Robotics Simplified

Generated 28 March 2019.

Contents

1 Preface **3**

1.1 Math . . . 3

1.2 Programming . . . 3

1.3 Libraries and Classes . . . 3

1.4 About comments . . . 3

1.5 Running the code . . . 4

2 Drivetrain Control **4**

2.1 Tank drive . . . 5

2.1.1 Implementation . . . 5

2.1.2 Examples . . . 5

2.1.3 Closing remarks . . . 6

2.2 Arcade drive . . . 6

2.2.1 Deriving the equations . . . 6

2.2.2 Implementation . . . 7

2.2.3 Examples . . . 8

2.2.4 Closing remarks . . . 8

3 Motor controllers **8**

3.1 Sample Controller Class . . . 9

3.2 Dead reckoning . . . 9

3.2.1 Implementation . . . 10

3.2.2 Example . . . 11

3.2.3 Closing remarks . . . 11

3.3 Bang–bang . . . 11

3.3.1 Implementation . . . 12
3.3.2 Examples . . . 12
3.3.3 Closing remarks . . . 13
3.4 PID . . . 13
3.4.1 Deriving the equations . . . 14
3.4.1.1 Proportional . . . 14
3.4.1.2 Integral . . . 15
3.4.1.3 Derivative . . . 15
3.4.1.4 Result . . . 16
3.4.2 Implementation . . . 16
3.4.3 Tuning the controller . . . 17
3.4.4 Examples . . . 18
3.4.4.1 Driving a distance . . . 18
3.4.4.2 Auto-correct heading . . . 18
3.4.5 Closing remarks . . . 19
3.5 Polynomial Function . . . 19
3.5.1 Horner's method . . . 20
3.5.2 Implementation . . . 20
3.5.3 Examples . . . 21
3.5.3.1 Driving a distance . . . 21
3.5.4 Generating a polynomial . . . 21
3.5.5 Closing remarks . . . 21

4 Odometry **22**

4.1 Heading from Encoders . . . 22
4.1.1 Calculate heading without gyro . . . 22
4.2 Line Approximation . . . 24

4.2.1 Deriving the equations . . . 24
4.2.2 Implementation . . . 25
4.3 Circle Approximation . . . 26
4.3.1 Deriving the equations . . . 26
4.3.1.1 Calculating R . . . 26
4.3.1.2 Rotating (x_0, y_0) around ICC . . . 27
4.3.1.3 Edge cases . . . 28
4.3.2 Implementation . . . 28

5 Resources **29**

5.1 Math . . . 29
5.2 Images . . . 29
5.3 p5.js . . . 30
5.4 VEX EDR . . . 30
5.5 Autonomous motion control . . . 30
5.6 PID . . . 30
5.7 Drivetrain Control . . . 30
5.8 Circle Approximation . . . 30

6 About **30**

6.1 Motivation . . . 30
6.2 Contributions . . . 31
6.3 Acknowledgements . . . 31

1 Preface

Although you don't need to know any robotics before reading through this website, there are some things that can make your learning experience much more pleasant.

1.1 Math

To understand the math on this website, it is best to understand basic algebra, trigonometry, geometry. None of the math should exceed high-school level and if it does, there will be links to resources to explain them. **Even if math weren't your strong suit and you skipped all of the equations, you'd still learn a lot!**

1.2 Programming

Understand basic concepts of programming will definitely come in handy before reading through the project. There are lots of great resources for learning programming:

- Reddit's r/learnprogramming forum.
- Codeacademy and their Learn Python 3 course.
- Project Euler for practicing programming on fun math-based problems.

Knowing syntax of Python would also be helpful, since all of the code examples discussed in the project are written in Python. If you don't know much about Python, but already know how to program in a different language, Dive into Python 3 is a great place to start.

If you don't know anything about programming but are interested in learning robotics anyway, you can still read through the chapters – the code is included mainly to showcase possible implementations of the discussed concepts.

1.3 Libraries and Classes

Throughout the project, there will be a lot of made-up classes like `Motor`, `Joystick` and `Gyro`. They are only used as placeholders for real classes that you would (likely) have, if you were implementing some of the concepts covered on this website on a platform of your choosing.

1.4 About comments

There are a LOT of comments in the examples of code that you are about to see. Way, way more than there should be. I agree that this is considered bad practice for practical purposes and that if you are a relative beginner, you shouldn't write code in a similar fashion.

However, since the purpose of the code on this website is educational and not to serve as a part of a codebase, I would argue that it is fine to use them to such extent.

1.5 Running the code

All of the code on this website has been tested on a VEX EDR robot programmed in Python using RobotMesh. If you want to try out the code yourself, doing the same would be the easiest way – all you'd have to do is substitute the made-up classes and methods for real ones from the vex library and run the code.

2 Drivetrain Control

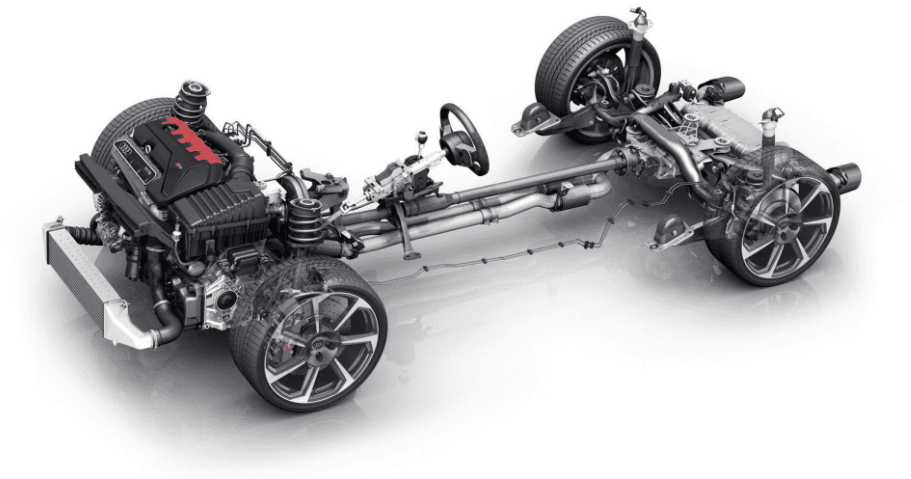

Figure 1: Drivetrain

The drivetrain of a vehicle is a group of components that deliver power to the driving wheels, hold them together and allow them to move. Some of the most popular ones are:

- **tank drive** – left and right side motors are driven independently – like a *tank*
- **mecanum drive** – similar to tank drive, but uses mecanum wheels, each of which is driven independently – allows for more maneuverability
- **swerve drive** – each wheel can rotate vertically around its axis – the robot can drive and rotate in any direction

For the purpose of this guide, however, we will only be discussing some of the most frequently used drivetrains, and the methods to operate them.

2.1 Tank drive

Tank drive is a method of controlling the motors of a *tank drive drivetrain* using two axes of a controller, where each of the axes operate motors on one side of the robot (see image below, or this video).

Figure 2: Tank Drive

2.1.1 Implementation

Suppose that we have objects of the `Motor` class that set the speed of the motors that take values from -1 to 1. We also have a `Joystick` object that returns the values of the y_1 and y_2 axes.

Implementing tank drive is quite straightforward: set the left motor to the y_1 axis value, and the right motor to the y_2 axis value:

```
def tank_drive(l_motor_speed, r_motor_speed, left_motor, right_motor):
    """Drives the robot using tank drive."""
    left_motor(l_motor_speed)
    right_motor(r_motor_speed)
```

2.1.2 Examples

The following example demonstrates, how to make the robot drive using tank drive controlled by a joystick.

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)
joystick = Joystick()

# repeatedly set motors to the values of the axes
```

```
while True:
    # get axis values
    y1 = joystick.get_y1()
    y2 = joystick.get_y2()

    # drive the robot using tank drive
    tank_drive(y1, y2, left_motor, right_motor)
```

2.1.3 Closing remarks

Tank drive is a very basic and easy way to control the robot. When it comes to FRC, it is a method used frequently for its simplicity, and because it is easier for some drivers to control the robot this way, compared to the other discussed methods.

2.2 Arcade drive

Arcade drive is a method of controlling the motors of a *tank drive drivetrain* using two axes of a controller, where one of the axes controls the speed of the robot, and the other the steering of the robot.

Figure 3: Arcade Drive

2.2.1 Deriving the equations

The equations used in the implementation are derived using linear interpolation. This allows us to transition from one coordinate to the other in a linear fashion, which provides a pleasant driving experience.

I won't show the derivation here, since it isn't too exciting (if you're interested, it can be found in this post on Chief Delphi), but rather an illustration to help you understand the derivation more intuitively.

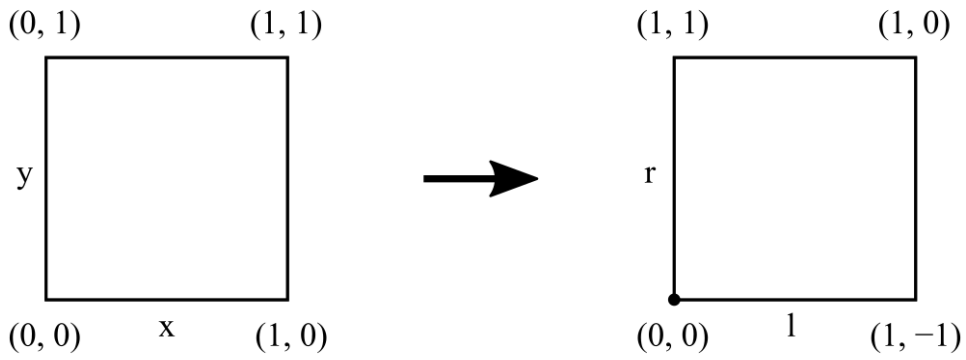

Figure 4: Arcade Drive Illustration

What we want is to transform our x and y coordinates to the speeds of the motor. To do this, let's focus on the values we would *like* to be getting in the 1st quadrant. The left rectangle in the illustration represents the joystick values (in the 1st quadrant) and right one the speeds of the left and right motor (in the 1st quadrant). I would suggest you see the visualization at the end of the article, it demonstrates this nicely.

The speeds of the left motor l are 1 in all of the corners (besides the origin). Using x, y, we can achieve the same values by saying that l is the bigger of the two... $l = \max(x, y)$.

As for the right motor r, As the x and y change their values, r goes from 1 (top left) to -1 (bottom right). We can model the same values using difference... $r = y - x$.

A similar observation can be made about the other quadrants. This is mostly to show that you can think about the derivations more intuitively (since this particular one it isn't too complicated).

2.2.2 Implementation

```
1  def arcade_drive(rotate, drive, left_motor, right_motor):
2      """Drives the robot using arcade drive."""
3      # variables to determine the quadrants
4      maximum = max(abs(drive), abs(rotate))
5      total, difference = drive + rotate, drive - rotate
6
7      # set speed according to the quadrant that the values are in
8      if drive >= 0:
9          if rotate >= 0:  # I quadrant
10             left_motor(maximum)
11             right_motor(difference)
12         else:            # II quadrant
13             left_motor(total)
14             right_motor(maximum)
15     else:
16         if rotate >= 0:  # IV quadrant
17             left_motor(total)
```

```
            right_motor(-maximum)
        else:             # III quadrant
            left_motor(-maximum)
            right_motor(difference)
```

2.2.3 Examples

The following example demonstrates, how to make the robot drive using arcade drive controlled by a joystick.

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)
joystick = Joystick()

# repeatedly set motors to the values of the axes
while True:
    # get axis values
    x = joystick.get_y1()
    y = joystick.get_y2()

    # drive the robot using adcade drive
    arcade_drive(x, y, left_motor, right_motor)
```

2.2.4 Closing remarks

If you are interested in reading more about this topic, I would suggest looking at this thread on Chief Delphi, where I learned most of the information about the theory behind arcade drive.

3 Motor controllers

It's nice to be able to drive the robot around using a joystick, but it would sometimes be more useful if the robot could drive **autonomously** (independently). This is where controllers come in.

A controller is a box that takes in information about the robot and a goal that we want the robot to achieve (like drive a certain distance / turn a certain angle) and, when asked, returns values it thinks the robot should set its motors to, to achieve that goal.

The are lots of various controllers to choose from that differ in many ways, such as:

- **Accuracy** – how accurate is the controller in getting the robot where it needs to be? How error-prone is it to unexpected situations (a bump on the road, motor malfunction,...).
- **Input** – what of information does the controller needs to function properly (and accurately)? Does the information have to be real-time?

- **Complexity** – how difficult is it to implement/configure said controller? How computationally intensive it is?

There is a whole field of study called control theory that examines controllers much more comprehensively than we can in a few short articles. That's why we're only going to talk about a select few.

3.1 Sample Controller Class

Although each of the controllers will operate quite differently, they will all have (nearly) identical functions, that will allow us to use them interchangeably, without breaking the rest of the code.

```
1  class SampleControllerClass:
2      """Description of the class."""
3
4      def __init__(self, ...):
5          """Creates a controller object."""
6          pass
7
8      def set_goal(self, goal):
9          """Sets the goal of the controller."""
10         pass
11
12     def get_value(self):
13         """Returns the current controller value"""
14         pass
```

`__init__` is called when we want to create a controller object. In the actual controller implementations, "..." will be replaced by the parameters that the controller takes.

`set_goal` is called to set the controller's goal, where goal has to be a number. Note that we will need to call this function before trying to call `get_value`, or things will break. This makes sense, because the controller can't really help you to reach the goal if you haven't specified it yet.

`get_value` will return the value that the controller thinks we should set the motors to, to achieve our goal. **All controllers will return values between (and including) -1 and 1,** since it's more convenient to do math on numbers in that range.

All of this will make more sense as we go through each of the controllers.

3.2 Dead reckoning

One of the simplest ways of controlling the robot autonomously is using dead reckoning.

It uses one of the first equations you learned in physics: $time = distance/speed$. We use it to calculate, how long it takes something to travel a certain distance based on its average speed.

Let's look at an example: say our robot drives an average of $s = 2.5\frac{m}{s}$. We want it to drive a distance of $d = 10m$. To calculate, how long it will take the robot, all you have to do is divide distance by speed: $t = d/s = 10/2.5 = 4s$.

This is exactly what dead reckoning does – it calculates the time it will take the robot to drive the distance to the goal. When asked, returns 1 if that amount of time hasn't elapsed yet and 0 if it has.

3.2.1 Implementation

There are two things that the controller needs: the average speed of the robot and a way to measure how much time had passed.

```
1  class DeadReckoning:
2      """A class implementing a dead reckoning controller."""
3
4      def __init__(self, speed, get_current_time):
5          """Takes the average speed and a function that returns current time."""
6          self.get_current_time = get_current_time
7          self.speed = speed
8
9      def set_goal(self, goal):
10         """Sets the goal of the controller (and also starts the controller)."""
11         self.goal = goal
12         self.start_time = self.get_current_time()
13
14     def get_value(self):
15         """Return the current value the controller is returning."""
16         # at what time should we reach the destination (d=d_0 + s/v)
17         arrival_time = self.start_time + (self.goal / self.speed)
18
19         # return +-1 if we should have reached the destination and 0 if not
20         if self.get_current_time() < arrival_time:
21             return 1 if self.goal > 0 else -1
22         else:
23             return 0
```

As we see, the parameters the `__init__` function is expecting to get are:

- `speed` – the average speed of the robot.
- `get_current_time` – a function that returns the current time to measure, whether the calculated time had elapsed.

3.2.2 Example

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)

# create a controller object and set its goal
controller = DeadReckoning(2.5, get_current_time)
controller.set_goal(10)

while True:
    # get the controller value
    controller_value = controller.get_value()

    # drive the robot using tank drive controlled by the controller value
    tank_drive(controller_value, controller_value, left_motor, right_motor)
```

This is an implementation of the problem proposed in the Introduction: make a robot drive 10 meters.

Notice how we used our previously implemented `tank_drive` function to set both motors to drive forward. We could have written `left_motor(controller_value)` and `right_motor(controller_value)`, but this is a cleaner way of writing it.

3.2.3 Closing remarks

Although this is quite a simple controller to implement, you might realize that it is neither accurate nor practical. If the robot hits a bump on the road or slips on a banana peel, there is nothing it can do to correct the error, since it doesn't know where it is.

Another important thing to keep in mind when using this controller is that if you want to change how fast the robot is driving/turning, you will need to re-calculate the average speed of the robot, which is very tedious.

We'll be focusing on improving accuracy in the upcoming articles by incorporating real-time data from the robot into our controllers.

3.3 Bang–bang

Although our previous controller was quite easy to implement and use, there is no way for it to know whether it reached the target or not. It pretty much just turns the motors on for a while and hopes for the best.

Bang-bang aims to fix this problem by using feedback from our robot. Feedback could be values from its encoders (to measure how far it has gone), gyro (to measure where it's heading), or really anything else that we wish to control. The important thing

here is that the data is **real-time**. The robot continuously tells the controller what is happening, so the controller can act accordingly.

Bang-bang implements the very first idea that comes to mind when we have real-time data available. The controller will return 1 if we haven't passed the goal yet and 0 if we have.

3.3.1 Implementation

The only thing the Bang-bang controller needs is the feedback function returning information about the state of whatever we're trying to control.

```
class Bangbang:
    """A class implementing a bang-bang controller."""

    def __init__(self, get_feedback_value):
        """Create the bang-bang controller object from the feedback function."""
        self.get_feedback_value = get_feedback_value

    def set_goal(self, goal):
        """Sets the goal of the bang-bang controller."""
        self.goal = goal

    def get_value(self):
        """Returns +1 or -1 (depending on the value of the goal) when the robot
        should be driving and 0 when it reaches the destination."""
        if self.goal > 0:
            if self.get_feedback_value() < self.goal:
                return 1    # goal not reached and is positive -> drive forward
        else:
            if self.get_feedback_value() > self.goal:
                return -1   # goal not reached and is negative -> drive backward

        # if it shouldn't be driving neither forward nor backward
        return 0
```

3.3.2 Examples

To make the robot drive a distance using this controller, we need a new `Encoder` class to measure how far the robot has driven. The objects of this class return the average of the distance driven by the left wheel and by the right wheel.

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)
encoder = Encoder()

# create a controller object and set its goal
controller = Bangbang(encoder)
controller.set_goal(10)

```

```
while True:
    # get the controller value
    controller_value = controller.get_value()

    # drive the robot using tank drive controlled by the controller value
    tank_drive(controller_value, controller_value, left_motor, right_motor)
```

Notice that pretty much nothing changed between this and the dead reckoning example. This is the main advantage of all of the controllers having the same functions – we can use controller objects almost interchangeably, allowing us to easily try out and compare the accuracies of each of the controllers, without messing with the rest of our code.

3.3.3 Closing remarks

This is already markedly better than our previous dead reckoning approach, but it is still relatively inaccurate: the robot's inertia will make the robot drive a little extra distance when the controller tells it that it shouldn't be driving anymore, which means it will likely overshoot.

We could try to fix this by saying that it should start driving backward once it passed the goal, but the only thing you'd get is a robot that drives back and forth across the goal (which may be amusing, but not very helpful).

In the upcoming articles, we will try to improve our approach and create controllers that don't just return 1 for driving and 0 for not driving, but also values in-between (when the robot should be driving slower and when faster).

3.4 PID

Our previous attempt at creating a controller that used feedback from the robot could be further improved by considering how the **error** (difference between feedback value and the goal) changes over time.

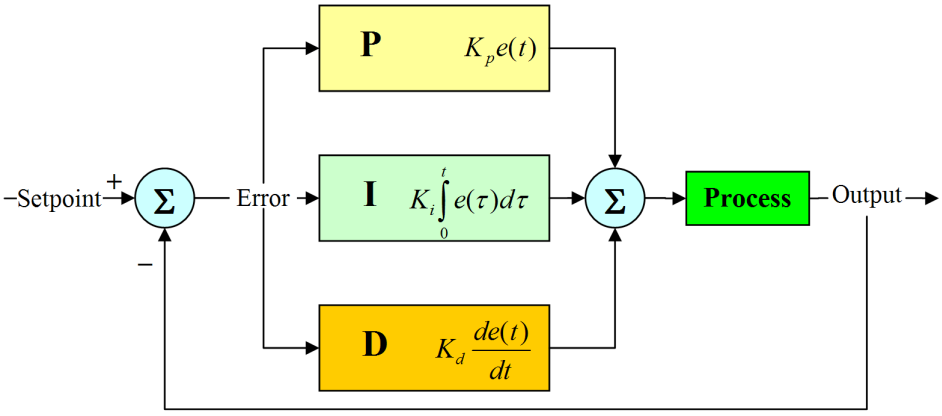

Figure 5: PID

Since PID is an abbreviation, let's talk about that the terms P, I and D mean:

- P stands for **proportional** – how large is the error now (in the **present**).
- I stands for **integral** – how large the error was in the **past.**
- D stands for **derivative** – what will the error likely be in the **future.**

The controller takes into account what happened, what is happening, and what will likely happen, and combine these things to produce the controller value.

3.4.1 Deriving the equations

Before diving into the equations, we need to define a terms to build the equations from:

- e – the current error (difference between robot position and its goal).
- Δe – difference between the current and the previous error.
- Δt – time elapsed since the last measurement.
- p, i, d – constants (positive real numbers) to determine, how important each of the terms are. This make it so that we can put more emphasis on some parts of the controller than others (or ignore them entirely).

3.4.1.1 Proportional

Proportional is quite straight forward – it only takes into account, how big the error is right now.

$$P = e$$

The problem with only using only P is that the closer we get to the goal, the smaller the value of this term is. The robot's movement would feel stiff and there is a chance that it wouldn't even reach the goal. That's why it needs to be complemented by the other parts, for the controller to be effective.

3.4.1.2 Integral

Integral adds the extra push that the proportional was missing, because it doesn't react to what is happening right now, but to what was happening in the past, by accumulating the error.

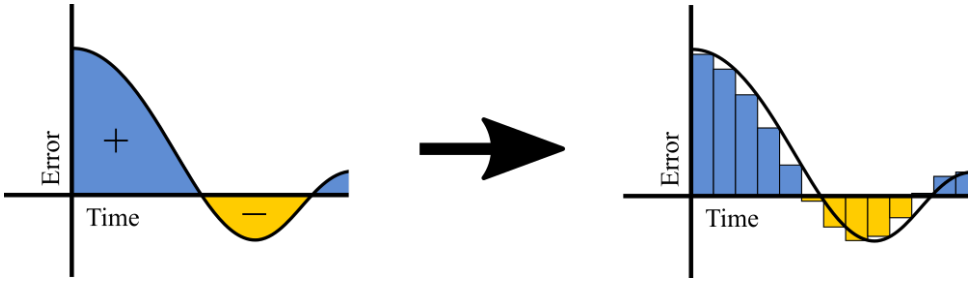

Figure 6: Integral

Calculating an integral means calculating area under a curve (in our case, the curve is error over time). With real-world measurements, we can't calculate the actual area, because we can only call the code so many times a second. That's why we will approximate the area by calculating rectangles that closely resemble the curve.

For each computation, the height of the rectangle is the error e, and width is the elapsed time Δt since the last measurement. To calculate the rectangular area, we multiply these two numbers together. I itself is the sum of all of these values.

$$I \mathrel{+}= e \cdot \Delta t$$

This can, however, introduce additional instability to the controller, since the values can accumulate and cause overshoot, and the reaction could also potentially be slow, due to windup.

3.4.1.3 Derivative

Derivative aims to further improve the controller by damping the values. We will calculate the rate of change of the error to predict its future behavior – the faster the robot goes, the bigger Δe is, and the more it will push back against the other terms.

$$D = \frac{\Delta e}{\Delta t}$$

A note to be made is that if $\Delta t = 0$, the derivative can't be calculated, because we would be dividing by zero – just something to keep in mind for the implementation.

3.4.1.4 Result

As we have previously said, the result is adding all of those terms, multiplied by their constants (determines the importance of the terms in the result).

$$P \cdot p + I \cdot i + D \cdot d$$

The only thing we need to keep in mind is that the values could exceed 1 (or -1). If they do, we will simply return 1 (or -1).

3.4.2 Implementation

The controller will need the p, i and d constants. It will also need a feedback function and, to correctly calculate the integral and derivative, a function that returns the current time.

```
1  class PID:
2      """A class implementing a PID controller."""
3
4      def __init__(self, p, i, d, get_current_time, get_feedback_value):
5          """Initialises PID controller object from P, I, D constants, a function
6          that returns current time and the feedback function."""
7          # p, i, and d constants
8          self.p, self.i, self.d = p, i, d
9
10         # saves the functions that return the time and the feedback
11         self.get_current_time = get_current_time
12         self.get_feedback_value = get_feedback_value
13
14     def reset(self):
15         """Resets/creates variables for calculating the PID values."""
16         # reset PID values
17         self.proportional, self.integral, self.derivative = 0, 0, 0
18
19         # reset previous time and error variables
20         self.previous_time, self.previous_error = 0, 0
21
22     def get_value(self):
23         """Calculates and returns the PID value."""
24         # calculate the error (how far off the goal are we)
25         error = self.goal - self.get_feedback_value()
26
27         # get current time
28         time = self.get_current_time()
29
30         # time and error differences to the previous get_value call
31         delta_time = time - self.previous_time
32         delta_error = error - self.previous_error
33
34         # calculate proportional (just error times the p constant)
35         self.proportional = self.p * error
36
```

```
37          # calculate integral (error accumulated over time times the constant)
38          self.integral += error * delta_time * self.i
39
40          # calculate derivative (rate of change of the error)
41          # for the rate of change, delta_time can't be 0 (divison by zero...)
42          self.derivative = 0
43          if delta_time > 0:
44              self.derivative = delta_error / delta_time * self.d
45
46          # update previous error and previous time values to the current values
47          self.previous_time, self.previous_error = time, error
48
49          # add P, I and D
50          pid = self.proportional + self.integral + self.derivative
51
52          # return pid adjusted to values from -1 to +1
53          return 1 if pid > 1 else -1 if pid < -1 else pid
54
55      def set_goal(self, goal):
56          """Sets the goal and resets the controller variables."""
57          self.goal = goal
58          self.reset()
```

To fully understand how the controller works, I suggest you closely examine the `get_value()` function – that's where all the computation happens.

Notice a new function called `reset` that we haven't seen in any of the other controllers. It is called every time we set the goal, because the controller accumulates error over time in the `integral` variable, and it would therefore take longer to adjust to the new goal.

It doesn't change the versatility of the controller classes, because we don't need to call it in order for the controller to function properly, it's just a useful function to have.

3.4.3 Tuning the controller

PID is the first discussed controller that needs to be tuned correctly to perform well (besides dead reckoning, where you have to correctly calculate the average speed). Tuning is done by adjusting the p, i and d constants, until the controller is performing the way we want it to.

There is a whole section on Wikipedia about PID tuning. We won't go into details (read through the Wikipedia article if you're interested), but it is just something to keep in mind when using PID, because incorrect tuning could have disastrous results.

3.4.4 Examples

3.4.4.1 Driving a distance

Here is an example that makes the robot drive 10 meters forward. The constants are values that I used on the VEX EDR robot that I built to test the PID code, you will likely have to use different ones.

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)
encoder = Encoder()

# create a controller object and set its goal
controller = PID(0.07, 0.001, 0.002, time, encoder)
controller.set_goal(10)

while True:
    # get the controller value
    controller_value = controller.get_value()

    # drive the robot using tank drive controlled by the controller value
    tank_drive(controller_value, controller_value, left_motor, right_motor)
```

3.4.4.2 Auto-correct heading

Auto-correcting the heading of a robot is something PID is great at. What we want is to program the robot so that if something (like an evil human) pushes it, the robot adjusts itself to head the way it was heading before the push.

We could either use values from the encoders on the left and the right side to calculate the angle, but a more accurate way is to use a gyroscope. Let's therefore assume that we have a `Gyro` class whose objects give us the current heading of the robot.

One thing we have to think about is what to set the motors to when we get the value from the controller, because to turn the robot, both of the motors will be going in opposite directions. Luckily, `arcade_drive` is our savior – we can plug our PID values directly into the turning part of arcade drive (the `x` axis) to steer the robot. Refer back to the Arcade Drive article, if you are unsure as to how/why this works.

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)
gyro = Gyro()

# create a controller object and set its goal
controller = PID(0.2, 0.002, 0.015, time, gyro)
controller.set_goal(0)  # the goal is 0 because we want to head straight

while True:
```

```
    # get the controller value
    controller_value = controller.get_value()

    # drive the robot using arcade drive controlled by the controller value
    arcade_drive(controller_value, 0, left_motor, right_motor)
```

3.4.5 Closing remarks

PID is one of the most widely used controllers not just in robotics, but in many other industries, because it is reliable, relatively easy to implement and quite precise for most use cases.

For motivation, here is a video demonstrating the power of a correctly configured PID controller.

3.5 Polynomial Function

Another way that we can get values that aren't just 1's and 0's is to model a function from points and get the speed of the robot from it – for example: let's say that we start at speed 0.2, drive at full speed when we're at half the distance and slow down to 0 when we're at the end.

Polynomial function is a great candidate for this task. We can pick points that we want the function to pass through and then use polynomial regression to get the coefficients of the function.

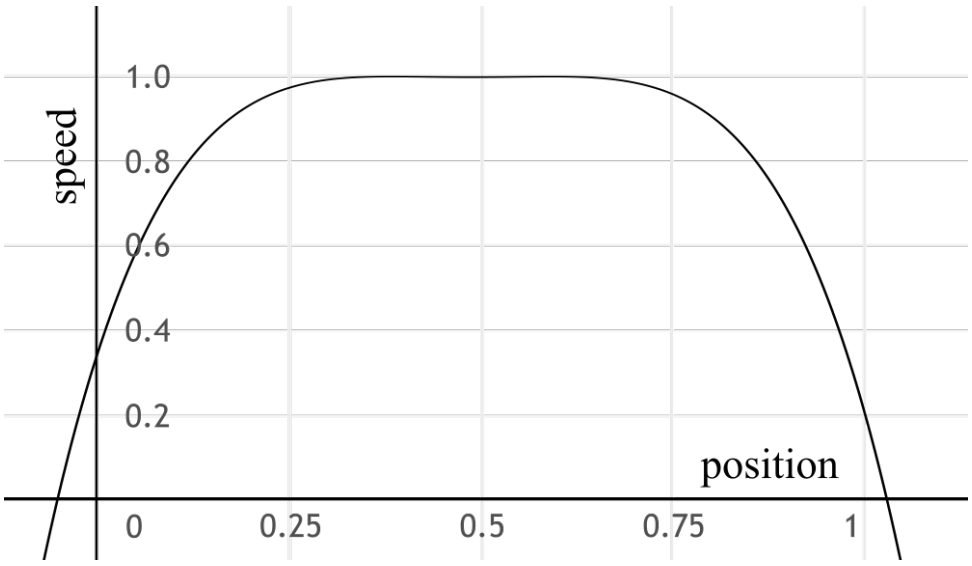

Figure 7: Polynomial function

One thing you should also notice is that the function starts at $x = 0$ and ends at $x = 1$. This is deliberate – it makes it easy for us to "stretch" the function a little wider if we want to drive some other distance, not just a distance of 1.

3.5.1 Horner's method

When it comes to programming, exponentiation tends to be quite expensive. Horner's method is an elegant solution to this problem. The concept is simple – change the expression by repeatedly taking out x, so there is no exponentiation.

$$2x^3 + 4x^2 - x + 5 \quad \rightarrow \quad x(x(x(2) + 4) - 1) + 5$$

This requires fewer multiplications, thus making the evaluation faster.

3.5.2 Implementation

The controller only needs the coefficients of the polynomial that we modeled, and a feedback function.

```
1  class PolynomialFunction:
2      """A class implementing a polynomial function controller."""
3
4      def __init__(self, coefficients, get_feedback_value):
5          """Initialises the polynomial function controller from the polynomial
6          coefficients and the feedback value."""
7          self.coefficients = coefficients    # the coefficients of the function
8          self.get_feedback_value = get_feedback_value   # the feedback function
9
10     def get_value(self):
11         """Returns the polynomial function value at feedback function value."""
12         # calculate the x coordinate (by "stretching" the function by goal)
13         x = self.get_feedback_value() / abs(self.goal)
14
15         # calculate function value using Horner's method
16         value = self.coefficients[0]
17         for i in range(1, len(self.coefficients)):
18             value = x * value + self.coefficients[i]
19
20         # if the value is over 1, set it to 1
21         if value > 1:
22             value = 1
23
24         # if goal is negative, function value is negative
25         return value if self.goal > 0 else -value
26
27     def set_goal(self, goal):
28         """Sets the goal of the controller."""
29         self.goal = goal
```

3.5.3 Examples

3.5.3.1 Driving a distance

Once again, the code is almost the exact same as the examples from nearly all of the other controllers.

```
# initialize objects that control robot components
left_motor = Motor(1)
right_motor = Motor(2)
encoder = Encoder()

# create a controller object and set its goal
controller = PolynomialFunction([-15.69, 30.56, -21.97, 6.91, 0.2], encoder)
controller.set_goal(10)

while True:
    # get the controller value
    controller_value = controller.get_value()

    # drive the robot using tank drive controlled by the controller value
    tank_drive(controller_value, controller_value, left_motor, right_motor)
```

3.5.4 Generating a polynomial

An alternative to "stretching" the polynomial to fit the goal is to specify the points the polynomial passes through and generate the coefficients *after* the goal is specified.

Say you have points $(0, 0.2)$, $(0.4, 1)$, $(0.6, 1)$ and $(1, 0)$. Since there are 4 points, the general form of the polynomial is $y = ax^3 + bx^2 + cx + d$ Using this information, we can create a system of linear equations:

$$\begin{aligned} 0.2 &= a(0)^3 + b(0)^2 + c(0) + d \\ 1 &= a(0.4)^3 + b(0.4)^2 + c(0.4) + d \\ 1 &= a(0.6)^3 + b(0.6)^2 + c(0.6) + d \\ 0 &= a(1)^3 + b(1)^2 + c(1) + d \end{aligned}$$

Solving this system of linear equations will give us the coefficients of the polynomial. We can apply this method to a polynomial of any degree d, if we have at least $d + 1$ number of points.

3.5.5 Closing remarks

Although this controller isn't as widely used as PID, it can sometimes outperform PID, namely in situations where the ranges of movement of the motors are restricted – forks of a forklift/robot arm.

4 Odometry

Odometry is the use of data from sensors of the robot to estimate its position over time.

This means that we can calculate where the robot is at the moment, by getting information from sensors such as an encoder, a gyro, or a camera, and performing calculations on said information.

4.1 Heading from Encoders

There are two things we need to know to approximate the position:

- **Distance** Δ – how much did we move by?
- **Heading** Δ – which way are we heading?

Assuming we have encoders on both sides of the robot, the distance is easy to calculate: we can read the values the encoders on both of the sides are reading and average them. Assuming we also have a gyro, heading is easy too: we can get the heading directly as the values the gyro is returning.

But what if we didn't have a gyro?

4.1.1 Calculate heading without gyro

Although gyro is arguably the most precise way to measure the current heading, it's not always available. It might be too expensive, impractical to include on a small robot, or not viable because of other conditions. In cases like these, it is good to know how to calculate heading only from the encoders.

Say the robot drove a small arc. The left encoder measured a distance l and the right side measured a distance r. The length between the two wheels of the drivetrain is c, the angle by which we turned is ω (measured in radians), and x is just a variable to help with our calculations.

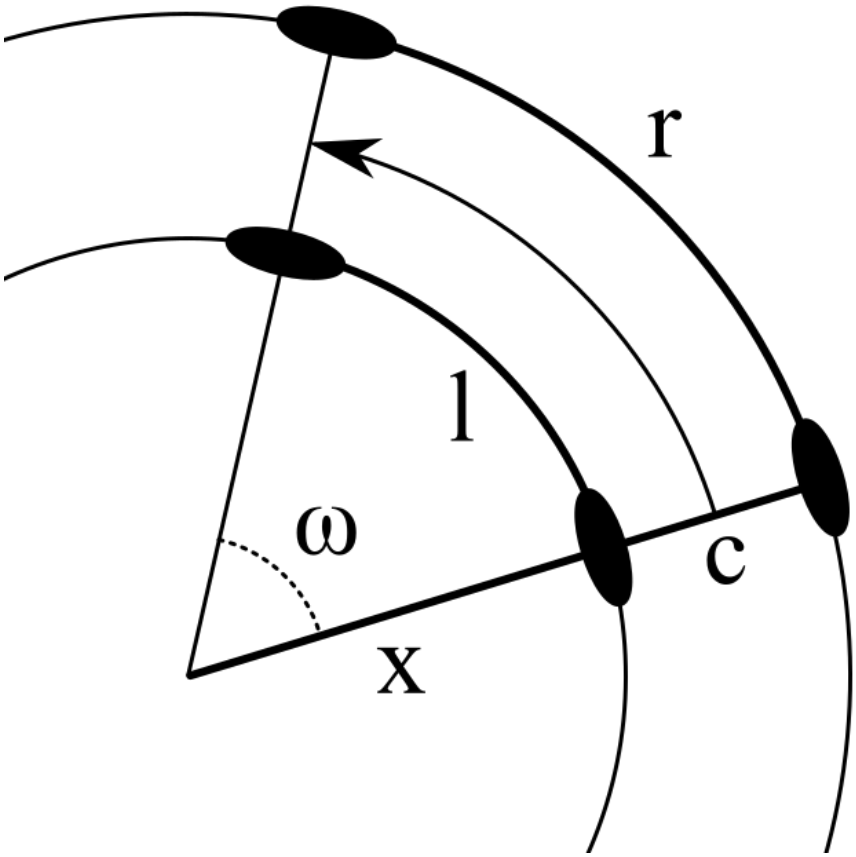

Figure 8: Heading from encoders

Let's derive the equations for the lengths of the arches l and r in terms of the other variables (here is an article about arc length, if you need further clarification):

$$l = x \cdot \omega \qquad r = (c + x) \cdot \omega$$

We can then combine the equations, simplify, and solve for ω:

$$\frac{l}{\omega} = \frac{r}{\omega} - c$$

$$\omega = \frac{r - l}{c}$$

And that's it! The angle can be calculated from the difference of the readings of the encoders, divided by the length of the axis.

4.2 Line Approximation

For our first approximation, let's make an assumption: instead of driving an arc, the robot will first turn by the specified angle, and only then drive the distance in a straight line.

This is quite a reasonable assumption to make for smaller angles: since we are updating the position multiple times per second, the angles aren't going to be too large.

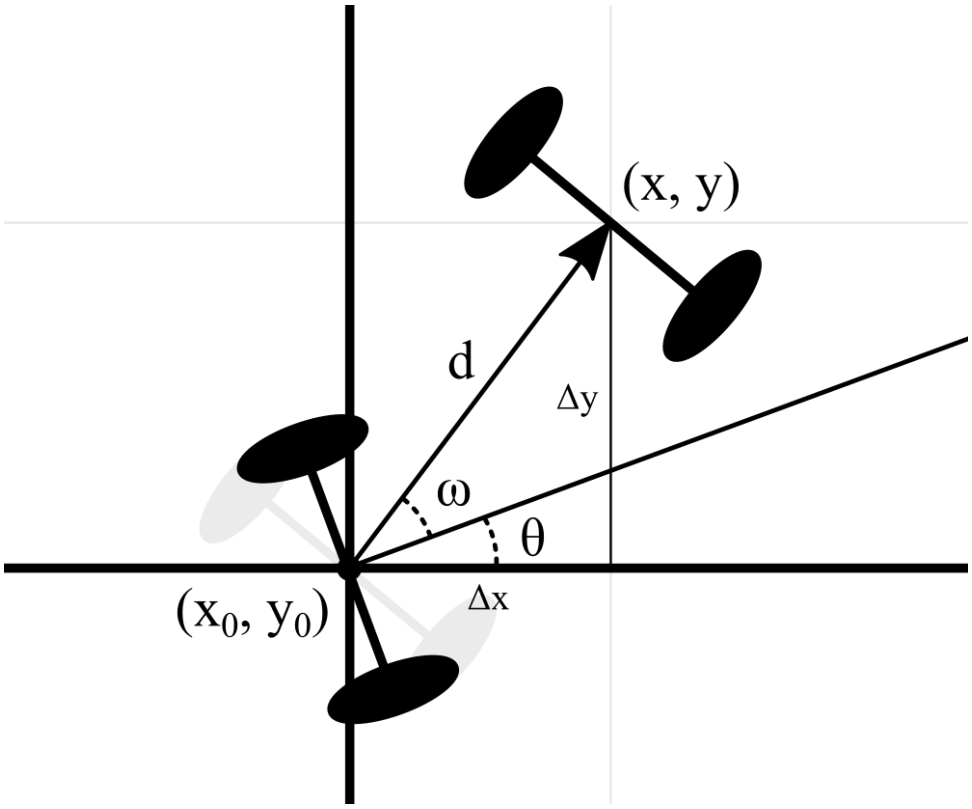

Figure 9: Line Approximation

4.2.1 Deriving the equations

The robot moved a distance d. It was previously at an angle θ and is now at an angle $\theta + \omega$. We want to calculate, what the new position of the robot is after this move.

One of the ways to do this is to imagine a right triangle with d being hypotenuse. We will use trigonometric formulas and solve for Δx and Δy:

$$sin(\theta + \omega) = \frac{\Delta y}{d} \qquad cos(\theta + \omega) = \frac{\Delta x}{d}$$

$$\Delta y = d \cdot sin(\theta + \omega) \qquad \Delta x = d \cdot cos(\theta + \omega)$$

The resulting coordinates (x, y) are $x = x_0 + \Delta x$ and $y = y_0 + \Delta y$.

4.2.2 Implementation

This is one of the possible implementations of a class that tracks the current position of the robot by getting information from both encoders using the aforementioned line approximation method.

Note that for the position estimation to be accurate, the `update()` function of the class needs to be called multiple times per second.

```
from math import cos, sin


class LineApproximation:
    """A class to track the position of the robot in a system of coordinates
    using only encoders as feedback, using the line approximation method."""

    def __init__(self, axis_width, l_encoder, r_encoder):
        """Saves input values, initializes class variables."""
        self.axis_width = axis_width
        self.l_encoder,  self.r_encoder = l_encoder, r_encoder

        # previous values for the encoder position and heading
        self.prev_l, self.prev_r, self.prev_heading = 0, 0, 0

        # starting position of the robot
        self.x, self.y = 0, 0

    def update(self):
        """Update the position of the robot."""
        # get sensor values and the previous heading
        l, r, heading = self.l_encoder(), self.r_encoder(), self.prev_heading

        # calculate encoder deltas (differences from this and previous readings)
        l_delta, r_delta = l - self.prev_l, r - self.prev_r

        # calculate omega
        h_delta = (r_delta - l_delta) / self.axis_width

        # approximate the position using the line approximation method
        self.x += l_delta * cos(heading + h_delta)
        self.y += r_delta * sin(heading + h_delta)

        # set previous values to current values
        self.prev_l, self.prev_r, self.prev_heading = l, r, heading + h_delta

    def get_position(self):
        """Return the position of the robot."""
        return (self.x, self.y)
```

4.3 Circle Approximation

Although the line approximation is relatively accurate, we can make it more precise by assuming that the robot drives in an arc, which, in reality, is closer to what the robot is really doing.

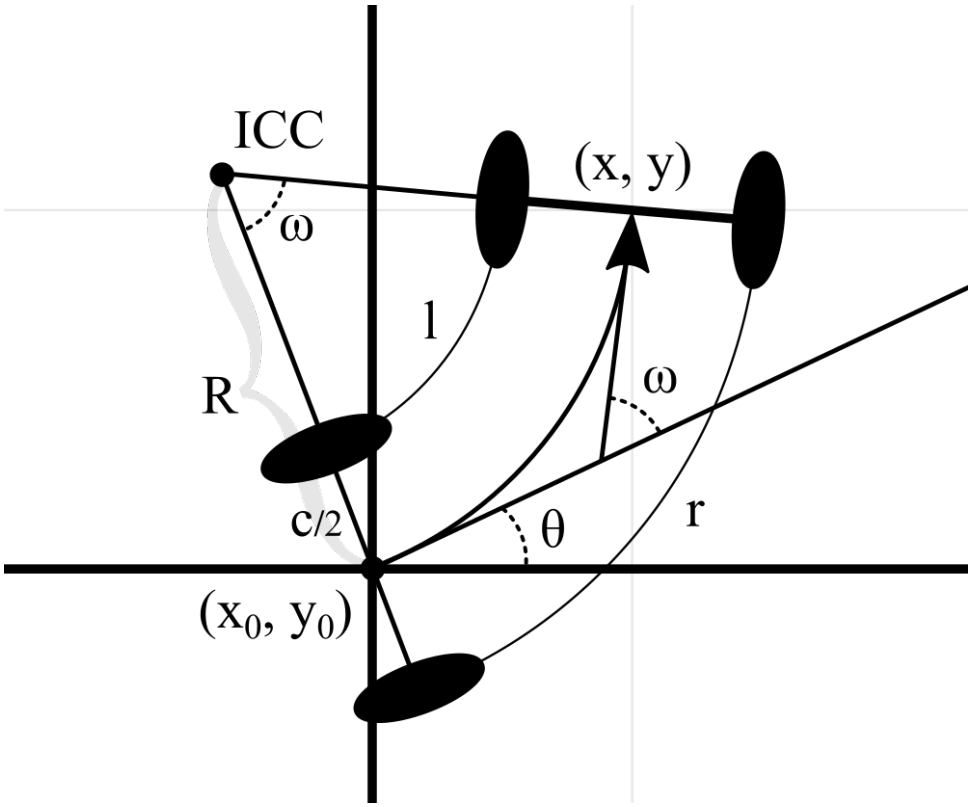

Figure 10: Circle Approximation

4.3.1 Deriving the equations

The robot moved in an arch, the left encoder measured a distance l and the right wheel a distance r. It was previously at an angle θ and is now at an angle $\theta + \omega$. We want to calculate, what the new position of the robot is after this move.

The way to calculate the new coordinates is to find the radius R of ICC (*Instantaneous Center of Curvature* – the point around which the robot is turning) and then rotate x_0 and y_0 around it.

4.3.1.1 Calculating R

Let's start by finding the formula for calculating R. We will derive it from the formulas for calculating l and r:

$$l = \omega \cdot \left(R - \frac{c}{2}\right) \qquad r = \omega \cdot \left(R + \frac{c}{2}\right)$$

Combining the equations and solving for R gives us:

$$R = \left(\frac{r+l}{\omega \cdot 2}\right)$$

From our previous article about Heading from Encoders, we know that $\omega = \frac{r-l}{c}$. If we don't have a gyro, we can just plug that into our newly derived formula and get:

$$R = \frac{r+l}{\frac{r-l}{c} \cdot 2} = \frac{r+l}{r-l} \cdot \frac{c}{2}$$

4.3.1.2 Rotating (x_0, y_0) around ICC

For this section, we will assume that you know how to rotate a point around the origin by a certain angle. If not, here is a video from Khan Academy deriving the equations.

First of, we will need the coordinates of the ICC. Since it's perpendicular to the left of the robot, we can calculate it by first calculating a point that is distance R in front of the robot:

$$ICC_x = x_0 + R\ cos(\theta) \qquad ICC_y = y_0 + R \cdot sin(\theta)$$

We will then turn the point 90 degrees to place it at distance R perpendicular to the robot by switching x_0 and y_0 and negating the first coordinate (using simple vector algebra):

$$ICC_x = x_0 - R\ sin(\theta) \qquad ICC_y = y_0 + R \cdot cos(\theta)$$

To rotate (x_0, y_0) around ICC (and therefore find (x, y)), we will first translate ICC to the origin, then rotate, and then translate back:

$$x = (x_0 - ICC_x) \cdot cos(\omega) - (y_0 - ICC_y) \cdot sin(\omega) + ICC_x$$

$$y = (x_0 - ICC_x) \cdot sin(\omega) + (y_0 - ICC_y) \cdot cos(\omega) + ICC_y$$

Plugging in the values for ICC_x, ICC_y and simplifying:

$$x = R\ sin(\theta) \cdot cos(\omega) + R \cdot cos(\theta) \cdot sin(\omega) + x_0 - R\ sin(\theta)$$

$$y = R\ sin(\theta) \cdot sin(\omega) - R \cdot cos(\theta) \cdot cos(\omega) + y_0 + R\ cos(\theta)$$

Using trigonometric rules, the equations can be further simplified to:

$$x = x_0 + R\ sin(\theta + \omega) - R\ sin(\theta)$$

$$y = y_0 - R\ cos(\theta + \omega) + R\ cos(\theta)$$

4.3.1.3 Edge cases

There is one noteworthy case of values l and r where our circle approximation method won't work.

If $r = l$ (if we are driving straight), then the radius cannot be calculated, because it would be "infinite". For this reason, our method can't be used on its own, because if the robot drove in a straight line, the code would crash.

We can, however, still employ or line approximation method, since in this case, it really is driving in a straight line! It is also a good idea to apply the line approximation method for very small angles, since it's almost like driving straight, and it's less computationally intensive to the circle approximation method.

4.3.2 Implementation

Here is the implementation, combining both of the approximation methods:

```
from math import cos, sin


class CircleApproximation:
    """A class to track the position of the robot in a system of coordinates
    using only encoders as feedback, using a combination of line and circle
    approximation methods."""

    def __init__(self, axis_width, l_encoder, r_encoder):
        """Saves input values, initializes class variables."""
        self.axis_width = axis_width
        self.l_encoder,  self.r_encoder = l_encoder, r_encoder

        # previous values for the encoder position and heading
        self.prev_l, self.prev_r, self.prev_heading = 0, 0, 0

        # starting position of the robot
        self.x, self.y = 0, 0

    def update(self):
        """Update the position of the robot."""
```

```
        # get sensor values and the previous heading
        l, r, heading = self.l_encoder(), self.r_encoder(), self.prev_heading

        # calculate encoder deltas (differences from this and previous readings)
        l_delta, r_delta = l - self.prev_l, r - self.prev_r

        # calculate omega
        h_delta = (r_delta - l_delta) / self.axis_width

        # either approximate if we're going (almost) straight or calculate arc
        if abs(l_delta - r_delta) < 1e-5:
            self.x += l_delta * cos(heading)
            self.y += r_delta * sin(heading)
        else:
            # calculate the radius of ICC
            R = (self.axis_width / 2) * (r_delta + l_delta) / (r_delta - l_delta)

            # calculate the robot position by finding a point that is rotated
            # around ICC by heading delta
            self.x += R * sin(h_delta + heading) - R * sin(heading)
            self.y += - R * cos(h_delta + heading) + R * cos(heading)

        # set previous values to current values
        self.prev_l, self.prev_r, self.prev_heading = l, r, heading + h_delta

    def get_position(self):
        """Return the position of the robot."""
        return (self.x, self.y)
```

5 Resources

Links to resources either directly used by the website (such as libraries), or those that helped me understand the concepts mentioned in the articles.

5.1 Math

The website uses KaTeX to render LaTeX equations.

5.2 Images

The images used to illustrate the concepts on this website are modified using Inkscape (free vector graphics editor) and GIMP (free bitmap graphics editor). CAD model images are generated with Fusion 360 (free CAD design software, assuming you are a student). I also used the TinyPNG website to compress the images.

5.3 p5.js

Visualizations on the website are created using the p5.js library. This example helped me understand how it worked with Jekyll. I use the p5js web editor to edit the visualizations before I put them on the website.

5.4 VEX EDR

To test the algorithms, I built a custom VEX EDR robot using this kit that the educational center VCT kindly lent me. The robot is programed in Python using RobotMesh studio (for more information, see the Python documentation).

5.5 Autonomous motion control

PythonRobotics is a great repository containing implementations of various robotics algorithms in Python.

5.6 PID

I studied a PID Python implementation before writing my own.

5.7 Drivetrain Control

There were a few helpful articles that helped me understand equations behind the more complex drivetrains:

- Arcade drive Chief Delphi forum post by Ether.
- Simplistic Control of Mecanum Drive paper.
- Swerve drive Chief Delphi forum post by Ether.

5.8 Circle Approximation

There were two main resources that helped me write the Circle Approximation article.

- Kinematics Equations for Differential Drive and Articulated Steering whitepaper
- Position Estimation presentation

6 About

6.1 Motivation

When I joined the FRC team Metal Moose and started learning about robotics, I didn't find many beginner-friendly resources for people like me. They were hard to

find and scattered all around the web. One had to be quite persistent to actually learn something on their own.

That is the main reason for the creation of this website – to serve as a resource for people that want to learn the concepts of robotics, without having to go though all the trouble of finding quality resources.

6.2 Contributions

I accept any and all recommendations/suggestions about this website. If there is anything you think should be changed/corrected, or you have a topic that you wish there was an article about, feel free to let me know.

You can do so either by email (`tomas.slama.131@gmail.com`), or by creating an issue on the website's GitHub repositories (see link on the top right of this page).

6.3 Acknowledgements

I would like to thank the following people for their help in making this project a reality:

- **Kateřina Sulková** for being loving, supportive, and especially helpful in writing the SOČ paper.
- **Matěj Halama** (*matejhalama.cz*) and **Jan Hladík** for help in designing the logo of the website.
- **VCT** for kindly lending me a VEX EDR kit to work with.