Reaction-Wheel Stabilized Bicycle Robot

Project Paper

Introduction to Robotics – EGN4060C

Due Date: April 17th, 2020

Team Members:

Eric Baker

Brett Burgess

Salomon Hassidoff

College of Engineering and Computer Science

Introduction

For our project we set out to build a bicycle robot that remains upright and stable using a reaction wheel as the balancing mechanism. The main model our project is aiming to tackle is that of an inverted pendulum. The inverted pendulum can be better visualized if you can imagine the bicycle rotating about the longitudinal axis, or the x-axis shown in the figure below. The reaction wheel stabilizes the bicycle robot if it tilts too much to one side by rotating in the same direction as it is falling which creates a counter torque; it operates on the principle of conservation of angular momentum, i.e. the angular momentum of the flywheel remains equal and opposite that of the bicycle robot.

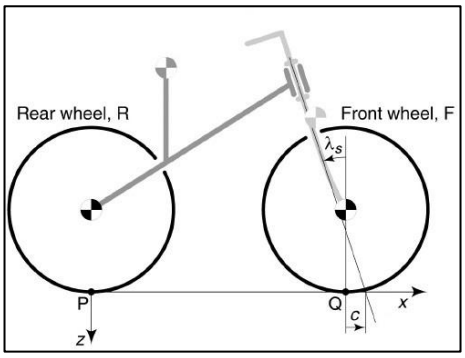

Figure 1: Bike Orientation Diagram

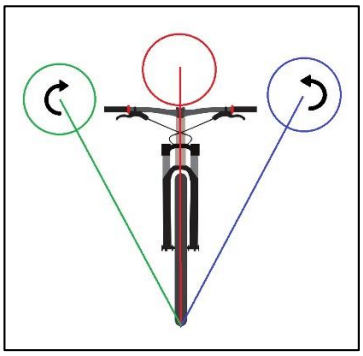

Figure 2: Reaction Wheel Spin Direction

There are several robots that make use of reaction wheels for stabilization that served as inspiration for our project. The most direct comparison is a robot called Murata Boy and Murata Girl which are self-balancing robots made by Murata Manufacturing. Murata Boy is a self-balancing, pedaling bicycle robot that balances using a reaction wheel seen in the chest of the robot rider. Murata Girl is very similar with the added complexity of needing to balance on the longitudinal and lateral axis because it uses a single wheeled unicycle robot. Murata Girl requires therefore requires two reaction wheels mounted facing along the two axes. There is also a robot called Cubli made by the Institute for Dynamic Systems and Controls at Zurich. Cubli is a cube with three reaction wheels integrated into its structure, each reaction wheel providing stabilization in one of the three special axes.

There is lots of active research that are looking into the design and application of reaction wheels in robotic systems. The field most known for making use of reaction wheels is astronautics. Satellites use reaction wheels or control moment gyros for attitude while in orbit. Two researchers from Norvik University in Norway aimed to design a reaction wheel small enough and light enough to be used in small form factor CubeSats [1]. Larger orbiting spacecraft such as the Kepler telescope, Dawn, Skylab, and the International Space Station all make use of reaction wheels or CMGs for 3-axis attitude control.

Figure 3: Murata Boy, Murata Girl, Cubli left to right

Method

Our robot relies on a feedback control loop to maintain a lean angle of 0°. Any deviations from this setpoint will force the bicycle to correct its angular position. The general structure of this feedback loop is shown in Figure 3.

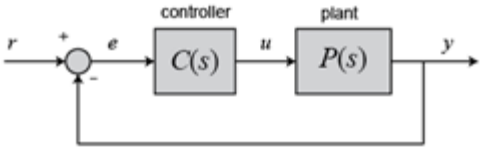

Figure 3: Feedback Loop

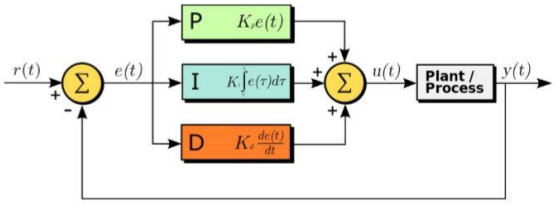

Figure 4: Feedback Loop w/ PID Controller

Looking at the block diagram above, the plant is the system that we want control. In our case, it is the bicycle. The purpose of the feedback loop is to constantly measure the error (e) between the desired output (r) and actual output (y). That error is fed into a controller which calculates the appropriate actuating signal (u) to reach the desired output.

The controller being used for this project is a PID controller [2]. It consists of a **P**roportional, **I**ntegral, and **D**erivative term as shown in Figure 4. The proportional term looks at the present error, the integral term looks at the past error, and the derivative term predicts future error. Together, they form a robust controller to effectively balance the bicycle.

The electronics used in the project include an Arduino Uno, an MPU 6050 accelerometer/gyroscope, a 12 Volt DC motor with a dual hall effect encoder, and an L298N motor controller. The MPU 6050 measures the angular acceleration of the bike and integrates that value to determine angular position [3]. The goal is to maintain a roll angle of 0° as best as possible. The Arduino Uno calculates and feeds the error through the PID controller, which determines the appropriate actuating signal. The controller output is given as:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

where $K_p, K_i, and\ K_d$ are the tuning parameters of the PID controller. This is converted to a Pulse Width Modulated (PWM) signal and written to the motor driver to control speed. The L298N comes equipped with an H-bridge. The Arduino is programmed to switch the polarity of the DC motor based on direction of tilt. Figure 5 illustrates a wiring schematic of our setup.

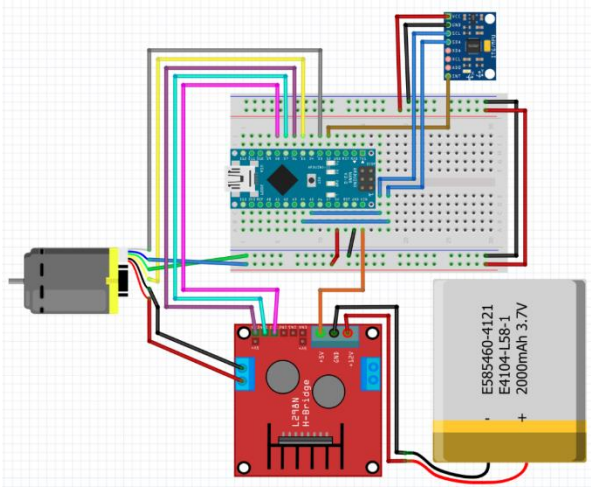

Figure 5: Wiring Schematic

Figures 6 and 7 below show the first iteration of our self-stabilized bicycle. The sole purpose of this prototype was to test the balancing mechanism before implementing other mechanical parts, such as steering.

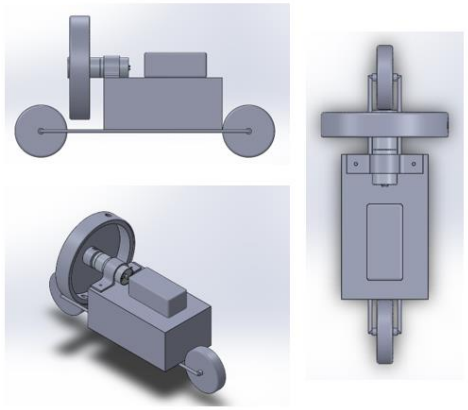

Figure 6: CAD Model

Figure 7: Physical Prototype

The second iteration of the design incorporated several changes that can be seen in Figure 8 and 9. To save on a lot of weight and space, an Arduino Nano was used which is a smaller microcontroller platform with the same capabilities as the Arduino UNO. Then, the Arduino Nano and the MPU6050 IMU were soldered onto a prototyping circuit board to further save on space and tidy up the wiring rather than using a breadboard as shown in Figure 10. The IMU I^2C connections were hardwired into the Arduino Nano, female pin connections were soldered to each Arduino Nano pin for any other connections, and both were powered on the same power bus.

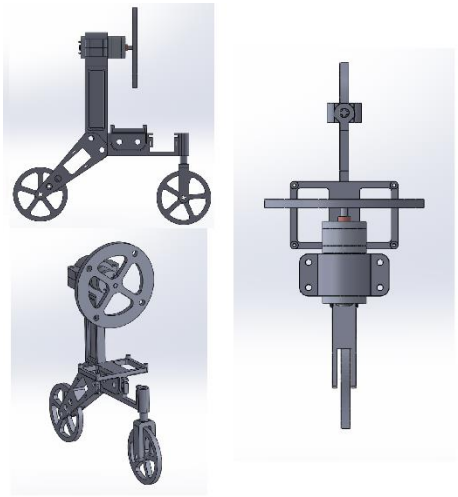

Figure 8: CAD Model 2nd Iteration

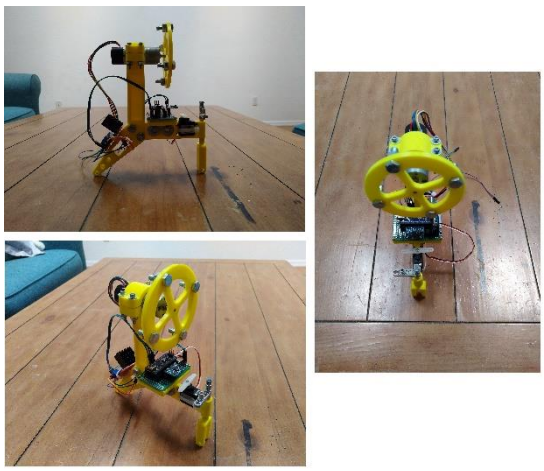

Figure 9: Physical Prototype 2nd Iteration

The frame, wheels and reaction wheel were 3D printed out of yellow PLA. To give some modularity to the design, the main chassis is a beam with holes for mounting electronics. All of the components were mounted using nuts and bolts found at home. In order to give the robot the capability to steer, a servo motor was connected to the front fork via a linkage. A servo was chosen for its simple integration with the Arduino as well as its relatively precise positional control. Steering functionality has not yet been implemented on the robot however.

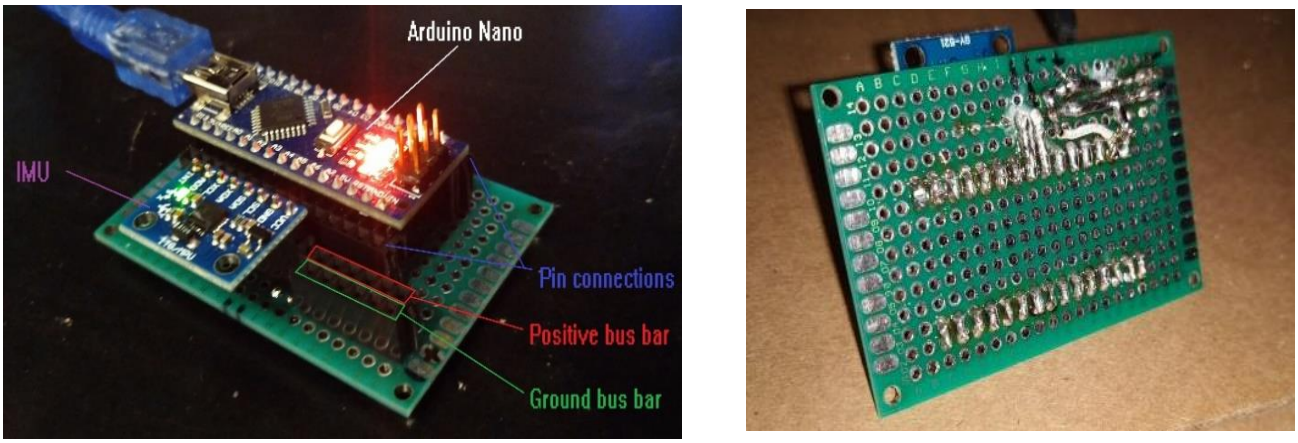

Figure 10: Microcontroller and IMU circuit board layout

Discussion

Tuning the controller was a manual process which made it difficult to select the most optimal values for stability. The robot did a decent job of staying upright but struggled with minor perturbations. Due to time constraints we could not make the bicycle robot as stable as we would have liked. Given more time and availability of materials we would have liked to try different controller tuning methods such as auto tuning or pole placement. We would also like to implement a secondary PID feedback control look that adds the information from the encoder on the DC motor. Using a motor encoder for a system such as this one is said to increase stability by a significant margin. Other features we did not have time to implement are steering and object avoidance.

One problem we experienced was a drift in IMU measurements, caused by the accumulation of integration errors. This made it difficult to accurately track the robot's angular position over time. We implemented a complementary filter, which fused the accelerometer and gyroscope data for a more accurate inclination angle. However, this gave marginal improvements. We decided to use a Kalman filter [4], which operates in two stages: prediction and correction. In the first stage, the Kalman filter estimates the current state of the robot with some uncertainty. After the next measurement is observed, the estimate is corrected with a weighted average of estimates and their certainties. This optimal estimation algorithm provided much better results than the complementary filter and made it easier to track the tilt of the robot.

At one point, the voltage regulator on our IMU sensor blew. Testing was halted for a few days and we attempted to simulate the robot in Gazebo whilst waiting for a replacement. The prototype was modeled in SolidWorks and exported as a URDF [5]. This provides Gazebo with a description of the links, joints, and inertial properties of the robot.

We then added a few plugins to help track joint states and implement controllers. The first plugin is called *gazebo_ros_imu_sensor* [6] which simulates an IMU sensor and provides orientation of the robot. The second plugin is called *gazebo_ros_control* [7] and it provides hardware interfaces for actuating joints. We fed the orientation through a PID loop and calculated the necessary motor velocity; however, we were unable to command the motor using a *JointVelocityController*. Further investigation is needed to get the simulation operating properly.

Conclusion

Since this is just a prototype, we plan on continuing development of the final product. After the robot is fully capable of self-balancing, we hope to add obstacle avoidance and path-planning. This would make an almost entirely autonomous bicycle. The full implementation of the original idea needs more time than initially planned. We would like to get our prototype working in simulation to more efficiently test the path-planning and obstacle avoidance. Once we can get the correct plugins and controllers simulated into ROS and Gazebo we will be able to better integrate our code and then transfer it over to the physical model.

When we first started the move over to simulation, our prototype did not have any steering. We wanted to get the original balance algorithm working before adding another variable. Moving forward, we plan to add steering into both the physical model (as seen in the second iteration) and into another simulated prototype in gazebo (a URDF of the second iteration).

References

[1] https://www.researchgate.net/publication/251889729_Reaction_wheel_design_for_CubeSats

[2] http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

[3] https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/

[4] https://github.com/TKJElectronics/KalmanFilter

[5] http://gazebosim.org/tutorials/?tut=ros_urdf

[6] http://gazebosim.org/tutorials?tut=ros_gzplugins

[7] http://gazebosim.org/tutorials/?tut=ros_control