Hands-on Robotics Development using Duckietown

This courses teaches the practicalities of programming robots. At the end, you will know how to write and deploy simple *agents* on your Duckiebot.

Contents

Part A
[RH1] Connecting and operating a Duckiebot

This part will take you through the most basic hardware and software skills you need in Duckietown. You will start from building your Duckiebot and learning the most frequently used terminal commands and go all the way to running your first Duckiebot demos!

Contents

UNIT A-1
Assembly duckumentation

We have prepared detailed instructions on how to build your Duckiebot, and, if you need, a whole Duckietown! Here, we will guide you to the relevant parts of the book that contain the specific instructions. Once you are done, you can continue with the next module.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Hardware
Results: Know how to build Duckiebots and Duckietowns.
Results: Know where to ask for help.

Contents

1.1. Assembling the Duckiebot

The content of the Duckiebox including a detailed set of instructions can be found in the assembly instructions (see exercise below). It is advisable to read through our hardware preliminaries (unknown ref opmanual_duckiebot/db-opmanual-hw-prel)

warning next (1 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/db-opmanual-hw-prel'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

section before you get your hands on your own Duckiebot.

The assembly instructions as well as the hardware preliminaries are part of the extensive documentation on Duckietown, which we refer to as the “Duckumentation”. The Duckumentation is an open-source set of documents that explains everything you need in order to find your way around the Duckietown universe.

If you cannot find the answer to a specific question you have, you can join our international Slack workspace. There you can ask the community about anything. When you sign up, please add your affiliation. It is always a pleasure to see Duckietown spreading around the world, and we are curious to find out where our new members come from.

If you run into any issues during the assembly, there are different ways to find help.

First, you can look at the FAQ sections that are on some pages of the Duckumentation. If this does not help you and you need further assistance, let us know via Slack.

Exercise 1. Duckiebot assembly.

Assemble the hardware of your Duckiebot according to the assembly instructions. Based on the Duckiebot model you have, please choose one of the following instruction guides:

`DB18` (unknown ref opmanual_duckiebot/assembling-duckiebot-db18)

previous **warning** next (2 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/assembling-duckiebot-db18'.
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

`DB19` (unknown ref opmanual_duckiebot/assembling-duckiebot-db19)

previous **warning** next (3 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/assembling-duckiebot-db19'.
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

`DB-beta` (unknown ref opmanual_duckiebot/assembling-duckiebot-db-beta)

previous **warning** next (4 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/assembling-duckiebot-db-beta'.
```

Location not known more precisely.
Created by function `n/a` in module `n/a`.

(only ETHZ students taking AMoD 2020 course)

1.2. Assembling your Duckietown

Now it is time to assemble your city. You can find useful instruction here (unknown ref opmanual_duckietown/dt-ops-assembly)

previous **warning** next (5 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckietown/dt-ops-assembly'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. Make sure you check out the city appearance specifications (unknown ref opmanual_duckietown/dt-ops-appearance-specifications)

previous **warning** next (6 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckietown/dt-ops-appearance-specifications'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

too!

UNIT A-2
Terminal basics

Working over the terminal is a skill that every roboticist-to-be needs to acquire. It enables you to work on remote agents or computers without the need for a graphical user interface (GUI) and lets you work very efficiently. Once you get the hang of it, you will find out for yourself how it can make your life easier.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (7 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Duckietown account (unknown ref opmanual_duckiebot/dt-account)

previous **warning** next (8 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/dt-account'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Results: Know how to use a terminal

Contents

2.1. Using a terminal

If you are completely new to working with a terminal, often also called "console" or "command line", a beginners tutorial can be found here. It makes sense to get to know the terminal very well, as this will save you a lot of time along the way.

If you are looking for an extensive list of commands that can be used from the terminal, this is the place to look at.

2.2. Using the Duckietown Shell

The Duckietown Shell, or `dts` for short, is a pure Python, easily distributable (few dependencies) utility for Duckietown.

The idea is that most of the functionalities are implemented as Docker containers, and `dts` provides a nice interface for that, so that users should not type a very long docker run command line. These functionalities range from calibrating your Duckiebot and running demos to building the duckumentation and submitting and evaluating for AI-DO. You will find the commands that you need along the way during the next steps.

If you followed all the steps in the laptop setup (unknown ref opmanual_duckiebot/laptop-set-up)

previous **warning** next (9 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

, you already installed `dts`. If not, now is the time to go back and do it.

UNIT A-3

Duckiebot Setup

Major efforts were made to make sure that the setup of your Duckiebot is as comfortable as possible for you. We created a set of instructions for initialization and calibration through which we will guide you here.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: an assembled Duckiebot.

Results: A Duckiebot that is ready to operate in Duckietown.

Contents

3.1. Initialization

First of all, you have to flash your SD card. Here you have the possibility to give your duckiebot a name and choose what network to connect to. We experienced people having issues when they called their Duckiebot `duckiebot`, so make sure to find a creative name that is different from that.

Follow the initialization instructions here (unknown ref opmanual_duckiebot/setup-duckiebot)

previous **warning** next (10 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opmanual_duckiebot/setup-duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

3.2. Make your Duckiebot move

As soon as you finished the initialization part successfully, it is time to make your Duckiebot move. Follow the instructions here (unknown ref opmanual_duckiebot/rc-control)

previous **warning** next (11 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/rc-control'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

to find out how you can maneuver your Duckiebot using your computer keyboard. This is also the moment to check whether you did a good job at wiring your motors. If your Duckiebot does not behave as you tell him to, this is probably due to the fact that some wires are crossed.

> Note: If this is the first time that you try to make your Duckiebot move, give it some time. It might take some time until the joystick pops up on your screen.

3.3. See what your Duckiebot sees

There is another key component missing now: the image stream from the camera. To find its way around in the city, a Duckiebot needs to be aware of what is going on around him and where he is allowed to drive and where not. To see the image stream from your Duckiebot, follow the instructions here (unknown ref opmanual_duckiebot/read-camera-data)

previous **warning** next (12 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/read-camera-data'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

3.4. Calibration

As with every real-world system, the hardware of the Duckiebot is always a little different. The "same" cameras or motors that you can buy off the shelf will never be exactly the same. Additionally, the camera might have been mounted in a slightly different orientation than it was supposed to. But don't worry, this is what we are going to take care of in this step.

We have two calibration procedures for the Duckiebot: one for the camera and one for the motors.

1) Camera calibration

The camera calibration procedure consists of two parts: the first one is the intrinsic camera calibration. It accounts for the differences between each camera and is therefore only dependent on the camera itself. If you did the intrinsic calibration, make sure to not play around with the lens of the camera anymore as it will invalidate the intrinsic calibration.

The second part is the extrinsic camera calibration. It accounts for the positioning of the camera relative to its environment (i.e. how you mount it on the Duckiebot). So if you mounted the camera at a slight angle with respect to the driving direction this part accounts for it.

Follow the instructions here (unknown ref opmanual_duckiebot/camera-calib)

previous **warning** next (13 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/camera-calib'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

to calibrate the camera of your Duckiebot.

For more detailed background information check out this link.

> **Exercise 2. Calibration.**
> During the camera calibration, the Duckiebot will run an automatic verification on the camera calibration. Check if the projection of the street on the actual picture fits. If it doesn't you have to redo the extrinsic calibration.

2) Wheel calibration

The Duckiebot uses a differential drive. Going forward in a straight line therefore depends on the motors turning at the exact same speed. As in reality every motor is slightly different, we have to account for these imprecisions using a wheel calibration procedure. In Duckietown we are currently using a gain-trim approach for that.

Follow the instructions here (unknown ref opmanual_duckiebot/wheel-calibration)

previous **warning** next (14 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/wheel-calibration'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

to run through the calibration procedure with your Duckiebot and help him drive straight.

Unit A-4
Networking basics

Networking is extremely vital in Duckietown. And we don't mean the networking events where duckies socialize (these are pretty fun), but rather the computer networks between the bots, your computers and the rest of the Duckietown equipment. These networks allow us to do some pretty cool stuff, like controlling your Duckiebot from your laptop or creating a centralized observation center that combines the video streams of all watchtowers. Networking's usefulness is only comparable with its complexity. Indeed, this is often the source of most confusion and problems for Duckietown newbies. That is why we will try to clarify as many things as we can from the very beginning.

Knowledge and activity graph

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (15 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

Requires: Duckiebot initialization (unknown ref opmanual_duckiebot/setup-duckiebot)

previous **warning** next (16 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/setup-duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

Results: Fundamental networking knowledge.

Contents

4.1. Why do we care about networking in the first place?

Your Duckiebot, just like your computer or your phone, is a network device and you connect to it through the network. You probably want to control it without having to attach a screen, a keyboard and a mouse to it, that would defeat the whole “autonomy” goal. In more complex projects, one computer can also be used to control dozens of devices at a time. And in one of the most challenging undertakings that we have attempted so far, we connect 50+ watchtowers into a single mega-hive. All this is enabled by smartly configured computer networks!

4.2. How do computer networks work?

A local network is setup with a *router* at the center, that allows all devices that connect to it to find each other and communicate. The role of the router is to direct (route) packages from a sender to a receiver. In big networks you cannot physically connect all devices to a single router. In this case, you can use *switches* to combine the network traffic from a number of devices onto a single connection to a router. The router must know which device is which and where to find it. To facilitate their communication, the router and the rest of the devices use *IP* and *MAC* addresses.

The MAC address is related to your hardware itself, to your computer (or more accurately, to the network interface). This means that it remains the same even if you move to the other end of the world and connect to a different network. If your computer supports both a WiFi and an Ethernet connection, then each one has a different MAC address. The MAC address is of the form: `0d:12:2c:a7:0d:27`, with each symbol being a hexadecimal (`0-9 + a-f`). More importantly, MAC addresses are unique: there is no other device in the world with the same MAC address as the WiFi adapter in your laptop. You can consider it as a citizen number: it is unique personal identifier. That makes MAC addresses extremely useful for routing messages reliably.

While MAC addresses have the benefit of stability, they are very clumsy to work with, imagine that every time you want to send a letter to your friend you need to write down their citizen number. And also imagine you are the mailman: it is very different to deliver mail if you don’t know where the person lives. Computers use IP addresses to handle these problems.

The IP address of a device is relative to the network it lives in. It is a sequence of numbers that are uniquely mapped to devices inside the network. It is coded on 32 bits. Most home networks use the range of IP from `192.168.1.1` to `192.168.1.255`, so you may have seen the numbers before. The structure of the IP address shows the hierarchical nature of the network architecture. This address will change as soon as you change network, and it is assigned by the network administrator. Typically this is handled by a DHCP server which, in most home networks is part of the router. In a local network, all addresses use the same subnetwork, which means that the first 24 bits of it are the same. If my IP is `192.168.1.23`, then my subnetwork is `192.168.1.`*xyz*. This makes it easy to determine which devices are on the same local network as me, as then the router can directly deliver my messages. If you are trying to connect to a device outside your local network (e.g., on the Internet), the router will need to find a way to deliver the message to it.

This concept is actually quite important. Your router will give you the address of any device on your *local* network, such that you can connect to it, but does not work for resources on the Internet, for example, `docs.duckietown.org`. Therefore, instead, it acts as an intermediary between your device and the Internet. The technical term for that is *gateway*. The router will mask any request that comes from you as if it comes from the router itself, and once it gets a reply from the remote server, it forwards that back to your device.

Even though using IP addresses is very convenient for computers, humans do not handle them that well. They change from time to time and are hard to memorize. Instead, we prefer to name our devices with memorable names such as `quackabot` or `duckiecar`. These names are called *hostnames* and you should have picked one for your Duckiebot when you initialized it. In Duckietown, we mostly use the hostnames for connecting to devices. However, the ability to find a device by hostname is non-standard and requires a protocol called *multicast DNS* (mDNS).

> Note: This mDNS protocol works by default on most home or office networks, but is blocked on large corporate networks like the ones of universities. If you have issues connecting to your Duckiebot thourgh the hostname, that is the most likely reason and you should first check with your network provider if mDNS is indeed blocked.

> **Exercise 3. Network utilities.**
>
> Now we will discuss some useful tools that can help understand the network on which you are.
>
> There is nothing simpler than finding your hostname: simply type `hostname` in a terminal. Now, make sure you are connected to a network first.
>
> We can use the `ifconfig` command to find some properties of this network. Open a terminal and type the command `ifconfig`. You might be missing the package that provides this command. If that is the case, install it and try again.
>
> The `ifconfig` command outputs a few paragraphs, one for each network interface. You typically will find one called something like `wlan0` (your wireless interface) and another one called `eth0` (your Ethernet interface). Look at the one through which you are connected at the moment. After the keyword `inet` you should see your IP address and after the keyword `ether` or `HWadress` you should get the MAC address of this interface.
>
> Can you determine what is your sub-network? How many devices can you put on this sub-network?

Now that you know what your network is, it is time to explore the devices on it. There are many ways to do this. If you know about a device that should be connected, like your Duckiebot, then you can directly try to find it. To do so, you can try to ping it. This will just “poke” the device to see if it is on the network and it is responsive to the poking. You can ping by IP address and a hostname. Pinging by IP address always works if a device is connected to the network. Pinging by hostname requries that mDNS is enabled, therefore if that fails it could mean that either your device is not connected, or that the mDNS traffic is being blocked on your network.

> **Exercise 4. Ping.**
>
> Open a terminal. Run `ping` *hostname*, where `hostname` is your Duckiebot's hostname. Does it work? What is the output? Now try `ping` *hostname*`.local` instead. Does this work? For the router to find a device with its hostname, it needs to know that the hostname is in the local network, not somewhere else on internet. In contrast, try to ping a server outside of the local network: `ping google.com`. You can stop pinging the Duckiebot by pressing `CTRL-C`.
>
> Now, when you pinged your Duckiebot, did you notice that there was an IP address in the output? Is it yours? No! It is the IP of the Duckiebot! You can now use this IP address and try pinging with it. Do you need to add the `.local` this time? Can you figure out why?
>
> This part will be very important for a lot of the things you will do in Duckietown. When a command involving your Duckiebot doesn't work, the first thing to try is to ping it and make sure it is still accessible.

> **Exercise 5. NMap.**
>
> We can now investigate what is on our network by using one of the many network mapping tools that exist out there. Keep in mind that depending on the network and the devices on it, you might not be able to see every device and every parameter.
>
> Since you know your IP address, you also know your sub-network. Using the tool `nmap`, we are going to search the whole sub-network. Try to run `nmap -sP` *YOUR IP*`/24` in a terminal. The `/24` part tells `nmap` to keep the 24 first bits the same in its search. If you don't put it, then nmap will search the complete space of address (which are the monstrous 2^32 addresses).
>
> The output should give you the list of all devices connected to your network, with their IP addresses and most of the time their hostnames. This way, you found your hostname and its IP, as well as other potentially present Duckiebots or computers.

4.3. Connecting to your Duckiebot

Now that we know what our local network is and how it works, we can this information to gain access to Duckiebots. The industry standard way of connecting to remote devices is a protocol known as *SSH* (Secure SHell). Then name describes it quite well: just in the same way that you can run shell commands on your computer in the terminal you can run shell commands, in a secure way, on a remote device. In this case, the remote device will be your Duckiebot.

> **Exercise 6. SSH.**
>
> Let's connect to our Duckiebot via SSH. Open a terminal and type `ssh` *username*`@`*hostname*`.local`. The username and hostname should be the ones you sup-

plied when you flashed your card. If you didn't set a username, then it should be the default value of `duckie`. If you are prompted to enter a password, again use the one you set when flashing, or if you didn't use the default `quackquack` password.

Now your terminal is not in your computer anymore but on the Duckiebot. Did the text before the place where you can enter you command change? Why? What do these things there mean?

You should now be in a shell in the Duckiebot. Try to move around with terminal commands like `cd` and `ls`, as explained in the terminal basics. Verify that these are not the directories and files you find on your computer. They actually are the ones on your robot.

Repeating the steps from one of the previous exercises, find the MAC address of your Duckiebot.

Once you are ready, you can exit the session on the Duckiebot and return to your computer by simply typing `exit` or by pressing `CTRL+D`.

You can connect to your bot without having to type a password (maybe that was already the case). This is done by using SSH keys (a *private* and a *public* one). You don't know this yet, but when you flashed the SD card on your computer, it added an SSH key to your computer and to the Duckiebot. With this, the Duckiebot recognizes your computer and won't ask for a password. On your computer, the key is in `~/.ssh`, and it is called `DT18_key_00`. If you in fact try to `ssh` in a Duckiebot on the network that was not flashed on your computer, you will have to know the password.

Exercise 7. SSH keys.

Open a new terminal and navigate to `~/.ssh` and open the file named `config`. What is in there? It is a list of know agents mapped with the key to use. When you run `ssh` *hostname* ssh will directly use the key and the provided Linux username (`duckie` by default).

UNIT A-5

Docker basics

If you are frequent user of Python, you have probably experienced that making your projects portable can sometimes be quite difficult. Your code might work only on a specific version of Python and requires specific versions of some particular libraries. But how can you make sure that the users of your code have the same installed? Thankfully, the Python community has develped wonderful tools to manage that, such as virtual environments and PyPI. Unfortunately, these tools stop short of extending their convenice outside the Python world. What about your parameters, libraries, packages written in different languages, binary executables, system configurations, and anything else that your code might need to run correctly? How do you make sure your user has all of this setup correctly? And what if you want this to work accross different hardware and operating systems? How difficult can achieving true portability be? In fact, it turns out, this is an engineering task that has taken thousands of the world's brightest developers many decades to implement!

Thanks to the magic of container technology we can now run any Linux program on almost any networked device on the planet. All of the environment preparation, installation and configuration steps can be automated from start to finish. Depending on how much network bandwidth you have, it might take a while, but that's all right. All you need to do is type a single command string correctly.

Docker is a tool for portable, reproducible, and self-contained computing. It is used to perform operating-system-level virtualization, something often referred to as *containerization*. While Docker is not the only software that does this, it is by far the most popular one.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (17 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Results: The very basic knowledge of using Docker

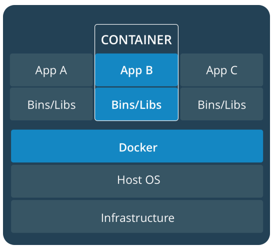
Contents

5.1. What's so special about containerization?

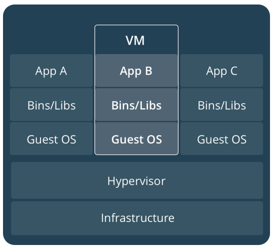
A (Docker) *container* is a packaging around all the software (libraries, configuration files, services, binary executable, etc.) that a computer needs to run a program. And by all, we don't simply mean the source code or the dependencies, we really mean all. Everything you need, from the lowest level OS components to the user interface. A container does not care what flavor or release of Linux you try to run it on, it has everything it needs to work everywhere inside it (it is a container, afterall). Not to mention that Linux Docker containers can generally be also executed on Mac OS and Windows as well!

Containerization is a process that allows partitioning the hardware and the core software (the kernel) of an operating systems in such a way that different containers can co-exist on the same system independently from one-another. Programs running in such a container have access only to the resources they are allow to and are completely independent of libraries and configurations of the other containers and the host machine. Because of this feature, Docker containers are extremely *portable*.

Containers are often compared to virtual machines (VMs). The main difference is that VMs require a host operating system (OS) with a hypervisor (another program) and a number of guest OS, each with their own libraries and application code. This can result in a significant overhead. Imagine running a simple Ubuntu server in a VM on Ubuntu: you will have most of the kernel libraries and binaries twice and a lot of the processes will be duplicated on the host and on the guest. Containerization, on the other hand, leverages the existing kernel and OS, keeps track of what you already have and adds only the additional binaries, libraries and code necessary to run a given application. See the illustration below.



(a) Using containers



(b) Using VMs

Figure 5.1. Comparison between containers and VMs (from docker.com)

Because containers don't need a separate OS to run they are much more lightweight than VMs. This makes them perfect to use in cases where one needs to deploy a lot of independent services on the same hardware or to deploy on not-that-powerful platforms, such as a Raspberry Pi - the platform Duckiebots use.

Containers allow for reuse of resources and code, but are also very easy to work with in the context of version control. If one uses a VM, they would need to get into the VM and update all the code they are using there. With a Docker container, the same process is as easy as pulling the container image again.

The same feature makes Docker containers great for development. If you mess up a configuration or a library in a container, all you need to do to fix it is, stop it, remove it, and try again. There is no trace left on your system and you cannot break down your OS by committing a simple stupid mistake in a container.

And the best part of it all, Docker containers are extremely portable. That means, that once you package your mindbogglingly-awesome Duckiebot code as a Docker container, you can very easily share it with your friends and anyone else in the world, who would be able to try it on their own robot with a single line in the terminal. Just as easily you can test it in simulation or even submitting for competing in the AI Driving Olympics!

5.2. What is it in a Docker container?

You can think of Docker containers as objects built from Docker images which in turn are built up of Docker layers. So what are these?

Docker images are build-time constructs while Docker containers are run-time constructs. That means that a Docker image is static, like a `.zip` or `.iso` file. A container is like a running VM instance: it starts from a static image but as you use it, files and configurations might change.

Docker images are build up from layers. The initial layer is the base layer, typically an official stripped-down version of an OS. For example, a lot of the Docker images we run on the Duckiebots have `rpi-ros-kinetic-base` as a base.

Each layer on top of the base layer constitutes a change to the layers below. The Docker internal mechanisms translate this sequence of changes to a file system that the container can then use. If one makes a small change to a file, then typically only a single layer will be changed and when Docker attempts to pull the new version, it will need to download and store only the changed layer, saving space, time and bandwidth.

In the Docker world images get organized by their *repository name*, *image name* and *tags*. As with Git and GitHub, Docker images can be stored in image registers that reside on the Internet and allow easy worldwide access to your code. The most popular Docker register is called *DockerHub* and it is what we use in Duckietown.

A Duckietown image stored on DockerHub has a name of the form `duckietown/rpi-ros-kinetic-base:daffy`. Here the repository name is `duckietown`, the image name is `rpi-ros-kinetic-base`, and the tag is `daffy`.

All Duckietown-related images are in the `duckietown` repository. The images themselves can be very different and for various applications.

Sometimes a certain image might need to have several different versions. These can be designated with tags. For example, the `daffy` tag means that this is the image to be used with the `daffy` version of the Duckietown code base.

It is not necessary to specify a tag. If you don't, Docker assumes you are interested in the image with `latest` tag, should such an image exist.

5.3. Working with Docker images

We will now take a look at how you can use Docker in practice. For this, we assume you have already set up Docker on your computer as explained in the Laptop Setup page.

If you want to get a new image from a Docker register (e.g., DockerHub) on your local machine then you have to *pull* it. For example, you can get an Ubuntu 18.04 image by running the following command:

```
$ docker pull library/ubuntu:18.04
```

You will now be able to see the new image you pulled if you run:

```
$ docker image list
```

Just like that you got a whole new OS on your computer with a single line in the terminal!

If you don't need this container, or if you're running down on storage space, you can remove it by simply running:

```
$ docker image rm ubuntu:18.04
```

You can also remove images by their `IMAGE ID` as printed by the `list` command.

If you want to look into the heart and soul of your images, you can use the commands `docker image history` and `docker image inspect` to get a detailed view.

5.4. Working with containers

Containers are the run-time equivalent of images. When you want to start a container, Docker picks up the image you specify, creates a file system from its layers, attaches all devices and directories you want, "boots" it up, sets up the environment, and starts a pre-determined process in this container. All that magic happens with you running a single command: `docker run`. You don't even need to have pulled the image beforehand, if Docker can't find it locally, it will look for it on DockerHub.

Here's a simple example:

```
$ docker run ubuntu
```

This will take the `ubuntu` image with latest tag and will start a container from it.

The above won't do much. In fact, the container will immediately exit as it has nothing to execute. When all processes of a container exit, the container exits as well. By default this `ubuntu` image runs `bash` and as you don't pass any commands to it, it exits immediately. This is no fun, though.

Let's try to keep this container alive for some time by using the `-it` switch. This tells Docker to create an interactive terminal session.

```
$ docker run -it ubuntu
```

Now you should see something like:

```
$ root@73335ebd3355:/#
```

Keep in mind that the part after `@` (the container's hostname) will be different - that is your `container ID`.

In this manual, we will use the following icon to show that the command should be run in the container:

```
$ command to be run in the container
```

You are now in your new `ubuntu` container! Try to play around, you can try to use some basic bash commands like `ls`, `cd`, `cat` to make sure that you are not in your host machine.

If you are sure about the difference between the host and the container, you might want to see what happens when you do `rm -rf /` **IN THE CONTAINER**. Do that extremely carefully because that wipes out all of the root of a system. You do not want to run this on your host. By running the above command in a Docker container you will destroy the OS inside the container - but you can just exit and start another one. If instead you have confused host and container, at this point you probably need to re-install your OS.

You can check which containers you are running using the docker `ps` command - analogous to the Linux `ps` command. Open a new terminal window (do not close the other one just yet) and type:

```
$ docker ps
```

An alternative syntax is

```
$ docker container list
```

These commands list all running containers.

Now you can go back to your `ubuntu` container and type `exit`. This will bring you back to your host shell and will stop the container. If you again run the `docker ps` command you will see nothing running. So does this mean that this container and all changes you might have made in it are gone? What about all these precious changes you made in it? Are they forever lost into the entropy abyss of electric noise in your computer's memory? Not at all, `docker ps` and `docker container list` only list the currently running containers.

You can see all containers, including the stopped ones with:

```
$ docker container list -a
```

Here `-a` stands for all. You will see you have two `ubuntu` containers here (remember the first one that exited immediately?). There are two containers because every time you use `docker run`, a new container is created. Note that their names seem strangely random. We could have added custom, more descriptive names, but more on this later.

We don't really need both of these containers, so let's get rid of one of them:

```
$ docker container rm container name
```

You need to put your container name after `rm`. Using the container ID instead is also possible. Note that if the container you are trying to remove is still running you will have to first stop it.

You might need to do some other operations with containers. For example, sometimes you want to start or stop an existing container. You can simply do that with:

```
$ docker container start container name
$ docker container stop container name
$ docker container restart container name
```

Imagine you are running a container in the background. The main process is running but you have no shell attached. How can you interact with the container? You can open a terminal in it with:

```
$ docker attach container name
```

Let's start again the container that we stopped before. You can check its container ID and name via `docker container list -a`. You can then start it again with command introduced above. You will see that the docker start command will only print the container ID and will return you back to the terminal. Rather uneventful, huh? Don't worry, your container is actually running: check that with `docker ps`.

But even though it is running, it seems you cannot do anything with it. But fear not, use the `docker attach` command to get back in the container's shell. Now you're back in and ready for the next adventure.

Often, you will need to run multiple processes in a single container. But how could you do that if you have only a single terminal? Well, Docker has a neat command for that: `docker exec`. The full signature of it is `docker exec CONTAINER_NAME/ID COMMAND`. Let's use that to create a file in our Ubuntu container that is already running. Open a new terminal and simply substitute the container name or ID in the signature above and use the command `touch /quackworld` which should create an empty file called `quackworld` in the container's root. The full command should look like that:

```
$ docker exec c73ee1f963a2 touch /quackworld
```

Verify that the file was indeed created by running it again, but this time with the command `ls \` instead, which will show you the contents of the root folder. Finally, verify that the change was made in the same container as the one to which you attached before by finding the file there and that the change was not made on your host by checking that you don't have a file called `quackworld` in your root folder.

Unit A-6
Basic Duckiebot operation

Now that you know more about how to assemble a duckiebot, how to use a terminal, how to set up a Duckiebot, how to handle a bit of networking and a bit of Docker, it is high time you learn how to use the basic functionalities of the Duckiebot. In this section, you will learn multiple ways to operate and manage existing functions of the Duckiebot.

Knowledge and activity graph

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (18 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Results: Know how to use the Dashboard, Portainer and the DT shell for demos.

Contents

6.1. Remote connection with a browser and an interface

One of the easiest way to use and get an overview of your Duckiebot's operations capacities is to use a Duckietown designed web interface, that we call the *Dashboard*. The dashboard will allow you to monitor and operate basic functionalities of the Duckiebot.

Exercise 8. Using the Dashboard.
To set up the dashboard, follow this tutorial (unknown ref opmanual_duckiebot/duckiebot-dashboard-setup)

previous **warning** next (19 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/duckiebot-dashboard-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. Once on the dashboard, explore the interface and try to understand its features. Through the dashboard you can, e.g., move the Duckiebot. You can find a tutorial on how to do so on (unknown ref opmanual_duckiebot/setup-ros-websocket-image)

previous **warning** next (20 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/setup-ros-websocket-image'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

You can even see what the Duckiebot is seeing Through the dashboard. You can follow the instructions from (unknown ref opmanual_duckiebot/image-dashboard)

previous **warning** next (21 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/image-dashboard'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

to do so.

The dashboard is really useful for quick debugging and for moving the Duckiebot. We suggest you use it every time you have doubts about the camera nor working or the motors not being plugged in the right way.

But this interface has its limits, as it hides everything that is actually running on the duckiebot. To better understand the duckiebot, let's take a look at what is under the hood : we will use portainer.

To manage and use containers, the command line interface is not so easy to use. But there exist a tool that create a nice interface to manage containers: *Portainer*. Portainer is itself a container that runs on a device. Let's learn how to use it.

Exercise 9. Using Portainer.

Luckily, We have one running directly on the duckiebots at startup. Go to *hostname*`.local:9000` on your web browser. You should arrive on an interface. Navigate on the side window to `Containers`. Here you will see all the containers that are running or that are stopped on your duckiebot.

> Look for the one that has `duckiebot_interface` in the name. This one contains all the drivers you need to drive around, use the camera and the leds.
>
> Select it, click on stop, then try to move your duckiebot around again with the dashboard. It doesn't work anymore. Select it again and start it. Now, find the `logs` button, right next to the name. This will open the logs output of the container. This can be very useful to debug new containers. In here you might see the error messages if something goes wrong.

With this interface, you can also attach a shell to the container, monitor its memory and cpu usage, and inspect its configuration.

Portainer is really helpful to manage images and containers that are already on the duckiebot, but what about if you want to create a new container or run a new demo. You could still do it from there, but it is not very intuitive. We commonly use the `dt shell`, that you already have installed.

6.2. Starting a demo using the DT shell

In the Duckietown world, demos are containers that contain a set of functionalities ready to work, if the rest of the Duckiebot is set up properly (e.g. dt-car-interface and dt-duckiebot-interface are running). This is also the moment where the work done in Section 3.4 - Calibration finally pays off. In order for the demo to work nicely, every Duckiebot must have undergone a calibration procedure to account for its motors' and camera's characteristics. In other words, the calibration procedure ensures that every Duckiebot will behave in the same way when it is given the same set of inputs or commands. The demos all follow the same workflow, which is described here (unknown ref opmanual_duckiebot/running-demos)

previous **warning** next (22 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/running-demos'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

> **Exercise 10. Try out the lane-following demo.**
> Let's now start a lane_following demo. To do so, follow these instructions (unknown ref opmanual_duckiebot/demo-lane-following)
>
> previous **warning** next (23 of 45) index
> warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/demo-lane-following'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

After following the instructions completely, you should have run the lane following demo, and seen the visual output of the lane filter node.

In the duckiebot operation manual, you can find the instructions for the other demos. We mainly use the indefinite_navigation one.

Part B
[RH2] Basic Development

In this part you will get to make your first small program that runs on your Duckiebot! But before that, we will cover some important tool and handy skill you need.

Contents

Unit B-1
Git and GitHub

Working on software in a group is great for development, but it automatically brings many pitfalls and issues. How to handle code that has been modified at the same time by two members of the group? How to keep an eye on what other members write in the code? How to keep enough history of the code to be able to go back to a stable version when something bad was added? How to do that when a few hundred people work on the same code and not go crazy. The answer is simple: **code versioning tools**. These tools allow communities to swiftly handle these issues. The most used one, and the one we will use, is **git**.

Knowledge and activity graph

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (24 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Results: Know how to extensively use a code versioning tools, git

Contents

1.1. Learning git

Git is a great tool, that is **mandatory** to anyone doing any sort of code. Learning how to use it is essential.

> **Exercise 11. Git tutorial.**
> To learn how to use all of git's functionalities, complete this tutorial.

1.2. What is github

"GitHub is a code hosting platform for version control and collaboration. It lets you and

others work together on projects from anywhere." (source : github.com)

Github is where all the code is stored. It provides tools to handle pull requests, issues, and much more. The Duckietown organization github page hosts all relevant code. It is comprised of many different repositories.

1.3. Being a good git citizen

Knowing how to use git is the first step. The second step, which is of the same importance, is knowing how to use it well.

1) Commits

- Commits need to be **granular**: One commit contains on fix, or one function. It cannot have two new functions, and three bug fix. This means that it is better to do too many commits that not enough. This is helpful when doing cherry picks, or when checking out a previous version of the code.
- Commits need to have **meaningful messages**: The message of the commit should describe its content.

2) Branches, forks, pull request and peer review

If you are going to work on a new function, but are not sure yet how it is going to go, then you cannot work on the master branch. This master branch needs to only receive code that has been tested, reviewed and approved by the team.

You then have **two solutions**:

- **Branching** On the main remote, you can branch out of the master branch, as explained in the above tutorial. Please give a relevant name to the branch (example : *"devel-new-flying-function"*). On repositories that you and a small team use a lot, this is the best option.
- **Forking** You can fork the main repo into you own workspace, and work from here. On repositories that are used by a lot of people, or that you very rarely will modify, this is the best option.

No matter the chosen solution, you then do your work, commit it, and then push it to github. On github, your branches will appear in your repository. When you feel like it is ready to be integrated to the master branch, you can open a pull request. This will allow your co workers to see the modifications you made.

What you need to do:

- Check that you are not committing wrong things by error.
- Provide a clear description of your work
- explain why it is relevant
- test it before opening the pull request, and explain that the test worked
- assign relevant co-workers to review the code

What the reviewers need to do (all in the github interface):

- Go through the modified code
- Comment directly on lines that raise questions and doubts

- Propose modifications
- And then, when all conversation are resolved, approve and merge the pull request

A pull request must never be approved and merged by the person who submitted it. Peer review is one of the most important part of software development. Not only it does allow for error proofing, but it also allows for someone to make a code suggestion alone. This way the code can be easily discussed and improved, even when it was functional to start with.

UNIT B-2
Python programs and environments

We assume you are already quite comfortable with Python. Nevertheless, when you work with big and complex projects, there are some subtleties that you must consider and some handy tools that can make your life easier. Let's take a look at some of these now.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (25 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Results: Developer knowledge of Python.

Contents

2.1. Define a basic project structure

In Duckietown, everything runs in Docker containers. All you need in order to run a piece of software in Duckietown is a Duckietown-compliant Docker image with your software in it.

A boilerplate is provided by the following repository.

The repository contains a lot of files, but do not worry, we will analyze them one by one. Click on the green button that says "Use this template".

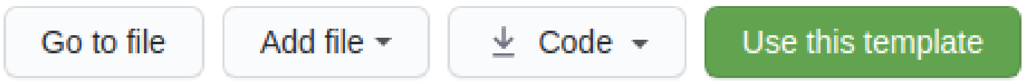
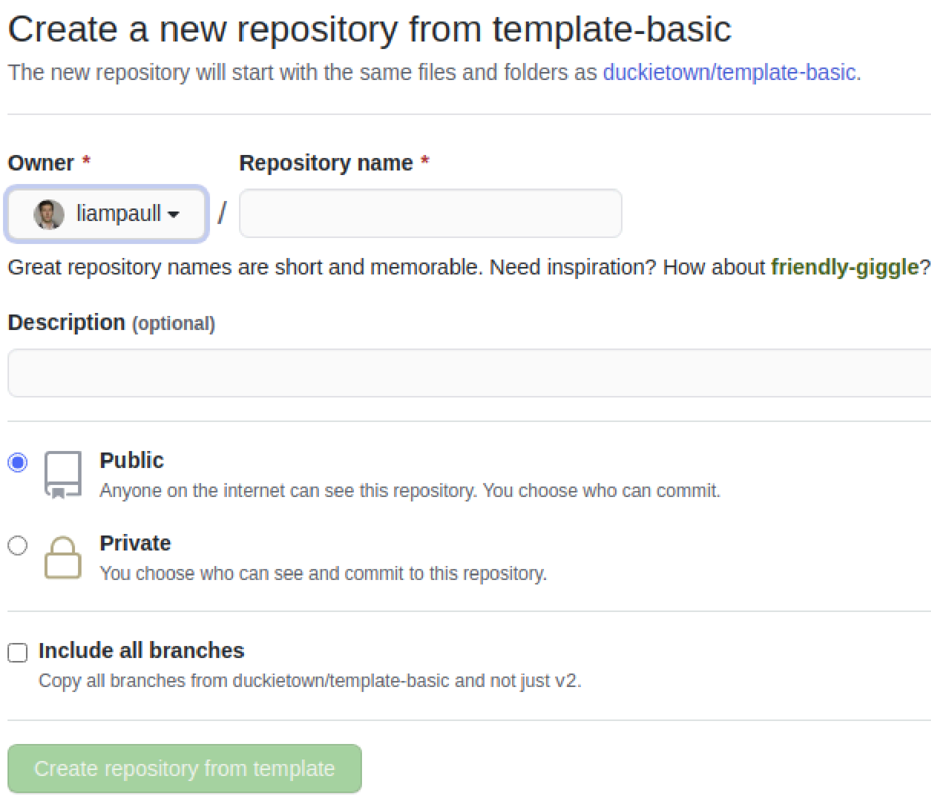


Figure 2.1

This will take you to a page that looks like the following:


Create a new repository from template-basic

The new repository will start with the same files and folders as duckietown/template-basic.

Owner * liampaull / Repository name *

Great repository names are short and memorable. Need inspiration? How about **friendly-giggle**?

Description (optional)

(● Public) Anyone on the internet can see this repository. You choose who can commit.

Private — You choose who can see and commit to this repository.

☐ Include all branches — Copy all branches from duckietown/template-basic and not just v2.

Figure 2.2

Pick a name for your repository (say `my-program`) and press the button *Create repository from template*. Note, you can replace `my-program` with the name of the repository that you prefer, make sure you use the right name in the instructions below.

This will create a new repository and copy everything from the repository `template-basic` to your new repository. You can now open a terminal and clone your newly created repository.

```
$ git clone https://github.com/YOUR_NAME/my-program
$ cd my-program
```

Note: Replace `YOUR_NAME` in the link above with your GitHub username.

The repository contains already everything you need to create a Duckietown-compliant Docker image for your program. The only thing we need to change before we can build an image from this repository is the repository name in the file Dockerfile. Open it using the text editor you prefer and change the first line from:

```
ARG REPO_NAME="REPO_NAME_HERE"
```

to

```
ARG REPO_NAME="my-program"
```

and then similarly update the `DESCRIPTION` and the `MAINTAINER` ARGs.

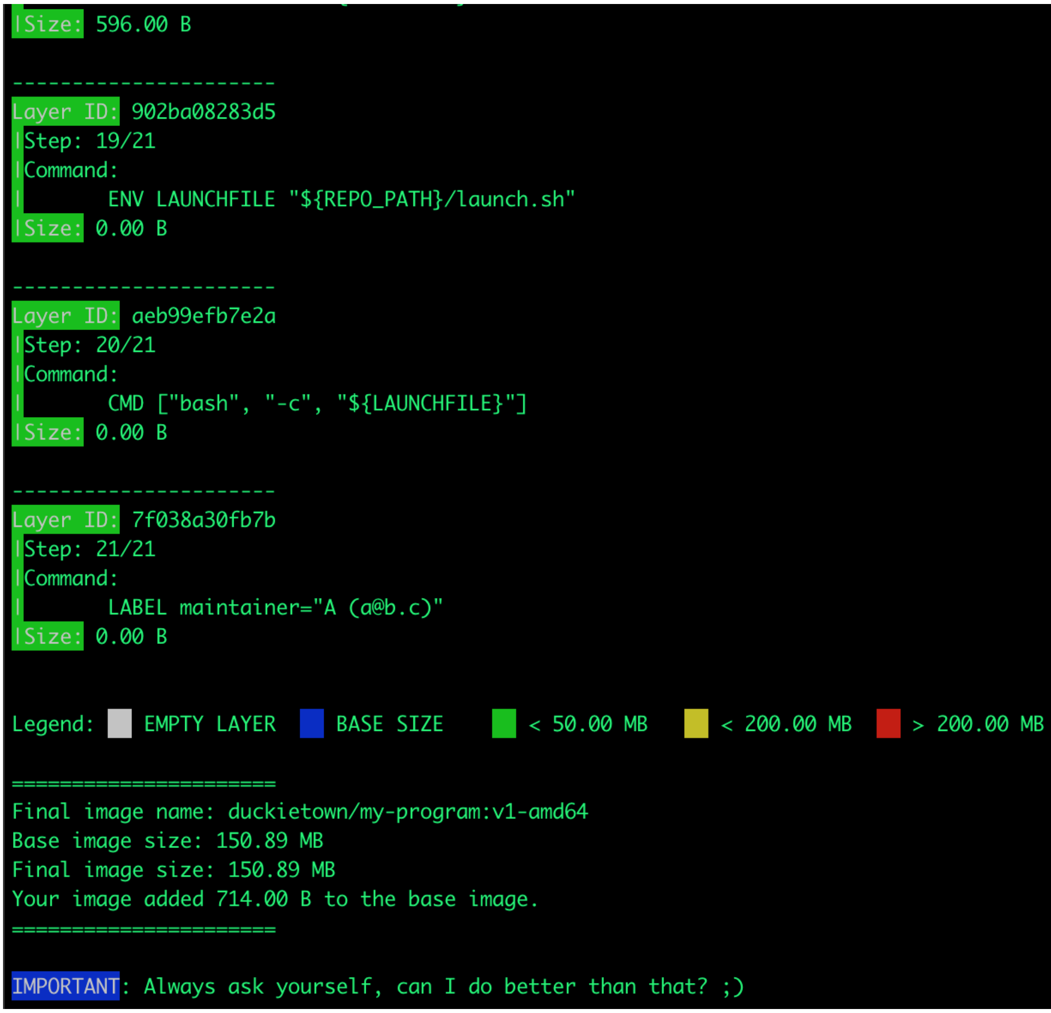
Save the changes. We can now build the image, even though there is not going to be much going on inside it until we place our code in it.

Now, in a terminal, move to the directory created by the `git clone` instruction above and run the following command(beware that it might take some time):

💻
```
$ dts devel build -f
```

> **Note:** If the above command is not recognized, you will first have to install it with `dts install devel`.

If you correctly installed Docker and `dts`, you should see a long log that ends with something like (but not necessary exactly like) the following:



```
|Size: 596.00 B

----------------------
Layer ID: 902ba08283d5
|Step: 19/21
|Command:
|       ENV LAUNCHFILE "${REPO_PATH}/launch.sh"
|Size: 0.00 B

----------------------
Layer ID: aeb99efb7e2a
|Step: 20/21
|Command:
|       CMD ["bash", "-c", "${LAUNCHFILE}"]
|Size: 0.00 B

----------------------
Layer ID: 7f038a30fb7b
|Step: 21/21
|Command:
|       LABEL maintainer="A (a@b.c)"
|Size: 0.00 B


Legend: ⬜ EMPTY LAYER 🟦 BASE SIZE   🟩 < 50.00 MB   🟨 < 200.00 MB   🟥 > 200.00 MB

======================
Final image name: duckietown/my-program:v1-amd64
Base image size: 150.89 MB
Final image size: 150.89 MB
Your image added 714.00 B to the base image.
======================

IMPORTANT: Always ask yourself, can I do better than that? ;)
```

Figure 2.3

You can now run your container by executing the following command.

```
$ dts devel run
```

This will show the following message:

```
The environment variable VEHICLE_NAME is not set. Using 'LAPTOP_HOSTNAME'.
WARNING: robot_type file does not exist. Using 'duckiebot' as default
type.
WARNING: robot_configuration file does not exist.
=&gt; Launching app...
This is an empty launch script. Update it to launch your application.
&lt;= App terminated!
```

Congratulations! You just built and run your first Duckietown-compliant Docker image.

2.2. Run a basic program on your Laptop

Now that we know how to build a Docker image for Duckietown, let's put some code in one of them.

We will see how to write a simple Python program, but any language should do it.

Open a terminal and go to the directory `my-program` created above. In Duckietown, Python code must belong to a Python package. Python packages are placed inside the directory code in `my-program`. Let go ahead and create a directory called `my_package` inside packages.

💻
```
$ mkdir -p ./packages/my_package
```

A Python package is simply a directory containing a special file called `__init__.py`. So, let's turn that `my_package` into a Python package.

💻
```
$ touch ./packages/my_package/__init__.py
```

Now that we have a Python package, we can create a Python script in it. Use your favorite text editor to create the file `./packages/my_package/my_script.py` and place the following code inside it.

```
message = "Hello World!"
print(message)
```

We now need to tell Docker we want this script to be the one executed when we run the command `docker run`. In order to do so, open the file `./launchers/default.sh` and replace the line

```
echo "This is an empty launch script. Update it to launch your application."
```

with the line

```
dt-exec python3 -m "my_package.my_script"
```

Note: Always prepend `dt-exec` to the main command in `./launchers/default.sh`.

If you are curious about why that is important, we can tell you that it helps us deal with an interesting problem called "The zombie reaping problem" (more about this in this article).

You can also create different custom executable scripts, if you want to know more about that check out the file ./readme.md.

Let us now re-build the image:

```
$ dts devel build -f
```

and run it:

```
$ dts devel run
```

This will show the following message:

```
The environment variable VEHICLE_NAME is not set. Using '774a2521b42e'.
Adding /code/my-program to PYTHONPATH
Adding /code/dt-commons to PYTHONPATH
Activating services broadcast...
Done!

Hello World!

Deactivating services broadcast...
Done!
```

Congratulations! You just built and run your own Duckietown-compliant Docker image.

2.3. Run a basic program on your Duckiebot

Now that we know how to package a piece of software into a Docker image for Duckietown, we can go one step further and write code that will run on the robot instead of our laptop.

This part assumes that you have a Duckiebot up and running with hostname `MY_ROBOT`. Of course you don't need to change the hostname to `MY_ROBOT`, just replace it with your robot name in the instructions below. You can make sure that your robot is ready by executing the command

```
$ ping MY_ROBOT.local
```

If we can ping the robot, we are good to go.

Let us go back to our script file my_script.py and change it to:

```python
import os
message = "Hello from %s!" % os.environ['VEHICLE_NAME']
print(message)
```

We can now modify slightly the instructions for building the image so that the image gets built directly on the robot instead of your laptop or desktop machine. Run the command

```
$ dts devel build -f --arch arm32v7 -H MY_ROBOT.local
```

As you can see, we changed two things, one is `--arch arm32v7` which tells Docker to build an image that will run on ARM architecture (which is the architecture the CPU on the robot is based on), the second is `-H MY_ROBOT.local` which tells Docker where to build the image.

Once the image is built, we can run it on the robot by running the command

```
$ docker -H MY_ROBOT.local run -it --rm --net=host duckietown/my-pro-
gram:latest-arm32v7
```

Please take into consideration that the image tag may be different, you can check the correct image name and tag in the end of the output the build command. If everything worked as expected, you should see the following output,

```
The environment variable VEHICLE_NAME is not set. Using 'MY_ROBOT'.
Adding /code/my-program to PYTHONPATH
Adding /code/dt-commons to PYTHONPATH
Activating services broadcast...
Done!

Hello from MY_ROBOT!

Deactivating services broadcast...
Done!
```

Congratulations! You just built and run your first Duckietown-compliant and Duckiebot-compatible Docker image.

2.4. Install dependencies using package managers (e.g., `apt`, `pip`)

It is quite common that our programs need to import libraries, thus we need a way to install them. Since our programs reside in Docker images, we need a way to install libraries in the same image.

The template provided by Duckietown supports two package managers out of the box:

- Advanced Package Tool (`apt`)
- Pip Installs Packages for Python3 (`pip3`)

List your apt packages or pip3 packages in the files `dependencies-apt.txt` and `dependencies-py3.txt` respectively before running the command `dts devel build`.

> **Exercise 12. Basic NumPy program.**
> Write a program that performs the sum of two numbers using NumPy. Add `numpy`

> to the file `dependencies-py3.txt` to have it installed in the Docker image.

Here you go! Now you can handle pip dependencies as well!

UNIT B-3
Become a Docker Power-User

We already introduced in Unit A-5 - Docker basics what Docker containers are and how you can start them and do basic operations. Recall that a Docker container is a closed environment and any change you do there cannot affect your host system or other containers. This can be great if you want to protect your laptop from possible mischief coming from inside a container, but at the same time limits what you can do with it. Thankfully, Docker has some very powerful ways to interact with your system and the outside world.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (26 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Docker basics

Results: Advanced knowledge of using Docker images and containers

Contents

3.1. Getting data in and out of your container

Docker provides a few ways to extract and import files from and to a container. We will look only at volume mounting as it is the most used and versatile way. In the simplest terms, mounting a volume to a container essentially means that you make a directory on your host machine available in the container. Then, you can think of these two directories as perfect copies of each-other: if you change something in one of them, it will be changed in the other as well. Therefore, if your container needs some data or configuration files to operate properly, or if you need to export your results out of it, volume mounting is the way to go. So, how does it work?

You can use `docker run` with the `-v host_dir:container_dir` option. Here `-v` is a shortcut for `--volume`. This specifies that `container_dir` in the container will be replaced with `host_dir` from your computer. Give it a try:

> **Exercise 13. Docker volume mounting.**
> Run a new Ubuntu container where you mount your home directory in the container's home directory:
>
> ```
> $ docker run -it -v ~:/home ubuntu
> ```
>
> In bash `~` is a shortcut for your home directory (`/home/your_username`). Now if you check which files are in the container's home directory by running `ls /home` you'd see the files you have on your host machine. Try to change one of them (hopefully one not that important file) or to create a new one with, for example, `touch test.txt`. The easiest way to modify a simple file is to append a string to its content with something like `echo "hello" >> test.txt`. Check in your host home folder if the changes appear there as well. Now do the opposite: make a change in your host and observe if there's a corresponding change in the container. To visualize the content of a simple file you can use the command `cat test.txt`.

3.2. Docker and networking

The default network environment of a Docker container (a bridge network driver) gives your container access to the Internet but not much more. If you run, for example, a web server in the container, you wouldn't be able to access it from your host. This is not ideal for us as most of the Duckietown code-base actually uses similar technologies to connect the various parts of the code.

However, by adding `--network host` to the `docker run` command, we can remove the network isolation between the container and the Docker host and therefore, you can use the full range of networking capabilities that your host has within the convenient environment in the container.

3.3. Handling devices

The Docker containers do not have access to the devices on your computer by default. Yup, if you put your code in a container it cannot use the camera, wheels and LEDs of your Duckiebot. No fun, right? Thankfully, just like with the network, Docker has a solution for that! You can manually allow each device to be available to your container or you can allow all of them by simply passing the `--privileged` option to `docker run`. You will see that option being often used in Duckietown.

3.4. Other fancy option

Docker provides many more options for configuring your containers. Here's a list of the most common ones:

TABLE 3.1. DOCKER RUN OPTIONS

Short command	Full command	Explanation
`-i`	`--interactive`	Keep STDIN open even if not attached, typically used together with `-t`.
`-t`	`--tty`	Allocate a pseudo-TTY, gives you terminal access to the container, typically used together with `-i`.
`-d`	`--detach`	Run container in background and print container ID.
	`--name`	Sets a name for the container. If you don't specify one, a random name will be generated.
`-v`	`--volume`	Bind mount a volume, exposes a folder on your host as a folder in your container. Be very careful when using this.
`-p`	`--publish`	Publish a container's port(s) to the host, necessary when you need a port to communicate with a program in your container.
`-d`	`--device`	Similar to `-v` but for devices. This grants the container access to a device you specify. Be very careful when using this.
	`--privileged`	Give extended privileges to this container. That includes access to **all** devices. Be **extremely** careful when using this.
	`--rm`	Automatically remove the container when it exits.
`-H`	`--hostname`	Specifies remote host name, for example when you want to execute the command on your Duckiebot, not on your computer.
	`--help`	Prints information about these and other options.

Examples

Set the container name to `joystick`:

```
--name joystick
```

Mount the host's path `/home/myuser/data` to `/data` inside the container:

```
-v /home/myuser/data:/data
```

Publish port 8080 in the container as 8082 on the host:

```
-p 8082:8080
```

Allow the container to use the device `/dev/mmcblk0`:

```
-d /dev/mmcblk0
```

Run a container on the Duckiebot:

```
-H duckiebot.local
```

Unit B-4
AIDO submissions

The Duckietown platform is one of many possibilities. In particular it is used for a international competition named AIDO. You will probably have part in it in one way or the other. You need to be able to participate in it.

Knowledge and activity graph

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (27 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Results: Know how to participate in AIDO.

Contents

4.1. Getting started

The AIDO book is complete and already has all the necessary instructions.

Exercise 14. Setup your account and software.
Follow the instructions accounts needed (unknown ref AIDO/cm-accounts)

previous **warning** next (28 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#AI-
DO/cm-accounts'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

and software requirements (unknown ref AIDO/cm-sw)

previous **warning** next (29 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#AI-
DO/cm-sw'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

4.2. Make a simple submission

To first get started and understand the workflow of AIDO submission, you will submit one with its default version.

Exercise 15. Make a simple submission.
Follow the instructions here (unknown ref AIDO/cm-first)

previous **warning** next (30 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#AI-
DO/cm-first'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. You will have to :

- retrieve a submission repository
- submit the default solution
- monitor your submission
- explore the leaderboard

On the AIDO website, find your submissions jobs, and play around with the following parameters:

- priority : changes the order of evaluation priority amongst your various submissions
- resetting : reset a job to make it restart
- retiring : removing a job from the evaluation queue

4.3. Customize a solution

Of course, the idea is not to submit the default solutions, but to improve them. This part is not mandatory, but you can go around and try to do better, by following the quickstart instructions (unknown ref AIDO/quickstart-lanefollowing)

previous **warning** next (31 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#AIDO/
quickstart-lanefollowing'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

UNIT B-5
Creating Docker containers

We spent a lot of time looking at how to use Docker containers and the image that they start from. But that still leaves a very important question open: how can you make your own image? Now you will have the opportunity to make your first image that will do some basic computer vision processing on your Duckiebot!

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (32 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Duckiebot initialization (unknown ref opmanual_duckiebot/setup-duckiebot)

previous **warning** next (33 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/setup-duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Docker basics

Requires: Docker poweruser skills

Results: Advanced knowledge of using Docker images and containers.

Contents

5.1. Where do Docker containers come from?

So far we saw that you can get a Docker image from the DockerHub by knowing its name. How do these images get on DockerHub? Well, the simple answer is that you register an account and then similarly to git, you can push one of your images to DockerHub. And how do you create an image in the first place?

A simple, though rarely practiced way is to convert a container in which you have made some changes into a new image. This can be done through the docker commit command. However, as this is not the preferred mode of operation we won't discuss it further. But you can find more details in the official documentation.

The more popular and accepted way is to build an image from a "recipe", called a Dockerfile. A Dockerfile is a text file that specifies the commands required to create a Docker image, typically by modifying an existing container image using a scripting interface. They also have special keywords (which are always CAPITALIZED), like `FROM`, `RUN`, `ENTRYPOINT`, and so on. For example, create a file called Dockerfile with the following content:

```
FROM ubuntu
RUN touch new_file1
CMD ls -l
```

The first line above defines the base image on top of which we will build our container. The second line simply executes the Linux command `touch new_file1` which creates a new file with this name. And the last line is the default command that will be run when the container is started (unless the user provides a different command).

Now, to build the image we can simply run:

```
$ docker build -t my_first_container:v1 .
```

The last part of this command denotes the directory (called *context*) which contains your Dockerfile. The `.` shorthand refers to the current directory.

You should see something like:

```
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
 --- ea2f90g8de9e
Step 2/3 : RUN touch new_file1
 --- e3b75gt9zyc4
Step 3/3 : CMD ls -l
 --- Running in 14f834yud59
Removing intermediate container 14f834yud59
 --- 05a3bd381fc2
Successfully built 05a3bd381fc2
Successfully tagged my_first_container:v1
```

Now run the command `docker images` in your terminal, and you should see an image called `my_first_container` with tag `v1`:

```
$ docker images
REPOSITORY          TAG        IMAGE ID         CREATED          SIZE
my_first_container  v1        05a3bd381fc2     2 seconds ago    88.9MB
```

An interesting observation is that the container size is `88.9MB`. Now, instead of needing to carry around a `88.9MB` file, we can just store the `4KB` text file and rest assured that all our important setup commands are contained within. In a sense, a whole OS, with our custom file inside is compressed to 3 lines of code.

Now, similarly to before, we can simply run:

```
$ docker run -it my_first_container:v1
total 64
drwxr-xr-x   2 root root 4096 Mar  7  2019 bin
drwxr-xr-x   2 root root 4096 Apr 24  2018 boot
drwxr-xr-x   5 root root  360 Sep 21 18:45 dev
drwxr-xr-x   1 root root 4096 Sep 21 18:45 etc
drwxr-xr-x   2 root root 4096 Apr 24  2018 home
drwxr-xr-x   8 root root 4096 May 23  2017 lib
drwxr-xr-x   2 root root 4096 Mar  7  2019 lib64
drwxr-xr-x   2 root root 4096 Mar  7  2019 media
drwxr-xr-x   2 root root 4096 Mar  7  2019 mnt
-rw-r--r--   1 root root    0 Sep 21 18:41 new_file1
drwxr-xr-x   2 root root 4096 Mar  7  2019 opt
dr-xr-xr-x 328 root root    0 Sep 21 18:45 proc
drwx------   2 root root 4096 Mar  7  2019 root
drwxr-xr-x   1 root root 4096 Mar 12  2019 run
drwxr-xr-x   1 root root 4096 Mar 12  2019 sbin
drwxr-xr-x   2 root root 4096 Mar  7  2019 srv
dr-xr-xr-x  13 root root    0 Sep 21 18:45 sys
drwxrwxrwt   2 root root 4096 Mar  7  2019 tmp
drwxr-xr-x   1 root root 4096 Mar  7  2019 usr
drwxr-xr-x   1 root root 4096 Mar  7  2019 var
```

Notice that as soon as we run the container Docker will execute the `ls -l` command as specified by the Dockerfile, revealing that `new_file1` was indeed stored in the image. However, we can still override `ls -l` by passing a command line argument: `docker run -it your/duck:v3 [custom_command]`.

5.2. Environment variables and Docker containers

Environment variables are often used to control the behavior of one or more programs. As the name hints, these variables are associated with a particular (terminal) environment and are shared among processes. In fact, all processes started from an environment inherit its set of environment variables. If you are curious, you can check out the Wikipedia article about them.

In bash you can set an environment variable with `export VAR_NAME=var_value`, and to check a variable's current value use `echo \$VAR_NAME`. Python allows you to easily get the environment variable of the environment where the program was started in through the `os` module and its dictionary `os.environ['VAR_NAME']`.

> **Exercise 16. Environment variables in Docker.**
> Open a terminal and set a new environment variable `MY_VAR` with any value you like. Then start an interactive Python session in the same terminal and check the value of `MY_VAR` using the above function.

In the Docker universe environment variables are particularly useful to configure a container when you run it. Imagine that your code can be run with different configuration variables (e.g. gain for the motors, exposure mode of the camera, etc.). Then you can set the value of this variable when you run the container, e.g.

```
$ docker run -e CAMERA_EXPOSURE='sport' my_fancy_camera:alpha
```

Then the Python code in the container can obtain the value you passed via the `os.environ` dictionary. In this way you make a single Docker image that can initialize containers with all sorts of configurations. Quite powerful, right?

5.3. Guide to the Dockerfile keywords

Here are some of the most commonly used Dockerfile keywords. You will see them in many of the Duckietown Dockerfiles and you will often make use of them. You can find much more information and details on how to use them on Docker's official documentation.

TABLE 5.1. DOCKERFILE KEYWORDS

Keyword	
FROM	Designates the base image on top of which thi
RUN	
CMD	Executes any shell command at run time, unless the user specifies another comm
ENV	Sets an environment var
COPY	Copies file from
WORKDIR	Change

5.4. Creating your first functional Docker image

Now that you know your way around Dockerfiles, it is time to finally build something meaningful that works on your Duckiebot! We are going to build a very basic vision system: we will try to measure how much of the image stream the camera sees is covered with pixels of a particular color.

Exercise 17. Creating a color detector in Docker.

Note: The following exercise will use the camera on your robot, bear in mind that only one process can access the camera at a time. Therefore, if there is another process on your bot that is already using the camera, your code will likely fail. Make sure that the `dt-duckiebot-interface` and any other container that can use the camera are stopped. You can use Portainer to do that.

We will divide the image that the camera acquires into `N_SPLITS` equal horizontal sectors. `N_SPLITS` will be an environment variable we pass to the container. Think of it as a configuration parameter. The container should find which color is most present in each sector. Or alternatively you can look at the color distribution for each split. It should print the result in a nicely formatted way with a frequency of about 1Hz.

You can start your Dockerfile from `duckietown/dt-duckiebot-interface:daffy-arm32v7`. Most of the stuff you need should already be in there. Make a `requirements.txt` file where you list all your pip dependencies. We would expect that you would need at least `picamera` and `numpy`. Using a `requirements.txt` file is a good practice, especially when you work with big projects. The Dockerfile then copies this file and passes it to pip which installs all the packages you specify there. Finally copy your code in the container and specify it should be the starting command. Here's an example Dockerfile:

```
FROM duckietown/dt-duckiebot-interface:daffy-arm32v7
# use daffy-arm64v8 if you are using a Duckiebot MOOC Founder's Edi-
tion

WORKDIR /color_detector_dir

COPY requirements.txt ./

RUN pip install -r requirements.txt

COPY color_detector.py .

CMD python3 ./color_detector.py
```

Make sure you understand what each single line is doing. Keep in mind that you might need to modify it in order to work for you.

Working with the camera can sometimes be tricky so you can use this template for `color_detector.py` to get started:

Use this if you are using a Raspberry Pi equipped Duckiebot:

```python
import picamera
import picamera.array
from time import sleep

with picamera.PiCamera() as camera:
    camera.resolution = (320, 240)

    while True:
        with picamera.array.PiRGBArray(camera) as output:
            camera.capture(output, 'rgb')
            output = output.array

            # You can now treat output as a normal numpy array
            # Do your magic here

            sleep(1)
```

Instead use this function template if you are using a Duckiebot MOOC Founder's Edition:

```
#!/usr/bin/env python3
import cv2
import numpy as np
from time import sleep


def gst_pipeline_string():
    # Parameters from the camera_node
    # Refer here : https://github.com/duckietown/dt-duckiebot-inter-
face/blob/daffy/packages/camera_driver/config/jetson_nano_camera_node/
duckiebot.yaml
    res_w, res_h, fps = 640, 480, 30
    fov = 'full'
    # find best mode
    camera_mode = 3  # 
    # compile gst pipeline
    gst_pipeline = """ \
            nvarguscamerasrc \
            sensor-mode= exposuretimerange="100000 80000000" ! \
            video/x-raw(memory:NVMM), width=, height=, format=NV12,
                framerate=/1 ! \
            nvjpegenc ! \
            appsink \
        """.format(
        camera_mode,
        res_w,
        res_h,
        fps
    )

    # ---
    print("Using GST pipeline: ``".format(gst_pipeline))
    return gst_pipeline


cap = cv2.VideoCapture()
cap.open(gst_pipeline_string(), cv2.CAP_GSTREAMER)

while(True):
    # Capture frame-by-frame
    ret, frame = cap.read()
    # Put here your code!
    # You can now treat output as a normal numpy array
    # Do your magic here

    sleep(1)
```

Once you have your `color_detector.py` file ready to be tested, you can build it directly on your bot by running:

```
$ docker -H DUCKIEBOT_NAME.local build -t colordetector .
```

Do you remember what `-H` does? It takes the context (the folder in which you are) and ships it to the device specified by `-H` and build the container there. Once the container is built (typically it takes more time the first time), you can test it with:

```
$ docker -H DUCKIEBOT_NAME.local run -it --privileged colordetec-
tor
```

If you want to run the image on a DB21M instead, you must mount the `argus_sock-et` volume to allow using the GStreamer pipeline from the Docker container.

```
$ docker -H DUCKIEBOT_NAME.local run -it --privileged -v /tmp/ar-
gus_socket:/tmp/argus_socket colordetector
```

Again there is the `-H` option (why?) and we also have the `--privileged` option. Do you remember what it does? Try to remove it and see what happens.

We omitted to mention what to do about a lot of implementation details which can significantly affect the performance of your color detector. For example, what should the value of `N_SPLITS` be? Should we consider the whole width of the image or just a central part? How many colors should we detect, which ones and what is the best way to do it? Should you use RGB or HSV color space? All this is left for you to decide. This is typically the case in robotics: you know what the final result should be, but there are multiple ways to get there and it is up to you to decide which is the best solution for the particular case. Experiment and find what makes your color detector really good. We recommend investing some time in this, as this color detector will be a building block in the next module.

5.5. Pushing to DockerHub

Say that you want to share your awesome color detector with your friend. How can you do that? You can of course repeat the same procedure as above, just replacing your Duckiebot's name with theirs. But that is cumbersome and requires them to have the code. DockerHub makes all this much easier. It allows you to push your image to their repository and then anyone can directly use it. That is where all the base images you saw so far come from.

To do this, first make sure you have a DockerHub account. Let's say your account name is `duckquackermann`. Then sharing your container with the world is as easy as building it under your account name:

```
$ docker -H DUCKIEBOT_NAME.local build -t duckquackermann/col-
ordetector .
```

Then push it to DockerHub:

```
$ docker -H DUCKIEBOT_NAME.local push duckquackermann/colordetec-
tor
```

Note: You will probably have to first connect your Duckiebot's Docker client with your DockerHub account. So first open an SSH connection to the robot and then run `docker login` in it. You will be prompted to provide your DockerHub username and password. If you want to be able to push images directly from your laptop, you should do the same there.

After you've pushed your image to DockerHub your code can be executed on any single Duckiebot around the world with a single command:

```
$ docker -H DUCKIEBOT_NAME.local run -it --privileged duckquacker-
mann/colordetector
```

Unit B-6
My First Duckietown Python Library

6.1. Get the Duckietown library template

A boilerplate is provided by the library template repository.

The repository contains a lot of files, but do not worry, we will analyze them one by one. Click on the green button that says "Use this template".

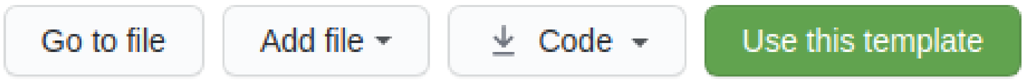
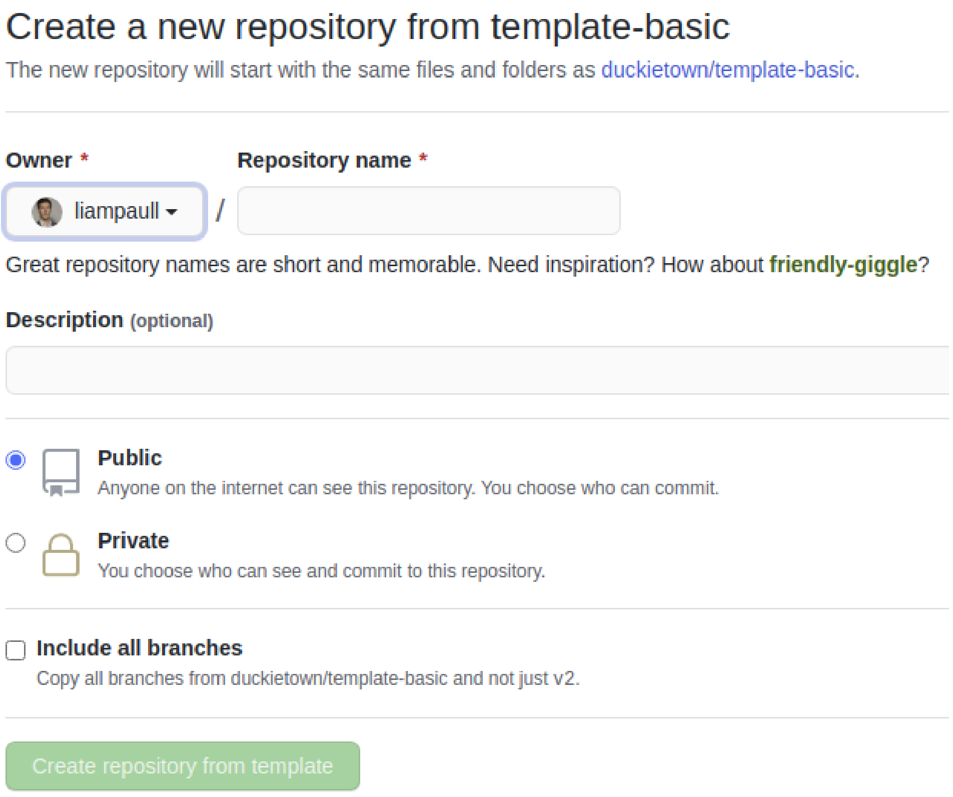


Figure 6.1

This will take you to a page that looks like the following:


Create a new repository from template-basic

The new repository will start with the same files and folders as duckietown/template-basic.

Owner * liampaull / Repository name *

Great repository names are short and memorable. Need inspiration? How about friendly-giggle?

Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Include all branches
Copy all branches from duckietown/template-basic and not just v2.

Figure 6.2

Pick a name for your repository (say `my-library`) and press the button *Create repository from template*. Note, you can replace `my-library` with the name of the repository that you prefer.

This will create a new repository and copy everything from the repository `template-library` to your new repository. You can now open a terminal and clone your newly created repository.

```
$ git clone https://github.com/YOUR_NAME/my-library
$ cd my-library
```

Note: Replace `YOUR_NAME` in the link above with your GitHub username.

6.2. Features of the library template

We have the following features:

- Unit-tests using Nose.
- Building/testing in Docker environment locally.
- Integration with CircleCI for automated testing.
- Integration with CodeCov for displaying coverage result.
- Integration with Sphinx to build code docs. (So far, only built locally.)
- Jupyter notebooks, which are run also in CircleCI as tests.
- Version bump using Bumpversion.
- Code formatting using Black.
- Command-line program for using the library.

6.3. Anatomy of the library template

This repository describes a library called "`duckietown_pondcleaner`" and there is one command-line tool called `dt-pc-demo.`

1) Meta-files

- `.gitignore`: Files ignore by Git.
- `.dtproject`: Enables the project to be built and used by `dts devel` tools
- `.bumpversion.cfg`: Configuration for bumpversion
- `Makefile`: Build tools configuration with Make

2) Python packaging

- `requirements.txt`: Contains the *pinned* versions of your requirement that are used to run tests.
- `MANIFEST.in`: Deselects the tests to be included in the egg.
- `setup.py`: Contains meta information, definition of the scripts, and the dependencies information.

3) Python code

- `src/` - This is the path that you should set as "sources root" in your tool

- `src/duckietown_pondcleaner`: Contains the code.
- `src/duckietown_pondcleaner/__init__.py`: Contains the `__version__` library.
- `src/duckietown_pondcleaner_tests`: Contains the tests - not included in the egg.

4) Docker testing

These are files to build and run a testing container.

- `.dockerignore`: Describes what files go in the docker container.
- `Dockerfile`: ...

5) Sphinx

- `src/conf.py`: Sphinx settings
- `src/index.rst`: Sphinx main file
- `src/duckietown_pondcleaner/index.rst`: Documentation for the package

6) Coverage

- `.coveragerc`: Options for code coverage.

7) Notebooks

- `notebooks`: Notebooks that are run also as a test.
- `notebooks-extra`: Other notebooks (not run as test)
- `notebooks/*.ipynb`: The notebooks themselves.

6.4. Creating your Library

Using the repo you have already created:

- Clone the newly created repository;
- Place your Python packages inside `src/`;
- List the python dependencies in the file `dependencies.txt`;
- Update the appropriate section in the file `setup.py`;

Make sure that there are no other remains:

```
$ grep -r . pondcleaner
```

Update the branch names in `README.md`.

1) Other set up (for admins)

The following are necessary steps for admins to do:

1. Activate on CircleCI. Make one build successful.
2. Activate on CodeCov. Get the `CODECOV_TOKEN`. Put this token in CircleCI environment.

6.5. How to use the utilities in the library template

1) Test the code

Test the code using Docker by:

```
$ make test-docker
```

This runs the test using a Docker container built from scratch with the pinned dependencies in `requirements.txt`. This is equivalent to what is run on CircleCI.

To run the tests natively on your pc, use:

```
$ make test
```

> **Note:** To do so you will need to have installed the libraries listed in the file `requirements.txt` on your computer.

For that we assume you have already setup a Python virtual environment.

> **Note:** To do so you will need to `pip install virtualenv` then `virtualenv duckietown` then `source duckietown/bin/activate`. In order to install the requirements to run the test do `pip install -r requirements.txt`.

2) Development

In the same virtual environment as above run:

```
$ python setup.py develop
```

This will install the library in an editable way (rather than copying the sources somewhere else).

If you don't want to install the deps, do:

```
$ python setup.py develop --no-deps
```

For example, this is done in the Dockerfile so that we know we are only using the dependencies in `requirements.txt` with the exact pinned version.

3) Adding tests

To add another tests, add files with the name `test_*py` in the package `duckietown_podcleaner_tests`. The name is important.

Tip: make sure that the tests are actually run looking at the coverage results.

4) Notes on using the notebooks

Always clean the notebooks before committing them:

💻
```
$ make -C notebooks cleanup
```

If you don't think you can be diligent, then add the notebooks using Git LFS.

5) Releasing a new version

Updating the version:

The first step is to change the version and tag the repo. **DO NOT** change the version manually; use the CLI tool `bumpversion` instead.

The tool can be called by:

💻
```
$ make bump    # bump the version, tag the tree
```

If you need to include the version in a new file, list it inside the file `.bumpversion.cfg` using the syntax `[bumpversion:file: &lt;FILE_PATH &gt;]`.

Releasing the package:

The next step is to upload the package to PyPy. We use twine. Invoke using:

💻
```
$ make upload  # upload to PyPI
```

For this step, uou need to have admin permissions on PyPy.

PART C

[RH3] Advanced Software Development

In this section, you will learn how to use the Robot Operating System (ROS) to enable different processes running on your Duckiebot to communicate with each other. You will also learn how to monitor/visualize these communications, change the behaviour of your robot on-the-fly, and work with ROS logs.

Contents

Unit C-1
Introduction to ROS

The official wiki describes ROS as:

```
... an open-source, meta-operating system for your robot. It provides
the services you would expect from an operating system, including hard-
ware abstraction, low-level device control, implementation of commonly-
used functionality, message-passing between processes, and package man-
agement. It also provides tools and libraries for obtaining, building,
writing, and running code across multiple computers.
```

You probably have some idea about what the above words mean. However, if this is your first encounter with ROS, you are already overestimating how complicated it is. Worry do not.

Putting it in very simple terms, as a roboticist, ROS is what will prevent you from reinventing the wheel at every step of building a robot. It is a framework which helps you manage the code you write, while providing you with a plethora of tools which will speed up the process.

Knowledge and activity graph

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (34 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Duckiebot initialization (unknown ref opmanual_duckiebot/setup-duckiebot)

previous **warning** next (35 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/setup-duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Docker poweruser skills

Results: Basic understanding of ROS

Contents

1.1. Why ROS?

Your Duckiebot is a very simple robot which has very few sensors. In case of DB-18 only one sensor: the camera. However if you are working with newer configuration such as DB-19 or DB-Beta you will have the camera plus the wheel encoders. Every robot also has two actuators: the motors. You can probably write all the code for the basic funtionality of a Duckiebot yourself. You start by getting images from the camera, processing them to detect lanes, generating suitable motor commands, and finally executing them. You create a single program for all of this which looks like this:

```
img = get_image_from_camera()
pose = get_pose_from_image(img)
cmd = get_command_from_pose(pose)
run_motors(cmd)
```

The next day, your Duckiebot crashes into a duckie which was crossing the road, so you want to add duckie detection into your program to prevent such accidents. You modify your program and it now looks like this:

```
img = get_image_from_camera()
pose = get_pose_from_image(img)
cmd = get_command_from_pose(pose)

if duckie_detected(img):
    cmd = EMERGENCY_STOP

run_motors(cmd)
```

You realize, however, that your Duckiebot is not at level 5 autonomy yet and you want to add manual control for difficult to navigate regions in the city. Your code now looks like this:

```
img = get_image_from_camera()
pose = get_pose_from_image(img)
cmd = get_command_from_pose(pose)

if mode == AUTONOMOUS:
    if duckie_detected(img):
        cmd = EMERGENCY_STOP
else:
    cmd = get_command_from_joystick()

run_motors(cmd)
```

It is easy to see that when you start thinking about having even more advanced modes of operation such as intersection navigation, Duckiebot detection, traffic sign detection, and auto-charging, your program will end up being a massive stack of if-else statements. What if you could split your program into different independent building blocks, one which only gets images from cameras, one which only detects duckie pedestrians, one which controlls the motors and so on. Would that help you with organizing your code in a better way? How would those blocks communicate with each other? Moreover, how do you switch from autonomous mode to manual mode while your Duckiebot is still running? And what will happen once you try to do this for advanced robots with a lot of sensors and a large number of possible behaviors?

1.2. Basics of ROS

Look at the following system

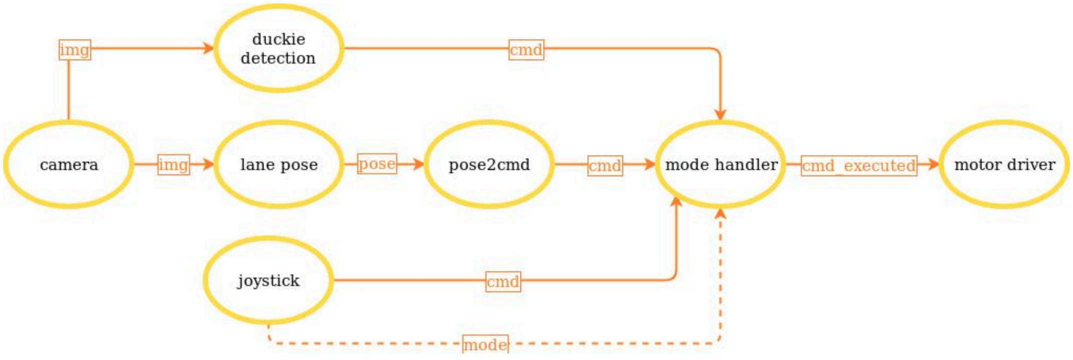


Figure 1.1

It performs exactly the same task as before. Unlike before, each of the building blocks is independent from the rest of the blocks, which means that you can swap out certain parts of the code with those written by others. You can write the lane pose extraction algorithm, while your friend works on converting that pose to a motor command. During runtime, the lane pose extractor and duckie detection algorithm run in parallel, just helping you utilize your resources better. The only missing piece to get a working system is making these blocks communicate with each other. This is where ROS comes in.

In ROS terminology, each box is a *node*, and each solid arrow connection is a *topic*. It is intuitive that each topic carries a different type of a *message*. The `img` topic has images which are matrices of numbers, whereas the `pose` topic may have rotation and translation components. ROS provides a lot of standard message types ranging from `Int`, `Bool`, `String` to images, poses, IMU measurements. You can also define your own custom messages combining different message types in one.

The nodes which send out data on a topic are called *publishers* of that topic and the ones which receive the data and use it are called *subscribers* of that topic. As you can seem from the diagram above, a node can be a publisher for one topic and subscriber for another at the same time.

You may have noticed a dashed arrow from the `joystick` node to the `mode_handler`. This represents that you can switch from manual to autonomous mode and vice versa using a button on your (virtual) joystick. Unlike sending images, which is a continuous flow of information, you will not keep switching modes all the time. ROS has a framework designed specifically for such case. This is called a *service*. Just like with messages, you can also define your own services. Here, the `mode_handler` node offers a service and the `joystick` node is the client of that service.

What manages the connections between nodes is the `rosmaster`. The `rosmaster` is responsible for helping individual nodes find one another and setting up connections between them. This can also be done over a network. Remember that you are able to see what your Duckiebot sees (unknown ref opmanual_duckiebot/read-camera-data)

previous **warning** next (36 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/read-camera-data'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

? That was because your laptop connected to the `rosmaster` of your Duckiebot. So, without knowing, you are already doing distributed robotics! It is important to keep in mind though that a single node can be managed by only one `rosmaster` at a time.

Another key building block of ROS are the *parameters* for each node. Recall when you calibrated your Duckiebot's wheels (unknown ref opmanual_duckiebot/wheel-calibration)

previous **warning** next (37 of 45) index
warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/wheel-calibration'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

or camera (unknown ref opmanual_duckiebot/camera-calib)

previous **warning** next (38 of 45) index

warning

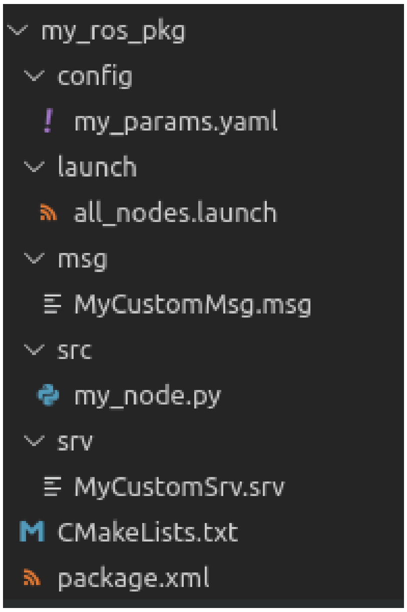
```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/camera-calib'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. These calibration parameters need to be stored somewhere so that they are not lost when your Duckiebot powers off. The ROS parameters are also very useful for configuring the nodes and therefore, the behavior of your robot. Say, that you want your lane controller to react faster, then you simply need to change the proportional gain parameter. You can hard-code that, but then changing it would require you to modify the source code. ROS offers a much nicer framework for handling hundreds of parameters for large robotics projects called `rosparam`. You can also use parameters in conjunction with services to dynamically modify their behaviour.

In ROS, code is organized in the form of *packages*. Each package is essentially a collection of nodes which perform very specific, related tasks. ROS packages also contain messages, services, and default parameter configuration files used by the nodes. A standard ROS package looks like this:



```
my_ros_pkg
  config
    my_params.yaml
  launch
    all_nodes.launch
  msg
    MyCustomMsg.msg
  src
    my_node.py
  srv
    MyCustomSrv.srv
  CMakeLists.txt
  package.xml
```

Figure 1.2

Note that the above diagram is just one of the ways to organize the flow of data. What happens actually on your Duckiebot is a little different.

1.3. Installation (Optional)

If you wish to install ROS on your computer, you can do so using this link. Please note that this might not be possible depending on your OS. Regardless of that, you should be able to use ROS through Docker, because it creates an environment which is completely independent of your OS. Quite powerful right? So much that all ROS development in Duckietown happens through Docker. This is why ROS installation on your pc is not mandatory. Keep in mind that currently all Duckietown ROS software works in ROS Noetic Ninjemys and if you want to use a native installation with your Duckiebot, you should install this version, otherwise you will likely run into compatibility issues. However, we strongly recommend using Docker for all ROS related software development.

1.4. ROS Tutorials

Tutorials on using ROS with Duckietown are covered in the next section. These tutorials are tailored to the Duckietown development process. Apart from this, we strongly recommend going through the official ROS tutorials. You should even try out the Beginner Level tutorials yourself if you have a native ROS installation. If not, read through them at least and proceed to the next section

1) Additional Reading

- ROS Graph Concepts

Unit C-2
Development in the Duckietown infrastructure

In this section, you will learn everything about creating a Duckietown-compliant Docker image with ROS.

Knowledge and activity graph

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (39 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Duckiebot initialization (unknown ref opmanual_duckiebot/setup-duckiebot)

previous **warning** next (40 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/setup-duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Docker poweruser skills

Requires: Basic understanding of ROS

Results: Developer knowledge of ROS

Contents

2.1. Basic Project Structure

In Duckietown, everything runs in Docker containers. All you need in order to run a piece of software that uses ROS in Duckietown is a Duckietown-compliant Docker image with your software in it.

A boilerplate is provided here. The repository contains a lot of files, but do not worry, we will analyze them one by one.

Click on the green button that says “Use this template”.

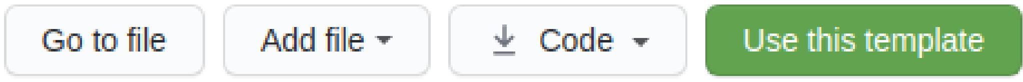
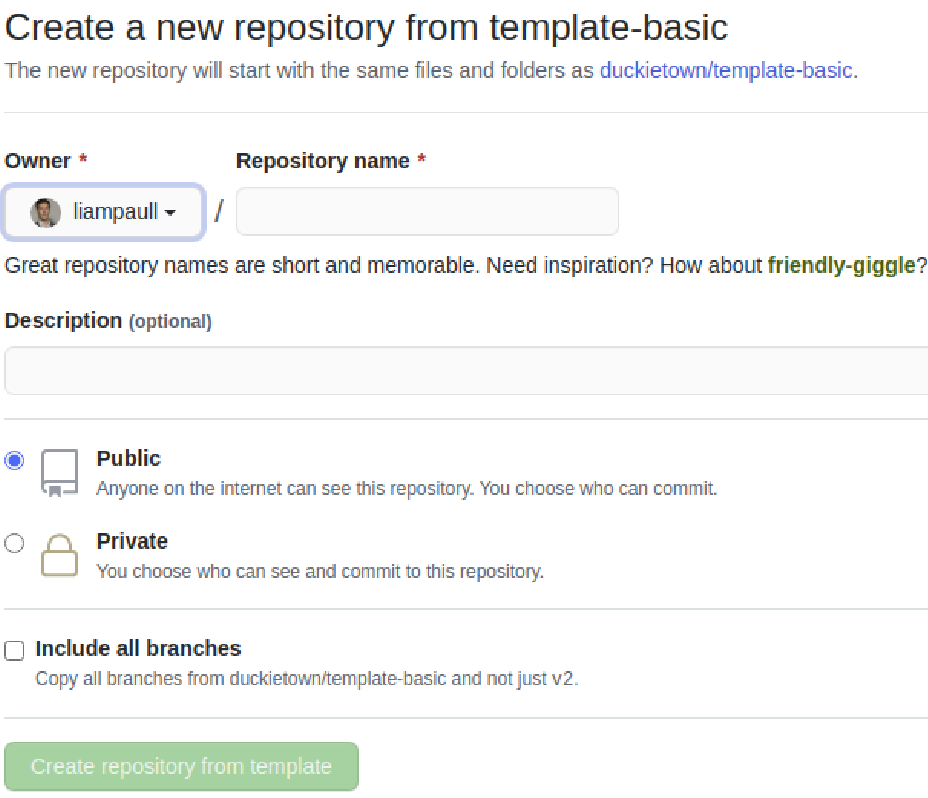


Figure 2.1

This will take you to a page that looks like the following:



Create a new repository from template-basic

The new repository will start with the same files and folders as duckietown/template-basic.

Owner * liampaull / Repository name *

Great repository names are short and memorable. Need inspiration? How about friendly-giggle?

Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

☐ Include all branches
Copy all branches from duckietown/template-basic and not just v2.

Create repository from template

Figure 2.2

Pick a name for your repository (say `my-ros-program`) and press the button *Create repository from template*. Note, you can replace `my-ros-program` with the name of the repository that you prefer, make sure you use the right name in the instructions below.

This will create a new repository and copy everything from the repository `template-ros` to your new repository. You can now open a terminal and clone your newly created repository.

```
$ git clone https://github.com/YOUR_NAME/my-ros-program
$ cd my-ros-program
```

NOTE: Replace `YOUR_NAME` in the link above with your GitHub username.

The repository contains already everything you need to create a Duckietown-compliant Docker image for your ROS program. The only thing we need to change before we can build an image from this repository is the repository name in the file `Dockerfile`. Open it using the text editor you prefer and change the first line from:

```
ARG REPO_NAME="<REPO_NAME_HERE>"
```

to

```
ARG REPO_NAME="my-ros-program"
```

Similarly update the `DESCRIPTION` and `MAINTAINER` ARGs.

Save the changes.

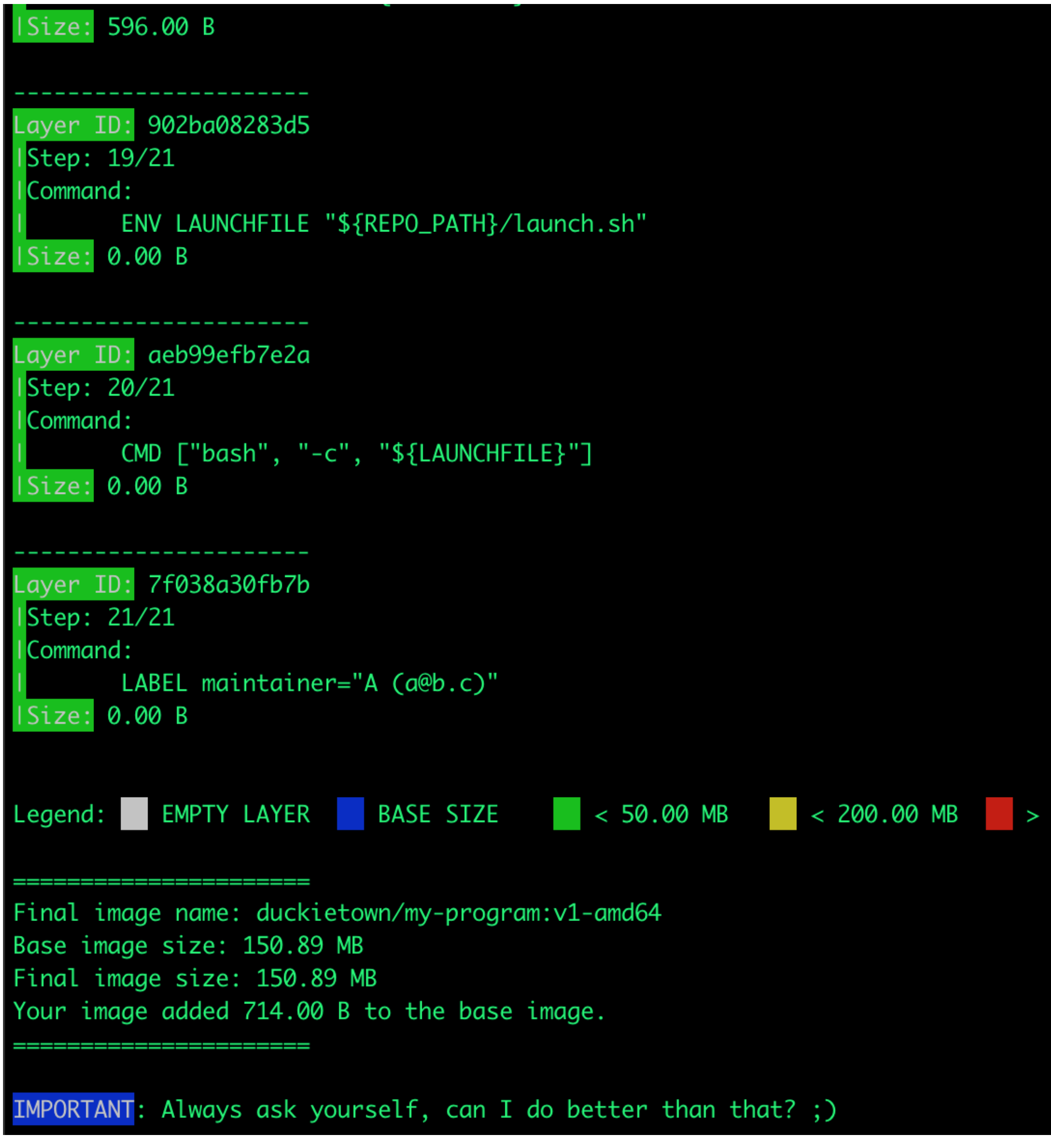
We can now build the image, even though there won't be much going on inside it until we place our code in it.

Open a terminal and move to the directory created by the git clone instruction above. Run the following command:

```
$ dts devel build -f
```

> Note: If the above command is not recognized, you will first have to install it with `dts install devel`.

If you correctly installed Docker and the duckietown-shell, you should see a long log that ends with something like the following:



```
|Size: 596.00 B

----------------------
Layer ID: 902ba08283d5
|Step: 19/21
|Command:
|       ENV LAUNCHFILE "${REPO_PATH}/launch.sh"
|Size: 0.00 B

----------------------
Layer ID: aeb99efb7e2a
|Step: 20/21
|Command:
|       CMD ["bash", "-c", "${LAUNCHFILE}"]
|Size: 0.00 B

----------------------
Layer ID: 7f038a30fb7b
|Step: 21/21
|Command:
|       LABEL maintainer="A (a@b.c)"
|Size: 0.00 B


Legend: ⬜ EMPTY LAYER  🟦 BASE SIZE   🟩 < 50.00 MB   🟨 < 200.00 MB  🟥 >

======================
Final image name: duckietown/my-program:v1-amd64
Base image size: 150.89 MB
Final image size: 150.89 MB
Your image added 714.00 B to the base image.
======================

IMPORTANT: Always ask yourself, can I do better than that? ;)
```

Figure 2.3

You can now run your container by executing the following command.

💻
```
$ dts devel run
```

This will show the following message:

```
The environment variable VEHICLE_NAME is not set. Using 'LAPTOP_HOST-
NAME'.
WARNING: robot_type file does not exist. Using 'duckiebot' as default
type.
WARNING: robot_configuration file does not exist.
= Launching app...
This is an empty launch script. Update it to launch your application.
= App terminated!
```

CONGRATULATIONS! You just built and run your first ROS-based Duckietown-compliant Docker image.

2.2. ROS Publisher on Laptop

Now that we know how to build a Docker image for Duckietown, let's put some code in one of them. We will see how to write a simple ROS program with Python, but any language supported by ROS should do it.

Open a terminal and go to the directory `my-ros-program` created above. In ROS, every ROS node must belong to a ROS *package*. ROS packages are placed inside the directory `packages` in `my-ros-program`. Let go ahead and create a directory called `my_package` inside `packages`.

```
$ mkdir -p ./packages/my_package
```

A ROS package is simply a directory containing two special files, `package.xml` and `CMakeLists.txt`. So, let's turn the `my_package` folder into a ROS package by creating these two files.

Create the file `package.xml` inside `my_package` using your favorite text editor and place/adjust the following content inside it:

```
<package>
  <name>my_package</name>
  <version>0.1.0</version>
  <description>
  This package is a test for RH3.
  </description>
  <maintainer email="YOUR_EMAIL@EXAMPLE.COM">YOUR_FULL_NAME</maintainer>
  <license>None</license>

  <buildtool_depend>catkin</buildtool_depend>
</package>
```

Create the file `CMakeLists.txt` inside `my_package` using your favorite text editor and place/adjust the following content inside it:

```
cmake_minimum_required(VERSION 2.8.3)
project(my_package)

find_package(catkin REQUIRED COMPONENTS
  rospy
)

catkin_package()
```

Now that we have a ROS package, we can create a ROS node inside it. Create the directory `src` inside `my_package` and use your favorite text editor to create the file `./packages/my_package/src/my_publisher_node.py` and place the following code inside it:

```
#!/usr/bin/env python3

import os
import rospy
from duckietown.dtros import DTROS, NodeType
from std_msgs.msg import String

class MyPublisherNode(DTROS):

    def         (self, node_name):
        # initialize the DTROS parent class
        super(MyPublisherNode, self).        (node_name=node_name,
node_type=NodeType.GENERIC)
        # construct publisher
        self.pub = rospy.Publisher('chatter', String, queue_size=10)

    def run(self):
        # publish message every 1 second
        rate = rospy.Rate(1) # 1Hz
        while not rospy.is_shutdown():
            message = "Hello World!"
            rospy.loginfo("Publishing message: '%s'" % message)
            self.pub.publish(message)
            rate.sleep()

if __name__ == '__main__':
    # create the node
    node = MyPublisherNode(node_name='my_publisher_node')
    # run node
    node.run()
    # keep spinning
    rospy.spin()
```

And don't forget to declare the file `my_publisher_node.py` as an executable, by running the command:

```
$ chmod +x ./packages/my_package/src/my_publisher_node.py
```

We now need to tell Docker we want this script to be the one executed when we run the command `docker run ...`. In order to do so, open the file `./launchers/default.sh` and replace the line

```
dt-exec echo "This is an empty launch script. Update it to launch your
application."
```

with the following lines

```
roscore &
sleep 5
dt-exec rosrun my_package my_publisher_node.py
```

Let us now re-build the image

```
$ dts devel build -f
```

Note: It is a good idea to make sure that the base image (`dt-ros-commons` in this case) is up to date. You can do so by adding the flag `--pull` to the command above, i.e., `dts devel build -f --pull`.

and run it

```
$ dts devel run
```

This will show the following message:

```
The environment variable VEHICLE_NAME is not set. Using 'b17d5c5d1855'.
... logging to /root/.ros/log/45fb649e-e14e-11e9-afd2-0242ac110004/
roslaunch-b17d5c5d1855-56.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://172.17.0.4:46725/
ros_comm version 1.15.8


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.8

NODES

auto-starting new master
process[master]: started with pid [67]
ROS_MASTER_URI=http://172.17.0.4:11311/

setting /run_id to 45fb649e-e14e-11e9-afd2-0242ac110004
process[rosout-1]: started with pid [80]
started core service [/rosout]

[INFO] [1602534741.100483]: [/my_publisher_node] Initializing...
[INFO] [1602534741.137653]: [/my_publisher_node] Health status
changed [STARTING] -> [STARTED]
[INFO] [1602534741.139893]: Publishing message: 'Hello World!'
[INFO] [1602534742.141385]: Publishing message: 'Hello World!'
[INFO] [1602534743.141426]: Publishing message: 'Hello World!'
[INFO] [1602534744.141346]: Publishing message: 'Hello World!'
```

CONGRATULATIONS! You just built and run your own Duckietown-compliant ROS publisher!

If you want to stop it, just use Ctrl+C.

2.3. ROS Publisher on Duckiebot

Now that we know how to package a piece of software into a Docker image for Duckietown, we can go one step further and write code that will run on the robot instead of our laptop.

This part assumes that you have a Duckiebot up and running with hostname *MY_ROBOT*. Of course, you don't need to change the hostname to *MY_ROBOT*, just replace it with your robot name in the instructions below. You can make sure that your robot is ready by executing the command

```
$ ping MY_ROBOT.local
```

If you can ping the robot, you are good to go.
Let us go back to our node file `my_node.py` and change the line:

```
message = "Hello World!"
```

to,

```
message = "Hello from %s" % os.environ['VEHICLE_NAME']
```

Since `roscore` is already running on the Duckiebot, we need to *remove* the following lines from `launch.sh`:

```
roscore &
sleep 5
```

We can now slightly modify the instructions for building the image so that the image gets built directly on the robot instead of your laptop or desktop machine. Run the command:

```
$ dts devel build -f -H MY_ROBOT.local
```

As you can see, we added the argument `-H MY_ROBOT.local`, which tells Docker to build the image on your *MY_ROBOT* instead of your laptop or desktop machine.
Once the image is built, we can run it on the robot by running the command:

```
$ dts devel run -H MY_ROBOT.local
```

If everything worked as expected, you should see the following output,

```
The environment variable VEHICLE_NAME is not set. Using 'riplbot01'.
[INFO] [1569609192.728583]: [/my_node] Initializing...
[INFO] [1569609192.747558]: Publishing message: 'Hello from riplbot01'
[INFO] [1569609193.749251]: Publishing message: 'Hello from riplbot01'
[INFO] [1569609194.749195]: Publishing message: 'Hello from riplbot01'
```

CONGRATULATIONS! You just built and run your first Duckietown-compliant and Duckiebot-compatible ROS publisher.

2.4. ROS Subscriber on Duckiebot

Now that we know how to create a simple publisher, let's create a subscriber which can receive these messages.

Let us go back to our `src` folder and create a file called `my_subscriber_node.py` with the following content:

```
#!/usr/bin/env python3

import os
import rospy
from duckietown.dtros import DTROS, NodeType
from std_msgs.msg import String

class MySubscriberNode(DTROS):

    def         (self, node_name):
        # initialize the DTROS parent class
        super(MySubscriberNode, self).        (node_name=node_name,
node_type=NodeType.GENERIC)
        # construct publisher
        self.sub = rospy.Subscriber('chatter', String, self.callback)

    def callback(self, data):
        rospy.loginfo("I heard %s", data.data)

if __name__ == '__main__':
    # create the node
    node = MySubscriberNode(node_name='my_subscriber_node')
    # keep spinning
    rospy.spin()
```

Once again, don't forget to declare the file `my_subscriber_node.py` as an executable, by running the command:

```
$ chmod +x ./packages/my_package/src/my_subscriber_node.py
```

Then add the following line in `./launchers/default.sh`

```
dt-exec rosrun my_package my_subscriber_node.py
```

after

```
dt-exec rosrun my_package my_publisher_node.py
```

Build the image on your Duckiebot again using

```
$ dts devel build -f -H MY_ROBOT.local
```

Once the image is built, we can run it on the robot by running the command

```
$ dts devel run -H MY_ROBOT.local
```

You should see the following output

```
[INFO] [1569750046.911664]: [/my_publisher_node] Initializing...
[INFO] [1569750046.914195]: [/my_subscriber_node] Initializing...
[INFO] [1569750046.924943]: Publishing message: 'Hello from riplbot01'
[INFO] [1569750047.926225]: Publishing message: 'Hello from riplbot01'
[INFO] [1569750047.928526]: I heard Hello from riplbot01
[INFO] [1569750048.926269]: Publishing message: 'Hello from riplbot01'
```

CONGRATULATIONS! You just built and run your first Duckietown-compliant and Duckiebot-compatible ROS subscriber.

As a fun exercise, open a new terminal and run (without stopping the other process

```
$ dts start_gui_tools MY_ROBOT
```

and then inside it, run

```
$ rqt_graph
```

Have you seen a graph like this before?

2.5. Launch files

You edited the `launch.sh` file to remove

`roscore &` when it was already running. What if there was something which starts a new `rosmaster` when it doesn't exist?

You also added multiple `rosrun` commands to run the publisher and subscriber. Now imagine writing similar shell scripts for programming multiple robot behaviors. Some basic nodes such as a camera or a motor driver will be running in all operation scenarios of your Duckiebot, but other nodes will be added/removed to run specific behaviors (e.g. lane following with or without obstacle avoidance). You can think of this as an hierarchy where certain branches are activated optionally.

You can obviously write a "master" `launch.sh` which executes other shell scripts for heirarchies. How do you pass parameters between these scripts? Where do you store all of them? What if you want to use packages created by other people?

ROS again saves the day by providing us with a tool that handles all this! This tool is called roslaunch.

In this section, you will see how to use a ROS launch file to start both the publisher and subscriber together.

Create a folder called `launch` inside your package and then create a file inside the folder called `multiple_nodes.launch` with the following content

```
<launch>

  <node pkg="my_package" type="my_publisher_node.py" name="my_publish-
er_node" output="screen"/>
  <node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node"  output="screen"/>

</launch>
```

Then replace the following lines inside `launch.sh` file

```
rosrun my_package my_node.py &
rosrun my_package my_node_subscriber.py
```

with

```
roslaunch my_package multiple_nodes.launch
```

Build and run the image again like above. You should get the same result.

You can read more about how to interpret launch files here.

2.6. Namespaces and Remapping

If you went through the above link on launch files, you might have come across the terms namespaces and remapping. Understanding namespaces and remapping is very crucial to working with large ROS software stacks.

Consider you have two Duckiebots - `donald` and `daisy`. You want them to communicate with each other so you use one `rosmaster` for both the robots. You have two copies of the same node running on each of them which grabs images from the camera and publishes them on a topic called `/image`. Do you see a problem here? Would it not be better if they were called `/donald/image` and `/daisy/image`? Here `donald` and `daisy` are ROS namespaces.

What if you were dealing with a robot which has two cameras? The names `/daisy/camera_left/image` and `/daisy/camera_right/image` are definitely the way to go. You should also be able to do this without writing a new Python file for the second camera.

Let's see how we can do this. First of all, we need to make sure that all the topics used by your Duckiebot are within its namespace.

Edit the `./packages/my_package/launch/multiple_nodes.launch` to look like this:

```
<launch>

  <group ns="$(arg veh)">

    <node pkg="my_package" type="my_publisher_node.py" name="my_publish-
er_node" output="screen"/>
    <node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node"  output="screen"/>

  </group>

</launch>
```

Then edit the roslaunch command in `./launch.sh` as follows:

```
roslaunch my_package multiple_nodes.launch veh:=$VEHICLE_NAME
```

Build and run the image. Once again run `rqt_graph` like above. What changed?

As a next step, we need to ensure that we can launch multiple instances of the same node with different names, and publishing topics corresponding to those names. For example, running two camera nodes with names `camera_left` and `camera_right` respectively, publishing topics `/my_robot/camera_left/image` and `/my_robot/camera_right/image`.

Notice how the `node` tag in the launch file has a `name` attribute. You can have multiple `node` tags with different names for the same python node file. The name provided here will override the name you give inside the python file for the node.

Edit the `./packages/my_package/launch/multiple_nodes.launch` file to have two publishers and two subscribers as below:

```
<launch>

  <group ns="$(arg veh)">

    <node pkg="my_package" type="my_publisher_node.py" name="my_publish-
er_node_1" output="screen"/>
    <node pkg="my_package" type="my_publisher_node.py" name="my_publish-
er_node_2" output="screen"/>
    <node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen"/>
    <node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_2"  output="screen"/>

   </group>

</launch>
```

Check `rqt_graph`. All communications are happening on one topic. You still cannot differentiate between topics being published by multiple nodes. Turns out doing that is very simple. Open the file `./packages/my_package/src/my_publisher_node.py` and edit the declaration of the publisher from

```
...
        self.pub = rospy.Publisher('chatter', String, queue_size=10)
...
```

to

```
...
        self.pub = rospy.Publisher('~chatter', String, queue_size=10)
...
```

All we did was add a tilde(`~`) sign in the beginning of the topic. Names that start with a `~` in ROS are private names. They convert the node's name into a namespace. Note that since the nodes are already being launched inside the namespace of the robot, the node's namespace would be nested inside it. Read more about private namespaces here

Do this for the subsciber node as well. Run the experiment and observe `rqt_graph` again. This time, switch the graph type from `Nodes only` to `Nodes/Topics (all)` and uncheck `Hide: Dead sinks` and `Hide: Leaf topics`. Play with these two "Hide" options to see what they mean.

All looks very well organized, except that no nodes are speaking to any other node. This is where the magic of remapping begins.

Edit the `./packages/my_package/launch/multiple_nodes.launch` file to contain the following:

```
<launch>

  <group ns="$(arg veh)">

    <node pkg="my_package" type="my_publisher_node.py" name="my_publish-
er_node_1" output="screen"/>
    <node pkg="my_package" type="my_publisher_node.py" name="my_publish-
er_node_2" output="screen"/>

    <node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen">
        <remap from="~/chatter" to="/$(arg veh)/my_publisher_node_1/
chatter"/>
    </node>

    <node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_2"  output="screen">
        <remap from="~/chatter" to="/$(arg veh)/my_publisher_node_2/
chatter"/>
    </node>

   </group>

</launch>
```

Check `rqt_graph`. Does it make sense?

Now, replace

```
<node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen">
    <remap from="~/chatter" to="/$(arg veh)/my_publisher_node_1/chat-
ter"/>
</node>
```

with

```
<node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen">
    <remap from="~/chatter" to="my_publisher_node_1/chatter"/>
</node>
```

Does it still work? Why?

How about if you replace it with this:

```
<node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen">
    <remap from="~/chatter" to="/my_publisher_node_1/chatter"/>
</node>
```

How about this?

```
<remap from="my_subscriber_node_1/chatter" to="my_publisher_node_1/chat-
ter"/>
<node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen"/>
```

Or this?

```
<remap from="~my_subscriber_node_1/chatter" to="~my_publisher_node_1/
chatter"/>
<node pkg="my_package" type="my_subscriber_node.py" name="my_sub-
scriber_node_1"  output="screen"/>
```

Can you explain why some of them worked, while some did not?

2.7. Multi-agent Communication

In this subsection, you will learn how to communicate between your laptop and the Duckiebot using ROS. Start by verifying that Portainer is running.

Next, ping your Duckiebot to find its IP address:

```
$ ping MY_ROBOT.local
```

Note down the address. Next, find the IP address of your computer. Note that you may have multiple IP addresses depending on how many networks you are connected to. If you have a Linux computer, you can find your IP using:

```
$ ifconfig
```

From the output, extract the IP address of the interface from which you are connected to your Duckiebot. For example, if you and your Duckiebot are both connected through WiFi, find your IP address from the WiFi connection.

Run the following command:

```
$ docker run -it --rm --net host duckietown/dt-ros-commons:daffy
/bin/bash
```

Right now, you are inside a ROS-enabled container which is connected to the `rosmas-`

`ter` running on your laptop. But you want to connect to the `rosmaster` on your duckiebot. To do this, inside the container, run:

💻
```
$ export ROS_MASTER_URI=http://MY_ROBOT_IP:11311/
$ export ROS_IP=MY_IP
```

Replace *MY_ROBOT_IP* and *MY_IP* with the IP addresses extracted above, in that order. More information about these environment variables here.

Now, run:

💻
```
$ rostopic list
```

You should see topics from your Duckiebot appearing here. Voilà! You have successfully established connection between your laptop and Duckiebot through ROS!

Are you confused about the `11311` above? You should not be. This is simply the default port number that ROS uses for communication. You can change it for any other free port.

UNIT C-3

Working with logs

Robotics is innately married to hardware. However, when we develop and test our robots' software, it is often the case that we don't want to have to waste time to test on hardware after every small change. With bigger and more powerful robots, it might be the case that a software can result in a robot actuation that breaks it or even endanger human life! But if one can evaluate how a robot or a piece of code would behave before deploying on the actual platform then quite some headaches can be prevented. That is why working in simulation and from logs is so important in robotics. In this section you will learn how to work with logs in ROS.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (41 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Docker poweruser skills

Requires: Developer knowledge of ROS

Results: Reading and processing bag files

Contents

3.1. Rosbag

A bag is a file format in ROS for storing ROS message data. Bags, named so because of their `.bag` extension, have an important role in ROS. Bags are typically created by a tool like `rosbag`, which subscribes to one or more ROS topics, and stores the sequence of messages in a file as it is received. These bag files can be played back in ROS ith the same topics that were recorded, or even using remapping to new topics. When a bag

file is replayed the temporal order of the different messages is always kept.
Please go through this link for more information.

3.2. Rosbag: Recording

You can use the following command to record bag files

```
$ rosbag record TOPIC_1 TOPIC_2 TOPIC_3
```

or simply

```
$ rosbag record -a
```

to record all messages being published.

> **Note:** Be careful on recording all the messages published in a ROS system. There might be quite a lot of topics creating very by bag files very quickly, especially using images.

3.3. Rosbag Python API: Reading

The following code snippet is a basic usage of the `rosbag` API to read bag files:

```
import rosbag
bag = rosbag.Bag('test.bag')
for topic, msg, t in bag.read_messages(topics=['chatter', 'numbers']):
    print msg
bag.close()
```

3.4. Rosbag Python API: Writing

The following code snippet is a basic usage of the `rosbag` API to create bag files:

```
import rosbag
from std_msgs.msg import Int32, String

bag = rosbag.Bag('test.bag', 'w')

try:
    s = String()
    s.data = 'foo'

    i = Int32()
    i.data = 42

    bag.write('chatter', s)
    bag.write('numbers', i)
finally:
    bag.close()
```

3.5. Exercises

All containers in the exercises below should be run on your laptop, i.e. without `-H` *MY_ROBOT*`.local`.

Exercise 18. Record bag file.

Using the following concepts,

- Getting data in and out of your container
- Communication between laptop and Duckiebot

create a Docker container on your laptop with a folder mounted on the container. You can use the image `duckietown/dt-ros-commons:daffy`. This time, however, instead of exporting the `ROS_MASTER_URI` and `ROS_IP` after entering the container, do it directly with the `docker run` command specifying environment variables. You already know how from here.

Run the lane following demo (unknown ref opmanual_duckiebot/demo-lane-following)

previous **warning** next (42 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/demo-lane-following'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. Once your Duckiebot starts moving, record the camera images and the wheel commands from your Duckiebot using `rosbag` in the container you just created (the one with the folder mounted). In order to save the bags once the container is stopped you should record them in the mounted folder. To do that navigate to the mounted folder using the `cd` command and then run

```
$ rosbag record /MY_ROBOT/camera_node/image/compressed
/MY_ROBOT/wheels_driver_node/wheels_cmd
```

Record the bag file for 30 seconds and then stop the recording using Ctrl+C. Use the `rosbag info filename.bag` command to get some information about the bag file. If the bag does not have messages from both the topics, check if you ran the container correctly(you can easily check that a topic is published using the `rostopic echo` functionality from within the container).

Stop the demo before proceeding.

Exercise 19. Analyze bag files.

Download this bag file.

Start by creating a new repository from the template, like in section C-2. Inside, the `./packages` folder, create a python file for this exercise. You do not need to create a ros package for this, you can simply launch a python script as you did in RH2. This is because reading a bag file does not actually require ROS, however, you can still choose to do so if you want. Using the following concepts,

- Getting data in and out of your container
- Creating a basic Duckietown ROS enabled Docker image

create a Docker image which can analyze bag files and produce an output similar to the one shown below. The min, max, average, and median values printed are statistics of the time difference between two consecutive messages. The `NNN` and `N.NN` are just placeholders, eg. `NNN` could be 100 and `N.NN` could be 0.05.

```
/tesla/camera_node/camera_info:
  num_messages: NNN
  period:
    min: N.NN
    max: N.NN
    average: N.NN
    median: N.NN

/tesla/line_detector_node/segment_list:
  num_messages: NNN
  period:
    min: N.NN
    max: N.NN
    average: N.NN
    median: N.NN

/tesla/wheels_driver_node/wheels_cmd:
  num_messages: NNN
  period:
    min: N.NN
    max: N.NN
    average: N.NN
    median: N.NN
```

Note: Make sure to mount the folder containing the bag file to the Docker container, instead of copying it.

Run the same analysis with the bag file you recorded in the previous exercise.

Exercise 20. Processing bag files.

Use the bag file which you recorded earlier for this exercise. Using the following concepts,

- Getting data in and out of your container
- Creating a basic Duckietown ROS enabled Docker image
- Converting between ROS Images and OpenCV Images

create a Docker image which can process a bag file. Essentially, you will extract some data from a bag file, process it, and write the results to a new bag file. Once again, create a new repository, and the necessary python file for this exercise inside the `./packages` folder. For the image message in the bag file, do the following:

- Extract the timestamp from the message
- Extract the image data from the message
- Draw the timestamp on top of the image
- Write the new image to the new bag file, with the same topic name, same timestamp, and the same message type as the original message

The new bag file should be generated in the mounted folder.

To verify your results, create a docker container exactly like you did in exercise 18. Make sure you place your processed bag file in the folder being mounted. Run the following command:

```
$ rosbag play processed_bag.bag --loop /MY_ROBOT/cam-
era_node/image/compressed:=/new_image/compressed
```

In a new terminal, use `dts start_gui_tools` to open a container connected to your robot and run `rqt_image_view` inside it. Can you see `/new_image/compressed`?

Stop the `rosbag play` using CTRL+C and now run the following command inside the same container:

```
$ rosbag play processed_bag.bag --loop
```

Again, use `start_gui_tools` but this time check `/MY_ROBOT/camera_node/image/compressed`. What's going on? Why? What does the last part of the first command do?

UNIT C-4
Robot behaviour with ROS

In this section you will extend some concepts already touched in an earlier exercise to work with ROS.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Laptop setup (unknown ref opmanual_duckiebot/laptop-setup)

previous **warning** next (43 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/laptop-setup'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Duckiebot initialization (unknown ref opmanual_duckiebot/setup-duckiebot)

previous **warning** next (44 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#op-
manual_duckiebot/setup-duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

Requires: Docker poweruser skills

Requires: Developer knowledge of ROS

Results: Basic robot behavior with ROS

4.1. ROS based color detector

Exercise 21. Converting the color detector to ROS nodes.

Using the following concepts:

- Creating a basic Duckietown ROS Publisher
- Creating a basic Duckietown ROS Subscriber
- Launch Files
- Namespaces and remapping
- Multi agent communication

- Recording bag files

Do the following:

- Create two repositories from the ROS template.
- Add all your python dependencies to the different `./dependencies-py.txt` files
- In the first one, add the code to extract images using your specific camera hardware(PiCamera or NVIDIA Jetson Nano camera) and publish it on a topic. This node will run on your Duckiebot and the node should run using a launch file. Remember to turn off all container that are instances of `duckiebot-interface` image and any other container which can use the camera.
- In the second one, add code to subscribe to that topic and extract color. Using concepts from roslaunch, use your `.launch` file to launch two nodes instances using the same script. This means that you are not allowed to have different Python files for each node. The first node detects the color red and the second detects yellow. You should use params within your `node` tag to let your detector know whether it is supposed to detect red/yellow. These nodes will run on your laptop. Once again, pass the required environment variables to connect the container on your laptop to the rosmaster of your Duckiebot using `docker run`.
- You should publish some debug images from within the color detection node. These debug images should have rectangles drawn in the region where the colors are detected. Note that we are not looking for perfect color detectors, as long as they produce reasonable output. You can draw multiple rectangles in the image if the multiple regions in the image have the requred color.
- If you are using `sensor_msgs/CompressedImage`, make sure that your image topic names end with `/compressed`. For example, instead of naming the topic `/my_image`, name it `/my_image/compressed`
- Record a bag file containing the original and debug images.

A sample debug image stream for the yellow color detector is shown here:

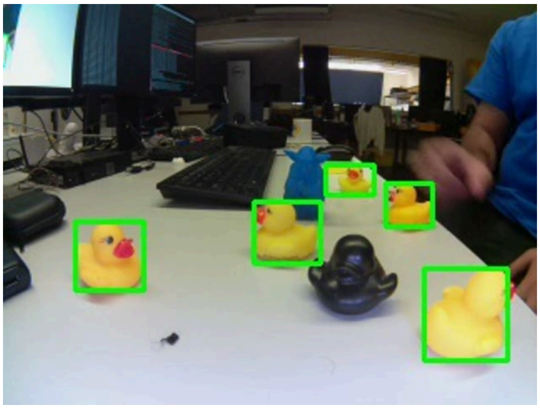


Figure 4.1. Sample Yellow Color Detector

Part D

[RH4] Implementing Basic Robot Behaviors

You are already a master of Docker and ROS and you can make small ROS programs that run on your robot. This is pretty nice but does it mean that you need to write everything from scratch if you want to change or improve an existing demo or functionality? Not the least bit!

Adding functionality to your Duckiebot while reusing the ROS nodes that are already implemented is incredibly easy and intuitive. That is where ROS and Docker really come in handy. In this module, we will do exactly that. We will use the already existing ROS nodes that control the camera, wheels, and LEDs of your robot and will implement a Braitenberg vehicle controller on top of them. But first, we will take a look at how Duckietown's code is organized.

Contents

UNIT D-1
Duckietown code structure

In order to develop new functionality within the Duckietown eco-system you need to know how the existing code is structured. This module will introduce you to the top-level structure and the references that can help you to find out more.

While on the outside Duckietown seems to be all about a simple toy car with some duckies on top, once you dive deeper you will find out that it is much bigger on the inside (just like a TARDIS). It's not only about cars, but also boats and drones. And you can run the same code on a real Duckiebot, in simulation, or in a competitive AI Driving Olympics environment. You can also use some of the dozens of projects done before. As we clearly cannot cover everything in a concise way, this module will instead focus only on the code that runs on a Duckiebot during the standard demos, e.g. Lane Following and Indefinite Navigation.

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Docker basics

Requires: ROS basics

Results: Knowledge of the software architecture on a Duckiebot

Contents

1.1. Main images and repositories

You probably noticed three container and image names popping up when you were running the demos, calibrating your Duckiebot, or developing some of the previous exercises: `dt-duckiebot-interface`, `dt-car-interface`, and `dt-core`. You probably wonder why there are three of these and what each one of them does?

Let's first look at the bigger picture: The container hierarchy in Duckietown.

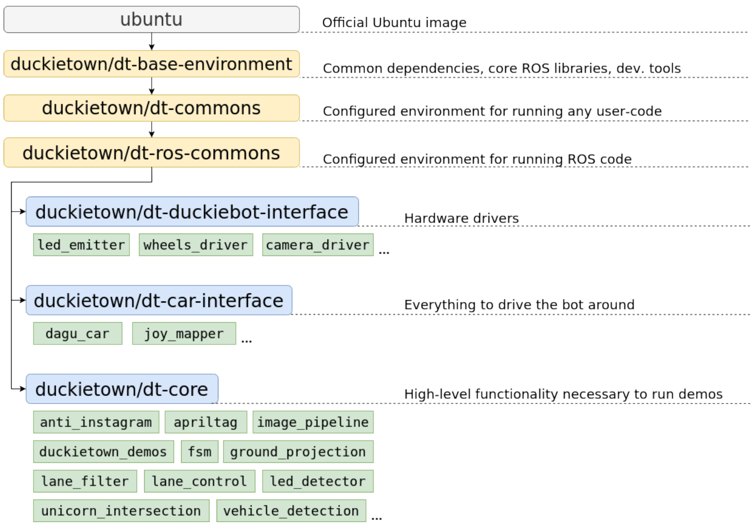

ubuntu
Official Ubuntu image
duckietown/dt-base-environment
Common dependencies, core ROS libraries, dev. tools
duckietown/dt-commons
Configured environment for running any user-code
duckietown/dt-ros-commons
Configured environment for running ROS code
duckietown/dt-duckiebot-interface
Hardware drivers
led_emitter
wheels_driver
camera_driver
...
duckietown/dt-car-interface
Everything to drive the bot around
dagu_car
joy_mapper
...
duckietown/dt-core
High-level functionality necessary to run demos
anti_instagram
apriltag
image_pipeline
duckietown_demos
fsm
ground_projection
lane_filter
lane_control
led_detector
unicorn_intersection
vehicle_detection
...

Figure 1.1. Docker image hierarchy

As you can see in the above image, all three of the containers actually inherit the same container. Recall that 'inheritance' in a Docker images means that the 'child' image has a `FROM` statement with the 'parent' image in its Dockerfile. We typically say that the 'child' *is based on* 'the parent'.

The image on which everything is based is `ubuntu`. It is simply the official Ubuntu image, with no added perks. Ubuntu 20.04 (Focal) is used for the `daffy` version of the Duckietown stack. Of course, as you can imagine, it is missing many key features that we would need. Also, it needs to be properly configured in order to work correctly with our software.

The `duckietown/dt-base-environment` adds many of the core libraries and configurations that we need. It installs development tools such as `vim`, `git`, `nano` and libraries for handling `i2c` devices, processing images, and efficiently doing linear algebra. It adds compilers, linkers, and libraries necessary for the compiling/building of software from source. Furthermore, we add `pip` and a bunch of handy `python3` libraries, such as `numpy`, `scipy`, `matplotlib`, and `smbus` (used to communicate with motors, LEDs, etc). Finally, `duckietown/dt-base-environment` also provides the core ROS libraries, including `rospy`: ROS's Python bindings. The version of `ROS` used for the `daffy` version of the Duckietown stack is ROS Noetic Ninjemys.

Then, `duckietown/dt-commons` builds on top of `duckietown/dt-base-environment`. We provide a number of Duckietown libraries here that deal with files handling, infrastructure communication, and everything else that makes our development tools run smoothly. This image configures the environment so that the hostname resolution is correctly performed also, and ensures that the environment variables pertaining to the type of the robot, its hardware, and its configuration are all properly set. It also makes sure that all Python libraries are discoverable, and that ROS is setup correctly.

Building on top of it we have `duckietown/dt-ros-commons`, which has everything you need in order to start developing code that directly works on your Duckiebot. However, as there are a few components that all Duckietown ROS nodes share, it is convenient to package them in an image. These are `duckietown-utils` (a library with a number of useful functions), `duckietown_msgs` (a ROS package that contains all the ROS message types used in Duckietown), and `DTROS`. `DTROS` is a 'mother' node for all other nodes in Duckietown. You have already seen it while working with publishers and subscribers in RH3, but we will look at it in more detail soon.

The `duckietown/dt-ros-commons` is also the place where we keep protocols that are key for the communication between nodes found in different repositories. By placing them here, we ensure that all repositories work with the exact same protocol, and hence we prevent communication issues. Currently, the only protocol there is `LED_protocol`, which is used by the `led_emitter_node` in `dt-duckiebot-interface`, which emits LED-encoded messages, and by the `led_detector_node` in `dt-core`, which interprets the messages encoded in the LED flashing of other robots.

Finally, `duckietown/dt-ros-commons` packs another handy node: the `ros_http_api_node`. It exposes the ROS environment as an HTTP API. The ROS HTTP API runs by default on any Duckietown device and allows access to ROS topics, parameters, services, nodes, etc, over HTTP, which is an extremely portable interface. This is the technology behind our web-based interfaces that communicate with ROS, such as the Duckietown Dashboard.

We finally can focus on `dt-duckiebot-interface`, `dt-car-interface`, and `dt-core`. The first, `dt-duckiebot-interface`, contains all the hardware drivers you need to operate your Duckiebot. In particular these are the drivers for the camera (in the `camera_driver` package), the ones for the motors (`wheels_driver`), and the LED drivers (`led_emitter`). Thanks to these nodes, you don't need to interact with low level code to control your Duckiebot. Instead, you can simply use the convenient ROS topics and services provided by these nodes.

The `dt-car-interface` image provides additional basic functionality that is not on hardware level. It is all you need to be able to drive your Duckiebot around, in particular the parts that handle the commands sent by a (virtual) joystick (the `joy_mapper` package) and the forward and inverse kinematics that convert the desired robot movement to wheel commands (`dagu_car` package). It might not be immediately clear at first why these are not part of `dt-duckiebot-interface` or `dt-core`. In some use cases, e.g. for the demos or controlling a robot via a joystick, it is beneficial to have these two packages. For others, e.g. when deploying a completely different pipeline, e.g. end-to-end reinforcement learning, one would prefer to interact directly with the drivers. We will see more examples of use cases shortly.

The `dt-core` image provides all the high level robot behavior that you observe when running a demo. The image processing pipeline, decision-making modules, lane and intersection contollers, and many others reside there.

If you are curious to see all the ROS packages available in each of these images, you can check out the corresponding GitHub repositories:

Note: Make sure to look at the `daffy` branches of these repositories!

- `dt-base-environment`
- `dt-commons`
- `dt-ros-commons`

- `dt-duckiebot-interface`
- `dt-car-interface`
- `dt-core`

As you will see in the nodes, there's a lot of inline documentation provided. You can also access in the 'Code documentation' section here in a more readable form.

> **Note:** Unfortunately, for the moment only `dt-ros-commons`, `dt-duckiebot-interface`, and `dt-car-interface` are documented. We are working on providing similar level of documentation for `dt-core` as well.

1.2. Various configurations of the Duckietown codebase

As we already mentioned, the Duckietown codebase can be used in various configurations: on a physical robot, in simulation, as an AI Driving Olympics submission, etc. Depending on how you want to deploy or use your code, you will be using different Docker images. Here we will take a look at a some of the most common use cases.

1) Driving with a (virtual) joystick

If you only want to drive your Duckiebot around, you need the `joy_mapper` node that translates the joystick `Joy` messages to car command messages, the `kinematics` node that in turn converts these to wheel command messages, and the `wheels_driver` node that controls the motors. So the `dt-duckiebot-interface` and `dt-car-interface` images are enough.

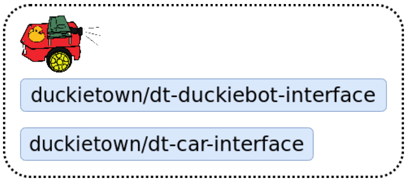

duckietown/dt-duckiebot-interface
duckietown/dt-car-interface

Figure 1.2. Driving with a (virtual) joystick

2) Driving through the Dashboard

As you have already seen, the Dashboard and the Compose interface also provide manual driving functionality. For this, one needs the same images as before, of course together with the Dashboard image itself:

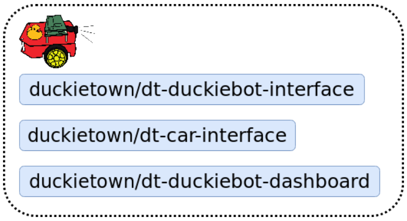

duckietown/dt-duckiebot-interface
duckietown/dt-car-interface
duckietown/dt-duckiebot-dashboard

Figure 1.3. Driving through the Dashboard

3) Running a demo on a Duckiebot

Running a demo requires to drive around together with the high-level processing and logic that reside in the `dt-core` image:

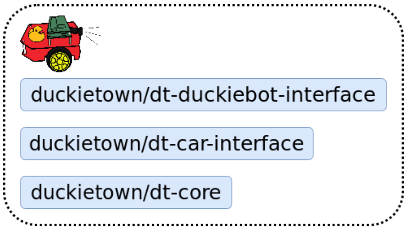

duckietown/dt-duckiebot-interface
duckietown/dt-car-interface
duckietown/dt-core

Figure 1.4. Running a demo on a Duckiebot

4) Running a demo in simulation

A demo can also be executed in simulation. In this case, instead of using the hardware drivers `dt-duckiebot-interface` provides, we substitute them with the simulator interface:

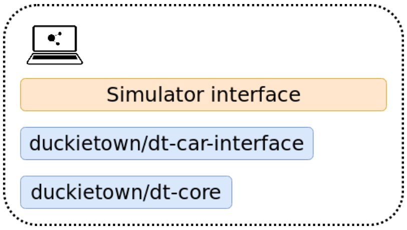


Figure 1.5. Running a demo in simulation

5) Evaluating AIDO submissions in simulation

An AI Driving Olympics submission is essentially a container that receives image data and outputs wheel commands. Therefore, it can replace the `dt-car-interface` and `dt-core` images and still use the same simulator framework. This can also be done in the cloud, and that is exactly how AIDO submissions get evaluated in simulation on the challenges server.

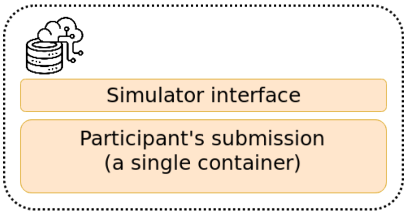


Figure 1.6. Evaluating AIDO submission in simulation

6) Evaluating AIDO submissions on a Duckiebot

The same submission image, with not a single change, can be also tested on a real Duckiebot! Simply substitute the simulator with the `dt-duckiebot-interface`. As the containers don't need to run on the same device, we can also use much powerful computers (also state-of-the-art GPUs) when testing submissions. This is the way that AI-DO submissions get evaluated in Autolabs. In this way, even if you don't have a Duckiebot, you can develop your submission in simulation, then submit it to be evaluated in simulations on the challenges server, and if it performs well, you can request remote evaluation on a real Duckiebot in a Duckietown Autolab!

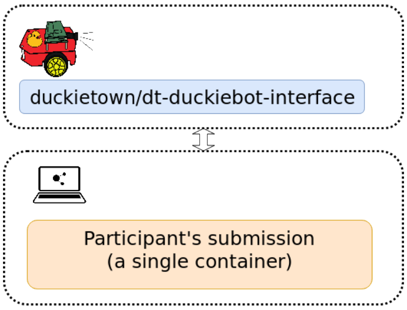


Figure 1.7. Evaluating AIDO submission on a Duckiebot

Unit D-2
Developing new Duckiebot functionality

You will now learn how to add your own code to already existing Duckietown codebase. In particular you will learn how to interface your nodes with the provided ones such that you don't have to rewrite already existing modules. Then, you will be able to master these skills by developing Braitenberg vehicle behavior on Duckiebots.

Knowledge and activity graph

Requires: Docker basics

Requires: ROS basics

Requires: Knowledge of the software architecture on a Duckiebot

Results: Skills on how to develop new code as part of the Duckietown framework

Contents

2.1. Exploring DTROS

The `DTROS` class is often referred to as the 'mother node' in Duckietown. It provides some very useful functionalities that the other nodes inherit. It has modified ROS Subscribers and Publishers which can be switched on and off. It also provides an interface to the ROS parameters of the node using it which allows dynamical changes while the node is running. For this reason we strongly suggest you to always base your nodes on `DTROS`. For some guidelines on how to structure a node in the Duckietown infrastructure, take a look at the dedicated chapter in the Developer Book. Instead of explaining all the details of `DTROS`, we instead invite you to investigate them yourself.

Exercise 22. Exploring how DTROS works.

First, take a look at the documentation of `DTROS` here. Find out how its functionalities are implemented by looking at their implementation in the `dt-ros-commons` repository here. In particular, make sure you can answer the following list of questions. To do that, it might be helpful to see how `DTROS` is being used in some of the other nodes. Take a look at `camera_node`, the `kinematics_node`, and the other nodes in `dt-duckiebot-interface` and `dt-car-interface`.

- How do you initialize the `DTROS` parent class? How do you start your node? What does `rospy.spin()` do? (*Hint: look at the nodes in* `dt_duckiebot_interface`)
- When should you redefine the `on_shutdown` method? Why do you still need to call the `on_shutdown` method of `DTROS`? (*Hint: look at the nodes in* `dt_duckiebot_interface` *and at the official ROS documentation*)

> - What is the difference between the `DTROS` `log` method and the native ROS logging? (*Hint: look at the* `DTROS` *implementation in* `dt-ros-commons`)
> - How are the parameters dynamically updated? Should you ever use `rospy.get_param()` in your node? If not, how should you access a ROS parameter? How do you initialize the parameters of your node? (*Hint: look at the nodes in* `dt_duckiebot_interface` *and at the official ROS documentation*)
> - What does the `~switch` service do? How can you use it? What is the benefit of using it?
> - What is the difference between the native ROS Subscriber and Publisher and `DTPublisher` and `DTSubscriber`?

2.2. Basic Braitenberg vehicle behavior

Through a series of exercises you will implement a very basic brightness- and color-based controller for your Duckiebot that can result in a surprisingly advanced robot behavior. In his book *Vehicles: Experiments in Synthetic Psychology*, Valentino Braitenberg describes some extremely basic vehicle designs that are capable of demonstrating complex behaviors. By using only a pair of ‘sensors’ that can only detect brightness, two motors, and direct links between the sensors and the motors, these vehicles can exhibit love, aggression, fear, foresight and many other complex traits.

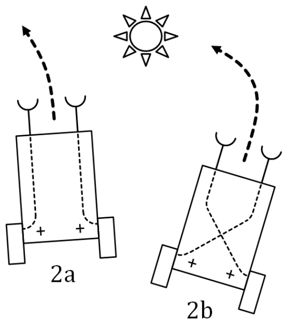


Figure 2.1. Avoiding and attracting Braitenberg behavior (illustration from [Thomas Schoch](https://commons.wikimedia.org/wiki/File:Braitenberg_Vehicle_2ab.png))

In the image above, the light intensity detected by a sensor is used proportionally to control a motor. Depending on whether each sensor is connected to the motor on the same or the opposite side, respectively avoiding or attracting behavior can be observed. These behaviors can further be combined if the robot also detects the color of the light.

Here’s an example video of how this Braitenberg behavior would look like on Duckiebots. When the light a Duckiebot sees is green, it has attracting behavior. Otherwise, it will be avoiding. By the end of this series of exercises you will be able to create similar Duckiebot controllers. Note that while this is recorded in a dark room, with a few smart tricks you can also make your robots work in well-lit spaces.

Figure 2.2

Exercise 23. Avoiding Braitenberg vechicles.

Using everything you have learnt so far, create a ROS node that implements the avoiding Braitenberg behavior. You should run this ROS node in a container running on your Duckiebot. Here are some details and suggestions you might want to take into account:

- Use the `dt-duckiebot-interface` and all the drivers it provides. In particular, you will need to subscribe to the images that the `camera_node` publishes and to publish wheel commands to `wheel_driver_node`. To do that simply make sure that the `dt-duckiebot-interface` container is running. Then, whenever you start the container with your code and `--net host` (why?), they will share their ROS Master, so that your subscribers and publishers can find each other.
- Use the nodes in `dt-duckiebot-interface` as a reference for code and documentation style. You will find a number of useful code snippets there. Also, it may be useful to visit the development book's chapter about structuring ROS nodes.
- Use the ROS template and create your package and node there. Don't forget to add the `package.xml` and `CMakeLists.txt` files, and to make your Python code executable, as explained before.
- Your controller needs to run in real time with a frequency of at least 10-12 Hz. Therefore, processing the input image at its full resolution might not be possible and you should consider reducing it. A neat way to do this is to change the configuration parameters of the `camera_node` running in `dt-duckiebot-interface`. In the template node code below that is already done for the exposure mode. Consult the ROS API docs and the code for the `CameraNode` class if you are not sure about which parameters you can change.
- For now ignore the color that your bot observes, focus only on the brightness of the image on its left and right side. If you still want to change the color of the LEDs, use the `set_pattern` service provided by the `led_emitter_node`. Its use is also documented on the ROS API docs. You do not need to call this service from inside your Python file. You would need to create a Docker container on your Duckiebot using `duckietown/dt-duckiebot-interface:daffy` as the image (why?) to run the required command. What other arguments should you use while creating this container?
- If your Duckiebot keeps on moving even after you stop your node, you will have to edit the provided `on_shutdown` method. Make sure that the last commands your

node publishes to `wheel_driver_node` are zero.

- You will need to publish `WheelsCmdStamped` messages to `wheel_driver_node`. You can see the message structure here.
- The template loads the kinematics calibration on your Duckiebot so you don't need to worry about trimming your Braitenberg controller. Simply use the provided `speedToCmd` method apply gain, trim, and the motor constant to your wheel commands. However, in order for that to happen you need to make sure to mount the `/data` folder of your Duckiebot, where all calibrations are stored, to your container. To do that, just add `-v /data:/data` to your Docker run.
- Once you have finished this exercise, you should have a Duckiebot which goes towards the left if your program senses that the right side has more brightness, and vice versa.

Template:

```
#!/usr/bin/env python3

import cv2
import numpy as np
import os
import rospy
import yaml

from duckietown.dtros import DTROS, NodeType, TopicType, DTParam,
ParamType
from sensor_msgs.msg import CompressedImage
from duckietown_msgs.msg import WheelsCmdStamped


class BraitenbergNode(DTROS):
    """Braitenberg Behaviour

    This node implements Braitenberg vehicle behavior on a Duckiebot.

    Args:
        node_name (:obj:`str`): a unique, descriptive name for the
node
            that ROS will use

    Configuration:
        ~gain (:obj:`float`): scaling factor applied to the desired
            velocity, taken from the robot-specific kinematics
            calibration
        ~trim (:obj:`float`): trimming factor that is typically used
            to offset differences in the behaviour of the left and
            right motors, it is recommended to use a value that re-
sults
            in the robot moving in a straight line when forward com-
mand
            is given, taken from the robot-specific kinematics calibra-
tion
        ~baseline (:obj:`float`): the distance between the two wheels
            of the robot, taken from the robot-specific kinematics
            calibration
        ~radius (:obj:`float`): radius of the wheel, taken from the
            robot-specific kinematics calibration
        ~k (:obj:`float`): motor constant, assumed equal for both
            motors, taken from the robot-specific kinematics calibra-
tion
        ~limit (:obj:`float`): limits the final commands sent to the
            motors, taken from the robot-specific kinematics calibra-
tion

    Subscriber:
        ~image/compressed (:obj:`CompressedImage`): The acquired cam-
era
            images
```

Exercise 24. Attracting Braitenberg vechicles.

You should be able to change the avoiding behavior of your robot into an attracting one by editing just a few lines of code. Give it a try! Once you have finished this exercise, you should have a Duckiebot which goes towards the right if your program senses that the right side has more brightness, and vice versa.

Exercise 25. Combined behavior Braitenberg vechicles.

Add a color detector to your Braitenberg controller node. If your Duckiebot sees green light (perhaps of a different Duckiebot) it should be attracted to it, otherwise it should be repelled by it.

If you have more than one robot, try to run your controller on a few of them. Set some to have green LEDs, and some red. Do you see complex behavior emerging? Changing the color of the LEDs can be done with the `set_pattern` service provided by the `led_emitter_node` in `dt-duckiebot-interface`. It is documented on the ROS API docs.

Can you devise even more complex behavior and interactions?

Part E
[RH5] Simulating and Modeling the Duckiebot

At this point you are familiar with implementing basic functionality for autonomous driving and know how to deploy this functionality to the Duckiebot. While this is a great foundation for implementing more complex behavior, it will not suffice for driving autonomously (and safely!) around Duckietown.

To be able to accomplish this, we will need more fine-tuned and robust control than what was implemented in the Braitenberg vehicle controller. As we will see, this will require us to be able to represent the state of the robot in some way, to model the dynamics of this state and to identify the parameters of this model. We will also need a way of testing complex behavior in a safe manner without having to risk the life of Duckies or without needing a physical Duckietown or Duckiebot. This is where a simulator will really come in handy.

In addition, understanding the intrinsic model representation of the differential drive robot will lead to better use of the Duckiebot. We will work on how to use the motor encoder data to derive a calibration procedure for the odometry.

Contents

UNIT E-1

Simulation in Duckietown

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Implementing Basic Robot Behaviors

Results: Experience with running and testing on the Duckietown simulator

Contents

1.1. Why simulation?

Daphne is an avid Duckietowner who loves Duckies. In her mission to “save the Duckies” from bugs in her code she used to spend a large portion of her time writing unit tests for her algorithms and ROS nodes. Some of these tests would check that the accuracy of her object detection pipeline was above a certain threshold, that the estimated offset of the Duckiebot from the lane given several input images was correct or that the output of the controller given several offsets gave sensible results. She noticed that this way of testing would fall short in several aspects:

- The number of hand-crafted edge cases was not representative of the number of situations the Duckiebot would encounter in a single drive
- Issues at the interface of these algorithms would not be caught
- To increase code coverage and maintain it, a lot of time would need to go into writing tests, mock ups, gathering and labelling test data, etc
- Quantifying controller performance was hard without having access to a model of the vehicle used to propagate the state forward in time

Daphne also found that having to charge her robot’s battery, setting up her Duckietown loop, placing her Duckiebot on the loop, connecting to it, and running the part of the pipeline that had to be tested everytime she or someone in her team wanted to merge new changes into the codebase was extremely time consuming.

More over, Daphne and her real Duckiebot only have access to a small Duckietown loop. But she wants to ensure that her algorithms work in the most complicated and busy environments of Duckietown.

All of the above were compelling reasons for Daphne to start looking at full-stack simulators that would allow her to simultaneously address the shortcomings of unit testing, the inconvenience of manual testing and the ability to test scenarios that are not possible or too risky in real life.

Luckily, she found just the right thing at the Duckietown gym.

Daphne’s story is the story of every autonomous driving company, whose mission is instead to “save the humans” and which cannot afford to make mistakes on the real roads, and which require automated integration testing tools that can be run faster-than-real-time under challenging conditions. As an example, Waymo has driven around 20 million miles on real roads, but around 15 billion miles in simulation!

1.2. The Duckietown Simulator

In this part of the exercise, you will become familiar with the Duckietown simulator by reading the setup instructions here: (unknown ref AIDO/dt-simulator)

previous **warning** (45 of 45) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#AIDO/
dt-simulator'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

and driving a robot around a virtual city. Of course, you are welcome to try the other many features of this simulator.

To demystify the simulator, here are a few tips to get started.

1) Minimal demo

To run a minimal demo of the simulator, you simply need a (virtual) environment with the gym_duckietown pip3 package installed.

To setup such an environment, the safest way is to run the following (you could also skip the virtual environment but you may have clashing packages installed):

```
$ cd ~ && virtualenv dt-sim

$ source dt-sim/bin/activate

$ pip3 install duckietown-gym-daffy
```

Now you need to create a simple python script with uses the gym-duckietown api to connect to the simulator, the API is very simple as you will see.

Create and run the following file, from within the environment you have setup above:

```
#!/usr/bin/env python3
import gym_duckietown
from gym_duckietown.simulator import Simulator
env = Simulator(
        seed=123, # random seed
        map_name="loop_empty",
        max_steps=500001, # we don't want the gym to reset itself
        domain_rand=0,
        camera_width=640,
        camera_height=480,
        accept_start_angle_deg=4, # start close to straight
        full_transparency=True,
        distortion=True,
    )
while True:
    action = [0.1,0.1]
    observation, reward, done, misc = env.step(action)
    env.render()
    if done:
        env.reset()
```

What do you observe? Does this make sense? Why is it driving straight? Can you make it drive backwards or turn? When is `done = True`? What is `observation`?

2) Driving around in the simulator

If you want to drive the robot around in simulation you might have read about the utility script `manual_control.py`. This is located in the root of the gym_duckietown repository and can be run after making sure that all the dependencies are met. Clone the repository and in the root of it run:

```
$ ./manual_control.py --env-name Duckietown-udem1-v0
```

You should be able to drive around with the arrow keys. If you are experiencing large delays and low frame rate, please replace the lines

```
pyglet.clock.schedule_interval(update, 1.0 / 30)

# Enter main event loop
pyglet.app.run()
```

by

```python
import time

...

dt = 0.01
while True:
    update(dt)
    time.sleep(dt)
```

Exercise 26. Creating a simulator ROS Wrapper.

How would we be able to exploit the powerhouse of dt-core in this simulator? By creating a ROS interface to the simulator! This will allow us to run the same code that we run on the duckiebot on the simulator.

In this exercise, you will create a ROS wrapper that maps wheel commands to actions and observations to camera images on a ROS topic.

To do so, you will leverage the skills you have obtained in the previous exercises where you used a ROS package template and created your own publisher and subscriber. This time, we encourage you to again use the ROS package template and to create a node which can both publish and subscribe to topics.

This link contains some important files that will be required to properly test your ROS wrapper. The `docker-compose.yaml` file spins up several containers at once through the simple command `docker-compose up` from the directory where the file resides. If you take a peek at the file you will see that these containers have familiar names. They are used to provide functionality to your Duckiebot, and in this scenario they are still needed to allow you to run demos in the simulator, to use the virtual joystick, etc. Inside `docker-compose.yaml` there are some lines which you will have to modify. You have to make sure that the data folder that you have downloaded from the link above is mounted on the containers so that the simulator is able to use your calibration and other configuration files.

Since now we are running our code on a fake robot (which is really our local machine) we need to modify a few things. In your `/etc/hosts` file, you will have to add the line `127.0.0.1 fakebot.local`. At the end of the Dockerfile in your ROS project (based on the Duckietown template) add the line: `ENV VEHICLE_NAME fakebot`.

Some apt packages you will need are: `freeglut3-dev`, `xvfb`

You will also need the `duckietown-gym-daffy` pip3 package

Finally, to ensure your publishers and subscribers parse the same ROS messages as the rest of the Duckietown pipeline, you might want to make use of `duckietown_msgs` (which is just a ROS package defined in `dt-ros-commons`).

Since your containers don't have a display, you will want to run these lines of bash code inside your container before running the wrapper.

```bash
dt-exec Xvfb :1 -screen 0 1024x768x24 -ac +extension GLX +render
-noreset
export DISPLAY=:1
```

3) Troubleshooting

Symptom: Despite following the above instructions, when I run my container I get an error like `pyglet.canvas.xlib.NoSuchDisplayException: Cannot connect to "None"`

Resolution: It could be that display :1 is in use or cannot be used by the docker container. Try to change the display number to a higher number (e.g. :33). Check out this post for more details.

UNIT E-2
Modeling the Duckiebot

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Terminal basics

Results: A good understanding of state representations and dynamic models for the Duckiebot

2.1. Representations

The following instructions are meant to let you test your understanding of representations from what you have learned in class through interactive Jupyter Notebook demos:

```
$ dts exercises init
```

Warning: a repository will be cloned inside the directory you are running the above command in.

```
$ cd mooc-exercises/representations

$ dts exercises notebooks
```

On your browser open the `.ipynb` file and start playing around!

2.2. Modelling of a differential drive vehicle

The following instructions are meant to let you test your understanding of modeling of a differential drive vehicle from what you have learned in class through interactive Jupyter Notebook demos:

```
$ cd mooc-exercises/kinematics

$ dts exercises notebooks
```

On your browser open the `.ipynb` file and start playing around!

UNIT E-3
Odometry with Wheel Encoders

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Understanding the DT insfrastructure

Requires: Working with ROS logs

Results: Being able to work with the wheel encoder data from the Duckiebots

3.1. Wheel Encoders

Encoders are sensors that are able to convert analog angular position or motion of a shaft into a digital signal. In Duckietown we use Hall Effect Encoders, which are able to extract the incremental change in angular position of the wheels. This is very useful, since it can be used to accurately map the position of the Duckiebot while it moves in the Duckietown.

> **Remark:** our encoders produce 135 ticks per revolution.

1) Encoders in Duckietown

The first task is to get familiar with how encoders work within the Duckietown pipeline. For this you will need a good understanding on how to build your own ROS-compliant Duckietown code. You will be required to create your own subscriber/publisher nodes, get the encoder information, and use it for the following tasks.

Similarly as with the Braitenberg Vehicles, we will be developing new Duckietown functionality. For this we will need the following:

- Creating a basic Duckietown ROS Publisher
- Creating a basic Duckietown ROS Subscriber
- Launch Files
- Namespaces and remapping

The data from each wheel encoders can be used to determined the distance travelled by each wheel:

$$\Delta X = 2\pi R N_{ticks} / N_{total}$$

•

$$\Delta X$$

is the distance travelled by each wheel;

•

$$N_{ticks}$$

is the number of ticks measured from each wheel;

•

$$N_{total}$$

is the number of ticks in one full revolution (in our case that's 135).

Below you can see the `WheelEncoderStamped.msg` from the Duckietown messages package:

```
# Enum: encoder type
uint8 ENCODER_TYPE_ABSOLUTE = 0
uint8 ENCODER_TYPE_INCREMENTAL = 1

Header header
uint32 data
uint16 resolution
uint8 type
```

Now you are ready to get started with the Duckietown encoders! Here's a useful (but definitely incomplete!) template that will help you get going. Note, this is not the only solution. Any solution which implements the required functionality will be considered correct.

Template:

```
#!/usr/bin/env python3
import numpy as np
import os
import rospy
from duckietown.dtros import DTROS, NodeType, TopicType, DTParam,
ParamType
from duckietown_msgs.msg import Twist2DStamped, WheelEncoderStamped,
WheelsCmdStamped
from std_msgs.msg import Header, Float32

class OdometryNode(DTROS):

    def         (self, node_name):
        """Wheel Encoder Node
        This implements basic functionality with the wheel encoders.
        """

        # Initialize the DTROS parent class
        super(EncoderNode, self).        (node_name=node_name,
node_type=NodeType.PERCEPTION)
        self.veh_name = rospy.get_namespace().strip("/")

        # Get static parameters
        self._radius = rospy.get_param( '/{self.veh_name}/kinemat-
ics_node/radius', 100)

        # Subscribing to the wheel encoders
        self.sub_encoder_ticks_left = rospy.Subscriber(...)
        self.sub_encoder_ticks_right = rospy.Subscriber(...)
        self.sub_executed_commands = rospy.Subscriber(...)

        # Publishers
        self.pub_integrated_distance_left = rospy.Publisher(...)
        self.pub_integrated_distance_right = rospy.Publisher(...)

        self.log("Initialized")

    def cb_encoder_data(self, wheel, msg):
        """ Update encoder distance information from ticks.
        """

    def cb_executed_commands(self, msg):
        """ Use the executed commands to determine the direction of
travel of each wheel.
        """

if __name__ == '__main__':
    node = OdometryNode(node_name='my_odometry_node')
    # Keep it spinning to keep the node alive
    rospy.spin()
    rospy.loginfo("wheel_encoder_node is up and running...")
```

Exercise 27. Get Wheel Encoder Data.
Do the following:

- Create a copy of the Duckietown ROS template.
- Create a subscriber node that is able to obtain the encoder information from both encoders.
- Run your node and the Keyboard Control node.
- Manually drive your Duckiebot around for ~10 seconds, and record a rosbag with the following parameters: encoder ticks (left and right), wheel commands.

Note: You could record the data from the topics directly with a rosbag (if Keyboard Control is running), but creating the subscriber node is necessary for the next step.

Exercise 28. Converting Wheel Encoder Information into Distance.
Do the following:

- Modify your previous code to also output the distance travelled by each wheel of the Duckiebot. Tip: this can be done by integrating the distance traveled by each wheel, but you need to take care of the direction of rotation of the wheels.
- Publish the distance travelled per wheel to a new topic.
- Manually drive your Duckiebot for ~10 seconds, and record a rosbag with the following values: wheel commands (left, right), encoder ticks (left, right), distance traveled per wheel (left, right).

3.2. Extracting Model Parameters

Now that we know how to work with the information from the wheel encoders, it is time to make something useful out of them. The task for now will be to implement some calibration functions with the encoders.

In Duckietown, Duckiebots are modeled using a differential drive model, which depends on several parameters such as the baseline (or distance between the two wheels of the robot), and the wheel radius. To simplify our lives, we assume these are constant values, the same across all Duckiebots. Nevertheless, in the real world this is often not the case, as you have already seen. To overcome this modeling limitation we usually perform wheel calibration, where we manually update some parameters from our configuration files (such as the trim value). While this helps to solve individual motor differences, it can still be improved by using the wheel encoders.

We can use the wheel encoders to obtain an accurate model, which extracts the parameters for your own Duckiebot! We will be updating the `baseline` and `radius` parameters used by the kinematics_node.py. Note that all these parameters can be modified using the `rosparam set` commands. However, to make the change permanent by writing it to the corresponding configuration file, you would need to use the `save_calibration` ros service call.

Exercise 29. Updating model parameters - Radius.

Do the following:

- Create a copy of the Duckietown ROS template or use your template from the previous exercise.
- Run Keyboard control and manually control your Duckiebot.
- Run your Duckiebot on a straight line for a fixed length (e.g. 1m, or 1 tile) and extract the value of the wheel radius. You might get slightly different values for each wheel, so take the average.
- Use `rosparam set` commands to update the radius parameter in your kinematics configuration file.
- After updating the parameters, make the change permanent by calling the `save_calibration` service to save the file: `rosservice call /DUCKIEBOT_NAME/kinematics_node/save_calibration`.

> **Remark:** the wheel radius can be found directly from the following formula.
>
> $$\Delta X = 2 * \pi * R * N_{ticks} / N_{total}$$

Deliverable: new radius value obtained. Can also be found in `/data/config/calibrations/kinematics/HOSTNAME.yaml`, after you called the `save_calibration` service.