A 3D Inverted Pendulum For Testing Control Algorithms (June 8, 2017)

Athina Ebert, Undergrad, UWEE, Kyle Hess, Undergrad, UWEE,
Ryan Mills, Undergrad, UWEE, XiaoCheng Zhang, Undergrad, UWEE

***Abstract*—This report aims to convey the design and approach of our capstone project, quick-balancing cube (QB^3). A three dimensional inverted pendulum. It will detail our initial approach to the project, problems we encountered, achieved functionalities, hardware and software components, and possible future improvements.**

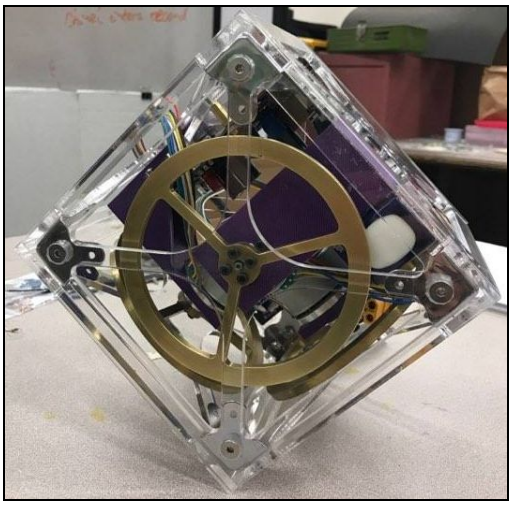


Figure 1: Completed cube assembly on edge.

Index Terms - Angular position, Controls, Counter-torque, Inverted pendulum, Linear Quadratic Regulator, Microprocessor, Moment of inertia, Motor drive, Wireless.

I. INTRODUCTION

Our goal was to create an inverted pendulum in the form of a cube. An inverted pendulum is an object with a center of mass above its pivot point. Having the center of mass above the pivot point makes for an unstable system. The object will tend to fall over when it is not perfectly balanced, or when no external force is applied. To balance the object and keep it stable, a controlled counter force must be applied. Generating the counter force for our cube will be accomplished by using the concept of the conservation of angular momentum.

Most inverted pendulum systems exist in two dimensions. For example, a two-wheeled balancing robot or an extended arm with a pivot point at one end.

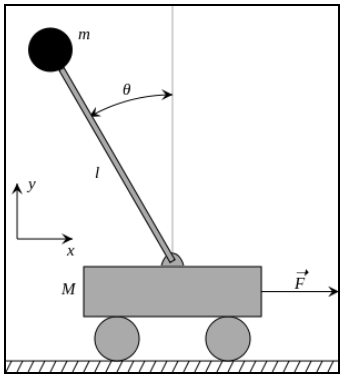


Figure 2: Moving cart with inverted pendulum.
www.wikipedia.com

These systems only require measurement and correction of angular position in a single plane whereas our cubic system requires the same type of control, but in three individual planes simultaneously. This poses an extra challenge compared to most other inverted pendulum problems because each of the three axes are inherently coupled together. This is a problem that is not frequently addressed in more simplified systems. We attempted to solve this complex problem by breaking it into three simple inverted pendulums problems. Achieving this goal would prove that some coupled systems could be simplified into more manageable problems. This is not the case.

Controlling the angular position of the cube will be achieved through a counter-torque generated by the three internal motors, each

spinning on an axis perpendicular to the others (x, y, z). These motors will accelerate high-inertia flywheels in order to generate a counter-torque on the cube body. The torque will be controlled such that it precisely cancels the rotation of the cube on one of its axes. Torque direction will be determined by the gyroscope sensor feedback and control algorithm.

II. BACKGROUND & RELATED WORK

Similar projects and research have been done by Faculty of Electronics, Telecommunications and Information Technology from University POLITEHNICA of Bucharest and Department of Mechanical and Process Engineering from ETH Zurich. The general approach of our project is similar: receive data from multiple IMUs to determine the angular position and velocity of the cube, and use motors to drive three reaction wheels to adjust the position of the cube. The detailed solution of our project might be different, i.e. we might use different IMUs, motor drives and control loops. Despite that, their research has been an important reference to us.

III. TECHNICAL APPROACH

A. Goals

The goals for the final product design are the following:

- The cube will be able to balance on a specified edge or corner.
- The cube must operate without any external inputs or physical interference
- The cube must be completely wireless while operating.

We chose to abandon the requirement of a "jump-up" feature due to the additional mechanical systems that were needed. Starting the cube in an unstable position proved challenging enough.

B. System

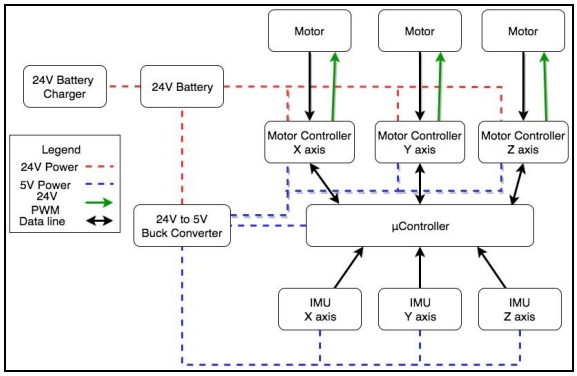


Figure 3: Project system schematic diagram,

The initial design of our project relied on the input of the angular velocity of three motors, values sent from three accelerometer-gyroscope IMUs to the microprocessor via I^2C serial bus. The microprocessor will then calculate the angular position and velocity of the cube from those values, then calculate the necessary current for each motor, and output a pulse width modulated (PWM) signal to the motor drivers.

C. Hardware

We are using the BNO055 IMU to measure the angular position of the cube.

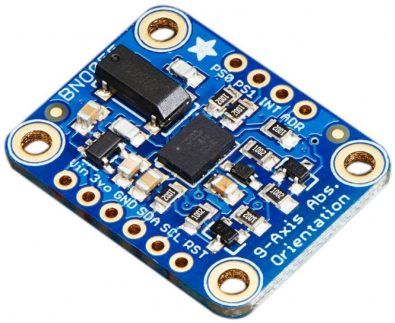


Figure 4: BNO055 'Absolute' IMU

It has nine degrees of freedom (DOF) with three-axis accelerometer, a three-axis gyroscope, a magnetometer (compass), and communicates via I^2C communication protocol. The data provided turns out to be much more stable and accurate than the six DOF MPU-6050 IMUs we initially chose.

In order to detect the exact angular position of the cube, we used three BNO055s, each to determine the pitch and roll angle of one face of the cube. However, the python library we found for the IMU was not designed for multiple IMU input due to the lack of an IMU port selection function. The original I^2C selection method was to get the default I^2C bus (*get_i2c_device()*), in which case we can only use I^2C-1.

```
# Use I2C if no serial port is provided.
# Assume we're using platform's default I2C bus if none is specified.
if i2c is None:
    import Adafruit_GPIO.I2C as I2C
    i2c = I2C
# Save a reference to the I2C device instance for later communication.
self._i2c_device = i2c.get_i2c_device(address, **kwargs)
```

Figure 5: Original I2C Setup Function

To solve that, we added a new parameter "bus" in the initialization function, along with using another method that allows us to select I^2C buses (*Device()*).

```
# Use I2C if no serial port is provided.
# Assume we're using platform's default I2C bus if none is specified.
if i2c is None:
    import Adafruit_GPIO.I2C as I2C
    i2c = I2C
# Save a reference to the I2C device instance for later communication.
if bus == -1:
    self._i2c_device = i2c.get_i2c_device(address, **kwargs)
else:
    self._i2c_device = i2c.Device(address=address, busnum=bus)
```

Figure 6: Updated I2C Setup Function

The updated library can be found at https://github.com/ryan1mills/QB-3/tree/master/Adafruit_Python_BNO055_QB%5E3.

Figure 7: Nanotec DF45 BLDC Motors

For the motors, we chose to use the Nanotec DF45. This is a 24V three phase brushless DC (BLDC) motor with Hall effect sensor outputs. We chose this motor due to its small size, high torque output and relatively low operating voltage (Appendix A). This motor has a maximum speed of 4840 RPM, a maximum power consumption of 65 watts and a torque rating of 0.13Nm. Which is more than enough to quickly accelerate our flywheels.

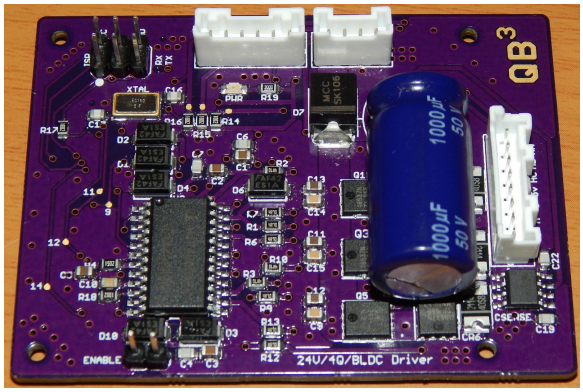


Figure 8: Complete DF45 Motor Driver PCB

To properly drive BLDC motors, we needed a digital motor driver board that operated in four quadrant (4Q) motoring (complementary switching). The only commercial 4Q drivers that matched our motors were either too expensive for our budget, or were too large to be physically located within the wireless cube. To overcome this problem, and since we had done so in an electric drives class, we designed our own motor drivers. These meet all of the requirements to drive the Nanotec DF45 BLDCs. These drivers are even capable of delivering a current sensor output signal to the microprocessor. This is crucial for accurate control of the motor torque. Current control is achieved by measuring the current delivered to each motor with the ACS712 Hall effect current transducer. This value is then utilized within our control algorithm to determine the necessary corrections that must be made to the motor output signals. The motor driver uses an on-board Atmel Atmega328p MCU to control the commutation of the motor phases based on inputs from the DF45 Hall effect sensors (Appendix D).

The various components inside of our system run at either 3.3V, 5V, 12V, or 24V. In order to power the cube from a 6S LiPo battery a voltage conversion board was designed. To create a

regulated 5V and 12V a switching buck regulator was used with the necessary capacitors, diodes, and inductors added. Because the battery voltage can vary between 21V and 25.2V a buck-boost controller was needed to achieve a consistent 24V.

For the microprocessor, we are using a Beaglebone Green Wireless. This allows for wireless communication, which has significant advantages when programming the processor while it is at the center of an enclosed cube. It also has several accessible GPIO, ADC, and I2C pins to allow for control of our system.

The BeagleBone being the brain of the cube has many connections coming and going from it. While it does have headers that wires can easily connect to it, the wires are loose and may come out while in operation. To solve this a PCB that connects to the headers securely was created. It has connectors that can easily connect the motor driver, power distribution board, and three IMUs. Also, since the hall effect sensors output 5V and the beaglebone uses 3.3V logic, a logic voltage converter was designed and used. Further, the current sensor, ACS712, can output up 5V, but the ADC on the BeagleBone is limited to 1.8V. Thus, a voltage divider was needed to reduce the voltage output of the current sensor.

The physical aspect of this challenge, counter torque, would be solved using three flywheels, one for each axis. We knew the general shape of the wheels we wanted and optimized the size to fit within the cube, but still have most of the mass as far from the center as possible. To do so we initially chose steel because it is an inexpensive and dense material. When we brought our design to be machined at the UW Aeronautics and Astronautics department, the shop manager suggested we switch to brass because it is easier to machine and he was able to provide the material from their scraps.

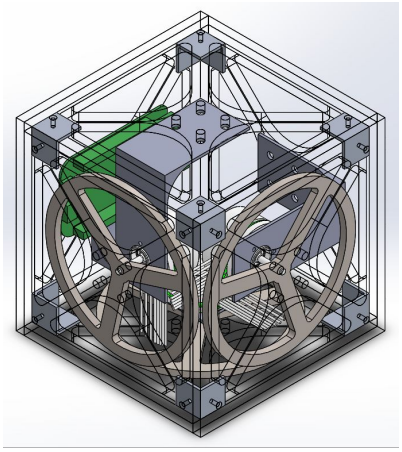


Figure 9: Model of acrylic cube casing, 3D-printed motor mount, motors and brass flywheels

Finally, we needed the physical cube structure. The design began with the following initial requirements:

- Encases all hardware components
- Relatively inexpensive materials
- Reasonable to fabricate given our limited resources and experience
- Robust enough to handle the stress of our operation

For the outer case we decided on acrylic because it is easy to cut into any shape or size in two-dimensions using a laser cutter. The next consideration was how the motors would be mounted within the cube such that there is one for each axis. To avoid interferences, each motor needed to be mounted to a separate perpendicular inner face. The resulting shape of the mount would have been very difficult to machine and heavy. For this, we decided to 3D print them with as much structural integrity as possible to make of for using a softer material. The remaining challenge was to mount all of the electrical components (beaglebone and cape, IMUs, driver boards, power distribution, battery) within the cube such that they are secure and semi-accessible. This was worked out near the end of the project when all of the rest of the hardware was finalized.

D. Software

No specific programming software is required for the Beaglebone Green. Right now we are using a combination of the Cloud9 IDE, a browser-based IDE and terminal, and an SSH terminal on a laptop for programming purposes. Python and C are the primary programming languages for the BeagleBone, and we have experimented with both. Most of the control algorithm software for the cube was written in Python due to the abundance of libraries for our IMUs and other GPIO functionality.

E. Control loop

We planned to use an LQR control loop algorithm for the controls system [1]. The state space representation of the LQR control loop is as follows:

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

Where: $x(t)$ is the current state of the system, A is the 'Dynamics matrix,' $u(t)$ is the input to the system (applied motor current), and $y(t)$ is the output.

$$x(t) = \begin{bmatrix} \theta_B \\ \dot{\theta}_B \\ \dot{\theta}_W \end{bmatrix}$$

Figure 10: System state matrix.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ \frac{g(M_B L_B + M_W L_W)}{I_B + M_W L_W^2} & -\frac{C_B}{I_B + M_W L_W^2} & \frac{C_W}{I_B + M_W L_W^2} \\ -\frac{g(M_B L_B + M_W L_W)}{I_B + M_W L_W^2} & \frac{C_B}{I_B + M_W L_W^2} & -\frac{C_W(I_W + I_B + M_W L_W^2)}{I_W(I_B + M_W L_W^2)} \end{bmatrix}$$

Figure 11: System 'dynamics' matrix

$$B = \begin{bmatrix} 0 \\ -\frac{k_m}{I_B + M_W L_W^2} \\ -\frac{k_m(I_W + I_B + M_W L_W^2)}{I_W(I_B + M_W L_W^2)} \end{bmatrix}$$

Figure 12: System input matrix

These matrices contain the parameters of the physical system that describe how the system behaves under certain conditions. Where:

Table 1. LQR Matrix Parameters

Name	Value	Unit
Reaction Wheel Mass (Mw)	0.175	Kg
Pendulum Mass (Mb)	2.7	Kg
Dist. From Pivot to CoG (Lb)	0.106	m
Dist. From Pivot to CoG of Wheel (Lw)	0.106	m
Moment of Inertia of Wheel+Motor (Iw)	8.689e-4	$Kg \cdot m^2$
Moment of Inertia of The Pendulum (Ib)	?	$Kg \cdot m^2$
Torque Constant (k_m)	0.0369	Nm/A

This control loop operates by predicting the future state of the system (x) and controlling the output parameters (motor current) to minimize the measured error. This is accomplished by continuously minimizing a quadratic cost function using parameters obtained by solving the Riccati differential equation. Luckily, MATLAB has built-in functions that do just that. All that we need to do is provide MATLAB with our dynamics matrix A and our input matrix B. MATLAB will then give us the parameters necessary for minimizing the quadratic cost function. This calculation will need to be performed about each of the three axes separately.

The variables in these matrices are as follows: M is mass, I is moment of inertia, L is length, g is gravitational acceleration, km is the torque constant of the motors and C is the coefficient of friction. The subscripts B and W indicate which part of the system the parameter relates to; "box" (cube) or "wheel" (reaction wheel) respectively.

IV. RESULTS & ANALYSIS

The construction process did not go as well as we expected. We originally decided to have three motor driver boards to control three motors. We purchased nine PCBs in total and

burnt out seven of them. However, the remaining PCB works perfectly fine, which enables the cube to balance on at least one of its edges.

A. Initial hardware troubles

The first motor driver PCBs that we had made were missing a single 5v connection to the MCU. This led to sporadic resets of the motor driver during operation, and ultimately, fried components and traces. The next PCBs should have worked fine, but voltage dips in the 5v supply had gone unnoticed and led to more random resets. This problem persisted for far too long without a proper diagnosis and set us back over two weeks. Our solution was to simply add in a 5v battery with a shorter ground loop.

B. First finished prototype testing

The first finished prototype contains two batteries, a power distribution board, BeagleBone Green, IMU, and three motors with attached flywheels. However, with only one functional motor driver, which means only one motor is operational.

Without knowing the moment of inertia of the cube along the pivot point, or the exact location of center mass, we were unable to apply an LQR algorithm and used a simpler PID control system instead. The cube was able to balance on one edge for several seconds and then slowly fall to one side as the motor speed saturated. Video of the cube balancing can be seen at https://youtu.be/Al67AnTL8aM.

C. Improvements to the controls

The initial PID parameters we used were determined by viewing the motor PWM signal duty ratio while moving the cube by hand. The values were increased from zero until the duty ratio changed rapidly in either direction. From here the response of the cube was observed, and the appropriate parameters were changed (tuned) for a more desirable action.

We found that quite large values for Kp and Ki were required to resist leaning in one direction or another. This results from the motor needing to accelerate quickly in either direction without speeding up in one direction continuously. The addition of a small Kd term helped eliminate oscillations, but also caused the motor to rapidly torque on the cube and expose weaknesses in the mounting brackets.

Unfortunately, the control system is very sensitive to the initial conditions of the cube. Each time the system runs it must start in an unstable, yet slightly balanced position to generate the set-point for the PID algorithm. If the starting position is slightly off to one side the motor will begin to accelerate until max speed is reached and the system cannot recover.

V. DISCUSSION

Comparing what we have achieved to the work from which we drew our inspiration would not be fair at this point. The work from the Cubli team clearly out-performed our own. However, our work does confirm the level of difficulty that is involved with actively balancing an unstable, nonlinear coupled system in three dimensions.

We are able to compare our results, for the most part, to those of E. S. Lupu [1] and her two-dimensional inverted pendulum problem. Our results confirm hers in the sense that tuning a PID control algorithm will never be perfect. The PID algorithm doesn't really 'know' anything about the system, it just tries its best. Lupu went a step further and conducted the same test with an LQR system as well, unfortunately we didn't get this far and cannot compare. We do assume however, that if we continued this project, and implemented LQR, we would see results similar to hers: large improvements over PID. This is due to the fact that LQR uses known parameters of a system to predict its response, and not just tuned values that produce a decent output.

VI. IMPACT

Our project has various significance in the field of Electrical Engineering and Astronautics. Using flywheels to adjust angular position of a certain object can be widely utilized in outer space tasks. Meanwhile, similar applications can make two-wheel automobile to be more practical. With a device that allows a car to

stand perfectly balanced on its wheels, it would be a lot easier for a car to climb up and down a hill.

VII. FUTURE WORK

The next step for this project is to finally implement a working LQR control algorithm. In order to do this, a few things will need to happen. First, the motor drivers must be redesigned with onboard power conversion, or commercial units must be purchased. Next, proper measurements of the system's center of gravity and inertia must be made. This information is crucial for the next step: A proper simulation. A mathematical model that closely resembles this unstable system must be developed for designing LQR.

If possible, reinforce the frame of the cube. It is surprising how much torque can be generated by such small motors, and the forces they impart flex parts of the system and introduce error.

VIII. CONCLUSION

In summary, we set out to balance a 3-dimensional, unstable system using the principles of conservation of angular momentum, and linear quadratic regulator control. Due to our approach of designing and building all of our own hardware we ran into many obstacles and time constraints. This led to a simplification of our project goals. In the end we were able to accomplish the following: Allow the cube to balance on a single edge for a period of time, operate completely wirelessly, and utilize our custom hardware. Imperfect project management throughout the process also contributed to the obstacles that we faced. Developing a clear project outline with good back-up plans initially is key. Overall this project was a success in terms of a robotics and controls problem.

IX. ACKNOWLEDGMENTS

Without the help we received from the following individuals and institutions none of this would have been possible:

- The A&A Machine Shop for their donation of materials and CNC machining services.
- Ben Ferleger for his informative lectures on developing LQR control algorithms.
- Professor Howard Chizeck for his consulting and feedback at each step of the project.
- Professor Rich Christie for his class on electric motor drives (EE 453).
- The UW College of Engineering, and the Electrical Engineering Department for funding this project.
- Adafruit Industries for their assortment of Python libraries for the BeagleBone.
- Michael Muehlebach, Gajamohan Mohanarajah, and Raffaello D'Andrea for their inspiring work on Cubli.

X. REFERENCES

[1] E. S. Lupu, "Cubic Structure Capable of Balancing", Faculty of Electronics, Telecommunications and Information Technology, University POLITEHNICA of Bucharest, Bucureşti, Romania, 2015.

[2] M. Gajamohan, M. Muehlebach, T. Widmer, R. D'Andrea, "The Cubli, A Reaction Wheel Based 3D Inverted Pendulum", Department of Mechanical and Process Engineering, ETU Zurich, Zurich, Switzerland, Feb. 2011.

[3] M. Gajamohan, M. Muehlebach, I. Thommen, R. D'Andrea, "The Cubli, A Cube that can Jump Up and Balance", Department of Mechanical and Process Engineering, ETU Zurich, Zurich, Switzerland, Feb. 2011.

[4] M. Gajamohan, M. Muehlebach, R. D'Andrea, "Nonlinear Analysis and Control of a Reaction Wheel-based 3D Inverted Pendulum", Department of Mechanical and Process Engineering, ETU Zurich, Zurich, Switzerland, Feb. 2011.

XI. BUDGET

Description	Cost
Motor Driver PCB V1 (Never received)	$25.25
Motor Driver Stencil	$14.36
Acrylic Sheet	$25.84
Battery Charger	$55.89
Batteries	$77.82
AVR Programmer	$16.95
BeagleBone Green	$48.00

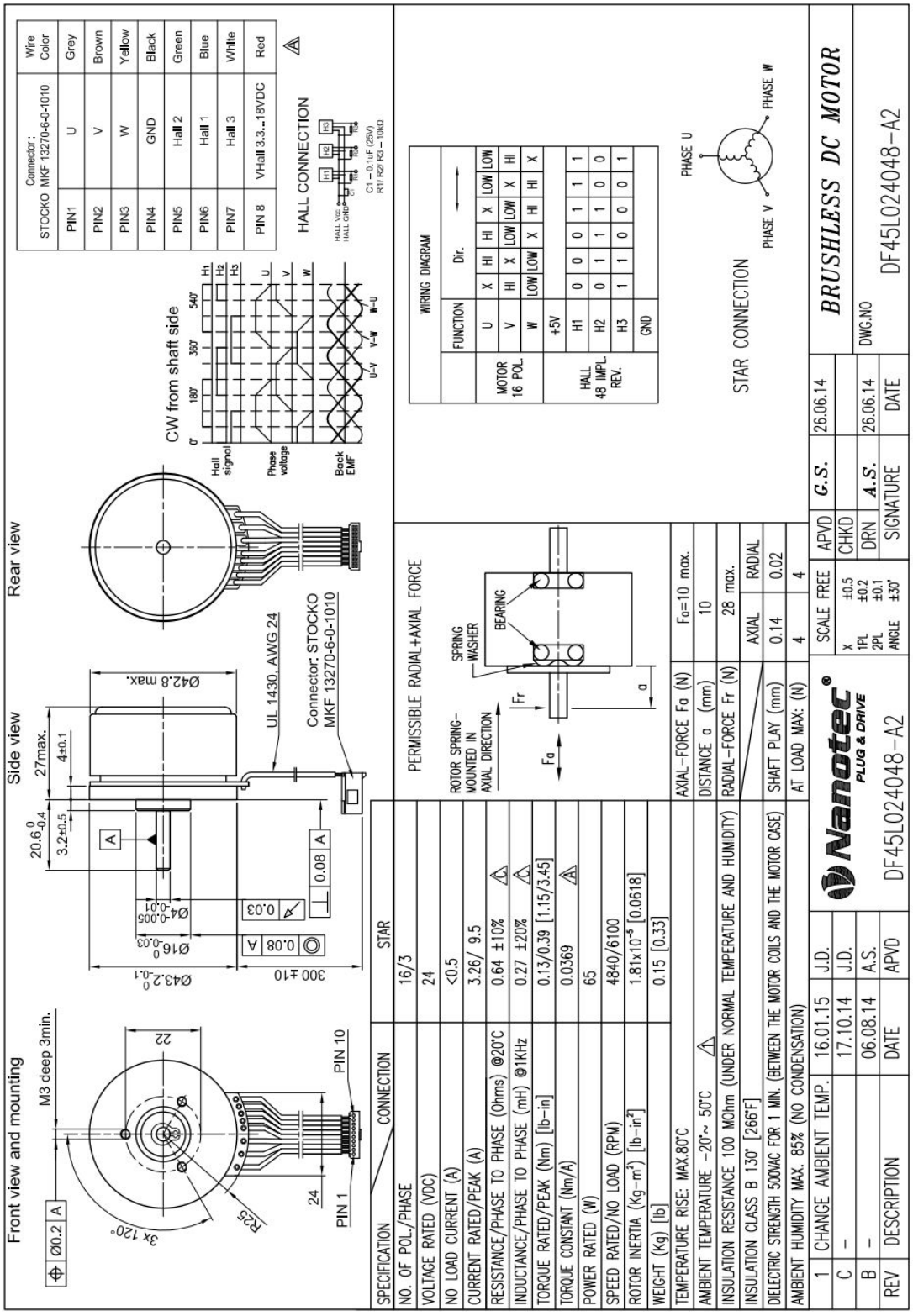
Brushless Motors	$348.99
3D Printer Filament	$22.99
IMU MPU 6050	$20.97
M3 x 10mm bolt	$7.34
Solder Paste	$15.54
3D Printer Filament Purple	$22.99
Angle Bracket	$9.11
M5 countersunk bolt	$11.59
M5 nylock bolt	$14.42
Double sided tape	$5.98
XT60 Connectors	$6.99
IMU	$30.51
Acrylic Sheet	$24.96
Center punch / Counter-sink bit	$25.59
Motor driver PCB V2	$61.50
Motor driver PCB V3	$64.00
Power Distribution PCB	$131.10
Driver Components	$4.99
Driver & BeagleBone Components	$316.89
ATMEGA328 & Resistors	$21.37
Power Dist & Driver Components	$359.66
Power & Driver Stencils	$33.23
Mounting Hub	$17.85
BeagleBone Cape PCB	$70.40
Cape components	$126.35
BeagleBone Cape stencil	$14.42
Electrical Components	$144.81
Electrical Components	$207.01
Header and Standoff	$20.98
M3 nyloc nuts	$6.36
M3 Countersunk bolts	$7.40
BNO055 IMU	$66.46
BeagleBone Black Wireless	$89.99
Motor driver PCB V4	$64.00
Fly wheels	$372.00
Symposium Poster	$46.00
Total Spent	$3,078.85

XII. RESOURCES

For this project we utilized various services, both on and off campus. One of the most helpful resources that we utilized was the CoMotion Makerspace in Fluke Hall. Their laser-cutter, 3D printers, PCB manufacturing equipment, and hand tools proved to be invaluable to us. The next most helpful resource was the Aeronautics & Astronautics machine shop. The shop donated brass for our flywheels and machined the flywheels for a very reasonable price. The next service, OSH Park custom PCBs, was critical for producing our own electronics hardware. Although this was not provided by the university, it was still a very useful resource. Finally, our lab space (B011) was perhaps the most important to us. Without a place to lock away our equipment and work indefinitely we would have struggled.

XIII. APPENDICES

A. Nanotec DF45 Motor Datasheet



B. Cube Balancing Code (Python)

```
import logging
import sys
import time
import math

import PID

from Adafruit_BNO055 import BNO055

import Adafruit_GPIO.I2C as I2C
import Adafruit_BBIO.PWM as PWM

#Reset the PWM signal for the motor driver.
#and start it at 50% (0 Torque).
PWM.stop("P9_14")
PWM.cleanup()
PWM.start("P9_14", 50, 20000, 0)

initCal = 0

#Initialize a PID loop object.
P = 25.0
I = 10.0
D = 5.0
myPID = PID.PID(P, I, D)
myPID.setSampleTime(0.0) #Sample the system as fast as possible.
myPID.SetPoint = 0.0 #Desired angle of the cube.
myPID.output = 50 #50% PWM.
out = 50

#Set max/min duty ratios of the motors.
maxPWM = 70
minPWM = 30


# Create and configure the BNO sensor connection.
bno = BNO055.BNO055(rst='P9_12', address=0x28, bus=2)
#bno2 = BNO055.BNO055(rst='P9_12', address=0x29, bus=2)
bno3 = BNO055.BNO055(rst='P9_12', address=0x28, bus=1)

# Enable verbose debug logging if -v is passed as a parameter.
if len(sys.argv) == 2 and sys.argv[1].lower() == '-v':
    logging.basicConfig(level=logging.DEBUG)

# Initialize the BNO055 and stop if something went wrong.
if not bno.begin():
    raise RuntimeError('Failed to initialize BNO055! Is the sensor connected?')

# Print system status and self test result.
status, self_test, error = bno.get_system_status()
print('System status: {0}'.format(status))
print('Self test result (0x0F is normal): 0x{0:02X}'.format(self_test))

# Print out an error if system status is in error mode.
if status == 0x01:
    print('System error: {0}'.format(error))
    print('See datasheet section 4.3.59 for the meaning.')

# Print BNO055 software revision and other diagnostic data.
sw, bl, accel, mag, gyro = bno.get_revision()
print('Software version:   {0}'.format(sw))
print('Bootloader version: {0}'.format(bl))
print('Accelerometer ID:   0x{0:02X}'.format(accel))
print('Magnetometer ID:    0x{0:02X}'.format(mag))
print('Gyroscope ID:       0x{0:02X}\n'.format(gyro))

print('Reading BNO055 data, press Ctrl-C to quit...')

#Start reading the sensors and controlling the motors.
runTime = time.clock()
while runTime < 10: #Run for 10.0 seconds


    # Read the Euler angles for heading, roll, pitch (all in degrees).
    heading, roll, pitch = bno.read_euler()

    # Read the calibration status, 0=uncalibrated and 3=fully calibrated.
    #sys, gyro, accel, mag = bno.get_calibration_status()

    if initCal < 5:
        myPID.SetPoint = pitch
        initCal += 1

    #Compute the new PID output based on the cube angle.
```

```
    myPID.update(pitch)

    #Set the motor duty ratio from 50%.
    out = 50 - myPID.output

    #Limit the motor duty ratio if above or below the max/min.
    if out > maxPWM:
        out = maxPWM
    if out < minPWM:
        out = minPWM

    #Send the duty motor duty ratio to the driver.
    PWM.set_duty_cycle("P9_14", out)
    runTime = time.clock()

#Set motor speed to 0 after running for a time
PWM.set_duty_cycle("P9_14", 50)
```

C. Generic PID Control Class (Python) - By: IVMECH MEKATRONİK

```
    # This file is part of IvPID.
    # Copyright (C) 2015 Ivmech Mechatronics Ltd. <bilgi@ivmech.com>
    #
    # IvPID is free software: you can redistribute it and/or modify
    # it under the terms of the GNU General Public License as published by
    # the Free Software Foundation, either version 3 of the License, or
    # (at your option) any later version.
    #
    # IvPID is distributed in the hope that it will be useful,
    # but WITHOUT ANY WARRANTY; without even the implied warranty of
    # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    # GNU General Public License for more details.
    #
    # You should have received a copy of the GNU General Public License
    # along with this program.  If not, see <http://www.gnu.org/licenses/>.

    # title           :PID.py
    # description     :python pid controller
    # author          :Caner Durmusoglu
    # date            :20151218
    # version         :0.1
    # notes           :
    # python_version  :2.7
    # ==============================================================================

    """Ivmech PID Controller is simple implementation of a Proportional-Integral-Derivative (PID) Controller in the Python
Programming Language.
    More information about PID Controller: http://en.wikipedia.org/wiki/PID_controller
    """
    import time

    class PID:
        """PID Controller
        """

        def __init__(self, P=0.2, I=0.0, D=0.0):

            self.Kp = P
            self.Ki = I
            self.Kd = D

            self.sample_time = 0.00
            self.current_time = time.time()
            self.last_time = self.current_time

            self.clear()

        def clear(self):
            """Clears PID computations and coefficients"""
            self.SetPoint = 0.0

            self.PTerm = 0.0
            self.ITerm = 0.0
            self.DTerm = 0.0
            self.last_error = 0.0

            # Windup Guard
            self.int_error = 0.0
            self.windup_guard = 20.0

            self.output = 0.0

        def update(self, feedback_value):
            """Calculates PID value for given reference feedback

            .. math::
                u(t) = K_p e(t) + K_i \int_{0}^{t} e(t)dt + K_d {de}/{dt}

            .. figure:: images/pid_1.png
               :align:   center

               Test PID with Kp=1.2, Ki=1, Kd=0.001 (test_pid.py)

            """
            error = self.SetPoint - feedback_value

            self.current_time = time.time()
            delta_time = self.current_time - self.last_time
            delta_error = error - self.last_error

            if (delta_time >= self.sample_time):
                self.PTerm = self.Kp * error
```

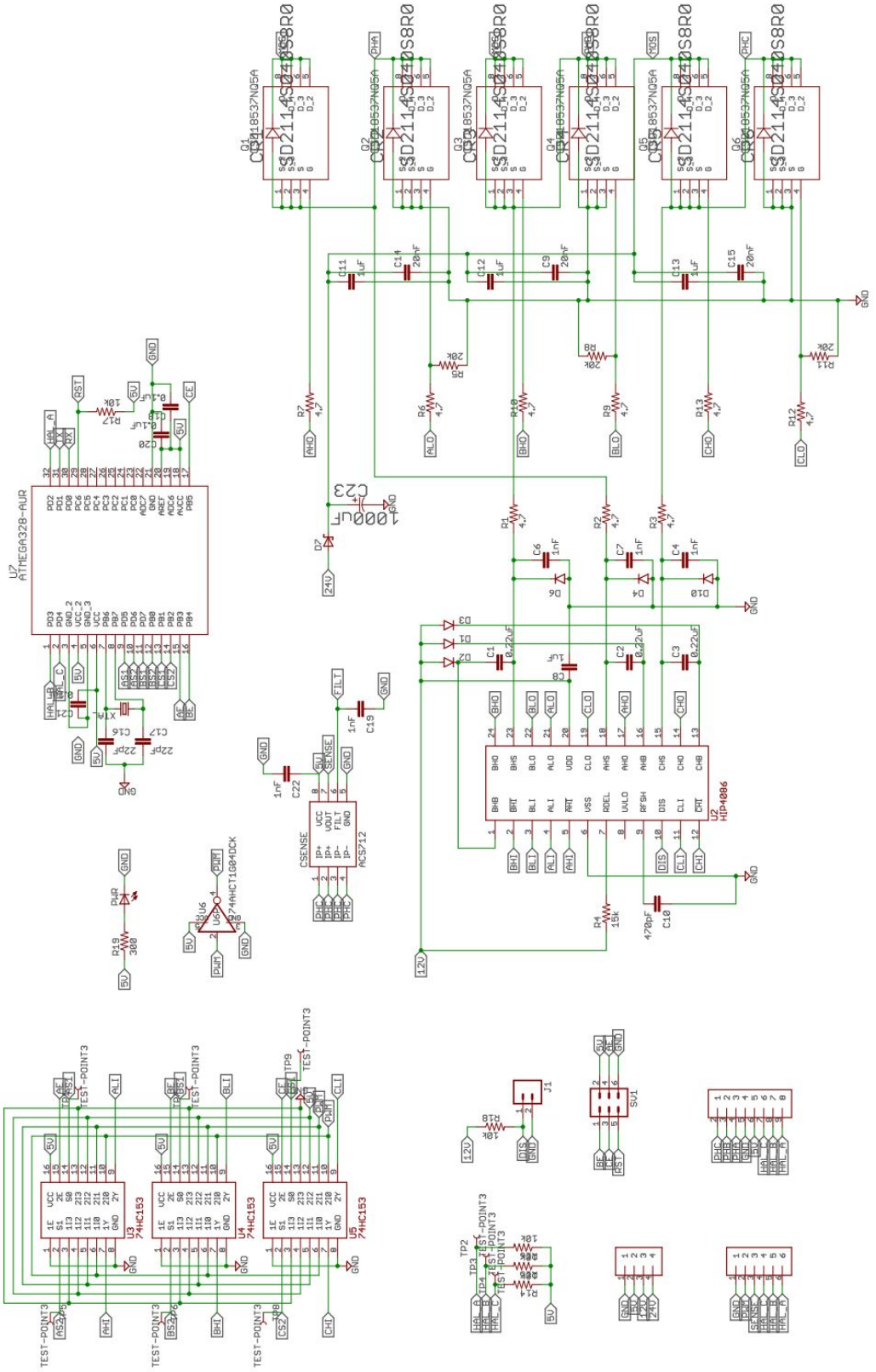
```
        self.ITerm += error * delta_time

        if (self.ITerm < -self.windup_guard):
            self.ITerm = -self.windup_guard
        elif (self.ITerm > self.windup_guard):
            self.ITerm = self.windup_guard

        self.DTerm = 0.0
        if delta_time > 0:
            self.DTerm = delta_error / delta_time

        # Remember last time and last error for next calculation
        self.last_time = self.current_time
        self.last_error = error

        self.output = self.PTerm + (self.Ki * self.ITerm) + (self.Kd * self.DTerm)

def setKp(self, proportional_gain):
    """Determines how aggressively the PID reacts to the current error with setting Proportional Gain"""
    self.Kp = proportional_gain

def setKi(self, integral_gain):
    """Determines how aggressively the PID reacts to the current error with setting Integral Gain"""
    self.Ki = integral_gain

def setKd(self, derivative_gain):
    """Determines how aggressively the PID reacts to the current error with setting Derivative Gain"""
    self.Kd = derivative_gain

def setWindup(self, windup):
    """Integral windup, also known as integrator windup or reset windup,
    refers to the situation in a PID feedback controller where
    a large change in setpoint occurs (say a positive change)
    and the integral terms accumulates a significant error
    during the rise (windup), thus overshooting and continuing
    to increase as this accumulated error is unwound
    (offset by errors in the other direction).
    The specific problem is the excess overshooting.
    """
    self.windup_guard = windup

def setSampleTime(self, sample_time):
    """PID that should be updated at a regular interval.
    Based on a pre-determined sampe time, the PID decides if it should compute or return immediately.
    """
    self.sample_time = sample_time
```

D. Atmega Motor Driver Code (C/Arduino)

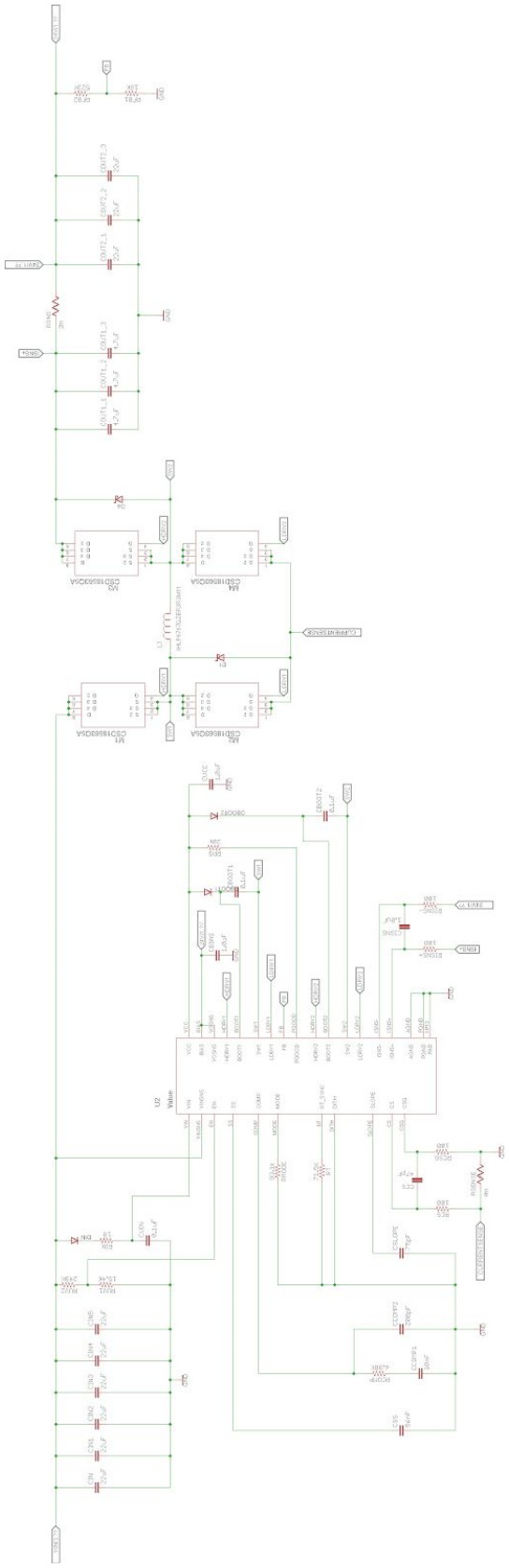
```
//SPEED
float RPM = 0.0;
float stepCount = 0.0;
int prevTime = 0;

// MUX BIT OUTPUTS
const int muxA0 = 5; //PD5
const int muxA1 = 6; //PD6
const int muxAe = 11; //PB3
const int muxB0 = 7; //PD7

const int muxB1 = 8; //PB0
const int muxBe = 12; //PB4
const int muxC0 = 9; //PB1
const int muxC1 = 10; //PB2
const int muxCe = 13; //PB5

// HALL SENSOR INPUTS
const int HallA = 2; //PD2
const int HallB = 3; //PD3
const int HallC = 4; //PD4

// HALL EFFECT READING VARS
char my_varA = 0;
char my_varB = 0;
char my_varC = 0;
int A;
int B;
int C;

// MOTOR STEP LOCATION
long curStep = 0;
long prevStep = 0;
long prevStep2 = 0;

void setup() {
  //Initialize Hall effect input pins
  pinMode(HallA, INPUT);
  pinMode(HallB, INPUT);
  pinMode(HallC, INPUT);

  //Initialize outputs/inputs
  DDRD = 0b11100000;
  DDRB = 0b11111111;

  //Detect the initial rotor step location
  A = digitalRead(HallA);
  B = digitalRead(HallB);
  C = digitalRead(HallC);
  if (A == 1 && B == 0 && C == 0) {
    curStep = 1;
  } else if (A == 1 && B == 1 && C == 0) {
    curStep = 2;
  } else if (A == 0 && B == 1 && C == 0) {
    curStep = 3;
  } else if (A == 0 && B == 1 && C == 1) {
    curStep = 4;
  } else if (A == 0 && B == 0 && C == 1) {
    curStep = 5;
  } else {
    curStep = 6;
  }

  //Initialize all MOSFETs LOW!
  turnOff();
}

void loop() {
  runMotor();
}

void runMotor() {
  // READ & GET THE HALL SENSOR STATES
  A = (PIND & (1 << PD2));
  B = (PIND & (1 << PD3));
  C = (PIND & (1 << PD4));

  prevStep = curStep;
  if (A == 4 && B == 0 && C == 0) {
    PORTD = 0b00000000;
    PORTB = 0b00010011;
    curStep = 1;
  } else if (A == 4 && B == 8 && C == 0) {
```

```
    PORTD = 0b01000000;
    PORTB = 0b00001010;
    curStep = 2;
  } else if (A == 0 && B == 8 && C == 0) {
    PORTD = 0b00100000;
    PORTB = 0b00100100;
    curStep = 3;
  } else if (A == 0 && B == 8 && C == 16) {
    PORTD = 0b00100000;
    PORTB = 0b00010001;
    curStep = 4;
  } else if (A == 0 && B == 0 && C == 16) {
    PORTD = 0b11000000;
    PORTB = 0b00001000;
    curStep = 5;
  } else if (A == 4 && B == 0 && C == 16) {
    PORTD = 0b10000000;
    PORTB = 0b00100100;
    curStep = 6;
  }
}

void turnOff() {
  PORTD = 0b01001110;
  PORTB = 0b00111101;
}

void getRpm() {
  // CALCULATE RPM
  if (curStep != prevStep) {
    stepCount++;
  }
  prevStep = curStep;
  if (millis() - prevTime >= 250) {
    RPM = (stepCount / 48.0) * 240.0;
    stepCount = 0.0;
    prevTime = millis();
  }
}
```

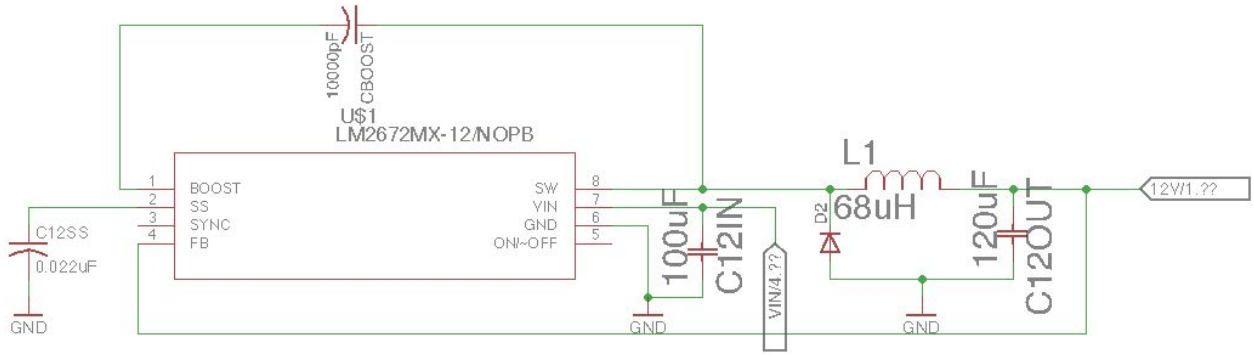
E. DF45 Motor Driver Schematic



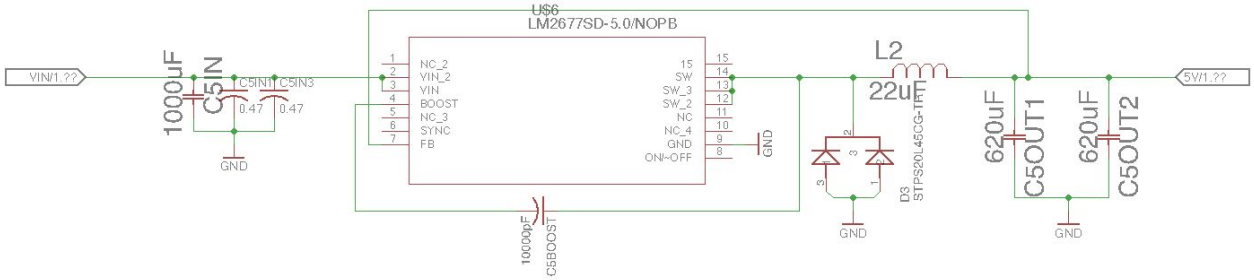
F. Power Distribution Schematic



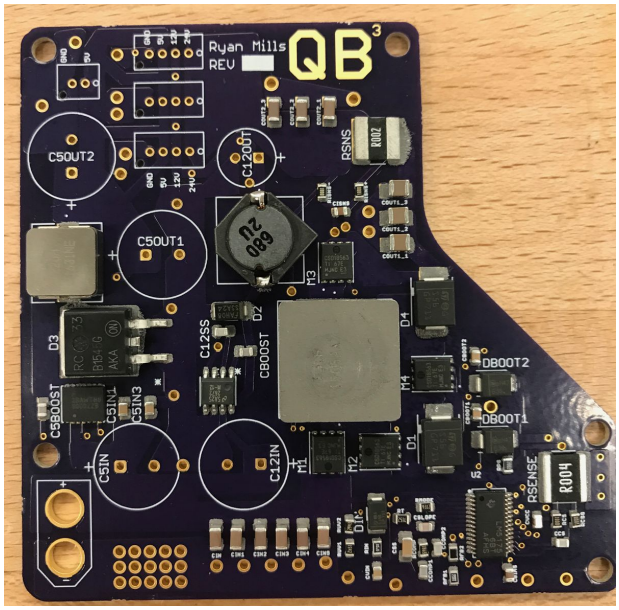
24V buck-boost converter



12V buck converter

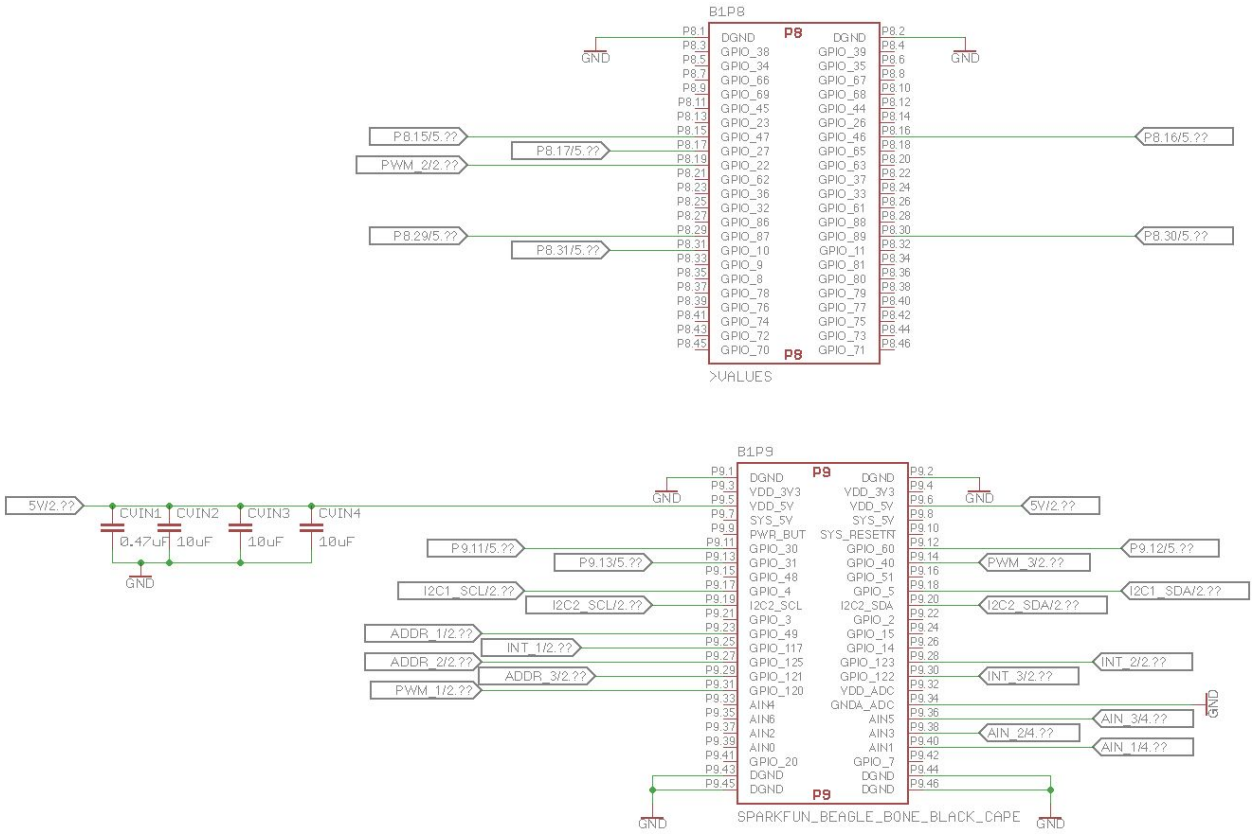


5V buck converter

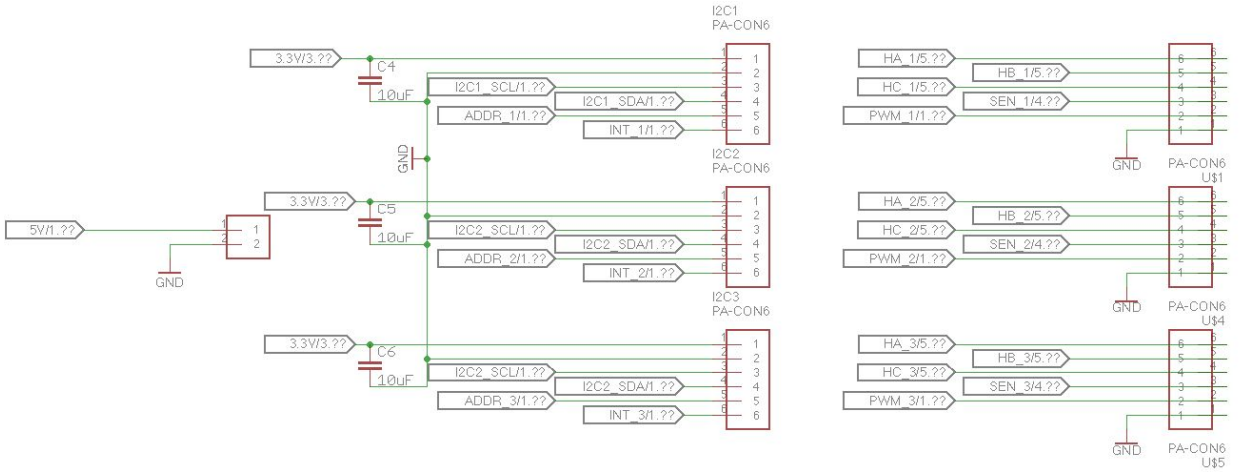


Power Distribution PCB

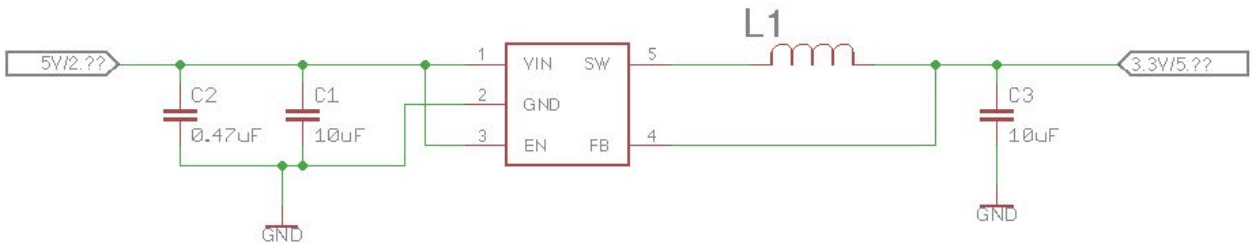
G. Beaglebone Cape



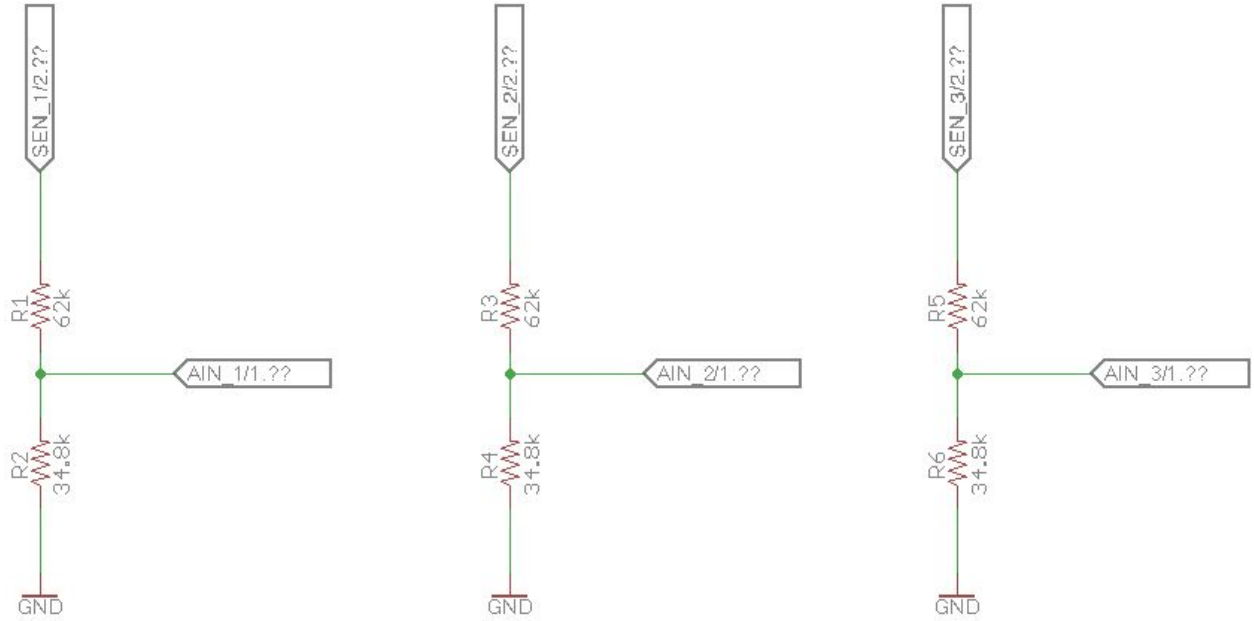
Connections to Beaglebone headers.



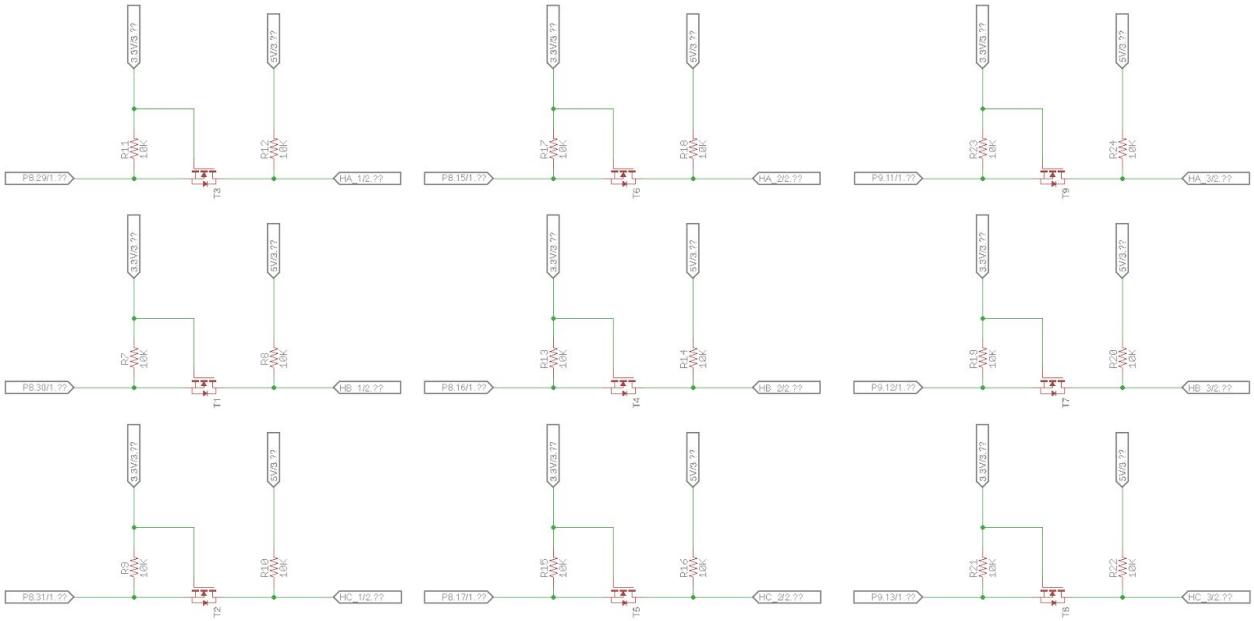
Connectors on cape to attach other components in system to BeagleBone.



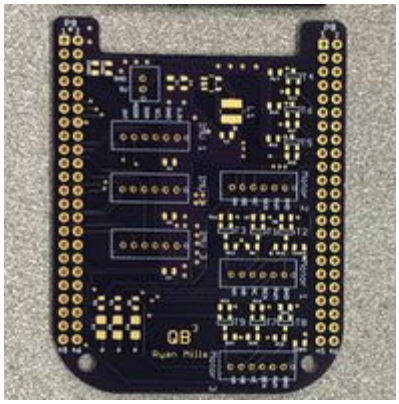
3.3V converter to power IMU and 5V to 3.3V logic.



5V to 1.8V voltage divider for Analog to Digital Converter with max 1.8V.



5V to 3.3V digital logic converter.



BeagleBone Cape PCB