The Pyramid

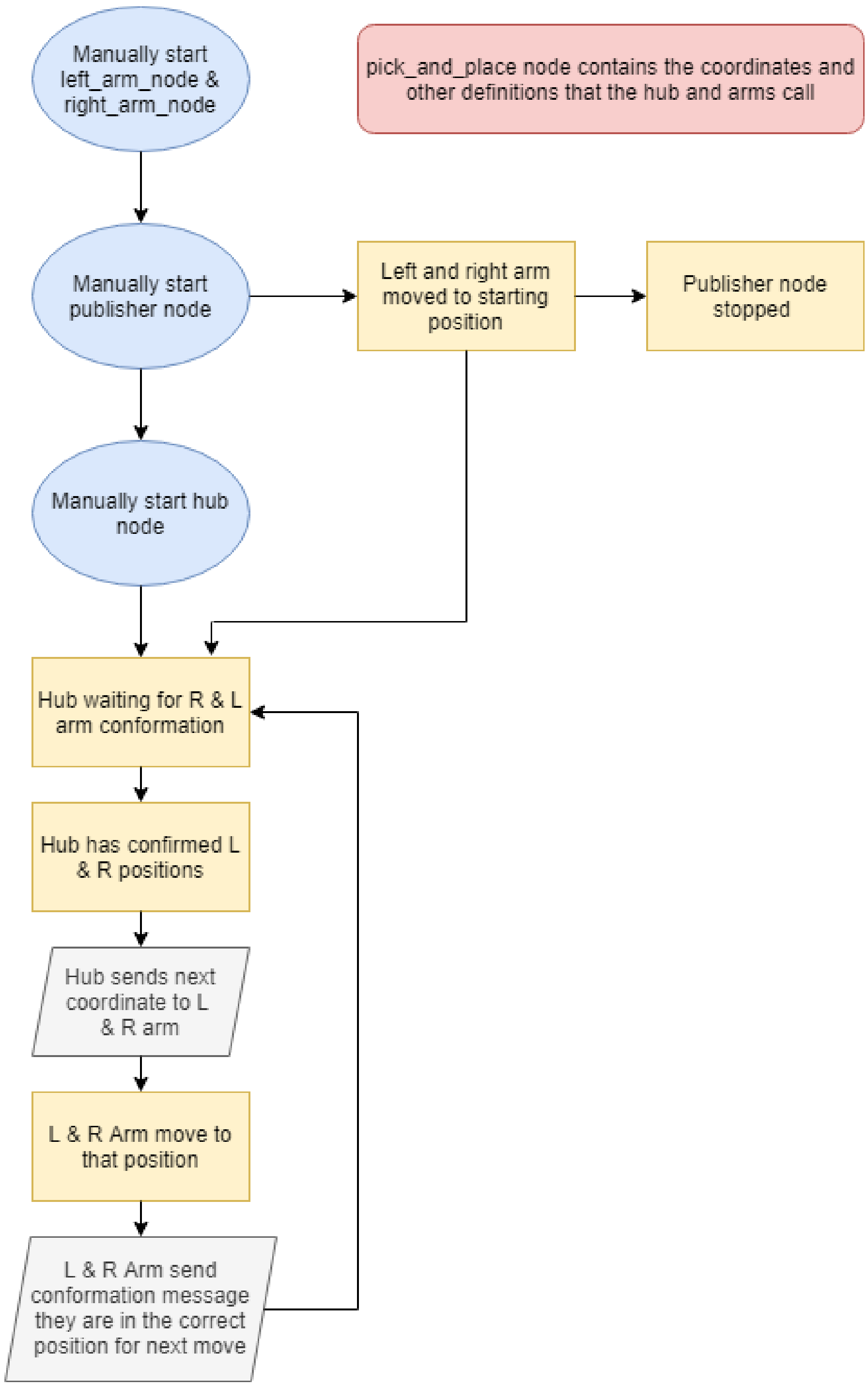
Robot De-Niro Documentation

Emily Branson, Minal Choudhary, Aisha Hussain, Gareth Jones, Visakan Mathivannan, Melisa Mukovic, Morteza Naimi

Table of Contents

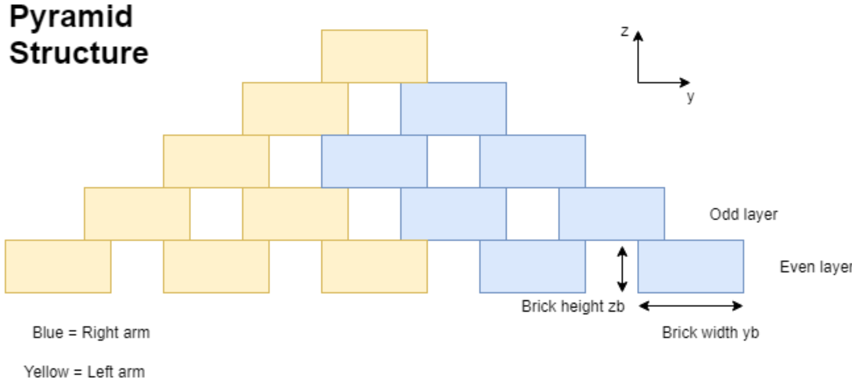
Visual Setup

Basic algorithm flowchart

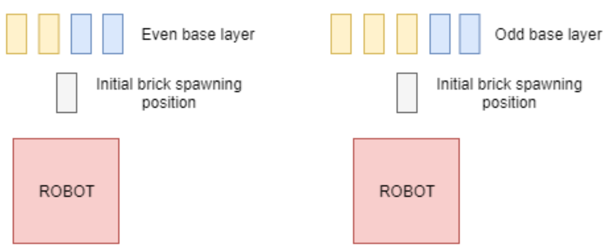


Pyramid & Robot Layouts

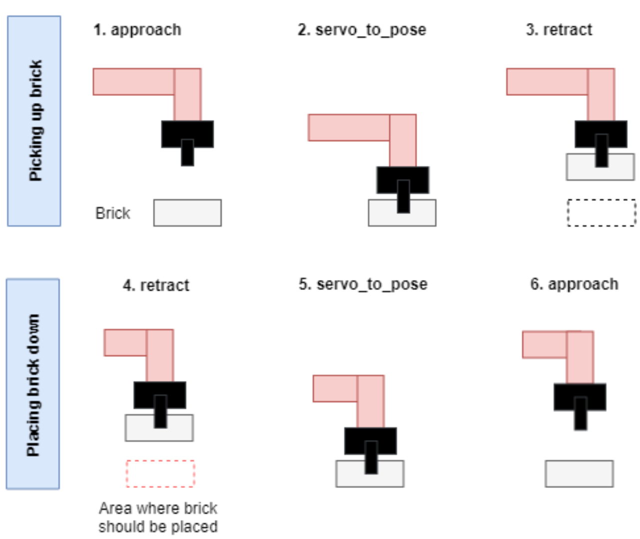
Pyramid Structure



Floor Layout



Robot Positions



Coordinates Generation

This is found in pick_and_place_py.

There are two functions for coordinate generation, def create_coordinates and def arrange_coordinates. def create_coordinates finds the coordinates based on brick number and layer, and then def arrange_coordinates arranges them in a specific way for each arm.

We arranged the bricks in the pyramid as below. Neighbouring bricks had a small gap between them to allow space for the gripper. The pyramid is centred on the robot, and the starting spawning brick is also right in front of the robot. Each arm will do its half of the pyramid structure, simultaneously placing bricks down in an alternating fashion. As you can see, the y coordinate (width) changes depending on brick and layer number.

def_arrange coordinates was necessary in putting alternating coordinates into a matrix to be sent to the right and left arm.

We initially started with only using one arm, and having the starting brick & structure all on this left hand side. However, we wanted produce a 'wow' factor, so decided to do that by having both arms in use simultaneously, which would build the pyramid even quicker.

Firstly we can input the number of layers want in the pyramid, which can be seen in the screenshot below. We create a matrix m which states the number of bricks in each layer (lines 294-297). x_structure is the x coordinate of the bricks (line 300). This value does not change. It is equal to:

- **x = (xs + xb + sc*0.05)**
- xs = starting x position coordinate
- xb = length of brick
- sc = scaling factor

```
291     #Input for number of layers
292     x = layer
293     print(layer)
294     x1 = list(range(1,x+1))
295     x1.reverse()
296     layers = np.asarray(x1)
297     m = layers.tolist() #this is a list of number of bricks in each layer. e.g. [3, 2, 1] i
298 
299     lay = [] #this is the matrix containint all the coordinates for the pyramid in x, y, z
300     x_structure = round(xs + xb + sc*0.05,4)
301 
```

We then create a loop from 0 to the number of layers in the pyramid, x, seen in the screenshot below. z_structure details the z coordinate of the bricks (line 303). This changes depending on which layer the brick is on:

- **z = zs + (x-j-1)*(zb) + 0.03**
- zs = starting z position of brick
- (x-j-1) = (number of layers - layer number currently on - 1). E.g. when doing layer 2 out of 6, the number of bricks above baseline will equal 6 - 2 - 1 = 3. Therefore put three brick heights up from the ground.
- zb = height of brick
- 0.03 = extra gap so the brick is dropped slightly above the previous layer

Next we look at the y positioning, (i.e. width along table - see Pyramids & Robot Structure). There is a shift of half a brick width along in the even layer compared to the odd. To create coordinates, we split each layer into a right and left half. For odd layers, these two halves were called pos_list (bricks on rhs) and neg_list (bricks on lhs), and for even layers these two halves were called init_list (bricks on rhs) and rev_list (bricks on lhs). Then we could give the left 'halves' (i.e. neg_list and rev_list) to the lh arm, and the right 'halves' (i.e. init_list and pos_list) to the right hand arm. See screenshot below.

Even layer (line 305 - 314):

We split the layer up in half. We started by having two lists containing half the bricks in that layer, init_list (right half) and rev_list (left half). Init_list and rev_list produces a range of numbers from 0 to (j+1)/2. This gives a shift of 0.5 to each value, thus shifting the even layer half a brick along compared to the odd one. Both of these lists are added into total_list. See screenshot below.

Odd layer (line 316 - 319):

We split the layer up in half. We started by having two lists containing half the bricks in that layer, pos_list and neg_list. Pos_list produces a range of integer numbers from 0 to (j+2)/2. (i.e. the y positions on the right). Neg_list produces a range of integer numbers from -j/2 to 0. (i.e. the y positions on the left). Both of these lists are added into total_list. See screenshot below.

y_structure is produced in a for loop using total_list produced in the even and odd layers.

- **y = ys + item*(yb + 0.03)**
- ys = starting y position
- Item = this is a value within the total_list of produced coordinates
- yb = width of brick
- 0.03 = gap between bricks

Finally we have c which is our list of x, y and z coordinates. We create a new matrix cnew in each loop and make this equal to c. We then append this to our lay matrix which contains a list of coordinates i.e. like [[x1, y1, z1], [x2, y2, z2], [x3, y3, z3]]. See screenshot below.

```
    for j in range(x):
        z_structure = zs + (x-j-1)*(zb) + 0.03

        if (j+1)%2 == 0: #even layers
            init_list = list(range(int((j+1)/2)))
            rev_list = list(range(int((j+1)/2)))

            for i in range(len(init_list)):
                init_list[i] = i+0.5
                rev_list[i] = -(i+0.5)

            rev_list.reverse()
            total_list = rev_list+init_list

        elif (j+1)%2 == 1: #odd layers
            pos_list = list(range(int((j+2)/2)))
            neg_list = list(range(int(-((j)/2)),0))
            total_list = neg_list + pos_list

        for item in total_list:
            y_structure = ys + item*(yb + 0.03)

            c = [round(x_structure,4), round(y_structure,4), round(z_structure,4)]

            cnew = [] #Creates a new matrix
            cnew = c #Makes cnew equal to the c coordinates
            lay.append(cnew)
```

Finally we arranged the coordinates in lay so they would be given in a certain order to the left and right arms. This can be seen in the screenshot below.

Firstly, a while loop was created. We wanted to sort the values in lay firstly according to their z height and then their y is arranged into right and left coordinates too. We appended values from lay from smallest to largest z values into list_of_index.

LArm: Then we found the largest y value in the sorted list_of _index to go into the left arm coordinates as the left arm was doing the leftmost large positive values of y to the midpoint of the pyramid of that layer. Then we appended the start position s into LArm, and then the smallest z, largest y coordinate into LArm, so it could go from the starting position to this coordinate. This coordinate was then removed for the next iteration.

RArm: Then we found the smallest y value in the sorted list_of _index to go into the right arm coordinates as the right arm was doing the rightmost negative values of y to the midpoint of the pyramid. Then we appended the start position s into RArm, and then the smallest z, smallest y coordinate into RArm, so it could go from the starting position to this coordinate. This coordinate was then removed for the next iteration. However, at the beginning of this section, we added an 'if len(lay) != 0:'. This will prevent the code going into this RArm section if there is no more values within the lay matrix (i.e. the LArm above it has removed the last brick). This prevents an index out of range error for having an empty list.

```
def arrange_coordinates(lay, s):
    RArm = []
    LArm = []

    while len(lay) != 0:
        z_smallest = lay[0][2]
        list_of_index = []
        for i in range(len(lay)):
            if lay[i][2] < z_smallest:
                list_of_index = [i]
            elif lay[i][2] == z_smallest:
                list_of_index.append(i)

        y_largest_index = list_of_index[0]

        for item in list_of_index:
            if lay[item][1] > lay[y_largest_index][1]:
                y_largest_index = item
        LArm.append(s)
        LArm.append(lay[y_largest_index])
        lay.remove(lay[y_largest_index])

        if len(lay) != 0:
            z_smallest = lay[0][2]
            list_of_index = []
            for i in range(len(lay)):
                if lay[i][2] < z_smallest:
                    list_of_index = [i]
                elif lay[i][2] == z_smallest:
                    list_of_index.append(i)

            y_smallest_index = list_of_index[0]
            for item in list_of_index:
                if lay[item][1] < lay[y_smallest_index][1]:
                    y_smallest_index = item
            RArm.append(s)
            RArm.append(lay[y_smallest_index])
            lay.remove(lay[y_smallest_index])
    return RArm, LArm
```

The Gazebo Environment:

Setting up the environment

The table is loaded into the simulation when this function is called:

```
def load_gazebo_models(table_pose=Pose(position=Point(x=0.82, y=0, z=-0.05)),
                       table_reference_frame="world"):
    # Get Models' Path
    model_path = rospkg.RosPack().get_path('baxter_sim_examples')+"/models/"
    # Load Table SDF
    table_xml = ''
    with open (model_path + "cafe_table/model.sdf", "r") as table_file:
        table_xml=table_file.read().replace('\n', '')

    rospy.wait_for_service('/gazebo/spawn_sdf_model')
    try:
        spawn_sdf = rospy.ServiceProxy('/gazebo/spawn_sdf_model', SpawnModel)
        resp_sdf = spawn_sdf("cafe_table", table_xml, "/",
                             table_pose, table_reference_frame)
    except rospy.ServiceException, e:
        rospy.logerr("Spawn SDF service call failed: {0}".format(e))
```

We then wrote code to spawn a brick at our previously defined starting point (x=0.49, y=0.01, z=0.752), in line 226 (see screenshot below). Like the table, the brick is referenced to 'world'. Line 237-240 spawns a new brick in the same location, and each new brick is given a new name in the format of new_brick_left(#), where # is a number from 1 to the number of bricks. This is done so there will always be a brick for the robot to pick up.

```
226 def load_brick_at_starting_point(brick_number, brick_pose=Pose(position=Point(x=0.49, y=0.01, z=0.752)),
227                 brick_reference_frame =  'world'):
228 
229     model_path = rospkg.RosPack().get_path('baxter_sim_examples')+"/models/"
230     # Load Table SDF
231 
232     brick_xml = ''
233 
234     with open (model_path + "new_brick/model.sdf", "r") as brick_file:
235         brick_xml=brick_file.read().replace('\n', '')
236 
237     try:
238         spawn_sdf = rospy.ServiceProxy('/gazebo/spawn_sdf_model', SpawnModel)
239         resp_sdf = spawn_sdf("new_brick_left{}".format(brick_number), brick_xml, "/",
240                              brick_pose, brick_reference_frame)
241     except rospy.ServiceException, e:
242         rospy.logerr("Spawn SDF service call failed: {0}".format(e))
243         # Spawn Block URDF
244     rospy.wait_for_service('/gazebo/spawn_urdf_model')
```

Deleting the Model and Bricks

When ROS is exited, the gazebo model containing the cafe table (line 256) and bricks (line 264) are deleted. The loop in line 262 below is there so the code can delete every brick since they all have different names, when the function is called.

```
249 def delete_gazebo_models():
250     # This will be called on ROS Exit, deleting Gazebo models
251     # Do not wait for the Gazebo Delete Model service, since
252     # Gazebo should already be running. If the service is not
253     # available since Gazebo has been killed, it is fine to error out
254     try:
255         delete_model = rospy.ServiceProxy('/gazebo/delete_model', DeleteModel)
256         resp_delete = delete_model("cafe_table")
257     except rospy.ServiceException, e:
258         rospy.loginfo("Delete Model service call failed: {0}".format(e))
259 
260 
261 def delete_bricks(brick):
262     for item in range(brick):
263         try:
264             resp_delete = delete_model('new_brick{}'.format(item))
265         except rospy.ServiceException, e:
266             rospy.loginfo("Delete Model service call failed: {0}".format(e))
```

Motion Planning

We modified the existing 'IK_pick_and_place_demo.py' file to create the basis for our motion planning. Inverse kinematics is used to do the motion planning.

Defining the Quaternion and Joint Angles

The joint angles and quaternion angle for the initial joint positions for the left arm are defined below, and can be found in left_arm_node.py:

```
    joint_angles = {'left_w0': -0.08042412041756966, 'left_w1': 1.6385748360237509,
                    'left_w2': -0.19069126257995908, 'left_e0': -0.490820123608967,
                    'left_e1': 1.6772691387232428,'left_s0': -0.4379951625489787,
                    'left_s1': -1.776227964501175}
```

```
    overhead_orientation = Quaternion(
                             x=-0.0249590815779,
                             y=0.999649402929,
                             z=0.00737916180073,
                             w=0.00486450832011)
```

Trial and error was used to modify this initial starting position so the arm was at a safe distance away from the table and bricks. These variables were also defined for the right arm and can be found in right_arm_node.py

Pre-existing Demo Code

The following code was given in the demo code and is found in 'pick_place_functions.py':

1. The function ik_request converts the coordinates, defined as pose, into joint angle positions.
2. The function _guarded_move_to_joint_position first checks if joint angles are defined, and if they are, moves the respective arm to the joint angles calculated for that specific set of coordinates.
3. The function _approach creates a copy of the current pose coordinate set and modifies the z part of the coordinate to include the current z distance plus the hover distance which is defined previously. This is fed back into the IK_request function to produce new joint angles. These angles are fed into _guarded_move_to_joint_position to move the robot to the new required pose. so that when the function approach is called, the end effector moves to this new location.

4. The same logic is applied for _servo_to_pose.
5. The function pick runs the above functions for movements 1 to 3, and function place runs the above functions for movements 4-6, as defined in Pyramid & Robot Structure image above.

Defining Pick and Place

The pick and place function for the left arm at a hover distance of hover_distance=0.1 is defined as pnp, and the robot is told to move to the positions defined by the joint angles above in these lines of code:

```
pnp = PickAndPlace('left', hover_distance)
pnp.move_to_start(joint_angles)
```

The same code is used for the right arm, except 'left' is replaced with right. This can be found in the arm node codes.

Integrating Coordinates to Motion Planning Code

The next step is to integrate the coordinates with the motion planning code. The coordinates for each position is outputted in this format: [x,y,z] where x, y and z are our length, width and height respectively. The code below converts each coordinate set in the matrix LArm for the left arm and RArm for the right arm into the correct format to be inputted into the pnp.pick and pnp.place function, which was given in the demo code.

```
for coord_set in LArm:
    block_poses.append(Pose(position=Point(x=coord_set[0], y=coord_set[1], z=coord_set[2]),orientation=overhead_orientation))
```

Each set of newly formatted coordinates are appended into 'block_poses'. The code will then run through each set of coordinates during the pick and place function. This can be found in the arm node codes.

Our Process

We first tested the code by manually writing out 3 sets of coordinates for just one arm - the initial position of the brick, position where we wanted it to be placed, and the second position we wanted it to be placed to. We then used this to create a loop which runs the pick and place functions so we could test the motion plan.

The next step was to actually pick and place a brick. When gazebo's brick position was inputted into the end effector position, we noticed that the robot's position was far away from the brick and that there was an offset present. A lot of time was spent identifying the offset amount, so we could coordinate the end effector and the actual brick location. We did this by, creating a loop which prints the coordinates the robot was told to go to (0,0,0) and actual end effector position in the gazebo. This was then used to work out the offset:

```
    def print_position(self):
        current_pose = self._limb.endpoint_pose()
        print(current_pose)
        x = current_pose['position'].x
        y = current_pose['position'].y
        z = current_pose['position'].z
        print(x, y, z)
```

Once we got this code working, we moved onto developing a motion plan for 2 arms.

Collision Avoidance and Improving Coordination

We set ourselves the challenge of using both arms simultaneously, to pick up bricks from the same starting point. One consistent problem we were facing was the collision of both robot arms - especially because one arm would typically finish its set of movements before the other, so would cause collisions a few iterations into the simulation. We therefore needed to think of a clever solution to this problem.

Using a 4-point Space for Both Arms:

Our collision-avoidance method uses a 4-point space that both robot arms circulate through; the pick position, the left and right outer 'collision avoiding' position and the place position. By having these motions that each arm circulates through, both arms will never catch up to one another/collide.

The code below shows a series of 'if' statements that run through each of these four movements. Each arm only goes into the next movement if ***both*** arms are ready in the correct final position from the previous step. This means they will always be synchronised and waiting for each other to move.

The 4 movements (see code below):

1) Right arm moves to right 'collision avoiding' position, left arm moves to pick.
2) Left arm moves to left 'collision avoiding' position, right arm moves to pick.
3) Right arm moves to right 'collision avoiding' position, left arm moves to place.
4) Left arm moves to left 'collision avoiding' position, Right arm moves to place.

For an in depth explanation of this, look at the video called 'robot pyramid code explained': https://imperialcollegelondon.app.box.com/s/05kedscpq0r26cp3qb7ounbtexipd7wr/folder/70047179358

We used the 'hub' node (further details on this in the following section) to ensure the arms wait for messages between nodes for their movements - avoiding the issue of one arm catching up to the other, and subsequently colliding.

Left-arm:

```
            if value == 1:
                pnp.move_to_start(joint_angles)
                brick+=2
                load_brick_at_starting_point(brick)
                msg.data = [2,topic_number]
                start = time.time()
                while time.time() - start < 2:
                    pub.publish(msg)
                topic_number+=1
            elif value == 2:
                pnp.place(block_poses[count])
                msg.data = [3,topic_number]
                count+=1
                start = time.time()
                while time.time()-start < 2:
                    pub.publish(msg)
                topic_number+=1
            elif value == 3:
                pnp.move_to_start(joint_angles)
                msg.data = [4,topic_number]
                start = time.time()
                while time.time()-start<2:
                    pub.publish(msg)
                topic_number +=1
            elif value == 4:
                pnp.pick(block_poses[count])
                msg.data = [1,topic_number]
                count+=1
                start = time.time()
                while time.time()-start<2:
                    pub.publish(msg)
                topic_number += 1
            check = True
```

Right-arm:

```
            if value == 1:
                pnp.pick(block_poses[count])
                count+=1
                msg.data = [2,topic_number]
                start = time.time()
                while time.time()- start < 2:
                    pub.publish(msg)
                topic_number+=1
            elif value== 2:
                pnp.move_to_start(joint_angles)
                brick +=2
                load_brick_at_starting_point(brick)
                msg.data = [3,topic_number]
                start = time.time()
                while time.time()-start < 2:
                    pub.publish(msg)
                topic_number +=1
            elif value == 3:
                pnp.place(block_poses[count])
                count+=1
                msg.data = [4,topic_number]
                start = time.time()
                while time.time() - start < 2:
                    pub.publish(msg)
                topic_number +=1
            elif value == 4:
                pnp.move_to_start(joint_angles)
                msg.data = [1,topic_number]
                start = time.time()
                while time.time() - start < 2:
                    pub.publish(msg)
                topic_number +=1
            check = True
```

Generating new topics:

```
        while not rospy.is_shutdown() and count < len(block_poses):
            if check == False:
                pub = rospy.Publisher('move_right_arm_hub{}'.format(topic_number+1), Int64MultiArray, queue_size =10)
                msg = Int64MultiArray()
                msg.layout.dim = [MultiArrayDimension('',2,1)]
```

One problem that we were running into was that the topics that we are publishing in would sometimes have the old information running within them. This meant that the arms would get confused with which stack of queues to be operating. To solve this, we introduced another control variable in which we are generating new topic names that we are publishing to in each iteration, by adding a number to it: **topic_number+1**. This meant there would always be a new topic number on each iteration.

Communication and Publisher

Publisher

The purpose of the publisher was to set up the robot into the starting position which enabled us to get the table and bricks into the correct location. The publisher refers to a position which moves both left and right arms from the tucked away position around to the front of the robot. Publisher then communicates with the communication node to inform the robot that it is in position and is ready to start the execution code. After the arms are set into this position, publisher.py is shutdown for processing power efficiency.

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

import rospy
from std_msgs.msg import Int32

pub = rospy.Publisher('chatter', Int32, queue_size = 10)
rospy.init_node('Starter_node')
r = rospy.Rate(10)
while not rospy.is_shutdown():
    pub.publish(4)
    r.sleep()
```

Communication Node

Within the understanding of this aspect of the code, we have internally labelled the communication node as the ‘hub’. The hub is the master node which controls what both arms are doing, when they are doing it, and when they need to move onto the next movement in the sequence to build the pyramid.

Topic Node

This part of the communication node uses topic numbers (line 24, 27) and sends the different motions to the left and right arm.

```
11  topic_number = 0
12  move_left = [6,8]
13  move_right = [5,7]
14
15  def callbackleft(data):
16          global move_left
17          move_left = data.data
18
19  def callbackright(data):
20          global move_right
21          move_right = data.data
22
23  def callback(data):
24          global topic_number
25          global move_left
26          global move_right
27          pub = rospy.Publisher('do_shit{}'.format(topic_number), Int64MultiArray, queue_size = 10)
28          msg = Int64MultiArray()
29          msg.layout.dim = [MultiArrayDimension('',2,1)]
30          r = rospy.Rate(10)
31          start = time.time()
32          print('here')
```

How It Works

As above, the publisher has moved the arms to the initial starting position. The hub then receives a message that the arms are both ready to start moving bricks. The hub sets the position to be in the 4th position of the loop which is when the right arm is out of the way and left arm is moving to pick. This means that when the communication node begins and the sequence starts (from position 1) and the next message it waits for will now be part of the loop.

This is when the left arm has picked up the first brick and right arm is moving to pick up a brick. The right arm moves into position and hovers over the brick, lowers itself down, grabs the brick and returns to the hover position. Once it's in this position it then publishes a message to the communication node to say that the if statement has been completed.

In the second if statement (line 41), the left arm moves out of the way to the left, and the right arm picks up a brick and returns to the hover position.

In the third statement (line 44), the right arm moves out of the way to the right and the left places the brick and returns to the hover positions.

Finally in the fourth statement, (line 47), the left arm moves out of the way and the right arm places the brick and returns to the hover positions. This allows the arms to be in the correct locations to start the loop again with the right arm moving out of the way and the left arm going to pick up another brick.

```
34          while time.time()-start < 5:
35                  #print(data)
36                  print(msg.data)
37                  #print('hi')
38                  if data == 1:
39                          msg.data = [1,topic_number]
40                          pub.publish(msg)  #left picked right moving to pick
41                  elif data== 2:
42                          msg.data = [2,topic_number]
43                          pub.publish(msg)        #right picked left moving to place
44                  elif data== 3:
45                          msg.data = [3,topic_number]
46                          pub.publish(msg)        #left placed right moving to placed
47                  elif data == 4:
48                          msg.data = [4,topic_number]
49                          pub.publish(msg)        #right placed left moving to pick
50                          #print('here')
51          move_right = [5,7]
52          move_left = [6,8]
```

For an in depth explanation of this, look at RobotPyramidCodeExplained in https://imperialcollegelondon.app.box.com/s/05kedscpq0r26cp3qb7ounbtexipd7wr/folder/70047179358

Listener Function

Within the communication node there is a listener function which constantly looks for the messages that are sent out from the loop to indicate whether the arms have completed their movements. These are recorded with two parts, the position number, indicated in the if statements above (e.g. in line 39, the number 1 is the position number), and the topic number. The position number remains the same, however the topic number is continually updated by a value of 1 (line 85) to ensure that the listener is constantly checking for the latest piece of information available and doesn't miss any updates in the movement of the arms if the topic number remains the same. More information can be found in 'Generating new topics' below.

```
def listener():
        global move_left
        global move_right
        global topic_number
        while not rospy.is_shutdown():
                #print("listening")
                '''
                rospy.Subscriber('move_left_arm_hub{}'.format(topic_number+1), Int64MultiArray, callbackleft)
                rospy.Subscriber('move_right_arm_hub{}'.format(topic_number+1), Int64MultiArray, callbackright)
                '''
                if topic_number%2 == 0:
                        msg_left = rospy.wait_for_message('move_left_arm_hub{}'.format(topic_number+1), Int64MultiArray)
                        msg_right= rospy.wait_for_message('move_right_arm_hub{}'.format(topic_number+1), Int64MultiArray)
                else:
                        msg_right = rospy.wait_for_message('move_right_arm_hub{}'.format(topic_number+1), Int64MultiArray)
                        msg_left =  rospy.wait_for_message('move_left_arm_hub{}'.format(topic_number+1),Int64MultiArray)


                move_left = msg_left.data
                move_right =msg_right.data

                #print('topic: {} \tleft: {}\t right: {}'.format(topic_number,move_left,move_right))

                #print('move right {}'.format(move_right))
                #print('move left {}'.format(move_left))

                if move_left[1] == topic_number and move_right[1] == topic_number:
                        topic_number +=1
                        print('move right {}'.format(move_right))
                        print('move left {}'. format(move_left))
                        callback(move_left[0])
```

Initializer Section

The initializer section of the code starts up the loop as shown in the diagram in Pyramid & Robot Structure section above. It starts off the communication node and starts the synchronization process. Within this loop it then defines the fact that the robot begins in the 4th position, as mentioned above, to ensure that the next movement is to get the robot arms into the first position of the sequence. The initializer is completed by enabling the listening function of the algorithm.

```
if __name__ == '__main__':
        rospy.init_node("hub")
        pub = rospy.Publisher('do_shit{}'.format(topic_number), Int64MultiArray, queue_size = 10)
        msg = Int64MultiArray()
        msg.layout.dim = [MultiArrayDimension('',2,1)]

        r = rospy.Rate(10)
        start = time.time()

        while time.time() - start < 3:
                msg.data = [4,topic_number]
                pub.publish(msg)


        listener()
```

Simulation Problems

Issued Faced

Running the simulation for the first few times with our code, we ran into a few initial problems - particularly with things like; the brick slipping out of the gripper, the brick bouncing when placed on the table, etc. To solve these, we needed to change a few variables:

- The friction coefficient of the gripper
- The friction coefficient of the brick
- The weight of the brick
- The slip of the brick
- The slip of the table
- The force of the gripper
- The width of the gripper
- The gripper strength: 4

Modifying the Brick

To add simplicity and efficiency in enabling changes to the brick we created a new urdf file called new.urdf which contained the specifications of brick which we have been using. This new file was constructed to enable us to test out different orientations of the brick, by changing the x and y values of the brick, without affecting the quaternion itself and keeping the brick centred around the start position within each orientation of the brick. See code screenshot below.

```
<robot name="block">
  <link name="block">
    <inertial>
      <origin xyz="0.025 0.025 0.025" />
      <mass value="0.5" />
      <inertia
      ixx="0.01666667"
       ixy="0.0"
       ixz="0.0"
        iyy="0.01666667"
         iyz="0.0"
          izz="0.01666667" />
    </inertial>
    <visual>
      <origin xyz="0.025 0.025 0.025"/>
      <geometry>
        <box size="0.086 0.192 0.062" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="0.025 0.025 0.025"/>
      <geometry>
        <box size="0.086 0.192 0.062" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="block">
    <material>Gazebo/Blue</material>
     <mu1>1000</mu1>
     <mu2>1000</mu2>
  </gazebo>
</robot>
```

Additional Changes Made:

Orientation of End Effector

To make our motion planning more ambitious and create more space, we changed the orientation of the initial brick at the starting position so that the end effector has to twist to place it in the correct position to build the pyramid. This uses the the quaternion multiply function to convert the original overhead position 90 degrees.

```
original_overhead = np.array([-0.0249590815779,
                          0.999649402929,
                          0.00737916180073,
                          0.00486450832011])
place_overhead_orientation = Quaternion(
                          x=original_overhead[0],
                          y=original_overhead[1],
                          z=original_overhead[2],
                          w=original_overhead[3])

second_overhead = quaternion_multiply(quaternion_from_euler(0,0,1.57),original_overhead)

pick_overhead_orientation = Quaternion(
                      x=second_overhead[0],
                      y=second_overhead[1],
                      z=second_overhead[2],
                      w=second_overhead[3])
```

Pick Hover Distance is Different for Picking or Placing Bricks

The pick approach hover distance was reduced to half the hover distance, to ensure that the movement was more accurate, as shown by following lines in the pick function:

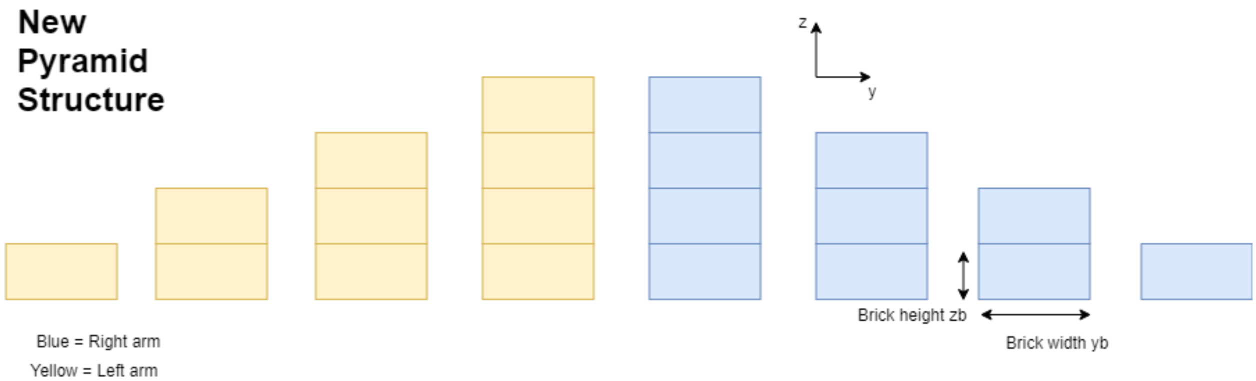
```
def pick(self, pose):
    self._hover_distance = self._hover_distance/2
    # open the gripper
    self.gripper_open()
    # servo above pose
    self._approach(pose)
    # servo to pose
    self._servo_to_pose(pose)
    # close gripper
    self.gripper_close()
    # retract to clear object
    self._retract()
    self._hover_distance = self._hover_distance*2
```

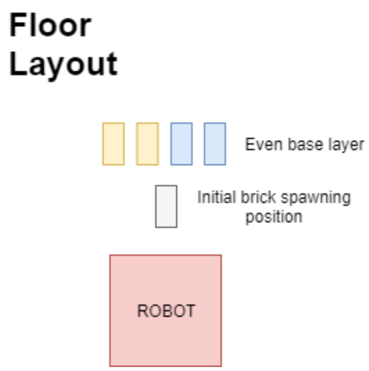
The place retract hover distance was decreased to 1, to ensure that there is no collision with the structure but increase the usable workspace

Structure of Pyramid has Changed

Upon further testing it was discovered that anything above 4 layers was outside of the reachable workspace of De-Niro meaning that the structure needed to be changed to incorporate the minimum brick allowance. To do this we kept the same idea of a pyramid but changed the structure of the pyramid and the way it has been built.

In this new structure the pyramid is formed using an increase in two bricks with every layer. In the case the pyramid will continue to be build at a maximum of four layers but beginning with eight bricks at the bottom, reducing the number of bricks by two at each layer. This can be depicted in the New Pyramid Structure figure below.





As mentioned earlier in the report, we used the functions def create_coordinates and def arrange_coordinates to create our pyramid. However, due to the pyramid structure change, we have now inserted a new definition called def create_coordinates_new, as seen in the screenshot below. This definition will produce a pyramid with only even layers.

Z_structure describes the z coordinate and depends on the layer number.

- z_structure = zs + (layer-(i/2))*zb + 0.03
- zs = starting z position
- (layer - (i/2))*zb = (number of layers - half the current layer number) * brick width
- 0.03 = gap above the bricks

Pos_list takes the brick coordinates on the left hand side, for example going from 3, 2, 1, 0 and Neg_list takes the bricks on the right hand side, for example going from -3, -2, -1, 0. tot_list = neg_list + pos_list then adds these two lists together.

Then total_list = [x+0.5 for x in tot_list] then adds 0.5 shift on all of these values, so that they are in ‘even’ positions.

Y_structure then finds out the position of bricks along the width of the table. This no longer depends on odd or even layers, as there are only even layers present. Therefore:

- y_structure = ys + item*(yb + 0.03)
- Ys = y starting position
- Item*(yb + 0.03) = this multiplies each element in total_list with the brick length + the gaps between bricks.

```
def create_coordinates_new(layer):
    sc = 1; #this scales all the brick dimensions and the gaps between the bricks
    #brick dimensions
    xbrick = 0.192; #this is the length of the brick
    ybrick = 0.086; #width of brick
    zbrick = 0.062; #height of brick
    #Scaled brick dimensions
    xb = xbrick*sc
    yb = ybrick*sc
    zb = zbrick*sc
    #starting position`
    xs = 0.49; #this is the starting position along length
    ys = 0.01; #starting position along height
    zs = 0.11; #starting position along width

    s = [xs, ys, zs] #this is the permanent starting position which is reeferenced as the centre of the brick

    #Input for number of layers
    x = layer
    x1 = list(range(1,x+1))
    x1.reverse()
    layers = np.asarray(x1)
    l = layers.tolist() #this is a layersist of number of bricks in each layer. e.g. [3, 2, 1] is 3 bricks in base layer, 
    m = [d*2 for d in l]
    lay = [] #this is the matrix containint all the coordinates for the pyramid in x, y, z which corrrespond to length, de
    x_structure = round(xs + xb + sc*0.05,4)

    for i in m:

        z_structure = zs + (layer-(i/2))*zb + 0.03
        pos_list = list(range(int((i)/2)))
        neg_list = list(range(int(-((i)/2)),0))
        tot_list = neg_list + pos_list
        total_list = [x+0.5 for x in tot_list]

        for item in total_list:
            y_structure = ys + item*(yb + 0.03)

            c = [round(x_structure, 4), round(y_structure, 4), round(z_structure, 4)]

            cnew = []
            cnew = c
            lay.append(cnew)
    return lay, s
```

Repeatability and Reliability

Our algorithm successfully works without dropping bricks over 3 times in a row. It can work up to layer 4 with the modified structure, which is the maximum distance the arms can reach in the workspace. This is demonstrated in the video's folder in box.