The pancreas unmanned ground vehicle

by

Benjamin Weinhold

B.S., Kansas State University, 2021

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Carl and Melinda Helwig Department of Biological and Agricultural Engineering
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2023

Approved by:

Major Professor
Dr. Daniel Flippo PhD

Copyright

Abstract

Global agricultural output must increase by 25 to 70% by 2050 to feed the world. The development of more resilient, higher yield crops using genotype to phenotype prediction is one promising method to achieve this growth. Progress in genotype to phenotype models has been constrained by the quantity of hand measurements necessary to accurately describe phenotype characteristics. The Pancreas unmanned ground vehicle was developed to fill this gap in capability.

The Pancreas is a four-wheeled unmanned vehicle which carries an electromagnetic inductance sensor to gather soil moisture data throughout the day. This sensor offers a reduction in the time required and an increase in the quantity of measurements taken over the typical soil core methods of measurement. The initial Pancreas prototypes were developed by Dr. Daniel Flippo and master's student Calvin Dahms. The author made alterations to these designs to reflect a change in operational requirements after the testing results of these prototypes. Broadly, the platform was made more robust, a path following algorithm and new control system were implemented, and a new power budget was developed.

Though these changes represent necessary improvements, the platform needs more work and testing to effectively perform its role. Increased power use, sensor accuracy, obstacle detection and avoidance, and durability remain problems to address in future work.

Table of Contents

List of Figures

1. Feeding the World

1.1. Projected Food Needs

The UN expects world's population to reach 9.8 billion by 2050 and the world does not currently produce enough food to support that many people [1]. Future human flourishing requires finding new ways to feed the world.

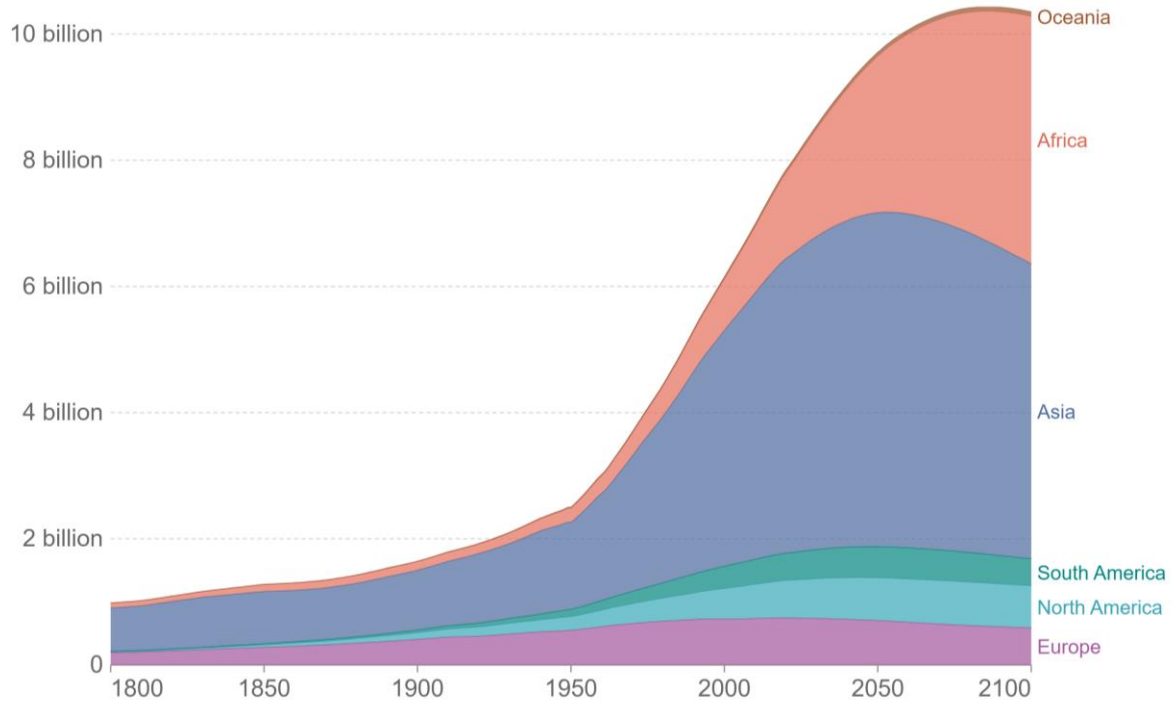


Figure 1: Global Population Growth with UN Predictions for 2100 [2]

Academics disagree on the precise amount by which global agricultural output must increase to feed the world. Well-regarded predictions from the early 2010's called for doubling the food supply by 2050 [3] [4]. The current number, taking into account the gains made in production in the intervening decade, could range from a 25% to a 70% increase in current output [5]. This substantial range is not particularly helpful for accurately assessing the danger to

civilization. However, with increasing globalization, even a small disruption in the supply chain can cause outsized impacts in fragile food supply systems, and the 25% deficit in food availability predicted in the most conservative projection would be catastrophic to significant swathes of the developing world with a high import dependency ratio. Import dependency ratio is a function of the imported food supply compared to the total available food supply.

Increasing the food supply and its resistance to externalities is not only necessary to sustain civilization, but also comes with benefits which range from a decrease in undernutrition related health issues to an increase in economic prosperity. In particular, an increase in agricultural GDP in developing nations is directly linked to a rapid reduction in undernutrition [6]. As countries modernize and a greater share of the population migrates to urban areas, countries that favor agricultural investment experience a greater reduction in malnutrition related illnesses like stunting, as shown in figure 2 below.

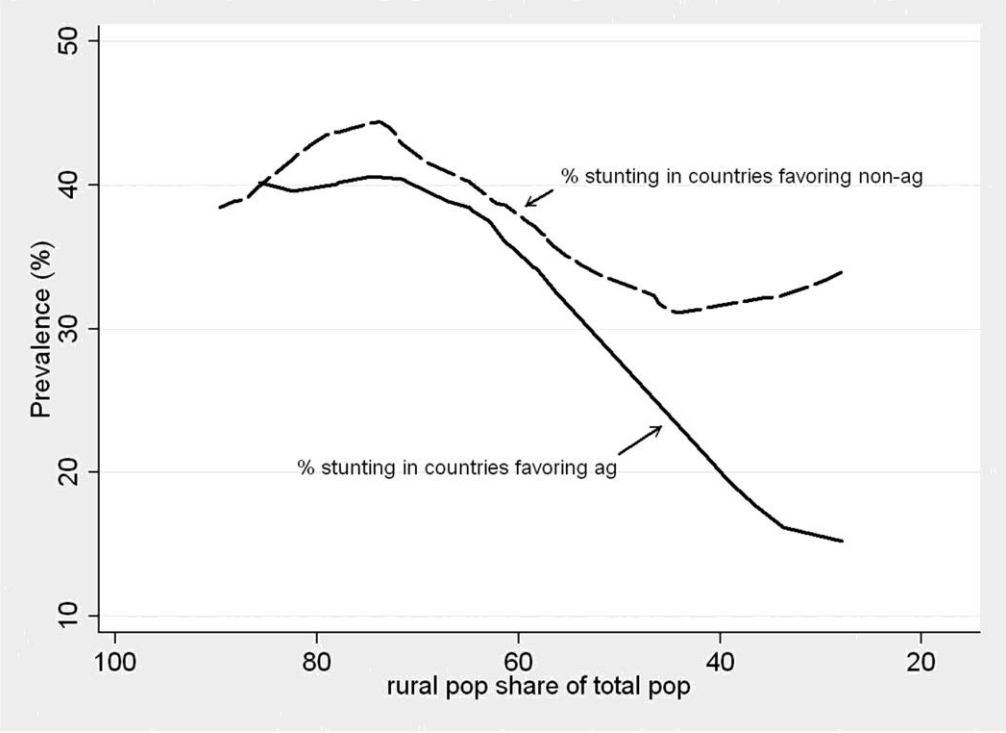


Figure 2: Prevalence of Stunting as a Function of Rural Population Share [6]

More productive farmers free up larger portions of a developing economy to invest in other industries, reduce domestic food prices, and thereby raise the standard of living. Increasing the productivity of farmers requires better methods of utilizing fertilizer, efficient agricultural machinery, and more resilient and productive crop varieties. The last category is the focus of this paper.

1.2. Current Cereal Production and Consumption, and Its Impacts

The five most consumed cereal grains globally are rice, wheat, corn/maize, barley, and sorghum [7]. Additionally, cereals make up a vital part of livestock feed. They account for almost 99% of all cereals produced globally [8]. Taken together, these five make up 46% of the calories consumed globally on a year-to-year basis as shown in figure 3 [9]. Of that, wheat made up 26% of all cereals produced in 2021, and so over 10% of total global calories.

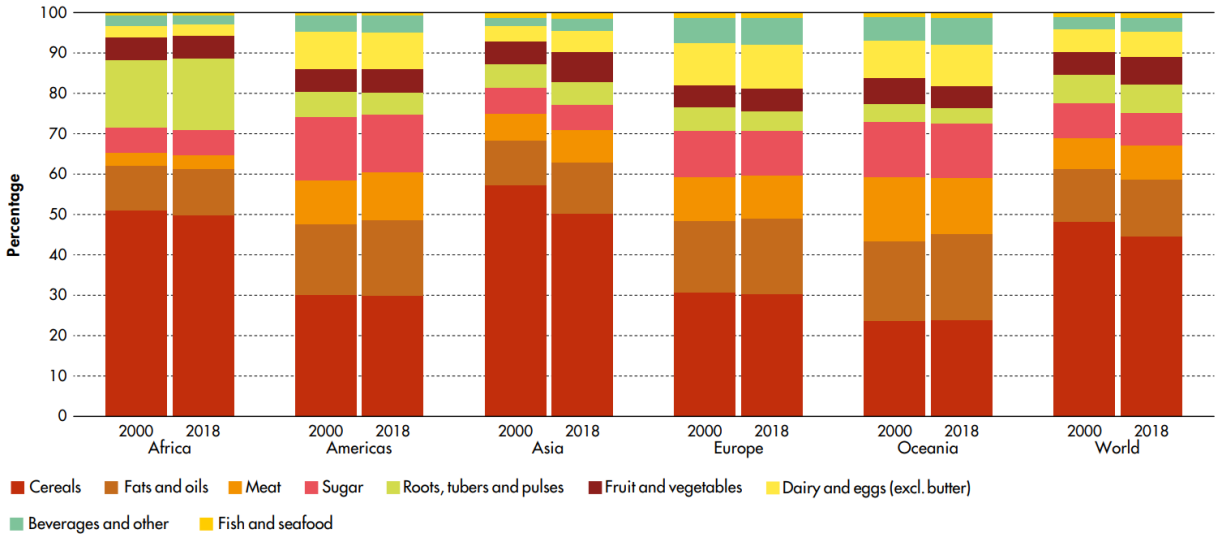


Figure 3: Dietary Energy Supply by Type [9]

In recent history the widespread adoption of fertilizers, mechanized farming, and better strains of cereal crops have greatly improved agricultural production. Over the past twenty years

wheat output has increased by just over 30% from roughly 600 million metric tons in 2000 to just under 800 million metric tons in 2021 [7].

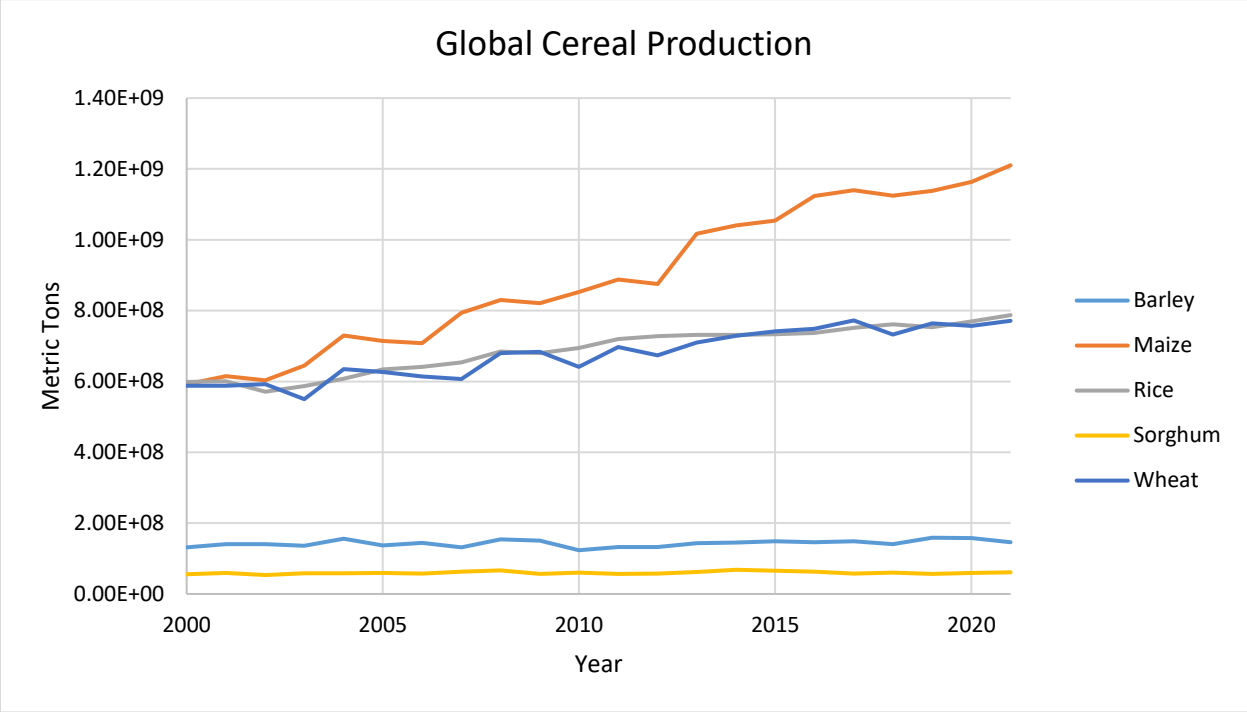


Figure 4: Global Cereal Production [10]

Although such growth could justify relaxation of research and development goals, these gains should serve as inspiration for further advancement, because the Food and Agriculture Organization of the United Nations (FAO) still categorizes 2.3 billion people as food insecure [9]. Mild food insecurity can be reasonable uncertainty about future food availability, and on the severe end meals are regularly missed.

Furthermore, recent conflict-related disruptions in the grain import market have highlighted the importance of wheat cultivation for global stability, because many regions such as north Africa and East Asia (excluding China) have a high import dependency ratio, being 52.4% and 69.0% as of 2018, respectively [11]. Disproportionate reliance on imports causes disproportionate impacts from shortages, as in the 2010-11 food price crisis. This crisis was

caused by a drought in Russia, Ukraine and central Asia, which make up about a third of global wheat exports. This was an exacerbating factor in the Arab Spring uprisings and civil unrest of the same year [12]. Similarly, the Ukraine war impacting those same wheat exporting regions has led to uncertainty, and the full effects have yet to be seen at the time of writing. The current state of food security when viewed holistically demonstrates a need for continued development toward increasing wheat productivity. Not only does the world need more productive wheat to supply a growing population, but also to make the food supply more resilient to external threats like natural disasters and conflict by supplying reserve capacity. To that end, research must produce new strains of wheat that use available land and fertilizer more efficiently to produce greater yield and that are less vulnerable to drought or other adverse conditions.

1.3. Wheat Phenotyping

In order to meet the pressing need for greater global wheat production capacity, Kansas State University, Oklahoma State University and Langston University, applied for and received EPSCoR grant 1826820 from the National Science Foundation (NSF) to research and develop modern crop models for genome to phenome prediction [13]. Wheat genome to phenome prediction is the process of mapping specific genetic traits to their expression in features like height, leaf size, and, most importantly, yield. Phenotyping, unlike genotyping, has been a slow and very labor-intensive process because researchers must measure the physical attributes of a plant in the field by hand. For this reason, it has become the constraining factor in many breeding operations [14]. Phenotyping provides critical data for predictive crop models, so work is necessary to develop more rapid data collection and analysis methods by leveraging new sensing technology and platforms such as computer vision and affordable UAVs to increase the quality

and quantity of data. A variety of such platforms are shown in figure 5. High-throughput research would allow farmers to select strains of wheat tailored to their growing conditions.

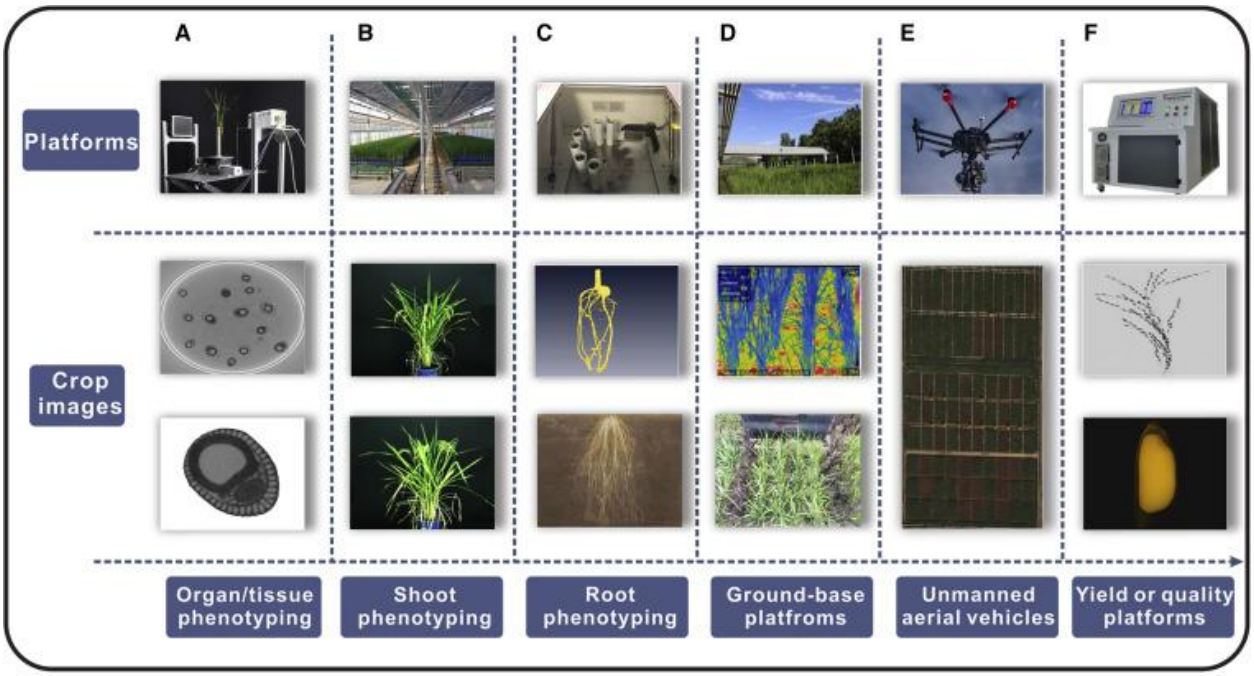


Figure 5: Modern Sensing Platforms and Roles [15]

The research of the EPSCoR grant takes a four-pronged approach: theory and computation, modeling, field testbed analysis, and field sensing. Each focus area has a dedicated team and sub goals. The sub-components of field sensing are areal imaging, gene-expression, and soil electrical conductivity sensing.

1.4. Autonomous Sensing and Data Collection

Electrical conductivity soil sensing has several benefits over the more common soil core analysis techniques. Soil core analysis is time consuming and does not capture certain time-dependent information about soil moisture in the field. Furthermore, each measurement only provides information about one point in the field, making it difficult to capture a complete

picture of field attributes [13]. Electrical conductivity (EC) does not require researchers to remove soil samples from the field. Instead, the sensor can be swept over the entire length of the phenotyping testbeds to gather data about soil properties and the root systems of the test wheat. Moreover, because the sensor reduces analysis time and requires fewer trips back to the lab, researchers can take more sensor passes at several different points in the day to provide insight on time-dependent characteristics of the field.

The grant proposal calls for an autonomous ground vehicle in order to take a large volume of repeatable measurements over the course of several days in the field. The vehicle would reduce man-hours spent in data collection and would potentially gather a larger quantity of consistent data. The EPSCoR proposal calls for several autonomous vehicles of this kind built of largely non-metallic components which could remain in the field for several days at a time. Application of these criteria would grow into the autonomous vehicle known as the Pancreas.

2. The Pancreas Platform Background

2.1. Design Criteria and Initial Concept

In order to fulfill the need for an autonomous sensing platform for wheat phenotyping, Dr. Daniel Flippo, Ph.D. initially designed the Pancreas robot to be a lightweight, solar powered, unmanned ground vehicle (UGV).

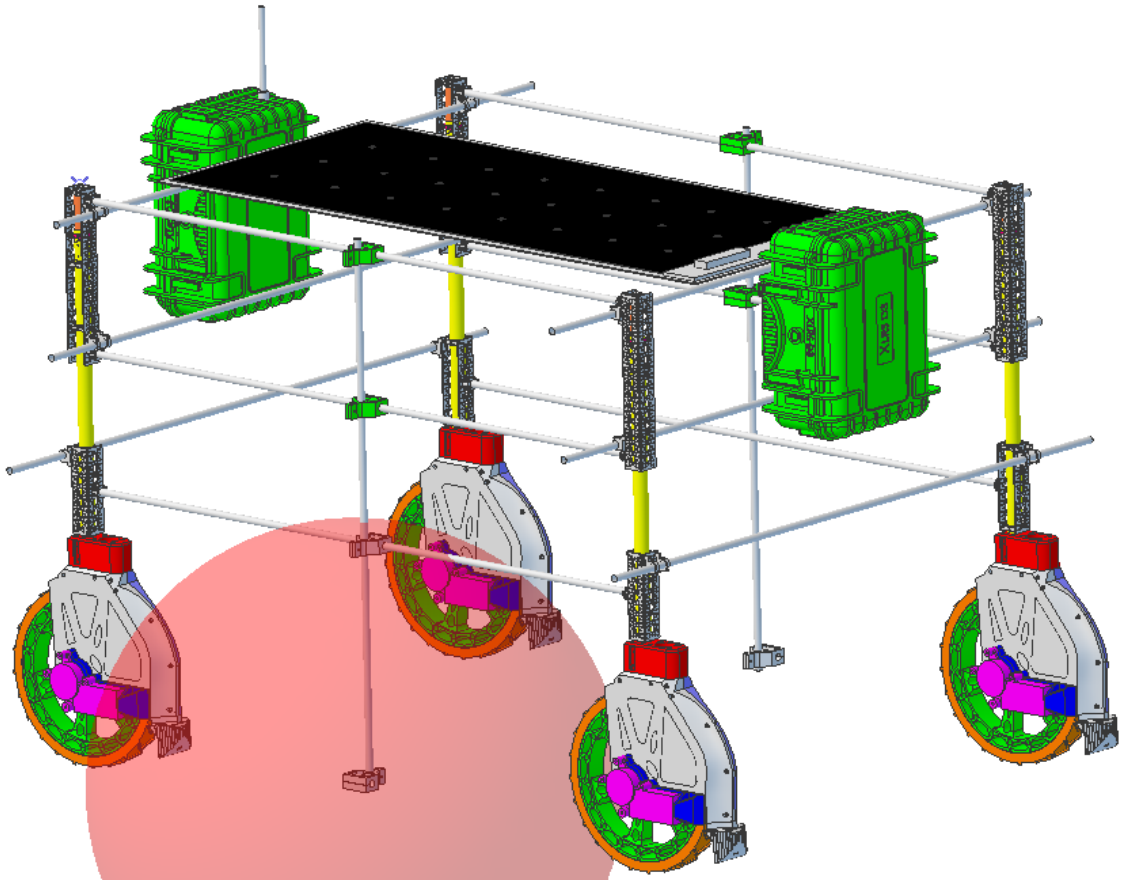


Figure 6: Initial Pancreas Concept

The four-wheeled vehicle design could straddle a row of the phenotyping test wheat and carry a Geophex Electromagnetic Induction sensor to autonomously take soil moisture measurements [Appendix B]. The sensor hanging in the middle of the robot would take measurements while the wheels would travel in the paths between wheat plots. The platform was

designed to remain in the field continuously for several days at time, except for maintenance and data collection. A set of solar panels charging an onboard battery pack were planned to enable this long-term field endurance.

Figure 7: Geophex GEM-2 (left), Solar Panels on First Prototype (right)

Because it was absolutely essential that the wheat not be crushed by the wheels of the robot, as that would negatively affect data collection for the larger EPSCoR phenotyping project, the platform required reliable path finding and obstacle avoidance. Master's student Calvin Dahms implemented the initial prototype and iterated on it over the course of 2020 and 2021. He built a second prototype, whose design features are described below.

2.2. Prototype Two Frame

After finding 1.27 cm rods too flexible, Dahms built the second vehicle out of 2.54 cm diameter carbon fiber tubing and 3D printed plastic components. These materials minimize interference with the underslung EM sensor which is sensitive to electromagnetic interference

from metallic and electrical components. However, some metallic components, like the drive and steering motors, the bearings and the bearing housings are integral to the platform's functionality, and could not be made from non-metallic materials. To minimize interference, these metallic components needed to be placed as far as possible from the sensor. Those consulting on the project initially considered it sufficient to place all electronics, metal, and motors at least 1m away from the EM sensor. The dimensions of the individual wheat plots also constrained the size and shape of the UGV's frame. The robot needed to straddle a single wheat plot which was 135 cm across and approximately 100 cm tall when fully grown, depending on the individual strain of wheat. The platform's motors and leg assembly also needed to fit in the 32 cm gap between plots.

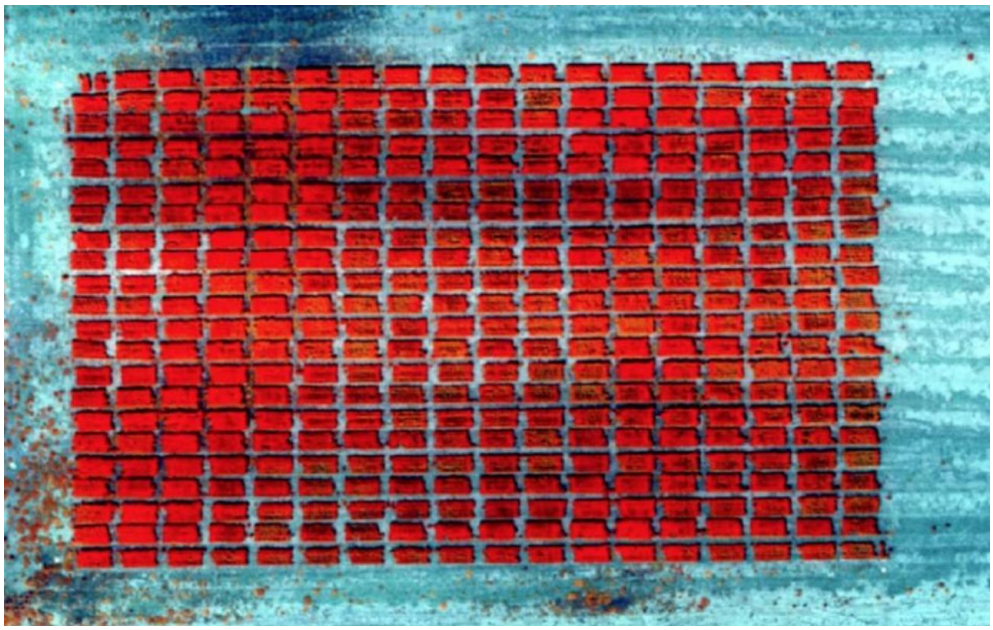


Figure 8: Overhead, Infrared View of Test Plot [16]

Based on these constraints, the frame was made of 12, 2.54 cm diameter carbon fiber tubes of 152 cm length with 3D printed ABS plastic gussets and fenders. The frame stands at 132 cm with sufficient clearance for a wheat row underneath.

Figure 9: Second Prototype in Field

The vehicle has four, independently-steered wheels, each initially powered by high torque servo motors which turn the whole carbon fiber leg on which the wheel is attached. All these features make for a lightweight maneuverable platform, able to carry the EM sensor for long periods in the field.

2.3. Prototype Two Power

To power the platform, two HQST, 100 W, 18 V solar panels, wired in series, charge two 11.1 V, 10.5 Ah, lithium polymer batteries through a solar charging regulator [Appendix B]. These batteries are also wired in series to make a 22.2 V battery pack. The solar panel's 100 W rating is based on a solar panel perpendicular to the incident sunlight during clear weather conditions. Because outside conditions vary, the amount of power generated by the panels is highly dependent on time of day and weather. As a result, the battery capacity must be large enough to buffer the peaks and troughs in power generation during normal operation. During the

initial prototype phase of development, the Pancreas was expected to draw about 100 W during use.

A 12 V regulator steps the power down from 22.2 V to 12 V for the control electronics, computer, and GPS. The onboard computer reduces voltage internally to 5 V and supplies power to some of the sensors.

2.4. Prototype Two Sensors

There is a small margin for error in navigating through the test wheat field. With that in mind the robot uses a suite of sensors to accurately follow a preset path and avoid the project wheat. With a gap between wheat plots of 32 cm, sub meter geo-location precision is needed for accurate path following along with object detection and avoidance protocols. The onboard GPS, Topcon B-125 unit, enables this precision. The B-125 is capable of sub centimeter accuracy with a position update rate of up to 100Hz [Appendix B], This rate is more than sufficient for a slow-moving agricultural robot. A single board computer called a Latte Panda processes the incoming GPS data over its USB port. In this stage of the design process, the path following algorithm had not been implemented, so the GPS data has been saved for later use.

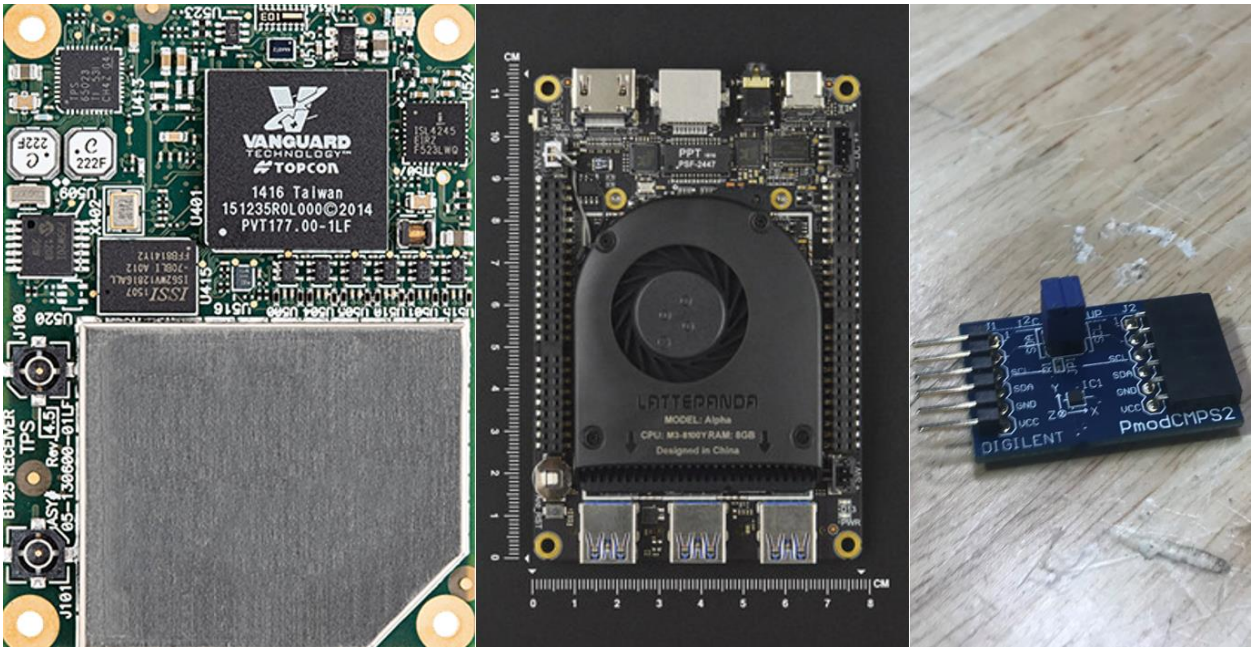


Figure 10: B-125 (left), Latte Panda Computer (center), PMod Compass (right)

The onboard compass, which supplies the vehicle's heading to orient the robot in its environment for path following, is a Pmod CMPS2. This is a 3-axis digital compass that communicates using the I2C protocol [Appendix B]. The compass, the voltage sensor and the current sensors communicate with the Latte Panda's built-in Arduino Leonardo which then feeds that data to the main computer to be recorded for future analysis.

Several sensors supply different channels of feedback about the robot's immediate environment to provide the object detecting capabilities needs of the robot. A pair of Adafruit VL53L0X Time of Flight Distance Sensors (TOF) are mounted on a front leg of the second prototype and determine how close and at what angle the wheat rows were relative to the robot. This sensor input could be used in the future to supply angle corrections to prevent collision with the test wheat. Angle calculations are made by taking measurements from each sensor and checking the difference between them. Another method of obstacle detection was proposed but not implemented on the second prototype. The method checks the color of the objects in front of

the wheel with the Sparkfun AS7265x Spectral Triad spectroscopy sensor to determine whether or not to avoid them. The author designed the mounts for the TOF and Spectroscopy sensors, shown below, and the code for the Spectroscopy sensor.

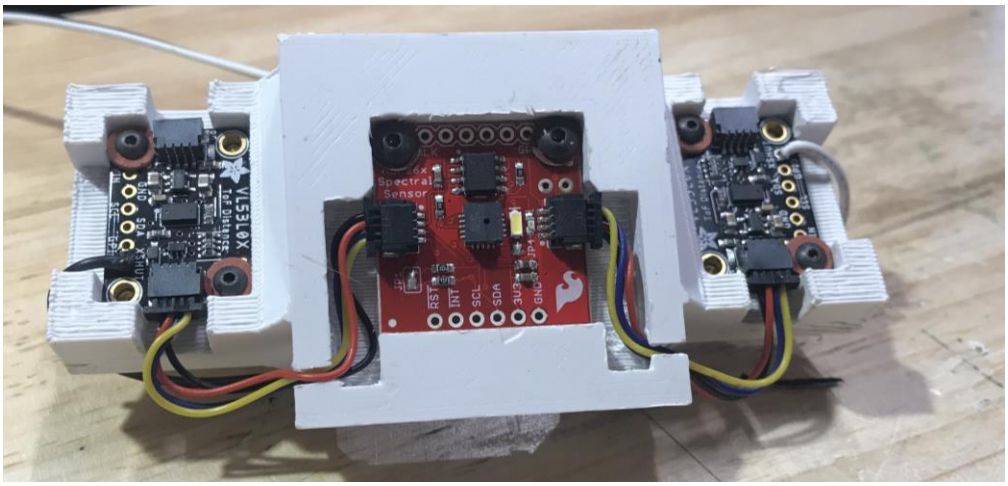


Figure 11: Distance (left and right) and Spectroscopy (center) Sensors

Color spectroscopy allows the platform to determine whether or not a plant is a weed or wheat to avoid because under IR and UV wavelengths different plants will have distinct reflectance. Finally, the most sophisticated method of obstacle detection and avoidance uses machine learning and computer vision to identify and avoid wheat from video taken on an OpenCV AI camera.

2.5. Prototype Two Controls and Propulsion

A high torque Lynx Motion Smart Servo steers each of the four wheels. Automobile window motors power the four drive wheels and are controlled by a pair of Sabertooth 2x32 motor controllers. The window motors are self-locking, worm gear motors with a high continuous torque rating of 3 Nm for their size. Four Accu-coder encoders from Encoder Outlet provided feedback from the window motors. A NI LabVIEW MyRio manages all the high level PID and

steering control which received user input from an iPad running a LabVIEW data dashboard to steer manually and enable test data collection for the initial prototype phase of the platform.

Figure 12: MyRio (left), Servo Motor (center), Motor Controller (right)

The platform turns using a method called Ackermann steering which is typically implemented by a mechanical linkage between two wheels. The Pancreas does this with digital controls. Ackermann steering reduces wheel slip by giving each wheel's turning arc the same center which reduces wear on the wheels and strain on the frame.

2.6. Prototype Two Feedback

Calvin Dahms' initial prototype design choices and alterations allowed for essential data collection and important insights for future work. The author's work on the platform began largely after the implementation of the second prototype. Data analysis determined that a 1 m bubble was insufficient to remove interference from electrical and metallic equipment on the UGV during sensing operations. As a result, the frame was expanded and the sensor was dropped closer to the ground. Because of the wheat row constraints, the frame had to be doubled in width to straddle two rows of wheat instead of one, rather than increased only enough to accommodate

the new metal-free bubble. In addition, National Instruments no longer supported the LabVIEW iPad control software, and the software was incompatible with future versions of LabVIEW. The author removed the MyRio and LabVIEW entirely in favor of a Python-based, radio control approach. Part of the motivation for this was the author's more significant experience with, and preference for Python.

In addition, the computer vision obstacle detection demonstrated issues that needed to be resolved before final implementation. Training a computer vision system to accurately specific identify objects in a scene requires large volumes of labelled data. The proof-of-concept machine learning algorithm used labelled images only of wheat heads and so, was of limited use for path following. Path following would require labelled images of paths between wheat rows, or the wheat blocks themselves. These do not exist because they are very domain specific. The only way to get labelled images of this kind would be to manually label thousands of images personally taken at the angles from which the robot would view the field. While this work may be an area for future research, the author put it aside in favor of implementing reliable path following.

The criteria updates for size necessitated substantial alterations to the frame and control system to support the increased load, motor power, and electrical requirements. Additionally, the change in computer control required many changes had to be made to the wiring and electronic layout of the platform.

3. Hardware Changes for Prototype Three

3.1. Frame Adjustment and Reinforcement

To expand the frame to the 3 m required to span two rows of wheat, the author replaced four of the 2.54 cm diameter, 1.5 m long carbon fiber tubes which made up opposite sides of the robot with 3 m long Fiber Reinforced Plastic (FRP) fiberglass tubes of the same diameter.

Figure 13: FRP (right) and Carbon Fiber (left)

The previous carbon fiber tubes had a wall thickness of 2.16 mm, and the replacement fiberglass tubes have a wall thickness of 6.35 mm. Fiberglass was selected instead of carbon fiber for its availability. At the time of construction, carbon fiber in the length required would have taken several months to arrive because of lingering supply chain issues and was cost prohibitive.

Figure 14: Pancreas with Fiberglass Extended Frame

Fiberglass on the other hand offered a reasonable trade off in stiffness and an increase in weight for being more available. The important characteristic for stiffness is flexural rigidity. The equation for flexural rigidity is shown in (1).

$$Flexural\ Rigidity = EI \tag{1}$$

Where E is Young's modulus and I is the area moment of inertia. The area moment of inertia for a tube is calculated in (2).

$$I_{tube} = \frac{\pi(r_o^4 - r_i^4)}{4} \tag{2}$$

Where $\boldsymbol{r_o}$ is the outer radius and $\boldsymbol{r_i}$ the inner radius. For anisotropic composites like the FRP fiberglass and carbon fiber used in the Pancreas, flexural or bending modulus is used instead of Young's modulus. This is because there are different material properties depending on fiber

orientation in the composite. These material constants are rough estimates because properties vary by manufacturer, and the study of the material properties of composites is well outside the scope of this paper. The industry standard flexural modulus ranges from roughly 5.5GPa to 9.5GPa for FRP tube. Carbon fiber of the type used on the platform has a flexural modulus that ranges from about 75GPa to 125GPa Filling in the equation for both materials using the lower estimate for each we have results for flexural resistance for carbon fiber and fiberglass in (3) and (4), respectively.

$$FR_{Carbon\ Fiber} = (75 * 10^9 Pa) * \pi * \frac{(\frac{0.0254m}{2})^4 - \left(\frac{0.0232m}{2}\right)^4}{4} = 465.8Nm^2 \quad \textbf{(3)}$$

$$FR_{FRP\ Fiberglass} = (5.5 * 10^9 Pa) * \pi * \frac{(\frac{0.0254m}{2})^4 - \left(\frac{0.0191m}{2}\right)^4}{4} = 76.4Nm^2 \quad \textbf{(4)}$$

As shown above, the carbon fiber is about six times more rigid than the FRP tube. In addition, the fiberglass weighs 0.358 kg/m of tube, while the carbon fiber weighs 0.126 kg/m of tube, or approximately three times lighter. In total this change from carbon fiber to FRP adds about 3kg in added weight, which is fairly reasonable considering the weight of all other reinforcing components added later.

The substantial increase in robot width causes an increase in frame flexibility and a larger moment arm which increases the torques from forces experienced by the robot. Excessive bending occurred during turns, resulting in the robot legs tilting inward and bowing the middle of the frame upward. This bending is caused by lateral force on the wheels coming from uneven terrain and small misalignments in the wheels and frame. This bending action forces the wheels to have substantial positive camber and reduces wheel contact with the ground. Furthermore,

reduced traction causes skidding. In addition, the bending places strain on the robot legs and wheel housings increasing the likelihood of component failure.

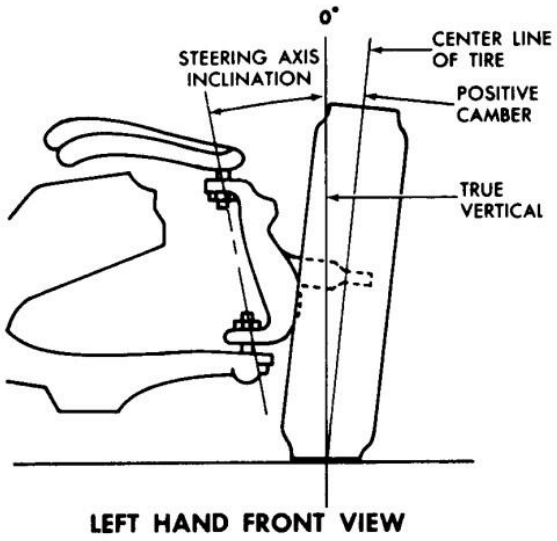


Figure 15: Positive Camber [17]

In order to mitigate this issue, the author modified the old gussets and leg supports in several ways. Moving the leg brace supports lower down the leg, toward the fender, left a much shorter unbraced section of leg. Next, the plastic section of the brace support was also redesigned to include bearings, which reduce friction and component wear.

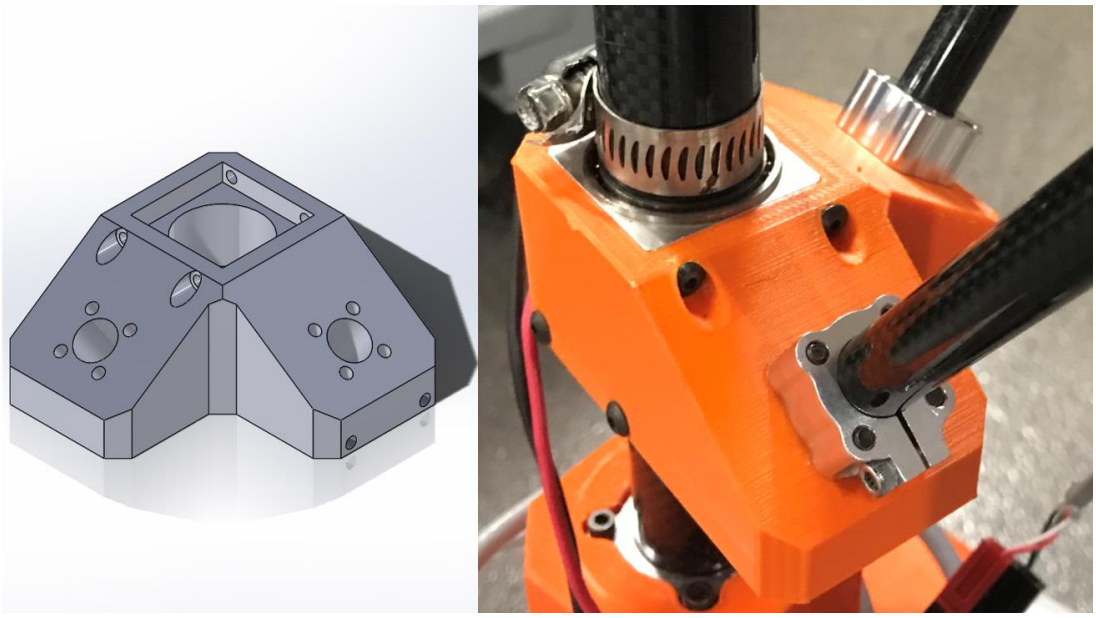


Figure 16: Brace Bearing Block

The carbon fiber sections of the supports also attach at a more obtuse angle, further increasing resistance to lateral forces.

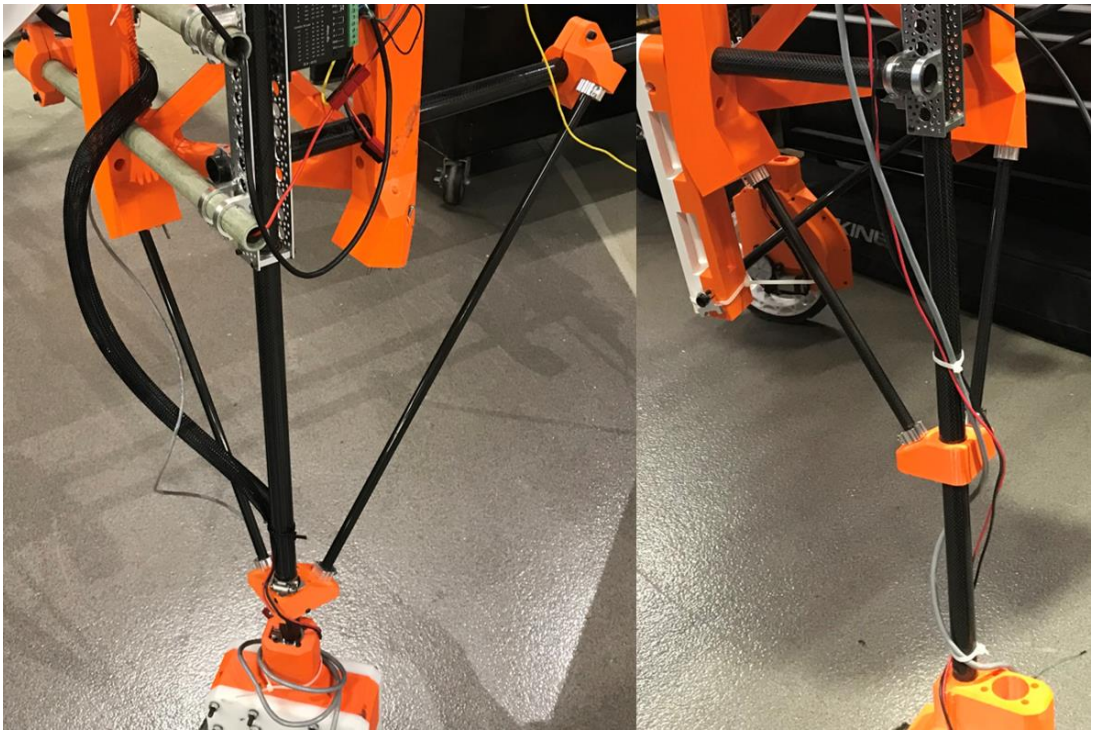


Figure 17: New Brace (left), Old Brace (right)

Because the brace attachment points to the upper frame are at an angle where they can no longer be fixed to the gusset, the author developed a new component for attaching the leg supports to the upper frame. The new components are made of two, 3D printed plastic parts which use brass insert nuts, heat formed into the plastic, and machine screws to clamp around the carbon fiber and fiberglass tubes of the upper frame.

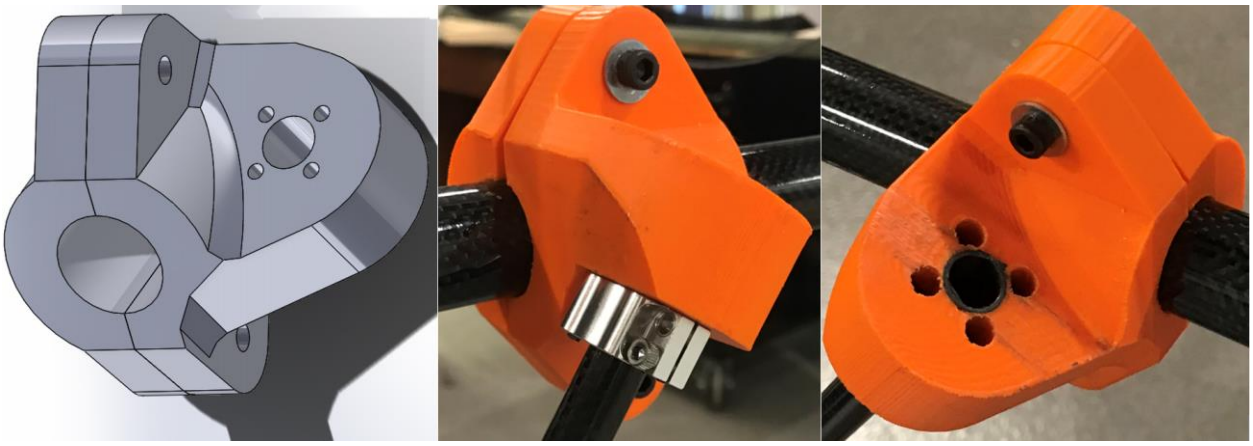


Figure 18: Brace Attachment Bracket

These adjustments stabilize the frame during operation and improve durability and mobility in the field by keeping more of the wheel in contact with the ground.

3.2. Reinforced Fenders

In addition to the leg supports being insufficient for the new, larger design, the wheel fenders also experienced excessive load, leading to failure. After driving a short distance, the sides of the fenders bent, and the wheel detached itself from the motor encoder and developed a severe cant to one side. To remedy this issue, the author designed a new fender.

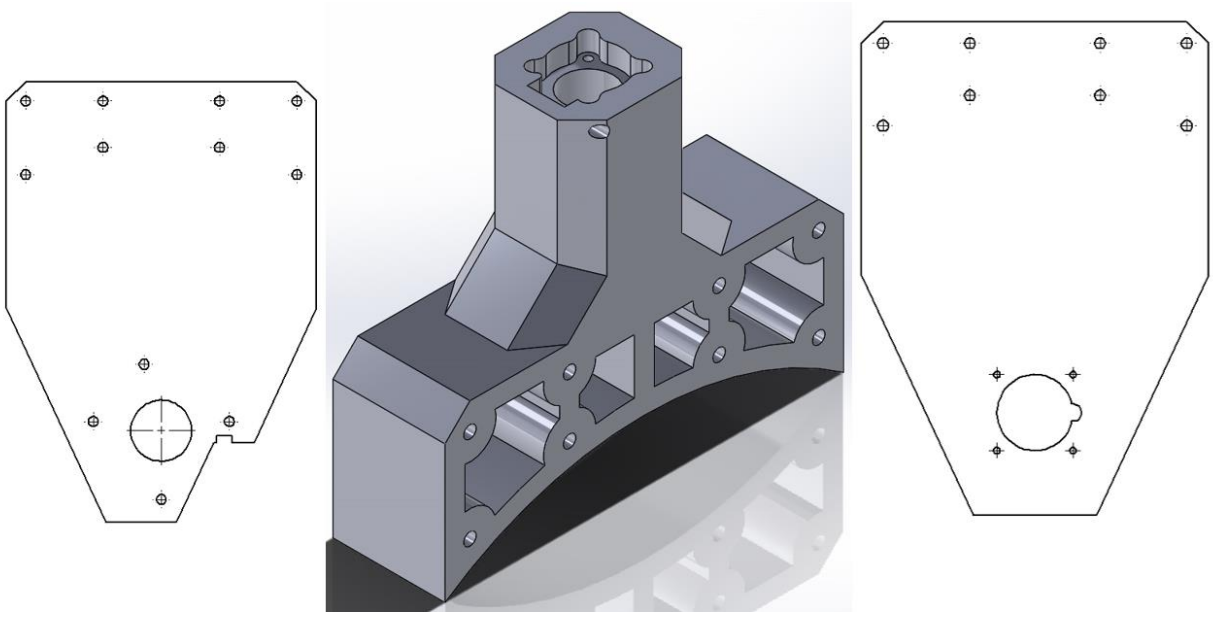


Figure 19: Left Fender Plate (left), Leg Socket (center), Right Fender Plate (right)

The new construction uses two, 1.9 cm thick sheets of HDPE plastic, which sandwich a 5.7 cm wide, 3D printed ABS plastic component with a socket that attaches to the robot leg. HDPE was chosen for its stiffness and ease of machining. The 3D printed component is attached to the HDPE sheets with 8 6mm bolts. The HDPE sheets, in turn, provide the attachment points for the drive motors and encoder. The middle 3D printed component's layer lines are perpendicular to the bolts securing the two sheets to it, which prevents the component from splitting when the bolts are tightened. Layer orientation needs consideration because 3D prints using FDM are anisotropic, meaning that they do not have the same strength in all directions and are vulnerable to layer separation under stress.

Figure 20: Wheel Assembly

The socket is slightly oversized to accommodate the robot leg, and a 25 mm Actorobotics clamping hub secures it [Appendix B].

Figure 21: Clamping Hub

The clamp is inset in the socket face and secured with machine screws and brass inserts. The new fenders are easy to disassemble and are substantially more robust than the previous iteration.

3.3. Stepper Motor Steering

In addition to frame and fender failure, the motors also struggled with the alterations. Several of the high-torque servo motors used for turning the robot legs burnt out under the increased load. Because the servos used were already on the upper end of what is commonly available in terms of torque rating for servo motors, the robot needed a new type of motor. The author selected four, Nema 23, 4.25:1 geared stepper motors for their high torque and robust construction as well as the angular precision that stepper motors have.

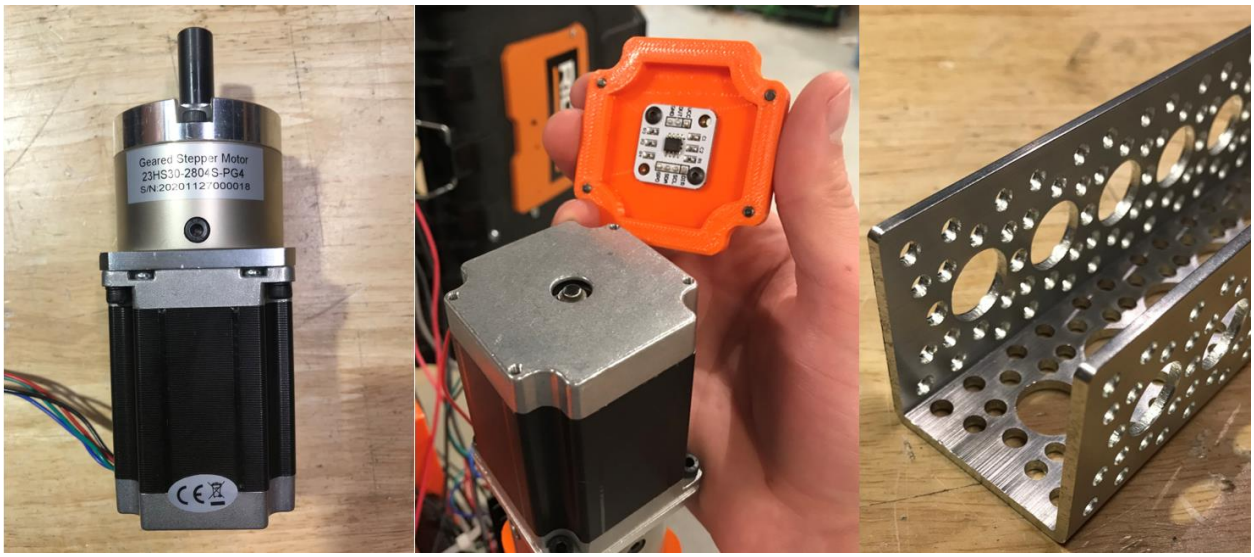


Figure 22: Stepper Motor (left), Encoder & Mount (center), U-Channel (right)

The motors have a holding torque of 8 Nm which is sufficient to turn the wheels under increased load on uneven terrain [Appendix B]. Closed loop control is necessary because the motors can still be forced out of alignment during operation and because accurate navigation requires precision control of the wheel angles. Using AS5600 magnetic rotary encoders mounted to the motor with a 3D printed component, one of the onboard micro-controllers can provide

corrections when a wheel is in an improper position. The motor is mounted in place over the robot leg using a 3D printed plastic part. The part secures the face of the motor to the top of the Actobotics U-channel at the corners of the Pancreas frame. The complete assembly is shown below alongside the CAD models in the figure below.

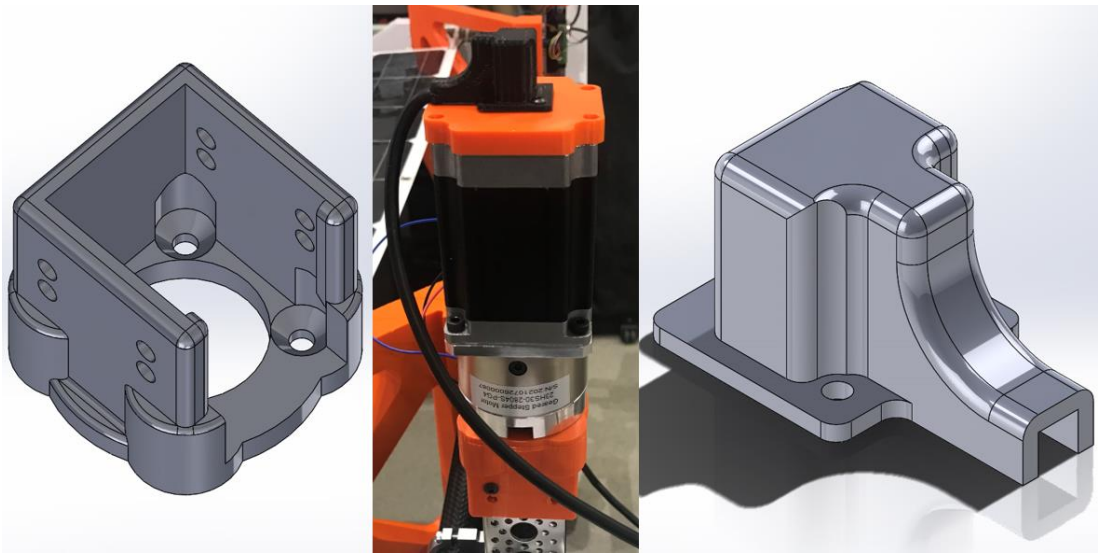


Figure 23: Stepper Motor U-Channel Mount CAD Model (left), Complete Assembly (middle), Encoder Cap CAD Model (right)

The author redesigned the 3D component with more material around the motor connection area after several instances of layer separation caused by rough handling during vehicle transit. The output shaft is attached to the carbon fiber robot leg with a friction fit collar and set screw. The output pulses from the microcontroller are converted into motor steps by four, TB6600 microstep drivers, which are also secured to the U-channel.

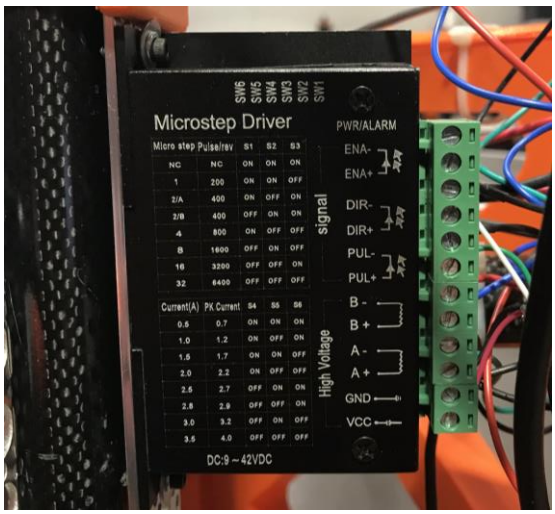


Figure 24: Microstepper

Because of the increased pin number, as well as the time delays when taking code blocking sensor readings, an additional Sparkfun RedBoard microcontroller, takes on the motor control functions of the platform [Appendix B].

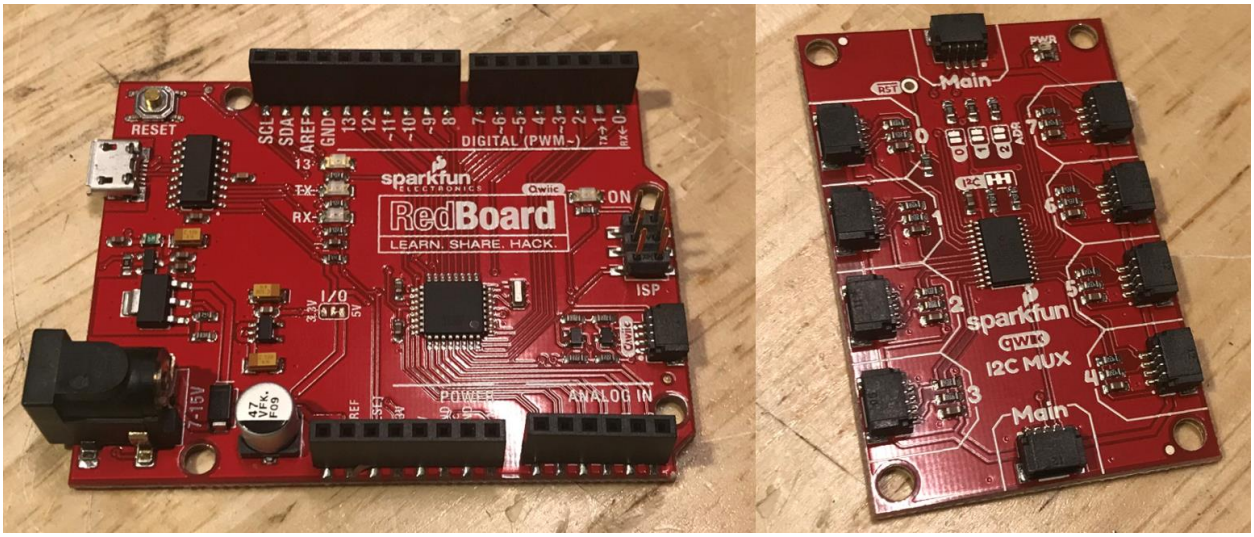


Figure 25: RedBoard (left), I2C Mux (right)

The Arduino-like board comes with a built-in quick disconnect, I2C port called a qiic port. This additional microcontroller communicates with the main computer code over USB. In addition, a Sparkfun I2C mux board was added to the control box to manage the encoder wires [Appendix

B]. The board switches between output ports allowing encoder feedback from all four motors while using only one quiic input port on the microcontroller.

3.4. Stronger Drive Motors

The window motors on the previous iteration were rated for a continuous torque of 2.9 Nm and were insufficient to move the weight of the larger robot up slight inclines and over rough ground. The previous iteration of the robot weighed in at 27.7 kg while the new version with all the reinforcements weighed in at 54.4kg without the final battery. After a short time at stall current, the built-in thermal fuse would trip, and the motor would shut off until it cooled down. A new 12V DC Bemonoc worm gear motor was selected to replace the old motors. The new motor had a rated continuous torque of 6 Nm which would have been able to propel the larger platform. However, because of quality control issues with these motors, a yet newer, higher quality motor from a more reputable supplier had to be selected.

Figure 26: Window Motor (left), Bemonoc Motor (center), AM Motor (right)

To properly specify the new motors, equation (5) was used to find the total force required to propel the vehicle, equation (6) to find the total torque required based on wheel diameter, and equation (7) to find the desired RPM:

$$Ftotal\ (N) = Crr * W + \frac{m * Vmax}{t} + W * \sin(\alpha) \quad \textbf{(5)}$$

$$Ttotal\ (Nm) = Ftotal * \frac{D}{2} * RF \quad \textbf{(6)}$$

$$Motor\ Speed\ (RPM) = Vmax * \frac{60}{\pi D} \quad \textbf{(7)}$$

Where Vmax is the maximum desired speed in m/s, Crr is the coefficient of rolling resistance, W is the robot weight, t is the time to accelerate to top speed, α is the maximum grade, m is the robot mass, D is the wheel diameter, and RF is the resistance factor of the gearbox and motor components. The motor specifications for torque are obtained by dividing the total required torque by the number of wheels on the robot. Values for Crr and RF were estimated using a reference table from a University of Florida Mechanical and Aerospace Engineering course [18]. Assuming a maximum grade of 3-5% and a robot speed of 0.45 m/s the necessary rated continuous torque of the motors needed to be around 6-7 Nm, and the speed needed to be at or above 28 RPM. The 226 series DC Gear Motor from AM Equipment met these requirements [Appendix B].

Because the 3D printed wheels were custom designed to fit the old motors, new couplers had to be made to attach the motor output shaft to the wheel. These were made by removing the old coupler from the burnt-out window motor, cutting it shorter, and machining a slot to accept the new motor shaft.

3.5. Robot Controller Adjustments and Part Changes

Because of the loss of the data dashboard app, the author devised a new method of robot control. The NI MyRio handled the PID control for the drive motors and the manual control functions. There are ways of integrating radio controllers for manual control and LabVIEW using FPGA. However, it was simpler to centralize all high-level control on the already-present Latte Panda single-board computer. The Panda's onboard Arduino is sufficient for the simple sensor communication, and the GPS already fed in data over USB serial. To manage the PID control of the drive motors, 4 Roboclaw Solo 60A motor controllers replaced the Sabertooth controller [Appendix B].

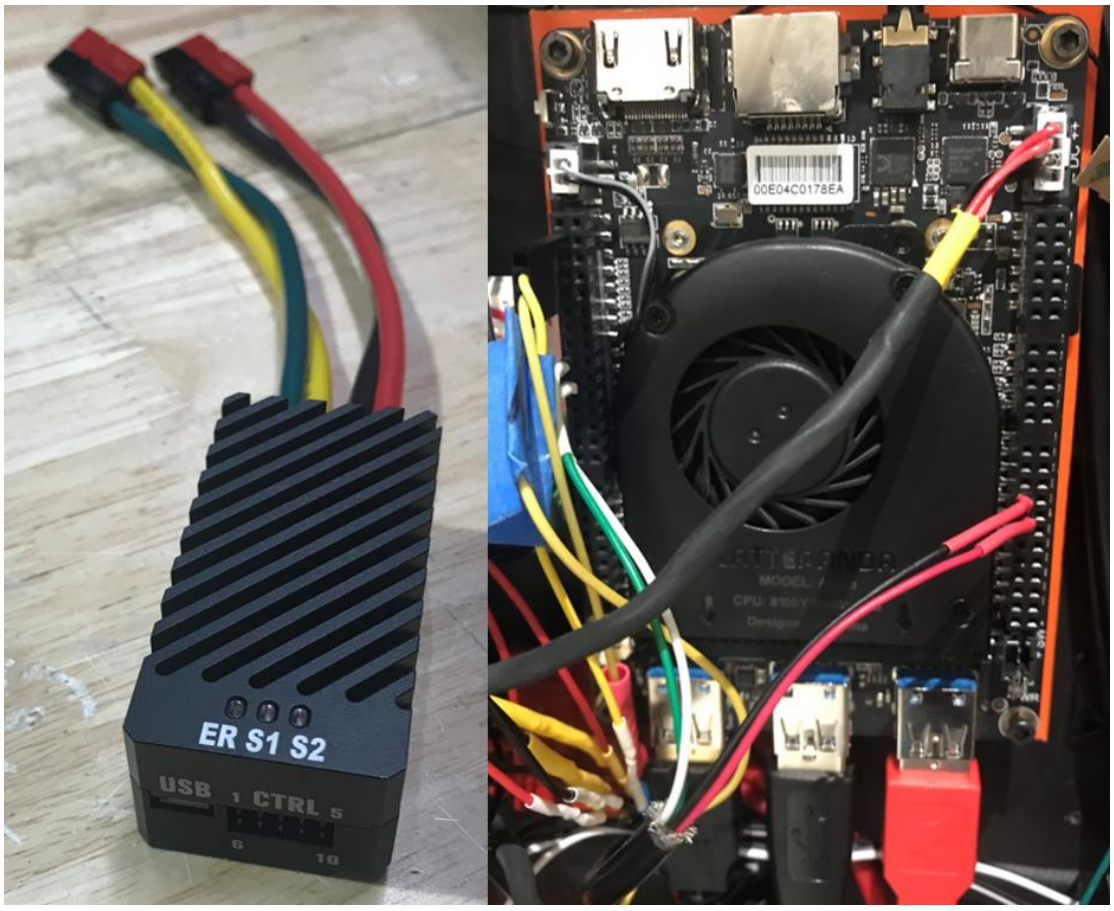


Figure 27: Roboclaw Motor Controller (left), Latte Panda Alpha (right)

The Solos come with built-in PID control and only require a speed command from the Redboard microcontroller mentioned above. The controllers had to be tuned to match the motor and wheels. This was accomplished by propping up the robot to allow the wheels to spin freely and writing new settings using Basic Micro's Motion Studio software.

3.6. Radio Antenna

A pair of new XBee Pro SB3 radio and antenna replaced the user input functions of the data dashboard [Appendix B]. The radio on the robot connects to the Latte Panda over USB. This radio is paired with the one connected to the operator's computer to send and receive messages from the Pancreas.

Figure 28: XBee Radio

The messages are sent asynchronously, meaning that the sending radio does not need a confirmation that its message has been received and so is non-blocking in the main control code. This method of communication allows for smoother robot control. In addition, because more

complex messages can be sent and received, debugging and testing are easier because robot feedback is more informative and customizable.

3.7. Power

The component additions, the motor changes, and the substantial increase in weight all increased power consumption. In addition, the new motors can briefly drop the voltage coming from the 12V regulator when at stall current. If the computer and the drive motors are on the same regulator, a drop in voltage, caused by motors drawing too much current, can shut off the computer. An additional regulator supplying the computer mitigates the voltage drop by providing a buffer. During testing, a larger provisional battery pack of two 22.2v 12.5 Ah, 277.5 Wh batteries wired in parallel was used until the precise power needs of the platform could be experimentally determined. A full wiring diagram of all components can be seen in Appendix C. In addition the updated CAD models can be found at BenW3/PancreasSolidModels: stl files for the robot (github.com).

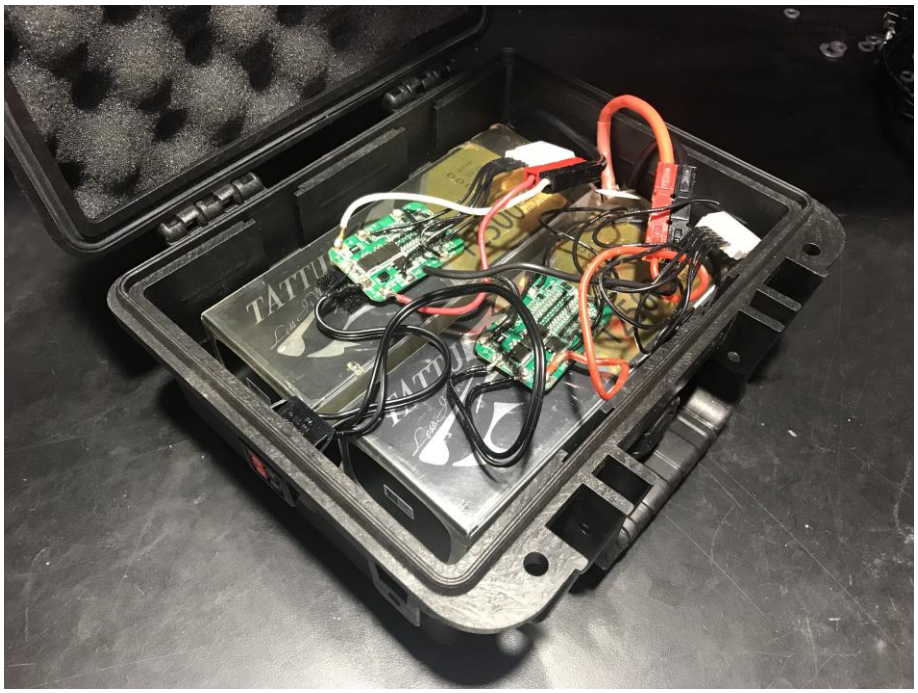


Figure 29: Provisional Battery Pack

4. Software Changes and Simulation for Prototype Three

4.1. Computer Control

With the move to solely using the Latte Panda, most of the control code already present on the previous iteration had to be re-made in Python, with the exception of the Arduino sensor code. The sensor code draws from Dahm's work and a helpful AS5600 encoder tutorial, which also inspired the design of the mount to which the encoder is attached [19]. Both the sending and receiving radio interface also had to be implemented. In addition, because the implementation on the previous prototype had only the manual method of control, the author had to design an autonomous path following strategy. The author switched the onboard computer's operating system to Linux because of its stability and reduced memory use. In, addition Linux is somewhat easier to use when implementing computer vision. The code and measured data are written to a detachable SD card which can be removed for either data analysis or code updates. The main body of the code runs from the Latte Panda terminal on startup. There is a radio command handler file and a function file which together make up the high-level control of the robot. The main Python code running on the Panda issues commands over serial to the two microcontrollers on the platform. The main code issues a serial command to the RedBoard microcontroller with a single angle for the steering and a duty cycle for the drive motors. The other built-in Arduino Leonardo microcontroller is issued a letter corresponding to the desired sensor reading and returns the reading over serial. The main code reads GPS by parsing the NEMA string that is continuously generated by the B-125 board. A block diagram of the system and information flow can be seen below. In addition, the full code for all systems is located on GitHub at:

https://github.com/BenW3/Pancreas. The most essential portions can also be found in Appendix A of this document.

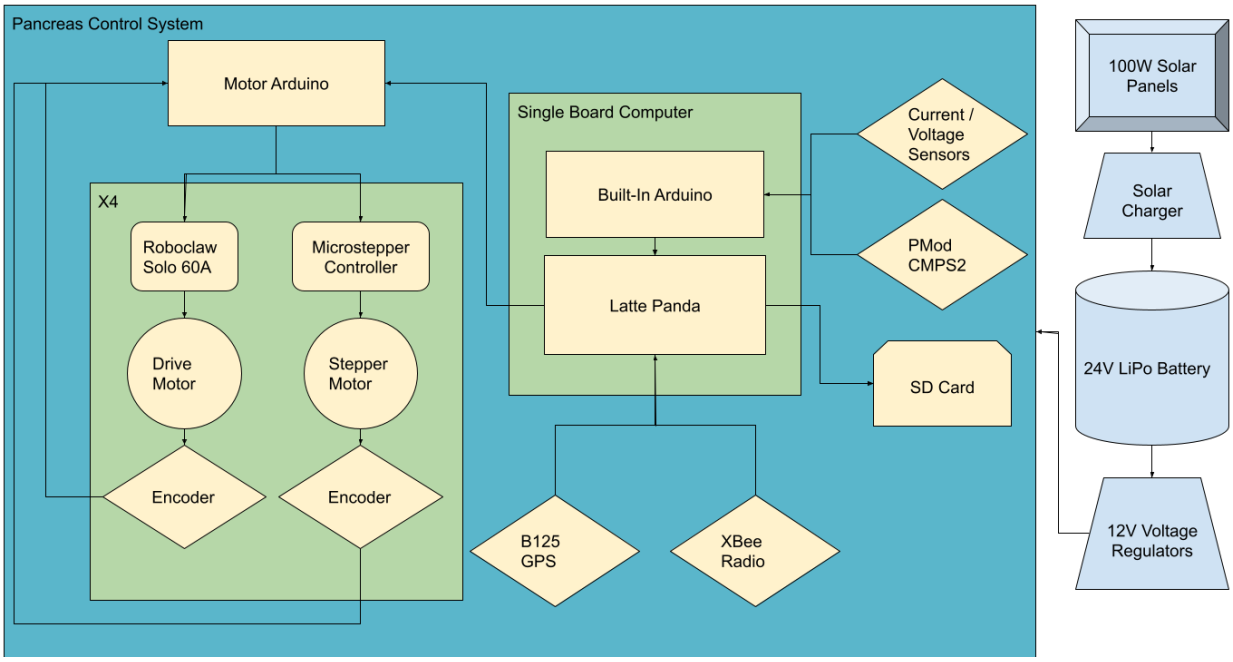


Figure 30: System Block Diagram

4.2. Radio Communication

The Xbee radio connects to an operator's laptop. This method of radio control is preferable to an RC controller because of its the ease with which new features and commands can be added. The operator can send a variety of text commands through the terminal to receive real time sensor data from the compass, GPS, voltage and current sensors, and control the platform in either manual or autonomous mode. The user can manually control the robot with the W, A, S, and D keys while recording GPS waypoints and set the speed with a text command.

4.3. Steering Methods

Like the previous iteration of the Pancreas, the current version also uses a form of Ackerman steering. In this iteration of the prototype, it is double-Ackerman steering, where both

the front and back wheels always have the same center around which they steer. This can enable a zero-turn radius, with the center of the turn in the middle of the robot, and it can reduce wheel slip.

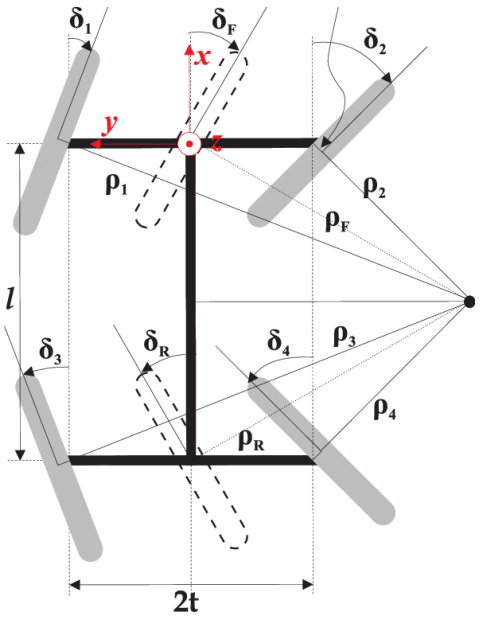


Figure 31: Double Ackerman Steering [20]

The high-level control code running on the Latte Panda sends the total desired turn angle for the robot to the motor control RedBoard where that signal is converted into the correct angles for each wheel. The equations to translate the desired turn angle into actual wheel angles are shown in (8) and (9):

$$\delta_1 = \tan^{-1}\left(\frac{l * \sin\theta}{l * \cos\theta + w * \sin\theta}\right) \quad \textbf{(8)}$$

$$\delta_2 = \tan^{-1}\left(\frac{l * \sin\theta}{l * \cos\theta - w * \sin\theta}\right) \quad \textbf{(9)}$$

Where δ_1 and δ_2 are the front left and front right wheel angles respectively, l is the robot length, w is the robot width, and θ is the desired robot turn angle. The rear wheel angle is simply the

negative of δ_1 and δ_2. The micro-controller then closes the loop by checking each desired wheel position against the actual position and adjusts the wheel angles with pulses sent to the microstepper. This was done with a custom Arduino library.

Because control and power cables necessarily run from the electrical box to the wheels, the legs of the robot cannot spin freely, otherwise those cables would wrap around the leg and jam it or disconnect themselves. To stop this, the stepper motors' turn angle has been limited by code to plus or minus 90°. When an angle greater than 90° or less than -90° is needed the stepper motor will turn 180° from the necessary angle and the drive motor will reverse direction. Although smoother steering is possible with a larger range of permitted turn angles, the disruption from jammed cables is not worth the gain.

When in manual control, the operator can also activate a synchronous steering mode from the control laptop and turn each wheel at the exact same angle. Switching between the two modes as needed allows for excellent maneuverability in confined spaces. The second mode was necessary for transportation to and from the field and for debugging purposes because the increased weight made the platform difficult to move physically.

4.4. GPS Waypoint Following

In order to follow a given set of GPS waypoints, the author selected and implemented the pure pursuit algorithm. Pure pursuit takes a set of line segments as input and finds a target point on the line segment currently being traversed based on some look-ahead radius. It then calculates the angle between the direction the platform is currently facing and the angle required to intersect the line segment at that look ahead point.

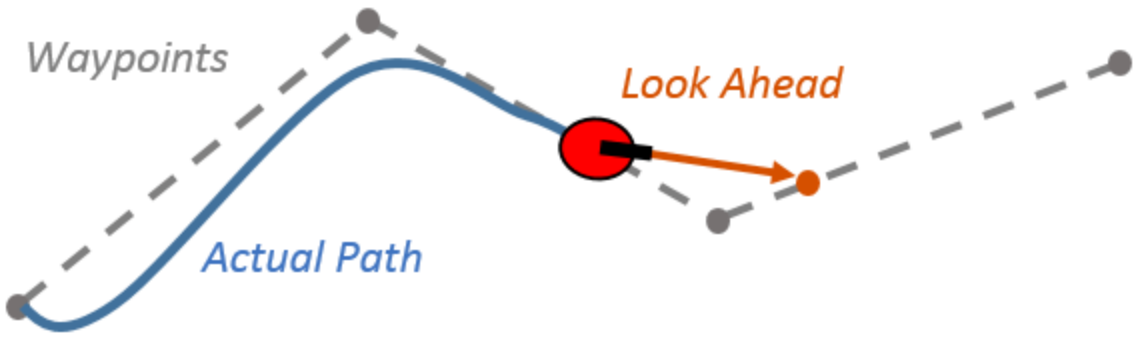


Figure 32: Pure Pursuit [21]

When the robot moves, the lookahead point also moves further along the line segment and will result in a new desired angle. This method produces smooth path following with the right adjustments to the lookahead radius because it gives the platform some time to adjust in advance of sharp turns, mitigating overshoot. In addition, the aggressiveness of the error correction can be tuned with PID control.

The GPS data itself is read every time the waypoint function loops around. This frequent reading is allowed by the 100Hz data update frequency of the B-125 board. The geometry done to determine the required correction angle is done in cartesian coordinates, so the raw latitude and longitude data needs to be converted. This is done by simple equirectangular projection, which flattens the coordinates down to a plane, so calculations need not be done on a spherical surface [22]. This method is crude, but effective for small scales. The equations are shown in (10) and (11).

$$x = R * \lambda * \cos(\varphi_{ref}) \quad \textbf{(10)}$$

$$y = R * \varphi \quad \textbf{(11)}$$

Where λ and φ are longitude and latitude converted to radians, respectively, R is the radius of the earth, and φ_{ref} is some reference latitude. This correction gives the appropriate circle of latitude to scale the x coordinate by at the reference point. In the case of the path following algorithm, the reference latitude is simply the first coordinate in the path. This centers the projection on wherever the platform happens to be.

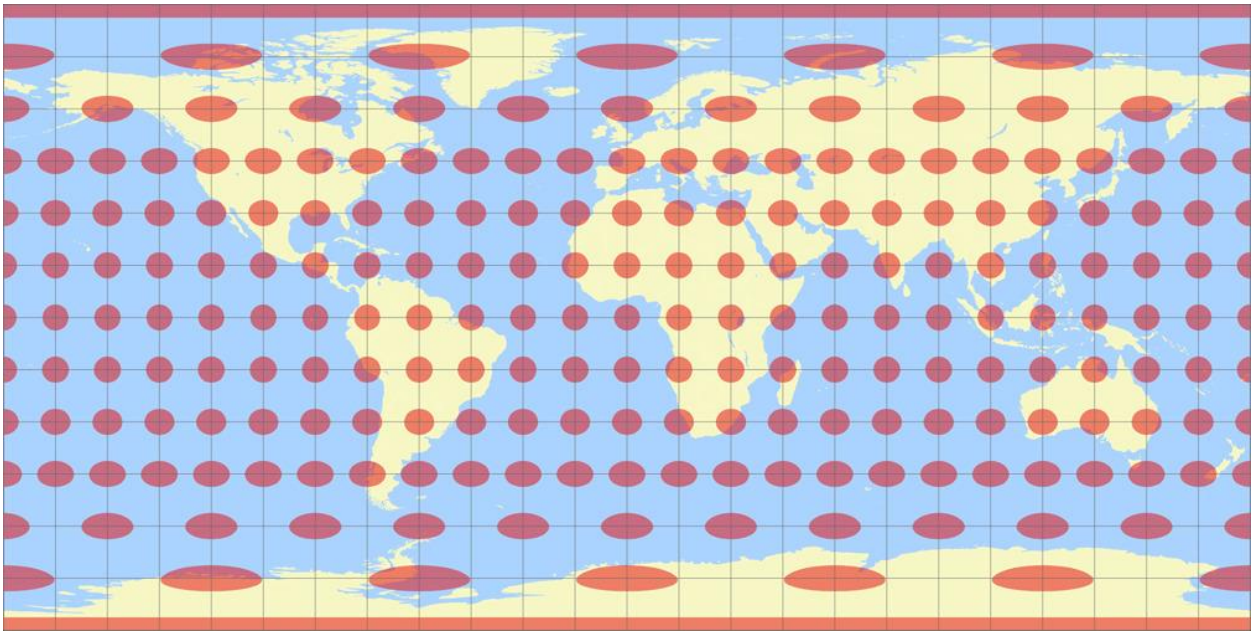


Figure 33: Equirectangular Projection Distortion, Reference Latitude at 0° [23]

The projection, like all global map projections, loses accuracy the further out from the reference coordinate the robot goes. In this case, it is north-south distance that is concerning. For the application considered here, a deviation from true distance of more than a few centimeters over a 40m span would be unacceptable. Based on my calculations the platform would need to travel almost 2km before the latitude to x coordinate conversion was distorted by 1cm over that span, which is well outside the operating range of the platform during field trials. The equations used are shown in (12) and (13).

$$Distance = R * \Delta\lambda * \cos(\varphi_{ref})$$

(12)

$$Distance + Tolerable\ Error = R * \Delta\lambda * \cos(\varphi_{failure})$$

(13)

Where "*Distance"* is the typical span of a test wheat plot and "*Tolerable Error"* was 1cm. After doing some simple algebra these equations can be rearranged to find the band of tolerable latitudes for navigation and with that the tolerable north-south range. The equation is shown in (14).

$$Tolerable\ Range = R * (\varphi_{ref} - \cos^{-1}(\cos(\varphi_{ref}) * \frac{Distance + Tolerable\ Error}{Distance}))$$

(14)

The initial path coordinates can be either imported from an external source, or taken by manually navigating through the desired path and recording the data with a command from the user. GPS, compass, time, and power data is collected both while recording and following a path. The data is then logged to a comma separated value file (.csv) for later analysis.

4.5. Path Following Simulation

In addition to physical testing, simulation was a useful tool for diagnosing errors and refining the control algorithm with quick turnaround time. The Webots open-source robotics simulator from Cyberbotics provided a simple, easy to use tool for the project. A simplified model of the robot frame was constructed in Webots with controllable steering, drive motors and virtual sensors. The simulation used a slightly simplified version of the Python code running on the robot for testing and debugging.

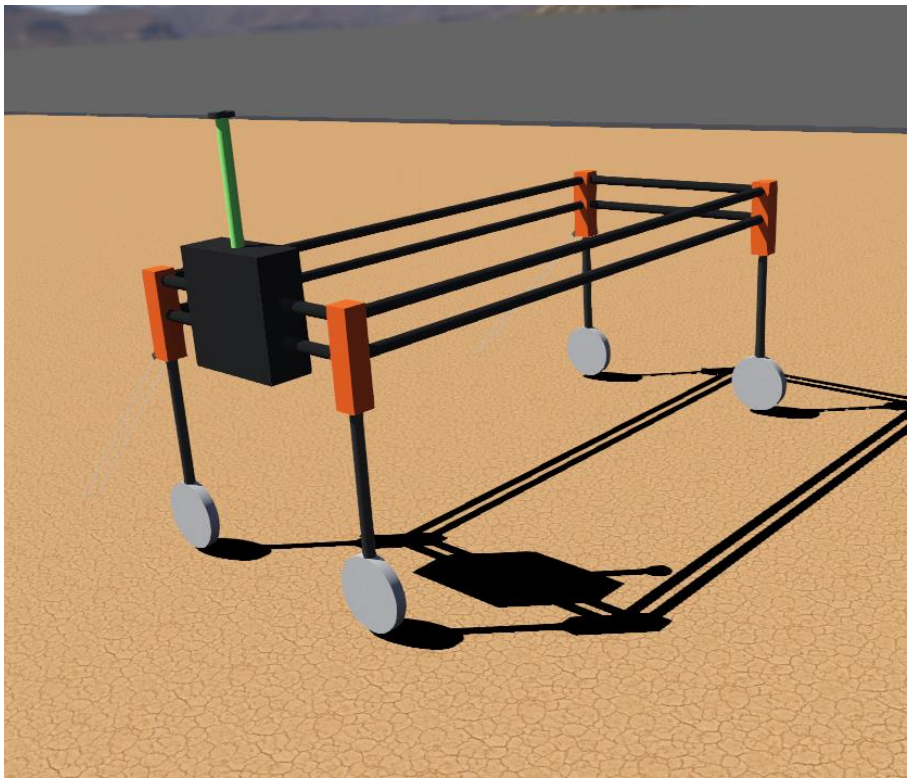


Figure 34: Webots Pancreas Simulation

After some changes, the simulated Pancreas performed reasonably well on a test course which approximated six passes through the test field. A look-ahead radius of 0.25 m, a proportional gain of 3.6, and integral and derivative gains of 0 performed well during testing. The simulation also gave a helpful estimate for how much space the platform needs to turn and its over/undershoot when path following to fall within tolerable margins. In addition to providing a benchmark for physical test results, the simulation also aided in resolving some persistent errors in the control code. A plot of performance alongside error can be seen in the figure below. The path traversed was roughly analogous to real test field conditions, matching the layout of the wheat plots.

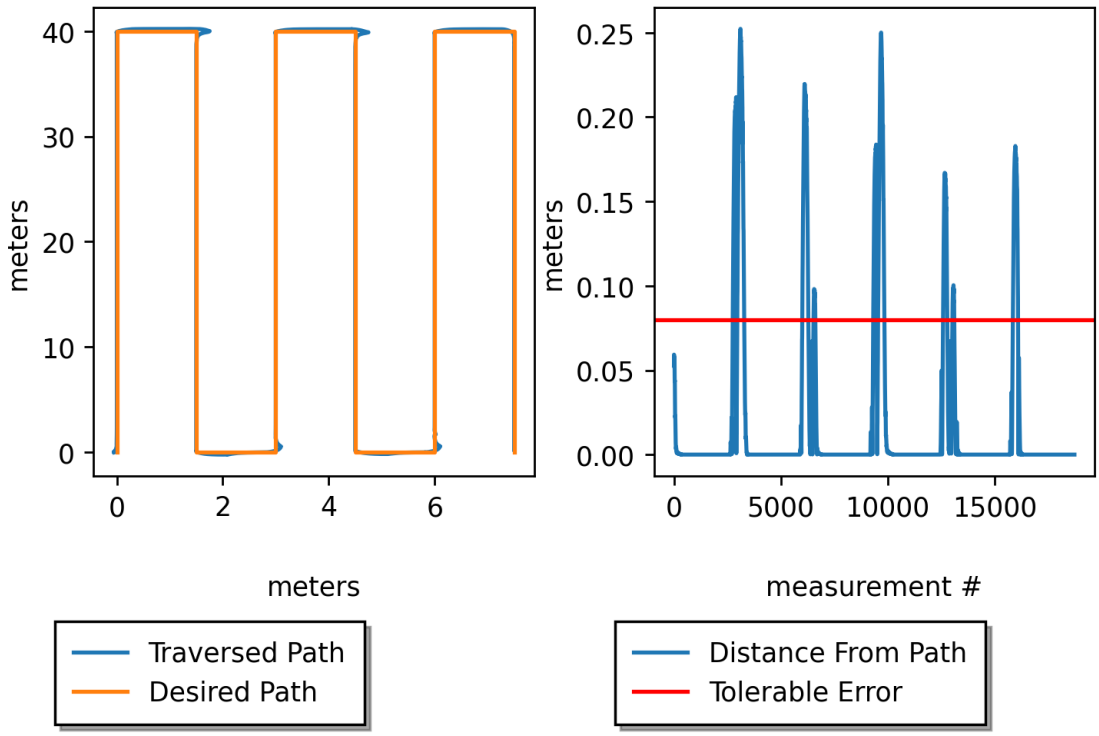


Figure 35: Simulated Robot Path and Tolerable Error Plot (NOTE: path not to scale)

The platform has 15 cm on either side of the wheel, assuming it begins centered in the 30 cm path, before it collides with the wheat plots. The thickness of the fenders and motor projecting from the side of the fender further reduces this distance by a total of 14.15 cm, to a margin of just under 8 cm on either side. In addition to precise centering on the waypoint path, the platform needs to be aligned with the path in terms of heading. A 3° deviation in heading will cause a collision with the wheat plots. This becomes an issue on the physical system, because the digital compass has a maximum error rating of 3°. With these settings, the transitory behavior subsides in roughly a meter. A plot of the heading error can be seen in the figure below. These plots suggest that the platform should be given a path that overshoots the ends of the test plot by a margin of a few meters to avoid collision.

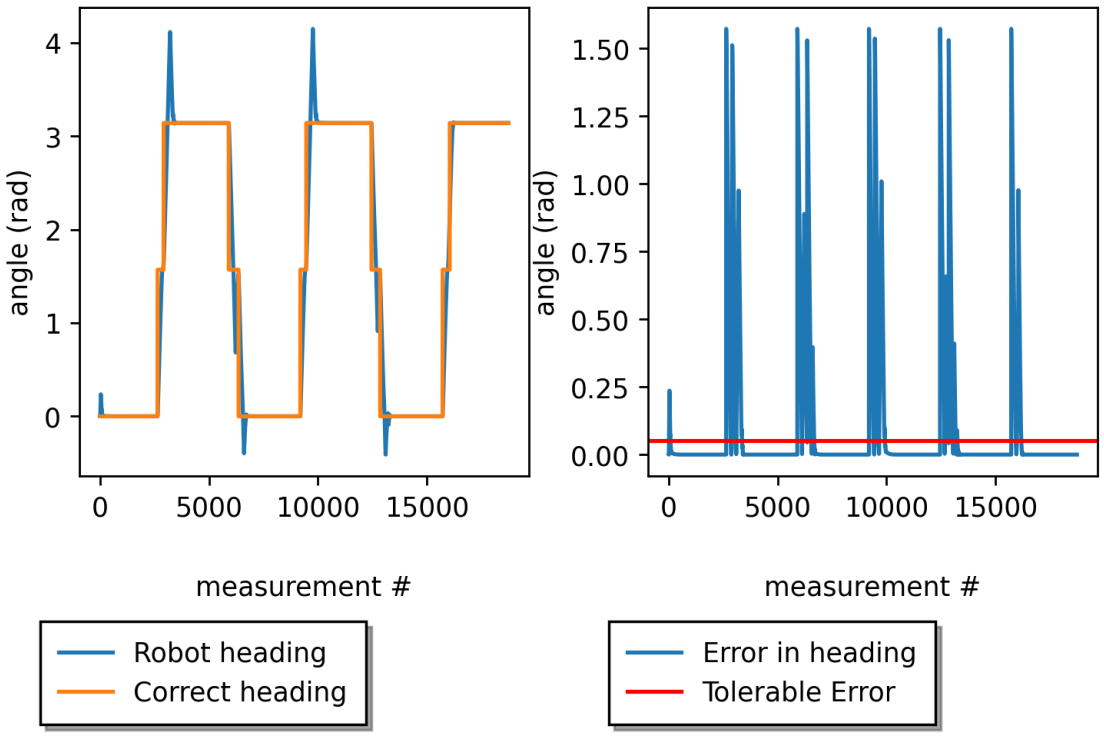


Figure 36: Simulated Robot Heading and Error

There are several key differences between the simulation and the physical platform. First, the simulated robot behaves as a rigid body with no deforming parts. Second, there is no error in the GPS and compass signal, and these signals have no delay from signal request to signal reading. Fortunately, there are signal processing tools like the Kalman filter which can resolve some of these real-world sensor errors, as detailed in the next section. Finally, the virtual terrain is perfectly flat, so there are no environment related disturbances to the path following. These differences could result in different values for the lookahead radius and PID coefficients, but the simulation provides a starting point from which to fine-tune the control coefficients.

4.6. Kalman Filtering

Rudolph E. Kalman designed the Kalman filter and published it in a 1960 article in The Transactions of the ASME [24]. The filter enables the smoothing of noisy measurements and

accurate estimations of a system's state. In the case of the Pancreas platform, the filter will be used to smooth its GPS signal by combining the raw GPS input with the compass readings and the assumed constant speed of the platform. The degree to which the data is smoothed can be adjusted by tuning the measurement noise covariance matrix. This matrix, called Q, represents how noisy the data is and how much it should be trusted. The filter works by updating the state of the system, in this case position, with the velocity estimate from the previous time step. It then compares this estimate with the current measurement and combines them based on how much it trusts the measurement. A visual of this can be seen in the figure below, from a University of North Carolina computer science course [25].

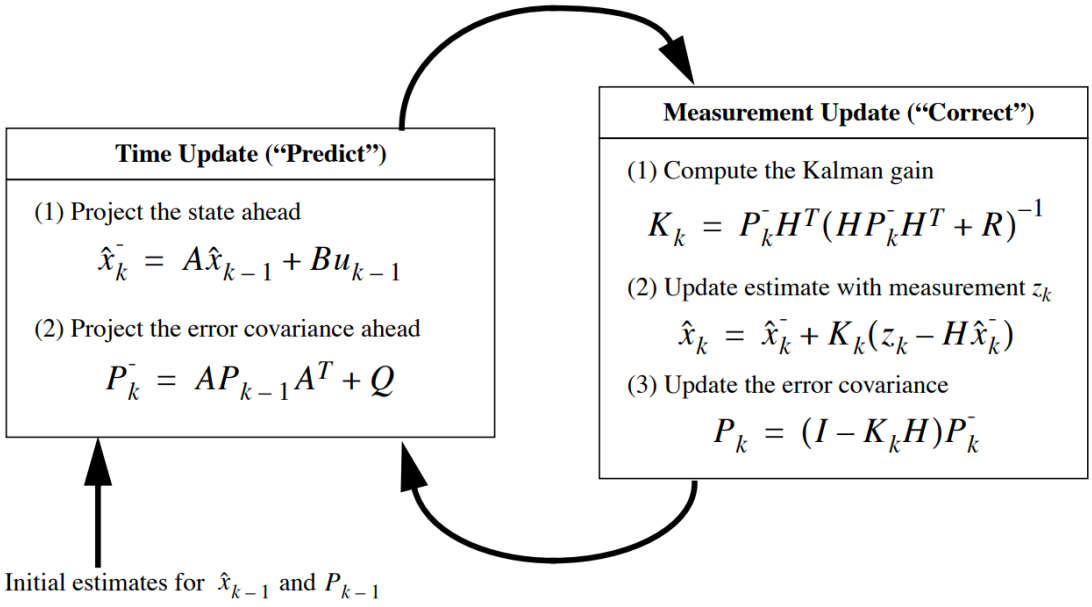


Figure 37: Basic Kalman Filter Function [25]

The process is notation heavy, but not difficult to implement in code. The author took inspiration from an online Python implementation of a Kalman filter [26]. However, the author made several changes to the state, covariance, and measurement matrices to reflect the

observations, which are represented by the $\boldsymbol{z_k}$ matrix, made by the platform as these are application specific, in this case being velocity and position. An example of smoothing is shown below in the following figures.

Figure 38: Raw Recorded Path

Figure 39: Filtered Path

Notably the accuracy of the GPS readings in the preceding figures was severely impacted by the presence of buildings, leading to a shift in the recorded waypoints by over a meter on portions of the path. This interference would not be present in the field but does demonstrate the effect of Kalman filtering. The actual path roughly matches the left-hand side of the loop shown in figure 39, but diverges as it travels back down the sidewalk.

5. Testing and Results

5.1.Path Following

Several issues with the frame and sensors of the robot led to path following failure. The frame has been in use for over a year, and wear on multiple components severely hampers accurate navigation. The connection between the steering motors and the robot's legs experienced wear to both the couplers and carbon fiber leg, leading to roughly 15° of slop in wheel angle. This causes the legs and frame to twist and bend, and it makes precise steering impossible. In addition, the Roboclaw motor controllers periodically fail, leading to one or more wheels being locked in place. Additionally licensing and proof of purchase issues led to an inability to acquire the RTK functionality of the GPS. Finally, during testing the author found that, despite being on a long pole away from the electronics box, the GPS antenna experienced radio interference from the computer and electrical box. The author determined this by testing GPS satellite acquisition with and without the computer turned on.

Figure 40: Radio Interference Solution

To resolve this, a large aluminum plate was secured on the pole under the antenna which greatly improved satellite acquisition and lock. Another potential fix mentioned in committee was to use thicker gage wire and the fewest ground points possible. Despite this interference fix, all the other issues resulted in unsatisfactory path following.

5.2. Updated Power Consumption

Several short tests pointed to the provisional battery pack being insufficient. To preempt power loss during longer tests, the author swapped the pack for a 24 V, 50 Ah, 1200 Wh, LiFePO4 battery from Dakota Lithium [Appendix B]. This battery added an extra 9 kg to the platform, but increased the battery life by over four times.

Figure 41: Dakota Lithium Battery

Field testing demonstrated an average power use of 218 W as shown below, which is over 3 times greater than the previous prototype. The platform struggled to exit a ditch which produced the power use spike at the end of the plot. This reflects off road conditions, and so was not removed as an outlier.

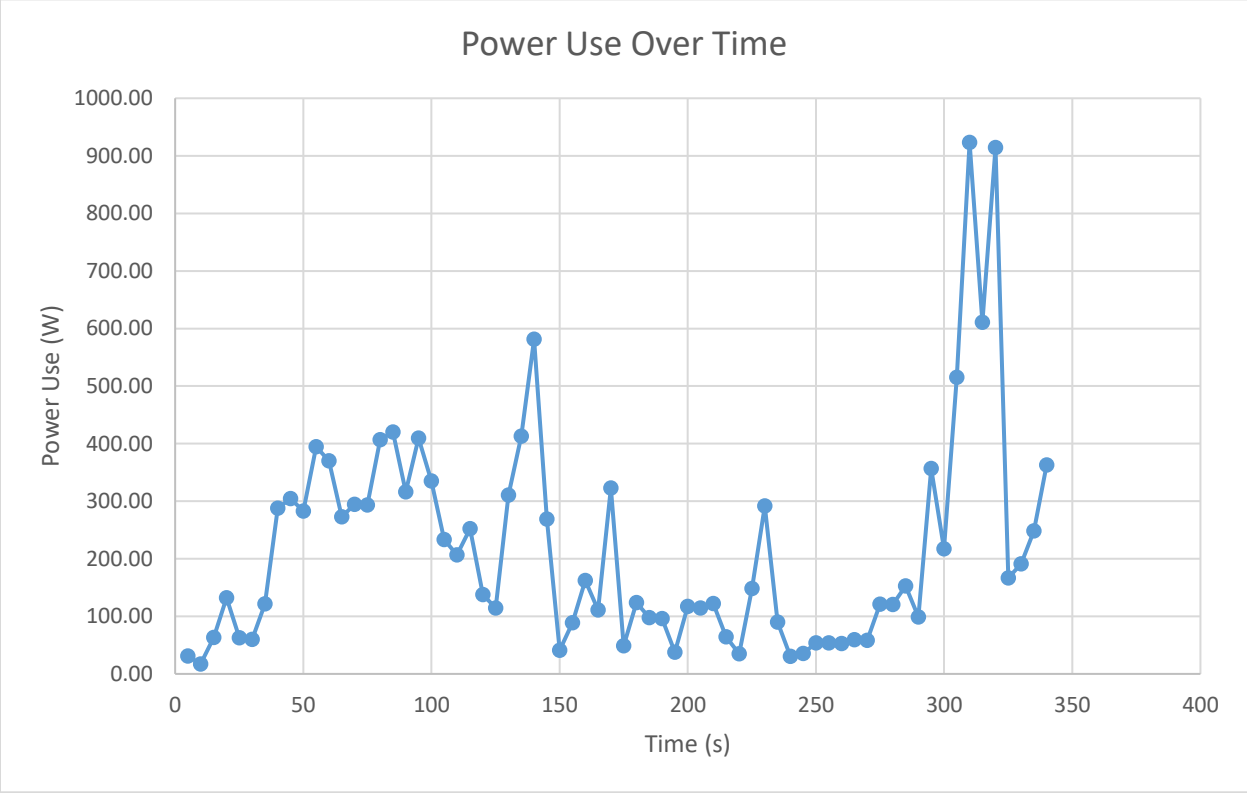


Figure 42: Power Use During Field Test

Based on these measurements, the platform can operate solely on battery for almost six hours. However, the current solar panel arrangement's limitations mean that the platform would have to stop, shut off power to the motors, and charge for part of the day. Dahm's solar panel trials give the total expected incoming power from the two panels as ~100 W during sunny, mid-day conditions and ~50 W during cloudy conditions [16]. Based on Dakota Lithium's charging specifications, the 24 V battery should be charged at 28.8 V and under 0.3 C (15 A) [27]. The

current solar panel setup would take 14.4 hours under ideal conditions to charge completely as it can maintain a current of just under 3.5 A at that voltage. Changes are necessary to avoid these excessive charging times.

Manhattan, Kansas receives, on average, 4.24 kWh per day per kilowatt of installed photovoltaic capacity [1] [28]. The average for April through September, which are the months of expected use, is 15% higher than the yearly average. This gives a daily average of 4.87 kWh/kWp during those months.

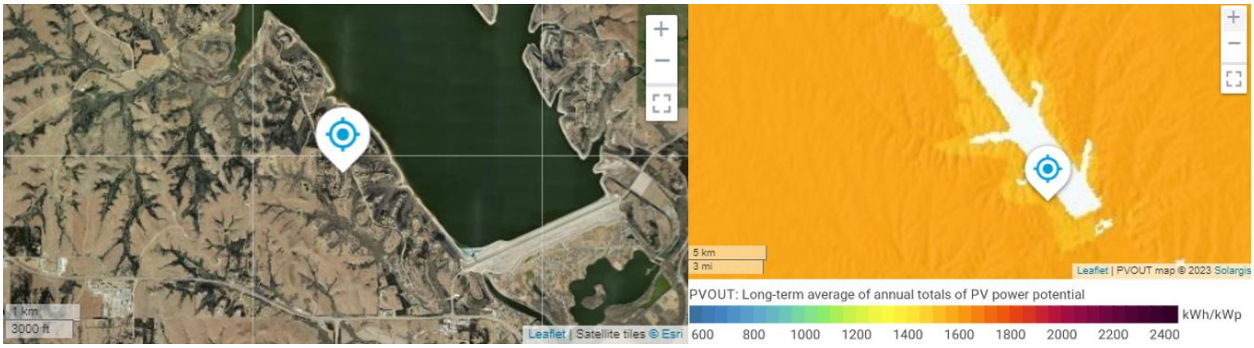


Figure 43: Satellite and Solar Potential Maps of Manhattan, KS [28]

Since the Pancreas has an effective 100 W of installed capacity, we can expect a daily average power budget of 487 Wh from average incoming sunlight. Based on the power use during regular operation, that gives 2.2 hours of run time for the platform. Given the power demands, the number of solar panels on the platform should be doubled supplying a power budget of 974 Wh. This would bring the operation time of the platform to 4.47 hours. This run time should be further reduced to compensate for idle power use. The platform draws just under 14 W when idle with the motors unpowered. Over the course of 24 hours this will drain 336 Wh. This decreases

[1] Information obtained from the Global Solar Atlas 2.0, a free, web-based application is developed and operated by the company Solargis s.r.o. on behalf of the World Bank Group, utilizing Solargis data, with funding provided by the Energy Sector Management Assistance Program (ESMAP) **[Statement required in terms of use]**

the operation time to 3.1 hours.

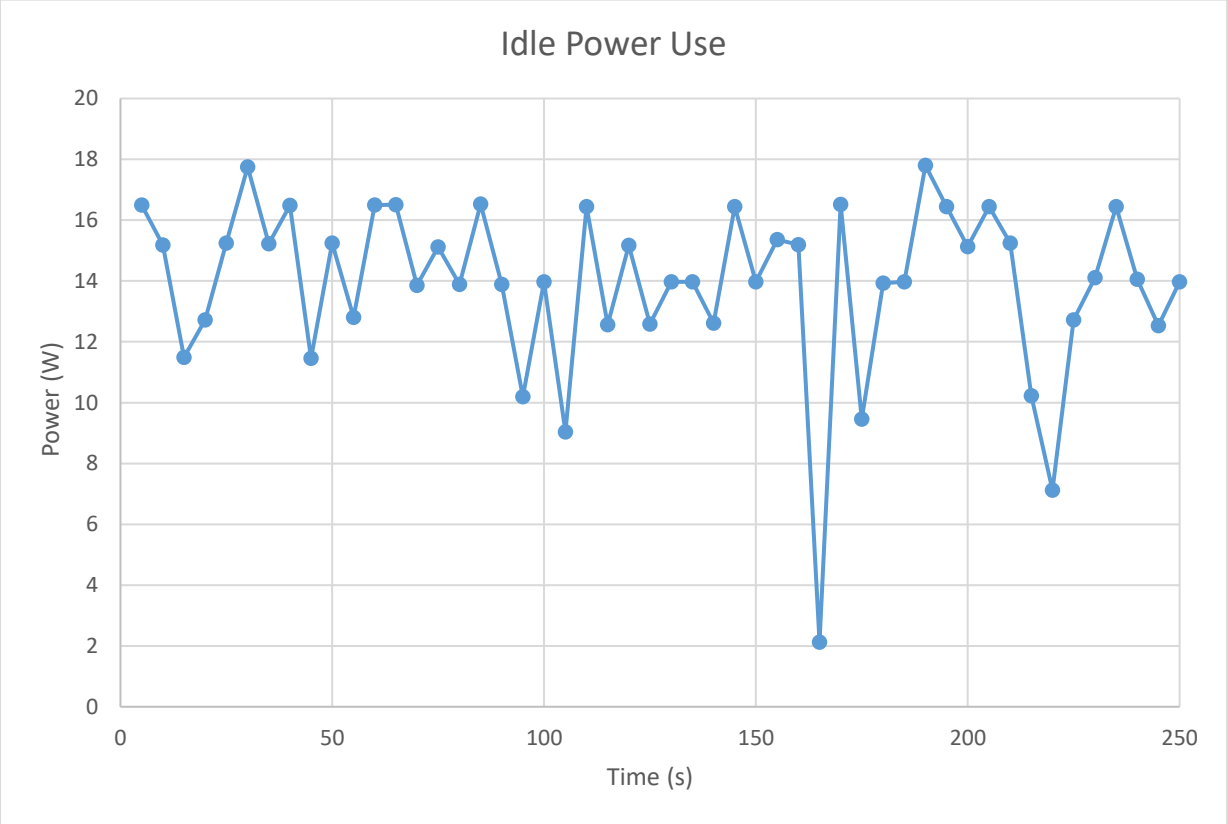


Figure 44: Idle Power Use

Idle power use was tested by shutting off power to the motors with a simple switch, but should be done with a 20 A relay which could be switched autonomously by the Latte Panda. The start and stop of operations should be determined by monitoring battery capacity and incoming solar power to check against experimentally determined threshold values. Then, when running, the platform should check its internal clock against its allotted daily runtime.

6. Conclusion and Future Work

The third iteration of the Pancreas unmanned ground vehicle makes several improvements to the previous prototype. First, the platform is larger to reduce interference from the motors and electronics on the EMI sensor. Next, it has a more centralized control system that leverages the power of the Latte Panda computer. It has more durable components to support the size increase. Finally, the Pancreas has a path following algorithm implemented and tested in simulation to follow GPS waypoints.

Future work is still necessary for the platform to reach its potential. Some issues are simple fixes and others require more in-depth solutions. A clerical error in measurement led to the Pancreas being a 30 cm shorter than necessary, so new 3.35 m long tubing is needed. The autonomous path following algorithm needs to be updated to include an automatic shut off when entering a dormant, charging state. The steering motor system needs improvements to its durability because some of the 3D printed couplers receive excessive wear. These components are now degrading the precision of the platform's steering and navigation. A more permanent solution would be desirable to simply printing new components of the same design. The accuracy of robot heading measurements is insufficient for navigation, so a new compass or compasses must be installed. RTK functionality needs to be implemented on the GPS system. To improve the Kalman filter, an inertial measurement unit (IMU) should be added to accurately gauge speed. The platform also needs to be weatherized. Finally, the obstacle avoidance system still needs to be implemented to provide feedback from the environment.

Simulation demonstrates that the Pancreas can work, but reliability remains a key issue for future development. The system is not yet ready for field use, but could be given these adjustments.

References

[1] United Nations, Department of Economic and Social Affairs, Population Division, "World Population Prospects: The 2017 Revision, Key Findings and Advance Tables," United Nations, New York, 2017.

[2] Our World In Data, "World population by region, including UN projections," Global Change Data Lab, 2022. [Online]. Available: https://ourworldindata.org/grapher/world-population-by-region-with-projections?time=1914..latest. [Accessed 1 March 2023].

[3] D. Tilman, C. Balzer, J. Hill and B. L. Befort, "Global food demand and the sustainable intensification of agriculture," *Proceedings of the National Academy of Sciences,* vol. 108, no. 50, pp. 20260-20264, 2011.

[4] N. Alexandratos and J. Bruinsma, "World agriculture towards 2030/2050: the 2012 revision," *ESA Working Papers 12-03,* 2012.

[5] M. C. Hunter, R. G. Smith, M. E. Schipanski, L. W. Atwood and D. A. Mortensen, "Agriculture in 2050: Recalibrating Targets for Sustainable Intensification," *BioScience,* vol. 67, no. 4, pp. 386-391, 2017.

[6] P. Webb and S. Block, "Support for agriculture during economic transformation: Impacts on poverty and undernutrition," *Proceedings of the National Academy of Sciences* , vol. 109, no. 31, p. 12309–12314, 2012.

[7] Food and Agriculture Organization of the United Nations, "Agricultural Production Statistics 2000-2021," *FAOStat Analytical Brief,* 2021.

[8] Statista, "Worldwide production of grain in 2022/23, by type (in million metric tons)* [Graph]," FAO, & US Department of Agriculture, 8 February 2023. [Online]. Available: https://www.statista.com/statistics/263977/world-grain-production-by-type/. [Accessed 3 March 2023].

[9] Food and Agriculture Organization of the United Nations, "Statistical Yearbook: World Food and Agriculture 2021," FAO, Rome, 2021.

[10] FAOSTAT, "Crops and livestock products," Food and Agriculture Organization of the United Nations, 2023. [Online]. Available: https://www.fao.org/faostat/en/#data/QCL. [Accessed 1 March 2023].

[11] FAOSTAT, "Cereal import dependency ratio (percent) (3-year average)," Food and Agriculture Organization, 2018. [Online]. Available:

https://data.un.org/Data.aspx?d=FAO&f=itemCode%3A21035. [Accessed 1 March 2023].

[12] A. L. Coulibaly, "The Food Price Increases of 2010-2011: Causes and Impacts," Library of Parliment, Ottawa, 2013.

[13] S. Welch, P. Alderman and F. F. Fotou, "RII Track-2 FEC: Building Field-Based Ecophysiological Genome-to-Phenome Prediction," National Science Foundation, Alexandria, 2018.

[14] D. Gomez-Garcia, F. Rodriguez-Morales, S. Welch and C. Leuschen, "High-Throughput Phenotyping of Wheat Canopy Height Using Ultrawideband Radar: First Results," *IEEE Geoscience and Remote Sensing Letters,* vol. 19, pp. 1-5, 2022.

[15] W. Yang, H. Feng, X. Zhang, J. Zhang, J. H. Doonan, W. D. Batchelor, L. Xiong and J. Yan, "Crop Phenomics and High-Throughput Phenotyping: Past Decades, Current Challenges, and Future Perspectives," *Molecular Plant,* vol. 13, no. 2, pp. 187-214, 2020.

[16] C. Dahms, "Pancreas Robot, A Thesis," Kansas State University, Carl and Melinda Helwig Department of Biological and Agricultural Engineering, Manhattan, 2022.

[17] AutoZone Inc., "Repair Guides," AutoZone Inc., 2023. [Online]. Available: https://www.autozone.com/diy/repair-guides. [Accessed 15 March 2023].

[18] College of Mechanical & Aerospace Engineering, "EML 2322L Drive Wheel Motor Torque Calculations," University of Florida, Gainesville.

[19] Curious Scientist, "AS5600 magnetic position encoder," 5 March 2021. [Online]. Available: https://curiousscientist.tech/blog/as5600-magnetic-position-encoder. [Accessed June 2022].

[20] C. Candido, "Auxiliary mechanisms for telerobotics.," NOVA University Lisbon, Lisbon, 2009.

[21] The MathWorks, Inc., "Pure Pursuit Controller," The MathWorks, Inc., [Online]. Available: https://www.mathworks.com/help/robotics/ug/pure-pursuit-controller.html. [Accessed 15 March 2023].

[22] E. W. Weisstein, "Equirectangular Projection," MathWorld--A Wolfram Web Resource, 13 March 2023. [Online]. Available: https://mathworld.wolfram.com/EquirectangularProjection.html. [Accessed 15 March 2023].

[23] T. Jung, "Equirectangular (0°) Map Projection Image," Compare Map Projections, 17 April 2011. [Online]. Available: https://map-projections.net/license/rectang-0:tissot-15-ssw. [Accessed 15 March 2023].

[24] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME--Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35-45, 1960.

[25] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," University of North Carolina, Chapel Hill, 2006.

[26] R. Sadli, "Object Tracking: 2-D Object Tracking using Kalman Filter in Python," Machine Learning Space, 26 February 2020. [Online]. Available: https://machinelearningspace.com/2d-object-tracking-using-kalman-filter/. [Accessed 13 March 2023].

[27] Dakota Lithium, "How to Charge Dakota Lithium and LiFePO4 Batteries," Dakota Lithium Batteries, 22 September 2021. [Online]. Available: https://dakotalithium.com/2021/09/22/how-to-charge-dakota-lithium-and-lifepo4-batteries/. [Accessed 10 April 2023].

[28] Solargis s.r.o, "Global Solar Atlas," Solargis s.r.o, 19 January 2023. [Online]. Available: https://globalsolaratlas.info/detail?c=38.812961,-97.007904,9&s=39.264158,-96.622009&m=site. [Accessed 11 April 2023].

[29] Y. Jiang, C. Li and A. H. Paterson, "High throughput phenotyping of cotton plant height using depth images under field conditions," *Computers and Electronics in Agriculture*, vol. 130, pp. 57-68, 2016.

Appendix A - Code

Command Handler/Main

```
#!/usr/bin/python3

import methods
from digi.xbee.devices import XBeeDevice, RemoteXBeeDevice, XBee64BitAddress
import os
import glob
from math import cos, sin
import sys
from time import perf_counter
import numpy as np
import traceback
# --------------------
# Pancreas Main Code
# --------------------
# ................
# Necessary code blocks to write
# ................
# - CHECK - PID control
# - CHECK - Error calculator
# - CHECK - Angle/Dist to pwm calculator
# - CHECK - GPS reader
# - CHECK - Compass reader
# - CHECK - Read/write to coordinate file
# - CHECK - Stop conditions
# - CHECK - Read/write to log file
# Obstacle sensor
# - CHECK - Read power
# ................
# --------------------

# --------------------
# Global Variables
# --------------------
powerReadingDelay = 10
aspectRatio = 0.0
average_timestep = 0.5
methods.getSerialPorts()
receiver = XBeeDevice(methods.radioPort, 9600)
```

```
remoteTransmitter = RemoteXBeeDevice(
    receiver, XBee64BitAddress.from_hex_string("0013A2004104110E"))
receiver.open()
print(str(methods.initArduinos()))


# --------------------
# Main Radio Message Handler
# --------------------
if __name__ == "__main__":
    print("Pancreas Online")
    # --------------------
    # Read Radio Messages Until Loop Ends
    # --------------------
    while True:
        message = ""
        try:
            data = receiver.read_data_from(remoteTransmitter, 3)
            message = data.data.decode("utf8")
            print(message)
        except:
            # print("no message")
            pass
    # --------------------
    # Manual Mode
    # --------------------
        if message == "manual":
            # print("manual mode activated")
            global pathName
            global powerName
            global CurrentTime
            global t1
            lat = []
            lon = []
            power = []
            heading = []
            dt = []
            logPath = False
            pathName = ""
            # --------------------
            # Continue Reading Messages
            # --------------------
            while message != '0':
                try:
                    data = receiver.read_data_from(remoteTransmitter, 0.1)
```

```
                    receiver.flush_queues()
                    message = data.data.decode("utf8")

                except:
                    message = ""
            # --------------------
            # Path and power recording
            # --------------------
                if message == "3":
                    # powerName = "powerDefault.csv"
                    CurrentTime = perf_counter()
                    t1 = perf_counter()

                    try:
                        print("Getting file name . . .")
                        receiver.flush_queues()
                        receiver.send_data_async(
                            remoteTransmitter, "Please supply a file name with a
.csv extension within the next 30s")
                        receiver.flush_queues()
                        data = receiver.read_data_from(remoteTransmitter, 30)
                        pathName = data.data.decode("utf8")
                        powerName = str("power"+str(pathName))
                    except:
                        pathName = "pathDefaultName.csv"
                        powerName = "powerDefaultName.csv"
                    try:
                        methods.logDataInit(pathName)
                        methods.logDataInit(powerName)
                    except Exception as e:
                        receiver.send_data_async(remoteTransmitter, str(e))
                    lat = []
                    lon = []
                    heading = []
                    power = []
                    dt = []
                    # power.append("starting file")
                    logPath = False
                    print("Getting gps . . .")
                    i = 0
                    while i < 5 and logPath == False:
                        try:
                            i += 1
                            [reflat, reflon, satnum] = methods.readGPS()[0:3]
                            aspectRatio = cos(reflat)
```

```
                            [x1, y1] = methods.latlonToXY(
                                reflat, reflon, aspectRatio)
                            robot_heading = methods.deg2rad(
                                float(methods.write_read('C',
methods.sensorArduino)))
                            initial_state = np.array([[x1], [y1], [sin(
                                robot_heading)*methods.velocityMagnitude],
[cos(robot_heading)*methods.velocityMagnitude]])
                            methods.filterInit(initial_state, average_timestep)
                            receiver.send_data_async(
                                remoteTransmitter, "try "+str(i)+",
"+str(reflat))
                            if satnum != 0:
                                logPath = True
                                receiver.send_data_async(
                                    remoteTransmitter, "Success!")
                                print("Recording path")
                        except Exception as e:
                            # receiver.send_data_async(remoteTransmitter, str(e))
                            pass

                if len(lat) > 20:
                    print(str(len(lat)) + " vals in list")
                    try:
                        data = []
                        i = 0
                        while i < len(lat):
                            data.append(
                                str(lat[i])+","+str(lon[i])+","+str(heading[i]) +
"," + str(dt[i]))
                            i += 1
                        methods.logDataUpdate(data, pathName)
                        lat = []
                        lon = []
                        heading = []
                        dt = []
                    except Exception as e:
                        # print(str(e))
                        # receiver.send_data_async(remoteTransmitter, str(e))
                        pass

                if len(power) > 20:
                    # print(str(len(power)) + " vals in list")
                    try:
                        methods.logDataUpdate(power, powerName)
```

```
                    power = []
                except Exception as e:
                    # print(str(e))
                    # receiver.send_data_async(remoteTransmitter, str(e))
                    pass
            # --------------------
            # Stop recording and write to file
            # --------------------
                if message == "4":
                    logPath = False
                    try:
                        data = []
                        i = 0
                        while i < len(lat):
                            data.append(
                                str(lat[i])+","+str(lon[i])+","+str(heading[i]) +
","  + str(dt[i]))
                            i += 1
                        methods.logDataUpdate(data, pathName)
                        lat = []
                        lon = []
                        heading = []
                        dt = []
                    except Exception as e:
                        # print(str(e))
                        # receiver.send_data_async(remoteTransmitter, str(e))
                        pass

                    try:
                        methods.logDataUpdate(power, powerName)
                        power = []
                    except Exception as e:
                        # print(str(e))
                        # receiver.send_data_async(remoteTransmitter, str(e))
                        pass

                if logPath == True:
                    if (perf_counter()-CurrentTime) > powerReadingDelay:
                        CurrentTime = perf_counter()
                        try:
                            power.append(methods.write_read(
                                'P', methods.sensorArduino))
                        except:
                            pass
                    try:
```

```
                        [lattitude, longitude, x, y, angle, sats, time, quality]
= methods.get_filtered_state(
                            aspectRatio, receiver, remoteTransmitter)
                        if sats != 0:
                            lat.append(lattitude[0])
                            lon.append(longitude[0])
                            heading.append(angle)
                            dt.append(perf_counter() - t1)
                            t1 = perf_counter()
                    except Exception as e:
                        # exception_type, exception_object, exception_traceback =
sys.exc_info()
                        # filename =
exception_traceback.tb_frame.f_code.co_filename
                        # line_number = exception_traceback.tb_lineno
                        # receiver.send_data_async(remoteTransmitter, str(e) +",
"+ str(exception_type)+", "+str(filename)+", "+str(line_number))
                        pass
                methods.writeToArduino(
                    'S'+methods.manualControl(message), methods.steeringArduino)

    # --------------------
    # Setting speed
    # --------------------
        elif message == "set speed":
            try:
                receiver.send_data_async(
                    remoteTransmitter, 'Input a speed in microseconds from 1500
to 2000. Current speed is ' + str(methods.Speed))
                receiver.flush_queues()
                data = receiver.read_data_from(remoteTransmitter, 30)
                speed = int(data.data.decode("utf8"))
                methods.setSpeed(speed)
            except Exception as e:
                print("Error with speed setting")
                print(str(e))
                receiver.send_data_async(remoteTransmitter, str(e))
    # --------------------
    # Read GPS
    # --------------------
        elif message == "gps reading":
            try:
                receiver.send_data_async(
                    remoteTransmitter, str(methods.readGPS()))
            except Exception as e:
```

```
                print("Failed to send GPS data")
                print(str(e))
                receiver.send_data_async(remoteTransmitter, str(e))
    # --------------------
    # Read Power
    # --------------------
        elif message == "power reading":
            try:
                receiver.send_data_async(
                    remoteTransmitter, methods.write_read('P',
methods.sensorArduino))
            except Exception as e:
                print("Failed to send power data, " + str(e))
                receiver.send_data_async(remoteTransmitter, str(e))
    # --------------------
    # Read Compass
    # --------------------
        elif message == "compass reading":
            try:
                receiver.send_data_async(
                    remoteTransmitter, methods.write_read('C',
methods.sensorArduino))
            except Exception as e:
                print("Failed to send compass data")
                receiver.send_data_async(remoteTransmitter, str(e))
    # --------------------
    # Test For Connection
    # --------------------
        elif message == "ping":
            receiver.send_data_async(
                remoteTransmitter, "Robot computer online")
    # --------------------
    # Run Autonomously From File
    # --------------------
        elif message == "autonomous":
            filelist = []
            counter = 1
            filestr = "Please type in the name of one of the available coordinate
files which you would like to follow or type CANCEL: \n"
            filelist.append(filestr)
            files = glob.glob('./*.csv')
            for f in files:
                filestr = str(f)+" ["+str(os.path.getsize(f))+" bytes"+"]"
                filelist.append(filestr)
                counter += 1
```

```
            print(filelist)
            i = 0
            receiver.send_data_async(
                remoteTransmitter, str(counter))
            receiver.flush_queues()
            while i < counter:
                receiver.send_data_async(
                    remoteTransmitter, str(filelist[i]))
                i += 1
                print(i)
            try:
                data = receiver.read_data_from(remoteTransmitter, 60)
                userFilename = data.data.decode("utf8")
            except:
                userFilename = 'CANCEL'
            if userFilename != 'CANCEL':
                try:
                    methods.waypointFollower(
                        0.0, 1.0, 0.0, 1, receiver, remoteTransmitter,
userFilename)
                    receiver.send_data_async(
                        remoteTransmitter, "Path following terminated")
                except Exception as e:
                    exception_type, exception_object, exception_traceback =
sys.exc_info()
                    filename = exception_traceback.tb_frame.f_code.co_filename
                    line_number = traceback.extract_tb(exception_traceback)
                    receiver.send_data_async(remoteTransmitter, str(
                        e) + ", " + str(exception_type)+", "+str(filename)+",
"+str(line_number))
                    print(str(e) + ", " + str(exception_type) +
                          ", "+str(filename)+", "+str(line_number))
            print(userFilename)
            message == ""

        elif message == "STOP ROBOT":
            break

    receiver.close()
```

Function File

```
# distance calculator
# https://www.hindawi.com/journals/ape/2014/507142/
```

```
from math import cos, sin, pi, acos, atan
from time import sleep
import serial
import pynmea2
from time import perf_counter
from numpy import genfromtxt, sign, sort
import csv
import serial.tools.list_ports
from CustomKalman import TwoDKalman
import numpy as np
# steeringArduino = serial.Serial(port = 'COM13', baudrate=9600, timeout=5)
logFrequency = 5  # How frequent should data be logged (s)
listSize = 100  # How large list is before logging
robotLength = 1.2192  # m, 4ft
robotWidth = 3.048  # m, 10ft
angleSignal = 0  # Radians
velocitySignal = 0  # Microseconds
mode = 1  # Steering mode
setpoint = 0  # for synchronous steering
Speed = 1600
speedset = -1
XVariance = 5  # noise variance in meters for longitude
YVariance = 5  # noise variance in meters for latitude
XVelocityVariance = 0.05  # noise varience in measured velocity
YVelocityVariance = 0.05  # noise varience in measured velocity
velocityMagnitude = 0.3  # m/s guess


def getSerialPorts():
    """
    This function finds the pancreas' connected devices and stores their serial
port names.
    """
    global steeringPort
    global sensorPort
    global gpsPort
    global radioPort
    radioPort = ""
    sensorPort = ""
    gpsPort = ""
    steeringPort = ""
    try:
        radioPort = list(*serial.tools.list_ports.grep('FT232EX'))[0]
    except:
```

```
        print("Radio not connected.")
    try:
        gpsPort = list(*serial.tools.list_ports.grep('Controller'))[0]
    except:
        print("GPS not connected.")
    try:
        sensorPort = list(*serial.tools.list_ports.grep('Leonardo'))[0]
    except:
        print("Sensor arduino not connected.")
    try:
        steeringPort = list(*serial.tools.list_ports.grep('USB Serial'))[0]
    except:
        print("Steering arduino not connected.")


def initArduinos():
    """
    This function initializes the two arduinos used in the rest of the program.
    """
    global steeringArduino
    global sensorArduino
    val = ""

    if steeringPort != "":
        try:
            steeringArduino = serial.Serial(
                port=steeringPort, baudrate=115200, timeout=0.1)
            val += 'steering connected'
        except Exception as e:
            val += e
    if sensorPort != "":
        try:
            sensorArduino = serial.Serial(
                port=sensorPort, baudrate=115200, timeout=0.1)
            val += ', sensors connected'
        except Exception as e:
            val += e
    return val


def setSpeed(input):
    global Speed
    Speed = input
```

```
def readGPS():
    """
    It reads the GPS Serial port and translates NMEA to usable values.
    """
    try:
        data = ""
        gps = serial.Serial(port=gpsPort, baudrate=115200, timeout=5)
        gps.flushInput()  # flush input buffer, discarding all its contents
        gps.flushOutput()
        sleep(.05)
        data = gps.readline()
        sleep(.05)
        gps.close()
        data = data.decode("utf-8")
        dataParse = pynmea2.parse(data)
        gpsLat = dataParse.latitude
        gpsLon = dataParse.longitude
        sats = dataParse.num_sats
        time = dataParse.timestamp
        qual = dataParse.gps_qual
        return gpsLat, gpsLon, int(sats), time, int(qual)
    except Exception as e:
        return e


def get_filtered_state(aspect_ratio, receiver, remoteTransmitter):
    """
    This function reads the GPS Serial port and translates NMEA to usable values.
    It returns lattitutde and longitude, cartesian coordinates, heading,
satellites connected, time, and gps quality.
    """
    try:
        data = ""
        gps = serial.Serial(port=gpsPort, baudrate=115200, timeout=5)
        gps.flushInput()  # flush input buffer, discarding all its contents
        gps.flushOutput()
        sleep(.05)
        data = gps.readline()
        sleep(.05)
        gps.close()
        data = data.decode("utf-8")
        dataParse = pynmea2.parse(data)
        gpsLat = dataParse.latitude
        gpsLon = dataParse.longitude
        sats = dataParse.num_sats
```

```
        if round(gpsLat) == 0 and round(gpsLon) == 0:
            sats = 0
            gpsLat = gps_lat_old
            gpsLon = gps_lon_old

        time = dataParse.timestamp
        qual = dataParse.gps_qual
        robotAngle = deg2rad(float(write_read('C', sensorArduino)))
        [xraw, yraw] = latlonToXY(gpsLat, gpsLon, aspect_ratio)
        xcenter = xraw+(robotWidth/2.0)*cos(robotAngle)
        ycenter = yraw-(robotWidth/2.0)*sin(robotAngle)
        measured_state = np.array([[xcenter], [ycenter], [sin(
            robotAngle)*velocityMagnitude], [cos(robotAngle)*velocityMagnitude]])
        [xfilter, yfilter] = filteredGPS(measured_state)
        [latAdjusted, lonAdjusted] = XYtolatlon(xfilter, yfilter, aspect_ratio)
        gps_lat_old = latAdjusted
        gps_lon_old = lonAdjusted
        return latAdjusted, lonAdjusted, xfilter, yfilter, robotAngle, int(sats),
time, int(qual)
    except Exception as e:
        receiver.send_data_async(remoteTransmitter, str(e))


def filterInit(initial_state, time_step):
    global gpsFilter
    global gps_lat_old
    global gps_lon_old
    gpsFilter = TwoDKalman(initial_state, time_step, XVariance,
                           YVariance, XVelocityVariance, YVelocityVariance)


def filteredGPS(current_state):
    [x, y] = gpsFilter.filter(current_state)
    return x, y


def deg2rad(deg):
    '''
    converts degrees to radians, pretty simple
    '''
    return deg*(pi/180.0)


def rad2deg(rad):
    '''
    converts radians to degrees, pretty simple
```

```
    '''
    return rad*180.0/pi


def distanceToWaypoint(xp, yp, xw, yw):
    """
    computes the distance to the waypoint
    """
    return ((xp-xw)**2+(yp-yw)**2)**(1/2)


def latlonToXY(lat, lon, aspectRatio):
    """
    This function converts lattitude and longitude to x and y values using simple
eqirectangular projection.
    X and y are in meters
    """
    r = 6371000
    lat = deg2rad(lat)
    lon = deg2rad(lon)
    y = r*lat
    x = r*lon*aspectRatio
    return x, y


def XYtolatlon(x, y, aspectRatio):
    """
    This function converts x,y back to lattitude and longitude using simple
equirectangular projection.
    X and y need to be in meters.
    """
    r = 6371000
    lat = rad2deg(y/r)
    lon = rad2deg(x/(r*aspectRatio))
    return lat, lon


def betweenWaypoints(x1, y1, x2, y2, xp, yp):
    """
    This function is for waypoint following. It decides whether to look at a
point on the line segment or one of the endpoints to follow.
    """
    xval = [x1, x2]
    xval = sort(xval)
    yval = [y1, y2]
    yval = sort(yval)
```

```
    if x1 == x2:
        if yp > yval[1] or yp < yval[0]:
            return False
    elif y1 == y2:
        if xp > xval[1] or xp < xval[0]:
            return False
    else:
        m = (y2-y1)/(x2-x1)
        intercept1 = y1+x1/m
        intercept2 = y2+x2/m
        a = yp+xp/m
        if intercept1 > intercept2:
            if (a-intercept1) > 0 or (a-intercept2) < 0:
                return False
        else:
            if (a-intercept1) < 0 or (a-intercept2) > 0:
                return False
    return True


def calcAngleError(x1, y1, x2, y2, xp, yp, robotAngle, r):
    """
    This is the error calculator for the pure persuit line following algorithm.
    The robot maintains a constant speed and only uses the error in desired and
current angle to navigate.
    """
    pt1Dist = ((xp-x1)**2 + (yp-y1)**2)**(1/2)
    pt2Dist = ((xp-x2)**2 + (yp-y2)**2)**(1/2)
    if x2 == x1:
        if y2 == y1:
            Acoef = 1
            Bcoef = -2*yp
            Ccoef = x2**2-2*xp*x2+xp**2+yp**2-r**2
            perpDist = pt1Dist+1
        else:
            Acoef = 1
            Bcoef = -2*yp
            Ccoef = x2**2-2*xp*x2+xp**2+yp**2-r**2
            perpDist = ((x2-xp)**2)**(1/2)
    else:
        m = (y2-y1)/(x2-x1)
        b = y2-m*x2
        Acoef = (m**2+1)
        Bcoef = 2*m*b-2*m*yp-2*xp
        Ccoef = xp**2+b**2-2*b*yp+yp**2-r**2
```

```
    if betweenWaypoints(x1, y1, x2, y2, xp, yp) == True:
        perpDist = abs(-m*xp+yp-b)/(m**2+1**2)**(1/2)
    else:
        perpDist = pt1Dist+1

distances = [pt1Dist, pt2Dist, perpDist]
minDist = distances.index(min(distances))
descriminant = Bcoef**2-4*Acoef*Ccoef

if distances[minDist] > r or descriminant < 0:
    if minDist == 0:
        x = x1
        y = y1
    elif minDist == 1:
        x = x2
        y = y2
    else:
        if y2 == y1:
            x = xp
            y = y2
        elif x2 == x1:
            x = x2
            y = yp
        else:
            x = (xp/m+yp-b)/(m+1/m)
            y = m*x+b

else:
    xpot1 = (-Bcoef + (descriminant)**(1/2))/(2*Acoef)
    xpot2 = (-Bcoef - (descriminant)**(1/2))/(2*Acoef)

    if x1 == x2:
        ypot1 = xpot1
        ypot2 = xpot2
        xpot1 = x2
        xpot2 = x2
    else:
        ypot1 = m*xpot1+b
        ypot2 = m*xpot2+b

    dpot1 = ((xpot1-x2)**2+(ypot1-y2)**2)**(1/2)
    dpot2 = ((xpot2-x2)**2+(ypot2-y2)**2)**(1/2)
    if (dpot2 > dpot1):
        x = xpot1
        y = ypot1
```

```
        else:
            x = xpot2
            y = ypot2

    xr = xp+r*sin(robotAngle)
    yr = yp+r*cos(robotAngle)
    ax = xr-xp
    ay = yr-yp
    bx = x-xp
    by = y-yp
    angleError = 0
    if (ax != 0 or ay != 0) and (bx != 0 or by != 0):
        try:
            angleError = acos(
                (ax*bx+ay*by)/(((ax**2+ay**2)**(1/2))*((bx**2+by**2)**(1/2))))
        except:
            pass
    cp = ax*by-bx*ay
    if cp < 0:
        angleError = -1*angleError
    return angleError


def writeToArduino(x, microcontroller):
    """
    This function communicates with the latte panda's onboard arduino or USB
connected arduino.
    It takes what is being written and a port name.
    """

    try:
        microcontroller.write(bytes(x, 'utf-8'))
        microcontroller.flushInput()  # flush input buffer, discarding all its
contents
        microcontroller.flushOutput()
    except Exception as e:
        print(str(e))
        pass
    return


def write_read(x, microcontroller):  # communucation btw the cpu and ard
    """
    This function communicates with the latte panda's onboard arduino or USB
connected arduino and receives a return signal.
```

```
    It takes what is being written and a port name.
    """
    try:
        microcontroller.flushInput()  # flush input buffer, discarding all its
contents
        microcontroller.flushOutput()
        microcontroller.write(bytes(x, "utf-8"))
        data = microcontroller.readline()
        data = data.decode("utf-8")
        return data
    except Exception as e:
        print(str(e))
        pass
    return

# This enables manual control from another computer


def manualControl(keystroke):
    """
    This function enables manual control from another radio receiver connected
computer.
    It returns the desired pwm inputs for steering and wheel motion.
    """
    global Speed
    global angleSignal
    global velocitySignal
    global mode
    global setpoint
    global speedset
    adjustedSpeed = Speed-1500

    if keystroke == "l":
        speedset *= -1

    if keystroke == "1":
        setpoint = 0
        mode = 1
    elif keystroke == "2":
        angleSignal = 0
        mode = 2

    if mode == 1:
        if keystroke == "a":
            angleSignal -= .05
```

```
        elif keystroke == "d":
            angleSignal += .05
    elif mode == 2:
        if keystroke == "a":
            setpoint -= 10
        elif keystroke == "d":
            setpoint += 10

    if keystroke == "w":
        velocitySignal = 1500 + adjustedSpeed
    elif keystroke == "s":
        velocitySignal = 1500 - adjustedSpeed

    if keystroke == "" and speedset < 0:
        velocitySignal = 1500
    return str(angleSignal)+","+str(velocitySignal)+","+str(setpoint)


def waypointFollwerVariableInits(ki, kp, kd, R):
    """
    This function initialized the starting values for autonomous waypoint
following.
    """
    global x
    global y
    global xPath
    global yPath
    global robotAngle
    global oldErr
    global oldTime
    global intErr
    global wpNum
    global kI
    global kP
    global kD
    global r
    global latPath
    global lonPath
    global heading
    heading = []
    latPath = []
    lonPath = []
    r = R
    kI = ki
    kP = kp
```

```
    kD = kd
    x = 0
    y = 0
    xPath = []
    yPath = []
    robotAngle = 0
    oldErr = 0
    oldTime = perf_counter()
    intErr = 0
    wpNum = 0


def computePID(err, dutycycle):
    """
    This is the pid control calculator, velocity may need to be adjusted
    """
    velocity = ((dutycycle-1500)/500) * \
        1.388  # m/s, constant derived from wheel diameter and measured rpm
    global oldTime
    global intErr
    global oldErr
    intMax = 2.0
    nowTime = perf_counter()
    deltaT = nowTime-oldTime
    intErr += err*deltaT
    if abs(intErr) > intMax:
        intErr = sign(intErr)*intMax
    derErr = (err-oldErr)/deltaT
    out = kP*err+kI*intErr-kD*derErr
    oldTime = nowTime
    oldErr = err
    return out


def logPath(lat, lon, name):
    """
    This writes a series of position values to a .csv file for later use.
    """

    with open(name, 'w') as f:
        f.write('')
    with open(name, 'a', newline='') as f:
        writer = csv.writer(f)
        i = 0
        while i < len(lat):
```

```
            writer.writerow([lat[i], lon[i]])
            i += 1


def logDataInit(name):
    with open(name, 'w') as f:
        f.write('')


def logDataUpdate(data, name):
    with open(name, 'a', newline='') as f:
        i = 0
        while i < len(data):
            f.write(data[i]+'\n')
            i += 1


def initializeWaypointFollower(name, receiver, remoteTransmitter):
    """
    This is to set the inital error so the derivative control doesn't do funny
things on startup.
    The function is just one pass through the waypoint follower loop without
actually issuing any motor commands.
    """
    global waypoints
    global refLat
    global aspectRatio
    global xwaypoints
    global ywaypoints
    global wpNum
    global oldErr
    global initial_state
    global xp
    global yp
    global xWaypoint
    global yWaypoint
    global xlastWaypoint
    global ylastWaypoint
    global gps_lat_old
    global gps_lon_old
    waypoints = genfromtxt(name, delimiter=',')
    receiver.send_data_async(
        remoteTransmitter, "length of chosen file: " + str(len(waypoints)))
    print(len(waypoints))
```

```
    refLat = deg2rad(waypoints[0, 0])
    aspectRatio = cos(refLat)
    xwaypoints = []
    ywaypoints = []
    i = 0
    while i < len(waypoints):

        [xpoints, ypoints] = latlonToXY(
            waypoints[i, 0], waypoints[i, 1], aspectRatio)
        xwaypoints.append(xpoints)
        ywaypoints.append(ypoints)
        i += 1
    i = 0
    failure = True
    while i < 5 and failure == True:
        try:
            [gps_lat, gps_lon, sats] = readGPS()[0:3]
            receiver.send_data_async(
                remoteTransmitter, "try "+str(i)+", lat:"+str(gps_lat) + ", lon:"
+ str(gps_lon))
            if sats != 0 and round(gps_lon) != 0:
                [xp, yp] = latlonToXY(gps_lat, gps_lon, aspectRatio)
                failure = False
                receiver.send_data_async(remoteTransmitter, "Success")
                gps_lat_old = gps_lat
                gps_lon_old = gps_lon
        except Exception as e:
            receiver.send_data_async(remoteTransmitter, str(e))
            print("No connection to GPS")
            pass
        i += 1

# Set first old error equal to the new one to stop weird inital derivative error
behavior
    robotAngle = deg2rad(float(write_read('C', sensorArduino)))

    initial_state = np.array([[xp], [yp], [sin(
        robotAngle)*velocityMagnitude], [cos(robotAngle)*velocityMagnitude]])
    filterInit(initial_state, 0.512)

    [xWaypoint, yWaypoint] = xwaypoints[wpNum], ywaypoints[wpNum]
    if wpNum == 0:
        [xlastWaypoint, ylastWaypoint] = [xWaypoint, yWaypoint]
    else:
        [xlastWaypoint, ylastWaypoint] = xwaypoints[wpNum-1], ywaypoints[wpNum-1]
```

```
    while distanceToWaypoint(xp, yp, xWaypoint, yWaypoint) < r:
        wpNum += 1
        if wpNum >= len(waypoints):
            break
        [xWaypoint, yWaypoint] = xwaypoints[wpNum], ywaypoints[wpNum]
        print("waypoint "+str(wpNum)+" reached")
    oldErr = calcAngleError(xlastWaypoint, ylastWaypoint,
                            xWaypoint, yWaypoint, xp, yp, robotAngle, r)


def waypointFollower(ki, kp, kd, lookahead, receiver, remoteTransmitter,
filename):
    """
    This is the meat and potatoes of the robot.
    It takes in a set of waypoints and follows them by issuing commands to the
onboard arduino.
    It needs pid control constants, the look ahead distance, a threshold at which
to stop going forward and focus on turning,
    a PWM speed (in microseconds) for the motors, a file name, and a radio
reciever and transmitter object.
    """
    global message
    global wpNum
    global latPath
    global lonPath
    global Speed
    global xlastWaypoint
    global ylastWaypoint
    global xWaypoint
    global yWaypoint
    global initial_state
    global aspectRatio
    global xp
    global yp
    global heading
    logDataInit("traversedPath.csv")
    logDataInit("errorOutput.csv")
    logDataInit("PIDoutput.csv")
    logDataInit("powerConsumption.csv")
    outputlist = []
    errplot = []
    timeplot = []
    pwrplot = []
    trueTime = []
    logTimer = perf_counter()
```

```
    waypointFollwerVariableInits(ki, kp, kd, lookahead)
    try:
        initializeWaypointFollower(filename, receiver, remoteTransmitter)
    except Exception as e:
        receiver.send_data_async(remoteTransmitter, str(e))
        return

    while True:
        # Take GPS measurement and compass measurement
        try:
            [lattitude, longitude, x, y, robotAngle, sats, time, quality] =
get_filtered_state(
                aspectRatio, receiver, remoteTransmitter)
            if sats != 0:
                latPath.append(lattitude[0])
                lonPath.append(longitude[0])
                heading.append(robotAngle)
                xp = x
                yp = y

        except Exception as e:
            receiver.send_data_async(remoteTransmitter, "GPS issue")
            print(e)
            pass

    # Read and select waypoint values
    # If robot within threshold distance increment to next waypoint

        while distanceToWaypoint(xp, yp, xWaypoint, yWaypoint) < r:
            wpNum += 1
            if wpNum >= len(waypoints):  # Check for more waypoints
                writeToArduino("S0,1500,0", steeringArduino)
                break
            [xWaypoint, yWaypoint] = xwaypoints[wpNum], ywaypoints[wpNum]
            if wpNum == 0:
                [xlastWaypoint, ylastWaypoint] = [xWaypoint, yWaypoint]
            else:
                [xlastWaypoint, ylastWaypoint] = xwaypoints[wpNum-1],
ywaypoints[wpNum-1]
            print("waypoint "+str(wpNum)+" reached")
        if wpNum >= len(waypoints):  # Check for more waypoints
            writeToArduino("S0,1500,0", steeringArduino)
            break
        try:
            data = receiver.read_data_from(remoteTransmitter, 0.1)
```

```
            message = data.data.decode("utf8")
            writeToArduino("S0,1500,0", steeringArduino)
            break
        except:
            pass

    # Calculate angle error
        try:
            err = calcAngleError(xlastWaypoint, ylastWaypoint,
                                 xWaypoint, yWaypoint, xp, yp, robotAngle, r)
            errplot.append(err)
            timeplot.append(perf_counter())
            if timeplot[-1] - logTimer > logFrequency:
                pwrplot.append(write_read('P', sensorArduino))
                trueTime.append(time)
                logTimer = timeplot[-1]
        # Run error through PID control for steering
            output = computePID(-err, Speed)
            outputlist.append(output)

        # Output to steering motors
            writeToArduino("S"+str(output)+","+str(Speed) +
                           ","+str(0), steeringArduino)
        except Exception as e:
            receiver.send_data_async(remoteTransmitter, str(e))
            print(e)
            pass

    # writing data to file
        if len(latPath) > listSize:
            data = []
            i = 0
            while i < len(latPath):
                data.append(str(latPath[i])+"," +
                            str(lonPath[i])+","+str(heading[i]))
                i += 1
            try:
                logDataUpdate(data, "traversedPath.csv")
                latPath = []
                lonPath = []
                heading = []
            except Exception as e:
                print(str(e))
                receiver.send_data_async(remoteTransmitter, str(e))
        if len(timeplot) > listSize:
```

```
        data1 = []
        data2 = []
        i = 0
        while i < len(timeplot):
            data1.append(str(timeplot[i])+","+str(errplot[i]))
            data2.append(str(timeplot[i])+","+str(outputlist[i]))
            i += 1
        try:
            logDataUpdate(data1, "errorOutput.csv")
            logDataUpdate(data2, "PIDoutput.csv")
            timeplot = []
            errplot = []
            outputlist = []
        except Exception as e:
            print(str(e))
            receiver.send_data_async(remoteTransmitter, str(e))
    if len(pwrplot) > listSize:
        data = []
        i = 0
        while i < len(pwrplot):
            data.append(str(trueTime[i])+","+str(pwrplot[i]))
            i += 1
        try:
            logDataUpdate(data, "powerConsumption.csv")
            trueTime = []
            pwrplot = []
        except Exception as e:
            print(str(e))
            receiver.send_data_async(remoteTransmitter, str(e))

# Check for obstacles

# Turn parallel to obstacle

# Log robot path

data = []
i = 0
while i < len(latPath):
    data.append(str(latPath[i])+","+str(lonPath[i])+","+str(heading[i]))
    i += 1
try:
    logDataUpdate(data, "traversedPath.csv")
    latPath = []
    lonPath = []
```

```
        heading = []
    except Exception as e:
        print(str(e))
        receiver.send_data_async(remoteTransmitter, str(e))

    data1 = []
    data2 = []
    i = 0
    while i < len(timeplot):
        data1.append(str(timeplot[i])+","+str(errplot[i]))
        data2.append(str(timeplot[i])+","+str(outputlist[i]))
        i += 1
    try:
        logDataUpdate(data1, "errorOutput.csv")
        logDataUpdate(data2, "PIDoutput.csv")
        timeplot = []
        errplot = []
        outputlist = []
    except Exception as e:
        print(str(e))
        receiver.send_data_async(remoteTransmitter, str(e))

    data = []
    i = 0
    while i < len(pwrplot):
        data.append(str(trueTime[i])+","+str(pwrplot[i]))
        i += 1
    try:
        logDataUpdate(data, "powerConsumption.csv")
        trueTime = []
        pwrplot = []
    except Exception as e:
        print(str(e))
        receiver.send_data_async(remoteTransmitter, str(e))
```

User Radio Control

```
from digi.xbee.devices import XBeeDevice, RemoteXBeeDevice, XBee64BitAddress
import pynput
import serial.tools.list_ports

# For windows, comment out the below try, except lines and replace radioPort with
a the name of the COM port that the radio is connected to.
print(list(*serial.tools.list_ports.grep('FT231X')))
```

```
try:
    radioPort = list(*serial.tools.list_ports.grep('FT231X'))[0]
except:
    print("Radio not connected.")
transmitter = XBeeDevice(radioPort, 9600)

remoteReceiver = RemoteXBeeDevice(
    transmitter, XBee64BitAddress.from_hex_string("0013A20040FCB774"))
transmitter.open()
inputName = False


def on_press(key):
    global inputName
    try:
        print(key.char)
        transmitter.send_data_async(remoteReceiver, key.char)
        if key.char == '0':
            print("manual stopped")
            return False
        elif key.char == '3':  # fix this
            print("manual paused, press enter once")
            inputName = True
            return False
        elif key.char == '5':
            try:
                data = transmitter.read_data_from(remoteReceiver, 3)
                message = data.data.decode("utf8")
                print(message)
            except:
                print("No transmission")
    except AttributeError:
        print('special key {0} pressed'.format(
            key))


def beginManual():
    listener = pynput.keyboard.Listener(on_press=on_press)
    listener.start()


if __name__ == "__main__":
    while True:
        x = input()
        message = ""
```

```
        if x == "ping":
            if remoteReceiver.reachable:
                print("Reciever online")
            else:
                print("Reciever offline")
            transmitter.flush_queues()
            transmitter.send_data_async(remoteReceiver, x)
            try:
                data = transmitter.read_data_from(remoteReceiver, 3)
                message = data.data.decode("utf8")
                print(message)
            except:
                print("Robot computer offline")

        elif x == "gps reading":
            transmitter.flush_queues()
            transmitter.send_data_async(remoteReceiver, x)
            try:
                data = transmitter.read_data_from(remoteReceiver, 10)
                message = data.data.decode("utf8")
                print(message)
            except:
                print("No data recieved")

        elif x == "power reading":
            transmitter.flush_queues()
            transmitter.send_data_async(remoteReceiver, x)
            try:
                data = transmitter.read_data_from(remoteReceiver, 3)
                message = data.data.decode("utf8")
                print(
                    "Power from batteries, power from solar, system input
voltage(W,W,V): " + message)
            except:
                print("No data recieved")

        elif x == "compass reading":
            transmitter.flush_queues()
            transmitter.send_data_async(remoteReceiver, x)
            try:
                data = transmitter.read_data_from(remoteReceiver, 3)
                message = data.data.decode("utf8")
                print(message)
            except:
                print("No data recieved")
```

```
        elif x == "set speed":
            transmitter.send_data_async(remoteReceiver, x)
            try:
                data = transmitter.read_data_from(remoteReceiver, 5)
                message = data.data.decode("utf8")
                print("\n"+message)
                speed = input()
                transmitter.send_data_async(remoteReceiver, speed)
            except:
                print("No response from robot")

        elif x == "manual":
            print("---------------------------------\nManual mode activated,
avialable commands are: \n   w,a,s,d - where a and d are for turning, and w and s
are forward and backward, respectively\n   l-lock forward or reverse\n   1 - for
dual Ackerman steering\n   2 - for synchronous steering\n   3 - to record a gps
coordinate path\n   4 - stop recording path and write to file\n   5 - read
transmission\n   0 - stop manual mode\n---------------------------------\n")
            transmitter.send_data_async(remoteReceiver, x)
            beginManual()

        elif inputName == True:  # fix this
            try:
                data = transmitter.read_data_from(remoteReceiver, 3)
                message = data.data.decode("utf8")
                print("\n"+message)
                name = input()
                transmitter.send_data_async(remoteReceiver, name)
            except:
                print("Robot computer offline")
            inputName = False
            print("manual resumed")
            beginManual()

        elif x == "autonomous":
            transmitter.send_data_async(remoteReceiver, x)
            try:
                data = transmitter.read_data_from(remoteReceiver, 5)
                message = data.data.decode("utf8")
                counter = int(message)
                i = 0
                while i < counter:
                    data = transmitter.read_data_from(remoteReceiver, 5)
                    message = data.data.decode("utf8")
```

```
                    print(message)
                    i += 1
                name = input()
                transmitter.send_data_async(remoteReceiver, name)
            except:
                print("No response from robot")
            while input("Press enter to refresh, or 0 then enter to exit.") !=
'0':
                try:
                    transmitter.flush_queues()
                    data = transmitter.read_data_from(remoteReceiver, 3)
                    message = data.data.decode("utf8")
                    print(message)
                except:
                    print("No error message recieved")

        elif x == "STOP ROBOT":
            transmitter.send_data_async(remoteReceiver, x)
        elif x == "STOP CONTROLLER":
            break
        else:
            print("---------------------------------\nUnrecognized command,
avialable commands are: \n   ping\n   set speed\n   gps reading\n   power
reading\n   compass reading\n   manual\n   autonomous\n   STOP ROBOT\n   STOP
CONTROLLER\n---------------------------------\n")

    print("loop ended")
    transmitter.close()
```

Kalman Filter

```
#based on: https://machinelearningspace.com/2d-object-tracking-using-kalman-
filter/

import numpy as np

class TwoDKalman():
    def __init__(self, initial_state, dt, stdX, stdY, stdVX, stdVY):
        self.dt = dt
        self.state = initial_state
        self.F = np.array([[1, 0, self.dt, 0],
                           [0, 1, 0, self.dt],
                           [0, 0, 1, 0],
                           [0, 0, 0, 1]])
```

```
        self.H = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0],
                           [0, 0, 1, 0],
                           [0, 0, 0, 1]])
        self.Q = np.eye(4)
        self.R = np.array([[stdX**2, 0, 0, 0],
                           [0, stdY**2, 0, 0],
                           [0, 0, stdVX**2, 0],
                           [0, 0, 0, stdVY**2]])
        self.P = np.eye(self.F.shape[1])
        self.I = np.eye(self.H.shape[1])
        # pass
    def update(self, measured_state):
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R
        kalman_gain = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))
        self.state = self.state + np.dot(kalman_gain, (measured_state -
np.dot(self.H, self.state)))
        # print(self.state.shape)
        self.P = (self.I - (np.dot(kalman_gain,self.H)))*self.P
        return self.state[0:2]
        # pass
    def predict(self):
        self.state = np.dot(self.F, self.state)
        self.P = np.dot(np.dot(self.F, self.P), self.F.T) + self.Q
        # pass
    def filter(self, measured_state, dt = None):
        self.predict()
        filtered_position = self.update(measured_state)
        return filtered_position
```

Motor Arduino Main

```
//-----------------------
//   DEFINES AND INCLUDES
//-----------------------
// Motor and steering
#define LWHEELMOTOR 2
#define RWHEELMOTOR 3
#define LRWHEELMOTOR 4
#define RRWHEELMOTOR 5
// #define WHEELMOTOR 1
#define LSTEPPIN 9
#define LDIRPIN 10
```

```
#define RSTEPPIN 11
#define RDIRPIN 12
#define LRSTEPPIN 13
#define LRDIRPIN 8
#define RRSTEPPIN 6
#define RRDIRPIN 7
//Power compass and communication
#define DECLINATION -70
#include <Servo.h>
#include <StepperMotorClosedLoop2.h>



//--------------------
//GLOBAL VARIABLES / OBJECTS
//--------------------
// Motor and steering
int steps = 850;
float gearRatio = 4.25;
unsigned char steeringAddress = 0x36;
int setPoint = 0;
float turnAngle;
float roboWidth = 10.0;
float roboLength = 4.0;
float leftAngle;
float rightAngle;
float rightRearAngle;
float leftRearAngle;
Servo LWheel;
Servo RWheel;
Servo LRWheel;
Servo RRWheel;
Servo Wheel;
StepperMotorClosedLoop lMotor(LSTEPPIN, LDIRPIN, 0, steps, gearRatio,
steeringAddress);
StepperMotorClosedLoop rMotor(RSTEPPIN, RDIRPIN, 1, steps, gearRatio,
steeringAddress);
StepperMotorClosedLoop lrMotor(LRSTEPPIN, LRDIRPIN, 2, steps, gearRatio,
steeringAddress);
StepperMotorClosedLoop rrMotor(RRSTEPPIN, RRDIRPIN, 3, steps, gearRatio,
steeringAddress);
String Comm = "";
char command;
String line = "";
String steeringString = "";
```

```
String forwardString = "";
String setpointString = "";
int i;
int j;
float steeringSignal;
int forwardSignal;
int signalStore;

//--------------------
//   SETUP
//--------------------

void setup() {
  Serial.begin(115200); // serial initialization
  delay(10);
  Serial.println("Setting up drive motors . . .");
  Serial.setTimeout(10);
  pinMode(LWHEELMOTOR, OUTPUT);
  pinMode(RWHEELMOTOR, OUTPUT);
  pinMode(LRWHEELMOTOR, OUTPUT);
  pinMode(RRWHEELMOTOR, OUTPUT);
  // pinMode(WHEELMOTOR, OUTPUT);
  // Wheel.attach(WHEELMOTOR);
  // Wheel.writeMicroseconds(1500);
  LWheel.attach(LWHEELMOTOR);
  LWheel.writeMicroseconds(1500);
  RWheel.attach(RWHEELMOTOR);
  RWheel.writeMicroseconds(1500);
  LRWheel.attach(LRWHEELMOTOR);
  LRWheel.writeMicroseconds(1500);
  RRWheel.attach(RRWHEELMOTOR);
  RRWheel.writeMicroseconds(1500);
  Serial.println(F("Connecting steering motors . . ."));
  lMotor.init();
  Serial.println(F("lmotor running"));
  delay(10);
  rMotor.init();
  Serial.println(F("rmotor running"));
  delay(10);
  lrMotor.init();
  Serial.println(F("lrmotor running"));
  delay(10);
  rrMotor.init();
  Serial.println(F("rrmotor running"));
  delay(10);
```

```
}

//--------------------
//   LOOP
//--------------------

void loop() {
  if (Serial.available()) {
    command = "";
    line = Serial.readString();
    Serial.flush();
    command = line[0];
    //    Serial.println(line);
    switch (command) {

      case 'S': //steering
        steeringString = "";
        forwardString = "";
        setpointString = "";
        j = line.length();
        if (j > 2) {
          i = 1;
          while (i < j) {
            if (line[i] == char(',')) {
              break;
            }
            steeringString += line[i];
            i++;
          }
          i++;
          while (i < j) {
            if (line[i] == char(',')) {
              break;
            }
            forwardString += line[i];
            i++;
          }
          i++;
          while (i < j) {
            setpointString += line[i];
            i++;
          }
          steeringSignal = steeringString.toFloat();
          forwardSignal = forwardString.toInt();
```

```
          forwardSignal = forwardSignal - 1500;
          signalStore = forwardSignal;
          setPoint = setpointString.toInt();
          steering(steeringSignal, setPoint);

          forwardSignal = signalStore *
lMotor.calculateStepsReversible(leftAngle);
          LWheel.writeMicroseconds(forwardSignal + 1500);
          forwardSignal = signalStore *
rMotor.calculateStepsReversible(rightAngle);
          RWheel.writeMicroseconds(forwardSignal + 1500);
          forwardSignal = signalStore *
lrMotor.calculateStepsReversible(leftRearAngle);
          LRWheel.writeMicroseconds(forwardSignal + 1500);
          forwardSignal = signalStore *
rrMotor.calculateStepsReversible(rightRearAngle);
          RRWheel.writeMicroseconds(forwardSignal + 1500);
        }
        break;

      default:
        break;
    }
  }
  lMotor.turnToAngle();
//  Serial.println(lMotor.returnAngle());
  rMotor.turnToAngle();
  lrMotor.turnToAngle();
  rrMotor.turnToAngle();

}


//--------------------
//STEERING
//--------------------

void steering(float controlSignal, int setpoint) {
  float Offset = float(setpoint) * PI / 1000.0;
  turnAngle = controlSignal;
  leftAngle = atan(roboLength * sin(turnAngle) / (roboLength * cos(turnAngle) +
roboWidth * sin(turnAngle)));
  rightAngle = atan(roboLength * sin(turnAngle) / (roboLength * cos(turnAngle) -
roboWidth * sin(turnAngle)));
  if (turnAngle > 0 && rightAngle < 0) {
```

```
    rightAngle = (PI) + rightAngle;
  }
  if (turnAngle < 0 && leftAngle > 0) {
    leftAngle = -(PI) + leftAngle;
  }
  leftRearAngle = leftAngle;
  rightRearAngle = rightAngle;
  leftAngle = -leftAngle;
  rightAngle = -rightAngle;
  leftAngle = leftAngle - Offset;
  rightAngle = rightAngle - Offset;
  leftRearAngle = leftRearAngle - Offset;
  rightRearAngle = rightRearAngle - Offset;
}
```

Arduino Closed Loop Stepper Motor Library

```
#include "Arduino.h"
#include "StepperMotorClosedLoop2.h"
#include "Wire.h"
#include "SparkFun_I2C_Mux_Arduino_Library.h"
StepperMotorClosedLoop::StepperMotorClosedLoop(int stepPin, int dirPin, int port,
int steps, float gearRatio, unsigned char address){
    _stepPin = stepPin;
    _dirPin = dirPin;
    pinMode(_stepPin, OUTPUT);
    pinMode(_dirPin, OUTPUT);
    _steps = steps;
    _port = port;
    _currentCount = 0;
    _desiredCount = 0;
    _stepsToGo = 0;
    _stepPin = stepPin;
    _dirPin = dirPin;
    _magnetStatus = 0; //value of the status register (MD, ML, MH)
    _numberofTurns = 0; //number of turns
    _correctedAngle = 0; //tared angle - based on the startup value
    _startAngle = 0; //starting angle
    _totalAngle = 0; //total absolute angular displacement
    _gearRatio = gearRatio;
    _resolution = (_steps/_gearRatio)/4096.0;
    _stepsNoGearbox = _steps/_gearRatio;
    _address = address;
    _pulseDelay = 100;
```

```
}

void StepperMotorClosedLoop::init() {
    Wire.begin(); //start i2C
    Wire.setClock(10000);
    _mux.begin();
    _mux.setPort(_port);
    _checkMagnetPresence(); //check the magnet (blocks until magnet is found)
    _readRawAngle(); //make a reading so the degAngle gets updated
    _startAngle = _degAngle; //update startAngle with degAngle - for taring
    _time = millis();

}

void StepperMotorClosedLoop::calculateSteps(float desiredPos) {
    _mux.setPort(_port);
    _desiredCount = (desiredPos * _steps/2)/PI;
    _desiredCount = _desiredCount % _steps;
    _readRawAngle();
    _correctAngle();
    _checkQuadrant();
    _currentCount = _totalAngle;
    if (abs(_currentCount - _desiredCount) > _steps / 2) {
        if (_currentCount > _desiredCount) {
            _stepsToGo = _steps + _desiredCount;
        } else {
            _stepsToGo = _desiredCount - _steps;
        }
    } else {
        _stepsToGo = _desiredCount;
    }

}




int StepperMotorClosedLoop::calculateStepsReversible(float desiredPos) {
    _mux.setPort(_port);
    int val = 1;
    _desiredCount = (desiredPos * _steps/2)/PI;
    _desiredCount = _desiredCount % _steps;
    _readRawAngle();
    _correctAngle();
```

```
    _checkQuadrant();
    _currentCount = _totalAngle;

    if (_desiredCount > _steps/4) {
      _stepsToGo = _desiredCount-_steps/2;

      val = -1;
    } else if (_desiredCount < -_steps/4) {
      _stepsToGo = _desiredCount+_steps/2;
      val = -1;
    } else {
      _stepsToGo = _desiredCount;

    }

    return val;
}

void StepperMotorClosedLoop::turnToAngle() {

    // if (millis()-_time > 200){
    _mux.setPort(_port);
    _readRawAngle();
    _correctAngle();
    _checkQuadrant();
    // _time = millis();
    // }
    if (_totalAngle > _stepsToGo+_steps/200) {
        digitalWrite(_dirPin, LOW);
        digitalWrite(_stepPin, HIGH);
        delayMicroseconds(_pulseDelay);
        digitalWrite(_stepPin, LOW);
        // delayMicroseconds(_pulseDelay);

    }
    else if (_totalAngle < _stepsToGo-_steps/200) {
        digitalWrite(_dirPin, HIGH);
        digitalWrite(_stepPin, HIGH);
        delayMicroseconds(_pulseDelay);
        digitalWrite(_stepPin, LOW);
        // delayMicroseconds(_pulseDelay);
    }
    // }
}
```

```
void StepperMotorClosedLoop::calibrateZero() {
    _mux.setPort(_port);
    _currentCount = 0;
    _desiredCount = 0;
    _readRawAngle();
    _startAngle = _degAngle;
}

void StepperMotorClosedLoop::_correctAngle() {
  _correctedAngle = _degAngle;

}

void StepperMotorClosedLoop::_checkQuadrant() {
    /*
  //Quadrants:
  4  |  1
  ---|---
  3  |  2
  */

  //Quadrant 1
  if(_correctedAngle >= 0 && _correctedAngle <= _stepsNoGearbox/4.0)
  {
    _quadrantNumber = 1;
  }

  //Quadrant 2
  if(_correctedAngle > _stepsNoGearbox/4.0 && _correctedAngle <=
_stepsNoGearbox/2.0)
  {
    _quadrantNumber = 2;
  }

  //Quadrant 3
  if(_correctedAngle > _stepsNoGearbox/2.0 && _correctedAngle <=
3.0*_stepsNoGearbox/4.0)
  {
    _quadrantNumber = 3;
  }

  //Quadrant 4
  if(_correctedAngle > 3.0*_stepsNoGearbox/4.0 && _correctedAngle <
_stepsNoGearbox)
  {
```

```
    _quadrantNumber = 4;
  }
  //Serial.print("Quadrant: ");
  //Serial.println(quadrantNumber); //print our position "quadrant-wise"

  if(_quadrantNumber != _previousquadrantNumber) //if we changed quadrant
  {
    if(_quadrantNumber == 1 && _previousquadrantNumber == 4)
    {
      _numberofTurns++; // 4 --> 1 transition: CW rotation
    }

    if(_quadrantNumber == 4 && _previousquadrantNumber == 1)
    {
      _numberofTurns--; // 1 --> 4 transition: CCW rotation
    }
    //this could be done between every quadrants so one can count every 1/4th of
transition

    _previousquadrantNumber = _quadrantNumber;  //update to the current quadrant

  }

  _totalAngle = (_numberofTurns*_stepsNoGearbox + _correctedAngle); //number of
turns (+/-) plus the actual angle within the 0-360 range
}

void StepperMotorClosedLoop::_checkMagnetPresence() {
      //This function runs in the setup() and it locks the MCU until the magnet
is not positioned properly

  while((_magnetStatus & 32) != 32) //while the magnet is not adjusted to the
proper distance - 32: MD = 1
  {
    _magnetStatus = 0; //reset reading

    Wire.beginTransmission(_address); //connect to the sensor
    Wire.write(0x0B); //figure 21 - register map: Status: MD ML MH
    Wire.endTransmission(); //end transmission
    Wire.requestFrom(_address, 1); //request from the sensor

    while(Wire.available() == 0); //wait until it becomes available
    _magnetStatus = Wire.read(); //Reading the data after the request

    //Serial.print("Magnet status: ");
```

```
    //Serial.println(magnetStatus, BIN); //print it in binary so you can compare
it to the table (fig 21)
  }

  //Status register output: 0 0 MD ML MH 0 0 0
  //MH: Too strong magnet - 100111 - DEC: 39
  //ML: Too weak magnet - 10111 - DEC: 23
  //MD: OK magnet - 110111 - DEC: 55

  //Serial.println("Magnet found!");
  //delay(1000);
}

void StepperMotorClosedLoop::_readRawAngle() {
      //7:0 - bits
  Wire.beginTransmission(_address); //connect to the sensor
  Wire.write(0x0D); //figure 21 - register map: Raw angle (7:0)
  Wire.endTransmission(); //end transmission
  Wire.requestFrom(_address, 1); //request from the sensor

  while(Wire.available() == 0); //wait until it becomes available
  _lowbyte = Wire.read(); //Reading the data after the request

  //11:8 - 4 bits
  Wire.beginTransmission(_address);
  Wire.write(0x0C); //figure 21 - register map: Raw angle (11:8)
  Wire.endTransmission();
  Wire.requestFrom(_address, 1);

  while(Wire.available() == 0);
  _highbyte = Wire.read();

  //4 bits have to be shifted to its proper place as we want to build a 12-bit
number
  _highbyte = _highbyte << 8; //shifting to left
  //What is happening here is the following: The variable is being shifted by 8
bits to the left:
  //Initial value: 00000000|00001111 (word = 16 bits or 2 bytes)
  //Left shifting by eight bits: 00001111|00000000 so, the high byte is filled in

  //Finally, we combine (bitwise OR) the two numbers:
  //High: 00001111|00000000
  //Low:  00000000|00001111
  //      -----------------
  //H|L:  00001111|00001111
```

```
  _rawAngle = _highbyte | _lowbyte; //int is 16 bits (as well as the word)

  //We need to calculate the angle:
  //12 bit -> 4096 different levels: 360° is divided into 4096 equal parts:
  //360/4096 = 0.087890625
  //Multiply the output of the encoder with 0.087890625
  _degAngle = _rawAngle * _resolution;

  //Serial.print("Deg angle: ");
  //Serial.println(degAngle, 2); //absolute position of the encoder within the 0-
360 circle
}

int StepperMotorClosedLoop::returnAngle() {
  return _totalAngle;
}

int StepperMotorClosedLoop::returnCommand() {
  return _stepsToGo;
}
```

Sensor Arduino

```
#include <Compass1.h>
#include <PowerSensors.h>
#define DECLINATION -70
unsigned char compassAddress = 0x30;
PowerSensors sensors;
Compass1 compass(compassAddress);
const int currentSensorPin = A2;    // define sensor pin
const int voltageSensorPin = A1;
const int currentSensorPin2 = A0; // define sensor pin
String Comm = "";
char command;
String line = "";

void setup() {
  Serial.begin(115200); // serial initialization
  delay(10);
  compass.CMPS2_init();                 // initialize the compass
  delay(10);
  sensors.init();
  delay(100);
  Serial.println("1");
  Serial.setTimeout(2);
```

```
}

void loop() {
  if (Serial.available()) {
    line = Serial.readString();
    command = line[0];
    switch (command) {

      case 'C': //compass
        Comm = String(compass.CMPS2_getHeading());
        Serial.println(Comm);
//        delay(100);
        break;

      case 'P': //power
        sensors.CurrentValue = sensors.readDCCurrent(currentSensorPin);
        sensors.CurrentValue2 = sensors.readDCCurrent(currentSensorPin2);
        sensors.VoltValue = analogRead(voltageSensorPin);
        sensors.Voltage = (sensors.VoltValue - 512) * 0.073170;
        sensors.Power = sensors.Voltage * sensors.CurrentValue;   // Power from
the batteries
        sensors.Power2 = 36 * sensors.CurrentValue2; // Power from solar
        sensors.PowerNet = sensors.Power2 - sensors.Power;   // Solar Power minus
Power used from Batteries
        Comm = String(sensors.Power) + "," + String(sensors.Power2) + "," +
String(sensors.Voltage);
        Serial.println(Comm);
        break;

      default:
        break;

      command = "";
    }
  }
}
```

Appendix B - Datasheets, Part Specifications and Features

Accu-Coder Encoder from Encoder Products

Part #: 15s-19m3-0500n5qpp-f00

MODEL 15S SPECIFICATIONS

Electrical

Input Voltage	5 VDC +10% Fixed Voltage 4.75 to 28 VDC max for temperatures up to 85° C 4.75 to 24 VDC for temperatures between 85° to 100° C
Input Current	140 mA max (65 mA typical for most configurations) with no output load
Output Format	Incremental – Two square waves in quadrature with channel A leading B for clockwise shaft rotation, as viewed from the encoder mounting face. See Waveform Diagrams.
Output Types	Open Collector – 20 mA max per channel Push-Pull – 20 mA max per channel Pull-Up – Open Collector with 2.2K ohm internal resistor, 20 mA max per channel Line Driver – 20 mA max per channel (Meets RS 422 at 5 VDC supply.)
Index	Once per revolution. 1 to 400 CPR: Ungated 401 to 10,000 CPR: Gated to output A See Waveform Diagrams.
Max. Frequency	Standard Frequency Response is 200 kHz for CPR 1 to 2540 500 kHz for CPR 2541 to 5000 1 MHz for CPR 5001 to 10,000 Extended Frequency Response (optional) is 300 kHz for CPR 2000, 2048, 2500, and 2540.
Electrical Protection	Reverse voltage and output short circuit protected. NOTE: Sustained reverse voltage may result in permanent damage.
Noise Immunity	Tested to BS EN61000-6-2; BS EN50081-2; BS EN61000-4-2; BS EN61000-4-3; BS EN61000-4-6; BS EN500811
Quadrature	67.5° electrical or better is typical,
Edge Separation	54° electrical minimum at temperatures > 99° C
Waveform Symmetry	180° (±18°) electrical (single channel encoder)
Accuracy	Within 0.017° mechanical or 1 arc-minute from true position (for CPR >189).
Commutation	Up to 12 pole. Contact Customer Service for availability.
Comm. Accuracy	1° mechanical

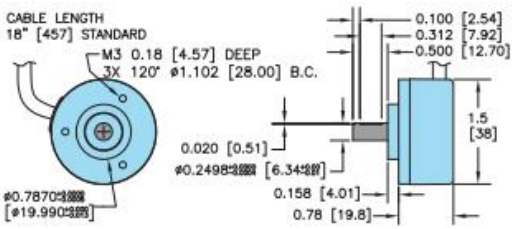
Mechanical

Max Shaft Speed	8000 RPM. Higher speeds may be achievable, contact Customer Service.
Shaft Material	Stainless Steel
Radial Shaft Load	5 lb max. Rated load of 2 to 3 lb for bearing life of 1.2 x 1010 revolutions
Axial Shaft Load	5 lb max. Rated load of 2 to 3 lb for bearing life of 1.2×10^{10} revolutions
Starting Torque	IP50- 0.05 oz-in IP64- 0.4 oz-in
Moment of Inertia	6.7×10^{-5} oz-in-sec^2 (4.8 gm-cm^2)
Weight	3 oz typical

Environmental

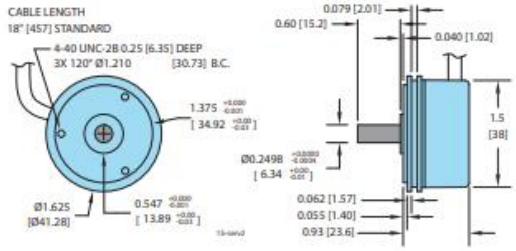
Storage Temp	-25° to 85° C
Humidity	98% RH non-condensing
Vibration	10 g @ 58 to 500 Hz
Shock	80 g @ 11 ms duration
Sealing	IP50 standard; IP64 available

MODEL 15S STANDARD SERVO MOUNT M1

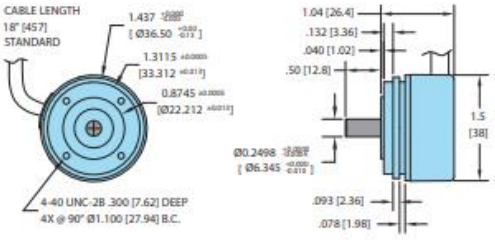


MODEL 15S SERVO MOUNT M2 & M9*

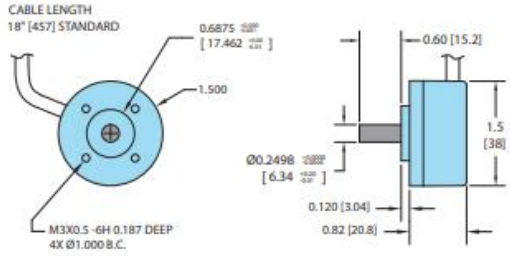
**M9 mount includes a 0.750" boss*



MODEL 15S SERVO MOUNT M5

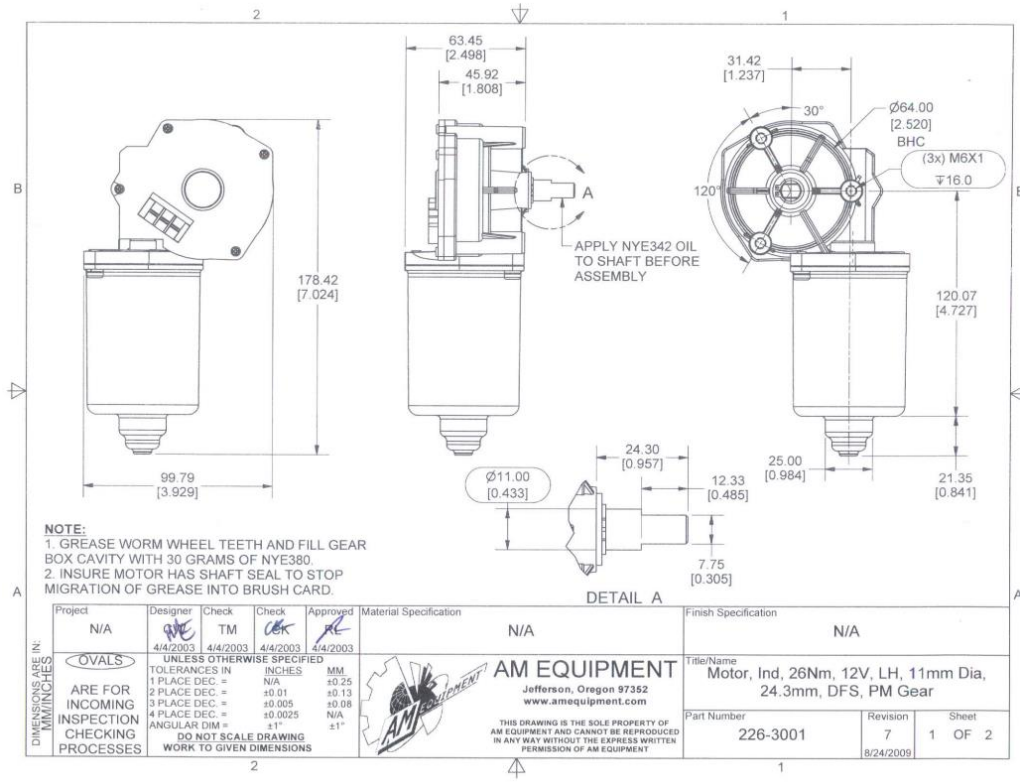


MODEL 15S SERVO MOUNT M6

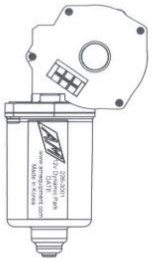


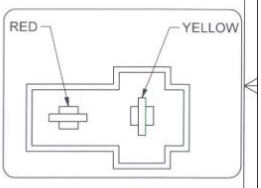
All dimensions are in inches with a tolerance of +0.005" or +0.01" unless otherwise specified. Metric dimensions are given in brackets [mm].

AM Equipment 226 Worm Gear Motor



CLOCKWISE MOTOR SHAFT ROTATION		
DATA POINT	DATA TYPE	VALUE RANGE
NO LOAD	CURRENT (A)	4.5 or <
	SPEED (rpm)	95.9 - 78.5
19.5Nm	CURRENT (A)	39.9 - 34.6
	SPEED (rpm)	39.7 - 32.9
PEAK POWER	POWER (W)	83.6 - 68.4
	TORQUE (Nm)	19.2 - 15.7
NOMINAL (PEAK EFFICIENCY)	POWER (W)	44.5 NOMINAL
	SPEED (rpm)	70.2 NOMINAL
	CURRENT (A)	12.8 NOMINAL
	TORQUE (Nm)	6.2 NOMINAL
COUNTER-CLOCKWISE MOTOR SHAFT ROTATION		
DATA POINT	DATA TYPE	VALUE RANGE
NO LOAD	CURRENT (A)	4.5 or <
	SPEED (rpm)	98.1 - 80.3
19.5Nm	CURRENT (A)	40.9 - 34.8
	SPEED (rpm)	39.4 - 32.0
PEAK POWER	POWER (W)	84.4 - 69.1
	TORQUE (Nm)	18.0 - 14.8
NOMINAL (PEAK EFFICIENCY)	POWER (W)	44.1 NOMINAL
	SPEED (rpm)	74.8 NOMINAL
	CURRENT (A)	11.0 NOMINAL
	TORQUE (Nm)	5.8 NOMINAL
BACKDRIVE 60Nm MINIMUM CW & CCW		





TERMINAL HOUSING 317-1057 (KET# MG 620042)
TERMINAL 317-1054 (KET# ST 740254-1)
MATE HOUSING 317-1056 (KET# MG 610043)
TERMINAL 317-1055 (KET# ST 730268-1)

RED (+), YELLOW (-) = CW
YELLOW (+), RED (-) = CCW *PREFERRED ROTATION*

Project N/A
Designer ME 4/4/2003
Check TM 4/4/2003
Check CK 4/4/2003
Approved RL 4/4/2003
Material Specification N/A
Finish Specification N/A
DIMENSIONS ARE IN: MM/INCHES
OVALS ARE FOR INCOMING INSPECTION CHECKING PROCESSES
UNLESS OTHERWISE SPECIFIED
TOLERANCES IN INCHES MM
1 PLACE DEC. = N/A ±0.25
2 PLACE DEC. = ±0.01 ±0.13
3 PLACE DEC. = ±0.005 ±0.08
4 PLACE DEC. = ±0.0025 N/A
ANGULAR DIM = ±1° ±1°
DO NOT SCALE DRAWING
WORK TO GIVEN DIMENSIONS
AM EQUIPMENT
Jefferson, Oregon 97352
www.amequipment.com
THIS DRAWING IS THE SOLE PROPERTY OF AM EQUIPMENT AND CANNOT BE REPRODUCED IN ANY WAY WITHOUT THE EXPRESS WRITTEN PERMISSION OF AM EQUIPMENT
Title/Name Motor, Ind, 26Nm, 12V, LH, 11mm Dia, 24.3mm, DFS, PM Gear
Part Number 226-3001
Revision 7 8/24/2009
Sheet 2 OF 2

AS 5600 Magnetic Encoder

Features

- Contactless angle measurement
- Simple user-programmable start and stop positions over the I²C interface
- Maximum angle programmable from 18° up to 360°
- 12-bit DAC output resolution
- Analog output ratiometric to VDD or PWM-encoded digital output
- Automatic entry into low power mode

Product parameters

Resolution [bit]	12
Interface	I²C
Output	Analog out / PWM / I²C
Max. Speed [rpm]	
Overvoltage Protection	No
Redundant	No
Supply Voltage [V]	3-3.6 and 4.5-5.5
Temperature Range [°C]	-40 to +125
Package	SOIC-8
Automotive Qualified	
Longevity Program	January 2031

Dakota Lithium 24V 50Ah Battery

SPECIFICATIONS

11 YEAR WARRANTY
World beating, best in class, eleven year manufacturer defect warranty.

STORAGE CAPACITY
52.5 ampere hours (Ah). Dakota Lithium batteries provide consistent power for all 52.5 amp hours. DL LiFePO4 batteries have a flat voltage curve, which means they have a steady power output as the battery discharges. The power output will not dramatically drop like similar sized SLA batteries. You get all the juice down to the last drop.

VOLTAGE
24V rated (26.6V resting voltage) Dakota Lithium 24V batteries can be used in series for up to 48V systems.

ENERGY
1394 Watts (Wh)

TERMINALS
M8 bolt terminals F12 (posts that screw in). Easy to adapt to different connection needs. (Max torque 15 ft. lbs.)

SIZE
12.99 in (330mm) L x 6.77 in (172mm) W x 8.66 in (220mm) H

WEIGHT
31.9 lbs (14.5Kg). That's 60% lighter than a SLA battery.

LIFECYCLES (BATTERY LIFESPAN)
Up to 80% capacity for 3,000 cycles in recommended conditions. The typical SLA has 500 cycles. Dakota Lithium batteries last so long that the price per use is a fraction of traditional batteries.

OPERATING TEMPERATURE
Ideal for rugged & harsh environments. Much better than SLA or other lithium's. -20'F min, +120'F max optimal operating temps (battery performs well down to -20'F). Avoid charging below 32'F.

DISCHARGE
52.5 A max continuous, 100 A max pulse 10 second pulse. The flat discharge voltage curve provides a 75% bigger capacity than a comparable 50Ah SLA battery.

CHARGE
25 A (0.5C) max, 30 V max. Included is a LiFePO4 compatible charger.

INCLUDES ACTIVE BMS PROTECTION
Contains a circuit that handles cell balancing, low voltage cutoff, high voltage cutoff, short circuit protection and high temperature protection for increased performance and longer life.

CERTIFICATIONS
All batteries are UN 38 certified. Dakota Lithium's cells are UL1642 certified and have been tested per IEC62133 standards. Meets all US & International regulations for air, ground, train, & marine transport.

ISO 9001:2015

CHARGER INCLUDED
Single 24V 5A LiFePO4 charger included.

Geophex GEM-2

GEM-2 Specifications

Operating Mode:	Frequency Domain
Number of frequencies:	Standard: programmable up to 10 simultaneous frequencies
Bandwidth:	25 Hz to 96 kHz
Sampling rate:	30 Hz or 25 Hz
Ski dimensions:	185 cm. x 12.5 cm.
Coil configuration:	coplanar
Maximum TX moment:	3 Amp-meters^2 at 330 Hz
Rechargeable battery:	12 VDC
Outputs:	Inphase and quadrature response in ppm Powerline amplitude
Communications:	BlueTooth or RS232 for computer Powered serial for GPS
Standard Gem-2 Includes:	Ski with electronics console Data logger, set up and tested carrying strap extra battery battery charger calibration ferrite soft case calibration & setup software CD quickstart guide
Optional extras:	Extra battery Extra battery charger Extra calibration ferrite Modified Garmin GPS for powered operation from the GEM2 Upgrade to aluminum hard shell carry case One-day training at Geophex location

HQST 100W Solar Panel

HQST – 100DB

100W Mono Bendable Solar Panel

Key Features

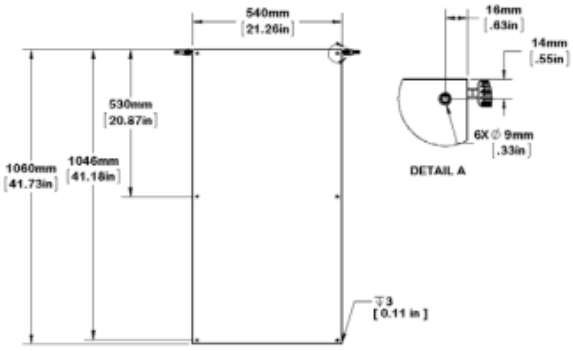
- Top Ranked PTC Rating
- High Module Conversion Efficiency
- Fast and Inexpensive Mounting
- Maximizes System output by reducing mismatch loss
- 100% EL testing on all HQST Solar Modules, Guaranteed no Hot Spots

Electrical Characteristics

Maximum Power at STC (Pmax)	100 W
Optimum Operating Voltage (Vmp)	17.7 V
Optimum Operating Current (Imp)	5.70 A
Open Circuit Voltage (Voc)	21.7 V
Short Circuit Current (Isc)	6.10 A
Maximum System Voltage	600 VDC
Maximum Series Fuse Rating	10 A

STC: Irradiance 1000W/m^2, Temperature 25°C, AM =1

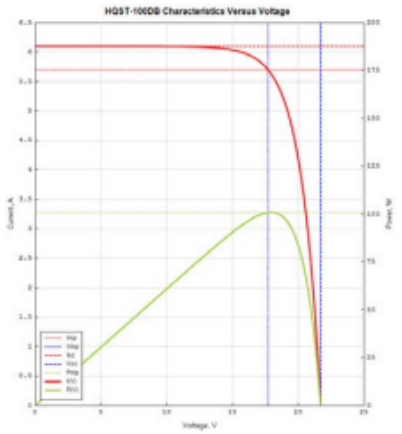
Module Diagram



Mechanical Properties

Solar Cell	Monocrystalline (125 x 125 mm)
# of Cells	32 (8 x 4 mm)
Dimensions	1060 x 540 x 3 mm (41.7 x 21.3 x 0.12 in)
Weight	2 kg. (4.4 lbs.)
Junction Box	IP65 Rated
Output Cables	14 AWG
Connectors	MC4 Connectors
Fire Rating	Class C

IV-Curve



Temperature Characteristics

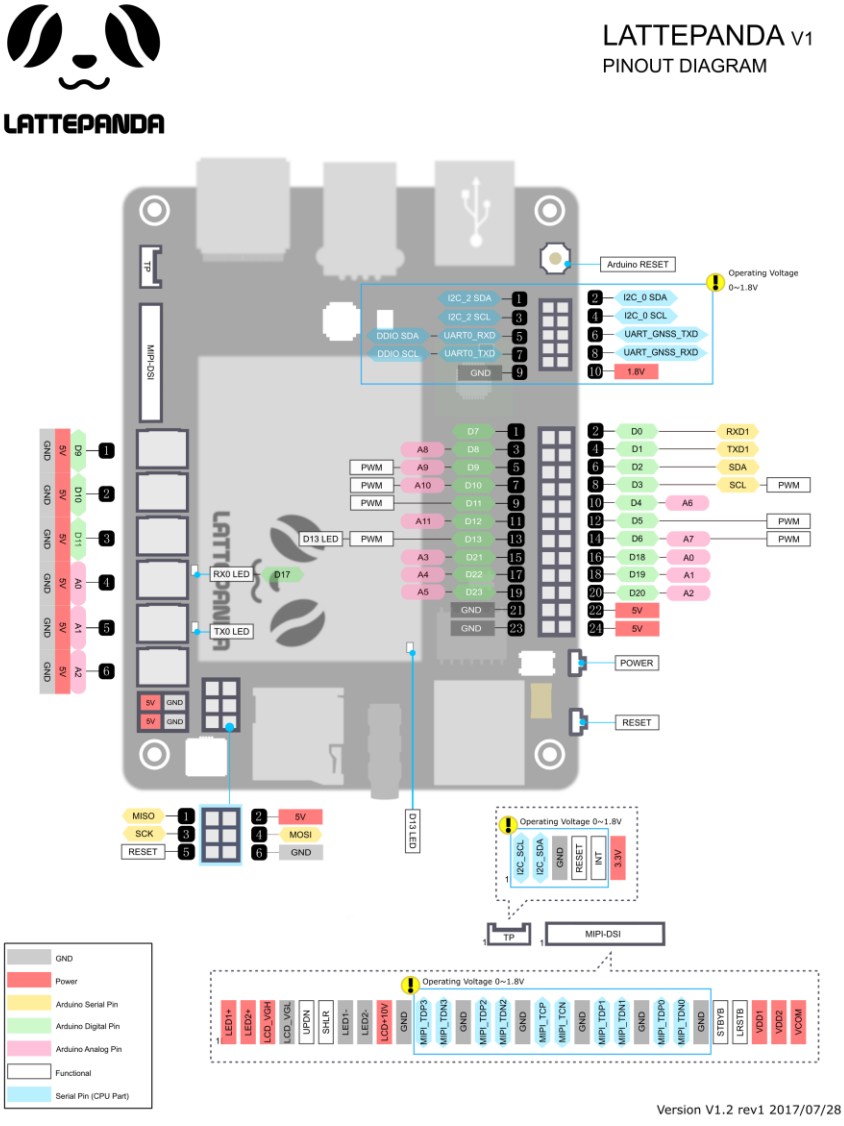
Operating Module Temperature	-40°C to +80°C
Nominal Operating Cell Temperature (NOCT)	47±2°C
Temperature Coefficient of Pmax	-0.38%/°C
Temperature Coefficient of Voc	-0.28%/°C
Temperature Coefficient of Isc	0.06%/°C

HQST Solar | www.hqsolarpower.com | sales@myhqsolar.com

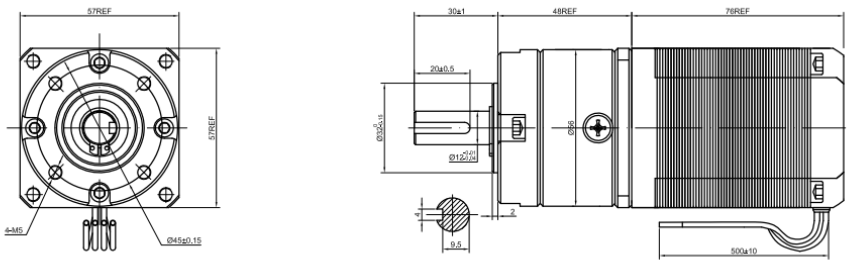
Update: January 2016

Latte Panda Alpha

Intel® Core™ M3-8100Y, Dual-Core, 1.1-3.4GHz
Intel® UHD Graphics 615
8GB Memory
Dual-Band 2.5GHz/5GHz Wi-Fi & Bluetooth 4.2 & Gigabit Ethernet
USB3.0 x3, **USB Type-C x1**
2 x M.2 PCIe (Support B&M Key and A&E Key)
Support Windows 10 & & Linux OS
Integrated Arduino Coprocessor ATMEL 32U4
Powered by PD adapter / 12V DC / 7.4V battery



Nema 23 Stepper Motor

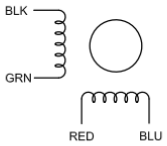


SPECIFICATION / CONNECTION	BIPOLAR
AMPS/PHASE	2.80
RESISTANCE/PHASE(Ohms)@25°C	1.13±10%
INDUCTANCE/PHASE(mH)@1KHz	5.40±20%
HOLDING TORQUE w/o GEARBOX(Nm)[lb-in]	1.89[16.73]
GEAR RATIO	4
EFFICIENCY	90.00%
STEP ANGLE w/o GEARBOX(°)	1.80
BACKLASH@NO-LOAD	<=1.5°
MAX.PERMISSIBLE TORQUE(Nm)	20.00
MOMENT PERMISSIBLE TORQUE(Nm)	30.00
SHAFT MAXIMUM AXIAL LOAD(N)	100.00
SHAFT MAXIMUM RADIAL LOAD(N)	200.00
WEIGHT(Kg)[lb]	——
TEMPERATURE RISE:MAX.80°C（MOTOR STANDSTILL;FOR 2PHASE ENERGIZED）	
AMBIENT TEMPERATURE -10°C~50°C[14°F~122°F]	
INSULATION CLASS B 130°C[266°F]	

TYPE OF CONNECTION (EXTERN)		MOTOR	
PIN NO	BIPOLAR	LEADS	WINDING
1	A —	BLK	A
2	A\ —	GRN	A\
3	B —	RED	B
4	B\ —	BLU	B\

FULL STEP 2 PHASE-Ex. ,
WHEN FACING MOUNTING END (X)

STEP	A	B	A\	B\	
1	+	+	-	-	CCW ↑
2	-	+	+	-	↓
3	-	-	+	+	
4	+	-	-	+	CW



		APVD		3.12.2020
		CHKD		
	1:1.5	DRN		
	SCALE	SIGNATURE		DATE

STEPPERONLINE®

STEPPER MOTOR

23HS30-2804S-PG4

PMod CMPS-2

±16 Gauss, Ultra Small, Low Noise 3-axis Magnetic Sensor

MMC3416xPJ

FEATURES

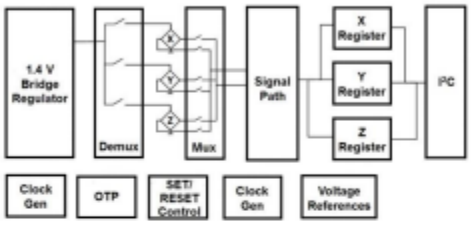
- **Fully integrated 3-axis magnetic sensor and electronic circuits requiring fewer external components**
- **Superior Dynamic Range and Accuracy:**
  - ✓ **±16 G FSR with 16/14 bits operation**
  - ✓ **0.5 mG/2 mG per LSB resolution in 16/14 bits operation mode**
  - ✓ **1.5 mG total RMS noise**
  - ✓ **Enables heading accuracy of ±1°**
- **Max output data rate of 800 Hz (12 bits mode)**
- **Ultra Small Low profile package 1.6x1.6x0.6 mm**
- **SET/RESET function**
  - ✓ **Allows for elimination of temperature variation induced offset error (Null field output)**
  - ✓ **Clears the sensors of residual magnetization resulting from strong external fields**
- **On-chip sensitivity compensation**
- **Low power consumption (140 µA @ 7 Hz)**
- **1 µA (max) power down function**
- **I^2C Slave, FAST (≤400 KHz) mode**
- **1.62 V~3.6 V wide power supply operation supported, 1.8 V I/O compatibility.**
- **RoHS compliant**

APPLICATIONS

- **Electronic Compass & GPS Navigation**
- **Position Sensing**

DESCRIPTION

The MMC3416xPJ is a complete 3-axis magnetic sensor with on-chip signal processing and integrated I^2C bus. The device can be connected directly to a microprocessor, eliminating the need for A/D converters or timing resources. It can measure magnetic fields within the full scale range of ±16 Gauss (G), with 0.5 mG/2 mG per LSB resolution for 16/14 bits operation mode and 1.5 mG total RMS noise level, enabling heading accuracy of 1° in electronic compass applications. Contact MEMSIC for access to advanced calibration and tilt-compensation algorithms.



FUNCTIONAL BLOCK DIAGRAM

An integrated SET/RESET function provides for the elimination of error due to Null Field output change with temperature. In addition it clears the sensors of any residual magnetic polarization resulting from exposure to strong external magnets. The SET/RESET function can be performed for each measurement or periodically as the specific application requires.

The MMC3416xPJ is packaged in an ultra small low profile BGA package (1.6 x 1.6 x 0.6 mm,) and with an operating temperature range from -40 °C to +85 °C.

The MMC3416xPJ provides an I^2C digital output with 400 KHz, fast mode operation.

©MEMSIC, Inc.
One Technology Drive, Suite 325, Andover, MA01810, USA
Tel: +1 978 738 0900 Fax: +1 978 738 0196
www.memsic.com

SPECIFICATIONS (Measurements @ 25 °C, unless otherwise noted; V_{DA} = V_{DD}= 1.8 V unless otherwise specified)

Parameter	Conditions	Min	Typ	Max	Units
Field Range (Each Axis)	Total applied field		±16		G
Supply Voltage	V_{DA}	1.62[1]	1.8	3.6	V
	V_{DD} (I^2C interface)	1.62[1]	1.8	3.6	V
Supply Voltage Rise Time				5.0	mS
Supply Current[2] (7measurements/second)	BW[1:0]=00, 16 bits mode		140		µA
	BW[1:0]=01, 16 bits mode		70		µA
	BW[1:0]=10, 14 bits mode		35		µA
	BW[1:0]=11, 12 bits mode		18		µA
Power Down Current				1.0	µA
Operating Temperature		-40		85	°C
Storage Temperature		-55		125	°C
Linearity Error (Best fit straight line)	FS=±16 G $H_{applied}$=±10 G		0.25		%FS
Hysteresis	3 sweeps across ±16 G		0.1		%FS
Repeatability Error	3 sweeps across ±16 G		0.1		%FS
Alignment Error			±1.0	±3.0	degrees
Transverse Sensitivity			±2.0	±5.0	%
Total RMS Noise	BW[1:0]=00, 16 bits mode		1.5		mG
	BW[1:0]=01, 16 bits mode		2.0		mG
	BW[1:0]=10, 14 bits mode		4.0		mG
	BW[1:0]=11, 12 bits mode		6.0		mG
Output resolution			16/14/12		bits
Max Output data rate	BW[1:0]=00, 16 bits mode		125		Hz
	BW[1:0]=01, 16 bits mode		250		Hz
	BW[1:0]=10, 14 bits mode		450		Hz
	BW[1:0]=11, 12 bits mode		800		Hz
Heading accuracy[3]			±1.0		degrees
Sensitivity	±16 G	-10		+10	%
	16 bits mode		2048		counts/G
	14 bits mode		512		counts/G
	12 bits mode		128		counts/G
Sensitivity Change Over Temperature	-40~85°C Delta from 25°C ±16 G		±3		%
Null Field Output			±0.1		G
	16 bits mode		32768		counts
	14 bits mode		8192		counts
	12 bits mode		2048		counts
Null Field Output Change Over Temperature using SET/RESET	-40~85 °C Delta from 25 °C		±5		mG
Disturbing Field[4]			25		G
Maximum Exposed Field				10000	G
SET/RESET Repeatability[5]			3		mG

[1] 1.62 V is the minimum operation voltage, or V_{DA} / V_{DD} should not be lower than 1.62 V.

[2] Supply current is proportional to how many measurements performed per second, for example, at one measurement per second, the power consumption will be 140 uA/7=20 µA.

[3] MEMSIC product enables users to utilize heading accuracy to be 1.0 degree typical when using MEMSIC's proprietary software or algorithm

[4] This is the magnitude of external field that can be tolerated without changing the sensor characteristics. If the disturbing field is exceeded, a SET/RESET operation is required to restore proper sensor operation.

[5] Perform SET/RESET alternately. SET repeatability is defined as the difference in measurement between multiple SET events. RESET repeatability is defined similarly.

Roboclaw Solo 60A

Electrical Specifications

Characteristic	Model	Min	Typ	Max	Rating
Main Battery	All	6		34	VDC
Logic Battery	All	6	12	34	VDC
Maximum External Current Draw (BEC)	All			1	A
Motor Current Per Channel	Solo 30A		30[2]	60[1,2]	A
	Solo 60A		60[2]	100[1,2]	
On Resistance	Solo 30A		4.3		mOhm
	Solo 60A		1.9		
Logic Circuit Current Draw	All		30mA		mA
Input Impedance	All		100		Ω
Input	All	0		5	VDC
Input Low	All	-0.3		0.8	VDC
Input High	All	2		5	VDC
I/O Output Voltage	All	0		3.3	VDC
Digital and Analog Input Voltage	All			5	VDC
Analog Useful Range	All	0		2	VDC
Analog Resolution	All		1		mV
Pulse Width	All	1		2	mS
Encoder Counters	All		32		Bits
Encoder Frequency	All			9,800,000	PPS
RS232 Baud Rate (Note 3)	All			460,800	Bits/s
RS232 Time Out (Note 3)	All	10			ms
Temperature Range	All	-40	40	100	°C
Temperature Protection Range	All	85		100	°C
Humidity Range	All			100 (4)	%

Notes:

1. Peak current is automatically reduced to the typical current limit as temperature approaches 85°C.
2. Current is limited by maximum temperature. Starting at 85°C, the current limit is reduced on a slope with a maximum temperature of 100°C, which will reduce the current to 0 amps. Current ratings are based on ambient temperature of 25°C.
3. RS232 format is 8Bit, No Parity and 1 Stop bit.
4. Condensing humidity will damage the motor controller.

Sparkfun I2C Mux Board

1 Features

- 1-to-8 Bidirectional Translating Switches
- I^2C Bus and SMBus Compatible
- Active-Low Reset Input
- Three Address Pins, Allowing up to Eight TCA9548A Devices on the I^2C Bus
- Channel Selection Through an I^2C Bus, In Any Combination
- Power Up With All Switch Channels Deselected
- Low R_{ON} Switches
- Allows Voltage-Level Translation Between 1.8-V, 2.5-V, 3.3-V, and 5-V Buses
- No Glitch on Power Up
- Supports Hot Insertion
- Low Standby Current
- Operating Power-Supply Voltage Range of 1.65 V to 5.5 V
- 5-V Tolerant Inputs
- 0- to 400-kHz Clock Frequency
- Latch-Up Performance Exceeds 100 mA Per JESD 78, Class II
- ESD Protection Exceeds JESD 22
  - ±2000-V Human-Body Model (A114-A)
  - 200-V Machine Model (A115-A)
  - ±1000-V Charged-Device Model (C101)

Sparkfun RedBoard

- ATmega328 microcontroller with Optiboot (UNO) Bootloader
- CH340C Serial-USB Converter
- AP2112 Voltage Regulator
- A4/A5 Jumpers
- 3.3V to 5V Voltage Level Jumper
- Input voltage - 7-15V
- 1 Qwiic Connector
- 20 Digital I/O Pins (6 PWM Outputs and 6 Analog Inputs)
- ISP Header
- 32k Flash Memory
- 16MHz Clock Speed
- All SMD Construction
- R3 Shield Compatible
- Improved Reset Button

TB6600 Microstep Driver

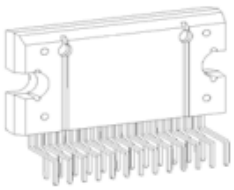
TOSHIBA BiCD Integrated Circuit Silicon Monolithic

TB6600HG

PWM Chopper-Type bipolar
Stepping Motor Driver IC

The TB6600HG is a PWM chopper-type single-chip bipolar sinusoidal micro-step stepping motor driver.
Forward and reverse rotation control is available with 2-phase, 1-2-phase, W1-2-phase, 2W1-2-phase, and 4W1-2-phase excitation modes.
2-phase bipolar-type stepping motor can be driven by only clock signal with low vibration and high efficiency.

TB6600HG



HZIP25-P-1.00F

Weight:
HZIP25-P-1.00F: 7.7g (typ.)

Features

- Single-chip bipolar sinusoidal micro-step stepping motor driver
- Ron (upper + lower) = 0.4 Ω (typ.)
- Forward and reverse rotation control available
- Selectable phase drive (1/1, 1/2, 1/4, 1/8, and 1/16 step)
- Output withstand voltage: Vcc = 50 V
- Output current: I_{OUT} = 5.0 A (absolute maximum ratings, peak)
  I_{OUT} = 4.5 A (operating range, maximal value)
- Packages: HZIP25-P-1.00F
- Built-in input pull-down resistance: 100 kΩ (typ.), (only TQ terminal: 70 k Ω (typ.))
- Output monitor pins (ALERT): Maximum of I_{ALERT} = 1 mA
- Output monitor pins (MO): Maximum of I_{MO} = 1 mA
- Equipped with reset and enable pins
- Stand by function
- Single power supply
- Built-in thermal shutdown (TSD) circuit
- Built-in under voltage lock out (UVLO) circuit
- Built-in over-current detection (ISD) circuit

Topcon B-125

TOPCON

B125 GNSS OEM Board

TRACKING	
Channels	226 Universal Tracking Channels™
Signals Tracked	GPS: L1, L2, L2C, L5 GLONASS: L1, L2, L3 Galileo: E1, E5a, E5b, E5AltBOC BeiDou: B1, B2 QZSS: L1, L2, L1C, L1-SAIF, L2C, L5 SBAS: L1 L-Band
ACCURACY[1] (RMS)	
Standalone	H: 1.2 m; V: 1.8 m
DGPS	H: 0.3 m; V: 0.5 m
SBAS	H: 0.8 m; V: 1.2 m
RTK	H: 5 mm + 0.5 ppm x baseline; V: 10 mm + 0.8 ppm x baseline
RTK Initialization	Time: < 10 seconds Reliability: > 99%
HD2	Heading 0.2°/D, where D is the inter-antenna distance in meters Inclination 0.3°/D, where D is the inter-antenna distance in meters
Velocity	0.02 m/second
Time	30 nsec
ACQUISITION TIME	
Hot / Cold Start	< 15 sec / < 44 sec typical
Reacquisition	< 1 sec
COMMUNICATION INTERFACES	
RS232	2x ports up to 460.8 kbps
LVTTL UART	1x ports up to 460.8 kbps
USB 2.0 (client)	1x port up to 480 mbps (High Speed)
CAN	1x port (without transceivers), CAN 2.0 A/B , NMEA2000 compliant
Ethernet	1x port supporting TCP/IP, FTP, Ntrip Server/Client

I/O	
PPS	1x output with 5 ns resolution, LVTTL, configurable edge, period, offset, and reference time
EVENT	1x input with 5 ns resolution, LVTTL, configurable edge and reference time
DATA AND MEMORY	
SD card support	Industrial SLC SD card, 20 Hz writing rate, up to 32GB capacity
Data Update/Output Rate	1 Hz – 100 Hz Selectable
Data Formats	TPS, RTCM SC104 2.x and 3.x, CMR/CMR+[2], BINEX
ASCII Output	NMEA 0183 versions 2.x, 3.x, and 4.x
ENVIRONMENTAL	
Temperature	Operating: -40°C to 85°C; Storage: -40°C to 85°C
Vibration	4g Sine Vibe (SAEJ1211); 7.7g Random Vibe (MIL-STD 810F)
Humidity	95%, non-condensing
Shock	Operational IEC68-2-27, 11ms, 40g Survival IEC68-2-27, 11ms, 75g
Acceleration	20g
POWER	
Voltage / Power Consumption	3.4 VDC to 5.5 VDC / 2.0 W typical
LNA Power	+3.4 V to +5.5 V (internal), +4.8 V to +5.16 V (external) at 0 – 120 mA
PHYSICAL	
Dimensions / Weight	55 mm x 40 mm x 10 mm / 20 g
Main Connector	80-pin Hirose
Antenna Inputs	2 ESD protected
Antenna Connectors	Hirose H.FL

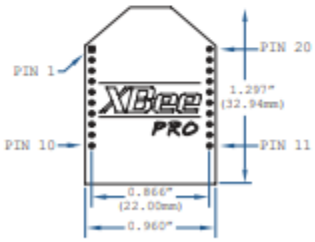
1. These specifications will vary depending on the number of satellites used, obstructions, satellite geometry (PDOP), occupation time, multipath effects, and atmospheric conditions. Performance may be degraded in conditions with high ionospheric activity, extreme multipath, or under dense foliage. For maximum system accuracy, always follow best practices for GNSS data collection.

2. CMR/CMR+ is a third-party proprietary format. Use of this format is not recommended and performance cannot be guaranteed. Use of industry standard RTCM 3.x is always recommended for optimal performance.

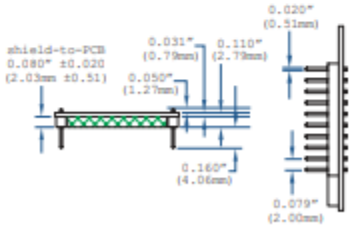
Xbee S3B Radio

SPECIFICATIONS	Digi XBee-PRO® XSC (S3)	Digi XBee-PRO® XSC (S3B)
HARDWARE		
PROCESSOR	ADF7025 transceiver, Atmel AT91SAM7S	ADF7023 transceiver, Cortex-M3 EFM32G230 @ 28 MHz
FREQUENCY BAND	902 MHz to 928 MHz	
ANTENNA OPTIONS	Wire, U.FL, RPSMA	
PERFORMANCE		
RF DATA RATE	10 Kbps	10 Kbps or 20 Kbps
INDOOR/URBAN RANGE*	Up to 1200 ft (370 m)	Up to 2000 ft (610 m)
OUTDOOR/LINE-OF-SIGHT RANGE*	Up to 6 mi (9.6 km)	Up to 9 mi (14 km) w/ dipole antenna Up to 28 mi (45 km) w/ high-gain antenna
TRANSMIT POWER	Up to 20 dBm (100 mW)	Up to 24 dBm (250 mW) software selectable
RECEIVER SENSITIVITY	-106 dBm	-109 dBm at 9600 baud -107 dBm at 19200 baud
FEATURES		
SPREAD SPECTRUM	FHSS	
OPERATING TEMPERATURE	-40° C to +85° C	
POWER		
SUPPLY VOLTAGE	3.0 - 3.6 VDC	2.4 to 3.6 VDC
TRANSMIT CURRENT	265 mA	215 mA
RECEIVE CURRENT	65 mA	26 mA
SLEEP CURRENT	45 uA	2.5 uA
REGULATORY APPROVALS		
FCC	MCQ-DigiXBeeXSC	MCQ-XBPS3B
IC	1846A-DigiXBeeXSC	1846A-XBPS3B
C-TICK	No	Australia

TOP VIEW



SIDE VIEWS



*Range figure estimates are based on free-air terrain with limited sources of interference. Actual range will vary based on transmitting power, orientation of transmitter and receiver, height of transmitting antenna, height of receiving antenna, weather conditions, interference sources in the area, and terrain between receiver and transmitter, including indoor and outdoor structures such as walls, trees, buildings, hills, and mountains.

WWW.DIGI.COM

Appendix C - Wiring Diagram

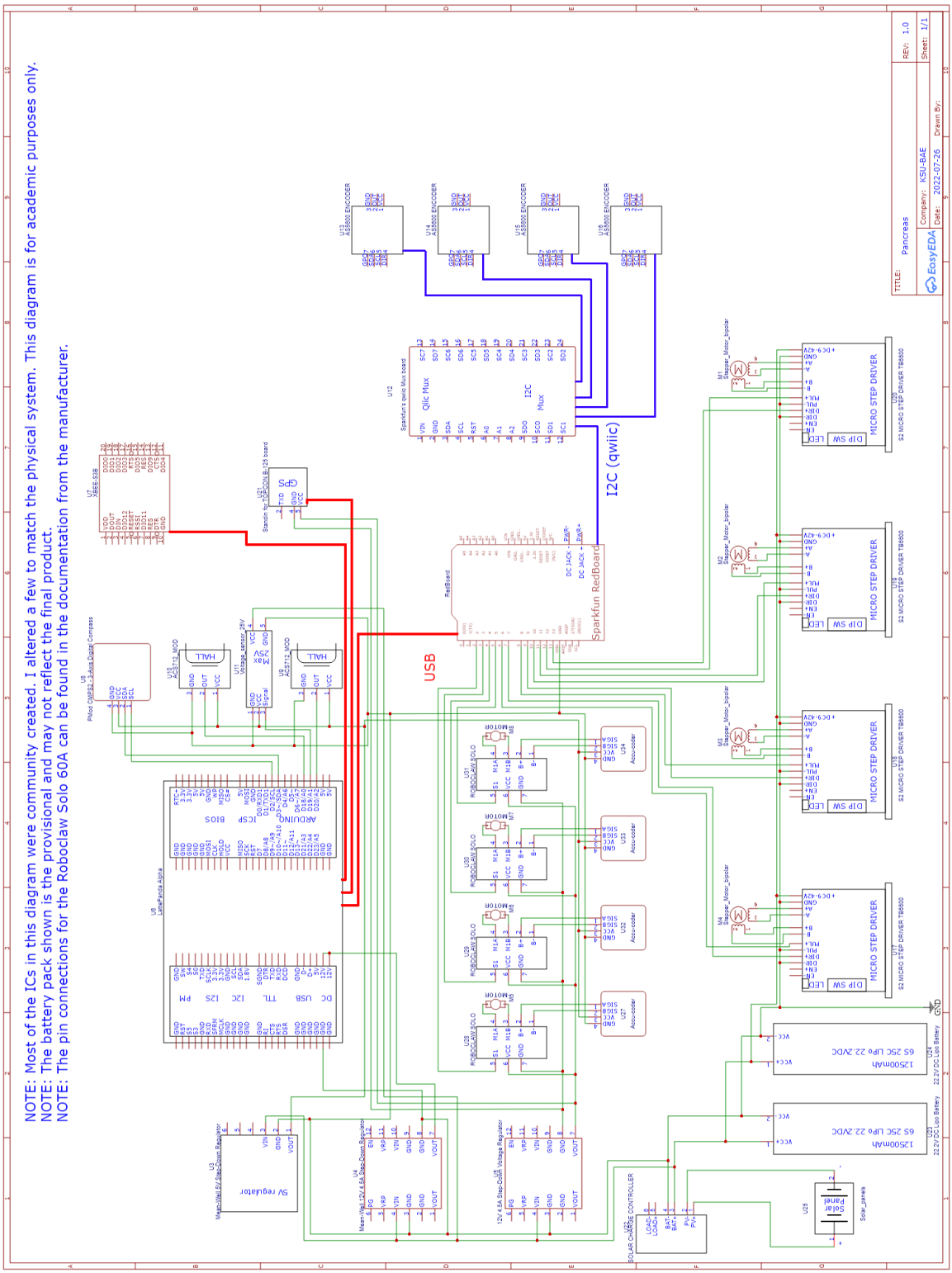


Figure 45: Wiring Diagram