💻🚙 05 - PID: Heading control

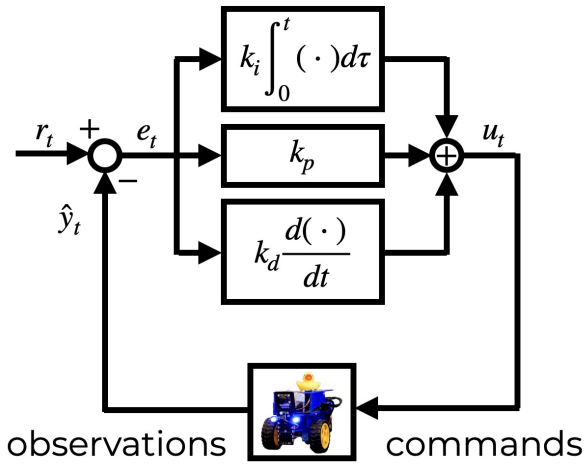
The controller is where the decision making part of a robot's mind.

Controllers output *commands*, which are signals that will be executed by our robot's actuators, with the objective of achieving a certain goal. The inputs to the controller are typically signals that helps define this goal.

For example, in the case of the Duckiebot, the (output) control signals are the linear velocity v and angular velocity ω of the robot. In other words, the controller will decide, at every time instant, at what speed should the Duckiebot go and how much should it steer left or right to, for example, stay in the lane.

The **P**roportional-**I**ntegral-**D**erivative (PID) controller is an example of a feedback controller where the goal is defined in terms of one or more tracking error signals (e_t), which are the input to the controller. The controller will try to bring the error ideally to zero, hence the system output (y_t) to match the reference signal (r_t).

$$e_t = r_t - y_t \to 0 \Rightarrow y_t \to r_t$$



A PID control loop.

Consider a Duckiebot driving in the middle of a road with a constant linear velocity (v_0). In the lane following task our goal is to make sure the robot stays in the middle of the lane by adjusting its angular velocity. Intuitively, whenever the robot deviates from the middle of the lane, we will adjust the angular velocity so the robot turns toward the middle of the lane. The question is: by how much?

The name proportional-integral-derivative comes from the fact that the controller adjusts the control signals proportionally to the error at every time instant, while also considers the integral (i.e., the sum, or accumulation) of the error over time, and the derivative of the error at each time step (i.e., the rate of change of the error over time).

The control command is calculated by considering the combination of these three components:

$$u_t = k_p e(t) + k_i \int_0^t e(\tau) d\tau + k_d \frac{de_t}{dt},$$

where k_p, k_i, and k_d are the proportional, integral, and derivative coefficients of the controller, respectively, and weigh the three terms.

The PID control problem is to determine values for these parameters (e.g., via trial and error) until the closed loop system performs reasonably well. Typically (but it depends strongly on the complexity of the system):

- Increasing k_p will decrease the the time it takes for the system to approach the reference point (i.e., rise time), but at the risk of overshooting the reference. A high k_p leads to an aggressive control.
- Increasing k_d will help to decrease this overshooting by preventing the robot from moving too quickly in a direction that increases the error.
- Increasing k_i will help eliminate the steady-state error (i.e., the remaining error once the system has converged), and compensate for unexpected external disturbances during operation.

For example, we can start by only adjusting k_p while keeping $k_i = 0$ and $k_d = 0$ until the controller is somewhat able to reach the target. We can then fix k_p at this value, and start adjusting k_d. Once we found a good value of k_d, that reduces the oscillations without slowing the system response down too much, we can proceed to adjust k_i as needed to mitigate steady state errors.

While the above approach to tune our PID controller may work in practice, there is **no guarantee** that our robot will be stable (e.g., it may oscillate around or even completely diverge from the reference point).

There are many approaches for designing PID controllers in a formal way and obtaining those guarantees. Talking about them though would require introducing more advanced system modeling methods (e.g., linearization, linear time-invariant systems, state-space representations, transfer functions), as well as analysis methods for dynamic systems (e.g., Bode diagrams, Nyquist plots, root locus) and synthesis techniques (e.g., loop shaping, pole placement), for actually determining the coefficients. There is a vast literature on PID control, and new variations on the theme are published every year

although it is already a very established method. A video introduction to control systems can be found online, e.g., in the Control Systems I course held at ETH Zurich.

The popularity of PID control is due to the fact, amongst other factors, that these formal methods are not necessary to reach a satisfactory outcome. "Rule of thumb" methods such as the Ziegler-Nichols can help us to tune our PID controller, but trail and error (or, "synthesis by iterations", a. k. a. "tweak until death") works, too!

Let's go ahead and design a PID controller!

Let's get started!

In this activity we will design a proportional, integrative, derivative (PID) controller to regulate the heading (θ_t) of a Duckiebot, while it is driving at a constant linear speed (v_0).

In reference to the PID control diagram above, we consider the following:

- $r_t = \theta_{ref,t} = \theta_{ref}$: the reference signal is a constant angle, expressed in radians
- $\hat{y}_t = \hat{\theta}_t$: the controlled variable is the heading of the Duckiebot. This controller will estimate the Duckiebot's heading based on the odometry model deigned in the odometry activity, so make sure you are happy about that before proceeding!
- $u_t = [v_0, \omega]^T$: the output of the controller, and input to the plant, will be two variables. The first will be the linear speed of the robot, which we will assume constant throughout this activity. The variable we will control is the angular rate ($\omega_t = \dot{\theta}_t = \frac{d\theta_t}{dt}$).

The objective of this activity is to determine the values of k_p, k_i, k_d such that we obtain "good" tracking performances.

Before coding away, let us review how computers actually calculate integrals and derivatives.

Calculating integrals in discrete time

The theory tells us that one of the component signals of the PID controller is proportional to the integral of the error over time:

$$e_{int}(t) = k_p \int_0^t e(\tau) d\tau$$

An integral is an *infinite* (pick the biggest number you can think of, *infinite* is strictly more than that) **sum** of *infinitesimal* (pick a positive number, the smallest you can think of, *infinitesimal* is stricly smaller than that) bits, a concept that assumes *continuity* of

time. Continuity means that, for any two instant in time you can think about, arbitrarily close to each other, there will always be *infinite* other instant between them.

But all computers in the world, including the one running on the Duckiebot, don't know how to do infinite or infinitesimal. "Think of the smallest" and "think of the biggest" are qualitative concepts that only humans can grasp.

Computers can do finite (instead of infinite or infinitesimal) though. Time, for a robot, is a sequence of instants. But when you make a computer take two consecutive instant, there is nothing in between. Computers have notion of *discrete* time, not continuous time.

The immediate repercussion of this fundamental limitation of computers is that we cannot really calculate integrals.

What we can do, is have them calculate a *finite sum* to approximate the actual integral.

$$e_{int}(t) = k_p \int_0^t e(\tau) d\tau \simeq \sum_{i=0}^{k} e_i \Delta t = (e_0 + e_1 + \cdots + e_{k-1} + e_k)\Delta t$$

Where in the above approximation we assumed for simplicity that all time instants are equally spaced by a constant *time step* Δt.

So how do we implment this integral component on our Duckiebots? We can note that:

$$e_{int,k} = (e_0 + e_1 + \cdots + e_{k-1} + e_k)\Delta t = (e_0 + e_1 + \cdots + e_{k-1})\Delta t + e_k \Delta t = e_{int,k-1} +$$

In [ ]:
```
# Run and do not edit this magic cell.
# It helps getting things to work throughout the Jupyter notebook - in parti

%load_ext autoreload
%autoreload 2
```

In [ ]:
```
import numpy as np

# How to calculate the integral term of the tracking error?

e_int_last = 0 # previous integral error, starts at 0
k = 10 # example current time step
e = np.ones(k) # assume the error is constant at every time instant (it will
dt = 0.1 # example sampling time (seconds)

# initiate the error
e_int_current = 0

for i in range (0, k):
    ei = e[i] # error at this time instant
    e_int_current = e_int_last + ei * dt # integral error at the previous in
    e_int_last = e_int_current # the present becomes the past for the future
```

```
print(f"The finite sum of the tracking error is {e_int_current}")
```

```
The finite sum of the tracking error is 0.9999999999999999
```

Calculating derivatives in discrete time

Derivates are the tool math uses to measure change. As you might imagine, derivatives are tremendously important operations as so many things in the universe change in some way. One could argue that derivatives are the most important of operators, and in fact open the doors to a whole field of math called *calculus*.

Time derivatives are defined as *the ratio of the difference of a function, evaluated at two infinitesimaly close to each other instants, and the time difference between them*.

In fancy words, derivates are *limits of the incremental ratio* functions.

$$\dot{e}_t = \frac{de_t}{dt} = \lim_{dt \to 0} \frac{e_{t+dt} - e_t}{dt}$$

Without getting in the details, that $\lim$ part means that the dt time difference is a very, very, small (positive) number. How small? The smaller you can imagine it the better you understood the derivative operation. *At the limit, it's zero*.

But computers cannot do this "at the limit", because it is a qualitative leap of the mind. This is a very human thing to do. Computer only know finite time. So they need to know how much is actually dt equal to (0.001? 0.0000001? or maybe even smaller that that?).

Computer cannot do derivatives, they can do *finite differences*, which approximate derivatives. There are many different formulations of finite differences, the simplest is called "Euler backwards" method. It basically approximates the derivative as a difference of the current and the previous function evaluation, divided by the time step.

$$\frac{de_t}{dt} \simeq e_{der,k} = \frac{e_k - e_{k-1}}{dt}.$$

Let's try it out!

In [ ]:
```
e_der = [0.0] # to initialize the derivative term

for i in range (1, k): # note how we start from 1, we can't calculate the de
    e_current = e[i] # error at this time instant
    e_previous = e[i-1] # error at the previous time instant
    e_der_ = (e_previous - e_current)/dt
    e_der.append(e_der_)

print(f"The finite difference of the tracking error is {e_der}")
```

```
The finite difference of the tracking error is [0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0]
```

We now have all the tools to write our first PID controller!

Remember that the objective is controlling the heading of the robot, by computing an angular speed of the robot, while the linear speed is held constant for simplicity.

Step 1: Implement the PID controller

Implement the function `PIDController` inside the file pid_controller.py.

Here are some tips on how to proceed.

Tip 1: start from the end

Keep your focus on the final objective: the function must output (at least) the control signals, i.e., v_0 and ω.

How do we calculate them?

- The velocity is constant, and it will be an input to the function. Straightforward.
- The angular speed is given by the PID equation we saw above.

Tip 2: proceed backwards

The PID equation has three terms. Each with an arbitrary coefficient. These will be used for tuning the controller. Go ahead and define them.

What else is needed? Ah, the error terms.

Tip 3: technical needs

We saw that there are some technical limitation on how computers calculate integrals and derivatives. Adjust your code to account for these limitations.

Tip 4: test and iterate

Seldomly things work out at the first shot. Test your code, *give meaning to the values it produces*, and fix where needed. Try to avoid hacks, i.e., solutions that work but do not fit with the theory / you do not understand why they should be there.

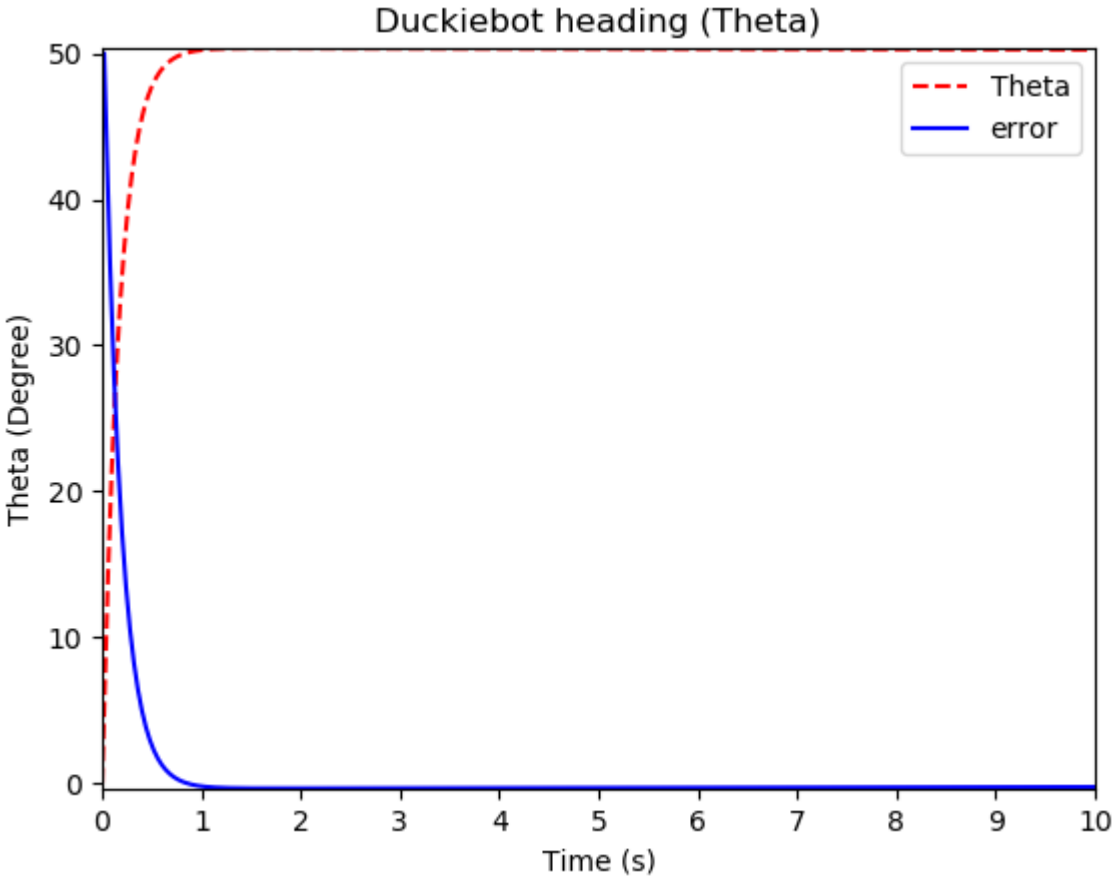
A complete solution is available for this activity in the `solutions_PID_controller.ipynb` file, but try to give it a shot yourself before checking!

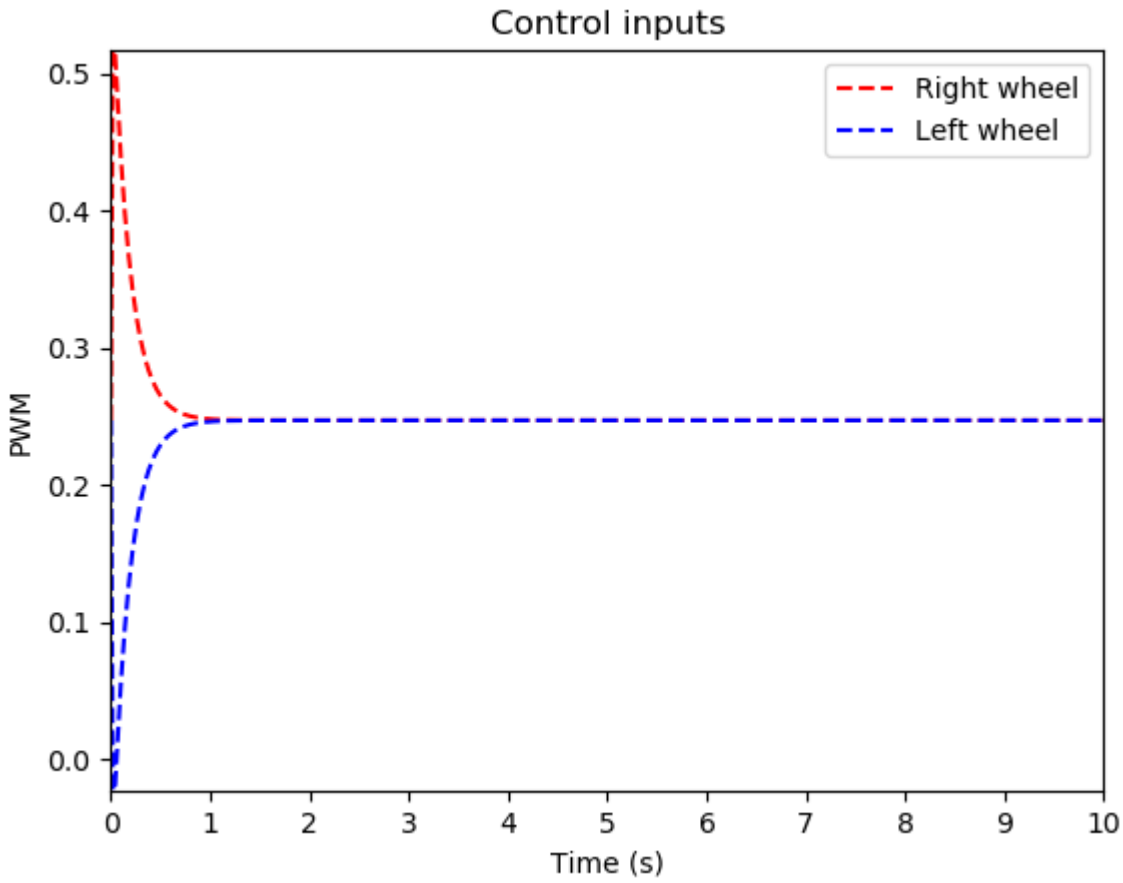
Sanity check

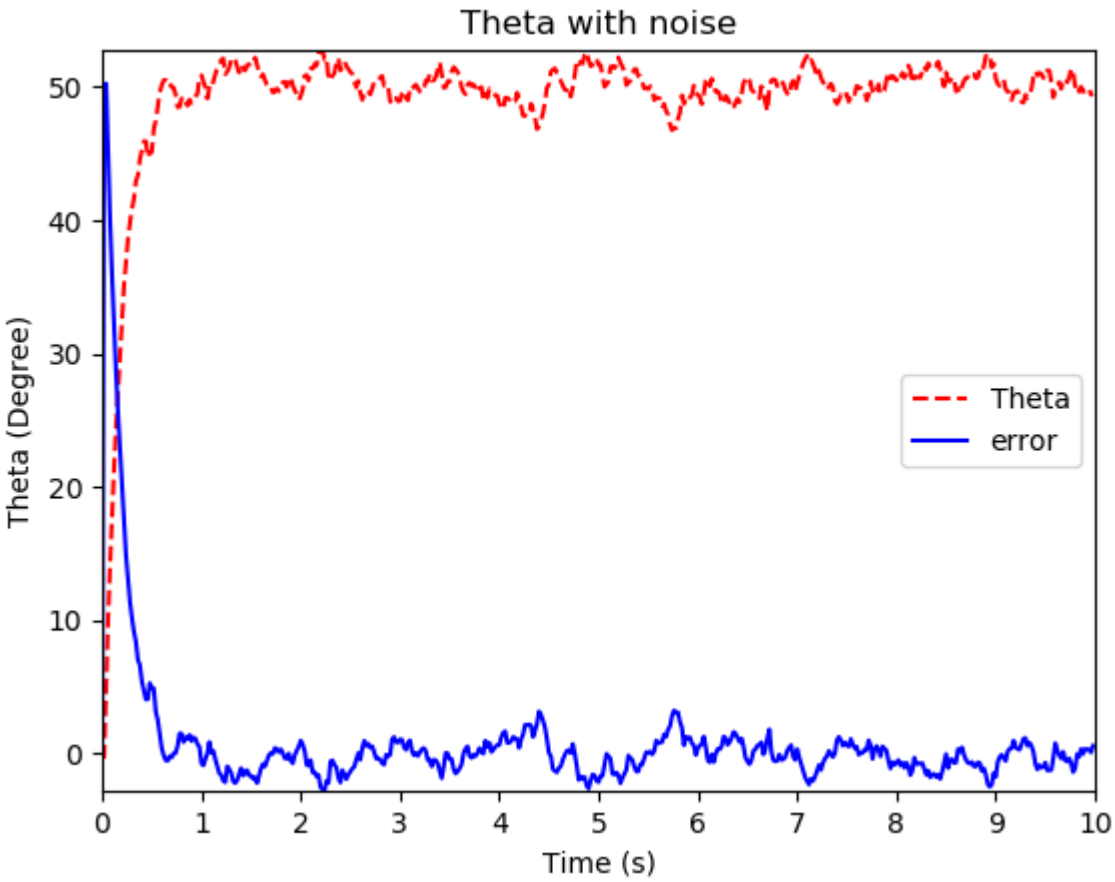
With this unit test you can do a quick test of your controller defined by the coefficients above. You can redefine the kinematic parameters as you wish to play around with various factors. We simulate a case without and with synthetic noise added to the measurements.

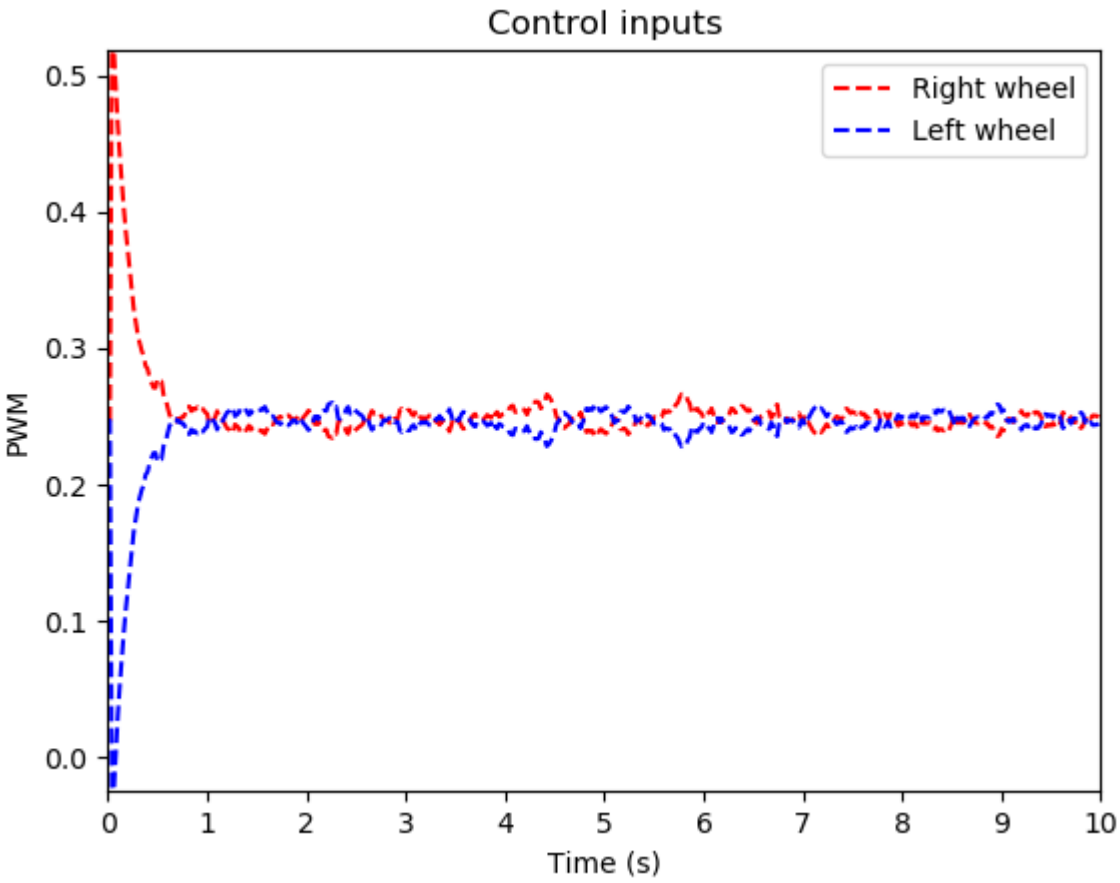
You can find the definition of this test in the unit_test function.

In [ ]:
```
from tests.unit_test import UnitTestHeadingPID

from solution.pid_controller import PIDController

import numpy as np

# This is for quick testing purposes only - you can try different values of
v_test = 0.2
R_test = 0.018 # m
baseline_test = 0.1 # m
gain_test = 0.6
trim_test = 0
theta_ref= np.deg2rad(50) # in rad

# Sanity check (not a faithful representation of the actual behavior, given
unit_test = UnitTestHeadingPID(R_test, baseline_test, v_test, theta_ref, gai
unit_test.test()
```




Control inputs
Right wheel
Left wheel
PWM
0.0
0.1
0.2
0.3
0.4
0.5
0
1
2
3
4
5
6
7
8
9
10
Time (s)


Theta with noise
Theta
error
Theta (Degree)
0
10
20
30
40
50
0
1
2
3
4
5
6
7
8
9
10
Time (s)



Step 2: Test the PID controller

Before running your PID controller on a real-world Duckiebot, which can cause real-world crashes, you first need to start in the simulator. Explore the limitations of your controller and tune the parameters for optimal behavior. This is the version of your controller that you will submit for evaluation in the next notebook.

When moving from the simulator to your Duckiebot, there are two main considerations:

1. Safety - Be sure to run your Duckiebot in a space where it cannot fall off of the edge of a table, go down stairs, or collide with anything fragile.

2. Behavior Changes - Your Duckiebot will perform differently from the simulated Duckiebot. The simulated hardware model cannot exactly match your specifically assembled Duckiebot, and the physical environment introduces many types of error not present in simulation.

Hint: Be sure to save your tuned controller coefficients that perform best in simulation before changing them to optimize your solution for your Duckiebot. The following values should be a good place to start from for each type of testing.

	Simulator	Duckiebot
kp	5	15
ki	0.2	1
kd	0.1	0.2

It is always a good idea to commit the various versions of your solutions to GitHub as you work on tuning.

💻 Test the controller against the simulator

1. Open a terminal on your computer, navigate to the `/duckietown-lx/modcon` directory, and type

   ```
   dts code build
   ```
2. Wait for the build to finish, then type:

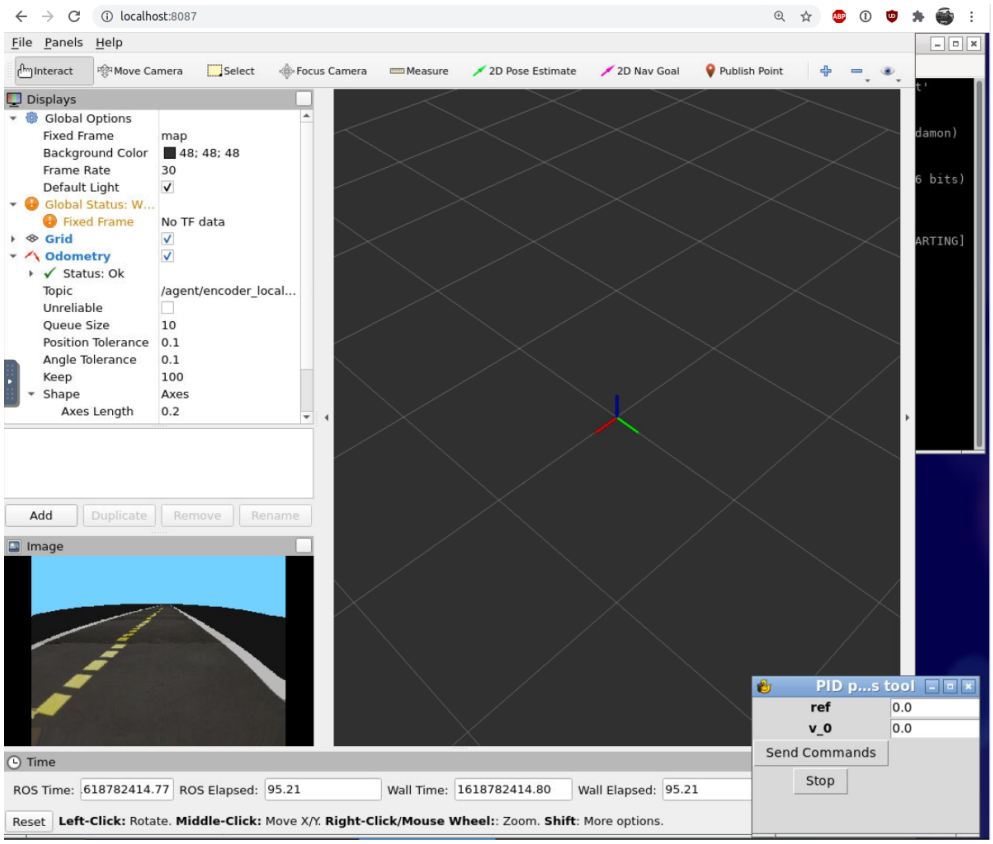
   ```
   dts code workbench --sim
   ```
3. Open VNC on you browser and click on the `PID` icon on your desktop. You will see the following open (it might take ~30second or more, depending on the specifications of your computer):

   - A preconfigured RVIZ: to visualize performance
   - an LX terminal: `Ctrl-C` here to return to VNC.
   - an interaction window with fields `ref` and and `v_0`, and buttons `Send Commands` and `Stop`.

In the RVIZ terminal you should see what the robot sees. Nothing should be moving. You will see some debugging data in the terminal running on your computer where you launched the activity from.

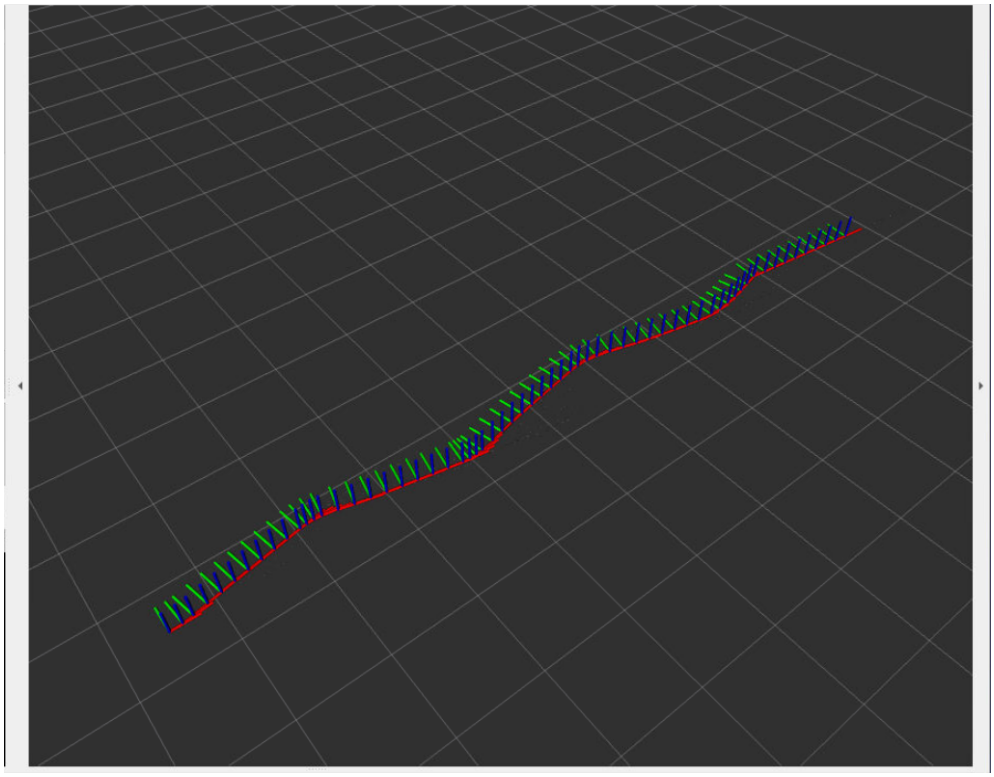
To initialize the testing of your controller, you will have to insert values for `ref` (in **degrees**) and `v_0` (between -1 and 1) and press on `Send Commands`. The controller you designed above will then start computing the ω and the robot will start driving.

Good example values for this activity are $ref = 10$ and $v_0 = 0.2$, but you can play with these parameters at runtime and see how your PID controller performs. Mind that you will go wasted if you crash, or exit the map. In this case, simply start the instance again - this is why we test in simulation!



The PID activity interface on VNC.

At any point you can press the `Stop` button to "emergency break" your Duckiebot.



Switching reference between $\pm 10^{\circ}$ at increasing velocities.

To test different solutions, change the `PIDController()` function above, save this file (`Ctrl-S`) and re-run the activity with `dts code workbench --sim`. Remember that the odometry functions you wrote before play an important role here too as the tracking errors are computed based on the odometry.

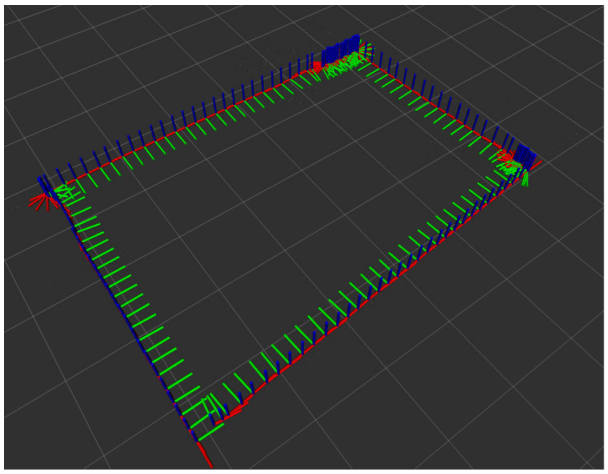
🚙 Test the controller on your Duckiebot

1. Open a terminal on your computer, navigate to the `/duckietown-lx/modcon` directory, and type

   ```
   dts code build
   ```
2. Wait for the build to finish, then type:

   ```
   dts code workbench -b ROBOTNAME
   ```
3. Follow the same instructions at point 3. of the simulation case.

Note: we suggest to start at very slow speeds with the physical Duckiebot, and get a hang of the interface first. Going wasted in simulation is just a matter of re-setting the instance. Having the Duckiebot go wasted in the physical world might be significantly more time consuming!



Duckiebot heading control with:

$$v_0 = 1, \theta_{ref} = [0^\circ, 90^\circ, 180^\circ, 270^\circ, 0^\circ]$$

To test different solutions, change the `PIDController()` function above, save this file (`Ctrl-S`) and re-run the activity with `dts code workbench -b ROBOTNAME`.

Note: Remember that the odometry functions you wrote before play an important role here too as the tracking errors are computed based on the odometry.

💡 Reflecting on the experience

After having played around with the PID controller, maybe testing different PID coefficients along with Duckiebot fixed speeds and distances to the reference

trajectories, you might have noticed a few things:

- Need for tuning: There is a coupling between stability, performance, coefficients and magnitude of the tracking error. A set of coefficients that worked "well" for a given linear speed and distance from the reference, might not work as well with a different set of values. PID control *can* produce good results, but it requires tuning to specific cases.

- Input saturations: real robots have saturations (e.g. `omega_max` in the kinematics parameters, which has been set in previous activities). This means that occasionally having the controller act more aggressively will not change the system's response, as there are other real world constraints. Saturations might represent physical limitations of hardware (intuitively, the higher the value of the input, the higher "energy" it will take to actuate that command. Energy is not free, nor infinitely available.), or overarching safety constraints that have been put in place to prevent dangerous things from happening. When inputs saturate, errors (especially the integral one) might continue building up, causing a situation that will break the logic of the PID controller. For this reason, typically `anti wind-up` mechanisms are implemented to prevent either input saturation, of the integral error building up too much. There are many fancy `anti wind-up` strategies; we implemented a simple one in the controller above.

- You never get perfect tracking, even with a very well tuned PID controller. Why?

- The robot doesn't have the slightest idea of what is going on around it. This is beacause we are using interoceptive sensors (wheel encoders) to create the robot's belief. The **real** state of the robot is unknown to it. Moreover, lack of exteroceptive sensors (e.g., a camera) makes the robot unable to "anchor" its belief to some real-world feature, and perform sanity checks. You might have noticed that a Duckiebot will *not* take that turn in your Duckietown autonomously, and that's because it has no notion of a map. For this very reason we will introduce cameras and computer vision in the future modules.