DE3-ROB1-CHESS Documentation

Release 0.0.1

Anna Bernbaum, Ben Greenberg, Josephine Latreille, Sanish Mist

Oct 13, 2018

Working with FRANKA Emika

CHAPTER 1

About

This is the documentation for the group CHESS Project for the Robotics 1 module in Design Engineering, Imperial College London, in March 2018.

The project is hosted on GitHub: https://github.com/nebbles/DE3-ROB1-CHESS.

1.1 The Authors

- Anna Bernbaum
- Benedict Greenberg
- Josephine Latreille
- Sanish Mistry
- Leah Pattison
- Paolo Ruegg
- Sylvia Zhang

For enquiries on this documentation or source code, please contact benedict.greenberg15@imperial.ac.uk

For help with our perception code, please contact either leah.pattison15@imperial.ac.uk or paolo.ruegg15@imperial.ac.uk

CHAPTER 2

Summary

The goal of this project was to create a fully automated chess playing robot by applying code to a FRANKA Panda arm. The project was written in Python and ROS was used to interface with FRANKA.

Open source code is used for the chess A.I., giving the robot the ability to calculate the best next move. Open Computer Vision libraries is used to process images of the board. This allows the robot to understand what the opponent's last move was. A virtual chess clock allows the user to finish their move, alerting FRANKA to calculate the next move. Custom built motion planning is used to create a path for FRANKA to follow and control the grippers. A smooth trapeziem velocity profile was applied, making the motion controlled, natural, and reliable. A manual calibration process was used to send accurate and precise coordinates to FRANKA.

Future improvements to the project could include a final implementation of automated calibration as well as a smoother, more continous trajectory. Additionally, the final implementation of the chess clock is yet to be completed. Overall, the main aims of this project were successfully achieved. A game of chess could be played between human and robot.

CHAPTER 3

Popular Links

- Using Python to control Franka (without ROS).
- Using Python to control Franka with ROS topics.
- Converting points between reference frames.

CHAPTER 4

Using Test Scripts

Throughout our project we used test scripts. These can be seen in the `tests` folder. To run these tests properly (e.g. `test_camera.py`) you should type the following into the terminal:

```
cd DE3-ROB1-CHESS/
python -m tests.test_camera.py
```

This is to ensure relative imports work properly, as every import is relative to the project level directory.

CHAPTER 5

Contents

5.1 Franka Arm

5.1.1 Starting up

The Arm should be connected to the workshop floor controller with the thick black cable and the controller box should be powered up. When booting, the arm LEDs will flash yellow. Once the arm is booted (solid yellow when completed) you can release the brakes via the web interface.

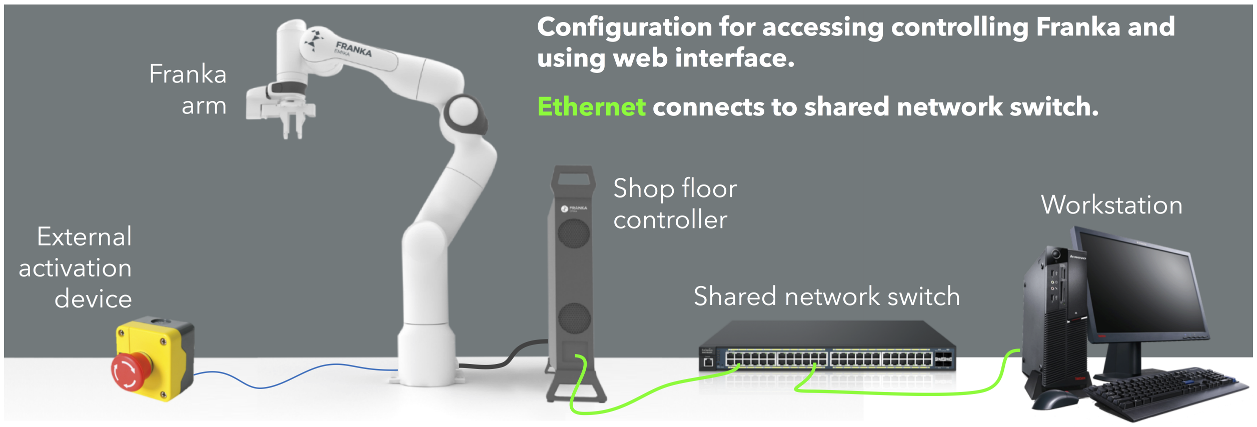

Configuration for accessing controlling Franka and using web interface.
Ethernet connects to shared network switch.
Franka arm
External activation device
Shop floor controller
Workstation
Shared network switch

Fig. 1: Wiring configuration for using the Franka.

Log into the controller web interface (http://192.168.0.88) with:

- Username: `robin`
- Password: `panda`

Note: Your browser may say the connection is not secure, or some other warning. If this is the case, click on the 'advanced' button, and then click 'proceed anyway'.

To release the brakes on the web interface:

1. Make sure the external activation device (EAD) is pressed down.
2. On the web interface, bottom right, *open* the brakes.
3. There will be a clicking noise from the arm.
4. Release the EAD button and the LED light should be solid white. This means the Arm is ready to be used.

Attention: The white light means the robot is in movement mode. The external activation device (EAD) / button is a software stop. It 'detaches' the arm from the controller which causes the arm to lock (yellow light on). It is **not** an emergency stop, as it does not cut power to the arm/controller.

With the Arm in movement mode, it can be manually manipulated by squeezing the buttons at the end effector. The buttons have a two stage press, and if you press too hard on the buttons, the arm will lock again. The Arm will automatically go into gravity compensation mode when manually moving it.

Tip: If the end-effector has recently been removed or readded, the gravity compensation may not be performing well. This is because the web interface settings have not been updated to account for the decrease/increase to expected weight. See the section on *Removing or adding the Franka's end-effector (hand)*.

Note: When designing motion controllers, we are recommended to use impedance control, not high gain control. This will mean we can reduce the stiffness of the arm when moving making it safer for collaborative environments.

5.1.2 Networking information

If you now want to use a workstation computer to control the Arm via the FRANKA Control Interface (FCI) libraries, **first ensure you have completed the above steps to unlock the Arm brakes**. Also check the ethernet cable is attached either to a network switch or directly to the shop floor controller.

Attention: According to FRANKA documentation: "the workstation PC which commands your robot using the FCI must always be connected to the LAN port of Control (shop floor network) and **not** to the LAN port of the Arm (robot network)."

With the main workstation computer *should* have a static IPv4 address for the computer in the Ubuntu network settings. The recommended values are seen below:

Device	IP Address	Notes
FRANKA Arm	192.168.1.0	This does not change
Shop floor (controller)	192.168.0.88	This does not change
Workstation (main)	192.168.0.77	Should be static (in settings)

Important: It is important to note that the IP address of the FRANKA Arm and shop floor controller are static and **should not be changed**. Use this table as reference.

You can confirm that the workstation computer is able to communicate with the workshop controller by pinging the IP address from the terminal:

```
$ ping 192.168.0.88
```

Note: Communicating with the Franka over the switch with a static IP does allow you to have internet access, just note that the gateway and DNS settings should be provided in the Ubuntu settings accordingly to make it work.

5.1.3 Removing or adding the Franka's end-effector (hand)

To remove or add the hand, first shutdown the arm completely. Secure/remove the hand using both screws and the attach/detach the interface cable.

Once the robot has restarted, go to the **Settings** in the web interface, go to **End effector** and set both the hand drop-down menu and toggle the gripper.

5.1.4 Franka Emika Software

Software updates can be found at: http://support.franka.de/

The software versions currently used in the robotics lab are:

Software	Version	Notes
Franka Firmware	1.0.9	Supports `libfranka < 0.2.0`
ros-kinetic-libfranka	0.1.0	Current ROS version is 0.2.0
franka_ros	??	Currently unused

Warning: The lab only supports libfranka 0.1.0 which is currently unavailable from `apt install`. **Do NOT uninstall ROS or libfranka on workstations which already have it installed**.

5.1.5 Shutting down the Arm

Enter the web interface for the Arm. In the lower right menu, lock the brakes. Then in the top right menu, select shutdown, and confirm.

Important: Remember to shutdown the controller from the web interface. This device is a computer, and should not be switched off from mains.

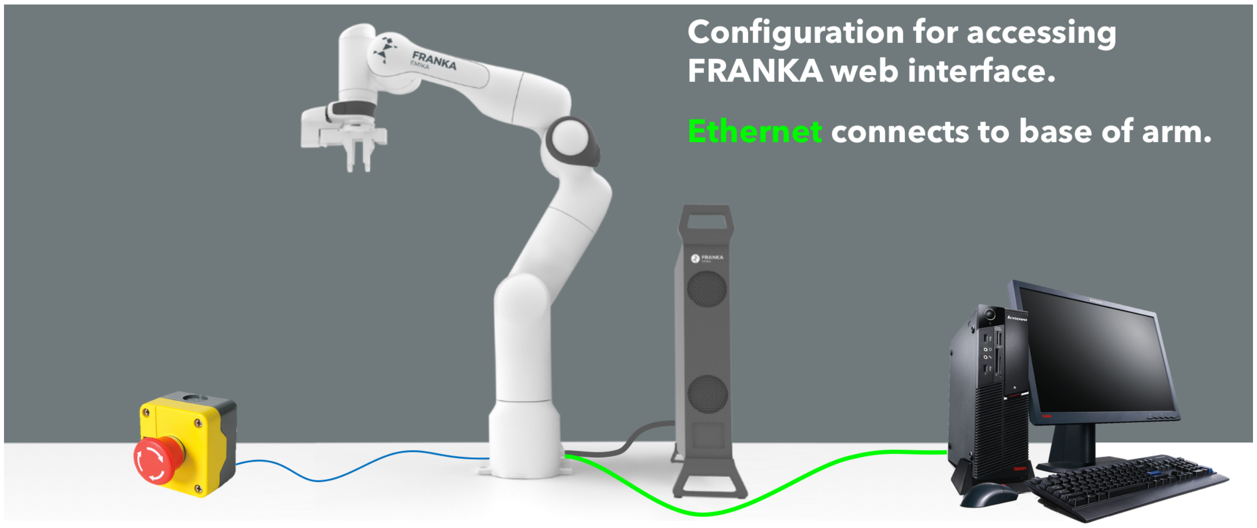

Configuration for accessing
FRANKA web interface.
Ethernet connects to base of arm.

Fig. 2: Wiring configuration fo accessing the Arm directly (through the web interface).

5.1.6 Appendix

In rare cases, you may need to access the Franka arm directly by connecting the ethernet cable as seen in the image below:

Log into the controller web interface (http://robot.franka.de) with:

- Username: `robin`
- Password: `panda`

5.2 Workstation set-up for Franka

Generally the set up procedure follows the guide provided on the FRANKA documentation website.

5.2.1 Requirements for FRANKA Workstation

1. *Install Ubuntu* (you may need to partition hard drive)
2. *Install the realtime kernel patch*
3. *Install ROS Kinetic*
4. *Install FRANKA Libraries*
5. *Install other libraries*

5.2.2 Install Ubuntu

On Macintosh

It is recommended you follow the following guide to install Ubuntu alongside your current operating systems on Mac:

https://apple.stackexchange.com/questions/257166/installing-ubuntu-on-mac-with-macos-and-windows-already-installed

On Windows PC

To install Ubuntu alongside Windows, you must first shrink you current partitions down so there is space to create a new partition. To do this, follow this dual boot guide.

If you are unable to shrink a partition with free space available, you need to defrag you partition. Unfortunately this may, or may not work just using Windows' built in application. If it doesn't work, use a third party tool, like the free AOMEI Partition Assistant to shrink your drive with space.

After you have shrunk your drive and you have a suitable amount of space for Linux (50-100GB ideally) you need to check how many partitions you currently have. If you already have 4 partitions, e.g.:

- System (Windows)
- Recovery
- HP_RECOVERY
- HP_TOOLS

Then you should delete the HP_RECOVERY partition using the disk management tool in Windows. This is because Master Boot Record (the partition map that Windows traditionally uses) only allows for a maximum of 4 partitions. (Also see: https://support.hp.com/gb-en/document/c00810279).

Note: You should check if your partition map is MBR (Master Boot Record) or GUID/GPT. If it is GUID/GPT then you do not need to delete any partitions in order to create a new one.

You should download the latest Ubuntu LTS version from the official website, then use a tool like RUFUS to create a bootable USB. Follow this guide to create a bootable USB for Linux: https://www.howtogeek.com/howto/linux/create-a-bootable-ubuntu-usb-flash-drive-the-easy-way/

When you have created your bootable USB, restart your computer and (if on an HP computer) continually tap the F9 key. This will lead you to the boot menu allowing you to select the USB that you just made to boot Linux.

You can now continue with this guide as before to install Linux. Once it is completely installed, you can move to the next section.

5.2.3 Install the realtime kernel patch

https://frankaemika.github.io/docs/installation.html#setting-up-the-real-time-kernel

Check you are running a real-time kernel with `uname -r`.

If you are not,

- Restart the computer
- Select *Advanced options for Ubuntu*
- Select the kernel with `rt` in the name.

5.2.4 Install ROS Kinetic

Follow the official installation guide to get ROS Kinetic at the following URL:

http://wiki.ros.org/kinetic/Installation/Ubuntu

5.2.5 Install FRANKA Libraries

Attention: It is recommended that you **DO NOT** install from source. There are lots of problems with compiling versions which work with the current firmware version of the Franka Arm.

Option 1: Install binaries via `apt`

Warning: Franka Emika have updated the version of libfranka distributed by ROS from 0.1.0 to 0.2.0 which means it no longer supports the firmware version used in the lab. For more information on software versions, see *Franka Emika Software*. Unless the Franka's firmware has been updated, you should now use *Option 2: Install and compile from source*.

Binary packages for `libfranka` and `franka_ros` are available from the ROS repositories. After setting up ROS Kinetic, execute:

```
sudo apt install ros-kinetic-libfranka
sudo apt install ros-kinetic-franka-ros
```

Note that if you use `apt-get` or `apt` to install a package on Ubuntu, you can use `dpkg -L <packagename>` to find where on the system the files for a particular package are installed.

Option 2: Install and compile from source

The following guide is based on a guide from Franka Emika (link).

Before building from source, please uninstall existing installations of `libfranka` and `franka_ros` to avoid conflicts:

```
sudo apt remove "*libfranka*"
```

To build `libfranka`, install the following dependencies from Ubuntu's package manager:

```
sudo apt install build-essential cmake git libpoco-dev libeigen3-dev
```

Then, download the source code by cloning `libfranka` from GitHub:

```
cd ~
git clone --recursive https://github.com/frankaemika/libfranka
cd libfranka
```

By default, this will check out the newest release of `libfranka`. If you want to build a particular version of `libfranka` instead, check out the corresponding Git tag. At the time of writing the firmware of the Franka means we need version `0.1.0` so we do:

```
git checkout tags/0.1.0
git submodule update
```

Tip: Use `git tag -l` will list available tags.

In the source directory, create a build directory and run CMake:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
```

You now need to add the path of this library to you system path so that libfranka can be found at runtime. To do this, when in the `build/` directory:

```
pwd
```

This returns a path such as `/home/<username>/libfranka/build`. We then add this to the **end** of our `~/.bashrc` file, such an example is:

```
nano ~/.bashrc

# add the following line to the end of the file
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/<username>/libfranka/build"
# now save and exit the file

source ~/.bashrc
```

Warning: Remember to put your own path to the build directory of `libfranka` instead of `/home/<username>/libfranka/build`.

Installing `libfranka` is now complete. If you now need to build the ros packages, you should use the guide found here.

5.2.6 Using the franka_ros library

This is only applicable is you installed `franka_ros` *with* `apt`.

The Franka ROS packages are intiated using the launch xml files. To do this you need to adjust the default IP address in these launch files:

```
roscd franka_visualization
cd launch/
nano franka_visualization.launch
```

Now change the value for `name=robot_ip` from `default=robot.franka.de` to `default=192.168.0.88`. You can then launch the package with:

```
roslaunch franka_visualization franka_visualization.launch
```

You can swap out the 'visualization' term for any other franka_ros package.

5.2.7 Setting Permissions

You need to ensure you have set the correct permissions for libfranka. Run:

```
source /opt/ros/kinetic/setup.bash
```

Hint: If this doesn't run, you may not have installed ROS Kinetic properly. Check ROS Kinetic install here.

We then need to source ros for the root user so that libfranka has permissions to use the realtime kernel:

```
sudo bash
source /opt/ros/kinetic/setup.bash
```

You now press **Ctrl+D** to exit out of sudo bash (the hash sign will change back to a dollar sign). You must also check that you have the correct realtime permissions for your own user. To do this, run:

```
ulimit -r
```

If the results is `99` then you have nothing more to do, is the result is `0` then go back and check you completed the last section of the realtime kernel setup.

Tip: Remember you can check if you are running your realtime kernel at any time by typing `uname -r` and looking for an `rt` after the kernal version.

5.2.8 Test with examples

To start testing you should move to *Controlling Franka & ROS* page to test out the workstation set up.

Remember, for this to work, you need:

- The FRANKA Arm must be in movement mode (white light).
- The workstation PC must be connected to the shop floor controller by ethernet.

Tip: You can confirm that the workstation computer is able to communicate with the workshop controller by pinging the IP address from the terminal: `ping 192.168.0.88`. For more information see the *Franka Arm* page.

5.2.9 Install other libraries

You may want to install OMPL:

http://ompl.kavrakilab.org/installation.html

Warning: The installation of OMPL takes several hours.

5.3 Controlling Franka & ROS

5.3.1 Starting ROS to control Franka

The recommended setup for controlling the franka can be seen in the tree below. The top level shows the devices required and their IP address, the second tier shows the commands needed to run the files:

```
.
├── Franka Control Box (192.168.0.88)
├── Host workstation computer (192.168.0.77)
│   ├── roscore
│   └── ./franka_controller_sub 192.168.0.88
└── Project workstation computer
    ├── roslaunch openni2_launch openni2.launch
    ├── camera_subscriber.py (this does not need to be run seperately)
    └── main.py
```

Tip: To test the image feed out without using it in `main.py` you can use the `test_camera.py` file in the `tests` folder.

Networking with other workstations

Instead of running the master node and subscriber nodes on your own workstation, these can be running on the main workstation in the lab instead. This means that libfranka won't need to be installed on your specific workstation.

To communicate over the lab network you need to change two main ROS variables. Firstly you need to find the IP address of your computer when connected to the lab network (via ethernet). To do this you can use `ifconfig` in a terminal window to give you your `<ip_address_of_pc>`.

You then need to run the following two commands in your terminal window (substitute in you IP address):

```
export ROS_MASTER_URI=http://192.168.0.77:11311
export ROS_IP=<ip_address_of_pc>
```

As you will see, this is connecting you to the static IP address of the main Franka workstation, `192.168.0.77`. In order for you to continue with running a Python publisher, you need to ensure that roscore and the subscriber is running on the main workstation.

Note: That this configuration of assigning IP addresses to ROS_MASTER_URI and ROS_IP is non-permanent, and is only active for the terminal window you are working in. This has to be repeated for every window you use to run rospy. Alternatively you can add these commands to your bashrc.

5.3.2 Running the subscriber

Once `roscore` is running, the subsciber has to run in the background to read the messages from our Python publisher and execute them with libfranka. To run the subscriber (from the project folder), run:

```
cd franka/
./franka_controller_sub 192.168.0.88
```

Sometimes there is an "**error with no active exception**" thrown by this executable. This can sometimes be solved by simply manually moving the arm using the buttons a bit. Then rerun the command above again.

Warning: This subscriber is compiled for `libfranka 0.1.0`. You can check your current `libfranka` version with `rosversion libfranka` command.

5.3.3 Using the publisher

First, make sure you are running `roscore` and the subscriber, `franka_controller_sub` on the main lab workstation.

Example

Assuming you are writing a script (`script.py`) that wants to use control franka, the files should be stored as:

```
.
├── README.md
├── franka
│   ├── __init__.py
│   ├── franka_control_ros.py
│   ├── franka_controller_sub
│   ├── franka_controller_sub.cpp
│   ├── print_joint_positions
│   └── print_joint_positions.cpp
└── script.py
```

To use the `FrankaRos` class in your own Python script would look something like this:

```python
from franka.franka_control_ros import FrankaRos

franka = FrankaRos(debug_flag=True)
# we set the flag true to get prints to the console about what FrankaRos is doing

while True:
    data = arm.get_position()
    print("End effector position:")
    print("X: ", data[0])
    print("Y: ", data[1])
    print("Z: ", data[2])
```

class `franka.franka_control_ros.`**`FrankaRos`**(*log=False*, *ip='192.168.0.88'*, *debug=False*, *init_ros_node=False*)

`get_position`()

Get x, y, z position of end-effector and returns it to caller.

Returns list as [x, y, z]

`grasp`(*object_width*, *speed*, *force*)

Grasp an object in the grippers. Note that this can only be called if the object width is smaller than the current distance between the grippers.

The grippers attempt to move to the width of the object defined with speed defined, and then proceed to apply a defined force against the object. If no object is present it evaluates as failed and moves to next task.

0 width defined == 2.2 cm difference real-world

Parameters

- **`object_width`** – width of target object to grab (float)
- **`speed`** – with which to move the gripper to object width (float)
- **`force`** – to apply to the object once the grippers are at object width (float)

`move_gripper`(*width*, *speed*)

Set gripper width by assigning `width` with a desired speed to move at. Note this must be called before grasping if gripper fingers are too close together `grasp()` call.

0 width defined == 2.2 cm difference real-world

Parameters

- **`width`** – desired distance between prongs on end-effector (in millimetres)
- **`speed`** – desired speed with which to move the grippers

`move_relative`(*dx*, *dy*, *dz*, *speed*)
Moves robot end effector in desired direction (in robot reference frame) given a target displacement and speed to travel at. The robot will not necessarily travel at this speed due to safety controls in the subscriber.

Parameters

- **`dx`** – float value displacement in robot reference axis
- **`dy`** – float value displacement in robot reference axis
- **`dz`** – float value displacement in robot reference axis
- **`speed`** – float value desired speed to target displaced position

`move_to`(*x*, *y*, *z*, *speed*)
Moves robot end effector to desired coordinates (in robot reference frame) given a target velocity. The robot will not necessarily travel at this speed due to safety controls in the subscriber.

Parameters

- **`x`** – float value position in robot reference axis
- **`y`** – float value position in robot reference axis
- **`z`** – float value position in robot reference axis
- **`speed`** – float value desired speed to target position

`send_trajectory`(*trajectory*)

`start_subscriber`()

`stop_subscriber`()

`subscribe_position_callback`(*data*)

`franka.franka_control_ros.`**`example_movement`**()
Used to test if the arm can be moved with gripper control.

Function moves the arm through a simple motion plan and then tried moving the grippers. To use this test, add the `-m` or `--motion-example` flag to the command line.

`franka.franka_control_ros.`**`example_position`**()
Used to test if position reporting is working from Arm.

It will repeatedly print the full arm position data and the XYZ position of the end-effector. To use this test, add the `-p` or `--position-example` flag to the command line.

5.3.4 Using Franka without ROS

Note: This method is deprecated. It is now recommended you use ROS to control the Arm using a Python publisher, such as the way described above.

Setting Permissions

To control the Franka Arm, Fabian and Petar have written a small collection of C++ files which can be compiled to run as executables and control the Franka Arm using libfranka.

You need to ensure you have set the correct permissions for libfranka. You can check that in *Using the franka_ros library*.

Downloading the C++ Executables and Python Class

Now that you have libfranka set up properly you can get use the C++ files provided. These resources can be found in the `/franka` directory of the repository. Firstly, go to your project directory in the terminal by using `cd <project_dir>`. If you have already downloaded the files before and are replacing them with an up-to-date version, run `rm -rf franka/` first. To download the necessary folder, run:

```
svn export https://github.com/nebbles/DE3-ROB1-CHESS/trunk/franka
```

Once this directory is downloaded into your project directory, you need to change directory and then make the binaries executable:

```
cd franka/
chmod a+x franka*
chmod a-x *.cpp
```

Warning: This next command will move the FRANKA. **Make sure you have someone in charge of the external activation device (push button)**.

These binaries can now be used from the command line to control the Arm:

```
./franka_move_to_relative <ip_address> <delta_X> <delta_Y> <delta_Z>
```

Alternatively, you can control the Arm using the easy custom Python class `Caller` (see below).

Python-Franka API with `franka_control.py`

The Python-FRANKA module (`franka_control.py`) is designed to allow easy access to the C++ controller programs provided by Petar. The provided Python module is structured as follows.

Example

To use the `FrankaControl` class in your own Python script would look something like this:

```python
from franka.franka_control import FrankaControl

arm = FrankaControl(debug_flag=True)
# we set the flag true to get prints to the console about what Caller is doing

arm.move_relative(dx=0.1, dy=0.0, dz=-0.3)
# we tell teh arm to move down by 30cm and along x away from base by 10cm
```

Note: This example code assumes you are following the general project structure guide. See below for more information. The code above would be called from a main script such as `run.py`.

General Structure of Project

The structure of the project is important to ensure that importing between modules works properly and also seperates externally maintained code from your own. An example of a project tree is:

```
.
├── LICENSE.txt
├── README.md
├── run.py
├── __init__.py
├── docs
│   ├── Makefile
│   ├── build
│   ├── make.bat
│   └── source
├── franka
│   ├── __init__.py
│   ├── franka_control.py
│   ├── franka_get_current_position
│   ├── franka_get_current_position.cpp
│   ├── franka_move_to_absolute
│   ├── franka_move_to_absolute.cpp
│   ├── franka_move_to_relative
│   └── franka_move_to_relative.cpp
├── my_modules
│   ├── module1.py
│   ├── module2.py
│   ├── module3.py
│   ├── __init__.py
└── test_script.py
```

5.3.5 Additional Resources

https://frankaemika.github.io/docs/getting_started.html#operating-the-robot

5.4 Camera

5.4.1 Description

In this project, an RGB-D camera was used to detect FRANKA and the chessboard. This part of the project allows us to fetch a single image from the camera livestream whenever it is needed. It is utilises OpenCV and is interfaced with ROS. RGB and depth information collected from the image frame is used in both automatic calibration procedures and within the Perception module for board and game state recognition.

5.4.2 Design

The external RGB-D camera is connected through USB. In order to use OpenNI-compliant devices in ROS and launch the camera drive we need to launch the rospackage for the device:

```
roslaunch openni2_launch openni2.launch
```

Warning: You must run this on the client machine. If you run the camera rospackage on the host computer running the FRANKA control loop it will cause the controller to crash.

The camera subscriber was initially written to use Python processes. This gave us a significant boost in speed in our main runtime loop however we updated our version to use a time synchronised thread in order to simplify the code. Functionally, the two versions provide the same output. The full version can be viewed below at the header *Camera Subscriber with Processes*.

In the version without processes, a class was created which initialised the subscribers as members:

```
        self.depth_sub = Subscriber("/camera/depth_registered/image_raw", Image)
        self.tss = message_filters.ApproximateTimeSynchronizer([self.image_sub, self.
↪depth_sub],
                                                                queue_size=10, slop=0.
↪5)
        self.tss.registerCallback(self.callback)

        time.sleep(0.5)
```

The callback function of the class was then used in the synchronised thread to update the members of the class storing the latest rgb and depth image:

```
        self.rgb_raw = img
        self.depth_raw = depth
        if self.debug:
            print("New images have been collected.")
```

This meant that during the runtime, when the latest image was needed, it could be fetched from the class (after converting into a CV image):

```
        cv_image = None
        depth_image = None
        while cv_image is None and depth_image is None:
            try:
                cv_image = self.bridge.imgmsg_to_cv2(self.rgb_raw, "bgr8")
            except CvBridgeError as e:
                print(e)

            try:
                depth_image_raw = self.bridge.imgmsg_to_cv2(self.depth_raw,
↪"passthrough")
                # noinspection PyRedundantParentheses
                depth_image = ((255 * depth_image_raw)).astype(np.uint8)
            except CvBridgeError as e:
                print(e)

        if self.debug:
            print("Image has been converted.")

        return cv_image, depth_image
```

You can download this simpler version of the camera subscriber `here`.

5.4.3 Camera Subscriber with Processes

`Download the processes version`

This version consists two classes and process function. The `CameraFeed` class creates a seperate process using the `main()` function. This function starts up the `ImageConverter` class which handles the Subscribers and image conversion into OpenCV format. A queue is created between the main function running in the new process and the original `CameraFeed` class running in the usual process.

Multiprocessing was used to spawn multiple subprocesses for parallel execution of tasks. First Queues are initialised holding RGB and depth images that need to be processed, and set the maxsize to `1`. Therefore, only one image can be held at a time. `get_frames()` method is used to to retrieve the results from `queue` and return to the caller:

```
    def get_frames(self):
        while self.rgb_q.empty() and self.depth_q.empty():
            time.sleep(0.1)

        # we fetch latest image from queue
        rgb_frame = self.rgb_q.get()
        depth_frame = self.depth_q.get()

        return rgb_frame, depth_frame
```

In the background process, the callback function is being called. This is attempting to add the latest image to the queue for the parent process to collect it. When it fetches a new image from the subscriber it will place it in the queue. If the queue is alread full, it overwrites the image. This ensures there is only ever one image in the queue. This occurs for both the RGB image queue and the depth image queue.

```
    def callback(self, img, depth):
        cv_image = None
        depth_image = None
        try:
            cv_image = self.bridge.imgmsg_to_cv2(img, "bgr8")
        except CvBridgeError as e:
            print(e)

        try:
            depth_image_raw = self.bridge.imgmsg_to_cv2(depth, "passthrough")
            # noinspection PyRedundantParentheses
            depth_image = ((255 * depth_image_raw)).astype(np.uint8)
        except CvBridgeError as e:
            print(e)

        if self.debug:
            print("Image has been converted.")

        if not self.rgb_q.empty() and not self.depth_q.empty():
            if self.debug:
                print("Queue has item in it.")
            # noinspection PyBroadException
            try:
                if self.debug:
                    print("Attempting to remove item from queue.")
                self.rgb_q.get(False)
                self.depth_q.get(False)
                if self.debug:
                    print("Successfully removed images from queue.")
            except Exception:
```

(continues on next page)

(continued from previous page)

```
                if self.debug:
                    print("\033[91m" + "Exception Empty was raised. Could not remove␣
↪from queue."
                          + "\033[0m")

        if self.debug:
            print("Queue should be empty now, putting in images.")

        self.rgb_q.put(cv_image)
        self.depth_q.put(depth_image)

        if self.debug:
            print("Images are now in queue.")
```

5.5 Calibration

Calibration allows coordinates to be sent to FRANKA in it's own reference frame for it to move to. By callibrating FRANKA, a conversion can take place between an alternative frame of reference to FRANKA's. Both manual and automated methods of doing this were devised.

To use our reference frame conversion code, use the following command in the terminal when in the directory you want it copied to:

```
svn export https://github.com/nebbles/DE3-ROB1-CHESS/trunk/tools/transform.py
```

5.5.1 Manual Calibration

In the interim before having complete automatic calibration, we used manual calibration to determine the position of the board relative to the FRANKA base frame. The FRANKA end-effector was moved to each of the inner corners as shown below and the positions reported by the FRANKA were then hardcoded.

When the `MotionPlanner` class is instantiated, the calibration is automatically taken from the hardcode values in our `__init__` function. In an automatic version, the hardcoded values would be replaced by coordinates gathered by the calibration procedure.

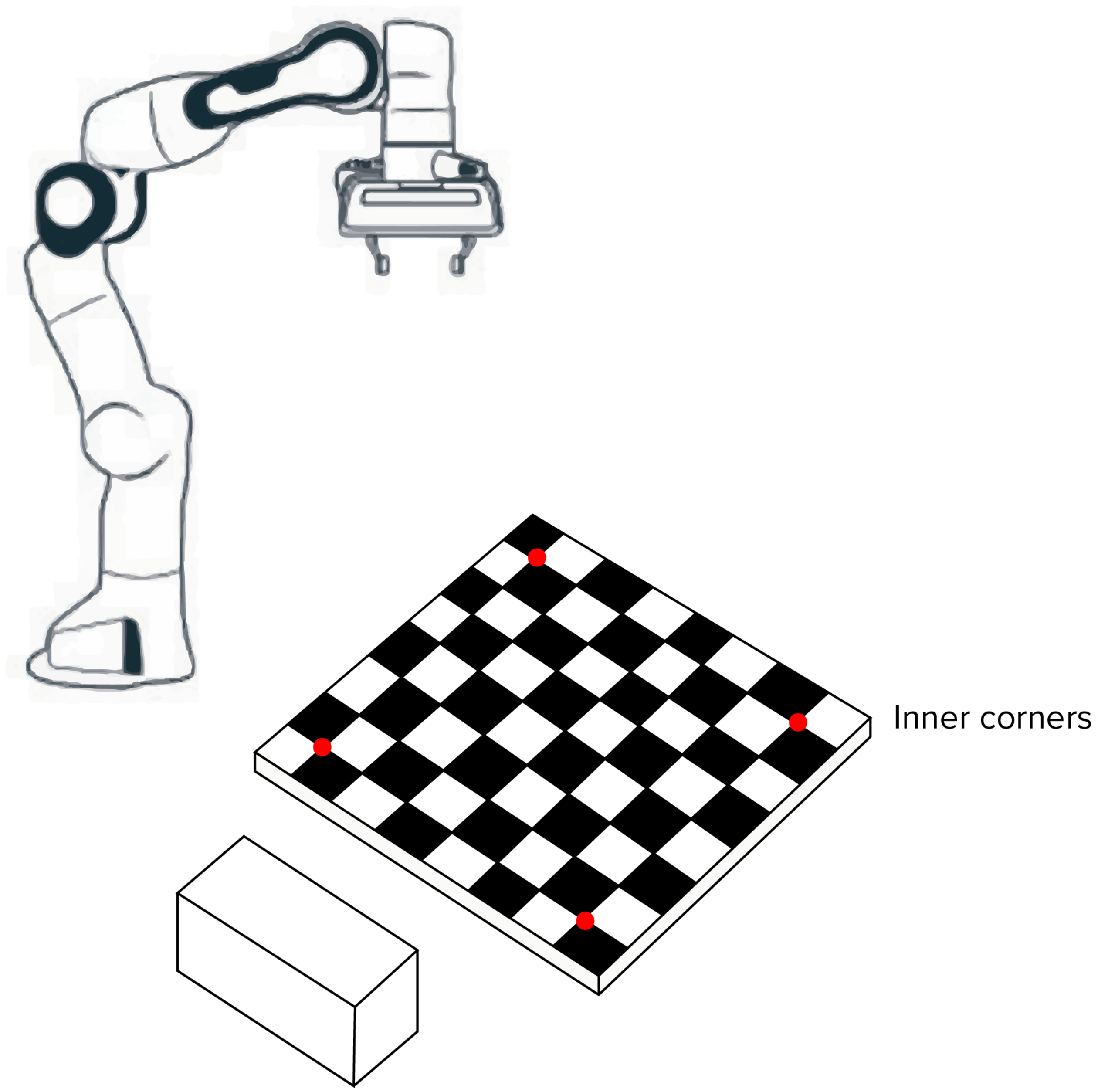
4 vectors are calculated from this information, along the x and y axis of the board:

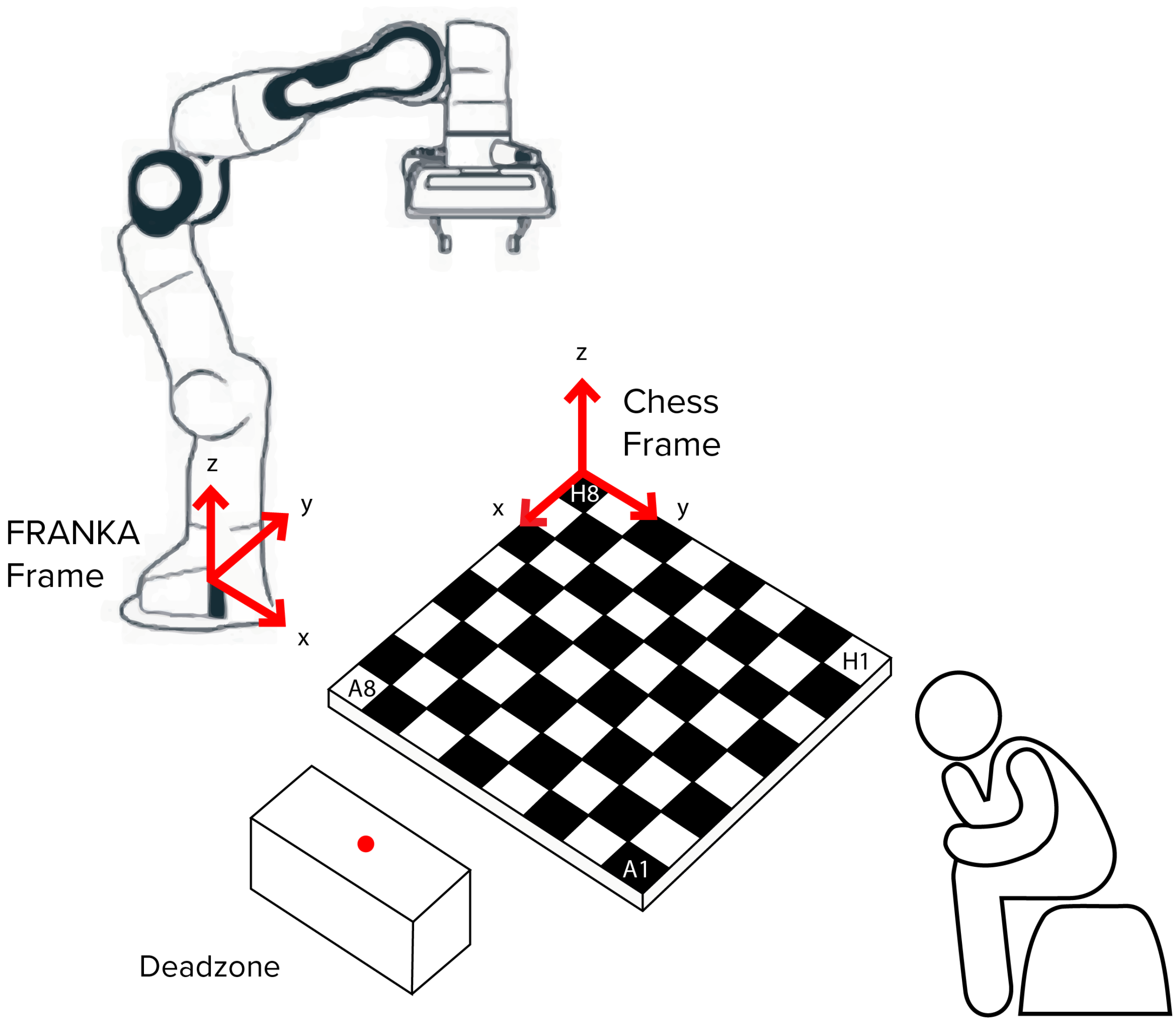
- *x axis 1:* H8 to A8
- *x axis 2:* H1 to A1
- *y axis 1:* H8 to H1
- *y axis 2:* A8 to A1

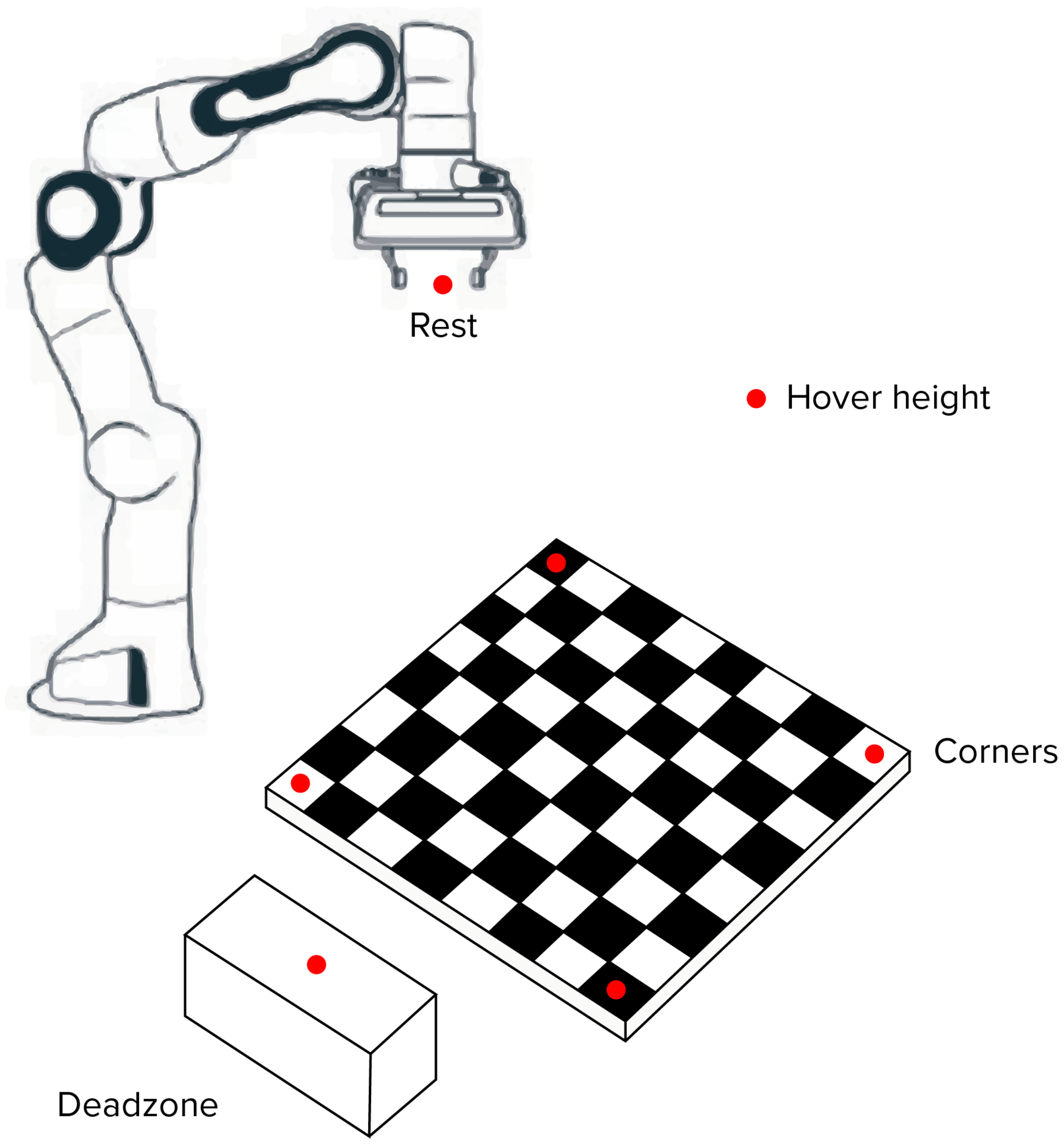
The x and y vectors are averaged respectively to give overall x and y directional vectors. These are then divided by 6 to give x and y unit vetors equivalent to a single board square. The H8 coordinate is stored and used as the starting point for all future location calculations.

The minimum z value of the board is hardcoded when manually moving the end-effector around. This value is then used to hardcode a hover height, rest position and deadzone location as global variables.

Derived positions:


Inner corners


z
Chess
Frame
x
H8
y
z
y
FRANKA
Frame
x
A8
H1
A1
Deadzone


Rest
Hover height
Corners
Deadzone

The coordinates of each square on the board are stored using 2 dictionaries, one for the numbers and one for the letters in algebraic notation. These locations are found by multiplying the unit vectors appropriately. For example, `f` is found by adding 2.5 times `x_unit_vector` to the H8 coordinate.

```
        self.letter_dict = dict([('h', [vector * 0.5 for vector in self.x_unit_
↪vector]),
                                 ('g', [vector * 1.5 for vector in self.x_unit_
↪vector]),
                                 ('f', [vector * 2.5 for vector in self.x_unit_
↪vector]),
                                 ('e', [vector * 3.5 for vector in self.x_unit_
↪vector]),
                                 ('d', [vector * 4.5 for vector in self.x_unit_
↪vector]),
                                 ('c', [vector * 5.5 for vector in self.x_unit_
↪vector]),
                                 ('b', [vector * 6.5 for vector in self.x_unit_
↪vector]),
                                 ('a', [vector * 7.5 for vector in self.x_unit_
↪vector])
                                 ])

        self.number_dict = dict([('8', [vector * 0.5 for vector in self.y_unit_
↪vector]),
                                 ('7', [vector * 1.5 for vector in self.y_unit_
↪vector]),
                                 ('6', [vector * 2.5 for vector in self.y_unit_
↪vector]),
                                 ('5', [vector * 3.5 for vector in self.y_unit_
↪vector]),
                                 ('4', [vector * 4.5 for vector in self.y_unit_
↪vector]),
                                 ('3', [vector * 5.5 for vector in self.y_unit_
↪vector]),
                                 ('2', [vector * 6.5 for vector in self.y_unit_
↪vector]),
                                 ('1', [vector * 7.5 for vector in self.y_unit_
↪vector])
                                 ])
```

Hardcoded width values were used for the gripping width, whilst the `board_z_max` was used to derive the correct height at which to pick up each piece.

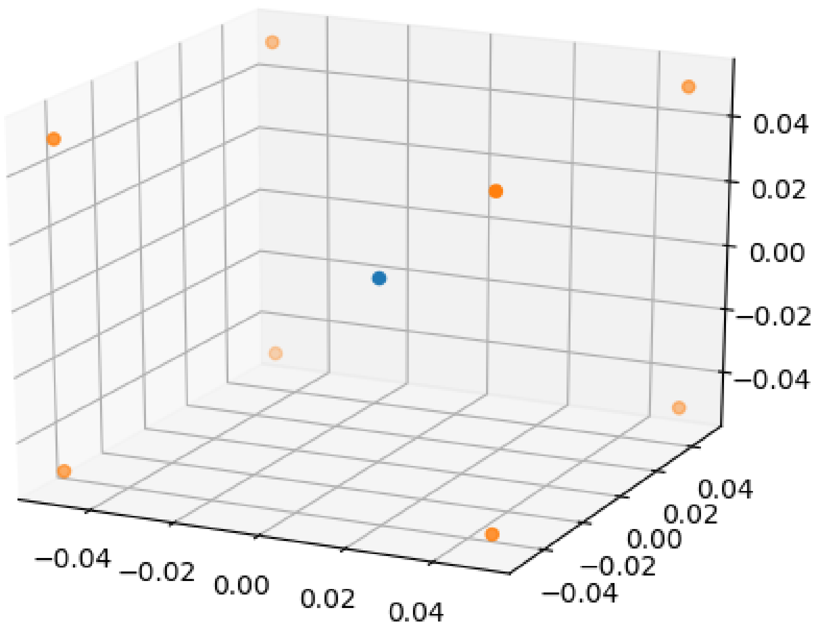
5.5.2 Automated Calibration

To advance beyond the manual hardcoded approach described above, a system was devised whereby the FRANKA end-effector could be placed in the centre of the field of view of the camera, allowing it to automatically calibrate itself from predetermined controlled motions. To calibrate the chess board, image recognition techniques described in the Perception module were used to relate the Franka base frame to the camera. A trackable marker located on the robot end-effector allowed the camera to locate the arm in space as it moved between 8 vertices of a cube (built around the starting position of the end-effector).

The `generate_cube` function generates an array of 9 `x, y, z` coordinates for calibration based on an end-effector (1 x 3 array) input position followed by the 8 coordinates of the each vertex of the cube.

Generating a cube

The generate_cube function generates an array of 9 `x, y, z` coordinates for calibration based on an end-effector (1 x 3) input position followed by the 8 coordinates of the each vertex of the cube. An array is created about `(0, 0, 0)` of edge length 0.1 metres.

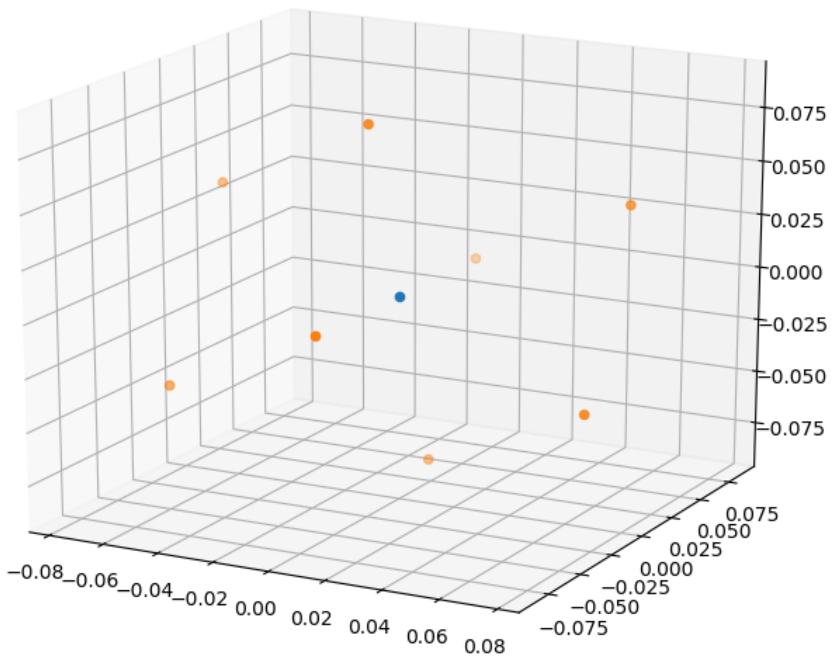


This array is then multiplied by 3 transformations matrices rotating it by 30 degrees in the x, y and z axes producing an 8 x 3 array of the transformed cube about `(0, 0, 0)`. When creating a transformation matrix between two frames, all of the points in the robotic frame need to be different in all three axes. If there are duplicates (say 4 points are planar on x-y) then we start accumulating errors in the transformation matrix. See *Converting between Reference Frames* later for more information on the transformation matrix.

```
    p_cube = np.array(
        [[-0.05, -0.05, -0.05], [-0.05, 0.05, -0.05], [-0.05, 0.05, 0.05], [-0.05, -0.
↪05, 0.05],
         [0.05, -0.05, -0.05], [0.05, 0.05, -0.05], [0.05, -0.05, 0.05], [0.05, 0.05,␣
↪0.05]])
```

This array is then multiplied by 3 transformations matrices rotating it by 30 degrees in the x, y and z axes producing an 8x3 array of the transformed cube about `(0, 0, 0)`. This is because when creating a transformation matrix between two frames we need each of our points in the robotic frame to be different in all three axes. If there are duplicates (say 4 points are planar on x-y) then we start accumulating errors in our transformation matrix. See *Converting between Reference Frames* later for more information on the transformation matrix.

A `for` loop was used to add each row of the array to the x, y, z end-effector positions offseting the transformed cube

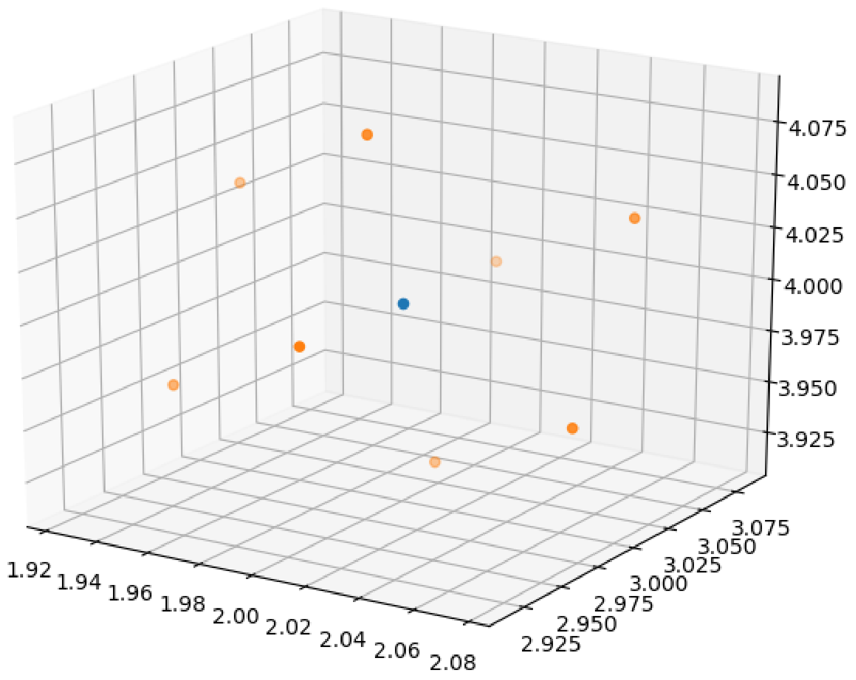

0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.08
−0.06
−0.04
−0.02
0.00
0.02
0.04
0.06
0.08
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075

by the current FRANKA end effector position. These are appended to the `cube_output` array.

```
    # Transformation matrices
    theta = np.radians(30)
    cos = np.cos(theta)
    sin = np.sin(theta)
    rot_x = np.array([[1, 0, 0], [0, cos, -sin], [0, sin, cos]])
    rot_y = np.array([[cos, 0, sin], [0, 1, 0], [-sin, 0, cos]])
    rot_z = np.array([[cos, -sin, 0], [sin, cos, 0], [0, 0, 1]])

    # Multiplying Initial Cube by Transformation matrices in X, Y & Z
    p_cube_rot_x = p_cube.dot(rot_x)
    p_cube_rot_xy = p_cube_rot_x.dot(rot_y)
    p_cube_rot_xyz = p_cube_rot_xy.dot(rot_z)
```

A for loop is used to add each row of the array to the x, y, z end-effector positions to offset the transformed cube by the current FRANKA end effector position. These are appended to the `cube_output` array.



Various markers including LEDs and ARUCO markers were trailled first. These were superceded once it was identified that a simple red cross marker was effective for the camera to detect. This was due to the good colour contrast between the marker and the rest of the frame, allowing a shape to be distinguished. OpenCV was used to detect the shape and colour of the marker. The `detect` function was used to find the area, perimeter and the number of sides of all the polygons found in a frame.

```
    for i in range(len(p_cube)):
        x_tr = x + p_cube_rot_xyz[i, 0]
        y_tr = y + p_cube_rot_xyz[i, 1]
        z_tr = z + p_cube_rot_xyz[i, 2]
        cube_output[i+1, 0] = x_tr
        cube_output[i+1, 1] = y_tr
        cube_output[i+1, 2] = z_tr
```

Detecting the Marker

Various markers including LEDs and ARUCO markers were tried first. These were superceded once it was identified a simple red cross marker was effective for the camera to detect. This was because the colour often had good contrast with the rest of the frame and therefore a shape could be easily distinguished. OpenCV was used to detect the shape and colour of the marker. The detect function was used to detect the area, perimeter and the number of sides of all the polygons found in a frame. To prevent detecting all polygons in the frame, the frame is resized and filtered for the red colour desired. The frame was converted from RGB to HSV to aid extracting objects of specific colour; this was done using a mask over a specific range of HSV values.

The detect function was used in the `find_cross_manual` and `find_cross_auto` functions. Parameters based on the number of sides, the total area and perimeter of the detect polygons were used to filter out other shapes and ensure that the red cross marker was detected.

```
def detect(c):
    """
    Used by find_cross function to detect the edges and vertices of possible polygons␣
↪in an image.

    :param c:
    :return:
    """
    peri = cv2.arcLength(c, True)
    area = cv2.contourArea(c)
    approx = cv2.approxPolyDP(c, 0.02 * peri, True)
    return len(approx), peri, area
```

Image moments were used to find the `x, y` coordinates of the centre of the shape, which are returned. To prevent detecting all polygons in the frame, the frame is resized and filtered for the red colour we desire. The frame was converted from RGB to HSV to aid extracting objects of specific colour; this was done using a mask over a specific range of HSV values.

```
        img = frame
        img_sized = imutils.resize(img, width=640)
        img_hsv = cv2.cvtColor(img_sized, cv2.COLOR_BGR2HSV)

        # cv2.imshow('frameb', imgsized)

        lower_red1 = np.array([170, 70, 50])
        upper_red1 = np.array([180, 255, 255])
        lower_red2 = np.array([0, 70, 50])
        upper_red2 = np.array([10, 200, 200])

        mask1 = cv2.inRange(img_hsv, lower_red1, upper_red1)
        mask2 = cv2.inRange(img_hsv, lower_red2, upper_red2)
        mask = mask1 | mask2
```

(continues on next page)

(continued from previous page)

```
        # cv2.imshow('mask', mask)

        contours = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_
↪SIMPLE)
        contours = contours[0] if imutils.is_cv2() else contours[1]
```

The detect function was used in the find_cross_manual and find_cross_auto functions. Parameters based on the number of sides, the total area and perimeter of the detect polygons were used to filter out other shapes and ensure that the red cross marker was detected.

```
            if 14 > shape[0] > 10 and 800 > shape[1] > 50 and 15000 > shape[2] > 100:
```

Image moments are used to find the x, y coordinates of the centre of the shape, which are returned.

```
                m = cv2.moments(c)
                c_x = int((m["m10"] / m["m00"]))
                c_y = int((m["m01"] / m["m00"]))
```

Automatic vs Manual Marker Detection

The manual detection method allows users to double check that if one marker is detected that the correct one has been selected, and if multiple detectors are selected they are able to enter the number of the correct marker. In the automatic mode the detected marker is automatically returned, however if multiple are detected they are able to switch to manual mode.

Find Depth

The `find_depth()` function returns the RGB value of a pixel at coordinates x, y on a frame. Only the first value in the list is required since this function is used on a greyscale depth frame. The `run_calibration()` function runs the calibration functions moving the end-effector between the 8 vertices of the cube (x, y, z), then detecting the marker at each of these 8 steps, applying the marker offset and producing the array of u, v, w positions (coordinates in the camera frame).

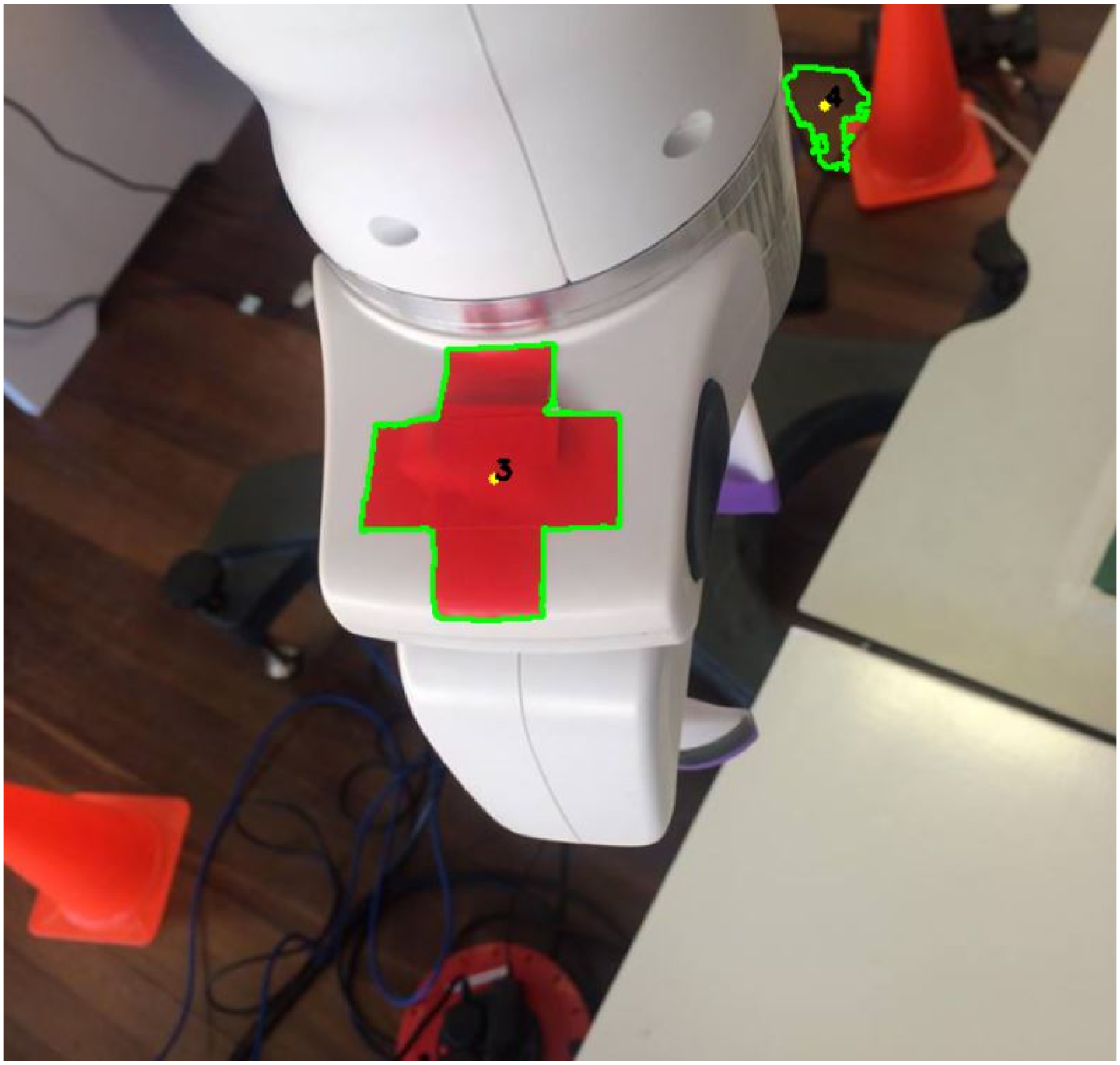
```
def find_depth(img, coordinate):
    """
    Finds depth of marker centre in a depth image.

    :param img:
    :param coordinate:
    :return:
    """
    # img = cv2.imread('test_3.jpg') #might not be needed if already reading a cv␣
↪image
    depth = img[(coordinate[1]), (coordinate[0])]
    coordinates = coordinate[0], coordinate[1], depth[0]

    return coordinates
```

Marker Offset

A hardcoded offset value is measured to adjust the perception u, v, w of the detected marker to the position of the end-effector of the robot. This is since the marker is not placed exactly at the position where FRANKA records its x,


3
4

y, z end effector positon.

Running Calibration

The `run_calibration()` function runs the calibration functions moving the end-effector between the 8 vertices of the cube (x, y, z), then detecting the marker at each of these 8 steps, applying the marker offset and producing the array of u, v, w positions.

Converting between Reference Frames

Overview

To be able to convert camera coordinates, provided by OpenCV tracking tools and other methods, a relationship between multiple reference frames needs to be maintained. The desired relationship relates the camera reference frame and the robot base reference frame used by the libfranka controller. This relationship is stored in a 4-by-4 transformation matrix, and is constructed using the following general formula:

$$aX = b$$

This is modelled on the idea that a coordinate in the main frame (e.g. RGB-D camera provides `u, v, w` coordinates) and converts it to the equivalent, corresponding coordinate in the robots reference frame (e.g. `x, y, z`) so that the robot can move to that point on the camera's view. `a` represents our camera coordinate, and `b` represents the output of our function, that mulitplies `a` with our transformation matrix `X`, which represents the same point but on the robots reference frame.

Creating the transformation matrix

To create the transformation matrix, you construct a set of linear equations that are then solved using a simple least squares algorithm, commonly used in linear regression. This algorithm tries to minimise the sum of squares for each solution to the set of linear equations.

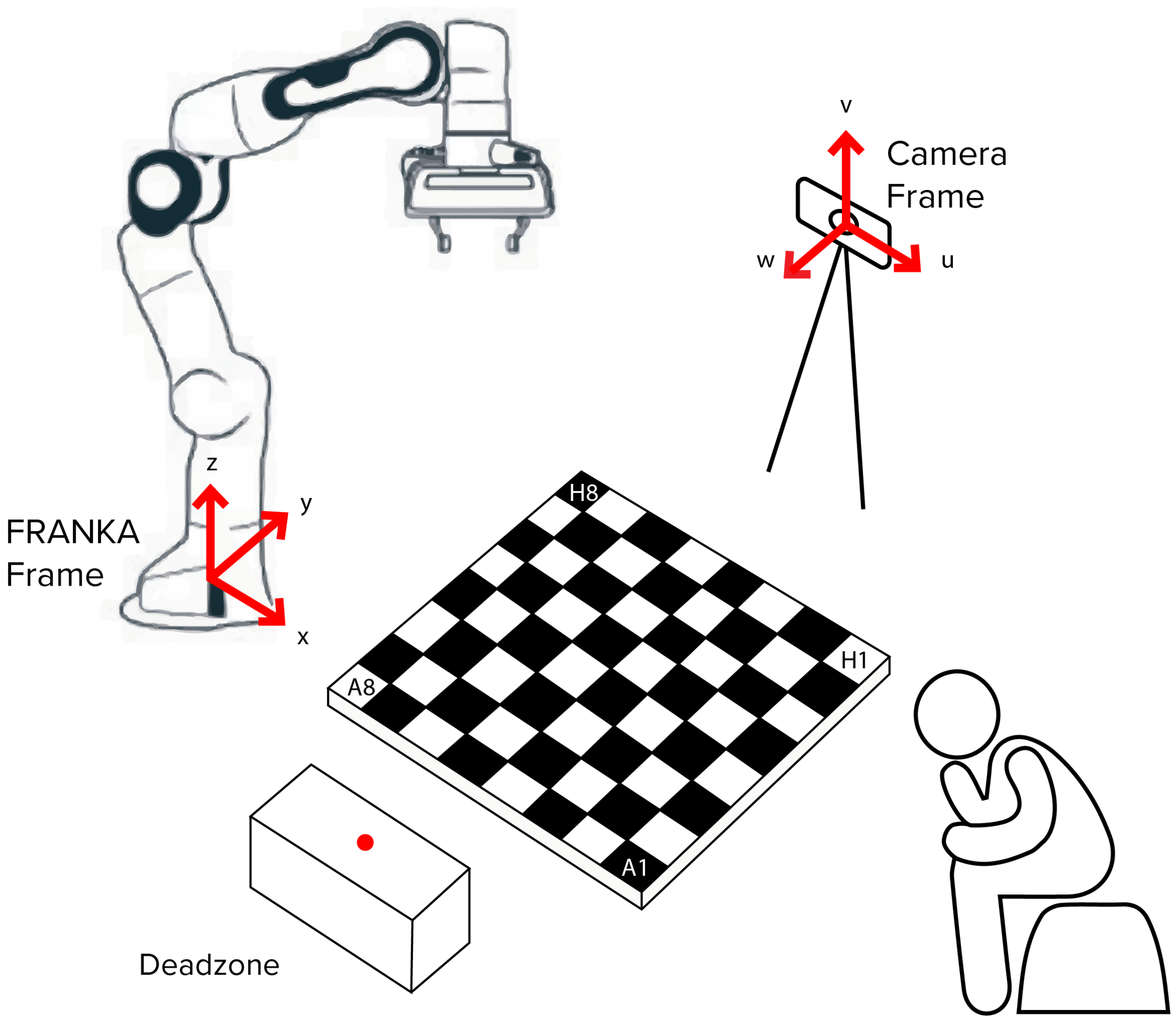
This set of linear equations is constructed using *calibration points*. These points (usually a minimum of 4) are a set of known, corresponding coordinates in both the cameras reference frame and the robots. These can be automatically sourced with a setup program, or manually. To manually get these points, the robots end effector are moved to a point in the field of view of the camera, and the robot would report its position (`x, y, z`). The camera would then detect the robot end effector in the field of view and report the location according to its own reference frame (`u, v, w`) and so these two points are the same point (they correspond) but are in different reference frames. A minimum of 4 calibration points are collected, ideally up to 8 or 10 because this will increase the accuracy of the transformation matrix, as there may have been a small error in the values reported by the camera or robot.

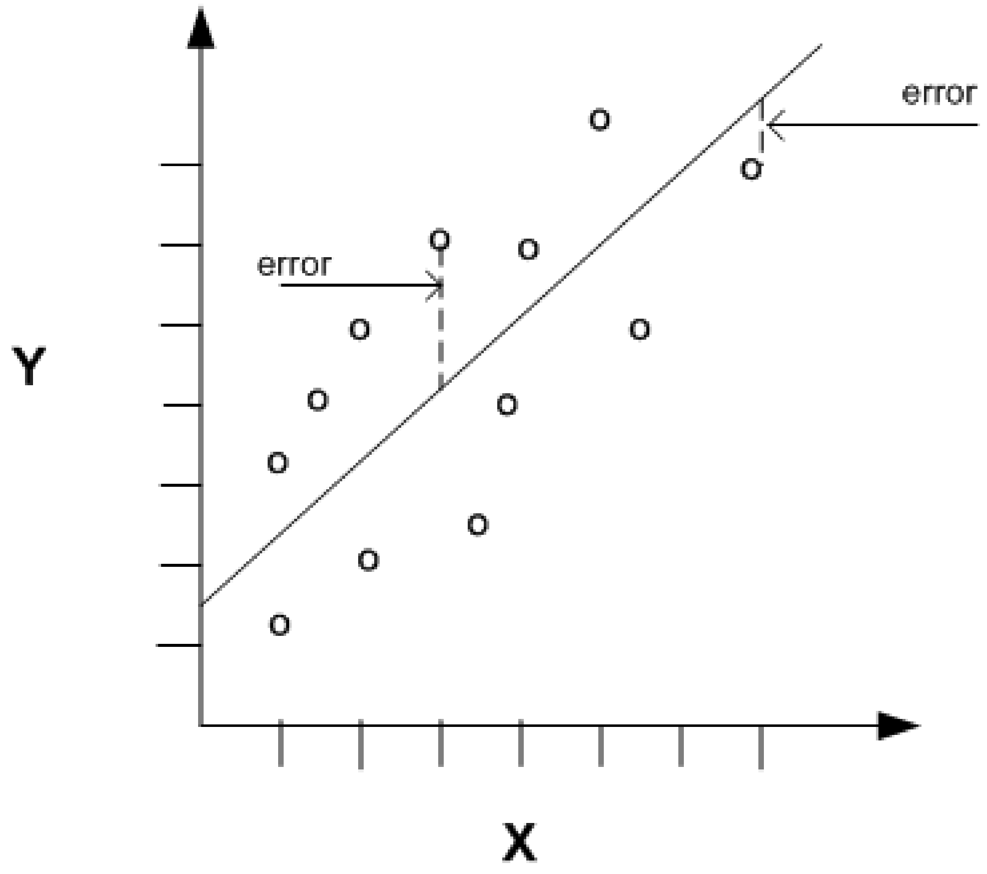
The calibration equation (of *n* calibration points) can now be solved for the unknowns in the transformation matrix, *X*.

$$\begin{bmatrix} u_i & v_i & w_i \\ & \vdots & \\ u_n & v_n & w_n \end{bmatrix} X = \begin{bmatrix} x_i & y_i & z_i \\ & \vdots & \\ x_n & y_n & z_n \end{bmatrix}$$

Where m_{ij} is the unknown in *X*,

$$X = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$


v
Camera
Frame
w
u
z
y
FRANKA
Frame
x
H8
H1
A8
A1
Deadzone


error
error
Y
X

In MATLab, the function for solving this equation is simply `X = a\b`, or less commonly written as `X = mldivide(a,b)`. The mldivide() function in MATLab is a complex one, and utilises many different possible algorithms depending on its inputs. To get the similar behaviour in Python, the numpy's lstsq function is used. It has similarites and differences which have been discussed {1} {2}, but ultimately provides the same functionality of returning a least square solution to the equation. The function is used as in our example below:

```python
import numpy as np
from numpy import random

num_pts = 4

A = random.rand(num_pts, 3)
one = np.ones((num_pts, 1))
A = np.column_stack([A, one])
print("A", A)
print("\n")

T = random.rand(3, 4)
xrow = np.array([0,0,0,1])
T = np.vstack([T, xrow])
print("T", T)
print("\n")

B = np.dot(A, T)
print("B", B)
print("\n")

x = np.linalg.lstsq(A, B, rcond=None)[0]
print("x", x)
```

Implementation

class `tools.transform.`**`TransformFrames`**(*frame_a_points*, *frame_b_points*, *debug=False*)

Stores the transformation relationship between original and target reference frames. The class should be instantiated with a minimum of 4 frame->frame calibration points.

`transform`(*coordinate*)

Transforms an x,y,z coordinate to the target reference frame.

`transform_reversed`(*coordinate*)

Transforms an x,y,z coordinate from the target reference frame back to the original.

5.6 Perception

5.6.1 Description

The perception module enables chess moves to be recognised using machine vision. It is based on OpenCV and runs on Python 2.7. An Asus Xtion camera provides frames as an input, which are then processed by the perception engine. It outputs a Black White Empty (BWE) matrix that is then passed on to the chess engine. This matrix is returned as a nested list, filled with 'E' for empty chess squares, 'W' if the square is occupied by a white piece, and 'B' if it's occupied by a black piece. As the initial setup of the chess pieces is constant, this matrix is sufficient to determine the state of the game at any time.

5.6.2 Design

There are several classes such as Line, Square, Board and Perception within the pereption module. The code works in the following sequence:

1) A picture of an empty board is taken and its grid is determined. 64 Square instances are generated, each holding information about the position of the square, its current state (at this stage they are all empty), and color properties of the square. The 64 squares are stored in a Board instance, holding all the information about the current state of the game. The Board instance is stored in a Perception instance, representing the perception engine in its entirety and facilitating access from other modules.

2) The chessboard is populated by the user in the usual setup. The initial BWE matrix is assigned, looking like this:

```
B B B B B B B B
B B B B B B B B
E E E E E E E E
E E E E E E E E
E E E E E E E E
E E E E E E E E
W W W W W W W W
W W W W W W W W
```

3) When the user has made his or her move, a keyboard key is pressed. This triggers a new picture to be taken and compared to the previous one. The squares that have changed (i.e. a piece has been moved from or to) are analysed in terms of their RGB colors and assigned a new state based thereupon. The BWE matrix is updated and passed to the chess engine, for instance to:

```
B B B B B B B B
B B B B B B B B
E E E E E E E E
E E E E E E E E
E E E E W E E E
E E E E E E E E
W W W W E W W W
W W W W W W W W
```

4) The chess engine determines the best move to make and the robot executes it. The user then needs to press the keyboard again to update the BWE to include the opponent's (robot) move. Upon pressing a key, the BWE might look like this:

```
B B B B B B B B
B B B B E B B B
E E E E E E E E
E E E E B E E E
E E E E W E E E
```

```
E E E E E E E E
W W W W E W W W
W W W W W W W W
```

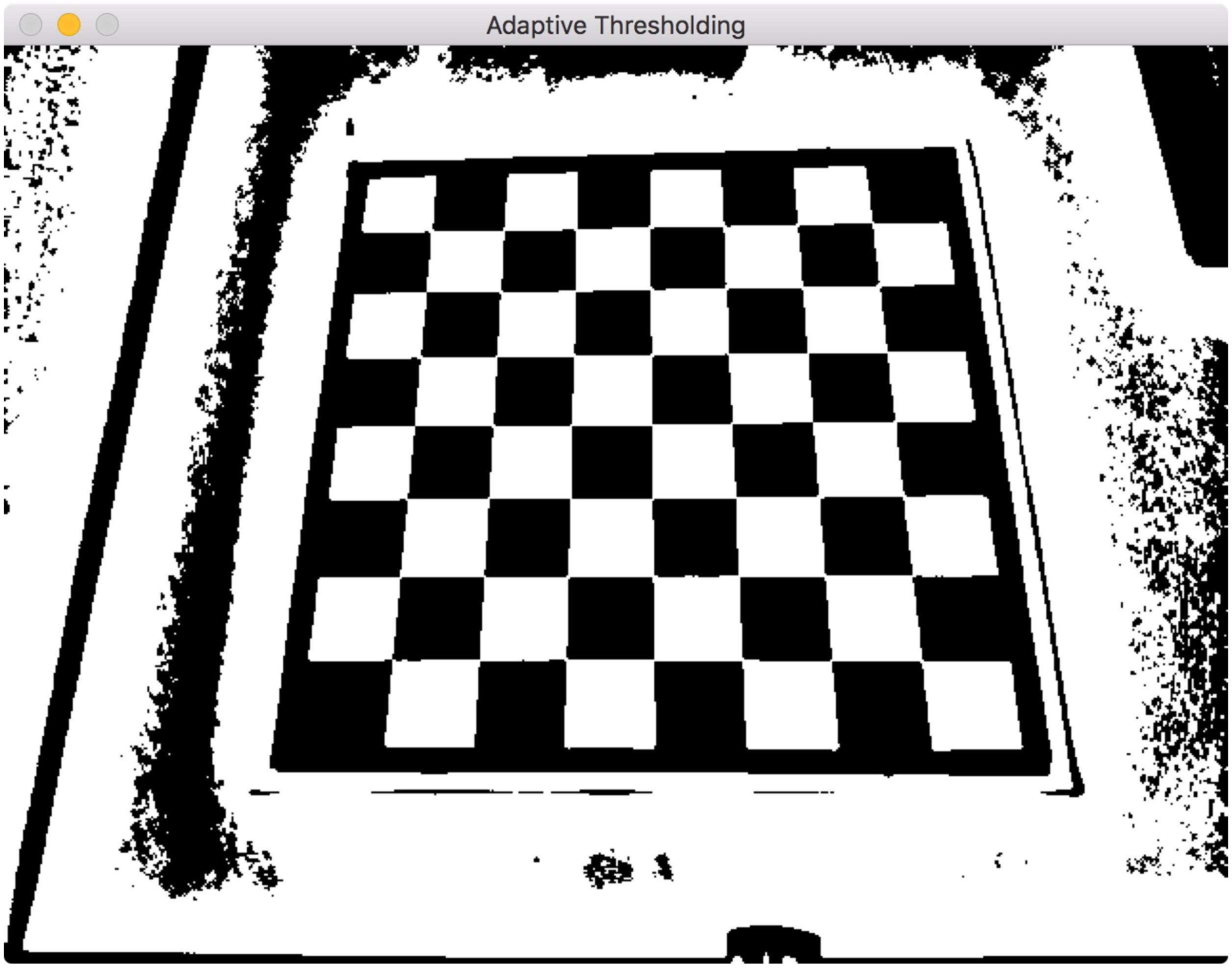
5) Return to step 3. The loop continues until the game ends.

5.6.3 Machine Vision

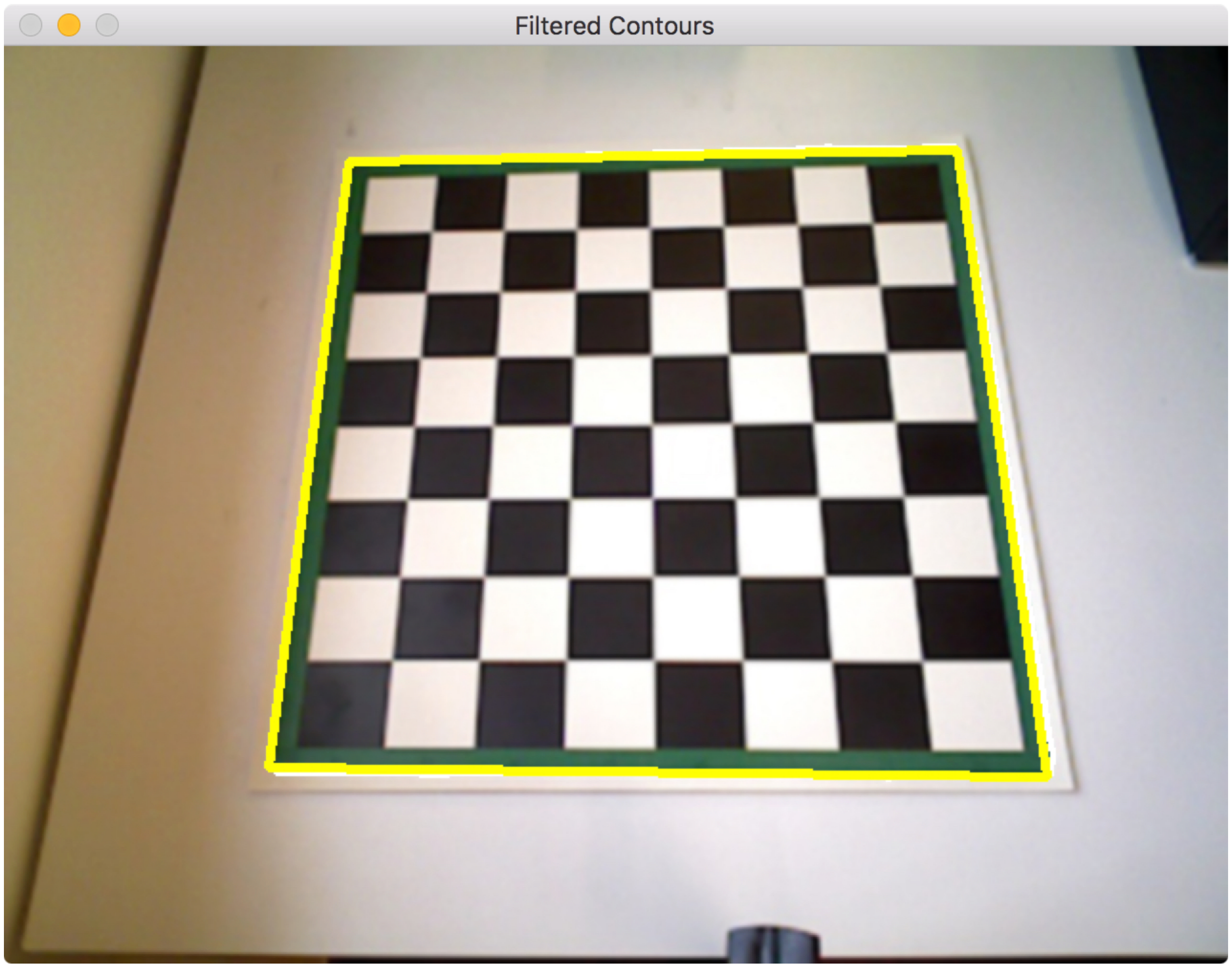
This section is concerned with how the machine vision works that achieves perception.

Thresholding, Filtering and Masking

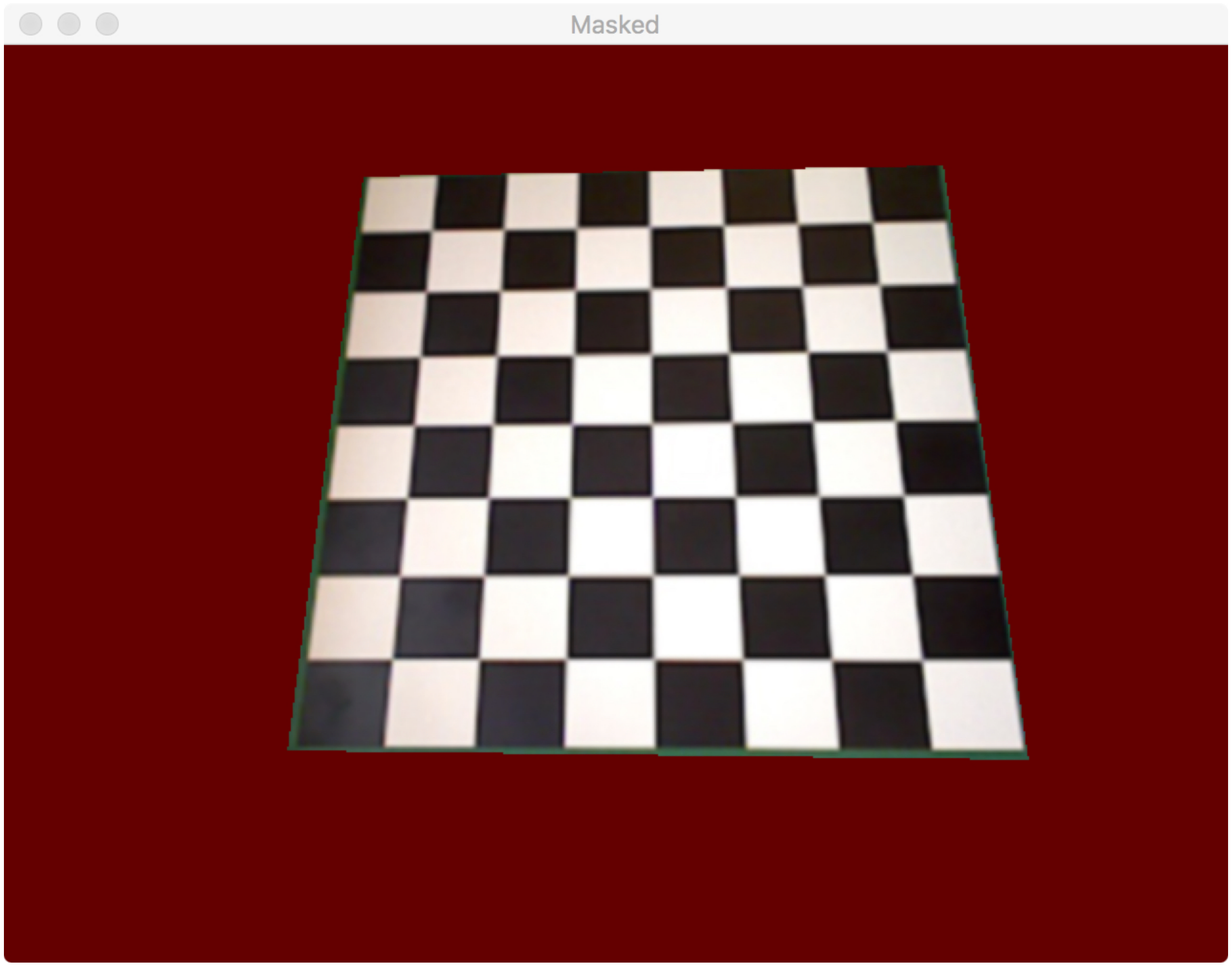
The first stage of image analysis takes care of thresholding, filtering and masking the chessboard. Adaptive thresholding is used to subsequently do contour detection, concerned with detecting the board edges.



A filter looks for squares within the image and filters the largest one, the chessboard. This was achieved by looking at the largest contour within the image that had a ratio of area to perimeter typical for a square.


Filtered Contours

The chessboard is masked and the rest of the image is replaced with a homogeneous color. We chose red for this purpose, as it was a color which did not interfere with other colors in the image.



Determining the chess corners and squares

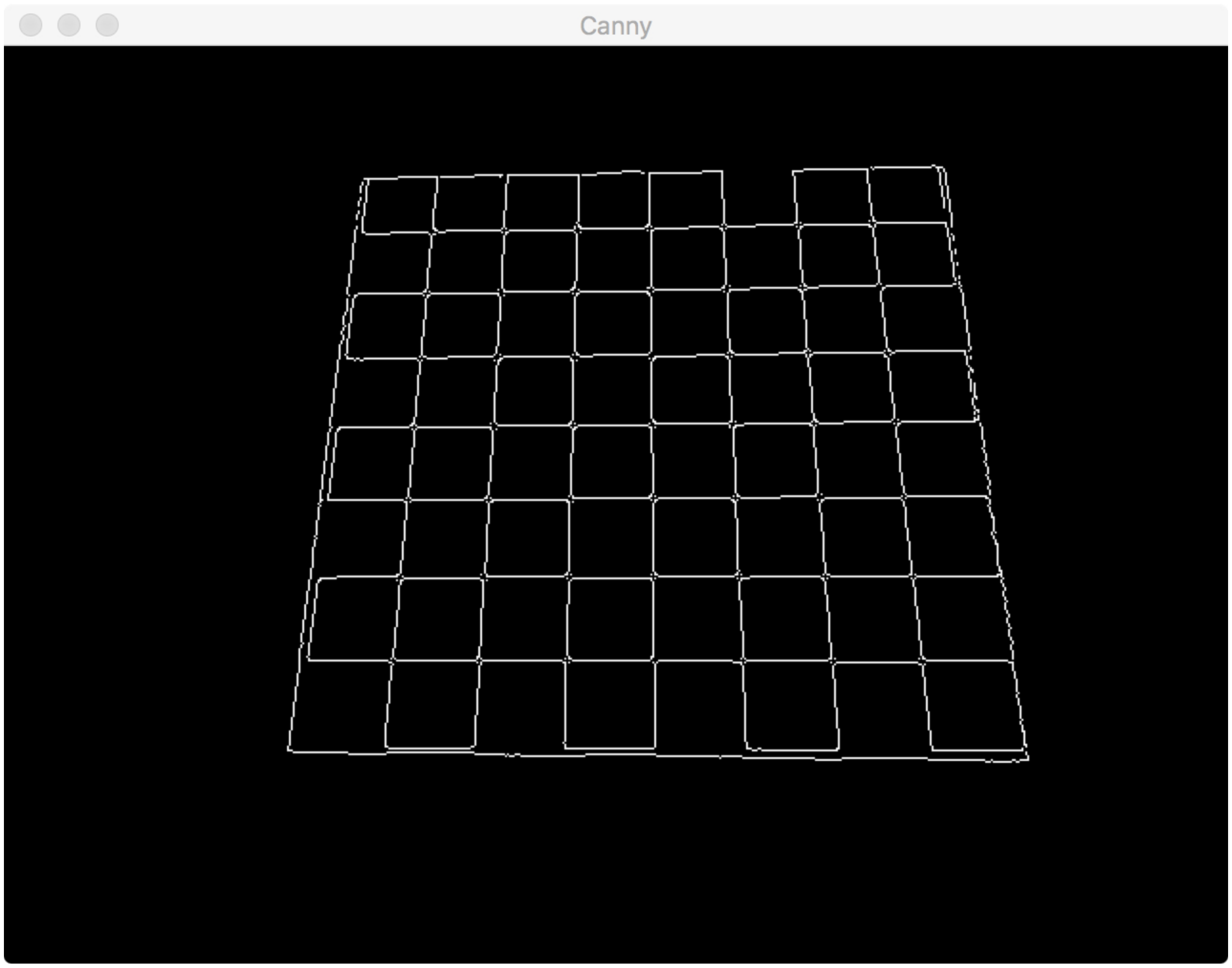
Canny edge detection is needed to determine Hough lines. It is an algorithm consisting of various stages of filtering, intensity gradient calculations, thresholding and suppression to identify edges. As shown below, it aids with identifying the chess grid.

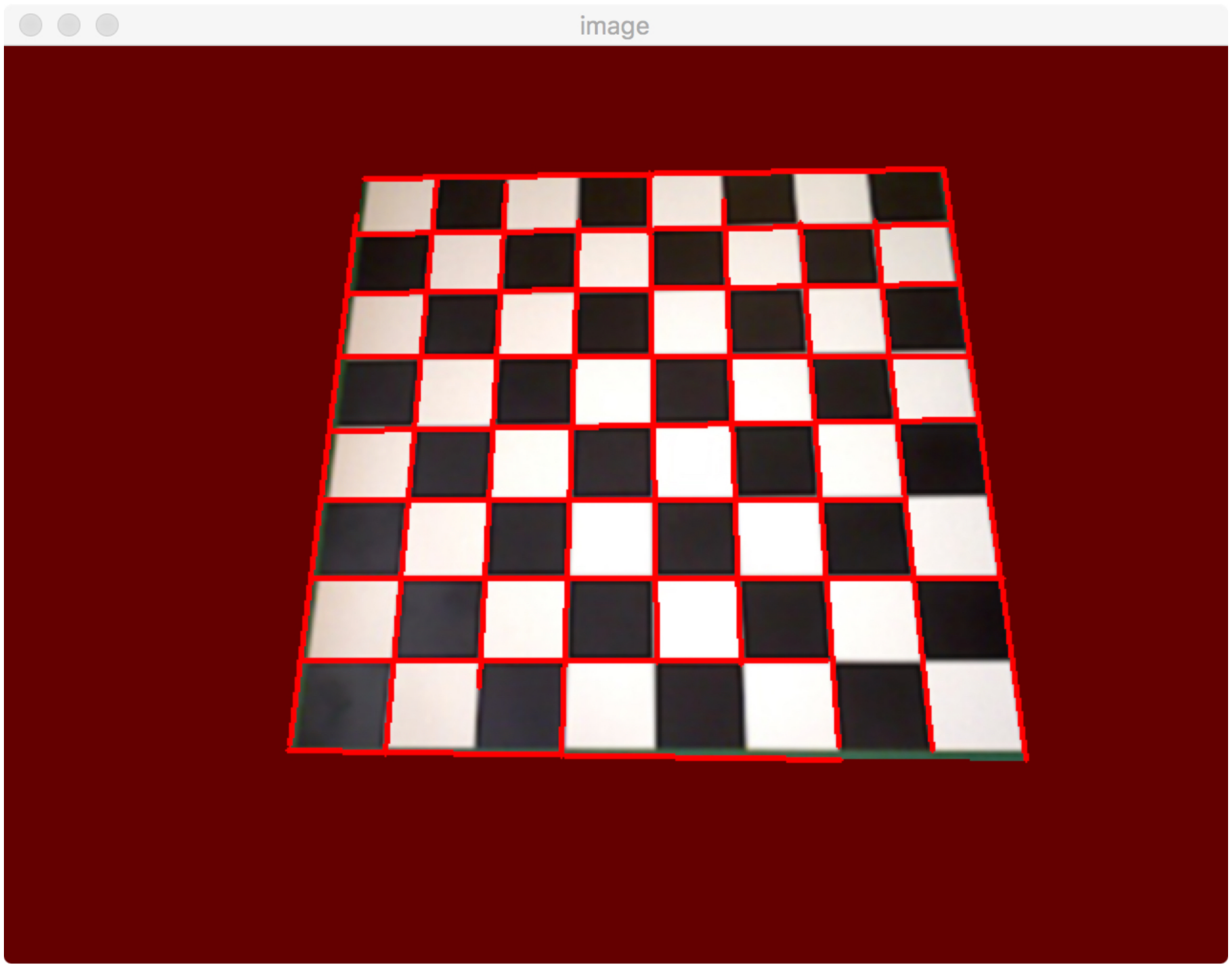
Hough lines are subsequently calculated from the Canny image. Lines are identified and instantiated with their gradients and positions. At that stage, similar lines are sometimes clustered together, so gradient filtering is applied to minimise the number of lines without losing the ones needed.

Intersections of Hough lines are found by equating two lines and solving. As there are still some duplicates, an algorithm now does the final filtering to ensure that only 81 points remain. The corner points (9 x 9) are assigned to rows and columns within the chessboard. 64 (8 x 8) Square instances are then generated. Each holds information about its position, index, and color average within the ROI area (shown as a circle in its centre). When the game is setup, the latter is the square's 'empty color', i.e. black or white.

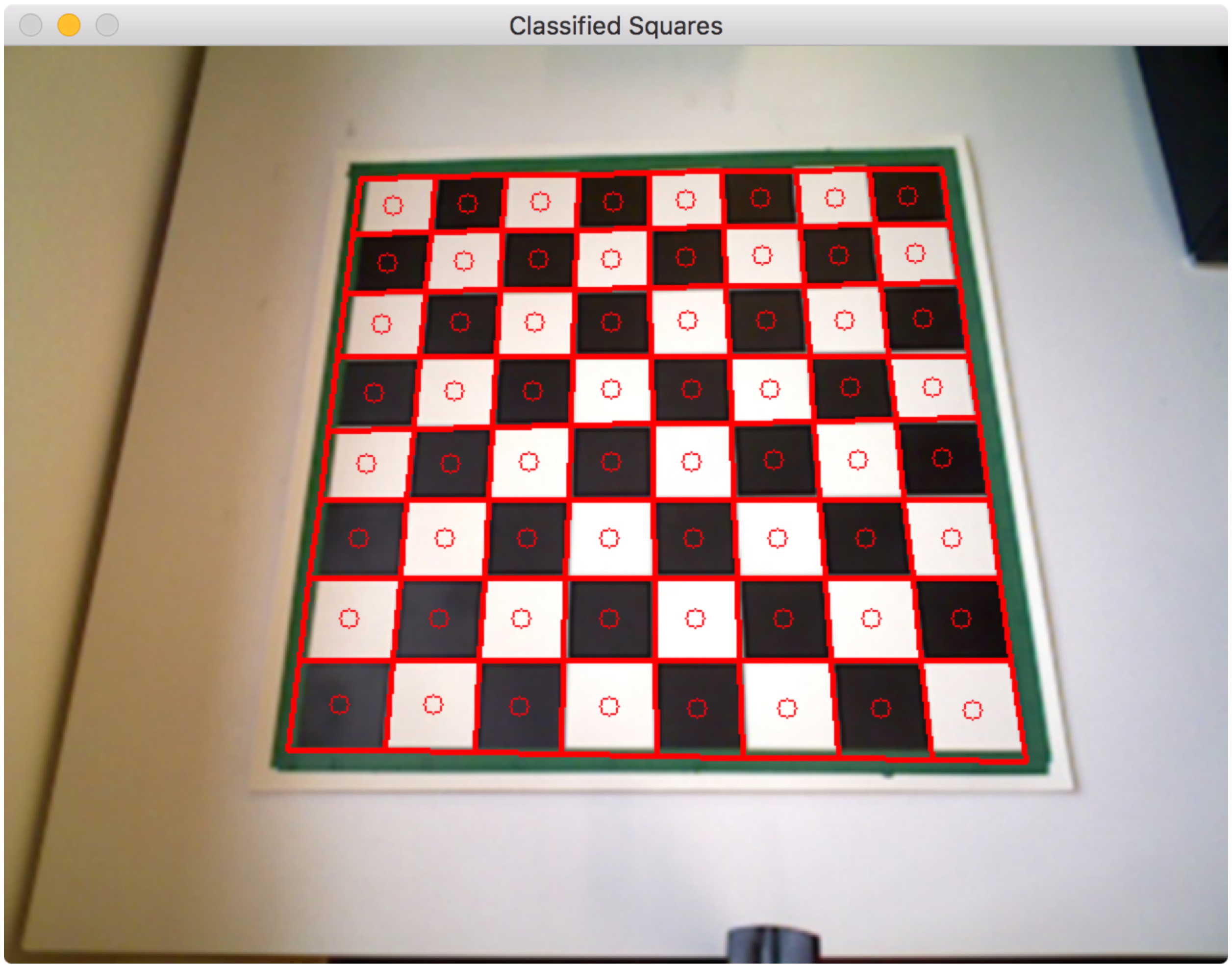
```
class Square:
    """
    Class holding the position, index, corners, empty colour and state of a chess␣
↪square
```

(continues on next page)


Canny



image


Classified Squares

(continued from previous page)

```
    """
    def __init__(self, position, c1, c2, c3, c4, index, image, state=''):
        # ID
        self.position = position
        self.index = index
        # Corners
        self.c1 = c1
        self.c2 = c2
        self.c3 = c3
        self.c4 = c4
        # State
        self.state = state

        # Actual polygon as a numpy array of corners
        self.contours = np.array([c1, c2, c3, c4], dtype=np.int32)

        # Properties of the contour
        self.area = cv2.contourArea(self.contours)
        self.perimeter = cv2.arcLength(self.contours, True)

        M = cv2.moments(self.contours)
        cx = int(M['m10'] / M['m00'])
        cy = int(M['m01'] / M['m00'])

        # ROI is the small circle within the square on which we will do the averaging
        self.roi = (cx, cy)
        self.radius = 5

        # Empty color. The colour the square has when it's not occupied, i.e. shade␣
↪of black or white. By storing these
        # at the beginnig of the game, we can then make much more robust predictions␣
↪on how the state of the board has
        # changed.
        self.emptyColor = self.roiColor(image)
```

The board can now be instantiated as a collection of all the squares. The sequence of functions called to generate the board is called within the makeBoard function, shown below:

```
    def makeBoard(self, image, depthImage):
        """
        Takes an image of an empty board and takes care of image processing and␣
↪subdividing it into 64 squares
        which are then stored in one Board object that is returned. Expanding to␣
↪depth calibration has not yet been
        finished.
        """
        try:
            # Process Image: convert to B/w
            image, processedImage = self.processFile(image)
        except Exception as e:
            print(e)
            print("There is a problem with the image...")
            print("")
            print("The image print is:")
            print(image)
```

(continues on next page)

(continued from previous page)

```
            print("")

        # Extract chessboard from image
        extractedImage = self.imageAnalysis(image, processedImage, debug=False)

        # Chessboard Corners
        cornersImage = extractedImage.copy()

        # Canny edge detection - find key outlines
        cannyImage = self.cannyEdgeDetection(extractedImage)

        # Hough line detection to find rho & theta of any lines
        h, v = self.houghLines(cannyImage, extractedImage, debug=False)

        # Find intersection points from Hough lines and filter them
        intersections = self.findIntersections(h, v, extractedImage, debug=False)

        # Assign intersections to a sorted list of lists
        corners, cornerImage = self.assignIntersections(extractedImage, intersections,
↪ debug=False)

        # Copy original image to display on
        squareImage = image.copy()

        # Get list of Square class instances
        squares = self.makeSquares(corners, depthImage, squareImage, debug=False)

        # Make a Board class from all the squares to hold information
        self.board = Board(squares)

        # Assign the initial BWE Matrix to the squares
        self.board.assignBWE()
```

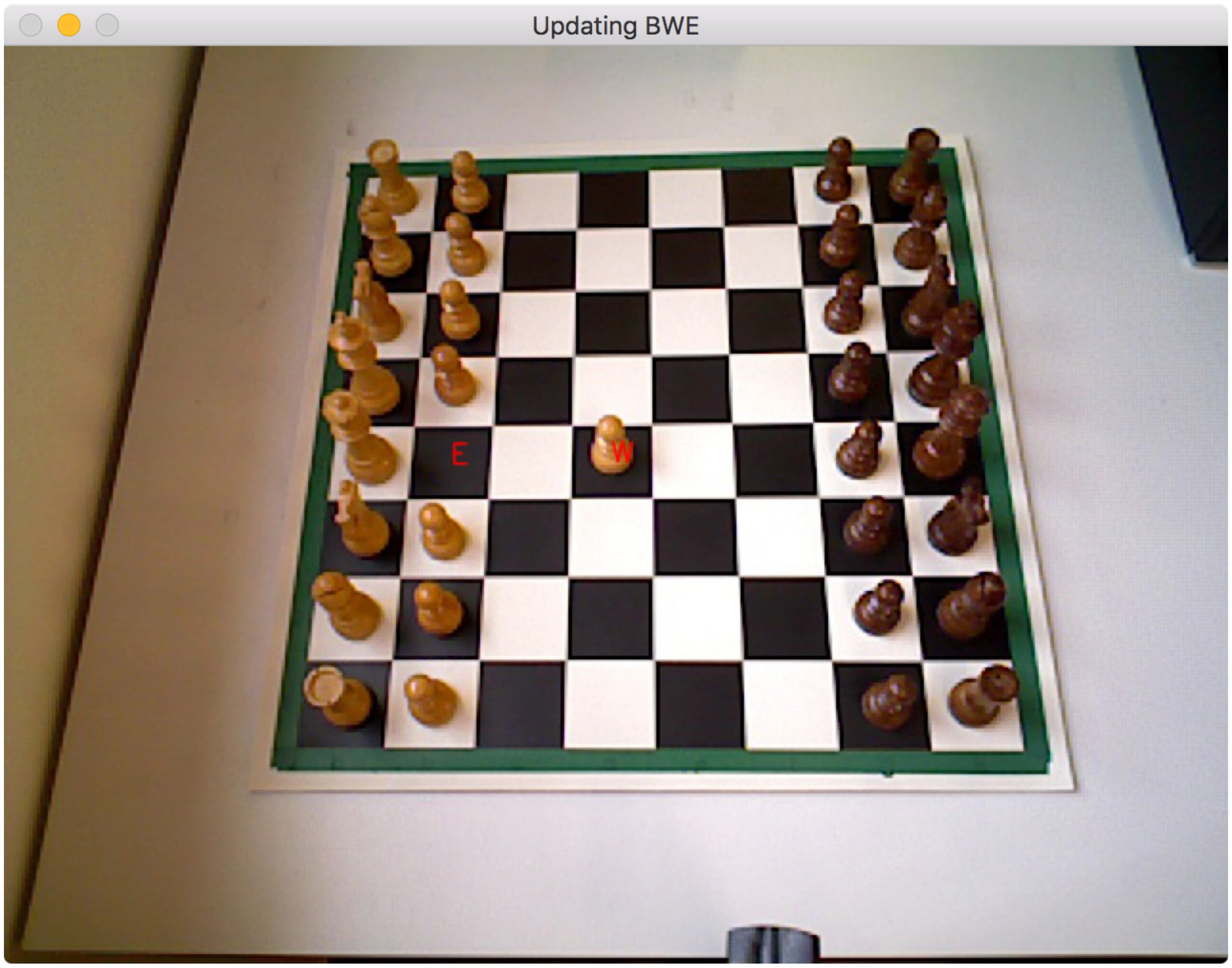
Updating the BWE matrix

When a piece is moved, the code detects changes between the previous and the current image. The centres of the bounding boxes surrounding that change region are matched with the squares. Two squares will be detected to have changed, as the centres of the change regions lie within them. A piece has been either moved from or to that square.

Both squares current Region of Interest (ROI) colors are taken and compared against their 'empty colors', i.e. their colors when not occupied by a piece. This distance is quantified by a 3-dimensional RGB color distance. The one with the smaller distance to its empty state must currently be an empty square, meaning a piece has been moved from it. Its old state (when the piece still was there) is saved temporarily, while its state is reassigned as empty. The non-empty square now takes the state of the piece that has been moved to it, i.e. the empty square's old state.

```
    def updateBWE(self, matches, current):
        """
        Updates the BWE by looking at the two squares that have changed and␣
↪determining which one is now empty. This
        relies on calculated the distance in RGB space provided by the classify␣
↪function. The one with a lower distance
        to the colour of its empty square must now be empty and its old state can be␣
↪assigned to the other square that
        has changed.
        """
```

(continues on next page)


Updating BWE
E
W

(continued from previous page)

```
        # Calculates distances to empty colors of squares
        distance_one = matches[0].classify(current)
        distance_two = matches[1].classify(current)

        if distance_one < distance_two:
            # Store old state
            old = matches[0].state
            # Assign new state
            matches[0].state = 'E'
            self.BWEmatrix[matches[0].index] = matches[0].state
            # Replace state of other square with the previous one of the currently␣
↪white one
            matches[1].state = old
            self.BWEmatrix[matches[1].index] = matches[1].state

        else:
            # Store old state
            old = matches[1].state
            # Assign new state
            matches[1].state = 'E'
            self.BWEmatrix[matches[1].index] = matches[1].state
            # Replace state of other square with the previous one of the currently␣
↪white one
            matches[0].state = old
            self.BWEmatrix[matches[0].index] = matches[0].state
```

5.6.4 Limitations

This perception module has limitations, which are mostly in terms of robustness and setup. With further development it should be able to recognise the chessboard grid even if it is populated. Changing light conditions make the perception engine very unstable, as the classification of states of chess squares relies on a constant light setting. There are still many improvements that can be made in terms of integrating the perception engine with the chess engine and the motion generation. There are inconsistencies with storing the BWE as a numpy array or as a nested list. Finally, this perception engine relies on having an image of the empty board first.

Please contact Paolo Rüegg under pfr15@ic.ac.uk in case you would like to continue working on this and require further information about this code.

5.6.5 Implementation

Documentation:

class `perception.mainDetect.`**`Perception`**(*board=0*, *previous=0*)

The perception class contains a Board instance as well as functions needed to generate it and output a BWE matrix. The updating of the BWE is done within the Board class.

`assignIntersections`(*image*, *intersections*, *debug=True*)

Takes the filtered intersections and assigns them to a list containing nine sorted lists, each one representing one row of sorted corners. The first list for instance contains the nine corners of the first row sorted in an ascending fashion. This function necessitates that the chessboard's horizontal lines are exactly horizontal on the camera image, for the purposes of row assignment.

bwe(*current*, *debug=False*)
Takes care of taking the camera picture, comparing it to the previous one, updating the BWE and returning it.

cannyEdgeDetection(*image*, *debug=False*)
Runs Canny edge detection

categoriseLines(*lines*, *debug=False*)
Sorts the lines into horizontal & Vertical. Then sorts the lines based on their respective centers (x for vertical, y for horizontal).

detectSquareChange(*previous*, *current*, *debug=True*)
Take a previous and a current image and returns the squares where a change happened, i.e. a figure has been moved from or to.

drawLines(*image*, *lines*, *color=(0*, *0*, *255)*, *thickness=2*)
Draws lines. This function was used to debug Hough Lines generation.

findIntersections(*horizontals*, *verticals*, *image*, *debug=True*)
Finds intersections between Hough lines and filters out close points. The filter relies on a computationally expensive for loop and could definitely be improved.

houghLines(*edges*, *image*, *debug=True*)
Detects Hough lines

imageAnalysis(*img*, *processedImage*, *debug=False*)
Finds the contours in the chessboard, filters the largest one (the chessboard) and masks it.

initialImage(*initial*)
This function sets the previous variable to the initial populated board. This function is deprecated.

makeBoard(*image*, *depthImage*)
Takes an image of an empty board and takes care of image processing and subdividing it into 64 squares which are then stored in one Board object that is returned. Expanding to depth calibration has not yet been finished.

makeSquares(*corners*, *depthImage*, *image*, *debug=True*)
Instantiates the 64 squares given 81 corner points.

printBwe(*bwe*)
Prints the BWE.

processFile(*img*, *debug=False*)
Converts input image to grayscale & applies adaptive thresholding.

showImage(*image*, *name='image'*)
Shows the image

class perception.boardClass.**Board**(*squares*, *BWEmatrix=[]*, *leah='noob coder'*)
Holds all the Square instances and the BWE matrix.

assignBWE()
Assigns initial setup states to squares and initialises the BWE matrix.

draw(*image*)
Draws the board and classifies the squares (draws the square state on the image).

getBWE()
Converts BWE from list of strings to a rotated numpy array

updateBWE(*matches*, *current*)
Updates the BWE by looking at the two squares that have changed and determining which one is now empty. This relies on calculated the distance in RGB space provided by the classify function. The one

with a lower distance to the colour of its empty square must now be empty and its old state can be assigned to the other square that has changed.

whichSquares (*points*)
Returns the squares which a list of points lie within. This function is needed to filter out changes in the images that are compared which have nothing to do with the game, e.g. an arm.

class perception.squareClass.**Square** (*position*, *c1*, *c2*, *c3*, *c4*, *index*, *image*, *state=''*)
Class holding the position, index, corners, empty colour and state of a chess square

classify (*image*, *drawParam=False*, *debug=False*)
Returns the RGB 3-dimensional distance from a squares current color to its empty color.

draw (*image*, *color=(0, 0, 255)*, *thickness=2*)
Draws the square onto an image.

getDepth (*depthImage*)

roiColor (*image*)
Finds the averaged color within the ROI within the square. The ROI is a circle with radius r from the centre of the square.

class perception.lineClass.**Line** (*x1*, *y1*, *x2*, *y2*)

draw (*image*, *color=(0, 0, 255)*, *thickness=2*)
Draws line onto an image.

intersect (*other*)
Finds intersections points between two lines.

perception.lineClass.**filterClose** (*lines*, *horizontal=True*, *threshold=40*)
Filters close lines.

5.7 Chess Engine

5.7.1 Description

The chess engine is an interface to any chess AI chosen. This project uses sunfish by thomasahle as the chess AI of choice. The chess engine spins up an instance of the sunfish program, and then sends the users plays to the AI. The AI then responds with what it decides should be the computer/robots move. To do this, some small modifications had to be made to the sunfish source code:

- Add a wait loop for the command queue to be filled with the user's response. Inform the engine if the move is invalid or not.

```
        pass_number = 1  # We set the first pass number before entering the loop
        while move not in pos.gen_moves():
            if pass_number > 1:  # if on the second pass, the previous must've been␣
↪invalid
                valid_queue.put(0)  # report to engine that the input was invalid

            command = command_queue.get(block=True)  # get a command from engine if␣
↪available

            match = re.match('([a-h][1-8])'*2, command)
            if match:
                move = parse(match.group(1)), parse(match.group(2))
```

(continues on next page)

(continued from previous page)

```
            else:
                # Inform the user when invalid input (e.g. "help") is entered
                print("Please enter a move like g8f6")

            pass_number += 1
```

- Inform the game engine if the user's move caused them to win.

```

        # After our move we rotate the board and print it again.
        # This allows us to see the effect of our move.
        print_pos(pos.rotate())

        if pos.score <= -MATE_LOWER:
            print("You won")
            valid_queue.put(1)  # inform engine that the move was accepted and user␣
↪won
            break
```

- Send whether the computer won or not. Also, send the computers move back to the chess engine through a seperate queue.

```
            print("Checkmate!")
            valid_queue.put(2)  # inform engine that the move was accepted and␣
↪computer won
        else:
            valid_queue.put(3)  # inform engine that the move was accepted and␣
↪sunfish will reply
        # The black player moves from a rotated position, so we have to
        # 'back rotate' the move before printing it.
        computer_move = render(119-move[0]) + render(119-move[1])
        reply_queue.put(computer_move, block=True)  # reply to engine
        print("My move:", computer_move)
        pos = pos.move(move)
```

These minimal changes mean that any AI that takes a users input as a chess command (e.g. 'a2a4') can be modified with minimal code to work with our chess engine.

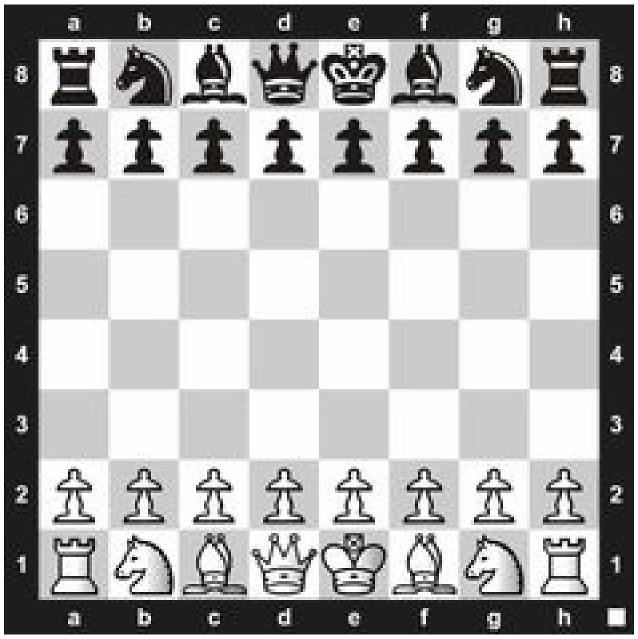
5.7.2 Design

The CE (chess engine) is written by the team, it can provide all the chess functionality needed.

Tip: Lower case letters represent black pieces (p,r,n,b,k,q), and upper case letters represent white pieces (P,R,N,B,K,Q). This follows the model used in *sunfish*.

The BWE matrix provided by the Perception module is taken. This Black-White-Empty matrix is simply a list of strings which represent each square on the board. The elements on the list correspond to the positions on the board as `[A8, B7, C7, ..., F1, G1, H1]`. For the image below, this would be: `['B','B','B',...,'W', 'W','W']`.

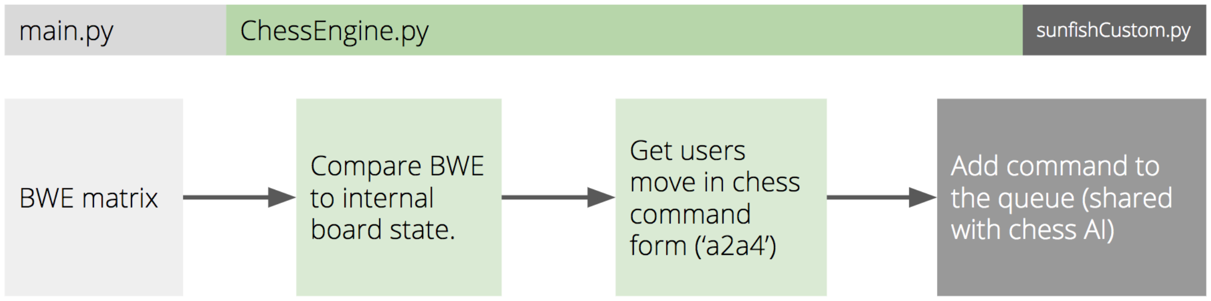
This matrix is compared with an internally stored matrix in the CE. This means the CE can understand:



- where the piece moved from
- where it moved to

and construct a chess command from it. This BWE matrix is of course checked for logical inconsistencies by measuring the change in number of black, white or empty squares in a single turn. Now that a potential chess move has been obtained, it is added to the command queue, a shared resource that both the CE and the chess AI (sunfish) have access to.

Hint: Moving pawn piece A2 to A4 at the beginning of the game would require command 'a2a4'


main.py
ChessEngine.py
sunfishCustom.py
BWE matrix
Compare BWE to internal board state.
Get users move in chess command form ('a2a4')
Add command to the queue (shared with chess AI)

This move is trialled internally on the AI, and it responds with a number of options.

- The move is invalid, in which case reporting of an illegal move back to the user needs to be carried out
- The move is valid and has caused the user to win, in which case the user should be told
- The move is valid and the computer responds with move (which may be a checkmate)

If the computer replies with a move, it will put it on the reply queue which is shared with the CE. The CE must now understand exactly what the chess AI is asking of it. It will have received a command such as 'n, b1c4' (i.e. knight b1 to c4). The CE then splits this command into the start and end position of the move. It then converts these positions into indices, which it uses to search its internally stored board for the type of piece that is moving. In addition it checks if there is already a piece existing at the end position.

The CE first updates its internal board to remember the move the computer just made, then returns the following information to the caller (function):

- Firstly, if there is a piece to be killed, its location and type.
- Then, the start location of the piece moving, and its type.
- Finally, the end location of the piece moving.

5.7.3 Limitations

The chess engine has limitations, which could be implemented in later versions:

- No support for pawn piece conversion (bringing pieces back onto the board).
- No support for special chess moves such as castling.

5.7.4 Implementation

Example usage:

```
from chess.engine import ChessEngine

bwe_list  # Get BWE from the camera

engine = ChessEngine()
code, result = engine.input_bwe(bwe_list)

if code == -1:
    print("There was a problem with the BWE matrix")
elif code == 0:
    print("Invalid move by user: ", result)
elif code == 1:
    print("The user won the game")
elif code == 2:
    print("The computer has won the game", result)
    franka_move(result)
    franka_celebrate()
elif code == 3:
    print("The computer's move is: ", result)
    franka_move(result)
else:
  print("Error code not recognised: ", code)
```

Documentation:

This engine is designed to interface with the modified sunfish file to provide a specialised interface between the other modules in this project and the chess logic underneath.

class chess.engine.**ChessEngine**(*debug=False*, *suppress_sunfish=True*)

Engine that manages communication between main program and chess AI Sunfish (running in separate process).

It's main purpose is to take a BWE matrix as the user's potential move and provide an analysis of this move by either reporting back its invalidity or the AI's response.

input_bwe(*bwe*)

Takes in the latest BWE and tries to input that to Sunfish AI.

Returns:

code [int]

- `-1`, BWE was invalid
- `0`, move was invalid
- `1`, user won the game with move
- `2`, computer won the game with checkmate
- `3`, computer did not win and

result Depending on `code`, this will vary:

- `-1`, None
- `0`, Move that was invalid as string
- `1`, None
- `2`, Move that computer is playing to win as tuple
- `3`, Move that computer is playing as tuple

Computer move is a list of tuples in the form:

- `[ (piece_to_move, move) ]`, or
- `[ (piece_to_kill, location), (piece_to_move, move) ]`

where,

- Pieces are single character strings.
- Locations are two character strings e.g. 'a2'
- Moves are 4 character strings e.g. 'a2a4'

start_sunfish_process()

Spins up external process for Sunfish AI.

Process communicates with three queues, the command queue (for user moves), the valid queue for confirming if the user queue is valid, and the reply queue for the Sunfish computer move response.

test()

This method is only used when debugging and developing the engine. It should not be called from other modules.

class chess.engine.**ChessState**(*debug=False*)

Class holding the ongoing state of the chess board.

compare_bwe(*new_bwe*)

Takes a BWE list and compares it to the existing game state. Return tuple of `(move_from, move_to)` indices for the single move that's detected. Does not verify if move is a legal one.

convert_to_index(*chess_pos*)

Takes an board position (e.g. 'a2') and converts to the corresponding board index (0-63).

convert_to_pos(*index_num*)

Takes an board index (0-63) and converts to the corresponding board position (e.g. 'a2').

get_bwe()

Returns the current game state as a BWE list.

get_bwe_move(*bwe*)
Takes a BWE list and returns the move that was made.

Attributes:

- bwe: A list of single character strings either 'B','W', or 'E'.

Returns:

- piece (str): Type of piece that moved e.g. `'P'`
- move (str): Move recognised in BWE e.g. `'a2a4'`

update_board(*bwe_matrix*)
Updates the game state with the latest BWE matrix after the user has played their turn. The BWE has been checked and the move has been checked with Sunfish.

exception chess.engine.**EngineError**(*message*)
Exception raised for errors in the game engine.

Attributes: message – explanation of the error

exception chess.engine.**Error**
Base class for exceptions in this module.

class chess.engine.**HiddenPrints**
Context manager for suppressing the print output of functions within its scope.

5.8 Motion

5.8.1 Description

This part of the project is responsible for providing the FRANKA arm with instructions on when and where to move. It takes in the algebraic notation (AN) of the start and end locations of the desired move, and then generates a motion plan to be sent to FRANKA for execution.

5.8.2 Grippers

Specialised grippers were designed to pick up the chess pieces.

Iteration 1:

A compliant design was used which allowed the grippers themselves to deform slightly when each piece was gripped. This allowed the uneven bumps of some pieces to be successfully gripped. They were 3D printed on the ProJet printer and then material from disposable silicone gloves was used to create the pads.

Iteration 2:

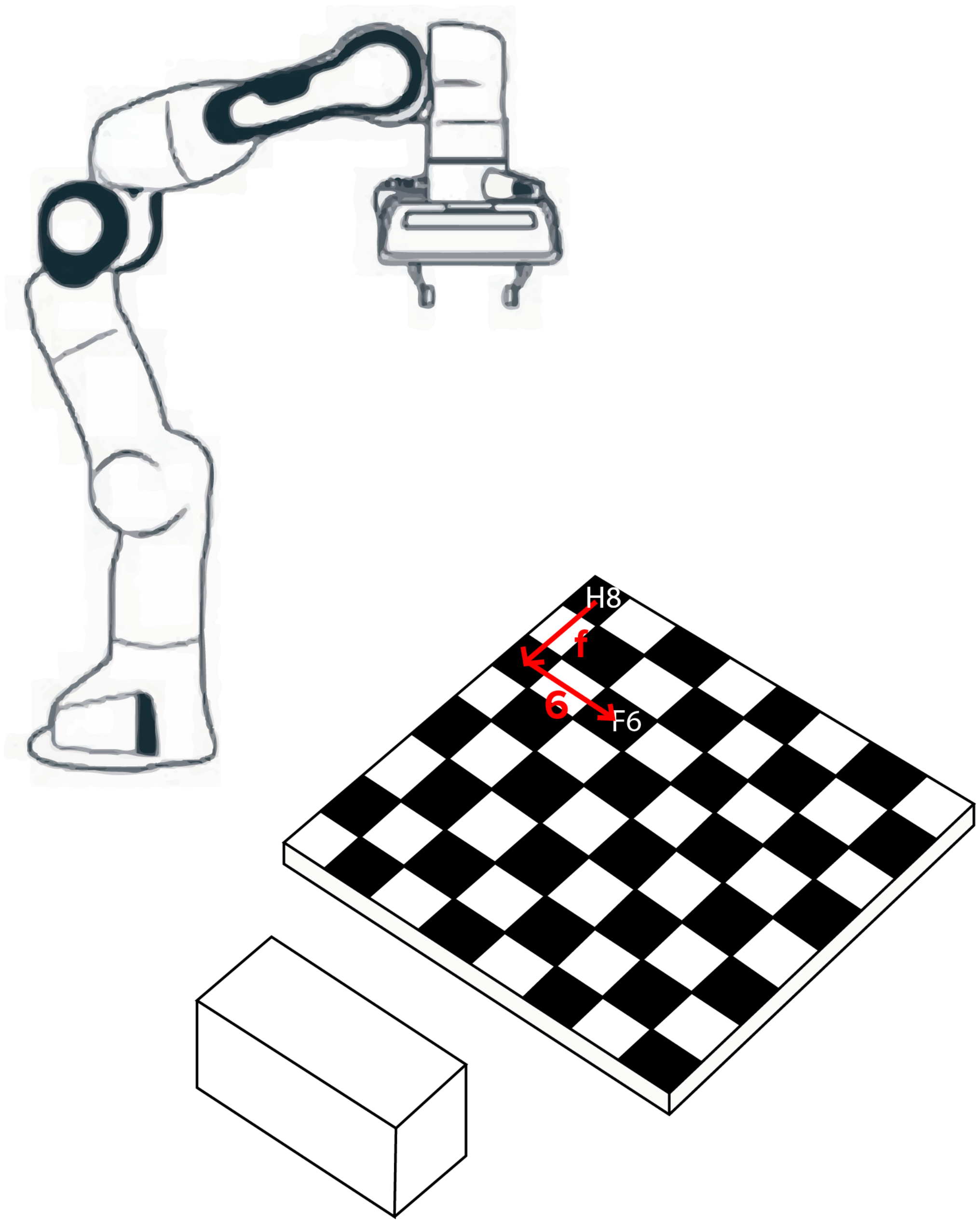
Small updates were made to the original design for the final versions. A taper angle was added to accomodate for pieces that are wider at the top than lower down. The width of the compressible part was increased to improve grip.

5.8.3 Generating the Trajectory

Firstly, the start and goal positions of the piece are converted from algebraic notation (AN) to `x, y, z` coordinates in FRANKA's reference frame. The AN is extracted from the output of the game engine, along with the piece type (king/queen etc). The conversion from AN is done by finding the x and y vectors from H8 to the selected square. Letter and number dictionaries of vectors are indexed through to find the necessary vectors (for more information on

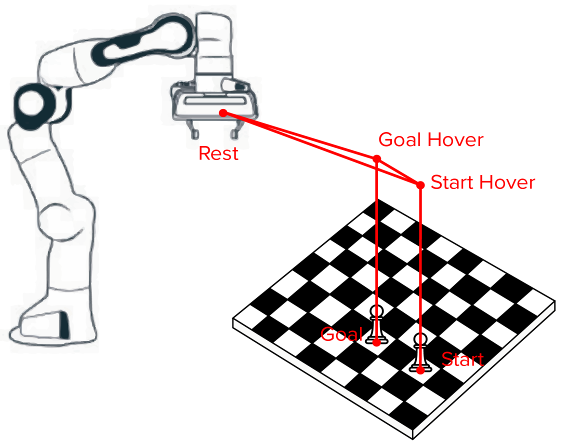
ROBOT INTELLIGENCE LAB

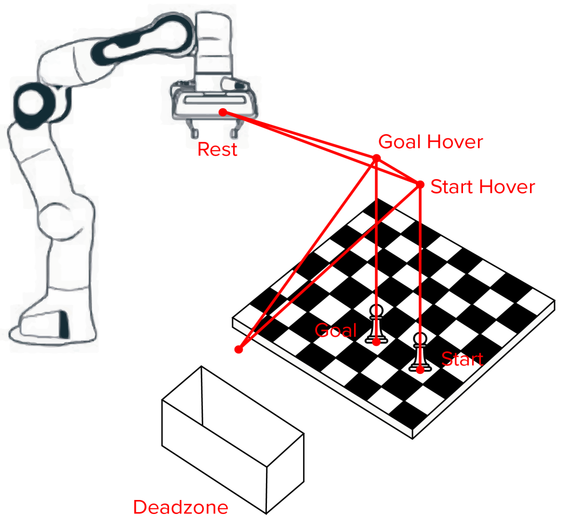
this see Calibration). These are then summed, giving the position vector of the square. For example, square F6 would be found by summing the F vector and the 6 vector, as shown below. The `z` coordinate is selected based on which piece it is, giving the best gripping height for that piece.



Note: These dictionaries are all created when `MotionPlanner` is instantiated.

The intermediate positions are then found; these are dependant on whether or not a piece has been killed, as shown below. The locations have been selected to allow FRANKA to follow a repeatable, direct path no matter what coordinates are parsed. The hover positions have `x, y` coordinates based on which pieces are being picked up while the `z` coordinate is hardcoded. The deadzone and rest positions are also hardcoded and created in the initialisation of the function.





Tip: The trajectory is vertically straight for all motions where a piece is gripped or ungripped on the board. This prevents the arm from colliding with any other pieces. The length of this section is determined by the hover height.

The positions are outputted in pairs of start and end locations, with every intermediate position represented. The start position is always set to the current position of FRANKA, rather than the end position of the previous command. This would account for any errors in FRANKA's motion.

```
            moves = np.array([[[current_position, self.rest], [current_position, ␣
↪move_from_hover],
                                [current_position, move_from]],
                               [[current_position, move_from_hover],
                                [current_position, move_to_hover], [current_position,␣
↪move_to]],
                               [[current_position, move_to_hover], [current_position,␣
↪self.rest]]])
```

The current position is forced at the beginning stage of each straight line movement.

Note: An additional move taking FRANKA from its current position to its rest position is always included in case the arm is not already in its rest location.

5.8.4 Executing the Trajectory

Before any trajectory is executed, the gripper is moved to it's ungripped position. This dimension is again hardcoded. Doing this at the beginning of the algorithm prevents any collisions between the end effector and the peices.

A single path is considered as a movement between 2 locations that involve a gripping action, for example start and goal, goal and deadzone or goal and rest. Only one path is executed at a time.

The path is extracted from the series generated by `generate_chess_moves` . A trapezoidal velocity profile is then applied as described in *Trapezium Velocity Profile* This path is sent to the FRANKA arm for execution. The current position of the end effector is updated.

It is then determined whether or not a gripping action should be executed, based on how many paths have already been run. This is possible as there are only 2 different series that could be executed (dead and not dead). If a gripping action is required, the desired gripping width is found using a global dictionary. The gripping action is executed by FRANKA.

This exectution process is repeated as many times as there are paths in the series, completing a single chess move.

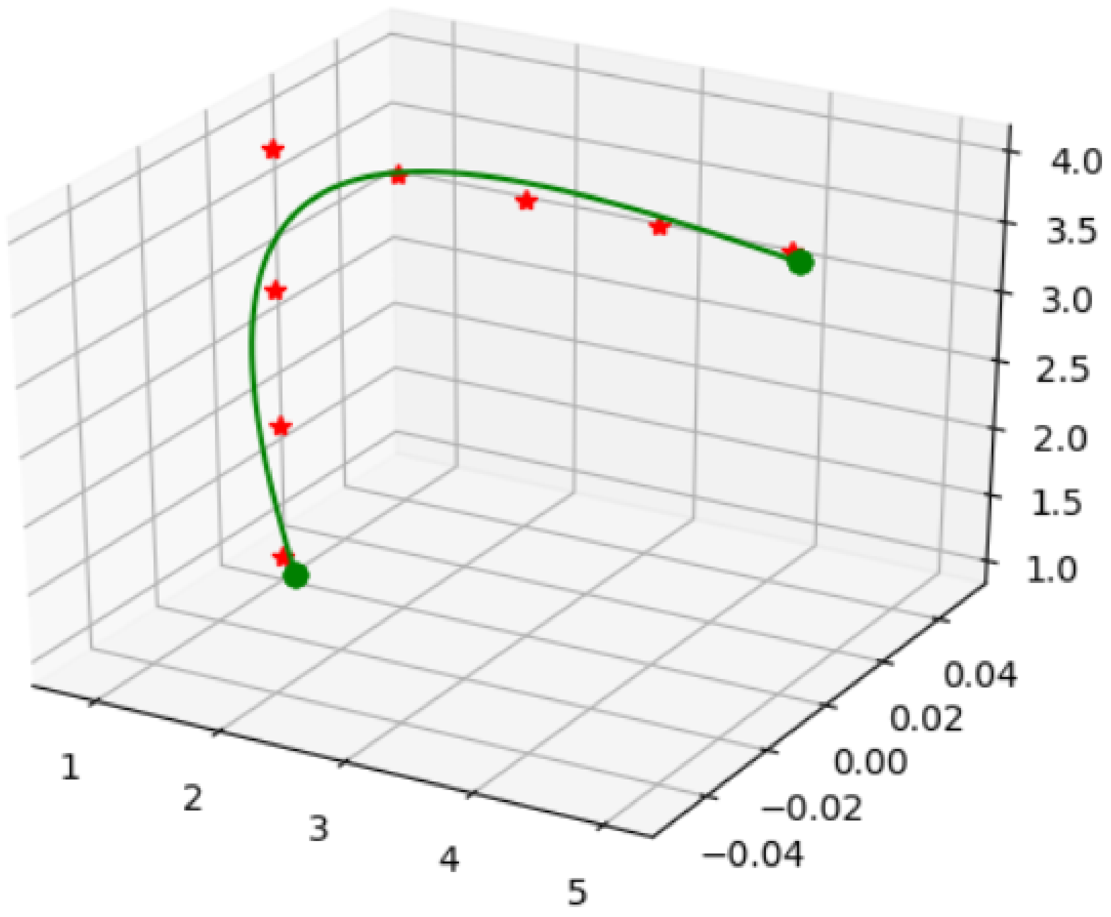
Tip: En passant moves have been disabled for this project, so the location of the killed piece will always be the same as the goal location.

5.8.5 Smoothing the trajectory

3 different methods were attempted to try and smooth the path to give a more natural movement.

1. Interpolation

The first method attempted was interpolation. This generated knot points along a given path and joined them up using a spline.



Tip: Interpolation generates new data points for a given set of preexisting data points. Spline interpolation then fits polynomials to these points to create a smooth curve. The result is a spline. These splines are easier to evaluate than high degree polynomials used in polynomial interpolation. {1}

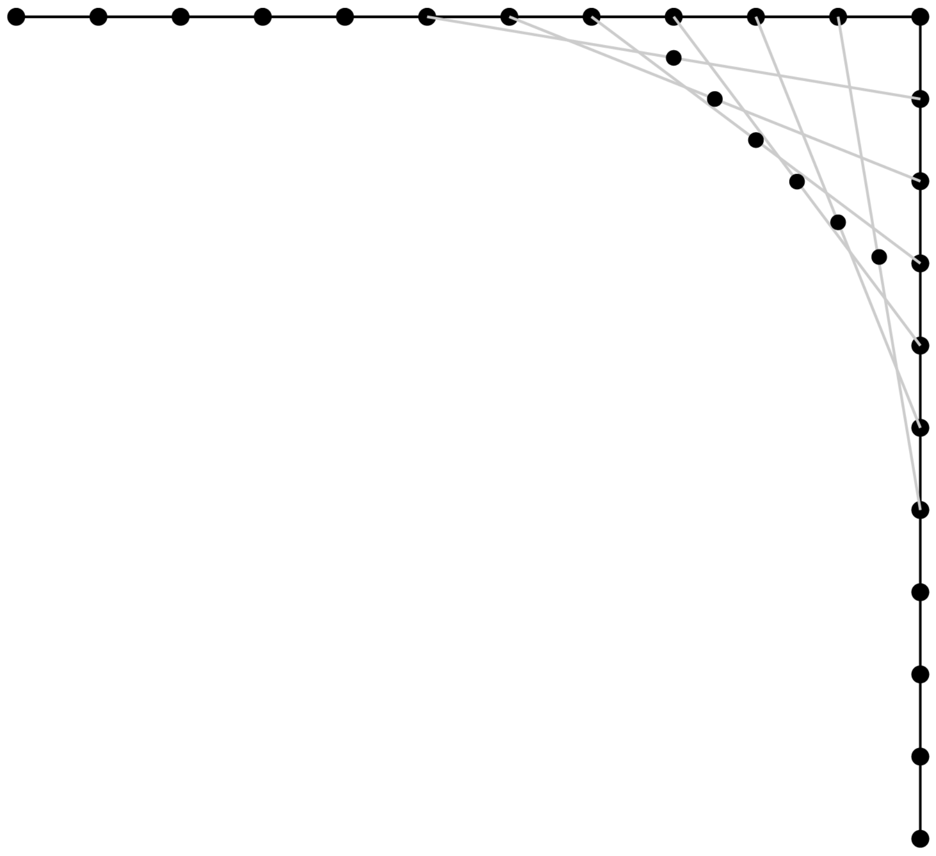
The B-spline representation of the 3-dimensional trajectory was found using a `scipy` function. This function takes in a list of coordinates that represent a trajectory as well as a smoothing factor. It returns the knot locations, the B-spline coefficients and the degree of the spline.

These outputs were then given to another `scipy` function that evaluates the value of the smoothing polynomial and its derivatives. It returns the coordinates of the knots and a list of coordinates making up the smoothed path.

The problem with this method was a lack of control over the path and difficultly in finding a consistently reliable smoothing factor.

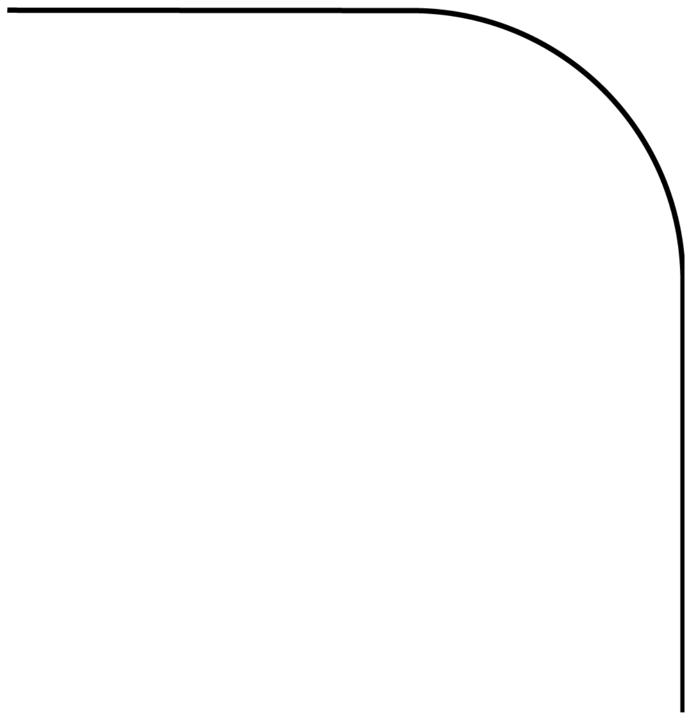
2. Tortoise and Hare

A discretised list of points was duplicated and shifted, such that one was several steps ahead of the other. The corresponding points in each list were then averaged, producing a third list of points. The result of this was a chamfered rather than smoothed corner.



3. Repeated Tortoise and Hare

To correct this problem, the same algorithm was run multiple times, chamfering the chamfer. This created a smooth path:



```
        for i in range(passes):
            trajectory_1_x = [item[0] for item in path]
            trajectory_1_y = [item[1] for item in path]
            trajectory_1_z = [item[2] for item in path]

            x_tortoise = trajectory_1_x[:-steps]  # remove last few coordinates
            x_hare = trajectory_1_x[steps:]  # first last few coordinates
            x_smooth_path = [(sum(i) / 2) for i in zip(x_tortoise, x_hare)]  #␣
↪average them

            y_tortoise = trajectory_1_y[:-steps]  # remove last few coordinates
            y_hare = trajectory_1_y[steps:]  # remove first few coordinates
            y_smooth_path = [(sum(i) / 2) for i in zip(y_tortoise, y_hare)]

            z_tortoise = trajectory_1_z[:-steps]  # remove last few coordinates
            z_hare = trajectory_1_z[steps:]  # remove first few coordinates
            z_smooth_path = [(sum(i) / 2) for i in zip(z_tortoise, z_hare)]
```

The problem here is that the distance between the points on the chamfered edges is smaller than the distance between points on unaffected edges. This did not work with the velocity profile algorithm, and was therefore not used in the final iteration of the motion code.

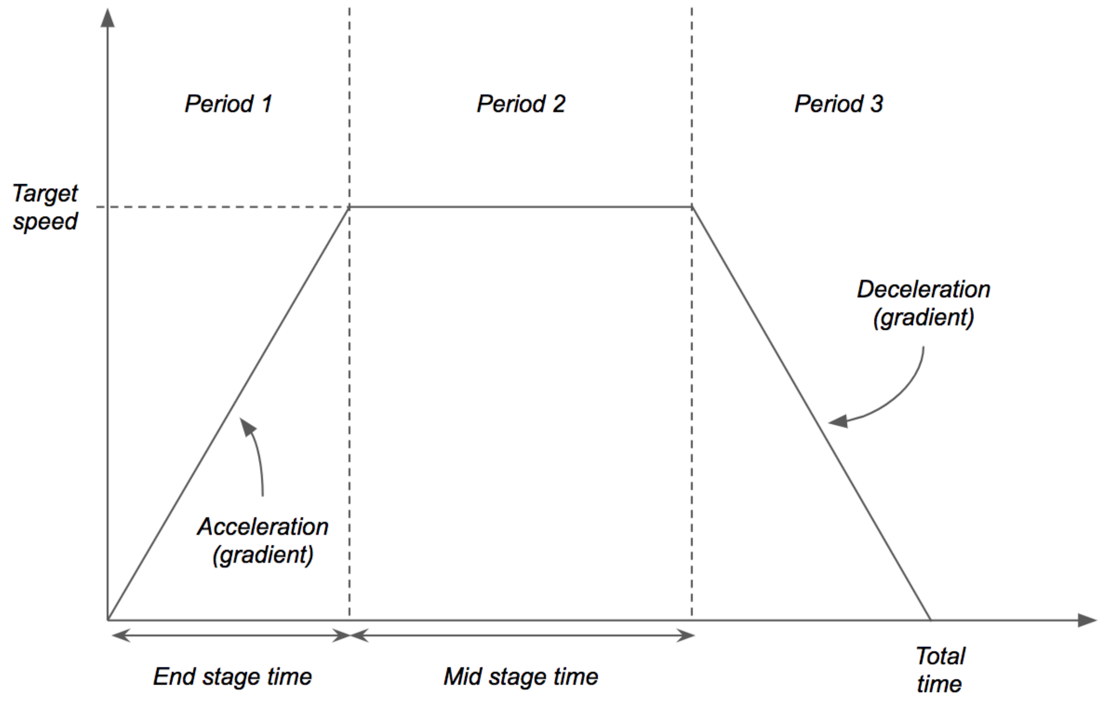
5.8.6 Trapezium Velocity Profile

The velocity profile applied is the industry standard trapezium profile. The acceleration period is mirrored to create the deceleration period. The middle section is scaled accordingly; keeping the end-effector at a constant speed. The profile is modelled around two main input parameters:

- The desired target speed of end-effector travel.
- The acceleration and deceleration of the end-effector.

Firstly, the path must be discretised into small, equally sized units. This `dx` value can be calculated to an ideal value: `acceleration * dt**2`.

There are two types of profile that had to be designed for. The simple case was that the distance of the travel (i.e. the length of the discretised path) is far enough for the end-effector to reach the target speed. In this case, *period 2* is scaled along the time axis accordingly.



The velocity profile is then constructed (remember that area under graph is displacement):

1. The end stage time is calculated: `target_speed / acceleration`
2. The end stage displacement is calculated: `end_stage_time * target_speed / 2`
3. The mid stage displacement is calculated: `path_length - 2 * end_stage_displacement`
4. The mid stage time is calculated: `mid_stage_displacement / target_speed`
5. The total time is calculated: `end_stage_time * 2 + mid_stage_time`

A time list from `0` to `total_time` in steps of `dt` are created.

Note: The `dt` variable is fixed by the control loop running the libfranka control loop. Therefore, it cannot be changed locally from `0.05` seconds in this trajectory generator.

A list of speeds is then created using this time list and the parameters calculated earlier (where `c` is the calculated intercept for the deceleration period):

```
        # sample speed graph to create list to go with time list
        speed_values = []
        c = (0 - (-acc) * time_list[-1])
        for t in time_list:
            if t <= end_stage_t:
                # acceleration period
                speed_values.append(acc * t)

            elif t >= end_stage_t + mid_stage_t:
                # deceleration stage
                speed_values.append(-acc * t + c)

            elif t > end_stage_t:
                # constant speed at target speed
                speed_values.append(target_speed)
```

The original discretised path list is now sampled using the speed list. For each speed, the corresponding number of samples in the path list is calculated (`speed_value * dt / dx`) and these sampled points are stored in a new list which can be sent directly to `franka_controller_sub`.

In another circumstance, the length of the path may be calculated to be to short to allow the end-effector to reach the target speed. As a result, the target speed is simply scaled down until the condition is met, allowing for a smaller triangular profile to be used.

The new target speed is calculated as:

$$targetSpeed = \sqrt{pathLength * acceleration}$$

5.8.7 Limitations

Feedback

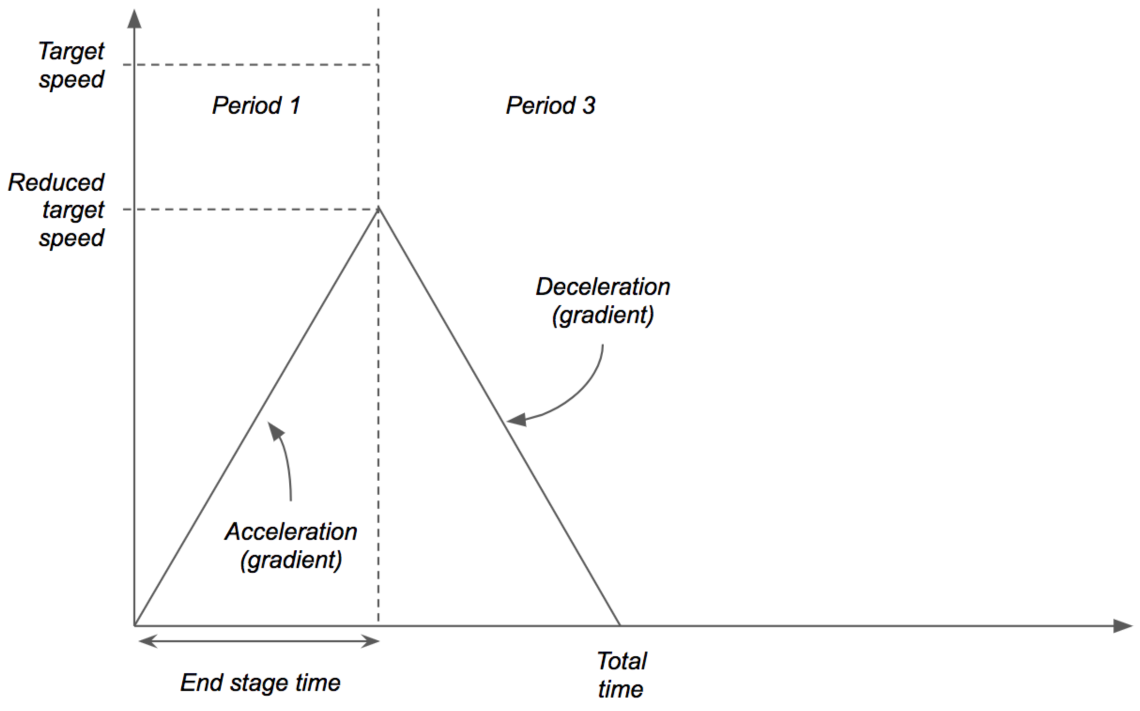
Unfortunately feedback control was not implemented to ensure the robot was moving accurately and successfully picking up pieces. However, it was found that FRANKA was accurate enough that it was not required for this project.

Smoothing

Smoothing was removed from the final version since the tortoise-hare method produced a path with uneven discretisation which prevented the velocity profile from successfully being applied.

Error in velocity profile

There is a small error in the velocity sampling when transitioning between acceleration/deceleration periods to the mid stage periods. This could be resolved in a future update by reducing the acceleration by a small amount so that the discretisation is in multiples of `dt`.



5.8.8 Implementation

Example usage:

```
# Create ROS node for this project runtime
  rospy.init_node('chess_project_node', anonymous=True)

# Create an object of the Franka control class
arm = FrankaRos()

# Create a planner object for executing chess moves
planner = MotionPlanner(visual=False, debug=True)

# LATER ON IN THE CODE...

# msg = [('n', 'h1g3')]  # example of chess move
planner.input_chess_move(arm, msg)
```

Documentation:

class `motion.`**`MotionPlanner`**(*visual=False*, *debug=False*)

`an_to_coordinates`(*chess_move_an*)

Converts chess algebraic notation into real world `x, y, z` coordinates.

Parameters **`chess_move_an`** – In form `[('p','a2a4')]` or `[('n','a4'),('p','a2a4')]`.

Returns Tuple of coordinate start(1), goal(2), died(3).

`apply_trapezoid_vel`(*path*, *acceleration=0.02*, *max_speed=0.8*)
Takes a path (currently only a start/end point (straight line), and returns a discretised trajectory of the path controlled by a trapezium velocity profile generated by the input parameters.

Parameters

- **`path`** – List of two points in 3D space.
- **`acceleration`** – Acceleration and deceleration of trapezium profile.
- **`max_speed`** – Target maximum speed of the trapezium profile.

Returns Trajectory as numpy array.

`static discretise`(*point_1*, *point_2*, *dx*)
Takes a straight line and divides it into smaller defined length segments.

Parameters

- **`point_1`** – First point in 3D space
- **`point_2`** – Second point in 3D space
- **`dx`** – Distance between points in discretised line.

Returns Numpy array of discretised line.

`discretise_path`(*move*, *dx*)
Discretise a moves path using object defined dx for unit.

Parameters

- **`move`** – List of points path goes through.
- **`dx`** – Displacement between two points on the target discretised path.

Returns Discretised path as numpy array.

`generate_moves`(*chess_move_an*, *franka*)
Generates a number of segments each a straight line path that will execute the move generated by the chess engine.

Parameters

- **`chess_move_an`** – Takes chess move in algebraic notation from chess engine
- **`franka`** – Takes franka control object as argument to find current position

Returns a list of lists that depict the full start to goal trajectory of each segment

`input_chess_move`(*arm_object*, *chess_move_an*)
After new move is fetched from the game engine, the result is passed to this function so that a trajectory may be generated. This is then executed on FRANKA.

Parameters

- **`arm_object`** – Takes object of Franka arm control class.
- **`chess_move_an`** – Takes chess move generated from chess engine.

`static length_of_path`(*path*)
Calculates the length of a path stored as an array of (n x 3).

Parameters **`path`** – List (length n) of list (length 3) points.

Returns The total length of the path in 3D space.

static smooth_corners(*path*, *size_of_corner*, *passes*)

Takes a discretised path and and rounds the corners using parameters passed into function call. Minimum number of passes is 1, which results in a chamfer.

Note This function is not currently used due to its error in smoothing the corners. It has not been implemented until a better version can be written.

5.9 Controller

The controller interfaces the different modules and in this way ensures the flow of the chess game. It handles the whole process from setting up the game and calibrating FRANKA through to to playing the actual game. The board is empty when main.py is started. The user is subsequently required to populate the board and the game is started. After having made a move, the user presses a keyboard button to trigger the chess clock. The move is detected and passed to Sunfish, which replies with the optimal next move. A motion plan for the move is generated and executed by the robot.

Note: Automatic calibration and the chess clock integration were not finished

5.9.1 Main.py File

Below is the specification for the modules of the Chess Project Python Program. This is to ensure that each module fulfills its function and that the overall goal of the project is achieved. Each module can then seperately be improved and updated, assuming compliance with this specification.

Clock module:

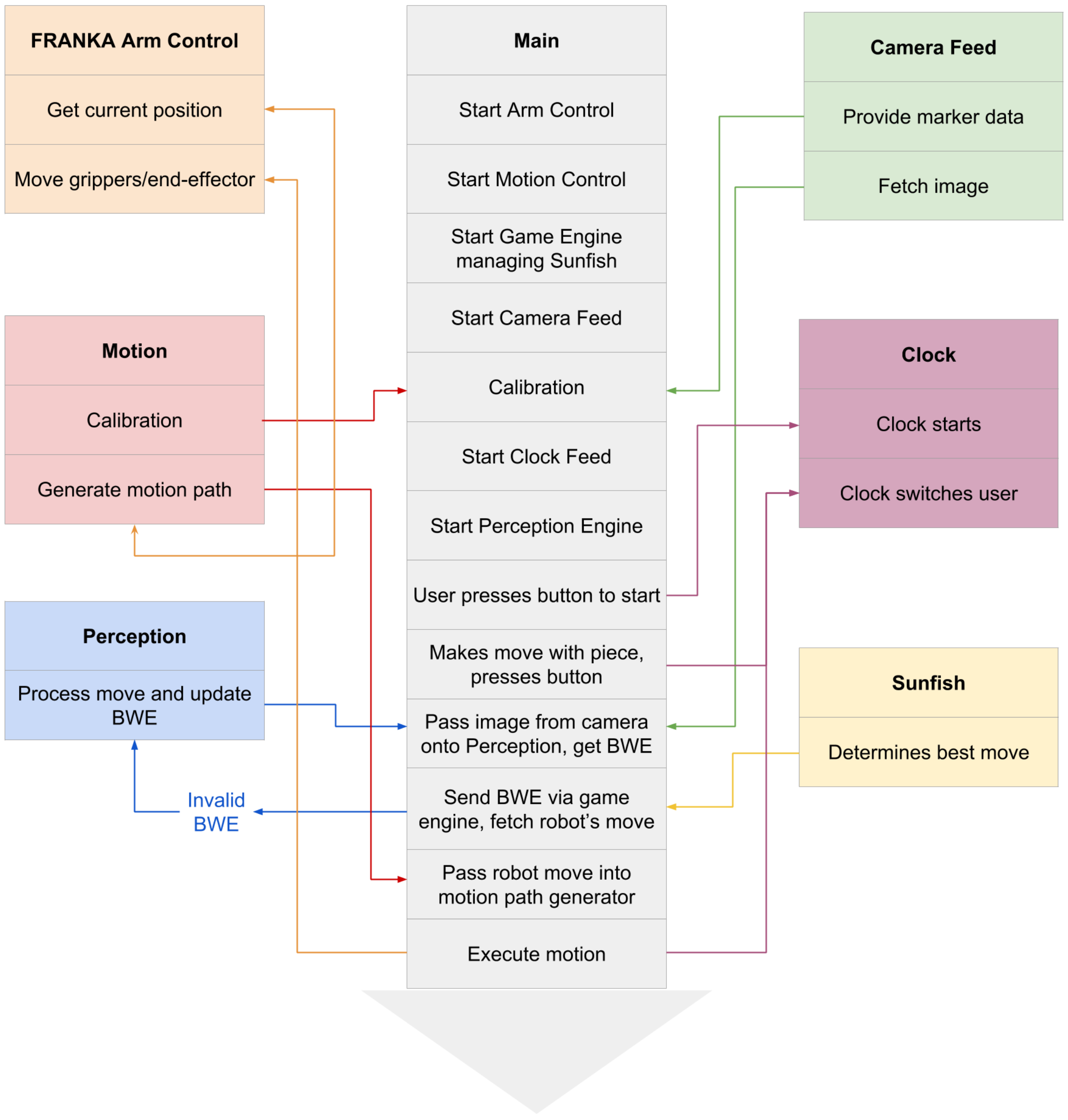
- Inputs:
  - Initiate module
  - Button press (type: mouse press)
- Outputs:
  - Up to date time for each player sent to display

Perception module:

- Inputs:
  - Initiate module
  - Camera feed (internal to module)
  - Global coordinates
  - Game Engine > Action command object:
    - Converts and saves all locations to real-world coordinates
- Outputs:
  - BWE Matrix (type: numpy array)
  - Synchronisation position of End effector?

Game Engine module:

- Inputs:


FRANKA Arm Control
Get current position
Move grippers/end-effector
Motion
Calibration
Generate motion path
Perception
Process move and update BWE
Invalid BWE
Main
Start Arm Control
Start Motion Control
Start Game Engine managing Sunfish
Start Camera Feed
Calibration
Start Clock Feed
Start Perception Engine
User presses button to start
Makes move with piece, presses button
Pass image from camera onto Perception, get BWE
Send BWE via game engine, fetch robot's move
Pass robot move into motion path generator
Execute motion
Camera Feed
Provide marker data
Fetch image
Clock
Clock starts
Clock switches user
Sunfish
Determines best move

  - BWE Matrix (type: numpy array)
- Outputs:
  - Action command object:
    * Stores whether or not there is a death first
    * Stores the piece type of death
    * Stores location of death
    * Stores move piece type
    * Stores action-move (piece type)
    * Stores action-move-from (location)
    * Stores action-move-to (location)

Motion module:

- Inputs:
  - Inner corner xyz locations from the perception module
  - Action command object:
  - Whether or not there is a death first
  - The piece type of death
  - Location of death
  - Move piece type
  - Action-move (piece type)
  - Action-move-from (location)
  - Action-move-to (location)
- Outputs:
  - Trajectory and gripping commands

5.10 Project Proposal

5.10.1 CHESS - Robotic Arm Playing Board Games

The goal of this project is to program a robotic arm to play board games such as chess. The robot used will most likely be the FRANKA Emika robotic arm (nicknamed "Panda") as it is more precise in positioning and grasping than DE NIRO. The outcome of the project will be demonstrated by performing a fully autonomous game of chess between a person and the robot. Perception will need to be added such as an RGB-D camera, a webcam, or a laser scanner.

You can see what the robot looks like here: http://www.imperial.ac.uk/robot-intelligence/robots/franka-emika/

The project includes:

1. **Forward and Inverse Kinematics:** To use the 7-dof kinematic model for Cartesian-to-Joint space mapping.
2. **Redundancy Resolution:** To resolve the 7-dof arm configuration while following a 6-dof hand pose trajectory.
3. **Perception:** Adding additional sensor for perceiving the environment and all objects of interest. It could be an RGB-D camera, a webcam, or a laser scanner.

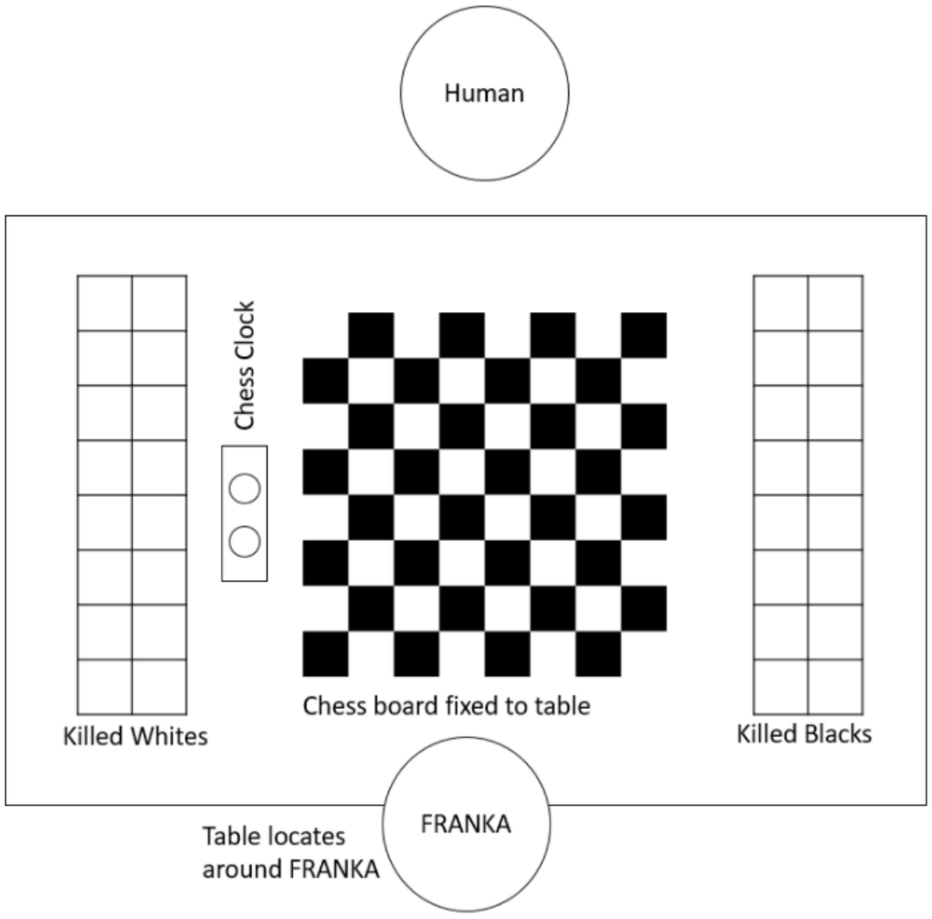
4. **Object detection and recognition:** To identify the chess pieces on the board as targets for the pick-and-place operations.
5. **Chess playing software:** Finding a suitable implementation of chess playing algorithm and integrating it within the project.
6. **Motion Planning:** Using OMPL/MoveIt or other motion planning libraries to generate viable collision-free trajectories for executing the movements.
7. **Motion Control:** To tune a Cartesian impedance controller or other for fast and smooth motion of the arm.
8. **Hand control:** To control the grasping with the 2 fingers of the hand.

Equipment: FRANKA Emika (Panda), RGB-D camera, laser scanner, chess board and pieces

5.11 Project Plan

5.11.1 Requirements

- **Sketch of a scenario:**



- **Flow chart of the sequence:**

1. Starting the game
    a. Set up game pieces
2. Playing the game
    a. Completing a move
    b. Timing the move
    c. Removing pieces
    d. Replacing pieces
3. Ending the game
    a. Relocating pieces to initial positions

5.11.2 Resources

- **Confirmed:**
    - FRANCA
    - Chessboard
    - Chess Pieces (custom to start with)
    - Camera (RGB-D, MS Kinect?)
    - Chess clock
- **Maybe:**
    - Sensors for end-effector (force detection, IMU?)
    - Customised chess board

5.11.3 Work Breakdown Structure

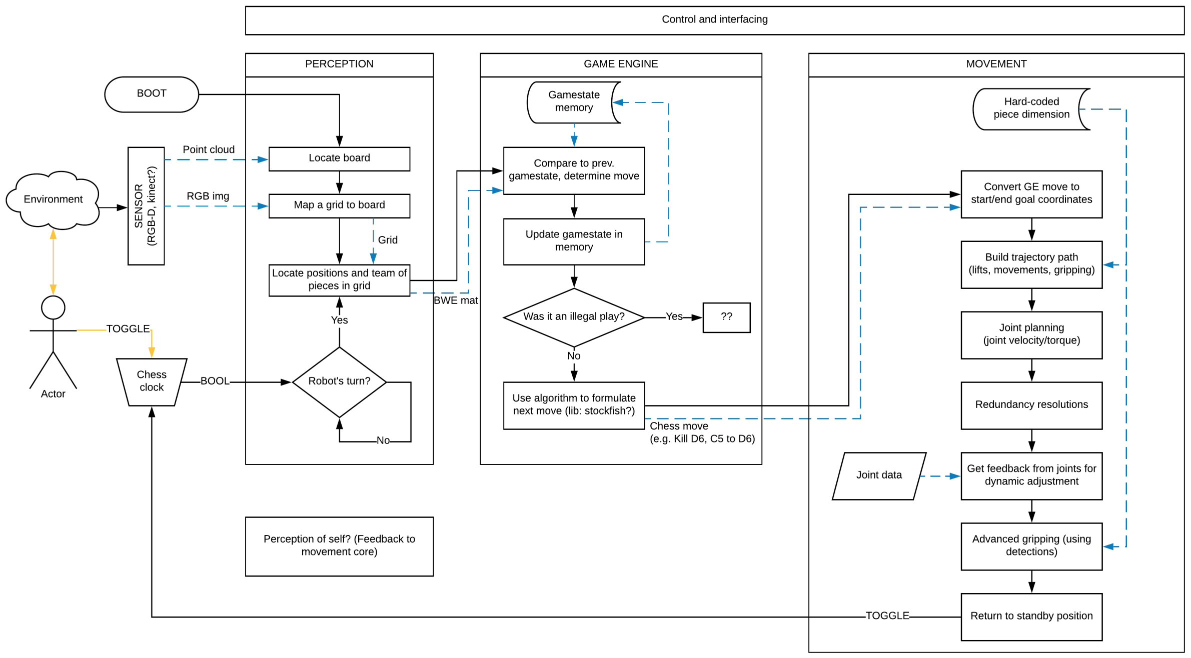
Each person's key areas of interest and work division and summarised in the table below:

Area of work	Starting teams (lead in bold)	Change in teams
Documentation	**Ben**	Ben
Physical build	**Sanish**, Paolo, Anna	*n/a - (expected completed first)*
Perception	**Leah**, Paolo	Leah, Paolo
Game Engine	**Josephine**, Ben	*n/a - (expected completed first)*
Movement	**Anna**, Sanish, Sylvia	Josephine, Anna, Sanish, Sylvia
Interface/Controller*	Ben	Ben

**implemented later on*

5.11.4 Gantt Chart

The Gantt chart is dynamic and can be viewed here: https://docs.google.com/spreadsheets/d/1-BJZdqYe8wWnCwUJs8sGYKyAoFGYXU7qOBCCHt1B_Ek/edit?usp=sharing


Control and interfacing
PERCEPTION
GAME ENGINE
MOVEMENT
BOOT
Point cloud
RGB img
Environment
SENSOR (RGB-D, kinect?)
Locate board
Map a grid to board
Grid
Locate positions and team of pieces in grid
BWE mat
Yes
Actor
TOGGLE
Chess clock
BOOL
Robot's turn?
No
Gamestate memory
Compare to prev. gamestate, determine move
Update gamestate in memory
Was it an illegal play?
Yes
??
No
Use algorithm to formulate next move (lib: stockfish?)
Chess move (e.g. Kill D6, C5 to D6)
Hard-coded piece dimension
Convert GE move to start/end goal coordinates
Build trajectory path (lifts, movements, gripping)
Joint planning (joint velocity/torque)
Redundancy resolutions
Joint data
Get feedback from joints for dynamic adjustment
Advanced gripping (using detections)
Return to standby position
TOGGLE
Perception of self? (Feedback to movement core)

5.11.5 Implementation

The hardware/software/component approaches to this project will be developed out further with modifications to this system diagram. This will come over the next few weeks.

5.11.6 Expected Difficulties

Difficulty	Mitigation strategy
Gripping outsized pieces (e.g. knight)	Standardised grip geometry at the bottom of each piece
Positional drift of chess pieces	Perception (active) and gripping (passive) system that can mitigate against drift
Large RGB-D tolerances leading to board misalignment	Maintain fixed board location
Integrating the game engine with the motion control	Dedicated project director and clear code structure
Reliably identifying board in a darker environment	Set up lighting rig to illuminate area
Grid assignment misaligning on boot	Maintain fixed board location
User does not remember to press chess clock	Time-out beeper, game manual

5.11.7 Risk Assessment & Safety Plan

In the FRANKA user manual, pages 62-86 describe an extensive risk assessment.

5.12 Helpful Resources

5.12.1 GitHub and Git

- https://guides.github.com
- Git basics: https://git-scm.com/book/en/v2/Getting-Started-Git-Basics
- The super simple beginners guide to Git: http://rogerdudler.github.io/git-guide/
- Understanding the workflow of git version control: https://www.git-tower.com/learn/cheat-sheets/vcs-workflow
- Guidance to git commands you may need in the command line: https://www.git-tower.com/blog/git-cheat-sheet/

5.12.2 Python

Structuring large Python projects

- Python documentation on what package, module, script are: https://docs.python.org/3/tutorial/modules.html
- Simple example of this structure and how it is documented: https://github.com/brandon-rhodes/sphinx-tutorial/blob/master/triangle-project/trianglelib

Writing Code: Python Conventions & Documentation

- Overall guide to documentation in Python http://docs.python-guide.org/en/latest/writing/documentation/
- The PEP8 on writing your code keeping to convention (supported by PyCharm): https://www.python.org/dev/peps/pep-0008/
- The PEP257 on documenting your code: https://www.python.org/dev/peps/pep-0257/
- Handling errors with exceptions and raising errors: https://docs.python.org/2/tutorial/errors.html

Differences between Python 2 and Python 3

- http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
- Porting code from Python 2 to Python 3: https://docs.python.org/3/howto/pyporting.html

Python commands

- `pyclean .` will clean the current directory of _pycache_ and .pyc

5.12.3 reStructuredText & Sphinx Markup

The documentation for this project is written in reStructuredText format. To assist this, the following resources should be acknowledged:

- RST Cheatsheet: https://github.com/ralsina/rst-cheatsheet/blob/master/rst-cheatsheet.rst
- RST Reference: http://docutils.sourceforge.net/docs/user/rst/quickref.html
- RST Directives: http://docutils.sourceforge.net/docs/ref/rst/directives.html
- Sphinx's RST Primer: http://www.sphinx-doc.org/en/stable/rest.html

- Official Sphinx Markup Constructs: http://www.sphinx-doc.org/en/stable/markup/

5.12.4 Sphinx & Read The Docs

The following resources were useful in the setting up of Sphinx and Read The Docs for this project:

- Screencast: https://www.youtube.com/watch?v=oJsUvBQyHBs
- RTD Guide: https://docs.readthedocs.io/en/latest/getting_started.html
- Documenting a project with Sphinx (lecture): https://www.youtube.com/watch?v=QNHM7q2hLh8

5.12.5 Getting Started with ROS

To 'get started' with learning ros, you may find doing the following helps you to understand ROS better:

1. In your home directory, ensure you have set up a complete catkin workspace.
2. Within that workspace, create a catkin package.

5.13 DE Programming Ground Rules

1. **Always put code on GitHub**, never use Google Drive or direct messaging.

1. This means when working on a particular function, you will be working in a smaller team and contributing to a particular branch you set up together.
2. Working on a branch in a small team (maybe 2-3) requires coordination so that you don't create conflicts when pushing your changes to the branch.
3. Remember to push changes to a branch, AND sync!

2. **When writing Python, keep to the** PEP8 Python styling guide.

   This style guide is usually enabled in PyCharm to help you. This means you get a squiggle and a yellow/brown mark next to a violation (settings are in `Preferences > Editor > Inspections > Python`). The key takeaways are:

1. Naming conventions using: ClassNames, function_names, variable_names
2. White space (PyCharm will give you help here)
3. Tabs vs Space, PyCharm actually converts the tab key, and 4 spaces, into the same thing by default (so just don't change this from default values, and you can use either).

3. **Comment your code! Plus see bullet 4.**

1. This can be in two main forms, inline or block. Again, keep to PEP8 to avoid differences between code. It's the simplest one to keep to.

4. **Document your code!**

   Use the PEP257 docstring convention and keep to it well - you can read the short version PEP257, or the complete PEP257.

1. This means that Sphinx (an automatic documentation generator) will work and display things properly.
2. An example of this can be seen on GitHub. It has library, modules, and comments.

5. UML Diagrams. . . USE THEM

6. Handle errors properly with exceptions. If necessary, build your own errors for safety measures and use try..except..raise blocks in your code.
7. Global variables are bad. Functions are better than types. Objects are likely to be better than complex data structures.
8. Avoid `from X import *` as much as possible. It's much better to import a specific function `from X import Y` or to use it in context of your code `import X; X.Y(arg)`. Otherwise we will need to start handling lists of import names.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index