2015

Phyro: Exploring an Untethered Model for Robots in CS-1

Philip Franchi-Pereira
Bard College

Phyro: Exploring an Untethered Model for Robots in CS-1

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Philip Franchi-Pereira

Annandale-on-Hudson, New York
May, 2015

Abstract

Robots are becoming more popular, both industrially and commercially. As new robots are created, designers must choose whether to tether to a primary computer. Many robots used in an introductory computing context, like the Scribbler robot and Fluke attachment, are tethered. Untethering educational robots is the next step in improving the way robots are used in CS-1. This project aims to demonstrate the advantages of untethered robots, using the Scribbler robot and Fluke singleboard computer attachment as a model. We developed the Phyro library to make programming an untethered Scribbler and Fluke easier for students. By comparing program performance and function execution time, it is shown that Phyro on an untethered Scribbler and Fluke can match and in some cases even outperform a tethered Scribbler and Fluke.

Contents

Dedication

To my parents. Thanks for getting me this far.

Acknowledgments

First and foremost, my advisor Keith O'Hara, without whom this project never would have gotten off the ground. To Jessica, Maya, Quincy, Katie and Darren, who helped me write, test, and prepare both Phyro and this paper. Finally, to all my other friends and colleagues at Bard College who offered kind words of encouragement.

1 Introduction

1.1 Motivation

Robots are used to teach students of all ages, from doctoral candidates in a robotics class to highschoolers in CS-1 and even autistic children in therapy[5]. Many robots used in the commercial and industrial markets are untethered, which is to say that they do not rely on a primary computer to do their computation for them. However, many robots developed for an educational context are still tethered in some way to a computer. As robots gain prominence in computer science and the world at large, it becomes increasingly important to allow them to be able to act independently from a primary computer. This project aims to demonstrate the effectiveness of an untethered approach to programming and controlling robots in an introductory computer science course. The Scribbler, along with its singleboard computer attachment, the Fluke (both of which are discussed in Chapter 2), is one such example of a wirelessly tethered robot, and is the model robot used in this project.

In order to eliminate the tethering aspect of using the Scribbler, it was neccesary to create a new library, Phyro. Phyro allows users to execute code that would normally be written for the computer communicating with the Scribbler. The history of robots in education has impacted the evolution of tethered robots, affecting how we use them in education today.

1.2 Educational Robotics

Robots appeal to a wide variety of students due to their presence in today's media and the tactile, "hands on" experience they introduce to a classroom. Advertising the use of robots in a class would most likely increase enrollment in computer science classes. However, robots can do more than boost enrollment numbers; they have the potential to fundamentally change the way students view their computers. The use of robots has been shown to increase student comprehension in CS-1 classes [8]

1.2.1 Robots in the classroom

Programming with robots can change the way students approach computers by showing them a different model of computation . Traditionally, computer science students learn to program in a linear fashion. As Stein [15]states:

"begin with a question. Describe the answer in terms of the question. Programming is the process of writing down the sequence of calculations required to get from a particular instance of the question to the corresponding instance of the answer. Computation is the process of executing those steps – the algorithm – to deduce the answer to a particular question."

This method of teaching reinforces the idea that programs written in the real world adhere to a very rigid, single process model. However, most programs created today interact with the real world. An App on a cell phone that constantly needs to be aware of users tapping the screen, cars that monitor tire pressure and gas levels, or even websites that ping servers for updates. These processes are not traditional finite procedures. They continually adapt to their respective surroundings. CS-1 classes taught using traditional styles often miss opportuntites to teach students that computers interact with their world on many different levels.

Robots present a novel platform for teaching this new, “world aware” model of computer science. They allow students to see how the programs they write interact and affect their surroundings. Even coding style is affected when programming for robots. While “Hello World!” might still be the first thing students learn to write, more complicated assignments no longer take the form “print out the batting averages of the baseball players in the list provided” and now resemble “make this robot find its way to the next room”. This is arguably not only a more meaningful kind of problem to solve, but can lead to more interesting problems later on.

1.2.2 Origins of Educational Robots

Robots have long been thought of as being educational tools. Ever since the creation of robots designed to demonstrated complex behavior (Elektro in 1939, followed by ELMER and ELSIE in 1948), engineers and professors alike have considered using robots in an educational environment. However, the first robotic learning platform, the LOGO Turtle robot, would not be developed until 1967. The LOGO platform began with the LOGO language, which was originally designed to teach young children the basics of Lisp, a popular programming language at the time. However, after the first year of its use in a classroom setting, LOGO’s designers chose to reinvent the language. By 1969, LOGO had been completely rewritten to work with the new “display turtles”, which today would be called a turtle simulator.

Shortly after the creation of the display turtle, the first turtle robot was created, and unceremoniously dubbed the “floor turtle”. Both the robot and the simulator introduced issues in 1970 that students and professors are still dealing with today. The display turtle consumed a large amount of resources and required a second, smaller computer to drive the monitor (a problem which is thankfully no longer a concern today). The floor turtle was physically tethered to a single, shared computer, and its sensors were too unreliable to be used in class. In the mid 1970's, portable graphics stations were developed to allow schools to have multiple simulators running at once. By the end of the decade, LOGO had been ported to multiple different computer platforms, including the TI 9900, which allowed for up to 28 turtles to exist at once [14].

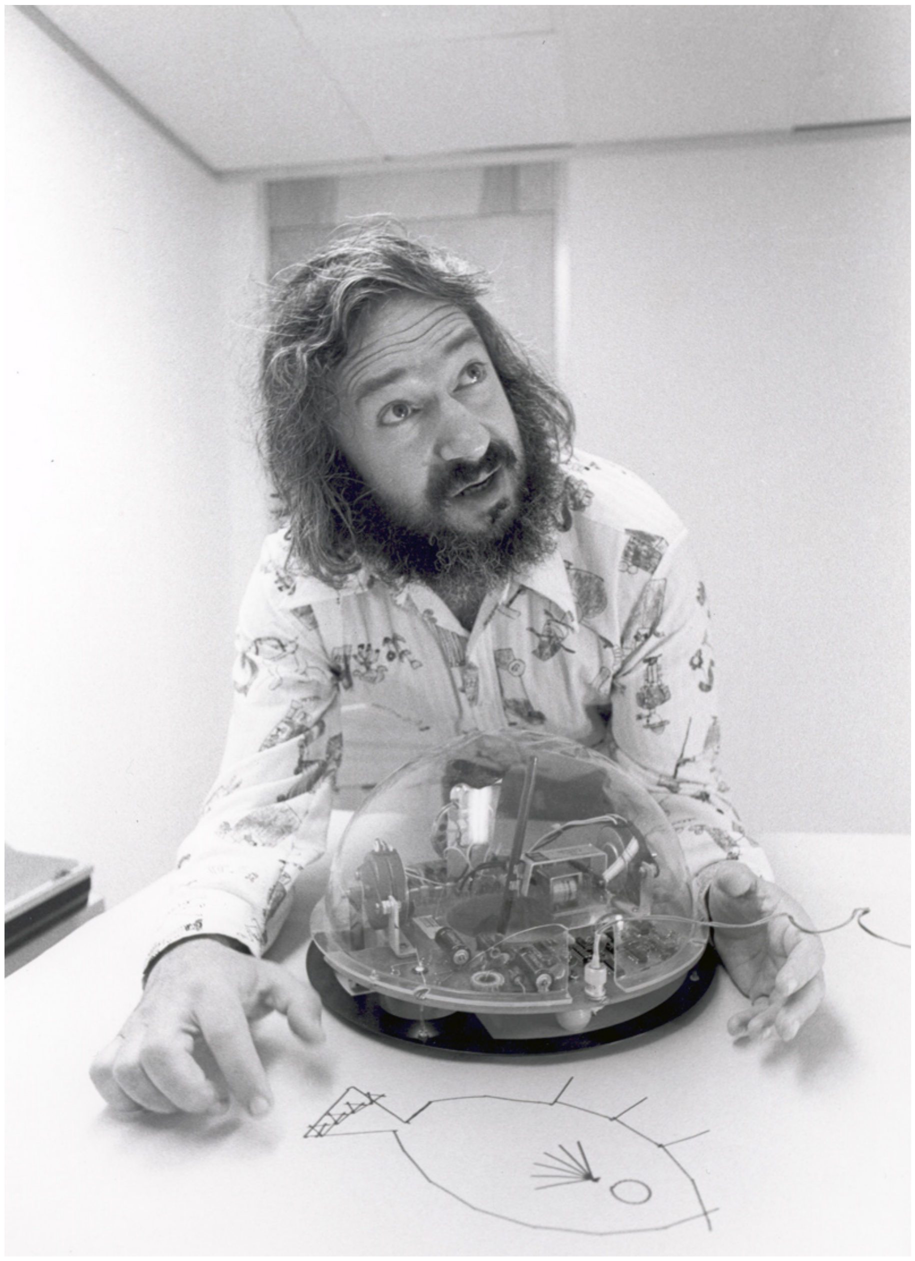


Figure 1.2.1: The Turtle robot along with LOGO's co-founder, Seymour Papert

With the introduction of personal computers in the 1980's LOGO saw a shift in priority from the physical realm of the floor turtle to the virtual simulator. Soon afterwards, the Lego company created Lego Mindstorms. Seymour Papert's book, Mindstorms: Childern, Computers and Powerful Ideas, served as the namesake and inspiration for Lego Mindstorms[7]. Originally called LegoLoGO, Lego Mindstorms were at first exclusively sold to schools. LegoLOGO allowed students to do more with their robots than just draw; they could create their own robots to fit the project. Unfortunately, LegoLOGO robots were still tethered, which led to the creation of the core “brick.” Lego robots were assembled around this brick, which was fully programmable and thus gave students more freedom with their assignments. The brick itself is a Lego piece, allowing it to interface with existing Lego pieces. It was made to be low cost, small and light enough to sell in a kit. While the first prototype was made in 1987, Lego unveiled the first Mindstorms kit in 1998.

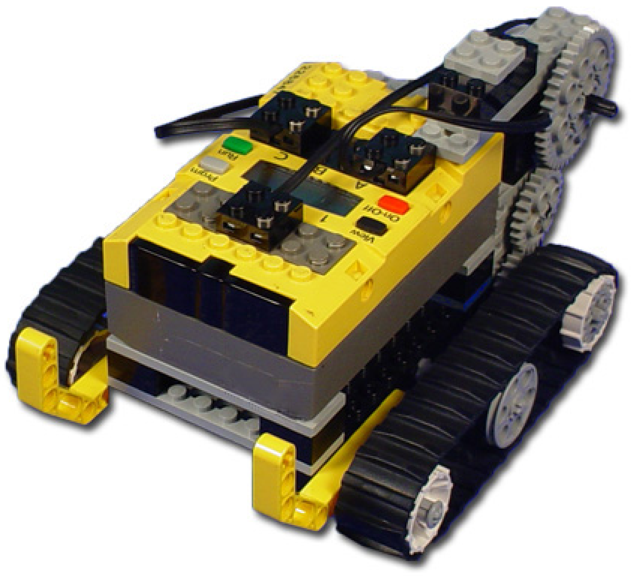


Figure 1.2.2: An example of a robot made with the Lego RCX Kit. Picture provided by www.robotc.net

The first kit, called the Robot Command eXplorers (RCX), came with two motors, two touch sensors and a light sensor. The brick itself boasted a 32KB of RAM and a 16 MHz processor. In 2006, Lego released a second brick, NXT, which came with a three servo motors, one light, sound, and distance sensor[12]. Its brick (labeled Ciara) had 64 KB of RAM and a 48MHz processor. The most recent brick, the EV3, runs Linux on a 300

MHz processor with 64 MB RAM and 16 MB of flash memory. EV3 kits contain two large motors, one medium motor, two touch sensors, one color sensor, one gyroscopic sensor, and one ultrasonic sensor[4]. Students can connect to the brick using either a USB connector, Wifi or Bluetooth, making it easy to program remotely [16].

While Lego Mindstorms certainly made a large impact on the way computer science was introduced to students, professors were not always thrilled to teach a class using Legos. In a hardware course, Lego's make for suboptimal building materials, and in a software course the variance between different students robots would take time away from the curriculum. One of the largest issues was that for most of the early 2000's no simulator existed for Lego Mindstorms, making working outside of the lab impossible for students [13]. Even with these flaws, Lego Mindstorms remains one of the largest platforms used in computer science education.

1.3 Educational Robots Today

In the early 2000's, the number of students enrolling in computer science courses began to decline. [2] Many educators took the opportunity to begin looking for new ways of exciting students about programming and computer science in general. Research began on using robots in an undergraduate and graduate level setting. While Lego Mindstorms was still a viable option for undergraduate students, researchers wanted robots with a "low floor and a high ceiling" [2]; the robot needed to be usable in both low level, introductory courses as well as high level robotics and A.I. courses

Researchers quickly agreed that any robotic platform needed to fulfill the same set of basic requirements. Perhaps the most important goal was that every student needed their own personal robot which could be taken home. Sharing a single robot between multiple students has shown to be an ineffective model for teaching [6]. Having smaller, individual robots that could be used in a dorm allowed students to correct mistakes in their programs outside of lab hours, making them more comfortable with the robot, and by extension the course material [10]. Designers also thought that the robot should be simple to use, but powerful enough to use in higher level classes. Robots needed to be feature heavy, so that the robot could solve more interesting problems.

The iRobot Create was the first of these robots to enter development. The Create is, for all intents and purposes, a Roomba vacuum cleaner that trades its cleaning capabilities for a cargo port. This cargo port happens to contain a DB-25 port for serial communications, giving programmers more control over the robots actions. The Roomba was initially released in 2002, and was an immediate success among hackers. iRobot soon began releasing models with features aimed at hackers, such as a new, open software interface and a Mini DIN serial port that allow the Roomba to be easily reprogrammed leading to the Create in 2009. While the Create is an interesting platform to work on for hobbyists, it is arguably too cumbersome to take home. Furthermore, it was seen as too complicated for a beginner to work on, even with the help of a professor.

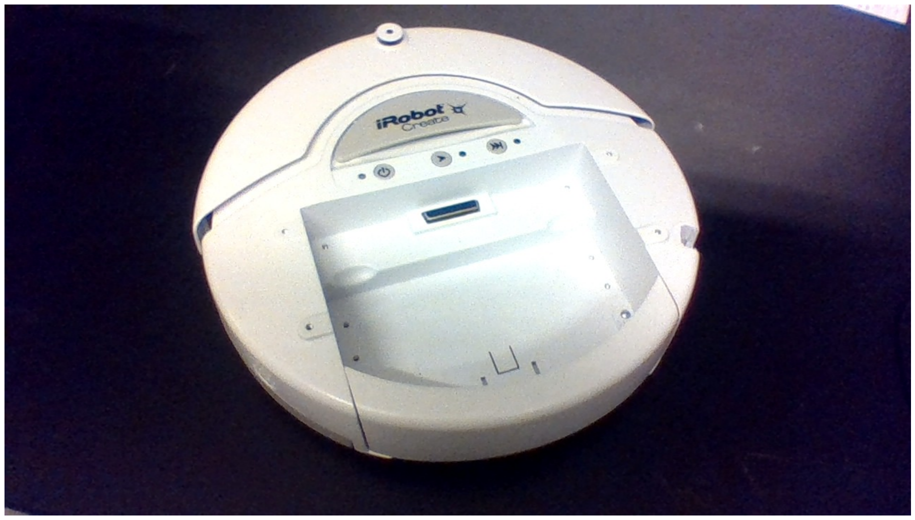


Figure 1.3.1: The iCreate v1, with its cargo port exposed

A study[11] conducted in 2007 used a Create paired with a Qwerk Controller to test the effectiveness of personal robots in a classroom setting. The results of this study revealed the shortcomings of the Create, leading the authors of the study to develop the Finch. The Finch is much smaller than the Create, fitting inside the average student's backpack. Furthermore, it comes with more sensors than the Create, with the current model including accelerometers, obstacle detectors, light sensors and temperature gauges. In order to reduce the cost of the robot, it was designed to be tethered to a primary computer, from which it draws power and receives instructions. In their study, the authors found that most professors they questioned used Java in their beginner level classes, and had no intentions to switch languages [11]. As such, the Finch is programmed using Java, and all of its sensors and actuators can be accessed from a single Finch class. While the Finch comes with significant improvements over the Create in a classroom environment, it still requires a tether to a primary computer.

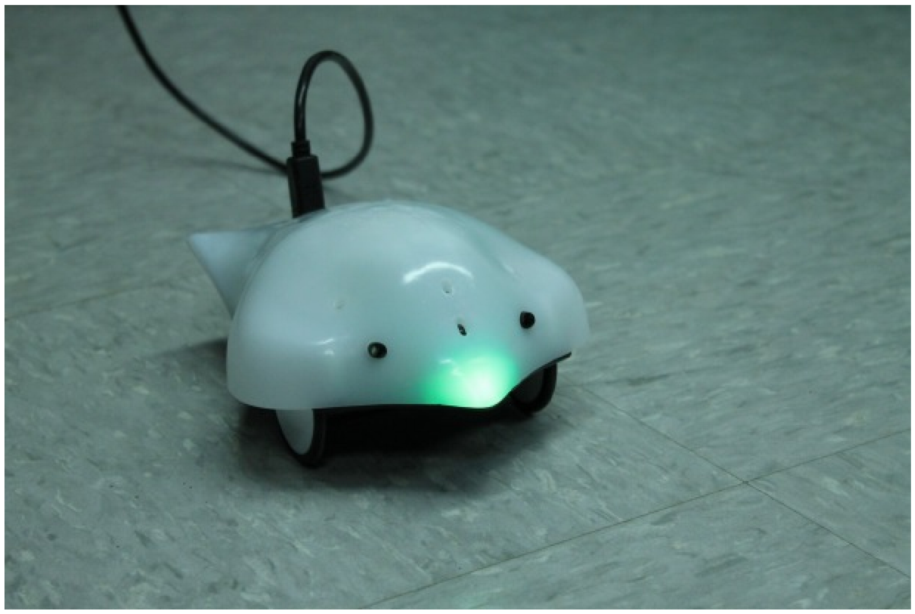


Figure 1.3.2: The Finch Robot

1.3.1 The Scribbler and Fluke

The Scribbler and its successor, the Scribbler2, represent another attempt to create a robot usable in a classroom environment. It was created by the joint effort of Parallax

Inc., Element Products Inc., and Bueno Systems Inc. The Scribbler comes with multiple sensors; 3 light sensors, 2 line sensors, a stall sensor, two IR sensors and an integrated pen port. All of these (except for the integrated pen port) can be with a reprogrammable microcontroller, and are put on display in the built in demos. Parallax created the first Scribbler with a Stamp micro-controller, which is programmed in BASIC. BASIC is easy to learn and gives experienced users a powerful way of creating interesting code. In 2010, Parallax released a new Scribbler, with wheel encoders, a microphone and a Propeller microcontroller. The Scribbler could originally be programmed in BASIC, but with the latest model, it can be programmed using either SPIN or C. SPIN code can be written using Parallax's SPIN IDE, or their visual programming language.

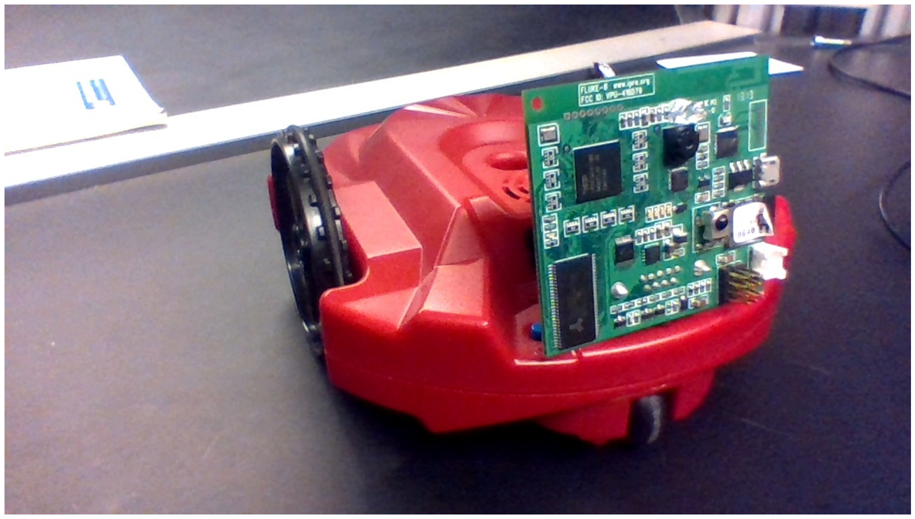


Figure 1.3.3: The Scribbler Robot along with its Fluke attachment

The Scribbler alone does not present any distinct advantages over its competitors. However, with the Fluke single-board computer add-on and the Myro library, students can not only program the Scribbler in Python, but can do so wirelessly over Bluetooth. Currently, the Scribbler and its Fluke attachment is a popular choice for teaching introductory level computer science classes, and is being used in dozens of universities.

1.3.2 Advantages and Disadvantages of a Tethered and Untethered Robot

The Finch presents itself as a model for physically tethered robots. It derives its power from the tether, freeing it of batteries and the variable behavior associated with their low levels. Programs are executed on the computer, allowing the Finch to have a simple microcontroller as its “brain”. Its creators have said that the Finch “is more of a computer peripheral than an autonomous agent”[11]. Tethering the Finch lowers its overall cost, making it more accessible to classrooms.

However, physically tethering a robot to a computer brings several drawbacks. The most important of these is the tether itself. The robot’s range of freedom is limited by the length of its tether. Untethered robots are not hampered by wires tying them to a computer, therefore giving them a larger degree of freedom.

On the other hand, the Scribbler and Fluke add-on are a model to other wirelessly tethered robots. Students can easily connect to the Fluke and control the robots actions in real time from a distance using the Python interpreter. This makes creating and debugging code much easier, as students can quickly see the changes that they made to their source code reflected in the real world.

While untethering a robot confers many advantages, these robots come with a separate list of issues as well. Untethered robots are generally battery operated, and the Scribbler is no exception. The Fluke is a singleboard computer attachment that tethers the Scribbler via Bluetooth to a computer. It makes up for this lack of power available by using less powerful hardware. A low power system means less powerful hardware, which in turn makes the Fluke and other devices like it less useful for a programmer. The Phyro library attempts to provide the speed of using an untethered approach with the simplicity of the wireless tethered approach.

Table 1.3.1. Tethered and Untethered Robots

	Tether	Untethered
Unwired	IPRE Fluke and Scribbler	NXT/Phyro
Wired	Finch	Scribbler

2
The IPRE Learning Environment: the Scribbler, Fluke, Myro and Calico

In 2006, The Institute for Personal Robots in Education (IPRE) was founded. The IPRE was the result of a coalition between Microsoft, Georgia Tech and Bryn MaWr, and was tasked with creating a better educational experience for students new to computer science. The IPRE approached this task by designing a robotic learning platform that was personal, easy to use and affordable. The result of their research was bundled in a Robotic Learning Kit: the Scribbler Robot, the Fluke attachment, the Myro library and the Calico IDE.

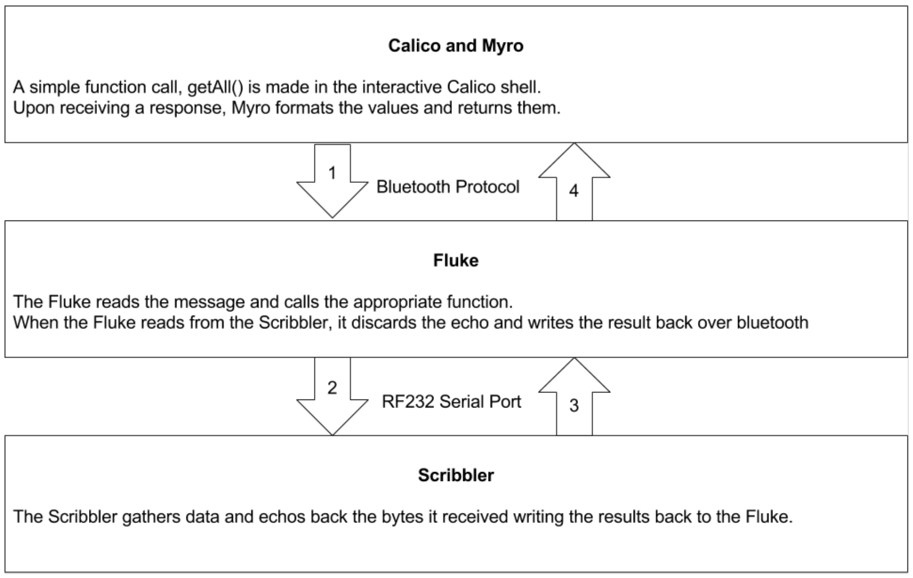

Calico and Myro
A simple function call, getAll() is made in the interactive Calico shell.
Upon receiving a response, Myro formats the values and returns them.
1
Bluetooth Protocol
4
Fluke
The Fluke reads the message and calls the appropriate function.
When the Fluke reads from the Scribbler, it discards the echo and writes the result back over bluetooth
2
RF232 Serial Port
3
Scribbler
The Scribbler gathers data and echos back the bytes it received writing the results back to the Fluke.

Figure 2.0.1: the IPRE Stack

2.1 The Fluke

The Fluke is a single board computer that acts as a middleman between the user and the Scribbler. It utilizes a 180MHz ARM9 processor, 32 MB of RAM, a 1 Megapixel camera, and a Class 1 Bluetooth radio[1]. The purpose of the Fluke is to shore up on the Scribbler's weaknesses; it improves on the Scribbler by allowing its users to program it wirelessly over Bluetooth. In this way students can create programs without needing to worry about a wire, which limits the physical distance their robots can travel while running programs. The Fluke can interface with any device that hosts a serial port, and many other robots have had software written to advantage of the Fluke's flexibility. However, the Fluke designers created it with the Scribbler in mind, and it ships with software allowing it to control the Scribbler by default. The Fluke also brings with it a whole new array of sensors. It has improved IR sensors that allow it to detect obstacles in its path with higher precision, and GPIO pins for attachments like small motors that can be easily controlled. Perhaps the most important feature the Fluke brings to the table is the camera. The camera was added to the Fluke later on in its developement as a meaningful way of introducing two-dimensional arrays to beginner programmers, an area instructors previously had struggled to teach using the basic Scribbler features.

The camera can take full color JPEGs and send them off-board wirelessly.

The Fluke allows users take segmented pictures. After capturing an image, the Fluke highlights the greatest color discrepancy and segments the picture into white, or "1", and the rest of the image in black, or "0". This leads to long strings of identical pixel values in a given row. The Fluke replaces these string with a number representing how many consecutive occurrences of a value it saw in that row, decreasing the amount of data it must process for an image. This process is called Run Length Encoding (RLE).

The Fluke2 has a fully functional version of Slackware Linux operating system running on-board, meaning it can be programmed like any other Linux computer. Upon booting up, the Fluke2 launches its Bluetooth Server process, `fluke2srv`, and prepares to read incoming messages from sent from a paired computer. Upon receiving a message, the process determines what command the user was calling by looking at the first byte of the message, and compares it to a table specified in fluke2cmd to see which associated function needs to be called. Many functions, like some of the sensor and camera functions, can be run entirely on the Fluke. These are simply called and executed locally, and the Fluke sends back a response after they have completed. Other functions need to interact with the Scribbler, either to get data from its sensors or give it commands to move or beep. For these, the Fluke writes the command byte it received from the user (along with any additional bytes passed as parameters) to the Scribbler's serial port. When the Scribbler is given a command by the Fluke, it will reply with an "echo" of the command it was given, along with any data it gathered during its execution. After filtering out the echo, the Fluke writes the results back to the Bluetooth serial port. Any programming language with support for Bluetooth communications can interface with the Fluke; however, the

Myro library, written in Python, is the most common method used by professors in a classroom.

2.2 Inner Workings of the Fluke Server

In order to understand the decisions made during the creation of Phyro, it is necessary to understand the software that runs onboard the Fluke. The Bluetooth server on the Fluke, `fluke2srv`, can monitor new camera images, Scribbler messages, and new Bluetooth data without needing to create new processes for each. It begins running by initializing structures that will contain error logs, camera parameters, Bluetooth communications data and Scribbler serial port data. Before starting any Bluetooth related processes, the server resets the Scribbler, prepares its serial port for reading and sets the camera back to its default gain and exposure settings. Once the Scribbler and Fluke are fully initialized, `fluke2srv` makes itself discoverable and begins listening for packets.

The heart of `mfluke2srv` is the `while` loop which polls the different structs it initialized earlier for new data, and responds to each one accordingly. The first struct polled, and perhaps the most complicated, is the camera struct. In order to capture an image, the system first needs to read from the camera file (`/dev/spidev0.0`) which triggers an image capture routine in the camera. The image is made available 0.4 seconds after the read. After the image is made ready, the section of memory it exists in is shared with `fluke2srv`. The server bypasses this delay when it polls the camera file for changes, which triggers a read and therefore a new image capture if no new data is in the file.

If a function requests a new image from the Fluke, `fluke2srv` will return any image taken in the last two seconds, and only take a new one if no such image could be found. Although the resulting image might have been taken before the `takePicture()` function was called, due to the frequency of camera reads this disparity is rarely noticed.

After checking the camera for any new images, the server checks the Scribbler serial port for any new data and copies it to the serial struct defined earlier. This serial struct is primarily used by the function `scribbler_passthrough()`, which is primarily responsible for reading and writing to the serial port. The server then goes on to check for new Bluetooth clients and any data in the receive buffer before finally transmitting data it sees in the transmit buffer. The server is constantly checking for all of these different events with each pass, which is not always beneficial. A function that wants to simply send a reply to one of the Fluke's clients needs to wait for the camera file and Scribbler serial port to be read, new clients to be identified, and Bluetooth data to be read in.

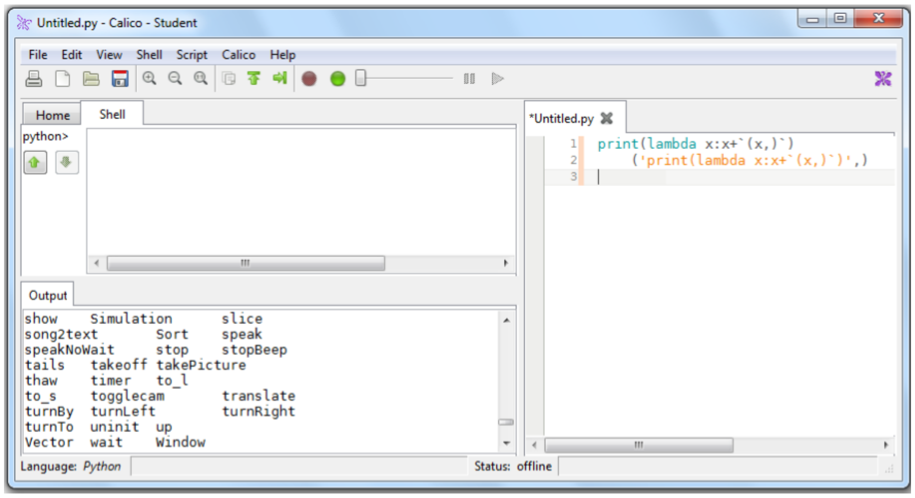
2.3 Myro

Myro is an interface for controlling robots, written in Python. Using Myro, students can send the Scribbler commands over Bluetooth through the Fluke. Myro was created to hide the difficulties of connecting to the Fluke over Bluetooth from the user.

The heart of Myro is the Scribbler class. In it exists a list of all the commands, represented in character form, that control the Scribbler. Myro functions acquire control over the port and write these characters, along with any other parameter bytes, to the Bluetooth serial port. It then awaits a response before relinquishing control. Most functions in Myro look more or less identical to each other, with some functions manipulating either the input or outputs before returning. Following this method Myro can quickly be updated to reflect changes made to either the Fluke or the Scribbler without needing to rewrite entire sections of the code. Having the Fluke and Scribbler handle most of the hard work of gathering and manipulating data means that Myro itself can be changed without, in theory, affecting the Scribbler and Fluke's functionality. Myro has other functions defined in it as well that simply make programming in python a bit easier. Students can call `wait()` instead of `time.sleep()`, and `flipCoin()` instead of `random.randint(0,1)`.

These functions make it easier for students to concentrate on coding the task at hand, rather than difficult, foreign syntax. While Myro was initially written for Python, implementations exist in C#, Java, and C. The Calico IDE goes further, and allows student to use Myro in an even wider variety of languages.

2.4 Calico


Untitled.py - Calico - Student
File Edit View Shell Script Calico Help
Home
Shell
python>
*Untitled.py
Output
Language: Python
Status: offline

```
print(lambda x:x+`(x,)`)
    ('print(lambda x:x+`(x,)`)',)
```

```
show    Simulation      slice
song2text       Sort    speak
speakNoWait     stop    stopBeep
tails   takeoff takePicture
thaw    timer   to_l
to_s    togglecam       translate
turnBy  turnLeft        turnRight
turnTo  uninit  up
Vector  wait    Window
```

Figure 2.4.1: Snapshot of the Calico IDE

The Calico IDE is the final piece of IPRE's Robot Education with the Scribbler. Calico was released in 2010, and was designed to provide a single framework for multiple programming environments with multiple contexts so that instructors and institutions would not need to limit their pedagogical choices. Its creators wanted an IDE students could use to learn multiple languages without needing to also familiarize themselves with a new work environment. Students would spend more time on their coursework and less time adapting to the specifics of a new tool.

Calico comprises four main components: an interface for the different programming languages it supports, an interface for different libraries, another interface for peer to peer communication and a text editor[3]. Calico comes with support for IronPython, Scheme, F#, Ruby and Boo programming languages. However, any language that conforms to the Common Language Infrastructure (CLI) can quickly be added in if a professor chooses to do so. In this way, Calico can share C# Myro with any other CLI language. Any function, and by extension library, written in one language, can switch contexts and be run in another. It is worth noting that Calico does not copy over or translate code from one language to another. Code written in Scheme will appear native to Scheme, even if it uses functions and data structures only present in Ruby.

Myro was one of the main libraries the creators of Calico chose to include in their environment. As such, Calico introduces even more features to enhance the Scribbler's educational utility. Calico allows students to use a graphical simulation of a Scribbler to test their code on the go. Projects that require specific conditions to run can be simulated and the code tweaked in response, so that students can begin the debugging process without needing to use the physical robot. Using the simulator and the Scribbler together means the student can always work on their projects, as long as they have their computer.

2.5 Strengths and Weaknesses

With all of the advantages that the IPRE Scribbler project brings, there are significant compromises and drawbacks that affect its use in the classroom. Perhaps the most irksome issue is power. The Scribbler requires six AA batteries to power itself. Batteries run out quickly, leaving the Scribbler as useless as a paperweight until the student can spend money to replace them. In order to mitigate this, the IPRE Robot Learning Kit comes with rechargeable batteries, and professors are highly encouraged to purchase communal chargers for the class.

More importantly for a classroom environment, Bluetooth is not the most eay-to-use method of communication. There are a variety of different Bluetooth devices on a variety of different platforms, each with their own unique instructions for interfacing with the

Fluke. While it is true that most laptops sold today have Bluetooth attachments, many desktops do not. Once again, the IPRE Robot Learning Kit fixes this issue by including a Bluetooth dongle, compatible with all major operating systems.

Myro is one of the best libraries available for communicating with the Fluke over Bluetooth because of the ease with which students with little to no experience can begin programming for the Scribbler. However, it still has flaws that cause difficulty for students. The Fluke2 and Scribbler2 brought improvements that would make many aspects of Myro irrelevant, as discussed in section 3.2. In order to maintain backwards compatibility Myro had to keep this functionality.

Most of these issues are troublesome, but are not much more than inconveniences to the average student in a beginner programming class. However, one of the complaints students and professors have about the Scribbler and Fluke is the Bluetooth delay. Every byte of data sent out by the Fluke has an overhead cost; sending a single picture takes over a second. This is explored in more detail in Chapter 4. The Phyro Library aims to reduce the amount of time a program spends waiting for a response from the Fluke by porting Myro directly onboard.

3 The Phyro Library

The Phyro library is an attempt to untether the Scribbler and Fluke from a primary computer. Phyro allows students to move their source code from a computer directly onto the Fluke, which eliminates any delay caused by Bluetooth. In this library any commands that would have been originally been function calls executed over Bluetooth are now performed locally by the Fluke.

3.1 How Phyro Works

Phyro was created with three goals in mind, which are listed in order of priority;

1. programs must be able to be executed entirely untethered,
2. programs should be the same in Myro and Phyro
3. finally Phyro must be at least as responsive as Myro.

The first goal was easily satisfied. Phyro can only be used properly on the Fluke, and requires no Bluetooth connection to be run. It is written using only Python and C, and can easily be copied over to a Fluke2.

Phyro's second goal is meant to make using the library easier for students to use. Students should not need to drastically change their source code to have it run on Phyro. Migrating code from the laptop to the Fluke should be as painless as possible, to encourage students to make the transition for some of their assignments. For these reasons, the Phyro interface was kept in Python, which limits its performance.

Phyro's third goal is to perform as well as Myro. Although removing the Bluetooth delay is a start, having Phyro run on the Fluke brings with it its own set of problems. The Fluke is not a particularly powerful computer, and Python is not a particularly efficient language. In order to maximize speed, Phyro functions are actually written in C. Almost every function that interacts in any way with either the Fluke's peripheries or the Scribbler are actually implemented as calls to the C library phyroC. This not only increases the speed with which Phyro programs run, but also makes Phyro accessible to more advanced users, who would rather create programs using C, without changing the way students interact with Phyro (through the python wrapper).

The Writer class in Phyro wraps the functions defined in `phyroC.c` (the portion of Phyro written in C) into Python. The Writer begins by opening the Scribbler serial port, the camera, initializing the error log and prepares the Scribbler for use. All the functions that interact with either the Fluke or Scribbler capabilities make calls to functions defined in this Writer class. In turn, the Writer class handles both reading and writing to the Scribbler serial port, as well as things like grabbing a picture from the camera or receiving sensor data.

Functions in Phyro divide themselves into three camps: Native functions, Scribbler functions and Fluke functions. Native functions are the simplest, as they are implemented entirely in Python, and they are identical in Phyro to the originals. Scribbler functions are simply calls to writer.write() and writer.read(), which write and read to the serial port on the Scribbler. Fluke functions use `ctypes` to call analogous in phyroC, and generally involve either the IR sensors, Fluke LEDs, or the camera.

3.2 Differences

Phyro was designed to allow students to transfer their source code onto the Fluke. The Myro Reference Manual was used to compare parameters and return values[9]. However, many functions were changed, for a variety of reasons. For reference, functions that can be called by a user in Myro are listed in Figure 3.1.1, along with descriptions of their use and whether or not they exist in Phyro. First and foremost were functions that exist in Myro, but were not implemented in the Fluke server. These are listed below

1. getIRMessage()
2. sendIRMessage()
3. setCommunicateLeft()
4. setCommunicateRight()
5. setCommunicateCenter()
6. setCommunicateAll()
7. setCommunicate()

These functions are defined in Myro, but their corresponding `fluke2srv` functions simply return 0 when called.

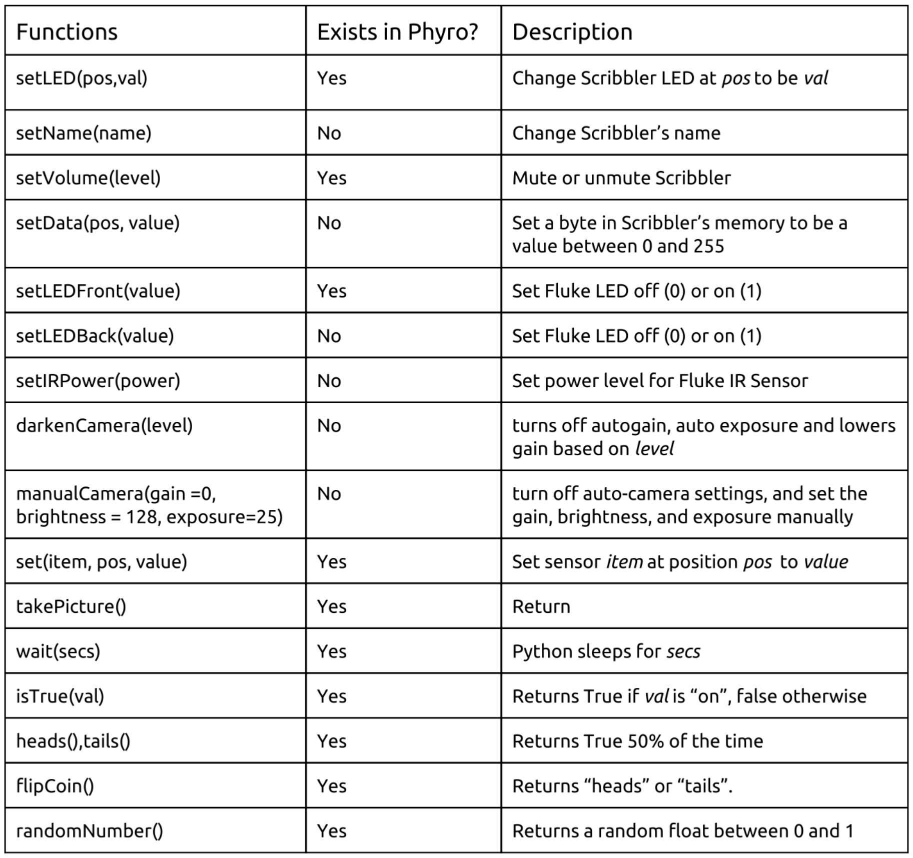
1. getIRMessage()
2. setIRMessage()
3. setCommunicate()

3.2.1 Scribbler Functions

For the most part, functionality remains unchanged from Myro to Phyro. However, below is a comprehensive list of those that were excluded, and explanations as to why:

Functions	Exists in Phyro?	Description
beep(time,freq1,freq2)	Yes	Scribbler makes will beep at the *freq1, freq2* specified for *time* in seconds.
move(trans,rot)	Yes	Scribbler will move forward based on *trans,* and rotate based on *rot*.
getAll()	Yes	Queries all of the Scribbler's sensors and returns the result
getLight(pos)	Yes	Reads a light sensors on the Scribbler, defaults to all
getIR(pos)	Yes	Reads the IR sensors on the Scribbler, defaults to all
getStall()	Yes	Reads the Stall sensor on the Scribbler
getName()	No	Returns the Scribbler's name
getPassword()	No	Returns the Scribbler's password
getVolume()	Yes	Returns 0 for volume off, 1 for volume on
getData()	No	Reads bytes stored in Scribbler Memory
getInfo()	No	Retrieve info about the robot
getBright("left"|"right"|"center"|0|1|2)	No	Reads the Fluke camera to detect intensity of light on the left, right and center. Defaults to all
getObstacle("left"|"right"|"center"|0|1|2)	Yes	Reads the Flukes IR sensors. The higher the value, the more likely there is an object. Max value of 6400.
getBattery()	Yes	Returns Battery Voltage
get(sensor)	Yes	Returns value for the sensor specified

Figure 3.2.1: Phyro Documentation Part 1



Functions	Exists in Phyro?	Description
setLED(pos,val)	Yes	Change Scribbler LED at *pos* to be *val*
setName(name)	No	Change Scribbler's name
setVolume(level)	Yes	Mute or unmute Scribbler
setData(pos, value)	No	Set a byte in Scribbler's memory to be a value between 0 and 255
setLEDFront(value)	Yes	Set Fluke LED off (0) or on (1)
setLEDBack(value)	No	Set Fluke LED off (0) or on (1)
setIRPower(power)	No	Set power level for Fluke IR Sensor
darkenCamera(level)	No	turns off autogain, auto exposure and lowers gain based on *level*
manualCamera(gain =0, brightness = 128, exposure=25)	No	turn off auto-camera settings, and set the gain, brightness, and exposure manually
set(item, pos, value)	Yes	Set sensor *item* at position *pos* to *value*
takePicture()	Yes	Return
wait(secs)	Yes	Python sleeps for *secs*
isTrue(val)	Yes	Returns True if *val* is "on", false otherwise
heads(),tails()	Yes	Returns True 50% of the time
flipCoin()	Yes	Returns "heads" or "tails".
randomNumber()	Yes	Returns a random float between 0 and 1

Figure 3.2.2: Phyro Documentation Part 2

1. getState() - This function was seldom used by students, and since updating Scribbler software was removed from Phyro, this function was removed as well.

2. getData() - This function was seldom used by students, and since updating Scribbler software was removed from Phyro, this function was removed as well.

3. setData() - As data cannot be retrieved, it also can not be set.

4. setSingleData() - see setData().

5. setEchoMode() - This function produces no results, as the Scribbler firmware does not respond to this command being called.

Most of these functions are only used to program the Scribbler directly, and therefore were not included in Phyro. The only exception is `setEchoMode`, which was originally intended to disable the echo the Scribbler sends back with each command. However, the `GET_ALL` function call is hard coded in the Scribbler's firmware, and cannot be disabled.

3.2.2 Fluke functions

Fluke functions in Phyro can behave very differently from those in Myro, as they do not need to either format their results in order to be sent over Bluetooth. Instead, phyroC can hand its values directly back to Phyro without any additional manipulation. Every Fluke function had to be rewritten to return their output to Phyro. However, some functions were more affected than others. Those are listed below.

1. takePicture("blob") - While this functionality still exists, Myro implements a run-length encoded (RLE) version of this function that speeds up transfer between the Fluke and a primary computer. Implementing RLE in Phyro introduces an unnecessary step, so this function simply returns the finished picture.

2. takePicture("jpeg—jpegfast") - saving a picture as a JPEG images not only takes time, but takes up too much space on an already limited system. While Phyro can convert to JPEG, that functionality can not be accessed from Python.

3. takePicture("grayjpg—grayjpg-fast") - see takePicture("jpeg").

Other functions were simply removed from Phyro, as they were seldom used in introductory classes. They are listed below.

1. darkenCamera() - change camera defaults
2. autoCamera() - change camera defaults
3. manualCamera() - change camera defaults
4. setWhiteBalance() - change camera defaults
5. get_cam_param() - change camera defaults
6. set_cam_param() - change camera defaults
7. setLEDBack() - Seldom used
8. getBright() - Seldom used
9. setIRPower() - Seldom used
10. reboot() - Seldom used
11. identifyRobot() - Seldom used
12. conf_window() - Seldom used
13. read_mem() - Used only to program the Scribbler
14. write_mem() - Used only to program the Scribbler
15. erase_mem() - Used only to program the Scribbler
16. set_scribbler_memory() - Used only to program the Scribbler
17. get_scribbler_memory() - Used only to program the Scribbler

These functions typically fall into three categories; functions that involve changing the cameras default parameters (which is typically not used by students), functions that are used to program the Scribbler, and functions that seldom receive use by students.

4
Results

In order for Phyro to be considered viable in a classroom setting, programs using it would need to perform at least as well as they do using Myro. To test this, the Bluetooth delay between the Fluke and a computer was measured and quantified. Furthermore, function call times were compared between the two libraries. Finally, sample programs were compared both quantitatively and qualitatively using both libraries.

4.1 Payload Sizes

It is important to understand the significance of the Bluetooth delay in Myro. Every byte sent from the computer to the Fluke takes additional time to be read, simply because Bluetooth is a slow medium of communication. Figure 4.1.1 shows the average amount of time needed to send payloads of information from Myro to the Fluke.

In order to take these measurements, a function was created in `fluke2srv` that would read in a number bytes from the Fluke's Bluetooth buffer, and then send back a single byte response. The time was recorded as the computer wrote the packets to the Fluke, and then again when it received the response. The results show that increasing the size of

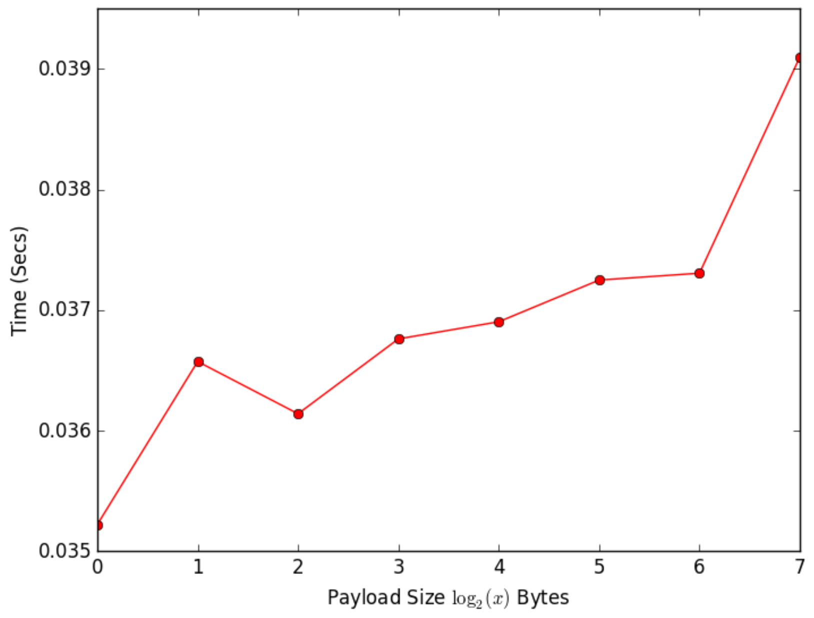


Figure 4.1.1: Average Time Taken to Send Packets vs Packet Size from Myro to the Fluke

the payload does not dramatically increase the time spent sending the payload. In fact, the average execution time for the largest payload of 128 bytes was only on average 0.039 seconds, which is only 0.004 seconds more than the average time taken to send 1 byte. That being said, Myro commands rarely send more than 9 bytes of data to the Fluke. Sequential calls to Myro functions each take upwards of .035 seconds, as they are starting a new write to the Bluetooth serial port each time, instead of writing all their bytes in one pass.

What is perhaps more important is the delay from the Fluke to Myro. The Fluke frequently sends packets of 9 to 10 bytes in the form of functions like `getAll()` and `getName()`. In comparison, functions like `takePicture()` can send kilobytes and even sometimes megabytes of information at time. Figure 4.1.2 shows the average amount of time needed for the Fluke to send a packet of increasing size back to Myro went requested. The results of the figure show that as payload size increases, so does the amount of time needed to send the data. However, the time required to send a payload of data does not double as the payload size increased. In fact, it seems as though the amount of overhead paid for additional bytes after is much smaller than the initial cost of sending any data at all.

Effectively, sending larger packets of data is more efficient than sending smaller packets, and so Myro functions, which rarely need to send more than 4 bytes back to the user, are paying a 0.004 second overhead for the few bytes they are sending.

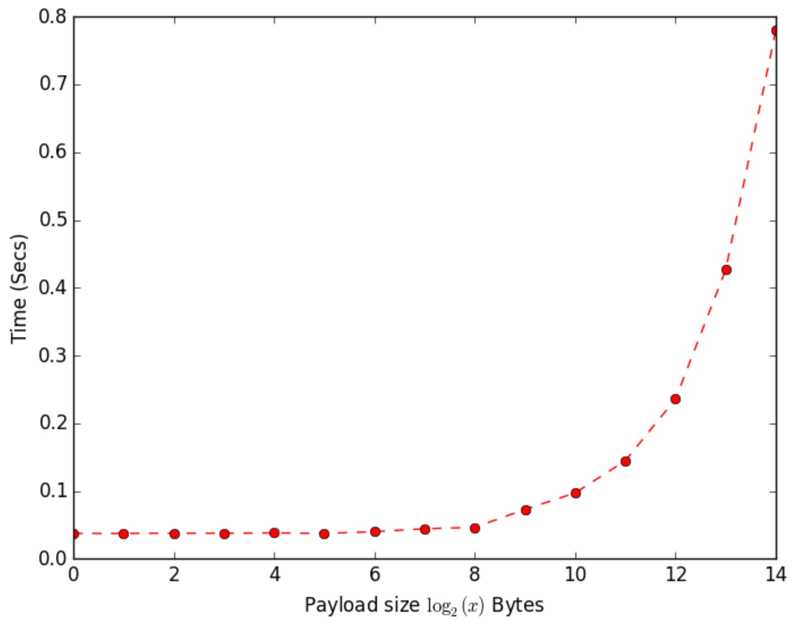
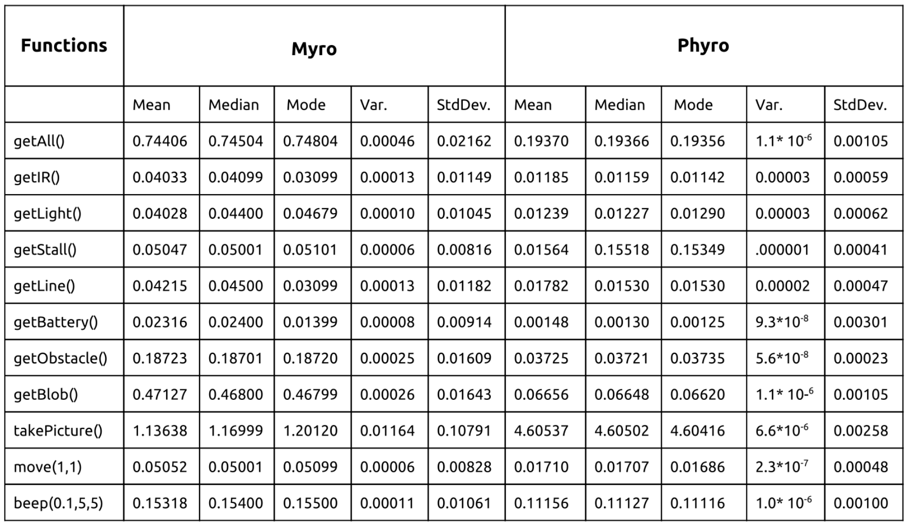


Figure 4.1.2: Average Time Taken to Send Packets vs Packet Size from the Fluke to a Computer

4.2 Micro Benchmarks

One of the easiest ways of comparing the Phyro library to Myro is to compare how long each of their functions take to execute. While a direct comparison is not ideal (programs created in classrooms are rarely as simple as printing sensor data), function calls to a robot can all too often become bottlenecks to performance. Comparing the amount of time a call to `getIR()` takes to execute in Phyro and Myro will at the very least give an idea as to how programs will perform using these libraries. Figure 4.2.1 compares the execution time of functions that students are most likely to use, namely those that read sensor data and control the Scribbler's movement.



Functions	**Myro**					**Phyro**				
	Mean	Median	Mode	Var.	StdDev.	Mean	Median	Mode	Var.	StdDev.
getAll()	0.74406	0.74504	0.74804	0.00046	0.02162	0.19370	0.19366	0.19356	$1.1* 10^{-6}$	0.00105
getIR()	0.04033	0.04099	0.03099	0.00013	0.01149	0.01185	0.01159	0.01142	0.00003	0.00059
getLight()	0.04028	0.04400	0.04679	0.00010	0.01045	0.01239	0.01227	0.01290	0.00003	0.00062
getStall()	0.05047	0.05001	0.05101	0.00006	0.00816	0.01564	0.15518	0.15349	.000001	0.00041
getLine()	0.04215	0.04500	0.03099	0.00013	0.01182	0.01782	0.01530	0.01530	0.00002	0.00047
getBattery()	0.02316	0.02400	0.01399	0.00008	0.00914	0.00148	0.00130	0.00125	$9.3*10^{-8}$	0.00301
getObstacle()	0.18723	0.18701	0.18720	0.00025	0.01609	0.03725	0.03721	0.03735	$5.6*10^{-8}$	0.00023
getBlob()	0.47127	0.46800	0.46799	0.00026	0.01643	0.06656	0.06648	0.06620	1.1* 10-^{6}	0.00105
takePicture()	1.13638	1.16999	1.20120	0.01164	0.10791	4.60537	4.60502	4.60416	$6.6*10^{-6}$	0.00258
move(1,1)	0.05052	0.05001	0.05099	0.00006	0.00828	0.01710	0.01707	0.01686	$2.3*10^{-7}$	0.00048
beep(0.1,5,5)	0.15318	0.15400	0.15500	0.00011	0.01061	0.11156	0.11127	0.11116	$1.0* 10^{-6}$	0.00100

Figure 4.2.1: Execution Time for Functions

It is clear that for most functions, Phyro presents a clear performance improvement. All but one of the functions tested saw immediate improvements in the amount of time it took for the user to receive a response. In fact, many functions in Myro take almost 4 times as long, and for most cases the value of the increase is about 3 times longer.

However, takePicture(), a key function in Myro, shows a significant hit in performance in Phyro. In Myro, it takes on average 1.1 seconds for takePicture to run to completion, whereas in Phyro it takes 4.6. This 3.5 second difference can be caused for a variety of different reasons, two of which immediately come to mind.

Firstly, the Fluke server will trigger image captures multiple times a second, meaning that if a user calls takePicture(), there is usually a picture already in the camera, ready to be processed. Phyro, on the other hand, will trigger the image capture only when takePicture() is called, meaning that the user needs to wait at least 0.4 seconds before the next image is taken.

Secondly, Myro uses `libjpeg.h` to compress images into a low quality JPEG, whose dimensions are 427 by 266. However, Phyro uses the full, uncompressed image, which is of size 1280 by 800. With only 32 Megabytes of RAM, the Fluke2 cannot store more than one or two copies of the image in memory at a time. These two factors most likely explain why Phyro is so much more inefficient than Myro at camera related functions.

4.3 Macro Benchmarks

While function call time is a good metric for comparing Phyro and Myro, students will not neccessarily notice the few milliseconds each individual function can save. They are perhaps more interested in the cumulative time it tooks to run their programs in each library. In order to test how responsive Phyro is in comparison to Myro, four programs were written that utilize some of Myro's most commonly used sensors. They represent the kinds of programs that students would create for a class.

As with all the programs shown below, the same code can be used in both Phyro and Myro on their appropriate platforms. The programs were first run using Myro, and then with Phyro. The Scribbler's battery was changed between the Myro tests and Phyro tests, to remove low battery as a factor affecting performance. All trials were conducted in the same fashion for both Myro and Phyro for each program.

The first program, shown in Figure 4.3.1, will make the Scribbler move forward until it spots an object, and then stop. Although this program is not particulary complicated, something of this caliber of difficulty could be reasonably assigned to a student in an introductory course. In order to compare the performance of this program using Myro to one using Phyro, the Scribbler was placed three feet away from a wall. Table 4.3.1 contains the results of these trials, which show that the average distance between the Scribbler and the wall being 10.5 inches. In contrast, the average distance when using Phyro was 14 inches, which is a drastic improvement. This is most likely due to the Bluetooth delay in Myro slowing down the programs response time to the Scribbler While the Phyro results had a higher variation, the average value was much higher than Myro's average.

Table 4.3.1. Effective Distance in Inches, of the Scribbler's IR Sensor using Phyro and Myro

Trial	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Avg
Myro	11	12	11	11	12	9	11.5	10	10.5	11.5	10	10.5	10	10.5	11	10.7
Phyro	12	12	14	13.5	14	14	15	16	12	13	14.5	14.5	13	13.5	14	13.67

The second program, listed in Figure 4.3.2, is very similar to the first. Instead of using the IR sensors, however, it uses the Fluke's obstacle sensors to detect an object. For this program, 10 trials were conducted, using the same methodology as the trials in the previous example. The results for this trial, shown in Table 4.3.2 also suggested Phyro to be faster than Myro. The average distance from the wall in this case was over an inch farther with Phyro than with Myro. While `getIR()` showed a larger discrepancy between the two libraries, `getObstacle()` has a shorter execution time in both libraries, most likely because it is a Fluke function and does not interact with the Scribbler.

Table 4.3.2. Distance Scribbler stopped from Wall using Obstacle Sensors in Myro and Phyro

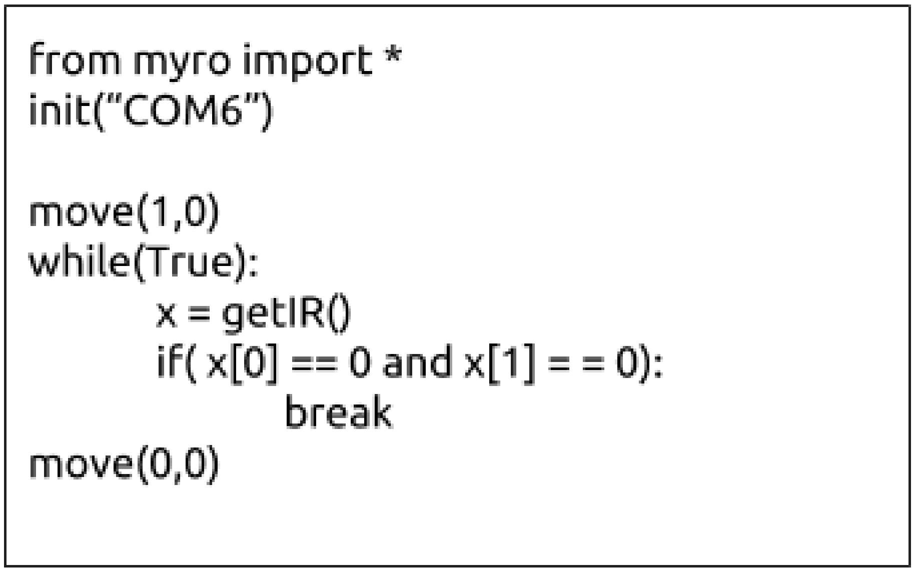
Trial	1	2	3	4	5	6	7	8	9	10
Myro	18	17.5	16.5	17.5	17.5	18	17	18	17	18
Phyro	18	19	18.5	18	17	18.5	18	19.5	17.5	19

The third program tested is background subtraction, and is shown in 4.3.3. This program is arguably the most complicated of the four programs tested. The Scribbler first takes a grayscale background picture, waits for two seconds, then takes another grayscale picture and "subtracts" the background from the new one to see what has changed in the time it has waited. Unfortunately, written in such a poorly optimized fashion, the Phyro process quickly runs out of memory. In fact, taking just the second picture uses up the Fluke2's limited resources. The results shown in Figure 4.3.4 were taken from Myro, which took on average 5.7 seconds to complete when the call to `wait()` was removed.

Then final program is shown in Figure 4.3.5. This program will allow the Scribbler to follow a line, and was considered by the creators to be one of the few programs that Myro cannot effectively perform, due to the Bluetooth delay. After reading in the line sensors, the Scribbler will move forward if both sensors see a line, rotate left or right if only one sensor sees the line, and backwards if no line is detected. In this fashion, the Scribbler can follow a line even through sharp turns and curves, and will retrace its steps if it accidentally runs off the line.

Quantitative measurements for this kind of test are difficult to create. While the average trial time was suggested as a metric, the Scribbler would often be stuck in an infinite loop, moving backwards and forwards as it saw a line, adjusted its course, and in the process of doing so lost the line. ther trials resulted in the Scribbler simply not seeing the line on the page, and moving over it. This may be due to the pigment in the ink of the marker.

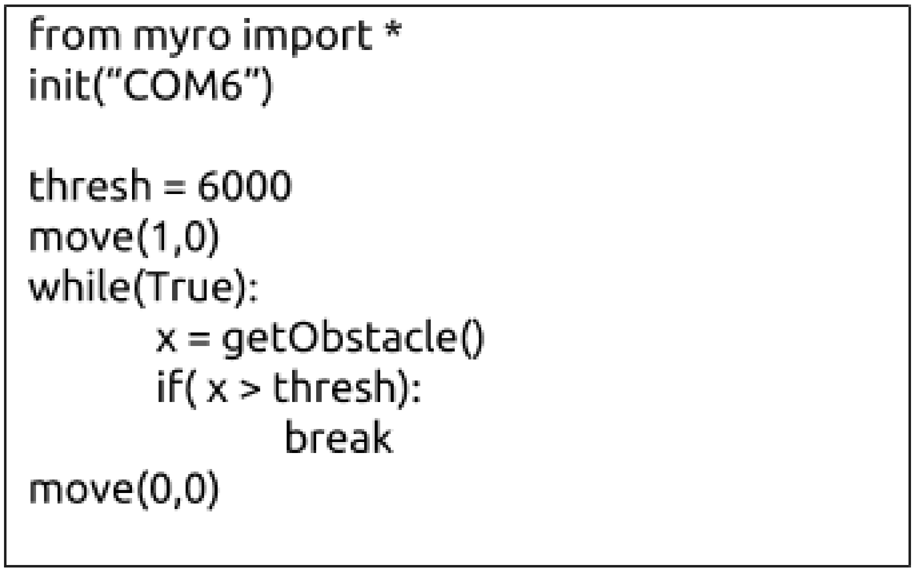
This program showed no dramatic differences in either Phyro or Myro. Unfortunately, the line sensors on the Scribbler are not consistent enough to allow either program to identify the line. Another version of the program was written in such a way that Scribbler would not reverse when losing sight of the line. Yet a third approach would pivot along one wheel instead of rotating in place. Neither of these produced significantly different results. More testing would need to be done, using better markers and more complicated logic in order to aquire more meaningful results.



```
from myro import *
init("COM6")

move(1,0)
while(True):
          x = getIR()
          if( x[0] == 0 and x[1] = = 0):
                    break
move(0,0)
```

Figure 4.3.1: Stop upon Detecting Wall Using IR Sensors



```
from myro import *
init("COM6")

thresh = 6000
move(1,0)
while(True):
          x = getObstacle()
          if( x > thresh):
                    break
move(0,0)
```

Figure 4.3.2: Stop upon Detecting Wall Using Obstacle Sensors

```
from Myro import *
init("COM23")

background = takePicture("gray")
width = getWidth(background)
height = getHeight(background)

newPicture = takePicture("gray")

dummyPicture = makePicture(width, height)

for p in getPixels(dummyPicture):
   setColor(p,makeColor(255,255,255))

for x in range(width):
   for y in range(height):
      oldPix = getPixel(background,x,y)
      newPix = getPixel(newPicture,x,y)
      dist = abs(getBlue(newPix) - getBlue(oldPix))
      if( dist > 35):
         dummyPix = getPixel(dummyPicture,x,y)
         setColor(dummyPix,getColor(newPix))
```

Figure 4.3.3: Background Subtraction Program

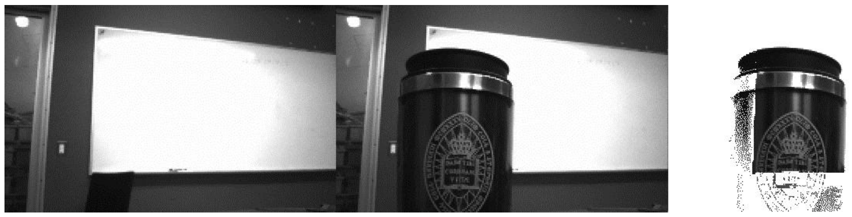
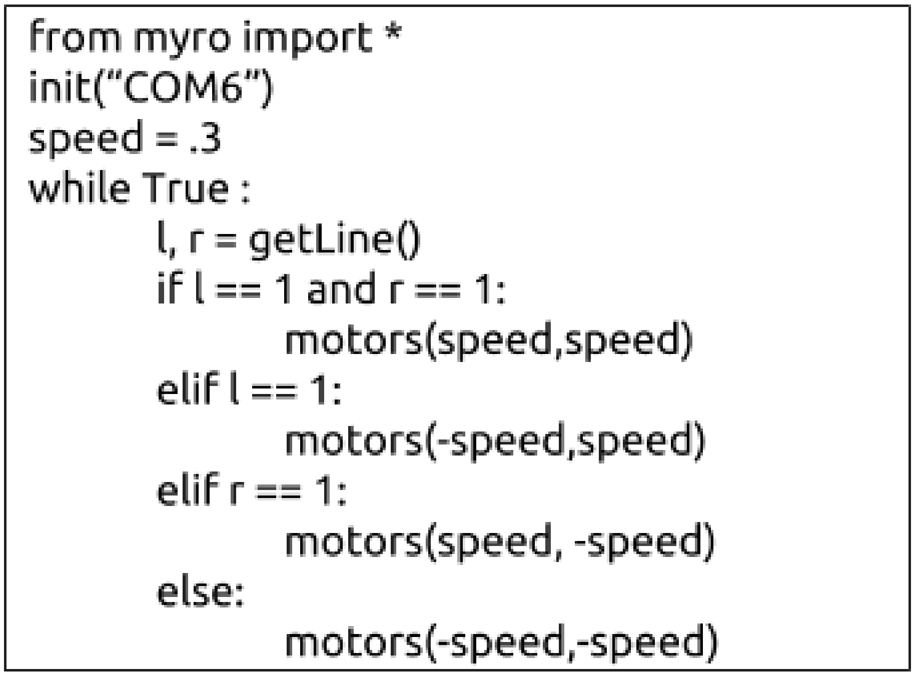


Figure 4.3.4: Images Taken using the Background Subtraction Program. In order from left to right; old, new, result



```
from myro import *
init("COM6")
speed = .3
while True :
        l, r = getLine()
        if l == 1 and r == 1:
                motors(speed,speed)
        elif l == 1:
                motors(-speed,speed)
        elif r == 1:
                motors(speed, -speed)
        else:
                motors(-speed,-speed)
```

Figure 4.3.5: Get Line Program

5 Conclusion

It is clear based on the results of the previous chapter that untethering the Scribbler and Fluke from a primary computer is a very real possibility. The Phyro library has shown improved performance in many areas over Myro. The results highlight Phyro's strengths and limitations; while it can gather data faster onboard than a computer could using Bluetooth, the computer can manipulate data much faster than Phyro programs can on a Fluke2.

Additionaly, while functions in Phyro are generally faster than their Myro counterparts, programs written in Phyro do not demonstrate the same level of improvement. All of the camera's functionality in Phyro relies on its ability to call `takePicture()`, and with that process taking 4.6 seconds, it is unlikely that this family of functions will ever surpass their Myro counterparts on the Fluke2.

5.1 Improving Phyro

Phyro shows improvements over Myro in many regards, but there are still areas which need more development. The most glaring issue in the library is camera functionality. While it is possible to take a picture with Phyro, the process is painfully slow. Future iterations of Phyro must prioritize improving the execution time of camera functions. One possible method of improvement could have Phyro constantly triggering camera captures in much the same way that `fluke2srv` does. Further more, Phyro could compress a captured image into a JPEG before returning it to the Python wrapper, decreasing its size from the megabyte range to the kilobyte range and perhaps making manipulation of the image less resource intensive. While these techniques might improve on Phyro's ability to manipulate pictures, the most severe bottleneck is the Fluke2 itself. Unfortunately, the Fluke has a very limited set of resources, which severely limits Phyro's ability to take pictures. Later iterations of the Fluke2, which would most likely increase the amount of available RAM (or perhaps even enough storage space for swap) might also allow Phyro to make use of the camera.

5.2 Future Works

While Myro's main market is the Scribbler and Fluke, it does have support for other robots as well. In the future, Phyro could also be made to work with other robots as well. Similarly, Phyro will eventually be integrated with either Calico or Jyro (Jupyter Myro). This will allow students to use Bluetooth to move a portion of their code over to the Fluke, and then use that code from their computer. This kind of functionality has two distinct advantages over Phyro's current system. First, this would make the process of using Phyro easier for students, as they would not need to mount the Fluke2's SD card to their computer to transfer their code. Second, and perhaps more importantly, is that users would be able to have the benefit of Phyro's enhanced performance while still being able to use their computers' resources for more intensive processes, like image manipulation. Students could use Phyro to move their Scribbler autonomously while taking and manipulating pictures with Myro. Allowing students to take advantage of the speed of Phyro's function response time with the power of a desktop computer would likely have a significant impact on the kinds of programs students could make in class.

Bibliography

[1] BetterBots, *Fluke 2 Product Information*, `http://www.betterbots.com/cshop/fluke2`.

[2] Tucker and Summet Balch Jay and Blank, *Designing Personal Robots for Education: Hardware, Software, and Curriculum*, IEEE Pervasive Computing **7** (2008), no. 2, 5–9.

[3] Douglas and Kay Blank Jennifer S. and Marshall, *Calico: A Multi-programming-language, Multi-context Framework Designed for Computer Science Education*, Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, 2012, pp. 63–68.

[4] Jordan Crook, *LEGO Mindstorms EV3: The Better, Faster, Stronger Generation Of Robotic Programming*, available at `http://techcrunch.com/2013/01/06/lego-mindstorms-ev3-the-better-faster-stronger-generation-of-robotic-programming/`.

[5] Kerstin Dautenhahn and Werry Iain, *Towards Interactive Robots in Autism Therapy*, Pragmat Cogn, 2007, pp. 1–35.

[6] Barry and Merkle Fagin Laurence, *Measuring the Effectiveness of Robots in Teaching Computer Science*, Technical Report 1, New York, NY, USA, 2003.

[7] Seymour Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, 1st ed., Basic Books, 1980.

[8] Keith J. O'Hara, *Leveraging Distribution and Heterogeneity in Robot Systems Architecture* (2011). AAI3500586.

[9] *Myro Reference Manual*, available at `http://wiki.roboteducation.org/Myro_Reference_Manual`.

[10] Jay and Kumar Summet Deepak and O'Hara, *Personalizing CS1 with Robots*, Proceedings of the 40th ACM Technical Symposium on Computer Science Education, 2009, pp. 433–437.

[11] Tom Lauwers and Illah Nourbakhsh, *Designing the Finch: Creating a Robot Aligned to Computer Science Concepts*, AAAI-10, 2009.

[12] Charlotte Simonsen, *Whats NXT? LEGO Group Unveils LEGO MINDSTORMS NXT Robotics Toolset at Consumer Electronics Show*, available at `http://wayback.archive.org/web/20101109061128/http://www.lego.com/eng/info/default.asp?page=pressdetail&contentid=17278&countrycode=2057&yearcode=&archive=false`.

[13] Elizabeth Sklar, Simon Parsons, and M Q Azhar, *Robots Across the Curriculum* (2007).

[14] Cynthia Solomon, *Logo, Papert and Constructionist Learning*, `https://logothings.wikispaces.com/`.

[15] Lynn Andrea Stein, *Rethinking CS101: Or, How Robots Revolutionize Introductory Computer Programming* (2007).

[16] Audrey Watters, *Lego MindStorms: A Histroy of Educational Robotics*, `http://hackeducation.com/2015/04/10/mindstorms/`.