Lesson 5: PID Control

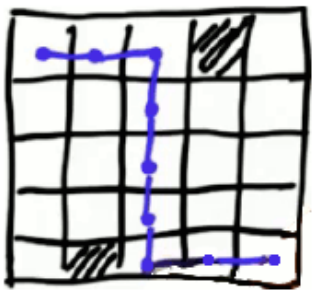
UDACITY

Robot Motion

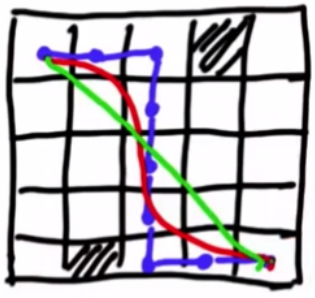
In this lesson you will learn about actual robot motion. In the previous lesson you learned how to find paths; now, you will learn about how to turn these paths into actual robot commands.

Specifically, you will learn how to generate smooth paths. And then you will learn about control, in particular, a method called PID control. And as usual you will get the chance to program these wonderful things.

You might remember that you originally did planning in a discrete world and the kind of plans you found looked like this:



A path like this has many disadvantages. You don't want the robot to go straight, take a 90-degree turn, then go straight again. A car cannot even make a 90-degree turn, and this route will force the robot to move really slowly around the corners. A much better path would look like the red curve in this next picture:



This is a much smoother path. In the extreme case, you might generate a path like the green line.

Question 1 (Robot Motion):

Suppose we have a robotic car in the top-left corner, facing right, and we want it to get to the bottom-right corner, facing right. Which path above would we prefer it to take: the blue path, red path, or green path?

Answer:

The answer is a bit subtle. We actually prefer the red path. As we saw earlier, the blue path has many disadvantages. The green path is also suboptimal: since the robot is facing to the right side, it would have to make an instantaneous 45-degree turn to take the green path. A better path would gently move around the corner, go down and gently move to the right. The green path might be shorter, but it requires two instantaneous turns, which are infeasible for the robot to make.

In the previous lesson on path-planning, you specified paths as a sequence of points in a two-dimensional grid, giving a path similar to the blue one above. Now we will see how to modify the blue path to become more like the smooth, red path.

For our purposes we will start with points x_i, where the variable i will vary from 0 to n. Think of these x_i as the nonsmooth path locations the planner from Lesson 4 found. Each x_i will denote a point with two coordinates, but the algorithm we will learn will apply to points in any number of dimensions.

We will create new variables, $y_0,\ldots,y_n$, and initialize each y_i to x_i. Then, we will smooth the path by minimizing the quantities $||x_i - y_i||^2$ and $||y_i - y_{i+1}||^2$ for all i.

The first expression is the square of the distance between the i^{th} original point and the i^{th} smooth point, and the second expression is the square of the distance between two consecutive smooth points.

Question 2 (Smoothing Algorithm):

What happens if we only minimize expressions of the first type: $||x_i - y_i||^2$?

Answer:

The value of each of these expressions is already 0 because we initialize each y_i to x_i, so we cannot make them any smaller. Thus, minimizing $||x_i - y_i||^2$ gives us the original path.

Question 3 (Smoothing Algorithm 2):

This question is the same, but about the second type of expression: what happens if we only minimize the expressions $||y_i - y_{i+1}||^2$?

Answer:

In this case, we will actually get no path at all. Minimizing these expressions is the same as making each y_i as similar to y_{i+1} as possible. The most similar they can be is the exact same point. Doing that for all the points would mean that $y_0 = y_1 = ... = y_n$, so we would get a single point and no path at all.

These two criteria are in conflict with each other: making the y_i close to the original points means that the path tends to be blocky and not smooth (like the blue path above), and making the y_i close to one another makes the path smoother but less true to the original path (like the green path above).

Question 4 (Smoothing Algorithm 3):

Suppose we optimize both expressions at the same time using the formula

$$||x_i - y_i||^2 + \alpha||y_i - y_{i+1}||^2$$

with an appropriate value for α. What sort of path do we get?

Answer:

In this case, you get a smooth path. In practice, we minimize both by minimizing a linear combination

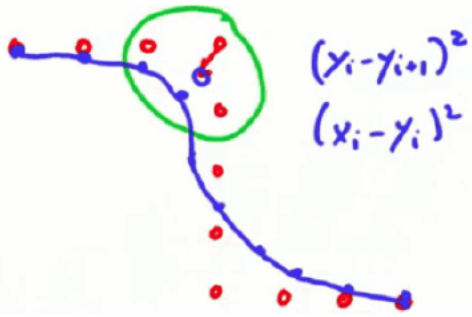
$$c_1||x_i - y_i||^2 + c_2||y_i - y_{i+1}||^2,$$

where c_1 and c_2 are parameters that determine the smoothness of the path. The larger c_2 is relative to c_1, the smoother the path will be, and the larger c_1 is relative to c_2, the more the path stays close to the original set of points. Finding the right balance between these two conflicting demands is what gives us a smooth path like the red one in our example above.

To see why, let's simulate the optimization.

Suppose we are given a solution to the planning problem like the red points in the next picture and we run the optimization algorithm. Consider a place like the first corner circled in green. By shifting the corner point in the down-left direction (the red arrow) towards a new position (the blue circle) and perhaps by shifting some other points, you can decrease the second error term

($||y_i - y_{i+1}||^2$) for both the first pair of points and second pair of points. However, you do this at the expense of the first error term ($||x_i - y_i||^2$), since you are now shifting y_i away from the original x_i. Depending on the weights (the c_1 and c_2) of these error terms, you might arrive at path like the following:



Although this new path suffers greater error from the $||x_i - y_i||^2$ term, it drastically reduces the distance between points and therefore reduces the error from $||y_i - y_{i+1}||^2$, thus minimizing the overall function $c_1||x_i - y_i||^2 + c_2||y_i - y_{i+1}||^2$.

How do we know the "right way" to optimize both of these expressions? One method is to use *gradient descent*: that is, for every time step, we adjust y_i in a way that minimizes the formula $c_1||x_i - y_i||^2 + c_2||y_i - y_{i+1}||^2$.

To do this, after initializing the points y_i, we then make the following assignments for every i between 1 and n-1:

$$y_i = y_i + \alpha(x_i - y_i) + \beta(y_{i-1} - 2y_i + y_{i+1}),$$

where α and β are real numbers. For now, let's take $\alpha = 0.5$ and $\beta = 0.1$ (note that this α is not exactly the same as the α in the last question.) We also keep track of the amount by which we change each y_i in a given pass through the i's.

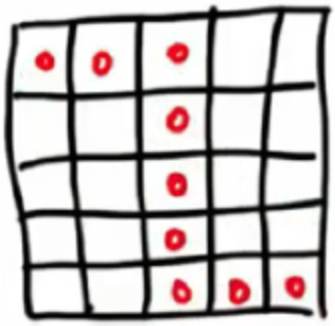
Notice that at each iteration, we assign to y_i the old value of y_i plus two terms. The first term moves us a little bit away from y_i toward x_i, and the second term moves a little bit away from y_i and towards the other smooth points y_{i+1} and y_{i-1}. That's how this procedure attempts to simultaneously minimize both $||x_i - y_i||^2$ and $||y_i - y_{i+1}||^2$.

We continue looping through i=1..n-1 until the total amount we change in a given pass is less than some tolerance parameter we pick beforehand. At that point, we end the iteration, and the values of the y_i's give us the final positions of the points that will determine our smooth path.

One last caveat: we do not apply this iteration to the first or the last point in the sequence, since the initial point and the destination point should remain the same. In other words, y_0 will always equal x_0, and y_n will always equal x_n.

Question 5 (Path Smoothing):

The path in the next exercise is in a 5 × 5 grid, starting at [0, 0] and going to [4, 4]:



Your job is to implement the function smooth(), which takes as an input the path, the two weighting factors, and a tolerance parameter.

```
def smooth(path, weight_data, weight_smooth, tolerance)
```

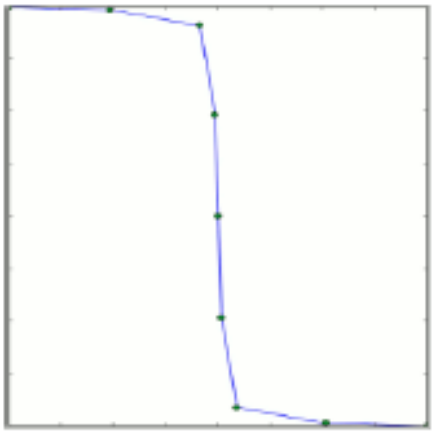
This should create a new path, named `newpath`, that consists of the points y_i from the equations we've discussed.

Notice that you cannot state make the assignment `oldpath = path`, because Python passes variables by reference. This would not make a copy of path but rather just make a pointer (in C terms) to `path`. Therefore, modifying `newpath` would modify `path` also, and you do not want that to happen. The code to make the deep copy is provided with the example.

The smoothing function should iteratively apply the update equation to all points except for the first and the last. It should does so until the total change observed in the update step becomes smaller than tolerance, at which point you can consider the smoother to have converged.

Use the provided code to print out the results after a test run of your implemented function. After hitting run you should see a result like the one below; on the left side you see the original path, and on the right you see that the new path is smoother than the original:

```
[0.000, 0.000] -> [0.000, 0.000]
[0.000, 1.000] -> [0.029, 0.971]
[0.000, 2.000] -> [0.176, 1.824]
[1.000, 2.000] -> [1.029, 1.971]
[2.000, 2.000] -> [2.000, 2.000]
[3.000, 2.000] -> [2.971, 2.029]
[4.000, 2.000] -> [3.824, 2.176]
```



```
[4.000, 3.000] -> [3.971, 3.029]

[4.000, 4.000] -> [4.000, 4.000]
```

```
# -----------
# User Instructions
#
# Define a function smooth that takes a path as its input
# (with optional parameters for weight_data, weight_smooth)
# and returns a smooth path.
#
# Smoothing should be implemented by iteratively updating
# each entry in newpath until some desired level of accuracy
# is reached. The update should be done according to the
# gradient descent equations given in the previous video:
#
# If your function isn't submitting it is possible that the
# runtime is too long. Try sacrificing accuracy for speed.
# -----------

from math import *

# Don't modify path inside your function.
path = [[0, 0],
        [0, 1],
        [0, 2],
        [1, 2],
        [2, 2],
        [3, 2],
        [4, 2],
        [4, 3],
        [4, 4]]

# ------------------------------------------------
# smooth coordinates
#

def smooth(path, weight_data = 0.5, weight_smooth = 0.1,
           tolerance = 0.000001):
    # Make a deep copy of path into newpath
    newpath = [[0 for col in range(len(path[0]))]
               for row in range(len(path))]
    for i in range(len(path)):
        for j in range(len(path[0])):
            newpath[i][j] = path[i][j]

    #### ENTER CODE BELOW THIS LINE ###

    return newpath # Leave this line for the grader!

# feel free to leave this and the following lines if you want to print.
newpath = smooth(path)

# thank you - EnTerr - for posting this on our discussion forum
for i in range(len(path)):
    print '['+ ', '.join('%.3f'%x for x in path[i]) +'] -> ['+
```

```
          ', '.join('%.3f'%x for x in newpath[i]) +']'
```

Answer:

```
#### ENTER CODE BELOW THIS LINE ###
change = tolerance
while (change >= tolerance):
      change = 0
      for i in range(1,len(path)-1):
            for j in range(len(path[0])):
                  d1 = weight_data * (path[i][j]-newpath[i][j])
                  d2 = weight_smooth * (newpath[i-1][j] +
                                        newpath[i+1][j] -
                                        2 * newpath[i][j])
                 change += abs(d1 + d2)
                 newpath[i][j] += d1 + d2

return newpath # Leave this line for the grader!
```

Question 6 (Zero Data Weight):

Take a minute to think about how the weighting parameters `weight_data` and `weight_smooth` affect the path smoothing. What is the expected output of the path smoothing algorithm if we set `weight_data` to zero? This is different from Question 3 because in this particular implementation, we do not change the first and last points in the path. Feel free to check the result using your code from the previous question!

A. The original path
B. A straight line
C. A single point

Answer:

The answer is B, a straight line, not a single point like we have seen before. The reason for this is that this time we are not changing the first and last points in the path, so the best way to minimize $||y_i - y_{i+1}||^2$ over all values of i is to distribute the points y_i equally along a straight line between the first and last points of the path (remember that the shortest path between two points is a straight line). If we had *not* fixed these two points, the result would indeed be a single point.

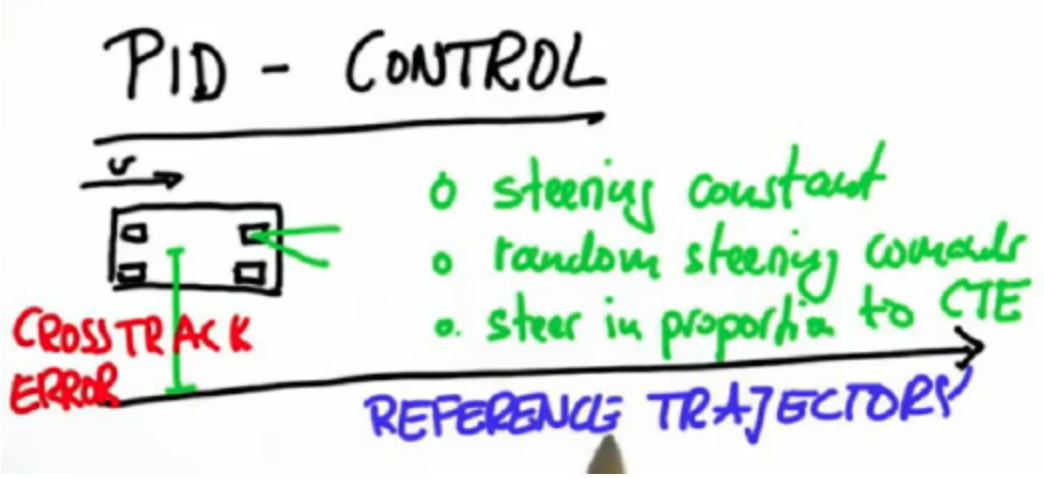
Now, if you do the opposite and set `weight_smooth` to zero, the result will be the original path. To see why, review the discussion in Question 2. You can also try these out on your code and see the results for yourself!

PID Control

This next section of the class talks about PID control. Control theory is a vast field, and someone can actually take many classes just to master the PID controller. For now, you will learn and implement the basics of this controller without worrying too much about the theory behind it, just to get a feeling for how it works. It will be fun, and you'll get the essence of what it means to control a car.

The Google car actually uses a version of this controller that is slightly more attuned to the specifics of the car, but the basic idea behind it remains the same.

Question 7 (PID Control):



Consider the car above, with a steerable front axle and two non-steerable wheels in the back. You want the car to drive along the black line, the reference trajectory. This trajectory can be thought of as a portion of the output of the smoother we just discussed. Assume the car has a fixed forward velocity, but you have the ability to set the steering angle of the car. How would you control the steering so the car follows the reference trajectory? Which of these is best suited to control the car?

A. Constant steering
B. Random steering
C. Steering proportional to the cross-track error (CTE), which is defined as the distance between the reference trajectory and the car

Answer:

The answer is C: you should steer in proportion to the cross-track error. The larger the error, the more you are willing to turn towards the target trajectory. As you get closer to the trajectory, you'll steer less and less.

It is easy to understand that the other two options are really bad. Constant steering would put you on a fixed circle, not a straight line. Random steering would essentially lead to something like a random walk behavior, which is not a good idea either.

This type of steering control is called a P controller, where P stands for "proportional." This is because you steer in proportion to the system error: in this case, the cross-track error. If α is the steering angle, we use the formula

$$\alpha = -\tau_p \, CTE_t.$$

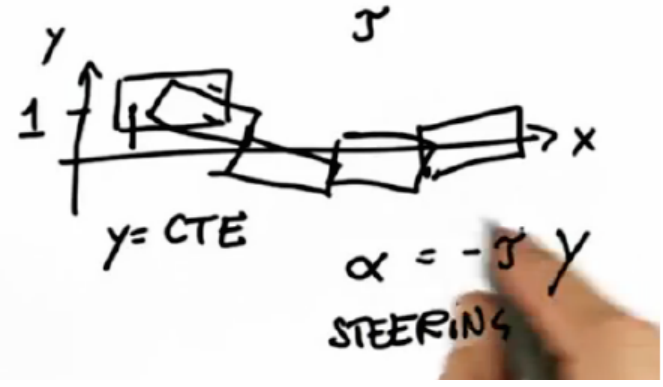
Question 8 (Proportional Control):

Suppose you steer in proportion to the cross-track error. That means that the steering angle is proportional by some factor, τ_p, to the cross-track error. What will happen to the car?

A. It never quite reaches the reference trajectory.
B. It overshoots the reference trajectory.
C. Either can happen.

Answer:

The answer is B: it overshoots. You can see that by noting that although the steering is straight, the car is already pointing into the trajectory when it reaches the trajectory. This happens no matter how small τ is, and because of that the car is forced to overshoot.



So, when applied to a car, a P controller will act like the one in the image: it will slightly overshoot and never quite

attain the correct trajectory. This behavior is called “marginally stable” (or often just “stable”) in the literature. This oscillation might be acceptable, however, if it is small enough for the particular application we are interested in.

Question 9 (Implement P Controller):

Now we will implement the P controller as we described. In the code below, we are given the robot class with the functions you are already familiar with: `init()`, `set()`, `set_noise()`, `move()`, etc. A `steering_drift` variable has also been added, but we won’t use it for now.

Implement the `run()` function, which takes the proportional control parameter (`param`) that governs the response of the steering angle to the cross-track error with the reference trajectory (which will be the x-axis for this problem). The program should initialize a robot at the position [0.0, 1.0, 0.0] with a speed of 1.0 and simulate the robot for 100 steps. This means you will have to set a steering angle value and simulate motion for each one of those steps.

The image above shows what is expected to happen. Considering that the reference trajectory is the x-axis, the cross-track error is given by the y-coordinate of the robot, and the car should oscillate above and below the reference trajectory.

Print the robot coordinates and the controller output for each step using the `print()` method for the robot class.

Below is the proposed implementation. It iterates through all simulation steps. First it gets the crosstrack_error from the robot’s y-coordinate, then it applies the control law to obtain the steering angle, and finally it applies the proper move command for that step.

```
# -----------
# User Instructions
#
# Implement a P controller by running 100 iterations
# of robot motion. The desired trajectory for the
# robot is the x-axis. The steering angle should be set
# by the parameter tau so that:
#
# steering = -param * crosstrack_error
#
# Your code should print output that looks like
# the output shown in the video. That is, at each step:
# print myrobot, steering
#
# Only modify code at the bottom!
# ------------

from math import *
import random
```

```
# --------------------------------------------------
# this is the robot class
#

class robot:

    # --------
    # init:
    #    creates robot and initializes location/orientation to 0, 0, 0
    #

    def __init__(self, length = 20.0):
        self.x = 0.0
        self.y = 0.0
        self.orientation = 0.0
        self.length = length
        self.steering_noise = 0.0
        self.distance_noise = 0.0
        self.steering_drift = 0.0

    # --------
    # set:
    # sets a robot coordinate
    #

    def set(self, new_x, new_y, new_orientation):

        self.x = float(new_x)
        self.y = float(new_y)
        self.orientation = float(new_orientation) % (2.0 * pi)

    # --------
    # set_noise:
    # sets the noise parameters
    #

    def set_noise(self, new_s_noise, new_d_noise):
        # makes it possible to change the noise parameters
        # this is often useful in particle filters
        self.steering_noise = float(new_s_noise)
        self.distance_noise = float(new_d_noise)

    # --------
    # set_steering_drift:
    # sets the systematical steering drift parameter
    #

    def set_steering_drift(self, drift):
        self.steering_drift = drift
```

```
# --------
# move:
#    steering = front wheel steering angle, limited by
#                                              max_steering_angle
#    distance = total distance driven, must be non-negative

def move(self, steering, distance,
         tolerance = 0.001, max_steering_angle = pi / 4.0):

    if steering > max_steering_angle:
        steering = max_steering_angle
    if steering < -max_steering_angle:
        steering = -max_steering_angle
    if distance < 0.0:
        distance = 0.0

    # make a new copy
    res = robot()
    res.length         = self.length
    res.steering_noise = self.steering_noise
    res.distance_noise = self.distance_noise
    res.steering_drift = self.steering_drift

    # apply noise
    steering2 = random.gauss(steering, self.steering_noise)
    distance2 = random.gauss(distance, self.distance_noise)

    # apply steering drift
    steering2 += self.steering_drift

    # Execute motion
    turn = tan(steering2) * distance2 / res.length

    if abs(turn) < tolerance:
        # approximate by straight line motion
        res.x = self.x + (distance2 * cos(self.orientation))
        res.y = self.y + (distance2 * sin(self.orientation))
        res.orientation = (self.orientation + turn) % (2.0 * pi)
    else:
        # approximate bicycle model for motion
        radius = distance2 / turn
        cx = self.x - (sin(self.orientation) * radius)
        cy = self.y + (cos(self.orientation) * radius)
        res.orientation = (self.orientation + turn) % (2.0 * pi)
        res.x = cx + (sin(res.orientation) * radius)
        res.y = cy - (cos(res.orientation) * radius)

    return res

def __repr__(self):
    return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y,
                                            self.orientation)
```

```
# ------------------------------------------------------------------------
# run - does a single control run
#

def run(param=0.2):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    #
    # Add Code Here
    #

run(0.1) # call function with parameter tau of 0.1 and print results
```

Answer:

```
def run(param=0.2):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    for i in range (N):
        crosstrack_error = myrobot.y
        steer = -param * crosstrack_error
        myrobot = myrobot.move(steer, speed)
        print myrobot, steer
```

Question 10 (Oscillations):

Now suppose we modify the parameter τ_p to 0.3. What will happen to the output path?

A. The path will oscillate more quickly.
B. The path will oscillate more slowly.
C. The oscillations will be unaffected.

Answer:

The answer is A: the car's path will oscillate more quickly. You can check this by examining the output of the program from the previous question. For a larger value, the robot reaches a negative y value faster. In fact, it crosses the line at step 13, whereas, for the parameter value 0.1 at that step, the robot's y-position is still 0.6. So, for a smaller value of control, the oscillation is much slower and the cross-track error compensation is much less.

We would want to avoid overshoot in the real world because driving in an oscillating car is dangerous. So the next question is, "is there a way to avoid the overshoot?" The trick for doing so is called PD-control.

In PD-control the steering angle is not only related to the crosstrack error by the gain parameter τ_p but also to the time derivative of the crosstrack error:

$$\alpha = -\tau_p \, CTE_t - \tau_d \, \frac{d}{dt} CTE_t$$

This means that when the car has turned enough to reduce the crosstrack error, it will not just go on trying to reach the x-axis but will notice that it has already reduced the error. The error becomes smaller over time. Eventually, the car counter-steers: that is, it steers up again. This allows the car to gracefully approach its target trajectory, assuming appropriate settings of differential gain versus proportional gain.

How can we compute the temporal derivative of the cross-track error? Since we are working with discrete time steps, we can use the following approximation. At time t:

$$\frac{d}{dt} CTE_t = \frac{CTE_t - CTE_{t-\Delta t}}{\Delta t}.$$

For now we will assume $\Delta t = 1$, so we can simplify the formula to:

$$\frac{d}{dt} CTE_t = CTE_t - CTE_{t-1}.$$

Now the car is controlled in way proportional not only to the crosstrack error but also to the difference between errors on current and previous step.

Question 11 (PD Controller):

Implement a controller that varies the steering direction proportionally according to `param1` and differentially proportionally (i.e., proportionally to the difference) according to `param2`. Observe the results produced by running it for 100 steps with `param1 = 0.2` and `param2 = 3.0`. The sequence of values should converge much more gently to zero. Moreover, as time goes on, the values should go down to 0 and stay at 0. That wasn't achieved with just a proportional controller.

```
# param1 is tau_p and param2 is tau_d
def run(param1=0.2, param2=3.0):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    #
    # ENTER YOUR CODE HERE
    #
```

Answer:

```
def run(param1=0.2, param2=3.0):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    crosstrack_error = myrobot.y
    for i in range(N):
        diff_crosstrack_error = myrobot.y - crosstrack_error
        crosstrack_error = myrobot.y
        steer = - param1 * crosstrack_error
                - param2 * diff_crosstrack_error
        myrobot = myrobot.move(steer, speed)
        print myrobot, steer
```

Systematic bias is a common problem in robotics. An everyday example of systematic bias is when the wheels of a car are out of alignment. In this case, though we might believe the wheels to be aligned perfectly straight ahead, they are actually aligned a little bit at an angle. For human drivers this often isn't that big of a concern: they just steer a little bit stronger.

What happens with a proportional controller? Let's add this line to the run function:

```
myrobot.set_steering_drift(10.0 / 180.0 * pi) # 10 degrees
```

This sets the steering error to be 10 degrees. For this question set the first parameter to 0.2, and the second to 0.0, in order to disable differential control and run your code from the previous question.

Question 12 (Systematic Bias):

What happens to the robot when it has improperly aligned wheels?

A. It remains just as before.
B. It causes large cross-track error.

Answer:

Of course, the answer is B, it causes a large cross-track error.

If you examine the output of your program, you should see that y is consistently between 0.7 and 0.9, which is a big error. Put differently, the robot oscillates with a fairly constant new offset error due to this bias. Even though the bias was in steering, it manifests itself as an increased CTE in the y direction.

Question 13 (Is PD Enough?):

Can the differential term solve the systematic bias problem? Try to use your program to help you answer the question.

Answer:

The correct answer is “no.” If you enter 3.0 for the differential term and run the program, you should still receive large y-values. These values will now converge to 0.87, but it's really far from the expected 0.0.

If you drive a car and your normal steering leads you toward a trajectory far away from the goal, and you notice that over a long period of time you are not getting closer, then you would adjust to the bias by steering more and more toward the goal to compensate for the bias.

How can we recognize a sustained bias? We can do this by adding up the cross-track errors over time: $\sum CTE_t$. The steering formula then becomes

$$\alpha = -\tau_p \, CTE_t - \tau_i \sum CTE_t - \tau_d \frac{d}{dt} CTE_t.$$

To see why this works, consider a scenario with a constant crosstrack error of 0.8. Using the updated equation, the component will increase by 0.8 for each time unit, which will in turn increase your and will eventually correct the robot’s motion.

This is called a PID controller, since it includes a proportional, differential, and now an integral term. Now let’s implement a PID controller in our code.

Question 14 (PID Implementation):

```
# param1 is tau_p, param2 is tau_d, and param3 is tau_i
def run(param1=0.2, param2=3.0, param3=0.004):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    myrobot.set_steering_drift(10.0 / 180.0 * pi) # 10 degree bias
    crosstrack_error = myrobot.y
    int_crosstrack_error = 0
    for i in range(N):
        int_crosstrack_error += crosstrack_error
        diff_crosstrack_error = myrobot.y - crosstrack_error
```

```
        crosstrack_error = myrobot.y
        steer = - param1 * crosstrack_error
                - param2 * diff_crosstrack_error
                - param3 * int_crosstrack_error
        myrobot = myrobot.move(steer, speed)
        print myrobot, steer
    # ENTER YOUR CODE HERE
```

Answer:

```
def run(param1, param2, param3):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    myrobot.set_steering_drift(10.0 / 180.0 * pi) # 10 degree bias
    int_crosstrack_error = 0
    for i in range(N):
        int_crosstrack_error += crosstrack_error
        diff_crosstrack_error = myrobot.y - crosstrack_error
        crosstrack_error = myrobot.y
        steer = - param1 * crosstrack_error
                - param2 * diff_crosstrack_error
                - param3 * int_crosstrack_error
        myrobot = myrobot.move(steer, speed)
        print myrobot, steer
    # ENTER YOUR CODE HERE
```

Parameter Optimization

One important question remains in order for us to effectively implement the PID controller: how can we determine the optimal parameters (also known as control gains) to use? What we will use is an algorithm called “Twiddle” that adjusts parameters one at a time to find an optimal set.

We will start with a parameter vector p = [0, 0, 0]. The vector p will represent our best guess so far for the optimal set of parameters. We’ll also initialize a vector dp = [1, 1, 1], which will represent the potential changes to p that we want to try.

First, we choose a parameter in p (let’s say p[0] for this example, though it could be any of the parameters), and increase that parameter by the corresponding value in dp (for example, if p[0] == 1 and dp[0] == 0.5, then we increase p[0] to 1.5). Then we run the PID planning algorithm using the parameters in p. If the error (that is, the function

$$c_1||x_i - y_i||^2 + c_2||y_i - y_{i+1}||^2$$

from before) is better than the best error we found so far, we keep the new value of p[0] and increase dp[0] by a factor of 1.1. If not, we return p[0] to its original value and *subtract* dp[0] instead of adding. Then rerun the PID planner; if it produces a path with a smaller error, then we keep this value of p[0] and multiply dp[0] by 1.1. If subtracting dp[0] didn’t produce a better error, then we put p[0] back to its original value and multiply dp[0] by 0.9, making the potential change smaller.

We continue this process as long as the sum of the entries in dp does not exceed some given threshold.

Question 15 (Parameter Optimization):

Implement the Twiddle algorithm to find optimal parameters for your PID controller. In the sample code, we iterate over 2*N steps. We also give the controller time to converge by ignoring the crosstrack error over the first N steps.

After only 107 steps in this implementation, Twiddle should converge to the optimal parameter vector [2.923, 10.327, 0.493]. Using this parameter vector, our PID controller starts out with an error around -195, and after only a few dozen iterations, the y error converges to nearly 0.0.

```
def run(params, printflag = False):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0
    err = 0.0
    int_crosstrack_error = 0.0
    N = 100
    myrobot.set_steering_drift(10.0 / 180.0 * pi) # 10 degree drift
    crosstrack_error = myrobot.y
```

```
    for i in range(N * 2):
        diff_crosstrack_error = myrobot.y - crosstrack_error
        crosstrack_error = myrobot.y
        int_crosstrack_error += crosstrack_error

        steer = - params[0] * crosstrack_error  \
            - params[1] * diff_crosstrack_error \
            - int_crosstrack_error * params[2]
        myrobot = myrobot.move(steer, speed)
        if i >= N:
            err += (crosstrack_error ** 2)
        if printflag:
            print myrobot, steer
    return err / float(N)

def twiddle(tol = 0.2): #Make this tolerance bigger if you're timing out!

    # -------------
    # Add code here
    # -------------

    return run(params)
```

Answer:

```
def twiddle(tol = 0.2): #Make this tolerance bigger if you're timing out!
    n_params = 3
    dparams = [1.0 for row in range(n_params)]
    params = [0.0 for row in range(n_params)]
    best_error = run(params)
    n = 0
    while sum(dparams) > tol:
        for i in range(len(params)):
            params[i] += dparams[i]
            err = run(params)
            if err < best_error:
                best_error = err
                dparams[i] *= 1.1
            else:
                params[i] -= 2.0 * dparams[i]
                err = run(params)
                if err < best_error:
                    best_error = err
                    dparams[i] *= 1.1
                else:
                    params[i] += dparams[i]
                    dparams[i] *= 0.9
```

```
    n += 1
    print 'Twiddle #', n, params, ' -> ', best_error
    return run(params)
```

From this you can see how the differential term compensates for steering drift, and the integral term compensates for systematic bias.

Observe the importance of the integral term by setting dparams[2] = 0.0: the error increases from 3.611e-17 to 0.00022. If you also remove the differential term by setting dparams[1] = 0.0, the error increases even more to 0.5529. Even if you comment out your steering drift, the error still remains significant at 0.1038. If you reintroduce the differential term, the error decreases back to 5.7e-11 without the steering drift.

Summary

In this lesson, you learned two important things about robot programming. You learned how to perform smoothing by turning a discrete path into a continuous smooth path. Then you learned how to make a robot follow this smooth path using a PID controller. The combination of a planner, a smoother, and a controller can achieve pretty good results. In fact, this is about the level of sophistication with which Stanley won the DARPA Grand Challenge.