A Comprehensive Analysis of Rhoades Botball Findings

Aden Makani
Rhoades Botball Club
Rhoades Department of Innovation
San Diego, CA, USA
aden.makani@gmail.com

Ahran Thaper
Rhoades Botball Club
Rhoades Department of Innovation
San Diego, CA, USA
ahranthaper@gmail.com

***Abstract*—For this year's BotBall competition we strategized the optimal method to engineer, design, and program our dual robot-system with a focus on accuracy and reliability. In order to maximize scoring efficiency during the limited time of each match (only 120 seconds), our dual-robot system used specialized multi-threading that utilized two processors. We created an "Alpha" Library (an object-oriented C++ template library) to provide a new level of reliability for controlling robotic motion not possible with previous methods. Deviating from procedural programming and switching to sensor-fused software increased the speed of our autonomous operations and improved consistency of scoring on a complex robotics system.**

***Keywords*—Botball, programming, Rhoades, robotics**

I. Introduction

Teams competing in the BotBall robotics competition face an unforgiving engineering and programming challenge that requires them to design, build, and program autonomous systems that can complete a complex sequence of tasks in 120 seconds or less. These tight constraints leave little margin for error. Any minuscule degree of navigational error or missed sensor reading can lead to catastrophic mechanical collisions or a missed scoring opportunity. In this paper, we present our comprehensive approach in developing a solution for this season's Botball game.

In the past, the use of open-loop control systems (navigation through set amounts of time) presented a significant challenge for our robotics team. For example, programs for our robots were often built using time-based actions that determined how long a battery would supply power to a motor (i.e., "msleep" to run for a predetermined amount of time). Time-based distances resulted in our robots maintaining excessive levels of variability, especially when high accuracy was necessary to perform complex operations. To put it simply, as the robot's battery continues to drain, the distance that it can travel within the same time-based pulse will continue to vary greatly from the previous test run. This is due to the battery capacity decreasing, impacting the voltage it can supply to a given motor. Additionally, variation can also occur from dissimilarity in board aspects such as table surface friction, dust buildup, and mechanical wear on components. This also causes time-based systems to become inaccurate. Our team concluded that we must change to a closed-loop system (i.e., one that continuously monitors the robot's position and makes real-time adjustments based upon measurements from sensors, as opposed to arbitrary timers) [1] in order to generate the required consistency needed to be successful against high-level competition.

We have enjoyed developing interesting hardware-to-software synergy to eliminate environmental variability through more sophisticated, systems-level programming.

II. Literature Review

Historically, C, C++, and Python have been utilized in BotBall. Each of these languages contains both advantages and disadvantages. Our goal was to discern which of these three languages would be most useful for programming our robots and most probable to continue using in the future. We created a scoring system from 1-10 to evaluate each language on its readability, simplicity, and adaptability for different situations. A 5/10 would represent the average for a language, or the bare minimum; a 5/10 in adaptability would represent the language's average compatibility across platforms, such as a language only working on a certain operating system.

A. The C Language

The most basic language tested and mastered by our team members was C. C is a simple yet elegant language, perfect for teaching beginners to our club. This language earned a score of 8/10 in readability, because of its beginner-friendly and objectively understandable syntax. It earned a score of 9/10 in adaptability because of its cross-platform uses and its general ability to handle different types of problems. It also earned a 9/10 score in simplicity because of its readability and clarity in demonstrating different programming principles [2]. We decided to utilize this language to teach basic programming techniques and skills to newcomers in our club.

B. The C++ Language

The C++ language is a powerful yet complex language. Its adaptability lets it work on almost any system and handle almost any problem thrown at it [3]. However, its power is balanced with a disadvantage; its debugging process can be ambiguous and frustrating. This gives C++ a 10/10 in adaptability for its many uses, but a 4/10 in simplicity because of its ambiguity [4]. Its readability suffers from this as well, and while it is manageable, it still gets a score of 6/10. This language was the language used to make the Alpha library, as it is a template library not meant for editing.

C. The Python Language

Python is well known as one of the most powerful, readable languages in the world. Its versatility is a match for C++, and its readability surpasses that of any other [5]. Its syntax is close to that of the English language itself and is therefore perfect for beginners. This gives Python a score of 10/10 on readability. Its beginner-friendliness along with its ability to suit complex projects gives it a score of 9.5/10 in adaptability. However, due to its readability, Python's syntax can be verbose and complex in order to make it understandable by the CPU as well, and errors with these small nuances are almost impossible to debug. This gives Python a 7/10 in simplicity.

D. The Final Verdict

We opted to use C++ for the main programming of this season. We chose this language as we admired its usability and special ease in tasks, such as multi-threading, and debugging related to it. The "Alpha" Library was written in C++ as well. Below is an example of the simplicity in C++ multi-threading versus Python and C.

```
C++:

std::thread t1(func, arg1, arg2);
t1.join();

Python:

t1 = threading.Thread(target=func, args=(arg1, arg2))
t1.start()
t1.join()
```

Fig. 1. This image demonstrates how multithreading in C++ is less verbose than in Python, enabling for faster writing.

III. Concept/Design

Our team utilized an efficient strategy along with both processors to complete several high-scoring tasks around the board.

A. Robot A

We assigned Robot A as our main scorer. Its main objective is to collect, sort, and deliver the drums dropped at 7-second intervals by the drum dispenser. We utilized a dual chamber collection system with a locking mechanism to ensure the drums remain in the container after being sorted by color and stored. While moving these drums from the dispenser to storage, an infrared sensor takes note of the color of the drum and stores it into the appropriate sub-container. The robot then travels to the far drum delivery station, where it delivers sub-container A's drums onto the pole smoothly and efficiently. It then travels to the previously closer drum delivery station, where it drops sub-container B's drums onto the pole. This path may seem counterintuitive. Therefore, we purposely designed this pathway to decrease the total distance traveled to ensure a timely completion point with Robot B at the upper starting box.

B. Robot B

Robot B was assigned as our secondary scorer. Robot B has three main goals: to capture botguy in less than 15 seconds, grab both cones, and to deliver these items into the upper starting box. Additionally, the robot will also push the upper level's Poms into the starting box for extra points. This robot is heavily reliant on our Alpha library for its calculated and expeditious movements.

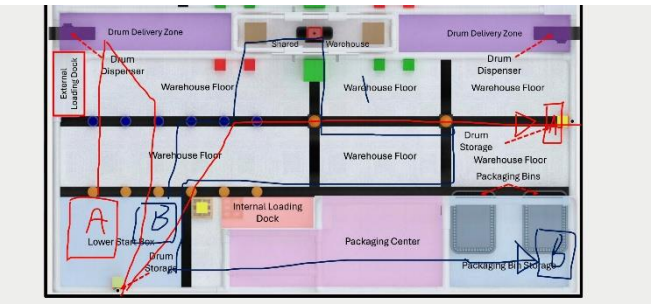

Fig. 2. This map depicts the routes of both robots on the game table.

C. The "Alpha" Template Library

The Alpha library, a student-made, complex, low-level template library, redefined standard functions such as "motor" and "msleep" to similar counterparts such as "Motor" and "Msleep" (case sensitive). These new functions implement the motor position counter, a device which tracks the motor's rotation relative to its axis. This system has been proven to be more accurate than traditional "msleep" which varies with battery percentages [6]. It also includes gyroscopic functions allowing for efficient and accurate turning with little to no variance. We implemented these functions through object-oriented syntax, allowing for each motor, servo, and attachment to be handled separately or in conjunction with one another. Additionally, the library is written in C++ with minor inline AArch64 (The Wombat CPU architecture) ASM. Below is a code snippet of a part of the redefined "Motor" class inside of the "Alpha" library:

```
class Motor;

class motorTraits{
    public:

    motorTraits(const int _speed, const Motor* _motor): speed(_speed), motor(_motor){}

    const uint speed{};
    const Motor* motor{};

    inline bool truth(const uint ticks) const;
};

class Motor final{

    public:

    Motor() = delete;

    explicit Motor(const uchar port_, const uint max_speed_ = 100): port(port_), max_speed(max_speed_){
        assert(port_ >= 0 && port_ <= 4);
        ::cmpc(this->port);
        Motor::all.push_back(this);
        this->gmpcx_handler = std::async(std::launch::async, Motor::gmpcx_waiter);
    }
    ~Motor(){
        this->gmpcx_handler.get(); // joins to make sure it does not become phantom or dangling
        this->freeze(); // to make sure motor doesn't go on forever
    }

    void freeze() const noexcept{
        ::freeze((this->port));
    }

    void cmpc() const noexcept{
        ::cmpc((this->port));
    }

    static void ac() noexcept;

    static int gmpc_avg(){
        int sum = 0;
        for(const motorTraits mot : Motor::running){
            sum += ::gmpc(mot.motor->port);
        }
        return sum/all.size();
    }

class Msleep final{

    public:
    explicit Msleep(const Tick tick_): ticks(abs((ll)(tick_()))){
        for(const motorTraits& mt : Motor::running){
            mt.motor->cmpc();
        }
        // running the motors
        // std::cout << "run start\n";
        while(abs(Motor::running[0].motor->gmpc()) < ticks){
            for(const motorTraits& mt : Motor::running){
                //std::thread([=]{
                    if(mt.truth(this->ticks)){
                        motor(mt.motor->port, mt.speed);
                    }
                //});
                //std::cout << mt.motor->port << '\n';
            }
        }

    }

    ~Msleep(){
        for(const motorTraits& mt : Motor::running){ // cleanup: freezes motors & clears necessary vectors
            mt.motor->freeze();
        }
        Motor::running.clear();
        //std::cout << "clear\n";
    }

    private:
    const ull ticks;
};
```

Fig. 3. This code defines counterparts for the "motor" and "msleep" functions with built-in usage of the motor position counter, known to be more accurate than traditional "msleep." The class created here with its operator overloading helps to simplify the syntax of using the motor position counter, defining its own syntax as almost identical to that of traditional methods (see "Implementation of the 'Alpha' Library").

D. Programming Concept

The programming for both robots is written in C++, a low-level, high-performance language designed for communicating with processors efficiently; this is exactly what is needed for BotBall. We chose this language specifically because of its ease and efficiency with multi-thread operations. The simplicity in performing tasks such as moving a servo while a motor is currently moving is made even easier with the simplicity of C++ multi-threading. This allows efficiency in completing multiple tasks around the board. It also allows a standardized approach to troubleshooting multiple problems and errors with programming, and its uniformity helps us solve such problems with ease. Lastly it helped us to achieve high code readability by utilizing namespaces and classes appropriately.

IV. Implementation

Our team utilized a uniform and strategic sequence when it comes to implementation.

A. Design Implementations

There are 3 steps we follow to implement any sort of strategy, taking it from its idea stage to practice. The first step is design; the idea is formulated into a LEGO or 3D print CAD design. The design is then prototyped and used in real practice. The last step involves modifying the design until it fits appropriately with our specified needs and overall strategy. For instance, Robot A's container to store drums was implemented using this same method; it was designed using CAD software, 3D printed, and its design was reiterated multiple times until it appropriately suited our needs.

Fig. 4. This is a screenshot of the CAD design for the drum container.

B. Software Modules

Software modules are added in a similar step process, with a quality control step added to the end. This end step allows for uniform management of code readability and its use in conjunction with our Alpha library. Software modules are typically limited, as design adjustments are favored for efficiency and manageability. It is also important to note that programming implementations differ from complete robot programs, and they are usually smaller-scale fixes such as strategic maneuvering of the robot to complete its tasks more effectively.

C. Implementation of Our Strategy

To maximize our scoring potential, we implemented a dual-robot strategy, taking advantage of both available processors. We divided the board's objectives based on mechanical suitability and time sensitivity. Robot A serves as our major scorer and contributor. It is responsible for the complex logistics of the drum dispenser, which are released every 7 seconds. Managing this continuous influx of elements required an extremely specialized mechanical design. Luckily, we were provided with a machine meant just for that, the precise and incredible 3D printer. We first modeled a custom, 3D-printed containment system with 2 separate hoppers, meant for catching and storing drums. Furthermore, we engineered a second effector which utilizes an infrared sensor, to determine the color of the drum, and subsequently dispense it into our storage component. A custom locking system, powered by a motor, was secured in place near the bottom of our container, preventing drums from falling during transit. We made it physically impossible for any drum to fall out unintentionally, as we would only release them on their designated storage post. Interestingly, we programmed Robot A to follow an intentional but contradictory path. We decided it will first deposit the first payload of drums at the farthest pole, before working its way back. This method of routing the robot reduced the overall distance the robot needed to travel to completion.

Robot B, the second scoring unit, was created by our team to work alongside Robot A. In terms of scoring. It may in total, only create 30-40% of our entire score, but its function is extraordinarily valuable with regard to receiving high-value assets within a short amount of time. With a high level of acceleration and precision when manipulating objects, the main goal of Robot B is to secure botguy in less than 15 seconds, retrieve both cones, and place the objects on the top level of the warehouse. Whilst Robot B makes its way to the upper starting box, it will simultaneously utilize an effector built on the front to bulldoze poms into the box for bonus points. Because Robot B will have to operate very quickly without interfering with Robot A's sort routine, it will require a very fast and reliable software backbone.

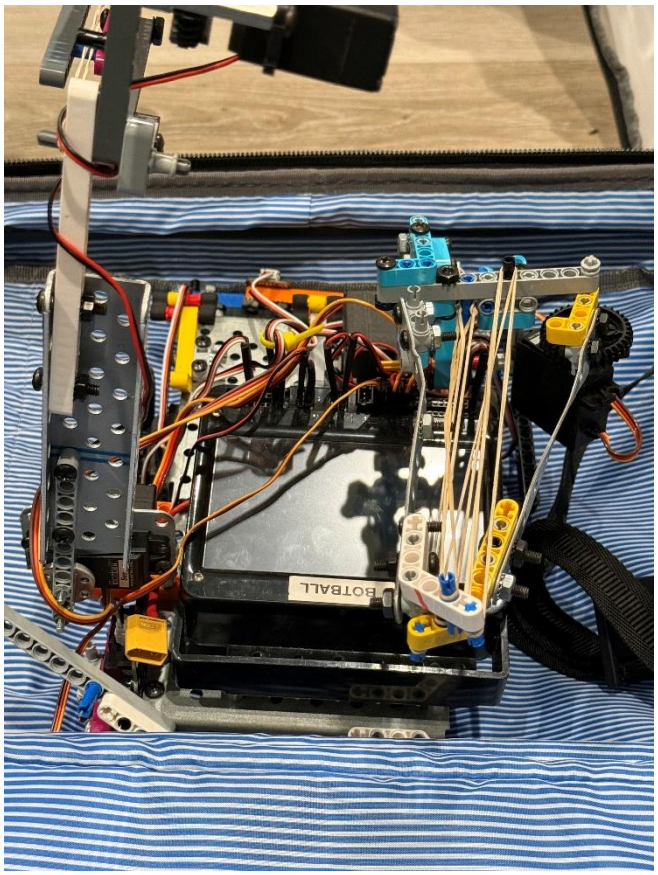

Fig. 5. This is a photograph of the various effectors on Robot B.

D. Implementation of the "Alpha" Library

Both robots' ability to work is accomplished through our collaborative software product - the "Alpha" library, developed entirely by students who took advantage of a complex low-level language, called C++. In using the Alpha Library, we were able to completely rewrite the way our robots usually move; we replaced traditional functions using "msleep" with custom alternatives, redefining KIPR's functions to automatically be converted to Motor Position Counter. The result of this restructuring is that we can now make our way throughout the board with a higher level of accuracy. By programming movements based upon actual sensors, we can create a path that is completely independent from the capacity of our batteries. Another major factor contributing to this capability is our use of gyroscope integration, providing us with perfect turning capabilities, and our use of inline "AArch64" (the KIPR Wombat's CPU architecture) assembly instructions, allowing fast execution of our code. Therefore, the Alpha Library provides us with a mathematically sound base upon which to build our routines. Furthermore, because of C++'s high support for multi-threading, our robots are able to run true concurrent systems and control the steering, effectors, and sensors simultaneously.

The "Alpha" library is one of the main ways to standardize the implementation of software modules. Basing blocks of code on this library and the standards that apply to it help us achieve successful code readability. This also helped address and troubleshoot issues with code from a uniform and unbiased standpoint. This library also used inline AArch64 Assembly to speed up run and compile time. This specific type of code is used in an instantaneous print function which transmits data to the console at impressive speeds, helping effectiveness, and efficiency within tasks. The Alpha library also includes functions involving the gyroscope including its calibration, and implementation across both robots. It is used as an accurate, closed loop turning system. The gyroscopic functions are also able to relentlessly turn until it reaches its target deviation, ensuring temporary or unexpected obstacles do not prove to be significant.

V. Conclusion/Results

This year's creation and use of our dual robot strategy demonstrated a marked improvement over past attempts at solving the problems with autonomous robotics. While developing and evaluating the shortcomings of traditional programming procedures and mechanical designs, we produced an innovative platform that is able to produce excellent scores while also exhibiting enhanced reliability and repeatability. Our approach supports how a generic library can negatively impact our final results. Creating an automated system to consistently excel required creating a personalized library, with a fully automated and sensor-controlled framework.

As seen from our rigorous testing, accurate function from our custom library proved how useful sensor-based movement was. Namely, Robot A was able to consistently perform the task of sorting and delivering eight drums onto the drum storage posts. Our custom library allowed us to manipulate through very complex motions (i.e., lifting and rotating) while maintaining the ability to release each of the payloads from the sub-containers in an efficient manner. In addition, the route of delivering to the most distant drum storage first, and then back towards the lower starting box, was a productive decision saving us significant time.

Furthermore, the primary goal of Robot B was to retrieve botguy within a narrow 15-second time frame. To do this, we used a strategy referred to as "motor over-torquing," which enables the robot to move quicker to retrieve botguy. A crucial constraint forced onto Robot B, was making sure It could complete its task without interfering with Robot A, which would jeopardize their individual goals. When Robot B finished its last objective (parking), the operations of both robots (A & B) finished relatively smoothly, without much stress to either robot.

The biggest success thus far into the season was implementing our Alpha library. We created our own C++ template framework which dramatically altered how our robots could communicate with their surrounding environment. Using a custom version of the "msleep" function which would automatically convert horizontal movement into motor position counter ticks. Subsequently, using this off-the-shelf robot motor function instead of variable-length "msleep" commands assisted in changing our robot's control from an open-loop system to a closed-loop control system. By utilizing the motor encoder count system and the gyroscope, we created enough insulation between the robots' paths and its overall environmental conditions (voltage drop on battery and/or how much friction there is on the table, etc.). Our gyroscope code made sure a turn programmed to turn 90 degrees, will always be exactly 90 degrees regardless of if it was the 1st or 50th time running it each day. In addition, we created a meticulous method for evaluating the gyroscope's accuracy called the "Hand of God", where we would physically prohibit the robot from turning. This test was meant to determine if the robot would incessantly attempt to reach its target position

regardless of the obstruction. Our results surprised us, witnessing the power of the gyroscope. Without the gyroscope's accuracy implemented in the Alpha library, none of our strategies would be possible.

Previous years of open-loop programming created operational restrictions on controlling the robot. Currently, programming with C++ allowed us to utilize processes which aided our ability to complete tasks in a more efficient manner. For example, Robot A had a major issue with time constraint; however, our Alpha library provided us with easily accessible multi-threading, saving us vital seconds. Our current local results show that producing a quality robot alone will not ensure consistency. Therefore, we must first create reliable software with reproducible results. With the extensive capabilities of our object-oriented syntax from the Alpha library (e.g. encoder-based tracking, or multi-threaded development), we have taken our robotic capabilities from simple automation systems to dependable, consistent, and autonomous robotic potentials. By using both namespaces and classes in C++ for robotics, we have standardized our debugging process, allowing us to make much quicker iterations and fix problems with greater ease. In the future, we plan to use the Alpha library to explore the feasibility of implementing PID (proportional, integral, derivative) control to effectively allow smoother acceleration and line following. Overall, through improved multi-threading efficiencies and more reliable navigation, the Alpha library has allowed us to develop a standardized and highly dependable coding platform for our team's benefit in the future.

Acknowledgements

We would like to thank PRIA and all of its staff for giving many students an educational and enjoyable robotics experience. Our thanks also goes to the Rhoades School, which has funded our Botball team. Gratitude is also due to our technology teacher Jeff Major, who introduced us to Botball and offered his time and resources to help our preparations. We deeply appreciate our family without whose constant support and committment we would not be able to to pursue our dreams. Finally we would like to thank our friends and encouragement, for their constant support, especially Zubin Sikchi, for his assistance in the editing of this paper.

References

[1] M. Ragno, "Open-Loop vs Closed-Loop Control Systems: Features, Examples, and Applications," *RT Engineering*, Aug. 09, 2023. https://www.rteng.com/blog/open-loop-vs-closed-loop-control-systems

[2] GeeksforGeeks, "C Language Introduction," *GeeksforGeeks*, Feb. 12, 2014. https://www.geeksforgeeks.org/c/c-language-introduction/

[2] "CSDL | IEEE Computer Society," *Computer.org*, 2026. https://www.computer.org/csdl/proceedings-article/latice/2014/3592a064/12OmNyUWR7M (accessed Apr. 09, 2026).

[3] U. Kirch-Prinz, P. Prinz, J. And, and B. Publishers, "A Complete Guide to Programming in C++." Available: https://www.idpoisson.fr/volkov/C++.pdf

[4] IBM I, "Ambiguous base classes (C++ only) " *Ibm.com*, Oct. 07, 2025. https://www.ibm.com/docs/en/i/7.4.0?topic=only-ambiguous-base-classes-c (accessed Apr. 10, 2026).

[5] Python Software Foundation, "What Is Python? Executive Summary," *Python*, 2025. https://www.python.org/doc/essays/blurb/

[6] A. Irwin, "Our Tests Show How Battery Percentage Can Affect EV Acceleration," *Car and Driver*, May 06, 2025. https://www.caranddriver.com/news/a64504438/ev-state-of-charge-acceleration-effect-test/ (accessed Apr. 10, 2026)