MOE, Miracles of Engineering

FTC Team 365

2019-20 Control Award Submission

Introduction

Preliminary program creation: C19

Programming laptop setup: C32

Template for tests: C45

Github: C97

Throughout the design of our robot, we have kept one universal theme in mind.

User Friendliness: The measure of how robust, simple, easy to maintain, and easy to use a robot is.

To accomplish this goal of user friendliness in Autonomous and TeleOp, we tried to keep the number of important components on the robot (sensors, motors, etc...) to a minimum while still vying to accomplish our goals in mind. The result has been a robot that places a greater priority on intricate algorithms than sensors.

Our robot utilizes a good number of sensors, but wherever one can be omitted (for example: a camera rather than multiple color sensors), we take that option. This results in less environmental variables that can impact robot performance, as the robot relies on its algorithms and math to do computation in the place of sensors that could sometimes provide faulty data.

When driving the robot, we try to keep controls as simple as possible in order to allow the driver to focus on making important decisions rather than be distracted or bothered with controlling the robot.

Along with programming for the sake of the robot in competition, we have also programmed for the sake of learning (such as creating our own Neural Network!) to involve ourselves in other forms and kinds of programming. For an explanation on our thought process & more experimental procedures, view the Additional Summary Information. Also, below most titles will be a listing of notebook pages grouped together by what stage in the development process they show.

The 6 main sections are as follows:

1. Autonomous Objectives
2. Sensors Used
3. Key Algorithms & Constructs
4. Driver Controlled Enhancements
5. Autonomous Program Diagrams

Table of Contents

Introduction *2*

Autonomous Objectives *6*

Autonomous Routine 6

Algorithmic & Programming Objectives 6

Sensors Used 7

Key Algorithms & Constructs *9*

Field Grid & MOEPS (MOE Positioning System) 9

Localization 12

Multithreading 18

Turning Methods 19

Jump Point Search/A*/Dijkstra's Pathfinding Algorithm 24

Proportional-Integral-Derivative (PID) Control 27

Skystone Detection 29

Driver Controlled Enhancements *31*

Adjustable Field-Centric Movement 34

Autonomous Routines 37

Additional Summary Information 38

Autonomous Objectives

The following objectives are what we planned for in our robot's autonomous modes.

Autonomous Routine

- Stone Delivery x1 – dropping off one skystone (10 pts.)
- Repositioning – moving foundation to the building zone (10 pts.)
- Navigating – parking over central tape (5 pts.)
- Placing x1 – placing skystone in foundation (4 pts.)

Algorithmic & Programming Objectives

- Establishing Field Grid & MOEPS (MOE Positioning System)
- Localization
- Multithreading
- Accurate Turning Methods
- Accurate Pathfinding with Intelligent Algorithms
    - Jump Point Search
    - "Rotational Symmetry"
    - Realignment
- Positional Proportional-Integral-Derivative (PID) Control

Sensors Used

Chassis requirements: C28

Foundation detection: C29

Encoders

4: Encoders placed on motors involved with the robot's mecanum drive—front left, front right, back left, and back right.

1: Encoder placed on the lift motor that controls the stone lift, allowing for more accurate and precise vertical control.

Inertial Measurement Unit (IMU)

1: IMU build into the REV Expansion Hub. This IMU is effectively a gyro sensor, with the capability to measure rotation on 3 axes. We primarily use the horizontal axis, or the one that measures rotation parallel to the ground for accuracy in any turns or rotational movement of the robot.

Logitech Webcam

1: Logitech Webcam used in the front of the robot. Using this rather than a phone camera allows for the phone to be safely protected within the robot, making sure that nothing goes wrong during the match. The Logitech Webcam is used for recognizing the Vumarks and aligning with the foundation.

Intel Realsense Tracking Camera T265

1: Intel Realsense Tracking Camera T265 used for localization and as a gyro. Using internal algorithms acting on data provided by two fisheye lenses and one IMU, the camera provides accurate positional and rotational data.

Limit Switch

1: Limit switch placed near the bottom of the lift to allow for hard resetting of the lift. In case the lift slips and encoder ticks are made to be inaccurate, the limit switch acts as a safety.

Odometry Wheels

1: Vertical odometry wheel placed on the side of the robot to allow the robot to continually localize (know its absolute x,y position). An odometry wheel is an unpowered wheel that moves only when the robot moves on the floor. (See more in *Odometry + Gyro Localization*)

1: Horizontal odometry wheel placed on the side of the robot to allow the robot to continually localize (know its absolute x,y position). An odometry wheel is an unpowered wheel that moves only when the robot moves on the floor. (See more in *Odometry + Gyro Localization*)

.

Key Algorithms & Constructs

Field Grid & MOEPS (MOE Positioning System)

PID and stereo-distance algorithms: C54

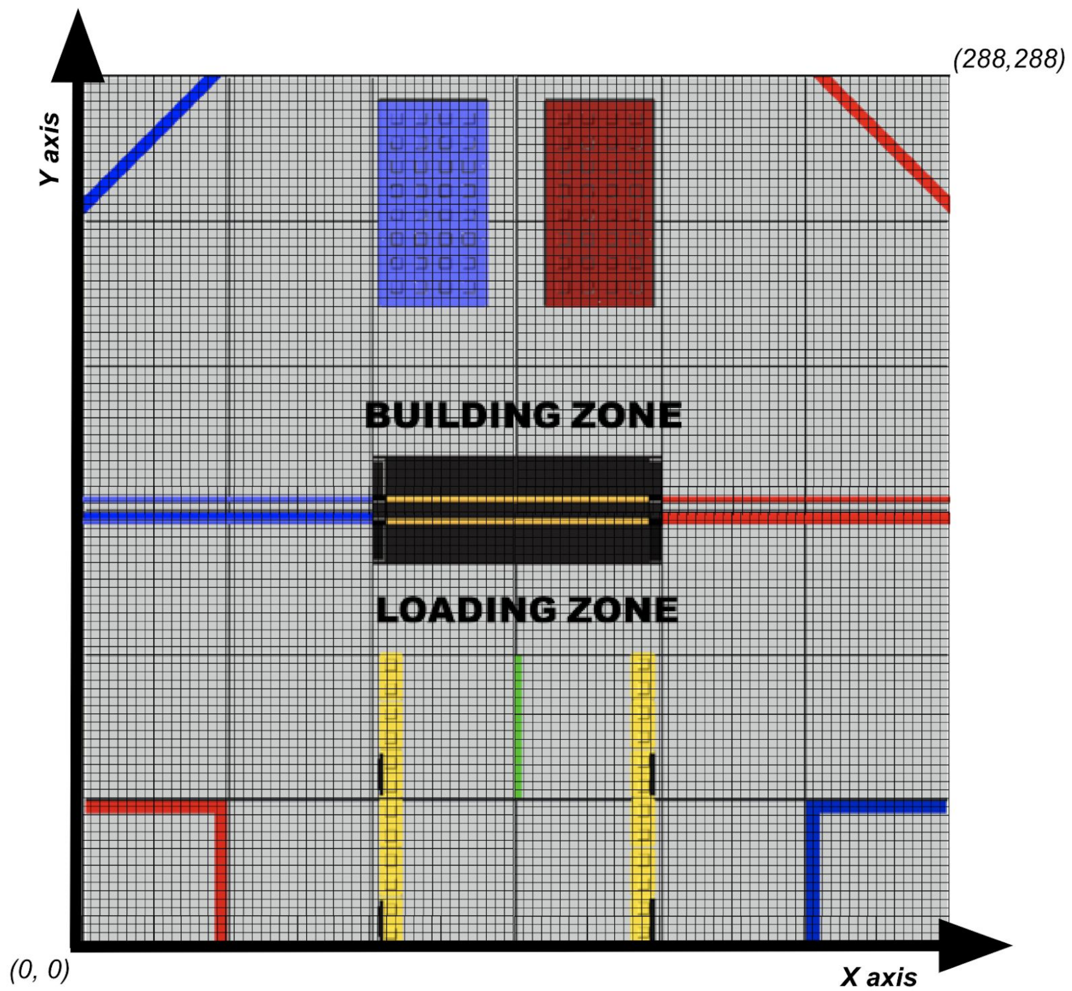
Field coordinate system: C88

Due to the importance of the two positioning systems described below in our programming structures, we have affectionately coined the term MOE Position System, or MOEPS, to describe the systems.

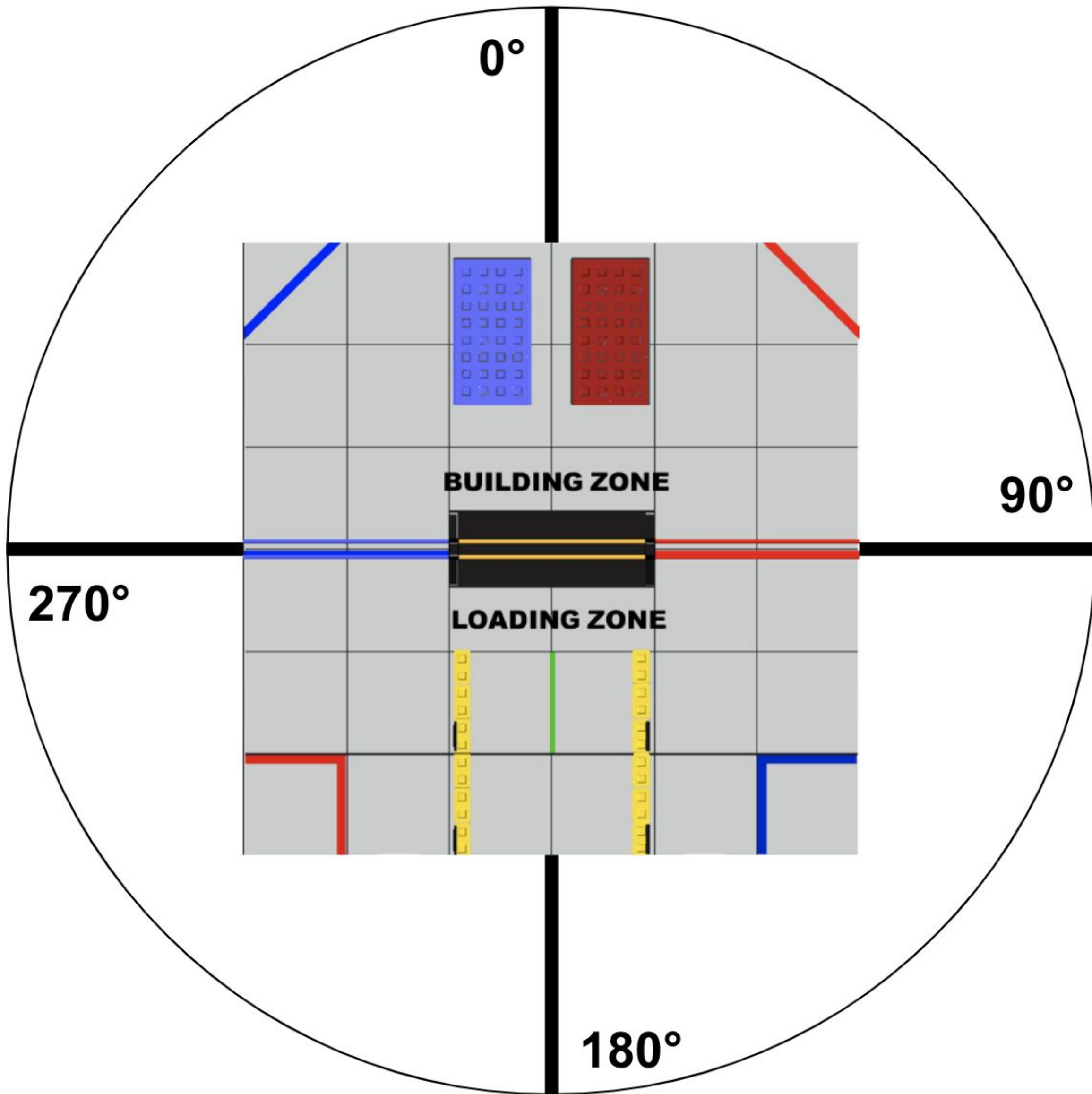
Many of the following approaches and techniques we use rely upon a grid of (x, y) points. To form this grid, we divided up the field into a grid on the Cartesian plane, from (0, 0) to (288, 288).

A real field is 12ft. x 12ft., and we divided up the field into a 288 x 288 grid, where each MOE unit = 0.5 inches. → 12 feet = 144 inches = 288 MOE units.

Additionally, the corner of the red building zone was arbitrarily chosen as (0, 0) of the grid.



Along with a positional (x, y) global map, we wanted to create an orientational global map to establish a consistent angle at any point on the map. The angle map was modeled off of the 2D *Euler Angle* system, which tarts 0 at the top and rotates clockwise for positive change.


0°
90°
270°
180°
BUILDING ZONE
LOADING ZONE

Localization

In terms of our programming team, localization means finding out the robot's exact global (x, y) position on the field. In this case, the robot would have to find out its global (x, y) position on the **MOEPS global field grid** (check *Defining The Field Grid*). To accomplish this, we make use of odometry wheel and an Intel SLAM T265 camera.

Intel Simultaneous Localization and Mapping (SLAM) T265 Camera

PID and stereo-distance algorithms: C54

Camera and position: C77

C++ and camera: C87

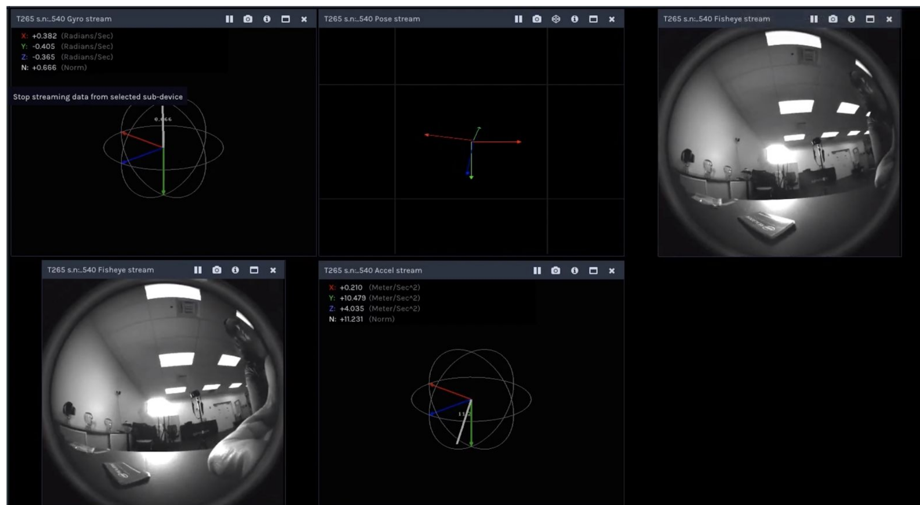
Intel camera and phone connections: C108

Intel testing: C113

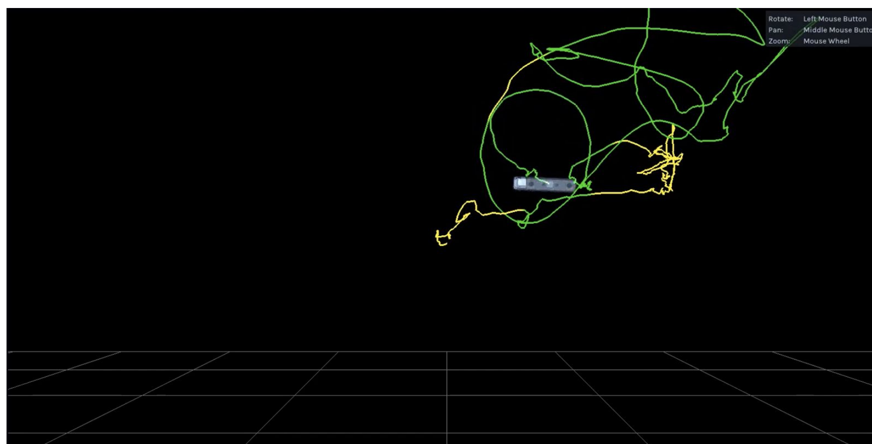
Better quaternion calculations: C118

The Intel Realsense Tracking Camera T265 is a Simultaneous Localization and Mapping (SLAM) camera. SLAM algorithms aim to construct an internal map of their surroundings and use that map to determine its position. Using two black and white fisheye lenses and one internal IMU, the camera has an internal processor with algorithms that perform said SLAM functions.

This, in turn, produces position data on the **x (horizontal), y (vertical), and z (depth-wise)** axes along with **rotational data (orientation)**. Traditionally, the camera has been used for drones and other similar applications, but we have found use for it in our robot.



The above image shows the views from the black and white fisheye lenses of the camera (bottom-left and top-right corners), which is used to determine position.

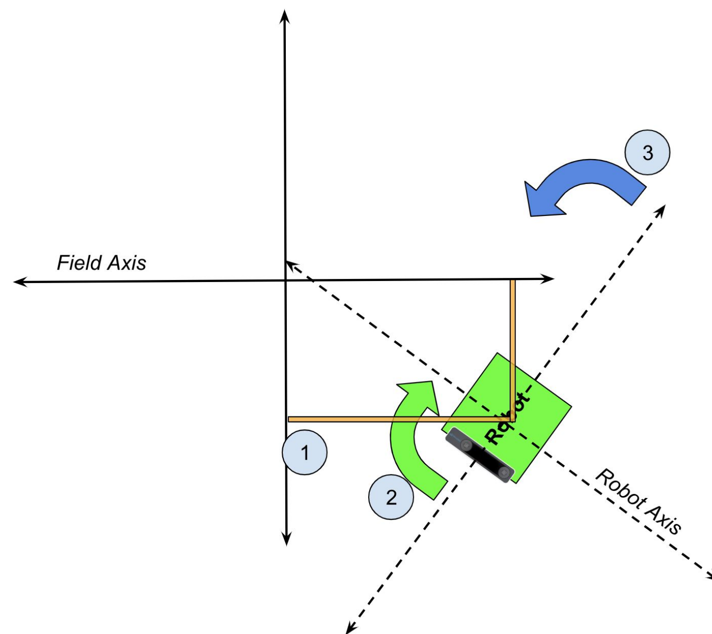


This shows a simplified view of the camera's tracking of position over time. The green and yellow lines show the previous positions of the camera in a 3D field.

In terms of this year's match, three data points from the camera come in handy:

- Horizontal position (x-axis)
- Depth-wise position (z-axis)
- Rotational position (θ)

The vertical position (y-axis) is not needed because our robot **hopefully** does not fly during the match.



Another issue we had to resolve was the conversion of the camera's axis to the field's axis. As shown above, the steps are as follows:

1. Calculate (*x, y*) offsets front field axis to robot (camera) axis
2. Rotate camera axis onto robot axis
3. Rotate robot axis to field axis
4. Add back (*x, y*) offsets to the point

After this correction method, we were able to successfully use the camera to determine the **absolute position** of the robot on the field.

Odometry + Gyro Localization

Servos and odometry pods: C88

Odometry programs: C92

Testing odometry: C101

Problematic odometry pods: C108

Odometry config: C221

Odometry values: C225

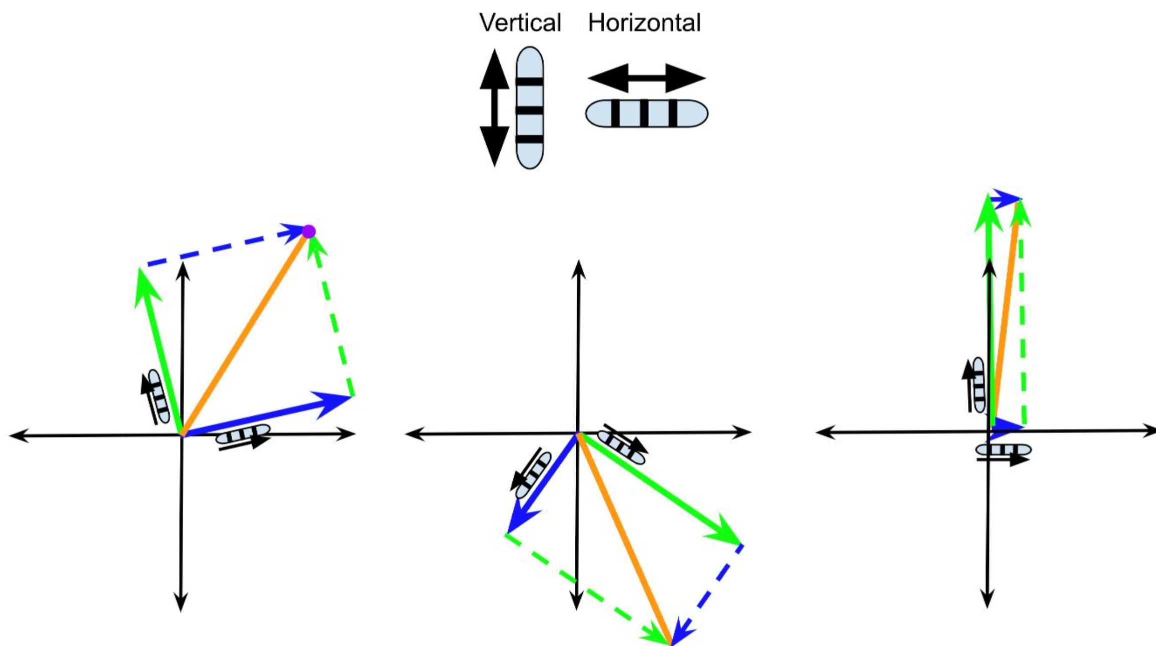
Odometry testing: C227

Although the above SLAM camera is accurate, it is occasionally not perfectly on target.

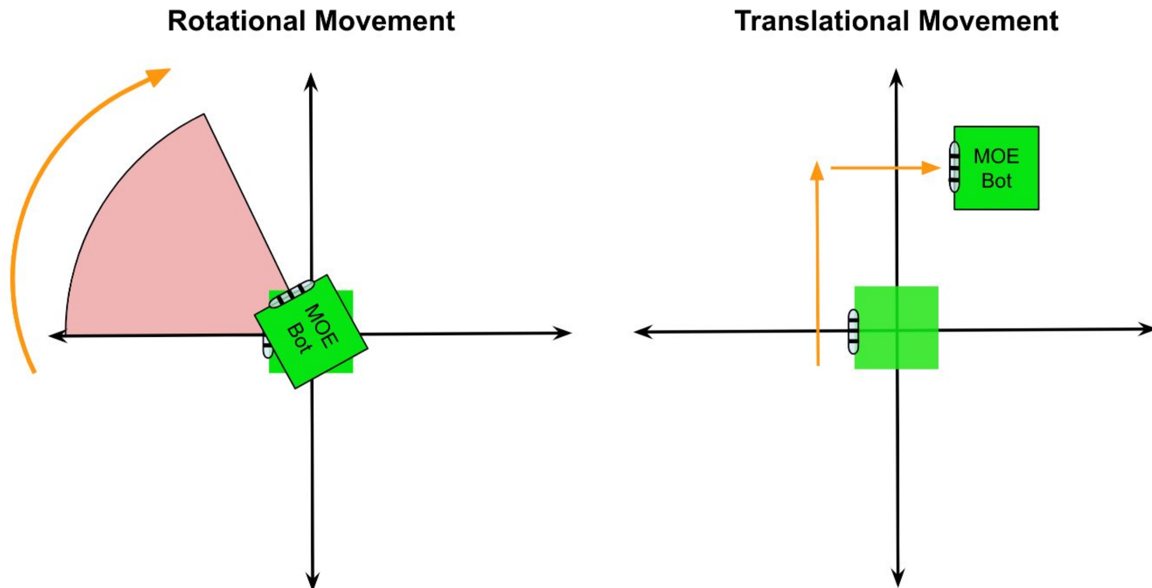
To further increase the accuracy of localization, we constructed odometry wheels, which allow for continual localization at any point on the field.

Odometry wheels are non-powered wheels bound to an encoder that allow for accurate measurement of robot's absolute movement. Since these wheels are not part of the drivetrain, when the robot drives forward into a wall, for example, the drive train motors would register a change in encoder tics while the odometry wheels would remain consistent since they only move with the robot. In our robot, we use 2 REV Through Bore encoders and 2 omni wheels, for a total of 2 odometry wheels.

The benefit of the REV Through Bore encoders, over the MA3 encoders we used last year, is that they are **incremental—they continually add on ticks**, making odometry more robust than before.

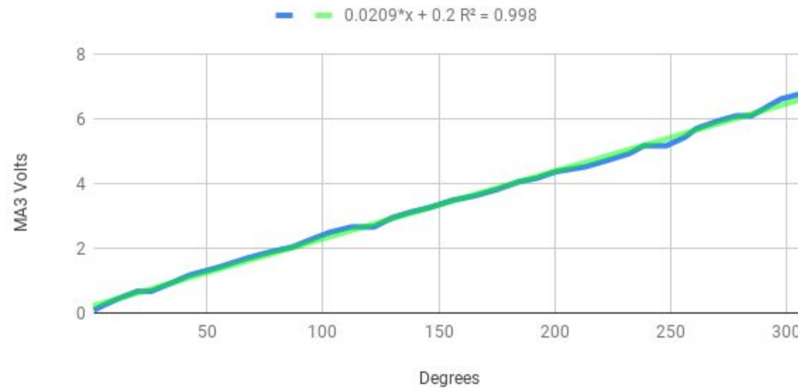


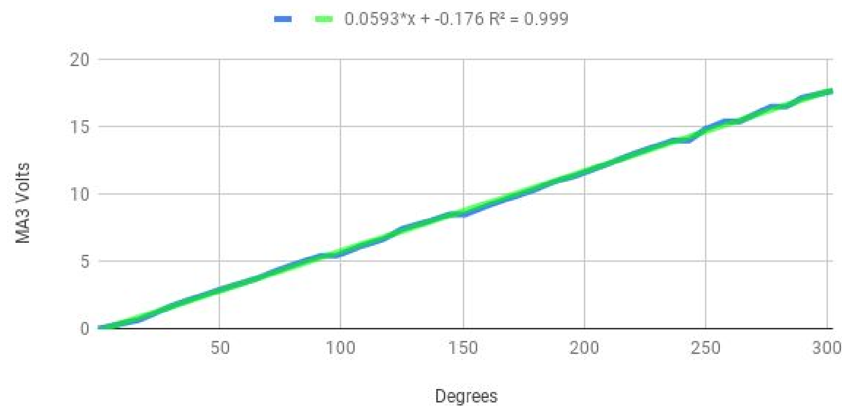
The two odometry wheels on our robot allow for accurate measurement of positional movement. Through the usage of the gyro sensor, we are able to do rotational math to figure out the robot's position at all times.



To use the odometry wheels reliably, we have to be able to distinguish between rotational movement and translational movement. The typical way to solve this issue is through a mechanical paradigm, which involves the use of additional odometry wheels opposite to each on of importance. By averaging the values of both wheels (positive & negative changes would cancel out), one could get an accurate measurement of only translational movement. Applying this methodology to our robot, we would have 4 odometry wheels on the robot. Due to the fact that we did not have space for 4 odometry wheels, our team has to resort to other options to discount rotational movement.

This has been done through the gyro and a measure we created known as “rotational offset”. This “rotational offset” would be used as a subtractor to discount any non-important odometry values. By dividing the angle difference (found between each refresh of the odometry wheels’ position) over 360, and multiplying by the “rotational offset”, an appropriate offset measure can be found for each wheel.





By calibrating & turning a "rotational offset" for each of the two odometry wheels, we can accurately discount any rotational motion, allowing only translational movement to be used in the calculation of the robot's position.

The above graphs show the number of volts that the encoders showed for a rotation of the robot. By using the equations above to estimate the rotational offset for each of the wheels, we can effectively & accurately cancel out rotational motion.

Multithreading

Multithreading is a technique by which a single set of code can be used by several processors at different stages of execution. In other words, a program can have multiple sets of instructions running at the same time. With multithreading, the robot is able to do *more than one task* at any given time. Since our robot is trying to accomplish all standard autonomous points, time is often cut close to 30 seconds.

Without the use of multithreading, the robot's autonomous routine would have to speed up its motors significantly to meet the allotted 30 seconds. This speeding up results in less accuracy, resulting in an autonomous that is more prone to error.

Although processes like Vuforia and TensorFlow may run on separate threads, we intentionally use our own threads or pull from other threads for the following purposes:

- Bringing down the lift mechanism used for dropping/hanging
- Pathing algorithm realignment (see *Vuforia Listener* in *Additional Summary Information* for more details)

We also used **Atomic variables** for thread-safe operation. When a global variable is dealt with between 2 or more threads, there is always the danger of it leaking data when operations on it are done at the same time. Since using a raw variable without synchronization or any other standard is considered bad practice, we decided to use Atomic variables for thread-safe operation. This way, when communicating between the Main Robot thread and the Vuforia thread, we can guarantee that no strange behavior in autonomous occurs because of loss of data between the two threads.

Turning Methods

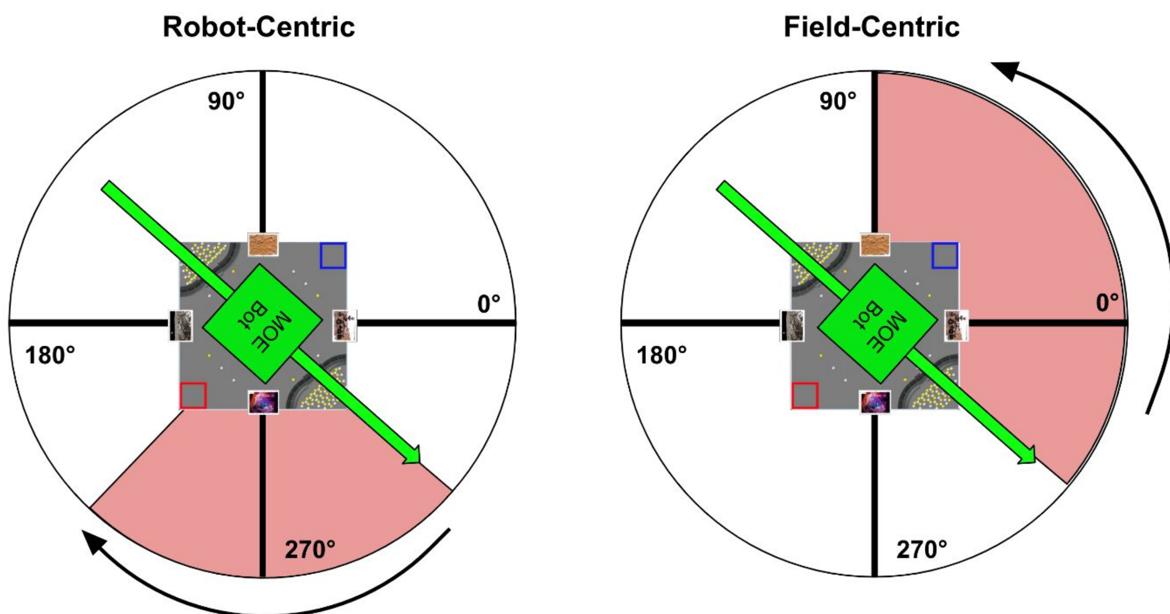
Turning in autonomous must be precise to the degree for repeatable results, which is why turning is dictated by the **IMU** sensor built into the ***REV Expansion Hubs***. Instead of turning by time, we turn by setting the powers the motors, and simply wait for the **IMU** to indicate that we are within the correct angle.

Field-Centric Turning & Robot-Centric Turning

In **field-centric turning,** the **MOEPS angle map** described above (see *Field Grid & MOEPS*), the robot turns to a given global angle on the field.

In **robot-centric turning** the robot turns to a given angle relative to itself.

Robot-centric vs. Field-centric Turning:



As shown in the diagram above, the robot turns 90° directly to the right in the robot-centric turn, while the robot is turning to the 90° mark in the field-centric turn. No matter the orientation, the robot will always turn to the same 90° mark in field-centric turning.

Jump Point Search/A*/Dijkstra's Pathfinding Algorithm

Prototyping pathing: C118

Pathfinding and angles: C126

Generic PID: C132

Slam camera localization: C137

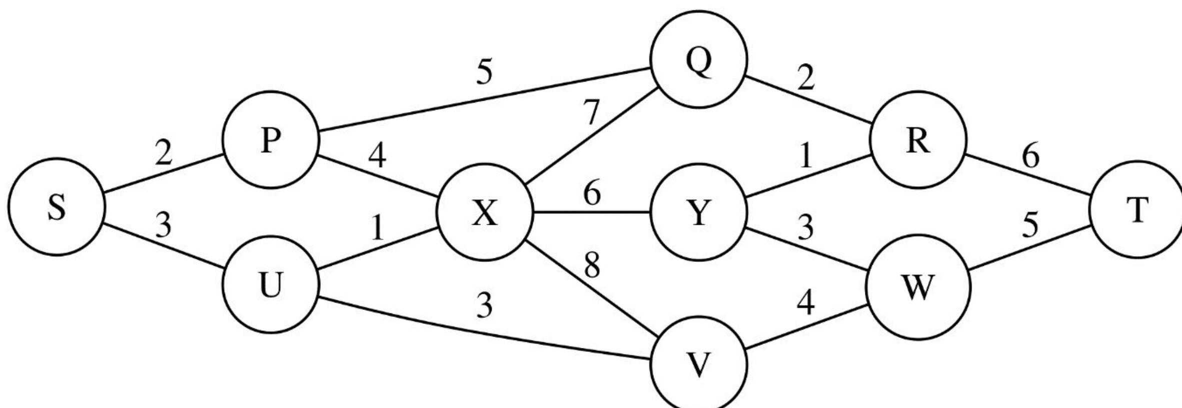
Autonomous points: C152

Introduction

The purpose of the algorithm is to allow the robot to dynamically figure out how to reach its destination. In many autonomous pathings, whenever there is a slight disturbance, the autonomous fails to finish. Rather than explicitly giving the robot a path to follow, the robot is given an end destination. Through localization, the robot figures out its (x, y) coordinate on the field and calculates on its own how to reach the destination.

The *A* (pronounced A Star)* and *Jump Point Search Algorithms* are similar to the popular *Dijkstra's Algorithm*, which is used for finding the shortest paths between nodes in a graph. The primary difference between Dijkstra's and the other two is that the pair utilize a "heuristic function", or an approximation function, to approximate a faster solution to Dijkstra's algorithm. Dijkstra's algorithm checks many more cases than the A* Algorithm, therefore taking longer to arrive at a similar answer. Since the field we are using is 288x288 (82944) nodes, we wanted to guarantee that processing speed would be fast. The algorithms commonly deal with graphs shown like the one below, but had to be specially adapted in our case to work with a 2D grid.

Visual representation of traditional graph in computer science:



To account for processing speed & time, Dijkstra's Algorithm has a worse case time complexity (when using lists) of **$O(N^2)$** where N = number of nodes on the graph, while A* *generally* has a time complexity of **$O(b^d)$,** where b = branching factor and d = depth of the solution on the search tree. However, both of these algorithms have a very slow runtime in certain circumstances, taking over 20 seconds to run. This is unacceptable when run in autonomous, which only has a period of 30 seconds. The Jump Point Search algorithm is an optimized version of the A* pathfinding algorithm that consistently brought our runtime below 2 seconds.

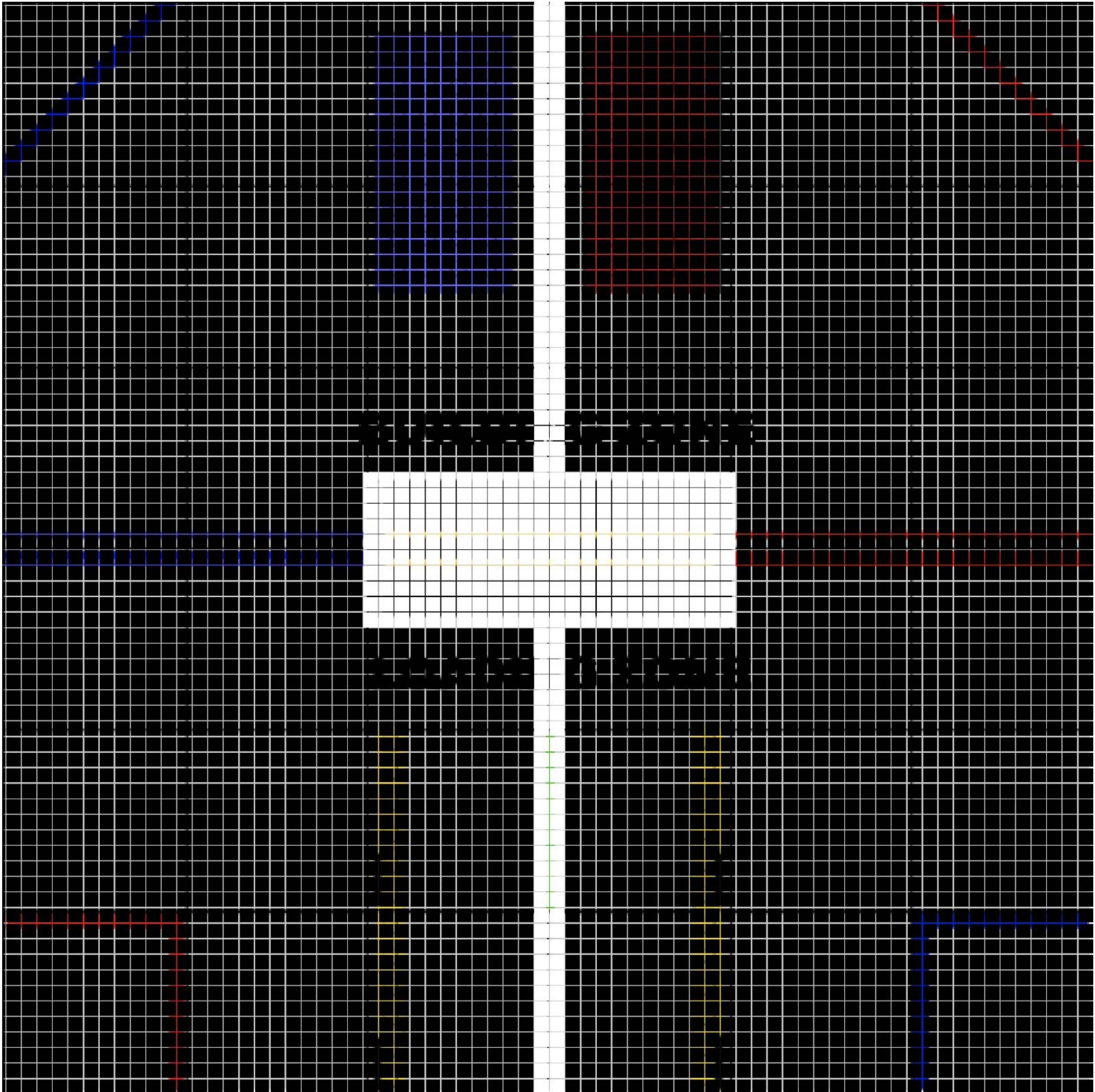
Note that the time complexity of A*/Jump Point Search is worse when using a very expensive heuristic cost function, but we are using the simple Euclidean distance, or the distance formula:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

With the pathfinding algorithms, the robot is able to move in **8 directions** to go from point A to point B. The directions are labelled as follows: **North, Northeast, East, Southeast, South, Southwest, West, and Northwest.**

Setup

To utilize the **pathfinding algorithms**, we needed to first setup a graph. To accomplish this, we took a 2D image of the field from Game Manual 2. After that, we wrote a **Python** script (utilizing the *PIL imaging library*) to go through the image, converting it to points we deemed as barriers (white) and points we deemed as free space the robot could travel on (black). The output was an image with the converted points as well as a 288x288 *array* that we would be able to use as our graph for the pathfinding algorithms.



Mapped Skystone Field: The above image is what the output array looked like visually. The barriers (the black squares) in the image above are represented by 1s, while the free space (the white squares) in the image above are represented by 0s.

Implementation

We implemented the algorithm in Java, making a separate class to handle the calculations. To verify that we wrote the algorithm correctly and be able to make predictions on robot movements, we made a simulation to show the pathfinding algorithm's path from any Point A to Point B visually:

To now use the algorithm in practice, we had to convert the results into a usable format by writing an algorithm to do so.

Path Conversion Algorithm

(Input) - Original Pathfinding Results: A series of points describing each point to go from point A to point B. *For example, getting from (0,0) to (5,5) could be:* (0,0) --> (0,1) --> (1,1) --> (2, 1) --> (3, 1) --> (3, 0) --> (4, 0) --> (4, 1) --> (5, 1) --> (5, 2) --> (5, 3) --> (5, 4) --> (5, 5)	**(Output) - Usable Results:** The number of inches in each direction the robot has to go, in order (each unit is 2 inches). *For example, getting from (0,0) to (5,5) could be:* FORWARD 2 in. --> RIGHT 6 in. --> BACKWARD 2 in. --> RIGHT 2 in. --> FORWARD 2 in. --> RIGHT 2 in. --> FORWARD 8 in.

The robot first turns towards the **0° mark** described in the **MOEPS global angle map (**the direction facing the Crater/Mars VuMark). The results from the pathfinding algorithm would then be translated into movements for the robot based on encoder ticks. The result of this extensive process is a robust and repeatable movement system that allows the robot to figure out its own path when given two points on the field. This simplifies the process of adjusting and programming autonomous, as well as allowing for a more robust and dynamic movement system.

Pathfinding Algorithm Error Correction

The pathfinding algorithm worked perfectly well in theory, but in practice, there were a few issues we had to fix in order of importance.

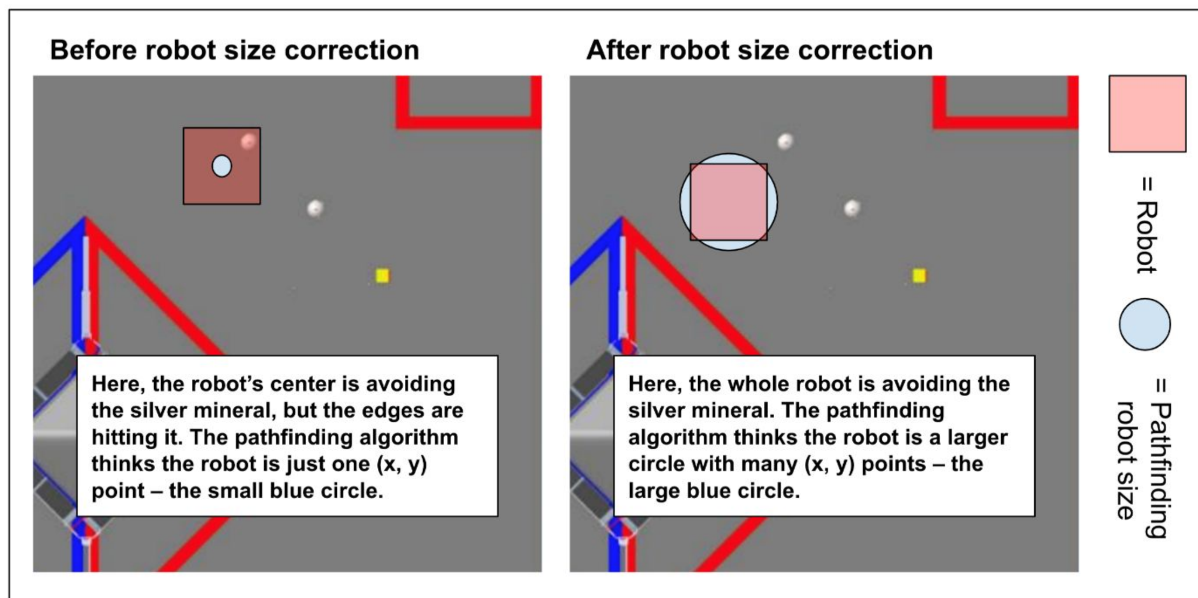
1. The **Jump Point Search algorithm** treated the robot as a single (x, y) point on the **global 288x288 grid** while the robot actually comprised at least a circle of many points with radius about 10 inches. This led to the edges of the robot crashing into parts of the field (lander, crater, sampling, etc...) while its center thought it was following the pathfinding algorithms as a single small point.
2. While moving, the robot would turn slightly off angle. It would be not exactly at its end destination due to slight turning while making up, down, left, and right movements.

Error #1 – Size Corrections

Conceptualization and implementation: C55, C97

Second iteration: C98, C99

To fix this error, we modified the size of the robot in the algorithms. Instead of treating the robot as a single point, we treated it as a collection of multiple points – when put together, these points would form the robot rather than one small point.



Error #2 – Turn Corrections

To fix this error, we took the IMU sensor's horizontal angle before the robot followed the A* algorithm. We then constantly tracked the gyro sensor's angle while the robot followed the A* algorithm.

If the IMU's angle strayed by more than 2°, the robot self-corrected itself back to the correct angle by again utilizing the gyro to turn back into position.

"Rotational Symmetry"

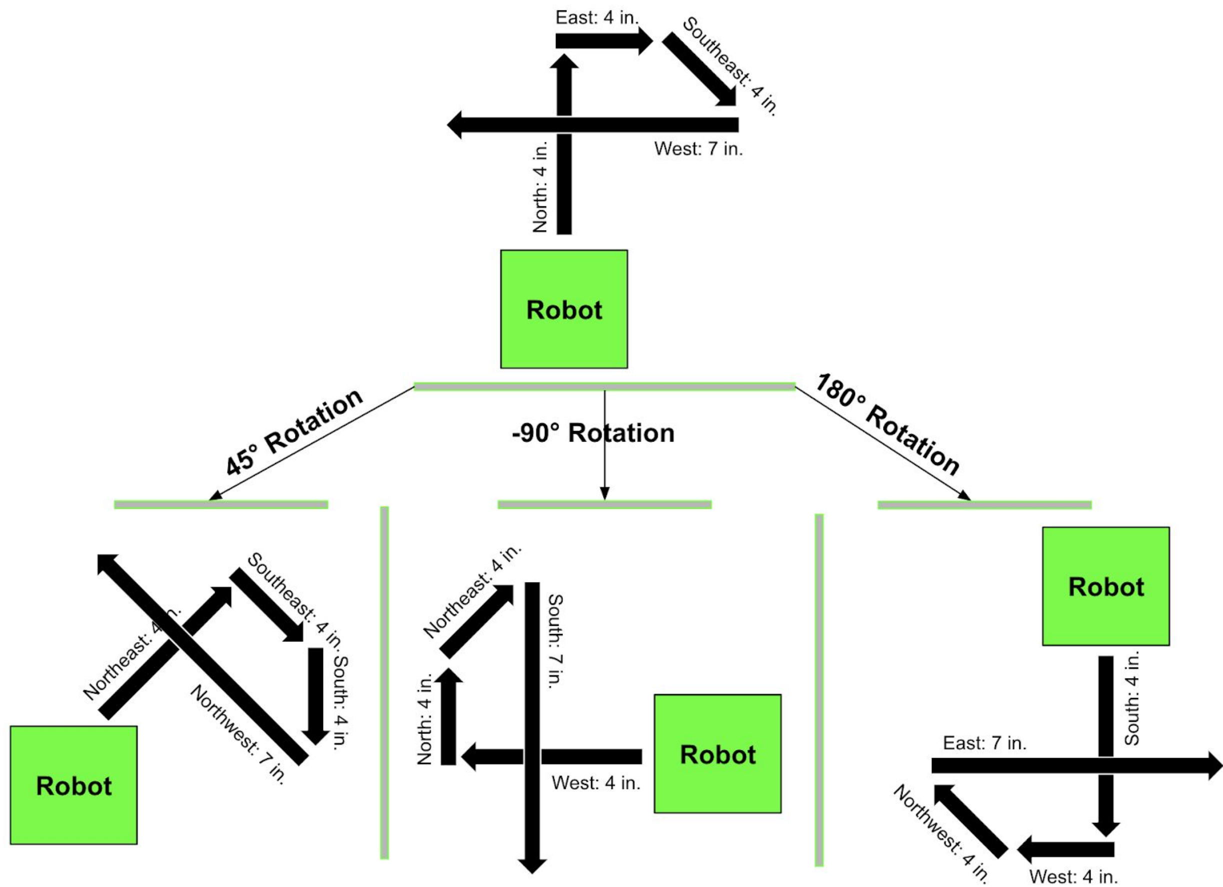
Conceptualization and Implementation: C150, C151

Another feature that we added to the pathfinding algorithm is the idea of "rotational symmetry". In other words, a given set of output instructions can be rotated by certain number of degrees while still preserving the relative directions of each movement.

The purpose of rotational symmetry is to allow for optimizations in accuracy and speed of robot movements. Since moving forwards and backwards is always faster than strafing, rotational symmetry allows the robot to take a pathing from the Pathfinding Algorithms and rotate the pathing instructions. The robot can then quickly turn and apply the rotated

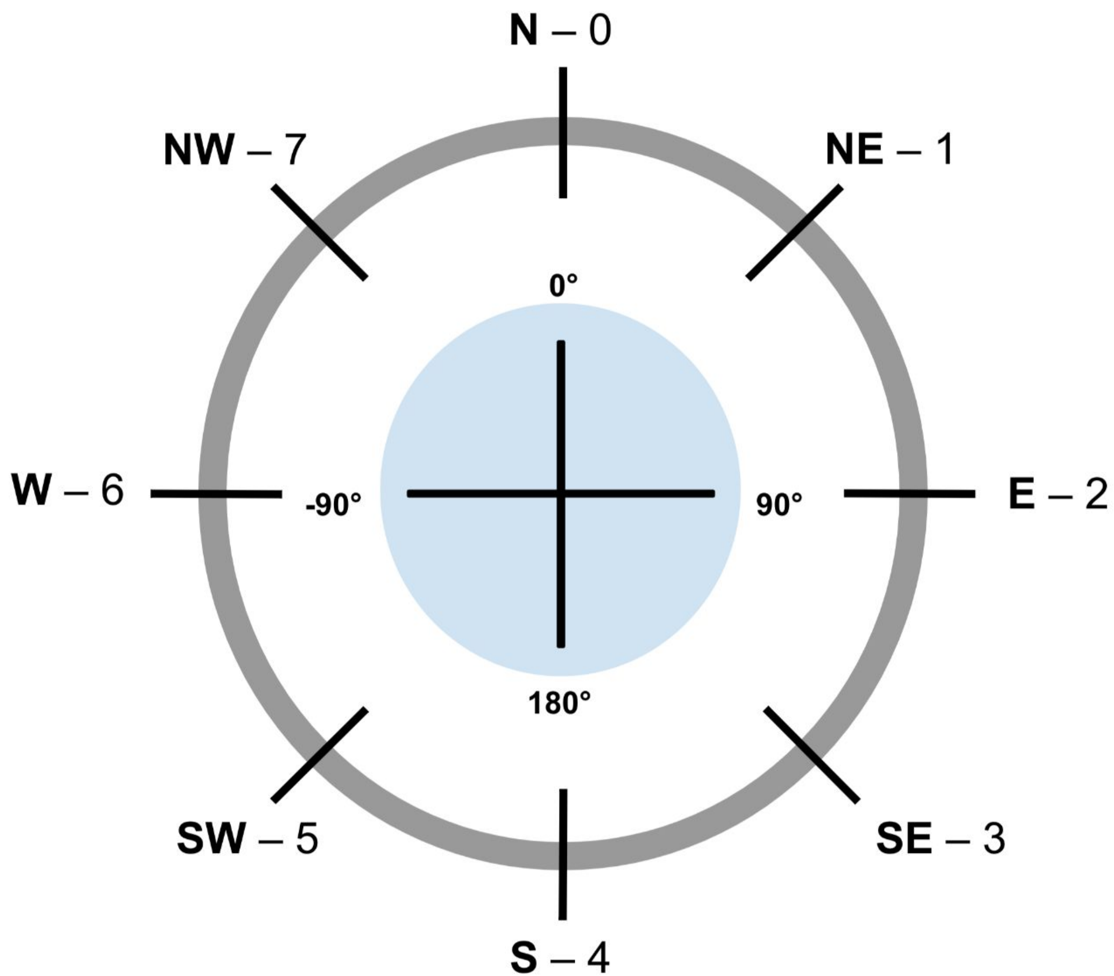
instructions to allow for more forwards and backwards movements, resulting in a more robust movement.

Examples of rotations done on set of output instructions:



The algorithm is accomplished by setting a numerical value to each of the directions in clockwise order. North all the way around to Northwest is numbered from 0 up to 7. This simple numbering pattern makes rotation much simpler than writing each direction's rotation to its right. To rotate a direction, divide the angle needed to rotate by 45° and add it to the number. In the case a value goes above 7, it is wrapped back around to start at 0 (ie: 9 becomes 2).

Visualization of Direction to Number mapping, along with degrees associated with rotations:



This system's simplicity becomes apparent when put into practice. For example, if an instruction says to move the robot North, the direction North's numerical value is 0. To rotate it by 90° clockwise, divide 90° by 45°, which equals 2. The 2 is added to North's numerical value, which results in 0+2 = 2. The 2 corresponds to the East direction, which is exactly a 90° clockwise rotation from North.

Proportional-Integral-Derivative (PID) Control

PID and stereo-distance algorithms: C54

Generic PID: C132

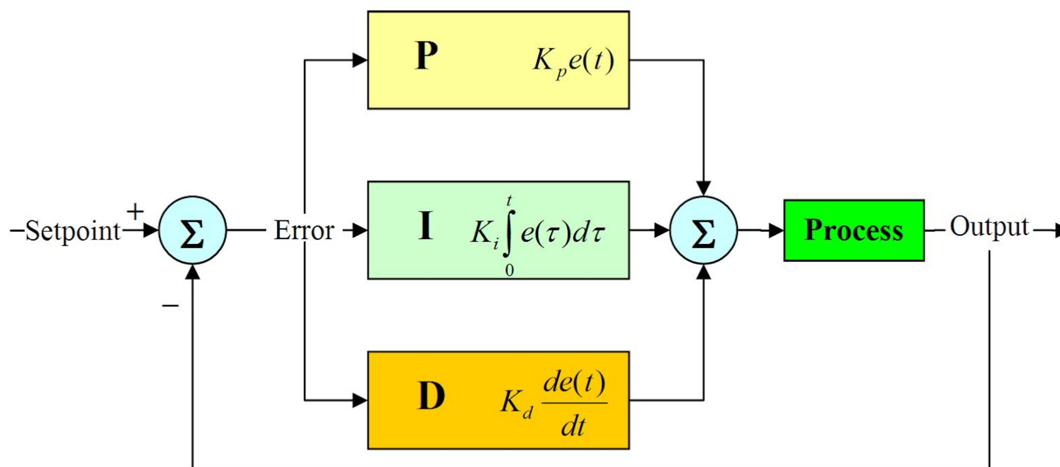
Slide testing: C194

Positional PID: C203, C204

SLAM Initial rotation: C216

Positional PID testing: C220

To achieve more stability and control in our movements, we utilized Proportional-Integral-Derivative (PID) control loops throughout our robot.



Credit: https://i2.wp.com/upload.wikimedia.org/wikipedia/commons/4/40/Pid-feedback-nct-int-correct.png

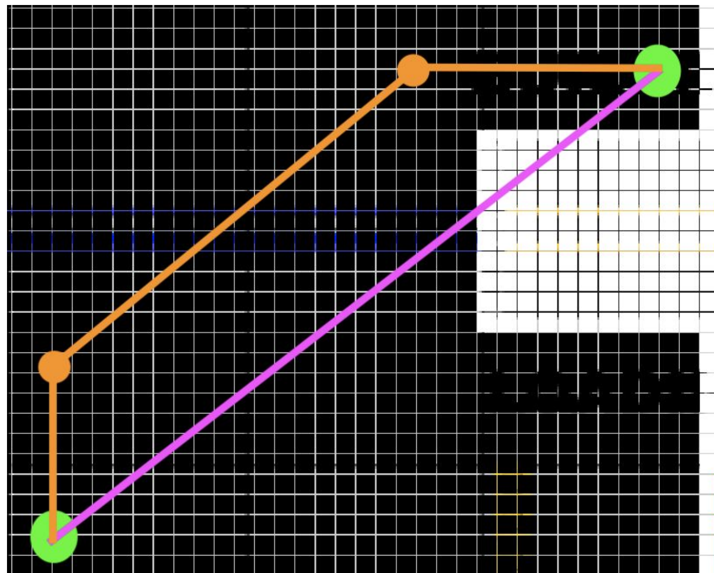
A PID generally works through defining a setpoint, or goal. By measuring the error between the current state and setpoint and summing **proportion (current error), integral (cumulative error), and derivative (change in error)** factors, an output value is reached. This value is subsequently applied to whatever process the PID is meant to be used for. By running this process in a loop, incredibly accurate and reliable motions or actions can be achieved by the robot.

We have PIDs used on our stone lift motor, so we may appropriately optimize the lift's movements while still maintaining accuracy and good ramp-up and slow-down rates.

We have also used PIDs for movement. Instead of PurePursuit, the method used last year that restricted us to only tank drive movement, a positional PID control system allows the robot to take advantage of its full strafing arsenal. The robot, to reach a destination, can rotate and move in any direction. Done successfully, this translates into beautiful, optimized movements where the robot efficiently reaches its destination.

To have a successful positional PID the following are required:

- Horizontal PID with tuned proportion (P), integral (I), and derivative (D) constants with output *h*
- Vertical PID with tuned proportion (P), integral (I), and derivative (D) constants with output *v*
- Rotational PID with tuned proportion (P), integral (I), and derivative (D) constants with output *r*
- Conversion method of changing *h, v,* and *r* into strafe (STR), forward (FWD), and rotation (ROT) values—this is done with field-centric code
- Pathfinding (done with Jump Point Search) and point injection



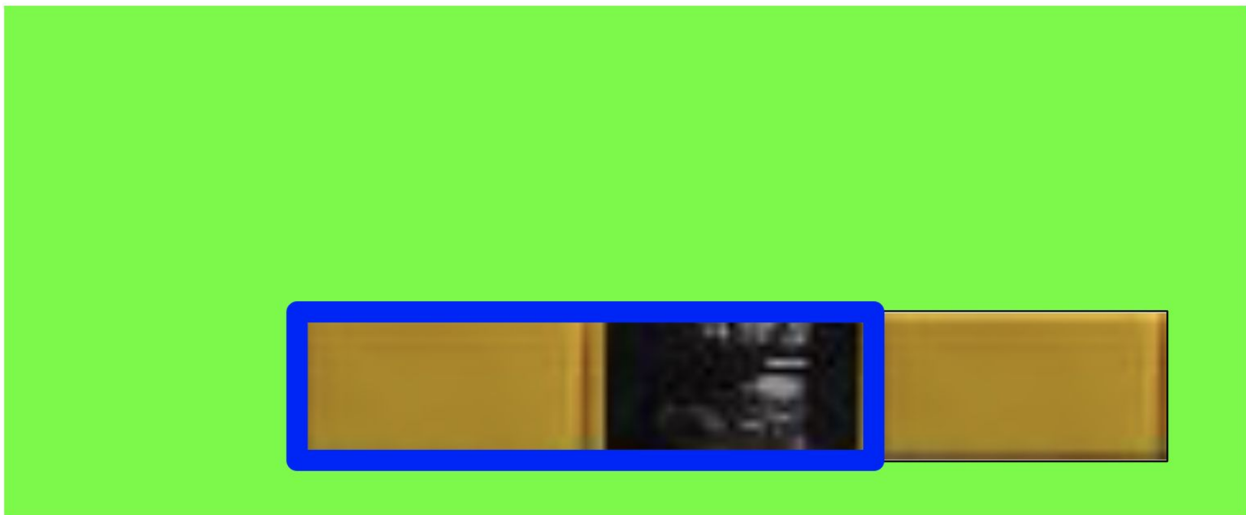
However, one downside of the positional PID is that it cannot inherently perform obstacle avoidance. For example, take the situation presented in the above image, where the starting and ending points are represented in green. Directly applying a PID would result in a crash, since the robot would traverse the purple path, hitting the white box. On the other hand, by using pathfinding (Jump Point Search) to find a path avoiding the white and injecting points along the way, the issue is resolved. The orange path in the image above shows one such path. Incrementally applying the positional PID to each set of two points in the path would allow the robot to reach its destination quickly and obstacle-free.

Skystone Detection

Skystone Detection: C38

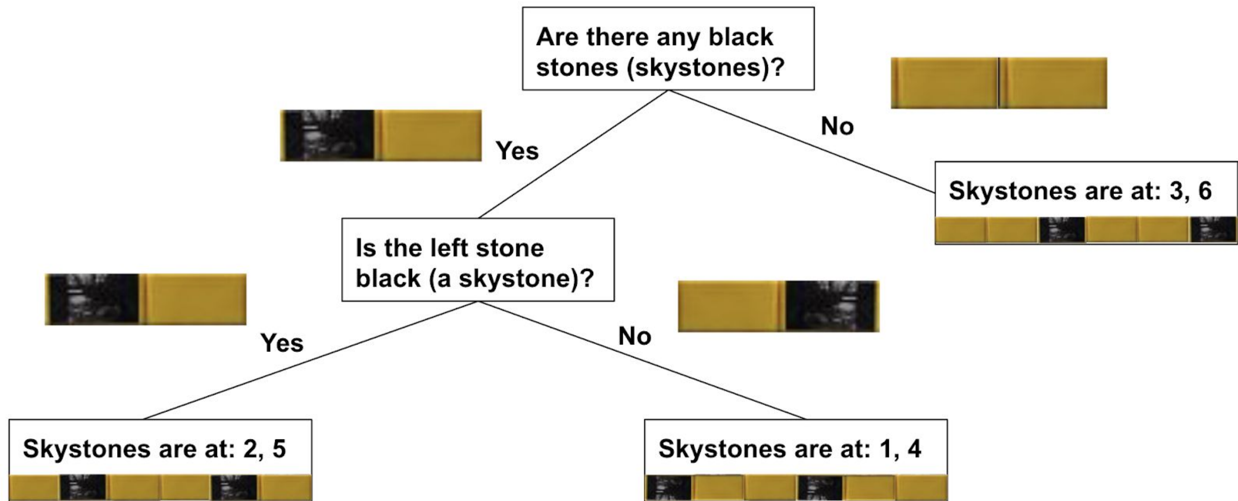
Pixel filter: C38

For the Skystone detection algorithm, we decided to use a Logitech Webcam instead of a color sensor, a commonly used approach. The color sensor is highly dependent upon external lighting conditions, and to combat this issue, significant mechanical effort and positioning is required. However, by using a Webcam, we essentially gather many data points that allow us to make better decisions; we gather a view of the whole field from a distance.



In the above image, the green represents the external noise in the camera position when autonomous is initialized. By cropping the above image to within the blue border (shown below), all the positions of the skystones can be determined.

Given the image above, we are able to write a relatively simple algorithm pixel comparison algorithm knowing that the image is always the left two skystones.



Using this information, we can use the positional PID to travel to the appropriate one, allowing us to have a much more robust autonomous.

Driver Controlled Enhancements

Adjustable Field-Centric Movement

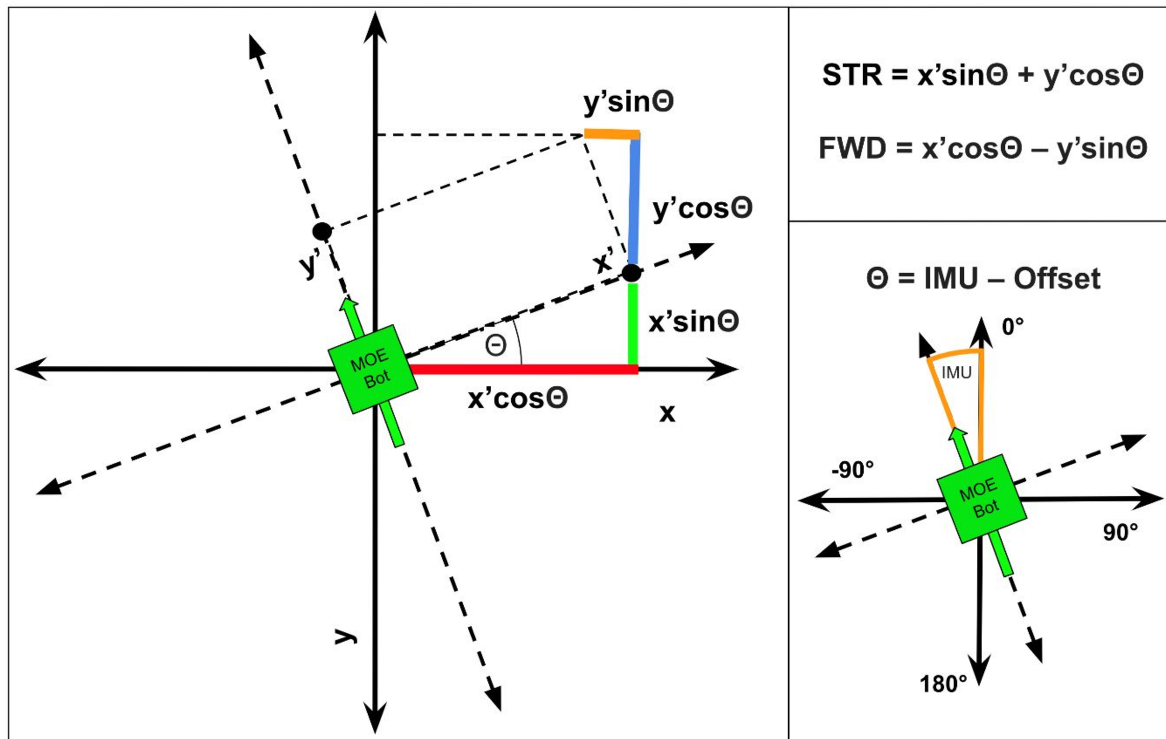
Conceptualization and implementation: C130, C131, C141

Mecanum calibration: C148

Drivetrain evaluation: C170

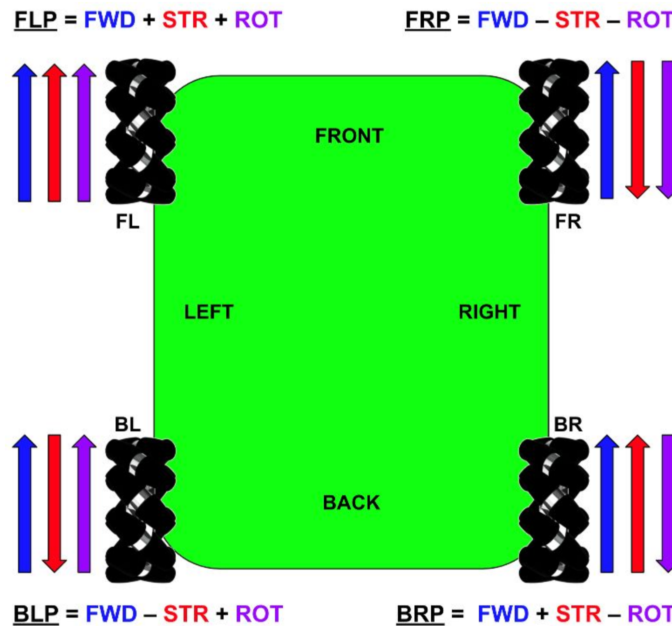
Since our robot uses a mecanum drive, it is capable of moving in any direction. Because of this, we are able to use **field-centric** motion rather than **robot-centric** motion. For the ease of the driver, we make all movements relative to the field rather than relative to the robot.

Note: In this diagram x' = Left Joystick X AND y' = Left Joystick Y



This diagram represents the **rotation of axes** that underlies the principles in field centric movement. The Θ is taken from the IMU, so that angle measurements are exact in the **field-centric** motion. :Inputs for x' and y' are taken from the left joystick, and the right joystick x controls the rotation of the robot. Also, the offset for the IMU is to allow the driver to set a custom 0° point for the robot.

In the below diagram, the arrows indicate which direction the wheel turns when given a positive value for FWD (forward), STR (strafe), or ROT (rotation).



The corresponding values of FLP, FRP, BLP, and BRP are fed into the motors based on controller input so that field-centric movement occurs.

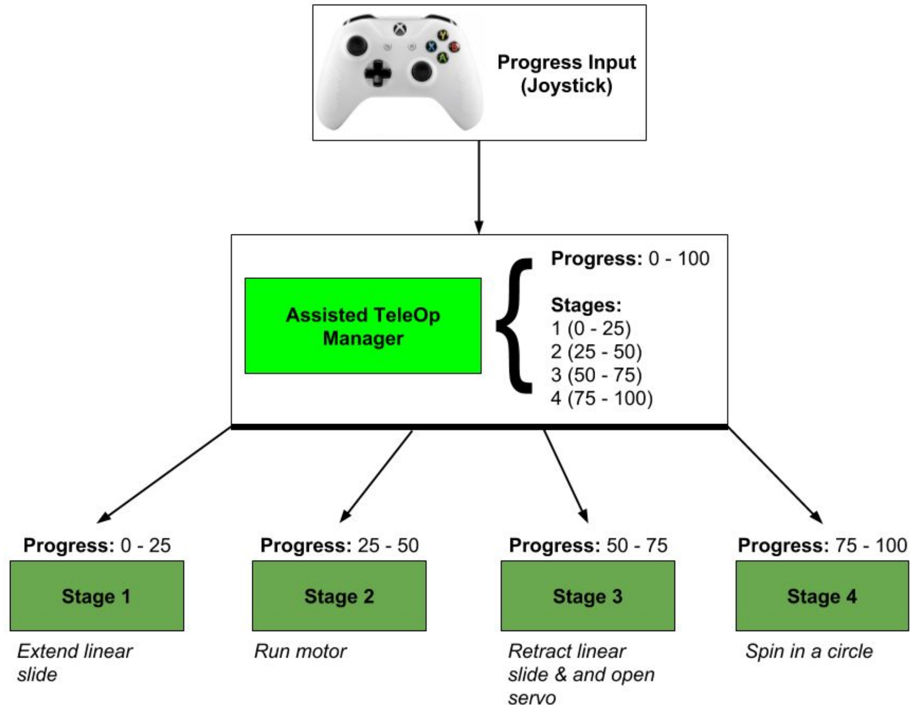
Assisted TeleOp

Teleop controls, gamepad manager: C156

Our robot uses "**Assisted TeleOp,**" in which the robot performs a series of actions in one press of a button.

To make our Assisted TeleOp functions adaptable, we created an AssistedTeleOpManager class that takes in an AssistedConfig. This allows us to simply write a configuration for an automated task, immediately and quickly integrating it into our code.

Each of our Assisted TeleOp configs uses a progress variable that can be increased or decreased to control which stage of action it belongs to. This brings flexibility into the AssistedTeleOp, since the progress variable could be automatically increased ***or*** be controlled by a single joystick. This heavily simplifies complex motions that would otherwise require many controller inputs.



Looking at the diagram above, the manager abstracts much of the logic of progressing through an Assisted TeleOp routine. By using the manager, moving forwards and backwards with an assisted routine becomes very simplified. All the programmer that uses the manager has to do is set the *progress* variable that controls the motion of the robot.

This infrastructure also rapidly speeds up development of Assisted TeleOp routines, since the programmer only has to focus on the logic of the Assisted TeleOp, not the mundane transitions between stages of Assisted TeleOp.

For example, our transfer mechanism, which sends the minerals from the harvester to the dispenser (and involves complex movements), can use this infrastructure to manage progress through stages.

Controls

Teleop controls: C132

Gamepad 1

Left Joystick: Move Robot

Right Joystick *Left, Right*: Turn robot

B: Run intake inwards (harvest stone)

Y: Reset 0° point (forward heading) for field-centric movement

Left Trigger: Make robot go faster

Right Trigger: Run intake outwards (dispense stone)

Left Bumper: Enable Foundation Servo

Gamepad 2

Left Joystick Up, Down: Move lift up, down

Right Stick Up, Down: Move lift up, down slowly

B: Toggles grab servo

Left Bumper: Move outtake servo out

Right Bumper: Move outtake servo in

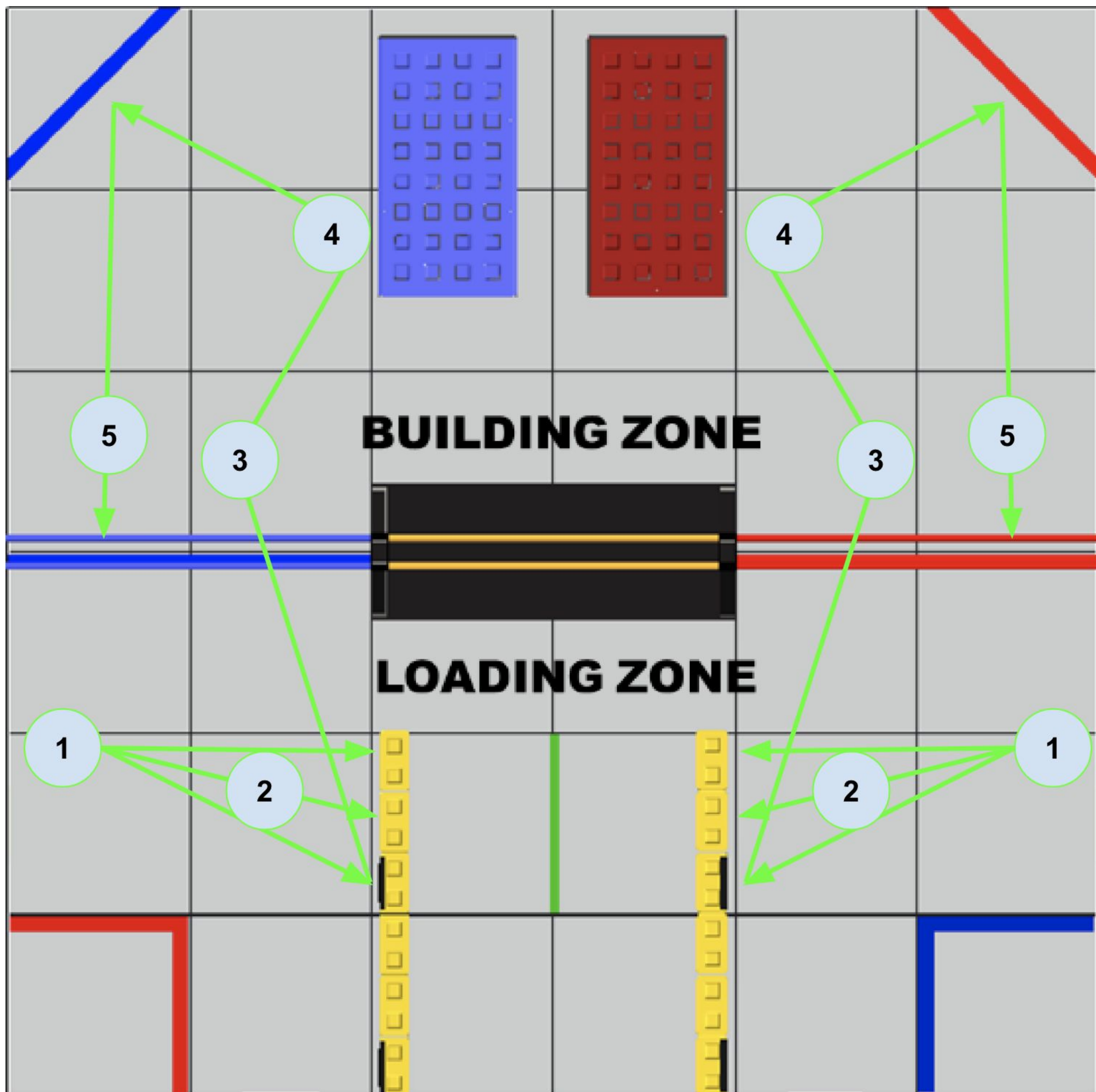
Autonomous Routines

General autonomous pathing: C11

Tweak autonomous code: C207

Tweak SLAM camera: C211

Primary Autonomous Pathing (29 pts.)



The image above represents our primary autonomous pathing, which assumes that the other robot will not interfere with our pathing. In other words, the other robot will stay still.

1. Detect skystone
2. Travel to quarry and collect skystone
3. Travel to foundation
4. Dispense skystone and grab foundation
5. Drag foundation to building zone and travel to parking zone

Initialization

1. Initialize motors, servos, sensors, and all other devices from the Hardware Map.
2. Initialize REV IMU sensor
3. Initialize Intel SLAM T265 camera
4. Initialize Vuforia
5. Set all servos to appropriate starting to positions

Steps

1. Detect skystone position

1. Use skystone detection algorithm on Webcam input to determine the left, center, or right position of the leftmost skystone relative to the robot

2. Travel to quarry and collect skystone

1. Path-find and use positional PID to travel to skystone's position in the quarry
2. Orient robot with left side parallel to skystones
3. Strafe left until harvester is aligned with skystone, using *x* position to determine alignment
4. Drive forward until skystone is collected, using *y* position to determine alignment

3. Travel to foundation (10 pts.)

1. Path-find and use positional PID to travel to foundation – the skystone has now crosse the parking zone

4. Dispense skystone and grab foundation (4 pts.)

1. Align front of robot parallel to the foundation’s left side
2. Dispense skystone
3. Grab foundation

5. Drag foundation to building zone and travel to parking zone (15 pts.)

1. Path-find and use turn-restricted positional PID (don’t want to swing around the foundation!) to travel to building zone, dragging the foundation along
2. Release the foundation
3. Path-find and use positional PID to travel to parking zone, using Webcam to avoid other robot and determine an open position

Alternative Autonomous Pathings

Foundation Grabbing (15 pts.): In this autonomous, the robot grabs the foundation, dragging it back to the building zone and releasing it. Subsequently, the robot travels to the parking zone.

Parking (5 pts.): In this autonomous, the robot simply parks immediately. This is used in the case of calibration, sensor, or alliance conflict issues.

Additional Summary Information

Text-To-Speech (TTS)

For additional fun and utility, we incorporated the *Google Text-to-Speech* technology that allows text to be read aloud in a human-like fashion.

On the field, our robot likes to let us know how it is doing through a variety of phrases, including when it is initialized. Certain phrases include:

- “Initialized Vuforia”
- “Initialization Complete”
- “Initialized Gyro”

Over time, hearing these phrases can become cumbersome; however, our robot likes to spice things up. When completing certain tasks in autonomous, the robot likes to show its patriotism and loyalty to our team. Certain phrases it uses include:

- “Go MOE”
- “Whooo!”
- “Hi _____” (where _____ may be someone’s name)
    - Note: this is pre-programmed, we have not yet integrated the facial recognition technology for the robot to detect people on its own

Our robot also has a fondness for music, which it may play on or off the field. More than anything else, our robot’s personality makes interacting with it more interesting and fun! :)