Connect Four Robot

Mar 13, 2020

Contents:

CHAPTER 1

Overview & Main File Breakdown

Warning: For best UI experience, please use the web version of the documentation as opposed to the PDF version.

In this project, the Franka Emika "Panda" robot was programmed to play a game of Connect 4 against a human. The human and robot took turns playing by dropping their tokens in their chosen column until one won the game. To do this, multiple tools such as Computer Vision, a Minimax Game Algorithm, Motion Planning and Collision Detection were implemented.

Due to the limited access to the physical robot, extensive simulations of the robot's motion were also performed, using Gazebo for visualisation.

Here is a quick introduction video of what the robot does.

The flow chart below shows an overview of the main program that plays the game.

Now here is a brief overview of our main script that plays the game.

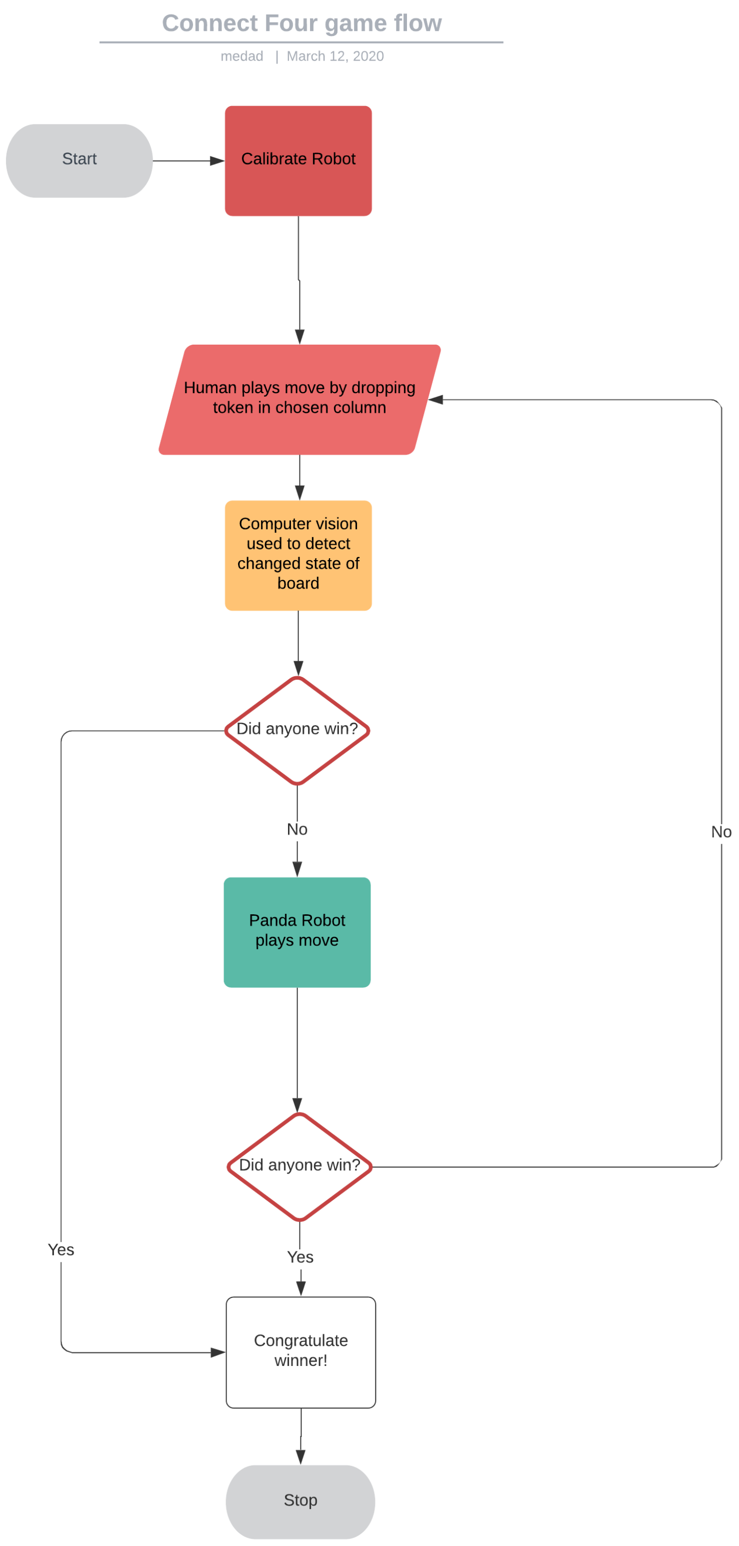
1.1 Importing Libraries

Warning: Although this is a python script, it will NOT run in an IDE in Windows. Many of the functions and libraries imported are specific for the ROS environment, which needs to be run on Ubuntu in a Virtual Machine with with necessary dependencies installed, or on computer with the necessary packages installed.

First it is necessary to import all of the required external functions and python libraries. Many driver functions were abstracted away in other python files to prevent cluttering up the main python file.

```python
# Import required python files
import c4_bot_functions as botfunc
import open_cv as vision
from c4_class import Connect4Robot
```

(continues on next page)


Connect Four game flow
medad | March 12, 2020
Start
Calibrate Robot
Human plays move by dropping token in chosen column
Computer vision used to detect changed state of board
Did anyone win?
No
Panda Robot plays move
Did anyone win?
No
Yes
Yes
Congratulate winner!
Stop

(continued from previous page)

```
# Import libraries
import sys
import copy
import rospy
import subprocess
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import tf
from IO import bcolors
from time import sleep
from math import pi
from std_msgs.msg import String, Float64MultiArray, MultiArrayDimension, Float64
from moveit_commander.conversions import pose_to_list
```

1.2 Debug Switches

To aid debugging, a number of boolean variables were used to switch on and off sections of code during development. This is due to sections of code being non-functional/insufficiently tested initially. OpenCV took a long time to develop and test, so in the meantime a switch was used so that the rest of the code could be tested without relying on computer vision. The switches are set at the beginning of the main code flow as follows.

```
simulation_status = True
visionworking = False
```

1.3 Initialisation

When everything has been imported, the Franka Emika robot needs to be set up and initialised. The following code shows the setup procedure for this robot.

```
# Set up Franka Robot
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('panda_demo', anonymous=True)
robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()
rospy.sleep(2)

# Get object frames
p = geometry_msgs.msg.PoseStamped()
#p = PoseStamped()
p.header.frame_id = robot.get_planning_frame()
p.pose.position.x = 0.4
p.pose.position.y = -0.301298
p.pose.position.z = -0.2
p.pose.orientation.x =  0.0
p.pose.orientation.y = 0
p.pose.orientation.z = 0.0
p.pose.orientation.w = 0.4440158
#scene.add_mesh("Connect4", p,"connect4.STL")
scene.add_box("table", p, (0.5, 1.5, 0.6))
```

(continues on next page)

(continued from previous page)

```
rospy.sleep(2)

display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',␣
↪moveit_msgs.msg.DisplayTrajectory, queue_size=20)

# This command makes ros to change the 'allowed_start_tolerance' to 0.05. Prevents␣
↪controller failure
ros_setup_message = """
rosservice call /move_group/trajectory_execution/set_parameters "config:
doubles:
    - {name: 'allowed_start_tolerance', value: 0.05}"
"""
subprocess.call(ros_setup_message, shell=True)

PandaRobot = Connect4Robot()
```

After setup, it is necessary to define all of the positions that the robot arm will need to visit during calibration and gameplay with labels. This allowed us to call all target positions merely with a string variable name. The positions were as follows: `"LeftCorner"`, `"RightCorner"` (for calibration), `"column_1"`, `"column_2" ... "column_7"` (for gameplay) and `"DiskCollection"` (resting position).

```
# Calibration positions
PandaRobot.define_coordinates([0.3, 0.35, 0.3, pi, 0, pi / 4])

# Initialise the positions the robot has to visit
PandaRobot.AddPosition("DiskCollection",
                    [PandaRobot.x1,
                        PandaRobot.y1 + 0.2,
                        PandaRobot.z1 + 0.1,
                        PandaRobot.roll1,
                        PandaRobot.pitch1,
                        PandaRobot.yaw1])

for i in range(0, 7):
    PandaRobot.AddPosition("column_"+str(i),
                        [PandaRobot.x1,
                            PandaRobot.y1 + PandaRobot.interpolation(i),
                            PandaRobot.z1,
                            PandaRobot.roll1,
                            PandaRobot.pitch1,
                            PandaRobot.yaw1])

PandaRobot.robot_init()
```

1.4 Calibration & Game Setup

Now that the robot has been set up, the physical elements of the game have to be set up and calibrated before the gameplay can begin. Two calibration positions were added that allowed the physical Connect 4 board to be manually aligned with the robot arm. Although it sounds inefficient, this was actually the most reliable way to set up the game under time pressure, leaving more time to debug and test gameplay and motion planning. The calibration sequence could be advanced by pressing Enter, leaving as much time as was needed to position the board correctly.

Note: In the code block below, one might notice that in the highlighted line, a robot position is called that has not been defined above: `PandaRobot.neutral()`. This is actually the same as `PandaRobot.DiskCollection()`, but the `neutral()` position is defined in terms of joint angles, rather than the end-effector position in cartesian space. This is to prevent the robot slowly working itself into a singularity, by resetting the joint angles before each game move. These calls are used interchangeably based on the context.

```
raw_input("Press Enter to move to DiskCollection point...")
PandaRobot.neutral()
raw_input("Press Enter to open gripper...")
PandaRobot.opengrip(simulation =simulation_status)
raw_input("Press Enter to close gripper...")
PandaRobot.closegrip(simulation =simulation_status)
raw_input("Press Enter to move to left corner...")
PandaRobot.MoveToPosition("LeftCorner")
raw_input("Press Enter to continue to right corner...")
PandaRobot.MoveToPosition("RightCorner")
raw_input("Press Enter to continue to game...")
```

Before the game can begin, the final step is to intialise all of the required static variables and variable states.

```
# Set player values for turn counter
PLAYER = 0
BOT = 1

# Set player piece values for board placement
PLAYER_PIECE = 1
BOT_PIECE = 2

# Set game algorithm difficulty (number of moves it looks ahead)
DEPTH = 4 # A higher value takes longer to run

# Initialise game
board = botfunc.create_board()
game_over = False
turn = 0 # Human goes first
```

1.5 Game Loop Breakdown

For the actual demonstration, the Computer Vision element of the project was not linked up to the column input, due to an issue with ROS Networking, so `visionworking = False`. This meant that someone was required to manually type in the column input for the human player's turn (however, this was cross-referenced & verified against the OpenCV output, to simulate a working system).

To avoid the whole loop crashing in the event of a mistyped entry, the input needed to be sanitised:

```
if turn == PLAYER:

        if visionworking == False:

            print("")
            botfunc.pretty_print_board(board)
            print("")

            # Sanitise the input
            while True:
```

(continues on next page)

(continued from previous page)

```
                try:
                    move = int(input("Human (Player 1) choose a column:"))
                except:
                    print("Sorry, I didn't understand that.")
                    continue

                if move not in range(0, 7):
                    print("Sorry you have keyed in a out of bounds column value")
                    continue
                else:
                    col = move
                    break
```

Once the input has been typed, this column value (assigned to `col`) is then passed into functions from the `c4_functions` file (imported as `botfunc`), to complete the piece placement and board state analysis.

```
if botfunc.is_valid_location(board, col):
    row = botfunc.get_next_open_row(board, col)
    botfunc.drop_piece(board, row, col, PLAYER_PIECE)

    if botfunc.winning_move(board, PLAYER_PIECE):
        game_over = True
        botfunc.pretty_print_board(board)
        print("Human Wins!")

    # Advance turn & alternate between Player 1 and 2
    turn += 1
    turn = turn % 2
```

Now that the turn has been advanced, it is the robot's turn to make a move. The minimax game algorithm scans the board state, generates the decision tree, and returns a `col` value relating to the column in which a piece should be placed to play the best possible move. This process is explained in further depth in the Connect 4 Algorithm section. This `col` value is then passed into the same function structure as above. In essence, the game is played and the piece is placed virtually before moving on to the robot arm movement.

```
if turn == BOT and not game_over:

    # Ask Ro-Bot (Player 2) to pick the best move based on possible opponent future␣
↪moves

    col, minimax_score = botfunc.minimax(board, DEPTH, -9999999, 9999999, True)
    print("Ro-Bot (Player 2) chose column: {0}".format(col))

    if botfunc.is_valid_location(board, col):
        row = botfunc.get_next_open_row(board, col)
        botfunc.drop_piece(board, row, col, BOT_PIECE)
        print("")
        botfunc.pretty_print_board(board)
```

Having assigned the required column for the next move, this can also be passed into the function calls for the robot arm movement.

Note: It was decided that the gripper should be manually closed with an Enter command, to minimise the risk of mis-collecting the Connect 4 piece.

```
print("Ro-Bot is currently heading to disk collection point")
# Execute motion sequence

PandaRobot.neutral()
PandaRobot.opengrip(simulation =simulation_status)
raw_input("Press Enter to close gripper...")

PandaRobot.closegrip(simulation =simulation_status)

print("Ro-Bot is currently dropping the piece. Please wait!")
rospy.sleep(0.3)

PandaRobot.MoveToPosition(str(col))
PandaRobot.opengrip(simulation =simulation_status)
PandaRobot.closegrip(simulation =simulation_status)

if botfunc.winning_move(board, BOT_PIECE):
    print("Ro-Bot Wins!")
    game_over = True

# Advance turn & alternate between Player 1 and 2
turn += 1
turn = turn % 2
```

1.6 Final Game Loop

The whole game loop is shown below, for completion:

```
while not game_over:
    if turn == PLAYER:

        if visionworking == False:

            print("")
            botfunc.pretty_print_board(board)
            print("")

            # Sanitise the input
            while True:
                try:
                    move = int(input("Human (Player 1) choose a column:"))
                except:
                    print("Sorry, I didn't understand that.")
                    continue

                if move not in range(0, 7):
                    print("Sorry you have keyed in a out of bounds column value")
                    continue
                else:
                    col = move
                    break

        # Note -  as it was not possible to connect up OpenCV to this input, this␣
↪version of the 'else' code block is NOT final or refined
        else:
```

(continues on next page)

(continued from previous page)

```
            # get new grid state from most recent capture
            vision.GetPositions('updated_gridstate.jpg')
            # analyse new grid state and get co-ordinate of most recent move
            new_move = vision.get_row_and_col(coordinates)
            # take the column index from the co-ordinate list, and assign to col
            col = new_move[1]

        if botfunc.is_valid_location(board, col):
            row = botfunc.get_next_open_row(board, col)
            botfunc.drop_piece(board, row, col, PLAYER_PIECE)

            if botfunc.winning_move(board, PLAYER_PIECE):
                game_over = True
                botfunc.pretty_print_board(board)
                print("Human Wins!")

            # Advance turn & alternate between Player 1 and 2
            turn += 1
            turn = turn % 2

    if turn == BOT and not game_over:

        # Ask Ro-Bot (Player 2) to pick the best move based on possible opponent␣
↪future moves

        col, minimax_score = botfunc.minimax(board, DEPTH, -9999999, 9999999, True)
        print("Ro-Bot (Player 2) chose column: {0}".format(col))

        if botfunc.is_valid_location(board, col):
            row = botfunc.get_next_open_row(board, col)
            botfunc.drop_piece(board, row, col, BOT_PIECE)
            print("")
            #botfunc.print_board(board)
            botfunc.pretty_print_board(board)

            print("Ro-Bot is currently heading to disk collection point")
            # Execute motion sequence

            PandaRobot.neutral()
            PandaRobot.opengrip(simulation =simulation_status)
            raw_input("Press Enter to close gripper...")

            PandaRobot.closegrip(simulation =simulation_status)

            print("Ro-Bot is currently dropping the piece. Please wait!")
            rospy.sleep(0.3)

            PandaRobot.MoveToPosition(str(col))
            PandaRobot.opengrip(simulation =simulation_status)
            PandaRobot.closegrip(simulation =simulation_status)

            if botfunc.winning_move(board, BOT_PIECE):
                print("Ro-Bot Wins!")
                game_over = True

            # Advance turn & alternate between Player 1 and 2
            turn += 1
```

(continues on next page)

(continued from previous page)

```
            turn = turn % 2

    if game_over:
        PandaRobot.neutral()
        print('Game finished!')
```

CHAPTER 2

Getting Started

In order to run the game script, you will have to install ROS and the simulation packages.

2.1 Set Up

2.2 Installing ROS

Note: It is assumed that you do **not** have ROS installed on your Ubuntu OS. The code we use here is tested on ROS Melodic on Ubuntu 18.1 and the following instructions show the installation steps for ROS Melodic.

First, make sure your Debian package index is up-to-date:

```
sudo apt update
```

It is recommened that you do the full ROS desktop installation that comes with: ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators and 2D/3D perception

```
sudo apt install ros-melodic-desktop-full
```

Now should have installed ROS Melodic on your computer as well as Gazebo (for visualising simulations) and RVIZ (for setting up and visualising motion planning)

You will need to install catkin, the ROS build system.

```
sudo apt-get install ros-melodic-catkin python-catkin-tools
```

2.3 Installing Movit

Movit is the motion planning Rviz plugin that can be interacted with in Rviz. To install the prebuilt binaries, type in to your terminal

```
sudo apt install ros-melodic-moveit
```

2.4 Installing Project folders

You would first clone the project repository into your home folder.

```
git clone https://github.com/BartyPitt/RoboticsProject.git
```

Now you have to make sure the submodules are updated.

```
cd RoboticsProject
git submodule sync
git submodule update --init --force --recursive
```

The submodules that you will be downloading are:

- franka_gazebo : Contains 3D models of robot, for collision detection and rendering.
- franka_ros : To use ROS to control the Franka Emika robot
- libfranka : Driver software to control the Franka Emika robot
- moveit_tutorials : Tutorial documentation files for using Movit motion planner(not really necessary)
- panda_moveit_config : Contains demos using Movit motion planner

Some of these submodules have been forked and customised for our particular Franka Emika Robot.

2.5 Compiling all necessary files

You have pulled all the dependencies for gazebo, Rviz but now you need to compile them. Go to your catkin workspace.

```
cd catkin_ws
```

Now you need to compile all the driver code. To do that, in your `catkin_ws` folder

```
catkin_make
```

You will find that it takes a minute or two to build the driver files.

Now you should all be ready to run your simulation code.

CHAPTER 3

Simulating the robot

Before testing on the real robot, it is vital to test your code on a simulator. We use Gazebo to visulise the movement of the Panda Emika robot to see if it behaves properly. Form experience, IF and ONLY if it works in the simulation is there a REMOTE chance that your code will work on the actual robot.

Word of advice: **Simulate, Simulate, Simulate**. Until everything works perfectly in the simulation.

Note that the actual Franka Emika robot costs several thousand Euros, and you will have a very limited time with the actual robot. You can save a lot of time by simulating everything on your computer. By simulating, you can potentially avoid getting the robot to do unexpected things. E.g spin around and smash into the wall behind. It would be a very expensive error.

Here is a video a full game simulation of the robot.

3.1 Starting up your simulation

Open five terminals, navigate to `/catkin_ws` and run the following commands in each:

Run Roscore, the master messaging core

```
roscore
```

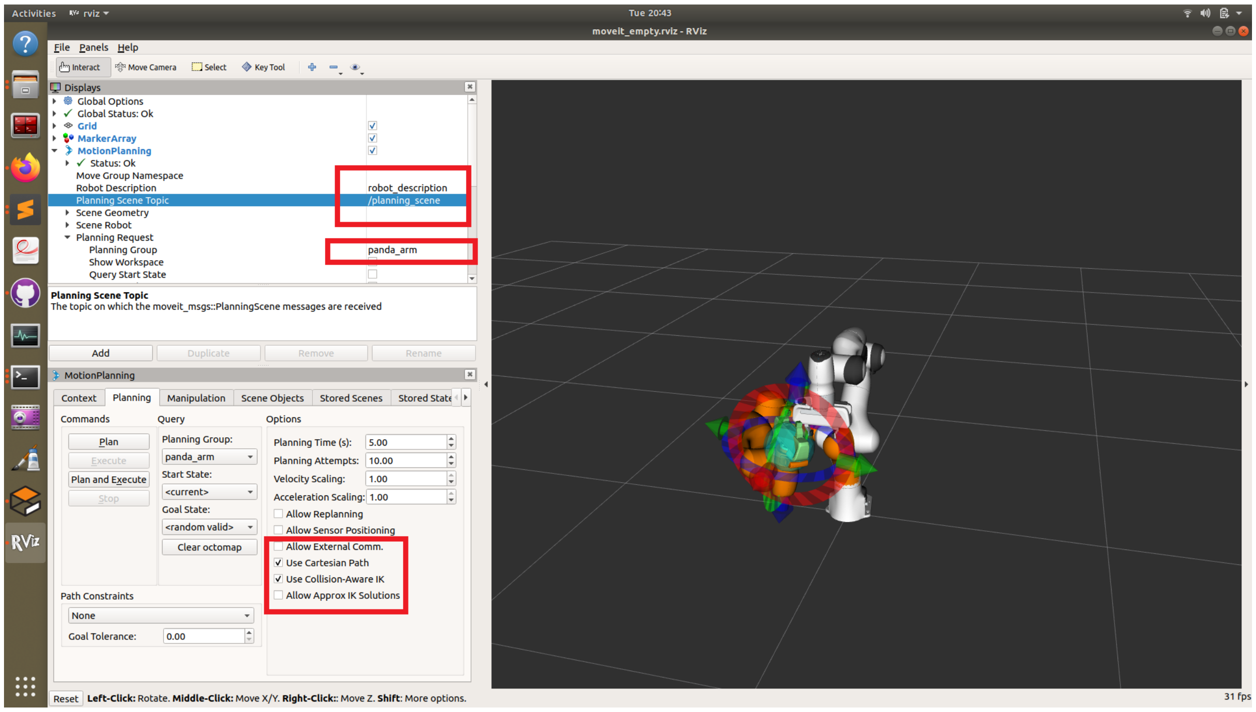
Open up Gazebo, the simulation software

```
source devel/setup.bash
rosrun gazebo_ros gazebo
```

Now open up Rviz, and open up the Movit motion planner plugin used for motion planning.

```
source devel/setup.bash
roslaunch panda_moveit_config demo.launch rviz_tutorial:=true
```

Now you need to activate motion planning using the Movit plugin. Add motion Motion Planning in the `Add` button. Then make sure `Planning Scene Topic` is set to `/planning_scene` . Also, set `Planning Request` to `panda_arm` . Under the `Planning` tab, make sure `Use Cartesian Path` is selected. You can set it up as shown in the image below.



Global Options
Global Status: Ok
Grid ✓
MarkerArray ✓
MotionPlanning ✓
Status: Ok
Move Group Namespace
Robot Description: robot_description
Planning Scene Topic: /planning_scene
Scene Geometry
Scene Robot
Planning Request
Planning Group: panda_arm
Show Workspace
Query Start State

Planning Scene Topic
The topic on which the moveit_msgs::PlanningScene messages are received

Planning Group: panda_arm
Start State: <current>
Goal State: <random valid>

Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
☐ Allow Replanning
☐ Allow Sensor Positioning
☐ Allow External Comm.
☑ Use Cartesian Path
☑ Use Collision-Aware IK
☐ Allow Approx IK Solutions

Path Constraints: None
Goal Tolerance: 0.00

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift**: More options.

Run the ‘’panda_publisher.py” utility that broadcasts movement of the joints and gripper such that gazebo knows that it has moved.

```
cd src/panda_publisher
python panda_publisher.py
```

Finally, spawn the robot arm in Gazebo.

```
source devel/setup.bash
roslaunch franka_gazebo panda_arm_hand.launch
```

3.2 Running Connect Four game code

Now that we have the simulation setup, we can run the code to move the robot and play the Connect Four game. We need to return to the home directory, `\RoboticsProject`, and open a new terminal. We then navigate to `franka_main`, where our python script to move the robot and play the game is stored. In your terminal opened in `\RoboticsProject`:

```
cd Franka_ws
python main.py
```

Now you should be able to see the game startup on your terminal. It will give your instructions to help you position the Connect Four board right under the robot’s gripper. More instructions on playing the game will be in the next page.

3.3 Simulation setup screencast

For your reference, here is video showing the whole setup operation that will allow you to run a full simulation.

CHAPTER 4

Robot Motion

The primary function of the robot was to pick up the Connect four game token and drop it into the right column. Some of the motions carried out by the robot can be seen in this video

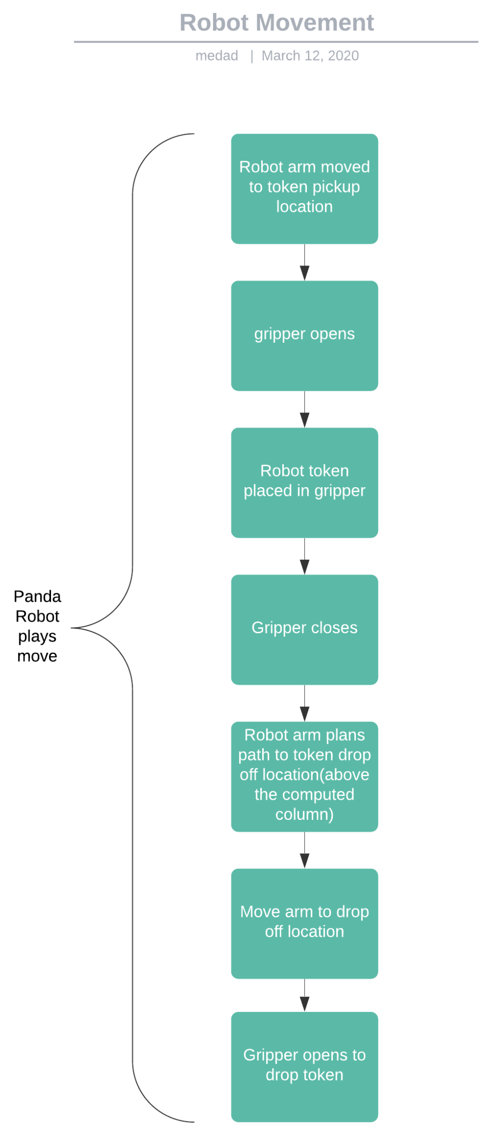
Here is an overview of the robot's motion when it plays a move

A separate python script was created which contained the robot class with methods related to its motion. The class methods could be called to move the robot to a cartesian point, open and close the gripper and calibrate the robot's position. Keeping the methods abstracted away in another file enabled us to keep the main python script clean and legible. The following in a breakdown of the methods within this Connect4Robot class.

4.1 Init

When an instance of the Connect4Robot is created, the method init() is automatically called. This defines the variables for opening and closing the gripper, which are used in their related methods. 2 groups are also created. These are "group", which includes all the joints in the arm of the robot and "group2", which includes the joints in the gripper. These are used by the moveit_commander library for moving the robot. Finally, a dictionary is initialised, which will be used for storing position names and cartesian coordinates.

```
def __init__(self, GripperSizeExtended=0.03, GripperSizeRetracted=0, group=moveit_
↪commander.MoveGroupCommander(
    "panda_arm"), group2 = moveit_commander.MoveGroupCommander("hand")):
    ''' Sets up the Inital setup conditions for the robot.
    '''
    self.GripperSizeRetracted = GripperSizeRetracted
    self.GripperSizeExtended = GripperSizeExtended
    self.group = group # All joints apart from the grippers
    self.__positions__ = dict()
    self.group2 = group2 # Gripper joints
```


Robot Movement
medad | March 12, 2020
Panda Robot plays move
Robot arm moved to token pickup location
gripper opens
Robot token placed in gripper
Gripper closes
Robot arm plans path to token drop off location(above the computed column)
Move arm to drop off location
Gripper opens to drop token

4.2 Calibration

The calibration method has 2 purposes. The first is to check that the robot is operating correctly, which is done by making it move to a position and then open and close its gripper. The second is to enable the connect 4 board to be positioned correctly in the real world. This is done by making the Panda robot move its end effector to above where the 1st column on the board should be. Once the user has aligned the board beneath it, they should press Enter, and the end effector will move above the last column on the board. Since the exact height above the board is not important, this is enough to enable the board to be correctly positioned.

```
def Calibration(self):
    ''' Calibration function to align board and test robot '''
    raw_input("Press Enter to move to DiskCollection point...")
    self.MoveToPosition("DiskCollection")
    raw_input("Press Enter to open gripper...")
    self.opengrip()
    raw_input("Press Enter to close gripper...")
    self.closegrip()
    raw_input("Press Enter to move to left corner...")
    self.moveto([self.x1, self.y1, self.z1, self.roll1, self.pitch1, self.yaw1])
    raw_input("Press Enter to continue to right corner...")
    self.moveto([self.x2, self.y2, self.z2, self.roll2, self.pitch2, self.yaw2])
    raw_input("Press Enter to continue to game...")
```

4.3 Define coordinates

This method enables us to reposition the board if we need to, as long as it remains perpendicular to the robot. We define where the left corner is going to be (as seen by the robot), and the right corner is automatically calculated. The coordinates of the left and right corners are then created as attributes so that all other positions in cartesian space can be defined relative to the board, and will auto-update if we change the location of the board. Being able to reposition the board is important so that we can test different places in the robot's task space which lead to more reliable motion planning.

```
def define_coordinates(self, LeftCorner, dx=0, dy=-0.468, dz=0):
    '''Defines top left corner of board (from pov of robot) relative to the robot and␣
↪moves to calibration points'''
    [x, y, z, roll, pitch, yaw] = LeftCorner
    RightCorner = [x + dx, y + dy, z + dz, roll, pitch, yaw]
    self.__positions__["LeftCorner"] = LeftCorner
    self.__positions__["RightCorner"] = RightCorner

    [self.x1, self.y1, self.z1, self.roll1, self.pitch1, self.yaw1] = LeftCorner
    [self.x2, self.y2, self.z2, self.roll2, self.pitch2, self.yaw2] = RightCorner
```

4.4 AddPosition

This function is designed to store a coordinate in cartesian form in a private dictionary. It originally stored the variables in the form of a Moveit Pose class, however this was later changed, as it is very difficult to both view the values as well as making it very difficult to modify the values. The function remained partially to interact with legacy code, and partially as it was thought it might be useful to add in a sanitization layer.

```
def AddPosition(self , PositionName , PositionCordinates):
                '''A setter function that sets up the positions for the robot to␣
↪travel to'''
                self.__positions__[PositionName] = PositionCordinates
```

4.5 Interpolation

This function was used to generate the coordinates of the columns. Interpolation was used as a method to avoid hard coding the column coordinates individually, and is used when the AddPosition method is called in the main function.

```
def interpolation(self, column):
ydistance = (self.y2-self.y1)/6 * (column-1)
return self.y1 + ydistance
```

4.6 Move to

This is a movement function that uses the moveit motion planner to move the robot. It takes in a position in cartesian list form, transforms it into the pose class, and then runs it directly through the motion planner. It then executes the plan.

```
def moveto(self, Position):
    '''Moves to a given position, in form [x,y,z,roll,pitch,yaw]'''

    # print("Moving to: ({},{},{}) with angle ({:.2f},{:.2f},{:.2f})".
↪format(*Position))

    # Converting the roll, pitch, yaw values to values which "moveit" understands
    pose_goal = self.CordinatesToPose(Position)

    self.group.set_pose_target(pose_goal)  # Set new pose objective
    plan = self.group.go(wait=True)  # Move to new pose
    rospy.sleep(0.5)
    # It is always good to clear your targets after planning with poses.
    self.group.clear_pose_targets()
```

4.7 Coordinates to pose

The human-legible cartesian position coordinates and rotations (x, y, z, roll, pitch, yaw), must be passed into a class for moveit to be able to interpret them. This starts by converting roll, pitch and yaw angles into quaternions and then converting these orientations as well as the Cartesian positions into a format understood by the moveit_controller library.

```
def CordinatesToPose(self, Position):
    '''Takes in a cordinate and transforms it into a pose'''
    x, y, z, roll, pitch, yaw = Position
    quaternion = tf.transformations.quaternion_from_euler(roll, pitch, yaw)

    pose = geometry_msgs.msg.Pose()
    pose_o = pose.orientation
```

(continues on next page)

(continued from previous page)

```
    pose_o.x, pose_o.y, pose_o.z, pose_o.w = quaternion
    # Defining target coordinates
    pose.position.x = x
    pose.position.y = y
    pose.position.z = z
    return pose
```

4.8 Move To Position

The function takes the name of a position and moves the robot to that position. It enabled us to feed in the position names defined in main.py.

```
def MoveToPosition(self ,Position):
    '''Takes the name of the position and moves the robot to that position.'''
    Cordinates = self.__positions__[Position]
    self.moveto(Cordinates)
```

4.9 Move joints

This is the command for direct joint control of the robot. For the most part the use of motion planners and inverse kinematics was preferred for this project. Most of the motion planning was done with the moveto() and the MoveToPosition() commands. This function was added so that after every run the robot could head to a set joint position, the idea behind this being that it stopped the robot from gradually working its way into a singularity, something that would happen within the simulations.

```
def movejoints(self, jointAngles):
    '''Takes in joint angles and moves to that pose'''
    joint_goal = self.group.get_current_joint_values()
    joint_goal = jointAngles
    self.group.go(joint_goal, wait=True)
    self.group.stop()
```

4.10 Neutral

Throughout the game the robot would slowly work itself into a singularity position after various successive moves, which meant it would become unable to move. In order to avoid this, a reset stage was required that would reconfigure the robot joints to a specific position after each move. Neutral() is a method which achieves this. It instructs the robot to move into a particular set of joint positions which orient it off to the side of the board. This method can be called after each time the robot plays a move, and can be used as the position from which it collects a disk.

```
def neutral(self):
    ''' Moves to disk collection position using joint angles.
        Joint angles used so that the robot doesn't work itself into singularity. '''

    self.movejoints([0.963,0.264,0.117,-1.806,-0.035,2.063,0.308])
```

4.11 Cartesian Path

Cartesian path is a function that takes in an Endposition for the robot to move to and uses the compute_cartesian_path() function to generate a cartesian path between the two. This function was useful, since for the most part it would keep the robot end effector along an easily predictable path. This gives much more stability than moveto(). The main difference between the two functions other than the motion planning is that Cartesian Path returns a true or false depending on weather or not it was successful.

```
def CartesianPath(self, Endposition , StartPosition = None , max_tries = 10):
        '''Takes an Endpositions and generates and then acts on a motion plan to the␣
↪Endposition using compute cartesian path. '''
        if StartPosition:
                StartPosition = self.CordinatesToPose(StartPosition)
        else:
                StartPosition = group.get_current_pose().pose

    Endposition = self.CordinatesToPose(Endposition)

    waypoints = []
        # start with the current pose
    waypoints.append(StartPosition)


    waypoints.append(Endposition)
    for i in range(max_tries):
                (plan, fraction) = group.compute_cartesian_path (
                                                            waypoints,   #␣
↪waypoint poses
                                                            0.01,        # eef_
↪step
                                                            0.0,         # jump_
↪threshold
                                                            True)        # avoid_
↪collisions
                if fraction == 1:
                        print("Motioned Planned Successfully")
                        break
    else:
                print("failed to run")
                return False

    self.group.execute(plan , wait = True)
    self.group.clear_pose_targets()
    return True
```

4.12 Robot Initialisation

Standard procedure, to clear the current targets to avoid conflicts.

```
def robot_init(self):
    ''' Clears targets, good to do after planning poses '''
    self.group.clear_pose_targets()
```

4.13 Gripper Control

We had two options for controlling the gripper, one by using movit commander's `go(joint_goal, wait=True)` function to move the gripper to the target location and using the `GraspGoal(width=0.015,speed=0.08, force=1)` function. Each had its drawbacks.

4.14 Using GraspGoal() function

When picking up the ConnnectFour token, ideally we would control both the position of gripper as well as the force it exerts. We do not want to exceed the maximum force that the gripper can produce, but we must ensure the token doesn't fall off due to a lack of force. We therefore tried using the `GraspGoal(width,speed,force)` function to set the gripper in place and exert a force on the token such that it did not fall off. However, we discovered that the gripper would grip the token, and then release its grip as soon as the `closegrip()` function came to an end. We could not figure out why it kept relaxing its grip.

```
from franka_gripper.msg import GraspAction, GraspGoal

    def closegrip(self, simulation=False, GripOveride=None):
        rospy.init_node('Franka_gripper_grasp_action')
        client = actionlib.SimpleActionClient('/franka_gripper/grasp', GraspAction)
        rospy.loginfo("CONNECTING")
        client.wait_for_server()
        action = GraspGoal(width=0.015,speed=0.08,force=1)
        rospy.loginfo("SENDING ACTION")
        client.send_goal(action)
        client.wait_for_result(rospy.Duration.from_sec(5.0))
        rospy.loginfo("DONE")
```

4.15 Using go() function

What worked in the end was DIRECTLY setting the gripper position to the fully closed postion by setting both gripper's position to `0` (fully closed). The gripper exerted a sufficient force to prevent the token from falling off. However, there was a good chance of failure when using this method. We set the gripper's position to `0` despite the connect 4 token getting in the way of the gripper fully closing. The robot could have thrown an error as the connect 4 token obstacle was getting in the way of the gripper fully closing, preventing it from going to the fully closed `0` position. However we discovered that due to the small size of the token and the flexiblity of the gripper pads, the grippers could close fully without detecting the connect 4 token obstacle.

The code for closing the gripper is as follows

```
def closegrip(self, simulation=False, GripOveride=None):
    ''' Function to open the grip of the robot '''
    joint_goal = self.group2.get_current_joint_values()
    joint_goal[0] = 0.0
    joint_goal[1] = 0.0
    self.group2.go(joint_goal, wait=True)
    self.group2.stop()
    if simulation == True:
        # For Gazebo simulation
        if GripOveride == None:
            GripOveride = self.GripperSizeExtended
        gripper_publisher = rospy.Publisher('/franka/gripper_position_controller/
↪command', Float64MultiArray,queue_size=1)
```

(continues on next page)

(continued from previous page)

```
        gripper_msg = Float64MultiArray()
        gripper_msg.layout.dim = [MultiArrayDimension('', 2, 1)]
        gripper_msg.data = [GripOveride, GripOveride]
        gripper_publisher.publish(gripper_msg)
        rospy.sleep(0.5)
```

Note: we have a separate function that broadcasts the gripper position to ROS. This is to ensure Gazebo sees the movement and displays accordingly. We create a `gripper_publisher` that publishes the new gripper position to the `/franka/gripper_position_controller/command` topic so that Gazebo can be updated.

CHAPTER 5

Motion planning

We used the default motion planner in Movit, which uses the **Open Motion Planning Library**,OMPL,an open-source motion planning library that primarily implements randomized motion planners. It was used to plan the movement of robot gripper between locations such as from the token pick up point to the drop off point above the column.

Occasionally we observed that the planner seemed to plan a circuitous plan for the robot, or occasionally spinning around to reach its destination. We minimised the risk of this happening by adding additional waypoints.

5.1 Collision Avoidance

The primary obstacle in the robot's workspace was the Connect 4 board. We had to tell Movit, the motion planner, to avoid getting any part of the robot to collide with the board.

Two 3D models were required for collision detection. The 3D model of the robot, including all its links and of the connect 4 board.

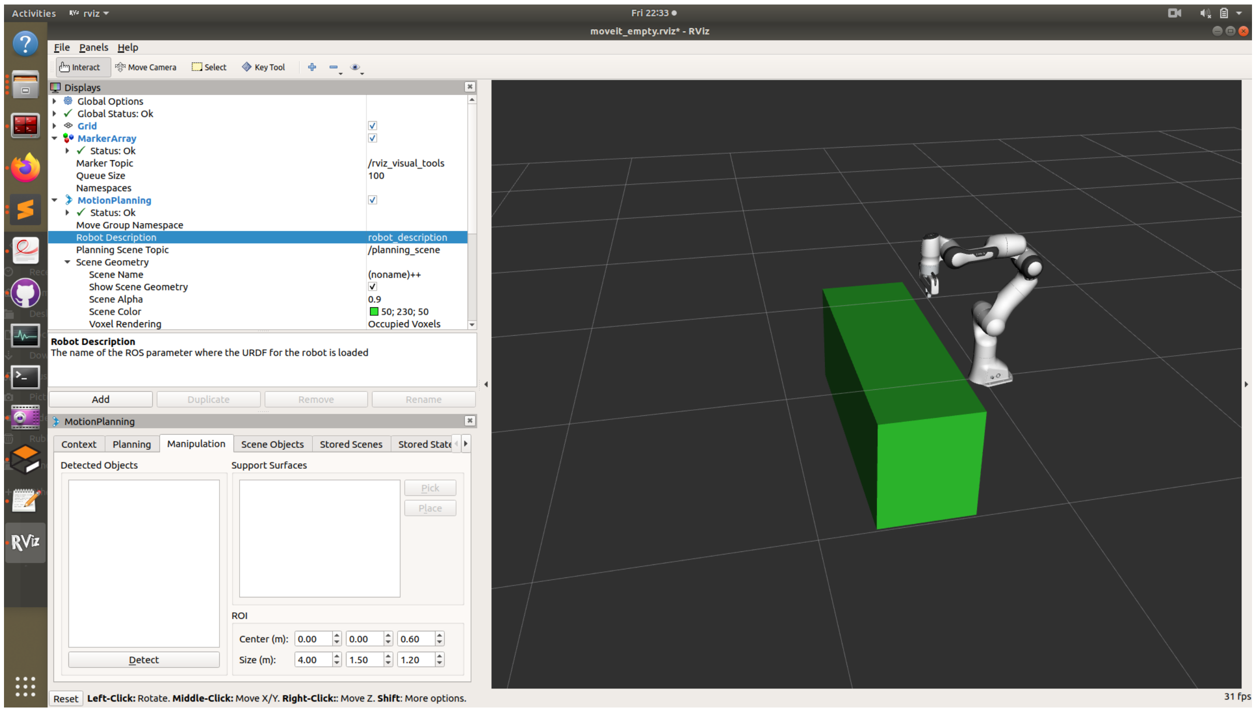
The 3D models of the model was supplied by the robot manufacturer in the form of .dae mesh files. These files can be found in `RoboticsProject\catkin_ws\src\franka_gazebo\meshes\visual`.

Note: The .dae mesh files are a lower resolution polygon mesh file compared to the .stl files supplied for visualisation. This is done to reduce computation load for collision detection.

In order to simplify computation, we defined the connect 4 board obstacle solely by a bounding box, a box that completely encompassed the volume of the connect 4 board.

For a comparison between the 3D models for visualisation collision detection, here is a photo of the 3D models of the connect 4 board. On the left is the bounding box obstacle used for collsion detection(rendered with Rviz) while on the right is the higher resolution model of the connect four board(rendered in Gazebo).

We could define the dimension, position and orientation of the obstacle with a few lines of code in the intial part of the main script running the game code. We define the position, orientation and dimensions of the box and pass that into the `scene.add_box(obstacle_name, pose, dimensions)` function that tells the motion planner the obstacle information.

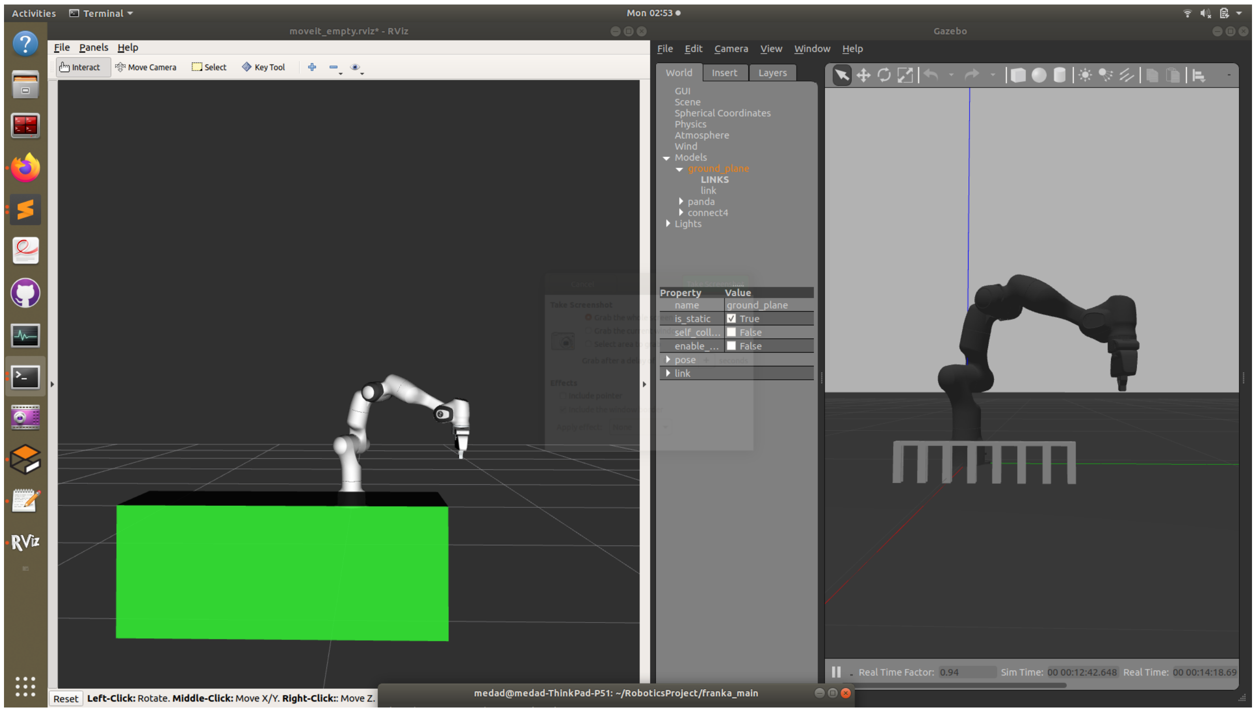

Activities
rviz
Fri 22:33
moveit_empty.rviz* - RViz
File Panels Help
Interact Move Camera Select Key Tool
Displays
Add Duplicate Remove Rename
MotionPlanning
Context Planning Manipulation Scene Objects Stored Scenes Stored State
Detected Objects
Support Surfaces
Pick
Place
Detect
ROI
Reset
31 fps

Global Options	
Global Status: Ok	
Grid	✓
MarkerArray	✓
Status: Ok	
Marker Topic	/rviz_visual_tools
Queue Size	100
Namespaces	
MotionPlanning	✓
Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	/planning_scene
Scene Geometry	
Scene Name	(noname)++
Show Scene Geometry	✓
Scene Alpha	0.9
Scene Color	🟩 50; 230; 50
Voxel Rendering	Occupied Voxels

Robot Description
The name of the ROS parameter where the URDF for the robot is loaded

Center (m):	0.00	0.00	0.60
Size (m):	4.00	1.50	1.20

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift**: More options.



Property	Value
name	ground_plane
is_static	✓ True
self_coll...	False
enable_...	False
pose	
link	

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z.

Real Time Factor: 0.94 Sim Time: 00 00:12:42.648 Real Time: 00 00:14:18.69

```
p = geometry_msgs.msg.PoseStamped()
p.header.frame_id = robot.get_planning_frame()
p.pose.position.x = 0.4
p.pose.position.y = -0.301298
p.pose.position.z = -0.2
p.pose.orientation.x =  0.0
p.pose.orientation.y = 0
p.pose.orientation.z = 0.0
p.pose.orientation.w = 0.4440158
scene.add_box("box", p, (0.5, 1.5, 0.6))
rospy.sleep(2)
```

Note: When you run the script to insert the box, you will notice that the obstacle(in green) does not immediately appear in the Rviz workspace GUI. The add_box method adds the object asynchronously so that RViz is not notified. The way to visualise this is to REMOVE the `motion planning` branch from the `display` tree in Rviz and add it back. Then the obstacle will appear. Note that you must make sure you have run scene.add_box method earlier. More information can be found in this question

CHAPTER 6

Open CV Overview

OpenCV is used to allow the robot to update the state of the board and know where the human has played without requiring any inputs. It is also used for error prevention and recovery, as will be further explained.

6.1 System Overview for Counter position detection:

This code is called by the decision making algorithm when it needs to know the current position of the placed counters.

In the game of connect 4 counters are placed into columns inside of the grid. One of the major tasks for machine vision is figuring out the current position of the counters in play.

It then needs to find the new counter that has been placed in the board by the human.

Currently the algorithm for counter detection can be split into five separate stages:

1. The webcam currently attached to the system takes a picture of the board in its current state.

2. The machine vision detects the four corners of the connect 4 grid. White Masks:

3. The system warps the image so that just the connect 4 grid is shown in a planer projection.

4. The image detects the position of all the counters in play and calculates in which column each of the counters falls into. New disk:

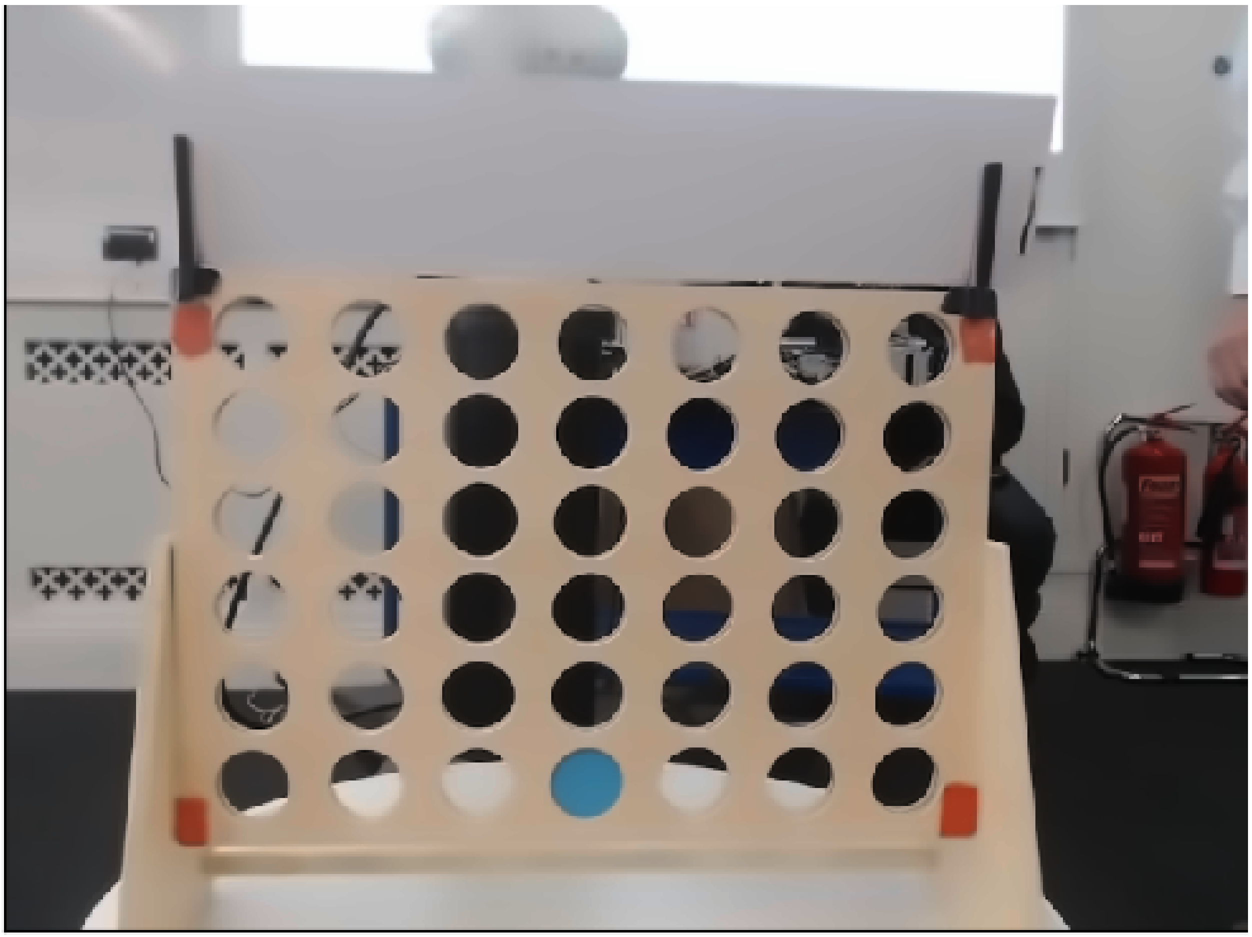
5. The location of all the counters is returned in the form of a numpy array. This is saved as board1

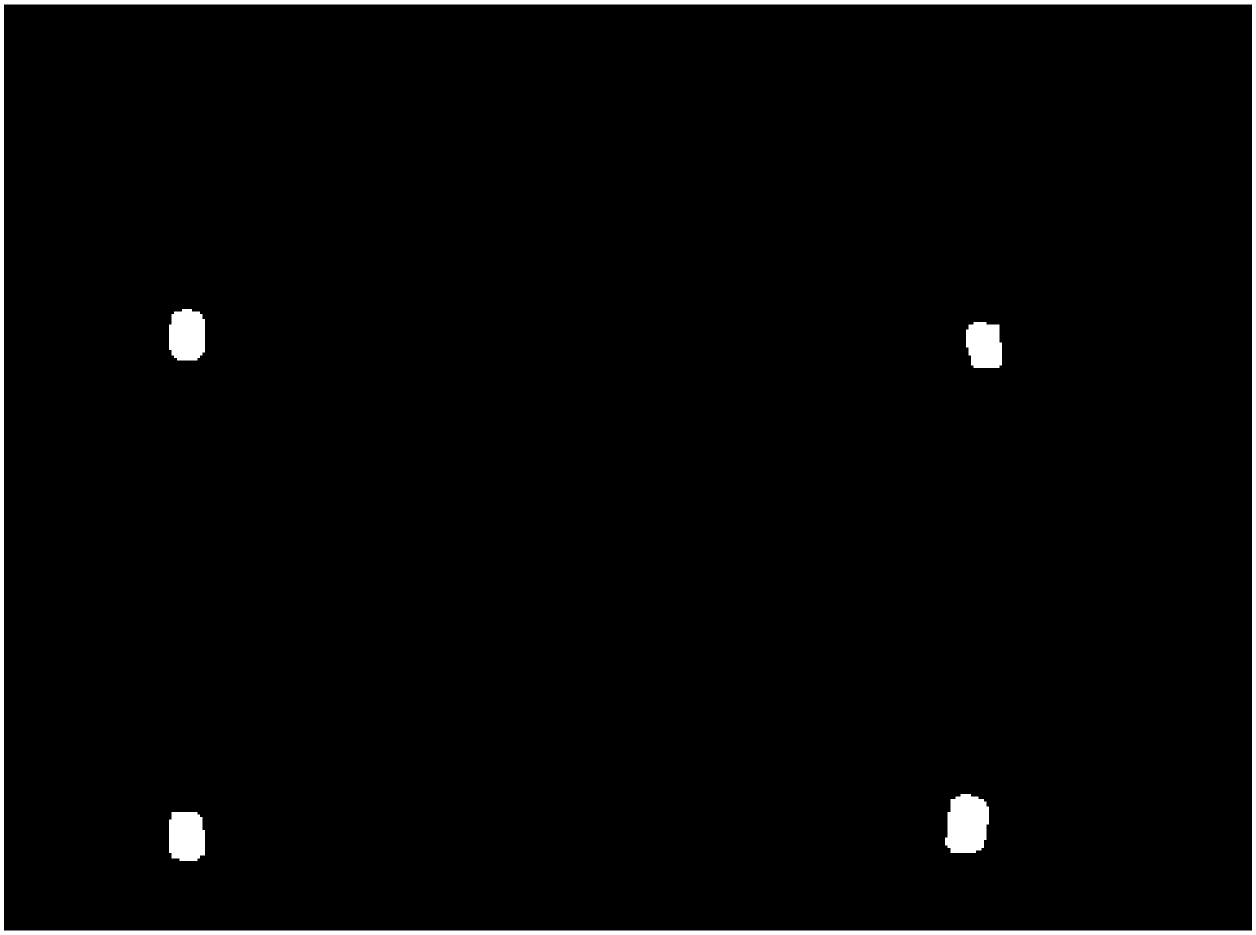
These steps are then repeated again. This outputs another board state in the form of a numpy array, saved as board2. Then board1 is subtracted from board2 to find where the new counter is.

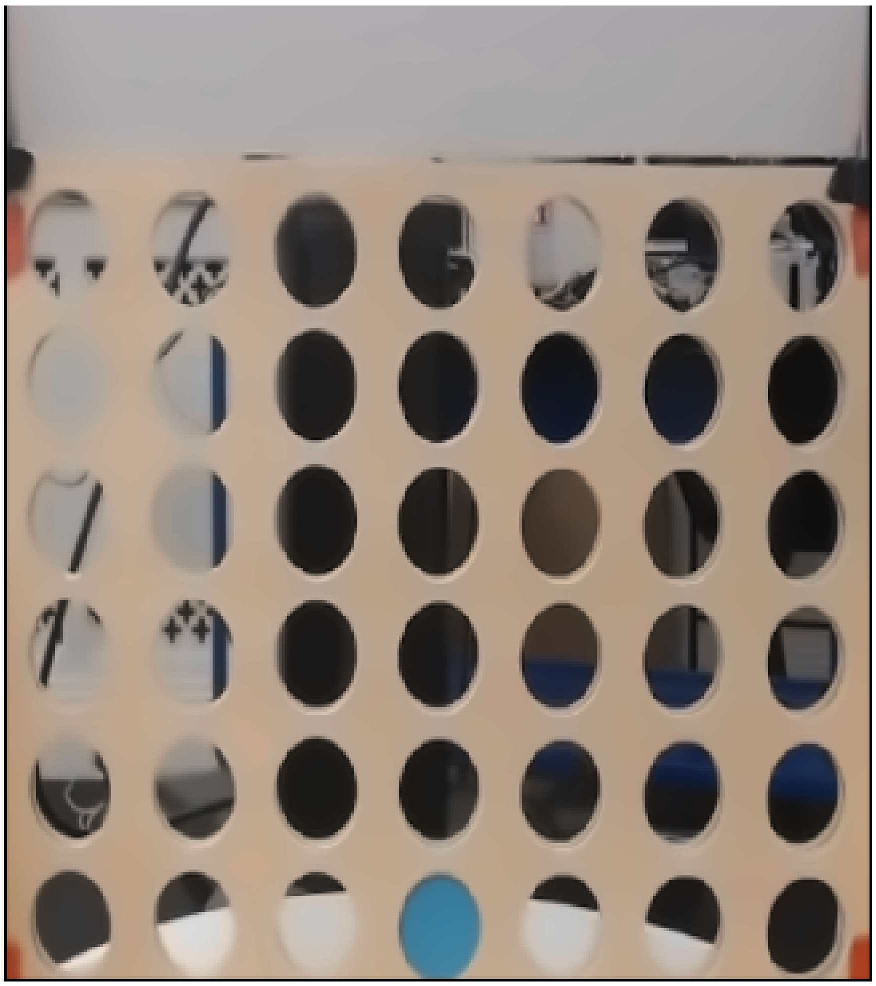
The column and the row of the new counter are found and the column is returned to the connect 4 playing algorithm to continue the game.

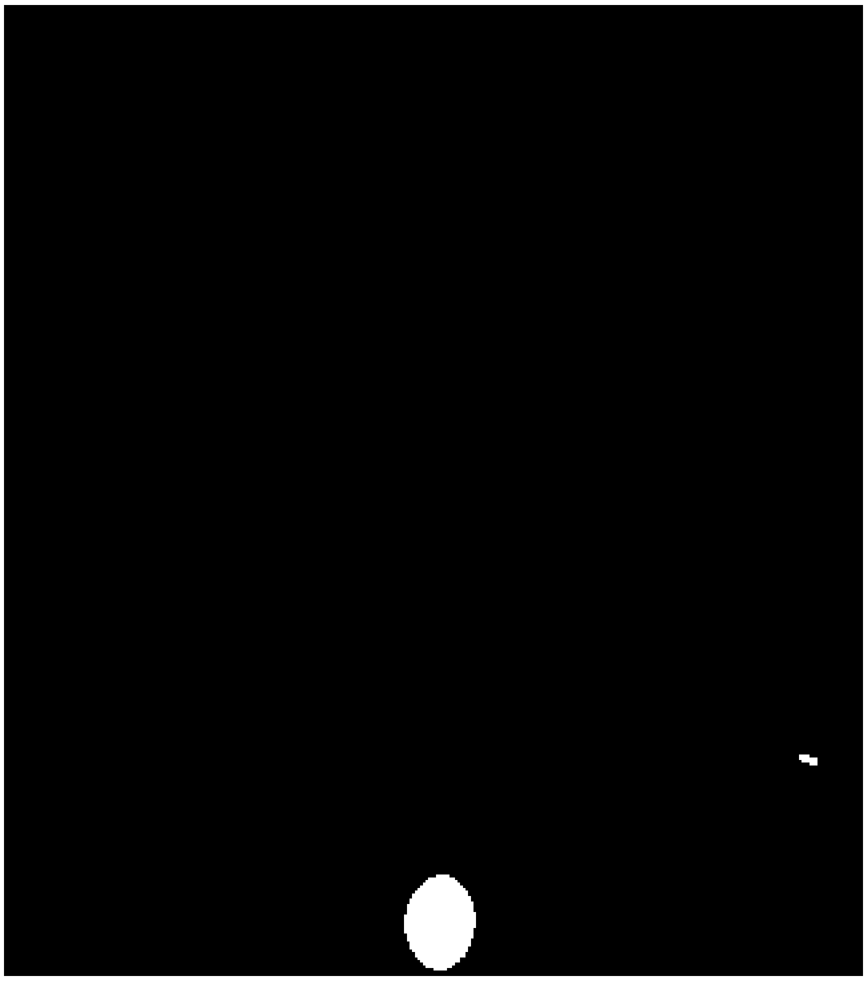
6.2 How the code works:

1. Taking a snapshot with the webcam, and then putting it through the position finding algorithm:









```
def SnapShotAndPossition():
    camera = cv2.VideoCapture(0)
    for i in range(10):
        __, frame = camera.read()
    frame = cv2.fastNlMeansDenoisingColored(frame,None,10,10,7,21)
    ImageInlineShow(frame)
    Board = GetPossitions(frame , Location = False)
    camera.release()
    return Board
```

Where the position finding algorithm is the following:

```
def GetPossitions(img ,Location = True):
    if Location:
         img = cv2.imread(img)

    SquareImage = TransformTheImage(img,200)
    #If the Extra space at the top starts causing problems.

    #The Blue mask
    lower_blue = np.array([90,130,80])
    upper_blue = np.array([115,255,255])

    blueContours, __ = ConvectionFunction(SquareImage,[lower_blue] , [upper_blue])
    blue_coordinates = ContourInfo(blueContours , 800)

    #The Yellow Mask
    lower_yellow = np.array([20,55,70])
    upper_yellow = np.array([45,191,200])

    yellowContours, __ = ConvectionFunction(SquareImage,[lower_yellow], [upper_
↪yellow])
    yellow_coordinates = ContourInfo(yellowContours , 800)

    mergedy = get_row_and_col(get_x_and_y_coord_from_contours(yellow_coordinates))
    mergedb = get_row_and_col(get_x_and_y_coord_from_contours(blue_coordinates))
    Board = ArrayfromCordinates(mergedb,mergedy)
    return disks_to_array(Board)
```

It reads an image from the given Image Location, flattens it, finds the yellow and the blue disks, and returns the rows and columns of each of the disks.

3. From the pixel coordinates of the disks, finding the row and column that they fall into: This fucntion takes the pixel coordinates in the form of [cX ,cY], and returns the row for cX, and column for cY

```
def get_row_and_col(coordinates):
Tolerance = 20 #a tolerance is added to check the coordinate in the row/column are␣
↪within a range.
xList = []
yList = []
KeyX = [55 , 155 , 250 , 345 , 450 , 545 , 640] #these are the estimated pixels in␣
↪which the coordinates for each column lie in
KeyY = [200 , 330 , 430 , 530 , 650 , 760] #these are the estimated pixels in which␣
↪the coordinates for each row lie in
for i in coordinates:
    y_coord = i[1]
    x_coord = i[0]
```

(continues on next page)

(continued from previous page)

```
    for n,x in enumerate(KeyX):
        if abs(x_coord - x) < Tolerance: #a tolerance is added to check the␣
↪coordinate in the row/column are within a range.
            xList.append(n)
            break
    else:
        print("x out" , x_coord)
        pass
    for n,y in enumerate(KeyY):
        if abs(y_coord - y) < Tolerance:
            yList.append(n)
            break
    else:
        xList.pop(-1)
        print("Y out" , y_coord)
return [cord for cord in zip(xList , yList)]
```

4. Converting the board into a numpy array: This function takes in the positions of all the disks on the board and returns a numpy array with -1 for the bot disks and 1 for the player disks

```
def disks_to_array(board):
    for x in np.nditer(board, op_flags=['readwrite']):
        if x[...] == 1:
            x[...] = -1
        if x[...] == 2:
            x[...] = 1
    return board
```

5. Finding the newly placed disk by the human: This function takes in the board state before the human plays (board1) and after they play (board2), and subtracts them from each other. Where the result is not 0 it returns the column and row of that position, which is where the new disk has been played

```
def where_is_the_new_disk(board1, board2):
    board_before = disks_to_array(board1)
    board_after = disks_to_array(board2)
    result = np.subtract(board_before, board_after)
    for x in np.nditer(result):
        if x[...] != 0:
            i, j = np.where(result != 0)
    return i, j #i is row, j is col
```

6. The column of the newly placed disk by the human is returned to the connect 4 playing algorithm.

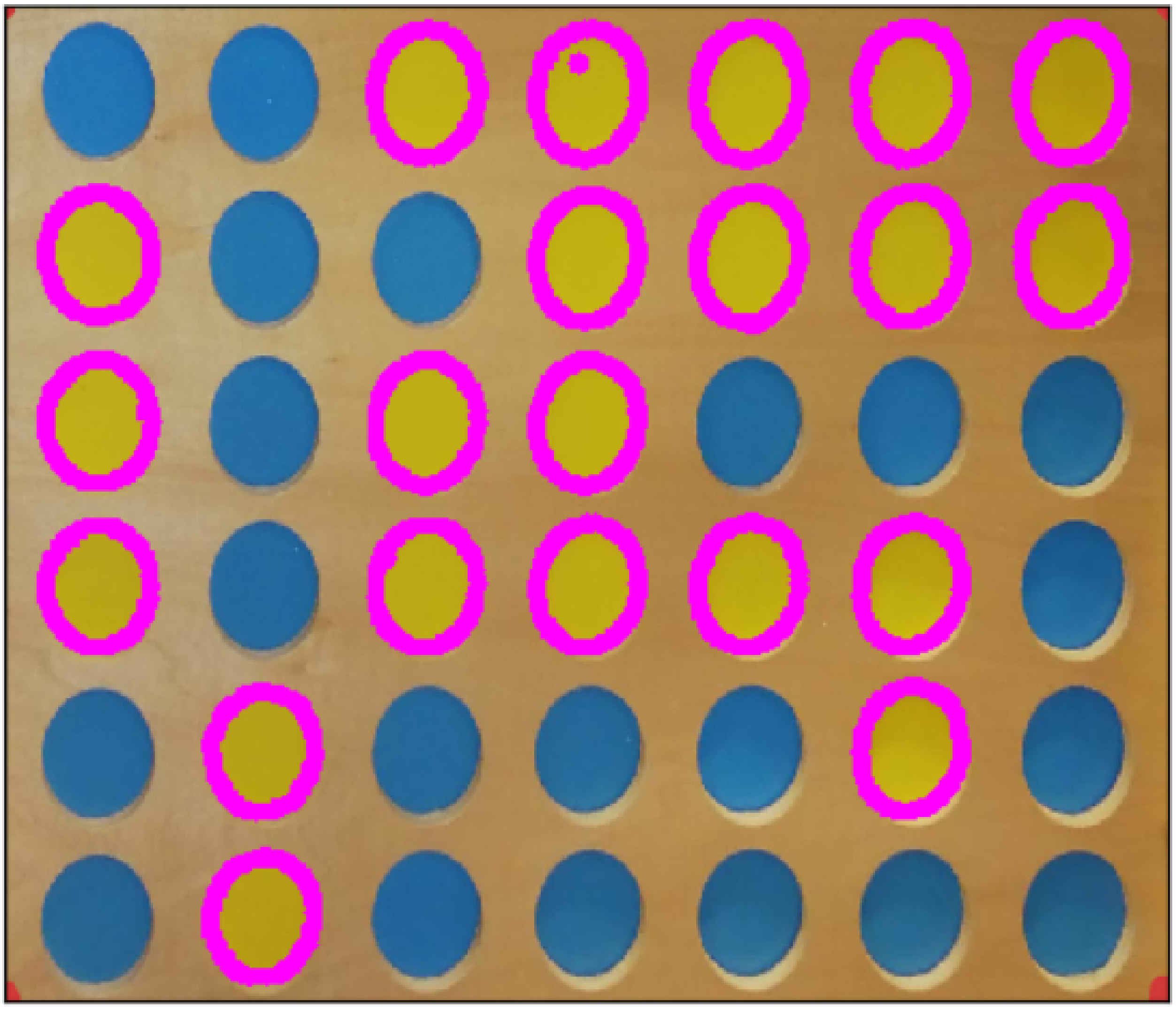
6.3 Error detection with OpenCV:

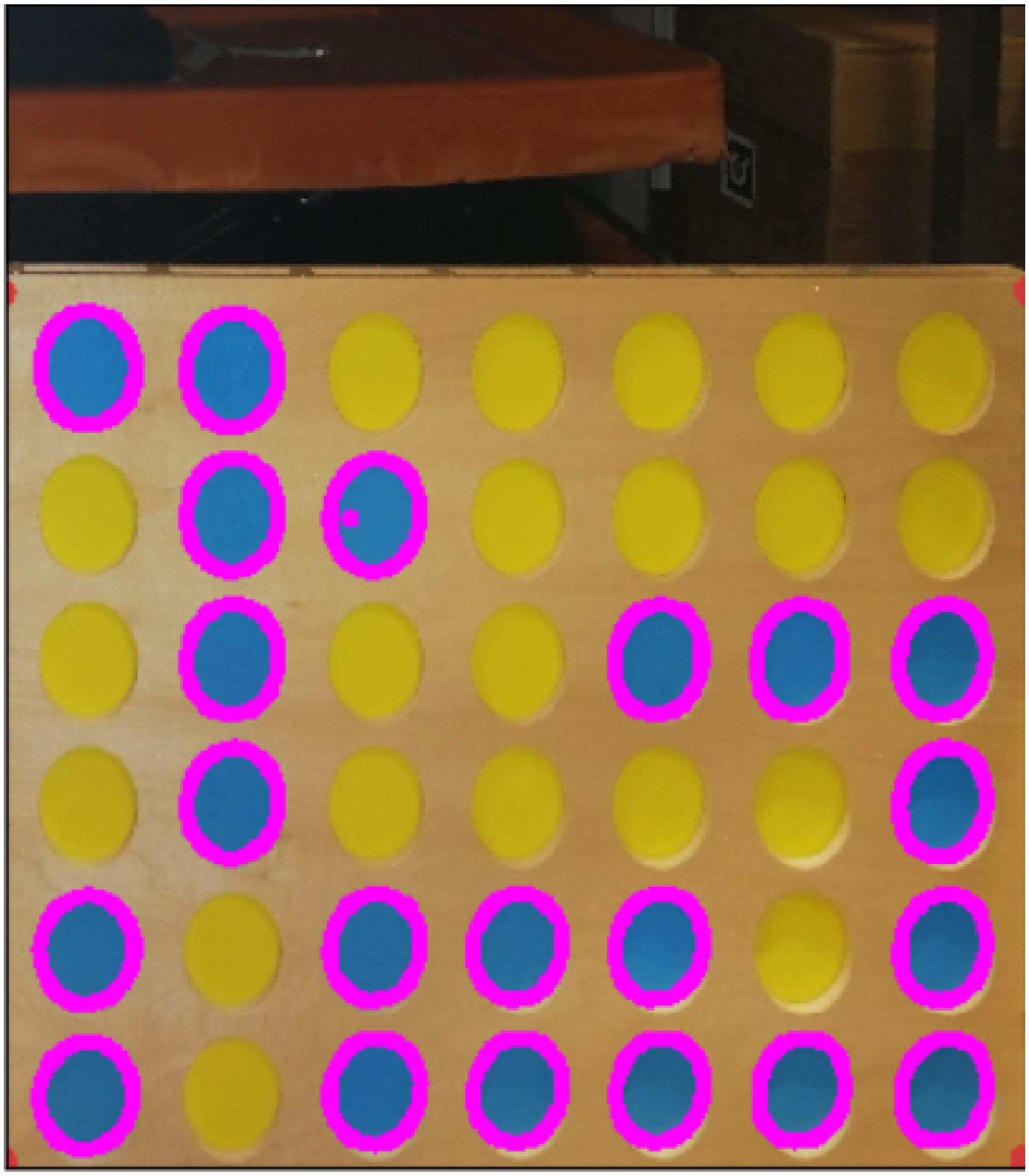
1. One method used to check for errors was to run the code to find the position of the newly played disk by the human twice, and ensure that in both instances the result is the same. Otherwise, repeat the same procedure until the new disk position is the same.

Firstly the yellow disks are found with the `GetPossitions` function mentioned above,

Then the blue disks, also found with `GetPossitions`:

Then the centre coordinates are extracted with the following function:





```
def get_x_and_y_coord_from_contours(coordinates):
    counter = -1
    coords = []
    for i in coordinates:
        counter += 1
        y_coord = coordinates[counter][0][1]
        x_coord = coordinates[counter][0][0]
        coords.append([x_coord, y_coord])
    return coords
```

And then the row and column of each of the coordinates is found using this function:

```
def get_row_and_col(coordinates):
    Tolerance = 20 #a tolerance is added to check the coordinate in the row/column␣
↪are within a range.
    xList = []
    yList = []
    KeyX = [55 , 155 , 250 , 345 , 450 , 545 , 640] #these are the estimated pixels␣
↪in which the coordinates for each column lie in
    KeyY = [200 , 330 , 430 , 530 , 650 , 760] #these are the estimated pixels in␣
↪which the coordinates for each row lie in
    for i in coordinates:
        y_coord = i[1]
        x_coord = i[0]
        for n,x in enumerate(KeyX):
            if abs(x_coord - x) < Tolerance:
                xList.append(n)
                break
        else:
            print("x out" , x_coord)
            pass
        for n,y in enumerate(KeyY):
            if abs(y_coord - y) < Tolerance:
                yList.append(n)
                break
        else:
            xList.pop(-1)
            print("Y out" , y_coord)
    return [cord for cord in zip(xList , yList)]
```

After this, the coordinates are used to make a numpy array:

```
def ArrayfromCordinates(Cordinates1, Cordinates2 = None):
    output = np.zeros((7,6))
    for co in Cordinates1:
        y,x = co
        output[y][x] = 1
    if Cordinates2 == None:
        return output
    for co in Cordinates2:
        y,x = co
        output[y][x] = 2
    return output
```

And then by taking multiple snapshots and performing this procedure twice the newly placed disk can be found:

```
def where_is_the_new_disk(board1, board2):
```

(continues on next page)

(continued from previous page)

```
    board_before = disks_to_array(board1)
    board_after = disks_to_array(board2)
    result = np.subtract(board_before, board_after)
    for x in np.nditer(result):
        if x[...] != 0:
            i, j = np.where(result != 0)
    return i, j #i is row, j is col
```

By running this code twice and ensuring that on both instances the disk position found is the same, it can then be verified that the board has been updated to the correct state.

2. Another method that was developed for error recovery but never implemented, was to draw a centre line on the column that the robot was about to place the disk in, and using a red marker on the centre of the robot gripper, check that the coordinate of the marker on the gripper is aligned with the target column.

First the top and the bottom coordinates of the disks in each column are found:

```
def find_top_and_bottom_coord_of_each_col(col_no, row_no, new_lst, points):
    col_1 = []
    col_2 = []
    col_3 = []
    col_4 = []
    col_5 = []
    col_6 = []

    for i in range(len(row_no)):
        #print(i)

        if new_lst[i][1] == 0 and new_lst[i][0] == 0:
            col_1.append(points[i])
            print('found top disk of column 1', new_lst[i], points[i])

        if new_lst[i][1] == 0 and new_lst[i][0] == 5:
            col_1.append(points[i])
            #print('found bottom disk of column 1', new_lst[i], points[i])

        if new_lst[i][1] == 1 and new_lst[i][0] == 0:
            col_2.append(points[i])

        if new_lst[i][1] == 1 and new_lst[i][0] == 5:
            col_2.append(points[i])

        if new_lst[i][1] == 2 and new_lst[i][0] == 0:
            col_3.append(points[i])

        if new_lst[i][1] == 2 and new_lst[i][0] == 5:
            col_3.append(points[i])

        if new_lst[i][1] == 3 and new_lst[i][0] == 0:
            col_4.append(points[i])

        if new_lst[i][1] == 3 and new_lst[i][0] == 5:
            col_4.append(points[i])

        if new_lst[i][1] == 4 and new_lst[i][0] == 0:
            col_5.append(points[i])
```

(continues on next page)

(continued from previous page)

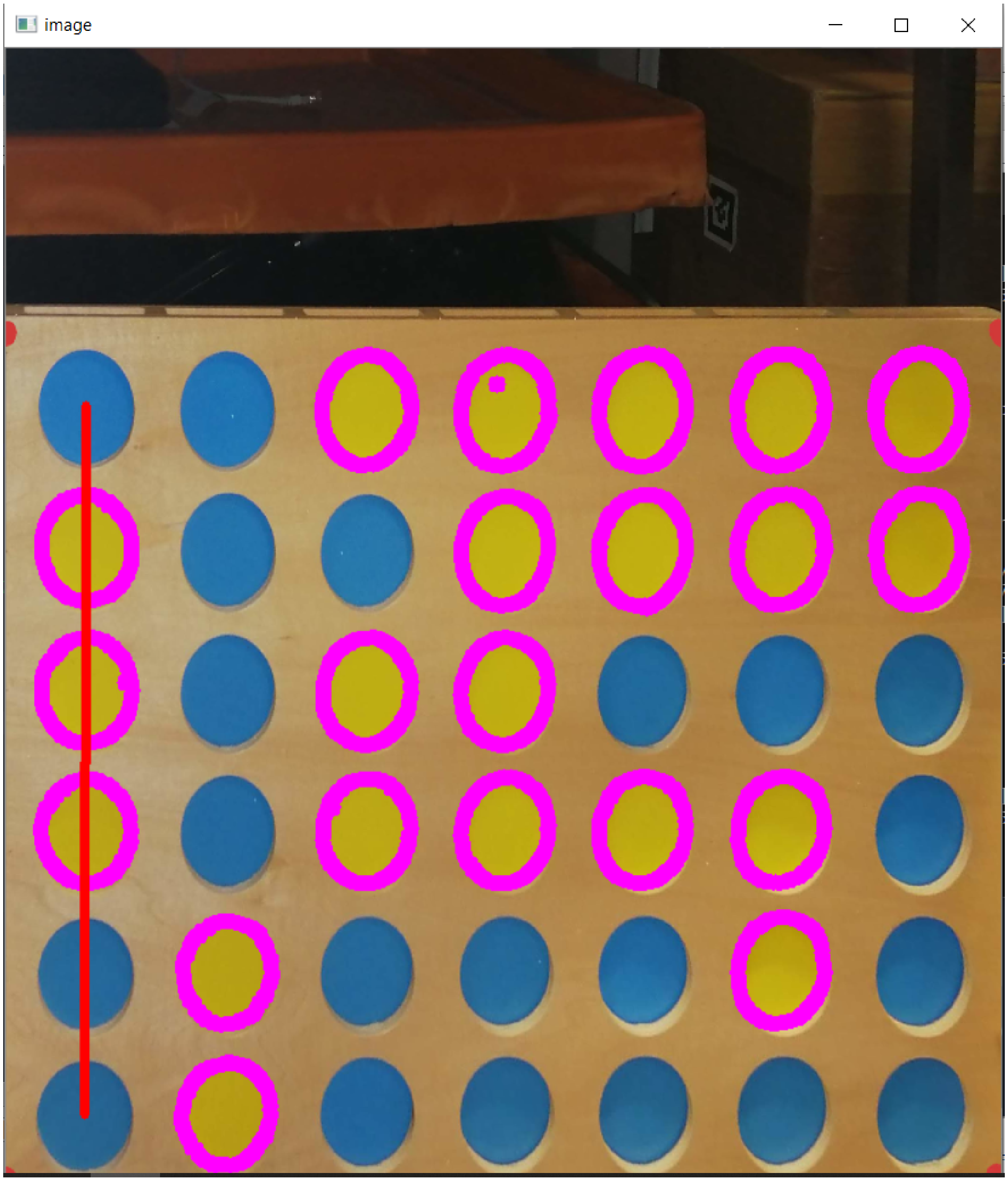
```
        if new_lst[i][1] == 4 and new_lst[i][0] == 5:
            col_5.append(points[i])

        if new_lst[i][1] == 5 and new_lst[i][0] == 0:
            col_6.append(points[i])

        if new_lst[i][1] == 5 and new_lst[i][0] == 5:
            col_6.append(points[i])
    return col_1, col_2, col_3, col_4, col_5, col_6
```

Then after this, the target column is used as an input for the following function, where it plots the centreline in the snapshot taken:

Then using the existing code to find the red markers in the function `GetPossitions`, the coordinate of that marker can be found, and calling the following function:

```
def get_row_and_col(coordinates):
    Tolerance = 20 #a tolerance is added to check the coordinate in the row/column␣
↪are within a range.
    xList = []
    yList = []
    KeyX = [55 , 155 , 250 , 345 , 450 , 545 , 640] #these are the estimated pixels␣
↪in which the coordinates for each column lie in
    KeyY = [200 , 330 , 430 , 530 , 650 , 760] #these are the estimated pixels in␣
↪which the coordinates for each row lie in
    for i in coordinates:
        y_coord = i[1]
        x_coord = i[0]
        for n,x in enumerate(KeyX):
            if abs(x_coord - x) < Tolerance:
                xList.append(n)
                break
        else:
            print("x out" , x_coord)
            pass
        for n,y in enumerate(KeyY):
            if abs(y_coord - y) < Tolerance:
                yList.append(n)
                break
        else:
            xList.pop(-1)
            print("Y out" , y_coord)
    return [cord for cord in zip(xList , yList)]
```

It can be verified whether or not the gripper marker lies in the same column as the target column wherein the robot wishes to place the next disk.



image

CHAPTER 7

Connect 4 Algorithm

In order for the robot to play competitively against a human, a minimax game algorithm is used to choose the best move in response to the human player. The algorithm's 'game loop' is implemented inside the main file, but for general tidiness all of the algorithm functions are stored in a separate file.

7.1 Setup Functions

Create a numpy array of zeroes to represent the Connect 4 board. This will be populated with numbered pieces throughout the game.

```python
def create_board():
    board = np.zeros((ROW_COUNT, COLUMN_COUNT))
    return board
```

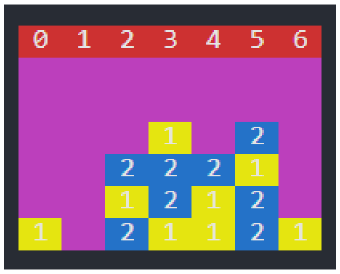
Set up the board to print out in the terminal in a way that makes it visually easy to play with the computer.

```python
def pretty_print_board(board):
    flipped_board = np.flipud(board)

    print("\033[0;37;41m 0 \033[0;37;41m 1 \033[0;37;41m 2 \033[0;37;41m 3 \033[0;37;
↪41m 4 \033[0;37;41m 5 \033[0;37;41m 6 \033[0m")
    for i in flipped_board:
        row_str = ""

        for j in i:
            if j == 1:
                #print(yellow)
                row_str +="\033[0;37;43m 1 "
            elif j ==2:
                row_str +="\033[0;37;44m 2 "
            else:
                #print black
                row_str +="\033[0;37;45m   "
```

(continues on next page)

(continued from previous page)

```
        print(row_str+"\033[0m")
```



```
 0  1  2  3  4  5  6


          1     2
       2  2  2  1
       1  2  1  2
 1     2  1  1  2  1
```

Note: Due to restrictions on the version of numpy, `np.flipud(board)` was used instead of the most up to date version: `np.flip(board)`. If you are using the most up to date version of numpy, you can update this function (although it will not break if you do not - numpy has reasonably good backwards-compatibility).

Warning: It's important to understand that the algorithm fills the board from the top down. In real life, the board fills up from the bottom. `np.flipud(board)` flips the board around a horizontal axis, making it the correct visual orientation for Connect 4. In future, for clarity and ease of understanding, the placement of the pieces within the board will be referred to in the same way that it would happen in real life (bottom up).

There are 4 functions that are used when placing a piece on the board.

1. To get all locations in the board that could contain a piece (i.e. have not yet been filled):

```
def get_valid_locations(board):
    valid_locations = []
    for col in range(COLUMN_COUNT):
        if is_valid_location(board, col):
            valid_locations.append(col)
    return valid_locations
```

2. To check if there is a valid location in the chosen column:

```
def is_valid_location(board, col):
    return board[ROW_COUNT - 1][col] == 0
```

3. To check which row the piece can be placed into (i.e. the next available open row):

```
def get_next_open_row(board, col):
    for r in range(ROW_COUNT):
        if board[r][col] == 0:
            return r
```

Note: This function serves as virtual 'gravity'. Instead of placing a piece anywhere in the column, by getting only the next open row, piece placement is restricted to the next available slot from the bottom of the board, as would happen in real life. This also means that the only input that is required is now the column (the row is automatically found and assigned).

4. Finally, to place a piece in the next available row, in the chosen column:

```python
def drop_piece(board, row, col, piece):
    board[row][col] = piece
```

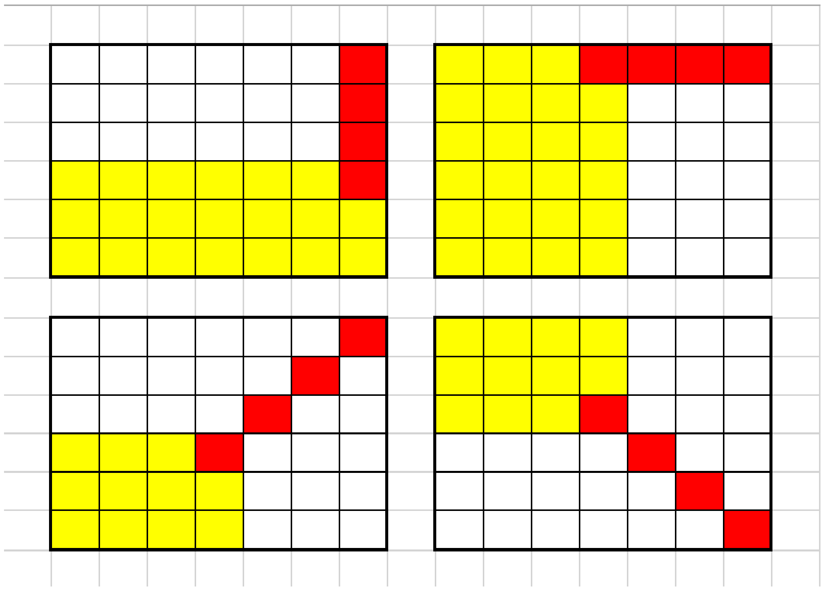
7.2 Analysis Functions

When the human player (Player 1) has made a move, the `drop_piece` function will update the numpy array `board` with a number 1 in the specified position. In order for the game algorithm (Player 2) to choose the best move to play in response, it has to understand and analyse the current board state. This is done using a 'windowing' technique. In the following function, horizontal, vertical, positive (upward sloping) and negative (downward sloping) diagonal windows are created. These windows are then used to scan all possible 4-piece sections of the board, and evaluate (score) each window based on its contents.

This evaluation is performed separately by the `evaluate_window` function, which is called within the `score_position` function, and explained in further detail below.

```python
def score_position(board, piece):
    score = 0

    # Score centre column
    centre_array = [int(i) for i in list(board[:, COLUMN_COUNT // 2])]
    centre_count = centre_array.count(piece)
    score += centre_count * 3

    # Score horizontal positions
    for r in range(ROW_COUNT):
        row_array = [int(i) for i in list(board[r, :])]
        for c in range(COLUMN_COUNT - 3):
            # Create a horizontal window of 4
            window = row_array[c:c + WINDOW_LENGTH]
            score += evaluate_window(window, piece)

    # Score vertical positions
    for c in range(COLUMN_COUNT):
        col_array = [int(i) for i in list(board[:, c])]
        for r in range(ROW_COUNT - 3):
            # Create a vertical window of 4
            window = col_array[r:r + WINDOW_LENGTH]
            score += evaluate_window(window, piece)

    # Score positive diagonals
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            # Create a positive diagonal window of 4
            window = [board[r + i][c + i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    # Score negative diagonals
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            # Create a negative diagonal window of 4
            window = [board[r + 3 - i][c + i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    return score
```

The figure below shows the scanning range for this `score_position` function. It is unnecessary to use every index of the board as a starting position for a scanning window, because in many positions some windows would then extend over the sides of the board. As a result, there are only 69 positions in which the scanning window needs to be deployed. The yellow highlight shows the applicable scanning range, and the red squares are an example of a scanning window in the maximum required position.



The `evaluate_window` function is called in the last line of each scoring block. The output of this evaluation function (a numerical score value) is stored in the `score` variable, which is updated every time a higher score is found. When the scanning is complete, the window with the best score is passed to the game algorithm to play a move. Note that this scoring mechanism is required, but the minimax function, which will be explained in further detail, makes some elements of this function much less important.

In any given scanning position, the contents of that window are evaluated for 'strength', e.g. a window that contains 3 consecutive pieces from the same player is a 'strong' state, and has a higher score. This means that the algorithm is more likely to try and create board states that are 'strong' - i.e. prioritise connecting 3 pieces together, rather than connecting 2.

```
def evaluate_window(window, piece):
    score = 0
    # Switch scoring based on turn
    opp_piece = PLAYER_PIECE
    if piece == PLAYER_PIECE:
        opp_piece = BOT_PIECE

    # Prioritise a winning move
    # Minimax makes this less important
    if window.count(piece) == 4:
        score += 100
    # Make connecting 3 second priority
    elif window.count(piece) == 3 and window.count(EMPTY) == 1:
        score += 5
    # Make connecting 2 third priority
    elif window.count(piece) == 2 and window.count(EMPTY) == 2:
        score += 2
    # Prioritise blocking an opponent's winning move (but not over bot winning)
    # Minimax makes this less important
    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
```

(continues on next page)

(continued from previous page)

```
        score -= 4

    return score
```

The final element of the analysis is a 'special case' variation of the `score_position` function. When 4 pieces are joined together, this signifies the game has been won. After every move, the board needs to be scanned by both the `score_position` function, and also the `winning_move` function, which will exit out of the game loop if it sees a winning move.

```
def winning_move(board, piece):
    # Check valid horizontal locations for win
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT):
            if board[r][c] == piece and board[r][c + 1] == piece and board[r][c + 2]␣
↪== piece and board[r][c + 3] == piece:
                return True

    # Check valid vertical locations for win
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT - 3):
            if board[r][c] == piece and board[r + 1][c] == piece and board[r + 2][c]␣
↪== piece and board[r + 3][c] == piece:
                return True

    # Check valid positive diagonal locations for win
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT - 3):
            if board[r][c] == piece and board[r + 1][c + 1] == piece and board[r +␣
↪2][c + 2] == piece and board[r + 3][c + 3] == piece:
                return True

    # check valid negative diagonal locations for win
    for c in range(COLUMN_COUNT - 3):
        for r in range(3, ROW_COUNT):
            if board[r][c] == piece and board[r - 1][c + 1] == piece and board[r -␣
↪2][c + 2] == piece and board[r - 3][c + 3] == piece:
                return True
```

7.3 Algorithm

The algorithm chosen to play Connect 4 is the minimax algorithm. Minimax is a backtracking algorithm which is commonly used in decision-making and game theory to find the optimal move for a player. This makes it a perfect choice for two-player, turn-based games.

In the minimax algorithm, the two players are the maximiser and minimiser. The maximiser is trying to get the highest score possible, and the minimiser is trying to get the lowest score possible. The best / worst scores are calculated by the `evaluate_window` function, and stored in the `score` variable, described in the previous section.

At the start of every turn, minimax will scan the board's remaining valid locations and calculate all possible moves, before backtracking and choosing the optimal move for that turn. This will be either the best or worst move, depending on whether it is the maximiser or minimiser's turn. The assumption is that minimax (maximiser) can play optimally, as long as the human player (minimiser) also plays optimally. This will not always be the case, but does not lead to significant gameplay problems.

Before implementing the minimax algorithm, the two game-terminating states need to be defined as terminal nodes.

If there is a winning move from either player, or if the board fills up without a win (leading to a draw), the game will end.

```
def is_terminal_node(board):
    return winning_move(board, PLAYER_PIECE) or winning_move(board, BOT_PIECE) or␣
↪len(get_valid_locations(board)) == 0
```

The minimax algorithm for the Connect 4 game is implemented below.

```
def minimax(board, depth, alpha, beta, maximisingPlayer):
    valid_locations = get_valid_locations(board)

    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            # Weight the bot winning really high
            if winning_move(board, BOT_PIECE):
                return (None, 9999999)
            # Weight the human winning really low
            elif winning_move(board, PLAYER_PIECE):
                return (None, -9999999)
            else:  # No more valid moves
                return (None, 0)
        # Return the bot's score
        else:
            return (None, score_position(board, BOT_PIECE))

    if maximisingPlayer:
        value = -9999999
        # Randomise column to start
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            # Create a copy of the board
            b_copy = board.copy()
            # Drop a piece in the temporary board and record score
            drop_piece(b_copy, row, col, BOT_PIECE)
            new_score = minimax(b_copy, depth - 1, alpha, beta, False)[1]
            if new_score > value:
                value = new_score
                # Make 'column' the best scoring column we can get
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value

    else:  # Minimising player
        value = 9999999
        # Randomise column to start
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            # Create a copy of the board
            b_copy = board.copy()
            # Drop a piece in the temporary board and record score
            drop_piece(b_copy, row, col, PLAYER_PIECE)
            new_score = minimax(b_copy, depth - 1, alpha, beta, True)[1]
```

(continues on next page)

(continued from previous page)

```
            if new_score < value:
                value = new_score
                # Make 'column' the best scoring column we can get
                column = col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return column, value
```

Note: The implementation of this minimax algorithm also contains Alpha-Beta pruning. There is no point following a decision-tree branch any further if the initial move scores less optimally than an alternative move that has already been discovered. Alpha-Beta pruning works to 'prune' away these branches, leaving a much smaller, more optimised decision tree.

This technique is used to reduce the time complexity of the algorithm, which in this context is important, as there are many other parts of the game loop that are time consuming (e.g. Motion Planning). The game algorithm can now run reliably in under 500ms, even when looking 4 moves into the future.

7.4 Limitations / Improvements

There are some key limitations to the algorithm, but they did not need to be directly addressed as they were outside the scope for this project.

1. Lack of scalability

Due to the hard-coded nature of the scanning procedure, the board size, the number of connected pieces required to win, and the scanning window size cannot be changed without causing major errors. This would not be particuarly difficult to fix, but would require a different, more adaptive scanning structure and further definition of static variables.

2. Incomplete win structure

During stress testing, it became clear that the algorithm would not make a winning move if there were two or more possible winning moves available. This is presumably because it could not decide between equally weighted branches, and therefore made the 'next best' move. This problem did not impact the algorithm's success rate, however, because as soon as the human player filled one of the possible winning spaces, the algorithm would win the game using the other.

CHAPTER 8

Miscellaneous

8.1 Documentation Instructions

This documentation has been created using the `sphinx` documentation generator. For creating the documentation for this project, `sphinx` must be installed on your computer. This can be done with:

```
pip install sphinx              # for Windows
apt-get install python-sphinx   # for Linux
```

You will also have to install `recommonmark` which can be done with:

```
pip install recommonmark        # for Windows
apt-get install recommonmark    # for Linux
```

Lastly, `sphinx_rtd_theme` , the theme used here, should be installed.

```
pip install sphinx_rtd_theme
```

To compile and view your documentation, navigate to `RoboticsProject/docs` and open a terminal there. On your terminal, type in

```
make clean html
```

This should compile your .rst documentation files to html in the `docs/_build/html` folder. To view what the webpage will look like, navigate to the `docs/_build/html` folder and open `index.html` in your browser.

Warning: If your `make clean html` command did not terminate successfully, you will not be able to find the `docs/_build/html` folder as it is generated by that command.

Note: When you rebuild the html files with `make clean html`, all you need to do is refresh your webbrowser to see the latest updated changes.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search