"VISION GUIDED OBJECT PICKING WITH UR10 ROBOT"

A Minor Project Report

Submitted in Partial Fulfillment of the Requirements for the degree

of

BACHELOR OF TECHNOLOGY
IN
INSTRUMENTATION AND CONTROL ENGINEERING

By
Masoom Lalani 20BIC025
Kathan Patel 20BIC022

Under the Guidance of
Vishal Vaidya

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION
SCHOOL OF TECHNOLOGY
NIRMA UNIVERSITY
Ahmedabad 382 481

DECEMBER 2023

CERTIFICATE

THIS IS TO CERTIFY THAT THE PROJECT REPORT ENTITLED **"VISION GUIDED OBJECT PICKING WITH UR10 ROBOT"** SUBMITTED BY MR.**MASOOM LALANI (20BIC025)** & MR.**KATHAN PATEL** (**20BIC022**) TOWARDS THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE IN **BACHELOR OF TECHNOLOGY (INSTRUMENTATION & CONTROL ENGINEERING)** OF **SCHOOL OF TECHNOLOGY** IS THE RECORD OF WORK CARRIED OUT BY HIM/HER/THEM UNDER MY/OUR SUPERVISION AND GUIDANCE. THE WORK SUBMITTED HAS IN OUR OPINION REACHED A LEVEL REQUIRED FOR BEING ACCEPTED FOR EXAMINATION. THE RESULTS EMBODIED IN THIS PROJECT WORK TO THE BEST OF MY/OUR KNOWLEDGE HAVE NOT BEEN SUBMITTED TO ANY OTHER UNIVERSITY OR INSTITUTION FOR AWARD OF ANY DEGREE OR DIPLOMA.

VISHAL VAIDYA **PROJECT GUIDE**	**PROF. H K PATEL** **HOD**

DATE:

CONTENTS

11. **Image Difference Algorithm**
    a. Converting Pixel Locations to Real-world Coordinates
    b. Creating an 8x8 Grid
    c. Generating CSV File for Chessboard Square Information
    d. Detecting Changes on the Chessboard
    e. Image Subtraction and Thresholding
    f. Morphological Operations for Noise Reduction
12. **Integration with Stockfish AI**
    a. Providing Chess Notation to Stockfish
    b. Receiving AI Suggested Moves
    c. Implementing Moves on the Chessboard
13. **Conclusion**
    - Achievements and Future Enhancements

Acknowledgement

We extend our heartfelt thanks to everyone who played a vital role in the successful completion of this project. A special expression of gratitude goes to Vishal Vaidya, our supervisor as well as mentor, whose unwavering support, motivating ideas, and guidance were indispensable at every stage—from dataset optimization and machine learning model training to model validation, web application deployment, and report writing. We are also deeply appreciative of Dr. Sharma for taking the time to proofread our work and rectify errors.

Our sincere thanks to the esteemed panelists for their invaluable guidance, steering us in the right direction to explore an idea with potential real-world applications.

Abstract:

This project explores the integration of the UR10e robotic arm, a collaborative robot designed by Universal Robots, into the realm of chess-playing applications. Leveraging the UR e-Series features, including a 10 kg payload capacity, a reach of 1.3 meters, and six degrees of freedom, the robotic arm is employed collaboratively with humans, enhancing its adaptability to shared workspaces. The integration is achieved through the implementation of ROS (Robot Operating System) and MoveIt!, providing a modular and standardized framework for seamless motion planning and control. The project further delves into path planning utilizing the Open Motion Planning Library (OMPL) and employs the Rapidly-exploring Random Tree (RRT) algorithm, with Covariant Hamiltonian Optimization for Motion Planning (CHOMP) as a backup. The application is complemented by a comprehensive image difference algorithm, involving a combination of image processing techniques, to facilitate the real-time interaction between the robot and the chessboard. The entire system orchestrates a dynamic interplay between the robotic arm, ROS, MoveIt!, and advanced planning algorithms, opening avenues for collaborative robotics in strategic games.

Robotic Arm- UR10e

The UR10e is a robotic arm manufactured by Universal Robots, a Danish company that specializes in collaborative robot (cobots) technology. Here are some key features and information about the UR10e robotic arm:

Overview:

The UR10e is part of Universal Robots' UR e-Series, which is a family of collaborative robotic arms designed for various industrial applications.

Collaborative Robot (Cobot):

The UR10e is classified as a collaborative robot, meaning it is designed to work alongside humans in a shared workspace. It has built-in safety features, such as force and torque sensors, that allow it to detect and respond to contact with objects or humans.

Payload Capacity:

The UR10e has a payload capacity of 10 kilograms (about 22 pounds). This means it can handle objects or tools weighing up to 10 kg during its operations.

Reach:

The robotic arm has a maximum reach of 1300 mm (1.3 meters or approximately 51 inches). This allows it to access a wide working area.

Degrees of Freedom (DoF):

The UR10e has six degrees of freedom, allowing it to move in six different directions. This flexibility enables the robot to perform a variety of tasks and maneuvers.

Programming Interface:

Universal Robots provides an intuitive and user-friendly programming interface for the UR10e. Programming can be done through a teach pendant, where operators can guide the robot through its motions and tasks.

What is ROS?

ROS stands for Robot Operating System. It is an open-source middleware framework used for developing robotic software. Despite its name, ROS is not an actual operating system but rather a set of software libraries and tools that provide services to hardware abstraction, device drivers, communication between processes, package management, and more.

ROS is widely used in the field of robotics to simplify the development of robotic systems by providing a standardized and modular framework. It enables developers to build and share code, making it easier to create complex and integrated robotic applications.

In the context of ROS (Robot Operating System), let's discuss some key components like packages, libraries, and other aspects:

1. **Packages:**
    - **Definition:** A ROS package is a directory that contains software related to a specific functionality or task. It includes code, configuration files, and other resources needed for that functionality.
    - **Structure:** Each package has a specific structure, including directories like **src** (source code), **launch** (launch files), and **config** (configuration files).
    - **Independence:** ROS packages are designed to be modular and independent, allowing for the development of specific functionalities that can be easily integrated into larger robotic systems.
2. **Libraries:**
    - **ROS Libraries:** ROS provides a set of libraries that help in developing robotic software. These libraries cover various aspects, such as communication, hardware abstraction, visualization, and more.
    - **Standard Libraries:** In addition to ROS-specific libraries, developers often use standard libraries like C++ Standard Template Library (STL) or Python libraries for general-purpose programming tasks.

3. **Nodes:**
    - **Definition:** A ROS node is an executable that performs a specific task. Nodes communicate with each other by passing messages, allowing for a distributed and modular architecture.
    - **Communication:** Nodes in ROS communicate using a publish-subscribe model, where one node publishes messages to a specific topic, and other nodes can subscribe to that topic to receive the messages.
4. **Topics:**
    - **Definition:** Topics are named communication channels used by ROS nodes. Nodes can publish messages to a topic, and other nodes can subscribe to that topic to receive the messages.
    - **Example:** Topics can represent sensor data (e.g., camera images), control commands, or any other type of information that needs to be shared between nodes.
5. **Services:**
    - **Definition:** Services in ROS provide a way for nodes to request a specific operation from another node. It is a synchronous form of communication.
    - **Example:** A service might be used to request a node to perform a specific action, and the service will return a response when the action is completed.
6. **Actions:**
    - **Definition:** Actions are a form of communication that allows nodes to perform long-running tasks with feedback and the ability to cancel the task.
    - **Example:** Actions could be used for tasks like robot navigation or grasping an object, where the node provides feedback on the progress of the task.

Overall, ROS provides a flexible and modular framework for developing robotic software, allowing developers to create reusable packages, libraries, and nodes to build complex robotic systems. It promotes collaboration and code sharing within the robotics community.

MoveIt! Commander

MoveIt! is a widely used motion planning framework for robotic systems. It facilitates the development of robotic applications by providing a set of tools and libraries for motion planning, control, perception, and simulation. MoveIt! Commander is one component of the MoveIt! framework, and it primarily deals with high-level motion planning and control.

Here's an overview of MoveIt! Commander and related packages:

1. **MoveIt! Commander:**
    - Purpose: MoveIt! Commander is a high-level interface that allows users to control robotic arms and plan motions easily.
    - Functionality: It provides a simple API (Application Programming Interface) for users to command robots, plan motions, and execute trajectories. It abstracts away much of the complexity involved in motion planning and control.
    - Usage: Users can employ MoveIt! Commander to send motion planning requests, execute trajectories, and interact with the robot at a higher level without dealing with the intricacies of low-level control.
2. **MoveIt! Planning Scene:**
    - Purpose: The MoveIt! Planning Scene package deals with representing and maintaining the current state of the robot and its environment.
    - Functionality: It manages information such as the current joint positions, collision objects, and the robot's kinematic model. This data is crucial for motion planning, collision checking, and executing trajectories without collisions.
3. **MoveIt! MoveGroup Interface:**
    - Purpose: The MoveIt! MoveGroup Interface is a component that simplifies interaction with a group of joints on a robot.
    - Functionality: It provides a way to specify a group of joints on the robot and perform various operations such as planning and executing motions for that group. This is useful when dealing with multi-jointed robots, as it allows users to focus on specific subsets of joints.

4. **MoveIt! Perception:**
    - Purpose: MoveIt! Perception integrates perception capabilities into the MoveIt! framework.
    - Functionality: It enables robots to perceive their environment, such as detecting objects, and use this information for motion planning. Perception data, when combined with motion planning, allows robots to perform tasks like picking and placing objects.
5. **MoveIt! Visualization:**
    - Purpose: MoveIt! Visualization is responsible for visualizing the robot, its environment, and planned trajectories.
    - Functionality: It provides tools for visualizing the robot model, the planning scene, and the trajectories that the robot is planning or executing. Visualization is crucial for debugging and understanding the robot's behavior.

These packages, along with others in the MoveIt! ecosystem, collectively provide a comprehensive framework for developing robotic applications. MoveIt! is widely used in research, academia, and industry for tasks such as robot manipulation, grasping, and path planning. It supports a variety of robotic platforms and is compatible with popular robotic simulation environments.

Commanding Goal Position in MoveIt!

In the context of MoveIt!, commanding the goal position refers to specifying the desired final position and, optionally, orientation of a robot's end-effector or a specific group of joints. The goal position represents where you want the robot to move within its workspace. Here are the key components involved in specifying a goal position in MoveIt!:

MoveIt! Setup: Before commanding any motion, you need to set up MoveIt! in your robotics application. This involves initializing the MoveIt! library, configuring the robot model, and creating instances of MoveIt! components such as RobotCommander and MoveGroupCommander.

Specify the Goal Pose: Once MoveIt! is set up, you'll define the desired goal pose. The pose typically includes both the position (x, y, z) and the orientation (represented by quaternion values: x, y, z, w) of the end-effector or joints you want to control. This can be done using appropriate data structures provided by MoveIt! in your programming language of choice (e.g., Python, C++).

Set the Goal Pose for the MoveGroupCommander: For a specific group of joints or the end-effector, you'll use the MoveGroupCommander to set the goal pose. The set_pose_target or similar function is used to define the target pose that you want the robot to achieve.

Plan the Motion: After setting the goal pose, you use MoveIt! to plan a motion trajectory from the current robot state to the specified goal pose. The planning process takes into account factors such as joint limits, collision avoidance, and kinematic constraints.

Execute the Trajectory: If the motion planning is successful, you can then execute the generated trajectory to move the robot to the specified goal position. The go or move function is commonly used for this purpose

Path Planning

Our primary planner used in motion planning is **OMPL**. OMPL, or the Open Motion Planning Library, is a widely used library for motion planning in robotics. It provides a variety of motion planning algorithms, and one of its notable planners is the Rapidly-exploring Random Tree (RRT).

Rapidly-exploring Random Tree (RRT): RRT is a popular probabilistic motion planning algorithm employed in robotics. It efficiently explores the configuration space by generating a random tree structure from the initial state towards the goal state. RRT rapidly explores the space by biased random sampling, expanding the tree toward unexplored areas. This approach is particularly effective in high-dimensional spaces and is widely utilized for path planning in various robotic applications.

In the context of OMPL, you can use the RRT planner to generate feasible paths for a robotic system. By incorporating RRT into the OMPL framework, developers and researchers can leverage its flexibility and efficiency for solving complex motion planning problems. The implementation details, parameters, and customization options for the RRT planner in OMPL can be found in the library's documentation.

If OMPL fails to work then we are using CHOMP as our backup, which stands for Covariant Hamiltonian Optimization for Motion Planning, is a motion planning algorithm used in robotics. It is particularly designed for continuous path planning problems, such as those encountered in robot motion planning tasks. CHOMP falls under the category of optimization-based planners and is known for its effectiveness in generating smooth and collision-free trajectories for robotic systems.

Covariant Hamiltonian Optimization for Motion Planning or CHOMP, is an optimization-based motion planning algorithm widely employed in robotics. Utilizing principles from Hamiltonian mechanics, CHOMP formulates motion planning as an optimization task, seeking

to generate smooth and collision-free trajectories for robotic systems. The algorithm minimizes a cost function that incorporates factors like trajectory smoothness and obstacle avoidance. Known for its ability to produce natural and efficient movements, CHOMP excels in collision avoidance, making it suitable for applications such as robot arm motion planning and grasping tasks. Its open-source implementations further enhance accessibility, allowing researchers and developers to integrate CHOMP into their robotic systems for continuous path planning in complex environments. While CHOMP is effective in specific scenarios, the choice of a motion planning algorithm ultimately depends on factors like system requirements and environmental considerations within the field of robotics.

Master Control

Selecting a Path: Imagine you have a robot, like the UR10e. When you want the robot to perform a task, you plan a specific path or trajectory that it should follow to get the job done. This path is essentially a series of coordinates that tell the robot where to move in its workspace.

Transferring Coordinates to the Master Controller: Once you've selected this path, you need to make sure it's safe for the robot to follow. You transfer each coordinate of the selected path to the master controller of the UR10e. This master controller is like the brain of the robot, and it oversees and manages all the movements.

Verification for Self-Collision and Workspace Violation: The master controller then performs a crucial task: it verifies the selected path. It checks whether, as the robot moves along the planned trajectory, any part of the robot collides with itself (self-collision) or if any part goes outside the allowed workspace (workspace violation). This verification step is essential to ensure the robot won't harm itself or operate in an unsafe zone.

Confirmation of Waypoint Achievement: If the verification process goes smoothly and the master controller gives the green light, it means that the selected path is safe and achievable. Each waypoint along the path is checked, and the master controller confirms that the robot can reach each of these points without encountering any issues.

In simpler terms, this process is like double-checking that the directions you've given to the robot won't make it bump into itself or go somewhere it shouldn't. It's a safety measure to ensure that the robot can follow the planned path without causing any harm or violating its workspace constraints. Once the verification is successful, you can be confident that the robot will move along the selected path safely.

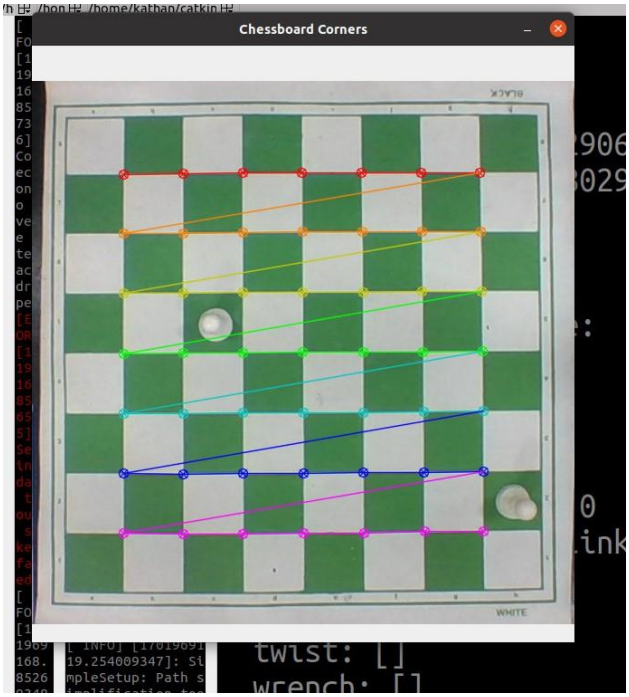
Mechanical Work

For the mechanical aspect we have 3D printed a camera stand which can accommodate majority of webcam. We went through this approach as we didn't knew which camera would be best fit for our project. At the end we ended up using MI webcam which was perfect for getting a good image difference. Also, we ended up 3D printing 2 straight claws which fitted exactly with our end actuator. This reason for using extend claws was that it can pick all pieces of different length and width.

Image Difference Algorithm

The main objective of this algorithm is to detect changes in the chess pieces moved by the player and feed this move (for example: e2e4, g1f3) to the Stockfish AI. Let's go through the algorithm step by step. First, the camera captures a photo of an empty chessboard and detects all the inner edge corners of the board using the built-in OpenCV library.

```
15  # Find the chessboard corners
16  ret, corners = cv2.findChessboardCorners(gray, (rows, cols), None)
17
18  if ret:
19      cv2.drawChessboardCorners(image, (rows, cols), corners, ret)
20      cv2.imshow('Chessboard Corners', image)
21      cv2.waitKey(0)
22      cv2.destroyAllWindows()
23  else:
24      print("Chessboard corners not found.")
25
```



After that, it finds the mean of all the (X, Y) coordinates of all the detected corners so that it can determine the center point of the chessboard. Following that, we create an 8x8 grid, taking the center of the grid as our mean center point. When constructing the grid, we ensure that the length of each square is exactly the same as the distance between two corners found during corner detection.

What we are doing here is that we are converting pixels location to real world location. The process of converting pixel locations to real-world coordinates involves a two-step transformation. Initially, the inverse transformation is employed to map pixel coordinates in an image to real-world coordinates in the camera's frame, considering intrinsic and extrinsic camera parameters. Subsequently, a forward orientation transformation is applied to further convert these camera frame coordinates into the robot's base link frame, aligning them with the robot's kinematics and spatial orientation in the world. This comprehensive transformation process ensures that the coordinates derived from pixel locations are accurately represented in the context of the robot's coordinate system, facilitating effective control and navigation within its workspace.

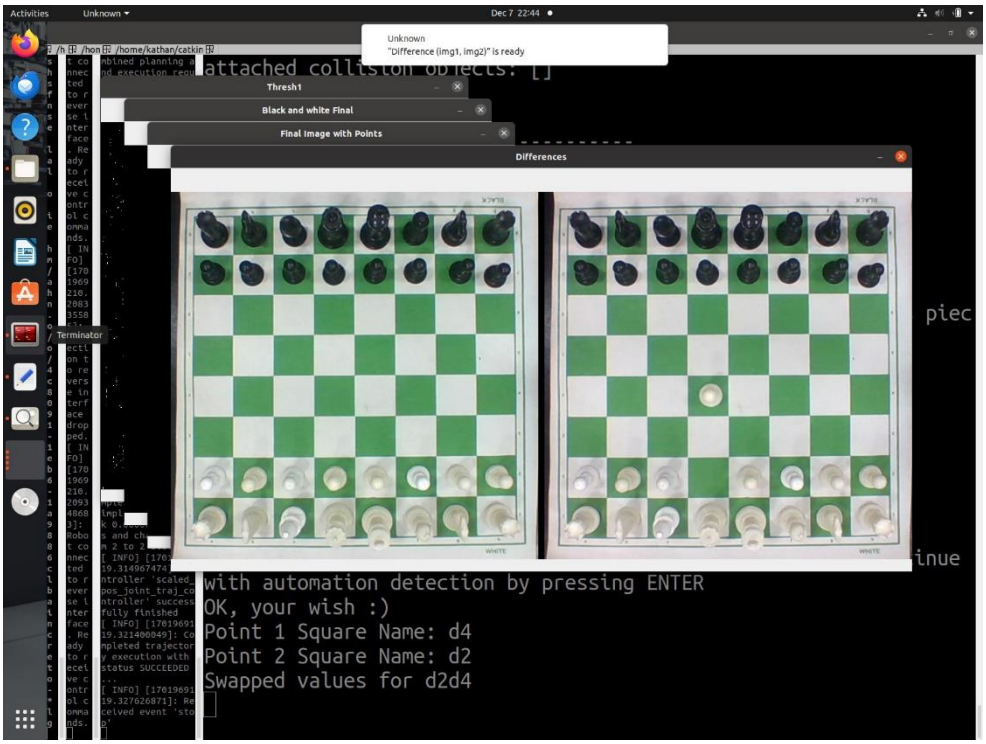
Now that we have created an imaginary 8x8 grid on our chess photo, precisely matching the size of the chessboard, we can easily calculate the mean coordinate of each of the 64 squares. With the 4 coordinates of the corners for each square, we compute the mean coordinate and assign it a name following the chessboard square naming notation, such as a1, b2, c4, f4, and so on. Subsequently, we generate a CSV file (an Excel file) with 4 columns and 64 rows to store this information.

	A	B	C	D
1	Square	x	y	EorF
2	a8	328	70	2
3	b8	395	70	2
4	c8	462	70	2
5	d8	529	70	2
6	e8	596	70	2

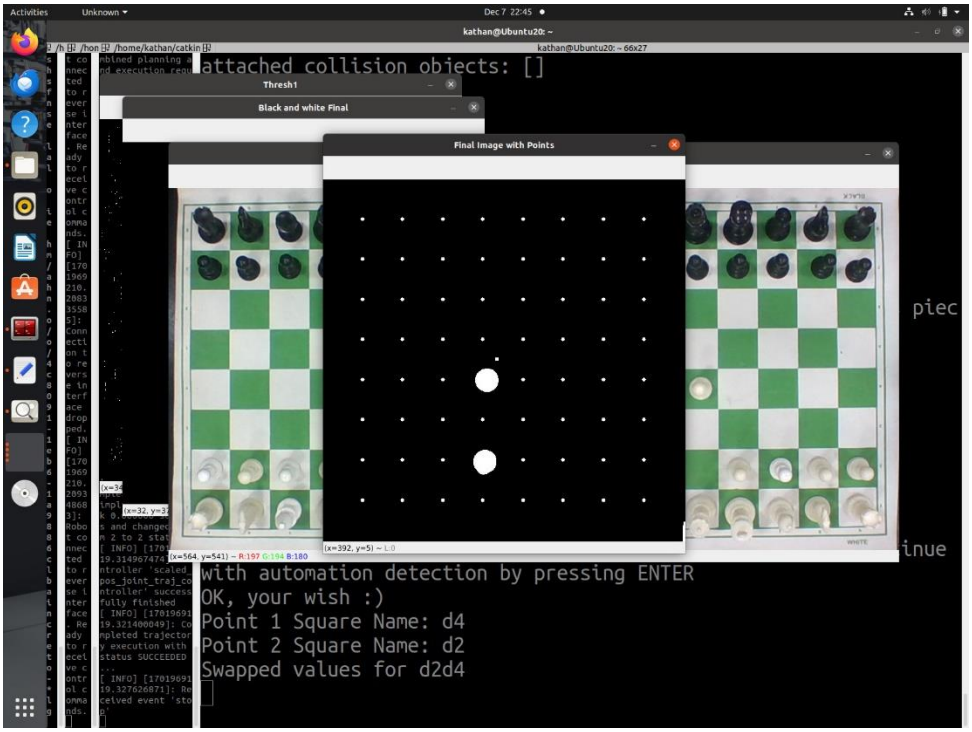
The "Square" column contains all the names of the squares in chess notation, and the "X" and "Y" columns contain the corresponding (X, Y) coordinates. The "EorF" column stands for "Empty or Filled." This column determines whether the square has a white piece, a black piece, or is empty (4 for white piece, 2 for black piece, and 0 for empty). Initially, rows from a1, b1, c1,... h1, and a2, b2, c2,... h2 will have an "EorF" value of 4, while rows from a7, b7, c7,... h7, and a8, b8, c8,... h8 will have an "EorF" value of 2. The rest of the squares are initially empty, so they will have a value of 0. This "EorF" value will be updated after every move to track whether the square is occupied by a white or black piece.

After this, the game begins, and the player moves the white piece. The algorithm is designed in such a way that it captures an image before the player moves the white piece, and it captures another image after the player has moved the white piece.

2 input images:



Final output of the image difference algorithm:



```
# Grayscale
gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Find the difference between the two images using absdiff
diff = cv2.absdiff(gray1, gray2)
cv2.imshow("Difference (img1, img2)", diff)

# Apply threshold
matrix, thresh = cv2.threshold(diff, 25, 255, cv2.THRESH_BINARY)
cv2.imshow("Thresh1", thresh)

# Apply morphological opening operation to clean up the binary image
kernel = np.ones((5, 5), np.uint8)
finalimage = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)
cv2.imshow("Black and white Final", finalimage)

# Load the chessboard coordinates from the CSV file
chessboard_coordinates = pd.read_csv('chessboard_coordinates.csv')

# Loop through the coordinates and draw white points on the finalimage
for _, row in chessboard_coordinates.iterrows():
    x = int(row['x'])  # Assuming 'x' is the column name for the x-coordinate in your CSV
    y = int(row['y'])  # Assuming 'y' is the column name for the y-coordinate in your CSV
    cv2.circle(finalimage, (x, y), 3, (255, 0, 0), -1)

# Show or save the final image with the drawn points
cv2.imshow("Final Image with Points", finalimage)
```

First, we convert both images to grayscale, and then we subtract the two images. By doing this, only the changes between the first and second images will be highlighted as white dots.
To enhance the image and reduce the noise of the image we need to make changes in our image difference photo. To do that first we first apply thresholding to the image for the better results. Thresholding is a basic image processing operation that is used to segment an image into regions based on intensity values. The idea is to set a threshold value, and then classify each pixel in the image as belonging to either one of two classes: pixels with intensity values below the threshold (background) or pixels with intensity values above the threshold (foreground).

```
# Apply morphological opening operation to clean up the binary image
kernel = np.ones((5, 5), np.uint8)
finalimage = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)
cv2.imshow("Black and white Final", finalimage)
```

In the world of images, this grid is used as a tool to make pictures clearer or change them in some way. It's like a filter or a stamp that you place on a picture to give it a new look.

Here's how it works:

1. **Filtering:** You put this grid (kernel) on top of each pixel in a picture.
2. **Changes Occur:** The numbers in the grid decide how much the color of each pixel should change. Some grids might make things blurrier, some might make edges stand out more, and others might emphasize certain details.
3. **Different Tasks:** Depending on what you want to do with a picture, you might use different grids (kernels). For example:
    - If you want to make things smoother, you might use a grid that blurs things.
    - If you want to make the edges sharper, you might use a grid that highlights differences in color.
4. **Structuring Element:** Sometimes, this grid (kernel) is used to look at groups of pixels around a point. It's like saying, "Hey, let's look at the neighbourhood around this pixel and decide what color it should be based on the colours of its neighbours."

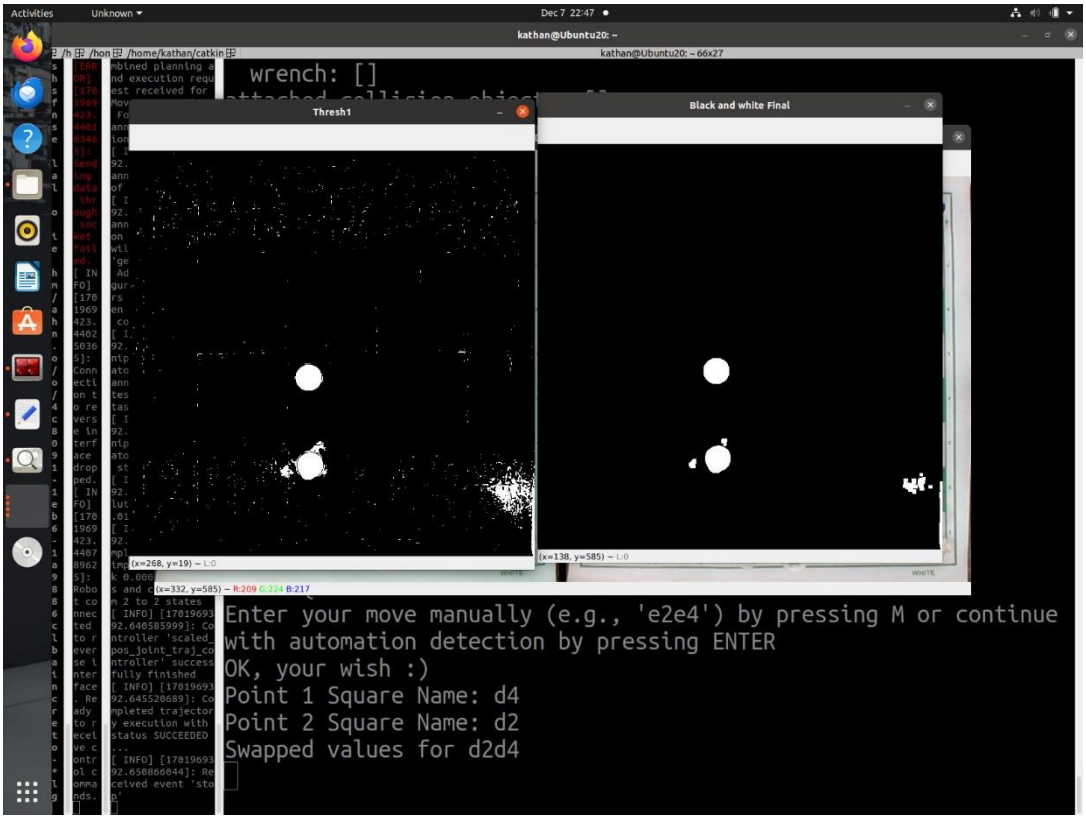
So, in simple terms, a kernel is just a little grid or pattern that helps us change or enhance images in different ways. It's a tool that image experts use to give pictures certain effects or make them better for different purposes.

Here we have applied morphological opening operation using the specified kernel. The opening operation consists of the following steps:

- **Erosion:** The binary image (**thresh**) is eroded, which means that small regions and details are removed. Erosion is particularly effective in eliminating noise and small objects.
- **Dilation:** The result of the erosion is then dilated. Dilation helps to restore the overall shape of larger objects and connect broken structures. It is effective in smoothing the boundaries and filling gaps.

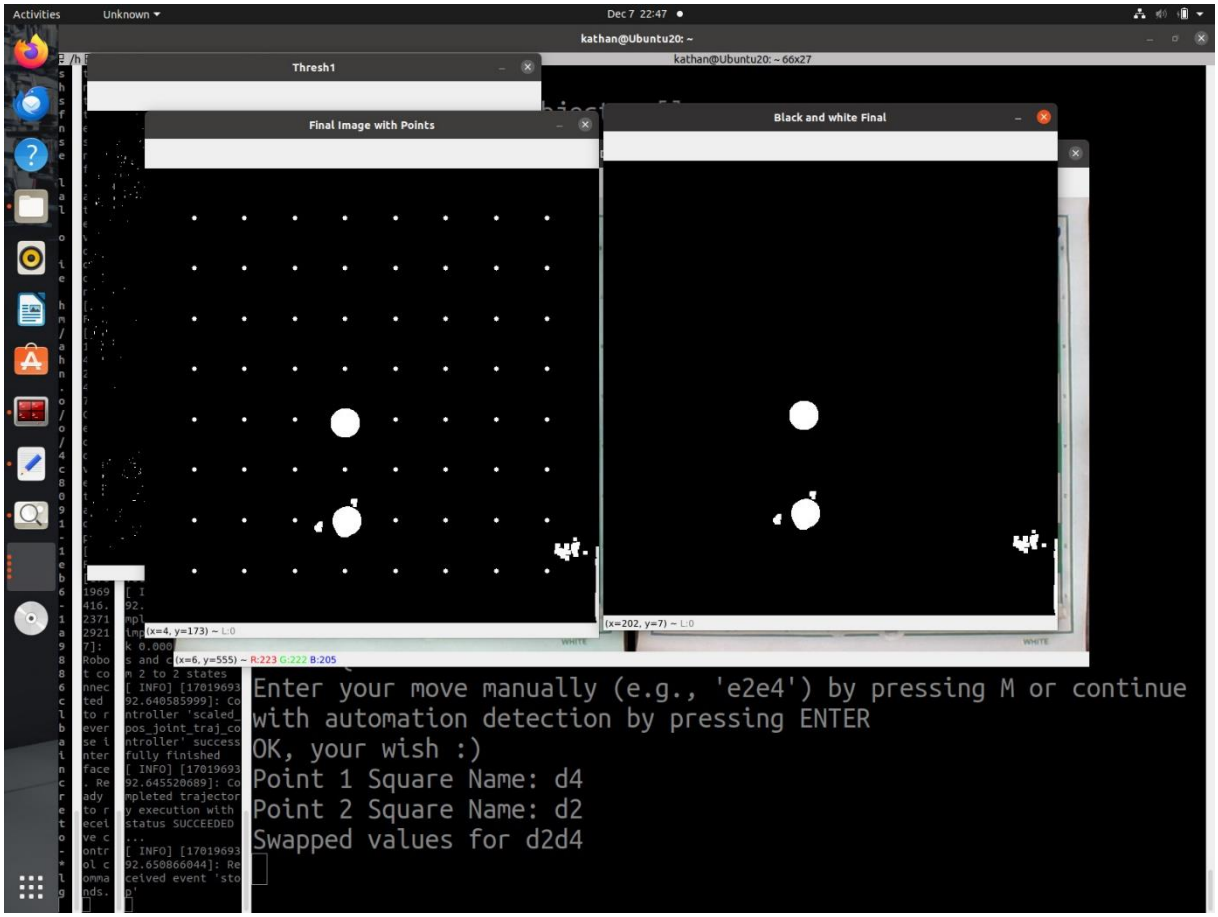
Combined Effect (Opening and Closing):

Opening (Erosion followed by Dilation): Opening is useful for removing small objects, smoothing boundaries, and separating objects that are close to each other. It is accomplished by applying erosion followed by dilation.

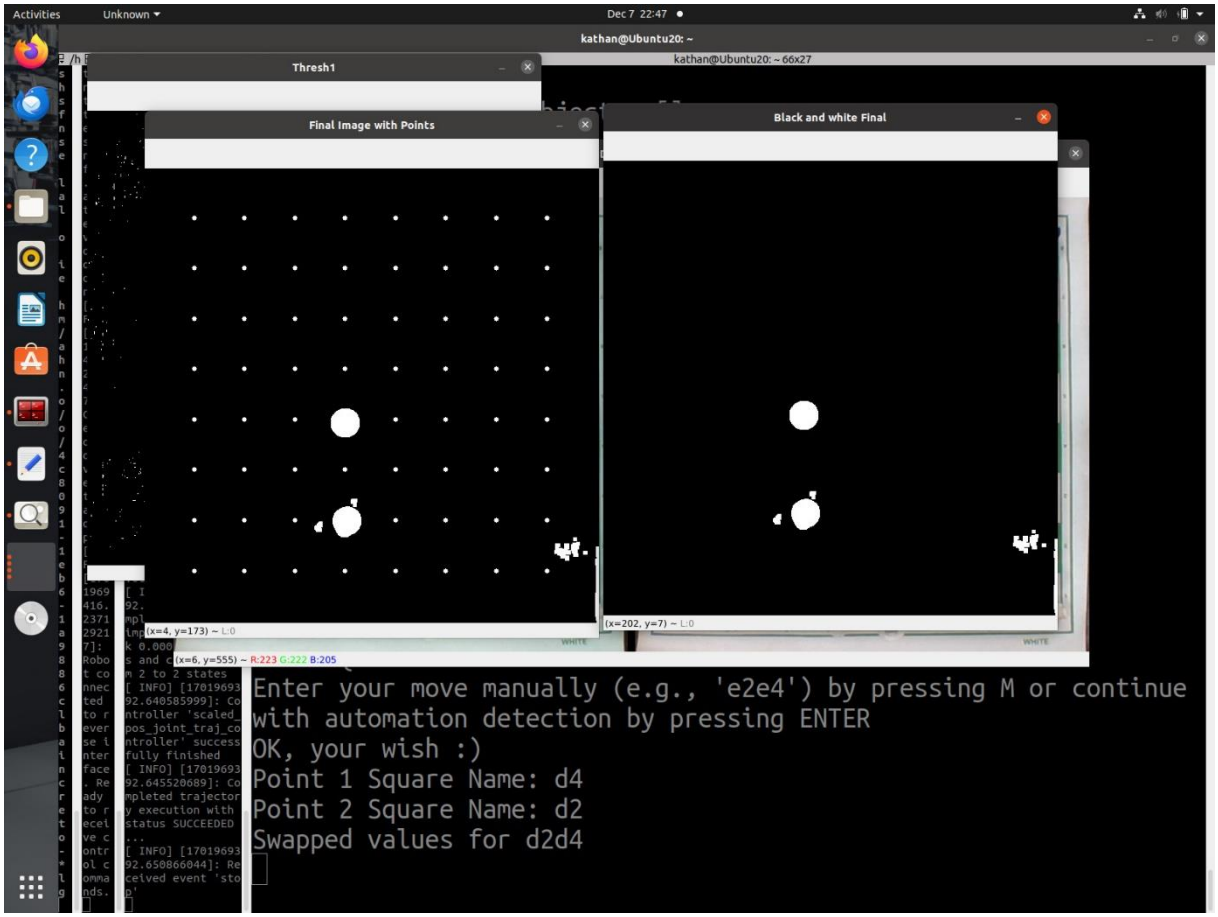


Thresholding followed by opening (Erosion followed by Dilation)

After enhancing our image and reducing noise in the image difference, we add and draw those 64 dots on our image. These 64 dots represent the mean coordinates of the 64 different chess squares obtained from our first image.



From here, we can observe two significant irregular white circles in the image and 64 dots on the grid. Next, we run a loop that provides us with two coordinates from those 64 dots , **which have the most white region near them**. This results in obtaining two chess square positions (like e2e4), indicating where White has moved its pieces. This notation is then provided to AI, specifically Stockfish. Stockfish suggests the next best move, and the UR10e robot moves the pieces from one square to another as indicated by Stockfish.



Now, initially, square e2 had a value of ‘4’ in EorF, and e4 had a value of ‘0’ in EorF. However, after White played e2e4, we simultaneously change the value of e4 to ‘4’ and e2 to ‘0’ in EorF in the CSV file. With every move, we make changes in the CSV file and update EorF, ensuring we stay up-to-date with whether a square has a White piece, Black piece, or is empty. Similarly, the CSV file will be updated in the EorF column for squares moved by UR10e.

Now, the question arises: why can't we just move the pieces and finish the game? What is the need for updating the CSV file's EorF column simultaneously?

The EorF column becomes crucial when White captures Black pieces and when deciding the square name sequence for providing notation to the Stockfish AI. For instance, if two dots appear on squares g1 and f3, the algorithm determines whether White played g1f3 or f3g1 by

examining the EorF column. If g1 has an EorF value of 4 (indicating a White piece on g1) and f3 has a value of 0, the algorithm deduces that White played 'g1f3'.

EorF is also valuable when White captures a Black piece. In such instances, both squares already have pieces, and the image displays two dots on those squares. However, utilizing the EorF column in the CSV file helps us discern which square had a piece of which color, providing the necessary information for notation.

To determine the notation what white has played we have set some if-else conditions:

```
# Check conditions and swap values in CSV file accordingly
eorf_square_name1 = chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name1, 'EorF'].values[0]
eorf_square_name2 = chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name2, 'EorF'].values[0]

if eorf_square_name1 == 0:
    # Swap values in CSV file
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name1, 'EorF'] = eorf_square_name2
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name2, 'EorF'] = 0

    # Swap square names
    square_name1, square_name2 = square_name2, square_name1

    print(f"Swapped values for {square_name1}{square_name2}")

elif eorf_square_name2 == 0:
    # Swap values in CSV file
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name1, 'EorF'] = 0
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name2, 'EorF'] = eorf_square_name1

    print(f"Swapped values for {square_name1}{square_name2}")

elif eorf_square_name1 == 4 and eorf_square_name2 == 2:
    # Swap values in CSV file
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name1, 'EorF'] = 0
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name2, 'EorF'] = 4

    print(f"Swapped values for {square_name1}{square_name2}")

elif eorf_square_name1 == 2 and eorf_square_name2 == 4:
    # Swap values in CSV file
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name1, 'EorF'] = 4
    chessboard_coordinates.loc[chessboard_coordinates['Square'] == square_name2, 'EorF'] = 0

    print(f"Swapped values for {square_name2}{square_name1}")

# Save the modified CSV file
chessboard_coordinates.to_csv('chessboard_coordinates.csv', index=False)
```

Integration with Stockfish AI

In the context of the UR10e robotic arm playing chess, the integration with Stockfish AI serves as a critical component for strategic decision-making during the game. Stockfish is a powerful open-source chess engine renowned for its advanced chess analysis and evaluation capabilities. By linking the robotic arm with Stockfish, the project elevates the chess-playing experience to a more sophisticated level.

The integration involves a seamless communication channel between the robotic arm and the Stockfish AI. As the robotic arm makes a move on the physical chessboard, an image capture mechanism detects the changes on the board. The algorithm processes this information, translating the move into chess notation. This notation is then relayed to Stockfish for analysis. Stockfish evaluates the current state of the chess game, considers potential moves, and determines the optimal next move for the robotic arm.

This integration not only introduces an element of artificial intelligence into the chess-playing scenario but also enables the robotic arm to make informed and strategic decisions based on Stockfish's analysis. The collaborative nature of the project emerges as the robotic arm and Stockfish work in tandem, blending the precision of robotic movements with the analytical prowess of a powerful chess engine. As a result, the chess game becomes a dynamic interplay between human opponents, the robotic arm, and the strategic insights derived from Stockfish AI, showcasing the fusion of robotics, artificial intelligence, and strategic gaming.

Conclusion

In conclusion, the integration of the UR10e robotic arm into a chess-playing scenario showcases the versatility and collaborative potential of modern robotic systems. The utilization of ROS and MoveIt! establishes a foundation for efficient and standardized robotic control, while the incorporation of advanced planning algorithms like RRT and CHOMP ensures adaptive and dynamic motion planning. The image difference algorithm acts as a key enabler, facilitating real-time communication between the robot and the chessboard, thereby enhancing the interactive and strategic aspects of the game. This project not only highlights the technical prowess of collaborative robotics but also points towards broader applications in human-robot interaction and collaborative decision-making scenarios. As robotics continues to advance, projects of this nature serve as a testament to the evolving capabilities and diverse applications of robotic systems in collaborative and dynamic environments.