Python model was constructed to provide a simulation of the movement and intersection of rays from a movable source to evaluate sensor performance and compare these results with practical experiments. The model allows for a number of configurable parameters:

- The trajectory of the light source 3D space, which moves in configurable discrete increments.
- The placement of any number of sensors and apertures, including their dimensions.
- The form of the output data, including as a static, or animated graphic.

Affording flexibility for the model to simulate any sensor topology under a variety of conditions.

4.4.2. Theory and Concept

The system is modelled in 3D space, consisting of planes and lines. Each line is defined by vectors representing position and normal direction, $\vec{A} = (a, b, c)$ and $\vec{u} = (\alpha, \beta, \gamma)$. The planes are defined by the vectors $\vec{P} = (l, m, n)$ and $\vec{n} = (\lambda, \mu, \nu)$, respectively.

Ray projection, from a source plane to a sensor plane, is modelled using the parametric equation of a 3D line (12). This allows each ray to be described in terms of a parameter t, which enables the calculation of the intersection points between the light rays and the sensor plane.

$$\frac{x-a}{\alpha} = \frac{y-b}{\beta} = \frac{z-c}{\gamma}(= t) \tag{12}$$

Where the intersection coordinates (x, y, z) occur within a target area, a hit occurs, representing illumination.

For any given combination of source plane, and sensor plane, the t parameter is calculated using the Line-Plane Intersection equation

$$t = \frac{\vec{n} \cdot \vec{P} - \vec{n} \cdot \vec{A}}{\vec{n} \cdot \vec{u}} \tag{13}$$

4.5. Material Analysis and Selection

4.5.1. Material Selection and Requirements

External Factors

Extreme Temperature Variations

Ionizing Radiation

Internal Factors

Mechanical Stresses

Outgassing and Vacuum

4.5.2. PCB Material Selection

High-Tg FR-4

Polyimide

Alumina

PTFE(Teflon)

Copper Foils

Kapton

Material Selection for PCB

4.5.3. CubeSat Chassis Material

Aluminium 6061- T651

Aluminium 7075

6061-T651 vs 7075

Emerging Materials and Future Trends

4.5.4. Quality Assurance and Reliability Standards in Space PCB Manufacturing

NASA Standards

ESA Standards

IPC Standards

US Militarty Standards(MIL-SPEC)

5. Results

5.1. Sensor Characterization

TO FINISHTO FINISHTO FINISHTO FINISHTO FINISHTO FINISHTO FINISH focus on the fundamental properties and performance of your photodiodes themselves, distinct from the other subsections. Here are some key elements that would belong specifically under SensorCharacterization:

Basic Photodiode Electrical Characteristics:

Dark current measurements Junction capacitance I-V characteristics in different lighting conditions Spectral response profiles (sensitivity vs. wavelength)

Individual Sensor Benchmarking:

Performance comparison between the 4 photodiodes (matching/differences) Responsivity measurements (A/W) Quantum efficiency calculations Detection threshold levels

Response Linearity:

Measurements showing linear range of the photodiodes Saturation point characterization Recovery time from saturation

Temperature Dependency:

Performance drift with temperature Baseline shift measurements Temperature compensation data

Aging/Stability Tests:

Long-term drift measurements Repeatability of measurements over time

This section should focus on the inherent properties of the photodiodes themselves - essentially providing the baseline characterization data that underpins all the other analysis. The other sections then build on this foundation by examining how these sensors perform when integrated into the complete system with amplification, angular positioning, enclosure effects, etc.

5.1.1. Functional Requirements

5.1.2. Design Approach

5.1.3. System Architecture

As shown in Figure the system architecture consists of various components.

5.2. Amplification Performance

This section provides results of the amplifier performance.

5.3. Photodiode Angular Response

This section discusses the results of the response of the solar sensor to angular changes of the light source.

5.4. Enclosure Effectiveness

This section discusses the effectiveness of the Photodiode enlosure.

5.5. Data Acquisition System Evaluation

This section provides results related to the Arduino DAQ.

5.6. System Performance Analysis

5.6.1. Operational Constraints Identified

5.6.2. Environmental Factors Impact

5.6.3. System Stability and Repeatability

5.6.4. Recommendations for Improvement

5.7. Comparative Analysis

This section compares the simulation with the prototype results.

5.7.1. Breadboard vs. Stripboard Results

5.7.2. Iteration Improvements Analysis

5.7.3. Performance Against Design Requirements

The performance ...

5.7.4. Design Evolution Assessment

The what now?

5.8. System Limitations And Considerations

This section discusses the limitations and future work.

5.8.1. Angle accuracy

6. Conclusions

7. Future Work

Bibliography

[1] J. Puig-Suari and C. Turner, “Development of the standard cubesat deployer and a cubesat class picosatellite,” 2001.

[2] K. S. Balaji, B. S. Anand, P. M. Reddy, V. C. B, M. D. P. Lingam, and V. K, “Studies on attitude determination and control system for 1u nanosatellite,” *ICCPCT*, pp. 616–625, 2023.

[3] I. Lopez-Calle and A. I. Franco, “Comparison of cubesat and microsat catastrophic failures in function of radiation and debris impact risk,” *Scientific Reports*, vol. 13, no. 1, -01-07 2023.

[4] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signals and Systems.* India: Pearson, 2013.

[5] Atmel, “Atmega328p 8-bit avr microcontroller with 32k bytes in-system programmable flash datasheet,” 2015. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[6] T. S. community, “butter scipy v1.15.2 manual.” [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html#scipy.signal.butter

[7] ——, “filtfilt — scipy v1.15.2 manual,” 2025. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.filtfilt.html

[8] N. I. Shopov, S. Hannisse, and S. Gupta, “Renewable energy demonstrator (red),” Glasgow Caledonian University, Tech. Rep., 2022.

[9] G. L. G. Burbui, U. Reggiani, and L. Sandrolini, “Prediction of low-frequency electromagnetic interferences from smps,” *ISEMC*, vol. 2, pp. 472–477, 2006.

A. Appendix - DAQ Code

A.1. Arduino DAQ full code

A.1.1. Arduino C++ Code

```
/*
* Reads 4 analog inputs (0-5V) for recording_dur seconds
* Streams data to PC while recording
*/
// change to true to print debug messages on Serial Monitor
// printing debug lines impacts transmission speed and messes csv
//  do not leave on during measurements!
bool debug = false;

const int analogInputs[] = {A0, A1, A2, A3};


// set up the global varialbes
unsigned long start_time;
const unsigned long recording_dur = 5000; // 25 seconds in milliseconds (ENSURE python code is greater than this)
unsigned long last_sample_time = 0;
const unsigned long min_samp_interval = 2; // Sample every 2ms (adjust for stability)
bool recording = false;
int sample_count = 0;

void setup() {
  // serial communication at 115200 bps
  Serial.begin(115200);

  // Set pins as input
  for (int i = 0; i < 4; i++) {
    pinMode(analogInputs[i], INPUT);
  }

  // Optimize ADC for faster sampling
  // Set ADC prescaler to 16 (default is 128)
  //
```

```
    // Bit: 7:enable; 6: initiate a convertion 5:
    //
    ADCSRA = (ADCSRA & 0xF8) | 0x04;

    // Wait for serial connection to establish
    delay(1000);

    // Send ready message
    Serial.println("ARDUINO_DAQ_READY");
  }

  void loop() {
    // Check if we received a command
    if (Serial.available() > 0) {
      String command = Serial.readStringUntil('\n');
      command.trim();

      if (command == "START") {
        if(debug) Serial.println("received START command");

        // Clear any remaining data in serial buffer
        while (Serial.available()) {
          Serial.read();
        }

        // Reset sample counter
        sample_count = 0;

        // Send header once
        Serial.println("Sample,Time(ms),A0(V),A1(V),A2(V),A3(V)");

        // Start recording
        recording = true;
        start_time = millis();
        last_sample_time = start_time;

        // Send confirmation
        Serial.println("RECORDING_STARTED");
      }
    }

    // If we're recording, collect and send data immediately
    if (recording) {
      if(debug) Serial.println("Recording!");
      unsigned long currentTime = millis();
      unsigned long elapsed_time = currentTime - start_time;

```

```
    // Check if we're still within the recording period
    if (elapsed_time <= recording_dur) {
      if(debug) Serial.println("elapsed time << duration");
      // Only sample at the specified interval
      if (currentTime - last_sample_time >= min_samp_interval) {
        last_sample_time = currentTime;

        // Increment sample counter
        sample_count++;

        // Start building the output string
        String data_string = String(sample_count) + "," + String(elapsed_time);

        // Multiplex through the four inputs sequentially
        for (int i = 0; i < 4; i++) {
          if(debug) Serial.println("reading input: " + String(i));
          int raw_value = analogRead(analogInputs[i]);
          float voltage = raw_value * (5.0 / 1023.0);
          data_string += "," + String(voltage, 3);
        }

        // Send the complete data string at once
        Serial.println(data_string);
      }
    }
    else {
      // End of recording
      recording = false;

      // Send notification that recording is complete
      Serial.println("RECORDING_COMPLETE");
      Serial.print("SAMPLES_COLLECTED:");
      Serial.println(sample_count);
      Serial.println("END_OF_DATA");
    }
  }
}
```

Listing A.1: C++ Code on Arduino

A.1.2. PC-side Python Serial Receive Script

```
import serial
import time
import matplotlib.pyplot as plt
import pandas as pd
```

```python
import os
import numpy as np
from scipy import signal

def apply_lowpass_filter(data, fs):
    """Apply a 4-pole low-pass Butterworth filter with 5Hz cutoff"""
    cutoff_freq = 2.0
    filter_order = 4

    nyquist = 0.5 * fs
    normal_cutoff = cutoff_freq / nyquist
    # analog=False implies  bilinear Transformation
    b, a = signal.butter(filter_order, normal_cutoff, btype='low', analog=False)
    filtered_data = signal.filtfilt(b, a, data)

    return filtered_data

# Load data from CSV, apply a 4-pole low-pass filter, and save the filtered data
def filter_and_save_data(filename):

    # Read the CSV data to pandas DataFrame
    #  It knows what are the column names
    df = pd.read_csv(filename)

    # Clean the dataframe - convert all columns to numeric
    for col in df.columns:                    # v - write NaN where it can't convert to number (in teh data, not column names)
        df[col] = pd.to_numeric(df[col], errors='coerce')

    # remove rows with NaN (not a number)
    df = df.dropna()

    # Samples not at exact distance from each other
    # Calculate the sampling frequency (median of differences)
    # numpy.diff to get the difference between samples
    time_diffs = np.diff(df['Time(ms)'])
    # numpy.median to get the median
    median_time_diff = np.median(time_diffs)  # in milliseconds
    fs = 1000.0 / median_time_diff  # Convert to Hz

    # Filter each analog channel:
    # ID column head for each channel
    analog_channels = ['A0(V)', 'A1(V)', 'A2(V)', 'A3(V)']
    # take each channel one at a time
    for channel in analog_channels:
```

```python
        # if name maches a column name
        if channel in df.columns:
            # add a new column _filtered, and send the array containing all the raw values to
            # have them filtered, and save them in the new _filtered column
            df[f"{channel}_filtered"] = apply_lowpass_filter(df[channel].values, fs)

    # Save the pandas Dataframe with filtered columns to a new CSV file
    filtered_filename = f"{os.path.splitext(filename)[0]}_filtered.csv"
    df.to_csv(filtered_filename, index=False)

    return filtered_filename

#
# Plot the DAQ data with original and filtered signals overlapped
#
def plot_data(filename):

    # Read the CSV data with pandas
    df = pd.read_csv(filename)

    # Initialize an empty list to store our analog channel names
    analog_channels = []

    # Look through all column names in the DataFrame
    for col in df.columns:
        # the column name starts with 'A'
        if col.startswith('A'):
            #  the column name ends with '(V)'
            if col.endswith('(V)'):
                # the column name does NOT contain '_filtered'
                if '_filtered' not in col:
                    # add this column name to our list
                    analog_channels.append(col)

    # Create color cycle for different channels
    colors = ['blue', 'green', 'red', 'purple']

    # Create a single plot with all channels overlapping
    plt.figure(figsize=(14, 8))

    # Plot original data (semi-transparent)
    for i, channel in enumerate(analog_channels):
```

```
        color = colors[i % len(colors)]
        plt.plot(df['Time(ms)'], df[channel], label=f'{channel} Original',
                linewidth=1.5, alpha=0.4, color=color, linestyle='-')

    # Plot filtered data (solid lines)
    for i, channel in enumerate(analog_channels):
        filtered_channel = f"{channel}_filtered"
        if filtered_channel in df.columns:
            color = colors[i % len(colors)]
            plt.plot(df['Time(ms)'], df[filtered_channel], label=f'{channel} Filtered',
                    linewidth=2.5, color=color, linestyle='-')

    # Set the y-axis range from 0 to 5V
    plt.ylim(0, 5)

    # Add labels and title
    plt.xlabel('Time (ms)')
    plt.ylabel('Voltage (V)')
    plt.title('Arduino DAQ - 4-Channel Readings with 4-Pole 5Hz Low-Pass Filter')
    plt.legend()
    plt.grid(True)

    # Add data summary
    duration = df['Time(ms)'].max() - df['Time(ms)'].min()
    sample_count = len(df)
    sample_rate = sample_count/(duration/1000) if duration > 0 else 0

    info_text = f"Data summary:\n" \
                f"Duration: {duration:.1f} ms\n" \
                f"Samples: {sample_count}\n" \
                f"Sample rate: {sample_rate:.1f} Hz\n" \
                f"Filter: 4-pole Butterworth, 5Hz cutoff"

    plt.figtext(0.02, 0.02, info_text, fontsize=10,
                bbox=dict(facecolor='white', alpha=0.8))

    # Save the plot
    plot_filename = f"{os.path.splitext(filename)[0]}_plot.png"
    plt.savefig(plot_filename, dpi=300, bbox_inches='tight')

    # Show the plot
    plt.tight_layout()
```

```
    plt.show()

def main():

    # ASk if to plot or measure
    what_to_do = int(input("Record new measurement (1) or plot existing (2): "))

    # User selected to plot old csv
    if(what_to_do == 2):
        plot_data(input("Insert the name of the csv: "))


    # User selected to record new measurement
    elif(what_to_do == 1):
        # Use a default port (COM3 for Windows, modify as needed)
        default_port = "COM3"  # Change to match your system

        print(f"Using  port: {default_port}")

        # Configure serial port
        try:
            ser = serial.Serial(default_port, 115200, timeout=2)
            print("Connected to Arduino!")
        except serial.SerialException:
            print(f"Error: Could not open port {default_port}")
            print("Please modify the default_port variable in the script.")
            return

        time.sleep(2)  # Wait for Arduino to reset

        # Flush buffers
        ser.reset_input_buffer()
        ser.reset_output_buffer()

        # Wait for Arduino ready
        print("Waiting for Arduino to be ready...")
        ready = False
        timeout = time.time() + 10 # don't wait too long

        while not ready and time.time() < timeout:
            line = ser.readline().decode('utf-8', errors='ignore').strip()
            if line == "ARDUINO_DAQ_READY":
                ready = True
                print("Arduino is ready!")
```

```python

        if not ready:
            print("Timed out waiting for Arduino. Make sure it's properly connected.")
            ser.close()
            return

        # Create a filename for this recording session
        filename = f"arduino_daq_data_{time.strftime('%Y%m%d_%H%M%S')}.csv"

        print(f"Starting data recording to {filename}...")
        print("Press Ctrl+C to stop if needed.")

        with open(filename, 'w', newline='') as file:
            # Send start command
            ser.write(b"START\n")

            recording = True
            data_lines = 0

            # Start time for timeout
            start_time = time.time()
            timeout_duration = 15  # seconds

            while recording and (time.time() - start_time) < timeout_duration:
                if ser.in_waiting:
                    line = ser.readline().decode('utf-8', errors='ignore').strip()

                    if "RECORDING_COMPLETE" in line:
                        recording = False
                        print("Recording complete!")
                    elif "END_OF_DATA" in line:
                        pass
                    elif line:
                        # Write the line to the file
                        file.write(line + '\n')
                        data_lines += 1

                        # Show progress occasionally
                        if data_lines % 100 == 0:
                            print(f"Recorded {data_lines} data points...")

        # Close the serial port
```

```
            if ser.is_open:
                ser.close()
                print("Serial port closed.")

            print(f"Recorded {data_lines} lines of data.")

            # Process the data
            print("Applying filters to data...")
            filtered_filename = filter_and_save_data(filename)
            print(f"Filtered data saved to {filtered_filename}")

            print("Generating plot...")
            plot_data(filtered_filename)
            print("Done!")

        else:
            print("Wrong Choice. Goodbye!")
            exit()
    if __name__ == "__main__":
        try:
            main()
        except KeyboardInterrupt:
            print("\nProgram terminated by user.")
```

Listing A.2: Python Serial Receive Script

B. Appendix - Software Model Code

B.1. Section 1 Title

B.1.1. subsectiontitle

B.2. Section 2 Title

B.2.1. subsectiontitle

C. Further Appendix

C.1. Section 1 Title

C.1.1. subsectiontitle

C.2. Section 2 Title

C.2.1. subsectiontitle