This course 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 ..................... 5
  o Unit A-1 - Assembly documentation ........................................ 6
  o Unit A-2 - Terminal basics ..................................................... 7
  o Unit A-3 - Duckiebot Setup .................................................. 8
  o Unit A-4 - Networking basics ............................................... 10
  o Unit A-5 - Docker basics ................................................... 15
  o Unit A-6 - Basic Duckiebot operation .................................... 21

Part B - [RH2] Basic Development ............................................. 23
  o Unit B-1 - Git and GitHub .................................................... 24
  o Unit B-2 - Python programs and environments ....................... 26
  o Unit B-3 - Become a Docker Power-User ................................ 33
  o Unit B-4 - AIDO submissions .............................................. 37
  o Unit B-5 - Creating Docker containers .................................. 39

Part C - [RH3] Advanced Software Development ......................... 46
  o Unit C-1 - Introduction to ROS .......................................... 47
  o Unit C-2 - Development in the Duckietown infrastructure ........ 52
  o Unit C-3 - Working with logs ............................................ 69
  o Unit C-4 - Robot behaviour with ROS .................................. 73

Part D - [RH4] Implementing basic robot behaviors ....................... 75
  o Unit D-1 - Duckietown code structure .................................. 76
  o Unit D-2 - Developing new Duckiebot functionality ............... 81

Part E - Preliminaries .............................................................. 87
  o Unit E-1 - Linux .................................................................. 88
  o Unit E-2 - Terminal operations ........................................... 89
  o Unit E-3 - Networking ...................................................... 90
  o Unit E-4 - Accessing computers using SSH ........................... 91
  o Unit E-5 - Python ............................................................ 96
Part F - Software development basics ...............................................................97
  o Unit F-1 - Git for development .................................................................98
  o Unit F-2 - Github ........................................................................................104
  o Unit F-3 - Basic Python code conventions ...............................................107
  o Unit F-4 - Python projects ........................................................................110
  o Unit F-5 - Python IDEs ...............................................................................111
  o Unit F-6 - Explore the world of Python .....................................................113
  o Unit F-7 - Bug squashing guide .................................................................115

Part G - Basic Docker development .................................................................118
  o Unit G-1 - Introduction to Docker for Robotics and Machine Learning ....119
  o Unit G-2 - Duckiebot Development using Docker ....................................133
  o Unit G-3 - Troubleshooting ......................................................................143
  o Unit G-4 - Creating Docker containers .....................................................144
  o Unit G-5 - Continuous integration ............................................................145
  o Unit G-6 - Duckietown Shell dts ...............................................................149
  o Unit G-7 - Using the AI-DO infrastructure ...............................................152

Part H - Appendices .........................................................................................153
  o Unit H-1 - Useful Linux commands ...........................................................154
  o Unit H-2 - Raspberry-PI commands ..........................................................161
  o Unit H-3 - Working with SD Cards .............................................................162
  o Unit H-4 - Mounting USB drives ...............................................................166
  o Unit H-5 - VIM ..........................................................................................167
  o Unit H-6 - Atom .........................................................................................168
  o Unit H-7 - Liclipse .....................................................................................170
  o Unit H-8 - PyCharm ..................................................................................172
  o Unit H-9 - Byobu .......................................................................................173

Part I - Advanced Duckietown development ................................................175
  o Unit I-1 - * Duckietown Shell for Developers .........................................176

Part J - ROS development basics - Exercises ............................................181
○ Unit J-1 - Exercise: Simple data analysis from a bag .........................182
○ Unit J-2 - Exercise: Bag in, bag out..................................................184
○ Unit J-3 - Exercise: Bag thumbnails ..................................................186
○ Unit J-4 - Exercise: Instagram filters................................................188
○ Unit J-5 - Exercise: Bag instagram.....................................................190
○ Unit J-6 - Exercise: Live Instagram ...................................................191
○ Unit J-7 - Exercise: Augmented Reality ............................................192
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 .................................................................6
Unit A-2 - Terminal basics ..................................................................................7
Unit A-3 - Duckiebot Setup ................................................................................8
Unit A-4 - Networking basics ...........................................................................10
Unit A-5 - Docker basics ..................................................................................15
Unit A-6 - Basic Duckiebot operation ...............................................................21
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

Section 1.1 - Assembling the Duckiebot.................................................................6

1.1. Assembling the Duckiebot

The content of the Duckiebox including a detailed set of instructions can be found in the assembly instructions. It is advisable to read through our hardware preliminaries 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.
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**

<table>
<thead>
<tr>
<th>Requires:</th>
<th>Laptop setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires:</td>
<td>Duckietown account</td>
</tr>
<tr>
<td>Results:</td>
<td>Know how to use a terminal</td>
</tr>
</tbody>
</table>

**Contents**

Section 2.1 - Using a terminal.................................................................7
Section 2.2 - Using the Duckietown Shell ..................................................7

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.

A list of commands that are frequently used can be found in the appendix.

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, 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**

Section 3.1 - Initialization .................................................................8
Section 3.2 - Make your Duckiebot move ..................................................8
Section 3.3 - See what your Duckiebot sees .............................................8
Section 3.4 - Calibration ...........................................................................9

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.

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 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.
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 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 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

<table>
<thead>
<tr>
<th>Requires:</th>
<th>Laptop setup.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires:</td>
<td>Duckiebot initialization.</td>
</tr>
<tr>
<td>Results:</td>
<td>Fundamental networking knowledge.</td>
</tr>
</tbody>
</table>

Contents

Section 4.1 - Why do we care about networking in the first place? ..........10
Section 4.2 - How do computer networks work?........................................10
Section 4.3 - Connecting to your Duckiebot ..........................................13

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 through 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 requires 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 use this information to gain access to Duckiebots. The industry standard way of connecting to remote devices is a protocol known as SSH (Secure SHell). The 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 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 supplied 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 your 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 known 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).
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 developed wonderful tools to manage that, such as virtual environments and PyPI. Unfortunately, these tools stop short of extending their convenience 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 across 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**

<table>
<thead>
<tr>
<th>Requires:</th>
<th>Laptop setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results:</td>
<td>The very basic knowledge of using Docker</td>
</tr>
</tbody>
</table>

**Contents**

- Section 5.1 - What’s so special about containerization? .............................................15
- Section 5.2 - What is it in a Docker container? .................................................................17
- Section 5.3 - Working with Docker images .................................................................17
- Section 5.4 - Working with containers .................................................................18

**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 con-
The 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, after all). 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 system 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.

![Comparison between containers and VMs](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:

```bash
$ docker pull library/ubuntu:18.04
```
You will now be able to see the new image you pulled if you run:

```bash
$ 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:

```bash
$ 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:

```bash
$ 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.

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

Now you should see something like:

```bash
$ 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.
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**

<table>
<thead>
<tr>
<th>Requires:</th>
<th>Laptop setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results:</td>
<td>Know how to use the Dashboard, Portainer and the DT shell for demos.</td>
</tr>
</tbody>
</table>

**Contents**

Section 6.1 - Remote connection with a browser and an interface ..................21
Section 6.2 - Starting a demo using the DT shell .............................................22

**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. 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 Section 9.4 - Option 2: Using the dashboard.

You can even see what the Duckiebot is seeing Through the dashboard. You can follow the instructions from Section 10.3 - Viewing the image stream on the Dashboard 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.

**Exercise 10. Try out the lane-following demo.**

Let’s now start a lane_following demo. To do so, follow these instructions.

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.................................................................24
Unit B-2 - Python programs and environments.................................26
Unit B-3 - Become a Docker Power-User ...........................................33
Unit B-4 - AIDO submissions............................................................37
Unit B-5 - Creating Docker containers..............................................39
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   | Results: Know how to extensively use a code versioning tools, git |

Contents

- Section 1.1 - Learning git .................................................................24
- Section 1.2 - What is github .............................................................24
- Section 1.3 - Being a good git citizen ..............................................24

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.

<table>
<thead>
<tr>
<th>Exercise 11. Git tutorial.</th>
</tr>
</thead>
<tbody>
<tr>
<td>To learn how to use all of git’s functionalities, complete this tutorial.</td>
</tr>
</tbody>
</table>

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: “develop-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 does it allow for error proofing, but also it allows for someone to made a code suggestion alone, that can then be easily discussed and improved, even when it was functional to start with.
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
- **Results:** Developer knowledge of Python.

**Contents**

- Section 2.1 - Define a basic project structure ..................................................26
- Section 2.2 - Run a basic program on your Laptop..............................................29
- Section 2.3 - Run a basic program on your Duckiebot.......................................30
- Section 2.4 - Install dependencies using package managers (e.g., `apt`, `pip`) 32

### 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.

First of all, you will need to make a copy of the template in your own GitHub account. To do so, go to the repository and click on the fork button.

![Fork button](image)

**Figure 2.1**

Now that you have a copy of the template, you can create new repositories based off of it. In order to do so, go to GitHub and click on the button `[+]` at the top-right corner of the page and then click on New Repository.
You will see a page that looks like the following:

**Create a new repository**
A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

In the section **Repository template**, select **YOUR_NAME**/template-basic. Pick a name for your repository (say my-program) and press the button **Create repository**. 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
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. To do that, we need to enable development features in the duckietown-shell by running the following in a terminal:

```
$ dts install devel
```

Now, in a terminal, move to the directory created by the `git clone` instruction above and run the following command:

```
$ dts devel build -f --arch amd64
```

If you correctly installed Docker and `dts`, 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: 7f038a30f67b
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 2.4
You can now run your container by executing the following command.

```
$ docker run -it --rm duckietown/my-program:v1-amd64
```

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!

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

Deactivating services broadcast...
Done!
```

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 code.

```
$ mkdir -p ./code/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 ./code/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 `./code/my_package/my_script.py` and place the following code inside it.

```python
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 `launch.sh` and replace the line

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

with the line

```bash
dt_exec python3 -m "my_package.my_script"
```

**Note:** Always prepend `dt_exec` to the main command in `launch.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).

Let us now re-build the image:

```bash
$ dts devel build -f --arch amd64
```

and run it:

```bash
$ docker run -it --rm duckietown/my-program:v1-amd64
```

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

Before we start, we need to configure the Duckiebot to accept new code. This is necessary because the Duckiebot by default runs only code released by the Duckietown community. In order to configure the robot to accept custom code, run the following command,

```
$ dts devel watchtower stop -H MY_ROBOT.local
```

**Note:** You need to do this once and the effect will be lost when the Duckiebot reboots.

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 `--arch arm32v7 -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-program:v1
```

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 |
| Requires: Docker basics |
| Results: Advanced knowledge of using Docker images and containers |

Contents

Section 3.1 - Getting data in and out of your container ..................................................33
Section 3.2 - Docker and networking ..................................................................................34
Section 3.3 - Handling devices ..............................................................................................34
Section 3.4 - Other fancy option ............................................................................................34

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:

```
$ docker run -it -v ~:/home ubuntu
```


Run a new Ubuntu container where you mount your home directory in the container's home directory:
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. 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.

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

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

#### 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
The duckietown platform is one of many possibilities. In particular it is used for an 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**

<table>
<thead>
<tr>
<th>Requires: Laptop setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Results: Know how to participate in AIDO.</td>
</tr>
</tbody>
</table>

**Contents**

Section 4.1 - Getting started ..................................................37
Section 4.2 - Make a simple submission .....................................37
Section 4.3 - Customize a solution ...............................................38

**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 here and here.

**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. 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
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 quick-start instructions.
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
- **Requires:** Duckiebot initialization
- **Requires:** Docker basics
- **Requires:** Docker poweruser skills
- **Results:** Advanced knowledge of using Docker images and containers.

Contents

- Section 5.1 - Where do Docker containers come from? .............................................39
- Section 5.2 - Environment variables and Docker containers........................................41
- Section 5.3 - Guide to the Dockerfile keywords..............................................................42
- Section 5.4 - Creating your first functional Docker image ............................................42
- Section 5.5 - Pushing to DockerHub ..............................................................................44

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:
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:
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>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Designates the base image on top of which this container image is built.</td>
</tr>
<tr>
<td>RUN</td>
<td>Executes any shell command at run time, unless the user specifies another command. This is the default command the container will execute.</td>
</tr>
<tr>
<td>CMD</td>
<td>Sets an environment variable</td>
</tr>
<tr>
<td>ENV</td>
<td>Executes any shell command at run time, unless the user specifies another command. This is the default command the container will execute.</td>
</tr>
<tr>
<td>COPY</td>
<td>Copies file from the container to the host.</td>
</tr>
<tr>
<td>WORKDIR</td>
<td>Change the working directory.</td>
</tr>
</tbody>
</table>

### 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. The `picamera` library allows only one process to 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. 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:

```bash
FROM duckietown/dt-duckiebot-interface:daffy-arm32v7
WORKDIR /color_detector
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY color_detector.py .
CMD python ./color_detector.py
```

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

```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)
```

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 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 \texttt{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 \texttt{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/colordetector
```

Then push it to DockerHub:

```
$ docker -H $DUCKIEBOT_NAME.local push duckquackermann/colordetector
```

**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
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 .................................................................47
Unit C-2 - Development in the Duckietown infrastructure ......................52
Unit C-3 - Working with logs..................................................................69
Unit C-4 - Robot behaviour with ROS .....................................................73
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 hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. 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
- **Requires:** Duckiebot initialization
- **Requires:** Docker poweruser skills
- **Results:** Basic understanding of ROS

Contents

- Section 1.1 - Why ROS? .................................................................47
- Section 1.2 - Basics of ROS .............................................................48
- Section 1.3 - Installation (Optional) ...............................................50
- Section 1.4 - ROS Tutorials ............................................................51

1.1. Why ROS?

Your Duckiebot is a very simple robot which has only one sensor (the camera), and two actuators (the motors). You can probably write all the code for the basic functionality 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:
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:

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

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:

```python
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 mode 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 controls 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

![System Diagram](image)

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. If you don’t want to write your own driver for the camera, you could very easily use one from any ROS robot using the PiCamera.

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.

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? 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 or camera. 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. You will also need parameters in conjunction with services. (Why?)

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:

![Figure 1.2](image)

When developing a large software stack, you may also find it easier to have all messages, services, and parameter files used by all nodes running on your robot in a single package rather than spread out inside packages which use them to avoid unnecessary redefinitions. The nodes, however, remain in their own packages. (Why? Does it have something to do with the fact that multiple nodes might use the same message, etc.?)

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 what OS you use, you should be able to use ROS through Docker (Why?). All ROS development in Duck-
Duckietown happens through Docker. This is why this step is not mandatory. Keep in mind that currently all Duckietown ROS software works in ROS Kinetic Kame 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 |
| Requires: Duckiebot initialization |
| Requires: Docker poweruser skills |
| Requires: Basic understanding of ROS |
| Results: Developer knowledge of ROS |

Contents

Section 2.1 - Basic Project Structure .................................................................52
Section 2.2 - ROS Publisher on Laptop...............................................................55
Section 2.3 - ROS Publisher on Duckiebot .......................................................59
Section 2.4 - ROS Subscriber on Duckiebot ....................................................60
Section 2.5 - Launch files ................................................................................62
Section 2.6 - Namespaces and Remapping .......................................................63
Section 2.7 - Multi-agent Communication .......................................................67

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.

First of all, you will need to make a copy of the template in your own GitHub account. To do so, visit this URL and click on the fork button.

![Fork button](image)

Now that you have a copy of the template, you can create new repositories based off of it. In order to do so, go to GitHub and click on the button [+ ] at the top-right corner of the page and then click on New Repository.
You will see a page that looks like the following:

**Create a new repository**
A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

![New repository selection](image)

In the section **Repository template**, select `YOUR_NAME/template-ros`. Pick a name for your repository (say `my-ros-program`) and press the button **Create repository**. Note, you can replace `my-ros-program` with the name of the repository that you prefer. Note that no capital letters are allowed and 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"
```

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 --arch amd64
```

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

```
$ docker run -it --rm duckietown/my-ros-program:v1-amd64
```

This will show the following message:

```
The environment variable VEHICLE_NAME is not set. Using '63734f6b4e7c'.
This is an empty launch script. Update it to launch your application.
```

**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:

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

find_package(catkin REQUIRED COMPONENTS rospkg)

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_node.py` and place the following code inside it:
#!/usr/bin/env python
import os
import rospy
from duckietown import DTROS
from std_msgs.msg import String

class MyNode(DTROS):
    def __init__(self, node_name):
        # initialize the DTROS parent class
        super(MyNode, self, node_name=node_name)
        # 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 = MyNode(node_name='my_node')
    # run node
    node.run()
    # keep spinning
    rospy.spin()

And don’t forget to declare the file my_node.py as an executable, by running the command:

```
$ chmod +x ./packages/my_package/src/my_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 `launch.sh` and replace the line

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

with the following lines
Let us now re-build the image

```
$ dts devel build -f --arch amd64
```

and run it

```
$ docker run -it --rm duckietown/my-ros-program:v1-amd64
```

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 1.12.14
```

```
PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rosdistro</td>
<td>kinetic</td>
</tr>
<tr>
<td>rosversion</td>
<td>1.12.14</td>
</tr>
</tbody>
</table>
```

```
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] [1569606196.137620]: [/my_node] Initializing...
[INFO] [1569606196.148146]: Publishing message: 'Hello World!'
[INFO] [1569606197.149378]: Publishing message: 'Hello World!'
[INFO] [1569606198.149470]: Publishing message: 'Hello World!'
```

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

Before you start, you need to configure the Duckiebot to accept new code. This is necessary because the Duckiebot by defaults runs only code released by the Duckietown community. In order to configure the robot to accept custom code, run the following command,

```
$ dts devel watchtower stop -H MY_ROBOT.local
```

**Note:** You need to do this every time you reboot your Duckiebot.

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 --arch arm32v7 -H MY_ROBOT.local
```
As you can see, we changed two things, one is \texttt{--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 \texttt{-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:

\begin{verbatim}
$ docker -H MY\_ROBOT\.local run -it --rm --net=host duckietown/my-ros-program:v1
\end{verbatim}

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

![The environment variable VEHICLE\_NAME is not set. Using 'riplbot01'.](image)

\begin{verbatim}
[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'
\end{verbatim}

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

\section*{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 \texttt{my\_node\_subscriber.py} with the following content:
#!/usr/bin/env python

import os
import rospy
from duckietown import DTROS
from std_msgs.msg import String

class MyNode(DTROS):
    def __init__(self, node_name):
        # initialize the DTROS parent class
        super(MyNode, self).__init__(node_name=node_name)
        # 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 = MyNode(node_name='my_node_subscriber')
    # keep spinning
    rospy.spin()

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

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

Then edit the following line from `launch.sh`

```
rosrun my_package my_node.py
```

to

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

Build the image on your Duckiebot again using

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

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-ros-program:v1
```
You should see the following output

```
[INFO] [1569750046.911664]: [/my_node] Initializing...
[INFO] [1569750046.914195]: [/my_node_subscriber] 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
```

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

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

with

```bash
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_node.py" name="my_node" output="screen"/>
    <node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber" 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_node.py" name="my_node_1" output="screen"/>
    <node pkg="my_package" type="my_node.py" name="my_node_2" output="screen"/>
    <node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_1" output="screen"/>
    <node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_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_node.py` and edit the declaration of the publisher from

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

to

```python
...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 subscriber 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_node.py" name="my_node_1" output="screen"/>
  <node pkg="my_package" type="my_node.py" name="my_node_2" output="screen"/>

  <node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_1" output="screen">
    <remap from="/chatter" to="$(arg veh)/my_node_1/chatter"/>
  </node>

  <node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_2" output="screen">
    <remap from="/chatter" to="$(arg veh)/my_node_2/chatter"/>
  </node>

</group>

</launch>

Check rqt_graph. Does it make sense?

Now, replace

<node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_1" output="screen">
  <remap from="/chatter" to="$(arg veh)/my_node_1/chatter"/>
</node>

with

<node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_1" output="screen">
  <remap from="/chatter" to="/my_node_1/chatter"/>
</node>

Does it still work? Why?

How about if you replace it with this:

<node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_1" output="screen">
  <remap from="/chatter" to="/my_node_1/chatter"/>
</node>
How about this?

```xml
<remap from="my_node_subscriber_1/chatter" to="my_node_1/chatter"/>
<node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_1" output="screen"/>
```

Or this?

```xml
<remap from="~my_node_subscriber_1/chatter" to="~my_node_1/chatter"/>
<node pkg="my_package" type="my_node_subscriber.py" name="my_node_subscriber_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-amd64 /bin/bash
```

Right now, you are inside a ROS-enabled container which is connected to the rosmaster 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` from the IP addresses extracted above, in that or-
der. More information about these environment variables here.

Now, run:

```bash
$ rostopic list
```

You should see topics from your Duckiebot appearing here. Viola! 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.
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
- **Requires:** Docker poweruser skills
- **Requires:** Developer knowledge of ROS
- **Results:** Reading and processing bag files

## Contents

- Section 3.1 - Rosbag
- Section 3.2 - Rosbag: Recording
- Section 3.3 - Rosbag Python API: Reading
- Section 3.4 - Rosbag Python API: Writing
- Section 3.5 - Exercises

### 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 serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics.

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
to record all messages being published.

3.3. Rosbag Python API: Reading

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

```python
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:

```python
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-amd64. 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. You already know it from here.

Run the lane following demo. 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). To do that navigate to the mounted folder using the `cd` command and then run

```bash
$ 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.

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 the previous section. Inside, the `.packages` folder, create a python file for this exercise. You do not need to create a ros package for this, however, you can still choose to do so. Since reading a bag file does not require ROS, you can do this without setting the necessary environment variables. 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, e.g. `NNN` could be 100 and `N.NN` could be 0.05.

```
/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 the first exercise of this section. Make sure you place your processed bag file in the folder being mounted. Run the following command:

```bash
$ rosbag play processed_bag bag --loop /MY_ROBOT/camera_node/image/compressed:=/new_image/compressed
```

In a new terminal, use `start_gui_tools` 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:

```bash
$ 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 an earlier exercise to work with ROS.

**KNOWLEDGE AND ACTIVITY GRAPH**

- **Requires**: Laptop setup
- **Requires**: Duckiebot initialization
- **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 file ./dependencies-py.txt
- In the first one, add the code to extract images using a PiCamera and publish it on a topic. This will run on your Duckiebot. The node should run using a launch file. Remember to turn off duckiebot-interface (can exist under different names) and any other container which can use the camera.
- In the second one, add the code to subscribe to that topic and extract color. Using concepts from roslaunch, create two nodes in your .launch file. Note 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 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 pro-
duce reasonable output. You can draw multiple rectangles in the image if the multiple regions in the image have the required 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:

![Sample Yellow Color Detector](image-url)

Figure 4.1. Sample Yellow Color Detector

Repeated use of ID "fig:example-embed"

These are the locations indicated:

- repeated-use
  
  Jump to element in output file.

- original-use
  
  Jump to element in output file.

Created by function detect_duplicate_IDS in module mcdp_docs.check_missing_links.
PART D

[RH4] Implementing basic robot behaviors

You are already 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 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 .................................................................76
Unit D-2 - Developing new Duckiebot functionality .............................................81
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 deep dive 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 bot, in simulation, or in a competitive AI Driving Olympics environment. You can also use some of the dozen projects done before. As we clearly cannot cover everything in a concise way, this module will instead focus only on the code the runs on a Duckiebot during the standard demos, e.g. Lane Following and Indefinite Navigation.

**KNOWLEDGE AND ACTIVITY GRAPH**

<table>
<thead>
<tr>
<th>Requires: Docker basics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires: ROS basics</td>
</tr>
<tr>
<td>Results: Knowledge of the software architecture on a Duckiebot</td>
</tr>
</tbody>
</table>

**Contents**

Section 1.1 - Main images and repositories{status=ready}..........................76
Section 1.2 - Various configurations of the Duckietown codebase...............78

1.1. Main images and repositories{status=ready}
You probably noticed three container and image names popping up when you were running the demos, calibrating your bot, or developing some of the previous exercises: `dt-duckiebot-interface`, `dt-car-interface`, and `dt-core`. You probably wonder why are there three of these and what does each one of them do?

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

The image from which everything starts is `ros:kinetic-ros-base-xenial`. It is an official ROS image that is configured to work smoothly with ROS Kinetic. Even though this image is already extremely powerful, it is not well suited to directly work on a Duckiebot. Therefore, we add a few additional components and configure it properly and the result is `duckietown/dt-ros-kinetic-base`.

The `duckietown/dt-ros-kinetic-base` image 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 `duckitown-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. We will look at it in more detail soon.

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 bot. 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 bot. 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 bot 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 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 controllers, and many others reside there.

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

- dt-ros-kinetic-base
- 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 it 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 bot 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.

![Figure 1.2. Driving with a (virtual) joystick](duckietown/dt-duckiebot-interface)

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:
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:

```plaintext
Figure 1.3. Driving through the Dashboard
```

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:

```plaintext
Figure 1.4. Running a demo on a Duckiebot
```

5) Evaluating AIDO submission 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.

```plaintext
Figure 1.5. Running a demo in simulation
```
6) Evaluating AIDO submission 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 `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 also the way that AIDO submissions get evaluated in Robotariums. 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 Robotarium!
UNIT D-2
Developing new Duckiebot functionality

You will now learn how to add your own code to already existing Duckietown code-base. 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 how to develop new code as part of the Duckietown framework

**Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 - Exploring DTROS</td>
<td>81</td>
</tr>
<tr>
<td>2.2 - Basic Braitenberg vehicle behavior</td>
<td>82</td>
</tr>
</tbody>
</table>

### 2.1. Exploring DTROS

The DTROS class is the often referred 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 this node that allows dynamically changed while the node is running. For this reason we strongly suggest that you always base your nodes on DTROS. Instead of explaining all the details of DTROS, we instead invite you to investigate them yourself.

**Note:** Currently dt-core is not using DTROS. Nevertheless, soon the nodes there will be converted to the DTROS framework as well.

**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 its 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 wheels_driver_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 onShutdown method? Why do you still need to call the onShutdown method of DTROS? (*Hint: look at the nodes in dt_duckiebot_inter-...*)
face and at the official ROS documentation)

• What is the difference between the DTROS log method and the native ROS logging?

• How are the parameters dynamically updated? What can you do to change how often that happens? Why is updateParameters called immediately after initializing the parameters? What is the use of the parametersChanged attribute? (Hint: see the implementation in camera_node)

• 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 DT-Publisher and DTSubscriber?

2.2. Basic Braitenberg vehicle behavior

Though a series of exercises you will implement a very basic brightness- and color-based controller for your Duckiebot that can result in surprisingly advanced robot behavior. In his book Vehicles: Experiments in Synthetic Psychology, Valentino Braitenberg describes some extremely basic 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

In the image above, the light intensity detected by a sensor is used proportionately to control a motor. Depending on whether the 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 bot 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 Duck-
iebot 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.

![Image of a robot in a dark room.

**Exercise 23. Avoiding Braatenberg vehicles.**

Using everything you have learnt so far, create a ROS node that implements the avoiding Braatenberg behavior. 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 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, they will share their ROS Master, so you connecting your subscribers and publishers to the ones of `dt-duckiebot-interface` can be directly done through simple `remap` commands in your launch file.

- 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.

- 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 frequency of at least 10-12 Hz. Therefore, processing the input image at its full resolution might not be possible. Consider reducing it (and potentially using only part of 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 for the `CameraNode` class if you are not sure which parameters you can change.

- For now ignore the color that your bot observes, focus only on the brightness. 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 will need to publish `WheelsCmdStamped` messages to `wheel_driver_node`. You can see their 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.
- If your Duckiebot keeps on moving even after you stop your node, you will have to edit the provided `onShutdown` method. Make sure that the last commands your node publishes to `wheel_driver_node` are zero.

Template:
import cv2
import numpy as np
import os
import rospy
import yaml
from duckietown import DTROS
from std_msgs.msg import Float64
from sensor_msgs.msg import CompressedImage, CameraInfo
from duckietown_msgs.msg import Twist2DStamped, WheelsCmdStamped

class BraitenbergNode(DTROS):
    
    """Handles the imagery.
    
    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 results in the robot moving in a straight line when forward command is given, taken from the robot-specific kinematics calibration
    ~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 calibration
    ~limit (:obj:`float`): limits the final commands sent to the motors, taken from the robot-specific kinematics calibration
    
    Subscriber:
    ~image/compressed (:obj:`CompressedImage`): The acquired camera images
    
    Publisher:
    ~wheels_cmd (:obj:`duckietown_msgs.msg.WheelsCmdStamped`): The wheel commands that the motors will execute
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!

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
Preliminaries

This part gives pointers to learn basic skills in computing.

Contents

Unit E-1 - Linux ........................................................................................................... 88
Unit E-2 - Terminal operations .................................................................................. 89
Unit E-3 - Networking .................................................................................................. 90
Unit E-4 - Accessing computers using SSH ............................................................... 91
Unit E-5 - Python ......................................................................................................... 96
UNIT E-1

Linux

These are some resources that can help you learn Linux.

Contents

Section 1.1 - Background reading .................................................................88

1.1. Background reading

- UNIX
- Linux
- free software; open source software.
UNIT E-2

Terminal operations

Contents

Section 2.1 - Shells ................................................................. 89

2.1. Shells
3.1. Background information

**Knowledge and activity graph**

Preliminary reading:

- Basics of networking, including
  - what are IP addresses
  - what are subnets
  - how DNS works
  - how .local names work
  - ...

→ XXX (ref to find).
UNIT E-4
Accessing computers using SSH

Contents

Section 4.1 - Background reading ................................................................. 91
Section 4.2 - Installation of SSH .................................................................. 91
Section 4.3 - Local configuration ................................................................. 91
Section 4.4 - How to login with SSH and a password .............................. 92
Section 4.5 - Creating an SSH keypair ....................................................... 92
Section 4.6 - How to login without a password .......................................... 94
Section 4.7 - Fixing SSH Permissions .......................................................... 95
Section 4.8 - ssh-keygen ........................................................................... 95

4.1. Background reading

- Encryption
- Public key authentication

4.2. Installation of SSH

This installs the client:

```
$ sudo apt install ssh
```

This installs the server:

This enables the server:

4.3. Local configuration

The SSH configuration as a client is in the file 

```
~/.ssh/config
```

Create the directory with the right permissions:

```
$ mkdir ~/.ssh
$ chmod 0700 ~/.ssh
```

Edit the file:
(We suggest you use VIM to edit files; see a tutorial here.)

Then add the following lines:

```
HostKeyAlgorithms ssh-rsa
```

The reason is that Paramiko, used by `roslaunch`, does not support the ECSDA keys.

### 4.4. How to login with SSH and a password

To log in to a remote computer with user, use:

```
$ ssh remote-user@remote
```

1) Troubleshooting

**Symptom:** “Offending key error”.

If you get something like this:

```
Warning: the ECDSA host key for ... differs from the key for the IP address '...'

Offending key for IP in /home/ user/.ssh/known_hosts: line
```

then remove line `line` in `~/.ssh/known_hosts`.

### 4.5. Creating an SSH keypair

This is a step that you will repeat twice: once on the Duckiebot, and once on your laptop.

The program will prompt you for the filename on which to save the file.

Use the convention

```
/home/ username /.ssh/ username@host name
/home/ username /.ssh/ username@host name.pub
```

where:

- `username` is the current user name that you are using (`ubuntu` or your chosen one);
• **host name** is the name of the host (the Duckiebot or laptop);

An SSH key can be generated with the command:

```
$ ssh-keygen -h
```

The session output will look something like this:

```bash
Generating public/private rsa key pair.
Enter file in which to save the key (/home/username/.ssh/id_rsa):

At this point, tell it to choose this file:

```
/home/username/.ssh/username@host name
```

Then:

```bash
Enter passphrase (empty for no passphrase):

Press enter; you want an empty passphrase.

Enter same passphrase again:

Press enter.
```

Your identification has been saved in /home/username/.ssh/username@host name
Your public key has been saved in /home/username/.ssh/username@host name.pub
The key fingerprint is:

```
```

The key's randomart image is:

```
+--->[ RSA 2048]----+
|     .    |
|     o   o  .  |
|      o = o  . o |
|       B . . * o|
|        s o    O|
|         o o  . E|
|          o o  o |
|           o  + |
|            .. .|
| +-----------------+
```

Note that the program created two files.
The file that contains the private key is
The file that contains the public key has extension \texttt{.pub}:

```
/home/ \texttt{username} /.ssh/ \texttt{username@hostname}.pub
```

Next, tell SSH that you want to use this key. Make sure that the file \texttt{~/.ssh/config} exists:

```
$ touch ~/.ssh/config
```

Add a line containing

```
IdentityFile ~/.ssh/PRIVATE_KEY_FILE
```

(Using the filename for the private key).

Comment

\begin{quote}
make sure to include the full path to the file, not just the filename.
\end{quote}

Check that the config file is correct:

```
$ cat ~/.ssh/config
...
IdentityFile ~/.ssh/PRIVATE_KEY_FILE
...
```

To copy the generated SSH key to the clipboard xclip can be used (Installation of xclip if necessary).

```
$ sudo apt-get install xclip
$ xclip -sel clip < ~/.ssh/\texttt{username@hostname}.pub
```

4.6. How to login without a password

\textbf{Knowledge and activity graph}

\begin{itemize}
\item \textbf{Requires}: You have two computers, called “\texttt{local}” and “\texttt{remote}”, with users “\texttt{local-user}” and “\texttt{remote-user}”. Here, we assume that \texttt{local} and \texttt{remote} are complete hostnames (such as duckiebot.local.).
\item \textbf{Requires}: The two computers are on the same network and they can ping each other.
\item \textbf{Requires}: You have created a keypair for \texttt{local-user} on \texttt{local}. This procedure is
First, connect the two computers to the same network, and make sure that you can ping `remote` from `local`:

```
local $ ping remote
```

Do not continue if you cannot do this successfully.

If you have created a keypair for `local-user`, you will have a public key in this file on the `local` computer:

```
/home/local-user/.ssh/local-user@local.pub
```

This file is in the form:

```
ssh-rsa long list of letters and numbers local-user@local
```

You will have to copy the contents of this file on the `remote` computer, to tell it that this key is authorized.

On the `local` computer, run the command:

```
local $ ssh-copy-id remote-user@remote
```

now you should be able to login to the remote without a password:

```
local $ ssh remote-user@remote
```

This should succeed, and you should not be asked for a password.

### 4.7. Fixing SSH Permissions

Sometimes, SSH does not work because you have the wrong permissions on some files.

In doubt, these lines fix the permissions for your `.ssh` directory.

```
$ chmod 0700 ~/.ssh
$ chmod 0700 ~/.ssh/*
```

### 4.8. ssh-keygen
UNIT E-5

Python

Contents

Section 5.1 - Background .................................................................96

5.1. Background
PART F
Software development basics

This part describes how to create a programming workflow, using Git, Python, Docker, and the AI Driving Olympics infrastructure.

Contents

Unit F-1 - Git for development ................................................................. 98
Unit F-2 - Github ...................................................................................... 104
Unit F-3 - Basic Python code conventions ............................................... 107
Unit F-4 - Python projects ................................................................. 110
Unit F-5 - Python IDEs ........................................................................ 111
Unit F-6 - Explore the world of Python .................................................. 113
Unit F-7 - Bug squashing guide ............................................................ 115
This guide provides a basic overview on how to version control with Git.

Contents

Section 1.1 - Background reading .................................................................98
Section 1.2 - Git setup ..............................................................................98
Section 1.3 - Git terms explained ...............................................................99
Section 1.4 - Git basic actions ..................................................................99
Section 1.5 - Git advanced tips .................................................................101
Section 1.6 - Git troubleshooting ..............................................................102

1.1. Background reading
You should read a good book about Git.
We suggest you work through the Github guides, such as:

- Github bootcamp
- Github handbook
- Github tutorial
- Github workflow

For the more advanced topics:

- Github branching
- Git Flow

And you should know nobody remembers all Git commands:

- Github cheatsheet

My first exercise.

1.2. Git setup

1) Installation

The basic Git program is installed using

```
$ sudo apt install git
```

Additional utilities for `git` are installed using:

```
$ sudo apt install git-extras
```
This include the `git-ignore` utility, which comes in handy when you have files that you don't actually want to have pushed to the remote branch (such as temporary files).

2) Setting up global configurations for Git

This should be done twice, once on the laptop, and later, on the robot.

These options tell Git who you are:

```
$ git config --global user.email "email"
$ git config --global user.name "full name"
```

Also do this, and it doesn’t matter if you don’t know what it is:

```
$ git config --global push.default simple
```

1.3. Git terms explained

1) Repository

A **repo** (short for **repository**), or Git project, encompasses the entire collection of files and folders associated with a project, along with each file’s revision history.

You can see the repositories for the Duckietown project [here](#).

2) Branch

A **branch** is a version of the main code, that you can work on and it’s changes won’t affect the main code until it is merged into the master branch. When several people collaborate on a project, it makes sense for each developer to work on his own branch.

3) Fork

A **fork** is basically a copy of someone else’s repository. Usually, you cannot create branches or change code in other people’s repos, that’s why you create your own copy of it. This is called **forking**.

1.4. Git basic actions

1) Fork a repository

To fork (creating a copy of a repository, that does not belong to you), you simply have to go to the repository’s webpage dashboard and click fork on the upper right corner.

2) Clone a repository

To clone a repository, either copy the HTTPS or SSH link given on the repository’s webpage. Then invoke following command to download the git repository onto the
local computer (actual directory you are in right now).

$ git clone git@github.com:username/repository.git

If you have SSH setup properly, you can directly download it. If you are using the HTTPS then github will ask for your credentials.

3) Create a new branch

After you successfully cloned a repository, you may want to work on your own branch in order not to cause any chaos in the master branch. It is usually protected against changes. For this, you can branch out from the master or any other branches by invoking the command

$ git checkout -b branch-name

To see which branch you are working on you can either use both of these commands

$ git branch
$ git status

The latter provides more information on which files you might have changed, which are staged for a new commit or that you are up-to-date (everything is ok).

4) Commit and Push changes

After you edited some files, you want to push your changes from the local to the remote location. In order to do so, first do a double-check on which files you have changed and if things look legitimate. Invoke

$ git status

and check the output. There will be several files, that show up in red. These are files you have changed, but not yet added for a future commit. Most of the time you want to push all your changes so you add them to your commit by executing

$ git add --all

If you do not want to add all files, single files can be added. Then you need to specify each single file

$ git add file-path

After you solved this, add a commit message to let collaborators know, what you have changed:
If everything went smooth without any issues you are ready to push your changes to your branch:

```bash
$ git push origin branch-name
```

5) Fetch new branches

If new branches have been pushed recently to the repository and you don’t have them you can invoke a

```bash
$ git fetch --all
```

to see all new branches and checkout to those.

6) Delete branches

To delete a local branch execute (you cannot be on the branch that you are going to delete!):

```bash
$ git branch -d branch-name
```

To delete a remote branch you need to push the delete command:

```bash
$ git push origin --delete branch-name
```

7) Open a pull request

If you are working on another branch than the master or if you forked a repository and want to propose changes you made into the master, you can open a so-called pull-request. In order to do so, press the corresponding tab in the dashboard of a repository and then press the green button New pull request. You will be asked which branch from which fork you want to merge.

8) Submitting issues

If you are experiencing issues with any code or content of a repository (such as this operating manual you are reading right now), you can submit issues. For doing so go to the dashboard of the corresponding repository and press the Issues tab where you can open a new request.

### 1.5. Git advanced tips

1) Delete branches
Delete local:

\[\textit{git branch -d} \textit{branch-name}\]

Delete remote:

\[\textit{git push origin --delete} \textit{branch-name}\]

Propagate on other machines by doing:

\[\textit{git fetch --all --prune}\]

2) Shallow clone

You can clone without history with the command:

\[\textit{git clone --depth 1 repository URL}\]

1.6. Git troubleshooting

1) Problem: \texttt{https} instead of \texttt{ssh}

The symptom is:

\[\textit{git push}\]

\textit{Username for 'https://github.com':}\n
Diagnosis: the remote is not correct.

If you do \textit{git remote} you get entries with \texttt{https:}

\[\textit{git remote -v}\]

\begin{verbatim}
origin https://github.com/duckietown/Software.git (fetch)
origin https://github.com/duckietown/Software.git (push)
\end{verbatim}

Expectation:

\[\textit{git remote -v}\]

\begin{verbatim}
origin git@github.com:duckietown/Software.git (fetch)
origin git@github.com:duckietown/Software.git (push)
\end{verbatim}

Solution:

\[\textit{git remote remove origin}\]

\[\textit{git remote add origin} \textit{git@github.com:duckietown/Software.git}\]
2) Problem: `git push` complains about upstream

The symptom is:

```
fatal: The current branch branch name has no upstream branch.
```

You have not associated the current branch to the remote.

Solution:

```
$ git push --set-upstream origin branch name
```
UNIT F-2

Github

This chapter describes how to create a Github account and setup SSH on the robot and on the laptop.

Contents

Section 2.1 - Create a Github account .................................................................104
Section 2.2 - Add a public key to Github ..............................................................104
Section 2.3 - hub ........................................................................................................106

2.1. Create a Github account

Our example account is the following:

Github name: greta-p
E-mail: greta-p@duckietown.com

Create a Github account (Figure 2.1).

Go to your inbox and verify the email.

2.2. Add a public key to Github

You will do this procedure twice: once for the public key created on the laptop, and later with the public key created on the robot.

Knowledge and activity graph

Requires: A public/private keypair already created and configured. This procedure is explained in Section 20.5 - Creating an SSH keypair.

Results: You can access Github using the key provided.
Since I am not as familiar with Linux an VIM it would have been great to say how we can get access to the public key: `sudo vim /home/username/.ssh/username@hostname.pub` and than copy it and add it on github SL

Go to settings (Figure 2.2).

Add the public key that you created:

To check that all of this works, use the command

```
$ ssh -T git@github.com
```

The command tries to connect to Github using the private keys that you specified. This is the expected output:

```
Warning: Permanently added the RSA host key for IP address 'ip address' to the list of known hosts.
Hi username! You've successfully authenticated, but GitHub does not provide shell access.
```
If you don’t see the greeting, stop.

2.3. **hub**

*hub* is a command line utility written by Github. It can be used as a substitute of *git*, in the sense that any command like *git* `xxx` can be rewritten as *hub* `xxx`. In addition, it provides several functions unique to Github, like creating pull requests.

1) **Installation**

Install *hub* using the instructions.

2) **Jump to project page**

Open the project page using:

```
$ hub browse
```

3) **Creating pull requests**

You can create a pull request using:

```
$ hub pull-request -m "description"
```
UNIT F-3

Basic Python code conventions

3.1. The Black Python formatter
You should use black, the standard Python formatter, which will make your code look like anyone else's code.

3.2. Tabs, spaces, indentation
Indentation is 4 spaces.
Never use tabs in Python files. The tab characters are evil.
If you find pre-existing code that uses tabs, be very careful in changing them. Do not use a tool to do it (e.g. “Convert tabs to spaces”); it will get it wrong.

3.3. Line lengths
Lines should be below 85 characters.
Long lines are a symptom of a bigger problem. The problem here is that you do not know how to program well, therefore you create programs with longer lines.
Do not go and try to shorten the lines; the line length is just the symptom. Rather, ask somebody to take a look at the code and tell you how to make it better.

3.4. The encoding line
All files must have an encoding declared, and this encoding must be utf-8:

```python
# -*- coding: utf-8 -*-
```

3.5. Sha-bang lines
Executable files start with:

```bash
#!/usr/bin/env python
```

3.6. Comments
Comments refer to the next line.
Comments, bad:

```python
from std_msgs.msg import String # This is my long comment
```

Comments, better:
3.7. Imports
Do not do a star import, like the following:

```python
from rostest_example.Quacker import *
```

Rather, import each symbol by name:

```python
from rostest_example.Quacker import MySymbol, Another
```

3.8. Logging
Do not use `print` for logging.
Rather, use the `logging` library:

```python
# initialize the logger
import logging
logger = logging.getLogger('my program')
logger.setLevel(logging.DEBUG)

# then, later
logging.info("info message")
logging.error("error message")
logging.warning("warning message")
```

3.9. Exceptions
Exceptions are good. Throw them all the time! You will regret if you don’t.
Get in the habit of catching and re-raising using the `from` keyword:

```python
try:
    some_function()
except BaseException as e:
    msg = 'Some function failed while I was doing this thing.'
    raise Exception(msg) from e
```

Python will show the full stack trace, telling you what exception caused what other.

3.10. `sys.exit`
It is extremely rare that calling `sys.exit` is the right thing to do. You should use ex-
exceptions instead.
In this unit we learn how to setup a pure Python project. This is suitable for creating a pure Python library that can be easily reused by others. Later, we will introduce additional complications, such as Docker and ROS. For now, we keep things simple and look at the basics.

Contents

Section 4.1 - Setting up the project structure .........................................................110
Section 4.2 - Virtual environments using pipenv .....................................................110
Section 4.3 - Unit tests ............................................................................................110
Section 4.4 - Running unit tests on the cloud ..........................................................110
Section 4.5 - Using a code-coverage tool .................................................................110

4.1. Setting up the project structure

4.2. Virtual environments using pipenv
• Explain why packages.
• Explain how PIP works.
• Explain why to keep different environments isolated.

4.3. Unit tests
• Explain why unit tests are necessary.

4.4. Running unit tests on the cloud
• Explain what is continuous integration.

4.5. Using a code-coverage tool
• Explain why you want to look at code-coverage.
UNIT F-5

Python IDEs

To be productive in coding, you need to have a proper IDE. An IDE is a necessary companion in your journey.

Contents

Section 5.1 - Why using an IDE.................................................................111
Section 5.2 - Python IDEs .............................................................................111
Section 5.3 - Detailed guides.................................................................111
Section 5.4 - Check offs .............................................................................112

5.1. Why using an IDE

Look at how quickly you can create Python files by using an IDE (Figure 5.1). In this case, the editor is importing the required symbols. Think about how long would it take to do it without an IDE.

Figure 5.1. Eclipse editor capabilities

5.2. Python IDEs

We suggest you install and master one of these:

- PyCharm educational version;
- VSCode;
- Liclipse.

5.3. Detailed guides

We have the following more detailed guides:

- Unit H-8 - PyCharm
- Unit H-7 - Liclipse
5.4. Check offs
6.1. Useful Python libraries

Python comes with batteries included (Figure 6.1). As you become a Python expert, you might want to check out the following libraries.

Plotting:
- matplotlib
- seaborn

Numeric computation and machine learning:
- numpy
- panda
- scipy

Computer vision:
- opencv
6.2. Useful Python tools

These are essential tools to use as needed:

- jupyter
- ipython
This unit describes how to debug your programs. Do read this accurately top-to-bottom.

If you think this is too long and too verbose to read and you are in a hurry anyway: that is probably the attitude that introduced the bug. It might be programming is not for you.

Contents

Section 7.1 - Historical notes .................................................................115
Section 7.2 - The basic truths of bug squashing .................................115
Section 7.3 - What could it be? .............................................................116
Section 7.4 - How to find the bug by yourself .................................116
Section 7.5 - How to ask for help? .....................................................117
Section 7.6 - How to give help ............................................................117

7.1. Historical notes

First, count your blessings. You are lucky to live in the present. Once, there were actual bugs in your computer (Figure 7.1).

![First actual case of bug being found.](image)

Figure 7.1. "First actual case of bug being found." Read the story here.

7.2. The basic truths of bug squashing

1) Truth: it is most likely something simple

The first truth is the following:

It is always something simple.

People tend to make up complicated stories in their head about what is happening. One reason they do that is because when you are frustrated, it is better to imagine to battle against an imaginary dragon, rather than a little invisible Leprechaun who is
playing tricks on you.
Especially in an easy environment like Linux/ROS/Python with coarse process-level parallelization, there is really little space for weird bugs to creep in. (If you were using parallel C++ code, you would see lots of heisenbugs). Here, the reason is always something simple.

2) Truth: the fault is likely yours

The second truth is the following:

While there are bugs in the system, it is more likely there is a bug in your code or in your environment.

7.3. What could it be?

1) 20%: Environment errors

Any problem that has to do with libraries not importing, commands not existing, or similar, are because the environment is not set up correctly. Biggest culprit: forgetting “source environment.sh” before doing anything, or rushing through the setup steps ignoring the things that failed.

2) 10%: Permission errors

Permission errors are most likely because people randomly used “sudo”, thus creating root-owned files where they shouldn’t be.

3) 9%: Bugs with the Duckietown software

Please report these, so that we can fix them.

4) 1%: Bug with ROS or other system library

Please report these, so that we can find workaround.

5) 10%: Problems with configuration files

Make sure that you have pulled duckiefleet, and pushed your changes.

Finally, given the questions we had so far, I can give you the prior distribution of mistakes:

6) 50%: Programming mistakes

Of these, 80% is something that would be obvious by looking at the stack trace and your code and could be easily fixed.

7.4. How to find the bug by yourself

1) Step 0: Is it late? Go to bed.

If it is later than 10pm, just go to bed, and look at it tomorrow.
After 10pm, bugs are introduced, rather than removed.

2) Step 1: Are you in a hurry? Do it another time.

Bug squashing requires a clear mind.
If you are in a hurry, it’s better you do this another time; otherwise, you will not find the bug and you will only grow more frustrated.

7.5. How to ask for help?

Many people just write: “I get this error: ... How can I fix it?”. This is not the best way to get help. If you don’t include the code and stack trace, it’s hard to impossible to help you.

The best way to get help is the following:
Gold standard: Provide exact instructions on how to reproduce the error (“Check out this branch; run this command; I expect this; instead I get that”). This makes it easy for an instructor or TA to debug your problem in 30 seconds, give you the fix, and probably fix it for everybody else if it is a common problem.
Silver standard: Copy the relevant code to a Gist (gist.github.com) including the error stack trace. Because we have no way to reproduce the error, this starts a conversation which is basically guesswork. So you get half answers after a few hours.

7.6. How to give help

1) Step 1: Consider whether there are enough details to provide an informed answer.

The worst thing you can do is guess work – this causes confusion.
I encourage the TAs to not answer any nontrivial question that is not at least at the silver standard. It is a waste of resources, it will likely not help, and it actually contributes to the confusion, with people starting to try random things until something works without understanding why things work, and ultimately creating a culture of superstitions.
PART G

Basic Docker development

Contents

Unit G-1 - Introduction to Docker for Robotics and Machine Learning...119
Unit G-2 - Duckiebot Development using Docker .................................133
Unit G-3 - Troubleshooting................................................................................143
Unit G-4 - Creating Docker containers.................................................................144
Unit G-5 - Continuous integration......................................................................145
Unit G-6 - Duckietown Shell dts ........................................................................149
Unit G-7 - Using the AI-DO infrastructure..............................................................152
UNIT G-1

Introduction to Docker for Robotics and Machine Learning

1.1. Docker is a tool for portable, reproducible computing

It would be nice to give a computer - any computer with an internet connection - a short string of ASCII characters (say via a keyboard), press enter, and return to see some program running. Forget about where the program was built or what software you happened to be running at the time (this can be checked, and we can fetch the necessary dependencies). Sounds simple, right? In fact, 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 now can run any Linux program on almost any networked device on the planet, as is. 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 the string correctly.

1.2. Docker containers are easy to install

Let’s say you have never used Docker. To get Docker, run this command on a POSIX shell of any Docker-supported platform:

```
$ curl -sSL https://get.docker.com/ | sh
```

Now you have installed Docker!

Suppose your friend, Daphne, has a Docker container. How can we run this container? Docker containers live inside registries, which are servers that host Docker images. A Docker image is basically a filesystem snapshot—a single file that contains everything you need to run her container.
You can fetch Daphne’s container by running the following command to pull it from her Docker Hub repository:

```
$ docker pull daphne/duck
```

Now you have Daphne’s Docker image. To see a list of Docker images on your machine, type:

```
$ docker images
```

Every image has an image ID, a name and a tag:

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>daphne/duck</td>
<td>latest</td>
<td>ea2f90g8de9e</td>
<td>1 day ago</td>
<td>869MB</td>
</tr>
</tbody>
</table>

To run a Docker container, type the repository name, like so:

```
$ docker run daphne/duck
```

Now you are running Daphne’s container. To verify it is running, type:

```
$ docker ps
```

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>...</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>52994ef22481</td>
<td>daphne/duck</td>
<td>...</td>
<td>happy_hamster</td>
</tr>
</tbody>
</table>

Note how Daphne’s duck container has a container ID, a base image, and a funny-
looking name, happy_hamster. This name is an alias for the container ID.

1.3. Containers come from other containers
So you have a terminal and an internet connection? Now it doesn’t matter what operating system you’re running! You can run almost any Linux program in the world with just a few keystrokes. No further steps are necessary. How neat is that? To have a tool that clones a program and its environment, fetches the appropriate dependencies, and runs it on any OS is a big time-saver. Suppose you have a program that runs on one computer. It is extremely likely to run on any other, regardless of the underlying OS or hardware.

But how do you create a Docker image? There are two ways. You can either snapshot a running Docker container, or you can write a plaintext recipe. First, let’s see how to create a snapshot:

```bash
$ docker run -it daphne/duck bash
```

This will launch Daphne’s container and drop us into a bash session within. Suppose we make a change to the Docker container like so:

```
root@295fd7879184:/# touch new_file && ls
total 0
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
```

However, this file is not permanent. Exit the container:

```
root@295fd7879184:/# exit
```

Then rerun it:

```bash
$ docker run -it daphne/duck bash
```

And run:

```
root@18f13bb4571a:/# ls
root@18f13bb4571a:/# touch new_file1 && ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

It seems like new_file has disappeared! Notice how the container ID (18f13bb4571a) is now different. This is because docker run daphne/duck created a new container from the image daphne/duck, rather than restarting our old container.

Let’s see all the containers on our machine:
It looks like 295fd7879184 a.k.a. merry_manatee survived, but it is no longer running. Whenever a container's main process (recall we ran merry_manatee with bash) finishes, the container will stop, but it will not cease to exist.

In fact, we can resume the stopped container right where we left off:

```
$ docker start -a merry_manatee
```

```
root@295fd7879184:/# ls -l
total 0
-rw-r--r-- 1 root root 0 May 21 20:52 new_file
```

Nothing was lost! Let’s open a new terminal (without leaving the current one) to see what other containers are running:

```
$ docker ps
```

```
CONTAINER ID   IMAGE             NAMES
295fd7879184   daphne/duck       merry_manatee
18f13bb4571a   daphne/duck       shady_giraffe
52994ef22481   daphne/duck       happy_hamster
```

Now suppose we would like to share the container shady_giraffe with someone else. To do so, we must first snapshot the running container, or commit it to a new image, giving it a name and a tag. This will create a checkpoint that we may later restore:

```
$ docker commit -m "fork Daphne's duck" shady_giraffe your/duck:v2
```

Wherever you see a funny-looking name like shady_giraffe in Docker, this is just another way to reference the container. We either can use the container ID, 18f13bb4571a or the designated name (ie. shady_giraffe). The above your can be your username or an organization you belong to on a Docker registry. This image repository will be called your/duck, and has an optional version identifier, v2. Now we can push it to the Docker Hub registry:
$ docker push your/duck:v2
$ docker images

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>daphne/duck</td>
<td>latest</td>
<td>ea2f90g8de9e</td>
<td>1 day ago</td>
<td>869MB</td>
</tr>
<tr>
<td>your/duck</td>
<td>v2</td>
<td>d78be5cf073e</td>
<td>2 seconds ago</td>
<td>869MB</td>
</tr>
</tbody>
</table>

$ docker pull your/duck:v2 # Anyone can run this!
$ docker run your/duck ls -l

This is a convenient way to share an image with your colleagues and collaborators. Anyone with access to the repository can pull our image and continue right where we left off, or create another image based on our own. Images can be created via the command line or by using something called a Dockerfile.

### 1.4. Containers come from recipes

The second way to create a Docker image is to write 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 daphne/duck       # Defines the base image
RUN touch new_file1   # new_file1 will be part of our snapshot
CMD ls -l             # Default command to be run when the container is started
```

Now, to build the image we can simply run:

```
$ docker build -t your/duck:v3 .        # Where '.' is the directory containing your Dockerfile
```

You should see something like:
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM daphne/duck
    --- ea2f90g8de9e
Step 2/3 : RUN touch new_file1
    --- e3b75gt9zc4
Step 3/3 : CMD ls -l
    --- Running in 14f834yud59
Removing intermediate container 14f834yud59
    --- 05a3bd381fc2
Successfully built 05a3bd381fc2
Successfully tagged your/duck:v3

Now run the command `docker images` in your terminal, and you should see an image called `your/duck` with tag `v3`:

```
$ docker images
REPOSITORY        TAG           IMAGE ID       CREATED       SIZE
daphne/duck        latest        ea2f90g8de9e   1 day ago     869MB
your/duck           v2            d78be5cf073e   5 minutes ago 869MB
your/duck           v3            05a3bd381fc2   2 seconds ago 869MB
```

This procedure is identical to the snapshot method we performed earlier, but the result is much cleaner. Now, instead of needing to carry around a 869MB BLOB, we can just store the 4KB text file and rest assured that all our important setup commands are contained within. Similar to before, we can simply run:

```
$ docker run -it your/duck:v3
```

Notice that as soon as we run the container, Docker will execute the `ls -l` command as specified by the Dockerfile, revealing `new_file1` was 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]`

1) Layer Caching

An important concept in Docker is the layers. One way to think of a layer is like a Git commit - a set of changes to a previous image or layer, identified by a hash code. In a Dockerfile, layers begins with a keyword. Let's have a look:
FROM daphne/duck
RUN touch new_file1                             # Defines a new layer
RUN mkdir config && mv new_file1 mkdir          # Each layer can have
multiple commands
RUN apt-get update && apt-get install -y wget   # Install a dependency
RUN wget https://get.your.app/install.sh        # Download a script
RUN chmod +x install.sh && ./install.sh         # Run the script

To build this image, we can run the command `docker build -t your/duck:v4 .`:

Sending build context to Docker daemon 2.048kB
Step 1/6 : FROM daphne/duck
---> cd6d8154f1e1
Step 2/6 : RUN touch new_file1
---> Running in a88a5e9ab8d0
Removing intermediate container a88a5e9ab8d0
---> 0473154b2004
Step 3/6 : RUN mkdir config && mv new_file1 mkdir
---> Running in e4c4dd614bd4
Removing intermediate container e4c4dd614bd4
---> 2201828019ff
Step 4/6 : RUN apt-get update && apt-get install -y wget
---> Running in 8fb56ef38bc8
... Removing intermediate container 8fb56ef38bc8
---> 3358calb8649
Step 5/6 : RUN wget https://get.your.app/install.sh
---> Running in e8284ff4ec8b
... 2018-10-30 06:47:57 (89.9 MB/s) - 'install.sh' saved [13847/13847]
Removing intermediate container e8284ff4ec8b
---> 24a22dc2900a
Step 6/6 : RUN chmod +x install.sh && ./install.sh
---> Running in 9526651fa492
# Executing install script, commit: 36b78b2
...
Removing intermediate container 9526651fa492
---> a8be23fea573
Successfully built a8be23fea573
Successfully tagged your/duck:v4

Layers are conveniently cached by the Docker daemon. If we run the same command again, Docker will use the cache instead of rebuilding the entire image:
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM daphne/duck
    ---> cd6d8154f1e1
Step 2/6 : RUN touch new_file1
    ---> Using cache
    ---> 0473154b2004
Step 6/6 : RUN chmod +x index.html && ./index.html
    ---> Using cache
    ---> a8be23fea573
Successfully built a8be23fea573
Successfully tagged your/duck:v4

If we later make a change to the Dockerfile, Docker will only need to rebuild the image starting from the first modified instruction. Suppose we add the line RUN echo "Change here!" to the bottom of our Dockerfile and rebuild:

Sending build context to Docker daemon  2.048kB
... Step 6/7 : RUN chmod +x index.html && ./index.html
    ---> Using cache
    ---> a8be23fea573
Step 7/7 : RUN echo "Change here!"
    ---> Running in 80fc5c402304
Change here!
Removing intermediate container 80fc5c402304
    ---> c1ec64cef9c6
Successfully built c1ec64cef9c6
Successfully tagged your/duck:v4

If Docker had to rerun the entire Dockerfile from top to bottom to every time we built an image, this would be terribly slow and inconvenient. Fortunately, Docker is smart enough to cache the layers which have not changed, and only rerun the minimum set of commands to rebuild our image. This is a very nice feature, however it can sometimes introduce unexpected results, especially when the cache is stale. To ignore the cache and force a clean rebuild, use the --no-cache flag.

What does Docker consider when deciding whether to use the cache? First is the Dockerfile itself - when an instruction changes, it and any subsequent instructions will need to be rerun during a build. Docker must also consider the build context. When we write docker build -t TAG ., the . indicates the context, or path where the build should occur. Often, this path contains build artifacts. For example:

```
FROM daphne/duck
COPY duck.txt .
RUN cat duck.txt
```

Now if we run the command
this will create a new file `duck.txt` in your build directory, and we will copy that file into the Docker image, then print its contents:

```
$ echo "Make way for duckies!" > duck.txt && docker build -t my/duck:v5.
```

As long as the first three lines of the Dockerfile and `duck.txt` are not modified, these layers will be cached and Docker will not need to rebuild them. However if the contents of `duck.txt` are later modified, this will force a rebuild to occur. For example, if we run

```
$ echo "Thank you. Have a nice day!" >> duck.txt
$ docker build -t my/duck:v5.
```

we have:

```
$ echo "Thank you. Have a nice day!" >> duck.txt
$ docker build -t my/duck:v5.
```

A common mistake when writing Dockerfiles is to COPY more than is necessary to perform the next build step. For example, if we write COPY . . at the beginning of the Dockerfile, then whenever a file in changed anywhere in the build context, this will trigger a rebuild of subsequent instructions which is not often what we want. If
we are conservative and only COPY what we need, this is a more efficient use of the cache and our builds will complete much more quickly. A general rule of thumb is source code be copied after dependencies and before compilation.

Like Git’s .gitignore, Docker has a .dockerignore file. If we add a line to the .dockerignore file, then all paths matching this line in the build context will be ignored. Docker also accepts more sophisticated pattern matching features like regular expressions and negation. For more details, refer to the Docker documentation.

2) Multi-stage builds

Docker’s filesystem is purely additive, so each layer will only increase the size of the final image. If you care about image size, it is often necessary to reduce the number of layers and tidy up unnecessary files. For example, when installing dependencies on Debian-based images, a common practice is to run

```bash
RUN apt-get update && apt-get install ... && rm -rf /var/lib/apt/lists/*
```

ensuring the package list is not baked into the image (Docker will only checkpoint the layer after an instruction is complete). But often, you only care about a single artifact, although building it can take several steps. To avoid this dance of chaining together commands and removing the garbage in a single instruction, we can use a technique called **multi-stage builds**. These allow you to build sequential images inside a **Dockerfile**, and copy resources from one to another, discarding the rest:

```bash
FROM your/duck:v3 as template1       # We can use `template1` as an alias later
FROM daphne/duck as template2
COPY --from=template1 new_file1 new_file2
FROM donald/duck as template3       # The last `FROM` will define *this* image
COPY --from=template2 new_file2 new_file3
CMD ls -l
```

Now let’s build and run this image:
Why would you want to use this feature? For example, one application of multi-stage builds is compiling a dependency from its source code. In addition to all the source code, the compilation itself could have separate build dependencies and setup requirements, all of which are quite incidental to our ultimate goal - to build a single file. If we’re especially unlucky, we might end up with gigabytes of transitive dependencies to build a tiny binary file, and waste a lot of disk space or time cleaning up the mess. Multi-stage builds allow us to build the file, discard the unnecessary bits, copy it to a fresh layer, and move on with our life, unburdened by intermediate dependencies.

1.5. Docker is not a silver bullet for complexity

When creating a new image, it may be tempting to reinvent the wheel. Your application is special, and has special requirements. But there are millions of Docker images in the wild. Unless you are doing something very special indeed, it’s best to keep things simple. Find a base image that accomplishes most of what you are trying to achieve, and build on top of it. Base images like Ubuntu (or the very popular Alpine Linux, due to its small footprint), are okay, but there is probably something more specific to your application’s requirements. Even the python base image can be fairly generic, there are many images which contain specific Python stacks.

It may also be tempting to use some random image you found on Docker Hub, which does exactly what you want. Congratulations! Maybe this is the case. But unless you are doing something very similar, it probably does some extra things that are inefficient, or even harmful to your application. If it provides a Dockerfile, inspect it first and see what’s inside. Maybe you can adapt the Dockerfile to suit your needs and get
rid of a lot of complexity. Try to find a happy medium between simplification and creating a Rube Goldberg image stack. It may work, but will not save you any headache in the long run. The best Docker images are often provided by the maintainer of your favorite library or dependency.

1.6. Running Docker remotely
You can run Docker commands locally, or on a remote Docker Host. To do so, run the command docker -H  REMOTE_HOST_NAME  . You do not need to open a SSH session simply to run a Docker command.

1.7. Useful Docker resources
We have found the following resources helpful for robotics and Machine Learning:

1) Resin
Resin.io is a very good source of base images for ARM devices. The best part of using Resin images, is that you can build them on x86 devices, such as your laptop or the cloud. Baked into every Resin image is a shim for the shell that will allow you to run ARM binaries on x86. To use this feature, you can adapt the StringUtils template below:

```
FROM resin/ BASE_IMAGE  # e.g. raspberrypi3-python
RUN [ "cross-build-start" ]
# Your code goes here...
RUN [ "cross-build-end" ]
CMD  DEFAULT_START_COMMAND
```

Resin uses qemu to cross-build images as described in this article. Also described is how to build and run non-Resin-based ARM images on x86 devices. This technique is not just for building images - it also works at runtime! When running an ARM image, simply use the qemu-arm-static binary as an entrypoint to your launch command:

```
$ docker run --entrypoint=qemu-arm-static -it your/arm-image bash
```

2) ROS
ROS.org builds nightly ARM and x86 images for robotics development. For each distro, there are packages like core, base, perception (including OpenCV), robot (for the robot) and others.

3) Hypriot
Hypriot, a base OS for RPi and other ARM devices, include support for Docker out of the box. Hypriot is lightweight, fast, and builds the latest RPi Linux kernels and Raspbian releases.

4) PiWheels

Not all Python packages (especially if they wrap a native library) can be run on all platforms. You might be tempted to build some package from its sources (and in rare cases, you may need to do so). But there is a good chance your favorite Python library is already compiled for Raspberry Pi on PiWheels. By running the following command (either in your Dockerfile or at runtime), you can install Python packages, e.g. opencv-python:

```bash
$ pip install opencv-python --index-url https://www.piwheels.org/simple
```

5) Graphical User Interfaces

Docker also supports GUIs, but you will need to configure X11 forwarding.

6) Docker Hub

Docker Hub is the central repository for Docker Images. Unless you have configured a separate registry, whenever you pull a Docker image tag, it will query the Docker Hub first. You can use the Docker Hub to upload Docker images, and configure automated builds from GitHub (with a 2-hour build timeout). Docker Hub does not support layer caching of any kind, so the build will always take a fixed amount of time.

Figure 1.2. Docker Hub auto-builds allow you to link a Dockerfile in a GitHub repository, and whenever that Dockerfile changes, the Docker image will be updated.

The Docker Hub also has features for configuring repository links and build triggers. These will automatically rebuild your Docker image when some event happens.
Repository links allow you to chain builds together across Docker Hub repositories. Whenever a linked repository is updated, your image will be rebuilt.

7) Docker Cloud

Docker Cloud is integrated with the Docker Hub (and may one day replace it). Builds are automatically published from Docker Cloud to Docker Hub. Notifications for email and Slack, as well as a longer build timeout (up to 4-hours) are supported. It also has features for configuring the build context and other useful build settings, such as enabling caching (unlike Docker Hub).
The following section will guide you through the Docker development process.

2.1. Prerequisites
Those who wish to use a physical Duckiebot will need these physical objects:
- Duckiebot
  - Raspberry Pi 3B+
  - Micro SD card (16GB+ recommended)
- Personal computer
- Internet-enabled router
- MicroSD card adapter
To interact with the Duckiebot, the computer must have the following software:
- POSIX-compliant shell
- Browser and/or Docker CE

2.2. Installation
- Software Prerequisites (Ubuntu/Debian):
  - wget/curl
  - apt-get
  - duckietown-shell
  - Docker (See Unit G-1 - Introduction to Docker for Robotics and Machine Learning for installation instructions)

First, you will need to set your Duckietoken:

```
$ dts tok set
```

Now, ensure that you have a valid Duckietoken:

```
$ dts tok verify $TOKEN
0 -> good JSON {'uid': number, 'exp': date}
1 -> bad error message
```

Place the Duckiebot’s MicroSD card into the MicroSD card adapter, insert it into the computer and run the following command:
The above command runs the `init_sd_card.sh` script, which will run an installer to prepare the SD card.

Follow the instructions, then transfer the SD card to the Raspberry Pi and power on the Duckiebot. On first boot, make sure the Raspberry Pi receives continuous power for at least five or ten minutes.

Your laptop should be connected to the same network as the Duckiebot, or alternatively, you will need to share internet from your laptop to your Duckiebot via an ethernet cable. Further details are described in the Duckiebot networking chapter.

Wait for a minute or so, and then visit the following URL:

http://DUCKIEBOT_NAME.local:9000/

You should be greeted by the Portainer web interface. This user-friendly web interface is the primary mechanism for interacting with a Duckiebot.

From here you can see the list of running containers on your Duckiebot:
If you prefer to use the command line, you can also connect to the Duckiebot via secure shell:

```
$ ssh USER_NAME@DUCKIEBOT_NAME.local
```

**Note:** Any Docker command can also be run remotely by using the hostname flag, `-H DUCKIEBOT_NAME`. You should not need to open an SSH connection simply to run a Docker command.

### 2.3. Running Simple HTTP File Server

All persistent data is stored under `/data` on the Duckiebot SD card. To access the data via the web browser, run:

```
$ docker -H DUCKIEBOT_NAME.local run -d \
  --name file-server \
  -v /data:/data \ 
  -p 8082:8082 \ 
  duckietown/rpi-simple-server:master
```

Go to the following URL: `http://DUCKIEBOT_NAME.local:8082/`

### 2.4. Testing the camera

Open Portainer Web interface and run the `duckietown/rpi-docker-python-picamera` container. Publish port 8081 and ensure that the container is run in “Privileged” mode.
$ docker -H $DUCKIEBOT_NAME.local run -d \
  --name picam \ 
  -v /data:/data \ 
  --privileged \ 
  -p 8081:8081 \ 
  duckietown/rpi-docker-python-picamera:master18

**Note:** The syntax `-H $DUCKIEBOT_NAME.local` may be omitted if you are running the command over SSH.

Visit the following URL: [http://$DUCKIEBOT_NAME.local:8082/image.jpg](http://$DUCKIEBOT_NAME.local:8082/image.jpg).

### 2.5. Testing ROS

It is best to first pull the base Duckietown Docker image using the following command:

$ docker -H $DUCKIEBOT_NAME.local pull duckietown/rpi-ros-kinetic-roscore:master18

Run the base Duckietown Docker image, opening a shell:
You can start a ROS environment on your laptop, which connects to the Duckiebot ROS Master:

```bash
$ docker -H $DUCKIEBOT_NAME.local run -it \
   --name roscore \
   --privileged \
   --net host \
   duckietown/rpi-ros-kinetic-roscore:master18
```

To allow incoming X connections, run `xhost +` on your computer.

**Note:** There is a more secure way to do this, if you are concerned about receiving arbitrary X11 connections.

The above command opens a “ROS” shell running on your laptop that is set to connect to DUCKIEBOT's ROS Master. To test the ROS connection, run `roswtf`:

```bash
$ roswtf
```

### 2.6. Test ROS Joystick

```bash
$ docker -H $DUCKIEBOT.local run -d \
   --name joystick-demo \
   --privileged \
   -v /data:/data \
   --net host \
   duckietown/rpi-duckiebot-joystick-demo:master18
```

### 2.7. Calibration

As described in Unit C-11 - Camera calibration and validation, print the calibration pattern and place the Duckiebot in the proper position.

1) **Extrinsic calibration procedure**

Launch the calibration container and follow the prompts:
You will first be asked to place the Duckiebot on the calibration pattern. Then you will be asked to place in a lane to test the calibration.

**Note:** Passing `-v /data:/data` is necessary so that all calibration settings will be preserved.

**Note:** You can run/launch the `rpi-simple-server` to see the results in your web browser; you can also download all files from `/data`. This is an easy way to view and download all calibration files and validation results.

### 2.8. Lane Following Demo

After the Duckiebot has been calibrated, you can now launch the Lane Following Demo.

```bash
$ docker -H DUCKIEBOT_NAME.local run -it \
--name lanefollowing-demo \
--privileged \
-v /data:/data \
--net host \
duckietown/rpi-duckiebot-lanefollowing-demo:master18
```

Wait for a few minutes for all nodes to be started and initialized.

You can test the Duckiebot by using the Joystick. Pressing R1 starts autonomous mode.

Pressing L1 puts the Duckiebot back in manual mode.

### 2.9. Development workflow

When developing Docker containers, there are two paths to deployment. You can write the Dockerfile on your laptop or an x86 machine, then build with the `RUN [ "cross-build-start" ]` and `RUN [ "cross-build-end" ]` commands. Once tested, you can deploy to the Duckiebot directly by running the following command:

```bash
$ docker save TAG_NAME | ssh -C duckie@DUCKIEBOT_NAME.local docker load
```

Alternately, you can build directly on an ARM device by creating a file named `Dockerfile.arm` (the `.arm` extension is just for the reader’s benefit), adding a base image and some build instructions, and running the command:
Note that ARM-specific Dockerfiles will not build on non-Mac x86 machines, and attempting to build one will cause an error on Docker Hub. However, once you have debugged the Dockerfile on an ARM device, you can easily port the entire build to x86 by enclosing it with `RUN [ "cross-build-start" ]` and `RUN [ "cross-build-end" ]` instructions, after the `FROM` and before the `CMD` directive, as seen here. Don't forget to publish to GitHub and set up a Docker Hub automatic rebuilds if you wish to automate the build.

2.10. Emulation

All Duckietown Docker images contain an emulator called QEMU - this allows us to run ARM images on x86 directly. To run a pure compute ROS node (i.e. one that does not require any camera or motor access) on a non-Mac x86 platform, you will need to provide a custom entrypoint to Docker when running the image. To do so, use the command `docker run ... --entrypoint=qemu3-arm-static YOUR_IMAGE [RUN_COMMAND]`, where `RUN_COMMAND` may be a shell such as `/bin/bash` or another command such as `/bin/bash -c "roscore"`. The qemu3-arm-static entrypoint is provided by duckietown/rpi-ros-kinetic-base, and may be updated in the future.

2.11. Common mistakes

1) exec user process caused “exec format error”

If you encounter this error, this means the container you are attempting to run is based on an image that is incompatible with the host's architecture. If you are trying to run an ARM image on an x86 host, you will need to use QEMU to emulate the ARM processor architecture. To run QEMU in Duckietown or Resin derived Docker image, use the flag `--entrypoint=qemu-arm-static` in your Docker run command. There is currently no solution for running x86 images on an ARM host, so you will need to build ARM-specific images for the Raspberry Pi.

2.12. Resources and References

1) SD Card Configuration and Flashing script

- https://github.com/duckietown/scripts

2) RPi Camera Test container

- https://github.com/rusi/rpi-docker-python-picamera
- https://hub.docker.com/r/duckietown/rpi-docker-python-picamera/
3) RPi Simple HTTP File Server
   - https://github.com/rusi/rpi-simple-server
   - https://hub.docker.com/r/duckietown/rpi-simple-server/

2.13. Duckiebot ROS containers
The following containers are very useful for getting started.

1) Base ROS container; opens **bash** when launched
   - https://github.com/duckietown/rpi-ros-kinetic-base
   - https://hub.docker.com/r/duckietown/rpi-ros-kinetic-base

2) Base ROS container with development tools and Duckietown dependencies (includes **picamera**)
   - https://hub.docker.com/r/duckietown/rpi-ros-kinetic-dev

3) **roscore** container - starts **roscore** when launched
   - https://github.com/duckietown/rpi-ros-kinetic-roscore
   - https://hub.docker.com/r/duckietown/rpi-ros-kinetic-roscore

4) Duckietown Base (monolithic) software container - opens **bash** when launched
   - https://github.com/duckietown/Software
   - https://hub.docker.com/r/duckietown/rpi-duckiebot-base

5) Joystick Demo container
   - https://github.com/duckietown/rpi-duckiebot-joystick-demo
   - https://hub.docker.com/r/duckietown/rpi-duckiebot-joystick-demo

6) Calibration container
   - https://github.com/duckietown/rpi-duckiebot-calibration
   - https://hub.docker.com/r/duckietown/rpi-duckiebot-calibration

7) Lane Following Demo container
   - https://github.com/duckietown/rpi-duckiebot-lanefollowing-demo
   - https://hub.docker.com/r/duckietown/rpi-duckiebot-lanefollowing-demo

8) Desktop ROS containers
   rosindustrial/roborobo-robot-nvidia:kinetic
   - https://github.com/robo-industrial/docker
osrf/ros:kinetic-desktop-full
- https://hub.docker.com/r/osrf/ros/

2.14. Docker Image Hierarchy

![Docker Image Hierarchy Diagram]

Figure 2.4. Docker Image Hierarchy

2.15. Misc

1) Building images:
$ cd image-builder-rpi
$ docker build . --tag TAG_NAME

2) Transferring Docker containers

$ docker save TAG_NAME | gzip | ssh -C duckie@DUCKEBOT_NAME.local docker load

Figure 2.5. Output of rqt_dep joystick (compilation dependencies)

Figure 2.6. Output of rqt_graph joystick (runtime dependencies)
UNIT G-3
Troubleshooting

**Symptom:** E: Failed to fetch http://packages.ros.org/ros/ubuntu/dists/xenial/main/binary-amd64/Packages Error writing to output file - write (28: No space left on device) Error writing to file - write (28: No space left on device) [IP: 64.50.233.100 80]

**Note:** Only happens in Mac since Docker is actually running VM with a fixed size

**Resolution:** Increase the size of your Disk image: Docker -> Preferences -> Disk and increase the slider and hit apply.
UNIT G-4

Creating Docker containers
UNIT G-5

Continuous integration

These are the conventions for the Duckietown repositories.

Contents

Section 5.1 - Never break the build .................................................................145
Section 5.2 - How to stay in the green .............................................................145
Section 5.3 - How to make changes to master: pull requests .......................146

5.1. Never break the build

The Software and the duckuments repository use “continuous integration”.
This means that there are well-defined tests that must pass at all times.
For the Software repository, the tests involve building the repository and running
unit tests.
For the duckuments repository, the tests involve trying to build the documentation
using make compile.
If the tests do not pass, then we say that we have “broken the build”.
We also say that a branch is “green” if the tests pass, or “red” otherwise.
If you use the Chrome extension Pointless, you will see a green dot in different places
on Github to signify the status of the build (Figure 5.1).

Figure 5.1. The green dot is good.

5.2. How to stay in the green

The system enforces the constraint that the branch master is always green, by pre-
venting changes to the branches that make the tests fail.
We use a service called CircleCI. This service continuously looks at our repositories.
Whenever there is a change, it downloads the repositories and runs the tests.
(It was a lot of fun to set up all of this, but fortunately you do not need to know how
it is done.)
At this page you can see the summary of the tests. (You need to be logged in with
your Github account and click “authorize Github”).
5.3. How to make changes to master: pull requests

It is not possible to push on to the master branch directly.

- See the Github documentation about pull requests to learn about the general concept.

The workflow is as follows.

1. You make a private branch, say `your name-devel`.
2. You work on your branch.
3. You push often to your branch. Every time you push, CircleCI will run the tests and let you know if the tests are passing.
4. When the tests pass, you create a “pull request”. You can do this by going to the Github page for your branch and click on the button “compare and pull request” (Figure 5.3).
You now have an opportunity to summarize all the changes you did so far (Figure 5.4). Then click “create pull request”.

Now the pull request has been created. Other people can see and comment on it. However, it has not been merged yet. At this point, it might be that it says “Some checks haven’t completed yet” (Figure 5.5). Click “details” to see what’s going on, or just wait.

When it’s done, you will see either a success message (Figure 5.6) or a failure mes-
Figure 5.6. The tests are done

Figure 5.7. The tests have failed

(7) At this point, you can click “squash and merge” to merge the changes into master (Figure 5.8).

Figure 5.8. The final step

1) Troubleshooting

If you see a message like “merge failed” (Figure 5.9), it probably means that somebody pushed into master; merge master into your branch and continue the process.

Figure 5.9. Merge failed
Duckietown Shell (dts) is a pure Python utility for Duckietown. The idea is that most of the functionality is implemented as Docker containers, and dts provides a nice interface for that, so that user should not type a very long `docker run` command line.

Contents

6.1. Requirements
The shell requires at least Python 3.6 to be installed.

6.2. Installation
You can install dt-shell by running

```
$ pip install duckietown-shell
```

Installing dts via pip will also install the following packages
- GitPython - A python library used to interact with Git repositories
- texttable - A module to generate a formatted text table, using ASCII characters.

1) Run the shell
You can launch the shell by running the command

```
$ dts
```

You should see the dts prompt

6.3. Commands
Duckietown Shell dts

dts is modular, and the pip package does not contain commands. The first time you launch the shell, the most recent version of the commands will be automatically downloaded and made available to you. A barebones set of commands will be automatically installed, others will be available but not installed.

1) List commands

You can list all the commands available in dt-shell by running

```
$ dts> commands [--core] [--installed] [--installable]
```

You should see something like the following

```
dts> commands
Core commands:
   commands
   install
   uninstall
   update
   version
   exit
   help

Installed commands:
  <empty>

Installable commands:
   aido18
   logs
```

Use the arguments --core, --installed, and --installable to filter the commands.

2) Update list of commands

Run the command dts> update to download the most recent version of the commands available.

**Note:** dt-shell uses git to update the commands. If you get an error, please make sure that git is installed.

3) Install a new command

Run dts> commands --installable to see the list of commands you can install.
Run dts> install command_name to install a new command.

4) Remove a command

Run dts> commands --installed to see the list of commands already installed.
Run dts> uninstall command_name to remove a command.
6.4. Custom commands

dts downloads all the commands from the git repository duckietown/duckietown-shell-commands. If you want to contribute to the shell and create new commands, please follow the instructions in the section Subsection 1.0.3 - Create a new command.

6.5. Troubleshooting
UNIT G-7
Using the AI-DO infrastructure

Contents
Section 7.1 - Intro.................................................................152

7.1. Intro
  • Why AIDO?
This part contains reference materials, such as a list of useful “recipes” for the command line.

Contents

Unit H-1 - Useful Linux commands ................................................................. 154
Unit H-2 - Raspberry-Pi commands ................................................................. 161
Unit H-3 - Working with SD Cards ................................................................. 162
Unit H-4 - Mounting USB drives................................................................. 166
Unit H-5 - VIM ......................................................................................... 167
Unit H-6 - Atom ...................................................................................... 168
Unit H-7 - Liclipse .................................................................................. 170
Unit H-8 - PyCharm .............................................................................. 172
Unit H-9 - Byobu .................................................................................. 173
UNIT H-1
Useful Linux commands

Contents

Section 1.1 - Getting help.................................................................155
  o Subsection 1.1.1 - man
Section 1.2 - Looking around.................................................................155
  o Subsection 1.2.1 - cd
  o Subsection 1.2.2 - ls
Section 1.3 - Showing authority .................................................................156
  o Subsection 1.3.1 - sudo
Section 1.4 - visudo .................................................................................156
Section 1.5 - Changing things around .........................................................156
  o Subsection 1.5.1 - cp
  o Subsection 1.5.2 - mv
  o Subsection 1.5.3 - mkdir
  o Subsection 1.5.4 - rm
  o Subsection 1.5.5 - touch
Section 1.6 - Restarting.............................................................................157
  o Subsection 1.6.1 - reboot
  o Subsection 1.6.2 - shutdown
Section 1.7 - UNIX processes.................................................................157
  o Subsection 1.7.1 - cat
  o Subsection 1.7.2 - tee
Section 1.8 - truncate .............................................................................157
Section 1.9 - Linux disks and files..............................................................157
  o Subsection 1.9.1 - fdisk
  o Subsection 1.9.2 - mount
  o Subsection 1.9.3 - umount
  o Subsection 1.9.4 - losetup
  o Subsection 1.9.5 - gparted
  o Subsection 1.9.6 - dd
  o Subsection 1.9.7 - sync
  o Subsection 1.9.8 - df
Section 1.10 - Users and permissions......................................................157
  o Subsection 1.10.1 - passwd
  o Subsection 1.10.2 - chmod
1.1. Getting help

1) man

This is an interface to the on-line reference manuals. Whenever you meet some unfamiliar commands, try use `man certain_command` before Googling. You will find it extremely clear, useful and self-contained.

1.2. Looking around

1) cd

Go to the directory you want. If you just use:

```
$ cd
```

then you will go to your home directory, i.e., `/home/user/`

To go up one directory use:

```
$ cd ..
```
2) `ls`

List all the files and documents in the current directory. From `~/.bashrc`, we know some commonly used alias. See more by `man ls`.

- `la` for `ls -a` which will list out all files and documents including the hidden ones (whose name starts with a dot).
- `ll` for `ls -l` which will display Unix file types, permissions, number of hard links, owner, group, size, last-modified date and filename.

### 1.3. Showing authority

1) `sudo`

Whenever you want to modify system files, you will need `sudo`. Commonly touched system files including `/etc`, `/opt` and so on. Since most of you have followed the guideline to use passwordless `sudo`, I would recommend that make sure what you are doing with `sudo` before you execute the command, otherwise you may need to reinstall the system.

#### 1.4. visudo

#### 1.5. Changing things around

1) `cp`

cp `fileA` `directoryB` will copy the file `A` to directory `B`. See more by executing `man cp`.

2) `mv`

`mv fileA directoryB` will move the file `A` to directory `B`. See more by executing `man mv`. This command can be used to rename a file, to do so execute: `mv fileA fileB`

3) `mkdir`

Make new directory. See more by `man mkdir`.

4) `rm`

Remove certain file. `rm -r` will also remove directories. More in `man rm`.

5) `touch`

Update the access and modification times of the input file to current time. See more by `man touch`. It can be used to create empty documents, for example: `touch duckye.txt` Will create an empty text file.
1.6. Restarting

1) reboot
This command must be executed as root. `sudo` required. This will reboot your laptop or Raspberry Pi. See more by `man reboot`.

2) shutdown
This command requires `sudo`. You can set a countdown to shutdown your machine. More by `man shutdown`.

1.7. UNIX processes

1) cat
Cat some file will return you the content. More in `man cat`.

2) tee
Read from standard input and write to standard output and files. More on `man tee`.

1.8. truncate

1.9. Linux disks and files

1) fdisk

2) mount

3) umount

4) losetup

5) gparted

6) dd

7) sync

8) df

1.10. Users and permissions
1) $ passwd
Update password of the current user. Old password needed.

2) $ chmod
chmod changes permission to a file or a directory. To make a file executable run:

```
$ sudo chmod +x FILE
```

3) groups

4) adduser

5) useradd

---

1.11. Downloading

1.12. $ curl

1.13. $ wget

1.14. Measuring CPU and IO usage

1) Measuring CPU usage using $ htop
You can use htop to monitor CPU usage.

```
$ sudo apt install htop
```

2) Measuring I/O usage using $ iotop
Install using:

```
$ sudo apt install iotop
```

---

1.15. Networking utils

1) $ hostname

2) $ ping: are you there?
3) ifconfig

$ ifconfig

1.16. Wireless networking

1) iwconfig

2) iwlist

iwlist is useful to understand the status of the Wifi networks.
For example, to get a list of wifi networks, use:

$ sudo iwlist interface scan | grep SSID

To check whether the interface support 5 GHz channels, use:

$ sudo iwlist interface freq

Example output:

wlx74da38c9caa0  20 channels in total; available frequencies :
    Channel 01 : 2.412 GHz
    Channel 02 : 2.417 GHz
    Channel 03 : 2.422 GHz
    Channel 04 : 2.427 GHz
    Channel 05 : 2.432 GHz
    Channel 06 : 2.437 GHz
    Channel 07 : 2.442 GHz
    Channel 08 : 2.447 GHz
    Channel 09 : 2.452 GHz
    Channel 10 : 2.457 GHz
    Channel 11 : 2.462 GHz
    Channel 36 : 5.18 GHz
    Channel 40 : 5.2 GHz
    Channel 44 : 5.22 GHz
    Channel 48 : 5.24 GHz
    Channel 149 : 5.745 GHz
    Channel 153 : 5.765 GHz
    Channel 157 : 5.785 GHz
    Channel 161 : 5.805 GHz
    Channel 165 : 5.825 GHz
Current Frequency:2.437 GHz (Channel 6)

Note that in this example only some 5Ghz channels are supported (36, 40, 44, 48, 149, 153, 157, 161, 165); for example, channel 38, 42, 50 are not supported. This means
that you need to set up the router not to use those channels.

1.17. Moving files between computers

1) scp

The Secure Copy (scp) copies files from different hosts using ssh for data transfer.

2) Download a file with SCP

To download a file named file.txt which is on hostname use this command:

```bash
$ scp hostname:/path/to/file.txt /path/where/you/wish/
```

to download file.txt to /path/where/you/wish/. If you want to download in the current directory simply use:

```bash
$ scp hostname:/path/to/file.txt .
```

3) rsync
2.1. **raspi-config**

`raspi-config` Opens the configuration tool of the Raspberry Pi, it requires root privileges, so run it using:

```
$ sudo raspi-config
```

2.2. **vcgencmd**

2.3. **raspistill**

`raspistill` Is the command to take a picture with the camera module of the Raspberry Pi. With your camera connected and enabled, run it with:

```
$ raspistill -o somename.jpg
```

2.4. **jstest**

2.5. **swapon**

2.6. **mkswap**
UNIT H-3
Working with SD Cards

Contents

Section 3.1 - Testing SD Card and disk speed .........................................................162
Section 3.2 - How to burn an image to an SD card..............................................162
Section 3.3 - How to shrink an image.................................................................163

3.1. Testing SD Card and disk speed

Test SD Card (or any disk) speed using the following commands, which write to a file called `filename`.

```bash
$ dd if=/dev/zero of=filename bs=500K count=1024
$ sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
$ dd if=filename of=/dev/null bs=500K count=1024
$ rm filename
```

Note the `sync` and the `echo` command are very important.

Example results:

524288000 bytes (524 MB, 500 MiB) copied, 30.2087 s, 17.4 MB/s
524288000 bytes (524 MB, 500 MiB) copied, 23.3568 s, 22.4 MB/s

That is write 17.4 MB/s, read 22 MB/s.

3.2. How to burn an image to an SD card

**KNOWLEDGE AND ACTIVITY GRAPH**

- **Requires:** A blank SD card.
- **Requires:** An image file to burn.
- **Requires:** An Ubuntu computer with an SD reader.
- **Results:** A burned image.

1) Finding your device name for the SD card

First, find out what is the device name for the SD card.
Insert the SD Card in the slot.
Run the command:
Find your device name, by looking at the sizes.
For example, the output might contain:

```
Disk /dev/mmcblk0: 14.9 GiB, 15931539456 bytes, 31116288 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

In this case, the device is /dev/mmcblk0. That will be the device in the next commands.
You may see /dev/mmcblk0pX or a couple of similar entries for each partition on the card, where X is the partition number. If you don’t see anything like that, take out the SD card and run the command again and see what disappeared.

2) Unmount partitions

Before proceeding, unmount all partitions.

**Warning:** Before unmounting partitions, make sure you found the correct device in the previous step. In particular, /dev/sda/ is the hard drive of your computer, and if you unmount its partitions, you can erase important data, including important files and the operating system you are not currently using.

Run `df -h`. If there are partitions like /dev/mmcblk0p n, then unmount each of them. For example:

```
$ sudo umount /dev/mmcblk0p1
$ sudo umount /dev/mmcblk0p2
```

3) Burn the image

Now that you know that the device is device, you can burn the image to disk.
Let the image file be image file.
Burn the image using the command `dd`:

```
$ sudo dd of=device if=image file status=progress bs=4M
```

**Note:** Use the name of the device, without partitions. i.e., /dev/mmcblk0, not /dev/mmcblk0pX.

**Note:** dd command with status=progress parameter only work for dd –version 8.24 ubuntu16.04.2

3.3. How to shrink an image
**Knowledge and Activity Graph**

**Requires:** An image file to burn.

**Requires:** An Ubuntu computer.

**Results:** A shrunk image.

**Note:** Majority of content taken from here

We are going to use the tool `gparted` so make sure it’s installed

```bash
$ sudo apt install gparted
```

Let the image file be `image file` and its name be `imagename`. Run the command:

```bash
$ sudo fdisk -l image file
```

It should give you something like:

<table>
<thead>
<tr>
<th>Device</th>
<th>Boot</th>
<th>Start</th>
<th>End</th>
<th>Sectors</th>
<th>Size</th>
<th>Id Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>duckiebot-RPI3-LP-aug15.img1</td>
<td>2048</td>
<td>131071</td>
<td>129024</td>
<td>63M</td>
<td>c</td>
<td>W95 FAT32 (LBA)</td>
</tr>
<tr>
<td>duckiebot-RPI3-LP-aug15.img2</td>
<td></td>
<td>131072</td>
<td>21219327</td>
<td>21088256</td>
<td>10.1G</td>
<td>83 Linux</td>
</tr>
</tbody>
</table>

Take note of the start of the Linux partition (in our case 131072), let’s call it `Start`. Now we are going to mount the Linux partition from the image:

```bash
$ sudo losetup /dev/loop0 imagename.img -o $((Start*512))
```

and then run `gparted`:

```bash
$ sudo gparted /dev/loop0
```

In `gparted` click on the partition and click “Resize” under the “Partition” menu. Resize drag the arrow or enter a size that is equal to the minimum size plus 20MB

**Note:** This didn’t work well for me - I had to add much more than 20MB for it to work.

Click the “Apply” check mark. Before closing the final screen click through the arrows in the dialogue box to find a line such a “resize2fs -p /dev/loop0 1410048K”.

Take note of the new size of your partition. Let’s call it `new size`.

Now remove the loopback on the second partition and setup a loopback on the whole image and run `fdisk`:

```bash
$ sudo fdisk -l
```
$ sudo losetup -d /dev/loop0
$ sudo losetup /dev/loop0 image file
$ sudo fdisk /dev/loop0

Command (m for help): enter d
Partition number (1,2, default 2): enter 2
Command (m for help): enter n
Partition type
p primary (1 primary, 0 extended, 3 free)
e extended (container for logical partitions)
Select (default p): enter p
Partition number (2-4, default 2): enter 2
First sector (131072-62521343, default 131072): start
Last sector, +sectors or +size{K,M,G,T,P} (131072-62521343, default 62521343): +new size K

Created a new partition 2 of type 'Linux' and of size 10.1 GiB.

Command (m for help): enter w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Re-reading the partition table failed.: Invalid argument

The kernel still uses the old table. The new table will be used at the next reboot or after you run partprobe(8) or kpartx(8).

Disregard the final error.
You partition has now been resized and the partition table has been updated. Now we will remove the loopback and then truncate the end of the image file:

$ fdisk -l /dev/loop0

<table>
<thead>
<tr>
<th>Device</th>
<th>Boot</th>
<th>Start</th>
<th>End</th>
<th>Sectors</th>
<th>Size</th>
<th>Id</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/loop0p1</td>
<td></td>
<td>2048</td>
<td>131071</td>
<td>129024</td>
<td>63M</td>
<td>c</td>
<td>W95 FAT32</td>
</tr>
<tr>
<td>/dev/loop0p2</td>
<td></td>
<td>131072</td>
<td>21219327</td>
<td>21088256</td>
<td>10.1G</td>
<td>83</td>
<td>Linux</td>
</tr>
</tbody>
</table>

Note down the end of the second partition (in this case 21219327). Call this end.

$ sudo losetup -d /dev/loop0
$ sudo truncate -s $(((end)+1)*512)) image file

You now have a shrunken image file.
It might be useful to compress it, before distribution:

$ xz image file
UNIT H-4

Mounting USB drives

First **plug in the USB drive** nothing will work if you don’t do that first. Now ssh into your robot. On the command line type:

```
$ lsusb
```

you should see your Sandisk USB drive as an entry. Congrats, you correctly plugged it in

```
$ lsblk
```

Under name you should see `sda1`, with size about 28.7GB and nothing under the `MOUNTPOINT` column (if you see something under `MOUNTPOINT` congrats you are done).

If it is not there already, make the directory to mount to:

```
$ sudo mkdir /data/logs
```

Next mount the drive

```
$ sudo mount -t vfat /dev/sda1 /data/logs -o umask=000
```

Test by running `lsblk` again and you should now see `/data/logs` under `MOUNTPOINT`.

4.1. Unmounting a USB drive

```
$ sudo umount /data/logs
```
To do quick changes to files, especially when logged remotely, we suggest you use the VI editor, or more precisely, VIM (“VI iMproved”).

5.1. **External documentation**
→ A VIM tutorial.

5.2. **Installation**
Install like this:

```bash
$ sudo apt install vim
```

5.3. **vi**

5.4. **Suggested configuration**
Suggested `~/.vimrc`:

```vim
syntax on
set number
filetype plugin indent on
highlight Comment ctermfg=Gray
autocmd FileType python set complete isk+=.,
```

5.5. **Visual mode**

5.6. **Indenting using VIM**
Use the `<>` command to indent.
To indent 5 lines, use `5`>`.
To mark a block of lines and indent it, use `<v`.
For example, use `<vJ>` to indent 3 lines.
Use `<c` to dedent.
**UNIT H-6**

**Atom**

6.1. Install Atom

Following the instructions here:

```
$ sudo add-apt-repository ppa:webupd8team/atom
$ sudo apt update
$ sudo apt install atom
```

After installing Atom, please open it once and close it again before proceeding with installing remote-atom!

6.2. Using Atom to code remotely

With Atom, you are able to remotely code on files located on your Duckiebot with a GUI. The benefit of using Atom is that you are able to install extensions such as an IDE for Python, a Markdown previewer, or just use custom themes to avoid coding in the terminal.

Follow these instructions:

Install remote-atom

```
$ sudo apm install remote-atom
```

Now, we need to edit our SSH config so that any data sent to the port 52698, which is the port remote-atom is using, is forwarded via SSH to our local machine. Edit the file “~/.ssh/config”. There, you add “RemoteForward 52698 127.0.0.1:52698” to your host. The resulting host will look similar to

```
Host lex
  User julien
  Hostname lex.local
  RemoteForward 52698 127.0.0.1:52698
```

Now, we need to connect to our duckiebot via SSH and install rmate (and simultaneously rename it to ratom)

```
$ sudo wget -O /usr/local/bin/ratom https://raw.github.com/aurora/rmate/master/rmate
$ sudo chmod +x /usr/local/bin/ratom
```

Now, you just need to launch Atom on your local machine, go to Packages->Remote
Atom->Start Server.
You can now edit a file in a terminal connected to your duckiebot via SSH by typing

```
$ sudo ratom filename
```

And atom will automatically open the file on your local machine. In the settings of remote-atom, you can also set the package to start the server automatically on launching atom on your local machine.
UNIT H-7
Liclipse

7.1. Installing LiClipse

Follow the instructions at this page.
At the moment of writing, these are:

```
$ wget http://www.mediafire.com/file/rwc4bk3nthtxcvv/liclipse_4.1.1_lin-
ux.gtk.x86_64.tar.gz
$ tar xvzf liclipse_4.1.1_linux.gtk.x86_64.tar.gz
$ sudo ln -s `pwd`/liclipse/LiClipse /usr/bin/liclipse
```

Now you can run it using liclipse:

```
$ liclipse
```

When it runs for the first time, choose “use this as default” and click “launch”.
Choose “Import -> General -> Existing project into workspace”. Select the folder
`~/duckietown`.

```
comment
Only Import -> General -> Projects from Folder or Archive, selecting `~/ducku-
ments` worked for me. JT
```

```
comment
This is not about the duckuments, it’s for duckietown - AC
```

If it asks about interpreters, select “auto config”.
When it shows “uncheck settings that should not be changed”, just click OK.

7.2. Set shortcuts for LiClipse

Go to “window/preferences/General/Keys”.
Find “print”, and unbind it.
Find “Quick switch editor”, and set it to `Ctrl`-`P`. (This is now the same as Atom.)
Find “Previous tab” and assign `Ctrl`-`Shift`-`[`. (This is now the same as Atom.)
Find “Next tab” and assign `Ctrl`-`Shift`-`]`. (This is now the same as Atom.)
Find “Show in (PyDev package explorer)” and assign `Ctrl`-`Shift`-`M`.

7.3. Shortcuts for LiClipse
Make sure that you can do the following tasks:

- Use the global browser: Press `Cmd-Shift-T`, type “what”. It should autocomplete to `what_the_duck`. Press enter; it should jump to the file.
- Switch among open editors with `Ctrl-P`.
- Switch between tabs with `Ctrl-Shift-[`, `-]`.
- See the current file in the directory, using `Cmd-Shift-M`.

**Table 7.1. LICLIPSE COMMANDS**

<table>
<thead>
<tr>
<th>On Linux</th>
<th>On Mac</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-Shift-T</td>
<td>Cmd-Shift-T</td>
<td>Globals browser</td>
</tr>
<tr>
<td>Ctrl-P</td>
<td>Cmd-P</td>
<td>Quick editor switch</td>
</tr>
<tr>
<td>Ctrl-Shift-[</td>
<td>Cmd-Shift-[</td>
<td>Previous tab (needs to be configured)</td>
</tr>
<tr>
<td>Ctrl-Shift-]</td>
<td>Cmd-Shift-]</td>
<td>Next tab (needs to be configured)</td>
</tr>
<tr>
<td>Ctrl-Shift-M</td>
<td>Cmd-Shift-M</td>
<td>Show in (PyDev package explorer)</td>
</tr>
<tr>
<td>Ctrl-1</td>
<td>Cmd-1</td>
<td>Find symbol</td>
</tr>
</tbody>
</table>

### 7.4. Other configuration for Liclipse

From the “Preferences” section, it’s suggested to:

- Get rid of the minimap on the right.
- Get rid of spellchecking.

Then, there is the issue of “code completion”. This is a love-it-or-hate-it issue. The choice is yours.
UNIT H-8

PyCharm
You need to learn to use Byobu. It will save you much time later. (Alternatives such as GNU Screen are fine as well.)

# 9.1. Advantages of using Byobu

# 9.2. Installation

On Ubuntu, install using:

```
$ sudo apt install byobu
```

# 9.3. Documentation

* See the screencast on the website http://byobu.co/.

# 9.4. Quick command reference

You can change the escape sequence from Ctrl-A to something else by using the configuration tool that appears when you type F9.

Commands to use windows:

| Table 9.1. Windows |
|--------------------|------------------|
| Using function keys | Using escape sequences |
| Create new window (F2) | Ctrl-A then C |
| Previous window (F3) | |
| Next window (F4) | |
| Switch to window | Ctrl-A then a number |
| Close window (Ctrl-F6) | |
| Rename window | Ctrl-A then 0 |

Commands to use panes (windows split in two or more):

<table>
<thead>
<tr>
<th>Table 9.2. Commands for panes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using function keys</td>
</tr>
<tr>
<td>Split horizontally (Shift-F2)</td>
</tr>
<tr>
<td>Split vertically (Ctrl-F2)</td>
</tr>
<tr>
<td>Switch focus among panes (Ctrl-T 1-5)</td>
</tr>
<tr>
<td>Break pane</td>
</tr>
</tbody>
</table>

Other commands:
TABLE 9.3. OTHER

<table>
<thead>
<tr>
<th></th>
<th>Using function keys</th>
<th>Using escape sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help</td>
<td>Ctrl-A then ?</td>
<td></td>
</tr>
<tr>
<td>Detach</td>
<td>Ctrl-A then D</td>
<td></td>
</tr>
</tbody>
</table>

9.5. Commands on OS X

Scroll up and down using fn option ↑ and fn option ↓.
Highlight using alt.
PART I
Advanced Duckietown development

This part contains development information for the Duckietown developers.

Contents

Unit I-1 - *Duckietown Shell for Developers* ........................................176
UNIT I-1

* Duckietown Shell for Developers

Contents

Subsection 1.0.1 - Downloading the commands
Subsection 1.0.2 - Understanding the structure of the repository
Subsection 1.0.3 - Create a new command
Section 1.1 - Update an existing command.........................................................180
Section 1.2 - Install third-party libraries.............................................................180

1) Downloading the commands

You can download the source code of the commands by forking the duckietown-
shell-commands repository and pulling your fork locally. The repository can be found
here.

2) Understanding the structure of the repository

Move to the directory where you pulled the commands repository. Run

```
$ tree .
```

You should be able to see a hierarchy of Python files that looks like the following,
Let us focus on the directories first. The directory `lib` is reserved to third-party libraries needed by the commands. Each directory (except for `lib`) defines a command. You will recognize some of the core commands (e.g., install, uninstall, version). The name of the directory defines the name of the command.

**Note:** A valid command name can contain only alphanumeric characters plus `_` and `-`.

Let us focus on files now. `dts` determines whether a command `mycommand` is installed by looking for the file `./mycommand/installed.flag`.

All the command directories contain a default `__init__.py` file. Such file is the same for all the commands. A standard template file `__init__.py.template` that you can copy if you wish to implement your own command is available in the main level of the repository.

All the magic happens in the file `command.py`, where the logic of the command is implemented. If you want to learn more about how to create your own command, read the section Subsection 1.0.3 - Create a new command.

This is all you need to know about the structure of the repository.

3) Create a new command

**Remark:** Before you can create a new command, you have to pull the source code locally as described in Subsection 1.0.1 - Downloading the commands. It is also important that you are familiar with the structure of the repository (described in Subsection 1.0.2 - Understanding the structure of the repository).
You can create a command `mycommand` by creating a directory in the main level of the repository with the following structure

```
./
  ...
  └── mycommand
      ├── __init__.py
      └── command.py
```

You can copy the files `__init__.py.template` and `command.py.template` that you find on the main level of the repo into your new command directory renaming them as `__init__.py` and `command.py` respectively.

This is enough to get your new command in `dt-shell`. Launch `dt-shell` and run

```
$ dt> mycommand
```

You should be able to see something like the following

```
dts> mycommand
You called the "mycommand" command, level 0, with arguments []
```

You can pass arguments to your command. For example, by running

```
$ dt> mycommand --arg1 value1
```

the shell will return

```
dts> mycommand
You called the "mycommand" command, level 0, with arguments ['--arg1', 'value1']
```

When the user types in the command `mycommand` and presses Enter, the file `./mycommand/command.py` will be used to serve the request. A valid file `command.py` will have the following structure
from dt_shell import DTCommandAbs

class DTCommand(DTCommandAbs):

    help = 'Brief description of the command'  # please redefine this help message
    # name = <read-only> a string with the name of the command
    # level = <read-only> an integer indicating the level of this command. Follows the directory hierarchy
    # commands = <read-only> a dictionary of subcommands

    @staticmethod
    def command(shell, args):
        # this function will be invoked when the user presses the [Return] key and submits the command
        #
        #   shell   is the instance of DTShell hosting this command
        #   args    is a list of arguments passed to the command
        #
        # PUT YOUR CODE HERE
        print(
            'You called the "%s" command, level %d, with arguments
            %r' % (
                DTCommand.name,
                DTCommand.level,
                args
            )
        )

    @staticmethod
    def complete(shell, word, line):
        # this function will be invoked when the user presses the [Tab] key for auto completion.
        #
        #   shell   is the instance of DTShell hosting this command
        #   word    is the right-most word typed in the terminal (usually the string the user is trying to auto-complete)
        #   line    is the entire command
        #
        # return a list of strings. Each string is a suggestion for the user
        #
        # PUT YOUR CODE HERE
        return ['suggestion_1', 'suggestion_2']

You can find the same template in the file command.py.template in the main level of the repository. You can recognize the default message printed above by the method command(shell, args) of the command mycommand.
The method `command(shell, args)` is invoked by the object `shell` when the user presses `Enter` and submits the argument `args` to the command. The argument `shell` is the instance of DTShell hosting this command, while `args` is the list of arguments passed to the command.

The method `complete(shell, word, line)` is invoked by the object `shell` when the user presses `Tab` and requests auto-complete. This method should return a list of strings, with each string being a suggestion for completing the command.

1.1. Update an existing command

**Remark:** Before you can create a new command, you have to pull the source code locally as described in Subsection 1.0.1 - Downloading the commands. It is also important that you are familiar with the structure of the repository (described in Subsection 1.0.2 - Understanding the structure of the repository).

You can follow the instructions in the section Subsection 1.0.3 - Create a new command to learn how the shell reacts to the input of the user and update your commands accordingly.

1.2. Install third-party libraries

You can add third-party libraries to the `.lib/` directory using `git submodule` if you have access to a public `git` repository that you can pull. Alternatively (but not suggested) you can simply copy your library in the `.lib/` directory and push it together with the commands.

`dt-shell` prepends the `.lib/` path to the `$PYTHON_PATH` environment variable before calling your command function. If you have your library in `.lib/my_lib/foo.py`, you can add the line

```
from my_lib import foo
```

at the very top of your `command.py` file.

**Note:** `dt-shell` will take care of updating your library when the user runs `dt> update` if you use `git submodule`. 
PART J
ROS development basics - Exercises

Contents

Unit J-1 - Exercise: Simple data analysis from a bag.................................182
Unit J-2 - Exercise: Bag in, bag out ..............................................................184
Unit J-3 - Exercise: Bag thumbnails ..............................................................186
Unit J-4 - Exercise: Instagram filters ............................................................188
Unit J-5 - Exercise: Bag instagram .................................................................190
Unit J-6 - Exercise: Live Instagram .................................................................191
Unit J-7 - Exercise: Augmented Reality ..........................................................192
UNIT J-1

Exercise: Simple data analysis from a bag

1.1. Skills learned
• Reading Bag files.
• Statistics functions (mean, median) in Numpy.
• Use YAML format.

1.2. Instructions
Create an implementation of `dt-bag-analyze` according to the specification below.

1.3. Specification for `dt-bag-analyze`
Create a program that summarizes the statistics of data in a bag file.

```
$ dt-bag-analyze [bag file]
```

Compute, for each topic in the bag:
• The total number of messages.
• The minimum, maximum, average, and median interval between successive messages, represented in seconds.

Print out the statistics using the YAML format. Example output:

```
$ dt-bag-analyze [bag file]
"topic name":
  num_messages: [value]
  period:
    min: [value]
    max: [value]
    average: [value]
    median: [value]
```

1.4. Useful APIs

1) Read a ROS bag
A bag is a file format in ROS for storing ROS message data. The package `rosbag` defines the class `Bag` that provides all the methods needed to serialize messages to and from a single file on disk using the bag format.

2) Time in ROS
In ROS the time is stored as an object of type `rostime.Time`. An object `t`, instance of `rostime.Time`, represents a time instant as the number of seconds since epoch (stored in `t.secs`) and the number of nanoseconds since `t.secs` (stored in `t.nsecs`). The utility function `t.to_sec()` returns the time (in seconds) as a floating number.

1.5. **Test that it works**

Download the ROS bag `example_rosbag_H3.bag`. Run your program on it and compare the results:

```bash
$ dt-bag-analyze example_rosbag.bag
/tesla/camera_node/camera_info:
   num_messages: 653
   period:  
      min: 0.01
      max: 0.05
      average: 0.03
      median: 0.03

/tesla/line_detector_node/segment_list:
   num_messages: 198
   period:  
      min: 0.08
      max: 0.17
      average: 0.11
      median: 0.1

/tesla/wheels_driver_node/wheels_cmd:
   num_messages: 74
   period:  
      min: 0.02
      max: 4.16
      average: 0.26
      median: 0.11
```
UNIT J-2

Exercise: Bag in, bag out

2.1. Skills learned
• Processing the contents of a bag to produce another bag.

2.2. Instructions
Implement the program `dt-bag-decimate` as specified below.

2.3. Specification of `dt-bag-decimate`
The program `dt-bag-decimate` takes as argument a bag filename, an integer value greater than zero, and an output bag file:

```
$ dt-bag-decimate "input bag" n "output bag"
```

The output bag contains the same topics as the input bag, however, only 1 in \(n\) messages from each topic are written. (If \(n\) is 1, the output is the same as the input.)

2.4. Useful new APIs
1) Create a new Bag
In ROS, a new bag can be created by specifying the mode `w` (i.e., write) while instantiating the class `rosbag.Bag`.
For example:

```python
from rosbag import Bag
new_bag = Bag('./output_bag.bag', mode='w')
```
Visit the documentation page for the class `rosbag.Bag` for further information.

2) Write message to a Bag
A ROS bag instantiated in `write` mode accepts messages through the function `write()`.

2.5. Check that it works
To check that the program works, you can compute the statistics of the data using the program `dt-bag-analyze` that you have created in (unknown ref exercises/exercise-bag-analysis)
You should see that the statistics have changed.
UNIT J-3

Exercise: Bag thumbnails

3.1. Skills learned

- Reading images from images topic in a bag file.

3.2. Instructions

Write a program `dt-bag-thumbnails` as specified below.

3.3. Specification for `dt-bag-thumbnails`

The program `dt-bag-thumbnails` creates thumbnails for some image stream topic in a bag file.

The syntax is:

```
$ dt-bag-thumbnails bag topic output_dir
```

This should create the files:

```
output_dir/00000.jpg
output_dir/00001.jpg
output_dir/00002.jpg
output_dir/00003.jpg
output_dir/00004.jpg
...
```

where the progressive number is an incremental counter for the frames.

3.4. Test data

If you don’t have a ROS bag to work on, you can download the test bag `example_ros-bag_H5.bag`. You should be able to get a total of 653 frames out of it.

3.5. Useful APIs

1) Read image from a topic

The `duckietown_utils` package provides the utility function `rgb_from_ros()` that processes a ROS message and returns the RGB image it contains (if any).

2) Color space conversion
In OpenCV, an image can be converted from one color space (e.g., BGR) to another supported color space (e.g., RGB). OpenCV provides a list of supported conversions. A ColorConversionCode defines a conversion between two different color spaces. An exhaustive list of color conversion codes can be found here. The conversion from a color space to another is done with the function cv.cvtColor.
UNIT J-4

Exercise: Instagram filters

4.1. Skills learned
- Image pixel representation;
- Image manipulation;
- The idea that we can manipulate operations as objects, and refer to them (higher-order computation);
- The idea that we can compose operations, and sometimes the operations do commute, while sometimes they do not.

4.2. Instructions
Create `dt-instagram` as specified below.

4.3. Specification for `dt-instagram`
Write a program `dt-instagram` that applies a list of filters to an image.
The syntax to invoke the program is:

```
$ dt-instagram image in filters image out
```

where:
- `image in` is the input image;
- `filters` is a string, which is a colon-separated list of filters;
- `image out` is the output image.
The list of filters is given in Subsection 4.3.1 - List of filters.
For example, the result of the command:

```
$ dt-instagram image.jpg flip-horizontal:grayscale out.jpg
```

is that `out.jpg` contains the input image, flipped and than converted to grayscale.
Because these two commute, this command gives the same output:

```
$ dt-instagram image.jpg grayscale:flip-horizontal out.jpg
```

1) List of filters
Here is the list of possible values for the filters, and their effect:
- `flip-vertical`: flips the image vertically
4.4. Useful new APIs

1) User defined filters

In OpenCV it is possible to define custom filters and apply them to an image. A linear filter (e.g., sepia) is defined by a linear 9-dimensional kernel. The \textit{sepia} filter is defined as:

\[
K_{\text{sepia}} = \begin{bmatrix}
0.272 & 0.534 & 0.131 \\
0.349 & 0.686 & 0.168 \\
0.393 & 0.769 & 0.189
\end{bmatrix}
\]

A linear kernel describing a color filter defines a linear transformation in the color space. A transformation can be applied to an image in OpenCV by using the function \texttt{transform}().
UNIT J-5
Exercise: Bag instagram

5.1. Instructions
Create `dt-bag-instagram` as specified below.

5.2. Specification for `dt-bag-instagram`
Write a program `dt-bag-instagram` that applies a filter to a stream of images stored in a ROS bag.
The syntax to invoke the program is:

```
$ dt-bag-instagram bag in topic filters bag out
```

where:

- `bag in` is the input bag;
- `topic` is a string containing the topic to process;
- `filters` is a string, which is a colon-separated list of filters;
- `bag out` is the output bag.

5.3. Test data
If you don’t have a ROS bag to work on, you can download the test bag `example_ros-bag_H5.bag`.

5.4. Useful new APIs

1) Compress an BGR image into a `sensor_msgs/CompressedImage` message

The `duckietown_utils` package provides the utility function `d8_compressed_image_from_cv_image()` that takes a BGR image, compresses it and wraps it into a `sensor_msgs/CompressedImage` ROS message.

5.5. Check that it works
Play your `bag out` ROS bag file and run the following command to make sure that your program is working.

```
$ rosrun image_view image_view image:=topic _image_transport:=compressed
```
UNIT J-6

Exercise: Live Instagram

6.1. Skills learned
• Live image processing

6.2. Instructions
You may find useful: Unit J-6 - Minimal ROS node - pkg_name. That tutorial is about listening to text messages and writing back text messages. Here, we apply the same principle, but to images.

Create a ROS node that takes camera images and applies a given operation, as specified in the next section, and then publishes it.

6.3. Specification for the node dt_live_instagram_robot name_node
Create a ROS node dt_live_instagram_robot name_node that takes a parameter called filter, where the filter is something from the list Subsection 4.3.1 - List of filters.

You should launch your camera and joystick from ‘~/duckietown’ with

$ make demo-joystick-camera

Then launch your node with

$ roslaunch dt_live_instagram_robot name_node.launch filter:=filter

This program should do the following:
• Subscribe to the camera images, by finding a topic that is called .../compressed. Call the name of the topic topic (i.e., topic = ...).
• Publish to the topic topic / filter/compressed a stream of images (i.e., video frames) where the filter is applied to the images.

6.4. Check that it works

$ rqt_image_view

and look at topic / filter /compressed
UNIT J-7

Exercise: Augmented Reality

7.1. Skills learned

- Understanding of all the steps in the image pipeline.
- Writing markers on images to aid in debugging.

7.2. Introduction

During the lectures, we have explained one direction of the image pipeline:

```
image -> [feature extraction] -> 2D features -> [ground projection]
-> 3D world coordinates
```

In this exercise, we are going to look at the pipeline in the opposite direction. It is often said that:

“The inverse of computer vision is computer graphics.”

The inverse pipeline looks like this:

```
3D world coordinates -> [image projection] -> 2D features -> [rendering] -> image
```

7.3. Instructions

- Do intrinsics/extrinsics camera calibration of your robot as per the instructions. Write the ROS node specified below in (unknown ref exercises/exercise-augmented-reality-spec)

previous warning next (2 of 3) index

**warning**

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#exercises/exercise-augmented-reality-spec'.

Location not known more precisely.

Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

Then verify the results in the following 3 situations.

1) Situation 1: Calibration pattern
• Put the robot in the middle of the calibration pattern.
• Run the program `dt_augmented_reality` with map file `calibration_pattern.yaml`.
  (Adjust the position until you get perfect match of reality and augmented reality.)

2) Situation 2: Lane

• Put the robot in the middle of a lane.
• Run the program `dt_augmented_reality` with map file `lane.yaml`.
  (Adjust the position until you get a perfect match of reality and augmented reality.)

3) Situation 3: Intersection

• Put the robot at a stop line at a 4-way intersection in Duckietown.
• Run the program `dt_augmented_reality` with map file `intersection_4way.yaml`.
  (Adjust the position until you get a perfect match of reality and augmented reality.)

4) Submission

Submit the images according to location-specific instructions.

7.4. Specification of `dt_augmented_reality`

In this assignment you will be writing a ROS package to perform the augmented reality exercise. The program will be invoked with the following syntax:

```
$ roslaunch dt_augmented_reality-robot name augmenter.launch
map_file:=map file robot_name:=robot name local:=1
```

where `map file` is a YAML file containing the map (specified in (unknown reference exercises/exercise-augmented-reality-map))

If `robot name` is not given, it defaults to the hostname.

The program does the following:
1. It loads the intrinsic / extrinsic calibration parameters for the given robot.
2. It reads the map file.
3. It listens to the image topic `/robot name/camera_node/image/compressed`.

4. It reads each image, projects the map features onto the image, and then writes the resulting image to the topic `!/robot name!/AR/!/map file basename!/image/compressed`

where `map file basename` is the basename of the file without the extension.

We provide you with ROS package template that contains the AugmentedRealityNode. By default, launching the AugmentedRealityNode should publish raw images from the camera on the new `/robot name/AR/map file basename/image/compressed` topic.

In order to complete this exercise, you will have to fill in the missing details of the Augmenter class by doing the following:

1. Implement a method called `process_image` that undistorts raw images.
2. Implement a method called `ground2pixel` that transforms points in ground coordinates (i.e. the robot reference frame) to pixels in the image.
3. Implement a method called `callback` that writes the augmented image to the appropriate topic.

7.5. Specification of the map

The map file contains a 3D polygon, defined as a list of points and a list of segments that join those points.

The format is similar to any data structure for 3D computer graphics, with a few changes:

1. Points are referred to by name.
2. It is possible to specify a reference frame for each point. (This will help make this into a general tool for debugging various types of problems).

Here is an example of the file contents, hopefully self-explanatory.

The following map file describes 3 points, and two lines.

```plaintext
points:
    # define three named points: center, left, right
    center: [axle, [0, 0, 0]] # [reference frame, coordinates]
    left: [axle, [0.5, 0.1, 0]]
    right: [axle, [0.5, -0.1, 0]]

segments:
    - points: [center, left]
      color: [rgb, [1, 0, 0]]
    - points: [center, right]
      color: [rgb, [1, 0, 0]]
```

1) Reference frame specification

The reference frames are defined as follows:

- `axle`: center of the axle; coordinates are 3D.
• camera: camera frame; coordinates are 3D.
• image01: a reference frame in which 0,0 is top left, and 1,1 is bottom right of the image; coordinates are 2D.
(Other image frames will be introduced later, such as the world and tile reference frame, which need the knowledge of the location of the robot.)

2) Color specification

RGB colors are written as:

\[[\text{rgb}, \ [R, G, B]]\]

where the RGB values are between 0 and 1. Moreover, we support the following strings:

- red is equivalent to \[[\text{rgb}, \ [1,0,0]]\]
- green is equivalent to \[[\text{rgb}, \ [0,1,0]]\]
- blue is equivalent to \[[\text{rgb}, \ [0,0,1]]\]
- yellow is equivalent to \[[\text{rgb}, \ [1,1,0]]\]
- magenta is equivalent to \[[\text{rgb}, \ [1,0,1]]\]
- cyan is equivalent to \[[\text{rgb}, \ [0,1,1]]\]
- white is equivalent to \[[\text{rgb}, \ [1,1,1]]\]
- black is equivalent to \[[\text{rgb}, \ [0,0,0]]\]

7.6. “Map” files

1) hud.yaml

This pattern serves as a simple test that we can draw lines in image coordinates:

```
points:
    TL: [image01, [0, 0]]
    TR: [image01, [0, 1]]
    BR: [image01, [1, 1]]
    BL: [image01, [1, 0]]

segments:
    - points: [TL, TR]
      color: red
    - points: [TR, BR]
      color: green
    - points: [BR, BL]
      color: blue
    - points: [BL, TL]
      color: yellow
```

The expected result is to put a border around the image: red on the top, green on the
right, blue on the bottom, yellow on the left.

2) **calibration_pattern.yaml**

This pattern is based off the checkerboard calibration target used in estimating the intrinsic and extrinsic camera parameters:

```yaml
points:
  TL: [axle, [0.315, 0.093, 0]]
  TR: [axle, [0.315, -0.093, 0]]
  BR: [axle, [0.191, -0.093, 0]]
  BL: [axle, [0.191, 0.093, 0]]

segments:
  - points: [TL, TR]
    color: red
  - points: [TR, BR]
    color: green
  - points: [BR, BL]
    color: blue
  - points: [BL, TL]
    color: yellow
```

The expected result is to put a border around the inside corners of the checkerboard: red on the top, green on the right, blue on the bottom, yellow on the left.

3) **lane.yaml**

We want something like this:

```
0
|   |          | . |             |   |
|   |          | . |             |   |
|   |          | . |             |   |
|   |          | . |             |   |
|   |          | . |             |   |
|   |          | . |             |   |
|   |          | . |             |   |
WW      L       WY      L         WW
```

Then we have:
points:
  p1: [axle, [0, 0.2794, 0]]
  q1: [axle, [D, 0.2794, 0]]
  p2: [axle, [0, 0.2286, 0]]
  q2: [axle, [D, 0.2286, 0]]
  p3: [axle, [0, 0.0127, 0]]
  q3: [axle, [D, 0.0127, 0]]
  p4: [axle, [0, -0.0127, 0]]
  q4: [axle, [D, -0.0127, 0]]
  p5: [axle, [0, -0.2286, 0]]
  q5: [axle, [D, -0.2286, 0]]
  p6: [axle, [0, -0.2794, 0]]
  q6: [axle, [D, -0.2794, 0]]

segments:
  - points: [p1, q1]
    color: white
  - points: [p2, q2]
    color: white
  - points: [p3, q3]
    color: yellow
  - points: [p4, q4]
    color: yellow
  - points: [p5, q5]
    color: white
  - points: [p6, q6]
    color: white

4) intersection_4way.yaml
points:
NL1: [axle, [0.247, 0.295, 0]]
NL2: [axle, [0.347, 0.301, 0]]
NL3: [axle, [0.218, 0.256, 0]]
NL4: [axle, [0.363, 0.251, 0]]
NL5: [axle, [0.400, 0.287, 0]]
NL6: [axle, [0.409, 0.513, 0]]
NL7: [axle, [0.360, 0.314, 0]]
NL8: [axle, [0.366, 0.456, 0]]
NC1: [axle, [0.372, 0.007, 0]]
NC2: [axle, [0.145, 0.008, 0]]
NC3: [axle, [0.374, -0.0216, 0]]
NC4: [axle, [0.146, -0.0180, 0]]
NR1: [axle, [0.209, -0.234, 0]]
NR2: [axle, [0.349, -0.237, 0]]
NR3: [axle, [0.242, -0.276, 0]]
NR4: [axle, [0.319, -0.274, 0]]
NR5: [axle, [0.402, -0.283, 0]]
NR6: [axle, [0.401, -0.479, 0]]
NR7: [axle, [0.352, -0.415, 0]]
NR8: [axle, [0.352, -0.303, 0]]
CL1: [axle, [0.586, 0.261, 0]]
CL2: [axle, [0.595, 0.632, 0]]
CL3: [axle, [0.618, 0.251, 0]]
CL4: [axle, [0.637, 0.662, 0]]
CR1: [axle, [0.565, -0.253, 0]]
CR2: [axle, [0.567, -0.607, 0]]
CR3: [axle, [0.610, -0.262, 0]]
CR4: [axle, [0.611, -0.641, 0]]
FL1: [axle, [0.781, 0.718, 0]]
FL2: [axle, [0.763, 0.253, 0]]
FL3: [axle, [0.863, 0.192, 0]]
FL4: [axle, [1.185, 0.172, 0]]
FL5: [axle, [0.842, 0.718, 0]]
FL6: [axle, [0.875, 0.271, 0]]
FL7: [axle, [0.879, 0.234, 0]]
FL8: [axle, [1.180, 0.209, 0]]
FC1: [axle, [0.823, 0.0162, 0]]
FC2: [axle, [1.172, 0.00117, 0]]
FC3: [axle, [0.845, -0.0100, 0]]
FC4: [axle, [1.215, -0.0181, 0]]
FR1: [axle, [0.764, -0.695, 0]]
FR2: [axle, [0.768, -0.263, 0]]
FR3: [axle, [0.810, -0.202, 0]]
FR4: [axle, [1.203, -0.196, 0]]
FR5: [axle, [0.795, -0.702, 0]]
FR6: [axle, [0.803, -0.291, 0]]
FR7: [axle, [0.832, -0.240, 0]]
FR8: [axle, [1.210, -0.245, 0]]

segments:
- points: [NL1, NL2]
  color: white
- points: [NL3, NL4]
7.7. Suggestions
Start by using the file `hud.yaml`. To visualize it, you do not need the calibration data. It will be helpful to make sure that you can do the easy parts of the exercise: loading the map, and drawing the lines.

7.8. Useful APIs

1)  Loading a map file:
To load a map file, use the function `load_map` provided in `duckietown_utils`:

```python
from duckietown_utils import load_map
map_data = load_map(map_filename)
```

(Note that `map` is a reserved symbol name in Python.)

2)  Reading the calibration data for a robot
To load the *intrinsic* calibration parameters, use the function `load_camera_intrinsics` provided in `duckietown_utils`:

```python
from duckietown_utils import load_camera_intrinsics
intrinsics = load_camera_intrinsics(robot_name)
```

To load the *extrinsic* calibration parameters (i.e. ground projection), use the function `load_homography` provided in `duckietown_utils`:

```python
from duckietown_utils import load_homography
H = load_homography(robot_name)
```

3)  Path name manipulation
From a file name like `/path/to/map1.yaml`, you can obtain the basename without extension `yaml` by using the function `get_base_name` provided in `duckietown_utils`:

```python
from duckietown_utils import get_base_name
filename = '/path/to/map1.yaml'
map_name = get_base_name(filename) # = 'map1'
```

4)  Undistorting an image
To remove the distortion from an image, use the function `rectify` provided in `duckietown_utils`:
from duckietown_utils import rectify
rectified_image = rectify(image, intrinsics)

5) Drawing primitives

To draw the line segments specified in a map file, use the `render_segments` method defined in the `Augmenter` class:

class Augmenter():
    
    # ...
    
    def ground2pixel(self):
        """Method that transforms ground points to pixel coordinates"""
        # Your code goes here.
        return "???"

    image_with_drawn_segments = augmenter.render_segments(image)

In order for `render_segments` to draw segments on an image, you must first implement the method `ground2pixel`. 

---

**EXERCISE: AUGMENTED REALITY**