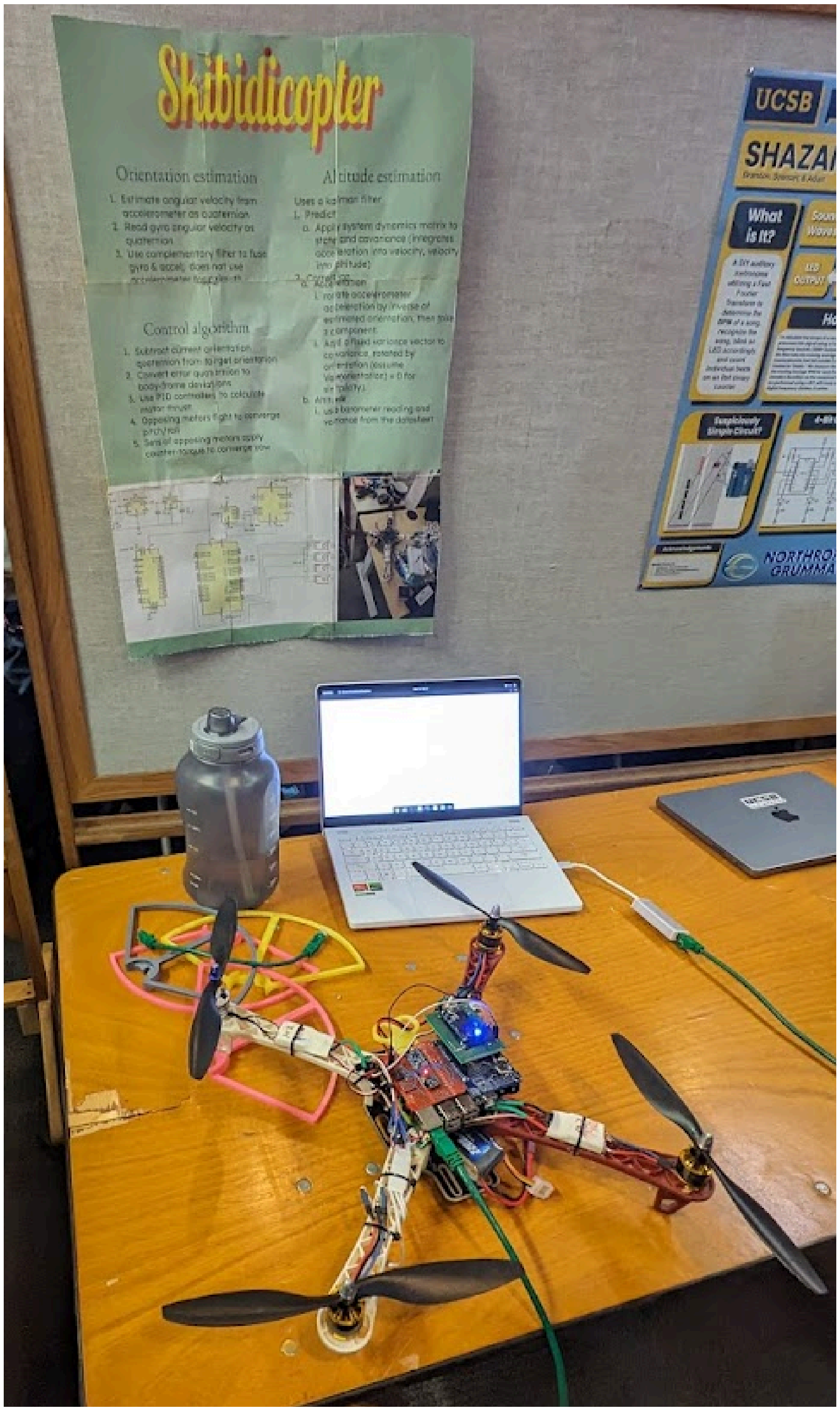
Quadcopter Final Report

Alex Petridis, Rohan Bandaru, Eric Shen



Device Functionality

There are three main components to ensuring the quadcopter can fly properly:

a. Orientation Estimation:
    - Estimate angular velocity from the accelerometer as a quaternion.
    - Read gyro angular velocity as a quaternion.
    - Use the complementary filter to fuse gyro & accel; do not use the accelerometer for azimuth.

b. Altitude Estimation:
    - Uses a Kalman Filter.
    - Predict:
        - Apply system dynamics matrix to state and covariance (integrates acceleration into velocity, velocity into altitude).
    - Correction:
        - Acceleration:
            - rotate accelerometer acceleration by the inverse of the estimated orientation, then take the z component.
            - Add a fixed variance vector to covariance, rotated by orientation (assume Var(orientation) = 0 for simplicity).
        - Altitude:
            - use barometer reading and variance from the datasheet.

c. Control Algorithm:
    - Subtract current orientation quaternion from target orientation.
    - Convert error quaternion to body-frame deviations.
    - Use PID controllers to calculate motor thrust.
    - Opposing motors fight to converge pitch/roll.
    - Diagonal motors apply counter-torques to converge yaw.

Hardware Components

- Components included in the Arduino kits:
    - Arduino Uno R3
    - MPU6050
    - BMP388
    - Resistors, capacitors, and wires
- Components we collected from IEEE lab/SecLab/family friends:
    - Raspberry Pi 3B
    - 4x ESCs
    - Buck/boost converter
    - LM7805 Voltage Regulator
    - 3 Cell LiPo Battery
    - Flight controller
    - Propellers (one was bent)
    - Quadcopter frame
    - Perforated board

- Components we needed to obtain:

- Collet-style propeller adaptors, $10 for 4 on Amazon
- leaded solder, $26 for a fancy roll on amazon
- External flux because the flux in the solder wasn't enough for the pads, $7 on amazon
- Electronic stack mount, 3D printed
- Propeller guards, 3D printed

Design Timeline

During most of the lab sessions, we worked on the original drone frame and wrote code on the Arduino.

We discovered that all of our hardware needed to be changed, and switched from using an Arduino to a Raspberry Pi very late in the project, so we had to rewrite everything from scratch to work on the new hardware, including the I2C drivers.

T-14 days to showcase (last lab session)

We finally got our parts together, but the motors for the drone we had built didn't work. Also, the software would likely not run on an Arduino, which was bad because it meant we'd need to completely restart the hardware. We got family friends to mail us a new drone frame, motors, and ESCs from an old project of theirs.

Friday, T-6 days to showcase

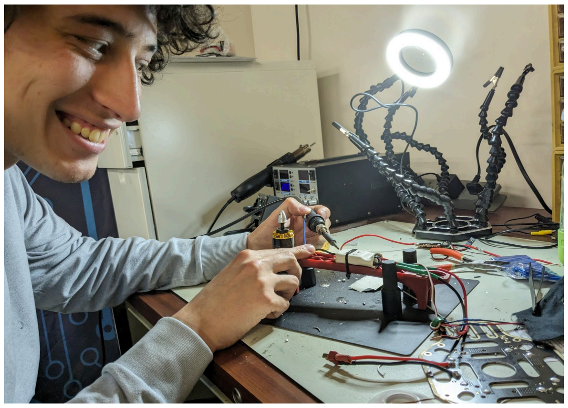
Revamped frame, removed old ESCs and residue, reassembled and soldered power. Validated ESCs using Arduino.

The parts arrived but didn't include the collets needed to mount the propellors on the motor shafts. They also didn't include a battery, which we had forgotten to order beforehand. We ordered a battery and propellor mounts from Amazon, but we had to wait till Monday for them to arrive.

One of the new motors had a different motor constant than the other motors, but we decided to deal with that problem in software.

The frame was missing screws and had mysterious, unlabelled ESCs that we weren't too sure about. We swapped them with the nicer ESCs from our previous drone, reassembled the frame with better cable management, and cleaned off adhesive residue and dirt.

Before soldering everything to the power distribution board, we validated that all 4 ESCs worked using a quick Arduino sketch.



We soldered the new ESCs to the frame's pads, but our iron couldn't melt the unleaded solder we had, so we had to use shit-tier leaded solder from an SMD rework kit, which didn't have a flux core.

Consequently, the wires barely stuck and had flaky connections.

Saturday, T-5 days to showcase

Started writing I2C drivers, Orientation code, and Kalman Filter (code day).

We started writing I2C drivers for the sensors in Java but didn't have our new electronics together yet, so we couldn't even test our drivers.

We also began writing control code, implementing Quaternion functionality, and setting up gyro rate integration. We also started writing a linear Kalman filter that we could later use to fuse our accelerometer and barometer readings to get altitude.

Sunday, T-4 days to showcase

Started electronics and more orientation + Kalman code.

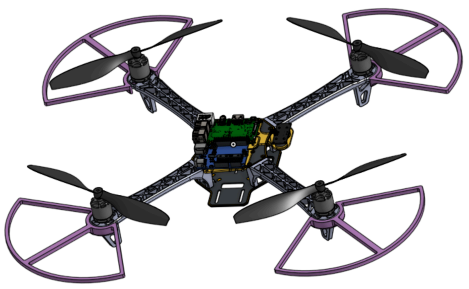
We realized we hadn't made a power supply for the Pi, so we designed one. We also realized that the drone's vibrations made the breadboard components loose, so would need to solder everything onto the perfboard.

Luckily, we found a roll of proper leaded solder, so we didn't need to use the SMD rework kit again. We soldered the sensors and power supply components together, using the kit perf board and a board we recovered from a botched Ethernet switch controller we found in the SecLab. The 30-gauge Kynar wire we were using kept breaking and was difficult to strip without cutting because our wire cutters were very blunt.

We wrote a Kalman filter to fuse the orientation from the accelerometer and gyro. We also worked on the orientation code.

Monday, T-3 days to showcase

Realized we needed prop inserts and made drone CAD. Some minor code.



The Amazon packages arrived, but the prop mounts had the wrong diameter, so we submitted 3D print jobs to the library for inserts that would allow the propellors to screw onto the motors' threads. While designing the prop mounts, we modeled the entire drone in CAD, taking the opportunity to plan the layout of our electronics stack. Placement of the IMU in the center of the frame would make our lives easier later.

We wrote a quick program to test which channel belonged to which flight controller axis.

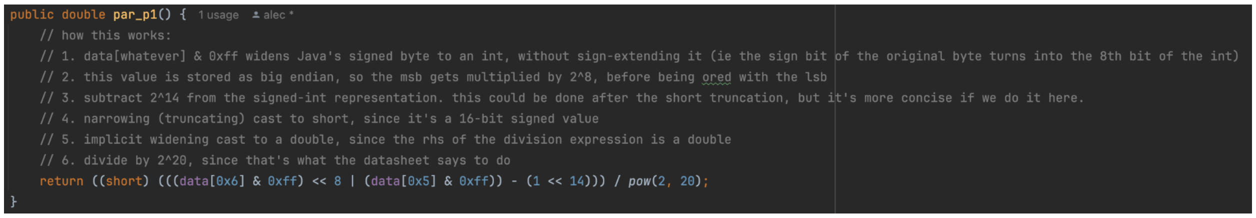
Tuesday, T-2 days to showcase

Finished electronics stack, soldering hell. Mounted electronics stack. Got orientation working with the renderer. Drivers working.

The prop inserts worked! We finally assembled our electronics stack and began mounting it onto the frame using a 3D-printed adapter. However, we soon realized that we had no bolts of appropriate diameter and length to mount our 3D-printed adaptor to the frame. Even after scavenging in the IEEE labs and Computer Security Lab, we couldn't find any bolts that fit. Instead of bolts, we used a combination of wire ties, which fixed the electronics in place vertically, and long, thin bolts taken from an Intel NUC case, which aligned the electronics mount with the frame in the x and y directions.

We accidentally mounted the electronics to the frame 90 degrees from the correct angle, which meant that the bottom of the frame obstructed the downward-facing ultrasonic rangefinder. We didn't want to reassemble the entire thing, so we scrapped the rangefinder from our design.

We discovered that our barometer and IMU drivers, which we previously had declared 'working', were very broken. We spent ages staring at the typo-laden BMP388 datasheet before realizing that the driver code had some missing brackets.



```java
public double par_p1() {  1 usage  alec *
    // how this works:
    // 1. data[whatever] & 0xff widens Java's signed byte to an int, without sign-extending it (ie the sign bit of the original byte turns into the 8th bit of the int)
    // 2. this value is stored as big endian, so the msb gets multiplied by 2^8, before being ored with the lsb
    // 3. subtract 2^14 from the signed-int representation. this could be done after the short truncation, but it's more concise if we do it here.
    // 4. narrowing (truncating) cast to short, since it's a 16-bit signed value
    // 5. implicit widening cast to a double, since the rhs of the division expression is a double
    // 6. divide by 2^20, since that's what the datasheet says to do
    return ((short) (((data[0x6] & 0xff) << 8 | (data[0x5] & 0xff)) - (1 << 14))) / pow(2, 20);
}
```

This is partially Java's fault for not supporting unsigned bytes, but if we had to do this in C, half of these operations would probably be UB or inadvertently cause truncations to bizarre, 9-bit Honeywell characters.

The BMP388 has two modes: 'normal' mode, where it periodically takes measurements and stores them in a FIFO, and 'forced' mode, where it takes measurements when they're explicitly requested over I2C. Normal mode was really broken, probably due to a weird race condition somewhere, but we eventually got tired of debugging it and decided to use forced mode instead.

Wednesday, T-1 days to showcase

Motor testing with Raspi. Failed motor characterization. Tried to figure out the error quaternion -> PID conversion. Stayed up in the SecLab.

We worked all night to debug our orientation software. We sent the drone's orientation to a computer via a UDP socket over Ethernet (with an IP assigned using a DHCP running on the Pi). We used JavaFX to visualize the drone's orientation.

Using a complementary filter to combine gyro rates with accelerometer data proved very difficult, especially since none of us were comfortable with quaternions. Orientation from the accelerometer is ambiguous since absolute azimuth cannot be determined without a magnetometer, which we didn't have. So, we had to extract global yaw from the gyro rates and add it back into the accelerometer roll/pitch.

Thursday, Day of showcase

Demonstrated “balance” in 1 axis

We finally hooked up the propellers to the drone and started working on code that would get the drone to fly now that all the helper classes were written and the sensor input was successfully converted into data useful to the flying algorithm. We applied the PID controller to send mapped throttle commands to the ESCs to balance on one axis. We started testing the code but ran into issues preventing all four ESCs from working correctly.

With hours left before the showcase, we called it quits and put together a demo of balancing on one axis using only two motors. At the showcase, we presented our real-time orientation renderer. The drone was too big to fly indoors anyway, but our classmates enjoyed seeing the live orientation demo. Pictured right is Alec explaining our drone to a very interested Northrop Gummy employee (the Skibidicopter has defence applications).

Friday, Day after showcase

Solved hardware PWM, ESC stability. Program stop state. Learned more about orientation.

One of our most mysterious unsolved bugs was the motors randomly briefly switching on when the program was running – even if the PWM duty cycle was 0. We eventually figured out that GC pauses during the on-period of our bit-bashed PWM signals pushed the pulse width past the threshold the ESCs considered to be 'on', making them briefly accelerate the motors. We fixed this by switching to ZGC, a fully concurrent GC built into HotSpot.

We also enabled the Raspberry Pi's two hardware PWM peripherals, so only 2 of the motors had to be bit-banged.

Our understanding of quaternions and their relationship with Euler angles and rotation matrices finally started to become coherent.

Software Design

Below is the pseudocode generated by ChatGPT from our Java source code (written by us). Here is a link to the full source code: https://github.com/rohanbandaru/ece5_quadcopter

Package drone:

MainLoop.java

```
FUNCTION MainLoop.main()
  PRINT "The program has started."

  TRY
    // Initializing sensors and motors
    Initialize barometer sensor (BMP388) on bus 1
    Initialize inertial measurement unit (MPU6050) on bus 1
    Initialize front-right motor (FRONT_LEFT_GPIO) with power 1.1
    Initialize back-left motor (BACK_RIGHT_GPIO) with power 1.1

    // Initializing orientation and altitude fusion
    Initialize orientation tracker (Orientation)
    Calibrate IMU with gravity vector (Vector3.K) is up
    Initialize altitude fusion (AltitudeFuser)

    // Initializing PID controller for pitch balancing
    Initialize PID controller for pitch balancing with parameters (0.5, 0, 0)

    // Initialize orientation from accelerometer
    Initialize orientation using accelerometer readings
    Store initial orientation as target orientation

    PRINT "Initialization completed."

    // Setting up network communication
    Open Datagram channel for UDP communication
    Set remote address to "10.42.42.2" and port 4444

    // Main loop
    WHILE Not interrupted by thread DO
      // Time update
      Calculate time difference since last update

      // Read sensor data
      Read accelerometer and gyroscope data from IMU
      Read altitude and variance from barometer

      // Update orientation and altitude fusion
      Update orientation using sensor data
      Update altitude fusion using sensor data

      // Calculate orientation error
      Calculate orientation error relative to target orientation
      Apply transformation to align with quadcopter frame

      // Calculate pitch, roll, and yaw errors
      Calculate pitch, roll, and yaw errors from orientation error

      // Apply PID correction for pitch balancing
      Calculate correction using PID controller
      Set throttle value

      // Print debug information
      PRINT "Time step:", time step, "Pitch error:", pitch error, "Correction:",
correction

      // Prepare and send sensor data over UDP
      Prepare sensor data packet
      Send sensor data packet to remote address

      // Update last update time
      Update last update time
    END WHILE

  END TRY
END FUNCTION
```

Package math:

FFT.java

```
FFT class:
  Methods:
    - fft(x: Complex[]): Complex[]
      - n = length of x
      - Base case: If n is 1, return array containing only x[0]
      - Check if n is a power of 2, throw exception if not
      - Compute FFT of even terms:
        - Create array even with size n/2
        - Fill even with even-indexed elements from x
        - Compute FFT of even using recursive call
      - Compute FFT of odd terms:
        - Reuse array odd by assigning it to even
        - Replace elements in odd with odd-indexed elements from x
        - Compute FFT of odd using recursive call
      - Combine evenFFT and oddFFT:
        - Initialize array y with size n
        - For k from 0 to n/2:
          - Compute kth angle
          - Compute wk using cosine and sine of kth angle
          - Compute y[k] by adding evenFFT[k] and wk times oddFFT[k]
          - Compute y[k + n/2] by subtracting wk times oddFFT[k] from evenFFT[k]
      - Return y

    - ifft(x: Complex[]): Complex[]
      - n = length of x
      - Create array y with size n
      - Take conjugate of each element in x and store in y
      - Compute forward FFT of y
      - Take conjugate of each element in y again
      - Divide each element in y by n
      - Return y
```

KalmanFilter.java

```
KalmanFilter class:
  Attributes:
    - observation: RealMatrix (H)
    - control: RealMatrix (B)
    - P: RealMatrix
    - x: RealMatrix

  Constructor:
    - KalmanFilter(observation: RealMatrix, control: RealMatrix)
      - Set observation and control matrices
      - Initialize P as identity matrix with dimensions of observation matrix
      - Initialize x as zero matrix with dimensions of observation matrix
  Methods:
    - predict(dynamics: RealMatrix, u: RealMatrix): void
      - Multiply dynamics matrix with x and add control multiplied by u to update
state estimation
      - Update P using dynamics matrix
      - Synchronize method

    - correct(y: RealMatrix, variance: RealMatrix): void
      - Compute innovation as difference between observed and predicted state
      - Compute innovation covariance
      - Compute Kalman gain
      - Update state estimation x and covariance P using Kalman gain, innovation, and
observation matrices
      - Synchronize method

    - state(): double[]
      - Return the state vector as an array
```

PID.java:

```
PID class:
  Attributes:
    - P_Gain: double
    - I_Gain: double
    - D_Gain: double
    - reset: boolean
    - lastError: volatile double
    - errorSum: double

  Constructor:
    - PID(kP: double, kI: double, kD: double)
      - Initialize P_Gain, I_Gain, and D_Gain with provided values

  Methods:
    - correction(dt: double, error: double): double
      - Compute proportional (P), integral (I), and derivative (D) terms
      - If reset flag is true, handle first iteration
      - Update error sum and compute derivative only if not first iteration
      - Update lastError
      - Return sum of P, I, and D terms

    - reset(): void
      - Reset error sum and set reset flag to true

    - setConstants(P: double, I: double, D: double): void
      - Set P_Gain, I_Gain, and D_Gain with provided values
```

ZFilter.java:

```
ZFilter Class:
  Attributes:
    - response: ImpulseResponse
    - a: Complex[]
    - T: double
    - index: int
  Constructor:
    - ZFilter(response: ImpulseResponse, order: int, t: double)
      - Initialize response, a, and T with provided values
      - Initialize index to 0

  Methods:
    - filter(value: double): void
      - Store the given value in the circular buffer a
      - Update the index for circular buffer wrapping

    - value(): double
      - Perform Fast Fourier Transform (FFT) on a
      - Apply filter gain to each frequency component (converts frequency to j*omega,
then bilinear transform to convert from s-domain to z)
      - Perform Inverse Fast Fourier Transform (IFFT) on filtered FFT result
      - Return the last value of the IFFT result

  Interface:
    - ImpulseResponse
      Methods:
        - gain(z: Complex): double
          - Abstract method to compute the gain for a given complex number z
```

Package pose:

AltitudeFuser.java

```
AltitudeFuser Class:
  Attributes:
    - altitudeFilter: KalmanFilter

  Constructor:
    - AltitudeFuser()
      - Initialize altitudeFilter using KalmanFilter with observation and control
matrices

  Methods:
    - update(dt: double, verticalAccel: double, accelVariance: double,
barometerAltitude: double, altitudeVariance: double): void
      - Compute state transition matrix
      - Predict the next state using KalmanFilter's predict method
      - Correct the prediction using KalmanFilter's correct method

    - altitude(): double
      - Return the altitude value from the Kalman filter state

    - verticalVelocity(): double
      - Return the vertical velocity value from the Kalman filter state

    - verticalAccel(): double
      - Return the vertical acceleration value from the Kalman filter state
```

Orientation.java

```
Orientation Class:
  Attributes:
    - orientation: volatile Quaternion
    - globalAccel: volatile Vector3

  Methods:
    - update(dt: double, gyroRates: Vector3, bodyAccel: Vector3, totalThrust: double): void
      - Compute orientation quaternion using gyroscopic rates and accelerometer readings
      - Update global acceleration based on body acceleration and orientation quaternion

    - initFromAccel(bodyAccel: Vector3): void
      - Initialize orientation quaternion from gravity accelerometer readings

    - compensateAccel(bodyAccel: Vector3, totalThrust: double): Vector3
      - Compensate for accelerometer readings considering motor thrust and drag
      - Adjust compensated acceleration to have a magnitude closer to 1

    - fromGyroRates(dt: double, gyroRates: Vector3): Quaternion
      - Compute quaternion representing rotation from gyroscopic rates
```

Package sensors:

BM388.java

```
BMP388 Class:
  Attributes:
    - delegate: I2CDevice
    - calibration: CalibrationData
    - zeroAltitude: double
    - pressureOversample: PressureOversample
    - iir: IIR

  Methods:
    - BMP388(delegate: I2CDevice): BMP388
      - Constructor to initialize BMP388 with provided I2CDevice

    - withDefaults(controller: int): BMP388
      - Static factory method to create BMP388 with default settings

    - read(): Reading
      - Read temperature and pressure data from BMP388 sensor
      - Compensate temperature and pressure readings
      - Compute altitude from pressure reading

    - temperatureVariance(): double
      - Calculate temperature variance

    - pressureVariance(): double
      - Calculate pressure variance based on oversampling and IIR filtering

    - configureOversampling(pressureOversample: PressureOversample): void
      - Configure pressure oversampling setting

    - configureIIR(iir: IIR): void
      - Configure IIR (Infinite Impulse Response) filter setting
    - setFifoConfig(config: FifoConfig): void
      - Configure FIFO (First In, First Out) settings

    - compensatedPressure(uncomp_press: int, temperature: double): double
      - Calculate compensated pressure based on uncompensated pressure and temperature

    - compensatedTemperature(uncomp_temp: int): double
      - Calculate compensated temperature based on uncompensated temperature

    - rawPressure(): int
      - Read raw pressure data from BMP388 sensor

    - rawTemperature(): int
      - Read raw temperature data from BMP388 sensor

    - readData(register: int, length: int): byte[]
      - Read data from specified register with given length

    - dataReady(): boolean
      - Check if sensor data is ready for reading

    - close(): void
      - Close the BMP388 sensor

BMP388.Inner Classes:
  - FifoConfig: record
    - Constructor
    - Static method: disabled()

  - PressureOversample: enum
    - Enum constants
    - variance(iir: IIR): double

  - IIR: enum
    - Enum constants

  - CalibrationData: record
    - Constructor
    - Getter methods for calibration parameters

  - Register: enum
    - Enum constants

  - Reading: record
    - Constructor
    - altitude(): double
    - altitudeVariance(): double
    - toString(): String
```

MPU6050.java

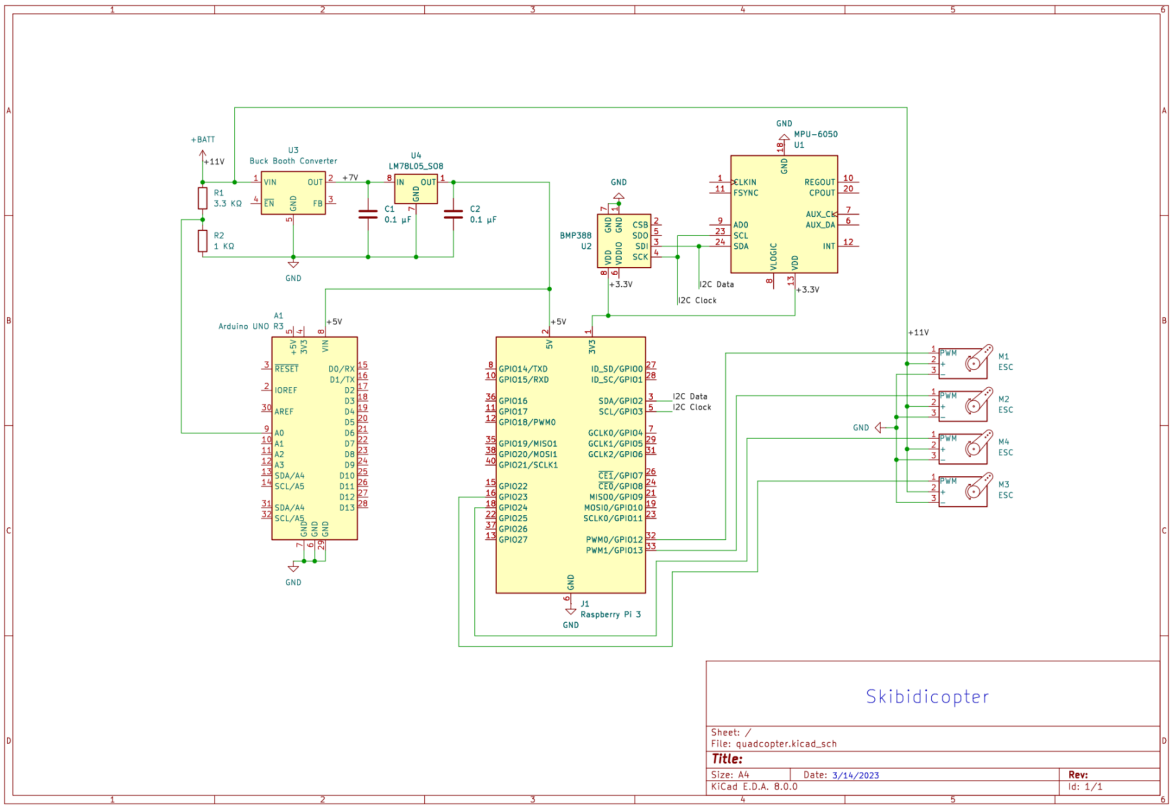
```
MPU6050 Class:
  Attributes:
    - delegate: I2CDevice
    - accelSpirit: Vector3
    - gyroSpirit: Vector3

  Constants:
    - G: double
    - GYRO_VARIANCE_VAL: double
    - GYRO_VARIANCE: Vector3
    - ACCEL_VARIANCE_VAL: double
    - ACCEL_VARIANCE: Vector3
    - DEFAULT_ADDRESS: int
    - OTHER_ADDRESS: int
    - GYRO_SENSITIVITY: double
    - ACCEL_SENSITIVITY: double
    - GYRO_CONVERSION: double
    - ACCEL_CONVERSION: double
    - TEMPERATURE_DIVISOR: double
    - TEMPERATURE_OFFSET: double
    - CALIBRATION_COUNT: int

  Methods:
    - MPU6050(controller: int)
      - Constructor to initialize MPU6050 with provided I2C controller

    - configure(): void
      - Configure MPU6050 sensor with default settings

    - calibrate(up: Vector3): void
      - Perform sensor calibration using provided up vector

    - read(): Reading
      - Read temperature, gyro, and accelerometer data from MPU6050 sensor

    - readGyro(): Vector3
      - Read gyroscope data from MPU6050 sensor

    - readAccelerometer(): Vector3
      - Read accelerometer data from MPU6050 sensor

    - readTemperature(): double
      - Read temperature data from MPU6050 sensor

    - processRawAccel(x: short, y: short, z: short): Vector3
      - Process raw accelerometer data and return Vector3

    - processRawTemperature(raw: short): double
      - Process raw temperature data and return temperature in degrees Celsius

    - processRawGyro(x: short, y: short, z: short): Vector3
      - Process raw gyroscope data and return Vector3

    - writeConfiguration(configurationName: String, register: byte, data: byte): void
      - Write and verify a device configuration value

    - readArray(startAddress: int, layout: MemoryLayout, count: int): MemorySegment
      - Read array of data from MPU6050 sensor
    - close(): void
      - Close the MPU6050 sensor

MPU6050.Inner Classes:
  - Registers: interface
    - Constants for register addresses in MPU6050 sensor

  - RegisterValues: interface
    - Constants for register values in MPU6050 sensor

  - Reading: record
       - Constructor
```

Package display:

OrientationRenderer.java:

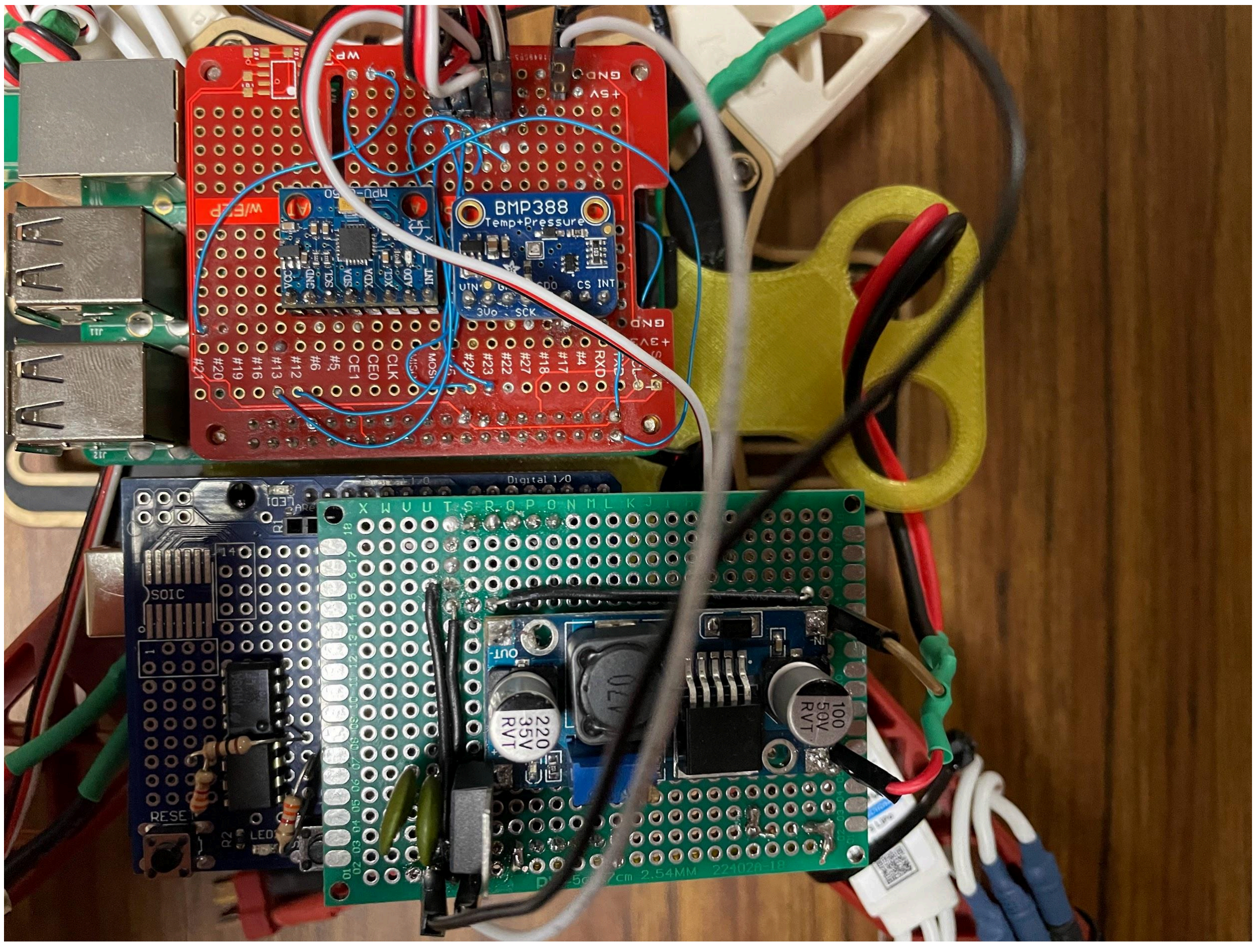
```
OrientationRenderer class (JavaFX Application):
  Attributes:
    - rotateY: int
    - rotateX: int
    - q: Quaternion
    - v: Vector3

  Methods:
    - start(primaryStage: Stage): void
      - Set primaryStage resizable to false
      - Instantiate block with dimensions 7x1x5 and red color
      - Set draw mode of block to line
      - Instantiate pointer with dimensions 0.1x0.1x2
      - Set draw mode of pointer to fill
      - Instantiate lines marking x, y, and z axes
      - Create and position camera with proper rotations and translation
      - Create root group and add camera, block, pointer, and axes lines
      - Create SubScene with root group and set camera
      - Create scene with root group
      - Set scene to primaryStage and show primaryStage
      - Create RotateTransition to continuously update block rotation
      - Set event handler to update block rotation based on quaternion and vector
      - Play RotateTransition

    - getAxisAngle(vec: double[]): AxisAngle
      - Calculate theta, Ax, Ay, Az, and angle from given quaternion vector
      - Return AxisAngle object

    - getPos(): double[]
      - Get quaternion and vector
      - Return quaternion components

  Main method:
    - Create virtual thread to receive quaternion and vector updates via
DatagramChannel
    - Launch JavaFX application with provided arguments
```

Circuit Schematics



Note: on the perf boards, we had 1 dual op-amp and 2 single op-amps configured to 50x gain, connected to the Arduino, which were meant to amplify current measurements made by sampling the voltage drop across the motors' traces on the drone frame PCB. We also had a voltage divider to measure the battery voltage. These were connected to the analog inputs on the Arduino, to provide the Arduino's ADC readings to the Pi by setting the Arduino up as an I2C slave. Unfortunately, we didn't have time to write the communication between the Arduino and Pi, but we tested all the circuitry and it seemed to work.

Circuit prototype



Testing

- Used multimeter on different parts of the circuit to test if the circuit is constructed properly and if the current flowing through is of the right magnitude.
- Used sensor data printed to the terminal to test the altitude estimation functionality.
- Used the OrientationRenderer application to test if the orientation and complementary filter code were written correctly.
- The motortest class is written to test and figure out how to arm each ESC correctly and the values needed to get it to spin at the right speed.
- A modified version of the Mainloop is used to test multiple propellers at the same time and ensure the corrections are working and the axes are balanced.

Conclusion, Next Steps

Our biggest problem was that we convinced ourselves that we had accomplished a lot more than we actually had, and generally had terrible time management. Ninety-nine percent of the work we needed to do was integration, and we didn't start any of that until a week before the deadline. When we did this, we realized that many of the assumptions we had made previously were incorrect and that it amounted to almost zero real progress.

Despite the many cascading debacles, we delivered a decent final result and won “Most Difficult Project” at the fair. We all learned a lot and aim to keep working on the quadcopter in the future. Our plans include firstly getting it to autonomously hover at a fixed altitude, possibly incorporating the rangefinder or perhaps computer vision to target waypoints and perform automatic landings. Staying up all night in the lab and breathing solder fumes with buddies was fun.