Design of the DSSL Testbench Hardware Architecture for Rapid Development and Execution of Hardware-in-the-Loop Control

Clayton Chu

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Aeronautics and Astronautics

University of Washington

2012

Reading Committee:

Mehran Mesbahi, Chair

Adam Bruckner

Christopher Lum

Program Authorized to Offer Degree:
Aeronautics & Astronautics

University of Washington

Abstract

Design of the DSSL Testbench Hardware Architecture for Rapid Development and Execution of Hardware-in-the-Loop Control

Clayton Chu

Chair of the Supervisory Committee:
Professor Mehran Mesbahi
UW Aeronautics & Astronautics

This thesis discusses the development of the MATLAB-based Distributed Space Systems Lab (DSSL) Hardware Architecture (or DHA), a software stack and design methodology to allow for rapid hardware and software development for controlling vehicles in MATLAB. The layers of the architecture are discussed in levels of increasing abstraction. The interaction between a MATLAB vehicle object and other objects is also considered, including interaction with controllers designed in Simulink. Several examples of the entire development cycle of a vehicle are provided (hardware, software, mathematical modeling), including a ground-based, two-wheel differential drive robot; and a two-propeller, vectorable-thrust blimp. The possibility of using the DHA for multi-agent systems is also discussed. A developer's guide is also provided.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF CODE SAMPLES

ACKNOWLEDGMENTS

The author would like to thank several people for their contributions to the development of the architecture presented in this thesis, and the hardware work that will be based on it.

Prof. Mehran Mesbahi has been extremely supportive of the author's efforts to synthesize the hardware architecture.

Josh Maximoff has acted as a great advisor of sorts, both in terms of MATLAB experience and as a computer engineer generally.

Georges-Henri Baron developed an underlying Java code to directly interface with Vicon, allowing users to pull timestamp and velocity information.

Airlie Chapman has been quite helpful in debugging the Arduino on the hardware side, as well as teaching how to use it in the first place.

Andrew Girardeau-Dale and Bao Le have been instrumental in testing the architecture by designing and building functioning hardware based on the stack.

The entire Distributed Space Systems Lab at the University of Washington has been nothing but supportive of the efforts to implement the hardware architecture.

Finally, Dr. Giampiero Campa from The MathWorks was gracious enough to answer questions about, help debug, and even fix upstream bugs in the ArduinoIO interface that the architecture is based on.

Chapter 1
INTRODUCTION

Control theory research tends to focus on the theoretical and mathematical aspects of control, with sometimes little regard for the execution of that control in real, physical systems. One reason for this is because of the non-control work required to successfully develop hardware systems; the focus of the control engineer should be on the control theory, not on programming embedded microcontrollers, at least in the early stages of the development cycle. Computer engineering issues such as dealing with poor floating point performance (requiring use of an off-chip math coprocessor); memory management; implementation of a finite state machine for cooperative multitasking; and the fact that a microcontroller has limited program and data memory all make rapid development of hardware and controllers cumbersome at best. Furthermore, these embedded systems do not allow control engineers to leverage powerful tools such as MATLAB and Simulink to implement control. There are software packages to compile Simulink models to embedded C code[13][14], but there are strict requirements on those models and only a small subset of MATLAB/Simulink may be used for embedded applications.

This is not to say that the computer engineering requirements for synthesizing controllers are irrelevant. Certainly in the later stages of development, the controller needs to be properly implemented to work within the framework of embedded systems. But in the early stages, controller design should not be slowed down or impeded due to these considerations.

This thesis presents a MATLAB-based architecture that addresses these issues. This architecture allows users to focus on the control engineering aspects of hardware

design and not consider the computer engineering difficulties. Once hardware is designed, users can rapidly develop software and controllers with minimal downtime for reconfiguring or recompiling code. It is the author's hope that this architecture can become a good first step in the hardware controller development cycle.

The goals of this architecture are as follows:

- **Make software engineering a lower priority**. Control engineers should be focused on control, not on the minute details of making robots respond to commands. The architecture should hide computer engineering details from the user to let him focus on high-level design and implementation.

- **Make controller implementation fast and simple**. Users should be able to rapidly development and run their controllers with zero downtime for hardware. Ideally, a user could just click "Save" after editing his Simulink controller, and the hardware would respond immediately.

- **Provide a framework to make projects manageable**. The three major components of a hardware controls project - hardware development, software development, and controller design - are certainly interdependent. But the framework presented allows for a segregation of each subsystem. This lets teams take a divide-and-conquer approach to development, with systems integration as a final step rather than required throughout the development cycle.

Report Structure

This document is comprised of an overview of the MATLAB-based hardware architecture and design methodology; some case studies that encapsulate the entirety of a project developed within that design methodology (i.e., hardware; software; and mathematics); and a discussion about the possibilities and pitfalls for using the DHA

in ways not developed so far. A user's guide at the end of this document allows developers to quickly start building vehicles in MATLAB. Developers only really need to be familiar with Chapters 2 and 3 and Appendix A to quickly begin, but reading other chapters will provide context and guidance as to how to successfully complete a hardware project.

Chapter 2 presents the DSSL Hardware Architecture (DHA). The architecture and accompanying design methodology is what allows developers to quickly design and build Arduino-based hardware and run control. Principles of object-oriented programming and how they relate to the architecture are presented. Each layer of the architecture is discussed from low-level to high level, and users can see how they fit into the context of software/hardware design. Also outlined is how the vehicle communicates with external processes, in particular the conduit for which Simulink-based controls may be designed and implemented on hardware.

Chapter 3 provides a software example: the architecture is used to simulate hardware-in-the-loop (HIL) control of a simple two-dimensional spring-mass-damper system. A simple model for the system is developed, and a controller designed via pole placement is used for both software-in-the-loop (SIL) and "simulated HIL" regulation of the system. Comparisons between true SIL and simulated-HIL are discussed, and in doing so the architecture is shown to be successfully commanding control properly.

Chapter 4 presents the first hardware developed with the DHA: an Arduino-powered blimp (known as Blimpduino). All phases of development - hardware design, software design, and mathematical modeling of dynamics and control - are discussed. The dynamics and stability properties of the blimp are derived and simulated, and an LQR-based controller is applied for regulation and tracking. SIL simulations are compared to real-life HIL execution.

Chapter 5 discusses the Frisbee bot, a differential drive ground vehicle. A simple PI controller was implemented to drive the bot to arbitrary waypoints. A mathematical model for the bot is shown, and software simulation is compared to hardware

execution. This bot is a good example of the power and flexibility of the DHA - the code used to run the robot live is straightforward and simple to read.

Chapter 6 presents a possible way to use the DHA for multi-agent systems. The main area of focus for the DSSL is controlling distributed networks of systems; this chapter discusses how users could set up their MATLAB environments to facilitate selective communication between agents via adjacency matrices. The difficulties and pitfalls of using the DHA in its current form to do so are also described, both in terms of hardware and software.

Appendix A is a tutorial meant for new developers to learn how to design MATLAB `DSSLVehicle` objects. It is a step-by-step guide to familiarize users with the DHA code base and get them started on designing high-level `Vehicle` objects and running control on them.

Chapter 2

THE DSSL HARDWARE ARCHITECTURE

The DSSL Hardware Architecture is a MATLAB/Simulink-based software stack and accompanying design methodology to perform hardware-in-the-loop control without need for more than basic electrical and computer engineering skills, while utilizing experience with MATLAB and Simulink. This stack and design methodology is simple enough for students to learn the basics of controlling physical plants, yet powerful enough to implement the most complex of controllers. Systems designed with this methodology may be rapidly developed and are easy to change or augment. Once the design of the software for a vehicle is completed, users may simply create Simulink controllers for those vehicles and watch them operate in real-time. Changes to the Simulink model can be done in real-time, and these changes are reflected with zero downtime to reprogram hardware. The architecture, then, offers truly rapid development of hardware and controllers.

There are three representations of a "vehicle" or "system":

- **Plant**: The physical implementation of the vehicle to be studied. Note that this is different than a "plant model", which is a mathematical construct. This is also known as the *hardware* or *platform*.

- **Model**: The mathematical representation of the underlying dynamics and control the vehicle will undergo.

- **Object**: The software representation of the vehicle in MATLAB.

In this methodology, each representation is needed to run real-time control on a

real-life system. Each representation and their interactions with each other will be described at length in this chapter. Each representation is separate, and as such, members of a large team can break into subteams to work on separate pieces of the system. In the end, all members of the team must work together for the vehicle to operate.

2.1 System Representations and Design

2.1.1 The Physical Plant

The plant representation of the system is the most straightforward. The software architecture requires that the plant is based on the Arduino microprocessor; this is to leverage the ArduinoIO application programming interface (API) released by The MathWorks[12]. This API contains C code to be compiled and uploaded to the Arduino and allows the Arduino to be controlled via MATLAB. With this API, typical Arduino-C commands such as `analogWrite()` can be performed live in MATLAB.

Creation of plants essentially consist of wiring actuators and sensors to the Arduino. While this may be a challenge all on its own, this process is no different than designing the physical plant without the software architecture. Thus, physical plant design is outside the scope of this thesis.

2.1.2 The Object

The software object representation is designed via object-oriented MATLAB. The architecture is designed to mirror the hardware design as much as possible; for instance, one might put an H-bridge motor controller on pins 6-12 of the Arduino, then in software instantiate an `HBridgeMotor` and attach it to the Object. The software architecture is heavily dependent on The MathWorks' `arduino` class. Figure 2.1 depicts the architecture; each layer only interacts directly with the layers above and below it. The implementation of this architecture will be discussed in Section 2.2; at a high

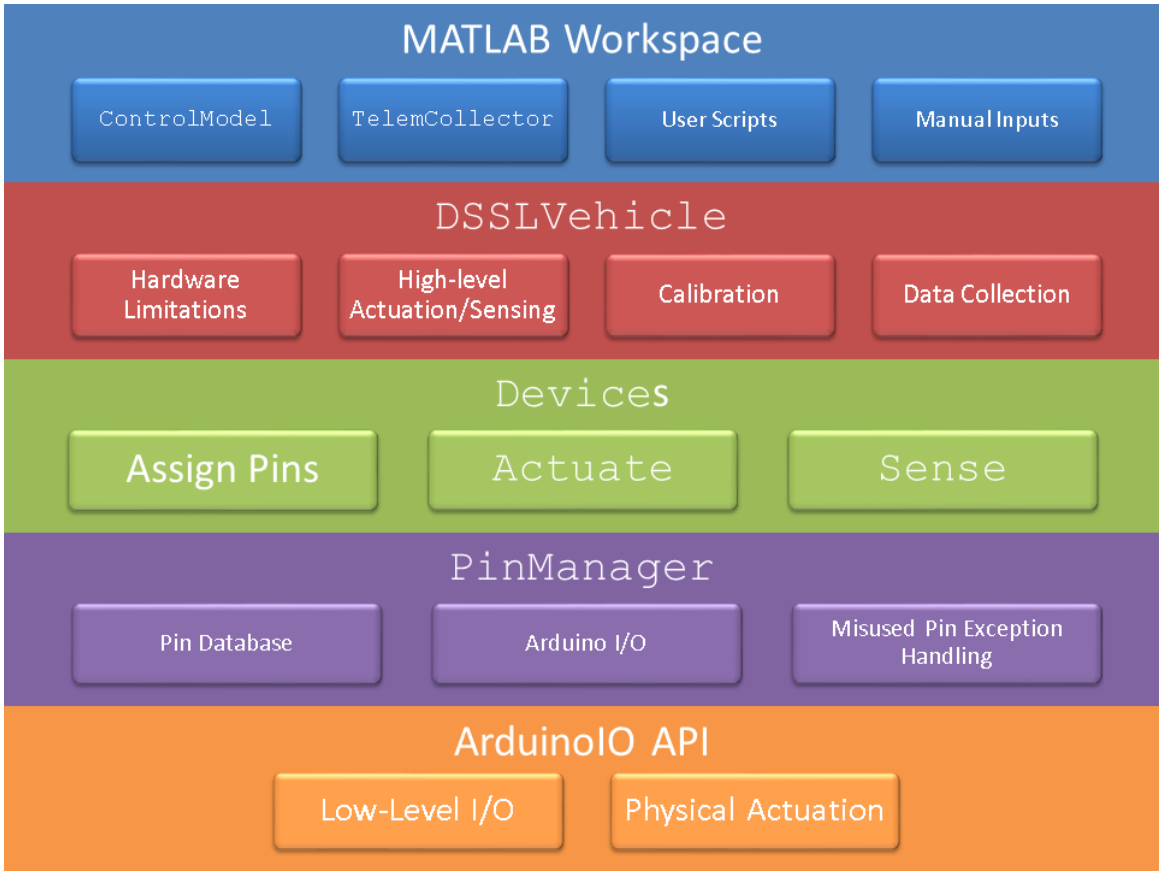


Figure 2.1: DHA System Architecture. The architecture is completely written in MATLAB. Each layer may only interact with the layer above and below it.

level, users only need to design the `DSSLVehicle` and maybe `Devices` if more are needed. No interaction with the `PinManager` or `arduino` objects is required; details about low-level implementation are therefore safely encapsulated from the user.

A basic `DSSLVehicle` class does the following:

- **Mimic design of hardware**. Just as adding a motor to the plant may be done by connecting pins to the right places, in software the call `DSSLVehicle.Motor = HBridgeMotor(pins)` is used to instantiate an `HBridgeMotor` device and connecting it to `pins` specified by the user. The `HBridgeMotor` object is then accessible by examining the `DSSLVehicle.Motor` data property.

- **Make it easy to use devices manually**. `Device` subclasses such as `Servo`s implement their own `actuate()` methods. There is no need to directly interact with the Arduino; rather than manually write signals to pins, one just calls `Device.actuate()`. `DSSLVehicle` actuation methods are just wrappers around `Device.actuate()` methods. This same setup is true for `Sensor`s and their `sense()` methods.

- **Quickly create new `Device`s**. Users simply inherit from the `Actuator` or `Sensor` class and implement their own `actuate()` or `sense()` functions.

- **Hide the `arduino` from the user**. A `PinManager` class is instantiated with every `DSSLVehicle`. It maintains pin assignments for `Device`s and ensures that `Device`s such as `Actuator`s cannot be set incorrectly, i.e., no reading of `Actuator`s and no writing to `Sensor`s. (An `InOut` class is available for devices that might require both reading and writing signals.) It also ensures that pins do not get overwritten; if a `Motor` tried to share the same pins as a `Sensor`, the `PinManager` will throw an exception. (The standard `arduino` will silently overwrite the first pin assignment instead.).

In tandem with the `DSSLVehicle` class, other classes are useful to assist users in visualizing vehicle dynamics and running control:

- `Vicon` and `ViconSubject`: An interface for a Vicon motion capture system. The Vicon system employs several infrared cameras to track IR beacons in space. With these classes, users can obtain position and heading information from IR beacons physically attached to the hardware.

- `TelemCollector`: A timer object which directs a `DSSLVehicle` to execute its `getTelem()` method, the data collection method.

- `ControlModel`: A wrapper for a Simulink model containing a controller. Given the system's current state, this will calculate the next control input u to be executed by the `DSSLVehicle`. Usage of this class will be discussed more thoroughly below.

Figure 2.2 depicts how these "helper" objects interact with the `DSSLVehicle`. Essentially, the `DSSLVehicle` only receives inputs and acts accordingly. An input from `TelemCollector` tells the `DSSLVehicle` to collect data, some of which may be from the `Vicon`; an input from `ControlModel` actuates the `DSSLVehicle`.

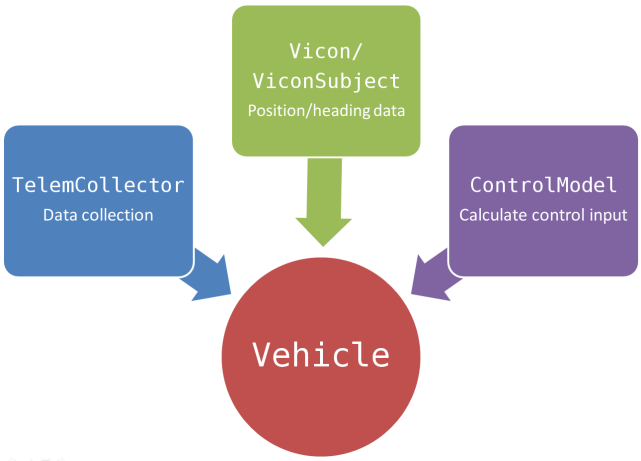


Figure 2.2: Interaction between `DSSLVehicle` and Other Objects

The base `DSSLVehicle` class is abstract and requires users to create their own concrete vehicles - i.e., to design their own software to interface to their hardware. All `DSSLVehicle` subclasses must export the following methods:

- `setupDevices()`: instantiate `Devices` to attach to the `DSSLVehicle`; assign pinouts and do any device initialization. This method is called on `DSSLVehicle` construction.

- `setupTelemetry()`: set up what data is to be collected and stored (and possibly examined either during analysis or live). This method is called on `DSSLVehicle` construction.

- `getTelem()`: collect the data. This method will be called on a clock defined by a `TelemCollector` object; this allows users to specify a data sampling rate for

data collection.

- `getCurrentOutput()` and `getCurrentInput()`: return current output y and control input u as sensed by the plant. These are inputs to a `ControlModel` object used to calculate control inputs.

- `convertUToMethod()`: translate an input vector u calculated by the controller (which is wrapped by a `ControlModel` object) into physical actuation. This method is the callback function for an `event.listener` object instantiated on `DSSLVehicle` construction. The callback executes when a `ControlModel` object calculates a new input u.

`DSSLVehicle`s should also contain actuation methods, such as "`runMotor()`" or "`pitchServo()`". These are just convenient wrappers for underlying `Device.actuate()` methods, so they are not totally necessary. But they are useful to ensure sane inputs to `actuate()` methods, or do pre- or post-processing of inputs, saving input history, etc. Additionally, "safe mode" functions should be used to shut down the vehicle in case of extremely low battery to protect the battery. `getTelem()` should call some `enterSafeMode()` method if it detects a low battery during routine data sampling.

2.1.3 The Model

The software architecture allows users to use a Simulink model for both software- and hardware-in-the-loop simulation. This is done by way of the `ControlModel` class. A `ControlModel` object operates on a timer and executes the Simulink model on timer expiration; users can select an operating frequency at which to calculate control inputs. This also means that by definition, the underlying system dynamics are assumed to be *discrete* rather than *continuous*. (However, using a continuous model rather than a discrete one may yield satisfactory results.)

The Simulink model for use with hardware-in-the-loop control contains only an observer and a controller. The inputs to the model are u_k and y_k, the previous control input and plant output (e.g., position, heading, etc). These are typically returned from the `DSSLVehicle` via `getCurrentInput()` and `getCurrentOutput()`, which is more robust than simply relying on the `ControlModel` to maintain that information. This is because the Simulink controller may not be aware of hardware limits on u, and unknown disturbances may vary y. The `DSSLVehicle` directly polls its sensors and actuators, so `ControlModel` can just ask for this information.

However, the state variables x are *not* necessarily available to the `DSSLVehicle`, so the `ControlModel` must maintain that history. In any case, the history of both the state estimation $\hat{x}$ and u as calculated by the `ControlModel` are maintained. Assuming the observer was built to be stable, $\hat{x}$ will converge to x.

The `ControlModel` will operate the Simulink model in a somewhat different fashion than a user might for software-in-the-loop integration; the needs of hardware-in-the-loop functionality dictate the program flow control:

- `ControlModel` imports user variables that were previously stored in `ControlModel.SimulinkVariables` into the Simulink workspace. This property is modifiable at any time and is useful to store parameters such as waypoints and controller gains that should be changeable on the fly. The current timestep `ControlModel.k` is also imported.

- `ControlModel` simulates exactly *one* timestep; that is, given an input u_k and output y_k (retrieved from the `DSSLVehicle`) the `ControlModel` will return u_{k+1} and $\hat{x}_k$, the estimated state. This timestep is equal to the period at which the `ControlModel` is run. (This also implies that variable-timestep solvers such as `ode45` are not applicable; indeed, `ControlModel` uses the fixed-timestep `ode1` integration method, for reasons that are discussed in Chapter 3.)

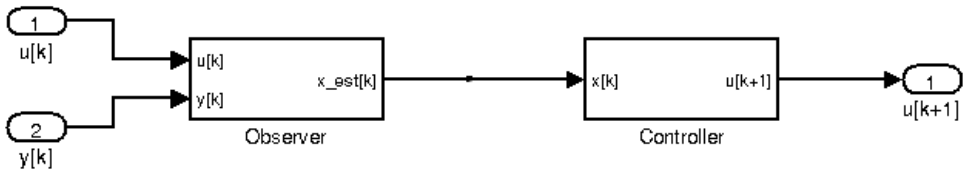


Figure 2.3: Observer/Controller Simulink Model for Hardware-In-The-Loop Control

- `ControlModel` forces Simulink to simulate the model for time $[t_0, t_1]$. `ControlModel` sets the model's initial state to the previous state estimate. (When $k = 0$, an externally supplied initial condition will be used; if none is supplied, it is assumed to be $\mathbf{0}$. If the observer is stable, the state error goes to zero so this assumption should not adversely affect the system for long.) Restrictions in model design could be added to simply integrate over $[t_k, t_{k+1}]$ for arbitrary k, but one goal of this architecture design is to be able to drop in arbitrary Simulink models without having to think of changes between software- and hardware-in-the-loop modeling. `ControlModel` tries to hide these differences and work them out behind the scenes to accomplish this.

  - The states of the system tend to just be position and velocity, so one obvious choice for the initial condition is the current position as detected by sensors and 0 for the velocity.

- After simulation is complete, `ControlModel` calls the `Vehicle.controlUToMethod()` method and converts u_{k+1} into actuation method calls. The method calls are then executed and actuators on board the physical plant are actuated.

Figure 2.3 depicts the prototypical observer/controller Simulink model for use with `ControlModel`. Users need to design the Observer and Controller subsystems,

but that needs to be done for software-in-the-loop simulation anyway. Both time-invariant and time-varying systems are supported (time-varying systems have access to the current timestep and frequency of control execution). Note that variables that may be imported such as waypoints or controller gains are not depicted here; they are made available in the `runControl()` workspace before controller execution.

2.2 Software Architecture Design

The software architecture presented is done in an object-oriented programming style. This is because the fundamental tenants of OOP - abstraction, encapsulation, and inheritance[10] - naturally fit the end goals of this project.

- **Abstraction**: Each layer of the architecture has no knowledge of how the layers above and below it are implemented. Users care about `DSSLVehicles` with attached `Devices`, but they do not care about how exactly the `Devices` work, just that they do. Motors turn on and off; the fact that motors are turned on and off by sending PWM inputs to the Arduino is unimportant for typical use.

- **Encapsulation**: Encapsulation ties strongly with abstraction. Each layer presents an interface to the layers above and below it to access the internals of that layer. `Devices` have `actuate()` and `sense()` methods that users can call to perform actuation/sensing. How this is done might be changed as development evolves, but the interface protects callers from those changes inadvertently breaking code.

- **Inheritance**: All vehicles are different: some fly and some drive. But they share basic characteristics - they all need to have pinouts set up; they all have data to collect; they all move; etc. Creating a common base class - the `DSSLVehicle`

class - ensures that while each of the derivative subclasses are different, the interface for how to use the `DSSLVehicle` is the same. For instance, a `ControlModel` broadcasts a control signal to execute, and the common interface ensures that the control signal is caught and read properly. How each `DSSLVehice` subclass executes the control signal is different, but how it receives that control signal is the same. Making use of inheritance results in less code being written, which ultimately means fewer bugs and less complexity.

Ultimately, the heavy reliance on OOP techniques means users spend more time rapidly developing vehicles and control for vehicles, rather than dealing with what could be a very complicated computer engineering project.

Each layer of the software architecture is discussed from lowest-level to highest-level in the next sections.

2.2.1 Hardware and `arduino`

The MathWorks' ArduinoIO API consists of `arduino.m` and `srv.pde`; the latter file is uploaded to the Arduino. The `arduino` object is essentially a communications layer to the hardware; calls such as `arduino.digitalWrite()` are translated into an opcode and transmitted to the Arduino via serial connection (either via USB or wireless XBee radio); `srv.pde` monitors the Arduino's serial line and executes commands based on the incoming opcode it reads[12]. `arduino.pde` is written in C.

So an `arduino` object allows a user to directly write or read the digital I/O pins, and essentially allows one to write prototype code in MATLAB to be interpreted on the fly rather than compiled and uploaded. This allows rapid changes and tweaks to `DSSLVehicle` design, which is one of the goals of this architecture. However, this presents one possible drawback to using this architecture: interpreted languages are inherently slower than compiled languages, so users must accept a performance penalty in terms of speed of execution. This is more than offset by the fact that

writing MATLAB code for control is significantly less complicated than writing it in C or using potentially expensive MATLAB addons to compile code to C. To write embedded C code effectively, a user would have to consider the performance costs and workarounds for doing complex math (and in this context, floating-point math is considered complex); an off-chip floating-point unit would need to be used, and a host of computer engineering issues would detract from designing the controller.

Using ArduinoIO allows the user to treat the Arduino as a dumb device which does no calculations and only writes/reads digital and analog I/O. Note that in this configuration, using more complicated devices that use SPI or I^2C communications protocols cannot be used; one would have to manually interface these devices and have them talk directly to MATLAB. This issue is not yet addressed in the software architecture's current form.

2.2.2 Hardware Abstraction Layer: `PinManager`

As `arduino.m` only acts as a hardware communications layer, it offers no protection to users from accidentally misusing it. For instance, attempting to write to a pin that was set as input will silently fail, but will *not* throw an exception - and the user will have no idea if the failure to write to the pin was a software or hardware error.

Moreover, in the OOP paradigm, the architecture should hide unimportant details from the user. Users should not need to know what pins to read/write from a device; they should be able to write/read a signal from the device itself. Once the pins are known, the user should never use them again. The `PinManager` object is used as a hardware abstraction layer to implement this information hiding.

`PinManager` has numerous uses, both as a hardware abstraction layer and as a layer of protection for the user:

- **Assign pins**. Pins are bound to a `Device`. Each individual pin is assigned a unique identifier called a *binder*. This abstracts away the unimportant informa-

tion: it does not matter what pin is being read or written to so much as how that pin is being used, and the binder expresses the pin's functionality. For instance, a pin that controls the PWM output on an H-bridge motor might be given the name `Motor1_pwm`. Once a pin is assigned to a binder, the pin can only be used by that binder until they are released back to the `PinManager`.

- **Protect pins from inadvertent I/O**. When binding pins to binders, the I/O type must be declared; `PinManager` will throw an exception if a user attempts to read a pin designated as output, or attempt to write to a pin designated as input. This is contrast to the naive `arduino`, where an incorrect I/O assignment might change the pin designation from input to output or vice versa, or silently fail.

- **Protect pins from misuse**. Once a binder is attached to a pin, only that binder may use the pin until specifically released. This protects the user from accidentally attempting to use pins on separate binders without first resolving the fact that there exists two separate objects attempting to access the same pins. Physically, devices typically cannot share the same physical pins on an Arduino, and this process is mirrored in software as well.

- **Ensure pin assignments are physically possible**. There are a limited number of pins that may be used as PWM or servo outputs, and generally speaking a limited number of digital and analog pins. There are also limitations inherent in the ArduinoIO library: for instance, if pin 9 is registered for use by a `Servo`, pin 10's PWM functionality is automatically disabled due to limitations within the `Servo.h` library that ArduinoIO uses. `PinManager` ensures that illegal assignments (attempting to access a pin that doesn't physically exist, etc.) do not happen.

The `PinManager` class prototype is shown in Code Sample 2.1, below.

Code Sample 2.1: `PinManager` Class Prototype

```
1  classdef PinManager < handle
2
3      % Database of pin assignments and reserved PWM/Servo pins
4      properties
5          digitalPins
6          analogPins
7          PWMPins   = [3 5 6 9 10 11]
8          ServoPins = [9 10]
9      end
10     methods
11         function obj = PinManager
12
13         % Assign a pin to a named device
14         function assign(obj, pinNum, device, binder, mode)
15
16         % Release a pin assigned to a binder
17         function release(obj, toRelease, type)
18
19         % Check if pin is assigned to a device
20         function [isAssigned pinData] = isAssigned(obj, pinNum)
21
22         % Find pins assigned to a device, show pin data
23         function [pins binders] = findDevice(obj, device)
24
25         % Find pins associated to a binder (named device)
26         function [pin device]   = findBinder(obj, binder)
27
28         % Get array of assigned pins
29         function allPins = getUsedPins(obj)
30
31         % Write a signal to a binder
32         function writeSignal(obj, binder, signal)
```

```
33
34          % Read a signal from a binder
35          function val = readSignal(obj, binder)
36      end
37  end
```

This hardware abstraction layer is used extensively at the `Device` level, but is designed such that a user should never have to interact directly with it in the typical development cycle.

2.2.3 Using Peripherals: `Devices`

`Devices` represent in software the physical actuators and sensors that might be added to a vehicle, such as motors, servos, ultrasonic sensors, etc. `Devices` are how `DSSLVehicles` interact with the world; as such, they are the next layer up in the presented software architecture in Figure 2.1. They are another layer of abstraction - users `actuate()` the `Actuators` and use `Sensors` to `sense()` the world around them; users should not worry about writing or reading signals, they should be worried about running actuators and reading sensors. The `Device` class encapsulates the complexity of writing and reading signals from the user.

The `Device` class is actually an abstract class, and its subclasses (`Actuator`, `Sensor`, and `InOut`) are also abstract classes. Each subclass exports a single method (or two methods, in the case of the `InOut`): `actuate()`, `sense()`, or both, respectively. These methods are simply wrappers around `PinManager.writeSignal()` and `readSignal()`, but they may do pre- or post-processing on input arguments.

Upon construction, `Devices` register with the `PinManager`. `PinManager` will throw an exception if pins had been previously reserved by other `Devices`. In addition, users may specify software limits such as min/max PWM and sensor capabilities. For instance, `Battery < Sensor` is used to read a battery voltage on an analog input. Since the Arduino can read a maximum of 5V on its output, a voltage divider must

be used to scale the true battery voltage to 5V. A multiplier option in the class lets users specify how the Arduino should scale its 0-5V input.

A sample `Device` class prototype is shown in Code Sample 2.2, below.

Code Sample 2.2: Sample `Device` subclass Prototype

```
1  classdef Actuator < Device % Device subclasses are Actuators, ...
       Sensors, or InOuts
2      properties (Abstract = true, SetAccess = protected)
3          Type    % will contain the name of the concrete subclass ...
               (HBridgeMotor, etc)
4          Binders % names of pins to be assigned
5      end
6
7      properties (Abstract = true)
8          Name  % unique name of the device instance - something like ...
               "Motor1"
9          Range % maximum PWM range of an actuator - typically [0 255] ...
               is okay
10     end
11
12     methods
13         obj = Actuator(PM, pins) % instantiate the object -
14                                  % feed pin assignments to the ...
                                       PinManager
15     end
16     methods (Abstract = true)
17         actuate(obj) % Actuate the device - call underlying PinManager
18                      % write/readSignal methods
19         end
20 end
```

2.2.4 The Big Picture: `DSSLVehicle`s

`DSSLVehicle`s are the reason this software architecture exists. Really, a `DSSLVehicle` is nothing more than a combination of `Device`s in one discrete package. It may be thought of as an interface that allows external processes (whether they be human interaction or other objects) to access `Device`s. For example, a `ControlModel` may calculate a control input to run and send that to the `DSSLVehicle`; the `DSSLVehicle` parses that control signal and runs `Device.actuate()` methods corresponding to the control signal. A `TelemCollector` used for data collection may tell the `DSSLVehicle` to `getTelem()` - to process `Sensor`s and retrieve position and heading information from the `Vicon`. In the spirit of abstraction and encapsulation, the `DSSLVehicle` is essentially just a collection of `Device`s.

Of course, `DSSLVehicle`s are a bit more nuanced than that in order to handle passing messages between `DSSLVehicle`s and other objects that need to talk to them. Messages are passed via `notify()` calls. This way objects do not "reach into" other objects to retrieve information. `ControlModel` objects calculate control inputs but should not directly run those control inputs on the `DSSLVehicle` itself. `TelemCollector` objects tell the `DSSLVehicle` to run its data collection methods, but should not access that data collection method itself.

`DSSLVehicle`s contain the following properties and methods:

- **Setup methods**: Methods to create pin assignments and define what telemetry will be collected.
- `Devices`: Handles to `Device` objects.
- `Device` **limits**: Ranges to further limit `Device` actuation/sensing. For instance, an `HBridgeMotor` may be capable of a maximum PWM value of 255, but the user desires to use it at no higher than a maximum PWM of 100.

- **Telemetry**: Collected position/sensor data over time.
- **Actuation methods**: Wrappers for `Device.actuate()` methods that incorporate custom limitations. May also push the resultant calls into telemetry.
- **Sensor methods**: Wrappers for `Device.sense()` methods.
- **Safe mode methods**: *(optional)* Shut down the plant (turn off motors, servos, etc) in case of emergencies, i.e., extremely low battery.
- **Calibration methods**: Methods to convert between a control input u and the equivalent PWM.
- **Higher-level methods**: Methods which are called upon from external processes a `ControlModel` or `TelemCollector` and operate at a high level. These methods include:
  - `getTelem()`: Run sensor methods and request position data from `Vicon` and store it to `Telemetry` property.
  - `convertUToMethod()`: Run actuation methods as directed by control input u. This method should contain calibration information to translate u to PWM outputs for actuators.

As long as `DSSLVehicle`s contain prototype methods and properties as listed above, they may be implemented arbitrarily and are done on a case-by-case basis. At this point, users should be able to operate their vehicles in real life and begin doing both manual and automatic control.

Chapter 3
DHA SOFTWARE AND HARDWARE TESTING

With the DHA completed, it is necessary to test for proper functionality of the software. A successful test should be able to answer the following questions affirmatively:

- Does hardware respond to actuation and sensor requests properly?
- Will a plant/controller behave in HIL execution the same way it behaves in SIL simulation?

Two simple but important tests were designed to ensure correct functionality of the DHA. A hardware test ensures that a designed `DSSLVehicle` operates correctly; a software test shows that the method of HIL control demonstrated in Chapter 2 is possible and operates correctly.

3.1 Hardware Test: Blimpduino Actuation and Sensing

A successful hardware test simply shows that the plant will respond to commands generated by MATLAB. But getting a successful result implies a lot about the `DSSLVehicle` object:

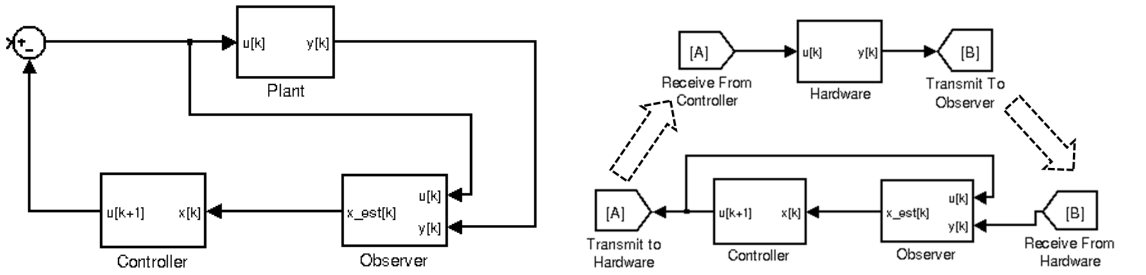
- The hardware design has been successfully implemented.
- Pin assignments were properly made in code.
- The `Devices` running on the `DSSLVehicle` are properly implemented.

- Wired or wireless communication has been successfully established.
- High-level actuation methods are valid.

Testing hardware may be as simple as running all sensing and actuation methods in a variety of edge cases. This is a consequence of debugging and may get done "just-in-time" rather than in a formal "testing phase" of a design.

3.2 `ControlModel` *Test*

The DHA is ineffective if designed controllers that work in SIL simulation cannot be used in HIL execution. Even though in HIL there will be dynamics that cannot be modeled in SIL simulation, the architecture should be able to reliably implement controllers with Simulink.



(a) SIL Simulation Block Diagram (b) HIL Execution with DHA Block Diagram

Figure 3.1: SIL and HIL Execution

Figure 3.1 depicts the differences between SIL and HIL execution. In SIL simulation (traditional Simulink model), data can be directly fed from block to block by simply connecting the proper Simulink blocks. In HIL execution, the DHA takes care of "connecting the blocks" together - the plant and controller do not know what is feeding them data, only that data is being fed for them to operate on. The DHA is operating properly if messages are successfully being passed between controller and

hardware. Note that this does not mean SIL and HIL execution will be exactly the same; it only means that the feedback loop between plant and controller is properly closed.

3.2.1 The `SimulinkVehicle`

To test the communications between a `DSSLVehicle` and a `ControlModel`, a `SimulinkVehicle < DSSLVehicle` object was developed. Since the object inherits from `DSSLVehicle`, it has all the same properties and methods as one would expect from a typical `DSSLVehicle` object, including methods like `convertUToMethod()`. But instead of doing some physical actuation, the `SimulinkVehicle` processes the input u provided by the `ControlModel` with a Simulink model, and returns the state. In this manner it is possible to do *simulated* hardware-in-the-loop (SHIL) execution.

This is different than SIL simulation. SHIL execution should behave the same way as SIL execution, but the DHA will handle transmission of messages between plant and controller. If they behave the same way, then the architecture is properly message handling for HIL execution.

Users of the DHA architecture should strongly consider instantiating their own `SimulinkVehicle` to do SHIL execution and compare this to both SIL and HIL performance.

Code Sample 3.1 shows a partial `SimulinkVehicle` code prototype. Again, it is very similar to the abstract `DSSLVehicle` class; the key differences are in using a Simulink model to perform plant execution, rather than execution in real life. Only major distinguishing features between the `SimulinkVehicle` and the `DSSLVehicle` are shown.

Code Sample 3.1: `SimulinkVehicle` Prototype

```
1 classdef SimulinkVehicle < DSSLVehicle
```

```
2       properties
3           SimulinkModel
4           SimulinkVariables
5           X0
6           Y0
7           TimeToSim
8       end
9       methods
10          function obj = SimulinkVehicle(model, varargin)
11          % SimulinkVehicle() - constructor
12          % ARGUMENTS
13          %     model     - String containing Simulink model
14          %     varargin  - List of options. See below.
15          % OPTIONS
16          %     REQUIRED
17          %         'numInps'     - number of inputs
18          %         'numOuts'     - number of outputs
19          %         'numStates'   - number of states
20          %         'X0'          - initial state
21          %         'Y0'          - initial output
22          %     OPTIONAL
23          %         'TimeToSim' - how long to simulate the model for ...
                (should
24          %         be equal to 1/ControlModel.controlFreq [s] [1/10]
25          %         'SimulinkVariables' - struct containing custom ...
                vars needed to
26          %                               run Simulink model
27          % RETURNS
28          %     obj - SimulinkVehicle object
29
30          function convertUToMethod(obj, u)
31          % convertUToMethod - simulate obj.SimulinkModel using input u,
32          %                    store results in obj.lastu and obj.lasty
```

```
33      end
34  end
```

3.2.2 Spring-Mass-Damper with Full State Feedback

To test functionality, a two-dimensional spring-mass-damper system is considered. Figure 3.2 depicts the system.

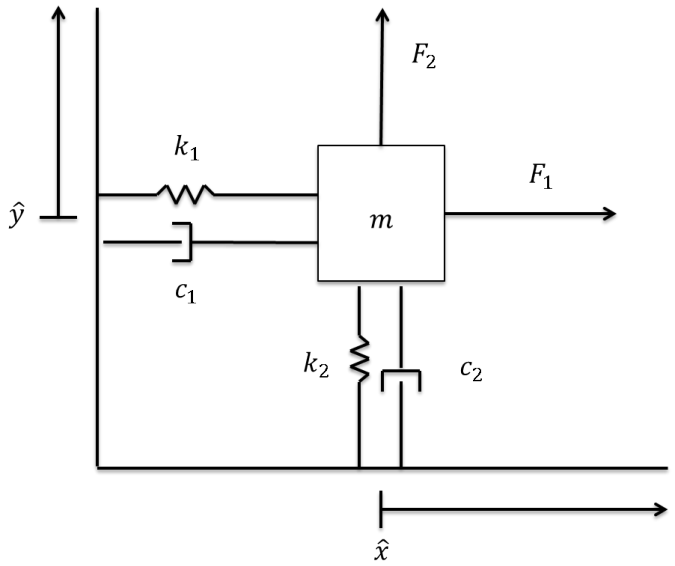


Figure 3.2: Two-dimensional Spring-mass-damper for Simulated HIL Test

z A mass m is connected to two separate springs and dampers of stiffnesses k_1 and k_2; and damping coefficients of c_1 and c_2, respectively. k_1 and c_1 are applied in the x direction, and k_2 and c_2 are applied in the y. Control inputs F_1 and F_2 are also applied the x and y directions. The equations of motion are straightforward; this is a second-order system in two dimensions:

$$m\ddot{x} = F_1 - k_1 x - c_1 \dot{x} \tag{3.1}$$
$$m\ddot{y} = F_2 - k_2 y - c_2 \dot{y}$$

Let $\vec{z} = [x\ \dot{x}\ y\ \dot{y}]^T$. The linear state-space form of Equations (3.1) is then

$$\dot{\vec{z}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{-k_1}{m} & \frac{-c_1}{m} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{-k_2}{m} & \frac{-c_2}{m} \end{bmatrix} \vec{z} + \begin{bmatrix} 0 & 0 \\ \frac{1}{m} & 0 \\ 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix} \vec{u} \tag{3.2}$$

where $\vec{u} = [F_1\ F_2]^T$.

If there is no damping, i.e., $c_1 = c_2 = 0$, then a non-zero initial condition will cause the system to oscillate about the origin forever.

If $k_1 = k_2 = m = 1$ and $c_1 = c_2 = 0$, then Equation (3.2) becomes

$$\dot{\vec{z}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{\text{A}} \vec{z} + \underbrace{\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}}_{\text{B}} \vec{u} \tag{3.3}$$

The eigenvalues of the state matrix A of (3.3) are

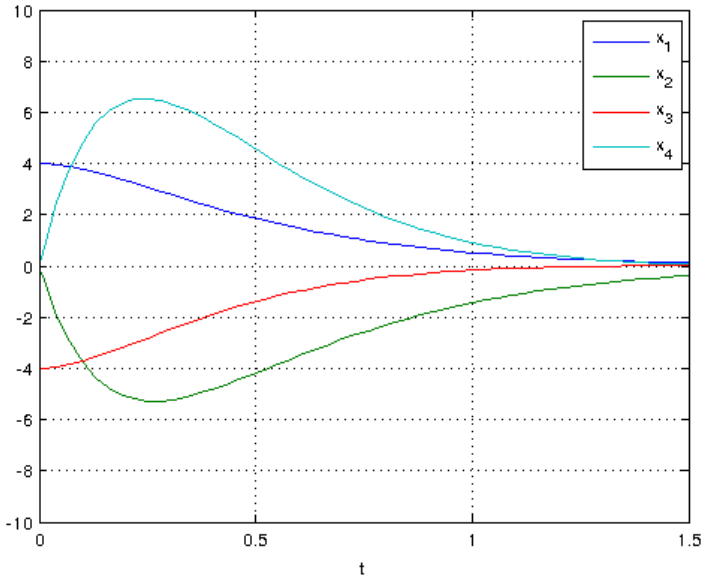
$$\lambda = \{\pm i, \pm i\}$$

Suppose it is desired to apply full-state feedback (pole placement) to move the eigenvalues to $\lambda_C = \{-2 \pm 0.1i, -2 \pm 0.1i\}$. By applying Ackermann's Formula (implemented in MATLAB as the `place()` function), a feedback law $u = -Kz$ can be found. For the desired eigenvalues, Ackermann's Formula yields

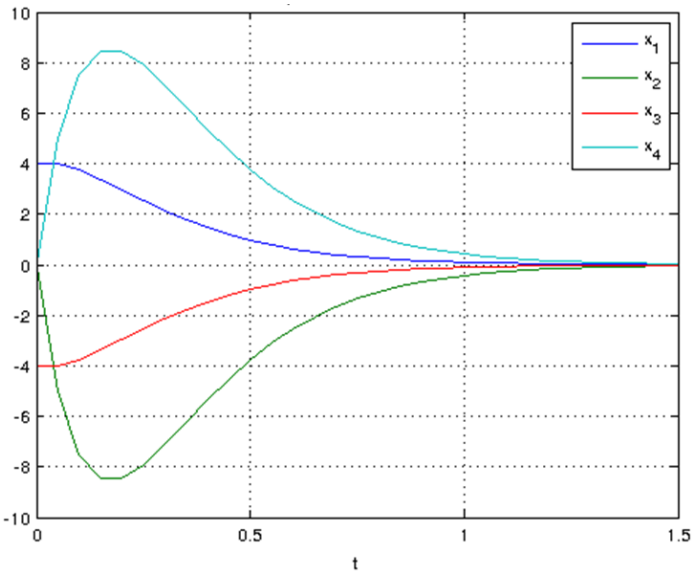
$$K = \begin{bmatrix} 3.01 & 4 & 0 & 0 \\ 0 & 0 & 3.01 & 4 \end{bmatrix} \tag{3.4}$$

3.2.3 SIL and SHIL Simulations

A traditional Simulink model was built with the state-space model from Equation (3.3) and controller from (3.4); this model was simulated using Dormand-Prince (`ode45`) and Euler (`ode1`) integration. An initial condition of $\vec{z}_0 = [4 \ 0 \ -4 \ 0]^T$ was used. The results of simulation are presented in Figure 3.3, below.



(a) Dormand-Prince (`ode45`)



(b) Euler's Method (`ode1`)

Figure 3.3: Spring-Mass-Damper with FSF: Software-in-the-Loop

A `SimulinkVehicle` was instantiated and its plant model property was set to the

state-space defined by Equation (3.3). A separate `ControlModel` was instantiated with the control law from (3.4). The `ControlModel.start()` method was called to begin data requests from the `SimulinkVehicle` for control calculation. After a short amount of time, the controller was deactivated with `ControlModel.stop()` and the state history was graphed. Results of the SHIL simulation is presented in Figure 3.4, below.

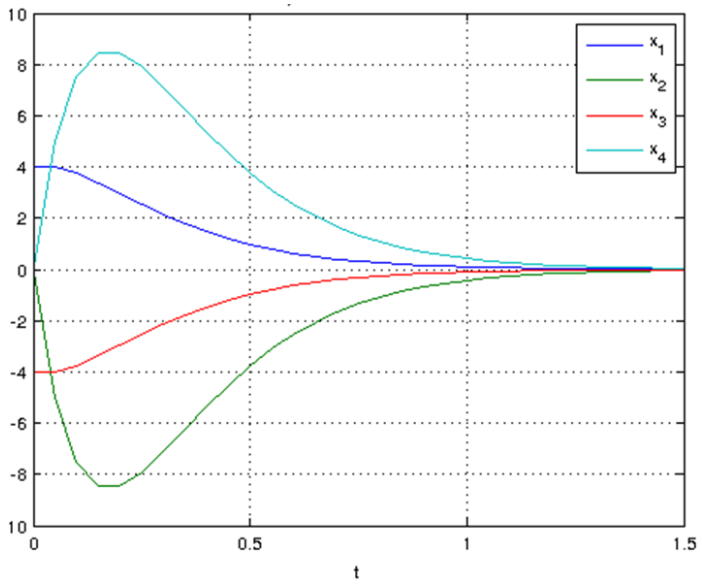


Figure 3.4: Spring-Mass-Damper with FSF: Simulated Hardware-in-the-Loop

The SHIL and SIL simulations are the exact same when using `ode1` integration, but using `ode45` integration the simulations do not compare. This behavior is at first odd, but this is actually not unexpected: physically, the `ControlModel` and `SimulinkVehicle` are acting in sequential order. `ControlModel` calculates a command input $u = -Kz$ based on single-timestep integration and `SimulinkVehicle` executes it. There is only one "function call" used to get [simulated] hardware to move from z_k to z_{k+1}, the `convertUToMethod()` call. Compare this to a more sophisticated integration technique such as RK4: RK4 uses multiple function calls to calculate the next state.

This result means that Euler integration *must* be used to compare SIL simulation to HIL execution; Dormand-Prince may not be an accurate enough representation of the HIL dynamics because of how `ControlModel` operates. This also means that the frequency of control execution must be very small to minimize error; Euler's method has an error of $O(h)$, so step sizes that are too large will lead to large error[1].

Chapter 4

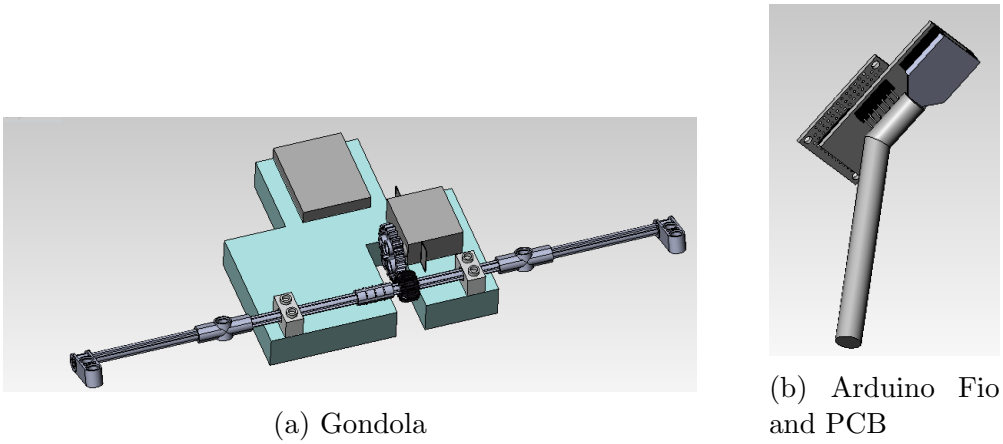
TWIN-ENGINE VECTORED-THRUST AIRSHIP: "BLIMPDUINO"

The Blimpduino is a small, indoor blimp powered by an Arduino Fio package. The DSSL hardware architecture allows the blimp to be developed and analyzed in three separate subsystems - model, plant, and object. Each subsystem will be discussed in the subsequent sections of this chapter.

4.1 Hardware Design

Figure 4.1: Blimpduino Hardware

The Blimpduino consists of a 52-inch mylar balloon; an Arduino Fio; an XBee wireless radio; two small motors; a servo; an H-bridge; a voltage regulator; a lithium polymer battery; and some resistors. The motors are spaced out 5 inches from each other and attached to a LEGO shaft connected to the servo. The completed hardware is shown in Figure 4.1. A CAD model of the physical layout is shown in Figure 4.2.

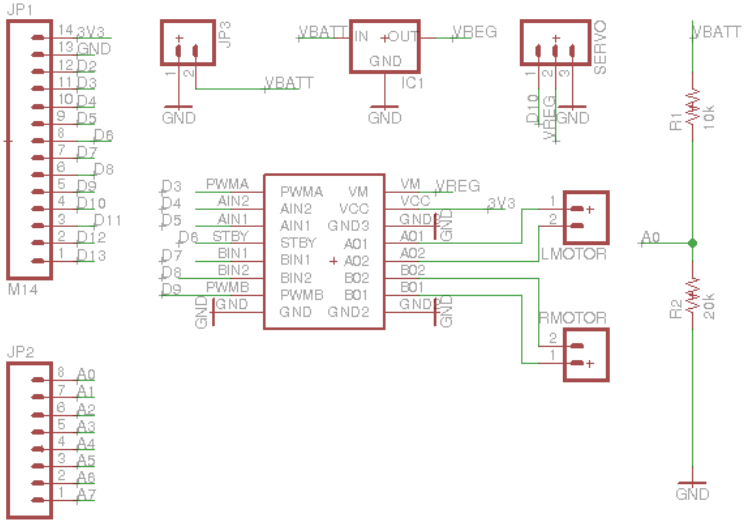


(a) Gondola

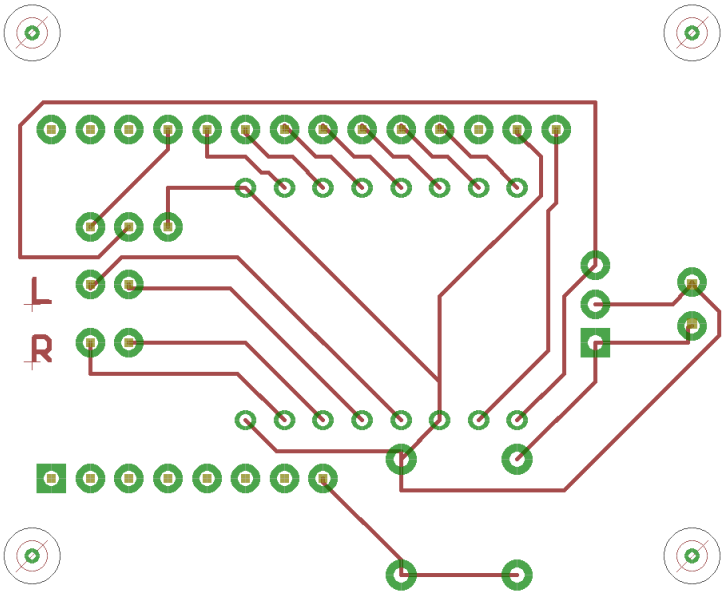
(b) Arduino Fio and PCB

Figure 4.2: Physical Blimp Layout

A PCB was designed and etched to minimize circuit weight. The circuit diagram and PCB layout are shown in Figures 4.3a and 4.3b. The Fio and PCB are draped below the gondola.



(a) Circuit Diagram



(b) PCB Layout

Figure 4.3: Blimpduino Electronics

The circuit is set up such that the H-bridge's logic inputs are adjacent to the Arduino's digital pins. The logic inputs dictate if a motor should be driven forwards

or backwards; a third PWM input simulates an analog signal to tell the H-bridge what speed to run the motors. A voltage regulator connects the lithium polymer battery to the H-bridge directly, stepping the voltage down from 7.4V to 5V. (The Arduino Fio can output a maximum of 3.3V, which is below the minimum required voltage for the H-bridge).

Two resistors in series form a 1:2 voltage divider to step down the maxmimum voltage of 5V to 3.3V. The stepped-down voltage is fed to an Arduino analog pin. This voltage gets resolved to 5V max in software; this allows the Arduino to detect the current battery level and put the vehicle in "safe mode", if necessary.

Propellers are attached to the motors and driven by an H-bridge controlled by the Arduino. This enables the motors to thrust in the forward or backwards direction. The motors are attached to a central shaft which is geared to the servo. The servo controls the pitch angle μ of the motors. This setup allows the blimp to vector its thrust. The shaft has a maximum range of 180 degrees. If a thrust angle larger than 180 degrees is required, the blimp can just reverse the direction of the thrust to compensate.

Figure 4.4 shows the calibration curves for possible propellers used by the Blimpduino. The calibration was done by setting each propeller face-up on a tared scale, then applying a voltage the motor. Motors 1 and 2 are used for the Blimpduino due to their similar characteristics when a positive voltage of 3.3V or less is applied; Motor 3 was damaged before calibration was completed and is not shown.

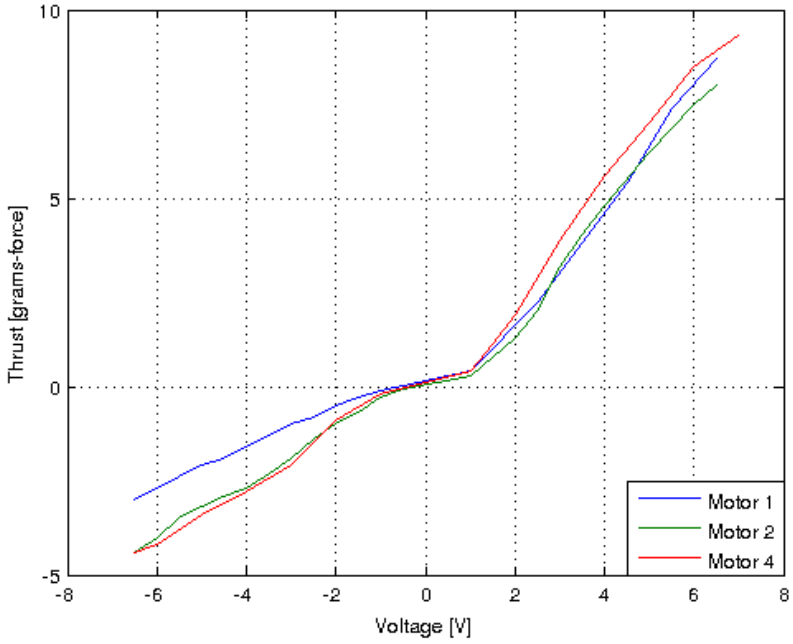


Figure 4.4: Blimpduino Propeller Calibration Curve

4.2 Software Design

The software was designed in MATLAB using the architecture and design methodology presented in Chapter 2 and Appendix A. Indeed, this vehicle is a case study in the effectiveness of using the DHA for design and implementation of hardware and control.

The `Devices` for the `Blimpduino` object include `HBridgeMotor < Actuator`, `Servo < Actuator`, and `Battery < Sensor`. Separate wrapper methods, `runMotor()` and `pulseServo()` allow users to specify a PWM or servo angle to run at. The `Blimpduino` implements telemetry collection, namely inertial position and velocity; PWM; servo angle; and battery. External processes may access these data via `getCurrentInput()` and `getCurrentOutput()`.

Since `Blimpduino` inherits from `DSSLVehicle`, most of the methods needed to run

the object are already written but only require overloading or populating. This means a "cookie-cutter" approach to writing the interface software is permissible. MATLAB will throw an exception if the user tries to build an incomplete object (i.e., the class is not completely concrete).

4.3 System Dynamical Model

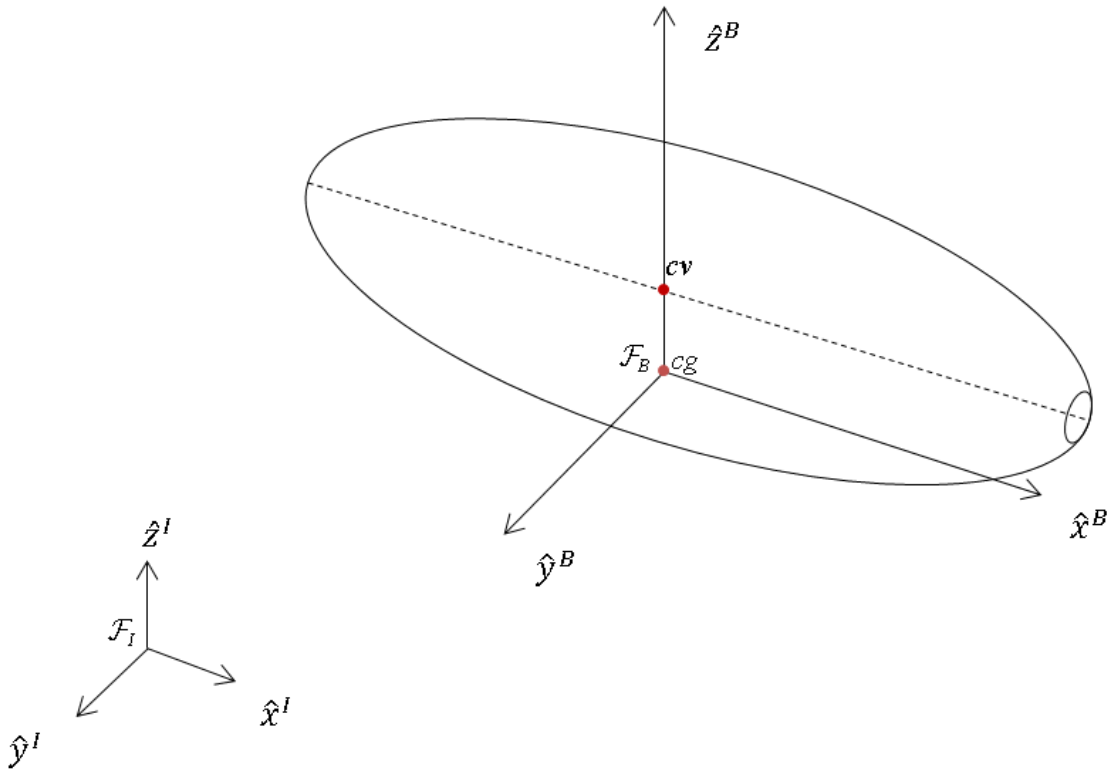


Figure 4.5: Blimp Coordinate Systems: $\mathcal{F}_I$ (Inertial) and $\mathcal{F}_B$ (Body)

An airship is like a fixed-wing aircraft in that it has six degrees of freedom: three translational and three rotational. The main difference is the generation of lift - the airship lifts by being filled with a low-density gas, rather than the motion of a wing through air. The difference in density between the air surrounding the airship and the gas inside the airship causes a buoyant force, rather than an aerodynamic lift force.

The body coordinate system is centered at the center of gravity. The $\hat{x}^B$ direction is parallel to the line that runs through the nose of the blimp; the $\hat{z}^B$ direction is up through the gondola; and the $\hat{y}^B$ direction forms the left-handed coordinate system by pointing through the starboard side of the blimp. Figure 4.5 depicts the coordinate system. (Note that the coordinate system is left-handed to follow the convention set by [15] for simplicity.)

The airship's translational and angular velocities in the $\hat{x}^B$, $\hat{y}^B$, and $\hat{z}^B$ directions are $\vec{v} = [u\ v\ w]^T$ and $\vec{\omega} = [p\ q\ r]^T$, respectively. Let $\vec{V} = [\vec{v}\ \vec{\omega}]^T$ be the vector of velocities; from Newton's second law, we may quickly write:

$$\Sigma\vec{F} = M\dot{\vec{V}} \tag{4.1}$$

The challenge is decomposing the forces and moments. To slightly simplify the model (and to account for the hardware platform the blimp is using), the following assumptions are made[15]:

- The blimp is a rigid body. Aeroelastic effects will be ignored.
- The blimp is symmetrical about $x - z$ plane.
- The blimp's center of buoyancy (cb) is located at the center of the lifting envelope; the center of gravity (cg) is below it, on the gondola.

In the upcoming sections, the forces acting on the blimp will be presented. A complete nonlinear model for the blimp is derived.

This model is adapted from van de Loo[15].

4.3.1 Mass Matrix M

Let the mass of the blimp be m, and the moments of inertia about the $\hat{x}^B$, $\hat{y}^B$, and $\hat{z}^B$ are I_x, I_y, and I_z respectively. (The off-diagonal cross-terms of the moment-of-inertia

matrix are negligible relative to the diagonal terms, so they are not included here.) Naively, the rigid-body mass matrix would be

$$M_{RB} = diag([m\ m\ m\ I_x\ I_y\ I_z]) \tag{4.2}$$

However, as the blimp moves forward it will also displace a large volume of air surrounding it. This displacement is not considered for traditional aircraft since the density of a fixed-wing aircraft or rotorcraft is significantly higher than the density of the air it travels through. For airships, the density of the envelope is comprised mostly of the lifting gas (helium or hydrogen), and this density is comparable to that of air. The effect of displacing this air, then, is not negligible.

To account for that, a "virtual mass" is considered:

$$M_A = diag([m_{A_x}\ m_{A_y}\ m_{A_z}\ I_{A_x}\ I_{A_x}\ I_{A_x}]) \tag{4.3}$$

The virtual mass is a function of the geometry of the airship and the density of the air being displaced[8]. Lamb's k-factors[6] for elliptical bodies may be found using the length and maximum diameter of the airship. The k-factors k_1 and k_2 represent a mass fraction of air displaced; k' is a ratio of the virtual moment of inertia to the true moment of inertia of displaced air, I_{zh}. A table of these factors is available in [6], and a spline interpolation was used to determine the appropriate k-factors for the Blimpduino.

The virtual masses are found as follows:

$$m_{A_x} = k_1 m \tag{4.4}$$

$$m_{A_y} = k_2 m \tag{4.5}$$

$$m_{A_z} = k_2 m \tag{4.6}$$

$$I_{A_x} = 0 \tag{4.7}$$

$$I_{A_y} = k' I_{zh} \tag{4.8}$$

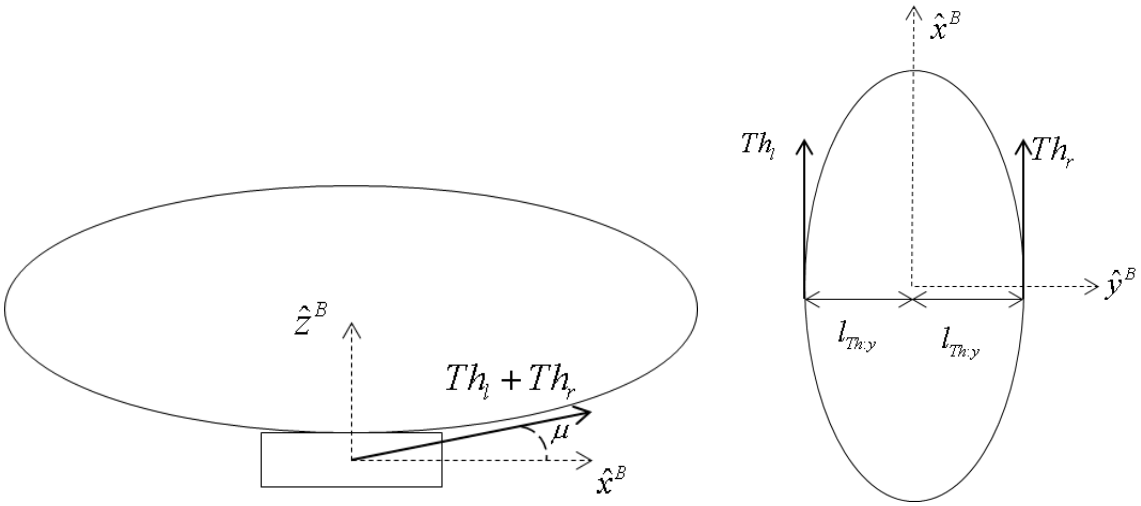
$$I_{A_z} = k' I_{zh} \tag{4.9}$$

$I_{A_x} = 0$ since the blimp is axisymmetrical about the x-axis; a roll about this axis will not displace any air.

The final mass matrix M is then

$$M = M_{RB} + M_A \tag{4.10}$$

4.3.2 Propulsive Forces $\vec{F}_p$



(a) X-Z Plane (b) X-Y Plane

Figure 4.6: Blimp Propulsive Forces

The Blimpduino has two vectorable thrusters that are nominally pointed directly in the x-direction. These thrusters are forced to point in the same direction; however, the thrusters are able to rotate about a central axis that is parallel to the y-axis. The angle between the thrust and the x-axis is the thrust angle μ.

Based on the geometry in Figure 4.6, the propulsive forces $\vec{F}_p$ are

$$\vec{F}_p = \begin{bmatrix} (Th_l + Th_r)\cos\mu \\ 0 \\ (Th_l + Th_r)\sin\mu \\ 0 \\ 0 \\ (Th_l - Th_r)\cos(\mu) l_{Th:y} \end{bmatrix} \tag{4.11}$$

where Th_l and Th_r are the left and right thrust forces; μ is the thrust angle; and $l_{Th:y}$

is the distance from each thruster to the origin along the y-axis.

4.3.3 Aerodynamic Damping $\vec{F}_d$

The Blimpduino's motion is damped due to air friction. The blimp produces negligible aerodynamic lift or moments, but drag in all six degrees of freedom is significant. The aerodynamic damping forces may be written as

$$\vec{F}_d = D(\vec{V})\vec{V} \tag{4.12}$$

where D is the damping matrix. Each component of D contains both linear and quadratic damping terms to account for laminar boundary layers and turbulent boundary layers:

$$D(\vec{V}) = -diag\left(\begin{bmatrix} D_u + D_{u^2}|u| \\ D_v + D_{v^2}|v| \\ D_w + D_{w^2}|w| \\ D_p + D_{p^2}|p| \\ D_q + D_{q^2}|q| \\ D_r + D_{r^2}|r| \end{bmatrix}\right) \tag{4.13}$$

4.3.4 Gravitational and Buoyant Forces $\vec{F}_g$

Generally speaking, the gravitational force acting on an airship is not the same as the buoyant force. For the Blimpduino's case, however, the blimp is ballasted to attempt to bring it to neutral buoyancy. The gravitational and buoyant forces therefore cancel out. The moment about the cb is not necessarily negligible, however. $\vec{F}_g$ may be written as:

$$\vec{F}_g = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -r_z F_b \cos(\theta)\sin(\phi) \\ -r_z F_b \sin(\theta) \\ 0 \end{bmatrix} \tag{4.14}$$

where F_b is the buoyant force and θ and ϕ are the pitch and roll angles, respectively. Since the blimp is at neutral buoyancy, $F_b = mg$. Note that because the blimp is stable in pitch and roll (see Section 4.4.1), this further reduces to $\vec{F}_g = \mathbf{0}$ when the blimp is at neutral buoyancy.

4.3.5 Coriolis Forces $\vec{F}_c$

The Coriolis force is a ficticious force that affects a body when its reference frame is non-inertial[11]. This occurs when there is some motion is both linear and rotational. The Coriolis force may be described mathematically as[15]:

$$\vec{F}_c = C(\vec{V})\vec{V} \tag{4.15}$$

where C is the Coriolis matrix. The Coriolis matrix is

$$C(\vec{V}) = \begin{bmatrix} 0_{3x3} & S(M_{11}\vec{v} + M_{12}\vec{\omega}) \\ S(M_{11}\vec{v} + M_{12}\vec{\omega}) & S(M_{11}\vec{v} + M_{12}\vec{\omega}) \end{bmatrix} \tag{4.16}$$

where $S(\bullet)$ is the skew-symmetric operator. M may be divided up into four three-by-three submatrices; M_{ij} corresponds to the submatrix in row i and column j (that is, rows and columns of the block matrix, not individual rows/columns of the full matrix).

4.3.6 The Navigation States

The body velocities are all that are needed to describe the dynamics of the blimp. But the translational and angular positions are needed to define the location of the blimp in space. The Euler angles $\Phi = [\phi \quad \theta \quad \psi]^T$ represent the roll, pitch, and heading angles of the blimp, respectively; the position variables $P = [\hat{x}^I \quad \hat{y}^I \quad \hat{z}^I]^T$ are the x, y, and z coordinates in the inertial frame, respectively. To translate from the body frame, $\mathcal{F}_b$ to the inertial frame, $\mathcal{F}_I$, the coordinate system only needs to be rotated by the Euler angles:

$$\mathcal{F}_I = C_{I/B}(\phi, \theta, \psi)\mathcal{F}_B \tag{4.17}$$

where the rotation matrix $C_{I/B}$ is given by[11]:

$$C_{I/B}(\phi,\theta,\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.18}$$

$C_{I/B}$ is orthogonal, i.e., $C_{I/B}^{-1} \equiv C_{I/B}^T$, so rotating from the inertial to the body frame may be done simply by transposing $C_{I/B}$ and premultiplying it to a vector in the inertial frame.

It is possible, then, to find the inertial position by integrating the body velocities in the body frame, then rotating the body position to the inertial frame; or by rotating the body frame velocities to the inertial and integrating them. It is more convenient to work with the body frame position to implement control (see Section 4.4), so this is implemented.

Finding the Euler angles based on the body rotation rates is not as straightforward, because the relationship between the body rotation rates $[p \quad q \quad r]^T$ and the Euler angle rate change $\dot{\Psi} = [\dot{\phi} \quad \dot{\theta} \quad \dot{\psi}]^T$ is not simply a rotation. The relationship is[11]

$$\dot{\Psi} = \underbrace{\begin{bmatrix} 1 & \tan(\theta)\sin(\phi) & \tan(\theta)\cos(\phi) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)/\cos(\theta) & \cos(\phi)/\cos(\theta) \end{bmatrix}}_{\mathrm{H}} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{4.19}$$

Equation (4.19) may be significantly simplified with applying some assumptions from Section 4.4; in particular, $p = q = \phi = \theta = 0$ for all time, so the transformation matrix above simplifies to I.

4.3.7 The Complete Nonlinear Airship Model

The complete equations, both for body velocities and for inertial positions/angles, are

$$\dot{\vec{V}} = M^{-1}\left[\vec{F}_d + \vec{F}_g + \vec{F}_p + \vec{F}_c\right] \tag{4.20a}$$

$$\begin{bmatrix} \dot{\vec{P}} \\ \dot{\vec{\Psi}} \end{bmatrix} = \begin{bmatrix} C_{I/B}(\phi,\theta,\psi) & 0 \\ 0 & H(\phi,\theta,\psi) \end{bmatrix} \vec{V} \tag{4.20b}$$

4.4 Design of Control

An LQR-style controller was chosen to control the Blimpduino. LQR control is applicable to linear systems, but Equation (4.20) is nonlinear due to the nonlinear dependence of $\vec{F}_p$ on the thrust angle (i.e., $\cos(\mu)$) and $\vec{P}$ on heading angle ψ (i.e., rotation matrix). Before applying LQR, the system must be linearized. Once this is done, due to the nonlinearities there is some extra logic required to make the LQR controller work properly, via gain scheduling.

4.4.1 Simplifications, Trim, and Linearization

The cg is directly below the cb on the blimp, so it is naturally stable in pitch and roll. Therefore it may be assumed that $\phi = \theta = 0$. Because of this natural stability, no attempt will be made to control it - so $p = q = 0$ as well. The twelve-state state space for the Blimpduino can be reduced to eight states - four velocities and four navigation.

Equation (4.20) is nonlinear due to the dependency of the navigation states (in the inertial frame) on ψ and the nonlinear term in $\vec{F}_p$ due to the variable thrust angle μ. In order to apply a linear control scheme, the system must be linearized around several possible airspeeds, heading angles, and thrust angles.

To find trim points, a quadratic objective function of the form $J = Q^T Q$ was minimized. $Q(x, u)$ is a diagonal matrix whose elements correspond to desired flight conditions:

- No acceleration: $\dot{u}, \dot{v}, \dot{w}, \dot{r} = 0$
- No angular velocity: $r = 0$
- Symmetrical thrust: $T_{dp} - T_{ds}$
- Forward speed: $u - |u|$

In addition, since the system must be linearized several times, some elements are set as functions of desired linearization point:

- Linearize around a particular airspeed: $V_{Desired} - V$
- Linearize around a particular heading angle: $\psi_{Desired} - \psi$
- Linearize around a particular thrust angle: $\mu_{Desired} - \mu$

Although the flight dynamics are not dependent on position, it was chosen to trim around $(x, y, z) = (0, 0, 1)$ to speed up the optimization by reducing the solution space.

Let the trim state and control for a particular triplet (V_D, ψ_D, μ_D) be denoted as $x^*(V_D, \psi_D, \mu_D)$ and $u^*(V_D, \psi_D, \mu_D)$, i.e., the trim states/controls are functions of the desired airspeed, heading angle, and thrust angle. The next step is to find a linearization about this trim point. The Jacobian, the matrix of all first-order partial derivatives of a function, can be used to find a linearization analytically. If the function to linearize is a vector-valued function $F(x, u)$, then the Jacobian is

$$J = \begin{bmatrix} F_{1_{x_1}} & F_{1_{x_2}} & \cdots & F_{1_{x_m}} & F_{1_{u_1}} & F_{1_{u_2}} & \cdots & F_{1_{u_n}} \\ F_{2_{x_1}} & F_{2_{x_2}} & \cdots & F_{2_{x_m}} & F_{2_{u_1}} & F_{2_{u_2}} & \cdots & F_{2_{u_n}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ F_{p_{x_1}} & F_{p_{x_2}} & \cdots & F_{p_{x_m}} & F_{p_{u_1}} & F_{p_{u_2}} & \cdots & F_{p_{u_n}} \end{bmatrix} \tag{4.21}$$

where $f_w = \partial f / \partial w$. We may approximate each partial derivative numerically via central finite difference method for a small perturbation.

J is a function of x^* and u^* and must be evaluated at each $x^*(V_D, \psi_D, \mu_D)$ and $u^*(V_D, \psi_D, \mu_D)$. Once that is done, a state matrix $A(V_D, \psi_D, \mu_D)$ and control matrix $B(V_D, \psi_D, \mu_D)$ can be found.

4.4.2 The LQR Solution and Gain Scheduling

Given a linear dynamical system $\dot{x} = Ax + Bu$, the LQR solution finds an *optimal* control law $u = -Kx$ such that the LQ problem is minimized[3]:

$$V = \int_0^T (x^T Q x + u^T R u) dt \tag{4.22}$$

Q and R are weighting matrices that penalize a particular state or control for being non-zero. A gain $K(t) = R^{-1} B P(t)$ is considered to be the "LQR solution"

for a given set of matrices (A, B, Q, R) if it satisfies the matrix Differential Riccati Equation[3]:

$$-\dot{P} = A^T P + PA + Q - PBR^{-1}B^T P \tag{4.23}$$

The elegance of the LQR solution is in the fact that it is optimal - it is the best possible way to minimize Equation (4.22) at any given point in the linear dynamical system's trajectory for a time $t \in [0\ T]$. If $T = \infty$, the problem is considered "infinite-horizon" and Equation (4.23) reduces to the Algebraic Riccati Equation:

$$0 = A^T P + PA + Q - PBR^{-1}B^T P \tag{4.24}$$

MATLAB implements finding K for the infinite-horizon LQR problem via the command `lqr()`; this approach is taken in designing the controller for the Blimpduino.

Because LQR is optimal and is robust in the classical-control sense[3], it is a useful initial choice to design a controller with. However, the LQR solution is only valid for linear systems. In the case of the Blimpduino, the nonlinearities in state and control mean that A and B are both functions of the current trim state and control, i.e., $A = A(x^*(V_D, \psi_D, \mu_D), u^*(V_D, \psi_D, \mu_D))$ and $B = B(x^*(V_D, \psi_D, \mu_D), u^*(V_D, \psi_D, \mu_D)$. Therefore the LQR solution K must also be a function of the trim state and control.

A gain-scheduling algorithm must be implemented to make use of LQR for nonlinear systems. Essentially, gain scheduling is a technique to use linear controllers for nonlinear plants by implementing a family of gain matrices and selecting one to use based on the current flight conditions. Several gains $K(V_D, \psi_D, \mu_D)$ may be found *a priori*, then a gain may be selected or interpolated on-the-fly based on the current flight conditions.

4.4.3 Flight Control Logic

There are a variety of flight modes the Blimpduino may be in. In order of highest to lowest precedence, these are:

- **Motion in the Z-direction**. If a desired waypoint is above or below the blimp, thrust vertically until the blimp is on the same plane as the waypoint.

- **Turning towards the waypoint**. If the blimp is pointed significantly away from the waypoint, rotate in place until the blimp is pointed reasonably towards the waypoint.

- **Motion towards the waypoint**. If the blimps is pointed towards the waypoint, fly towards it. If the error in heading angle is small, correct it on-the-fly rather than stop and turn in place.

This flight control logic removes some of the nonlinearity from the system - the thrust angle is always in the x^b or z^b directions, and perturbations in heading angle are made small for flight towards a given waypoint by rotating the blimp in place if that perturbation is large.

Rather than adjusting K based on the logic, the state is adjusted instead - this way a costly recalculation of the LQR gain, on-the-fly or offline, can be avoided. For instance, if motion in the z direction is desired, the state error is set such that all error is 0 except the z motion.

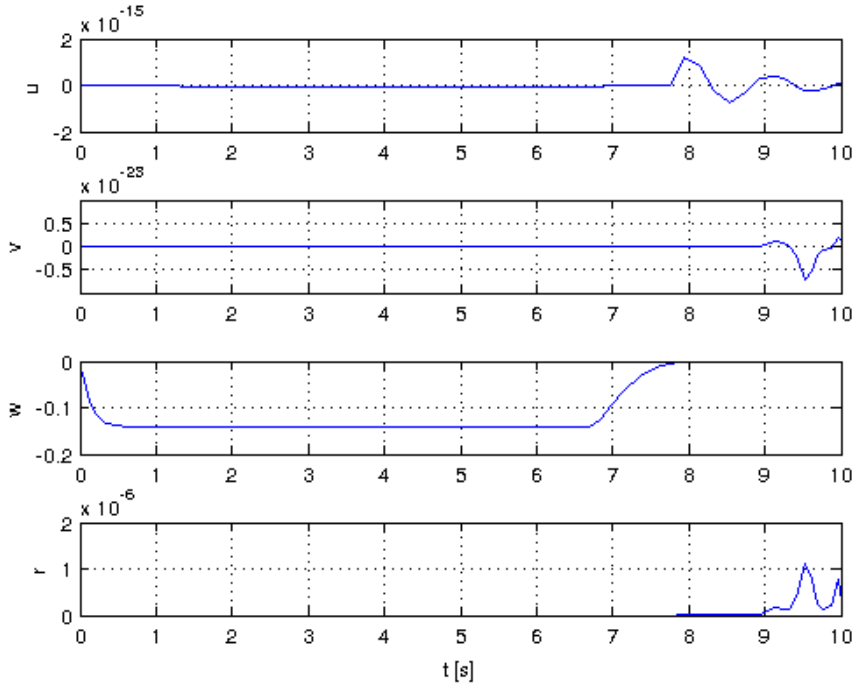
4.5 Simulation Results

Several scenarios were simulated. In each case, an initial condition of zero velocity and arbitrary position/heading were chosen. The scenarios were broken down according to the logic outlined in Section 4.4.3; a flight path consists of three separate motions, so simulating each separate motion is sufficient to show that the LQR-based controller is working properly.

The thrusters are limited to $[-0.0637\ 0.0637]$ N to accurately model propeller capabilities based on the calibration data from Figure 4.4, unless otherwise noted.

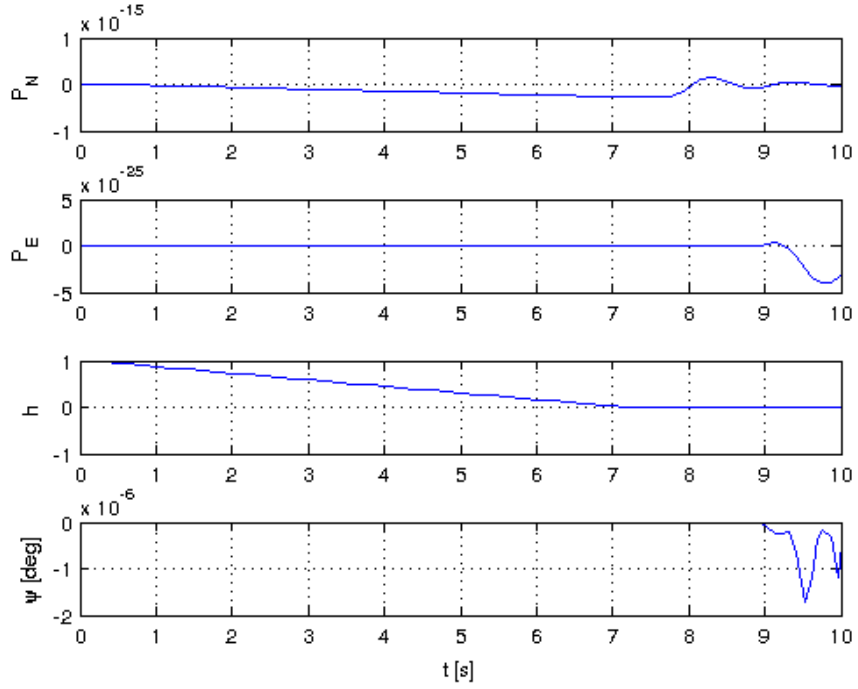
4.5.1 Vertical Motion

The blimp was simulated to descend 1 meter. Figure 4.7 shows the states and controls during the descent operation.

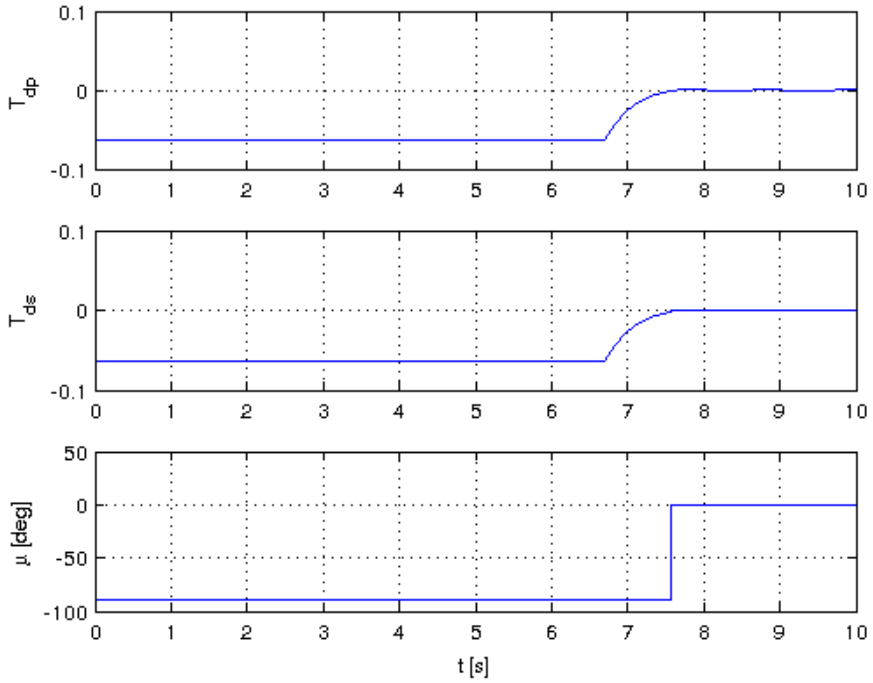


(a) Velocity States

Figure 4.7: Blimpduino Vertical Motion SIL Simulation



(b) Navigation States



(c) Control Inputs

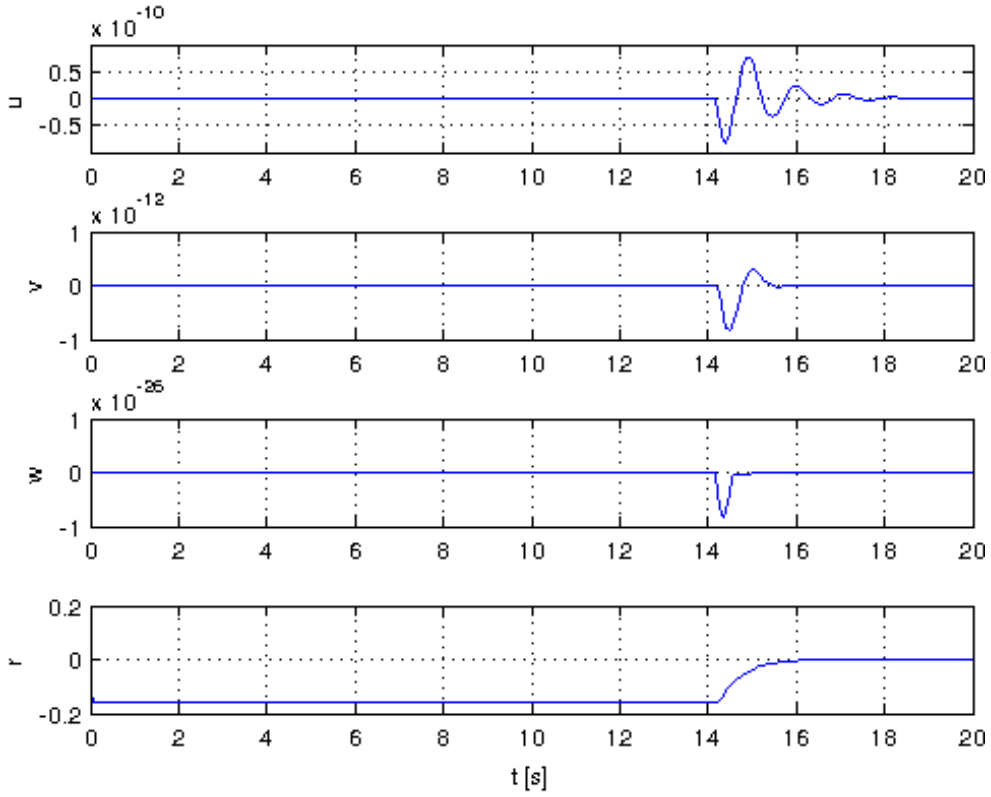
Figure 4.7: Blimpduino Vertical Motion SIL Simulation (cont.)

Notice in Figure 4.7c that the thrust angle is -90 degrees, indicating that the thrusters are pointed straight down; the thrusts are also negative. This has the same effect as if the thrust angle and thrusts were all positive, but physically the thrust angle servo can only pivot from -180 to 0 degrees. This hardware limit is reflected in this simulation.

Based on this simulation, the blimp is able to follow a command of vertical motion.

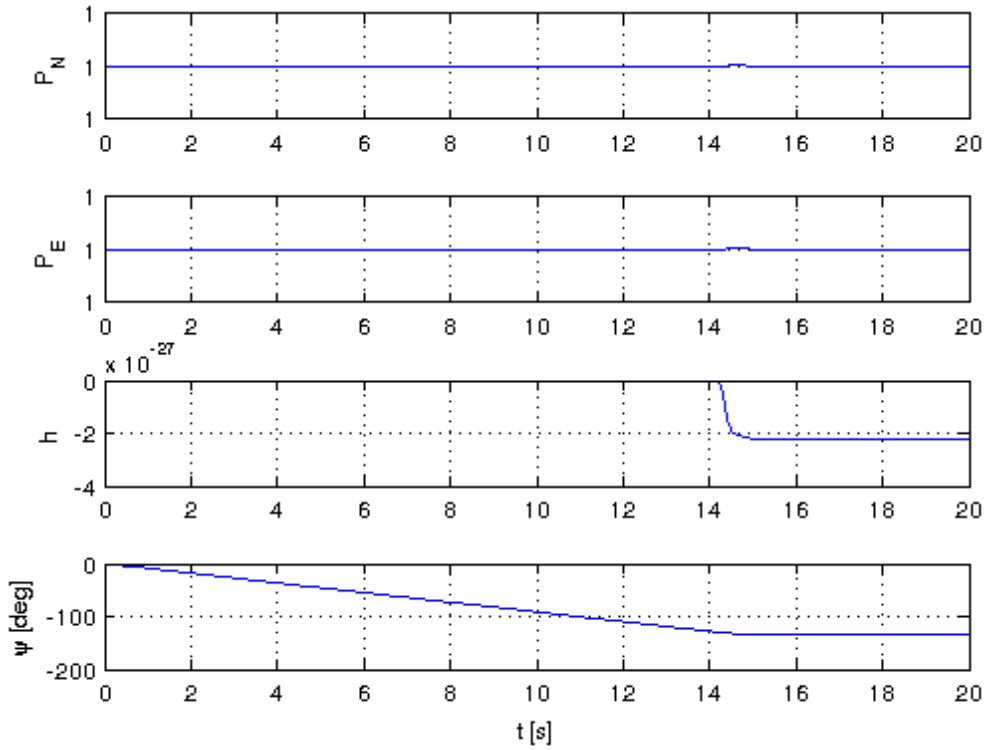
4.5.2 Turning In Place

To simulate turning in place, the blimp was set at $(1, 1)$ facing the $\hat{x}^I$ direction and directed to face the origin $(0, 0)$ - i.e., turning in place -135 degrees. Figure 4.8 shows the states and controls during the turning operation.

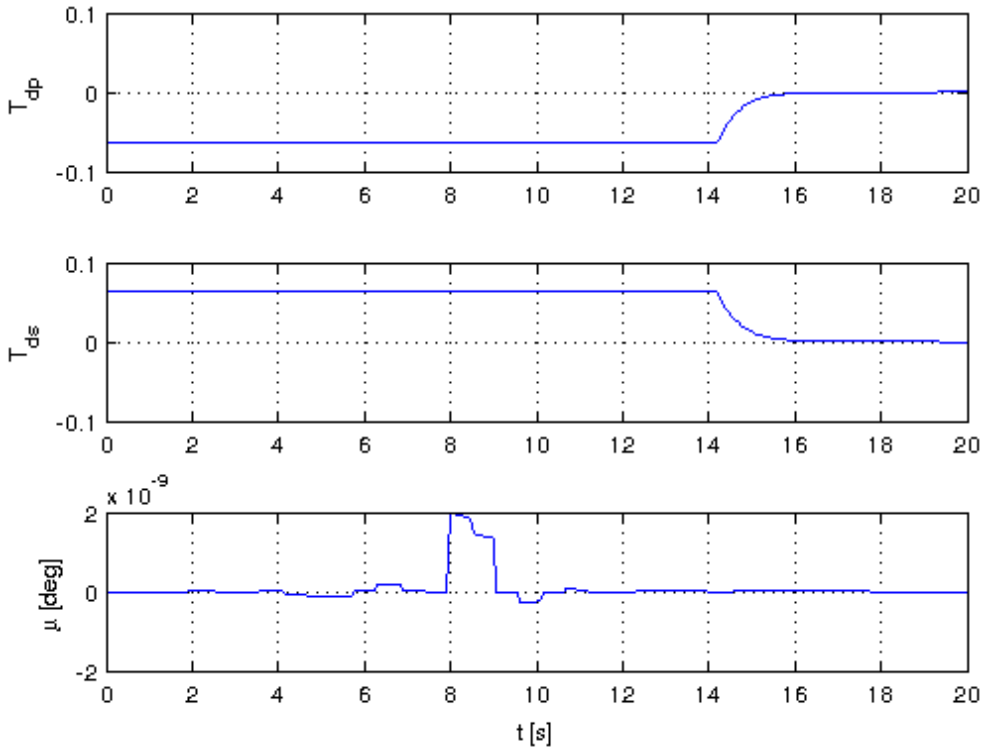


(a) Velocity States

Figure 4.8: Blimpduino Turn-In-Place SIL Simulation



(b) Navigation States



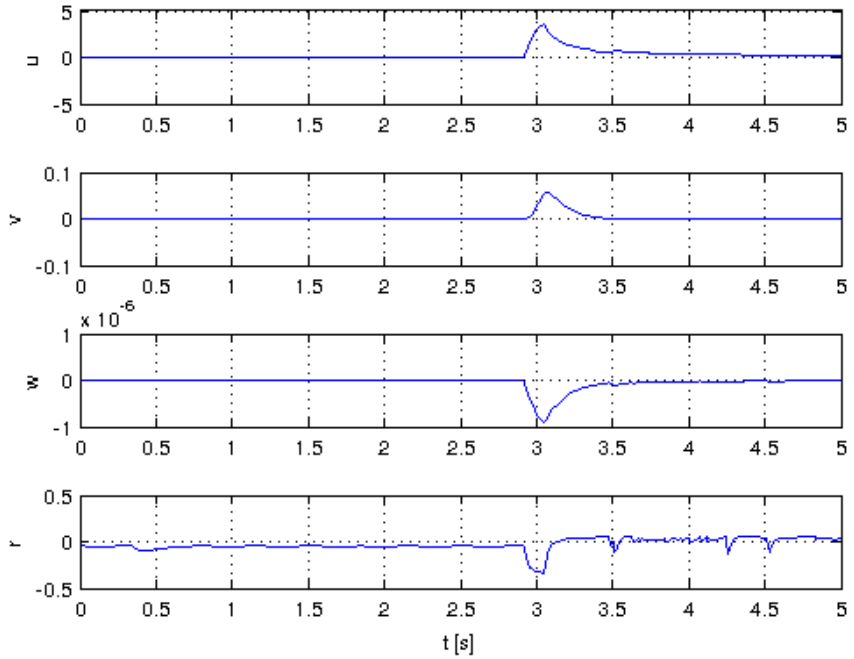
(c) Control Inputs

Figure 4.8: Blimpduino Turn-in-Place SIL Simulation (cont.)

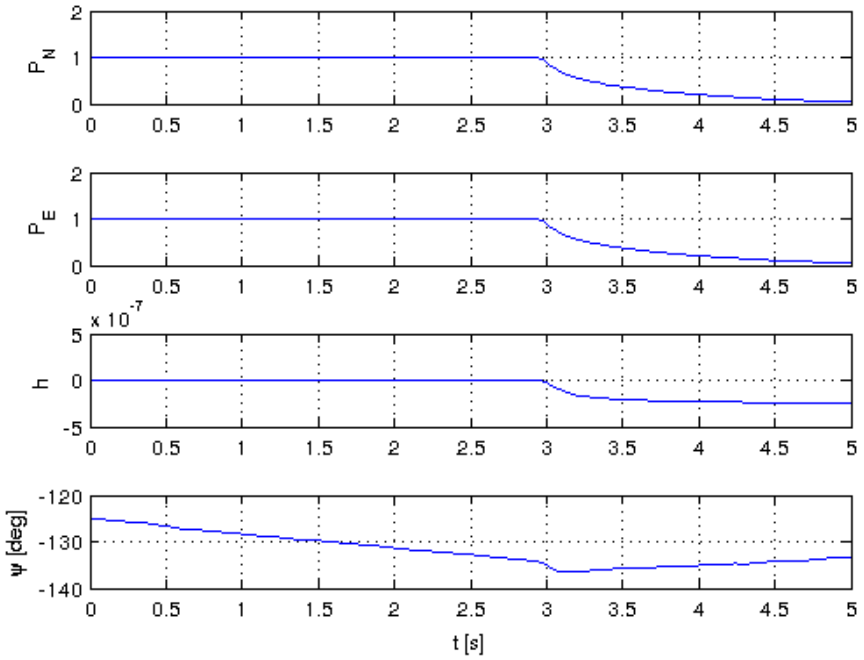
The heading angle is smoothly adjusted from 0 to -135 degrees with the two thrusters saturated and pointing in opposite directions; the heading angle is able to reach the commanded heading with zero steady-state error in finite time.

4.5.3 Forward Motion

For this simulation, the blimp was given an initial inertial position of $(1, 1, 1)$. Because the servomechanism problem may be transformed to the regulator problem[3], the only destination waypoint tested was $(0, 0, 1)$. (The blimp was trimmed to fly at a height of 1 meter, so in the linearized dynamics this corresponds to the origin). The initial pointing was -125 degrees (i.e., 10 degrees away from the origin directly in front of the blimp). Figure 4.9 depicts the states and control as well as the motion in the x-y plane.

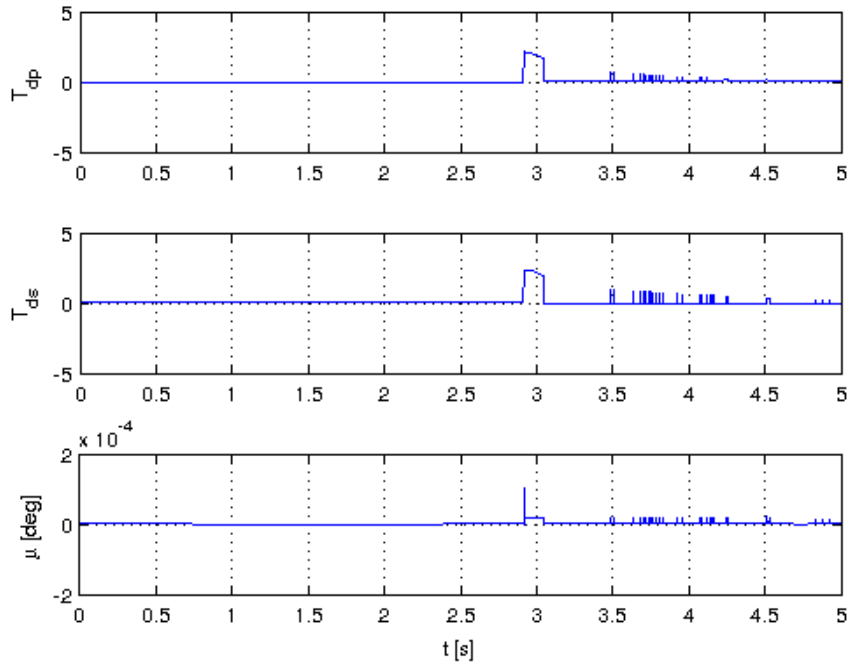


(a) Velocity States

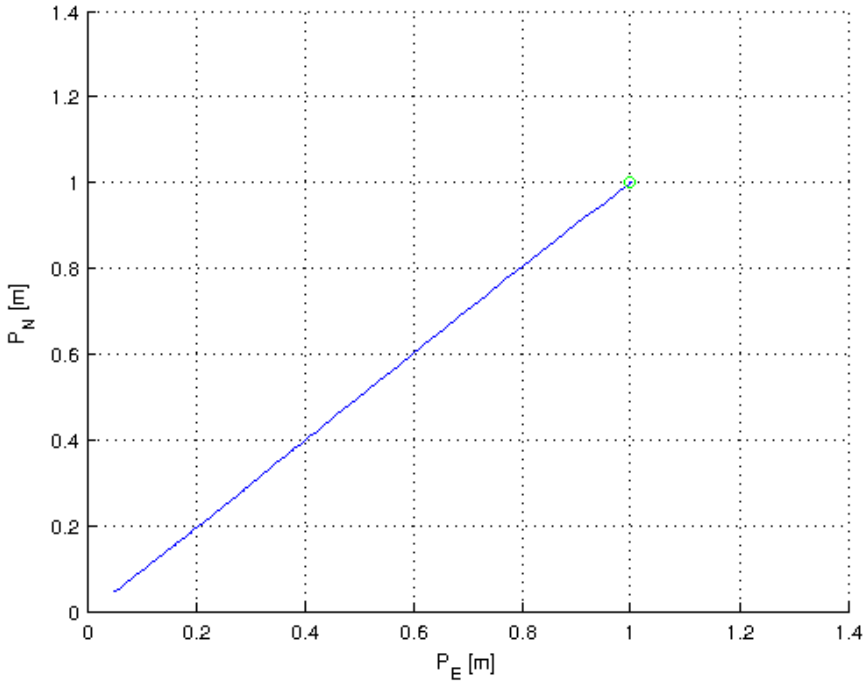


(b) Navigation States

Figure 4.9: Blimpduino Straight Flight SIL SImulation



(c) Control Inputs



(d) Trajectory

Figure 4.9: Blimpduino Straight Flight SIL Simulation (cont)

Due to numerical issues (namely, thrashing of the control inputs), the hardware limit was removed for this simulation. Even with the hardware limit removed, the control inputs continued to thrash slightly, so simulation was halted after 5 seconds. In hardware testing, this thrashing cannot occur since the controller will give the hardware a small but finite time to execute a command before attempting to send a new command.

That said, the simulation indicates that the blimp is able to rotate itself

It is interesting to note that steady state error will occur if the blimp is not directly pointed at the origin, depending on the tolerances set inside the LQR controller; in HIL, it is unlikely that the blimp will ever be pointed exactly towards the origin, so steady state error *will* occur. However, in simulation the cg is considered as a point mass; in HIL execution the entire blimp must be considered, so the steady state error may be negligible in comparison to the size of the blimp. If the steady-state error is too large, applying linear-quadratic-integral control (implemented in MATLAB as `lqi()`) would help to reduce steady-state error to zero[16].

4.6 Hardware Results

Communications between the hardware and MATLAB have been successfully established; the blimp is able to take hardware commands to pitch the servo and activate motors. This was done both in static testing and in manual flight.

Because the original mylar balloon is metallic, the Vicon system is unable to properly see IR markers on it; the balloon is highly reflective and adds a lot of noise to the camera systems. Originally, a large black tissue paper cloak was added to the blimp to reduce this noise. The blimp's mass margin is very small, though, and adding this tissue paper made the blimp too heavy to achieve neutral stability. A smaller, non-reflective envelope was adapted for use with the gondola package.

Error in the hardware execution is abound, which may or may not be avoidable:

- Reaching neutral buoyancy is extremely difficult; the mass margin on the blimp is quite small, so any leaks will require refilling the envelope often.

- Once the blimp reaches neutral buoyancy, external disturbances due to air circulation in the lab environment begin to dominate. Care was taken to reduce this, but this cannot be totally removed.

- The Vicon motion-capture system is not noise-free; position estimates are not exact, and heading angle estimates based on those readings may be erroneous. A Kalmann filter could be implemented to reduce this type of error.

- Aerodynamic damping data were taken from [2], which discusses a similarly sized blimp. These data may not be correct for the Blimpduino and should be found empirically. (With the change to a non-reflective envelope that is different in shape and size, the damping data are certainly not correct.)

- Calibration data only considers steady state. Transients may be significant for the motors, but this was not captured during calibration.

- The actuators used (motors and servo) are not finely tuned, so it unlikely the actuators were able to execute the commanded control inputs exactly. In particular, the servo tends to saturate when the motors are perfectly horizontal and there is some hysteresis in the servo.

Unfortunately, since the new envelope is smaller than the original, it was very hard to get positive buoyancy without filling the balloon to nearly maximum capacity. During a refill it burst and damaged the foam mounting bracket that held all the parts in place. This occurred during the debugging phase of HIL execution, and no data was collected before the failure.

Chapter 5

DIFFERENTIAL DRIVE GROUND VEHICLE: "FRISBEE"

The Frisbee robot is a simple, two-wheeled differential drive ground vehicle. Control for this vehicle is fairly simple: the robot turns towards a desired waypoint, then drives forward. Classical P and PI control are used for this vehicle, and compared.

5.1 Hardware Design

Physically, the Frisbee bot is a two-wheeled, differentially driven ground vehicle that is cylindrical in shape. Figure 5.1 depicts a CAD model of the vehicle.

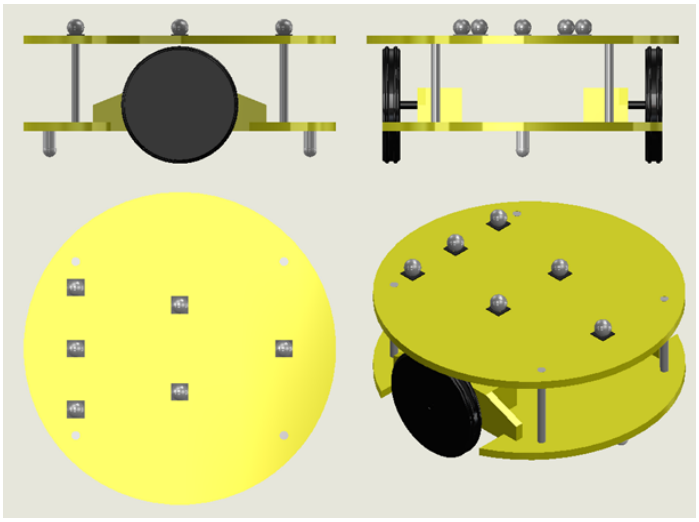


Figure 5.1: CAD Model of the Frisbee Bot

The diameter of each wheel is 7cm, and the diameter of the body is 18cm. IR beacons on top of the Frisbee are used in conjunction with the Vicon motion-capture

system (and by extension the `Vicon` and `ViconSubject` objects).

The vehicle is powered by an Arduino Duemilanove, which is based on the ATmega328 microprocessor. The MathWorks' ArduinoIO software is installed on the chip to allow users to directly control it with MATLAB, either manually or via the DHA architecture (see Chapter 2). An XBee wireless radio is attached to the Duemilanove to allow for communication between the MATLAB ground station and the Frisbee. A 12V lithium polymer battery provides power for the entire system. Figure 5.2, below, depicts the internals of the Frisbee bot.

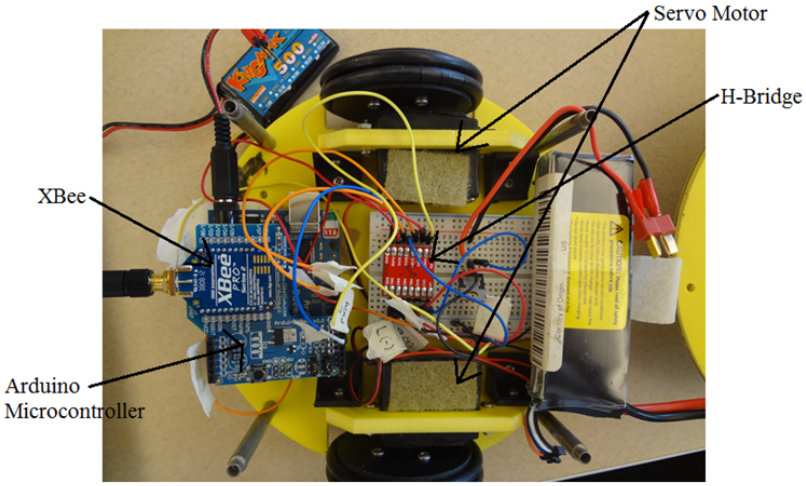


Figure 5.2: Frisbee Bot Electronics

The wheels are driven by servos that can be driven both forwards and backwards; these servos are controlled via an H-bridge. A calibration of wheel rotation speed as a function of voltage is shown in Figure 5.3. The Arduino can output 256 discrete PWM signals (0-255) corresponding to 0-5V on its digital I/O pins; knowing that, a calibration curve of wheel rotaton speed versus PWM can also be created.

Calibration was done by applying an external voltage and finding how long it takes for the wheels to rotate a prescribed number of times. This yields a curve of revolutions per second as a function of voltage, which can then be translated to rad/s

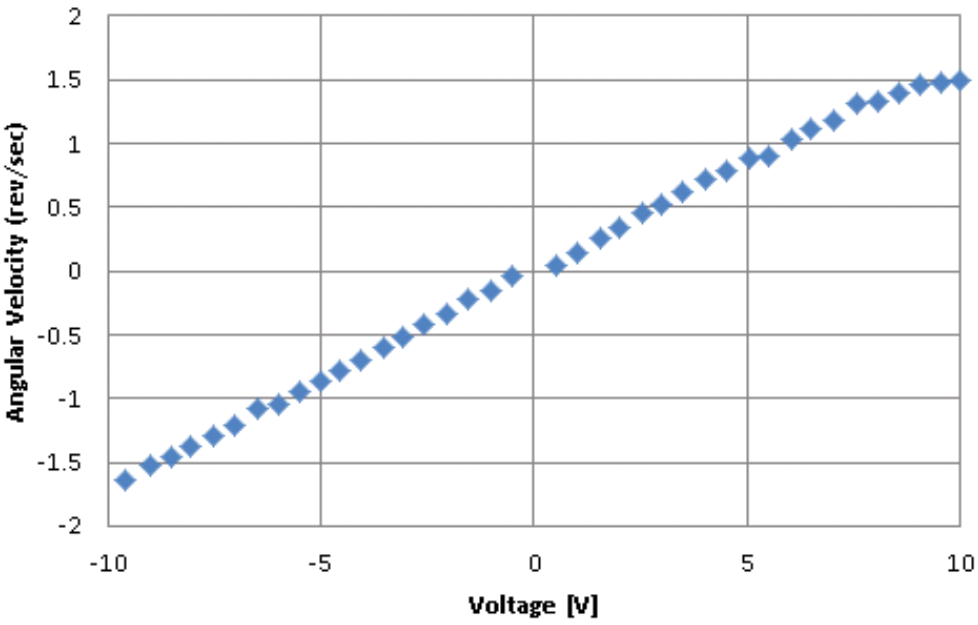


Figure 5.3: Calibration of Wheel Velocity

vs PWM. (This translation is done in the `convertUToMethod` function.) The curve is not exactly linear, particularly near the extremes of applied voltages (0V and 10V), but it is close; furthermore, a spline interpolation is used to look up table values; the calibration curve is smooth enough for this interpolation technique to work properly. Finally, the Arduino can only output a maximum of 5V anyhow; the maximum wheel speed for the Frisbee is then about 1 rev/s, or 2π rad/s.

5.2 Software Design

The Frisbee bot software is written in MATLAB using the DSSL Hardware Architecture design methodology. The `FrisbeeBot < DSSLVehicle` object uses an `HBridgeMotor < Device` to drive the motors on board; the motors are differentially driven. A `runMotor()` method wraps around the underlying `HBridgeMotor.actuate()` command. To run control, the translation from individual wheel velocity to voltage is directly written into the `convertUToMethod()` command, since the `calibrate()`

and `readCalibration()` commands were not implemented at the time.

Since `FrisbeeBot` inherits from the abstract `DSSLVehicle` class, most of the code required to run the bot is actually already written - only a few methods need to be overloaded or reimplemented to tweak the code

The complete `FrisbeeBot` vehicle code may be found on the DSSL Subversion server (see Appendix A).

5.3 System Kinematics

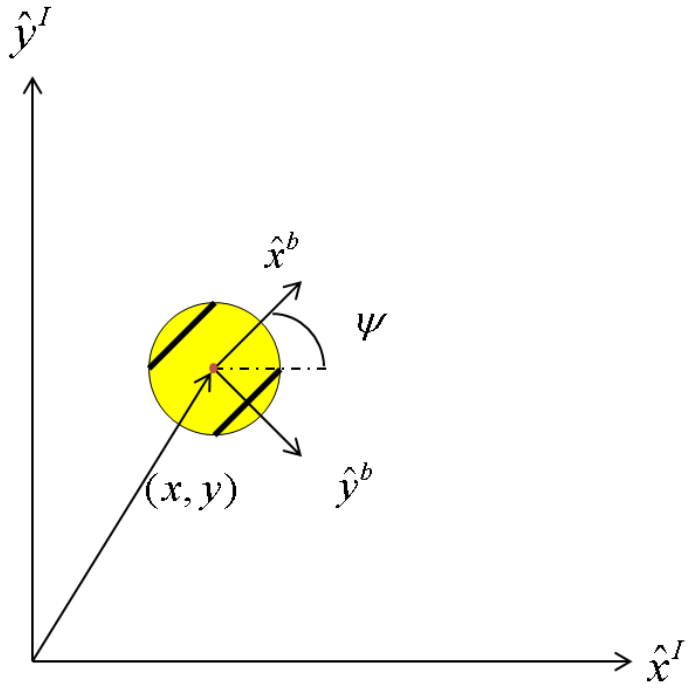


Figure 5.4: Frisbee Reference Coordinate Systems

Figure 5.4 shows the coordinate system of Frisbee bot. A Cartesian coordinate system is used, with a heading angle ψ being the angle between the front of the robot and the x-axis of the inertial reference frame. The robot is circular with a radius of R.

Let v_L and v_R be the velocity of the left and right wheels; the velocity V of the vehicle is then

$$V = \frac{v_L + v_R}{2} \tag{5.1}$$

The angular velocity $\dot{\psi}$ is

$$\dot{\psi} = \frac{v_R - v_L}{R} \tag{5.2}$$

The kinematic equations (i.e., $x(t)$ and $y(t)$ may be solved analytically, given initial conditions x_0, y_0, and ψ_0[9]:

$$x = \frac{V}{\dot{\psi}} \left[\sin(\psi\Delta t)\cos(\psi_0) - (1 - \cos(\psi\Delta t))\sin(\psi_0)\right] + x_0 \tag{5.3}$$

$$y = \frac{V}{\dot{\psi}} \left[\sin(\psi\Delta t)\sin(\psi_0) - (1 - \cos(\psi\Delta t))\cos(\psi_0)\right] + y_0 \tag{5.4}$$

$$\psi = \dot{\psi}\Delta t + \psi_0 \tag{5.5}$$

where $\Delta t = t - t_0$. The control inputs for the robot are v_L and v_R, so we may rewrite Equations (5.3)-(5.5) as

$$x(t, v_L, v_R) = \frac{R(v_R + v_L)}{2(v_R - v_L)} \left[\sin(\psi\Delta t)\cos(\psi_0) - (1 - \cos(\psi\Delta t))\sin(\psi_0)\right] + x_0 \tag{5.6}$$

$$y(t, v_L, v_R) = \frac{R(v_R + v_L)}{2(v_R - v_L)} \left[\sin(\psi\Delta t)\sin(\psi_0) - (1 - \cos(\psi\Delta t))\cos(\psi_0)\right] + y_0 \tag{5.7}$$

$$\psi(t, v_L, v_R) = \frac{v_R - v_L}{R}\Delta t + \psi_0 \tag{5.8}$$

5.4 Design of Control

It is desired to direct the Frisbee bot to move to a designated point (x, y) in the inertial frame. PI control is used to move the bot. The logic for the controller is as follows:

- If the bot is not pointed towards the target, rotate it towards the target (using the shortest arc length) until it is pointed towards the target.
- Otherwise, drive towards the target, using PI control.

PI control is a "classical" control approach[4], and as such the problem must be posed as a single-input, single-output (SISO) problem. The input u will be the total velocity of the vehicle V as described in Equation (5.1); the output will be the distance between the current location and the target location (i.e., the error). The objective is to regulate the error to zero.

The system is time-varying, as is the controller; but the `ControlModel` as currently implemented can only handle time-invariant systems because of the one-timestep-at-a-time approach (see Chapter 3). This problem is mitigated by using `persistent` variables that maintain the current error integral and using `ControlModel`'s maintained timestep and knowing the frequency of running the controller. The error integral associated with the I term is calculated by using the trapezoid rule to calculate the error of the current timestep, then adding it to the running sum of error so far.

Gains K_P and K_I were empirically determined based on hardware performance.

5.5 DHA Implementation

The controller was built using MATLAB and imported to Simulink via an `Interpreted MATLAB` block. This was a fairly arbitrary design choice; the controller could have been just as easily be built in pure Simulink. The controller described in Section 5.4 was

designed assuming that only V, the total forward velocity of the Frisbee bot, is the control input; this must be translated in the logic to account for the fact that two physical controls exist (left wheel and right wheel velocities) via Equations (5.1) and (5.2). This controller was placed in a Simulink model following the design constraints required from Chapter 2.

Once the `FrisbeeBot` vehicle was implemented, a MATLAB script was written to run the control and collect data appropriately. Code Sample 5.1 shows the script used to run the robot.

Code Sample 5.1: `FrisbeeBot` Control Script

```
% Configuration settings
SERIAL_PORT = 'COM5';
WAYPOINTS = {[1 1]; [0 1]; [0 0]; [1 0]};
SIMULINK_MODEL = 'FB_Control_Simulink';
ERR_TOL = 1e-3;

% Vehicle setup
V = Vicon.GetInstance;
V.Initialize;
FB = Frisbee(SERIAL_PORT);
FB.AddViconSubject(V, 'Frisbee');

TC = TelemCollector(FB);

% The ControlModel has no states - it only does control, no observing.
CM = ControlModel(FB, SIMULINK_MODEL, 'numInps', 2, 'numOuts', 3, ...
    'numStates', 0);

% Turn on data collection
TC.start;

```

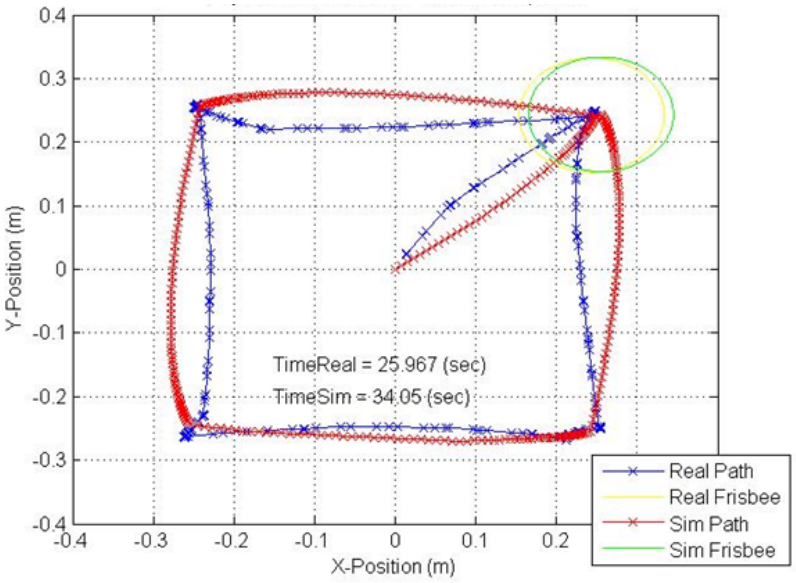
```
% Execute waypointing
for i = 1:length(WAYPOINTS)
    [t X Y Z Vx Vy Vz] = FB.getViconData('head');
    [Xdes Ydes] = disperse(WAYPOINTS{i});

    Xd = [Xdes; Ydes];

    % Import the reference states to the Simulink controller.
    CM.SimulinkVariables = v2struct(Xd);

    CM.start; % Turn on controller and keep checking the error —
              % once the error is small enough, exit loop and switch ...
                 datapoints.
    while norm([X-Xdes; Y-Ydes]) > ERR_TOL
        [t X Y Z Vx Vy Vz] = FB.getViconData('head');
        drawnow; % needed to empty event queue
    end
    CM.stop; % Turn off control to reset the timestep to zero.
    wait(5);
end
```

Code Sample 5.1 shows every aspect of running a hardware vehicle - it sets up the `Vehicle`, the `TelemCollector`, the `ControlModel`, then feeds information into the controller to get it to run. One objective of the DHA is to allow users to write simple-to-use, understandable and clean code. The DHA allows users to think about the steps needed to run control at a high level: "turn on Vicon, turn on the vehicle, enable data collection, enable control, feed reference points into the controller."

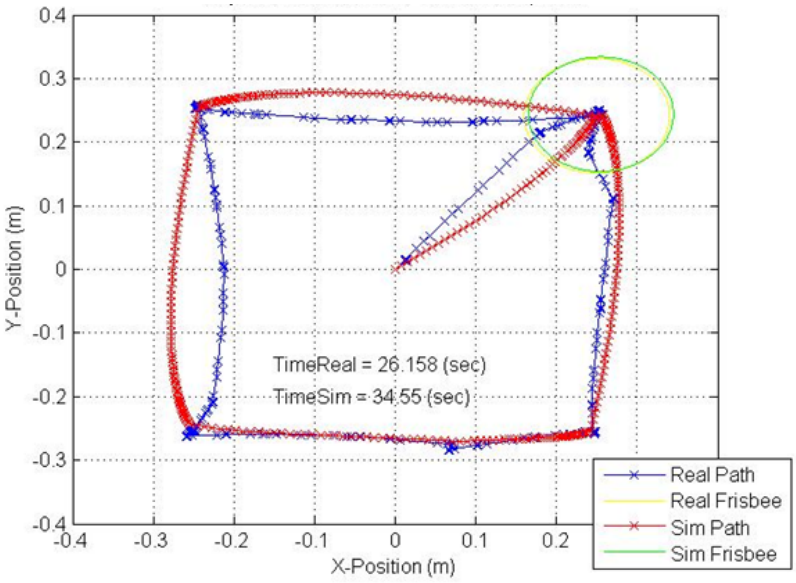
5.6 Results

The Frisbee was commanded to travel from $(0, 0)$ to $(0.25, 0.25)$, then move in a square autonomously (as shown in Code Sample 5.1). The result of both SIL and

HIL execution for P and PI control are depicted in Figures 5.5a and 5.5b.



(a) P Control



(b) PI Control

Figure 5.5: SIL and HIL Execution for Path-Planned Control of the Frisbee Bot

In all cases, the Frisbee successfully reaches the targets of $(\pm 0.25, \pm 0.25)$, but the paths themselves are a little different. The time to execute PI control to drive in a square is slightly larger than just using P control; this is expected, since PI control typically overshoots the target slightly in exchange for reducing steady state error. Then, the SIL paths seem to "bow out" compared to the HIL paths, which "bow in". This may have occurred due to the control logic that requires the robot pivot in place until it gets to within a certain pointing tolerance. Tightening the tolerance may make these squares "tighter", and as the tolerance gets closer to zero the paths for both should become perfect squares. Making the tolerance too tight for HIL simulation, however, will result in the robot spinning around forever. This is due to the finite precision of the Vicon motion-capture system - even if the controller executed infinitely quickly, the noise in the Vicon heading angle reading will make it very difficult for the robot to exactly point at an angle. One way to improve the performance besides tightening the pointing angle tolerance is to apply a filter. Or, of course, more complicated control (i.e., MIMO control) can be applied.

In terms of the architecture's performance, the robot executes exactly as directed based on Code Sample 5.1. The ability to follow waypointing with a controller built in Simulink means that the DHA is working properly, and the fact that the HIL performance is comparable to SIL simulation indicates that the architecture is able to handle running real-time control of hardware vehicles.

Chapter 6

USING THE DHA FOR MULTI-AGENT SYSTEMS

The examples shown thus far are all of single systems to be controlled, but the DHA can handle multi-agent systems as well. This chapter briefly discusses using MATLAB to build multi-agent systems for hardware control. No discussion on control of multi-agent systems will be made; rather, this chapter shows how a user can set up the MATLAB environment to build a distributed system.

6.1 Distributed Systems

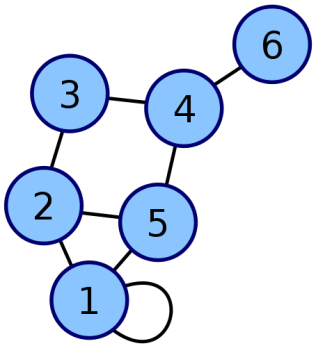
A classic example of a distributed system is a formation of autonomous planes flying in formation; a plausible objective would be to have each plane fly in a tight formation without being directly commanded by a ground station to do so, i.e., to autonomously figure out an optimal way to all fly in formation given certain conditions. Generally speaking, a distributed system is a multi-agent system where the individual agents do not necessarily have all the information about the entire system[7]. The knowledge an agent has is limited to a few neighbors' state. The problem becomes, then - given a distributed system, how can the system as a whole be controlled if each agent does not have complete knowledge of the system?

This is an active area of research within the Distributed Space Systems Lab at the University of Washington, and part of the goal of this software architecture is to assist with this effort by creating hardware distributed networks to apply some sort of control to them.

6.2 How DHA Fits In to Distributed Systems

Every `DSSLVehicle` instantiated in the MATLAB workspace is discrete, just as in a real distributed system each agent is discrete. Mathematically, not only must each agent's dynamics be defined, but also the connectivity between it and all other agents in the system - i.e., which agents can "talk to" each other.

This may be represented easily as a graph[7] - nodes represent agents, and edges represent the connectivity between certain agents. The graph in turn may be represented as an adjacency matrix[5], which is easy to implement in MATLAB. Figure 6.1 depicts the graph and adjacency matrix.



(a) Undirected Graph

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(b) Adjacency Matrix

Figure 6.1: Representations of a Distributed/Networked System

Suppose a user wants to create a distributed system with n agents; in MATLAB, he could simply create an n-by-1 cell array of agents. There would also need to be n `ControlModels` (but not `TelemCollectors` - for reasons outlined in Section 6.3). Code Sample 6.1 shows a script that might be used to build the graph from Figure 6.1.

Code Sample 6.1: Example of a Distributed System with `DSSLVehicles`

```
n = 6;
connectivity = [..]; % See above adjacency matrix
agents = cell(n,1);
CM     = cell(n,1);

% Initialize all DSSLVehicles
agents{1} = DSSLVehicleSubClass(..);
..
agents{n} = DSSLVehicleSubClass(..);

for i = 1:n
    % Have each agent collect data about all the other agents
    % it can, and store that in the ControlModel via SimulinkVariables
    otherAgentData = cell(n,1);
    for j = 1:n
        if connectivity(i,j) == 1
            otherAgentData{j} = agents{j}.GetCurrentOutput();
        else
            % Controller will have to deal with the nan's
            otherAgentData{j} = nan;
        end
    end
    CM{i}.SimulinkVariables = v2struct(otherAgentData);

    % Run all agents' runControl code.
    for j = 1:n
        CM{i}.runControl();
    end

    % Add a delay for a little while - match the frequency each bot
    % would run control at
    pause(0.02);
```

```
33 end
```

This is much more verbose than other codes to run control, but a lot of this comes from the fact that information needs to be passed around from agent to agent. Furthermore, in this example there are no `TelemCollector`s involved. `GetCurrentOutput()` will need to be modified to make calls to `getTelem()` to operate asynchronously.

The controller for each agent has access to other agents' outputs via the `SimulinkVariables` properties, and in this script information is only transmitted according to the adjacency graph. Ideally, each controller would run in parallel (and users could use a `parfor` loop if the Parallel Computing Toolbox is available); to ensure that each controller executes in a timely fashion without other agents changing state too much, the `controlFreq` should be kept relatively low.

6.3 Constraints on Hardware Distributed Systems with the DHA

There are a few problems with this setup, and most problems are due to poor scaling; for large networks, computer engineering problems such as concurrency start to become important.

Each `DSSLVehicle` needs its own `TelemCollector` to operate, as well as its own `ControlModel`. For large networks, the memory cost may become unacceptably large. For `TC` this could be mitigated by using events and listeners to operate the `getTelem()` methods, but this is more complicated for `CM`. An **instance manager** class could be in charge of firing each `CM.runControl()` callback.

There also needs to be a separate serial port for each Arduino to be attached to the ground station, and an XBee radio pair for each agent. From a software perspective, with too many agents the MATLAB event queue will tend to get full and commands may be executed behind schedule (or dropped entirely, based on the users' timer configurations). To ensure that all commands are executed in a timely fashion, users will have to decrease the frequency of data collection or control. The queue cannot be

parallelized via the Parallel Computing Toolbox (or at least, this is not documented), so multithreading will not help.

Some of these constraints can be mitigated by manually choosing to run the callbacks on a user basis, rather than automatically with timer objects. In that case, the user could use `parfor` loops to execute data collection and control code in parallel. The tradeoff here is the user must manually keep track of performing the executions, so encapsulation must be violated. Ideally, the ground station would have a CPU available for each `DSSLVehicle` in the `parfor` loop, otherwise the agents will have large response latencies. This is really only a problem for very large networks, but the definition of "very large" will depend largely on the power of the ground station running MATLAB.

Chapter 7
CONCLUSION

This thesis serves as a guide for future developers of the DSSL Testbench to use MATLAB to rapidly design hardware vehicles for use with MATLAB/Simulink. A technical description of the architecture was presented, along with several case studies that underline the ability to rapidly create code to do hardware control. It is the author's hope that this will serve as a stepping stone for implementing more complex vehicles and the controllers to run them, all within the familiar, powerful, control-engineer-friendly confines of MATLAB. Ultimately, it is the hope of the author that little to no computer engineering experience outside that of a typical control engineer's education is needed to efficiently use the architecture. Of course it is helpful to understand the building blocks that make up the DHA, but the design choice of using object-oriented programming ensures that users only have to understand the functionality at a high level.

Each case study depicts the three areas of focus that a team must consider in the project: hardware design, software design, and mathematical modeling/controller design. A project that is structured such that these areas are interdependent yet implemented separately should be successful fairly rapidly, once users are familiar with the architecture and design methodology.

7.1 Future Work

There are several improvements that could be made within the architecture's framework. As it stands, the architecture is certainly usable, but there are ways to make it more intuitive or useful.

- **Implement in hardware a multiple-vehicle controller**. The DSSL's research focus is in distributed networks of systems. It has been shown theoretically how a network could be implemented in MATLAB via connectivity matrices, but this has not been tested. A good start would be to do a simple lead/follow of two vehicles.

- **Allow users to import settings from a configuration file**. Currently, there are several steps involved in setting up objects, and these steps involve manually setting up pinouts or hardware limitations. A configuration file would let users instantiate vehicles with arbitrary properties quickly. (`save`ing and `load`ing the objects may lead to undesirable results, since other objects will point to the vehicle. If the object is loaded but is not contained in the same memory as before, the resultant behavior will likely be anomalous.)

- **Create a GUI to generate new `Vehicle` or `Device` code**. Users could specify what `Device`s they want on their `Vehicle`s, then leave skeleton code where users must fill in the needed methods to properly run the `Vehicle`. This is akin to MATLAB's GUIDE interface to allow users to build GUIs quickly. Code-generated-code is inherently difficult to debug, however.

- **Simplify Simulink API calls inside `ControlModel` and `SimulinkVehicle`**. Unfortunately, the Simulink API is not nearly as well documented as the MATLAB language, and it is likely possible to simplify the code and make it more robust. For instance, there is no simple way to check how many states an observer/controller has until it is run for the first time. It is not very elegant to initially run the controller without the user's knowledge just for this information; the fix is to force the user to specify the number of states ahead of time. This may or may not be acceptable (since it does not agree with the OOP principles of encapsulation and information hiding).

- **Implement XBee-API Mode with Java**. Currently, wireless communication via XBee radio takes place via AT mode, which essentially acts as a serial passthrough. To configure the radios, they must be placed in AT-Command mode or configured manually with external software such as X-CTU. XBee-API mode is much more robust than AT mode for communications and makes it simpler to have radios communicating in mesh networks. Furthermore, this allows users to connect only *one* XBee radio to MATLAB to communicate to each agent in the network; currently, for n agents there must be n XBees attached to the computer. To implement API mode, the entire ArduinoIO API must be rewritten (both the MATLAB half and the C half) to avoid using serial communications. In particular, using XBee-API mode will require embedding Java inside the ArduinoIO MATLAB code.

BIBLIOGRAPHY

[1] Steven C. Chapra. *Applied Numerical Methods with MATLAB for Engineers and Scientists*. McGraw-Hill, 2nd edition, 2005.

[2] Filipe Manuel da Silva Metelo and Luis Ricardo Garcia Campos. Vision based control of an autonomous blimp (videoblimp). Technical report, Universidade Técnica de Lisboa, September 2003.

[3] Peter Dorato, Chaouki T. Abdallah, and Vito Cerone. *Linear Quadratic Control: An Introduction*. Kreiger Publishing Company, 2000.

[4] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamical Systems*. Pearson Higher Education, Inc, 6th edition, 2010.

[5] Chris Godsil and Gordon Royle. *Algebraic Graph Theory*. Spring-Verlag New York, Inc, 2001.

[6] Horace Lamb. The inertia-coefficients of an ellipsoid moving in fluid. Technical Report and Memoranda No. 623, British Aeronautilc Research Committee, 1918.

[7] Mehran Mesbahi and Magnus Egerstedt. *Graph Theoretic Methods in Multiagent Networks*. Princeton University Press, 2010.

[8] Max M. Munk. The aerodynamic forces on airship hulls. Technical Report NACA No. 184, NACA, 1923.

[9] Gregor Novak and Martin Seyr. Simple path planning algorithm for two-wheeled differentially driven (2wdd) soccer robots. In *WISES'04*, pages 91–102, 2004.

[10] Michael L. Scott. *Programming Language Pragmatics*. Academic Press, 1999.

[11] Brian L. Stevens and Frank L. Lewis. *Aircraft Control and Simulation*. John Wiley & Sons, 2nd edition, 2003.

[12] The MathWorks Classroom Resources Team. MATLAB Support Package for Arduino (aka ArduinoIO Package). [http://www.mathworks.com/matlabcentral/fileexchange/32374].

[13] The Mathworks, Inc. Simulink Coder. [http://www.mathworks.com/products/simulink-coder/].

[14] The MathWorks, Inc. xPC Target. [http://http://www.mathworks.com/products/xpctarget/].

[15] Jasper van de Loo. Formation flight of two autonomous blimps. Master's thesis, Technische Universiteit Eindhoven, October 2007.

[16] P.C. Young and J. C. Willems. An approach to the linear multivariable servomechanism problem. *International Journal of Control*, 15(5):961–979, May 1972.

Appendix A

DHA DEVELOPER'S GUIDE

The purpose of this Appendix is to help new users develop new `DSSLVehicle` objects from scratch and use it to accomplish HIL execution.

Chapter 2 is required reading. This chapter discusses the architecture from a high-level point of view; this tutorial might not make that much sense if the context is not understood.

It is assumed the user is fluent with MATLAB.

This guide was written assuming the user is running Linux. However, DHA (and MATLAB, for that matter) is OS-agnostic; if there are variations between OSes, they will be pointed out.

What's covered in this guide

- Setting up your development environment
  - `PATH` setup
  - Subversion basics - concepts and actions
    - The concept of revision control is interesting - you might want to read more about it from the Subversion manual[1]
  - Setting up the Arduino IDE (needed to upload ArduinoIO code to your Arduino)
- Designing `DSSLVehicle`s

[1]See `http://tortoisesvn.net/docs/nightly/TortoiseSVN_en/` - this isn't the true SVN manual, but it's SVN from a graphical, Windows user perspective.

  - Constructor requirements
  - Setting up pinouts and data collection
  - Required and optional methods
  - Interfacing your `DSSLVehicle` with external processes

- Designing `Device`s
- Designing Simulink Models for `ControlModel`
- HIL Execution
  - Interfacing with the Vicon motion-capture system via `Vicon` and `ViconSubject`
  - Collecting and visualizing data with `TelemCollector` and `Telem`
  - Running control with `ControlModel`s

What's <u>not</u> covered in this guide

Developers are responsible for understanding the concepts below.

- Arduino basics (language, IDE)
  - Take a look at `www.arduino.cc` for tutorials. It's strongly recommended you understand the basics of Arduino-C (the "Wiring" language) to interface hardware to software - in particular, how to do I/O and the differences between analog and digital signals.
- Hardware design/development
  - An undergraduate course in basic circuits and instrumentation (AA 320, for instance) should be enough background to build basic circuits. Soldering and wirewrapping skills are also highly recommended.

- SIL simulations (including development/design of models and controllers)
  - Users should be familiar with modern control theory (state-space form) and using Simulink. Classical control theory (transfer functions) might be enough for very simple systems (e.g., SISO), but this hasn't been tested rigorously.
- Object-oriented MATLAB (concepts and techniques)
  - OOP principles are introduced in undergraduate-level Java or C++ courses. The MATLAB documentation[2] is the best resource to learn about MATLAB-specific OOP functionality.

A.1 Setting Up Your Development Environment

It is strongly recommended that you use your own computer for development; the DSSL workbench computer should only be used when using Vicon and doing HIL execution. (Actually, you can even talk to Vicon with your own laptop as long as the Vicon hardware is on and the lab computer is running the Vicon Nexus software.)

A.1.1 Subversion Basics

The DSSLHardware server is kept under version control via Subversion. You will need to install a Subversion client to access it.

- Windows: TortoiseSVN (`tortoisesvn.net`)
- Linux: `subversion` - from your distro's package manager
  - Consider a GUI interface like RabbitVCS (`rabbitvcs.org`) to make SVN usage easy.

[2] `http://www.mathworks.com/help/techdoc/matlab_oop/ug_intropage.html`

- Mac OS X: SCPlugin (`scplugin.tigris.org`)

This tutorial will be using Linux and RVCS, but users of TSVN and SCP should be able to follow along. They all do the same thing.

SVN Concepts

In a naive client-server setup, users might accidentally overwrite each others' work if two people are working on the same file at the same time. One idea to get around this is to *lock* a file so only one person can work on a file at a time. This is implemented with Microsoft Sharepoint to allow collaborative editing between teams. But this isn't effective either - what if two people are working on entirely separate sections of code in the same file?

Subversion instead allows anyone to work on any file they want. Users *check out* a part of or the entire repository of files to work with, then *commit* changes they may have made to source code. SVN will seamlessly merge in changes to files from two separate people if the changes don't overlap. (If they do, a helpful `diff` tool can be used to determine which changes should be merged and which ones should be left out.)

SVN also maintains a revision history and allows users to roll back commits if necessary. This way if you work on a file and something breaks, you can always revert to a known good state for that file - even if that state was months previously. An optional but highly recommended log file may also be written along with a commit.

There are a couple of commands that need to be run fairly regularly in order to make good use of this robust server:

- **Update**: Receive the latest changes from other users and merge them into your local copy of the repository.

- **Revert**: Revert all changes you've made to a local copy. This is useful if you've made so many mistakes you want to start from a last known good configuration.
- **Add**: Flag a file to be added to the server on the next commit (see below).
- **Remove**: Flag a file to be removed from the server on the next commit.
- **Commit**: Submit changes to your files to the server. These changes will be downloaded by other users on the next update.

The next few sections cover enough basic usage of Subversion to use with the DSSLHardware repository, but there are much better tutorials available on the Internet. Check them out for yourself.

Checking out the DSSLHardware Repository

The DSSLHardware repo is located at

`svn://128.95.33.121/srv/svn/DSSLHardware/`

Right-click in your file manager and select "RabbitVCS SVN → Repository Browser", then point the URL to the server location above. Figure A.1 depicts the repo browser view. (TSVN and SCP users have very similar commands and repo browser windows.)

You'll be asked to log in - as of this writing, the SVN administrator is Josh Maximoff (`maximoff@u.washington.edu`). Contact him to set up an SVN account.

From here, you can view the entire repository. Right-click the `DSSLHardware` folder and select "Checkout...". Set the destination properly (the default might be good enough; you might want to make the repository a subfolder in your Documents folder, however), then click "OK". The server will then begin the transfer of files to your computer.

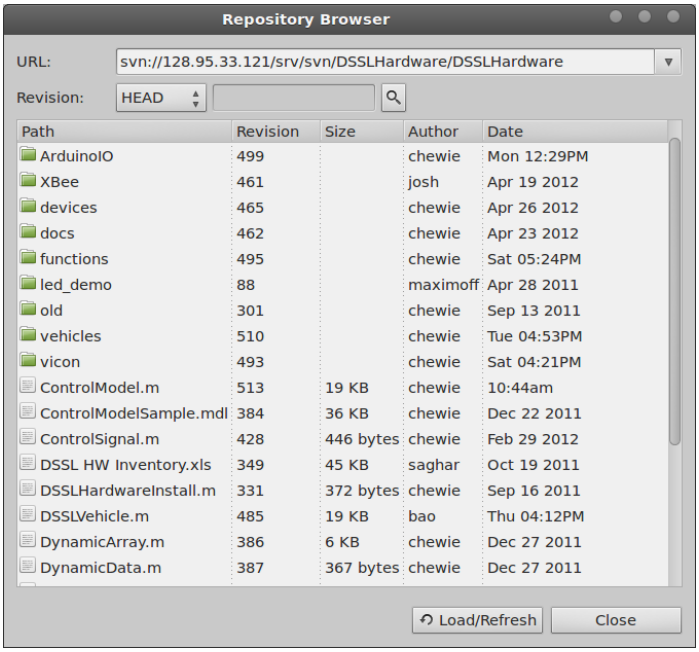


Figure A.1: Repository Browser

A Typical Workflow

All the SVN commands you need are a right-click away. Leave a file manager window open to access them. Figure A.2, depicts emblems your SVN client will denote for the status of some files. (These icons are from RabbitVCS; TortoiseSVN and SCPlugin will have different icons, but they should be similar.)

Before you begin workflow for the day, you'll need to `svn update` (either from the command line or from your file manager) - this will fetch the latest updates from the server to any code you might be working on. **Note**: This includes code you are actively editing! Changes might have gotten merged into your work. To make sure those changes don't interfere with what you're working on, your development team should ensure that any people working on the same code works on separate parts - i.e., no two people should work on the same functions.

If conflicts *do* arise, you can use the `svn diff` tool to resolve them. The offending

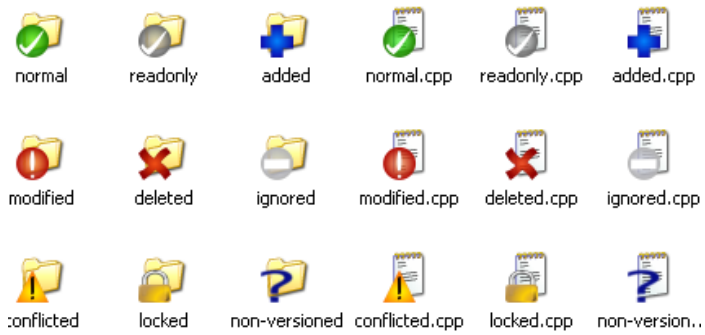


Figure A.2: Subversion Status Icons

lines will be marked. Merge or change the lines, then mark the conflict with `svn resolved`.

Once you're ready to submit your changes, `svn commit` them. You can commit several files all at once. Make sure you fill out the changelog before committing - in case problems arise down the road, the logs may be examined to see if something was changed. **Do not commit files until the changes are tested and known to work properly** - commits are not for incremental work but are a kind of milestone. Commits should not break other code somewhere else - so new additions to code should be made as error- and bug-free as possible.

At some point you'll probably add new files or folders. By default, SVN will ignore files until you've specifically marked them for adding (or you manually select them for commitment in the commit window). You can manually flag them to be added to the server with `svn add`.

Likewise, you'll at some point want to delete files or folders. **Deleting them from your local computer does not it from the server** - you'll need to flag the delete with `svn delete`. (This will also remove the files from your file manager view). Commit deletions to the server ASAP - other users may write to those folders/files and those conflicts are much harder to resolve.

Rename files with `svn rename` rather than naively clicking on the name and chang-

ing it, and commit renames to the server ASAP also - before you edit the renamed file. SVN uses two revisions to do file renaming - one is to delete the old file, one is to create a new file with a different name and copy the contents over.

You should never run into problems if you use the update and commit commands regularly. At some point, there might be so many problems that you just have to re-download the repository. If that's the case, just copy all your local changes somewhere else, delete the entire repo, then re-checkout; merge your changes, and commit immediately.

A.1.2 Setting up your `PATH` with `startup.m`

It is assumed that you have checked out the DSSLHardware repository to `$DSSLHARDWARE`; `$DSSLHARDWARE` may be something like `C:\Documents and Settings\User\DSSLHardware` (Windows) or `/home/User/Documents/DSSLHardware` (*NIX-based OS).

Add the below lines to your `startup.m` file. This code will add all the folders to your MATLAB `PATH` on startup.

Code Sample A.1: `startup.m` Additions for DHA

```
1  HOME = cd;
2  DSSLpath = ...
3      genpath_exclude(...
4          'DSSLHARDWARE', ... % Use your own DSSLHardware path!
5          {'\.svn', 'old', 'led_demo', 'slprj', 'docs'} ...
6  );
7
8  % Feel free to include more directories to ignore.
9
10 addpath(DSSLpath);
11 clear DSSLpath;
12 cd HOME;
```

```
13 clear HOME;
```

A.1.3 Setting up the Arduino IDE

Install the Arduino 1.0 IDE from the Arduino website. In `$DSSLHARDWARE/ArduinoIO` there is an extra library, `AFMotor.zip`. Unzip this and place it in the `libraries` subfolder of your Arduino IDE installation path. This file is required to upload the ArduinoIO program to the Arduino.

There are several dependencies you need to properly compile and upload code to an Arduino. See the Arduino Playground page concerning installation[3] to properly install these dependencies, if needed.

A.2 Designing `DSSLVehicle`*s*

The documentation for every class (properties and methods) is very good; take advantage of it!

A.2.1 Class Constructor

The main job of the constructor is to parse user options such as configuration settings and pinouts. The constructor then calls the `DSSLVehicle` constructor; runs `setupDevices()`; and `setupTelemetry()`. Basically anything that is needed to initialize the `DSSLVehicle` object goes here. You can provide defaults for the pinouts and telemetry in those initialization methods (see below).

Use MATLAB's `InputParser` class to simplify argument handling. The `InputParserExtended < InputParser` class, available in the DSSLHardware repository, simplifies functionality a bit (allows users to write required arguments in any order).

[3] http://arduino.cc/playground/Learning/Linux

A.2.2 Defining the `Devices` *and Pinouts*

Your `DSSLVehicle` needs `Devices` - actuators and sensors - to run. For instance, the Blimpduino has two motors powered by an H-bridge; a servo; and a battery voltage sensor. (It also has a separate pull-up for the H-bridge, but this is due to the PCB trace layout.)

Define some properties to contain the `Device` objects to instantiate. (All these properties should be set to private - you will provide the interface to interact with the `Devices`.) Then instantiate the `Devices` in the `setupDevices()` method (you have to write this yourself). `Device` constructors require two arguments: a `PinManager` object to manage the `Device`, and the pinouts. The `PinManager` is instantiated on `DSSLVehicle` construction and stored as a private property, so you only have to pass `obj.PM` for this argument. The second argument is the pinout, which is an n-by-2 cell array. Each row is a separate pin: the first column is for the binder name, the second is the pin number. The last argument is a struct or cell array of option-value pairs.

Code Sample A.2 shows how the `Blimpduino` instantiates its `Devices`. (Note that only the relevant properties and methods for the `Blimpduino` is shown.)

Code Sample A.2: `Blimpduino` Pinout Setup

```
1  classdef Blimpduino < DSSLVehicle
2  % Device placeholders — we will instantiate Devices and store the ...
       handles here.
3  properties (Access = private)
4      Motor
5      Servo
6      MotorStandby
7      Battery
8  end
9  methods
10     function setupDevices(obj)
```

```
11          MotorPins = {...
12            'LeftMotorFWD' , 5; 'LeftMotorBACK' , 4; 'LeftMotorPWM' , 3;
13            'RightMotorFWD', 7; 'RightMotorBACK', 8; 'RightMotorPWM', 11;
14          };
15
16          ServoPin = {...
17              'ServoPWM', 10;
18          };
19
20          MotorSTBYPin = {...
21              'HBridgeEnable', 6;
22          };
23
24          BatteryPin = {...
25              'Battery', 'A0';
26          };
27
28          BatteryOpts = {...
29              'VRef'      , 3.3;
30              'multiplier', 3  ;
31              'min_batt'  , 6.5;
32          };
33
34          obj.Motor        = HBridgeMotor(obj.PM, MotorPins);
35          obj.Servo        = Servo(obj.PM, ServoPin);
36          obj.MotorStandby = OutputPin(obj.PM, MotorSTBYPin);
37          obj.Battery      = Battery(obj.PM, BatteryPin, BatteryOpts);
38
39          % Instantiate a listener for a low battery signal,
40          % enter safemode if needed
41          addlistener(obj.Battery, 'LO_BATT', @obj.safeMode);
42
43          % Pull the HBridgeMotor standby HIGH — i.e., enable motors
```

```
44          obj.MotorStandby.writeSignal(true);
45
46      end
47  end
```

The code for all `Device`s are in `$DSSLHARDWARE/devices`. Check each file for further documentation and usage. In particular, look at the optional arguments you might want to use, if some exist (for instance, for the `Battery` sensor).

Currently there are four `Device`s available for use:

- `HBridgeMotor` - Power motors via an H-bridge (which allows forwards and backwards operation).
  - Physical H-bridges have a limit as to how many motors can run off of them. `HBridgeMotor` has no such limit (and in fact, you can power multiple H-bridges with one `HBridgeMotor` object as long as the pinouts are set up correctly).
- `Servo` - Attach an RC servo to the Arduino.
  - Due to limitations of the underlying Arduino library that provides this functionality, `Servo`s can only be attached to pins 9 or 10. Using a `Servo` will disable PWM functionality on these pins - i.e., if you attach a `Servo` to pin 9, pin 10 will not be usable for other `Actuator`s.
- `OutputPin` - Low-level pin. Used when direct control of a pin's PWM is required, but using this `Acutator` keeps the interface consistent. You should never manipulate the `arduino`'s pins directly.
- `Battery` - A voltage divider used to read a battery voltage.

- The hardware required to set this up varies from battery to battery. Check the documentation to set up the proper voltage divider and software multiplier.
- You can set a reference voltage (defaults to 5V for the standard Arduino; set this to 3.3V if you're using the Arduino Fio) to properly resolve the analog-to-digital signal.
- A `LO_BATT` message is broadcast if the battery voltage is lower than a user-definable threshold. Instantiate a listener to listen for this message if you want notification that the battery is running low. This is particularly important for LiPo batteries, where the battery can be permanently damaged if it falls below a particular voltage.

A.2.3 Data Collection: Visualization and Collection

Setting up Telemetry

The `DSSLVehicle` constructor needs to contain a call to the `setupTelemetry()` function. This function will set up what data is to be collected.

These are all defined by the user. Code Sample A.3 shows the `Blimpduino`'s implementation for `setupTelemetry()` as an example.

Code Sample A.3: `Blimpduino.setupTelemetry()` Example

```
1 function setupTelemetry(obj)
2     obj.Telemetry.Position = ...
3         Telem({'X' 'Y' 'Z'}, ...
4               {'XLabel', 't [s]', 'YLabel', 'Pos [m]'});
5
6     obj.Telemetry.Velocity = ...
7         Telem({'X' 'Y' 'Z'}, ...
8               {'XLabel', 't [s]', 'YLabel', 'Vel [m/s]'});
```

```
9
10      obj.Telemetry.Battery  = ...
11          Telem('blimp', ...
12                {'XLabel', 't [s]', 'YLabel', 'Voltage [mV]'});
13
14      obj.Telemetry.PWM      = ...
15          Telem({'PWM1' 'PWM2'}, ...
16                 {'XLabel', 't [s]', 'YLabel', 'PWM (arduino)'});
17       obj.Telemetry.Servo   = ...
18          Telem({'servo'}, ...
19                {'XLabel', 't [s]', 'YLabel', 'Servo Angle (arduino)'});
20  end
```

Each field of `obj.Telemetry` contains a different data type to collect: for the `Blimpduino`, position; velocity; battery; motor PWM; and servo angle are all collected. Each field is a `Telem` object that is newly instantiated. The arguments are:

1. **Sub-data** (cell array). It's natural to consider graphing X, Y, and Z position versus time; you can store related data together in the same `Telem` object. Doing so will let you visualize all the data together (see below). Note that this argument is required, even if you have only one variable to track (for instance, `VBatt`). This is because `Telem` inherits from `dynamicprops` - the subdata name is used as a dynamic property which contains the actual data. (See below sections.)

2. **Plot options** (cell array). You can add an `XLabel`, `YLabel`, or `Title` just like a typical MATLAB plot. You can also set whether the graph should plot in real-time; and change the time frame the visualization is available for (watch the last 30 seconds, etc). As always, the full list of options are available in the documentation for the `Telem` class.

Implementing Data Collection

The main way data collection is done is through the `getTelem()` method. This method is called by a `TelemCollector` (more on this class later). `getTelem()` simply runs through all of the `Device` sensing methods (see section A.2.4) and also requests Vicon position and velocity information. A snippet of the `Blimpduino.getTelem()` method is shown below in Code Sample A.4.

Code Sample A.4: `Blimpduino.getTelem()`

```
1  function getTelem(obj)
2
3      [t x y z Vx Vy Vz] = obj.getViconData('central');
4      uptime = obj.uptime;
5
6      obj.Telemetry.Position.X.appendData([uptime x]);
7      obj.Telemetry.Position.Y.appendData([uptime y]);
8      obj.Telemetry.Position.Z.appendData([uptime z]);
9      obj.Telemetry.Velocity.X.appendData([uptime Vx]);
10     obj.Telemetry.Velocity.Y.appendData([uptime Vy]);
11     obj.Telemetry.Velocity.Z.appendData([uptime Vz]);
12
13     Vbatt = obj.Battery.sense();
14     obj.Telemetry.Battery.blimp.appendData([obj.uptime Vbatt]);
15 end
```

Reading on-board sensors is done through `Sensor.sense()` and reading Vicon is done through `obj.getViconData()` on a named marker in the Vicon system. The `DSSLVehicle` maintains its own uptime (time since it was first instantiated), and typically this should be the abscissa on the graphs. Notice the use of `appendData()` here: the points are not appended to a matrix naively, but are added to a `DynamicArray` object. This is further discussed below.

Operating on Data

The data collected by the `DSSLVehicle` are stored in a special `DynamicArray < handle` object. This is used instead of naively appending data to a matrix because it deals better with constantly growing data. Instead of naively growing a matrix one row at a time (actually, MATLAB will allocate another entire matrix; copy over the matrix; add the data to the last, empty elements; and then free the old memory), a `DynamicArray` will dynamically preallocate blocks of memory as old blocks get filled via cell arrays. This significantly reduces the time cost of dynamically allocating memory just-in-time.

But `DynamicArray` data can't be operated on like usual - you need to retrieve the underlying data as a standard array. `Telem.writeData()` writes out all data to the workspace for you. The output argument is a struct whose fields are the data types the `Telem` object contained; the values are the arrays of data. Code Sample A.5 shows how a user might retrieve data from a `DSSLVehicle`.

Code Sample A.5: Data Retrieval from `Telem` Object

```
1 >> positionData = Blimpduino.Telemetry.Position.writeData()
2 positionData =
3 X: [10x2 double]
4 Y: [10x2 double]
5 Z: [10x2 double]
```

To clear out all data, use `Telem.clearData()`. Arguments to both `writeData()` `clearData()` are optional and will only write/clear the subdata property specified. See the documentation for details. `DynamicArray` is useful for more than just telemetry collection if you are in need of fast dynamic memory allocation. See its documentation for details also.

A.2.4 Methods for Actuation and Sensing

All the `Actuator < Device`s objects export a single method: `actuate()`. While it is possible to run your `DSSLVehicle` by directly calling the `Device.actuate()` methods, you should write methods that are wrappers for the lower-level `actuate()` calls. These wrapper functions should accomplish the following:

- **Apply user-definable hardware limits**. For example, the `HBridgeMotor` object can run a PWM output between 0 and 255; for your own `DSSLVehicle`, it might be desirable to limit the output between 0 and 100. Hardware limits on the `Device` level describe what the physical device is capable of handling, while hardware limits on the `DSSLVehicle` level describe what user's desired capability regardless of what it's physically capable of.

- **Simplify hardware calls**. `Device`s that are already provided tend to be tested and known to be working; changing the interface (i.e., input and output arguments) may break someone's code. If you want a different interface, you should write that interface at the `DSSLVehicle` level and write the wrapper code to provide the interface yourself to ensure nobody else's code breaks if they're using the `Device` interface.

- **Let high-level commands encapsulate low-level ones**. A command such as `turnLeft()` might depend on the coordination between two separate motors controlled by an `HBridgeMotor`. `turnLeft()` can call multiple `actuate()` commands to accomplish this; the user can then just ask his `DSSLVehicle` to `turnLeft()` and the hardware will respond to the high-level command appropriately.

- **Maintain a history of `Device` control input**. To run control (see Section

A.2.5), the controller must have knowledge of the current input vector u. Actuation methods should translate the device I/O into this vector for access later.

`Sensor < Device` objects behave just like `Actuator`s, so the above discussion also applies to them (of course they `sense()` rather than `actuate()`).

A.2.5 Implementing an Interface to Run Control

To use control, a `ControlModel` object has to be aware of the hardware's current input u and output y (in the control theory sense of the words). To respect the OOP concepts of encapsulation, it is not good practice for the `ControlModel` to directly examine a `DSSLVehicle`'s properties directly. (Furthermore, every system will define u and y differently, so most of the time it wouldn't make sense to do this anyway). To run control, users must implement three methods. The control method prototypes are shown in Code Sample A.6, below.

Code Sample A.6: `DSSLVehicle` Control Method Prototypes

```
1    function u = getCurrentInput(obj)
2        % Return obj's last commanded input
3
4    function y = getCurrentOutput(obj)
5        % Return obj's last output — from data collection
6
7    function convertUToMethod(obj, u)
8        % Read u, then convert those numbers to physical actuation
```

Actuation methods should record the latest value of u for `getCurrentInput()` to access; likewise, whenever the sensors are activated (typically with `getTelem()`), the latest value of y should also be made available to access.

`convertUToMethod()` is where the plant and object designs meet. The Arduino analog outputs are PWM. Users need to create calibration curves to translate between

inputs physically required by controllers (i.e., thrust outputs, angular velocities, etc.) and the PWM that the Arduino can provide.

A.2.6 Adding Calibration Curves

The `calibrate()` method lets you add your hardware calibration data to your `DSSLVehicle` without hard-coding in. It is an n-by-1 cell array, where each element is a different calibration curve for each of your n inputs u. The first column is the control input, the second column is the translation to PWM. You can also add calibration data from a MAT-file.

For most Arduinos, V_{cc} is 5V, so the scale factor to convert from Volts to PWM is 5/255. For the Arduino Fio, $V_{cc} = 3.3$V. Keep that in mind if you're using the Fio.

To use the calibration data, call the `readCalibration()` function. It uses a spline interpolation to translate between u and the corresponding PWM (or vice versa). You should be translating from u to PWM in your implementation of `convertUToMethod()`, and from PWM to u in your actuation functions (this is for `ControlModel`'s benefit).

A.2.7 `Device` Design

Several `Device` classes already exist. See section A.2.2 for a list of available ones.

There will be times when you need to create your own `Device` classes. These classes are just wrappers around pins, so you'll need to do two things:

1. Build the constructor
    - Take in pin arguments and send them to the superclass constructor
    - Parse any extra options such as hardware limits
2. Build the `actuate()` or `sense()` methods that call `PinManager` I/O methods

All `Devices` inherit from `Actuator`, `Sensor`, or `InOut`, but the only differences between these are the abstract methods they export. Pin assignments are done by the superclass; the individual `Device` constructor's job is just to make sure the pin assignments aren't malformed (i.e., only 2 elements per row, minimum number of pin assignments met, etc).

The exported methods are just wrappers for the `PinManager` I/O. Write the arguments you'll need to call, parse those arguments, and call `PinManager` I/O methods as needed.

Working on the `Device` level requires you understand the `PinManager` interface. A simple tutorial isn't really possible for this. Read the documentation for this class and look at other `Device` code to see how they operate.

A.3 `ControlModel`*: Simulink Design and Operation*

The `ControlModel` is what calculates your control inputs to execute on live hardware. This section discusses how to design a Simulink model for use with HIL control; and what `ControlModel` actually does to facilitate execution. Not only should you read the documentation for this class, but you should also completely understand the *underlying source code* as well. `ControlModel` was made as robust as possible but there are probably still unseen bugs with it.

A.3.1 Simulink Model Design

A Simulink controller/observer designed for HIL execution has a couple of strict requirements. This is because of how the `ControlModel` will execute the Simulink model in real-time.

- **There must exactly 2** `InPorts`. The first `InPort` is for u_k, the second is for y_k. Even if your controller/observer does not rely on u_k or y_k to operate, these

blocks are required (and they're not needed, you can add `Terminator` blocks to them).

- **There must be exactly 1** `OutPort`. This is for the new control input to run, u_{k+1}. It is calculated by your controller based on u_k and y_k.

- **All** `In/OutPorts` **must have their port dimensions set to -1 (inherited)**. This is the default Simulink behavior. The `runControl()` method will change the port dimensions to match the expected number of inputs/outputs before simulation, then gets set back to inherited afterwards. If you require port dimensions that aren't inherited, you might want to write your own Simulink callback function to make sure the port dimensions are correct for your own execution.

Note that the last state x_k is *not* an input to your Simulink model. It is assumed you need to use an observer; if you don't, then $y_k = x_k$ anyhow. It is also not an explicit output; Simulink automatically makes state information available if needed.

You can use variables in your blocks if desired (rather than hardcoding values). This will probably make your Simulink model more robust. Importing variables from the base workspace into the `ControlModel` is simple; this is further discussed in Section A.4.5.

A.3.2 `ControlModel` Operation

On `ControlModel` object construction, the input Simulink model has a couple of simulation options set. It checks for blocks that might contain initial conditions such as `Integrators` or `State-Space` blocks. The solver is set to `ode4`, the `StartTime` is set to 0, and the `StopTime` and `FixedStep` is set to the period defined by the user-defined `controlFreq`. The solver will integrate exactly one timestep.

When control is active, on timer expiration (the clock runs at a frequency of `controlFreq` Hz) the `runControl()` method is called. This method updates the timestep, requests current input and output information from the `DSSLVehicle` and properly formats it for Simulink model execution. It then imports the `CM.SimulinkVariables` struct into the function workspace, if defined; this gives the Simulink model access to any variables it might need to operate. Then, `sim()` gets called and simulates the model for one timestep. It collects and formats the output information properly, then calls the `Vehicle.convertUToMethod()` method to execute the newly calculated control input. Finally, it records the new control input and new state (if applicable) into a `Telem` object.

A.4 HIL Execution

Now that your `DSSLVehicle` is complete, it's time to run it on hardware live. The following sections will walk you through talking to your `DSSLVehicle` and executing commands and control; the high-level, step-by-step process is as follows:

-1. Install the ArduinoIO C code to the Arduino.

0. Ensure Arduino can connect to MATLAB via serial device.

1. Instantiate your `DSSLVehicle`.

2. Set up the `Vicon` and ensure it is talking to the lab Vicon Nexus software.

3. Associate the `DSSLVehicle` to a Vicon Subject.

4. Instantiate a `TelemCollector`; start data collection.

5. Instantiate a `ControlModel`; start the controller.

6. Disable the controller and `TelemCollector` once the hardware exhibits its desired control.

7. Write out the collected telemetry data for further analysis.

A.4.1 Install `srv.pde` to the Arduino

The ArduinoIO API contains a file `srv.pde` that needs to be compiled and uploaded to your Arduino. This file is in `$DSSLHARDWARE/ArduinoIO/pde/srv`. Open the file in the Arduino IDE, then compile and upload it. See the Arduino website for details on how to use the Ardiuno IDE to do this. (**Note**: The latest Arduino IDE, Arduino 1.0, will convert `srv.pde` to `srv.ino`. For backwards compatibility reasons, leave `srv.pde` on the server and let Arduino IDE convert it to the right file format locally.)

A.4.2 Connecting to Your Arduino through MATLAB

You can connect to the Arduino via USB or XBee and MATLAB will recognize it as a serial device. To check for connected devices, just run

```
>> instrhwinfo('serial')
```

This will return all available serial ports. Note that MATLAB is not plug-and-play - if you disconnect/reconnect the device, `instrhwinfo` won't recognize it. You'll need to restart MATLAB (or make a guess as to what serial port your device is connected to).

Once you know what serial port your device is connected to, connect to it with

```
>> arduino(s) % or Blimpduino(s), etc
```

where `s` is the serial port you found from `instrhwinfo()`.

If you're connecting to an `arduino` object, you can play around with its low-level I/O methods at this point; if you're running your own `DSSLVehicle`, you can run some of the actuation methods you've designed for it.

A.4.3 Interfacing a `DSSLVehicle` to Vicon

Once you're connected, the next step is to activate the Vicon process (if your `DSSLVehicle` needs one). George-Henri Baron wrote a good step-by-step tutorial on how to use it manually; the `DSSLVehicle` will hide some of this from you. Basically you'll need to instantiate a new `Vicon` object, then pass that object to your `DSSLVehicle`. Also pass in the name of the Subject as defined in the Vicon Nexus software that you want to associate your `DSSLVehicle` with. Code sample A.7 shows an example of this usage with the `Blimpduino`:

Code Sample A.7: Sample Usage of `Vicon` Object

```
1 >> B = Blimpduino('/dev/ttyUSB0');
2 >> vicon_setup_script; % See George-Henri's tutorial for more Vicon ...
     usage
3 >> V = vicon.GetInstance();
4 >> V.Initialize();
5 >> B.AddViconSubject(V, 'Blimp');
```

You can simply use `DSSLVehicle.getViconData(marker)` to retrieve the position and velocity of a specified `marker` string as specified by the Vicon Nexus software. Position is provided directly by Vicon; velocity is calculated via backwards finite difference. To maintain quality velocity information, `getViconData()` should be called often (probably within the `getTelem()` method which gets called on a clock.)

A.4.4 Using a `TelemCollector` for Data Collection

The `TelemCollector` (`TC`) object is simply a preconfigured wrapper for a MATLAB `Timer` object. The `TC` constructor takes in the `DSSLVehicle` as input. The underlying `Timer` gets its timer expiration callback set to `Vehicle.getTelem()`. Making the `DSSLVehicle` do data collection is very simple: just run `TC.start()`. Run `TC.stop()`

to stop. Code Sample A.8 shows how to use the `TelemCollector`.

Code Sample A.8: `Blimpduino` Data Collection with `TelemCollector`

```
% Assume a Blimpduino object B was already instantiated
>> TC = TelemCollector(B);
>> TC.start;
Telemetry collection started
% Wait a while for data to get collected
>> TC.stop;
Telemetry collection stopped
>> posData = B.Telemetry.Position.writeData();
```

Frequency of data collection can be set by the user (see the `TelemCollector` documentation); the default is a frequency of 200 Hz.

A.4.5 Using a `ControlModel` to Run Control

Operating a `ControlModel` (`CM`) is exactly the same as operating a `TelemCollector`: users toggle a timer on or off and the timer expiration callback function executes via `start()` and `stop()`. In this case, it runs the `CM.runControl()` method which executes the Simulink model designed from Section A.3.

The required arguments for `ControlModel` are:

- The `DSSLVehicle` you're operating on.
- A string containing the path to the Simulink model you're using.
- Number of inputs.
- Number of outputs.
- Number of states (only for models that have observers).

- Initial state (only for models that have observers).

`ControlModel` uses input and output data that is generated by `Vehicle.getTelem()`, so you should have a `TelemCollector` timer running to make sure that data is available for your controller to use.

Code Sample A.9 shows how to use a `ControlModel` and `TelemCollector` to run control.

Code Sample A.9: Sample `ControlModel` Usage

```
% B is a Blimpduino object; all states are observable.
>> TC = TelemCollector(B);
>> CM = ControlModel(B, 'Blimpduino_Controller', 'numInps', 3, ...
    'numOuts', 6, 'numStates', 0);
>> TC.start;
Telemetry collection started
>> CM.start;
Controller active
% Wait a while for the controller to drive the blimp
>> CM.stop;
Controller disabled
>> TC.stop;
Telemetry collection stopped
```

If you need to, you can import variables from your workspace into the `ControlModel` by setting the `SimulinkVariables` property to a struct. Use the `v2struct()` function (a custom function listed on the MATLAB File Exchange and a part of the DSSL-Hardware repository but not part of MATLAB itself) to do this easily. An example of this is shown in Code Sample A.10.

Code Sample A.10: `v2struct` Usage

```
1 >> A = 10;
2 >> B = -5;
3 >> C = eye(2);
4 >> myStruct = v2struct(A,B,C)
5 myStruct =
6     A: 10
7     B: -5
8     C: [2x2 double]
```

Debugging Your Simulink Model

At some point you will need to debug your Simulink model. `ControlModel.runOnce()` may be used instead of `CM.start()` to run the controller for exactly *one* timestep. You can step into the `runControl()` method and then step into the Simulink execution itself. After the timestep is done, the `Vehicle.controlToZero()` method is called to halt hardware execution. The `TelemCollector` must be active even for debugging purposes, since `runControl()` depends on the availability of data which is provided by the `TC`.

A.4.6 Writing Scripts for `DSSLVehicle`s

As of this writing, there is no simple way to queue waypoints for a `DSSLVehicle` to follow - you have to do it one waypoint at a time, and you have to manually feed the `DSSLVehicle` with this information. But just as you would build MATLAB functions and tie them together with scripts, you can use scripts to control the flow of information between your objects. Code Sample A.11 shows a sample script to run the `Frisbee` differential drive robot.

Code Sample A.11: Sample `Frisbee` Controller Script

```
clear all;
clc;

% Initialize Vicon
V = Vicon.GetInstance;
V.Initialize;

% Initialize Frisbee and add a ViconSubject to it
F = Frisbee('/dev/ttyUSB0');
F.AddViconInstance(V, 'Frisbee');

% Set up timer objects to do data collection and control
TC = TelemCollector(F);
CM = ControlModel(F, 'FB_Control_Simulink', 'numStates', 0, ...
    'numInps', 2, 'numOuts', 3);

% Set up the waypoints for the bot to travel in
WAYPOINTS = {[0;0], [1;0], [1;1], [0;1]};

% Start TC collection
TC.start;

% Execute controller to run the waypoints we want
for i = 1:length(WAYPOINTS)
    Xd = WAYPOINTS{i};
    CM.SimulinkVariables = v2struct(Xd);

    [t X Y] = F.getViconData('head');
    CM.start;

    % Execute control until error tolerance is low
    while norm([X-Xd(1), Y-Xd(2)]) > 1e-3
        drawnow;
```

```
33      end
34
35      % Stop the controller and wait a while before using the next ...
            waypoint
36      CM.stop;
37      pause(5);
38  end
```

drawnow and the MATLAB Event Queue

Since all the execution for DHA acts "in the background", events such as callback execution, `notify`s, etc., are pushed into the MATLAB event queue so they may be processed one at a time. If something is actively executing "in the foreground", you need to make sure the event queue is flushed once in a while. Consider Code Sample A.12, which defines a simple `Timer` object:

Code Sample A.12: MATLAB Event Queue and Blocking

```
1  clear all;
2  clc;
3
4  t = timer;
5  set(t, 'Period', 1);
6  set(t, 'ExecutionMode', 'fixedSpacing');
7  set(t, 'TimerFcn', @(event, data) disp('Timer rollover!'));
8
9  start(t);
10
11 while(1)
12     drawnow;
13 end
```

If the `drawnow` call is commented out, the timer does not execute its callback function. The `while` loop is "blocking" execution of the event queue!

What does this mean for you? If you write a script which contains a "do-nothing" `while` loop, presumably to wait for your `DSSLVehicle` to go from point A to point B, you need to include a `drawnow` call at the end of that loop instead of leaving it blank. Feel free to do something else inside the `while` loop, but if the loop executes too long the event queue will get backed up and things won't execute properly.