DESIGN AND DEVELOPMENT OF OPEN-LOOP SUN TRACKING INSTRUMENT

This research project is accomplished, with affiliation to

Physical Research Laboratory

By,

Yogeshkumar Patel

In the partial fulfillment of the requirements for the degree

Masters of Science in Physics

(St. Xavier's College, Ahmedabad)

Under the guidance of

Dr. Harish Gadhavi

Space and Atmospheric Sciences Division

Physical Research Laboratory, Navrangpura Campus– Ahmedabad

Jan 22' - Apr 23'

Declaration

I hereby declare that this written submission represents my own work in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above can cause disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Yogeshkumar A. Patel

Date: _____

Acknowledgments

I am very thankful to my mentor Dr. Harish Gadhavi, scientist at Physical Research Laboratory, Navrangpura - Ahmedabad for encouraging me greatly in my learning phase and guiding me for the project.

I am very grateful to Mr. Malaidevan, Physical Research Laboratory, Navrangpura- Ahmedabad for always helping me solve my queries and motivating me to keep moving forward.

I would also like to thank Dr. Rajesh Iyer, Head of the Department of Physics-Electronics, St. Xavier's College, and Mr. Tejas Turakhia for always encouraging me and providing support whenever required for this project.

I would like to thank Parth Dineshkumar Patel for his support in the related software-related guidance.

I am also very thankful to my parents who have always believed in me throughout this journey and have always provided me their great support and encouragement which has constantly pushed me further to cross my limits and to give my best to achieve new things.

Abstract

Aerosol optical depth (AOD) is an important measure of the amount of aerosols, such as dust, smoke, and pollution, in the Earth's atmosphere. It is defined as the amount of light that is absorbed or scattered by aerosols in the atmosphere, relative to the amount of light that would be transmitted through a clear atmosphere. AOD has several important implications for climate and environmental studies. Sun-photometer is required to estimate AOD which in turn requires it to be pointed toward Sun. We have the sun photometer which can be pointed toward Sun manually but for automated operation and to make observations from a platform such as a balloon or ship, an automated sun-tracker is required. We have tried to make a modular platform for the sun tracker which can take the readings automatically. For this purpose we used microcontroller boards **Raspberry pi pico** as well as **Arduino UNO** and for the programming purpose we used two different languages first is **C++** and second is **Micropython**. The sun-tracker included sensors and devices such as Li-Ion cells, Buck converter, GPS, Servo motors, Accelerometer modules, etc. We aim to make a sun tracker for Sun-photometer which is having a field of view of 2.5º.

Table of Contents

Introduction

An essential metric for measuring how much an atmospheric aerosol affects weather, climate, and quality of air is called aerosol optical depth (AOD). The ratio of the amount of solar radiation scattered and absorbed by aerosol particles to the amount of solar radiation that would be scattered and absorbed if the atmosphere were entirely clean is known as the aerosol optical depth, or AOD [1]. A sun tracker instrument is necessary to correctly measure AOD. Sun trackers are gadgets that measure the quantity of direct and dispersed sunlight reaching the ground by tracking the sun's position in the sky throughout the day. Sun trackers are equipped with a solar sensor that monitors the strength of solar radiation and a motorized mount that rotates the sensor to keep it aimed at the sun [2]. Suntracker data can be utilized to calculate the AOD by monitoring the difference between direct and dispersed solar radiation [3]. Sun trackers are widely utilized in a variety of applications, including air quality monitoring, climate research, and weather forecasting [4].

AOD is a critical measure for determining the effect of atmospheric aerosols on climate change, weather patterns, and human health. Aerosols are microscopic particles floating in the atmosphere that can be solid or liquid and can come from both natural and man-made sources. By reflecting or absorbing sunlight, these particles can have a major influence on the Earth's energy balance, changing the quantity of solar radiation reaching the Earth's surface [5]. The quantity of solar radiation scattered and absorbed by aerosol particles is computed as the ratio of the amount of solar radiation scattered and absorbed if the atmosphere were fully clean [1]. The higher the AOD value, the more aerosols there are in the atmosphere and the greater their influence on climate change and air quality. AOD is commonly measured at various wavelengths using special devices such as sunphotometers or lidars [6]. Sun trackers are essential for monitoring AOD because they offer reliable readings of direct and dispersed sunlight throughout the day. Sun trackers come in a variety of configurations, including single- and dual-axis models, and are used to follow the sun's passage across the sky [2]. A single-axis sun tracker tracks the sun's daily movement from east to west, whereas a dual-axis sun tracker tracks the sun's elevation angle [7]. Because the amount of dispersed light in the atmosphere fluctuates with the location of the sun and changes during the day, this sort of equipment is required [8]. AOD is measured using a variety of approaches, including satellite-based observations, ground-

based sunphotometers, and lidars [9]. Each strategy has advantages and limitations, and the method used is determined by the unique application and data needs [10]. Sun trackers are crucial for reliable ground-based AOD measurements, especially in densely populated regions with high levels of air pollution, where the amount of dispersed light can be significantly greater than in rural settings [11]. Sun trackers are used for a number of purposes, such as air quality monitoring, climate research, and weather forecasting. AOD measurements can assist identify the origins of air pollution, estimating the impact of wildfires, and tracing the movement of dust and smoke plumes in air quality monitoring [12]. AOD observations may be utilized in climate research to investigate the influence of aerosols on the Earth's energy balance, cloud formation, and precipitation patterns [13]. AOD measurements in weather forecasting can assist enhance the accuracy of short-term weather predictions as well as identify the presence of fog or haze, which can impair aviation safety [14].

Finally, Aerosol Optical Depth is an important measure for determining the influence of atmospheric aerosols on air quality, climate, and weather patterns. Sun trackers, which follow the sun's position in the sky and measure the quantity of direct and dispersed sunlight reaching the ground, are required for reliable AOD readings. Sun trackers are widely utilized in a variety of applications, including air quality monitoring, climate research, and weather forecasting. Further study is needed to better understand the influence of atmospheric aerosols on climate change and human health, as well as to create novel AOD measurement technology.

Theory

Greenwich Mean Time is the yearly average (or 'mean') of the time each day when the Sun crosses the Prime Meridian at the Royal Observatory Greenwich. So from that we can know about the universal time or GMT (as there is no difference between that).

$$greenwichtime = hour - timezone + \left(\frac{minute}{60}\right) + \left(\frac{second}{3600}\right)$$

From that we got GMT or UT the second quantity is day number here we calculate according to below equation

$$daynum = 367 * year - \frac{7 * year + \frac{month + 9}{12}}{4} + 275 * \left(\frac{month}{9}\right) + day - 730531.5 + \frac{greenwichtime}{24}$$

here day number is calculated according to the above equation. The term (M+9)/12 in the formula for the Julian Day Number is used to adjust the calculation for the fact that the year in the Julian calendar starts on March 25th, not January 1st as it does in the Gregorian calendar which is the calendar system in use today.

The factor of 275*month/9 is used to account for the varying rate at which the Sun moves along the ecliptic throughout the year. Here 730531.5 is the day number of 1st January, 2000 (J2000) for the reference of the J2000.

Latitudes are horizontal lines that measure the distance north or south of the equator. Longitudes are vertical lines that measure east or west of the meridian in Greenwich, England. We can estimate the sun's mean longitude it means the geocentric celestial longitude which the sun would have if its apparent annual motion in the ecliptic were at a uniform average or mean angular velocity.

$$meanlongitude = daynum * 0.01720279239 + 4.894967873$$

mean longitude of the sun can be get by this equation here 4.894967873 represent the longitude at the J2000 and 0.01720279239 represents the mean daily motion of the sun along ecliptic.

In celestial mechanics, the mean anomaly is a parameter that describes the position of a planet or other celestial object in its orbit around the Sun. It is defined as the angle between the perihelion (the closest point in the orbit to the

Sun) and the current position of the object, measured concerning the center of the elliptical orbit.

$$meananomaly = daynum * 0.01720197034 + 6.240040768$$

Here 0.01720197034 is the term that represents the daily motion of the sun in its orbit in radiance. Here 6.2400407680 is the corresponding anomaly to j2000.

The sun's ecliptic longitude is defined as the angle subtended at the earth between the vernal equinox and the sun.

$$\begin{aligned} &Eclipticlongitude \\ &= meanlongitude + 0.03342305518 * sin(meananomaly) \\ &+ 0.0003490658504 * sin\,(2 * meananomaly) \end{aligned}$$

Here 0.03342305518 represents the longitude of the ascending node of the lunar orbit on January 1, 2000.

the obliquity of the ecliptic is the tilt of the earth's axis of spin as measured from the plane of the ecliptic this angle is about 0.409 radiance and it changes every day due to various factors such as gravitational interactions with other celestial bodies and the precession of the Earth's axis.

$$Obliquity = 0.409877234 - 6.981317008 * 10^{-9} * daynum$$

The right ascension of a celestial object is a measure of its position in the sky relative to the vernal equinox, which is the point on the celestial sphere where the ecliptic intersects the celestial equator at the beginning of spring in the Northern Hemisphere.

$$ascension = tan^{-1}\left(cos\frac{(Obliquity) * sin(Eclipticlongitude)}{cos(Eclipticlongitude)}\right)$$

Declination is an astronomical term that refers to the angular distance of a celestial body north or south of the celestial equator. The celestial equator is an imaginary line on the celestial sphere that represents the projection of the Earth's equator into space. Declination is measured in degrees, minutes, and seconds of arc, and it is used to locate celestial objects in the sky relative to the celestial equator.

$$Declanation = sin^{-1}\big(sin(Obliquity) * sin(Eclipticlongitude)\big)$$

Sidereal time is a system of timekeeping used by astronomers to measure the position of celestial objects in the sky. It is based on the rotation of the Earth on its axis relative to the stars, rather than relative to the Sun as in solar time.

$$Hourangle = Siderealtime - ascension$$

Coordinates of the sun can be given by the following equations

$$Elevation = sin^{-1}\big(sin(Declanation) * sin(observer'slatitude) + cos(Declanation) * cos(observer'slatitude) * cos(Hourangle)\big)$$

$$Azimuth = tan^{-1}\frac{\big(-cos(Declanation) * cos(Observerslatitude) * sin(Hourangle)\big)}{\big(sin(Declanation) - sin(Observer'slatitude) * sin(Elevation)\big)}$$

Here the values of Azimuth and Elevation are in radiance.

Instrumentation

To track the sun using an open loop algorithm, one needs the date, time and coordinates of the device Which we aimed to obtain using GPS. We used micro-controller board named Raspberry Pi Pico 2040 which works on Firmware MicroPython v1.19.1 to process the GPS data.

The Raspberry Pi Pico 2040 which is a microcontroller board designed by the Raspberry Pi Foundation, is based on the RP2040 microcontroller, which is a dual-core Arm Cortex-M0+ processor with 264KB of RAM. The board also features 26 GPIO pins, including 3 analog input pins, a programmable LED, and various other features.

The 26 GPIO pins on the Pico 2040 are labeled GP0 to GP25 and can be used for digital input or output. These pins can be configured as either INPUT or OUTPUT and can also be used for PWM, interrupts, and other functions. Additionally, three of these pins (GP26, GP27, and GP28) can be used for analog input, allowing the Pico 2040 to read analog voltage levels between 0 and 3.3 volts. The analog voltage input is converted into a digital value using a 12-bit ADC built into the RP2040 microcontroller.

The Pico 2040 features a clock frequency of 133 MHz, which is significantly faster than the 16 MHz clock frequency of the Arduino Uno. This means that the Pico 2040 can perform operations and execute instructions much more quickly than the Uno. However, the higher clock frequency also means that the Pico 2040 may consume more power and generate more heat than the Uno.

The Pico 2040 also includes a variety of other features, including a programmable LED connected to GP25, a reset button, and various power and ground pins. The board can be powered by a USB-C connector, and it can also be programmed and powered through its SWD (Serial Wire Debugging) port.

One of the unique features of the Pico 2040 is its support for MicroPython and CircuitPython, which are high-level programming languages designed for microcontrollers. These languages allow developers to write code in Python, which is a popular and easy-to-learn programming language. The Pico 2040 also supports C and C++ programming languages, making it a versatile platform for a wide range of projects.

Here's a brief overview of the pins on the Raspberry Pi Pico:

There are 26 GPIO pins in total, labeled GPIO0 to GPIO25, that can be configured for various purposes, such as digital input/output, analog input, and PWM.

There are three analog pins on the Raspberry Pi Pico: GP26, GP27, and GP28. These pins can be used as analog inputs to read voltages from sensors and other analog devices.

UART is a serial communication protocol that allows two devices to communicate with each other. The Raspberry Pi Pico has two UART interfaces: UART0 and UART1. The pins for UART0 are GP0 (TX) and GP1 (RX), while the pins for UART1 are GP4 (TX) and GP5 (RX).

I2C is a serial communication protocol that allows multiple devices to communicate with each other using only two wires. The Raspberry Pi Pico has two I2C interfaces: I2C0 and I2C1. The pins for I2C0 are GP8 (SDA) and GP9 (SCL), while the pins for I2C1 are GP16 (SDA) and GP17 (SCL).

SPI is a serial communication protocol that allows high-speed data transfer between devices. The Raspberry Pi Pico has two SPI interfaces: SPI0 and SPI1. The pins for SPI0 are GP10 (MOSI), GP11 (MISO), and GP12 (SCK), while the pins for SPI1 are GP19 (MOSI), GP20 (MISO), and GP21 (SCK).

There are several other pins on the Raspberry Pi Pico that can be used for various purposes, including PWM, interrupt handling, and more. These pins include GP2, GP3, GP6, GP7, GP13 to GP15, GP18, GP22 to GP25, and RUN.

The temporal resolution of the Raspberry Pi Pico 2040 depends on the specific application and how it is programmed. The RP2040 microcontroller has a built-in timer system that can be configured to generate interrupts at specific intervals. The timer system can be used to measure time intervals, generate pulses, or trigger events based on specific time conditions.

The Pico 2040's clock frequency of 133 MHz allows for very precise timing, with a maximum resolution of approximately 7.5 nanoseconds per clock cycle. This means that the microcontroller can perform many operations and execute many instructions in a very short amount of time, allowing for high-speed applications.

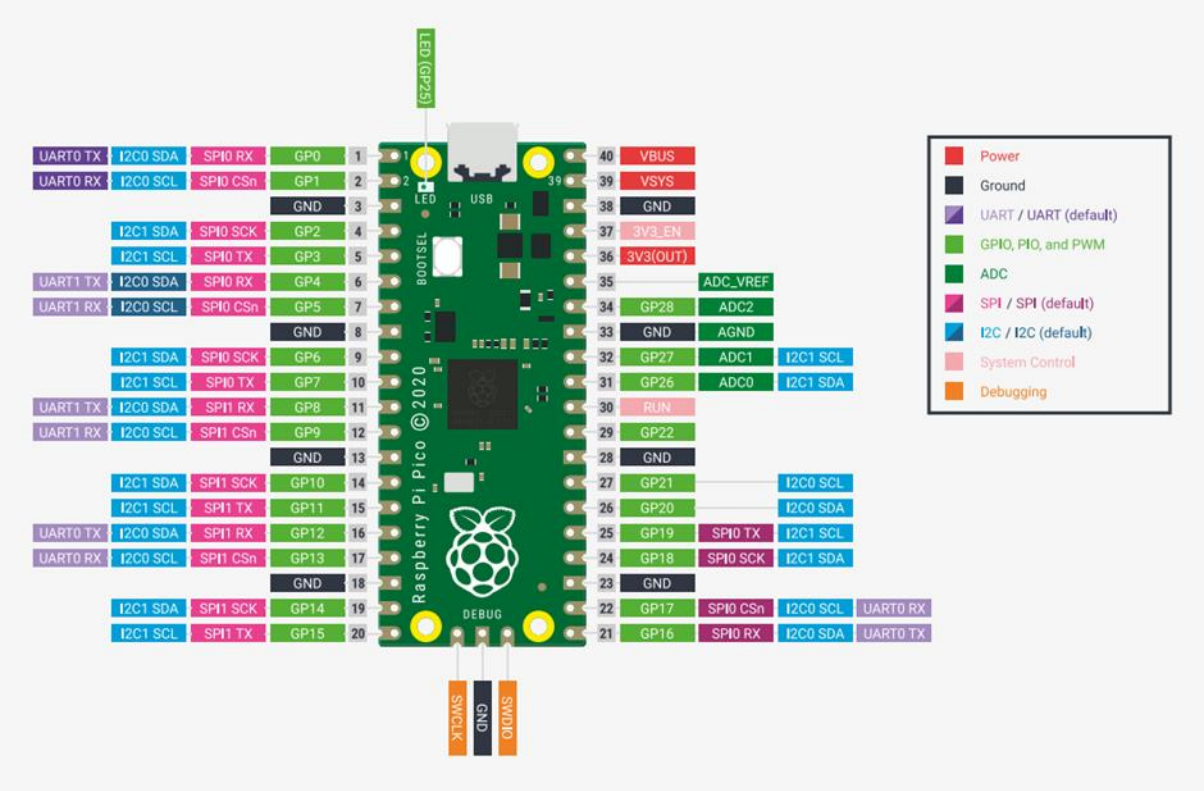


Fig 1 :- Pin diagram of raspberry pi pico 2040

However, the temporal resolution of the Pico 2040 is ultimately determined by the programming language and algorithms used to control the microcontroller. For example, if the microcontroller is programmed to read a sensor value at a specific rate, the temporal resolution will depend on how accurately the timer system can be configured to trigger the sensor readings.

In general, the Pico 2040 is capable of high-speed and high-precision timing, making it well-suited for applications that require accurate timing or fast data acquisition. However, achieving optimal temporal resolution requires careful programming and configuration of the microcontroller's timer system.

GPS, or Global Positioning System, is a satellite-based navigation system that provides location and time information anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

The GPS system consists of a network of at least 24 satellites orbiting the Earth, as well as ground control stations and GPS receivers. Each satellite broadcasts a signal containing information about its location and the current time. GPS receivers on the ground receive these signals and use the information to calculate their location and time.

To determine its location, a GPS receiver compares the time the signals were transmitted by the satellites with the time the signals were received. The time

difference between when the signal was transmitted and when it was received gives the GPS receiver an estimate of the distance to each satellite. By combining the distances to at least four satellites, the GPS receiver can then calculate its location using a process called trilateration.

The GPS system also uses a technique called "differential GPS" to improve accuracy. This involves comparing the GPS signal from a stationary receiver at a known location with the GPS signal from a mobile receiver. The difference in the two signals can be used to correct errors in the GPS signal caused by atmospheric effects and other factors.

Overall, GPS is an incredibly useful technology that has a wide range of applications, from navigation and surveying to tracking and monitoring.

The GPS NEO 6M is a common GPS module used in many applications, such as drones, robotics, and navigation systems. The NEO 6M module works by receiving signals from GPS satellites and processing them to determine the user's location.

The NEO 6M module has a built-in receiver chip that is capable of receiving signals from up to 22 GPS satellites simultaneously. The module also includes an onboard microcontroller that processes the satellite data and provides a variety of output formats, including NMEA, binary, and u-blox.

The NEO 6M module communicates with a microcontroller or other host device using a serial interface and provides information such as latitude, longitude, altitude, speed, and time. The module also includes a backup battery that allows it to retain satellite data and provide faster location fixes upon power-up.

The NEO 6M module is known for its low power consumption, making it suitable for use in battery-powered applications. It is also relatively easy to use, with a simple command set that allows users to configure the module for their specific application. Overall, the NEO 6M is a popular and reliable GPS module that is widely used in many different types of projects.

Here we got the position of the sun but that data will be inside the Raspberry Pi Pico for transmitting this data to the Arduino Uno we need to establish communication between both of them here are the details about Arduino Uno.

Arduino Uno is a popular open-source microcontroller board designed for easy prototyping of electronic projects. It is based on the ATmega328P microcontroller and has 14 digital input/output pins, 6 analog inputs, and a 16 MHz quartz crystal oscillator.

Here's a breakdown of the pins on the Arduino Uno:

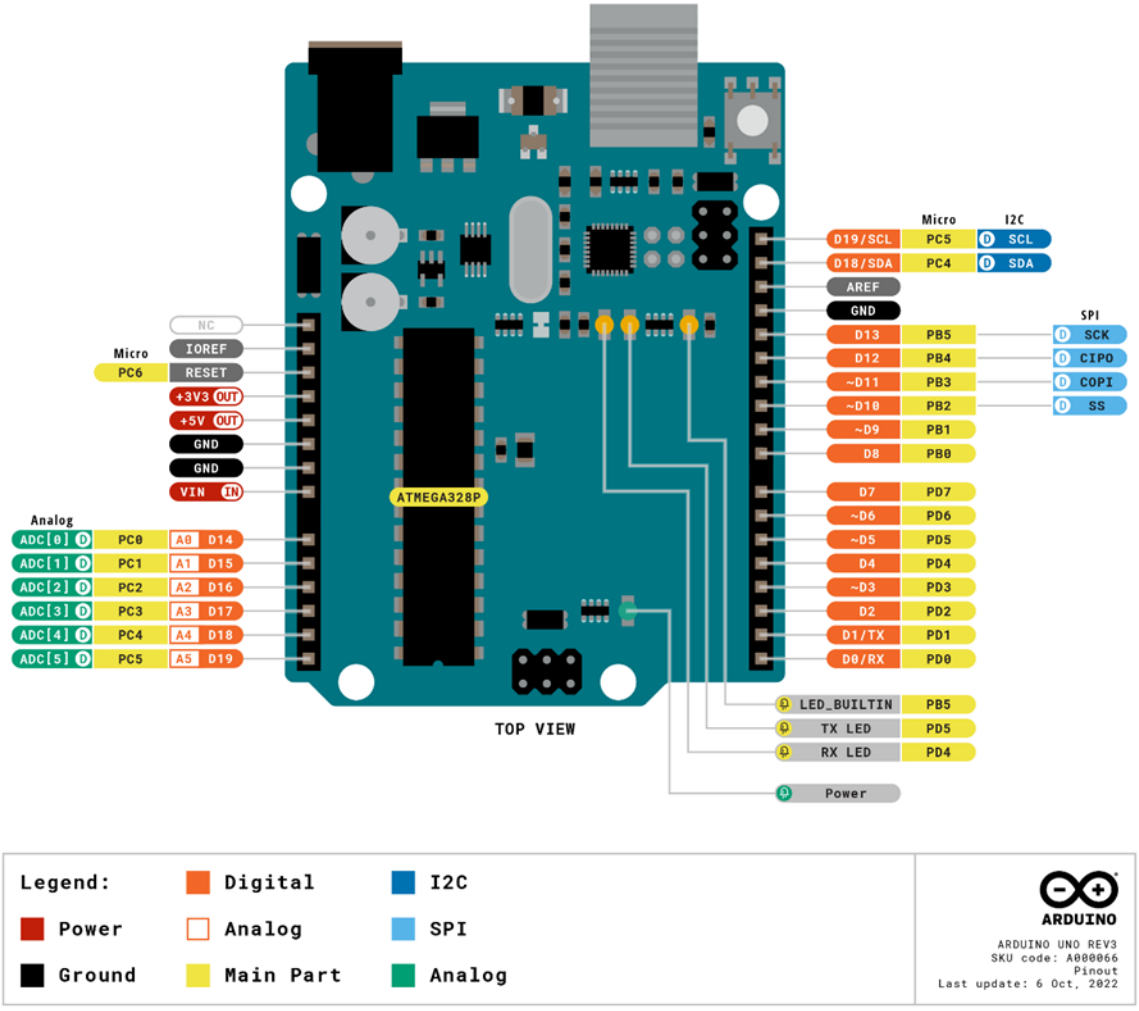


Fig 2 :- Pin diagram of Arduino Uno

Digital pins (D0-D13): These are general-purpose input/output (GPIO) pins that can be used for digital input or output. They can be configured as either INPUT or OUTPUT and can also be used for PWM (pulse width modulation) and interrupts. D0 and D1 are also used for serial communication (RX and TX, respectively).

Analog pins (A0-A5): These are analog input pins that can read analog voltage levels between 0 and 5 volts. They can also be used as digital input/output pins (A0 is equivalent to D14, A1 is equivalent to D15, and so on).

Power pins: The Arduino Uno has several power pins, including a 5V pin that can provide up to 500mA of current, a 3.3V pin that can provide up to 50mA, and a Vin pin that can be used to power the board using an external power source (6-20V DC).

Ground pins: The Arduino Uno has several ground pins that are connected to the board's ground plane.

Reset pin: This is a special pin that can be used to reset the microcontroller.

ICSP header: This header provides access to the ATmega328P's SPI (Serial Peripheral Interface) pins, which can be used for programming and debugging the microcontroller.

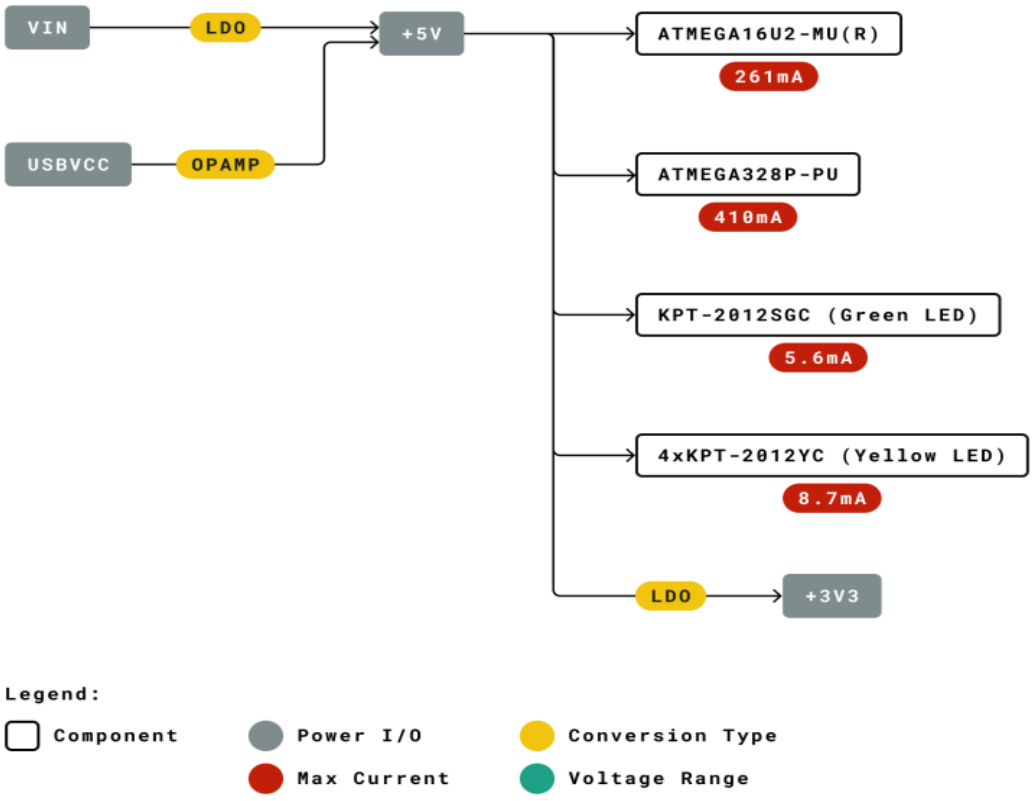


Fig 3 :- Power distribution in Arduino Uno

AREF pin: This pin can be used to set the analog reference voltage used by the analog inputs.

In addition to the above pins, the Arduino Uno also has a built-in LED connected to pin 13, which can be used for debugging and simple output. The board also has a USB connector that can be used for programming and power, as well as a barrel jack connector for external power.

Analog pins on the Arduino Uno are input pins that can read analog voltage levels between 0 and 5 volts. These pins are labeled A0 to A5 and can also be used as digital input/output pins, with A0 equivalent to D14, A1 to D15, and so on.

Analog pins on the Arduino Uno work by converting an analog voltage input into a digital value. This is done using an analog-to-digital converter (ADC) built into

the microcontroller. The ADC samples the voltage at the analog pin and converts it into a digital value, which can then be read by the microcontroller.

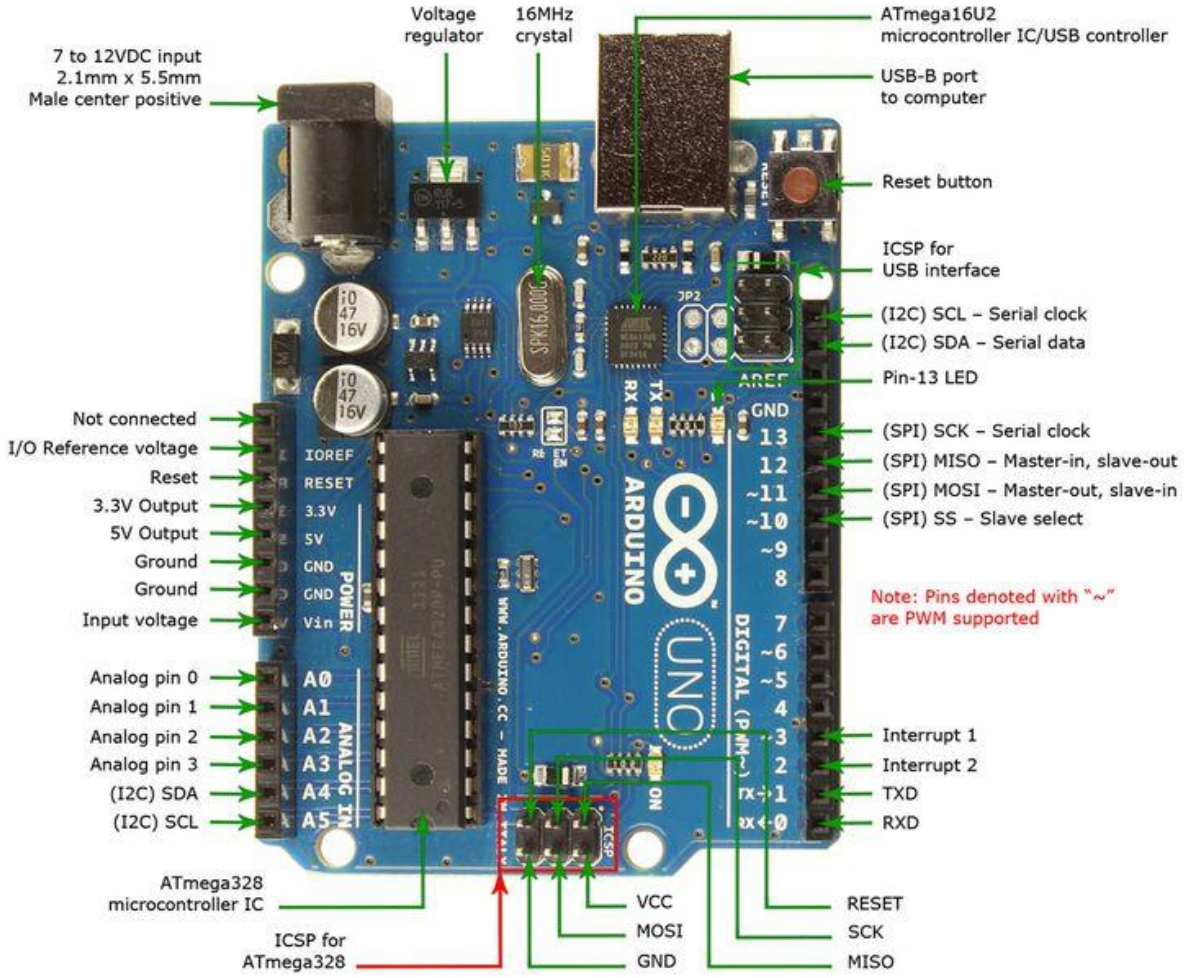


Fig 4 :- Pin diagram of Arduino Uno

The resolution of the ADC on the Arduino Uno is 10 bits, which means it can represent analog values from 0 to 5 volts with a resolution of 1024 (2^10). This resolution determines the smallest change in voltage that can be detected by the analog pins. For example, if the voltage range is divided into 1024 steps, each step represents a change of 0.00488 volts (5V / 1024).

To use the analog pins on the Arduino Uno, you can use the analogRead() function in the Arduino IDE. This function reads the analog value from the specified analog pin and returns a value between 0 and 1023. The value returned by analogRead() can be mapped to the desired range of values using the map() function.

Analog pins can be used for a variety of applications, such as reading sensors that output analog voltages, or controlling analog devices such as motors and lights with PWM.

Clock frequency, also known as clock speed, refers to the rate at which a processor or microcontroller's clock generates pulses. These pulses are used to synchronize the operation of various components within the processor or microcontroller.

In the case of the Arduino Uno, the clock frequency is 16 MHz, which means the clock generates 16 million pulses per second. This clock frequency determines how quickly the microcontroller can perform operations and execute instructions.

A higher clock frequency generally means that the microcontroller can perform operations more quickly and execute more instructions per second. However, a higher clock frequency also means that the microcontroller may consume more power and generate more heat.

In some cases, it may be necessary to adjust the clock frequency of the microcontroller to optimize performance or reduce power consumption. This can be done using various techniques, such as changing the clock divider settings or using an external oscillator. However, modifying the clock frequency can also affect the operation of the microcontroller, so it should only be done with caution and with a thorough understanding of the system.

Overall, the clock frequency is an important parameter to consider when designing and programming systems based on microcontrollers like the Arduino Uno. It can have a significant impact on the performance and power consumption of the system and should be carefully considered during the design process.

Here Arduino works with one main sensor module named mpu-6050 here other options are also available like bmx 160, Adxl 345 .

Here Adxl345 sensor’s stability analysis is shown in the graph as shown in the graph it is stable with the time as well as on board low pass filter is there that’s why the filtered output is much more smoother than the basic one.

The second sensor is mpu6050 which is 3 axis gyroscope and 3-axis accelerometer sensor module this module contains a complimentary filter (DMP) inside a module which give us the upper hand in the digital motion process than

Adxl345.

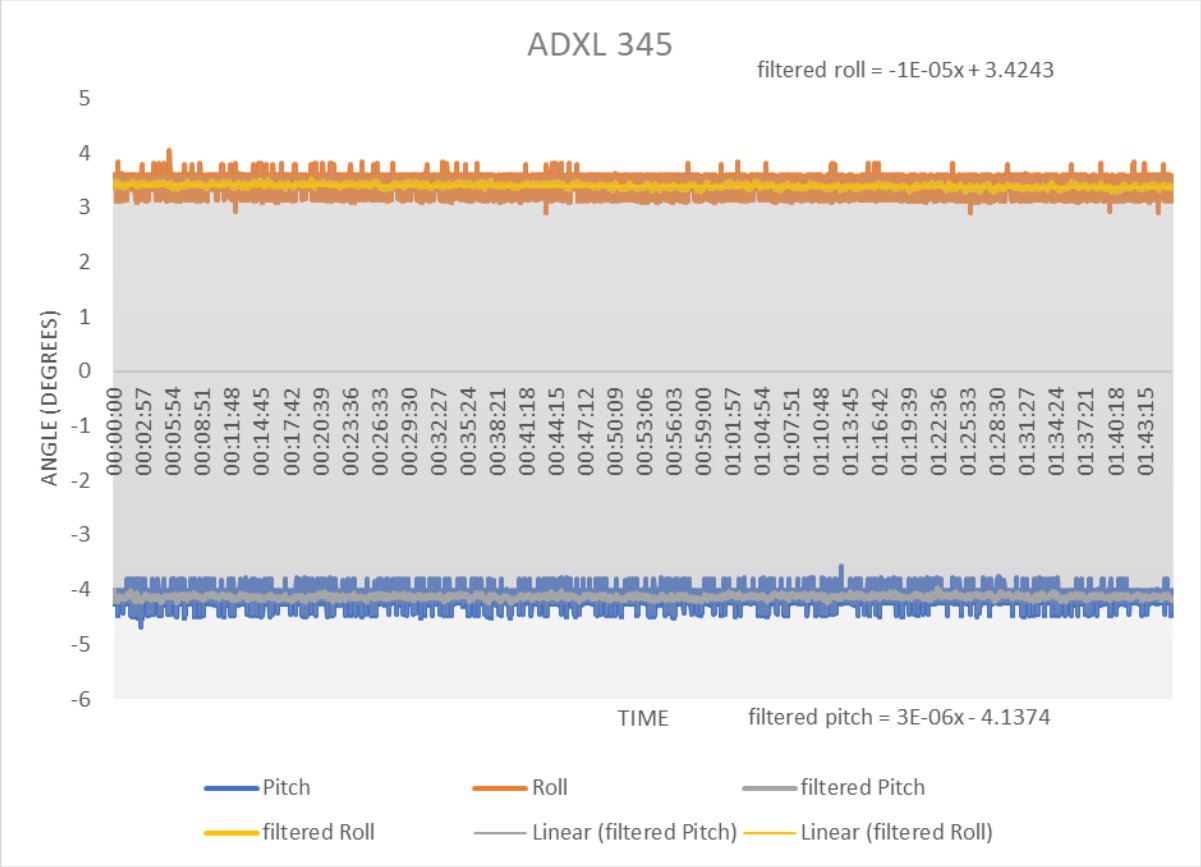
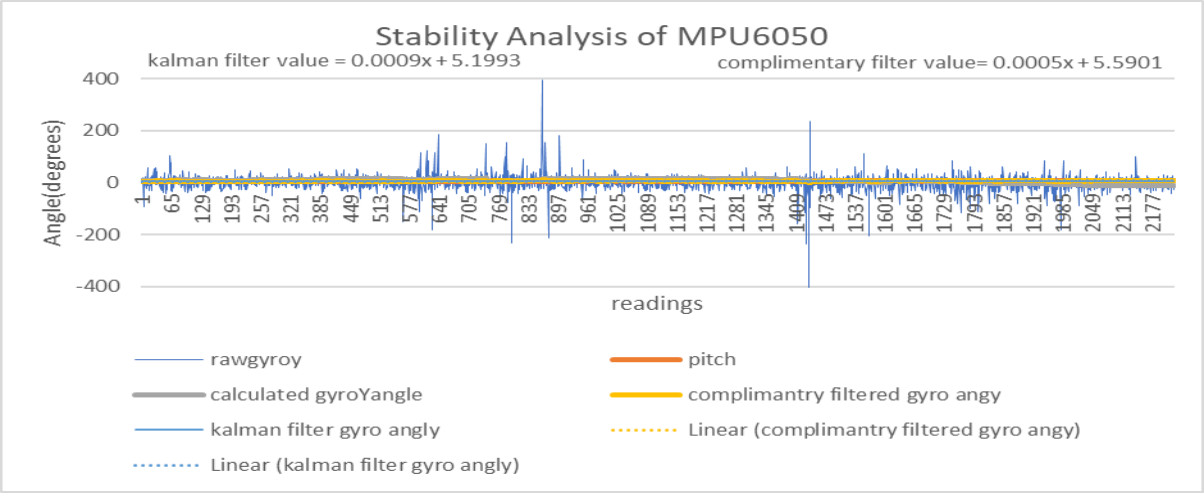


Fig 5 :- Stability analysis Of Adxl345 Sensor module

Here we took readings every second so we got total of 2224 readings for stability analysis over the constant angle here with the use of complementary filter we got a slope of 0.0001 which can be neglected with the use of the Kalman filter our slope will be more accurate approximately 0.00004 which is good but it comes with the price of higher computation power requirement as well as it needs more complex codes also which will end up with more complexity over the system here raw data from the sensor but complimentary filtered values are nearly same and nearly unchanged over a time same for y-axis also which makes it perfect sensor to get the angle values. both graphs are shown in the figure

Here to use this angle we used Servo motors which take inputs as a PWM(Pulse Width Modulus) which can be done by using Raspberry Pi Pico or Arduino Uno

here we used Uno full circuit diagram of the system as given below.



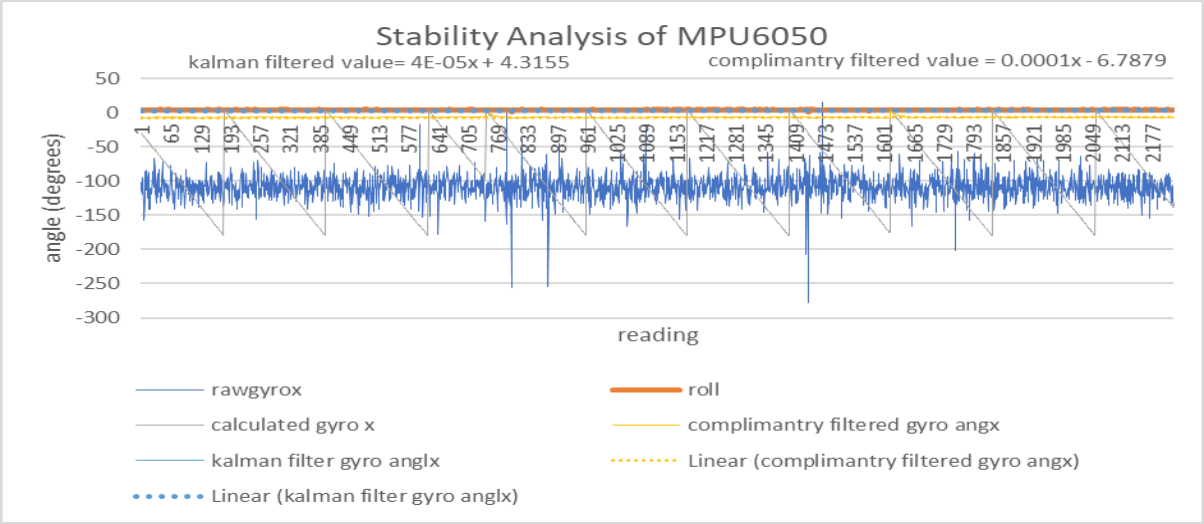


Fig 6 & 7:- Stability analysis of MPU6050 sensor module

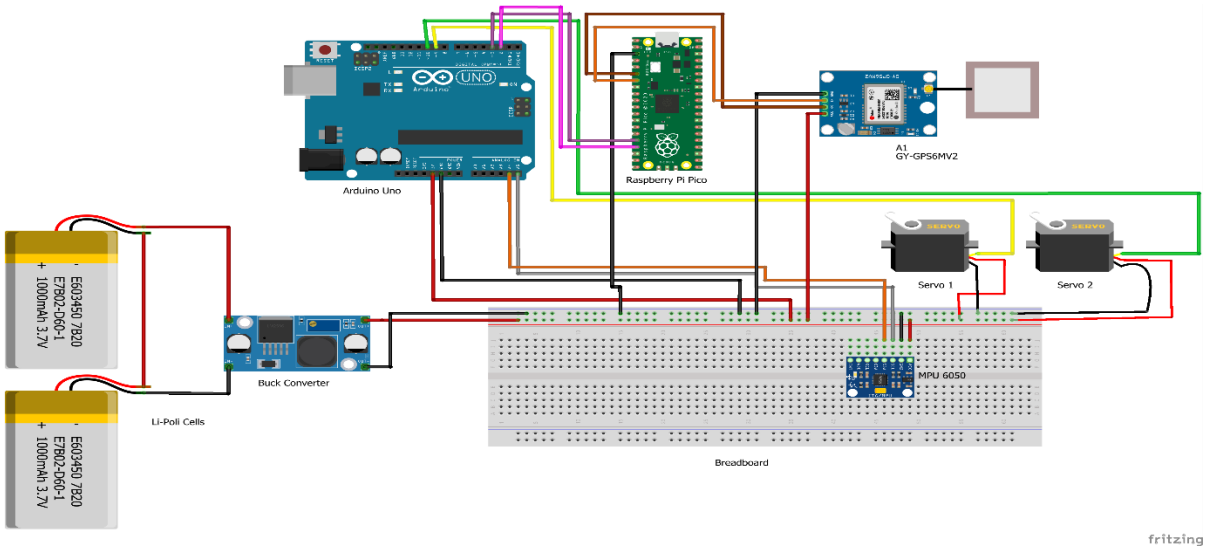


Fig 8 :- circuit diagram of open-loop sun tracking instrument

Here we powered up the whole system by using 2 batteries of 10000 mAh each and the voltage regulator module LM2596 for 5V Regulated voltage here the connection between Raspberry Pi Pico Arduino Uno, GY-GPS6MV2 , MPU6050 as well as Servo connection is also there here we need to care about one thing is that the motors need much more current than any other devices and it will also highly depends upon which motor we are using as well as how much weight we put on the motors which can vary the speed of the motors (for continuous servo) and it makes one disadvantage over the continuous servo motors but positional servo motors can be used as precise till 1-degree precision . Here is the problem comes here we want to use this platform to track the sun. the sun itself having an angular diameter of 30’ minutes which means(0.5 degrees).

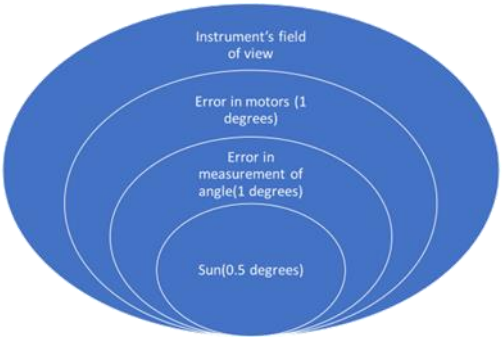


Fig 9:- solution for error

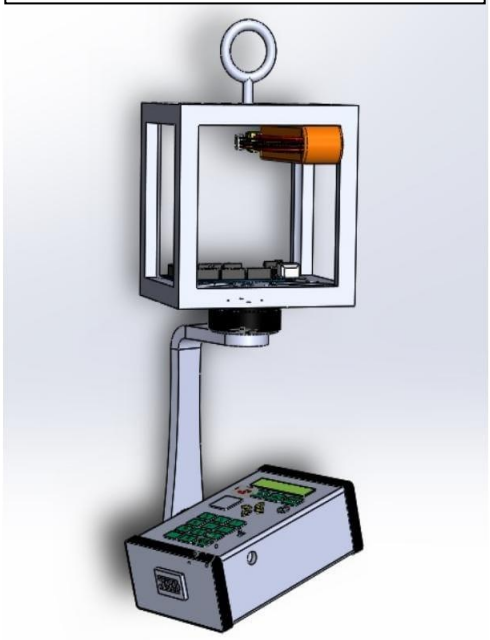


Fig 10 :- CAD design of Instrument with gimbal motor

For the solution we need to take the NA (numerical aperture) of the instrument here we use the sun photometer which has field of view of 2.5 degrees. So, we get that data from the sun rays so that's how we can get the data for the instruments. suppose we had the error of 1 degree in the measurement of the angle so the total error can increase to 2 degrees since we had the field of view (NA in the case of fibers) we still get the data. Here we had two types of errors.

- Error in the measurement of angle: -
  - when we are talking about the angle of the measurement it depends upon the imu sensor here our imu sensor gives us some data it can be possible that provided data has errors of 1 degree.

- Error in the execution of motor:-
  - When we talk about motors we should understand how it works first of all servo motors have an encoder , potentiometer, and motor inside them. all of these make the servo motor precise till 1 degrees rotation but as we know that it is not a very reliable method when temperature varies and it is possible because it contains a resistor that varies with the temperature. The encoder also had its limits. Servo had a lack of smoothness but the gimbal works pretty smoothly. Both of the errors can be adjusted by the use of the gimbal motors and motor drivers. But it can cost us higher power consumption as well as weight also increases. here is some comparison between both of them

	Weight	Power consumption
Servo (MG 995)	55 gm	4.8 V , 0.7 A (3.5 W)
Gimbal (GB 54-1)	137 gm	4s

Table 1 :- motor comparison

- Hence Gimbal motors consumed much more power than Servo motors but the smoothness in gimbal motors is unmatched and also they can carry much more load than servo motors as it depends upon how much current we give to the motor.
- Right now we are using servo motors for lower power consumption it also had the advantage of the lower complexity in the code and it didn't require a motor driver . here is our model which uses gimbal motors for rotation.
- Here the important thing about the angles is the reference (which angle we should take as 0 degrees) here we are use the horizontal coordinate systems of the celestial sphere.
- So that's why we use north as our reference for Azimuth

Cardinal point	Azimuth
North	0º
East	90º
South	180º
West	270º

Table 2:- directional references for Azimuth

- We use altitude angle from the horizontal plane(it is the angle between the object and the local horizon).

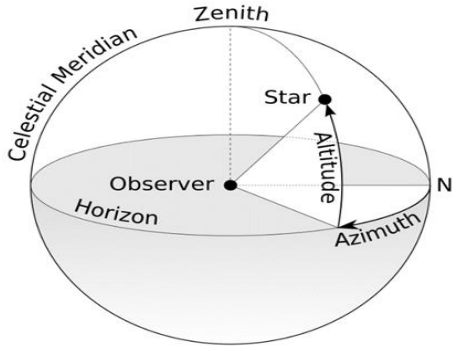


Fig 11:- horizontal co-ordinate system in celestial sphere

Result and Conclusion

- Here from this method we got accurate coordinates of the sun using the open-loop method(~0.01° precession)[18].

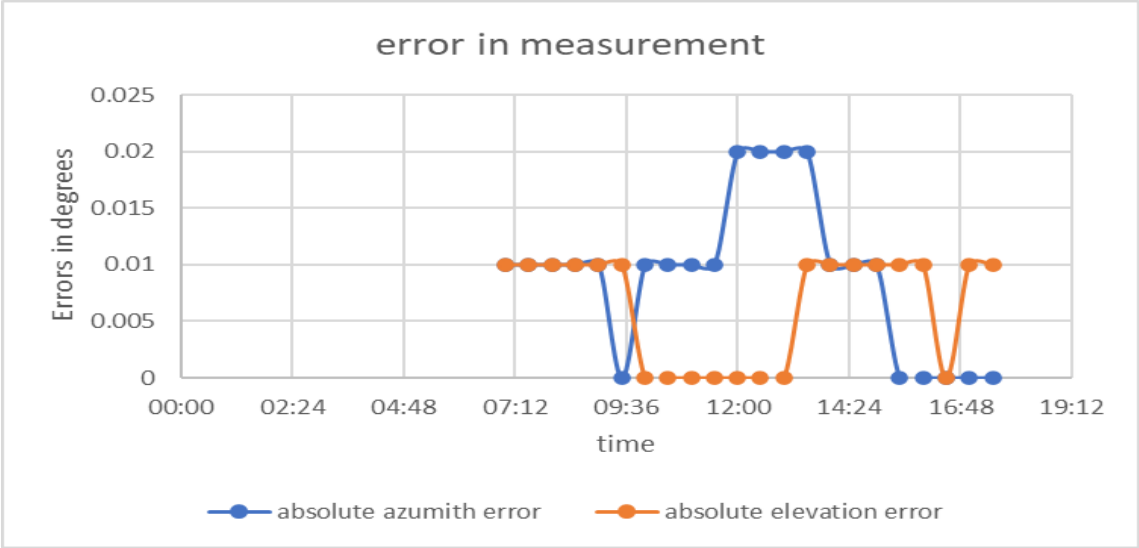


Fig 12:- error in measurement

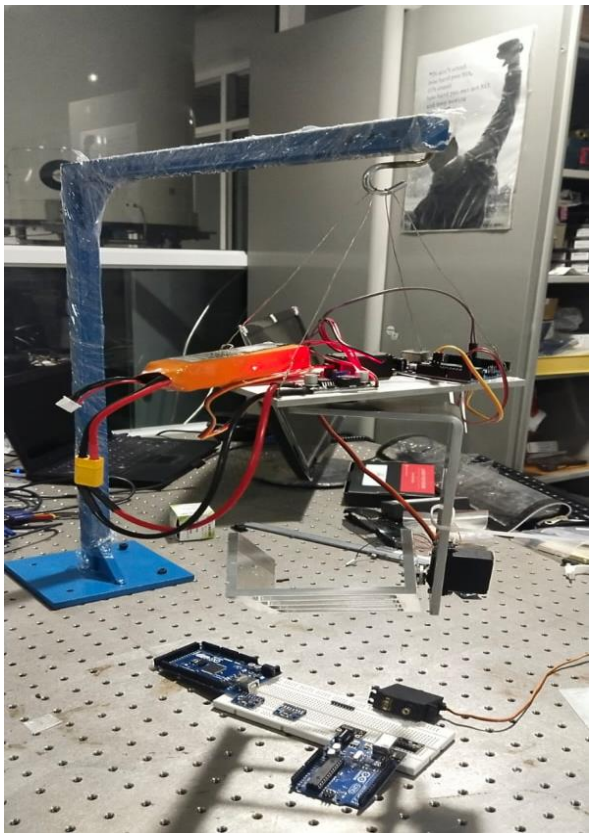


Fig 13 :- Experimental setup of the instrument

- The method we are using right now is not so sufficient for continuous data collection
- As it whole system is made as an open loop system it does have not any kind of feedback here so it makes the method somewhat less precise than close loop systems when we consider external disturbance added in the whole system so it makes the system much unstable for longer period which results into data lose. For that purpose, we uses mpu 6050 (IMU) sensor here
- MPU 6050 can work fine with proportional feedback but it is not so smooth transition .
- Here we get the settling time of nearly 1 second (using pid tunning) for response time with transient behavior which is not so good because for 1-

second sun can be out of the instrument’s field of view. It results in us data loss or unstable system

- As the temperature changes the microcontroller peripheral devices and modules are either getting some sort of error or they can’t work efficiently.
- Here mpu6050 does not contain any kind of magnetometer so we need to calibrate in the north direction manually

Solutions & Future Aspects

- continuous data collection can be possible using higher clock frequency microcontrollers and peripheral devices
- We can get the 0.1 second response time using an external processor and software work but we need to make that work on standalone also that's why we need more computational power as well as more efficient and precise motors.
- As a platform it needs some sort of cooling system to make sure the temperature variance will not take place as it affects the process as well as the performance of the microprocessor, sensors, and other peripheral devices.
- We need to use either a pid system or a model predictive control system based on a system which had higher computation power and higher clock frequency.
- We need to change the sensor mpu 6050 with bmx 160 which is 16 bit accelerometer, gyroscope and geomagnetic sensor.

References

1. Kaufman, Y. J. (1993). The atmospheric effect on solar radiation and atmospheric aerosols. In Advances in Atmospheric Remote Sensing with Lidar (pp. 27-42). Springer, Berlin, Heidelberg.
2. Gueymard, C. A. (2008). Solar radiation and daylight models. Academic Press.
3. Holben, B. N., Eck, T. F., Slutsker, I., Tanre, D., Buis, J. P., Setzer, A., ... & Vermote, E. (1998). AERONET—a federated instrument network and data archive for aerosol characterization. Remote sensing of environment, 66(1), 1-16.
4. Jethva, H., & Satheesh, S. K. (2016). A review of satellite-based aerosol detection techniques for air quality applications. Atmospheric environment, 125, 404-415.
5. IPCC. (2013). Climate change 2013: The physical science basis. Working Group I contribution to the fifth assessment report of the Intergovernmental Panel on Climate Change.
6. Mishra, A. K., Shibata, T., & Matsui, T. (2017). Aerosol optical depth retrieval over land surfaces from AVIRIS-NG using a kernel-driven BRDF model. Remote Sensing, 9(11), 1162.
7. Luria, M., Maman, S., & Sinvani, M. (2013). Design and development of a dual-axis sun tracker with two degrees of freedom. IEEE Transactions on Industrial Electronics, 60(4), 1452-1462.
8. Diner, D. J., Beckert, J. C., Reilly, T. H., Bruegge, C. J., Conel, J. E., Kahn, R. A., ... & Martonchik, J. V. (1998). Multi-angle Imaging SpectroRadiometer (MISR) instrument description and experiment overview. IEEE Transactions on Geoscience and Remote Sensing, 36(4), 1072-1087.
9. Liou, K. N. (2002). An introduction to atmospheric radiation. Academic Press.
10. Dubovik, O., & King, M. D. (2000). A flexible inversion algorithm for retrieval of aerosol optical properties from Sun and sky radiance measurements. Journal of Geophysical Research: Atmospheres, 105(D16), 20673-20696.
11. Holben, B. N. (1986). Characteristics of maximum-value composite images from temporal AVHRR data. International journal of remote sensing, 7(11), 1417-1434.
12. Kumar, K. R., Kumar, N. K., & Reddy, B. S. (2010). Temporal analysis of aerosol parameters over an urban station using ground-based remote sensing. International Journal of Remote Sensing, 31(3), 631-642.
13. Ramanathan, V., Crutzen, P. J., Kiehl, J. T., & Rosenfeld, D. (2001). Aerosols, climate, and the hydrological cycle. Science, 294(5549), 2119-2124.
14. Devidal, J. L., & Boucher, O. (2002). Retrieval of the aerosol optical thickness using ground-based measurements of solar radiation at two different altitudes. Journal of Geophysical Research: Atmospheres, 107(D24), ACH 8-1-ACH 8-15.
15. Britannica, T. Editors of Encyclopaedia. "celestial coordinates." Encyclopedia Britannica, February 20, 2018. https://www.britannica.com/science/celestial-coordinates.
16. Datasheet of adxl,mpu6050,Arduino uno, raspberrypi pico, bmx 160 , neo gps 6m, voltage regulator , servo mg 995
17. John Clark Craog ,Sun Position: Astronomical Algorithm in 9 Common Programming Languages, May 31,2017 , ISBN-13 978-1546576259, ISBN-10 1546576258
18. NOAA solar calculator ,Global monitoring laboratory (https://gml.noaa.gov/grad/solcalc/)

Code

```
from machine import Pin, UART, SoftI2C
import utime, time
gpsModule = UART(1, baudrate=9600)
print(gpsModule)
buff = bytearray(255)
TIMEOUT = False
FIX_STATUS = False
latitude = ""
longitude = ""
satellites = ""
GPStime = ""
h = ""
m = ""
s = ""
def getGPS(gpsModule):
    global FIX_STATUS, TIMEOUT, latitude, longitude, satellites, GPStime
    timeout = time.time() + 8
    #gpsModule.readline()
    #buff = str(gpsModule.readline())
    #print(buff)
    while True:
        gpsModule.readline()
        buff = str(gpsModule.readline())
        #print(buff) #debug
        parts = buff.split(',')
        if (parts[0] == "b'$GPGGA" and len(parts) == 15):
            if(parts[1] and parts[2] and parts[3] and parts[4] and parts[5] and parts[6] and parts[7]):
                print(buff)

                latitude = convertToDegree(parts[2])
                if (parts[3] == 'S'):
                    latitude = -latitude
                longitude = convertToDegree(parts[4])
                if (parts[5] == 'W'):
                    longitude = -longitude
                satellites = parts[7]
                h = print(parts[1][0:2])
                m = print(parts[1][2:4])
                s = print(parts[1][4:6])
                GPStime = parts[1][0:2] + ":" + parts[1][2:4] + ":" + parts[1][4:6]
                FIX_STATUS = True
                break
        if (time.time() > timeout):
            TIMEOUT = True
            break
        utime.sleep_ms(1000)
def convertToDegree(RawDegrees):
    RawAsFloat = float(RawDegrees)
    firstdigits = int(RawAsFloat/100)
    nexttwodigits = RawAsFloat - float(firstdigits*100)
```

```
    Converted = float(firstdigits + nexttwodigits/60.0)
    Converted = '{0:.6f}'.format(Converted)
    return str(Converted)
i=0
while i==0:
    i = i+1
    getGPS(gpsModule)
    if(FIX_STATUS == True):
        print("Printing GPS data...")
        print(" ")
        print("Latitude: "+latitude)
        print("Longitude: "+longitude)
        print("Satellites: " +satellites)
        print("Time: "+GPStime)
        print("h"+h)
        print("m"+m)
        print("s"+s)
        print("----------------------")
        FIX_STATUS = False
    if(TIMEOUT == True):
        print("No GPS data is found.")
        TIMEOUT = False
# -*- coding: utf-8 -*-
"""
Created on Sun Nov 27 01:51:40 2022
@author: yap21
"""
# sunpos.py
import math
execfile("oncegps.py")
```

```
def gmt_to_ist(gmt_time, timezone='Asia/Kolkata'):
    """
    Converts a given time in GMT format to IST format.
    Args:
        gmt_time (str): A string representing the time in the format 'HH:mm:ss'
        timezone (str): An optional parameter specifying the timezone to convert to. Defaults to 'Asia/Kolkata'.
    Returns:
        str: A string representing the time in the specified timezone in the format 'HH:mm:ss'
    """
  # Define a dictionary of timezone offsets in seconds
    timezone_offsets = {
        'GMT': 0,
        'Asia/Kolkata': 19800,  # GMT +5:30 (5.5 hours = 19800 sec)
        'US/Eastern': -18000,  # GMT -5:00
        # Add more timezone offsets as needed
    }
    # Split the input time string into hours, minutes, and seconds
    gmt_hours, gmt_minutes, gmt_seconds = map(int, gmt_time.split(':'))
    # Calculate the total number of seconds in the input time
    gmt_total_seconds = gmt_hours * 3600 + gmt_minutes * 60 + gmt_seconds
    # Calculate the time difference between GMT and the target timezone in seconds
```

```
    timezone_offset = timezone_offsets.get(timezone, 0)

    # Calculate the total number of seconds in the target timezone

    target_total_seconds = gmt_total_seconds + timezone_offset

    # Calculate the hours, minutes, and seconds in the target timezone

    target_hours, target_minutes = divmod(target_total_seconds, 3600)

    target_minutes, target_seconds = divmod(target_minutes, 60)

    # Format the resulting time as a string and return

    return f"{target_hours:02}:{target_minutes:02}:{target_seconds:02}"

# Convert a GMT time to IST

GPStime = gmt_to_ist(GPStime)

print("Latitude: "+latitude)

print("Latitude: "+latitude)

print("Longitude: "+longitude)

print("Satellites: " +satellites)

print("Time: "+GPStime)

parts = GPStime.split(":",2)

hour = parts[0]

#hour = int(hour)

minute =parts[1]

#minute = int(minute)

second = parts[2]

latitude = float(latitude)

longitude = float(longitude)

#print("h"+h)

#print("m"+m)

#print("s"+s)

print("----------------------")

        #print(buff) #debug

        #parts = buff.split(',')

#print("h"+h)

#print("m"+m)

#print("s"+s)

#from datetime import datetime

def sunpos(when, location, refraction):

# Extract the passed data

    year, month, day, hour, minute, second, timezone = when

    latitude, longitude = location

# Math typing shortcuts

    rad, deg = math.radians, math.degrees

    sin, cos, tan = math.sin, math.cos, math.tan

    asin, atan2 = math.asin, math.atan2

# Convert latitude and longitude to radians

    rlat = rad(latitude)

    rlon = rad(longitude)

# Decimal hour of the day at Greenwich

    greenwichtime = hour - timezone + minute / 60 + second / 3600

# Days from J2000, accurate from 1901 to 2099

    daynum = (367 * year - 7 * (year + (month + 9) // 12) // 4 + 275 * month // 9 + day - 730531.5 + greenwichtime / 24)

# Mean longitude of the sun

    mean_long = daynum * 0.01720279239 + 4.894967873

# Mean anomaly of the Sun
```

```
    mean_anom = daynum * 0.01720197034 + 6.240040768

# Ecliptic longitude of the sun

    eclip_long = (mean_long + 0.03342305518 * sin(mean_anom) + 0.0003490658504 * sin(2 * mean_anom))

# Obliquity of the ecliptic

    obliquity = 0.4090877234 - 0.000000006981317008 * daynum

# Right ascension of the sun

    rasc = atan2(cos(obliquity) * sin(eclip_long), cos(eclip_long))

# Declination of the sun

    decl = asin(sin(obliquity) * sin(eclip_long))

# Local sidereal time

    sidereal = 4.894961213 + 6.300388099 * daynum + rlon

# Hour angle of the sun

    hour_ang = sidereal - rasc

# Local elevation of the sun

    elevation = asin(sin(decl) * sin(rlat) + cos(decl) * cos(rlat) * cos(hour_ang))

# Local azimuth of the sun

    azimuth = atan2( -cos(decl) * cos(rlat) * sin(hour_ang), sin(decl) - sin(rlat) * sin(elevation),)

# Convert azimuth and elevation to degrees

    azimuth = into_range(deg(azimuth), 0, 360)

    elevation = into_range(deg(elevation), -180, 180)

# Refraction correction (optional)

    if refraction:

        targ = rad((elevation + (10.3 / (elevation + 5.11))))

        elevation += (1.02 / tan(targ)) / 60

# Return azimuth and elevation in degrees

    return (round(azimuth, 2), round(elevation, 2))

def into_range(x, range_min, range_max):

    shiftedx = x - range_min

    delta = range_max - range_min

    return (((shiftedx % delta) + delta) % delta) + range_min

if __name__ == "__main__":

# Close Encounters latitude, longitude

    location = (latitude, longitude)

    hour = int(hour)

    minute = int(minute)

    second = int(second)

    #location = (23.0225, 72.5714)

# Fourth of July, 2022 at 11:20 am MDT (-6 hours)

    #when = (year,month,day,hour,minute,seconds,timezone)

    #when = (2023, 2, 24, 13, 50, 0, +5.5)

    when = (2023, 2, 24, hour, minute, second, 0)

    #when = (2023, 2, 22, h, m, s, +5.5)

# Get the Sun's apparent location in the sky

    azimuth, elevation = sunpos(when, location, True)

# Output the results

    print("\nWhen: ", when)

    print("Where: ", location)

    print("Azimuth: ", azimuth)

    print("Elevation: ", elevation)
```

```
# When:  (2022, 7, 4, 11, 20, 0, -6)
# Where:  (40.602778, -104.741667)
# Azimuth:  121.38
# Elevation:  61.91
from machine import UART
import time
#from timer import delay
# Set up UART on pins 16 (TX) and 17 (RX) at a baud rate of 115200
uart = UART(0, baudrate=115200, tx=machine.Pin(12), rx=machine.Pin(13))
print(uart)
# Read a line of data from the UART
#data = uart.readline()
execfile("fakest.py")
uart.write(f' azimuth: {azimuth}')
uart.write(f',Elevation : {elevation}')
uart.write(f' | ')
#uart.write(b"here is the data\n")
while(1==1):
 #   pass
    # Write a line of data to the UART
    execfile("fakest.py")
    #uart.write(f'{azimuth}')
    #uart.write(',')
    #uart.write(f'{elevation}')
    #uart.write("\n")
    uart.write(f'{azimuth},{elevation}')
    print(f'{azimuth},{elevation}')
    print("sent suc")
    time.sleep_ms(1000)
    #uart.write("When: ", when)
    #uart.write(",where: ", location)
    #Alternatively, you can use the uart object as a file-like object
    # to read and write data:
    #uart.write(b"Hello, world!\n")
#    data = uart.readline()
 #   delay(1)
```

For Arduino uno we uses c/c^{++} programming language.

```
/* Get tilt angles on X and Y, and rotation angle on Z
 * Angles are given in degrees
 * License: MIT
 */
#include "Wire.h"
#include <MPU6050_light.h>
#include <Servo.h>
Servo pitchs;
Servo rolls;
float X,Y,Z,pitch,roll, yaw;
#include <SoftwareSerial.h>
int a0,b0,a,b;
String message ;
SoftwareSerial uart(3, 2);
MPU6050 mpu(Wire);
unsigned long timer = 0;
void setup() {
  Serial.begin(115200);
  uart.begin(115200);
```

```
  Wire.begin();
  byte status = mpu.begin();
  Serial.print(F("MPU6050 status: "));
  Serial.println(status);
  while(status!=0){ } // stop everything if could not connect to MPU6050
  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  // mpu.upsideDownMounting = true; // uncomment this line if the MPU6050 is mounted upside-down
  mpu.calcOffsets(); // gyro and accelero
  Serial.println("Done!\n");
  pitchs.attach(9);
  rolls.attach(10);}
void loop() {
  mpu.update();
  String message = uart.readStringUntil('\n');
  if((millis()-timer)>1000){ // print data every 10ms
  Serial.print("X : ");
  X = mpu.getAngleX();
  Y = mpu.getAngleY();
  Z = mpu.getAngleZ();
  String message = uart.readStringUntil('\n');
  pitch = 180 * atan2(X, sqrt(Y*Y + Z*Z))/PI; // calculate the pitch angle
  pitch = pitch +90; // here 90 is to make the pitch from 0 to 180 insted of -90 to 90 for servo
  pitch = pitch + b; // set the angle accordingly your elevation here b is elevaion
  roll = 180 * atan2(Y, sqrt(X*X + Z*Z))/PI;
  roll = roll + 90; // here 90 is to make the pitch from 0 to 180 insted of -90 to 90 for servo
  roll = roll + a; // here a is azimuth
  pitchs.write(pitch);
  rolls.write(roll);
  Serial.println();
  Serial.print(pitch);
  Serial.print(roll);
  timer = millis();
  }
}
```