High Assurance Controller of a Self-Balancing Robot

Team 26

Patrick Gmerek
Amanda Voegtlin
Justin Patterson
Ethan Lew

Table of Contents

Background and Requirements

Introduction

Cyber-physical systems are increasingly being used to automate and enhance aspects of daily life. With the rise of the embedded systems, devices like robots and intelligent networks are being used to replace human labor in factories. Further, robots are being installed in offices and homes, where they can guard, care, service, explore, and entertain inhabitants. The level of intelligence and articulation promise to continue to bring immense profit to society. That is, of course, if the robots are reliable and malfunction rarely. With the high level of complexity in robots comes the *curse of dimensionality*---the amount of states that a robot can exhibit is so large that errors are hard to find exhaustively because they are so rare. This emergent complexity demands that verification must take up more of the design process; this has been observed to be a large bottleneck, with up to 80% of the overall design costs being spent on verification.

This Capstone explores the design, implementation, and verification of a two-wheeled inverted pendulum (TWIP). Though not immensely complex, the robot system contains enough instability to make it a suitable sandbox for controller design and verification. The TWIP is a human-like robot that stands upright above two wheels. This upright posture causes it to want to fall over, making it a form of an inverted pendulum. In order to maintain its posture, a controller actuates motors at the wheels to counteract the falling movement. This gives the robot an innately clumsy motion as it stabilizes. The robot is considered to be *underactuated* as it has fewer actuators than degrees of freedom. The problem then is to design a controller that can stabilize this underactuated system about its equilibrium point.

In addition, the inverted pendulum system is notably nonlinear. Nonlinearity can greatly complicate controller design and verification. Linear control theory does not necessarily apply with regards to stability. For example, gain and phase margins cannot readily be attributed to the closed loop system. Of course, *linearization* is a popular technique to deal with designing controllers for nonlinear plants. This technique proves fruitful for the robot chosen as it is approximately linear about its equilibrium point. Popular linear controller techniques like PID, LQR and H∞ are shown to have suitably large regions of attraction.

Nonlinearity still complicates the problem of verification as all of the nonlinear dynamics must be incorporated, in some way, into the evaluation of stability. An exploration of general *dynamical theory* offers insight into how characteristics of the robot can be extracted, including stability. Equations can be constructed that describe the dynamics and kinematics of the TWIP. These equations compose a *mathematical model* that contains both *functions* and *parameters*. Functions are capable of evolving in time whereas parameters are time invariant. For example, consider the ever-popular pendulum model: the angular position and rate are functions of time

but the gravitational acceleration and the length from the center of mass are unchanging parameters.

Dynamical systems permit a system to start at an initial condition; in the case of the pendulum, it would have an initial angular position and rate. Then, the system evolves and starts to sway back and forth, assuming a *trajectory* in its *phase space*. Notably, the system is attracted to a fixed point when the angular rate is slow enough. It is repelled from an equilibrium point when it is swung fast enough. This idea, *attraction* and *repulsion*, starts to characterize stability to some degree. If a point is near an attractor, it is likely to be stable. Of course, notable exceptions occur, and those exceptions will be explored. Further, the concept of a *limit cycle*--that a system converges or diverges from a single, periodic path--turns out to be a critical aspect to the robot considered.

These equations produce *simulations* of the TWIP system when an *ordinary differential equation solver* is used to approximate the state evolution. Simulations are critical to gain intuition of the controlled system as well as approximate bounds of the worst case scenario that still produces an upright robot. Additionally, the simulation can be used as a diagnostic when the validation stage fails, providing insight on where errors occur in the validation pipeline.

The language of continuous time evolution does not readily present itself to be run on a computer. Being continuous both in time and in state, a computer would be incapable of testing every trajectory in finite time. So, another mathematical framework is required to translate the continuous dynamical behavior into a form that can be verified in reasonable time. This problem can be addressed by *symbolic dynamics*, which encodes collections of trajectories into a finite quantity of admissible words. Each word corresponds to a region of state space, and connections can be drawn between words that corresponds to the evolution of trajectories in the phase space. Effectively, this encoding *discretizes* the phase space into a cyclical directed graph.

Importantly, a technique for verification must be employed to prove or disprove that the robot built matches its specification. This project utilizes the *model checking* methodology to verify that a system satisfies its requirements against all acceptable stimuli. Model checking involves producing a model and a specification that the model should satisfy. The previously mentioned dynamical system can serve as a model to be verified. The specification, however, requires additional formalism.

An extension to computation tree logic (CTL) is utilized to rigorously specify the system requirements. This logic language is desirable because it adds path quantification and temporal operation to conventional propositional logic. Accordingly, existential and universal behavior can be described as well a discrete time dependent behavior. This permits general model requirements like *safety*, *reachability*, *liveness* and *fairness* to be readily defined. As mentioned, an extended version of CTL is used as conventional branching logic has no concept of

continuous state variables. Inequalities with real numbers translate real values subspaces to boolean values, comprising a language dubbed CTL-A (CTL for Analog).

Integration to existing verification tools speeds up the design cycle immensely. The satisfiability modulo theorem (SMT) model checker Kind2 is used in the project. As its name suggests, Kind2 is a K-induction based verification tool that is a successor to the original Kind SMT checker. It features both compositional and modular reasoning as a way of verifying discrete finite automata. Supporting real values, it is capable of verifying real valued data flow in addition to boolean equations. It is not, however, capable of working with transcendental functions; the linearization and discretization techniques outlined previously must be used. Notably, Kind2 is capable of generating implementations of its nodes in languages that can run in an embedded environment. Hence, it is possible that verified systems can be *transpiled* and run on the robot itself, having a high guarantee of operating as expected.

Deploying verified code in an embedded environment segues into the last aspect of the TWIP project: building a working robot. Care must be taken to select parts and distribute the mass in ways that don't deviate too greatly from the model used. For example, the robot assumes symmetric distribution of weight on both wheels and so placing too many components on one side can adversely affect the high assurance aspects of the project. Fortunately, good controller design can add hefty margins to these decisions as the robot will have improved robustness.

The ideal TWIP model uses torque as its control input, but the input perceived by the embedded system is a pulse-width modulation value that corresponds to root-mean-square (RMS) voltage. Accordingly, a model that accurately describes the build requires a model for the brushed DC motors. The model, fortunately, is linear except for the motor's *deadzone region*. The linear control law is modified to compensate for this motor imperfection with little change to the rest of the design.

Critically, accurate sensing is required in order to create feedback; reading orientation of the robot is not a trivial process. The system responsible for calculating the robot's orientation in time is known as an Attitude Heading and Reference System (AHRS). Orientation representation is a challenge in itself, as a hierarchical composition of *Euler angles* can result in *gimbal lock*. In order to overcome this constraint, algorithms utilize *quaternion* orientation representation. The mathematics of hypercomplex number systems needs to be formulated and related to the AHRS problem.

Additionally, an IMU needs to be selected and filtered properly. The AHRS system chosen uses Magnetic, Angular Rate and Gravity (MARG) measurements to determine orientation. The accelerometer, gyroscope and magnetometer used all have considerable noise that requires heavy filtering. The IMU needs to be able to reject noise with as little delay in the feedback loop as possible. Traditional FIR/IIR filter design is shown to be highly inefficient due to the delay incurred as the signal propagates through the tap values. *Sensor fusion* is used to

characterize the noise using a variety of sources and predict whether a change is value is actually the orientation changing. First, a *complementary filter* was implemented to fuse the accelerometer and gyroscope values. Next, sensor covariance is incorporated via a *kalman filter* (KF) and *extended kalman filter* (EKF) to improve the sensor accuracy. Lastly, other techniques like the *Madgwick* and *Mahony* filter were tested.

Formal Requirements

Our task was to implement a controller for a self-balancing robot, and formally verify the controller. Creating a robot that balanced itself was just the proof-of-concept piece of this task, though it did form a large portion of the work.

Our formal requirements were worked out over the course of the project with our industry sponsors at Galois Inc, and the critical portions were these:

- Be well-documented with a repeatable design process, especially with respect to hardware
- Be formally verifiable
- Be an inverted pendulum robot which balances on two wheels unassisted
- Implement PID controller and document accuracy/performance characteristics
- Have all components designed and implemented with verification in mind
- If a second controller is implemented, we must be able to measure its improvement over a “simple PID” controller
- It will utilize the embedded Rust programming language for its principle control program
- Investigate different model checkers such as Kind2 or Alloy
- Output telemetry data to enable debugging

Mathematical Preliminaries

A mathematical body of work called dynamical system theory provides a framework with which to model the robot’s change in time. A *dynamical system* is a group of functions, $\{f^t\}_{t\in\mathbb{R}}$ that describe the *flow* of a point in some geometrical space, meaning that $f^t(x_l) \subset \mathbb{R}^{\times}$. Importantly, such systems are not restricted to the field $\mathbb{R}$, but a system $\{f^n\}_{n\in\mathbb{N}}$ can occur in discrete time; in this case the system described a *cascade* of a point changing in some geometrical space. In the case of modeling a this robotic system, this geometric space is often referred to as the state space or phase space. The elements of a dynamical system can be functions that describe solutions to a system of ordinary differential equations. In discrete time, the elements of a dynamical systems can be functions that describe solutions to an *iterated map* or a system *difference equations*.

It is helpful to think of dynamical behavior this generally because models can be translated into forms that permit themselves to be verified more readily. If the model was described by differential equations, it would be easy to solve for specific *orbits*. Studying a system at this level is known as *local analysis*. Extracting *global behavior* would be much more challenging with a trajectory or orbit solver. For example, stability verification will require the solution to attractor and regions of attraction. Further, verification of controllability/observability is a global phenomenon that is of interest. The concept of *symbolic encoding* can overcome this limitation and is motivated by dynamical system theory. It is helpful, then, to explore the relevant theory needed to utilize global analyses.

As soon as the system is identified as dynamical, a wealth of properties and tools become available for analysis. This section will briefly summarize such properties. A dynamical system is said to be *linear* if the evolution of its state vector equals a constant matrix A multiplied by the state vector itself.

$$\dot{x} = Ax$$

Likewise, a system is said to be *nonlinear* is the change is dependent not on a weighted sum of first power state variables. A highly relevant nonlinear system is the differential equation describing pendulum with no energy dissipation,

$$\ddot{\theta} + \frac{g}{L}\sin(\theta) = 0.$$

This is equivalent to

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = -\frac{g}{L}\sin(y_1). \end{cases}$$

where y_1 describes the angular position θ and y_2 describes the angular rate $\dot{\theta}$. What has been gained by *converting* the system in this form? The system's *phase space* is made more evident by this representation, being a phase describing pairs of angular position and rate. The flow of the system can be shown by superposing a vector field on this plane; the orbits are the integral curves produced by tracing the evolution from an initial condition.

When the changes of both angular position (velocity) and angular rate (acceleration) are zero, meaning that $\dot{y}_1 = 0, \dot{y}_2 = 0$, the system has no desire to change position in phase space as the magnitude of the vector field at that point is zero. These points are called *critical points* of the system, and can characterize the long term behavior of the system. That is, a system has the ability to be *attracted* and *repelled* from critical points, or sometimes both depending on its position from the critical point. Going back to the pendulum example, the critical points are:

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = 0 \text{ iff } \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \in \left\{ \begin{bmatrix} k\pi \\ 0 \end{bmatrix} : k \in \mathbb{Z} \right\}$$

The *Hartman-Grobman theorem* justifies that one can use a locally linearized system to analyze the behavior of a nonlinear system about its equilibrium point. This theorem is critical later on when justifying why a linear model approximation can be used to optimally configure a controller about the uncontrolled robot's unstable equilibrium point. Consequently, a linearized system can be derived by taking the *Jacobian* of the system with respect to the state variables and then substituting the state variables with the values at equilibria,

$$\frac{\partial F}{\partial y} = \begin{bmatrix} \frac{\partial F_1}{\partial y_1} & \frac{\partial F_1}{\partial y_2} \\ \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{L}\cos(y_1) & 0 \end{bmatrix}$$

Substituting,

$$\left.\frac{\partial F}{\partial y}\right|_{y_1=0} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{L} & 0 \end{bmatrix}$$

$$\left.\frac{\partial F}{\partial y}\right|_{y_1=\pi} = \begin{bmatrix} 0 & 1 \\ \frac{g}{L} & 0 \end{bmatrix}$$

Observe that at $y_1 = 0$, the eigenvalues are $\{j\sqrt{Lg}/L, -jLg/L\}$ and are entirely imaginary. This implies that points immediately around this critical point periodically orbit about this point with constant amplitude. This matches intuition as no energy is lost in the system, so the pendulum will keep oscillating forever. On the other hand, consider $y_1 = \pi$. The eigenvalues are $\{\sqrt{Lg}/L, -\sqrt{Lg}/L\}$. These values are entirely real, and of opposite sign. This means that this equilibrium point is hyperbolic, having both an attraction and repulsion aspect. Hyperbolic critical points are of considerable interest as they have the ability to result in interesting dynamics, and in higher dimensions, can create *chaotic behavior*. This point is when the pendulum is considered inverted, and will be the point at which a controller will attempt to stabilize the robotic system.

Now consider the *damped pendulum system*. It can be defined as

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = -\frac{g}{L}\sin(y_1) - by_2. \end{cases}$$

This means that the system is resisting movement proportional to the angular rate of the pendulum. Using the same process as before, the eigenvalues reveal a system *bifurcation*. Depending on the values of b, the system could have real or complex eigenvalues. Hence, there exists a fundamental change in the system's behavior at that point. Note that the hyperbolic point remains hyperbolic.

Next, consider the conservative properties of the system. A system is considered to be conservative if

$$\nabla \cdot F = \left(\frac{\partial F_1}{\partial x_1} + \frac{\partial F_2}{\partial x_2} + ... + \frac{\partial F_n}{\partial x_n} \right) = 0.$$

From this definition, it is clear that the original pendulum system is conservative while the dissipated one is not. This is intuitive when considering the energy properties of the physical system. However, some of the implications are worth considering. Consider the image of a covering in the phase space C_0,

$$F(C_0) = \{F(x) : x \in C_0\},$$

it is clear that the area of the image will be the same as the original covering if the system is conservative. This is not the case with a non-conservative, where there exists images that don't have a common area. Critically, controllers designed for the robot will make the system dissipative, affecting the difficulty of verifying the system's phase space.

Modeling

Every aspect of the TWIP design and implementation rests on an accurate model. The simulator, model checker and even controller utilize some insight gained through modeling. This model needs to describe the kinematic behavior, or the behavior of how the parts move in relation to one another. For example, it is necessary to know how the wheel speeds corresponds to the system's velocities. Dynamic behavior, or how the system responds to applied forces, also needs to be described. The applied forces to the robot's wheels will be the primary output of any controller design. The model should be expansive enough to cover critical parameters, but not too complex so as to make it impossible to evaluate symbolically or numerically. Further, it would be ideal if the model was hierarchical, permitting some of the dynamics to be separated and incorporated at different stages during the project's lifetime.

The robot is considered to be a *chain* of *rigid bodies* which rotate at a *joint*. A description of all possible movements of a robot is contained in its configuration space. The smallest number of real-valued coordinates is the degrees of freedom (DOF) of the system. The TWIP considered has four degrees of freedom: an (x, y) position on a plane, a heading angle θ and a tilt angle α. The space described by the cartesian product of these coordinates is the *configuration space*, or *C-space*. It is critical to note that the shape of the C-space is independent of the coordinate system chosen; all representations are *topologically equivalent* to one another.

Kinematics

Kinematics describes how changes in the Cartesian space of the system changes the joint spaces of the system. The robotic system is able to convert angular motion of the wheels into angular and linear motion of the platform and payload. For the model, the system will be considered to be a chain of rigid bodies. This restriction permits using Newton's laws of point mass as approximation of the bodies' motions. Any robot built using these models in mind will need to have minimal backlash and friction forces acting on it.

Wheeled robotics can either be classified as *omnidirectional* or *nonholonomic*. Omnidirectional wheeled robots have no unifying constraints applied to them, while nonholonomic ones possess a single *Pfaffian constraint*. The TWIP is nonholonomic, which simplifies the kinematic derivation of the robot. Physically speaking, this means that the wheels can only roll forward, incapable of slipping side to side. Mathematically, consider a set of generalized coordinates for a rigid-body chassis robot, q, then there exists a transformation such that

$$A(q)\dot{q} = 0.$$

This is another way of saying that the *configuration space* of the robot is equally dimensional to the euclidean space that it is described. No transformation acting on the coordinates themselves evaluate to zero, but one acting of the rate of change of the coordinates can. This can be seen intuitively by considering a directional wheel rolling on a plane. The configuration space of the robot is described by the (x, y) coordinates that the contact point is positioned at, the steering angle of the wheel θ and the angular orientation of the wheel ϕ. The configuration space then is $\mathbb{R} \times \mathbb{R} \times T \times T$, being the cartesian product of a plane and a torus.

First, a coordinate system should be defined to start constructing equations for the kinematics analysis. Using *generalized coordinates*, the quantities are $(x_0, y_0, \theta, \theta_l, \theta_r)$. The coordinates (x_0, y_0) are the location of the system on the plane of locomotion (ground). θ is the yaw of the system, or heading angle, being the direction orthogonal to the axis of the wheels and aligned with the ground. (θ_l, θ_r) are the angular positions of the left and right wheels, whose time derivatives are the angular rate of the wheels.

The system is assumed to only be able to move forwards and backwards relative to the orientation of the wheels. Accordingly, side-to-side slipping is not allowable. Accordingly, the linear velocity of the system has no components aligned with the wheels,

$$\dot{y}_0 \cos(\theta) - \dot{x}_0 \sin(\theta) = 0.$$

A track positions two wheels laterally below the payload; the track width of the system affects how fast the difference of the speed of wheels changes the yaw of the system, or heading angle. Thus, in the absence of linear motion,

$$\begin{cases} r\dot{\theta}_r = \frac{d}{2}\dot{\theta} \\ r\dot{\theta}_l = -\frac{d}{2}\dot{\theta} \end{cases}$$

And, being an expression of linear motion already, the linear velocity of the entire system can be added,

$$\begin{cases} r\dot{\theta}_r = \frac{d}{2}\dot{\theta} + \dot{x}_0\cos(\theta) + \dot{y}_0\sin(\theta) \\ r\dot{\theta}_l = -\frac{d}{2}\dot{\theta} + \dot{x}_0\cos(\theta) + \dot{y}_0\sin(\theta) \end{cases}$$

The kinematics, then, can be expressed as

$$\begin{bmatrix} \dot{x}_0 \\ \dot{y}_0 \\ \dot{\theta} \\ \dot{\theta}_r \\ \dot{\theta}_l \end{bmatrix} = \begin{bmatrix} \frac{r\cos(\theta)}{2} & \frac{r\cos(\theta)}{2} \\ \frac{r\sin(\theta)}{2} & \frac{r\sin(\theta)}{2} \\ \frac{r}{d} & \frac{r}{d} \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{\theta}_r \\ \dot{\theta}_l \end{bmatrix}$$

This model describes how the angular rate of the wheels changes the location and heading angle of the system. Note that it does not include tilt, meaning the robot can be fallen down and still hold these kinematics. However, as soon will be shown, it would be convenient to relate this motion to the dynamical system's state space. Angular position of the wheels are not in the state space, but the angular rate of the yaw and linear velocity is. Consider the relation,

$$\begin{bmatrix} \dot{\theta}_r \\ \dot{\theta}_l \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & \frac{1}{d} \\ \frac{1}{r} & -\frac{1}{d} \end{bmatrix} \begin{bmatrix} v \\ \dot{\theta} \end{bmatrix},$$

meaning that,

$$\begin{bmatrix} \dot{x}_0 \\ \dot{y}_0 \\ \dot{\theta} \\ \dot{\theta}_r \\ \dot{\theta}_l \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \\ \frac{1}{r} & \frac{1}{d} \\ \frac{1}{r} & -\frac{1}{d} \end{bmatrix} \begin{bmatrix} v \\ \dot{\theta} \end{bmatrix}.$$

This analysis relates how changes in the system's state space also change the motion of rigid bodies of the entire chain system. Dynamics analysis, next, will show how the quantities v and $\dot{\theta}$ evolve in response to the actuation from the motors.

Motor Modeling

Figure 1 shows a lumped element circuit of a conventional DC motor. The windings create a highly inductive load with resistance due to the size of the wires used. Notably, an additional voltage source should be considered to account for the back electromotive force (EMF) that opposes the input V_{RMS}. This is from the coupling between the coils and motor armature, causing the motor to act like a generator when disturbed. The system can be described by the governing equation,

$$V_{RMS} = I_a R_a + L_a \dot{I}_a + \mathcal{E}$$

It is desired to relate the input voltage V_{RMS} to the torque applied to the armature. It can be assumed that current is directly proportional to torque and that the back EMF is directly proportional to angular rate. This leads to the differential system,

$$\begin{cases} V_{RMS} = I_a R_a + L_a \dot{I}_a + K_e \omega \\ \dot{\omega} = \frac{K_m I_a}{J} - \frac{b\omega}{J} \end{cases}$$

This yields the linear state space representation

$$\begin{bmatrix} \dot{I}_a \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} -R_a/L_a & -K_e/L_a \\ K_m/J & -b/J \end{bmatrix} \begin{bmatrix} I_a \\ \omega \end{bmatrix} + \begin{bmatrix} 1/L_a \\ 0 \end{bmatrix} V_{RMS}$$

$$\tau = \begin{bmatrix} K_m & 0 \end{bmatrix} \begin{bmatrix} I_a \\ \omega \end{bmatrix} + 0$$

and the transfer function

$$H(s) = \frac{K_m J s + b K_m}{J L_a s^2 + (J R_a + b L_a) s + K_m K_e}$$

When addressing the control of the motor in discrete time, it is helpful to have a discrete time version of the dynamics. So, an analogous system on the z-plane with sampling period T is

$$H(z) = \frac{(K_m b T^2 + J K_m T) z^2 + 2 K_m T^2 b z + (K_m b T^2 - J K_m T)}{(J L_a + J R_a T + L_a T b + K_e K_m T^2) z^2 + (2 K_e K_m T^2 - 2 J L_a) z + (J L_a - J R_a T - L_a T b + K_e K_m T^2)}$$

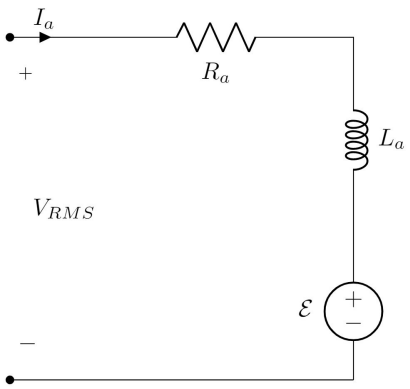

Figure 1: A motor can be modeled as a series RL circuit with an additional variable voltage source to account for the back EMF.

Verifying The Controller

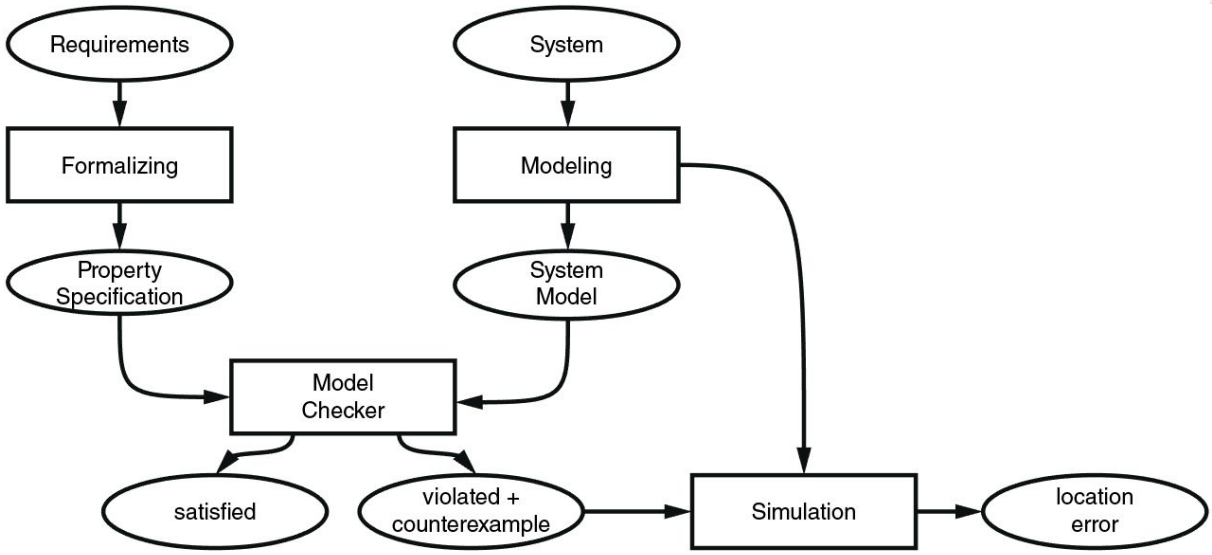

Figure 2: Verification plays a central role in the development pipeline. The model checker is not only capable of discovering when a design is correct but can also help find and diagnose errors.

Even though the controller designs enabled the TWIP system to remain upright, no analysis was performed to determine how stable the system is. All previous work exploited some aspects of the system's dynamics in order to improve response; the approaches themselves failed to yield guarantees of the system's overall stability. Accordingly, the next step required is formally verifying whether the TWIP meets a desired specification. Digital systems

have a long history with formal verification techniques---their binary nature lends themselves well to closing a chain of evidence between possible inputs and desired system behavior.

Dynamical, and also analog, systems have numerous complications when comparing response behavior to a specification. Firstly, the continuous time domain restricts all possible inputs from being explored in finite time. Secondly, high variability can exist in the components, making it difficult to verify a system without extensive measurements. Lastly, the models used in the verification process have inaccuracies, so uncertainty can originate from the model itself. There exists an implicit assumption that the system model is sufficiently accurate in the validation of its controllers. This assumption invites additional verification via inverse modeling post-build: the model's behavior and actual behavior should be compared.

System modeling and controller design provides insight on how well the two-wheeled inverted pendulum robot can perform under initial conditions. It does not completely describe the system's stability, however. A prominent formal verification technique called *model checking* can determine if a controller under consideration meets some desired specification. Typical verification properties are

- Safety - Does instability even exist?
- Liveness - Can the system stabilize from an initial condition?
- Fairness - Under what conditions can stability occur repeatedly?
- Real-time - Does stability occur fast enough?

These properties address the stability problem in different ways. Safety is a condition that finds if any failure can occur in the system after a finite amount of time---the robot has fallen over. Finding that unsafe conditions exist, liveness seeks to find the conditions where successful stabilization occurs. Fairness considers the total environment that the system is operating in; if something in the environment causes the robot to fail stabilization, a fairness condition implies that the controller is executing infinitely often. Lastly, the real-time specification seeks whether or not a desired outcome meets some time deadline.

These properties are useful to specify because they are logically well-defined. A commonly applied language in model checking is CTL (Computation Tree Logic). The language supports *path quantifiers* and *temporal operators*, capable of describing conditions for behavioral path and time dependence. Traditionally, CTL is generated by the following grammar,

$$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi)$$
$$\mid \mathrm{AX}\phi \mid \mathrm{EX}\phi \mid \mathrm{AF}\phi \mid \mathrm{EF}\phi \mid \mathrm{AG}\phi \mid \mathrm{EG}\phi \mid \mathrm{A}[\phi \mathrm{U} \phi] \mid \mathrm{E}[\phi \mathrm{U} \phi]$$

This language is required to act on a model defined by transitions to and from logical states. This requires the definition of a *Labeled Transition System (LTS)*. A LTS $T = (Q, Q_0, \Sigma, R)$ consists of

- A set of states Q
- A set of initial states Q_0
- A set of events Σ
- A state transition relation $R \subseteq Q \times \Sigma \times Q$

Note that CTL is a branching tree logic, meaning that the future is not fully specified by the present. This language contrasts with LTL (Linear Time Logic) which has a single path temporally. This is a critical feature for verifying dynamical systems, whose phase space can be mapped to branching behavior even if the system is itself deterministic. Further, some desired continuous time behavior is undefined in conventional CTL. A new extended language, called CTL-A, incorporates boolean evaluation of the real values encountered in analog and dynamical systems. It is defined as

$$v, z, \phi ::= z \geq v | z \leq v | \perp | \top | p | (\neg\phi) | (\phi \wedge \phi) | (\phi \vee \phi) | (\phi \Rightarrow \phi) | (\phi \Leftrightarrow \phi) \\ | \mathrm{AX}\phi | \mathrm{EX}\phi | \mathrm{AF}\phi | \mathrm{EF}\phi | \mathrm{AG}\phi | \mathrm{EG}\phi | \mathrm{A}[\phi \mathrm{U} \phi] | \mathrm{E}[\phi \mathrm{U} \phi] | A^{-1}$$

where z is a continuous state variable and v is a constant real value. The inequality operators are associated with *half-planes*, whose combinations define arbitrary Manhattan polytopes in the system phase space. For example, consider the CTL-A equations,

$$\begin{cases} \Phi & = (x_1 > 2) \wedge (x_1 < 3) \wedge (x_2 > 1) \wedge (x_1 < 3) \\ \Psi & = \mathrm{EF}(\Phi) \end{cases}$$

This can be read as finding a region in state space whose trajectories have the ability to eventually flow into Φ, being a rectangle with two vertices $(2, 1), (3, 3)$. Using this language, the model checking properties can be described,

- Safety - $AG(\Phi)$ "All paths always satisfy Φ "
- Liveness - $AG(\Phi \Rightarrow AF(\Psi))$ - "From all paths satisfying Φ, it always follows eventually that Ψ is satisfied "
- Fairness - $AG(AF(\Phi))$ - "All paths always satisfy that all paths eventually satisfy Φ"

Provided that Φ adequately describes a stable region for a dynamical system, CTL-A provides a useful framework for verifying stability.

In practice, evaluating these requirements in CTL is challenging as the paths are continuous in nature and uncountably infinite. Traditionally, available model checking tools are equipped to evaluate the path of *discrete finite automata* and don't accept the ODE solvers and transcendental functions required to calculate a continuous time dynamical system path. *System discretization*, of course, can address one of these problems. The phase space itself, though, still needs to be discretized in some way. One of the more common techniques to address this issue is known as *cell mapping*. It is defined and discussed in the following section.

Often a collection of values in the state space are attributed to a logical state. In the case of the TWIP, when the robot's tilt angle and tilt angular rate are nearly zero, the robot can be considered *stabilized*. Otherwise the robot can either be *stabilizing* or *unstable*---the difference being that one is able to transition to a *stabilized* state while the other not. The margins between *stabilized* and *stabilizing* are arbitrary, specified by the behavioral description. Of course, the controller design attempts to minimize both the *stabilized* and *unstable* regions, while maximizing the *stabilizing* region. Importantly, it would unreasonable to expect arbitrary convergence time, and so the state transitions must occur within a desired time interval. Good controller designs are able to converge quickly while still possessing the acceptable region sizes mentioned.

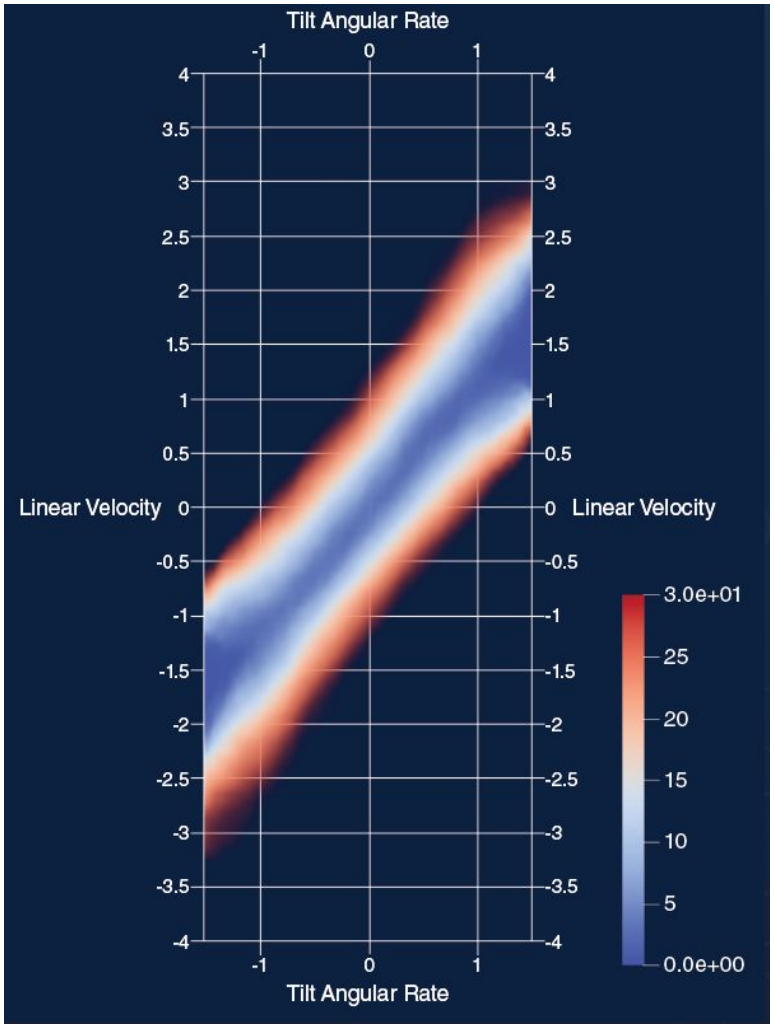

Figure 3: The stability convergence time between linear velocity and angular rate. Note that the robot has enhanced stability when the tilt angular rate and velocity are similar, able to converge quickly to the stable region.

A region of attraction can be found for the robot by evaluating the liveness conditions of a polytope which describes a stable area in phase space. The benefit of this approach is that transients can be described as well, meaning that some areas can be specified as taking too long to converge.

Cell Mapping

It has been seen that the process realized by a dynamical system is produced from its initial conditions. In general, initial conditions near one another share a similar fate. However, this assumption can be violated if areas of the system exhibit rich dynamical behavior. This is the case for a pendulum near its unstable equilibrium; the point is a hyperbolic fixed point and the orbits around it are *homoclinic*. In 1935, the mathematician Garrett Birkhoff applied

symbolic dynamics near homoclinic orbits and showed that it was capable of describing the behavior around this fixed point effectively.

The demand to characterize homoclinic orbits was heightened by the interest to describe *chaotic systems*, a class of determinate systems that exhibit high irregularity. These systems are so sensitive to changes in initial conditions that any *local analysis*--analysis involving looking at individual orbits--is ineffective. Many arguments have been made whether the chaotic systems are really a valid classification as they are absolutely deterministic like any other conventional dynamical system. The distinction seems only to be the accuracy required in the initial conditions to produce a meaningful trajectory. Nonetheless, as measurements made in the engineering and scientific disciplines have noise present, so the distinction is useful in practice. Accordingly, a collection of *global analysis* methods was developed to characterize chaotic systems. While the robot system being designed is far from chaotic, the global methods are useful for stability verification.

Unsurprisingly, *symbolic dynamics* looks similar to the transition systems previously defined: the discretized dynamical system is related to a sequence of abstract symbols corresponding to system state. The transitions are created from the shift operation that results in the system evolution. The construction of the symbolic image is performed by partitioning the phase space in a collection of finite *covers* that relate to a symbol. The objective, then, is to partition a space in such a way that the global dynamical properties are made evident from inspection of the symbolic transition system.

A function in a dynamical system acts *smoothly* on a compact manifold M in state space or $f : M \to M$. This topological notion justifies the concept of *cell-mapping*. The manifold M permits itself to be partitioned into finitely many other closed manifolds M_i, being part of a collection C,

$$C = \{M_1, M_2, ..., M_n\}.$$

Given the previous condition of f, it is clear that there exists an *image* of M_i, or $f(M_i)$. For M_i, there also exists a subset of C that contains $f(M_i)$, i.e.,

$$C(i) = \{M_j : M_j \cap f(M_i) \neq \varnothing\}.$$

Hence, a *directed graph* G can be created to describe the symbolic image of f with respect to the covering C. This construction affords one the ability to use graph techniques to resolve various aspects of a dynamical system. The following relations between the original system and the new symbolic image holds:

- trajectories agree with the admissible paths on the graph
- the entire symbolic image reflects the global structure of the dynamical system itself

- the number of coverings control the accuracy of the symbolic system's approximation of the dynamical one

The smooth property of the system being analyzed implies that the graph analysis will converge to accurate system dynamics as the number of cells increase; that is, *subdivision* of the state space allows the graph adjacency to correspond to the orbits realized in the system. Of course, this claim requires rigorous justification--another section will serve to formalize these concepts.

Lastly, it should be noted that the symbolic dynamics analysis is capable of addressing a number of problems commonly encountered in system analysis. Example ones include

- Construction of Periodic Orbits
- Construction of attractors and regions of attraction
- Estimation of Lyapunov exponents
- Verification of controllability

Each of these problems are relevant for the design and verification of the TWIP.

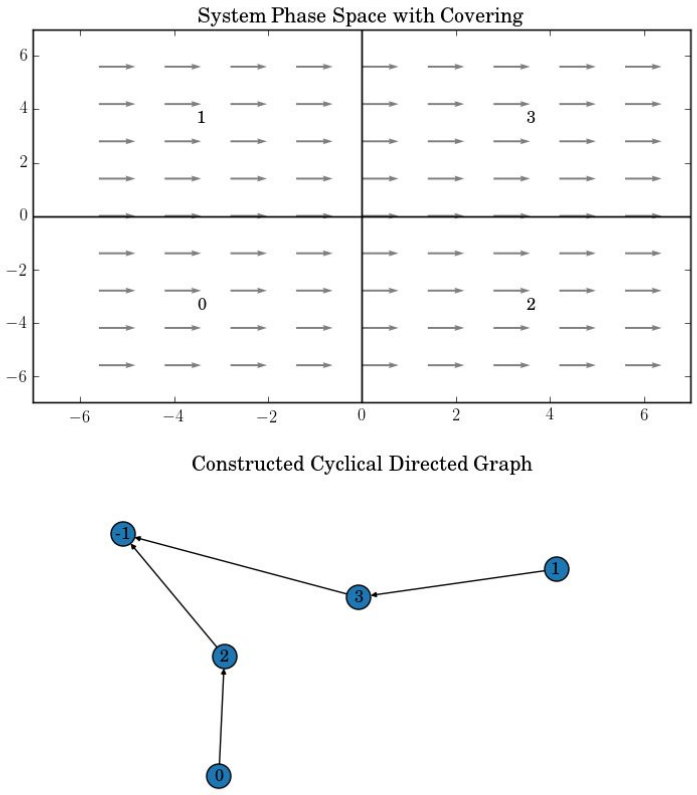

Figure 4: A dynamical system's phase space and symbolic encoding

Cell mapping was used to in figure 4 construct a graph that describes the evolution of covering in a simple state space. Here, it is possible to see that the image of the "1" covering evolves into

the “3” covering. For long time behavior, it is clear that all flows leave the compact subspace because “-1”, being a region outside, is the only absorbing node of the graph. If another node, or cyclical group of nodes, were found, an attractor could be inferred to exist in those regions.

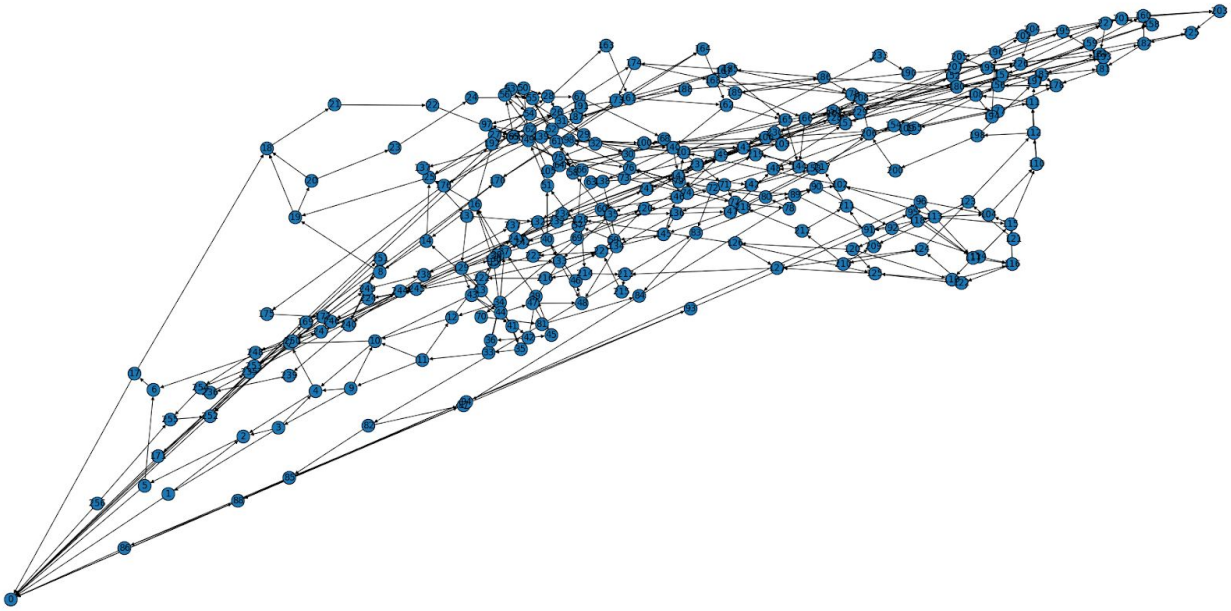

Figure 5: Symbolic encoding of the TWIP system’s phase space

The graph in figure 5 shows the Markov partition used on the TWIP’s phase space. It is clear that many nodes leave the area being partitioned, being unstable for this analysis. However, cycles can be found using techniques like Tarjan’s algorithm that relate to attraction of the dynamical system. This means that finding the basin of attraction becomes a graph reachability problem when using symbolic encoding.

Design

Our initial design proposed these functional blocks:

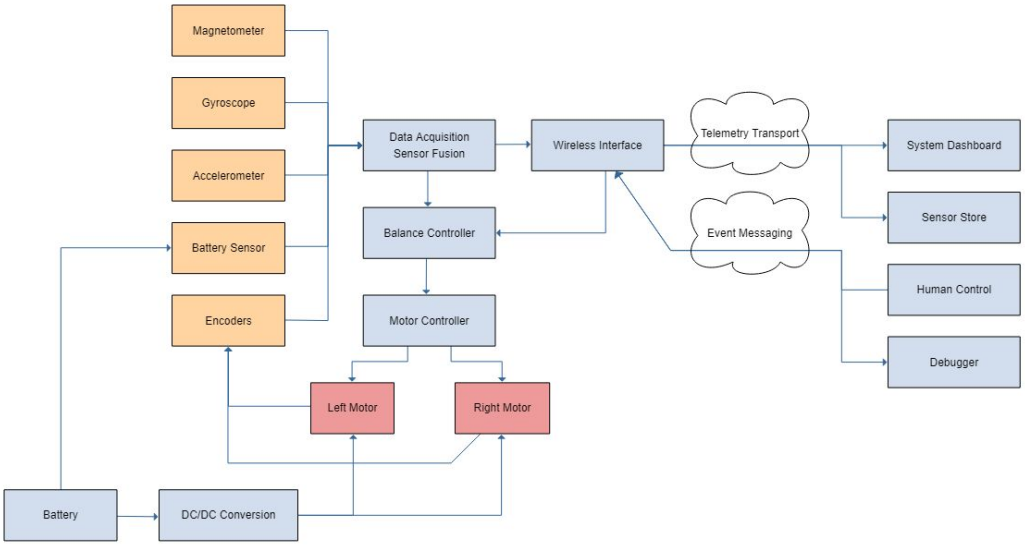

Figure 6: The functional blocks of the robot build

Picking a Control System: the PID Controller

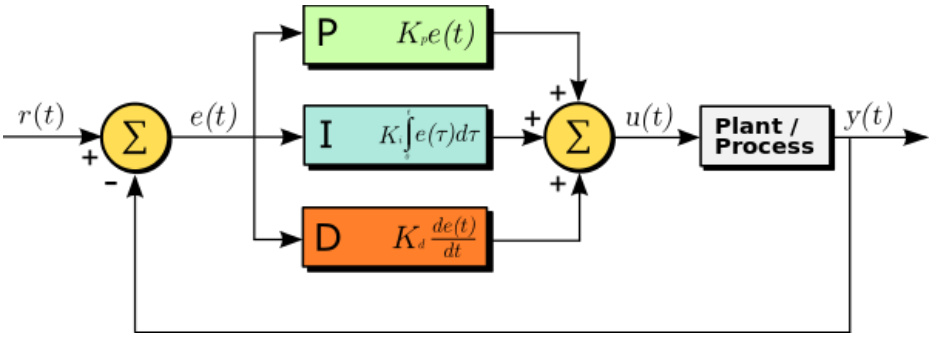

Figure 7: A basic PID controller diagram. Source: Wikipedia, May 2019

A proportional–integral–derivative controller (PID controller, or three-term controller) is a control loop feedback mechanism widely used in systems that require continuously modulated control. A famous example is the thermostat, which has to turn on a heater or air conditioner based on the difference between a detected temperature and a desired temperature, without overshooting the target. A PID controller accepts the error value *e(t)* as the difference between a desired setpoint (in our case, equilibrium) and a measured process variable (in our case, pitch

or yaw) and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively), hence the name. The proportional term is a response to the current error, the integral term corrects for the sum of previous errors (to prevent drift in response), and the derivative term is an anticipatory correction-- since it's the derivative term, it is measuring how swiftly the error is growing or shrinking, so it can engage a stronger response to a rapidly growing error. In the case of a hot room it will crank the air conditioner up high, or in our case, in response to a fall it will exert strong corrective torque on the wheels.

The basic PID controller is tuned by adjusting the constant terms in P, I, and D, and a basic version is little more than an equation, but there are additional improvements that can be made in programming one that account for common output issues. The PID.rs library in Appendix A1 incorporates these changes, but we will summarize the problems we addressed here.

Controller Design Process: Overcoming Basic Issues

Maximum Outputs

The PID controller is just a calculator, it doesn't automatically know about the physical limitations of the hardware it's in. Sometimes it will try to output an impossible number and the motors will saturate. The solution was to program in boundary values, based on the hardware that will use the output; in our case, limits on motor torque.

Derivative Kick

This is a thing that happens when the setpoint (The new balance point) of the controller changes (as it often does in a moving robot), causing an overreaction in the motor output to fix. In a straightforward implementation, the derivative of the error is going to be equal to the negative derivative of the input, except when the setpoint of the robot is changing.

$$dError/dt = dSetpoint/dt - dInput/dt$$

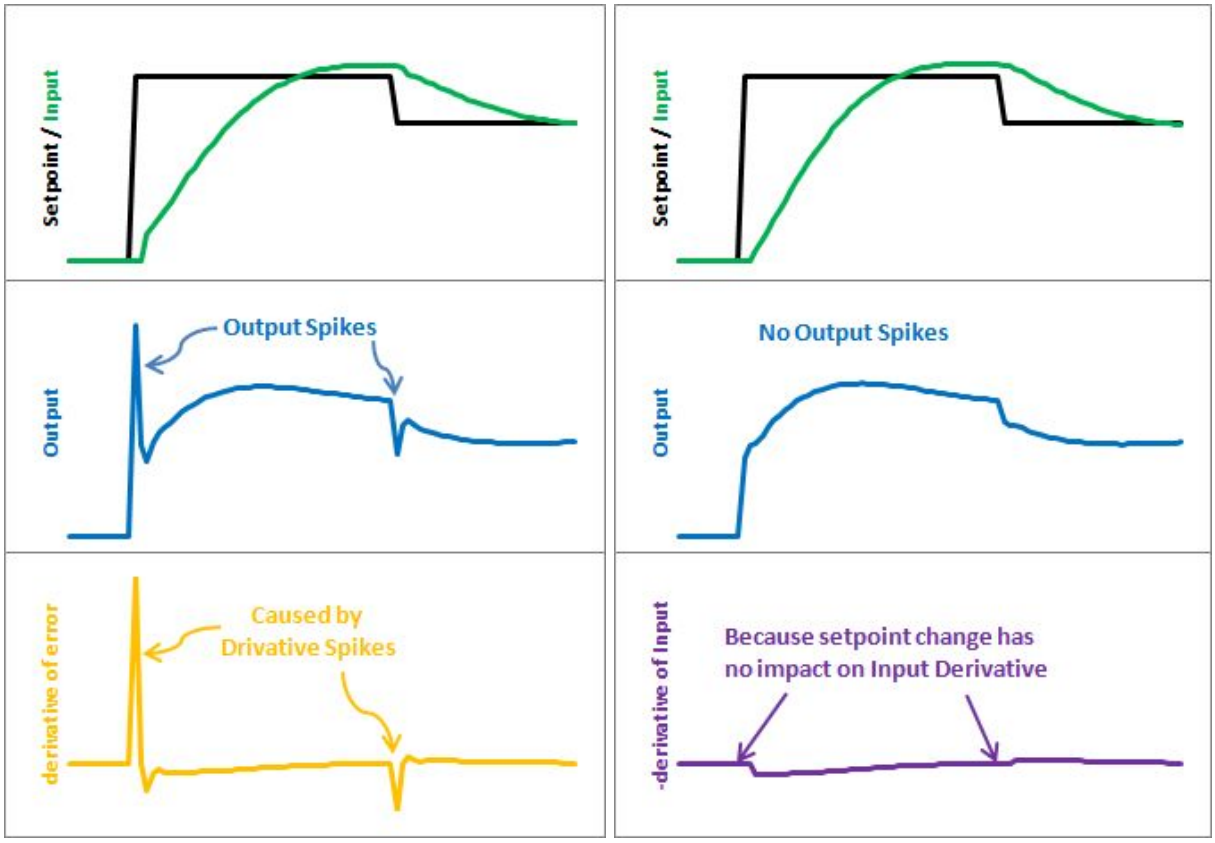

Figure 8: (left) Output before correction (right) Output after correction. Images sourced from Brett Beauregard's *Improving The Beginner's PID: Derivative Kick,* retrieved May 2019

Our setpoint is going to be changing quite a lot (the robot is, after all, intended to move and to always correct its balance) and we don't want these jerky spikes in motor output. We resolve this by changing how the derivative is calculated in our code. So we force the controller to call the derivative of the error as equal to the negative derivative of the input all the time. This means that the setpoint now has no impact on the error.

$$dError/dt = -dInput/dt$$

Controller direction

While this one isn't worthy of an output graph, we still needed the TWIP to have a concept of backwards, especially for turning in place. Since the motors are identical but facing opposite directions, going forward requires sending one direction to one motor and the opposite direction to the other. The robot doesn't know which side is its front or back, only which direction it is being told to go. We handled this with simple additions of constants to our code, allowing it to apply torque in the correct direction to handle a tilt.

On-The-Fly Tuning Changes

We need to adjust the PID constants from time to time without resetting the robot entirely. When we do this, the integral term will multiply past values by the new Ki term-- not what we wanted! The result of this is an undesirable bump in the output of the controller as it multiplies old error sums by the new constants. We overcome this by changing how the Ki term is treated.

Instead of using the basic interpretation of the KI term, where it is multiplied by the integral:

$$K_I \int e\,dt \approx K_{I_n}\left[e_n + e_{n-1} + \ldots\right]$$

We bring the Ki term inside the integral.

$$K_I \int e\,dt = \int K_I\, e\,dt$$
$$\int K_I\, e\,dt \approx K_{I_n} e_n + K_{I_{n-1}} e_{n-1} + \ldots$$

In practice, the effect of this is that when Ki changed, there is no bump because all the old Ki terms are already stored inside the integration of past values.

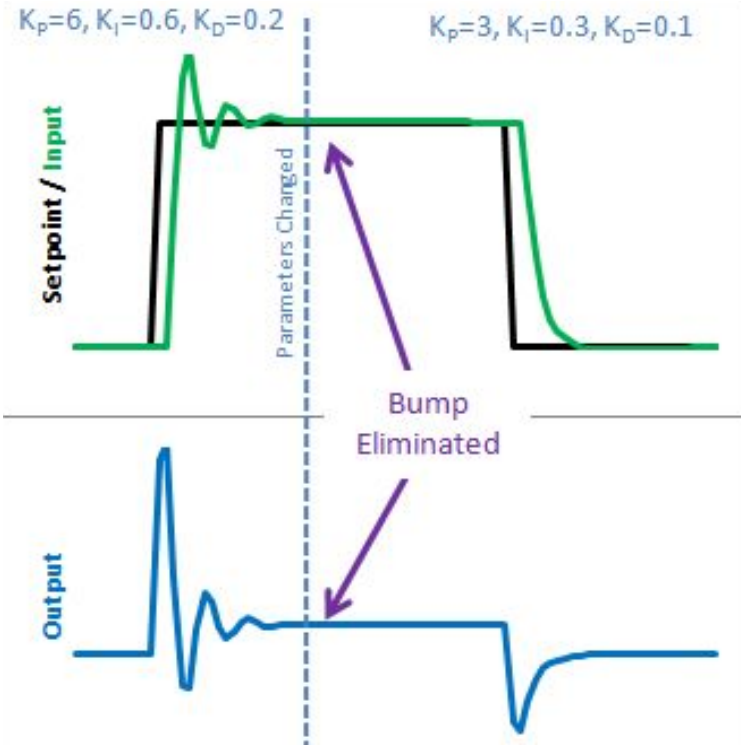

Figure 9: (top) Output before correction (bottom) Output after correction. Images sourced from Brett Beauregard's *Improving The Beginner's PID: Tuning Changes,* retrieved May 2019

Sample Rate

The PID controller we used was sampled at 100Hz. The cornering frequency (the frequency at which change of slope occurs in asymptotic bode plot, which can be either the frequency at which a pole or a zero occurs) of the motor was 30Hz, meaning our minimum sample time had to be 60Hz. But more is better, so we went with 100Hz.

Motor Deadzone

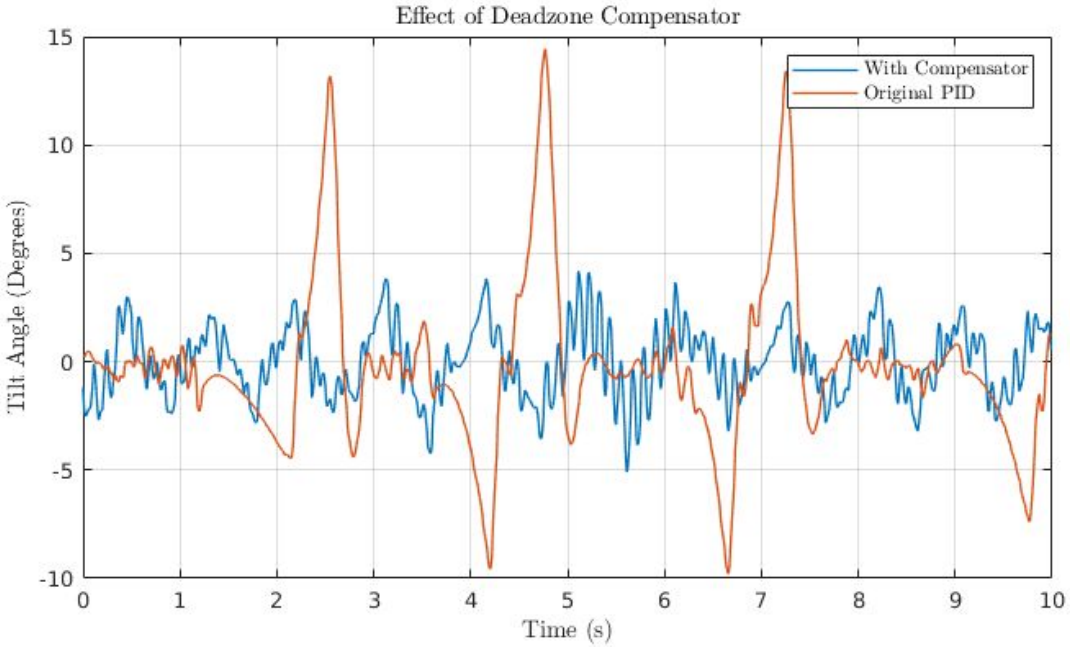

Figure 10: The Motor Deadzone graph

The controller derived thus far obeys a linear control law with an additional integrator used to remove steady state error. However, certain assumptions were made with regard to actuation that weren't always true depending on the motors used. When the system is near its set point for a brief period of time, a small control signal is needed to make small corrections to its stability. When the control signal is small enough, the motors may not respond at all because the static friction present in the gearbox keeps the armature inactive. This phenomenon is known as a *motor deadzone*, and it makes a linear controller ineffective near its equilibrium point.

Naturally, this deadzone creates large amplitude steady state tilt behavior. When the initial control signal proves ineffective, a PID controller's integral term will keep increasing until the static friction is overcome, sending the robot back to equilibrium after some delay. Once returned to equilibrium, the cycle continues--the system starts to sway and then the system eventually sends a control signal large enough to move the load back.

The linear control law can be modified to compensate for the deadzone in the motors. The revised law can be expressed as

$$\begin{cases} h = Kx \\ u = h + \text{sgn}(h)b \end{cases}$$

h is the linear controller and u has a bias term added in order to compensate for the minimum signal strength b needed to make the armature move. This bias is added in the same direction that the linear controller specifies, hence the $\text{sgn}(h)b$ factor. Effectively, a bang-bang controller is added in with the existing controller. The trade-off, unfortunately, is that although the steady state amplitude is greatly reduced, the steady state frequency is higher as the robot is able to compensate more quickly. Visually, this causes the robot to “shiver”, vibrating in place.

Hardware Considerations and Selection Process

The physical characteristics of the components and chassis directly influence the dynamics and stability of the robot and, subsequently, the robustness required by the balancing controller. The most critical components for the stability of the robot are the Inertial Measurement Unit (IMU), wheels, and motors. However, other components and design considerations are discussed at the end of this section.

The IMU is crucial to the stability of the robot because its ability to measure the orientation of the robot in space is the primary input to the controller. While filtering the data from the IMU is unavoidable, having quality components on the IMU itself will make processing the data simpler, faster, and more accurate. Other factors affecting the IMUs accuracy such as shielding, orientation, and fastening techniques will not be discussed in this section; only attributes of the IMU itself will be explored.

A magnetic, angular rate and gravity (MARG) system was modeled and used for orientation detection. An accelerometer, the device used to detect gravity, is able to measure gravitational acceleration direction in addition to the linear accelerations experienced by the body of the system. Likewise, a magnetometer can measure the global magnetic field added with the local magnetic disturbances present. Finally, the angular rate of the body is found via a gyroscope, capable of finding the change of rotation over time but not the absolute rotational orientation with respect to the earth. Worse, MEMS gyroscopes are magnetically sensitive, meaning that considerable drift error will occur if AHRS techniques like dead reckoning are solely employed. These incomplete detections, alongside the considerable noise sources mentioned, means sensor fusion is required in addition to denoise filtering.

The three principal traits, then, that complicated the IMU considerations are Magnetic Sensitivity, Convergence Time, and Degrees of Freedom (DoF) Limitations. Magnetic sensitivity

was largely addressed by fusing the magnetometer with the gyroscope in such a way that local disturbances could be detected and rejected, although some IMUs available on the market promised considerably fewer offset errors than others. Convergence time is dependent both on the “cleverness” of the algorithm to detect noise and the sampling rate of the IMU. Some of the available IMU options had superior oversampling, meaning that the onboard sensor processor could filter out some noise before the data was even transported to the central processor. The algorithm fusing the data also has a convergence time aspect; if the MARG data is quickly changing in response to orientation, a filter could reject it because it contains either low pass characteristics or mistakes it for noise. Filters like the 6 DoF complementary and 9 DoF Mahony both can have inferior convergence time if improperly tuned. Lastly, the DoFs available on the sensor can affect accuracy; many orientation sensors on the market skip the magnetometer, assuming that the application’s environment is magnetically static.

The application torque through the wheels is the only way a TWIP robot can balance. Therefore, the torque output of the motors has the greatest effect on the stability of the robot. Motor qualities such as rotational speed and deadzone also influence stability, but to a lesser extent; rotational speed primarily dictates top speed of the robot whereas deadzone influences the ability for the robot to make small changes.

When choosing motors for a TWIP robot, first determine the torque required to satisfy your requirements for stability. This torque value is dependent on the robot’s mass and wheel size. For our approximately 1.2 kilogram robot with 80mm wheels, we chose motors with 1 kg * cm torque and a rotational speed of 560 Revolutions per Minute (RPM). At this point, having a simulation of your model is useful because it allows you to analyze a motor’s performance in your system needing to purchase anything. A rotational speed of 500 to 600 RPM is ideal for a small TWIP robot with medium sized wheels; this speed allows the robot to respond quickly to changes in tilt angle and should provide a top speed greater than the average walking pace. However, as long as a motor continues to provide ample torque, there are no downsides to having a faster rotational speed.

Unlike torque and rotational speed, deadzone is difficult to measure and, in our experience in this project, seldom provided with a motor’s specifications. Deadzone in motors is a function of other parameters such as static friction and gear backlash. Refer to the *Motor Deadzone* section under *Controller Design Process: Overcoming Issues* for more information. In our experience, the motors used on the first robot had a very large deadzone; this manifested itself as steady state oscillations without deadzone compensation. With deadzone compensation, these motors applied torque abruptly to maintain stability while at rest. Large deadzones limit the amount of control the robot has when applying small torques; this is readily observed when a TWIP robot is standing in place because only minute forces are required to maintain stability. Unfortunately, if deadzone is not specified by the motor manufacturer, it’s difficult to determine the size of a motor’s deadzone without physically measuring the motor.

For the purposes of maintaining consistent power delivery and to streamline the charging process, both robots used boost/buck converters to regulate the power supplied to both the motors and the microcontroller. This meant that both on-board batteries would be depleted evenly and a constant voltage would be supplied to all components for a wide range of States of Charge (SoC). We recommend that future designs retain this feature; we had no issues with power for the duration of the project and this design kept performance consistent regardless of battery charge. Regarding the selection of buck/boost converters, we found that screw-type terminals and an integrated LCD to monitor input/output voltage were convenient features to have during construct.

Hardware Testing

Hardware testing served two main purposes for this project. One of them being the ability to determine parameters used in the model, and the other being to perform validation of the model.

The main piece of hardware that was tested was the Pololu 3262 brushed DC motor. Using a power supply and digital multimeter, motor parameters such as the armature resistance and inductance were determined.

With respect to validation, the components that were tested were the IMU and the DC motor. Further detail is provided in the latter, under the *Validation* section.

Writing The Software

Many Tools Needed

In the course of the project, we stumbled upon the need for a lot of tools that weren't controller software or verification software, and this section is meant to be a guide to those souls who come after us. We wrote code to simulate the performance of the TWIP to test our assumptions about the fundamental dynamics equations, also in Python but with its 3D images done in Blender. We used SerialPlot to stream information from our IMU. We ended up needing GUIs for calibration and telemetry, also written in Python.We needed MATLAB and Simulink for graphing the performance of test parts, and as plugins to other verification software. We surely needed a lot of linux-capable machines, and to know how to navigate problems with them. We needed Rust and Embedded Rust guides, dozens of them, to navigate the multiple releases of that codebase and to make it accomodate our hardware, and to find the specific Rust crates for our microcontroller to expose its pins and functions. We wrote the first controller software in Sketch. We explored lots of verification and model-checking tools: CoCoSim, S-TaLiRo,

Simulink, KIND2, and wrote the code for KIND2 in a language called Lustre. There were a wealth of things no one on the team had ever seen before, and we each had an opportunity to fall into entirely new worlds of code languages and math concepts in the course of this project.

Here, we will spend time on some of the larger pieces, more or less in the order that they appeared.

The Simulator

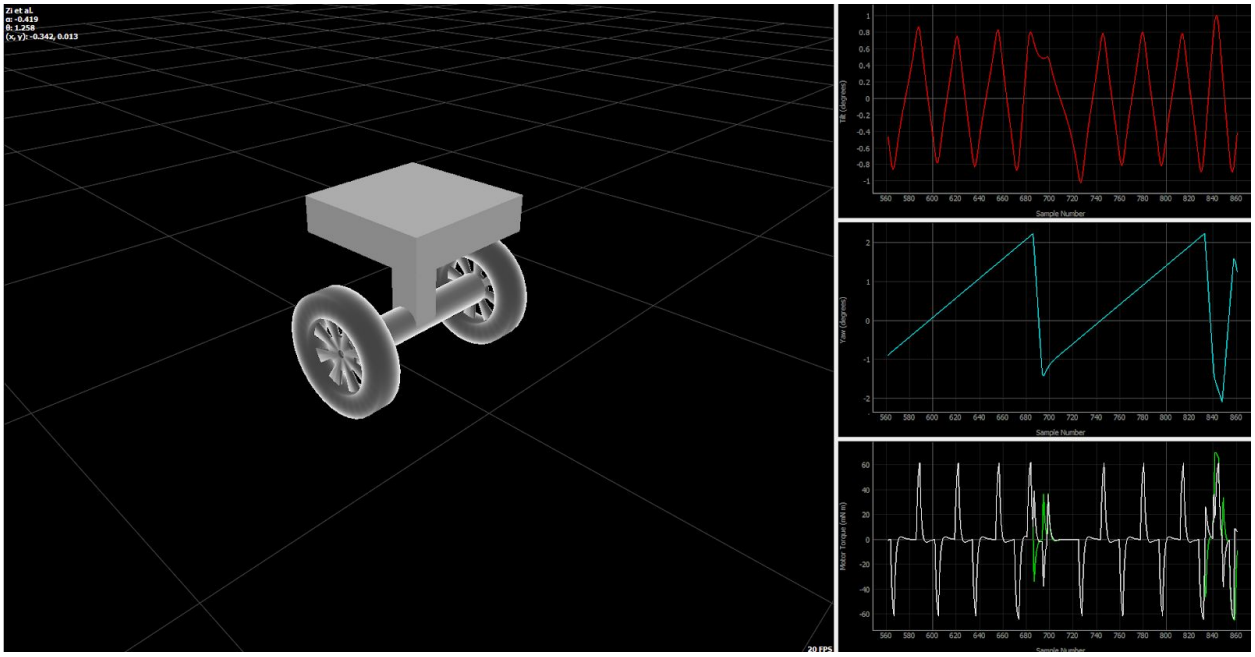

Figure 11: (left) Screenshot of the simulator used to test the model checker results. (right) Telemetry output: (top right): tilt angle (center right): yaw angle (bottom right): motor torque

A simulator was designed and written to load the counterexamples generated by the model checker and study the reasons for failure. The simulator is a mixed domain orbit solver, being able to run the iterated maps of a discrete-time controller as well as the robot's continuous time dynamics together. After the addition of motor dynamics, a stiff ODE solver was added in order to reduce the computation time; the simulator offers both a real-time (online) and offline mode. Inertial and geometric properties could be set and reloaded in order to perform elementary sensitivity studies. Lastly, the simulator was lightweight and could be bundled and run on a compute cluster for brute force verification. A process pool was added so that all cores on a CPU can solve for orbits in parallel.

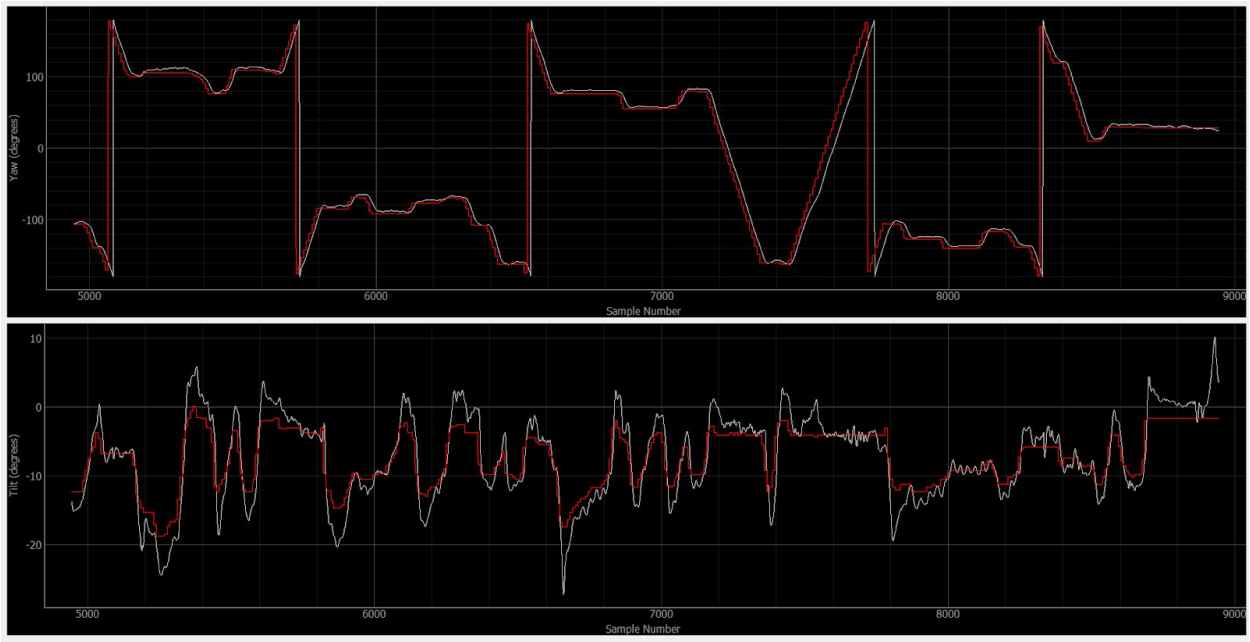

Figure 12: Screenshot of the telemetry output. (top) Yaw (bottom) Til. Note that the red signal is the setpoint.

A wireless telemetry system was implemented in order to facilitate validation. Displaying the setpoint in red, it is possible to observe how quickly the robot can converge on a new setpoint or reject disturbances. Further, this level of real-time oversight allows heuristical tuning methods to be employed. For enjoyment, Ziegler-Nichols tuning was performed using the data available from the telemetry dashboard.

Controller Program

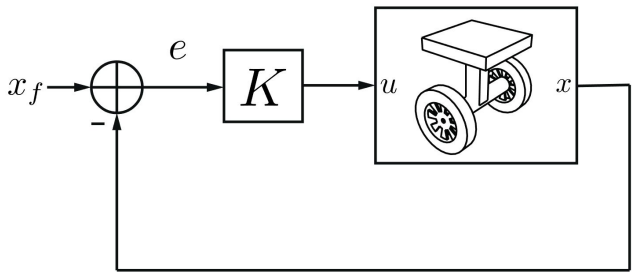

Figure 13: A simplified PID control loop

We decided to go with a bluetooth joystick to stream input to our robot. This meant that we needed to incorporate a bluetooth chip and a bluetooth library into our code. A Kalman filter was also needed, to smooth our input and output and to prevent the robot for jerkily responding to every bump in signal. On-board gyroscope, accelerometers and magnetometers provided the sensing mechanism for the robot to judge its position. The first robot had two PID controllers

implemented; one kept track of forward-backward tilt, and one kept track of yaw. This meant that if you rotate the robot in place with your hands, it corrected itself to its previous position, and if it was nudged with a foot it would return to its upright position. At rest, the robot did its best to simply stand in place.

The control program, at a high level, was this:

SETUP:
Set up hardware connections
Set up all constants
Create a telemetry buffer
Create bluetooth signal
Setup PID for tilt
Setup PD for Yaw (not PID)
Initialize PWM controls
Set up FIFO buffer

Check for how long since last sample
If past set startup delay, proceed:
    Turn on "Ready" LED on robot
    Compute PID for current weights and setpoint
    Grab Bluetooth data for current setpoint
        Send bluetooth data to telemetry graph
    Parse setpoint from bluetooth
    Calculate new PID
    Apply tilt values to both motors equally
        If yaw change detected, apply sign change to one of the motors.
    Convert PID outputs to PWM for the motors
    If robot has not fallen over,
        Apply PWM to motors
    Else,
        Apply PWM to motors with 0 power
    LOOP:
        Update yaw buffer
        Update tilt buffer
        Check for current relationship to the setpoint
        If bluetooth data is coming in from joystick, put in buffer and update yaw and tilt setpoints
        Convert bluetooth data to floats
        Recalculate setpoint

Coding in Rust

Rust is a still-evolving language, and your mileage may vary. As we were working with the STM32F303DISCOVERY board, we had access to a wealth of online tutorials to get oriented. Appendix F outlines how to flash Rust code to a microcontroller, a non-trivial task. Appendix G outlines some of the changes needed on verified code before it can be used on an embedded system like ours. The purpose of using Rust for this was manifold: First of all, Rust is designed to run at great speed with high reliability; its main focus is on *safety*. It achieves this, in part, by performing checks at compile time for pointer-related undefined behavior, the major silent killer in C and C++, as well as defaulting to immutable variables, forcing the user to think about which values really need to be updated during runtime. Its type system expresses regions through lifetimes, and the lifetime of a region can even be used as a parameter for type constructors. These things in combination let the programmer effectively avoid aliasing issues of shared mutability, trying to use a resource after it has been freed, dangling pointers, and so on. In addition, Rust uses linear types by default; it has no move constructors, replacing them with drop flags that show whether a variable belongs to a given scope. And since it checks for all of these things at compile time, the programmer can be assured that if the program compiles, then the program is safe.

But embedded programming is another animal altogether. The challenges of learning Rust syntax in an embedded environment were not trivial, and were not fully conquered by the end of this project; while some trivial examples were written and did compile, in the end the validated robot controller was written in C, with only a PID library ever fully implemented in Rust. The repository associated with this project does contain some Rust example work for future engineering students to bridge off of.

Testing and Validation

Figure 14 shows the setup of the *Prony Brake* used for motor testing. The concept behind the Prony Brake is that we are able to calculate torque through measured force. For the motor to produce torque, a shoelace was placed under equal tension. When the motor is powered, there exists a net force F_{net} that is perpendicular to the rotating wheel. The net force is obtained by the difference of the two measured forces. Hence, the driving torque is calculated by

$$\tau = rF_{net}$$

where r is the radius of the drive wheel.

Measurements were obtained for different torque values by applying more force to the drive wheel (increase Prony Brake height). In addition, measurements were obtained for the motor rotating in both CW and CCW directions.

The relationship between torque and current is shown in Figure 15. Recall, the equation for torque of a DC motor; that is,

$$\tau = K_m I_a$$

where, K_m is the torque coefficient. As shown in Figure 15, there is a linear relationship between torque and armature current. Hence, the torque coefficient was found to be 0.0993 N-m/A.

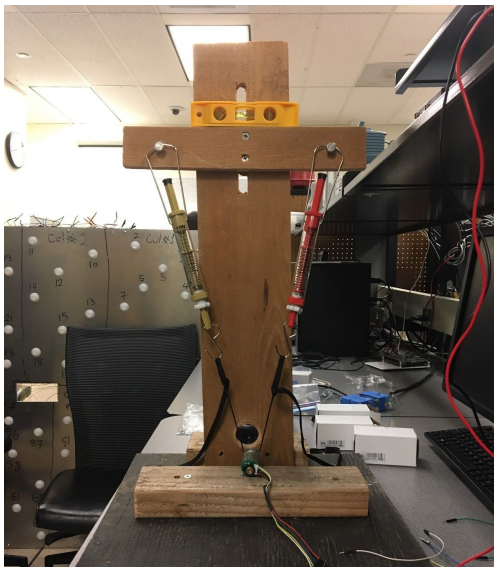

Figure 14: Prony Brake setup

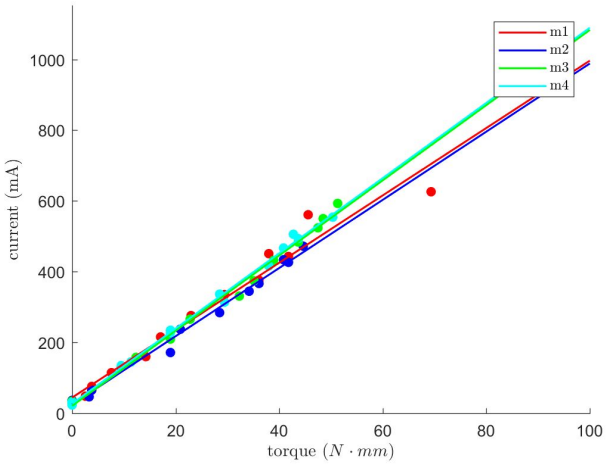

Figure 15: Current vs Torque for four Pololu motors

Using the equations for a DC motor in the Torque Modeling section, measurements were carried out to determine a relationship between angular speed ω and the back EMF voltage $\mathcal{E}$.

Note that the speed measurements were obtained through the use of a photo tachometer. Figure 16 shows a linear relationship between the two. In result, the EMF coefficient K_e was found to be 0.1893 V/rad/sec.

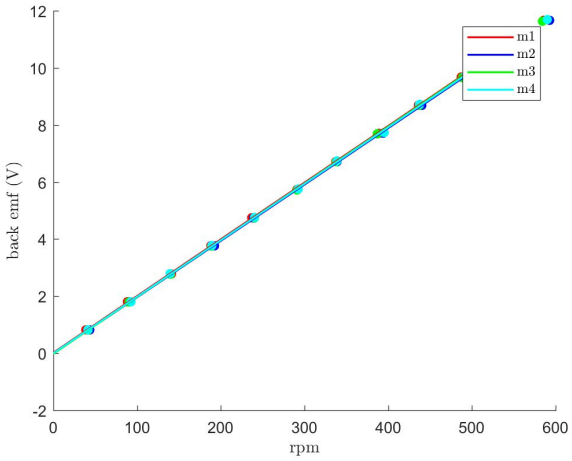

Figure 16: Back EMF vs RPM for four Pololu motors

Figure 17: Phototachometer used to measure the RPM of the motor

The Tools

The tools tools needed to carry out the testing of the DC motors were basic lab equipment, a *Prony Brake*, and a photo tachometer. A power supply was needed for the motors, and a digital multimeter was used to measure current, resistance, and inductance. For testing motor torque, the *Prony Brake* was constructed from scratch. Two 20-Newton spring scales and a shoelace (or polycord) were used within the *Prony Brake*. In addition, the photo tachometer was used to measure the RPM of the motors. Note that reflective tape may be used on the

rotating drive wheel to stabilize the readings on the photo tachometer. The measurements were recorded in Microsoft Excel and further analyzed using Matlab.

Results

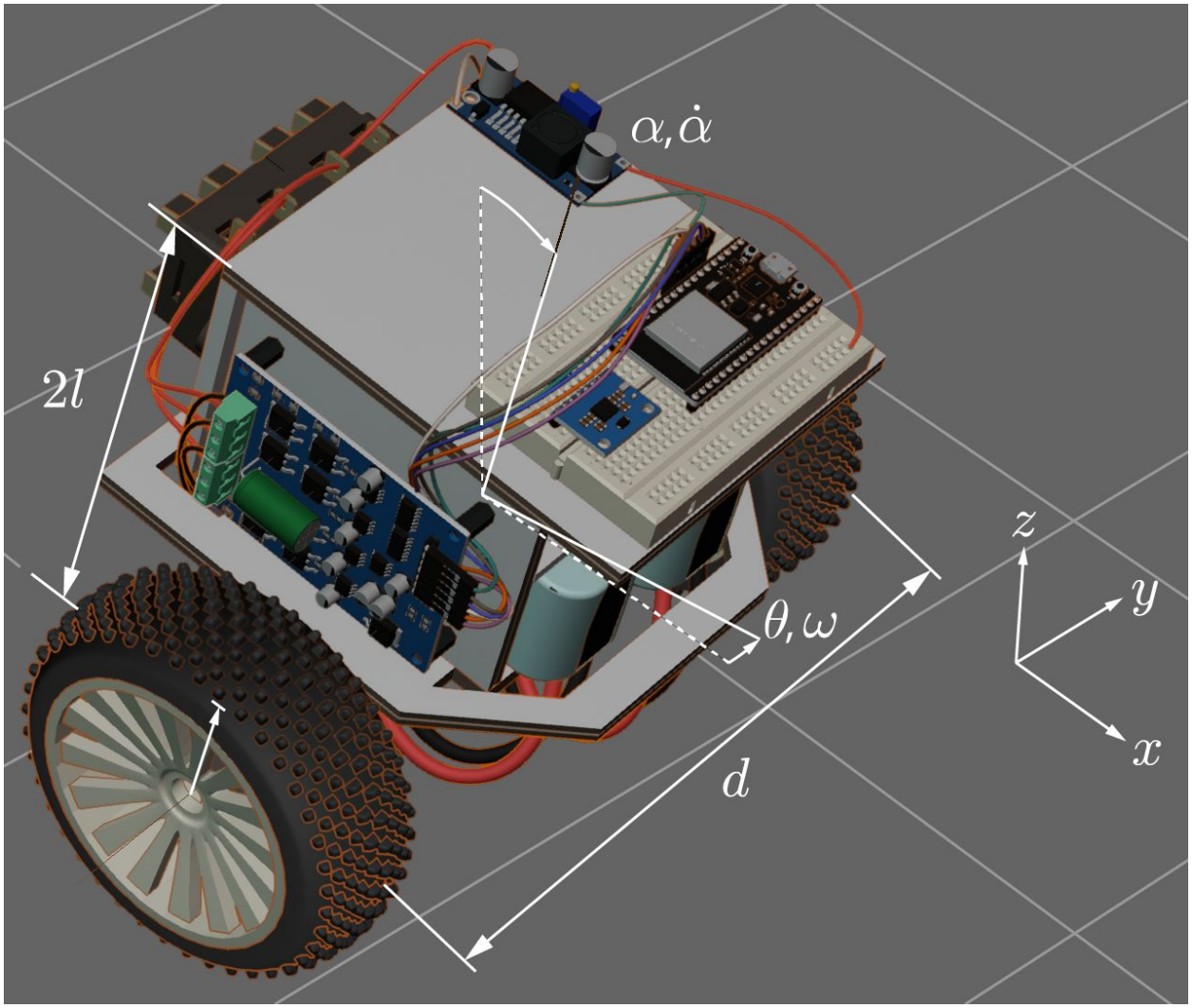

Figure 18: The first prototype, modeled in Blender

This project shows the complete model, design and implementation of a TWIP robot. These methods serve to illustrate how model checker can be used for hybrid systems. The robot's features were extensively modeled using differential equations and then translated to other forms via techniques like linearization and orbit coding. A methodology was created to convert continuous state behavior into a finite element graph that can be verified using traditional model checking software. Finally, the robot was built and shown to have gained improved stability from the high assurance design pipeline that was employed to create it.

Conclusion

The chief accomplishments of this team was the discovery and implementation of a process by which model-checking tools for digital systems can be used to verify hybrid systems. From the onset of the project, we were tasked with developing a model of our system, exploring different verification tools, and finding methods by which these tools can be applied to our system. We achieved this goal by familiarizing ourselves with each tool's features and limitations with respect to the requirements of our system and assignment. Not all tools were relevant or useful to the systems, but a few --namely Kind2, Simulink, and S-Taliro-- proved directly applicable to our project.

The system of a TWIP robot in inherently nonlinear. The primary goal of this project was to find a model-checking tool that was capable of verifying our system; the challenge of this goal was that most of these tools were suitable for digital, linear systems. Our system, however, was nonlinear and mixed signal. In order to adapt our system for these model-checkers, we discretizing the continuous time, continuous state behavior of our system using symbolic encoding. This allowed for our system to be formally verified and, more importantly, the results of this verification closely aligned with brute force simulation. Having established a process to linearize a nonlinear system, we satisfied one of the primary goals outlined in the original proposal.

Another outcome of this project was the construction of two TWIP robots, one of which performed well enough to be demonstrated live at the Capstone poster session. At the poster session, judges, other students, and passersby could view and even operate our robot. Additionally, a live, wireless feed of telemetry data provided a clear means by which we could explain the different parameters of our system.

Next Steps

As the TWIP system is inherently nonlinear, future work would be to design a nonlinear controller. Similarly, the closed loop stability of the robot would need to be formally verified and compared against the PID controller.

On the implementation side, further work is needed to develop the controller in embedded Rust. Again, the performance should be compared with our C-based implementation.

References

[1] Jorge Aparic, A self-balancing robot coded in Rust, (2018), Github Repository, https://github.com/japaric/zen

[2] Stefan Kroboth, Rust on a Microcontroller, (2018), Github Repository, https://github.com/stefan-k/STM32F303DISCOVERY

[3] A. Greig, A. Richter, J. Munns, J. Pallant, D. Egger, The Embedded Rust Book, (2019), Github Repository, https://rust-embedded.github.io/book

[4] [Various Authors], The Cargo Book, (2019), Github Repository, https://doc.rust-lang.org/stable/cargo/index.html

[5] [Various Authors], Template to develop bare metal applications for Cortex-M microcontrollers, (2019), Github Repository, https://github.com/rust-embedded/cortex-m-quickstart

[6] [Various Authors], Generate Rust register maps (`struct`s) from SVD files, (2019), Github Repository, https://github.com/rust-embedded/svd2rust

[7] [Various Authors], Jorge Aparic, Board Support Crate for the STM32F3DISCOVERY, (2019), Github Repository, https://github.com/japaric/f3

[8] Rust By Example, https://doc.rust-lang.org/rust-by-example/

[9] Crate: Cortex-M-Quickstart; A template for building applications for ARM Cortex-M microcontrollers, (2018), github repository, https://rust-embedded.github.io/cortex-m-quickstart/cortex_m_quickstart/

[10] Brett Beauregard. *Improving The Beginner's PID Controller*. Project Blog. http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/ Accessed May 02, 2019.

[11]"STM32F303 Peripheral Coverage." *STM32 Tips and Tricks*, http://stm32.agg.io/rs/STM32F303.html

[12] "STM32F303 User Manual" https://www.st.com/content/ccc/resource/technical/document/user_manual/8a/56/97/63/8d/56/41/73/DM00063382.pdf/files/DM00063382.pdf/jcr:content/translations/en.DM00063382.pdf

[13] “STM32F303 Datasheet”
https://www.st.com/resource/en/datasheet/stm32f303vc.pdf

[14] “STM32F303 Reference Manual”
https://www.st.com/content/ccc/resource/technical/document/reference_manual/4a/19/6e/18/9d/92/43/32/DM00043574.pdf/files/DM00043574.pdf/jcr:content/translations/en.DM00043574.pdf

[15] S. N. Vukosavic, *Digital Control of Electrical Drives.* Springer, 2007.

[16] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel, “Verification of automotive control applications using s-taliro,” *American Control Conference*, 2012.

[17] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivancic, and A. Gupta, “Probabilistic temporal logic falsification of cyber-physical systems,” 2011.

[18] T. Kahsai, P. L. Garoche, H. Bourbouh, C. Pagetti, T. Loquen, and E. Noulard. (2017) Cocosim. [Online]. Available: https://coco-team.github.io/cocosim/

[19] Li, Z., Yang, C., & Fan, L. (2012). Advanced control of wheeled inverted pendulum systems. Springer Science & Business Media.

[20] Drechsler, R. (Ed.). (2004). Advanced formal verification (Vol. 122). Norwell: Kluwer Academic Publishers.

[21] Zecevic, A., & Siljak, D. D. (2010). Control of complex systems: Structural constraints and uncertainty. Springer Science & Business Media.

[22] Lynch, K. M., & Park, F. C. (2017). Modern Robotics. Cambridge University Press.

[23] Strogatz, S., Friedman, M., Mallinckrodt, A. J., & McKay, S. (1994). Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering. Computers in Physics, 8(5), 532-532.

[24] Osipenko, G. (2006). Dynamical systems, graphs, and algorithms. Springer.

[25] Baier, C., & Katoen, J. P. (2008). Principles of model checking. MIT press.

[26] Kunkel, P., & Mehrmann, V. (2006). Differential-algebraic equations: analysis and numerical solution (Vol. 2). European Mathematical Society.

[27] Grasser, F., D'arrigo, A., Colombi, S., & Rufer, A. C. (2002). JOE: a mobile, inverted pendulum. IEEE Transactions on industrial electronics, 49(1), 107-114.

[28] Ha, Y. S. (1996). Trajectory tracking control for navigation of the inverse pendulum type self-contained mobile robot. Robotics and autonomous systems, 17(1-2), 65-80.

[29] Dellnitz, M., Froyland, G., & Junge, O. (2001). The algorithms behind GAIO—Set oriented numerical methods for dynamical systems. In Ergodic theory, analysis, and efficient simulation of dynamical systems (pp. 145-174). Springer, Berlin, Heidelberg.

[30] Dellnitz, M., & Hohmann, A. (1997). A subdivision algorithm for the computation of unstable manifolds and global attractors. Numerische Mathematik, 75(3), 293-317.

[31] Goher, K. M., Tokhi, M. O., & Siddique, N. H. (2011). Dynamic modeling and control of a two wheeled robotic vehicle with a virtual payload. ARPN Journal of Engineering and Applied Sciences, 6(3), 7-41.

[32] Nawawi, S. W., Ahmad, M. N., & Osman, J. H. S. (2008). Real-time control of a two-wheeled inverted pendulum mobile robot. World Academy of Science, Engineering and Technology, 39, 214-220.

[33] England, J. P., Krauskopf, B., & Osinga, H. M. (2004). Computing one-dimensional stable manifolds and stable sets of planar maps without the inverse. SIAM Journal on Applied Dynamical Systems, 3(2), 161-190.

[34] Diebel, J. (2006). Representing attitude: Euler angles, unit quaternions, and rotation vectors. Matrix, 58(15-16), 1-35.

[35] Mirzaei, F. M., & Roumeliotis, S. I. (2008). A Kalman filter-based algorithm for IMU-camera calibration: Observability analysis and performance evaluation. IEEE transactions on robotics, 24(5), 1143-1156.

Appendix B: Bill of Materials

Part	Part Number	Price	Quantity	Subtotal
Arduino Uno	**A000073**	$21.40	**1**	$21.40
Raspberry Pi Model 3B+		$38.30	**1**	$38.30
Samsung 32GB 95MB/s (U1) MicroSD EVO Select Memory Card with Adapter	**MB-ME32GA/AM**	$7.99	**1**	$7.99
STMICROELECTRONICS STM32F3DISCOVERY EVAL KIT, STM32 F3 SERIES DISCOVERY	**STM32F3DISCOVERY**	$31.50	**1**	$31.50
ST-LINK/V2(CN Version) ST MCU Microcontroller STM8 STM32 JTAG SWD SWIM In-circuit Debugger Programmer Emulator @XYGStudy	**-**	$36.99	**1**	$36.99
Pololu 9.7:1 Metal Gearmotor 25Dx48L mm LP 12V with 48 CPR Encoder	**3262**	$34.95	**4**	$139.80
Pololu 25D mm Metal Gearmotor Bracket Pair	**2676**	$7.45	**1**	$7.45
Blomiky 2 Pack 11.1V 3S 2200mAh 25C Lipo Battery Pack with T Deans Plug and Balance Charger for RC Airplane Helicopter Car RC Truck Boat Quadcopter T 11.1V 2200mAH 2	-	$36.99	**1**	$36.99
Buck Boost Converter Display, DROK Buck-Boost Board DC 5.5-30V 12v to DC 0.5-30V 5v 24v Adjustable Constant Current Voltage Step UP Down Voltage Regulator 3A 35W Power Supply Module	**-**	$13.98	**3**	$41.94
Adafruit Motor/Stepper/Servo Shield for Arduino v2.3 Kit	**1438**	$18.90	**2**	$37.80
GY-521 MPU-6050 MPU6050 3 Axis Accelerometer Gyroscope Module 6 DOF 6-axis Accelerometer Gyroscope Sensor Module 16 Bit AD Converter Data Output IIC I2C	**MPU6050**	$5.79	**3**	$17.37
Pololu 1083 Universal Aluminum MOUNTING HUB for 6mm Shaft Pair, 4-40 Holes	**43220-262879**	$13.89	**1**	$13.89
80 x 10mm Black Robot Wheels	**595660**	$11.99	**1**	$11.99

Atomic Market HC-06 Bluetooth Serial Pass-Through Module Wireless Serial Communication Compatible with Arduino	**HC-06**	$8.99	**1**	$8.99
VETOMILE 6-way Fuse Box Blade Fuse Block Holder Screw Nut Terminal 5A 10A 15A 20A Free Fuses LED Indicator Waterproof Cover for Automotive Car Marine Boat	**-**	$9.99	**1**	$9.99
MICTUNING 100PCS ATC/ATO Assorted Standard Blade Fuse Set - (2A 3A 5A 7.5A 10A 15A 20A 25A 30A 35A) - Car Truck SUV Boat Automotive Replacement Fuses	**MIC-ABF-146**	$7.99	**1**	$7.99
HVAZI 200pcs Metric M2.5 M3 304 Stainless Steel Hex Socket Head Cap Screws Assortment Kit	**BTM25M3200P**	$11.99	**1**	$11.99
Hilitchi 180pcs M4 Stainless Steel Hex Socket Head Cap Screws Nuts Assortment Kit with Box (M4)	**HSH180**	$12.99	**1**	$12.99
HVAZI 205pcs Metric M2.5 M3 M4 M5 M6 M8 M10 M12 Stainless Steel Nylon Hex Lock Nut Assortment Kit	**FSLM205P**	$14.90	**1**	$14.90
TOTAL FOR ONE UNIT	$360.52			
TOTAL INCLUDING SPARE PARTS	**$510.26**			

Appendix C: CAD Models

Chassis of the Second Robot. Designed and Rendered in Autodesk Fusion360

Appendix D: How To Put Rust On A STM32F303 DISCOVERY Microcontroller

Start with a Linux environment. We used a Linux Ubuntu 18.04 system.

1. Install Rust by following the directions at https://rustup.rs. At time of this report, to install on Linux took the following terminal command:
   1.1. **curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh**
   1.2. Then, switch to the "nightly" channel by typing **rustup default nightly**
   1.3. Install itm using **cargo install itm**
2. Install dependencies
   2.1. **sudo apt-get install \ gcc-arm-none-eabi \ gdb-arm-none-eabi \ cutecom \ openocd \ bluez \ rfkill**
3. Create these two files in `/etc/udev/rules.d` with the contents shown below.
   3.1. **$ cat /etc/udev/rules.d/99-ftdi.rules**

```
# FT232 - USB <-> Serial Converter
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", GROUP="uucp"
```

   3.2. **cat /etc/udev/rules.d/99-openocd.rules**

```
# STM32F3DISCOVERY rev A/B - ST-LINK/V2

ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3748", GROUP="uucp"

# STM32F3DISCOVERY rev C+ - ST-LINK/V2-1
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", GROUP="uucp"
```

   3.3. Save them and then reload the rules by typing
   sudo udevadm control --reload-rules
   3.4. Check that you are a member of the uucp group by typing
   groups $(id -nu)
   It should return:

```
(..) uucp (..)
     ^^^^
```

   3.4.1. If you aren't, add yourself using **sudo usermod -a -G uucp $(id -u -n)**
   3.4.1.1. Then check again with groups $(id -nu)
   3.5. Reboot your machine with `sudo reboot`
4. Connect the STM32F303 to your machine with the USB cable, plugged in to its "USB ST-LINK" port in the middle of the board.
   4.1. Verify its permissions and connection status using **lsusb | grep -i stm**
   4.1.1. You should get back something like:

```
Bus 003 Device 004: ID 0483:374b STMicroelectronics ST-LINK/V2.1
    ^^^         ^^^
```

4.1.2. Take note of the bus number and device number for the next step.

4.1.3. `ls -l /dev/bus/usb/MYBUSNUMBER/MYDEVICENUMBER`

4.1.4. Get back something like:

4.1.5. `crw-rw-r-- 1 root uucp 189, 262 Oct 27 00:00 /dev/bus/usb/003/004`

4.1.5.1. If “uucp” doesn’t show up, check your udev rules from step 3 and reload them.

5. Setting up OpenOCD

5.1. OpenOCD is how we are going to see what is happening on the board. Make sure the STM32 is plugged in and powered. Then, in a separate terminal, type **`openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg`**

5.1.1. You should see output like this:

```
Open On-Chip Debugger 0.9.0 (2016-04-27-23:18)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport
"hla_swd". To override use 'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control.
The results might differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.914184
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

5.1.1.1. If not, consult general troubleshooting.

5.1.1.2. If so, then OpenOCD is set up! Kill it, we don’t need it yet.

6. Using Cortex-M-Quickstart

6.1. Install dependencies

6.1.1. `cargo install cargo-clone`

6.1.2. `sudo apt-get install gdb-arm-none-eabi`

6.1.3. `cargo install cargo-edit`

6.1.4. `rustup target add thumbv7em-none-eabihf`

6.2. Begin by cloning the quickstart repo:

```
cargo clone cortex-m-quickstart --vers 0.3.4
```

6.3. Edit the Cargo.toml file that appears, changing its name, author, and version. *NOTE: The name cannot contain spaces, underscores are preferred.*

6.4. (IF YOU ARE USING THE F3 CRATE, SKIP TO 6.4.2) Open the memory.x file that appears and ensure that it contains the following section, and that the numbers match. If not, change them:

6.4.1. `MEMORY`

```
{
  /* NOTE K = KiBi = 1024 bytes */
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH =} 40K
```

6.4.2. **If you're using the f3 crate,** delete memory.x and build.rs by typing

`rm memory.x`

`rm build.rs`

6.4.2.1. And then open Cargo.toml and add this:

```
[dependencies.f3]
version = "0.6.1"
features = ["rt"]
```

6.5. Edit the .cargo/config file and add the following section to the bottom:

6.5.1. `[build]`

`target = "thumbv7em-none-eabihf"`

6.6. We used the f3 board support crate. If using some kind of board support crate, this is the part where we add it:

6.6.1. `cargo add f3`

7. Write the application in src/main.rs. This is a huge step, and outside the scope of a quick guide. If you are simply testing to find out if your build worked, build one of the examples by typing `rm -r src/* && cp examples/hello.rs src/main.rs`
8. Plug in your board, if it isn't already. Then build your application by typing `cargo build --release`
   8.1. It may challenge you by saying things like "no panic-semihosting". Try to clear these errors as they appear by typing `cargo add [thing it says I don't have]`, and then try to build it again.
   8.2. It may also challenge you about the naming conventions of your work. If these are warnings, and not errors, you can skip fixing them by commenting out #![deny(warnings)] at the top of the example file.
9. If step 9 worked, great! In the directory you build your application in, in a separate terminal, run openocd by typing `openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg`
10. In the terminal you built your application in, run gdb by typing
    10.1. `arm-none-eabi-gdb -q target/thumbv7em-none-eabihf/release/[MY_PROGRAM_NAME_GOES_HERE]`

10.1.1. We had to install and use gdb-multiarch instead, so if this fails try `gdb-multiarch -q target/thumbv7em-none-eabihf/release/[MY_PROGRAM_NAME_GOES_HERE]`

10.2. Gdb should have launched. If it did, great! Into the gdb terminal type `target remote :3333`

10.3. Flash the device by typing `load`

10.4. Set your program counter to start at main by typing `break main`

10.5. Start stepping through with `c`

10.6. In the OpenOCD terminal, you'll see output (if there is any!) And if any lights are supposed to start blinking on the board, they now will be. You did it! In only 52+ steps!

10.6.1. When you're done, type `quit`**.** You may have to type it more than once.

Appendix E: Converting KIND2 output to Embedded Rust

Because embedded programs rely on a system that may not have an OS, or any other kind of program running at all, adapting the non-embedded output of the KIND2 tool requires first understanding how embedded rust programs compile.

The smallest embedded program that compiles, taken from the Embedded Rust Book, is:

```
#![allow(unused_variables)]
#![no_main]
#![no_std]

fn main() {
use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}
```

The third line, ***#![no_std]***, is where the bulk of the problems arise from. It means that this program does not implement the standard library for Rust programs. Since a great many utility crates rely on the standard library (and it requires close inspection for each one to determine whether they d0 or not), simple tasks like telling the time or calculating an integral requires extra attention to whether or not a crate will compile. It is easier to write an embedded program line by line than it is to adapt an existing program of any real length, but in the case of our KIND2 output, that's not an option.

To get our feet wet, we used the KIND2 input code from appendix A2, and used the kind2rust option in KIND2 to create the Rust code in appendix A3.

Appendix F: Additional Verification Tools

S-TaLiRo

S-TaLiRo is a Matlab toolbox used for verification of a Cyber-Physical System Model. Unlike Kind2, where properties of a system are verified, S-TaLiRo performs a stochastic search of system trajectories that falsify a temporal logic specification [cite fainekos]. The toolbox searches for paths with minimal robustness by using global optimization techniques which include the simulated annealing method, cross-entropy method, uniform sampling, and the genetic algorithm. The notion of a robustness metric is a measure of how satisfiable a trajectory is from the temporal specification. A positive robustness corresponds to a trajectory, along with a neighborhood of trajectories, that satisfies the specification, where as a negative robustness implies that the trajectory does not meet the specification [cite fainekos].

Logical operators are *conjunction* ($\wedge$), *disjunction* ($\vee$), *negation* ($\neg$), *implication* ($\rightarrow$), and *equivalence* ($\leftrightarrow$). Common temporal operators are *eventually* ($\Diamond_{\mathcal{I}}$), *always* ($\Box_{\mathcal{I}}$), and *until* ($\mathcal{U}_{\mathcal{I}}$), where $\mathcal{I}$ is a timing constraint. If there is no timing constraint then the specification is a Linear Temporal Logic (LTL) formula; otherwise, it is a Metric Temporal Logic (MTL) formula.

In the latter, we present the verification of a discrete PID controller using S-TaLiRo. First, let us consider the system in Figure F.1.

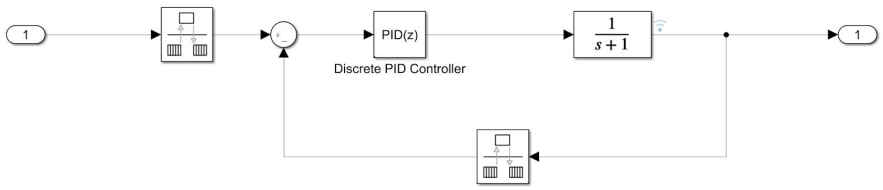

Figure F.1: Simulink model of discrete PID controller cascaded with the plant. The plant has the transfer function $\frac{1}{s+1}$.

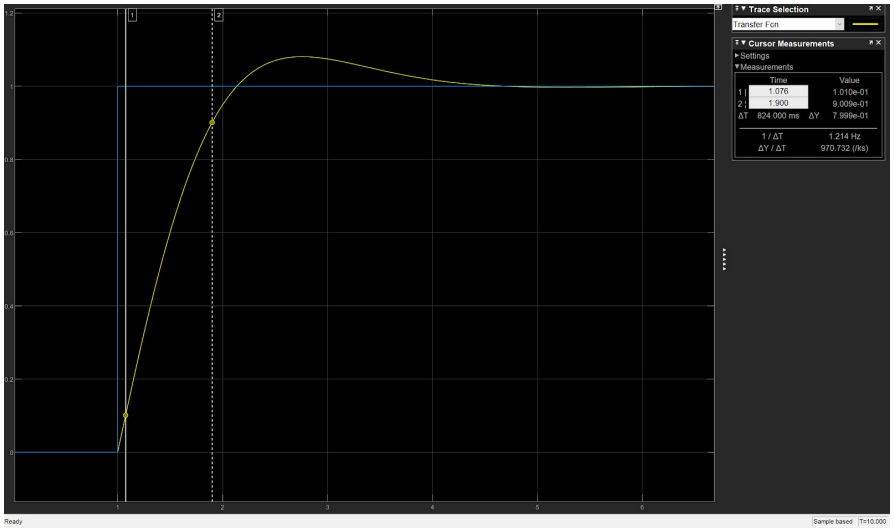

Figure F.2: Step of response of system. Note that the rise time of the output is approximately 0.8 sec.

Noting the rise time in the Figure above, the MTL formula we want to falsify is

$$r_{10} \rightarrow \Diamond_{[0,0.1]} r_{90}$$

which reads as "when the output is greater than 10% of the input, the output will be greater than 90% of the input some time between 0 and 0.1 seconds."

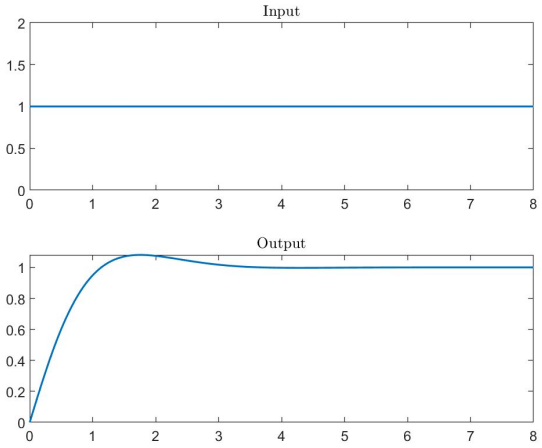

Figure F.3: S-TaLiRo output

Although the S-TaLiRo output is similar to that in Fig. F.2, the MTL formula was not falsified. We expect the formula to be falsified because the simulated rise time was found to be 0.82 seconds. The optimization method used was uniform random sampling for a total of 100 tests. The best (minimum) robustness value was found to be 0.0029261 at a program run time of 203 seconds.

Using the *always* operator, the MTL formula was then changed to

$$\Box(r_{10} \rightarrow \Diamond_{[0.8,0.85]} r_{90})$$

which reads "it will always be true that when the output is greater than 10% of the input, the output will be greater than 90% of the input some time between 0.8 and 0.85 seconds."

At a run time of 12.8655 sec, the best robustness was found to be -0.1659. Hence, the MTL formula was falsified. We expect the formula to be satisfied because the simulated rise time was found to be 0.82 seconds.

Experimenting with and without the *always* operator, the S-TaLiRo output contradicted the simulation for both cases.