ECE 495: Intro to Robotics Final Project Report

High Level Description

The goal of our project is to create a Tic-Tac-Toe playing robot. We use a robot arm to move game pieces that are spawned into Gazebo and place the correct X or O piece on the board based on the user's input location. There are currently two ways to play the game. The first option is to play a 2-player game with both sides being controlled by humans. In this scenario the Panda Arm places pieces on the board as the player's take turns inputting their choices into the command line. The second option is to play against the computer AI. In this mode the player will select whether they want to be X or O and then the game randomly selects who moves first. The player will alternate moves with the AI until a winner is determined or there is a tie.

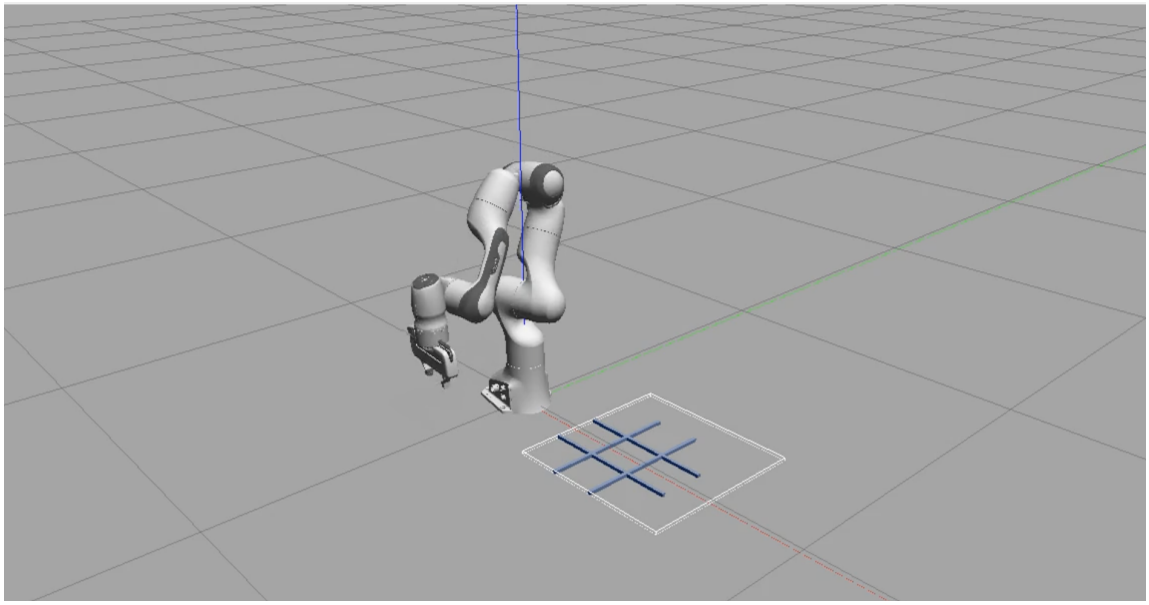
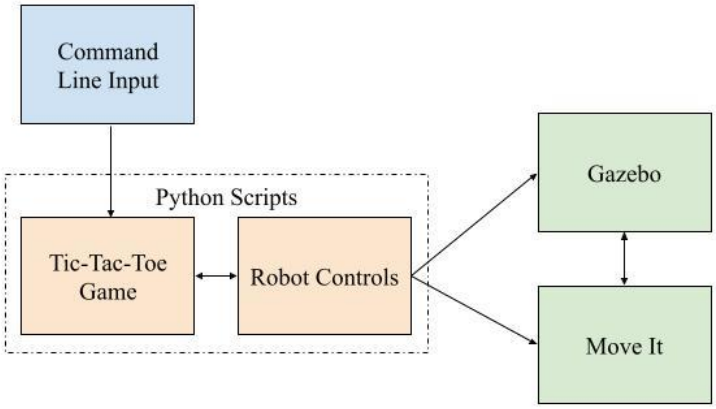


Fig 1: Panda Robot with Game Board

Systems Diagram



Code Explanation

Our code has two parts. We have a tic-tac-toe game that runs via command line in one python file and we have all the robot's movement functions in the other file. By splitting up our project like this, our team was able to independently work on tic-tac-toe and robot movement simultaneously. When the time came to combine both parts together we simply had to import and call the robot's movement functions in the tic-tac-toe script when either a player or an AI moved.

Tic-Tac-Toe Script

The script has two classes that have very similar functionality. TicTacToeAI starts a game against the AI while TicTacToe is 2 player. The code is very similar and as such a walkthrough of the functions will be provided for only the TicTacToeAI class.

The class construction handles setup such as determining the player and computer's letter, creating the board, deciding who goes first, and starting the game loop. It also includes setup for the Panda Arm as well which handles the ROS aspect.

```
class TicTacToeAI():
    def __init__(self):
        xpieces = ['X1','X2','X3','X4','X5']
        opieces = ['O1','O2','O3','O4','O5']
        board = ['Board']
        self.tutorial = MoveGroupPythonIntefaceTutorial()
        self.tutorial.removePieces(xpieces+opieces+board)
        time.sleep(1)
        spawnBoard(board[0])
        self.theBoard = [""] * 10
        self.playerLetter, self.computerLetter = self.inputPlayerLeter()
        self.turn = self.whoGoesFirst()
        print("The " + self.turn + " will go first.")
        self.gameIsPlaying = True
        self.startGameLoop()
```

The drawBoard function is simply a helper function which is used to draw a formatted board in the command line interface.

```
def drawBoard(self, board):
    print(" {:s} | {:s} | {:s} ".format(board[7], board[8], board[9]))
    print(" --------- ")
    print(" {:s} | {:s} | {:s} ".format(board[4], board[5], board[6]))
    print(" --------- ")
    print(" {:s} | {:s} | {:s} ".format(board[1], board[2], board[3]))
```

The inputPlayerLetter function simply asks the user which letter they want to be and updates the game variables appropriately in class initialization. The whoGoesFirst method randomly selects who plays first.

```
def inputPlayerLeter(self):
    # Lets the player type which letter they want to be.
    # Returns a list with the player's letter as the first item, and the computer's letter as the second.
    letter = ""
    while not (letter == "X" or letter == "O"):
        print("Do you want to be X or O")
        letter = input().upper()

    if letter == "X":
        return ("X", "O")
    else:
        return ("O", "X")

def whoGoesFirst(self):
    if random.randint(0, 1) == 0:
        return "AI"
    else:
        return "Human"
```

The playAgain function simply asks the user and takes in their input. The makeMove function takes in the board and updates it with the move made by the appropriate player. The isWinner function checks all possible win cases for TicTacToe and returns if any have occurred.

```
def playAgain(self):
    print("Do you want to play again? (yes or no)")
    return input().lower().startswith("y")

def makeMove(self, board, letter, move):
    board[move] = letter

def isWinner(self, board, letter):
    return ((board[7] == letter and board[8] == letter and board[9] == letter) or  # across the top
            # across the middle
            (board[4] == letter and board[5] == letter and board[6] == letter) or
            # across the bottom
            (board[1] == letter and board[2] == letter and board[3] == letter) or
            # down the left side
            (board[7] == letter and board[4] == letter and board[1] == letter) or
            # down the middle
            (board[8] == letter and board[5] == letter and board[2] == letter) or
            # down the right side
            (board[9] == letter and board[6] == letter and board[3] == letter) or
            # diagonal
            (board[7] == letter and board[5] == letter and board[3] == letter) or
            # diagonal
            (board[9] == letter and board[5] == letter and board[1] == letter))
```

The main difference between the 2-player game and the AI one is the logic in the function below. The code first gets a copy of the board to compute on. First it tests whether a move can be made immediately allowing the AI to win and places the piece there if so. If not it checks whether the player can make a move that would win and blocks that. Otherwise it takes the preference of playing in any corners if they are available followed by the center and finally the sides. Algorithm referenced from here.

```
def getComputerMove(self, board, computerLetter):
    if computerLetter == "X":
        playerLetter = "O"
    else:
        playerLetter = "X"

    # Here is our algorithm for our Tic Tac Toe AI:
    # First, check if we can win in the next move
    for i in range(1, 10):
        copy = self.getBoardCopy(board)
        if self.isSpaceFree(copy, i):
            self.makeMove(copy, computerLetter, i)
            if self.isWinner(copy, computerLetter):
                return i

    # Check if the player could win on their next move, and block them.
    for i in range(1, 10):
        copy = self.getBoardCopy(board)
        if self.isSpaceFree(copy, i):
            self.makeMove(copy, playerLetter, i)
            if self.isWinner(copy, playerLetter):
                return i

    # Try to take one of the corners, if they are free.
    move = self.chooseRandomMoveFromList(board, [1, 3, 7, 9])
    if move != None:
        return move

    # Try to take the center, if it is free.
    if self.isSpaceFree(board, 5):
        return 5

    # Move on one of the sides.
    return self.chooseRandomMoveFromList(board, [2, 4, 6, 8])
```

Finally we have the last two functions of our TicTacToe code. A helper isBoardFull is used to check if all spaces are full for the tie condition. The startGameLoop function handles the main sequential game logic. While the gameIsPlaying state is True we alternate between the Human and AI's turn. In each one of those turns, we draw the state of the board and allow either the Human to input a move or use the AI's calculated move. Finally, we go through a series of checks to determine if a winner is found or the game is tied.

```
def isBoardFull(self, board):
    # Return True if every space on the board has been taken. Otherwise return False.
    for i in range(1, 10):
        if self.isSpaceFree(board, i):
            return False
    return True

def startGameLoop(self):
    while self.gameIsPlaying:
        if self.turn == "Human":
            # Human's turn
            self.drawBoard(self.theBoard)
            move = self.getPlayerMove(self.theBoard)
            self.makeMove(self.theBoard, self.playerLetter, move)
            self.tutorial.make_move(move, self.playerLetter)

            if self.isWinner(self.theBoard, self.playerLetter):
                self.drawBoard(self.theBoard)
                print("You have won the game!")
                self.gameIsPlaying = False
            else:
                if self.isBoardFull(self.theBoard):
                    self.drawBoard(self.theBoard)
                    print("The game is a tie!")
                    break
                else:
                    self.turn = "AI"
        else:
            # AI's Turn
            move = self.getComputerMove(self.theBoard, self.computerLetter)
            self.makeMove(self.theBoard, self.computerLetter, move)
            self.tutorial.make_move(move, self.computerLetter)

            if self.isWinner(self.theBoard, self.computerLetter):
                self.drawBoard(self.theBoard)
                print("The computer has won. You lose!")
                self.gameIsPlaying = False
            else:
                if self.isBoardFull(self.theBoard):
                    self.drawBoard(self.theBoard)
                    print("The game is a tie!")
                    break
                else:
                    self.turn = "Human"
```

Robot Functions

For spawning in the board and the X/O pieces we use this type of function. Takes in a name since gazebo model names have to be unique in the scene. We get around this to spawn pieces by naming our pieces X or O followed by a number 1-5.

```
# This is how we spawn the board into our gazebo scene using the spawn model client
def spawnBoard(name):
  spawn_model_client = rospy.ServiceProxy('/gazebo/spawn_sdf_model', SpawnModel)
  spawn_model_client(
    model_name=name,
    model_xml=open('/home/tpc14/tictactoe/custom_pieces/tic_tac_toe_board/model.sdf', 'r').read(),
    robot_namespace='/moveit_commander',
    initial_pose=Pose(position= Point(0.738252,0.302994,0),orientation=Quaternion(0,0,0,0)),
    reference_frame='world'
  )
```

For deleting any extraneous boards/pieces after a run:

```
def removePieces(self, names, timeout=4):
    del_model_prox = rospy.ServiceProxy('/gazebo/delete_model', DeleteModel)
    for name in names:
        del_model_prox(name)
```

For taking in a move number and transforming that into cartesian movements we use these functions. The idea here is that one of our robot's "home" positions (after a move_to_board) is directly above the #2 position on the tic-tac-toe board (board with numbered positions is drawn at the top of the move_to_board_position function). Knowing this and the fact that all adjacent positions (not diagonal) are exactly $\pm$0.127666m away from each other, you can create a multiplication factor for both the x and y cartesian trajectory vectors for all numbered board positions. For example, just by supplying the desired position 9 (top right) we would move the EE 1 * 0.127666m to the right and 2 * 0.127666m up.

```
def transformFunc(num):
    if num == 0:
        return 1
    else:
        return num-2

def getMovements(move):
    movements = [0,0]
    movements[0] = transformFunc(move%3)
    movements[1] = math.floor((move-1)/3)
    return movements
```

```
def move_to_board_position(self,position):
    # Board positions
    # 7 | 8 | 9
    # 4 | 5 | 6
    # 1 | 2 | 3

    move = getMovements(position)
    group = self.group

    waypoints = []

    wpose = group.get_current_pose().pose
    wpose.position.x += move[1] * 0.127666
    wpose.position.y += -move[0] * 0.127666
    waypoints.append(copy.deepcopy(wpose))


    # We want the Cartesian path to be interpolated at a resolution of 1 cm
    # which is why we will specify 0.01 as the eef_step in Cartesian
    # translation.  We will disable the jump threshold by setting it to 0.0 disabling:
    (cartesian_plan, fraction) = group.compute_cartesian_path(
                                    waypoints,   # waypoints to follow
                                    0.01,        # eef_step
                                    0.0)         # jump_threshold
    self.execute_plan(cartesian_plan)
    self.printEECoordinates()
```

For opening and closing the gripper with force to make sure pieces didn't slip from the EE's grasp:

```
# Opening the EE gripper using the franka gripper action client
def open_gripper(self):
  client = actionlib.SimpleActionClient('franka_gripper/grasp', GraspAction)
  client.wait_for_server()
  action = GraspGoal(width=0.08,speed=0.08,force=1)
  client.send_goal(action)
  wait = client.wait_for_result()
  if not wait:
    rospy.logerr("Action server not available!")
    rospy.signal_shutdown("Action server not available!")
  else:
    return client.get_result()

# Closing the EE gripper using the franka gripper action client
def close_gripper(self):
  client = actionlib.SimpleActionClient('franka_gripper/grasp', GraspAction)
  client.wait_for_server()
  action = GraspGoal(width=0.055,speed=0.03,force=20)
  action.epsilon.inner = 0.005
  action.epsilon.outer = 0.002
  client.send_goal(action)
  wait = client.wait_for_result()
  if not wait:
    rospy.logerr("Action server not available!")
    rospy.signal_shutdown("Action server not available!")
  else:
    return client.get_result()
```

This is one of the two home positions we calculated the joint goals for. We used these to make sure the rest of the robot would never hit the pieces coming back from picking up or dropping a piece. For moving the EE up and down we used very similar functions except they used cartesian $\pm$ y movements to get the EE to the correct height.

```
def move_to_board(self):
  group = self.group

  ## Planning to a Joint Goal
  ## ^^^^^^^^^^^^^^^^^^^^^^^^
  ## The Panda's zero configuration is at a
  # `singularity <https://www.quora.com/Robotics-What-is-meant-by-kinematic-singularity>`_ so the first
  ## thing we want to do is move it to a slightly better configuration.
  # We can get the joint values from the group and adjust some of the values:
  joint_goal = group.get_current_joint_values()
  joint_goal[0] = 0.33383182800560984
  joint_goal[1] = -0.782783609353852
  joint_goal[2] = -0.2096302313904701
  joint_goal[3] = -2.339148939786515
  joint_goal[4] = -0.12285217911277968
  joint_goal[5] = 1.529797990773984
  joint_goal[6] = 1.047124117937443

  # The go command can be called with joint values, poses, or without any
  # parameters if you have already set the pose or joint target for the group
  group.go(joint_goal, wait=True)

  # Calling ``stop()`` ensures that there is no residual movement
  group.stop()
```

Finally we could abstract all of these away into a higher level function called make_move. Using all the functions discussed before it only needs the desired numbered board position and piece type to automate the entire piece dropping process. This means that for our tic-tac-toe interfacing code we only call this one line to make the robot move the way we want it to.

```
def make_move(self,position, piece_type):

  self.move_to_piece_factory()
  self.open_gripper()
  if piece_type == 'X':
    spawnX("X"+str(self.x_num))
    self.x_num += 1
  else:
    spawnO("O"+str(self.o_num))
    self.o_num += 1
  self.move_down_pick_up()
  self.close_gripper()
  self.move_up()
  self.drop_piece(position)
```

My Efforts

The part of the project I started working on was the setup to find a robot compatible with Gazebo, MoveIt, and that has an end-effector. Unfortunately, I couldn't find a single scara robot model online and had to pivot to another robot. I chose to use the Panda robot because it had a built in end effector. With the Github repo referenced in Challenges I was able to get the panda robot visualized in both gazebo and rviz with motion planning enabled and set up basic movement functionality for the team to expand on. The second aspect of the project I worked on was the TicTacToe python game. I coded up two fully functioning modular tic-tac-toe games. The first one was the 2-player game that takes in user input from two people on the command line and gives a visual representation of the game state on the screen. The second one is a 1-player game that allows a human to play against an AI where the AI always plays an optimal move. The code was modular enough to allow our team members to easily insert their movement logic to combine the backend game code and the robot movement. Finally, I was able to help solve the spawning issue which allowed us to finally import the game board at the start and spawn the correct pieces as needed.

Video/GIF

https://youtu.be/AAifj87mxgs

Human Integration

When thinking about the human interaction required for the physicalized version of this project, we need to mainly consider the task space and work space. The work space is the area that the end effector can reach in its various configurations while the task space extends to all possible configurations of the robot. Since our robot would be moving around large objects, it is not enough to restrict access to the workspace. We also need to make sure that no humans could be hit by moving pieces. This means extending the restricted area to include an extra radius around the end effector. It is important to place the controls for the game outside of this radius to allow the human to play the game. Additionally, the ability to reset pieces by the robot is important. We can try to minimize human interaction in the robot's task space so that there are no accidents that may occur when a human manually resets the game state.

Challenges

The first challenge faced by our team was finding a suitable robot arm for this task. The UR robots we used in the midterms did not have end-effectors available and we could not find any models for the SCARA robot which was initially selected in our proposal. Therefore, we had to pivot to using the Panda arm, the only robot arm which was readily available with a usable end-effector. Unfortunately, while the Panda is compatible with MoveIt for motion planning, it is

not readily usable with Gazebo and requires many intricate changes for compatibility. Thankfully, we were able to find a Github repo that set up an environment for the Panda arm which is compatible with both Gazebo and MoveIt. The second issue we faced was with respect to object spawning. While we followed a user solution, we could not replicate the spawning behavior and were met with errors. The solution eventually ended up being that we could not store our SDF models in the catkin_ws directory and instead moved these models to our repo folder in the home directory and this allowed us to finally spawn the game board and game pieces into Gazebo using a python script.

Future Work

One of the main stretch goals listed for our project was to be able to have our Tic-Tac-Toe robot arm go up against the other team's robot. While we were not ultimately able to achieve this stretch goal, we did work to standardize most of our feature set so that this may be done relatively easily in the future. Game items like our board and the game pieces were standardized between teams for integration. Additionally, we worked to standardize the movement logic between teams so that we could work out the spawn locations for the game pieces and also easily alter the movement logic so that one robot can use the same movement functions but alter the locations behind the scenes to account for being on the opposite side of the board. A scoring system to keep track of best of 3 or best of 5 games would also be a really great feature to have if there was more time.

Beyond this, we think it would be really cool to test out this functionality with a real Panda robot arm. Some changes would need to be made as of course pieces can't be spawned out of thin air in real life. However, creating new set locations for the arm and each piece would not be too difficult and could allow us to run our current code without many changes.

Takeaways

- Using cartesian trajectory planning is useful when the robot can easily move its EE to your desired location. However, if you care about where the rest of the robot's links are relative to objects in your scene, supplying the robot with joint goals makes the robot's movement much more predictable and easier for debugging. For example, our robot has 2 home positions we calculated the quaternions for to make sure our panda arm didn't hit any pieces, but moving to board positions/picking up pieces were all cartesian trajectory planning.
- We found that while our scripts would work the majority of the time on the virtual machine, there were certain periods of time when the robot would not even be able to access its own current joint states (without changing any code). Since our script will fail if the robot can't access its joint states within a time threshold, we recommend installing gazebo and running more advanced robot functionality on local vms to get rid of latency issues.