Duckietown Software development guide

This part is about how to develop software in Duckietown
PART A
Development environment

This part describes how to set up a proper development environment, including ergonomics.
UNIT A-1
IDEs
Unit A-2

Shells
UNIT A-3
Using Makefiles
PART B

Git for development

This guide provides a basic overview on how do version control with git.

0.1. Background reading

- Github handbook
- Github tutorial
- Github workflow
- Github branching
- Github cheatsheet
- Github bootcamp
1.1. **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).

1.2. **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"
```
2.1. Github 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.

The main Duckietown repository is https://github.com/duckietown/Software/

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.

2.2. Github basic commands

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
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:

```
$ git commit -m "commit-message"
```

If everything went smooth without any issues you are ready to push your changes to your branch:

```
$ 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

```
$ 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!):

```sh
$ git branch -d branch-name
```

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

```sh
$ 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.

For example you encounter a bug or a mistake in this operating manual, please visit this repository to open a new issue.
3.1. **Problem:** **https** instead of **ssh**

The symptom is:

```sh
$ git push
Username for 'https://github.com':
```

Diagnosis: the `remote` is not correct.
If you do `git remote` you get entries with `https:`

```sh
$ git remote -v
origin https://github.com/duckietown/Software.git (fetch)
origin https://github.com/duckietown/Software.git (push)
```

Expectation:

```sh
$ git remote -v
origin git@github.com:duckietown/Software.git (fetch)
origin git@github.com:duckietown/Software.git (push)
```

Solution:

```sh
$ git remote remove origin
$ git remote add origin git@github.com:duckietown/Software.git
```

3.2. **Problem:** **git push** complains about upstream

The symptom is:

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

Solution:

```sh
$ git push --set-upstream origin `branch name`
```
PART C

Python programming basics

This part describes the basics of Python programming and some of the conventions for Duckietown.
1.1. Background reading
- Python
- Python tutorial

1.2. Python packages

1.3. Python virtual environments
Install using:

```
$ sudo apt install virtualenv
```

1.4. Useful libraries
```
matplotlib
seaborn
numpy
panda
scipy
opencv
...`
```

1.5. Context managers

1.6. Exception hierarchies

1.7. Object orientation - Abstract classes, class hierarchies

1.8. Idioms
```
segment_list = copy.deepcopy(segment_list)
- add_duckietown_header
-`
```
UNIT C-2
Duckietown code conventions

2.1. Python

1) Tabs

Never use tabs in Python files. The tab characters are evil in Python code. Please 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.

2) Spaces

Indentation is 4 spaces.

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 do 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.

4) The encoding line

All files must have an encoding declared, and this encoding must be utf-8:

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

5) Sha-bang lines

Executable files start with:

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

6) Comments

Comments refer to the next line. Comments, bad:

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

Comments, better:

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

2.3. Exceptions

2.4. Script entry points

```python
def summary():
    fs = get_all_configuration_files()

    if __name__ == '__main__':
        wrap_script_entry_point(summary)
```

1) Imports

Do not do a star import, like the following:

```python
from rostest_example.Quacker import *
```
UNIT C-3

Working with YAML files

YAML files are useful for writing configuration files, and are used a lot in Duckietown.

3.1. Pointers to documentation

3.2. Editing YAML files

3.3. Reading and writing YAML files in Python
4.1. **IPython**

How to enter IPython:

```python
from IPython import embed()

a = 10
embed() # enters interactive mode
```
PART D

Python programming basics exercises
PART E

Basic Docker development
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
```

Note how Daphne's duck container has a container ID, a base image, and a funny-look-
ing 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:

```
$ 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:

```
$ 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

```
total 0
-rw-r--r-- 1 root root 0 May 21 21:32 new_file1
```

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:

```bash
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
  --- e3b75gt9zyc4
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
----> 3358ca1b8649
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
----> a8be23fe573
Successfully built a8be23fe573
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
----> a8be23fe573
Successfully built a8be23fe573
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 
  ---> a8be23fca573 
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

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

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:
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM daphne/duck
       ---> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
       ---> e0e03d9e1791
Step 3/3 : RUN cat duck.txt
       ---> Running in 590c5420ce29
Make way for duckies!
Removing intermediate container 590c5420ce29
       ---> 1633e3e10bef
Successfully built 1633e3e10bef
Successfully tagged 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

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

we have:

Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM ubuntu
       ---> cd6d8154f1e1
Step 2/3 : COPY duck.txt .
       ---> f219efc150a5
Step 3/3 : RUN cat duck.txt
       ---> Running in 7c6f5f8b73e9
Make way!
Thank you. Have a nice day!
Removing intermediate container 7c6f5f8b73e9
       ---> e8a1db712ae
Successfully built e8a1db712ae
Successfully tagged 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

```
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:

```
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 `Dockerfile` 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:

```
$ 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 1.4. 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.
Figure 1.6. 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).

Figure 1.8. Docker Cloud allows a longer build timeout, and has more sophisticated build configuration features.
UNIT E-2

Duckiebot Development using Docker

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 E-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:

```
$ dts init_sd_card
```
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 alternately, 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:

![Portainer Container View](image)

Figure 2.2. Portainer Container View

You can attach a console to a running container and interact with it via the browser:
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 -v /data:/data -p 8082:8082 duckietown/rpi-simple-server:master18
```

Go to the following URL: [http://DUCKIEBOT_NAME.local:8082/](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.
Figure 2.6. Portainer PiCam Demo

```bash
$ 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.

### 2.5. Testing ROS

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

```bash
$ 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_NAME 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 B-10 - 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](#).

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:

```
$ 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](https://github.com/duckietown/scripts)

2) **RPi Camera Test container**

   - [https://github.com/rusi/rpi-docker-python-picamera](https://github.com/rusi/rpi-docker-python-picamera)
   - [https://hub.docker.com/r/duckietown/rpi-docker-python-picamera/](https://hub.docker.com/r/duckietown/rpi-docker-python-picamera/)

3) **RPi Simple HTTP File 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**
   - https://github.com/ros-industrial/docker
   - https://hub.docker.com/r/rosindustrial/ros-robot-nvidia:
• https://hub.docker.com/r/osrf/ros/

### 2.14. Docker Image Hierarchy

![Docker Image Hierarchy Diagram](image)

**Figure 2.8. Docker Image Hierarchy**

### 2.15. Misc

1) **Building images:**
2) Transferring Docker containers

```
$ docker save \(\text{TAG\_NAME}\) | gzip | ssh -C duckie@\(\text{DUCKIEBOT\_NAME}\)\.local docker load
```

Figure 2.10. Output of `rqt_dep joystick` (compilation dependencies)

Figure 2.12. Output of `rqt_graph joystick` (runtime dependencies)
UNIT E-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.
PART F
Basic Docker development exercises
PART G
Creating programs that work

This part describes techniques for testing and continuous integration.
UNIT G-1

Creating unit tests with ROS

catkin_make -C catkin_ws/ --pkg easy_logs
UNIT G-2
Continuous integration

These are the conventions for the Duckietown repositories.

2.1. 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 2.2).

Figure 2.2. The green dot is good.

2.2. How to stay in the green

The system enforces the constraint that the branch master is always green, by preventing 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”).
2.3. 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 2.6).
(5) You now have an opportunity to summarize all the changes you did so far (Figure 2.8). Then click “create pull request”.

Open a pull request
Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

Figure 2.8. Preparing the pull request

(6) 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 2.10). Click “details” to see what’s going on, or just wait.

Figure 2.10. Wait for the checks to finish

When it's done, you will see either a success message (Figure 2.12) or a failure message
Figure 2.12. The tests are done

Figure 2.14. The tests have failed

(7) At this point, you can click “squash and merge” to merge the changes into master (Figure 2.16).

Figure 2.16. The final step

1) Troubleshooting

If you see a message like “merge failed” (Figure 2.18), it probably means that somebody pushed into master; merge master into your branch and continue the process.

Figure 2.18. Merge failed
UNIT G-3

Bug squashing guide

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.

3.1. Historical notes

First, count your blessings. You are lucky to live in the present. Once, there were actual bugs in your computer (Figure 3.2).

Figure 3.2. "First actual case of bug being found." Read the story here.

3.2. 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.
3.3. 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.

3.4. 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.

So, first of all, run what-the-duck. Then, fix the errors that what-the-duck shows you.

This is the proper way to run what-the-duck:
3.5. 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.

3.6. How to give help

1) Step 1: Ask for the output of `what-the-duck`

If there are errors reported, the students should fix those before worrying about their current problem. Maybe you or they don’t see the connection, but the connection might be there.

Also, in general, errors in the environment will cause other problems later on.

2) Step 2: 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 H

Working with images
UNIT H-1
Duckietown utility library

1.1. Images
This section contains the documentation about the utility functions used for image processing available in the duckietown_utils Python package.

1) Function write_image_as_jpg

**Description:** Takes an BGR image and writes it as a JPEG file.

```python
Are we sure that the encoding is right? -AC
```

**Prototype:**

```python
write_image_as_jpg(image, filename)
```

**Defined in:** image_writing.py

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>numpy.ndarray</td>
<td>The BGR image to save as JPEG file.</td>
</tr>
<tr>
<td>filename</td>
<td>str</td>
<td>The path of the JPEG file.</td>
</tr>
</tbody>
</table>

**Returns:** None.

2) Function rgb_from_ros

**Description:** Takes a ROS message containing an image and returns its RGB representation.

**Prototype:**

```python
rgb_from_ros(msg)
```

**Defined in:** image_conversions.py

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>sensor_msgs.Image or sensor_msgs.CompressedImage</td>
<td>Message containing the image to extract.</td>
</tr>
</tbody>
</table>

**Returns:** numpy.ndarray :: RGB representation of the image contained in the ROS message msg.

3) Function d8_compressed_image_from_cv_image

**Description:** Takes a OpenCV image (BGR format), compresses it and wraps it into a
ROS message of type `sensor_msgs.CompressedImage`.

Prototype:

```python
d8_compressed_image_from_cv_image( image_cv )
```

Defined in: `image_jpg_create.py`.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image_cv</td>
<td><code>numpy.ndarray</code></td>
<td>BGR representation of the image to compress.</td>
</tr>
</tbody>
</table>

Returns: `sensor_msgs.CompressedImage`: A ROS message containing a compressed version of the input `image_cv`.  

PART I

Working with images - Exercises
Unit I-1

Exercise: Basic image operations

1.1. Skills learned
- Accessing command line arguments.
- Reading and writing files.
- Working with pixel-based image representations.
- OpenCV and duckietown_utils APIs for reading/writing images.

1.2. Instructions
Create an implementation of the program `dt-image-flip0`, specified below.
If this exercise is too easy for you, skip to Unit A-2 - Exercise: Basic image operations, adult version.

1.3. Specification of `dt-image-flip0`
The program `dt-image-flip0` takes as an argument a JPG file with extension `.jpg`:

```
$ dt-image-flip0 file.jpg
```

and creates a file called `file.flipped.jpg` that is flipped around the horizontal axis.

(b) The original picture.

(d) The flipped output

Figure 1.2
Example input-output for the program `dt-image-flip`.

1.4. Useful APIs
1) Load image from file

The OpenCV library provides a utility function called `imread` that loads an image from a file.

2) Flip an image

comment
This is the kind of thing that they need to figure out how to do with pixels. -AC

3) Write an image to a file

The `duckietown_utils` package provides the utility function `write_image_as_jpg()` that writes an image to a JPEG file.
Exercise: Basic image operations, adult version

2.1. Skills learned
- Dealing with exceptions.
- Using exit conditions.
- Verification and unit tests.

2.2. Instructions
Implement the program `dt-image-flip` specified in the following section.
This time, we specify exactly what should happen for various anomalous conditions.
This allows to do automated testing of the program.

2.3. `dt-image-flip` specification
The program `image-ops` expects exactly two arguments: a filename (a JPG file) and a directory name.

```
$ dt-image-flip file outdir
```

If the file does not exist, the script must exit with error code 2.
If the file cannot be decoded, the script must exit with error code 3.
If the file exists, then the script must create:
- `outdir/regular.jpg`: a copy of the initial file
- `outdir/flip.jpg`: the file, flipped vertically.
- `outdir/side-by-side.jpg`: the two files, side by side.
If any other error occurs, the script should exit with error code 99.

(b) The original picture.

(d) The output `flip.jpg`
2.4. Useful APIs

1) Images side-by-side

An image loaded using the OpenCV function `imread` is stored in memory as a NumPy array. For example, the image shown above (Figure 2.2b - The original picture.) will be represented in memory as a NumPy array with shape (96, 96, 3). The first dimension indicates the number of pixels along the Y-axis, the second indicates the number of pixels along the X-axis and the third is known as number of channels (e.g., Blue, Green, and Red).

NumPy provides a utility function called `concatenate` that joins a sequence of arrays along a given axis.

2.5. Testing it works with `image-ops-tester`

We provide 4 scripts that can be used to make sure that you wrote a conforming `dt-image-flip`. The scripts are `image-ops-tester-good`, `image-ops-tester-bad1`, `image-ops-tester-bad2`, and `image-ops-tester-bad3`. You can find them in the directory `/exercises/dt-image-flip/image-ops-tester` in the `duckietown/duckuments` repository.

The script called `image-ops-tester-good` tests your program in a situation in which we expect it to work properly. The 3 “bad” test scripts (i.e., `image-ops-tester-bad1` through `image-ops-tester-bad3`) test your code in situations in which we expect your program to complain in the proper way.

Use them as follows:

```
$ image-ops-tester- [scenario] candidate-program
```

**Note:** The tester scripts must be called from their own location. Make sure to change your working directory to `/exercises/dt-image-flip/image-ops-tester` before launching the tester scripts. If the script cannot be found, `image-ops-tester- [scenario]` will return 1.

`image-ops-tester- [scenario]` will return 0 if the program exists and conforms to the specification (Section 2.3 - `dt-image-flip` specification).
If it can establish that the program is not good, it will return 11.

**Bottom line**

Things that are not tested are broken.
PART J
Basic ROS development
Unit J-1

Introduction to ROS

Robot Operating System (ROS) is an open-source framework for software development on robots. It is widely used in research and industry and we use it for running our duckiebots.

This guide will give an overview of ROS specific to Duckietown as well as point out some useful tools for developing with ROS. If you are looking for a general guide to ROS there exist excellent tutorials on the official wiki.

Additionally there is a course at ETHZ Programming for Robotics - ROS although they use C++.
2.1. ROS Version

The version of ROS we are running on the duckiebots is ROS Kinetic. The current long term service version is ROS Melodic however it is quite new and not as well documented as Kinetic which will be supported until April 2021. You can install Melodic on your personal laptop and will be able to communicate with the duckiebot but may have issues building the duckiebot software on your own laptop.

2.2. ROS Installation

See the official guide for installing ROS.

2.3. ROS Master

The ROS master is a process which manages the connections between different nodes. It is typically run using the `roscore` command. We launch a container specifically for the ROS master with:

```
$ docker -H hostname.local run -dit --privileged --name roscore --net host --restart unless-stopped duckietown/rpi-ros-kinetic-roscore:master18
```

Let's try to communicate with ROS master on our laptops. First we need to tell our computer where ROS master is. we do this by setting the environment variable `ROS_MASTER_URI` with the command

```
$ export ROS_MASTER_URI=http://hostname.local:11311/
```

Keep in mind this variable setting will not persist if you close the terminal. One option is to put that line in your `.bashrc` or `.zshrc` but then you wont be able to connect to any ROS master on your machine unless you reset it.
Rosbags are a system for logging data in ROS.

3.1. Commands

$ rosbag record <topic names> -O bagname
$ rosbag info
UNIT J-4

Node configuration mechanisms
Software development in ROS takes place in a root directory called a catkin workspace, usually called \texttt{catkin_ws}. Inside this directory is a \texttt{src} folder which contains all the packages. ROS is built by several packages which each perform a specific role such as defining rosnodes, actions or message definitions. See \texttt{joy_mapper} as an example of a ROS package. It can be launched independently with

\begin{verbatim}
$ roslaunch launch/joy_mapper_test.launch
\end{verbatim}

however we usually launch these nodes as part of a container.

5.1. Catkin Tools

Catkin tools are recommended as they let you use \texttt{catkin build} rather than \texttt{catkin_make} as well as several other commands.

5.2. Creating Nodes

Best practice is to define a class which contains several member functions. You can bind those functions as callback functions when subscribing to a topic or call on them when you want to publish something. You should also make use of timers rather than use while loops which block.

For example in \texttt{joy_mapper} the JoyMapper subscribes to the “joy” topic with callback function \texttt{cbJoy}. \texttt{joy_mapper} also uses a timer with callback \texttt{cbParamTimer} which is called every second to update parameters.

Notice that the main function simply creates the rosnode, instantiates the JoyMapper class then calls \texttt{rospy.spin()} which keeps the process at that line until it is called to exit.
This document outlines the process of writing a ROS package and nodes in Python. To follow along, it is recommended that you duplicate the pkg_name folder and edit the content of the files to make your own package.

### 6.1. The files in the package

1) **CMakeLists.txt**

We start with the file `CMakeLists.txt`. Every ROS package needs a file `CMakeLists.txt`, even if you are just using Python code in your package.

```bash
# documentation about CMakeLists.txt
```

For a Python package, you only have to pay attention to the following parts.

The line:

```bash
project(pkg_name)
```

defines the name of the project.

The `find_package` lines:

```bash
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  duckietown_msgs # Every duckietown packages must use this.
  std_msgs
)
```

You will have to specify the packages on which your package is dependent. In Duckietown, most packages depend on `duckietown_msgs` to make use of the customized messages.

The line:

```bash
catkin_python_setup()
```

tells `catkin` to setup Python-related stuff for this package.

* **ROS documentation about setup.py**
2) package.xml

The file `package.xml` defines the meta data of the package. Catkin makes use of it to flush out the dependency tree and figures out the order of compiling.

Pay attention to the following parts.

- `<name>` defines the name of the package. It has to match the project name in `CMakeLists.txt`.
- `<description>` describes the package concisely.
- `<maintainer>` provides information of the maintainer.
- `<build_depend>` and `<run_depend>`: The catkin packages this package depends on. This usually matches the `find_package` in `CMakeLists.txt`.

3) setup.py

The file `setup.py` configures the Python modules in this package.

The part to pay attention to is

```python
setup_args = generate_distutils_setup(  
    packages=['pkg_name'],
    package_dir={'': 'include'},
)
```

The `packages` parameter is set to a list of strings of the name of the folders inside the `include` folder.

The convention is to set the folder name the same as the package name. Here it’s the `include/pkg_name` folder.

You should put ROS-independent and/or reusable module (for other packages) in the `include/pkg_name` folder.

Python files in this folder (for example, the `util.py`) will be available to scripts in the catkin workspace (this package and other packages too).

To use these modules from other packages, use:

```python
from pkg_name.util import *
```

6.2. Writing a node: talker.py

Let’s look at `src/talker.py` as an example.

ROS nodes are put under the `src` folder and they have to be made executable to function properly.

→ You use `chmod` for this; see Section 12.1 - `chmod`.

1) Header

Header:
# Imports module. Not limited to modules in this package.
from pkg_name.util import HelloGoodbye
# Imports msg
from std_msgs.msg import String

The first line, `#!/usr/bin/env python`, specifies that the script is written in Python. Every ROS node in Python must start with this line.

The line `import rospy` imports the `rospy` module necessary for all ROS nodes in Python.

The line `from pkg_name.util import HelloGoodbye` imports the class `HelloGoodbye` defined in the file `pkg_name/util.py`.

Note that you can also include modules provided by other packages, if you specify the dependency in `CMakeLists.txt` and `package.xml`.

The line `from std_msgs.msg import String` imports the `String` message defined in the `std_msgs` package.

Note that you can use `rosmsg show std_msgs/String` in a terminal to lookup the definition of `String.msg`.

2) Main

This is the main file:

```python
if __name__ == '__main__':
    # Initialize the node with rospy
    rospy.init_node('talker', anonymous=False)

    # Create the NodeName object
    node = Talker()

    # Setup proper shutdown behavior
    rospy.on_shutdown(node.on_shutdown)

    # Keep it spinning to keep the node alive
    rospy.spin()
```

The line `rospy.init_node('talker', anonymous=False)` initializes a node named `talker`.

Note that this name can be overwritten by a launch file. The launch file can also push this node down namespaces. If the `anonymous` argument is set to `True` then a random string of numbers will be append to the name of the node. Usually we don't use anonymous nodes.

The line `node = Talker()` creates an instance of the `Talker` object. More details in the next section.

The line `rospy.on_shutdown(node.on_shutdown)` ensures that the `node.on_shutdown` will be called when the node is shutdown.
The line `rospy.spin()` blocks to keep the script alive. This makes sure the node stays alive and all the publication/subscriptions work correctly.

### 6.3. The Talker class

We now discuss the Talker class in `talker.py`.

1) Constructor

In the constructor, we have:

```python
self.node_name = rospy.get_name()
```

saves the name of the node. This allows to include the name of the node in printouts to make them more informative. For example:

```python
rospy.loginfo('[%s] Initializing.' % (self.node_name))
```

The line:

```python
self.pub_topic_a = rospy.Publisher('~topic_a', String, queue_size=1)
```

defines a publisher which publishes a `String` message to the topic `~topic_a`. Note that the `~` in the name of topic under the namespace of the node. More specifically, this will actually publish to `talker/topic_a` instead of just `topic_a`. The `queue_size` is usually set to 1 on all publishers.

- For more details see [rospy overview: publisher and subscribers](https://github.com/ros/design-guidelines/blob/master/rospy-overview.md#publisher).

The line:

```python
self.sub_topic_b = rospy.Subscriber('~topic_b', String, self.cbTopic)
```

defines a subscriber which expects a `String` message and subscribes to `~topic_b`. The message will be handled by the `self.cbTopic` callback function. Note that similar to the publisher, the `~` in the topic name puts the topic under the namespace of the node. In this case the subscriber actually subscribes to the topic `talker/topic_b`.

It is strongly encouraged that a node always publishes and subscribes to topics under their `node_name` namespace. In other words, always put a `~` in front of the topic names when you defines a publisher or a subscriber. They can be easily remapped in a launch file. This makes the node more modular and minimizes the possibility of confusion and naming conflicts. See [the launch file section](https://github.com/ros/design-guidelines/blob/master/launch-guide.md) for how remapping works.

The line:

```python
self.pub_timestep = self.setupParameter('~pub_timestep', 1.0)
```
Sets the value of `self.pub_timestep` to the value of the parameter `~pub_timestep`. If the parameter doesn’t exist (not set in the launch file), then set it to the default value 1.0. The `setupParameter` function also writes the final value to the parameter server. This means that you can `rosparam list` in a terminal to check the actual values of parameters being set.

The line:

```python
self.timer = rospy.Timer(rospy.Duration.from_sec(self.pub_timestep),
     self.cbTimer)
```

defines a timer that calls the `self.cbTimer` function every `self.pub_timestep` seconds.

2) Timer callback

Contents:

```python
def cbTimer(self,event):
    singer = HelloGoodbye()
    # Simulate hearing something
    msg = String()
    msg.data = singer.sing("duckietown")
    self.pub_topic_name.publish(msg)
```

Every time the timer ticks, a message is generated and published.

3) Subscriber callback

Contents:

```python
def cbTopic(self,msg):
    rospy.loginfo("[%s] %s" %(self.node_name,msg.data))
```

Every time a message is published to `~topic_b`, the `cbTopic` function is called. It simply prints the message using `rospy.loginfo`.

6.4. Launch File

You should always write a launch file to launch a node. It also serves as a documentation on the I/O of the node.

Let’s take a look at `launch/test.launch`. 74 MINIMAL ROS NODE - PKG_NAME
For the `<node>`, the `name` specify the name of the node, which overwrites `rospy.init_node()` in the `__main__` of `talker.py`. The `pkg` and `type` specify the package and the script of the node, in this case it's `talker.py`. Don't forget the `.py` in the end (and remember to make the file executable through `chmod`).

The `output="screen"` direct all the `rospy.loginfo` to the screen, without this you won't see any printouts (useful when you want to suppress a node that's too talkative.) The `<param>` can be used to set the parameters. Here we set the `~pub_timestep` to 0.5. Note that in this case this sets the value of `talker/pub_timestep` to 0.5.

The `<remap>` is used to remap the topic names. In this case we are replacing `~topic_b` with `~topic_a` so that the subscriber of the node actually listens to its own publisher. Replace the line with

```
<remap from="~topic_b" to="talker/topic_a"/>
```

will have the same effect. This is redundant in this case but very useful when you want to subscribe to a topic published by another node.

### 6.5. Testing the node

First of all, you have to `catkin_make` the package even if it only uses Python. `catkin` makes sure that the modules in the include folder and the messages are available to the whole workspace. You can do so by

```
$ cd ${DUCKIETOWN_ROOT}/catkin_ws
$ catkin_make
```

Ask ROS to re-index the packages so that you can auto-complete most things.

```
$ rospack profile
```

Now you can launch the node by the launch file.

```
$ roslaunch pkg_name test.launch
```

You should see something like this in the terminal:
... logging to /home/username/.ros/log/
d4db7c80-b272-11e5-8800-5c514fb7f0ed/roslaunch-robot name-15961.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://robot name.local:33925/

SUMMARY
=======
PARAMETERS
  * /rosdistro: $ROS_DISTRO
  * /rosversion: 1.11.16
  * /talker/pub_timestep: 0.5

NODES
/
  talker (pkg_name/talker.py)

auto-starting new master
process[master]: started with pid [15973]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to d4db7c80-b272-11e5-8800-5c514fb7f0ed

process[rosout-1]: started with pid [15986]

started core service [/rosout]

process[talker-2]: started with pid [15993]

[INFO] [WallTime: 1451864197.780158] [/talker] ~pub_timestep = 0.5

Open another terminal and run:

$ rostopic list

You should see

/rosout
/rosout_agg
/talker/topic_a
In the same terminal, run:

```bash
$ rosparam list
```

You should see the list of parameters, including `/talker/pub_timestep`.

You can see the parameters and the values of the `talker` node with

```bash
$ rosparam get /talker
```

### 6.6. Adding a command line parameter

You can register a parameter in the launch file such that it is added to the ROS parameter dictionary. This allows you to call `rospy.get_param()` on your parameter from `talker.py`.

Edit your launch file to look like this:

```xml
<launch>
  <arg name="pub_timestep" default="0.5" />
  <node name="talker" pkg="pkg_name" type="talker.py" output="screen">
    <param name="~pub_timestep" value="$(arg pub_timestep)"/>
    <remap from="~topic_b" to="~topic_a"/>
  </node>
</launch>
```

Previously, you should have had the line `<param name="~pub_timestep" value="0.5" />` inside of the node tags. This sets a parameter of value 0.5 to be called `/talker/pub_timestep`. (Remember that the tilde prefixes the variable with the current namespace). By adding the line `<arg name="pub_timestep" default="1" />`, we are telling the program to look for a parameter on the command line called `pub_timestep`, and that if it doesn't find one, to use the value one. Then, `value=$(arg pub_timestep)` retrieves the value set in the previous line.

Within `talker.py`, we can get the value of the inputted parameter. You should already have the line:

```python
self.pub_timestep = self.setupParameter("~pub_timestep", 1.0)
```

This calls the talker's `setupParameter` method, which contains the line:

```python
value = rospy.get_param(param_name,default_value)
```

Where `~pub_timestep` is passed in as `param_name` and 1.0 is passed in as the default value. Now that we have edited the launch file to accept a command line argument, `value` should be the value which is given on the command line, rather than 0.5.

You can test that this works by calling `roslaunch` with the added parameter:

```
roslaunch MINIMAL ROS NODE - pkg_name ...
```
This should cause the time between messages to become three seconds.
The functions `rosparam list` and `rosparam info [param]` are useful in debugging issues with registering a parameter.

### 6.7. Documentation

You should document the parameters and the publish/subscribe topic names of each node in your package. The user should not have to look at the source code to figure out how to use the nodes.

### 6.8. Guidelines

- Make sure to put all topics (publish or subscribe) and parameters under the name-space of the node with ~. This makes sure that the IO of the node is crystal clear.
- Always include the name of the node in the printouts.
- Always provide a launch file that includes all the parameters (using `<param>`) and topics (using `<remap>`) with each node.
Here are just a few useful commands that can help with debugging

7.1. Nodes, Topics, Services

See all topic names:

```
$ rostopic list
```

Print topic messages to terminal:

```
$ rostopic echo topic name
```

See frequency of messages published on topic:

```
$ rostopic hz topic name
```

See which node is publishing and subscribing to a topic:

```
$ rostopic info topic name
```

7.2. rqt

rqt is a collection of packages and tools for debugging in ROS

See a graph of all nodes and topics:

```
$ rqt_graph
```

Here is an example graph.

7.3. rviz

rviz is a tool for visualizing data from topics.
On the left panel you can add visualizations based on message type for topic.
UNIT J-8

Naming Conventions

8.1. Nodes

8.2. Topics

8.3. Messages
PART K

ROS development basics - Exercises
UNIT K-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 K-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 **Unit A-3 - Exercise: Simple data analysis from a bag**.
You should see that the statistics have changed.
UNIT K-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` (unknown ref software_devel/duckietown-utils-library)
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 K-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 then 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
- `flip-horizontal`: flips the image horizontally
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 \texttt{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()}. 

- \texttt{grayscale}: Makes the image grayscale
- \texttt{sepia}: make the image sepia
UNIT K-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 `d8compressed_image_from_cv_image()`.
I will ignore this because it is an external link.

> I do not know what is indicated by the link 'software_devel/duckietown_utils-d8_compressed_image_from_cv_image'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

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 K-6

Exercise: Live Instagram

6.1. Skills learned

- Live image processing

6.2. Instructions

You may find useful: (unknown ref software_devel/ros-python-howto)

<table>
<thead>
<tr>
<th>previous warning (7 of 7) index</th>
</tr>
</thead>
<tbody>
<tr>
<td>warning</td>
</tr>
</tbody>
</table>

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#software_devel/ros-python-howto'.

Location not known more precisely.

Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

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 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 K-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:

\[
\text{image} \rightarrow [\text{feature extraction}] \rightarrow 2D \text{ features} \rightarrow [\text{ground projection}] \rightarrow 3D \text{ world coordinates}
\]

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:

\[
3D \text{ world coordinates} \rightarrow [\text{image projection}] \rightarrow 2D \text{ features} \rightarrow [\text{rendering}] \rightarrow \text{image}
\]

7.3. Instructions

- Do intrinsics/extrinsics camera calibration of your robot as per the instructions.
- Write the ROS node specified below in Section 9.4 - Specification of dt_augmented_reality.

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 \texttt{dt\_augmented\_reality} with map file \texttt{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 \texttt{dt\_augmented\_reality} with map file \texttt{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 Section 9.5 - Specification of the 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.

```
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:

```
[rgb, [R, G, B]]
```

where the RGB values are between 0 and 1.

Moreover, we support the following strings:

- **red** is equivalent to [rgb, [1,0,0]]
- **green** is equivalent to [rgb, [0,1,0]]
- **blue** is equivalent to [rgb, [0,0,1]]
- **yellow** is equivalent to [rgb, [1,1,0]]
- **magenta** is equivalent to [rgb, [1,0,1]]
- **cyan** is equivalent to [rgb, [0,1,1]]
- **white** is equivalent to [rgb, [1,1,1]]
- **black** is equivalent to [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:

```yaml
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:
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)
```

(Notice 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`.