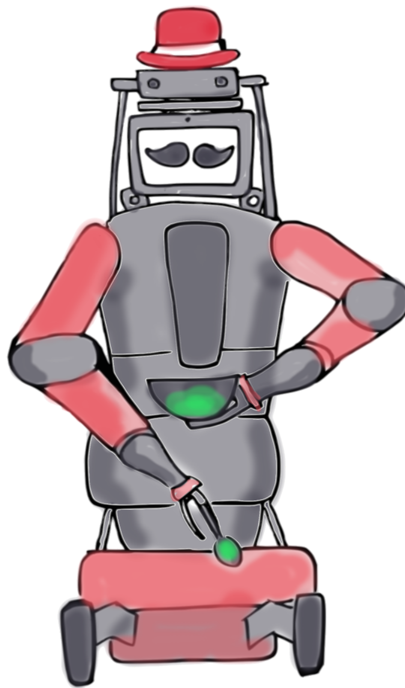


Feeding Robot Engineering Design

FRED

Ina Roll Backe
Grace Chin
Lidia Dynes Martinez
Jacob Mitchell
Matilda Supple

Contents

1 Introduction 3
1.1 Problem Definition 3
1.2 Market Analysis 3

2 Project Management 4
2.1 Roles 4
2.2 Gantt Chart 4
2.3 Risk Assessment & Safety Plan 5

3 Technical Implementation 6
3.1 High Level Overview 6
3.1.1 ROS 6
3.1.2 Feeding Process 6
3.1.3 File Directory 7
3.2 Cameras 7
3.2.1 Hand Camera 8
3.2.2 Kinect Camera 8
3.3 Object Detection 9
3.3.1 Subsystem Design and Analysis 9
3.3.2 Implementation 9
3.3.3 Issues 9
3.4 Facial Analysis 11
3.4.1 Subsystem design and analysis 11
3.4.2 Implementation 11
3.5 Publishing Topics 14
3.6 Calibration 14
3.6.1 Calibration methods 14
3.6.2 Calibration Step-by-step 16
3.7 Moving Baxter 18
3.7.1 Control Baxter Robot 18
Receives Baxter Information 19
Control Limbs 20
Control Grippers 20
Playback Recorded Trajectory 20
3.7.2 Motion Planning 21
3.8 FRED Feeding 22
3.8.1 Feeding Flowchart 22
3.8.2 Safety 23

4 Demonstrations 23
4.1 Scenario 23

4.2 Final Demonstration Step-by-step 24
4.2.1 Food Detection Demo 25
4.2.2 Baxter Moving to Mouth Demo 25
4.3 Video Guide 30
4.4 Safety 30

5 Conclusion 30

6 References 30

7 Appendix 31
7.1 FRANKA 31
7.1.1 Initial Test Using FRANKA Interface 31
7.1.2 Camera and FRANKA Calibration 31
7.1.2.1 Calibration Procedure 31
7.1.2.2 Calibration Development 32
7.1.3 Control System 33
7.1.4 Physical Component 33
7.2 Cameras 34
7.2.1 Astra Installation 34

1 Introduction

1.1 Problem Definition

The goal of this project was initially to program the FRANKA Emika robotic arm (nicknamed "Panda") to detect and pick up food and then feed a person. This was then changed to the Baxter Robot. The outcome of the project will be demonstrated by performing the whole scenario fully autonomously, including the perception of the person, the person's mouth, etc.

1.2 Market Analysis

The target market for this project would be people who are unable to feed themselves using their arms. Similarly to existing products such as Obi (Figure 1), this device would be useful for individuals lacking upper extremity motor control, such as those with ALS, Cerebral Palsy or Parkinson's disease [1]. In order to operate the device the user must be able to chew and swallow freely as well as have the cognitive capability to operate a simple device. It could also be used by amputees or people with phocomelia syndrome. To control the device the commands should therefore be handsfree, for example, facial gestures or voice control.

Figure 1: Obi Feeding Robot

2 Project Management

2.1 Roles

Below shows what different people in the team mainly worked on. Because everyone helped each other during the project and some jobs could not have been done alone, below does not accurately reflect what everyone did.

Ina Roll Backe	Camera installations, object detection, supply physical components, video recording, report & presentation formatting
Grace Chin	Robot control, calibration, motion planning, video recording
Lidia Dynes Martinez	Feeding logic flow, motion planning
Jacob Mitchell	Camera installations, facial detection, mouth detection, video recording
Matilda Supple	Object detection, supply physical components, presentation, report & presentation formatting, video recording

2.2 Gantt Chart

	Term 2										EASTER
	2	3	4	5	6*	7	8	9	10”	11^	1
Planning the project (sensors, research, dividing roles)											
Facial Detection											
Object Detection											
Control											
Motion & motion planning											
Implementing robot sections together											
Refining & debugging											
Report writing											
Video editing											

* = Interim review demo; ” = Live demo; ^ = Report, documentation & source code

The reason for many activities occuring in week 10 is due to the following big fallbacks:

- change in robot from FRANKA to Baxter in the last week before the Live Demo
- change in camera from Astra to Kinect camera the day before the demo

The changes are described in more detail in the Technical Implementation section and the previous work done on FRANKA and the RGBD camera are recorded in the Appendix.

2.3 Risk Assessment & Safety Plan

The robot should stop and start with the given signals of the user. Other safety measures are described in the risk assessment.

Risk	*Cause*	*Severity*	*Mitigation*
Injure user	Dangerous coding	High	Implement preventative measures in code
Water Damage equipment	Spilled liquids in lab	High	Avoid liquids in lab
Hits surrounding viewers	Audience is too close to the robot	High	Ensure viewers are a safe distance away from the robot
Incomplete Project	Bad project management High task difficulty	High	Plan project carefully Seek out help if required
Robot breaks hindering testing	Accidents	Mid	Responsible use of robot, ask for help if you are unsure
Poke the user's eye	False trajectory	Mid	User wears safety goggles
Throwing of items	Insufficient gripping force whilst moving	High	Ensure all items are light and will not make an impact
Team members get sick	Illness	Mid	Regular meetings and good internal communication

3 Technical Implementation

3.1 High Level Overview

3.1.1 ROS

The system is built upon Robot Operating System (ROS).

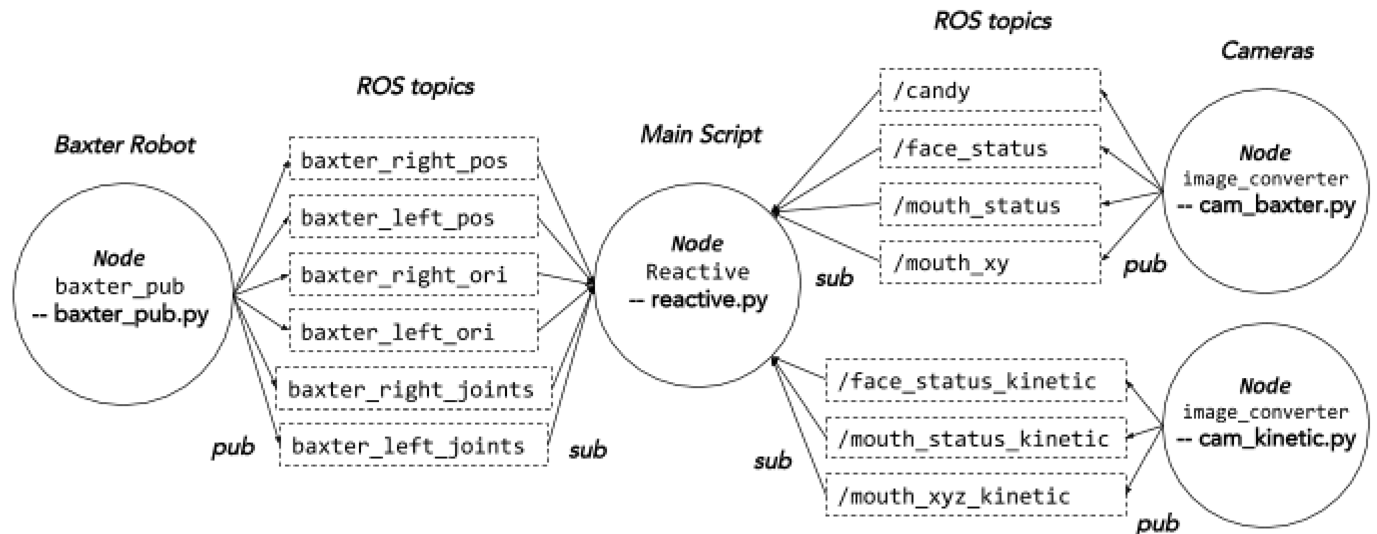


Figure 2 : ROS System Diagram

3.1.2 Feeding Process

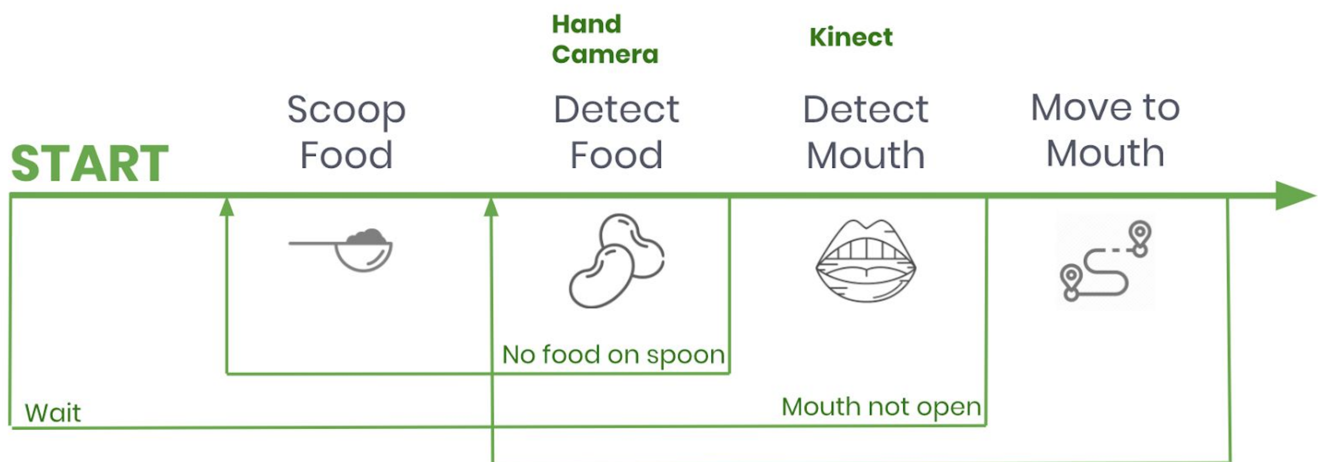


Figure 3 : Process diagram

3.1.3 File Directory

This report refers to different files in our source code. The source code should be placed in the catkin_ws of the lab desktop. For the reader's ease of understanding, here is how the source code was organised from **/catkin_ws**:

```
/catkin_ws   /src   /fred              /src   /other

             ...    /trac_ik_python           calibration_roni.py
                    ....                      inverse_kinematic.py
                                              joint_playback.py
                                              joint_position_printer.py
                                              move_joints_to.py
                                              message_filters
                                              rospkg
                                              rospy

                                              /perception

                                              baxter_control.py
                                              baxter_pub.py
                                              calibration.py
                                              cam_kinetic.py
                                              cam_baxter.py
                                              overall_control.py
                                              perception_sub.py
                                              gripping.rec
                                              scooping.rec
```

3.2 Cameras

Two main cameras were used in the setup; one RGB camera embedded into the hand of the Baxter robot and one externally placed RGB-D Kinect camera. To connect to a camera RGB and depth video stream within a python script, you need to subscribe to the camera topics.

To access the video stream for analysis with opencv, a package called cv_bridge is required to convert the image subscription to an image format. Cv_bridge needs to be defined in the init function to be used later.

```
#initialise bridge:
self.bridge = CvBridge()

# in main code:
```

```
try:
  cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
except CvBridgeError as e:
  print(e)
```

Now the image is in RGB format that OpenCV can understand.

3.2.1 Hand Camera

The hand camera is part of the baxter ros environment. Therefore to access the topic, you must run the python code within catkin and launch baxter:

1. `$ cd catkin_ws`
2. `$ bash baxter.sh`

Because the hand camera has only one available topic (RGB), a simple rospy subscription can be used.

```
self.image_sub = rospy.Subscriber("/cameras/right_hand_camera/image",Image,self.callback)
```

The subscriber requires a callback function to send the subscribed video stream to.

3.2.2 Kinect Camera

To access the kinect camera, first the driver needs to be installed along with openni.

In order to access the RGB and depth camera streams at the same time, a function called approximate time synchroniser is required. This bundles both video streams together even if they have different time stamps and sends them to the determined function together.

```
# initialise subscribers:
image_sub = Subscriber("/camera/rgb/image_rect_color",Image)
depth_sub = Subscriber("/camera/depth_registered/image_raw", Image)

# bundle together subscribers
tss = message_filters.ApproximateTimeSynchronizer([image_sub, depth_sub],queue_size=10,
slop=0.5)

# hit up the callback function with both images
tss.registerCallback(self.callback)
```

3.3 Object Detection

3.3.1 Subsystem Design and Analysis

The object detection was primarily designed to get the the coordinates and direction of the candy pieces for the end effector to pick up. This was later replaced by a prerecorded scooping/picking up motion. Hence, the object detection was used to make sure that the robot had candy on its spoon after attempting to scoop/pick it up.

3.3.2 Implementation

A python script was written to process the frames from the RGB D camera to complete the object detection. The first section of the script pre-processes the frames by applying a Gaussian smoothing, an hsv conversion and then a green colour threshold. The resulting mask is shown in figure 4.
The script then uses `cv2.findContours` to identify the contours of the white sections of the mask. These contours are then processed in a for loop. The centre of each contour is then found and labelled using the `cv2.moments`. When working with Franka the aim was to use the gripper to pick up the candy. To make sure it was in the correct orientation to pick it up, a line through the centre of the candy was created also using `cv2.moments` and the the contours, lines, centres, and text were drawn on the initial frame and displayed (figure 5).

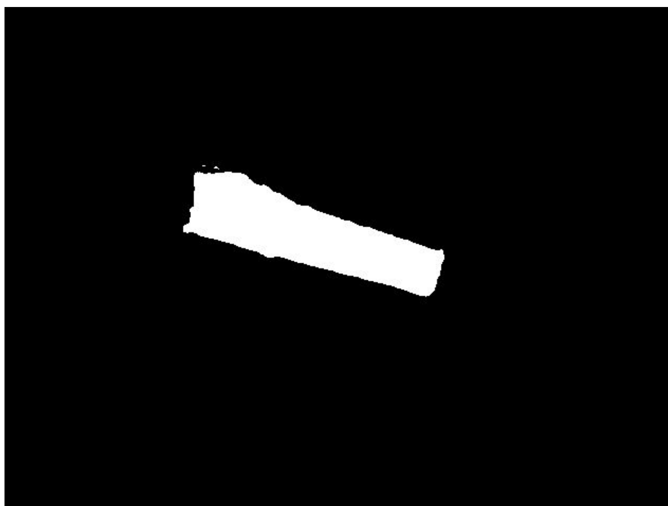


Figure 4: Mask

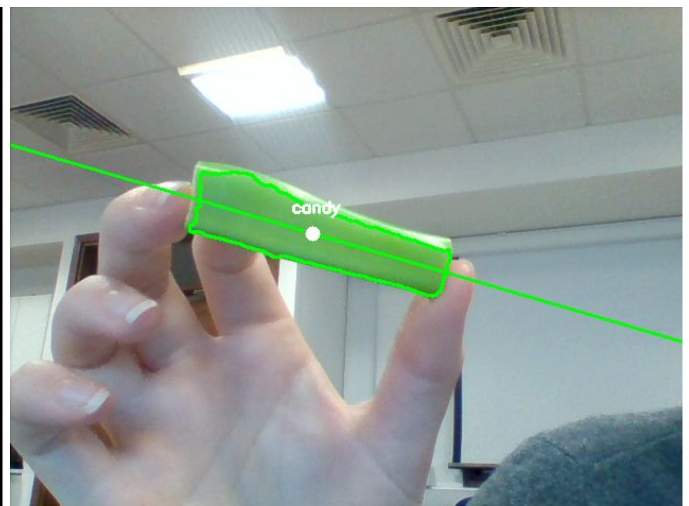


Figure 5: Initial Output

When changing robots this method was discarded and instead a spoon was used to scoop the candy. This code was then used to detect whether or not there was candy in the spoon.

3.3.3 Issues

An issue encountered was that when detecting the contours, due to different lighting the script would pick up small pixels of green on the outlines of the candy and treat them as separate pieces of candy (figure 6). This was negated by finding the area of each contour and discarding contours that are smaller than the specified area threshold.

Figure 6: Pixels of green

Additionally, when changing to the built in camera within the Baxter arm the colors were not as vivid due to the lower quality of the camera. The hsv threshold had to be finely tuned to the new colours of the candy and the surrounding conditions. To get the threshold just right, the mask was put on the original image, to see exactly which colours were being discarded and not.

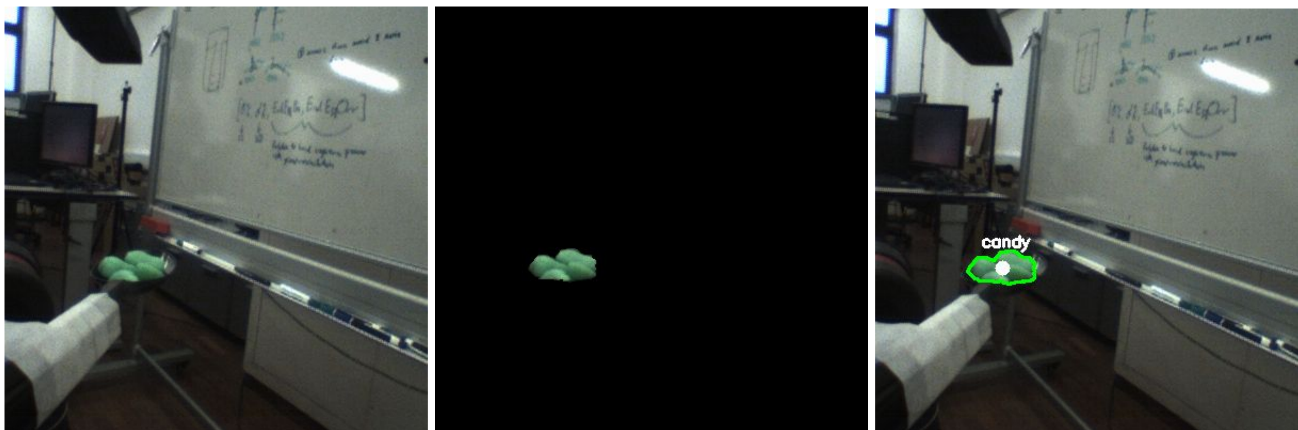


Figure 7: Steps of Final Object Detection

The object detection returns a `candy_state` of `True` or `False`. To make the system more robust, a counter for the status was implemented. Only when the code has picked up five counts of candy, will it actually publish a `candy_state = 'True'`.

3.4 Facial Analysis

3.4.1 Subsystem design and analysis

Facial detection is a key component of the feeding project. The system needed to analyse a live video feed and return the location of the center point of the user mouth.

3.4.2 Implementation

Detailed facial feature detection was used to output the X, Y and Z coordinates of the center of the mouth. The python packages used for this task were OpenCV and Dlib. A useful resource for installing Dlib and running feature detection can be found here. Instructions for installing OpenCV can be found here.

The mouth detection can be split up into 5 Sections:

1. Face detection - localising face in an image
2. Facial landmark detection - detect key facial features within face ROI
3. Mouth center - locate center point coordinates of mouth using landmark points
4. Depth - determine depth value using the Depth
5. Mouth state - Determine the state of the mouth (open/closed)

1. Face detection

Face detection was achieved using the built in dlib function `get_frontal_face_detector()`. This function takes an input cv image and outputs a list of rectangular ROIs that contain faces identified in an image.

2. Facial landmark detection

Landmark detection used another built in dlib function called `shape_predictor()`. This function takes the image and the ROI and outputs a list of facial landmarks with corresponding X and Y pixel coordinates. Figure 8 shows the numbering system of the facial landmarks.

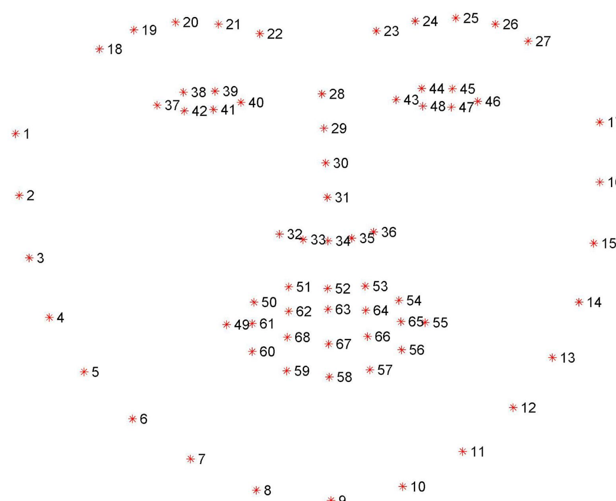


Figure 8: Numbering System of Facial Landmarks

FRED

3. Mouth center

The center of the mouth was determined as the coordinates between facial landmarks 63 and 67.

Figure 9: Feature Detection Mouth Centre

The coordinates of the mouth center was calculated using the following code:

```
mouth_center_x = mouth_bottom[0] +(mouth_top[0]-mouth_bottom[0])/2
mouth_center_y = mouth_bottom[1] +(mouth_top[1]-mouth_bottom[1])/2
```

4. Depth

To determine the depth value of the mouth coordinate, the depth image from the RGB-D camera was utilised. The depth image is converted to a black and white array with values between 0 and 255. The larger the value the further away the object. However, it must be identified that the depth readings appeared in bands. The depth reading resets to zero approximately every meter or so. This is probably some kind of error or might be something integral to depth cameras.

Figure 10: Depth Image Bands

Both the RGB and D images have the same size and are taken from virtually the same position, therefore the XY coordinates of the mouth center can be used to index the corresponding pixel depth value. Because the infrared and the RBG camera on the Kinect are slightly offset from each other, a pixel offset value for both x and y was adjusted manually.

```
mouth_center_z = depth_image[mouth_center_x-55, mouth_center_y+20]
```

5. Mouth State

A form of communication between the person and the robot is the state of the mouth. If the mouth is open, then the user is ready to receive food. If closed, then the user is not ready to receive food. A simple ratio between points of the mouth was used.

$$\text{Mouth Ratio} = (P63 - P67)/(P61 - P65)$$

If the mouth ratio is greater than 1, the mouth state is set to open.

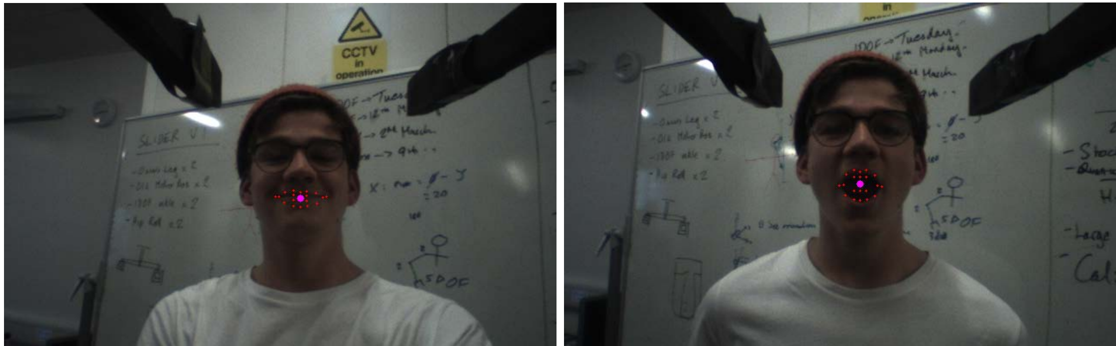


Figure 11: Closed and open mouth states (from left to right)

3.5 Publishing Topics

Sections 3.3 and 3.4 have shown how useful pieces of information can be acquired using image analysis of video stream subscriptions. This information is required by other python nodes such as for calibration and controlling baxter. To share the information between nodes the data is published to a topic with the following example code:

```
#initialise node
self.mouth_xyz_pub = rospy.Publisher("/mouth_xyz_kinetic", Point, queue_size=10)
#assign value in main code
xyz = Point()
xyz.x = mouth_center_z
xyz.y = mouth_center_x
xyz.z = mouth_center_y
#publish in main code
self.mouth_xyz_pub.publish(xyz)
```

This topic can now be subscribed to from any other python node as long as this script is running.

3.6 Calibration

Calibration is required to transform the mouth coordinates in respect of the camera, *[u, v, w]*,to the mouth coordinates in respect of the robot, *[x, y, z]*. The calibration also takes into account the extra length of the spoon.

3.6.1 Calibration methods

Two different calibration methods were considered when trying to calibrate FRANKA and Baxter robot.

Method 1: Transformation Matrix Method
The transformation matrix method, which is also described very clearly by the Chess Team, uses the following equation:

$$aX = b$$

where X is the transformation matrix, a is the input and b is the output

Two different codes (the GTA, Ronnie's, and the Chess Team's) were used to find the transformation matrix, however, unfortunately both did not work. A suggested reason is because of scaling issues: The camera takes coordinates in terms of pixels, where as the robot takes the coordinates in terms of metric units. This makes it difficult to transform from one frame of reference to another.

NEXT STEPS OPPORTUNITY ↠
Taking the coordinates of the mouth in metric units by the camera was possible using point cloud, however due to the restriction of time, this could not be managed. This would help with finding the transformation matrix, which would greatly help with the accuracy of the calibration to consider the tilts of the camera.

Method 2: Linear Regression

A more fundamental method that was used is plotting the different frame axes for the camera, *(u, v, x)* with the different frame axis for the robot *(x, y, z),* and then finding the relationship for each, as shown in Figure 12.

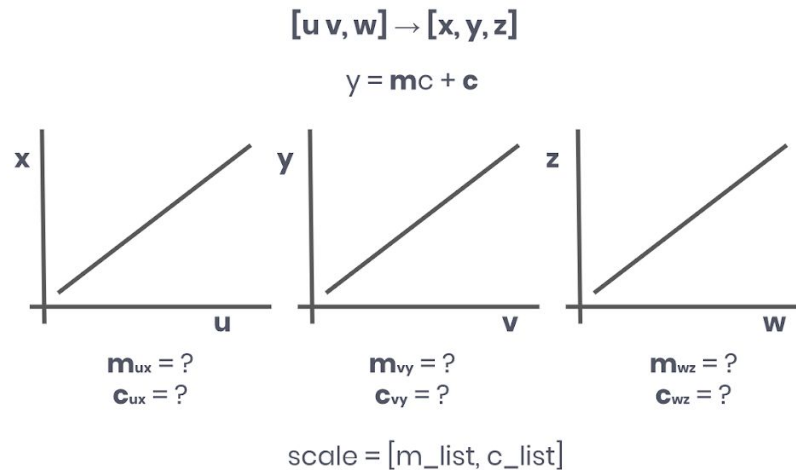


Figure 12: Image showing how the scale contains a list of m and a list of c for each axis relationship

Using the scale, the relationship between the camera frame and the robot frame is found and easily converted between each other. Although this method resolves any scaling issues, this method assumes that axes *x* and *u*, *y* and *v*, *z* and *w*, are collinear. However this is not always the case.

For Baxter robot, Method 2 was used. The code is in `calibration.py`. Because the Kinect camera used is located on Baxter robot's head, the axes of the robot and the camera were assumed to be collinear. Although the axes were collinear, manual experimentation had to be used to distinguish which camera axes was collinear with which robot axes.

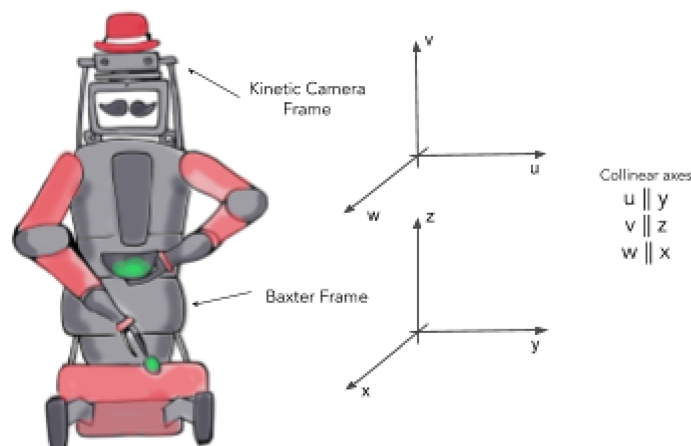


Figure 13: Diagram showing the collinear axes in the camera and Baxter robot reference frame.

3.6.2 Calibration Step-by-step

Calibration is achieved through finding coordinate points in the camera reference frame and the robots reference frame. The code for this is in `calibration.py`. This can be done through getting the mouth position and the robot's end effector position. Two people are prefered for the calibration process. The following are the steps for calibration, which is guided in the terminal Figure 14.

1. Ensure that the Baxter robot is set up, turned on, and enabled as described in this link.
2. Open the terminal in the Lab computers.
3. Execute the following in the terminal: >> `cd catkin_ws`
4. Split the terminal horizontally into four sections. This can be done by right clicking on the terminal. Each divided terminal section should now be in the catkin_ws directory.
5. For each terminal section, execute the following: `bash baxter.sh`
6. Execute the following for each of the different section terminals:
    a. Terminal 1 | This is for launching the Kinect camera. Although streams of red text might popup, as long as yellow text indicating the camera has successfully been enabled is shown, the launching has been successful.
    ```
    >> roslaunch openni2_launch openni2.launch
    ```
    b. Terminal 2 | This runs the python program that will activate detection code for the Kinect camera. Two windows showing what the camera views should come up. One is in grayscale and the other in colour. The grayscale should have a white dot on where the program detects as where the mouth is.
    ```
    >> cd src/fred/src
    >> python cam_kinetic
    ```
    c. Terminal 3 | This runs the baxter publisher which helps the BaxterControl class to receive information from Baxter. (Section 3.6.1 discusses why this needs to be done.)
    ```
    >> cd src/fred/src
    >> python baxter_pub.py
    ```
    d. Terminal 4 | This runs the main calibration program.
    ```
    >> cd src/fred/src
    >> python calibration.py
    ```
7. In the terminal where you executed `python calibration.py`, follow the instructions stated:

```
How many points would you like to calibrate with?: 5
Would you like to see current camera pos value? [Y/n]:

('Camera position:', [317.0, 173.0, 232.0])
Would you like to record this? [Y/n]:
('Camera coordinates :', [[317.0, 173.0, 232.0]])
('End effector coordinates :', [])
Would you like to see current end effector position value? [Y/n]:

('End effector position:', [0.267, -0.543, 0.347])
Would you like to record this? [Y/n]:
('Camera coordinates :', [[317.0, 173.0, 232.0]])
('End effector coordinates :', [[0.267, -0.543, 0.347]])
Would you like to see current camera pos value? [Y/n]:
```

Figure 14: Calibration Guide in Terminal

- Input the number of points you would like to record.
- Move the cardboard mouth to desired position. Record the mouth position.
- Move the robot arm end effector to the mouth. Record the end effector position.
- Repeat steps 3 & 4 until sufficient points are recorded.
- Linear regression is done with the points and a scale between the two frame of references are found. From there, test the scale and ensure sufficient accuracy. The scale contains a list of m values and a list of c values.

3.7 Moving Baxter

The control of Baxter robot was not developed extensively due to the abrupt change in choice of robot from FRANKA to the Baxter robot one week before the set live demo. The transition between robots was due to the lack of python end effector orientation control available for the FRANKA robot. Other differences between FRANKA and Baxter are summarised in the table below:

	Franka Robot	Baxter Robot
End Effector Control	absolute position (x, y, z) relative position (dx, dy, dz)	joint angle controls cartesian position (x, y, z) *Inverse kinematics, IK solver*
End Effector Info	position (x, y, z)	position (x, y, z) orientation (qx, qy, qz, qw)
Cameras	Separate RGBD camera	On board cameras: Kinect and Hand

Progress on controlling the FRANKA robot are described in the Appendices X. The website shows both information and documentation for running FRANKA and Baxter.

3.7.1 Control Baxter Robot

Baxter's Python Interface is used to control the robot. Research SDK Example Programs of how it is used are in the Desktop of the lab computers and are also in the Baxter Research Robot website.

For ease of access, a `BaxterControl` class was created in `baxter_control.py`.
`BaxterControl` has the following attributes:

`self.default_arm`
Some of the methods have the option of inputting a limb of choice. If no limb is inputted as a parameter in the methods, the default arm ("right") will be the used.

`self.ee_face_forward`
This is a list of values, which define the end effector orientation such that the end effector is in the correct scooping orientation.

`BaxterControl` has methods can be divided into the following categories:

- Receives Baxter Robot Information
- Controls Limbs
- Controls grippers
- Playback Recorded Trajectory

Receives Baxter Information

Figure 15 shows a summary of the methods in `BaxterControl` and how it gets information from **baxter_pub.py**. In **baxter_pub.py**, key informations of Baxter are formatted and published into different topics as messages. The formatted information are then accessed through subscribing to the different topics in the methods of **baxter_control.py**.

> **baxter_pub.py** must be running in a different terminal for **BaxterControl** to receive information about the robot. Run: `python baxter_pub.py`

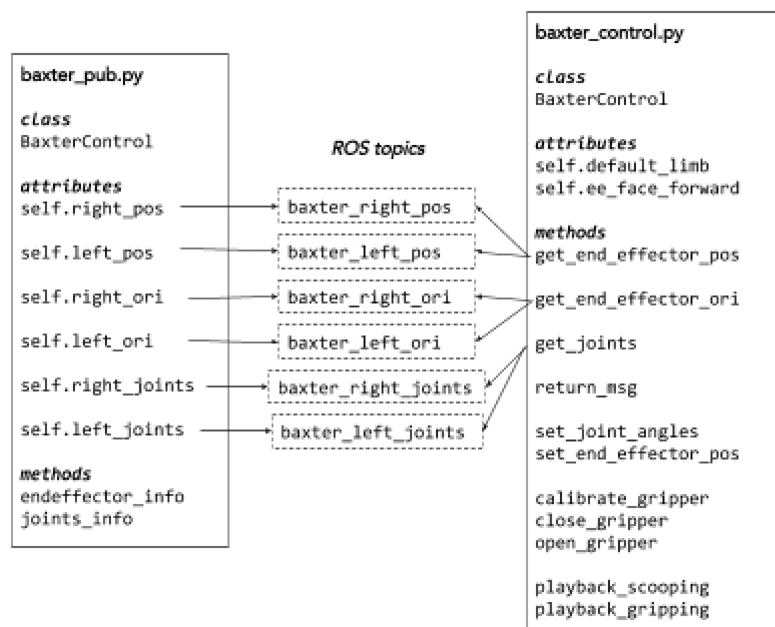


Figure 15: Summary of the methods in **BaxterControl**

The class, `BaxterControl` ,contains the following methods:

get_end_effector_pos(*limb=None*)
Gets the 3 values that define Baxter's end effector position.

get_end_effector_ori(*limb=None*)
Gets the 4 values that define Baxter's end effector orientation.

get_joints(*limb=None*)
Gets the 7 joint angles (in radians) of Baxter's defined limb.

return_msg()
Callback function of the subscriber methods to return the messages given.

Control Limbs

`set_joint_angles(joints, limb=None)`
joints = [...] ← list of 7 joint angles
Sets Baxter's limb to the given joint angles.

`set_end_effector_pos(x, y, z, qx=None, qy=None, dz=None, qw=None, limb=None)`
Sets the robot limb's end effector to a given absolute position (`x,y,z`) and orientation (`qx,qy,qz,qw`) . If the orientation is not set, the orientations defined as `self.ee_face_forward` will be used.

Setting the end effector position requires an inverse kinematics solver, which calculates solutions of possible joint angles for the robot to reach the desired end effector position. The inverse kinematic solver used is the TRAC_IK solver. To download the TRAC-IK library in the directory as shown in section *3.1.3 File Directory*, run the following in the terminal:

```
sudo apt-get install ros-kinetic-trak-ik
```

And then run the following in the correct directory:

```
git clone https://bitbucket.org/traclabs/trac_ik.git
```

Control Grippers

`calibrate_gripper(limb=None)`
Calibrates the gripper. Calibration of the gripper must be done before any movement of the gripper.

`close_gripper(limb=None)`
Closes the gripper.

`open_gripper(limb=None)`
Opens the gripper.

Playback Recorded Trajectory

To playback the trajectory, the server must be initiated.
This is done by running the following in a separate terminal:

```
rosrun baxter_interface joint_trajectory_action_server.py --mode velocity
```

`playback_scooping()`
Playbacks the recorded joint angles of the robot scooping candy.
The angles are recorded in a file called `scooping.rec` .

`playback_gripping()`

Playbacks the recorded joint angles of the robot gripping onto candy.
The angles are recorded in a file called `gripping.rec` .

The playbacks were recorded using the provided example program (**`joint_position_recorder.py`**) and the methods were adapted from the example playback program (**`joint_trajectory_file_playback.py`**).

3.7.2 Motion Planning

When controlling the end effector to reach a certain *x, y, z* coordinate, sometimes an error would occur. This is because there may not be a solution for the inverse kinematics, which means that there are no joint angles that would allow the robot's end effector reach the desired position.

To solve this, a projection of equally spaced points from the starting position to the end position would be found. The robot's arm would be programmed to attempt to reach the closest projection point. If the attempt fails, the next projection point would be given to the robot to try. This continues until the robot has finished trying all the projected points. This method is visualised in Figure 16 below.

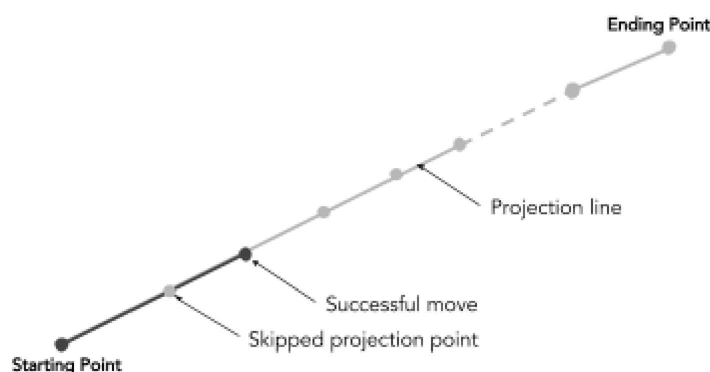


Figure 16: The diagram shows the example of the projection method being used. The dark grey show attempted points and the light grey indicate the planned line of projection and points on it.

Although Baxter's arm would successfully reach at close proximity to the desired position using this method, the movement of the arm whilst going towards the position is still uncontrolled. This means that orientation of the end effector can not be controlled whilst going from one position to another.

NEXT STEPS OPPORTUNITY ↠
There are many different joint angles that would allow the robot's end effector to reach a certain position. The robot currently would use a random motion to reach to the desired position, causing the food to drop on the way after scooping. Using motion planners, such as MoveIt, would allow more advanced trajectory control and solve this problem. However, due to time constraints, this could not be done. According to a GTA, Roni, at least two more full days will be needed for this.

Because of this lack of control, the gripper was used to grip strips of candy as oppose to scooping pieces of the candy.

3.8 FRED Feeding

3.8.1 Feeding Flowchart

Unfortunately due to time restrictions and errors in planning, the different components of our control code could not be combined together. Despite this, we had a concrete plan as to how our control code was designed to function; this is detailed below.

We set out with the aim of keeping our code as modular as possible for clarity and extensibility. From a high level behavioural standpoint, our robot had to either perform a recorded action (in order to traverse to the food) or perform reactive control (in order to reach the mouth of the subject). It also had to decide when to use which control subsystem.
For this reason, we split our logic into three classes:

- `OverallControl`: For deciding when to switch between reactive and recorded control.
- `ReactiveControl`: Code for performing reactive actions to reach the subject's mouth
- `BaxterControl`: Code for performing recorded actions to retrieve food (see section 3.6.1)

`OverallControl`

This class has the responsibility of switching between the recorded and reactive control modules based upon the current state of the robot's inputs, which were in turn retrieved via a publish subscribe queue. This class may be externally controlled through a high-level interface which contains the simple methods `begin_execution` and `stop_execution`. As their names suggest, these methods would begin or stop the robot's execution at whichever stage the robot is at; this was achieved through registering and unregistering subscriptions to the robot's state.

Once a state update was published to the queue, the `__candy_callback__` method would be called. This would decide whether or not to switch to either reactive or recorded control through the `__switch_to_reactive__` or `__switch_to_recorded__` methods respectively. These would in turn pass responsibility to `ReactiveControl` or `BaxterControl` instances in order to perform the desired control switch.

`ReactiveControl`

This class was responsible for performing reactive control through an interface consisting of the three methods `turn_on` and `turn_off`.

`turn_on` and `turn_off` would subscribe or unsubscribe from publishing queues respectively in order to gain information about the current positions of the food and open mouth; this had the effect of starting or stopping reactive control.

When updates to the food or mouth points were pushed to the queue, the current recorded positions of food and mouth would be updated, and a new trajectory and speed would be set based on these values; this is performed through a call to `__update_values__`. Assuming regular updates were pushed

to our pub-sub queue, this would allow for a smooth, continuously updating trajectory towards the subject’s open mouth which would slow down as it approached the opening to avoid injury.

3.8.2 Safety

Danger protocol: Retract its arm away from the user by approximately up to 30 cm depending on the workspace of the robot arm and awaits for danger to no longer be there before continuing.

Neutral protocol: A default position will be set for the robot to return to.

<table>
<tr><th>Possible unexpected events</th><th>Robot’s response</th></tr>
<tr><td>The user sneezes.</td><td rowspan="3">The robot will not be able to detect the user’s face and will execute the danger protocol.</td></tr>
<tr><td>The user turns away.</td></tr>
<tr><td>The user covers mouth.</td></tr>
<tr><td>The user cannot be reached by the robot.</td><td rowspan="2">The robot will stop moving and execute the neutral protocol until the unexpected event is resolved.</td></tr>
<tr><td>Robot cannot detect any food.</td></tr>
<tr><td>The user shakes his/her head.</td><td rowspan="2">The quick movement of mouth will be detected and the danger protocol will be executed before the neutral protocol is executed.</td></tr>
<tr><td>The user is speaking</td></tr>
<tr><td>Food drops from the spoon.</td><td>The robot will detect that there is no food and collect more food. Food anywhere but the bowl will be ignored.</td></tr>
<tr><td>The user does not eat the food on the spoon.</td><td>The robot will hold its position and wait until food is collected.</td></tr>
</table>

4 Demonstrations

4.1 Scenario

Plastic spoons and the bowl were stuck onto the grippers of the Baxter robot using tape. Many broke in the process of testing. Whenever the robot was disabled, the arm of the robot would go down causing the cutlery to snap. Cardboard faces were used for safety reasons during testing. Solid green candy were used for safety and food detection reasons. Liquid foods have high risk of damaging the robot.

Figure 17- Spoon, fork, bowl and cardboard face

Originally the bowl was stuck on to a separate surface as oppose to the hand of the robot (see Figure 18). However, it was found that more calibration and detection work would need to be done so that the robot would be able to recognise the bowl and scoop from it. To fix the position of the bowl in reference to the robot, the bowl was attached to the robot's left end effector so that the robot is guaranteed to be able to reach the food easily (see Figure 19).

Figure 18 - Scooping from the surface of a bin

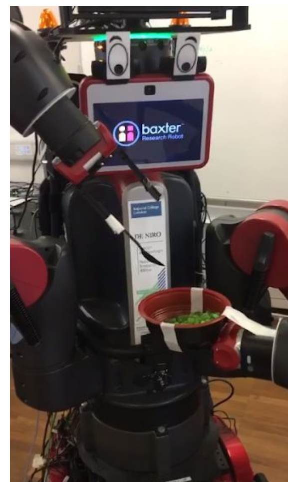


Figure 19 - Scooping from the robot's left end effector

4.2 Final Demonstration Step-by-step

This section is the detailed procedures for how to demonstrate everything that was accomplished. As mentioned previously, due to the lack of time, the different functions were not put together.

4.2.1 Food Detection Demo

This demo will program will view the spoon using the Baxter hand camera and detect whether or not their is food in the spoon. Figure 20 will show the outcome, viewing the food through Baxter hand camera.

Below is the step-by-step method to run the demo:

1. Open a terminal and split in two.
2. Execute the following in the terminal:
   ```
   >> cd catkin_ws
   >> bash baxter.sh
   >> roslaunch openni2_launch openni2.launch
   ```

3. In the second terminal execute the following:
   ```
   >> cd catkin_ws
   >> bash baxter.sh
   >> cd src/fred/src
   >> python cam_baxter.py
   ```

Figure 20: Viewing Food through Baxter Hand Camera.

4.2.2 Baxter Moving to Mouth Demo

This demo will program Baxter to scoop food and to put it to the detected mouth position from the Kinect camera. To program the robot to grip candy as oppose to scoop candy, simply replace `bc.playback_scooping()` to `bc.playback_gripping()` in the main part of `reactive.py`. See Figure 21 for the location of this code.

FRED

```
def main(args):
    # initialise node
    rospy.init_node('Reactive', anonymous=True)

    # Create instance of classes
    bc = BaxterControl() # see baxter_control.py for class explanation
    rc = ReactiveControl() # see above for class explanation

    # Robot control starts
    bc.playback_scooping() # playback of scooping
    rc.turn_on() # starts moving hand towards mouth

    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")

if __name__ == '__main__':
    ''' when reactive.py is ran, the following is executed '''
    print("Reactive Control")
    main(sys.argv)
```

Figure 21: The gray boxed line shows where the code can be changed so that Baxter is gripping candy as oppose to scooping candy.

Below is the step-by-step method to run the demo:

4. Baxter must be calibrated beforehand (see section 3.5.2 to see how).
5. Input the scale values from calibration into the `__mouth_callback__` function in `reactive.py` python file. Figure 22 shows where.

```
# Callback for mouthxyz changes
def __mouth_callback__(self, mouth_point):
    print('mouth callbacks')

    mouth_point = [mouth_point.x, mouth_point.y, mouth_point.z]

    # Scale values found from calibration.py - !!! Replace m_list and c_list after calibration
    m_list = [-0.00011107096733081952, 0.0022319768342585443, -0.00022221026524383612]
    c_list = [0.92590354598174474, -0.60855149454118618, 0.63769496828478978]

    # Calibrating mouth point from camera --> mouth point from robot
    output_pt = []
    for di in range(3):
        new_value = mouth_point[di]*m_list[di]+c_list[di]
        round_v = math.ceil(new_value * 1000.0) / 1000.0
        output_pt.append(round_v)
    mouth_pt = Point()
    mouth_pt.x = output_pt[0]
    mouth_pt.y = output_pt[1]
    mouth_pt.z = output_pt[2]

    # mouth_point is defined with calibrated values
    self.mouth_point = mouth_pt
    self.__update_values__()
```

Figure 22: The grey box outlines where in the code the new scale values should be placed.

6. Ensure that the Baxter robot is set up, turned on, and enabled as described in this link. The physical components should also be prepared and taped onto Baxter as described in section 4.1 Scenario.
7. Open the terminal in the Lab computers.
8. Execute the following in the terminal: >> `cd catkin_ws`
9. Split the terminal horizontally into five sections. This can be done by right clicking on the terminal. Each divided terminal section should now be in the catkin_ws directory.
10. For each terminal section, execute the following: `bash baxter.sh`
11. Execute the following for each of the different section terminals:

a. Terminal 1 | This is for launching the Kinect camera. Although streams of red text might popup, as long as yellow text indicating the camera has successfully been enabled is shown, the launching has been successful.

```
>> roslaunch openni2_launch openni2.launch
```

b. Terminal 2 | This runs the python program that will activate detection code for the kinetic camera. Two windows showing what the camera views should come up. One is in grayscale and the other in colour. The grayscale should have a white dot on where the program detects as where the mouth is.

```
>> cd src/fred/src
>> python cam_kinetic
```

c. Terminal 3 | This runs the playback server so that playback can be done. (Section 3.6.1 discusses why this needs to be done.)

```
>> rosrun baxter_interface joint_trajectory_action_server.py --mode velocity
```

d. Terminal 4 | This runs the baxter publisher which helps the BaxterControl class to receive information from Baxter. (Section 3.6.1 discusses why this needs to be done.)

```
>> cd src/fred/src
>> python baxter_pub.py
```

e. Terminal 5 | This runs the main program that uses the various data from the other running programs.

```
>> cd src/fred/src
>> python reactive.py
```

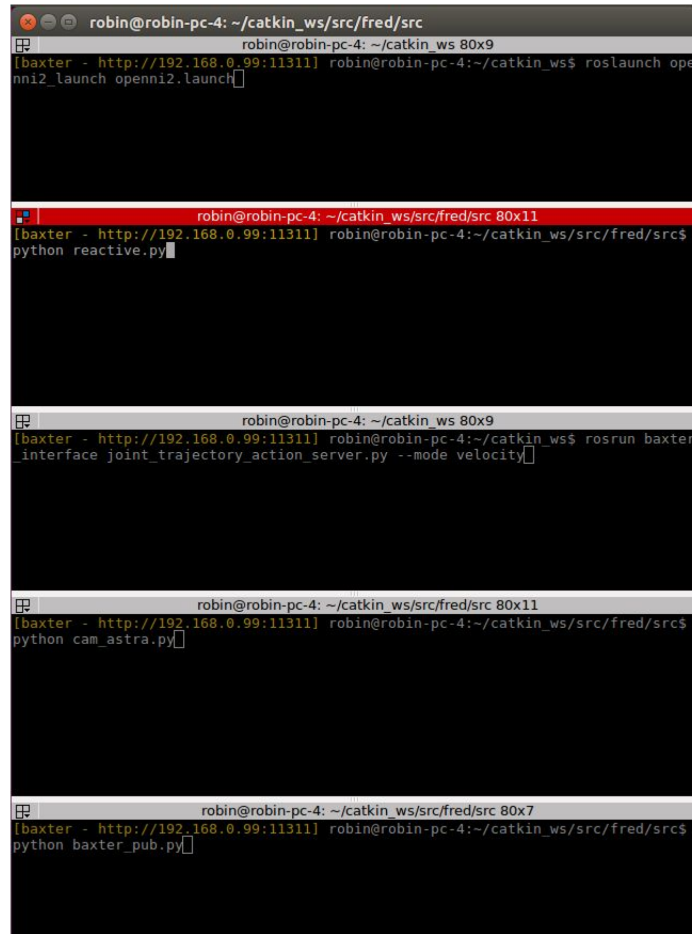


Figure 23: Screenshot before running everything in the terminals. It shows what each terminal should be running.

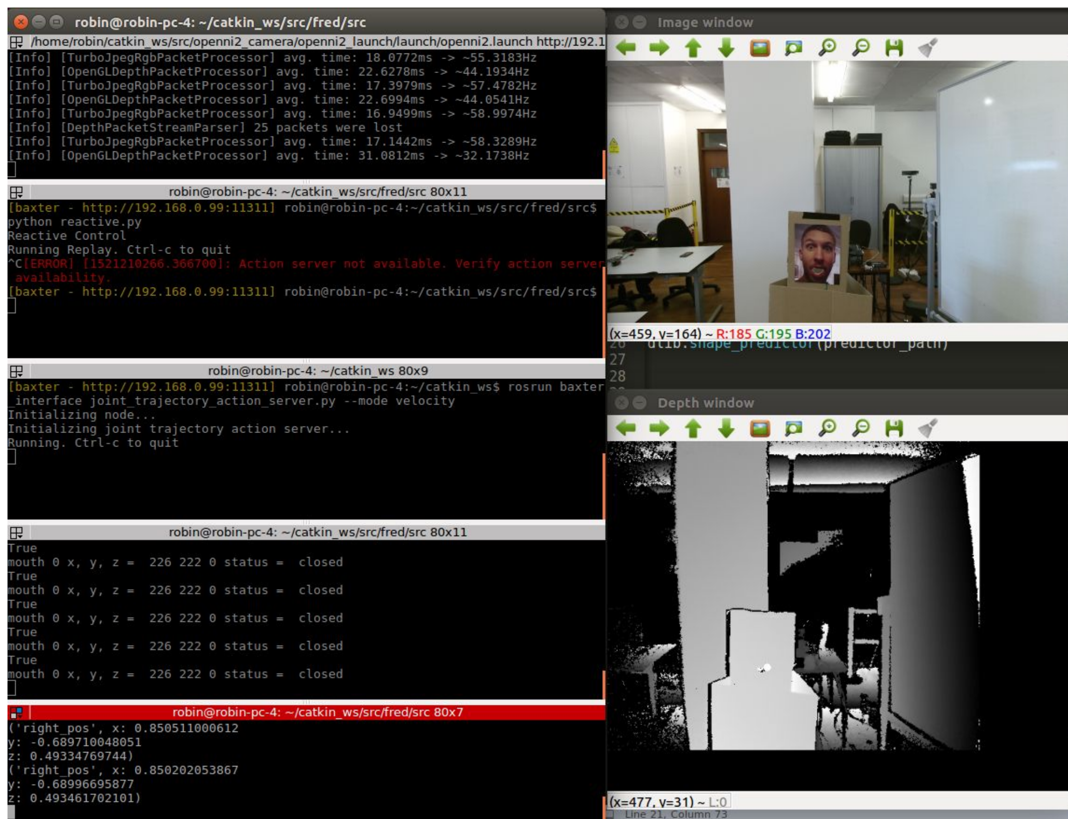


Figure 24: A Screenshot after everything in the terminal is running apart from `reactive.py`. `reactive.py` will be executed in the second terminal from the top.

NEXT STEPS OPPORTUNITY ↠
To significantly reduce the procedures required to run this demo (particularly step 8) and save time in the future, a launch file can be made. A launch file would automatically launch everything that needs to be launched in the separate files when run.

12. The robot should move after a while. Patience may be required as the computer may crash.

NEXT STEPS OPPORTUNITY ↠
Since this program frequently crashes the computer when executed, a separate computer can be used to run the camera's launch program (described in step 8a.) or rewriting certain parts of the programs can be done to make everything run more efficiently.

4.3 Video Guide

The attached long video tutorial contains the following information:

- *Calibration* - refer to *3.5.2 Calibration Step-by-step* for written instructions
- *Baxter feeding* - refer to *4.2.3 Baxter Moving to Mouth Demo* for written instructions

4.4 Safety

During testing and demonstrations, one person was always holding the emergency button in case anything were to go wrong. Also safety glasses was worn by anyone involved. During most testing, cardboard with print out faces were used as opposed to real faces, as demonstrated in Figure 25.

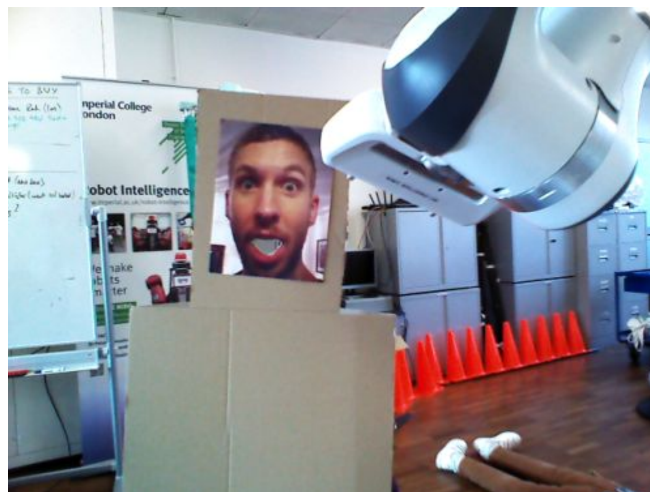


Figure 25: Cardboard Face Used for Safe Facial Detection

5 Conclusion

Despite obstacles, such as changing of robots, the outcome of this project was Baxter successfully detecting a face, detecting and scooping food and being able to move a spoon to a face. Possible opportunity for next steps in the project are acknowledged. The feeding logic flow was also composed. Although this was not fully autonomous for the demo, the components were working and the group were proud of their achievements and have created a good foundation for further improvement to implement feeding using Baxter.

6 References

[1] https://meetobi.com/

Franka Emika GmbH, 2017. *Franka Control Interface.* [online] Available at: <https://frankaemika.github.io/docs/index.html> [Accessed 6 March 2018].

Documentation http://de3-rob1-feeding.readthedocs.io/en/latest/

7 Appendix

7.1 FRANKA

7.1.1 Initial Test Using FRANKA Interface

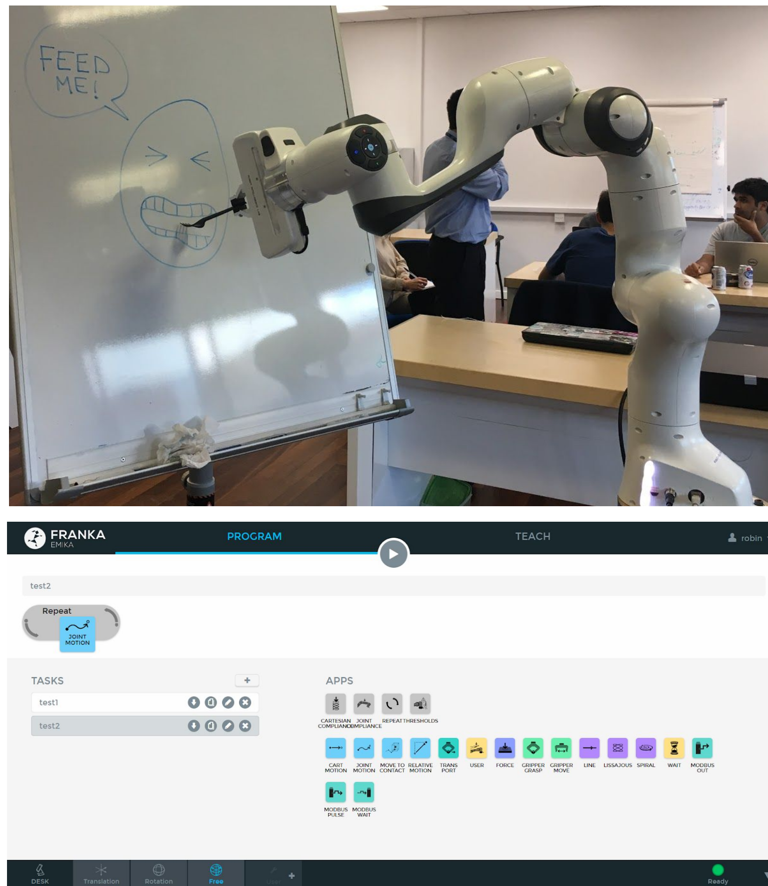


Figure 26: Using Franka Interface

7.1.2 Camera and FRANKA Calibration

7.1.2.1 Calibration Procedure

Calibration is achieved through finding coordinate points in the camera reference frame and the robots reference frame. This can be done through getting the mouth position and the robot's end effector position. Two people are prefered for the calibration process. The following are the steps for calibration, which is guided in the terminal (Figure 27):

8. Call the calibration code function and follow the instructions stated in the terminal.

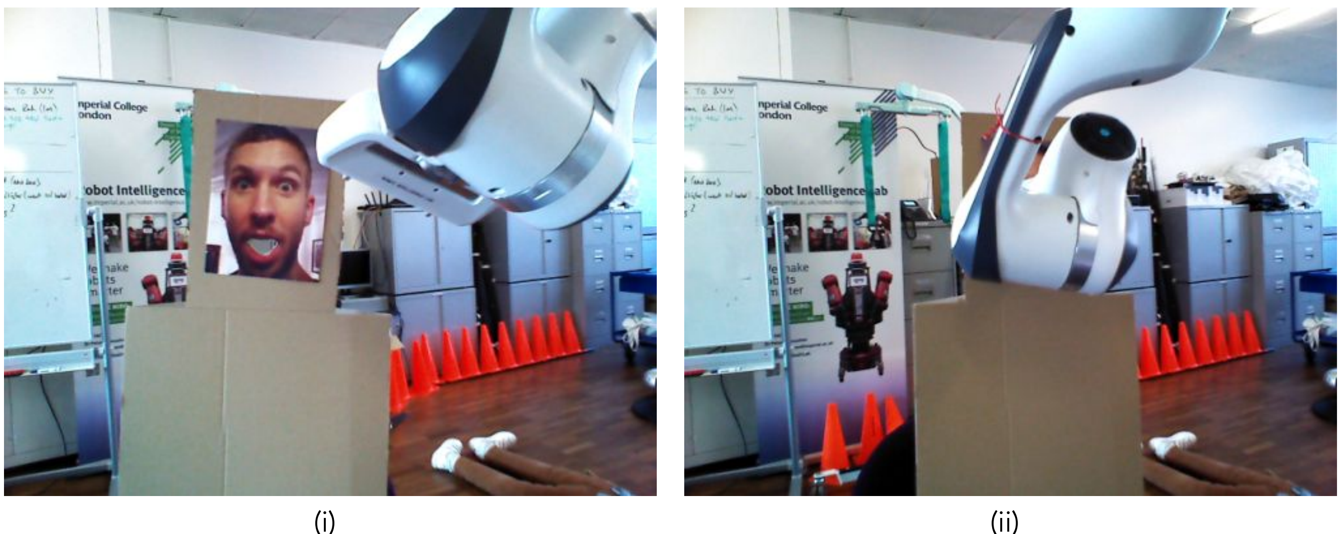
```
robin@robin-pc-4:~/DE3-ROB1-FEEDING$ python fred_feed.py
How many points would you like to calibrate with?: 5
Would you like to see current camera pos value? [Y/n]:

('Camera position:', [317.0, 173.0, 232.0])
Would you like to record this? [Y/n]:
('Camera coordinates :', [[317.0, 173.0, 232.0]])
('End effector coordinates :', [])
Would you like to see current end effector position value? [Y/n]:

('End effector position:', [0.267, -0.543, 0.347])
Would you like to record this? [Y/n]:
('Camera coordinates :', [[317.0, 173.0, 232.0]])
('End effector coordinates :', [[0.267, -0.543, 0.347]])
Would you like to see current camera pos value? [Y/n]:
```

Figure 27: Terminal

2. Input the number of points you would like to record.

3. Move the cardboard mouth to desired position. Record the mouth position. (Figure 28 (i))

4. Move the robot arm end effector to the mouth. Record the end effector position. (Figure 28 (ii))



(i) (ii)

Figure 28

5. Repeat steps 3 & 4 until sufficient points are recorded.

6. Linear regression is done with the points and a scale between the two frame of references are found. From there, test the scale and ensure sufficient accuracy.

7.1.2.2 Calibration Development

The calibration was tested and the result was very unstable. The left images (Figure 29) show how the robot arm seems to lack depth control, whilst the robot arm in the right is far better. This was the result of the inaccuracy of the depth measure from the camera and also the overshooting of the arm. Due to the change in robot, investigation towards resolving this problem was discontinued.

Figure 29

7.1.3 Control System

FRANKA has a libfranka library (Franka, 2017), which is a C++ interface library that is used to control the robot. With the help of Petar Kormushev and Fabian Falck, a terminal command can be used to control the very basic functions of the robot. From the Chess team's code, the commands are made accessible in Python code through a FrankaControl function class library. Petar Kormushev and Fabian Falck later extended their help by creating additional python and gripper controls for FRANKA through ROS.

7.1.4 Physical Component

To allow FRANKA to scoop, an alternative end effector was designed using given documentation measurements.

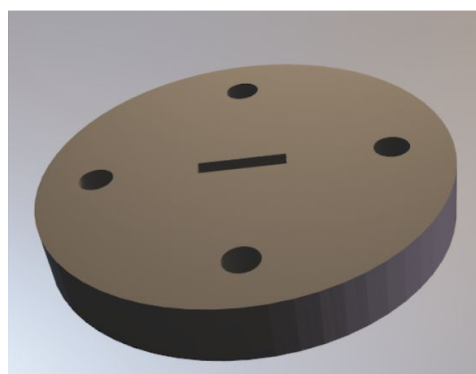


Figure 30: Plastic spoon would be slotted in the rectangular hole.
The circular holes would be for the screws.

7.2 Cameras

7.2.1 Astra Installation

Install SDK and Open a Simple Viewer:

```
cd ~/Downloads
# wget "https://www.dropbox.com/sh/p2uowlt3swdrfno/AACfEbv7ejIU-4FHy4Fyi0ZWa?dl=1" -O
Tools_SDK_OpenNI.zip

sudo pip install gdown
gdown "https://drive.google.com/uc?id=0B9P1L--7Wd2vSktrZXFYMEZOWXM" -O Tools_SDK_OpenNI.zip

mkdir ~/Downloads/Tools_SDK_OpenNI
cd ~/Downloads/Tools_SDK_OpenNI
unzip ~/Downloads/Tools_SDK_OpenNI.zip

cd 2-Linux
tar zxvf OpenNI-Linux-x64-2.2-0118.tgz

cd OpenNI-Linux-x64-2.2
sudo ./install.sh

cd ~/Downloads/Tools_SDK_OpenNI/2-Linux/OpenNI-Linux-x64-2.2/Samples/Bin
./SimpleViewer  # This should open a viewer for depth image
```

Use Astra camera with `openni2-camera` ROS package:

```
sudo apt-get install ros-kinetic-openni2-camera ros-kinetic-openni2-launch

cd ~/Downloads/Tools_SDK_OpenNI/2-Linux/OpenNI-Linux-x64-2.2/Samples/Bin
sudo cp libOpenNI2.so /usr/lib/libOpenNI2.so
sudo cp OpenNI2/Drivers/* /usr/lib/OpenNI2/Drivers/
```

Then edit `/usr/lib/pkgconfig/libopenni2.pc` to be like below:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/openni2

Name: OpenNI2
Description: A general purpose driver for all OpenNI cameras.
Version: 2.2.0.3
Cflags: -I${includedir}
Libs: -L${libdir} -lOpenNI2 -L${libdir}/OpenNI2/Drivers -lDummyDevice -lOniFile -lORBBEC
-lPS1080 -lPSLink

cd ~/ros/kinetic/src
wstool set ros-drivers/openni2_camera https://github.com/ros-drivers/openni2_camera.git --git
```

```
-v indigo-devel -y -u

cd ros-drivers/openni2_camera
source /opt/ros/kinetic/setup.bash
catkin bt
```

To launch the camera:

```
source ~/ros/kinetic/devel/setup.bash
roslaunch openni2_launch openni2.launch
```

To visualise in RVIZ:

```
rosrun rviz rviz
```