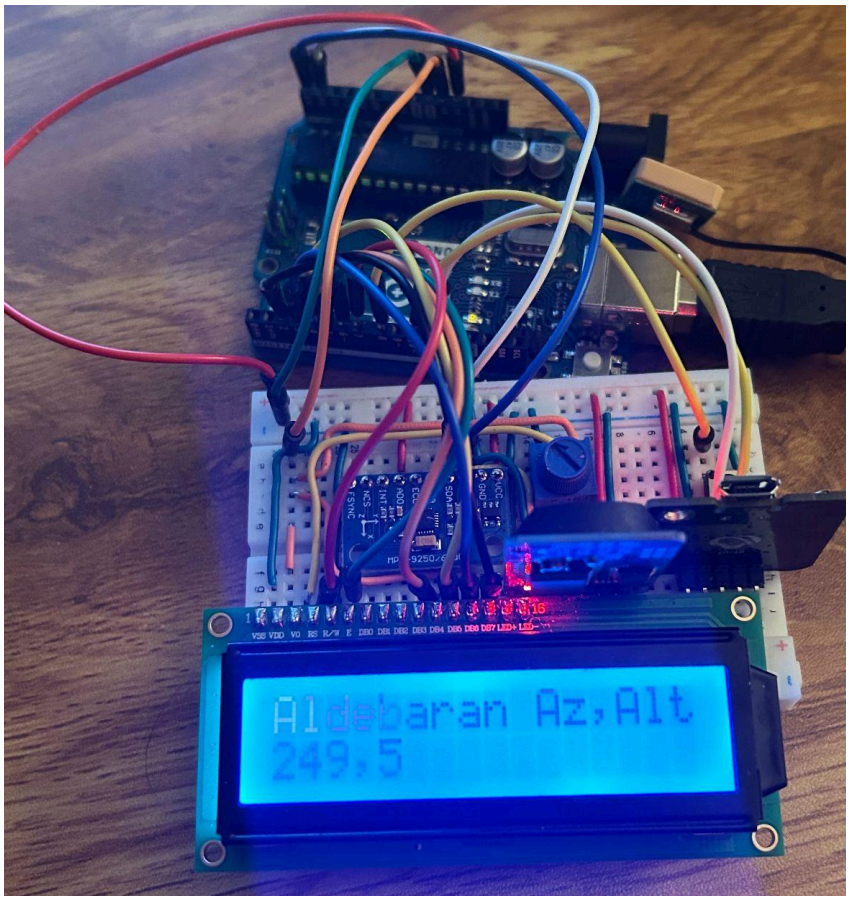
Wes Scarpa, Bridget Valdez, & Alessandra Nefedenkova

COMP 349: Robotics

Professor Cianci

May 4, 2025

Modular Astrological Compass



Abstract

Our project utilizes an Arduino and its sensor data to create a handheld astrological compass designed to guide users toward celestial bodies using astrological calculations. Implementing a GPS module, real-time clock (RTC), and internal measurement unit (IMU), allows our system to

dynamically calculate and update the user's location, time, and orientation. These values are then utilized in collaboration with the SkyMap library to compute the azimuth and altitude of a given celestial body. The IMU values are compared to the calculated coordinates and use the difference to determine the relative location. The user is presented with an LCD screen and button that allows the selection of an object, and its relative position. Through handling raw sensor data to do calculations, combining the live readings of multiple sensors for a single output, calibrating sensors to environmental factors, adapting to sensor limitations, and handling sensor noise, we deepened our understanding of real-time sensors, I²C protocols, and the challenges of combining several hardware components into one cohesive system. Exploring each of the sensor modules allowed us to learn about the variability in sensor data and taught us how to go from isolated testing to merging our understanding of each component into one interconnected system. Our project gave us insight into the uncertainties of embedded system design, teaching us how to code, work with hardware, and think about these systems in general.

Objectives

- Incorporate necessary modules and components
    - GPS module to record live longitude and latitude coordinates
    - IMU module to determine orientation through yaw, pitch, and roll
    - RTC clock module to keep track of and update time
    - LCD screen to display target celestial body and degrees until alignment
    - Button to allow user to select celestial body to locate
- Accurately calculate the azimuth and altitude of celestial body

    - Use RTC and GPS readings along with known right ascension and declination values to calculate approximate azimuth and altitude

- Compare user orientation to celestial body
    - Convert IMU magnetometer readings for heading and accelerometer readings for pitch to approximate azimuth and altitude
    - Calculate difference between current orientation and target celestial body
    - Use these readings to update directional values on LCD screen

Process

Our astrological compass is built around a coded system that takes in real-time data from the user's physical environment, like GPS coordinates, time, and device orientation, and combines it with astronomical models to calculate the exact position of a target object in the sky.

We wrote a program for an embedded system that calculates the azimuth and altitude of a celestial object based on a specified date, time, and geographic location, and gives a real-time position of celestial objects relative to the user's current location and orientation. The code translates raw data from our sensors into actionable directions. Before working with any data from sensors, we had to make sure that the code for calculating the Azimuth and Altitude of the target object works correctly.

We used the SkyMap library to perform the following sequence of calculations:

1. **Input values**: We hardcoded the date, time, latitude, longitude, and the celestial object's Right Ascension (RA) and Declination (Dec). For initial testing, we used Sirius (RA =

101.52°, Dec = -16.7424°), and Los Angeles coordinates (Lat = 34.06°, Long = -118.24358).

2. **Convert time to UTC**: We used the function SKYMAP_hh_mm_ss2UTC() to convert local time we get from our RTC clock module into UTC using a timezone offset.

3. **Calculate J2000 offset**: Using SKYMAP_j2000(), we computed the number of days since January 1, 2000 (J2000 epoch). This is necessary for time-based astronomical calculations.

4. **Compute Local Sidereal Time (LST)**: Using SKYMAP_local_sidereal_time(), we used the J2000 value, UTC time, and longitude to determine LST, which tracks the rotation of the sky above a specific location.

5. **Calculate Hour Angle**: With SKYMAP_hour_angle(), we subtracted the star's RA from the LST to compute the Hour Angle, which represents how far the star is from the local meridian.

6. **Get Azimuth and Altitude**: Using SKYMAP_search_for_object(), we input the Hour Angle, Dec, and our latitude to compute the object's azimuth (horizontal compass direction) and altitude (vertical angle in the sky)

All calculations were printed to the serial monitor to confirm accuracy. This allowed us to verify that the code could correctly locate celestial objects from any known location and time without needing hardware input. After confirming the code worked with static values, we integrated sensor modules to supply live, real-world data in place of hardcoded values. This involved interfacing the GPS, RTC, and IMU modules with the Arduino microcontroller. Before

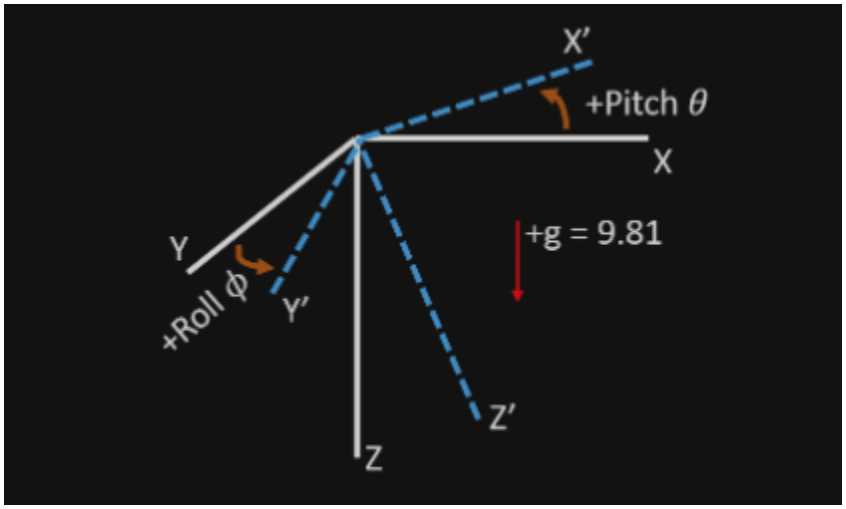
integrating with the main calculation code, we verified each module independently. For both the GPS and RTC, we confirmed the accuracy and update rate of the data using serial printouts and cross-checking values with known time and location.

1. **GPS Module:** We connected a GPS module to the Arduino using TXD and RXD pins to obtain real-time latitude and longitude data. The GPS outputs serial data, which we parsed using the TinyGPS++ library that handles reading the raw data of the sensor and returns coordinates in decimal degrees. These values replaced the hardcoded location inputs in the astronomical calculation.

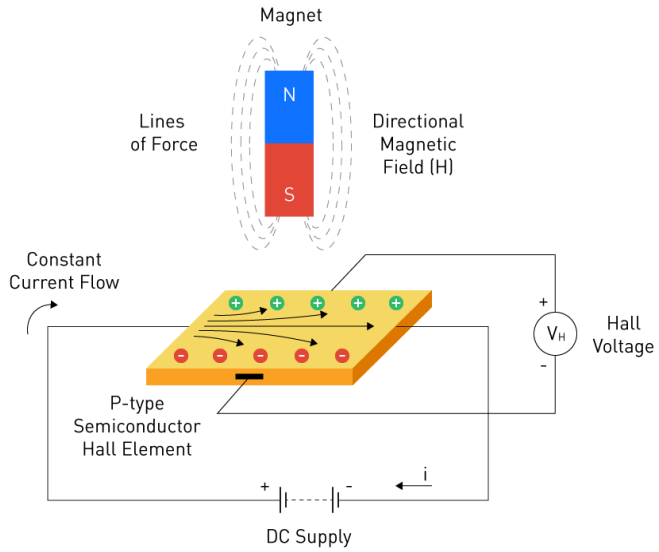
2. **RTC Module**: The RTC (Real-Time Clock) module provided accurate year, month, day, hour, minute, and second values, which we used as input to the SkyMap time functions. To accomplish that, we set the current time once and connected a small battery, which allows the device to store and keep track of elapsed time using the waveform generated by an oscillating crystal. This enables the device to maintain accurate time even when it's not connected to the computer. We connected it via I²C (SDA/SCL pins).This ensured time continuity even when the device was disconnected from power. We tested the RTC module using the RTClib library and verified output by printing timestamps to the serial monitor.

3. **IMU/ MPU Sensor**: The IMU tracks the device's pitch, roll, and heading (yaw) in real time, which we use to determine the direction the user is currently pointing. We connected it via I²C to read the data collected by the sensor. Using the MPU, we calculated the device's orientation and heading: the heading is determined by the magnetometer, which outputs a voltage based on the device's alignment with the Earth's

magnetic field, while the pitch is calculated using the accelerometer, which constantly detects the direction of gravity, providing a reference point for its orientation to be calculated use mechanics.



Pitch and Roll from Accelerometer. https://mwrona.com/posts/accel-roll-pitch/



Hall Effect within Magnetometer.
https://www.monolithicpower.com/en/learning/resources/hall-effect-sensors-a-comprehensive-guide

To improve usability and interface design, we implemented an LCD screen and a button in our Arduino setup. We used a library to be able to send characters easily to the LCD. The displayed real-time outputs for the difference in our azimuth and altitude and those of the target star gives users a clear visual readout of the star's position. Additionally, we programmed a scroll button that lets the user cycle through a list of predefined stars. It's connected to a digital pin on one side and grounded on the other. When the button is pressed, the pin reads a LOW signal, which we used to trigger a shift to the next entry in the hashmap. This feature added a layer of interactivity and functionality to our astrological compass, moving it from a test device to a more complete, user-facing instrument.

Before calculating altitude (pitch), we first ensured that GPS, IMU (which includes a magnetometer and accelerometer), and RTC were properly connected and working. To get the IMU values from magnetometer and accelerometer, we worked with the raw data from the sensor instead of using a library. We connected the component with I²C to read the data, but did all the calculations manually using sensor values. The IMU's magnetometer allowed us to determine the device's heading or yaw, which corresponds to azimuth: the compass direction the device is facing, measured from magnetic north. As long as the device remained level (with minimal pitch or roll), the heading was accurate and stable. However, we observed a ~7° discrepancy between our device's heading and that of a phone compass app. This is due to the IMU measuring magnetic north, while the phone typically references true north; the difference between them, known as magnetic declination, varies by location and accounts for the error. Additionally, any

tilt in the device distorts the magnetometer's reading since it changes the direction the magnetic field passes through it, so accurate heading measurements required keeping the device flat. We attempted many different ways to correct for tilt, (including a tilt correcting algorithm that combines values from both accelerometer and the magnetometer, that is often referenced as a viable solution by other users of the IMU); however, it did not seem to be working at all and made readings worse. We also tried a rotation matrix that would convert the measured yaw, pitch, and roll of the device's reference frame into azimuth and altitude of the user's reference frame. Although promising, this also was not successful.

To get the altitude angle (pitch) from our device, we use readings from the accelerometer in the IMU. An accelerometer measures acceleration forces in all directions. When the device is stationary, the only force acting on it is gravity, so it essentially becomes a tool for detecting orientation relative to the ground. To calculate pitch (how much the device is tilted forward or backward), we apply a formula:

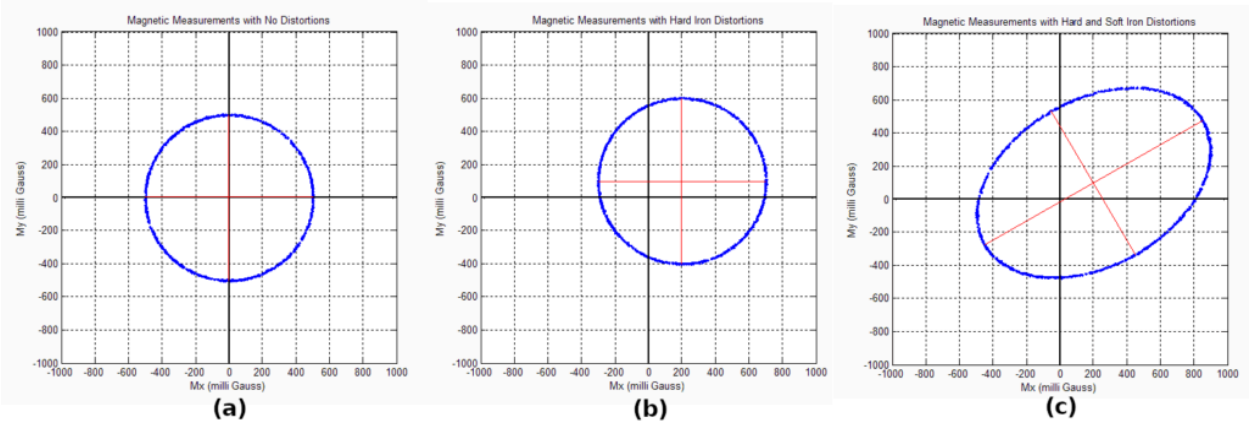
pitch = arctangent of (acceleration in the y-axis / acceleration in the z-axis)

This works because gravity affects all axes differently depending on the device's tilt. When the device tilts, the distribution of gravity shifts between the y and z axes. By comparing these two, we can infer the tilt angle with simple trigonometry.

To get the yaw and altitude working properly on our device, we implemented two calibrations: one for the accelerometer and one for the magnetometer. We calibrated the accelerometer by placing the device flat on a surface to determine the offset values and adjust for them. For the magnetometer, calibration involved rotating the device 360 degrees to identify the sensor's maximum and minimum values. We used a well-established calibration method that accounts for

both hard and soft iron distortion—returning an offset to correct for hard iron and a scaling factor to correct for soft iron. Hard Iron distortions are created by objects that produce a constant magnetic field near your sensor: batteries, or the Arduino board itself can shift the entire magnetic field and offset magnetometer readings in all directions. Soft Iron Effects (Field Distortion) are created by non-magnetic but conductive materials that bend or distort magnetic fields around: wires, or aluminum make the magnetic field stretched or squashed.

This approach was highly effective and significantly improved the accuracy of our readings.



Hard and Soft Iron Distortion.

https://atadiat.com/en/e-magnetometer-soft-iron-and-hard-iron-calibration-why-how/

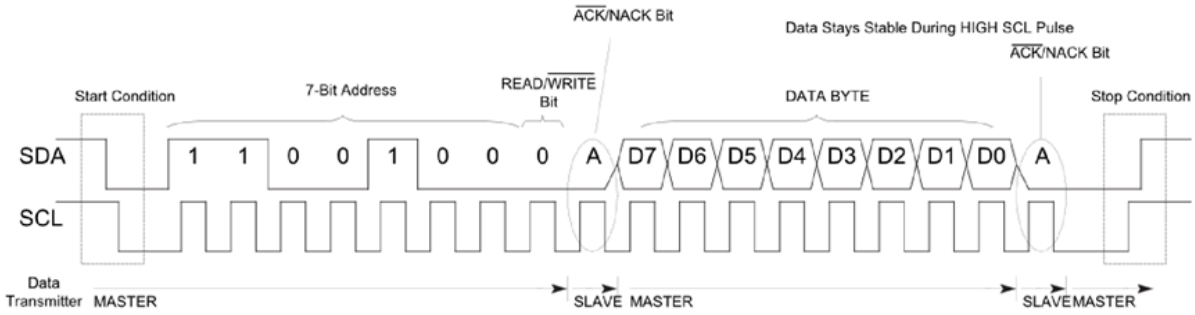
I²C protocol:

One of the key technical systems we worked with during the project was the I²C (Inter-Integrated Circuit) communication protocol, which the Arduino uses to exchange data with multiple components. The Wire library in Arduino is built on this protocol. I²C is a serial communication method that allows multiple devices to connect using just two shared lines: SDA - Serial Data Line, which is used to transmit data between the controller (Arduino) and the components (like

the IMU and RTC); and SCL - Serial Clock Line that provides the timing signal that ensures both devices stay in sync during communication.

Each device on the I²C bus has a unique 7-bit address embedded in its hardware. During setup, we encountered issues where the Arduino couldn't detect certain components. To troubleshoot, we implemented a code that scans the I²C bus for all connected device addresses and confirms their presence. Doing so, we realized that the IMU and the RTC initially had the same addresses: 0x68. However, by connecting the AD0 pin on the IMU to 3.3V changed the device's address to 0x69, giving us two distinct addresses to read and write from for both devices. This allowed us to use the two devices at the same time. Understanding how to scan for and interpret these addresses was critical in resolving connection issues.

We also learned how the Arduino initiates communication using SDA by sending a sequence of bits: the device address and a read/write bit that tells the component whether it should receive data (read) or respond with data (write). Initially, we faced challenges understanding why some devices weren't responding. As explained previously, that came down to addressing conflicts. This taught us the importance of mastering low-level communication protocols when integrating sensors into a system. Through trial, error, and debugging, we gained hands-on experience with I²C's structure: how to scan for device addresses, and how to ensure timing and communication alignment across all connected components.



I²C Protocol Bit Transmission.

https://www.analog.com/en/resources/technical-articles/i2c-primer-what-is-i2c-part-1.html

With the GPS supplying current latitude and longitude, the RTC providing precise time, and the IMU giving us the device's heading (yaw) and pitch, we could now calculate our own azimuth and altitude. Using the SkyMap library, we also computed the target star's azimuth and altitude based on its Right Ascension, Declination, and our real-time coordinates. We then compared these two sets of values—our current orientation vs. the star's location—and calculated the angular difference in both azimuth and altitude. This difference represents how far off the device is from pointing directly at the star. We displayed these two values on an LCD screen, providing the user with a clear and responsive readout. By integrating all sensor data, processing it, performing calculations, and displaying visual feedback, the user is able to adjust their device orientation in real time to align with the target object.

Challenges

1. The Arduino setup kept breaking down: wires would come loose or short, and sometimes the Arduino would stop uploading code entirely. We suspect this was due to the inconsistent connections through the breadboard. To troubleshoot, we started testing

components one by one with minimal code and connections, reseating the wires carefully, and replacing wires when necessary. Stability improved once we simplified our layout and secured the wiring more intentionally.

2. Our LCD screen powered on, but wouldn't display anything, even though the code worked previously and wasn't changed. We suspected a wiring or contrast issue, or a problem with the screen itself. After carefully examining the setup, we realized that a pin was mistakenly connected to 5 volts instead of a pin on the arduino. This did not allow the LCD screen to communicate properly with the arduino. Easy fix!

3. The yaw value (compass heading) from our IMU had a constant drift/ increasing value, even when the device was stationary. We realized that we bought a component without a magnetometer. This meant we couldn't get stable yaw readings, which are essential for comparison to celestial azimuth. We ended up replacing the sensor with another IMU that included a magnetometer and confirmed that it provides a proper heading relative to magnetic north. Orientation readings became much more consistent which solved the drift problem, as well.

4. Another limitation we ran into was with the GPS module. In open spaces, it worked fine, but indoors or near buildings, it took a while to get a satellite lock or sometimes didn't lock at all. We learned that this is a common issue with GPS hardware, so during testing, we made sure to place the GPS near windows or outside. As well as that, we hard coded a default coordinate to Occidental College, so if it did not read a signal it could still function with accurate results, assuming the user is somewhere near Occidental.

5. Two of our components, the RTC and IMU, use I²C communication, which meant they needed to share the same SDA (A4) and SCL (A5) pins on the Arduino. Initially, this caused confusion, as we thought the components might be conflicting with one another on the same lines. After some research, we confirmed that multiple I²C devices can be connected in parallel to the same SDA and SCL lines, as long as each device has a unique I²C address. Using the Wire library, we wrote code to scan the bus and identify all connected devices by their address, making sure to correctly specify which device we're writing to or reading from in each function.

6. Our most important issues were related to the magnetometer: Tilt compensation, calibration, and noise handling. For tilt compensation we tried several methods so that the heading would be accurate while the device was not level. These included sensor fusion libraries, a tested tilt compensation algorithm, and a matrix rotation to convert device relative yaw, pitch, and roll values into user relative orientation sometimes used in VR applications. These all made our measurements completely inaccurate and were very difficult to implement. At the moment, tilt compensation is a work in progress. For calibration, we were successful in increasing the accuracy of our device's readings by measuring the sensor values at all angles and creating offsets and scalars based on the distortion of the environment. And lastly we were able to reduce noise with a low pass filter algorithm that is commonly used on magnetometer readings: $Mag_{filtered} = (\alpha Mag_{raw} + (1 - \alpha) Mag_{prev-filtered})$ where alpha is the smoothing factor. This allowed the heading value to be quite steady and accurate.

Future Work

- As future work, we plan to add a directional arrow to visually indicate the device's heading. To ensure smooth and accurate movement of this arrow, implementing a PID controller would be beneficial. Without it, the arrow could appear jittery due to fluctuations in IMU readings. A PID system would help dampen sudden changes in orientation and allow the arrow to "ease into" the correct direction rather than snapping or jumping. This kind of control would be especially important if we later incorporate physical movement, such as servo motors.

- To make the device more robust, we could design and 3D print a custom enclosure that houses the Arduino, sensors, display, and battery in a compact and protective case. This could include a handle, compass ring, or even an adjustable phone mount if we wanted to expand the UI to a mobile app. A custom PCB could make it easier to produce a bunch of the device, it would reduce wire clutter, reduce size, and improve portability.

- One major expansion would be connecting our system to a motorized telescope mount. Instead of just pointing the user in the direction of a celestial object, the device could send motor commands to physically rotate a telescope toward the desired star. This would turn the astrological compass into a functional star tracker or auto-alignment tool for astronomers and astrophotographers. We could also build in a star catalog with a menu interface, so users could select an object and have the system point to it automatically.

- Right now, the code is set up to calculate the position of a single target star using hardcoded RA/Dec values. In future versions, we could add planets and more stars to create a full star or planet database (using a HashMap or external file) that stores multiple

objects and their coordinates. The user could scroll through a menu, select a target, and have the system recalculate azimuth/altitude in real time.

Reflection

Through the building and development of our astrological compass, we discovered the complexity in connecting multiple sensor modules to create a reliable unified system. As we began our planning we decided we needed GPS, RTC, and IMU. We quickly learned how easy it was to overlook important features we may need. Our first IMU lacked a magnetometer, which led to a significant amount of drift in yaw readings, and an LCD we had ordered ended up being much smaller than we had expected. Through these errors we learned how to read datasheets with more care and to think thoroughly about the needs of our project before deciding on a module. The IMU component turned out to be more complex than we initially expected, as it involved reading data from two subcomponents—the accelerometer and the magnetometer—both of which required separate calibration.

Next, we explored each module in isolation. Our previous experience with different sensors and readings allowed us to get meaningful readings from our new sensor modules with relative ease. We found exploring each module in isolation before unifying our system to be extremely useful to understand what each module did for our project and learn how to wire it properly. The main discovery we had in this process was hard and soft iron calibration. As we began testing our IMU with a magnetometer we found that its readings were inconsistent. We found that hard iron distortions are created by objects that produce a magnetic field and soft iron distortions are deflections or alterations in the existing magnetic field. We learned a solution was to take note of readings given when rotating the device 360 degrees. We then find the midpoint between the

maximum and minimum values, this value is subtracted from readings to get more accurate measurements.

As we got our sensors working we began learning about what calculations we would need to accurately find the position of stars. While we had expected to learn a lot about astronomy we had ended up learning a lot more. We discovered stars' positions are found using azimuth and altitude which are calculated using the time, right ascension, and declination. These values were new to us and required some reading and explanation to fully understand what we needed to find in a library, get from our sensors, and hard code in our system. Through these explorations we found that SkyMap had the best functions to calculate azimuth and altitude. SkyMap had the best calculations but had no saved right ascension and declination for stars. This meant we had to dedicate unexpected time to creating a hashmap of these values for stars we wanted to include in our compass.

Finally, integrating everything into one cohesive system highlighted the importance of clean wiring and modular code organization. Early on in our experimentation we found that loose wires and tangled cables occasionally led to accidental disconnections and some confusion. While it felt easy to connect all of our components right away and just get it working without precision we learned that this led to some issues later in our project. We learned having clean wiring and taking note of connections led to a smoother testing process. Additionally we found that the RTC and IMU shared the Arduino’s A4 and A5 lines, which were for SDA and SCL connections. This realization led to us learning about I²C connections. We found that each device has a unique 7-bit address. This means these modules could share pins and still have accurate and separate communication.

Week One:

- Outlined the goals of our final project
- Using the goals, listed what modules or components we would need
- Ordered the components not already in our kit

Week Two:

- Researched how to determine relative star position
- Explored different libraries for star mapping and chose SkyMap
- Began exploring how to get necessary readings from GPS and IMU
- Discovered GPS only works inside and IMU has drift in yaw readings

Week Three:

- Worked on circuiting and implementing LCD to display GPS readings
- Tested SkyMap with our GPS data to verify its accuracy
- Ordered RTC
- Began connecting our components

Week Four:

- Ordered an IMU for magnetometer
- Discovered issues with circuit and LCD

- Now that the modules were connected we began implementing the sensor readings with the SkyMap test code

Week Five:

- Began implementing new MPU
- Found out that we needed to calibrate for hard and soft iron interference
- Resolve tilt and heading errors
- Final testing