



High Assurance Controller of a Self-Balancing Robot

Team 10

Forsman, Kulakevich, Mejia-Rodriguez, Wang



Electrical and Computer Engineering
Portland State University

Table of Contents

Abstract	3
Introduction & Motivation	3
Purpose and background of TWIPs in general	3
Background of Controller Design & Modeling	3
Background of Verification	4
Formal Requirements	4
Project Management	5
Organizational Tools	5
Work Assignments	5
Schedule	5
Issues	6
Modeling & Controller Design	7
Programming	10
Porting Code To Different Controller	10
Prototype	10
Serial data issue	12
New Robot	12
Compiling Embedded Rust	12
Cortex-m-quickstart	12
cbindgen	13
Writing Rust functions for C++	14
Use with Arduino	14
Controller Translation To Lustre	14
Verification	16
Overview	16
Verification of Controller	17
Next Steps	18
Testing	18
Deliverables	21
Conclusion	22
References	23
Appendix A - Test Plan	24
Appendix B - KIND2 Modifications	47

Abstract

Mobile robots are an important branch of robotics, and research on mobile robots dates back to the 1960s. There are two main reasons for the rapid development of mobile robots: one is that its application fields are becoming more and more extensive; the other is the rapid development of related disciplines. As the application field of mobile robots continues to expand, the environment and tasks faced are becoming more and more complex. Robots must often work in work spaces with small spaces and many corners. How to perform various tasks flexibly and quickly in a complex environment is a hot spot of current research. The task of exploring the process of implementing and verifying controllers in a “safe” programming language (Rust) for a two-wheeled self-balancing robot was proposed in this context.

This project ported the previous team’s PID controller code to a new microcontroller, and implemented a new PID controller written in Rust as well as Wifi-based data recording and control. Efforts were successful to improve the Lustre/Kind 2 verification tool’s Rust code generator to generate Rust code that can be directly loaded onto the microcontroller. A fuzzy logic controller was designed, and implemented on the microcontroller, using Rust code generated from Kind 2. A new robot was built using upgraded parts and tested to confirm that the controllers worked to keep it upright. Attempts at verification of the controllers was ultimately unsuccessful, but insight into the next steps for verification and the project in general were gained.

Introduction & Motivation

Purpose and background of TWIPs in general

Tools and practices for designing and verifying high assurance software systems are not fully standardized. In order to explore the processes involved in this type of development, a test case using a control system for a two-wheeled inverted pendulum (TWIP) will be explored and documented. In specific, this capstone is based on the design, implementation, and verification of the control system for a TWIP. The TWIP is a complex system with nonlinearity, strong coupling, and an uncertain model. The self-balancing of the TWIP is achieved by the motor-driven wheels that respond to tilt and yaw data quantified by sensors and microprocessors.

Background of Controller Design & Modeling

In this design, the dynamics of the inverted pendulum are represented by nonlinear differential equations that are very difficult to solve analytically. An approach to getting around this is using

system conversion to convert the equations to a first-order differential equation that a programming language such as MATLAB can numerically calculate. The approach presented by [1] uses equations and a model that takes in the torque control inputs (one for each motor) and outputs displacement, velocity, tilt, and yaw. We will focus on a tilt controller that uses the tilt output of the system to provide the appropriate torque to each wheel to control the system. There exist many controllers for nonlinear systems such as LQR, NMSS PIP, PID, sliding mode control, and fuzzy logic control. It must be noted that the majority of these controllers assume a linear system that can be achieved by linearizing the TWIP about its upright position. However, a fuzzy logic controller yields promising results for nonlinear systems and is robust to modeling errors. Additionally, a PID controller is simple to implement and yields good results as long as the system maintains within its linear range. We will design each of these through the use of MATLAB and Simulink.

Background of Verification

In order to confirm that a piece of hardware or software will perform as designed, verification must be performed. One method of verification is called Model Based Verification, which uses mathematical abstraction for hardware or software processes and confirms that specified properties hold true for all cases, or are false for certain cases. This can be attempted using simulations or numerical analysis, but these methods are not as exhaustive as model-based techniques and can also suffer from problems such as roundoff errors and discretization of continuous systems. While verification procedures and processes are fairly standardized, combining verification with high assurance programming languages is not, and the translation between verified code and high assurance code is not trivial. When moving between two programming languages there is the chance that human error, or some aspect of the language will cause the previously verified property to become invalidated. The model checker specified in the PDS, Kind 2, gets around this issue by being able to generate code in the high-assurance language specified in the PDS, Rust. Using these facts, we attempted to leverage Kind 2 to verify and generate the controller code.

Formal Requirements

Unlike many other projects, our requirements were created by our Galois industry sponsors at the start of the project and we made slight modifications over the course of the project. The most important requirements are listed below:

- Extend the existing code for the inverted pendulum robot to include a Rust control library.
- Modify the Lustre/Kind2 Rust code generator to generate embedded-friendly Rust, which can be directly imported in your library.
- Develop and identify a model of a nominal self-balancing robot in Octave or MATLAB.
- Isolate the Software Under Test as a severable block to analyze only the linear/non-linear controller for specific properties.

- Develop and identify a dynamics model of the robot in Octave or MATLAB.
- Develop and verify a controller with a wider range of stable input conditions and compare its performance with the PID controller through both simulations and in the real system.
- Build another robot with the upgraded parts provided by Galois.

Project Management

Organizational Tools

We employed several organizational tools that were extremely helpful in communication and file/version tracking and storage. For communication we created a Discord server, which worked well as we were able to send messages when an issue came up, as well as have voice and video chats, along with screen sharing when the speed of communication needed to be increased.

For our file storage and tracking we used a GitHub repository of our own creation, as well as one that was set up by our sponsor and previous team. Each member used their own interface for the repository, with some using the command line interface and others using programs like GitKraken. This arrangement worked well to ensure file changes were trackable and accessible to everyone involved with the project.

All of the code and documentation for this project are contained in the repositories located at <https://github.com/GaloisInc/HighAssuranceControllerOfSelfBalancingRobotCapstone/> and <https://github.com/Artem1199/ECE-HACoSR2/>, with the latter being work that was done by only our team, and the former containing work from both the previous team and our team. These repositories are up to date.

Work Assignments

Due to each of our previous experience and interests, we separated the workload into four sections: the controller, the robot, the code, and the verification. These were assigned as follows:

Robot (build new one and perform testing) - Yuqi

Controller Selection and Design - Ignacio

Code (Port of previous code, Rust PID, and Kind 2 Rust generation) - Artem

Verification - Andrew

Schedule

We met formally at least once a week to go through the check-in questions that were suggested (What did you do last week? What will you do this week? Any blocks?), and these meetings were recorded in the Wiki of our repository, located at <https://github.com/Artem1199/ECE-HACoSR2/wiki>.

Additionally, we had many informal chats, both text based and voice based, throughout most weeks.

Due to the nature of the work, there were various dependencies between all of the separate areas. The controller selection and design had to occur before the verification of the controller, the previous team's code had to be ported to the new board before any of the new controllers could be implemented on the robot, and the controllers had to be implemented in order to perform any testing. In order to keep this all organized and on track, we produced a schedule using Microsoft Project, located here:

<https://github.com/Artem1199/ECE-HACoSR2/tree/master/Docs/Project%20Schedule>.

The schedule was followed fairly accurately until about March 15th when everything went off the rails due to COVID-19, at which point nothing was accurate and other major problems occurred. The same general order of execution was followed, but the dates are wildly inaccurate and some aspects were not fully completed.

Issues

We ran into a few hardships, mostly surrounding the COVID-19 restrictions. These included:

- Hardware for the new robot was inside a lab in the Fourth Avenue Building that we weren't able to access. Eventually we were able to negotiate access to the lab, but shortly after the campus was closed to all students. Shortly after that, we were able to negotiate a one-time access to the lab, during which a member of the team was able to retrieve the hardware.
- Due to the aforementioned lab closure, we were unable to test the physical hardware as thoroughly as desired. The home lab of the team member with the hardware has some testing equipment, but was missing some of the equipment we would have used to characterize the new robot's physical dynamics
- As we could not meet in person, all physical testing fell on one person, and our ability to contribute and collaborate meaningfully to each other's focus was restricted.
- There were various personal issues including a hospitalization, and a loss of childcare for team members.

These trials are only mentioned in order to give a full account of the project, and to reflect on the problems that were overcome. By having a collaboration site (GitHub) and a chat client (Discord), most of the potential communication problems were navigated successfully, and the project as a whole went better than expected. While the requirements list was renegotiated to move the physical hardware building and testing to the non-essential list of requirements, these requirements were still completed.

Modeling & Controller Design

Obtaining a valid model for the TWIP is important for making sure that the controllers we design are good in simulation before we implement it on the system. Tuning a controller on the actual system is much more time consuming and can result in damage to the actual robot. Much research on the model for a TWIP was done by the previous capstone team. They implemented the following differential equations in MATLAB through a system conversion:

$$\begin{aligned}
 \dot{y}_1 &= y_2 \\
 \dot{y}_2 &= \frac{\cos(y_1) I_m \left(-m l y_2^2 \sin(y_1) - \frac{\tau_l}{r} - \frac{\tau_r}{r} - d_l - d_r \right) + m g l \sin(y_1) \left(M + 2M_w + m + \frac{2I_w}{r} \right)}{(l^2 m + I_m) \left(M + 2M_w + m + \frac{2I_w}{r^2} \right) - (\cos(y_1))^2 l^2 m^2} \\
 \dot{y}_3 &= y_4 \\
 \dot{y}_4 &= \frac{2d \left(\frac{\tau_l}{r} - \frac{\tau_r}{r} + d_l - d_r \right)}{I_p + 2 \left(M_w + \frac{I_w}{r^2} \right) d^2} \\
 \dot{y}_5 &= y_6 \\
 \dot{y}_6 &= \frac{(l^2 m + I_m) \left(-m l y_2^2 \sin(y_1) - \frac{\tau_l}{r} - \frac{\tau_r}{r} - d_l - d_r \right) + m^2 g l^2 \sin(y_1) \cos(y_1)}{(l^2 m + I_m) \left(M + 2M_w + m + \frac{2I_w}{r^2} \right) - (\cos(y_1))^2 l^2 m^2}
 \end{aligned}
 \quad
 \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} \alpha \\ \dot{\alpha} \\ \theta \\ \omega \\ x \\ \dot{v} \end{bmatrix}$$

Figure 1: The dynamic equations for the TWIP in the appropriate form for MATLAB.

MATLAB's ode45 numerical solver can then be used to implement these equations in both script and in Simulink. The previous team also developed a model of the TWIP in Simulink which is shown below with two PID controllers attached to it:

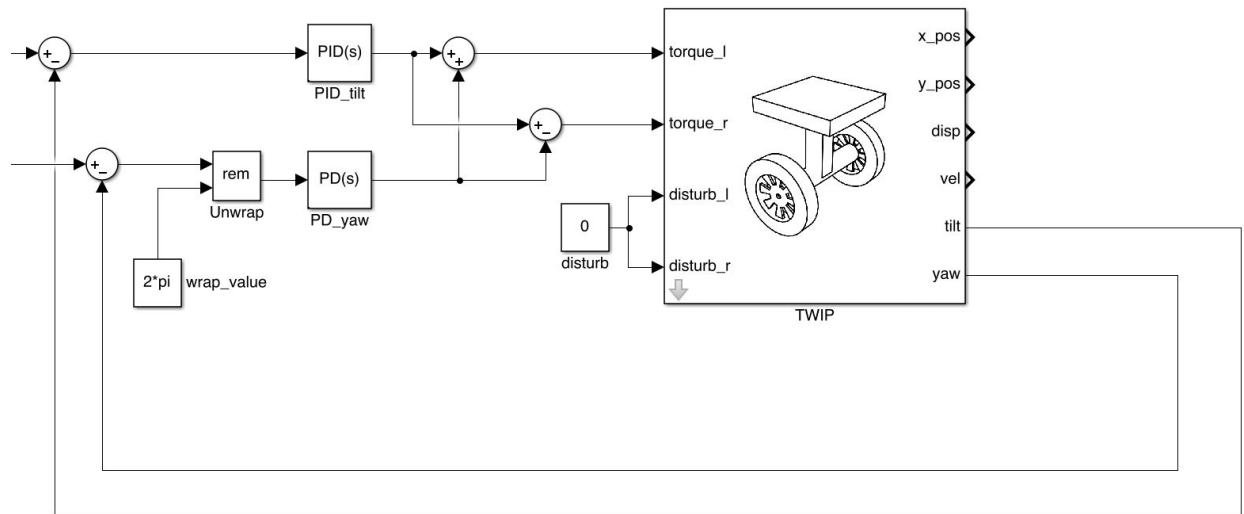


Figure 2: The Simulink block diagram of the TWIP.

It should be noted that this model has both a tilt and yaw controller while we only implemented a tilt controller. Nevertheless, it is important to know that the yaw controller allows the robot to sense angular displacement and return to its initial angular position when moved by exterior disturbances. We used this model in our initial controller design.

Fuzzy logic control (FLC) has been used on TWIP before and has been proven superior to conventional linear controllers such as PID controllers. Additionally, it is robust to modeling errors, which is important in a design such as this where only an approximation of the model can be attained. FLC has three main sections, which are fuzzification, inference, and defuzzification. Fuzzification takes an input and determines its location on the input membership function plot, inference uses the rule base to determine the appropriate location of the output on the output membership function, and defuzzification performs a mathematical operation on the fuzzy output to turn it into a crisp output. The process is depicted in the block diagram below:

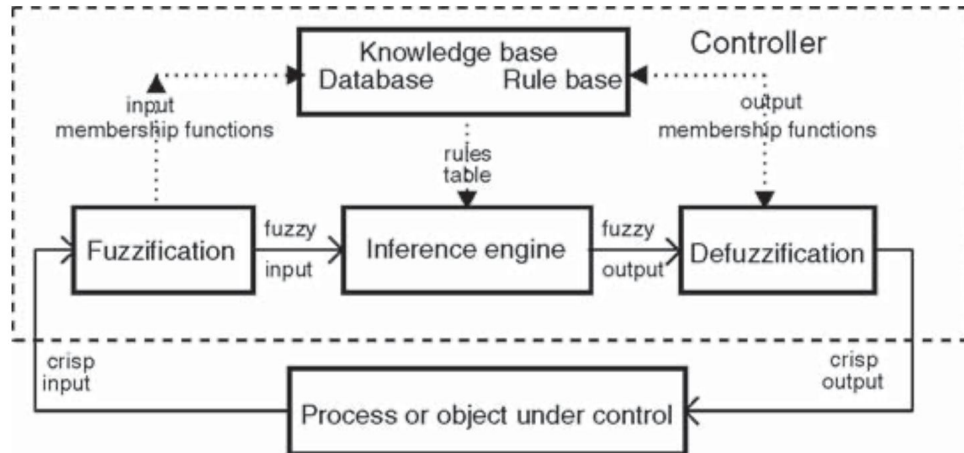


Figure 3: The general fuzzy control system block diagram[2].

To simplify matters, MATLAB has a Fuzzy Logic Toolbox that can be used to develop a fuzzy logic controller. The input and output membership functions typically take the form of triangular and rectangular membership functions as shown below:

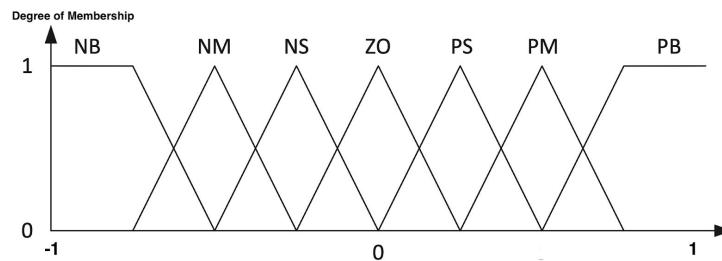


Figure 4: The membership function plot for the Fuzzy logic controller inputs and outputs.

It should be noted that the inputs and outputs are typically normalized so that the maximum input and output are ± 1 . The inputs and outputs can then be denormalized by adding gain blocks on the controller. Ultimately, a MATLAB script written by Matt Clarke (Industry Sponsor) was used to implement the controller in Simulink. This script is essentially a more efficient version of the Fuzzy Logic Toolbox. Additionally, we decided to use a proportional-plus-derivative (PD) fuzzy logic structure which seemed to be a common method for TWIPs in many of the articles we explored on this topic. The final membership functions corresponding to the proportional and derivative inputs and output are shown below for the tilt controller:

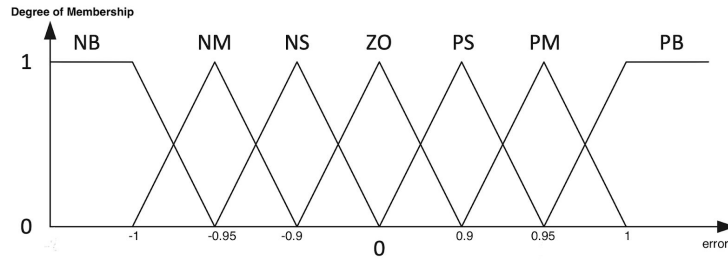


Figure 5: The error input membership function plot.

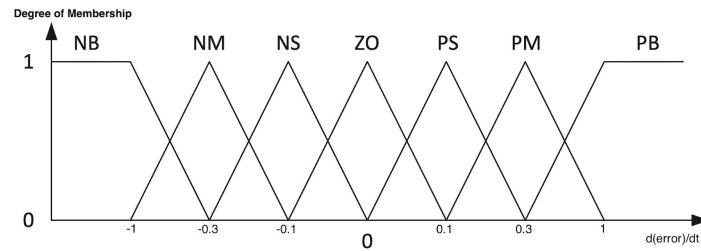


Figure 6: The derivative of the error input membership function plot.

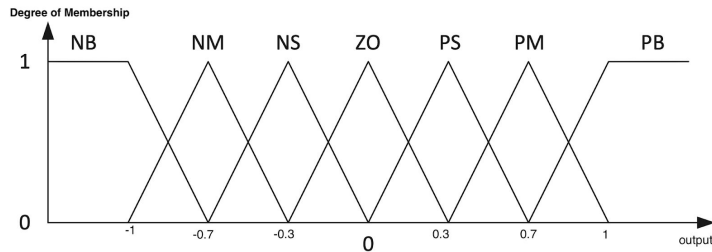


Figure 7: The output membership function plot.

The output membership function will have to be scaled by the 70-80% of the stall torque of the motors. This is so that the motors operate far from their stall torque, which prevents the motors from being damaged. The result after some PD tuning for the tilt of the system is as follows:

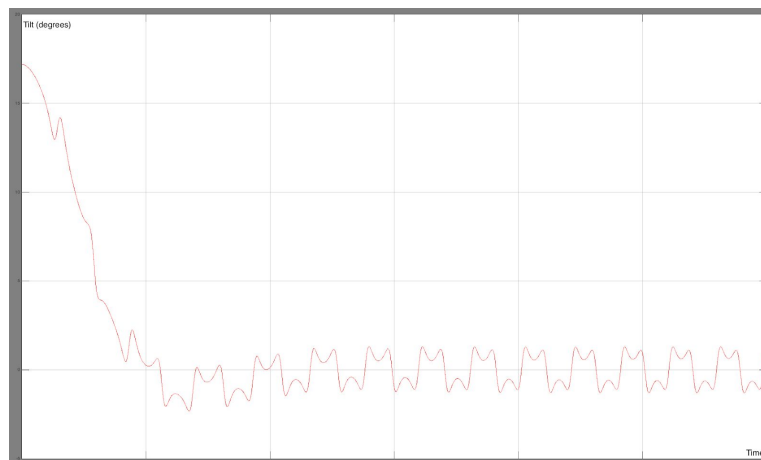


Figure 8: The tilt of the robot in response to a 17 degree initial condition.

The system is stable given an initial condition of about 17 degrees. However, we experienced small oscillations as the robot approached its steady-state tilt of zero. From here we decided to implement the controller on the actual robot and inspect how closely the simulations reflected reality.

Programming

Porting Code To Different Controller

Prototype

The original prototype robot was programmed on an ESP32 which had bluetoothSerial capabilities. This is a Xtensa processor that does not support Rust compilation yet. Galois picked out a new processor Arduino board and that was the MKR1010 which holds a Cortex-M0+ Arm based processor. It also has a second chip that can deal with BluetoothBLE or Wifi. The prototype robot used PWM to control the motor which was sent to a dual-h bridge to be amplified for the motors. So the following diagrams shows how the robots were rewired from the old processor to the new processor:

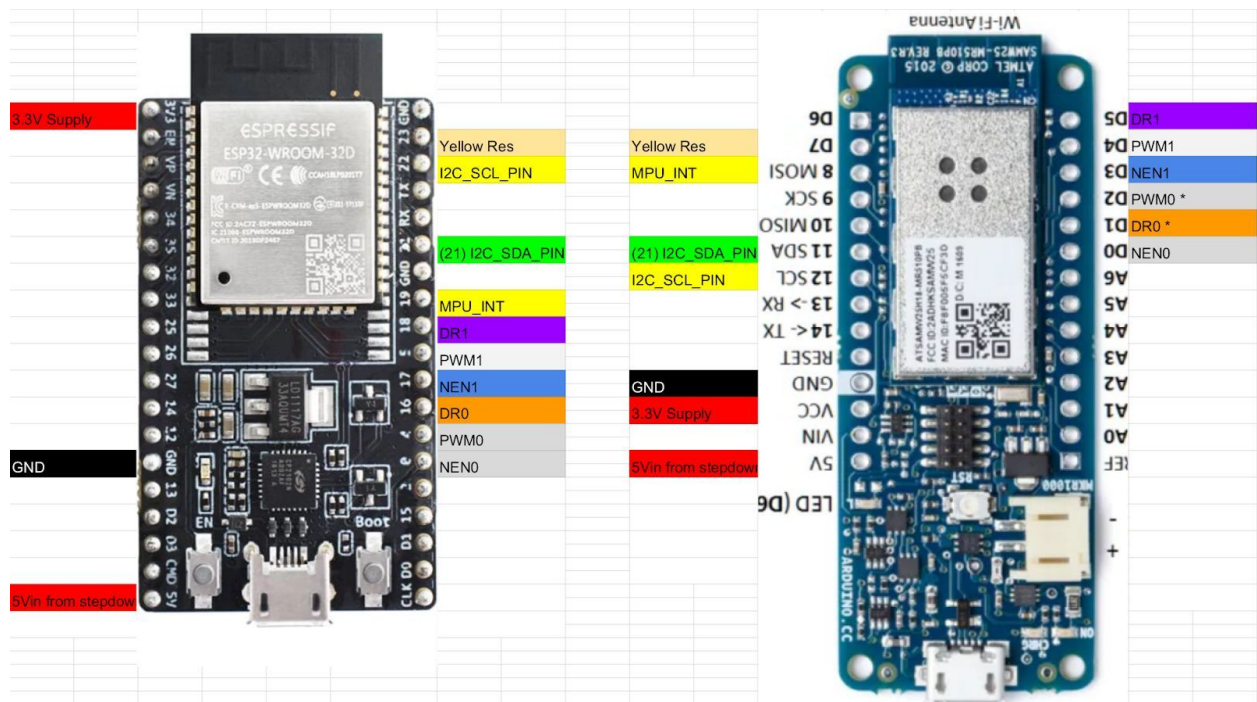


Figure 9: Diagram showing how the robots were rewired in comparison to the old board.

The one challenge that arose from porting the code to a different processor board was the change from BluetoothSerial to BluetoothBLE. We decided to use wifi communication instead

because it seemed simpler to implement at the time than BluetoothBLE. It was also challenging to get wifi to work consistently without hanging up, because reading data that arrives to the processor from the wifi chip is also very time consuming. This meant that wifi communication had to be as small and unobtrusive as possible so the processor didn't have to spend too much time handling it. The floats are encoded into bytes by the processor and decoded by another machine. Wifi communication isn't perfect though, and the jump from Arduino -> Router -> PC often causes hang ups which make controlling the robot very difficult. The heuristical data that is received from the processor was tested, and is very reliable. Here is a block diagram of the data communication between the Arduino and a host machine:

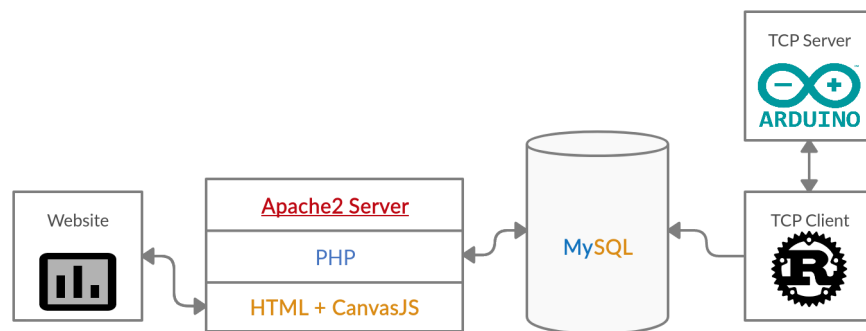


Figure 10: A block diagram of the wireless data communication between the Arduino and a host machine.

The website is written in PHP / javascript and it just displays data out of a MySQL database. A small Rust TCP client is able to connect to the Arduino and send commands or receive data from the Arduino. TCP decodes the data that is received from the Arduino back into floating point values and prints them to a serial output. This TCP client can also write to an SQL database for the website to display. Having it work with MySQL is pretty tedious, so the Rust plotter library + piston may be a better option for displaying the data in the future.

Serial data issue

With the new board there seems to be an issue with Serial data communication when using it with the Arduino IDE. This was tested by running the robot with a basic `Serial.print()` output enabled. It was found that after about 1-2 minutes the robot would freeze up when trying to deal with the MPU6050 data that arrives through interrupts. Without the `Serial.print()` output the robot can run for hours without hanging up. It's likely that the `Serial.print()` takes too long to process and causes the MPU6050 FIFO buffer to overflow. This can also be very dangerous because after a hang up the robot motor wheels will continue to run. There was an attempt to mitigate this problem by adding a fail state where the robot just blinks a RED LED instead of running the motors. `Serial.print()` should only be used for minor debugging on this processor and avoiding outside of debug will save a lot of headaches.

New Robot

The new robot didn't have many changes, the only significant difference was the addition of the Adafruit motor shield. This motor driver used I2C communication instead of PWM. That means all of the code using PWM was removed, and the motors were interacted with using I2C.

Compiling Embedded Rust

Cortex-m-quickstart

Compiling embedded Rust code is not incredibly challenging, the process is well documented online. There is a very helpful repository that makes this process incredibly easy, called the "cortex-m-quickstart" repository. The front page discusses everything that needs to be done to compile code for a Cortex-M0+. Basically it involves installing `cargo generate` which creates a template folder with a few examples of Rust code that can be run on the processor. A few of the configuration files in the template have to be modified to fit the processor specifications, so the correct memory size needs to be recorded in the template. This template has already been created and modified for the specific processor used in this project. The template has been saved with the rest of the Arduino code.

cbindgen

Making the library compile code that can be used with C++ is a bit more challenging. To generate a library that can be used by the Arduino, but can also be compiled for an embedded environment, the crate type has to be set to:

```
crate-type = ["staticlib"]
```

This is done in the cargo.toml file. In the code itself an error handler has to be defined, currently the default provided by cortex-m-quickstart is used:

```
#[alloc_error_handler]
fn alloc_error(_layout: Layout) -> ! {
    asm::bkpt();
    loop {}
}
```

This has to be specified at the top of the main rust code with:

```
#![feature(alloc_error_handler)]
```

and

```
#[global_allocator]
static ALLOCATOR: CortexMHeap = CortexMHeap::empty();
```

Another command is also required, the `no_std` command. This will remove the Rust standard library that cannot be used with embedded Rust.

```
#![no_std]
```

To use Rust code in C++ a very useful software tool exists called `cbindgen`. This program will generate a C++/C header files from the Rust code. The functions from Rust can be used by C++. To expose those the Rust functions to `cbindgen` a couple things are required in the code:

```
#[no_mangle]
```

This line needs to go before every function that you want exposed. As well as

```
pub extern "C" fn
```

For every function that needs to be exposed.

Writing Rust functions for C++

Using Rust functions in C++ has some drawbacks, namely the interfacing between classes and methods. There doesn't seem to be an easy way to call Rust class methods from C++. People able to use methods is critically important when it comes to using the code generated from Kind2. The method that was found for this project involved creating a basic Rust function that can interface with C++, and having that Rust function deal with the class and methods. This exact code used is extensively documented in the `otis-arduino` repository, but a summary of the process involves. [3] Allocating memory for the Rust struct in C++ using `new`. Then passing a pointer to that memory to Rust. Rust can then use an `unsafe` bit of code to dereference the point and turn it into a mutable reference. Rust can then call methods from that struct using the mutable reference. When using a constructor method, Rust can overwrite the data inside the mutable reference and return nothing. When using a method for computation Rust can override values inside the mutable reference, but can also return a result value to C++. A very useful guide exists on this process called the *The (unofficial) Rust FFI Guide* [12] that goes into depth on various ways to communicate between Rust and other languages.

For this project the code the Rust code was separated into multiple files. A `lib.rs` file held all of the functions that called the Rust controller methods. The controllers generated by Kind2 are held in a separate file that is called by the `lib.rs` file.

Use with Arduino

Using the Rust code on the Arduino IDE is a bit strange, but not very difficult. The code had to be placed in the Arduino libraries folder, and enabled in the library's properties file. The process has also been documented in the `otis-arduino` repository [3].

Controller Translation To Lustre

Two different controllers were written in Lustre to test the Kind 2 Lustre to Rust code generator. A quick overview of Lustre, it is a language made to be verifiable. It's used in a lot of critical control software for airplanes, helicopters, and nuclear power plants. The program has function and nodes, nodes are where functions are generally declared. Functions are memoryless nodes. Lustre also has three main types, real, int, and bool, these are f64, i64, and bool in Rust respectively. Lustre is also capable of using arrays, but this option is not available when translating from Lustre to Rust.

A PID controller was written based on Brett Beauregard PID controller. [6] The lustre PID code that was written has three different functions. The first Compute node checks to see if enough time has elapsed since the last calculation. If enough time has elapsed, then it calls another node called PID_Calc to do the actual calculation. The PID_Calc node stores some of the previous error values and calculates the new output. PID_Calc also calls the a limit function that limits our output from -255 to 255 which are the motor shield output limits. The Lustre code itself doesn't store the PID constants or setpoint, those values are passed in from C++ to make it easier to tune the robot. The Lustre code does store the error values and reuses them with the `0.0 -> pre` command. The pre command tells the program to use the value from the previous state instead of the current value.

A second fuzzy logic controller was written in lustre heavily based on Matthew Clark and Kuldip Rattan's Matlab fuzzy logic controller that is written in matlab. The matlab based controller is heavily based on recursion and matrices. Lustre can't use arrays when generating Rust code, so the matrices were expanded out. Recursion doesn't work well either, so the recursion was expanded out. Based on an input, the controller picks a value out of a matrix and calculates the output.

This fuzzy logic controller only accounted for a proportional and a derivative component, but no integral. In a real world test this controller worked about as well as a standard PID controller, the only downside was the lack of an integral component. This meant that the robot had a hard time reaching its setpoint, and would have some setpoint error. This was resolved by adding the integral component from the PID controller to the fuzzy logic controller. This way we had the `P` and `D` components of the fuzzy logic controller and the `I` component of a PID.

Kind2 Lustre to Rust Modifications

Kind2 is capable of generating Rust code. This is done by running the command:

```
`kind2 <controller name>.lus --compile true`.
```

Kind2 generates the entire Rust folder, and the code can be tested right away using

```
`cargo run`.
```

The main Rust file generated by the compiler has a trait called ``Sys`` that defines the methods for every struct in the program. Each node and function has its own struct to hold the input and output values. Each node and function also has an ``init`` and a ``next`` method that do the actual calculations and return the values. The ``init`` method is used to construct the struct, and afterwards the next method can be used to compute output. Basically the code can be used by calling ``init`` to create a struct, then calling ``next`` to calculate the output. In addition to the basic functionality the compiler also generates additional functions that make this program usable in a terminal environment. Several of the instructions in ``helpers`` are used to read the input of the command line, and to output values to the command line. This is obviously not possible in embedded systems so it had to be changed. Changes to the compiler only needed to be made to one file: `LustreToRust.ml`. This file is entirely written in Ocaml. This language was not necessary to learn to make these modifications. The Ocaml code essentially treats the Rust code that it plans on generating as strings that it modifies the strings based on the Lustre code that is input. It was possible to modify the code that is generated without modifying any Ocaml code by simply editing the strings. Significant changes were made to the strings to remove the command line features, and to remove the ``std`` library features. Additionally the code generated all of its Rust functions with error handling that would normally print an error message to the command line, that was removed since the errors would not be usable when interfacing with C++. Some changes were also made to the way it dealt with structs, instead of creating a new struct each time, the program changes the values in a mutable reference of that struct. This way the generated controller can be compatible with the mutable reference we create from the pointer passed in by C++. A more extensive list of the changes is located in the appendix B.

Notably assertions were not touched, and won't work properly when generated. These were not necessary for writing the controllers, so they were never accounted for. After modifying the `LustreToRust.ml` file, Kind2 can be rebuilt, and running the Kind 2 binary allows for someone to generate embedded Rust code using the compile command mentioned earlier. This generated code is entirely compatible with an embedded system, but it is still not C++ compliant.

The easiest way to use this controller is to place the generated controller Rust file in the ``cortex-m-quickstart`` folder mentioned above. A function is then necessary to receive a pointer

from C++, dereference the pointer and convert it to a Rust mutable reference, and then use that reference to call the ``init`` and ``next`` functions from the generated Rust code.

Verification

Overview

As mentioned in the background section, the method of verification for this project was specified to be the Kind 2 model checker by the sponsor in their project proposal. One of the first problems encountered was the installation of the software. There are numerous dependencies and if the user is not familiar with using a command line interface, this can be a huge stumbling block. However, there exists a Docker container, which allows the user to download the entire installation with minimal effort. A link to the installation procedure is included in the references section of this report [8].

Once the software was installed, we attempted to verify the previous team's PID implementation, using their specified properties. This route was taken because if we were able to verify their PID, then our PID written in Rust, should have also been verifiable, and they had already laid the groundwork as far as coding went. They had implemented their state space model of the system as well as their PID controller in Lustre, and set the stability condition as a tilt angle of less than 0.1 radians for the time interval 71 to 99. There are two aspects of the verification we explored: verification of the controller, and verification of the controller's interaction with the physical system.

Verification of Controller

Verification of the controller involved verifying the property of the controller's output never being greater than the maximum amount allowed by the microcontroller. For the previous team's PID, this was a value of 1000. This property was verified, but this is trivial, due to the implementation of a sub-node (a function that the controller calls) which sets the controller's output to the maximum allowed by the microcontroller. While it is good to verify that the controller's output will stay within the bounds of what the microcontroller can output to the motors, it doesn't say much about the stability of the whole system. In order to prove the system stable, the controller must be combined with the model of the system.

Initially this caused the verification to fail, due to the lack of specification regarding the time intervals before 71, or after 99. In order to get around this, we needed to include the specification that the count variable was either less than 71, between 71 and 99 with the tilt angle under a certain value, or greater than 99.

After adjusting the logical expression to include the intervals before 71 and after 99, the verification did not immediately fail. However, now the program appeared to stall and never complete either verifying or proving the property to be variant. We attempted to solve this issue by using the timeout option for Kind 2. By employing the "--timeout_wallclock <number of seconds>" option, we were able to stop the verification after a specific number of seconds, in order to see how far into the verification we had reached. This was helpful to prove that the verification was working up to whatever point it stopped, but it did little to solve the greater problem of complete verification.

Initially, only one computer was doing the verification processing: a 2015 Macbook Air with a 2.2 GHz Intel Core i7 and 8 GB of DDR3 RAM. When the processing was moved to an AMD RYZEN 7 2700 8-Core 3.2 GHz with 16 GB DDR4 RAM, verification that the previous system wouldn't complete was able to finish, and we found that the system was invariant only up to 97 steps.

After further refining the system's boundaries and conditions by limiting the initial tilt angle to 0° (an idea gleaned from a paper written about TWIP controller verification using different software [7]) and the number of iterations of the system to 100 (equivalent to number of loops multiplied by the sampling time, in this case $100 \times 1/100 \sim 1$ second), and using only two of the SMT-solver engines (Bounded Model Checking and IC3), we achieved verification of the property of stability. This stable condition specified that the iteration count was greater than 0 and the angle of tilt was less than $\sim 12^\circ$. This proves that verification is feasible given some adjustments of both expectations and implementation of the controller.

At this point we began bumping up against the project's deadlines and so were not able to further explore how we might begin to alter the verification specifications or coding in order to provide a more complete verification.

Next Steps

There are a few main avenues of proceeding in order to alleviate the issues we encountered in the verification step:

- Define the specifications for what is considered stable. One misstep we made was never rigorously defining what would be considered stable, and this made it difficult to define the property we wanted to check in Kind 2. Does the robot need to be able to come back from being completely on its side, due to some disturbance or are we only considering the system without disturbances? What initial conditions are allowed (tilt angle)? How long does the system need to maintain stability? These are all questions that need to be answered before the code can be written with enough specificity that the verification can be trusted.

- Rewrite the Lustre code so there is only one node and the rest of the code is in function form. The Kind 2 documentation notes “Non-linear arithmetic has a huge impact [] on the performances of the underlying SMT solvers” and while the system model of the robot is linear, the controller node is not, as it includes sub-nodes that introduce non-linearity in the input variable. The difficulty in this approach involves figuring out how to specify things like the accumulation node and the backwards difference node without using the operators that Kind 2 does not support in functions, which include all of the operators that refer to a previous state of the function. If successful, this has the potential to reduce processing time significantly, but may be more trouble to implement than the savings in processing time. The Kind 2 documentation has a section on functions that would be a good starting place [8].
- Use better hardware, or outsource the processing to a service like AWS. This may work, but it is hard to tell if the issues we encountered are due to a lack of processing power, or some inherent problem with the way the verification code is written.

Testing

A test plan was written, taking into account the limitations on testing due to restricted access to the labs at PSU. This test plan is included in the appendices. Testing was done mostly on the physical robot to verify that changes made to the system did not introduce new unknown issues. The first set of tests involved comparing the PWM output of the old ESP32 processor to the new MKR1010 processor.

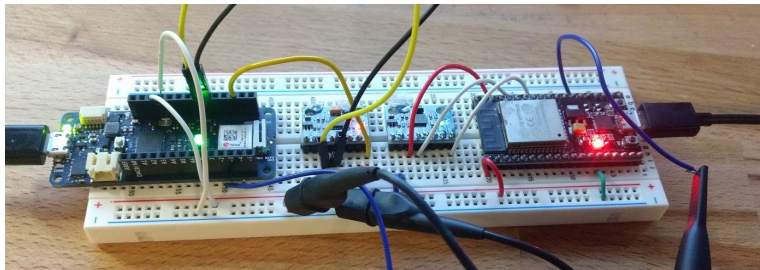


Figure 11: The test setup for the two controller boards.

The PWM outputs were compared using identical inputs, and the outputs were found to be pretty much exactly the same:



Figure 12: The PWM outputs of the two boards for the same inputs.

After building the new robot this same test was performed on the output of the Adafruit motor shield board. This time the outputs were not equal, and that was mostly because the adafruit board is limited to an output of 1600 Hz while the original code runs at 30Khz. The Duty cycles were otherwise the same, so this issue was mostly ignored.



Figure 13: The outputs of the Adafruit motor shield board with both processors.

To help confirm that the code that was being generated by Rust was valid, the PID controller that was written in Rust was also compared with Brett Beauregard C++ controller. Both the generated Rust controller and the C++ controller had identical inputs and PID constants. The robot was physically required to balance from various locations. The response between the Rust PID and the C++ PID seemed nearly identical. Both controllers were easily able to recover from both extremes of the robot inputs.

A similar test was used to compare the Fuzzy logic controller to the PID controller. Both controllers were compared from various input locations.

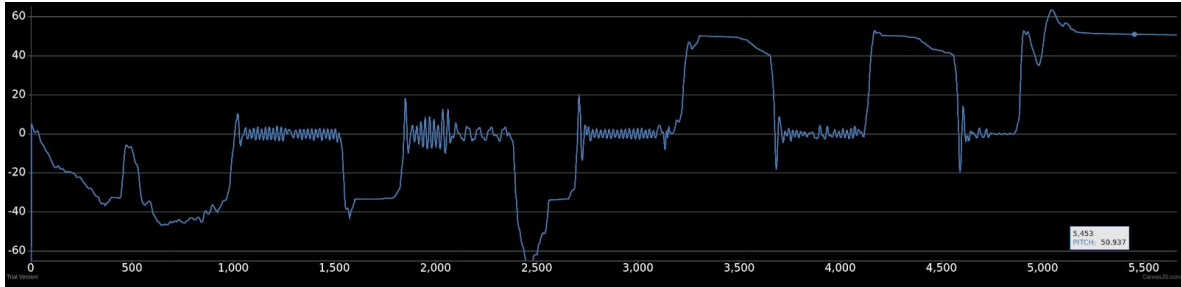


Figure 14: PID recovery from extremes

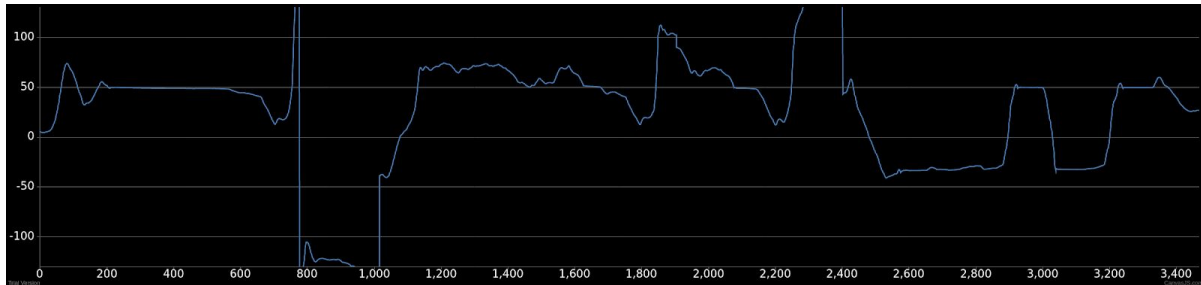
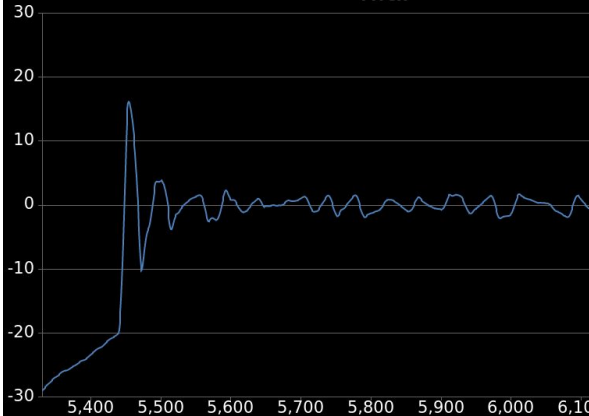
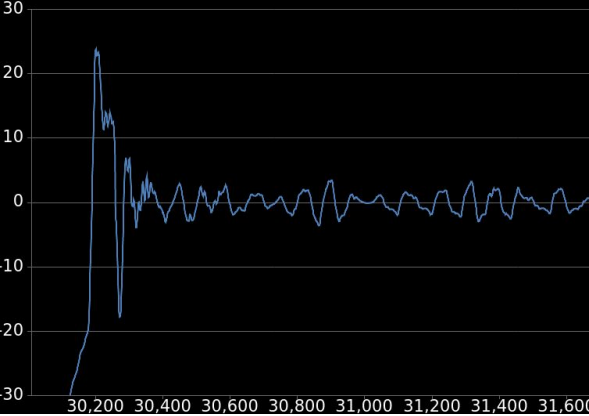


Figure 15: Fuzzy controller recovery from extremes

The graphs above show the controller's ability to balance from an extreme. The PID controller is able to go from an extreme of ~60 degrees to its balance point where it slightly oscillates up and down. The fuzzy logic controller was never able to hit the balance point, and usually overshoot the setpoints significantly. Overall the PID controller was slightly more capable than the fuzzy controller, but the difference was often negligible. Comparing the two controllers is not entirely fair in this situation because a strong influence on the performance of the controller is tuning. The PID controller was tuned fairly extensively, while the fuzzy controller has significantly more parameters to modify and was not as extensively tuned.

The graphs above were generated by the website running on apache2. The website was also verified by comparing the serial data output from Arduino IDE to the data recorded in MySQL. There was no packet loss and the data was within 1% error of accuracy.

	
<p><u>Rust PID Controller</u></p> <p>Start from -20.18°, overshoot: 15.85°</p> <p>Oscillations between -1.85° to 1.65°</p> <p>Settling time to 10% of 20°: 0.82 sec.</p> <p>Reached 5% of 180° in 0.64 sec.</p> <p>Qualitative: PID consistently was able to balance itself from a starting point of 20+ degrees away from setpoint. The robot could nearly balance itself from the ground with the PID.</p>	<p><u>Rust Fuzzy Controller</u></p> <p>Start from -20.18°, overshoot: 23.71°</p> <p>Oscillations between -3.0° to 3.3°</p> <p>Settling time to 10% of 20°: Never reached within 10%;</p> <p>Reached 5% of 180° in 1.79 sec.</p> <p>Qualitative: The robot had great difficulty balancing from 20 degrees away. The majority of the time the robot overshoot the setpoint.</p>

(Data is presented at 100 samples per second.)

Deliverables

Below is a list of all of our hardware and software deliverables for this project:

Hardware

- 2 robots and related supplies (charger, arduino boards, etc.)

Software

- Implemented Rust PID
- Implemented Rust Fuzzy Logic Controller
- Modified Kind2 Embedded Rust Binary & LustreToRust.ml
- MATLAB implemented controller

Conclusion

The main achievements of this project include the development of a fuzzy logic controller, and the comparison of its performance with the PID controller through simulation and real-world testing, the expansion of the existing code of the inverted pendulum robot to include the Rust control library, modification Lustre/Kind 2 Rust code generator to generate "embedded friendly" Rust, which can be imported directly onto the microcontroller. This generator was successfully used for both the PID as well as the fuzzy logic controller.

We successfully used the upgraded parts provided by Galois to build another robot, and transplanted the code of the previous team to the new board. The controller code was isolated from the overall embedded code and implemented in Lustre, where it would be possible to use Kind 2 to verify it. While the verification requirement was not completed, certain roadblocks that would stymie successive teams have been identified and possible solutions addressed.

All of this was accomplished despite the limitations imposed by the COVID-19 virus, and the importance of flexibility and creativity were underscored. While there are still many avenues to be explored and aspects of this project that need to be completed and improved on, this year's team feels like a solid foothold was created.

We recommend that following teams pursue these avenues to continue the project:

- Port all embedded code into Rust so as to eliminate the issues caused by the interfacing of Rust and C
- Implement arrays into the Rust generator from Kind 2
- Continue developing a suite of non-linear controllers, with a recommendation towards a Non-Minimal State Space controller
- Characterize the new robot's dynamics
- Continue attempting to verify the controllers, with the possibility of using a computer cluster to overcome some of the time constraints inherent in verifying a complex system

The progress made in this project is a step in the direction of a more standardized process of implementing high-assurance control systems and we hope that following teams are able to continue the work.

References

- [1] Z. Li, C. Yang, and L. Fan, *Advanced control of wheeled inverted pendulum systems*. London: Springer, 2013.
- [2] L. Reznik, *Fuzzy controllers handbook how to design them, how they work*. Oxford: Newnes, 1997.
- [3] GitHub. 2020. *Artem1199/Otis-Arduino*. [online] Available at: <https://github.com/Artem1199/otis-arduino/tree/6c62d9b780cc5ca6abff5f2492f3ba903a413212> [Accessed 11 June 2020].
- [4] GitHub. 2020. *Artem1199/Otis-Server*. [online] Available at: <https://github.com/Artem1199/otis-server/tree/d4b680f804dd0024e1f270156c189b10fb3cbe98> [Accessed 11 June 2020].
- [5] GitHub. 2020. *Artem1199/Otis-Tcp*. [online] Available at: <https://github.com/Artem1199/otis-tcp/tree/26e13c5c7a3a1544f5fac511028aa15508fe81b3> [Accessed 11 June 2020].
- [6] Beauregard, B., 2020. *Br3ttb/Arduino-PID-Library*. [online] GitHub. Available at: <https://github.com/br3ttb/Arduino-PID-Library> [Accessed 11 June 2020].
- [7] H. Gul, J. Ahmad, F. Gul and M. Ilyas, "Modeling and formal verification of inverted pendulum based two-wheeled transportation vehicle," 2012 Proceedings of SICE Annual Conference (SICE), Akita, 2012, pp. 113-118.
- [8] "Kind 2" *Kind 2*. [Online]. Available: https://kind.cs.uiowa.edu/kind2_user_doc/home.html. [Accessed: 11-Jun-2020].
- [9] Baier, C., & Katoen, J. P. (2008). Principles of model checking. MIT press.
- [10] Doc.rust-lang.org. 2020. *The Rust Programming Language - The Rust Programming Language*. [online] Available at: <<https://doc.rust-lang.org/book/>> [Accessed 12 June 2020].
- [11] Rust-embedded.github.io. 2020. *Introduction - The Embedded Rust Book*. [online] Available at: <https://rust-embedded.github.io/book/> [Accessed 12 June 2020].

[12]S3.amazonaws.com. 2020. *More Complex Objects - The (Unofficial) Rust FFI Guide*. [online] Available at: <https://s3.amazonaws.com/temp.michaelfbryan.com/objects/index.html> [Accessed 12 June 2020].

Appendix A - Test Plan

ECE 412

High Assurance Controller of Self-balancing Robot: Test Plan

Authors:	Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang				Group # 10
Github	https://github.com/Artem1199/ECE-HACoSR2/tree/85f0d0fe3be21060c7b8cf96b7f15a4fa912cec2				
Version #:	1.0	2.0			
Date:	10-Apr-2020	27-May-2020			

1.0 Introduction

The test plan was developed to document and track the essential information required to soundly define methods for testing the two wheeled inverted pendulum robot (TWIP) and ensure that it functions correctly based on the specifications defined in PDS. The team broke down the test planning based on the component modules, applied unit testing, and integration tests. The intended audience for the test plan is the project team, student peers, and professors.

2.0 References

2.1 Documents

Product Design Specification	Version 1	February 18, 2020
https://github.com/Artem1199/ECE-HACoSR2/blob/master/Docs/PDS/2020_PSU_Capstone_10_PDS.pdf		

Galois Project Proposal		
https://github.com/GaloisInc/HighAssuranceControllerOfSelfBalancingRobotCapstone/blob/master/ECE_Casptone_High_assurance_controller_II.pdf		

3.0 Resources

3.1 Equipment

- I Oscilloscope
- I Digital Multimeter
- I Soldering Iron
- I Protractor

3.2 Project Specific Hardware

- I Arduino MKR WiFi 1010.
- I Arduino ESP-32

- | TWIP (Robot built by previous team)
- | TWIP charging unit.

3.3 Software

- | Arduino IDE (1.8.10) and related drivers
- | MATLAB/Simulink
- | Lustre

4.0 Objectives

The objective of this document is to test the two-wheeled inverted pendulums (TWIP) against all of our requirements. This will be done by evaluating modules in our system and performing some of the types of tests listed below.

4.1 Unit Test

Multiple individual units will have to be tested using various equipment to verify the functionality of individual modules.

4.2 Integration Test

Integration tests involve combining multiple units and testing them together to verify that the modules will interact as expected before the whole system is entirely closed up inside the box.

4.3 Functionality Test

Functionality testing will be used to confirm basic functionality, without strict parametric tests. The modules should be able to turn on and switch between states after certain inputs.

4.4 Stress Test

Stress testing will verify the stability and reliability of the system to determine robustness.

4.5 Parametric Test

Parametric testing will verify that the observed data is distributed according to our design parameters.

4.6 Acceptance Test

The purpose of this test is to evaluate the system's compliance with the design requirements and assess whether it is acceptable or not.

5.0 System Tests

5.1 Unit Tests

5.1.1: PWM Accuracy test (Test Id:RT-UT-01)

PDS Related Requirement: change the "otis_arduino" code so it works with Arduino MKR WiFi 1010.

Test Writer:Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)						
Test Case Name:		PWM Accuracy			Test ID #:	RT-UT-01
Description:		Compare PWM expected output consistency of old board and new board.			Type:	black box
Tester Information						
Name of Tester:		Artem			Date:	13 April
Hardware Version:		1.0			Time:	1:00 AM
Setup:		ESP board and MKR1010 boards with PWM output code.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Setup MKR1010 Cortex m0+ boards and old ESP board with identical PWM control code and test vectors.	Code should compile and upload.	X			

2	Create varying PWM inputs. Measure PWM output.	N/A	X			Created input by mapping MPU6050 input data to PWM output.
3	Measure PWM at 0% duty cycle	Frequency, and duty cycle of both boards should match within 5%.	X			0% error.
4	Measure PWM at 25% duty cycle	Frequency, and duty cycle of both boards should match within 5%.	X			<1% error on output.
5	Measure PWM at 50%	Frequency, and duty cycle of both boards should match within 5%.	X			<1% error on output.
6	Measure PWM at 75%	Frequency, and duty cycle of both boards should match within 5%.	X			<1% error on output.
7	Measure PWM at 100%	Frequency, and duty cycle of both boards should match within 5%.	X			<1% error on output.
Overall test result:			X			See images in the same folder for more info.

5.1.2: PWM Response Rate (Test Id:RT-UT-02)

PDS Related Requirement: change the “otis_arduino” code so it works with Arduino MKR WiFi 1010.

Test Writer:Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)						
Test Case Name:		PWM Response			Test ID #:	RT-UT-02
Description:		Compare PID controller output on MKR1010 and ESP processor boards and response time comparison.			Type:	black box
Tester Information						
Name of Tester:		Artem			Date:	14 April 2020
Hardware Version:		1.0			Time:	2:38AM
Setup:		ESP boards connected to MPU6050 with oscilloscope measuring PWM outputs.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Assemble ESP boards on breadboard with MPU6050 communication. MKR1010 reuse current processor on B.B.	ESP: establish communication w/ I2C, MKR1010: N/A.	X			N/A

2	Upload current working TWIP code with "PID.h" library for both ESP and MKR1010 boards. Match setpoints, and P,I,D constants, and SampleTimes on both boards.	Code should compile and upload.	X			Modified outputSum += ki*error to outputSum = ki*error. MKR limit set to 255, then mapped magnitude to 1k, ESP limit set to 255.
3	Physically move MPU6050 to setpoint tilt value.	Controller outputs should be within 5%.	X			<1% error on output.
4	Tilt MPU6050 on MKR1010 robot to one extreme of robot tilt, record value, match value on ESP-32. Record PID output. Set an oscilloscope to measure PWM output continuously.	Controller outputs should be within 5%. Compare PWM output change response time. Response time should be within 10%.	X			<1% error on output. Loop process time for both boards is within 1%.

5	Tilt MPU6050 on MKR1010 robot to opposite extreme of robot tilt, record value, match value on ESP. Record PID output. Set an oscilloscope to measure PWM output continuously.	Controller outputs should be within 5%. Compare PWM output change response time. Response time should be within 10%.	X			<1% error on output. Loop process time for both boards is within 1%.
Overall test result:			X			PASSED.

5.1.3: PWM Response Rate (Test Id:RT-UT-03)

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)						
Test Case Name:		Rust PID library			Test ID #:	RT-UT-03
Description:		Compare C++ library and Rust library.			Type:	black box
Tester Information						
Name of Tester:		Artem			Date:	25 May 2020
Hardware Version:		2.0			Time:	
Setup:		MKR1010 board with Rust PID library generated by Lustre and C++ PID libraries.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments

1	Modify code to implement both Rust and C++ PID libraries. Upload to MPU6050.	Code should upload to MKR1010.	X			
2	Physically move MPU6050 to setpoint tilt value.	Output of C++ PID and MKR1010 PID libraries should be identical.	X			
3	Tilt MPU6050 on MKR1010 robot to one extreme of robot tilt.	Output of C++ PID and MKR1010 PID libraries should be identical.	X			
4	Tilt MPU6050 on MKR1010 robot to opposite extreme of robot tilt.	Output of C++ PID and MKR1010 PID libraries should be identical.	X			When finding the setpoint, the outputs of the serial are slightly different. i.e. Serial says we are 185 deg from setpoint while tcp says we are -175 degree away. This is technically correct either way.
Overall test result:			X			

5.1.4: TCP Data Accuracy (Test Id:RT-UT-04)

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)			
Test Case Name:	Rust PID library	Test ID #:	RT-UT-04

Description:		Compare serial data output to tcp data output.			Type:	black box
Tester Information						
Name of Tester:		Artem			Date:	May 26 th , 2020
Hardware Version:		2.0			Time:	
Setup:		MKR1010 board with Rust PID library generated by Lustre and C++ PID libraries.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Modify code to serial print data out data at the same time as sending it over tcp.	Could should compile.	X			
2	Move the robot through the ranges of motion for pitch.	N/A	X			
3	Compare results from database to tcp output over 30 second period.	Serial data and TCP data should be within 2% error value wise. And less than 2% missed data points.	X			At around 180 degrees, one output will print ~185 while the other prints ~-175. Technically both are correct.
Overall test result:			X			PASS

5.1.5: Adafruit motor shield PWM accuracy (Test Id:RT-UT-05)

Test Writer:Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)						
Test Case Name:					Test ID #:	RT-UT-05
Description:		Compare the PWM output to the motors			Type:	black box
Tester Information						
Name of Tester:		Artem			Date:	27 May 2020
Hardware Version:		2.0 and 1.0			Time:	4:00PM
Setup:		ESP boards connected to MPU6050 with oscilloscope measuring PWM outputs.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Program PWM output through arduino using methods in test case 5.1.2. Setup PWM output for adafruit shield.	Code should compile and upload.	X			
2	Setup oscilloscope to measure output of both MKR1010 PWM and adafruit shield.	N/A	X			
3	Physically move MPU6050 to setpoint tilt value.	Controller outputs should be within 5%.	X			Duty cycle is within 5%. Frequency is not.

4	Tilt MPU6050 on MKR1010 robot to one extreme of robot tilt, record value, match value on ESP-32. Record PID output. Set an oscilloscope to measure PWM output continuously.	Controller outputs should be within 5%. Compare PWM output.	X			With a 55/255 input, duty cycle in the same for both at ~23% within 1% error. PWM duty cycles for both old and new robot are identical. Frequencies are not, this is a limitation of the adafruit motor board.
5	Tilt MPU6050 on MKR1010 robot to opposite extreme of robot tilt, record value, match value on ESP. Record PID output. Set an oscilloscope to measure PWM output continuously.	Controller outputs should be within 5%. Compare PWM output.	X			With a 215/255 input, duty cycle in the same for both at ~83% within 1% error. PWM duty cycles for both old and new robot are identical. Frequencies are not, this is a limitation of the adafruit motor board.
Overall test result:			X			PASSED.

5.2 Integration Test

5.2.1 Tilt Controller test (Test Id: RT-IT-01)

PDS Related Requirement: Develop and verify a controller with a wider range of stable input conditions and compare its performance with the PID controller through both simulation and in the real system.

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)						
Test Case Name:		Tilt controller		Test ID #:		RT-IT-01
Description:		The tilt controller is the heart of the self balancing robot. It is tested to ensure that the control input (output of the controller) is as expected for several different inputs. There are 2 different tilt controllers: PID and Fuzzy Mamdani. The same type of test is conducted for both.		Type:		white box
Tester Information						
Name of Tester:				Date:		
Hardware Version:		1.0		Time:		
Setup:		Disconnect the tilt control input from the robot and probe this point with an oscilloscope.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Disconnect control input from robot and attach an oscilloscope probe to the control input.	Will see a PWM signal generated on the screen of the oscilloscope.				
2	Move the device to a known tilt angle.	A PWM of different duty cycle should appear on the oscilloscope.				

3	Translate the control signal to a motor torque given the manufacturers data.	The torque from the motors is the same as simulated in Simulink.				No valid simulink model to compare
Overall test result:						

5.2.2 Yaw Controller Test (Test Id: RT-IT-02)

PDS Related Requirement: Develop and verify a controller with a wider range of stable input conditions and compare its performance with the PID controller through both simulation and in the real system.

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)			
Test Case Name:	Yaw controller	Test ID #:	RT-IT-02
Description:	The yaw controller allows the orientation of the robot to remain at a certain angle. It is tested to ensure that the control input (output of the controller) is as expected for several different inputs. The yaw control system uses PD control.	Type:	white box
Tester Information			
Name of Tester:		Date:	
Hardware Version:	1.0	Time:	
Setup:	Disconnect the yaw control input from the robot and probe this point with an oscilloscope.		

Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Disconnect control input from robot and attach an oscilloscope probe to the control input.	Will see a PWM signal generated on the screen of the oscilloscope.				
2	Move the device to a known yaw angle.	A PWM of different duty cycle should appear on the oscilloscope.				
3	Translate the control signal to a motor torque given the manufacturers data.	The torque from the motors is the same as simulated in Simulink.				No valid simulink model to compare
Overall test result:						

5.3 Functionality Test

5.3.1 Physical Robot Balance performance C++ vs Kind2 PID (Test Id: RT-FT-01)

PDS Related Requirement 1: Extend the existing code for the inverted pendulum robot to include a Rust control library

PDS Related Requirement 2: Modify Lustre/Kind2 Rust code generator to generate embedded friendly Rust, which can be directly imported in your library.

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)			
Test Case Name:	PID C++ vs Rust	Test ID #:	RT-FT-01

Description:		Test to compare performance of PIDs implemented in C++ and Rust. Functionality should be near identical	Type:		white box	
Tester Information						
Name of Tester:		Artem		Date:		27 May 2020
Hardware Version:		2.0		Time:		5:00PM
Setup:		No external measurement tools are needed. Just set the robot upright and turn it on.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Rust and C++ PID Hold the robot such that it has a tilt angle of approximately zero.	The robot is held upright.	X			Robot has no issue balancing at the center.
2	For Rust and C++ Turn the robot on and release it.	The robot may jerk around, but ultimately remains upright.	X			Robot has no issue balancing at the center.
3	With the C++ PID library enabled installed. Move the robot to both sizes of the tilt angle, and find the offset before the robot cannot recover. Record this offset.	N/A			X	PID , recovers from +/-40 degrees from offset.

4	With the Rust PID library enabled installed. Move the robot to both sizes of the tilt angle, and find the offset before the robot cannot recover.	The Rust PID point of no return should be within 5% of the C++ PID point of no return.	X			Rust PID is able to recover from +/- 40 degrees from offset. Can't any further offset because of robot 's physical body.
5	Pick robot off table. Hold robot by hand. See if robot is able to balance itself while in hand	Rust PID and C++ PID should have identical responses.	X			Robot will bounce between -30 to +30 degrees, but is able to balance itself the majority of the time.
Overall test result:			X			Rust PID and C++ PID are functionally very similar.

5.3.2 Physical Robot Balance performance Rust PID vs Rust Fuzzy Cont. (Test Id: RT-FT-02)

PDS Related Requirement 1: Extend the existing code for the inverted pendulum robot to include a Rust control library

PDS Related Requirement 2: Modify Lustre/Kind2 Rust code generator to generate embedded friendly Rust, which can be directly imported in your library.

PDS Related Requirement 6: Develop and verify a controller with a wider range of stable input conditions and compare its performance with the PID controller through simulation and in the real system.

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)			
Test Case Name:	PID Rust vs Fuzzy Rust	Test ID #:	RT-FT-02

Description:		Compare the Rust Fuzzy controller for tilt input and Rust PID controller for various inputs.			Type:	white box
Tester Information						
Name of Tester:		Artem			Date:	27 May 2020
Hardware Version:		2.0			Time:	7:00PM
Setup:		No external measurement tools are needed. Just set the robot upright and turn it on.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	For the Fuzzy logic controller hold the robot such that it has a tilt angle of approximately zero.	The robot is held upright.	X			Robot has no issue balancing at center.
2	For the fuzzy logic controlled robot, turn the robot on and release it.	The robot may jerk around, but ultimately remains upright.	X			Robot has no issue balancing at center.
3	With the Rust PID library enabled. Move the robot to both sizes of the tilt angle, and find the offset before the robot cannot recover. Record this offset.	N/A			X	PID , recovers from +/-40 degrees from offset.

4	With the Rust Fuzzy logic controller enabled. Move the robot to both sizes of the tilt angle, and find the offset before the robot cannot recover.	The Fuzzy logic controller should have a better response than the PID controller.			X	Fuzzy logic controller is unable to recover past about 15 degree offset from the setpoint.
5	Pick robot off table. Hold robot by hand. See if robot is able to balance itself while in hand	The fuzzy logic controller should have a better response than the PID controller.			X	Fuzzy logic controller doesn't have nearly as good performance.
Overall test result:					X	Fuzzy logic controller isn't as good as the PID. Potentially can get better with better tuning.

5.4 Stress Test

5.4.1 Test 1: IMU and Web server measurement test (Test Id: RT-ST-01)

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)			
Test Case Name:	PWM Accuracy	Test ID #:	RT-ST-01
Description:	Communication and PID calculation robustness.	Type:	black box
Tester Information			

Name of Tester:		Artem			Date:	27 May 2020
Hardware Version:		2.0			Time:	7:35PM
Setup:		MKR1010 board with TCP communication enabled and Rust Heap memory based PID.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Program MKR1010 with Wifi TCP enabled and Rust heap allocated PID.	Code should compile, Arduino should connect to wifi and start writing data.			X	
2	Run the program for 2 hours. Monitor data output to the server. After 8 hour verify data is still live by moving the robot.	MKR1010 should consistently output live data to the server. MKR1010 should not hang up losing communication with MPU6050 or with a web server.		X		Start at 7:38PM, no issues. 7:43PM, no issues. 7:50PM, no issues. 8:10PM, connection lost. Attempt 2: 20:41PM to 20:49PM runs no issues until a connection loss. Still NO hangup issues.
Overall test result:				X		Connection issues even without serial data transfer. But still a significant improvement.

5.4.2 Test 2: IMU and Serial data measurement test (Test Id: RT-ST-02)

Test Writer: Team 10(Ignacio Mejia-Rodriguez, Artem Kulakevich, Andrew Forsman, Yuqi Wang)

Test Case Name:		PWM Accuracy			Test ID #:	RT-ST-02
Description:		Communication and PID calculation robustness.			Type:	black box
Tester Information						
Name of Tester:		Artem			Date:	
Hardware Version:		2.0			Time:	
Setup:		MKR1010 setup to stream MPU6050 data through the serial port. Leave MKR1010 running for a long duration.				
Step	Action	Expected Result	Pass	Fail	N/A	Comments
1	Program MKR1010 with Wifi disabled, and serial data enabled.	Code should compile, Arduino should connect to wifi and start writing data.			X	
2	Run the program for 2 hours. Monitor data output to the server. After 8 hour verify data is still live by moving the robot.	MKR1010 should consistently output live data. MKR1010 should not hang up losing communication with MPU6050.		X		Attempt 1: Hangup occurs within less than 1 minutes. Attempt 2: Hang up in less than 3 minutes. Attempt 3: Hang up in less than 2 minutes.
Overall test result:				X		Serial.print data transfer is pretty much unusable in the current software. This seems to be a common issue for this board.

5.5 Parametric Test

Verifying that our controller performs as simulated is a very important part of this project. This is a task in progress and is more detailed than what can be included in the tabular form. Separate documents will be written specifically for the verification of the controller.

5.6 Acceptance Test

It is important to note that this project is research based rather than focused on creating a product. The system's acceptance is based on the ability to establish formal verification for the TWIP. In other words, the basic functionality of the robot allows us to have a complex system that we can formally verify.

5.6.1: PWM Response Rate (Test Id:RT-AT-02)

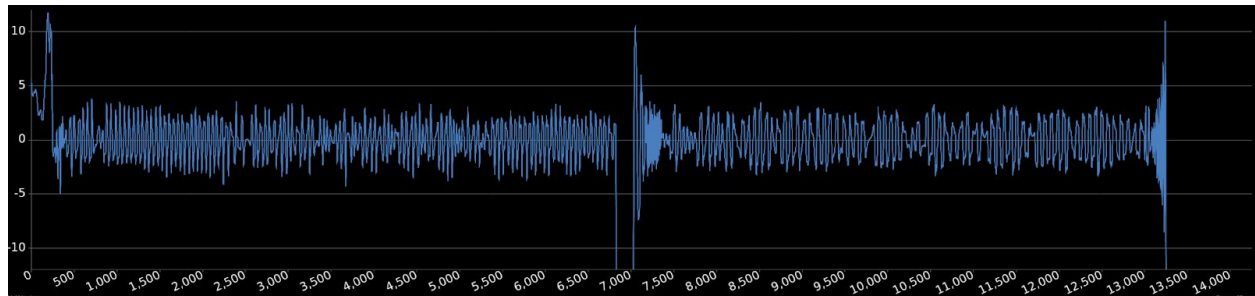


Figure 1: Tilt data for Fuzzy-I controller on the left, PID controller on the right after impact in the center.

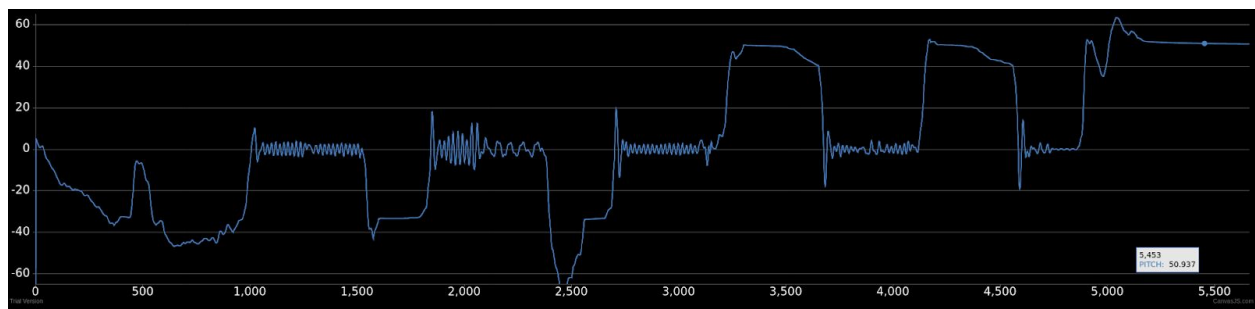


Figure 2: PID controller recovering from laying on the floor poth in positive and negative.

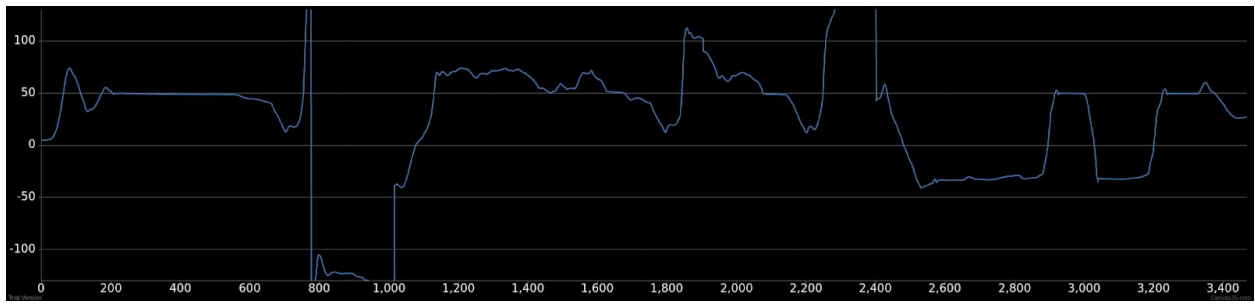


Figure 3: Fuzzy controller failing to recover from laying down position. Usually overshooting the other end.

Appendix B - KIND2 Modifications

Here is a list of the general modifications I made:

1. Commented out `clap_and_run()` function, there's no point to run this as a separate program in embedded.
2. Commented out references to `std`
3. Commented out all there cmd window interfaces, i.e. `InputReader`
4. Commented out all `Result` types, Removed `OK()` returns, and Removed `Err`
5. Entirely replaced `read_init` (old `read_init` is still generated, but another function is used)
6. Entirely replaced `read_next` (ditto above)
 1. Critical change here, I modified the input to a mutable reference, this way I could keep things FFI compliant with C++. I believe the original functionality had it return a new object to replace the old one instead of modifying the same object each time.
 2. This means that I do not return the next state anymore, instead I override the values in the current structure. **This is an area where I can see potential functional errors.** (See lines 241 - 244) So far though the controllers I have generated are working properly.
7. Commented out `run()`
8. Commented out `pub mod parse ()` function, used for command window, not needed for functionality.
9. Commented `arity()`, again cmd window.

Changes I did not make:

1. I did not deal with assertions, this is potential because of my ignorance of Kind2, but I did not see how this could be useful in embedded Rust. That means if your lustre code has assertions, the generated Rust code **will not** work properly.