January 17, 2019

Thermonuclear Falcons
18-0288

January 17, 2019

Introduction

As with any Botball team, the code and programming portion of the robots is an integral component of their functionality and usability. The code's purpose is to allow the robot to sense its environment, move around and interact with it, and organize its actions into a sequence that can be performed on the game board to complete objectives. This means it allows the robot to orient itself, controls its movement, and allows it to pick up or move objects into designated areas to score points. The code was written by our programming team, consisting of myself (Ziad) as the lead, along with programmers Juntae, Maytham, Oroni, and Waddah.

Best Practices

Criterion	Completed?	Notes
Code uses functions to organize code	✓	The code contains a wide variety of functions that help keep it organized
Code uses comments to document functions' purposes	✗	Comments are included rather sparingly and are usually used as notes to self or others rather than real documentation
Code includes comments documenting arguments and return values of functions	✗	Return values of functions are usually emphasized by variable or function names and are not commented
Variable names are descriptive and convey their use in the code	✓	Variables are clearly named with identifiers that clarify their purpose in the code
Code avoids usage of unnamed constants other than 0, 1, and/or 2	✓	The code contains a `namespace` for constants, which are identified by descriptive names
Code is formatted to show flow of control	✓	The code is formatted clearly and allows a reader to identify the flow of control
Comments do not contain blocks of old or unused code	✓	Old code is instead either deleted or accessed using version history on our Github repository

The code is structured in a logical and easily readable manner, as well as being free of potential bugs that may be difficult to notice, such as memory leaks or typing issues. Functions and variables follow specific naming conventions which not only help organize them, but also clarify their purpose. Important values are either defined as preprocessor macros or passed in as arguments to avoid reliance on ambiguous “magic numbers”.

However, the code does not have comments explaining the use of functions such as `robot::direct()` or `delta_angle()`, which may hinder the reader’s understanding of what these functions do. The code also does not comment on the return value of some functions, which may also hinder understanding of the code.

Reliability

The code is highly reliable and safe for operation for a number of reasons. It is structured in a way that clearly emphasizes the flow control and makes the robot’s process of operation very easy to understand. The robot combines a coordinate-and-bearing-based positional system in combination with the compass sensor to determine its position and orientation at all times.

The code allows the robot to detect errors in its position using the compass while it is driving in a straight line, combining it with a mathematically optimized curve function to make the robot smoothly set itself back on course if it is skewed or tilted off the correct angle. Additionally, the code facilitates significant amounts of error checking through use of compile-time error checking using templates and `static_assert`. This ensures that errors can be caught during compilation, without a need to even test the robot on the board. The code also uses multithreading for certain processes, and as a part of this we ensured that there are no deadlocks in the code that could cause conflict over memory addresses. Finally, the clear and well-organized object structure of the code and its inheritance patterns makes the code more reliable to use by including generalized functions usable with either robot while differentiating their functionality slightly to suit each robot using `virtual` functions. This reduces confusion over parameters or purpose when using different robots.

However, the code is not perfectly reliable and can be improved. The use of the tape or ultrasonic sensors could be added in to reinforce the robot’s positional system, as well as using the accelerometer in combination with double numerical integration to determine the robot’s forward movement more accurately.

Maintainability

As mentioned earlier, the code uses an object-oriented structure and places the Wallaby functions such as `mav()` beneath a layer of abstraction that makes them more user-friendly and efficient to use. Both the `Demobot` and `Create` classes inherit from a base `Robot` class that includes the base layer for actions such as movement and orientation. Individual `Servo` objects are then added as components to a respective robot and accessed using the dot “`.`” operator as a class member. This structure makes it incredibly easy to determine which function belongs to which class and what its purpose is, allowing any relevant additions or modifications to be made to the correct function.

Important or relevant constants, such as the robot’s default speed, are placed into their own `namespace` called `def`, preventing the use of magic numbers that obfuscate code and make it difficult to maintain. Functions and variables are named rather clearly as to

indicate their purpose, which helps easily identify the flow of the code and where modifications ought to be made. One of the most important aspects of our code's maintainability is the fact that we have set up a private Github repository that allows us to keep our local versions constantly up to date without having to rely on USBs or store them on the Wallabies, as well as providing us with a backup of the code should we have any issues with a Wallaby. This also helps us keep track of what changes have been made to the code through commit messages, allowing all programmers to know how the code's latest version works. Comments are also left on the code on buggy or untested portions of the code to let other programmers know what requires fixing or what a source of error may be.

The one way to improve our code's maintainability is to improve its documentation and commenting. At the moment, there are some functions that use code that can be complex, or are included in one file and used in another with little explanation. Including comments to explain a function's purpose and behavior would make it far easier to maintain and use the code in the future.

January 17, 2019

Effectiveness

The code is highly effective at performing its task. The coordinate system mentioned earlier allows it to keep track of its location and face or go to a certain point on demand, helping us trace a path using very simple and easy instructions. Additionally, the robot uses multithreading to allow it to perform multiple functions at once, such as moving a servo while operating the drivetrain. This is highly effective since it makes the movement much smoother and allows the robot to perform tasks very quickly.

The code could be made more effective by including communication between the robots. This would save a lot of time on the board by allowing one robot to detect randomized objectives using the camera, for example, while the other robot performs them immediately without having to have one robot perform both the checking and the objective itself.

Code Excerpt — Before

```
class robot {
    public:
        dv::camera camera;

    protected:
        double _theta = 0;
        point _position;
        virtual void direct(int, int, int) = 0;

        double deltaTheta(double newTheta) {
            return (newTheta - _theta);
        }

        double deltaTheta(point goal) {
            double dx = goal.x() - _position.x();
            double dy = goal.y() - _position.y();
            double newTheta = atan2(dy, dx);
            return (newTheta - _theta);
        }

    public:
        robot(
            double startingTheta,
            double startingX,
            double startingY)
            : camera(
                CONST::CAMERA_CONF,
                0,
                1,
                2,
                CONST::POINT::LEFT,
                CONST::POINT::CENTER,
                CONST::POINT::RIGHT),
            _theta(startingTheta),
            _position(startingX, startingY) {
        }

        virtual ~robot() {
        }

        inline const point& position() const {
            return _position;
        }
```

This is our initial setup of some of the code we were using, which we decided to build upon and improve. The code did not perform any compile time checking, and relied solely on coordinates in order to orient itself and move around the board, with essentially no sensory input involved.

It relied almost entirely on the `_position` member in order to give information about its location, while `_theta` was updated using estimations by trigonometric formulae rather than real sensory data. The code also included significant amounts of magic numbers and some convoluted variable and function names that were difficult to read.

This was of course improved by including more compile-time checking using templates, as well as incorporating the compass sensor in order to improve the accuracy of the robots. It has also been organized by adding more descriptive variable and function names and eliminating any inclusion of magic numbers in favor of `static constexpr` constants that are clearly named and defined.

January 17, 2019

Code Excerpt — After

```
template    <typename angular>
class   robot
{
    static_assert(is_angular_sensor<angular>::value, "Must be an angular sensor");
public:
    using angular_t = angular;
protected:
    std::unique_ptr<angular_sensor> __angular;
    bot::clock                      __clock;
    std::thread                     __calibration;
public:
    robot() : __angular(std::make_unique<angular_t>()) {  }
    robot(float s_theta) : __angular(std::make_unique<angular_t>(s_theta)) { }
    void    calibrate(uint32_t speed)
    {
        static_assert(std::is_same<angular_t, compass>::value, "Rotation values must be used with compass");
        // Should take approximately 20000ms since that's the calibration period
        // for the compass
        int time = 20000;
        auto    __turn_fn = [this, speed, time]()
        {
            // Sleeps to wait until the compass says "Go"
            msleep(2100);
            direct(speed, -speed);
            msleep(time);
            direct(0, 0);
        };
        __calibration = std::thread(__turn_fn);
        __angular->calibrate();
        __angular->start();
        __calibration.join();
    }
    void    calibrate()
    {
        static_assert(std::is_same<angular_t, gyro>::value, "Zero-argument function must be used with gyro");
        __angular->calibrate();
        __angular->start();
    }
    virtual ~robot() = default;
    virtual void    direct(int speed_l, int speed_r) = 0;
    virtual void    turn(float offset, uint32_t speed = def::turn_speed)
    {
        float   __start = __angular->theta();
        float   __final = unit_circle(__start + offset);
        direct(-sign(offset) * speed, sign(offset) * speed);
        while (!error_minus(__angular->theta() - __final, 0.01f))
        {
            msleep(def::sleep);
        }
        direct(0, 0);
    }
```

The code above makes use of `static_assert`, compile-time checking using templates, displays the compass calibration function, and uses clear variable, function, and parameter names. It also includes no magic numbers (an identifier from `namespace def` is actually used above) and uses the compass to ensure accuracy of turning.