Automated Control of Multiple Ferromagnetic Bots Using a 2D Solenoid Grid

Abstract

This report presents the development and optimization of an autonomous control system for multiple ferromagnetic robots (bots) operating on a grid of solenoids. Aimed at automating coordinated tasks such as shape-changing of a compliant elastic band attached to the bots, navigation through obstacles, and object manipulation within a planar workspace, this work replaces the manual activation of solenoids with an integrated system of camera-based tracking, Arduino control, and Python algorithms. The vision-based algorithm autonomously determines the required shapes and paths for the bots. This streamlines the sequence of solenoid activations. The development process addressed technical challenges related to hardware connectivity, camera calibration, and algorithmic adjustments, implementing solutions such as recalibrating camera settings and optimizing coil activation thresholds to enhance accuracy and reliability. This report details the methodologies, including camera calibration, positional measurement, and the main processing logic, and explores the function logic and algorithms for coil activation and updates. The findings underscore the significance of adaptive algorithms and systematic troubleshooting in creating robust and efficient autonomous robotic systems, with potential applications in various fields requiring precise, untethered actuation of small-scale bots.

Introduction

Magnetic systems have emerged as a favored approach in robotics due to their unique advantages, particularly in enabling untethered operations at small scales. Historically, controlling magnetic robots has relied on global magnetic fields. This approach presents significant challenges in independently manipulating multiple robots. The primary difficulty with global magnetic fields is their uniform influence across the entire workspace, resulting in simultaneous and often undesirable movements of all bots when attempting to control just one. In contrast, the use of a solenoid grid to generate local magnetic fields offers a distinct advantage by allowing for the independent control of individual bots within specific areas. This method effectively overcomes the limitations posed by global magnetic fields, enabling precise and isolated actuation of each robot.

This work focuses on the solenoid grid-based setup for independently actuating ferromagnetic robots (bots), developed and tested at the M2D2 lab of IISc Bengaluru [1]. While this setup has demonstrated the potential for precise control, the process has traditionally relied on manual sequencing of coil activations, which is both time-consuming and prone to errors. For complex and coordinated tasks, such as navigating through obstacles and manipulating objects, manual control is infeasible and inefficient.

To address these challenges, this project aims to automate the control of bots using vision-based algorithms and Arduino-controlled solenoids. Automation is essential for enhancing the efficiency and precision of the system, allowing it to autonomously determine the necessary shapes, paths, and solenoid activations required for various tasks. The vision-based algorithm will identify the positions of the bots and nearby solenoids, generating a series of target positions and corresponding coil activation sequences. This approach ensures that the bots can autonomously navigate, avoid obstacles, and manipulate objects with high accuracy and reliability.

The development of this autonomous bot control system involved overcoming several technical challenges related to hardware connectivity, camera calibration, and algorithmic adjustments. By implementing targeted solutions such as recalibrating camera settings, optimizing coil activation thresholds, and refining bot movement algorithms, the system achieved improved accuracy and reliability. This report details the methodology, including camera calibration, positional measurement, and the main processing logic, and discusses the function logic and algorithms for coil activation and updates. The findings underscore the significance of adaptive algorithms and systematic troubleshooting in creating robust and efficient autonomous robotic systems, with potential applications in various fields requiring precise, untethered actuation of small-scale bots.

Problem Statement:

The objective of this project is to develop an automated system for coordinated tasks performed by multiple ferromagnetic robots. These bots, consisting of slender structures, are equipped with a compliant elastic band. The compliant band can deform to change its shape, allowing the bots to navigate through obstacles and manipulate objects in a planar workspace. Currently, these actions are manually controlled by listing the solenoids' activation sequence for each bot. The goal is to create a vision-based algorithm that determines the compliant band's shape at different positions in the workspace and automatically generates the solenoid activation sequence based on the desired ferrobot path. Integration of camera-based tracking, Arduino control, and Python-based algorithms will enhance the precision, efficiency, and overall functionality of this magnetic robotic system, enabling autonomous navigation, obstacle avoidance, and target object manipulation.

Experimental Setup:

A setup consisting of a grid of electromagnets (solenoids) precisely controls the movement of ferrobots (bots) within a designated work area. The strength of the magnetic fields attracting the bots can be adjusted by varying the current supplied to each solenoid. An overhead camera tracks the bots' positions and sends this information to a controller for precise manipulation.

The bots themselves are equipped with special holders containing 20mm diameter ferromagnetic balls. These holders have built-in springs for smooth movement and Teflon feet to minimize friction. Cleverly, the magnetic field can pull the bots down, causing the holders to contact the surface and create friction. This acts as an automatic braking system, enhancing control and stability. This system allows for the manipulation of multiple bots within the defined workspace.

Methodology For Motion of Bots

A group of four bots connected by a flexible compliant elastic band, forming a circle and equidistant from each other, is tasked with moving towards a target object, grabbing it, and returning. This task is achieved through a combination of various motion and formation changes, controlled by specific code modules. Initially, the positions of the bots are measured using a camera calibration module, and their positions are captured using a position measurement function. The bots, starting in a random shape, can translate in any direction while maintaining their shape. Using the square formation module, the bots first form a square shape. Then, the square shape is transformed into a diamond shape using the diamond formation module. From the diamond shape, the bots change to an I shape via the I formation module, and then to a C shape using the C formation module. In the C shape, the bots translate towards the target object. Upon reaching the target, they form an O shape using the O formation module to grab the object. Finally, the bots move back to their starting position by translating back.

Modules and Their Functions

1. **Camera Calibration Module**: Measures the positions of the bots using a camera, ensuring accurate position data for subsequent movements.
2. **Position Measurement Function**: Captures frames and measures the positions of the bots, providing the necessary input data for the motion modules.
3. **Translation Module**: Allows the bots to translate up, down, left, or right while retaining their current shape. This module is used for basic movements.
4. **Square Formation Module**: Converts the bots from any random shape into a square shape, setting up a structured starting point for further transformations.
5. **Rotation Module**: Allows the bots to rotate around a fixed centre.
6. **Diamond Formation Module**: Transforms the bots from a square shape into a diamond shape, preparing them for the next formation change.
7. **I Formation Module**: Changes the bots from a diamond shape to an I shape, further adjusting their configuration.
8. **C Formation Module**: Reconfigures the bots from an I shape to a C shape,
9. **Object Grasping (O Formation):** In an “O” shape, the bots collectively grab the object.

Sub Modules Common in All Codes and Their Functions:

1. **Finding influential coils:**

To determine the influential coils for attracting the bot, we need to consider the bot's position and its attraction range. The bot can only be attracted from a specific distance of 33 cm or less. Therefore, any coils located within a radius of 33 cm from the bot's current position are deemed influential coils. These coils have the capability to influence the bot's movements and are crucial for directing it towards the desired task.

2. **Four coil act and send data to Arduino:**

There are PCA9685 modules, each capable of sending out 16 signals. Twelve such boards are connected to the Arduino, with each board having a different address. Each solenoid is connected to the power source through a gate operated by a PWM signal from the Adafruit module. There is a mapping between a given solenoid and the particular pin in a particular board that controls the solenoid. To control the solenoid, we need to know which board and which solenoid to activate. The Python code must communicate with the Arduino code in real-time using serial communication, conveying the board number and pin number determined by the algorithm to the Arduino.

General Flow of Code and Common Module in All the Codes

It aims to autonomously navigate a group of four bots to designated target positions using a combination of visual feedback, positional analysis, and magnetic actuation. Initially, video frames are processed to detect and locate all four bots in both world and image spaces. The centre position of the detected bots is calculated, and vectors from this centre to each bot help determine angles crucial for assigning bots to their respective target positions. A magnetic actuation system controlled by an Arduino then activates specific coils, guiding each bot towards its target based on calculated angles and distances. Continuous position updates ensure real-time adjustments to coil activations, optimizing navigation efficiency. The process iterates until all bots reach their targets, ensuring precise and efficient navigation through careful monitoring and adjustment of bot positions and coil activations. The flowchart of this process is shown in Fig. 1.

1. Objective:

- To autonomously navigate a group of four bots to designated target positions using visual feedback, positional analysis, and magnetic actuation.

2. Bot Detection:

- Acquire and process video frames to detect bots.
- Identify and record the coordinates of the bots in both world and image spaces.
- Ensure all four bots are detected before proceeding.

3. Center Calculation:

- Calculate the center position of the detected bots.
- Compute vectors from the center to each bot.
- Determine the angles between these vectors to assist in bot assignment.

4. Target Position Generation:

- Generate target positions according to the requirement.
- Assign each bot to a target position based on the calculated angles and distances.

5. Magnetic Actuation System:

- Control a magnetic actuation system using an Arduino.
- Activate specific coils in a grid to guide the bots towards their targets.

6. Bot Navigation:

- Calculate the angle between the bot's current position and the target position.
- Determine the nearest active coil based on this angle.
- Move the bot towards the target by activating the most effective coil.

7. Position Update:

- Continuously update the position of the bots by acquiring new frames.
- Re-evaluate and adjust the active coils based on the updated positions.
- Assess the influence of neighboring coils to optimize navigation.

8. Target Reaching Verification:

- Compare the current and target coordinates of the bots to check for proximity.
- Check if the bots have reached their target positions within a specified tolerance.

9. Iterative Process:

- Repeat the steps of position update and coil activation until all bots successfully reach their assigned target positions.
- Save and analyze frames at each step to monitor progress and ensure accuracy.

10. Outcome:

- Achieve precise and efficient navigation of multiple bots in a controlled environment using the developed method.

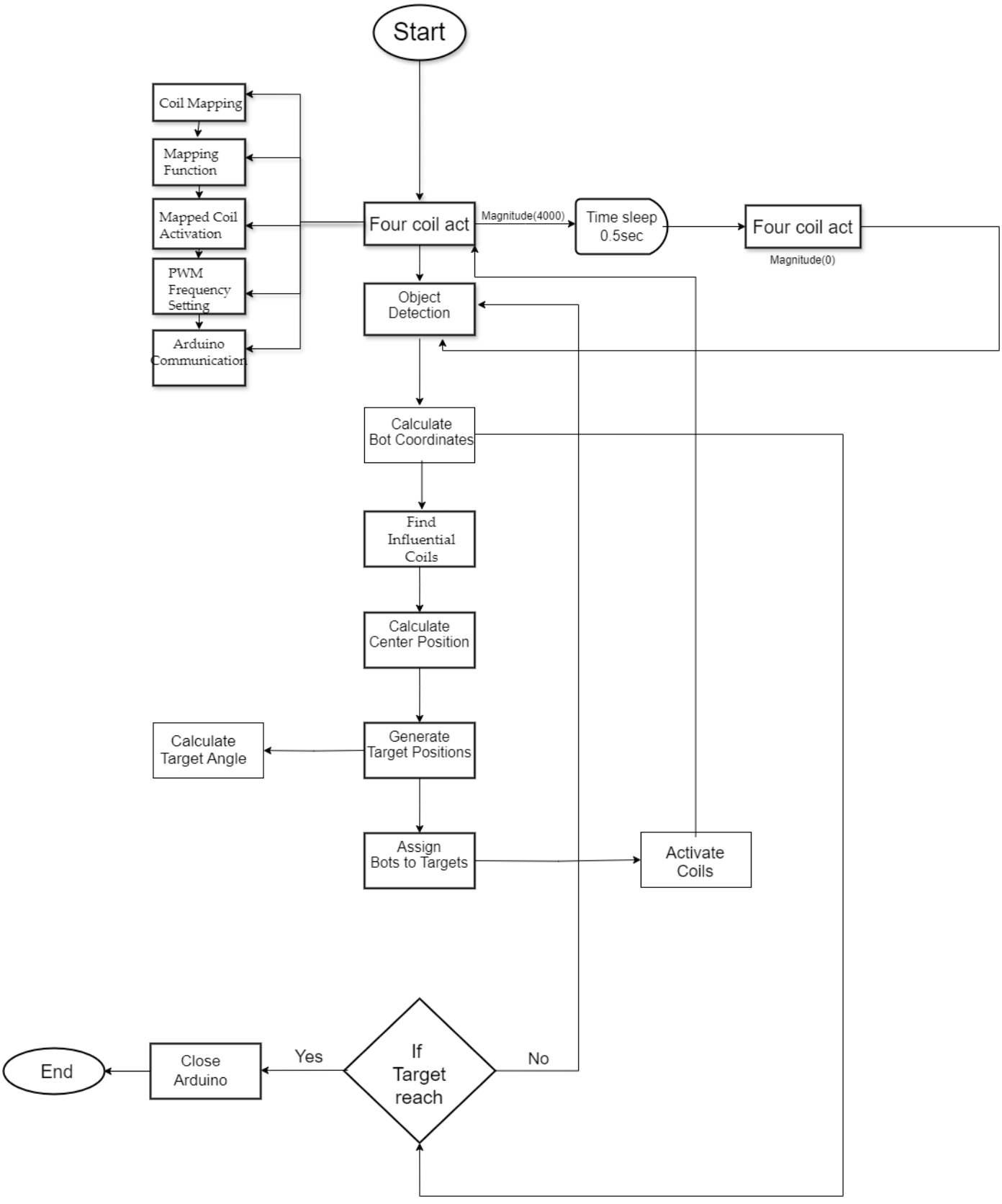

Figure 1 Flowchart of process

Translation

1. Input Parameters:

- The function accepts the bot's current coordinates and the distance to be moved along the specified axis.

2. Distance Conversion:

- It converts the distance from millimetres to meters for standardization, as positions are typically represented in meters.

3. Coordinate Update:

- The function updates the coordinates by adding the converted distance.

4. Output:

- It returns the new coordinates as a tuple, reflecting the bot's updated position.

Rotation around a fixed centre

1. Input Parameters:

The function accepts:

- Movement angles for each robot.
- Distances to move for each robot.
- A central reference position.
- A directional flag indicating clockwise or counterclockwise movement.

2. Direction Handling:

- If moving counterclockwise, adjust the angles slightly positively.
- If moving clockwise, adjust the angles slightly negatively.

3. Displacement Calculation:

- Calculate the x and y displacements for each robot using trigonometric functions based on the adjusted angles and distances.

4. Position Update:

- Add these displacements to the central reference position to get the new coordinates for each robot.

5. Output:

- Return the updated positions as an array.

Square Formation

1. To generate target positions:

Inputs:

- The distance from the center position to each target position.
- The central reference position.

Process:

1. Define a list of rotation angles: 225°, 315°, 45°, and 135° (four corner points for a square, as arctan2 function is used).
2. Initialize an empty list for target positions.
3. Extract x coordinate and y coordinates of center from center position.
4. For each angle in rotation angles:
    - Convert the angle to radians.
    - Calculate x coordinate of target position using the cosine of the angle and the distance from the center position to each target position.
    - Calculate x coordinate of target position using the sine of the angle and the distance from the center position to each target position.
    - Append the calculated x and y coordinate of target position to target positions.
5. Return the list of target positions.

2.To assign target positions to the bots:

Inputs:

- The coordinates of the bots.
- The target positions.

Process:

1. Calculate the center position of all the bots from the coordinates of each bot.
2. Calculate vectors from the center position to each bot.
3. Calculate the angles of these vectors.
4. Initialize an empty list to assign bots the target position`.
5. Create a nested list with four empty sub lists.
6. Sort the angles and obtain their sorted indices.
7. Assign the target positions to bots based on the sorted angle indices:
    - Assign the first sorted index to the first target position.
    - Assign the second sorted index to the second target position.

- Assign the third sorted index to the third target position.
- Assign the fourth sorted index to the fourth target position.

8. Return the list, which contains the target positions assigned to the nearest bots.

Together, these functions generate a set of target positions around a central point and then assign each bot to the nearest target position based on their current locations and directional angles. This approach ensures efficient and systematic allocation of target positions to bots.

Diamond formation

1. Generate Target Positions:

Inputs:

- The distance each target is from the center.
- The central reference coordinates.

Outputs:

- A list of calculated target positions.

Steps:

1. Define a set of rotation angles.
2. Initialize an empty list to store target positions.
3. For each angle, compute the x and y coordinates of the target position using trigonometric functions.
4. Add each calculated target position to the list.
5. Return the list of target positions.

2. Assign Bots to Nearest Target Positions:

Inputs:

- The coordinates of the bots.
- The target positions to be assigned.

Outputs:

- A list of target positions assigned to the nearest bots.

Steps:

1. Calculate the center position from the bot coordinates.
2. Compute vectors from the center to each bot.
3. Determine the angles of these vectors.
4. Sort the bots based on their angles.
5. Assign each robot to the nearest target position based on the sorted angles.
6. Return the list of assigned target positions.

Vertical I formation

1. To Generate Target Positions

Inputs:

- The specified distance from the center to each target.
- The coordinates of the central reference point.

1. Define Angles:

- Specify a set of angles [265, 355, 85, 175] for determining the directions of target positions around the center. The angles are a bit distorted to avoid collision of the bots while moving .

2. Calculate Initial Target Positions:

For each angle in the defined set:

- Convert the angle to radians.
- To compute the x-coordinate, start with the centre x-coordinate. Then, you add the product of the required distance and the cosine of the angle (converted from degrees to radians).
- To compute the y-coordinate, start with the centre y-coordinate. Then, you add the product of the required distance and the sine of the angle (converted from degrees to radians).
- Store the calculated coordinates in target positions.

4. Adjust Y Coordinates:

- Modify the y-coordinate of the first target by decreasing it by 0.03.
- Modify the y-coordinate of the third target by increasing it by 0.03.
- Ensure these modified y-coordinates have a minimum value of 0.03, as this is the minimum possible y-coordinate on the board.

5. Adjust X and Y Coordinates for Specific Targets:

- Adjust the second target's x-coordinate by decreasing it by 0.09 and its y-coordinate by increasing it by 0.06.
- Adjust the fourth target's x-coordinate by increasing it by 0.09 and its y-coordinate by decreasing it by 0.03.

This ensures that, due to the limited workspace, the bots do not collide with each other while trying to reach the target position, and a proper I formation is achieved from diamond shape.

Output:

- Return the list of adjusted target positions.

2. To Assign Bots to Nearest Target Positions

Inputs:

- A list of robot coordinates.

- A list of predefined target positions.

1. Calculate Central Position:

- Determine the central position from the robot coordinates.

2. Compute Vectors and Angles:

- Calculate vectors from the central position to each robot.
- Compute the angles of these vectors relative to a reference direction.

3. Sort Angles and Assign Targets:

- Sort the calculated angles.

- Assign target positions to bots based on the sorted order of their angles.

Output:

- Return the list of target positions assigned to the nearest bots.

C Shape Formation from I shape

Steps:

1. Initialization:

- Identify the bots with the minimum and maximum y-coordinates.

2. Find Extremes:

- Iterate over the robot coordinates to find the bots with the lowest and highest y-coordinates.

3. Adjust Extreme Positions:

- For the bot with the maximum y-coordinate, move it 0.066 meters to the right.
- For the bot with the minimum y-coordinate, move it 0.066 meters to the right.

4. Update Positions:

- Create a new list of bot positions, updating the positions of the bots with the minimum and maximum y-coordinates.
- Keep the positions of the other bots unchanged.

5. Sort Positions:

- Sort the updated list of positions by y-coordinate to ensure the positions are ordered correctly.

6. Calculate Range and Number of Bots:

- Calculate the range of y-coordinates between the updated minimum and maximum y-coordinates.
- Define the number of bots.

7. Form C Shape:

- Adjust the y-coordinates of the bots (excluding the ones with the minimum and maximum y-coordinates) to form a "C" shape.

8. Output:

- Return the list of updated positions that form a "C" shape.

O Formation from C Shape

Steps:

1. Initialization:

- Identify the initial bots with the minimum and maximum y-coordinates.

2. Identify Extremes:

- Loop through the list of robot coordinates to find the bots with the smallest and largest y-coordinates.

3. Adjust Extreme Positions:

- For the robot with the maximum y-coordinate, move it slightly to the right by 0.066 meters.
- For the robot with the minimum y-coordinate, move it slightly to the right by 0.066 meters.

4. Update Positions:

- Create a new list to hold the updated positions of all bots.
- Replace the original positions of the minimum and maximum y-coordinate bots with their new positions.
- Keep the positions of the other bots unchanged.

5. Sort Positions:

- Sort the updated list of robot positions by their y-coordinates to ensure they are in the correct order for forming the "C" shape.

6. Calculate Range and Adjustments:

- Calculate the range of y-coordinates between the adjusted minimum and maximum y-coordinates.

7. Form C Shape:

- Adjust the y-coordinates of the bots (excluding the ones with the minimum and maximum y-coordinates) as needed to help form a "C" shape. In this case, the y-coordinates remain unchanged.

8. Output:

- Return the list of bot positions that have been adjusted to form a "C" shape.

Four Coil Actuation and Send Data to Arduino

1. Coil Mapping:

- There is a predefined dictionary `coil_map` that maps specific coil identifiers to different values. For instance, coil numbers like 40,43,47,46,72, 127, 174 mapped to coil numbers 185,186,188,189,187, 191, 192 respectively using this dictionary.

2. Mapping Function:

- There's a helper function (`map_coil(coil)`) that checks if a given coil identifier exists in `coil_map`. If it does, the function returns the mapped value; if not, it returns the original coil identifier. This ensures consistency and correct mapping of coil identifiers.

3. Mapped Coil Activation:

- Once the coil is mapped using the `map_coil` function, the function prints which coil is activated based on the mapping. This step ensures clarity on which physical component or actuator is being controlled.

4. Board and Pin Calculation:

- After determining the mapped coil, the function calculates two important parameters: board number and pin number. These values are crucial for identifying the specific hardware connections on the Arduino board that will control the coil.

5. PWM Frequency Setting:

- Another essential aspect is setting the Pulse Width Modulation (PWM) frequency. This value is determined by taking the minimum of a given magnitude (`mag`) and a predefined maximum limit (4090). PWM frequency controls the power supplied to the coil, regulating its operation effectively.

6. Arduino Communication:

- Finally, the function communicates with an Arduino microcontroller. It sends the calculated board number, pin number, and pwm frequency values to the Arduino. This communication ensures that the Arduino can activate the specified coil with the correct PWM frequency, enabling precise control over magnetic actuators or similar devices.

Finding Influential Coils

1. Initialization:

- Start with empty lists to store influential coils and their coordinates.

2. Find Closest Coil:

- Compute the distance from the bot's current position to each coil in the system.
- Identify the coil that is closest to the bot based on this distance.

3. Identify Neighboring Coils:

- Define a set of neighboring coils around the closest coil. These include coils immediately adjacent as well as diagonally adjacent within a predefined distance.

4. Evaluate Neighboring Coils:

- For each neighboring coil identified:
- Calculate its distance from the bot.
- Determine if the distance falls within a specified range (`0.005 < coil_dist < 0.035`), indicating proximity to the bot.

5. Store Influential Coils:

- Collect the IDs of coils that meet the distance criteria as influential coils.
- Record the coordinates of these influential coils for further processing or action.

6. Return Results:

- Output the list of influential coil IDs and their coordinates.
- Provide these lists to the calling function for subsequent use, typically in tasks where the bot needs to navigate or interact based on proximity to these coils.

Challenges Faced and Solutions Implemented in Autonomous Bot Control System

1) Issue: Faulty Coils

Challenge: Some coils did not switch on. This issue was attributed to loosen connections, power supply from the boards got disconnected and malfunctioning green boards.

Solution: Re-routing connections to alternative green boards resolved the problem, ensuring all coils were operational.

2) Issue: Camera Position Calibration Error

Challenge: A 3 mm error in camera position calibration led to inaccuracies in identifying bot positions and determining influential coils.

Solution: Adjustments were made in the camera calibration process to minimize errors. The system's algorithms were also revised to account for the calibration discrepancy, ensuring more accurate bot positioning and coil identification.

3) Issue: Time-consuming Camera Initialization

Challenge: Initialization of the camera before each run consumed significant time, delaying the execution of subsequent code.

Solution: A strategy was devised to initialize the camera once at the beginning and keep it running throughout multiple operations. This optimization streamlined the process and reduced overall runtime.

4) Issue: Ineffective Bot Movement Towards Coils

Challenge: Bots occasionally failed to move towards influential coils even after activation, leading to operational failures downstream in the code.

Solution: An algorithmic solution was implemented where bots activate coils, move towards them, and then their positions are updated based on actual movement captured in subsequent frames. This autocorrecting mechanism ensured the system proceeded accurately even if bots did not initially respond as expected.

5) Issue: Bot Identification Failure

Challenge: Inaccuracies in bot detection due to overlapping circles in the identification process (handled by pos_mes_fun), Lighting Interference and other external factors.

Solution: Initially, this problem was addressed by adjusting the circularity test parameter, reducing its intensity from C6-0.6 to 0.7. While this change improved the detection of the actual bots, it also led to the unintended consequence of non-bot objects being mistakenly identified as bots.

To resolve this, the code was further refined to include a condition that required the detection of exactly four bots before proceeding with any camera capture operations. This ensured that the system only

continued when the correct number of bots was identified, thereby improving the accuracy and reliability of the bot detection process.

Additionally, the bots were painted white, leaving a black circle on top of each one. This modification helped in image detection by eliminating all other closed boundary surfaces.

When direct sunlight falls on the board, the bots are not detected properly. Therefore, it is preferred to add an obstacle to prevent sun rays from directly hitting the board.

Since the bots have a specific size and are placed on the first line of the board (coil no. 1 to 16), there is a possibility that the top portion of the bot might not be captured during camera calibration. As a result, they might not get detected. To address this, it is advisable to check the image during camera calibration and ensure there is some gap between the square and the edge of the image to accommodate the height of the bot.

6) Issue: Lighting Interference with Binary Board Detection

Challenge: Fluctuations in binary board detection caused by varying lighting conditions, particularly direct sunlight.

Solution: The OpenCV thresholding code (cv2.threshold) was adjusted dynamically by setting an appropriate threshold value (95), tailored to the current lighting conditions in the lab, ensuring stable binary image detection under varying light intensities.

7) Issue: Vertical Distance Impact on Position Measurement

Challenge: The precision of bot position measurements was affected by the vertical distance from the camera to the binary code.

Solution: Adjustments were made in the system to account for varying heights of the binary code (handled by pos_mes_fun) due to change in the board, ensuring accurate position measurements regardless of vertical distance.

8) Issue: Angle Calculation Error near Borderline

Challenge: Inaccuracies in calculating angles using the arctan2 function, especially near the -180° to 180° boundary, due to slight misalignment of bots.

Solution: Implementing boundary conditions in the angle calculation code ensured accurate angle determination, mitigating errors caused by borderline cases.

9) Issue: Bots Oscillating Between Positions

Challenge: Bots oscillated between two positions due to ineffective switching of influential coils or slight positioning errors.

Solution: Increasing the higher threshold value (`0.019`) for positional tolerance prevented bots from oscillating between positions, ensuring stable movement towards targets.

Adjusting the lower threshold value (`0.005 m`) ensured bots only activated coils when necessary, reducing redundant coil activations caused by minor positional discrepancies.

10) Issue: Target Oscillation Between Adjacent Coils

*Challenge***:** Target positions generated automatically occasionally oscillated between adjacent coils, preventing bots from reaching the intended targets.

Solution: Setting a tolerance value (0.019) for target proximity ensured targets were positioned effectively, minimizing oscillations between adjacent coils and improving overall system stability.

11) Issue: Identifying Influential Coils:

Challenge: There's always a discrepancy between the coil's position and the bot's position on that coil. Due to this error, sometimes the updated influential coils are not very appropriate.

Solution: To find the influential coils, the threshold value was set between 0.005 and 0.035. A threshold of 0.005 accommodates the error caused by the discrepancy between the coil's position and the bot's position on that coil. A threshold of 0.035 identifies all the nearby coils that can attract the bots.

12) Issue: Error Accumulation Over Time

Challenge: Since errors cannot be completely eliminated, they can only be minimized. Small errors accumulate over time and cause noticeable changes in the behavior of the code.

Solution: After completing one loop, the position is captured, and further steps are taken to ensure real-time operation and auto-correction. This approach helps minimize the impact of accumulated errors.

13)Issue: Speed and Stability of the bots.

Challenge: Execution of code encountered delays due to excessive print statements, slow bot movement, and unpredictable behavior in bots lacking an autobraking mechanism.

Solution: Streamlining code execution involved eliminating unnecessary print statements. These statements, while helpful for debugging, introduced delays. By minimizing console output, overall performance significantly improved.

For bots equipped with an autobraking mechanism, the sleep time was reduced from 2 seconds to 0.5 seconds. This adjustment allowed them to respond more swiftly without compromising stability.

Bots lacking an autobraking feature exhibited unpredictable behavior. To prevent overshooting their targets, the time the coil remained switched on was shortened. However, instances of deflection and slipping were observed, necessitating further fine-tuning.

A master code efficiently initialized Arduino and sequentially called functions from different files. Minimizing function call overhead and reinitialization contributed to better overall efficiency.

Experimental Demonstrations:

The project involved orchestrating the movement of bots using a combination of various modules to accomplish specific tasks. A master code was developed to demonstrate the bots' capabilities in moving in

different directions, rotating, and executing object-grabbing maneuvers. The flowchart, depicted in Figure 2, illustrates the sequence of actions undertaken by the bots. The movement of bots is shown in fig. 3. Initially, commands are received and processed to initiate movement commands. The code facilitates the bots' ability to perform object-grabbing actions, integrating sensor feedback to ensure precise execution. Overall, the master code coordinates these actions seamlessly, showcasing the bots' versatility and functionality in performing complex maneuvers for task completion.

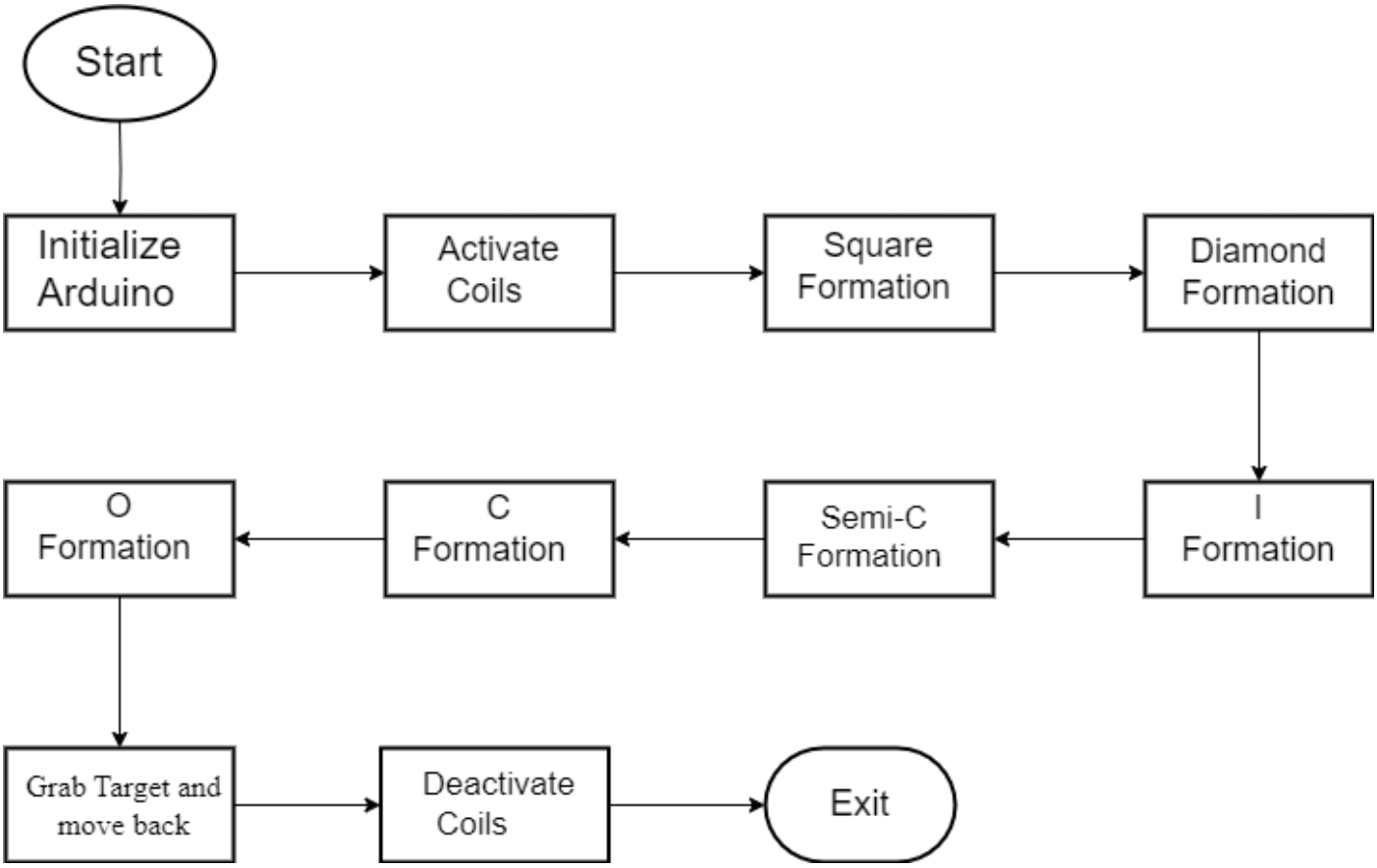

Figure 2 Flowchart of Experimental Demonstration

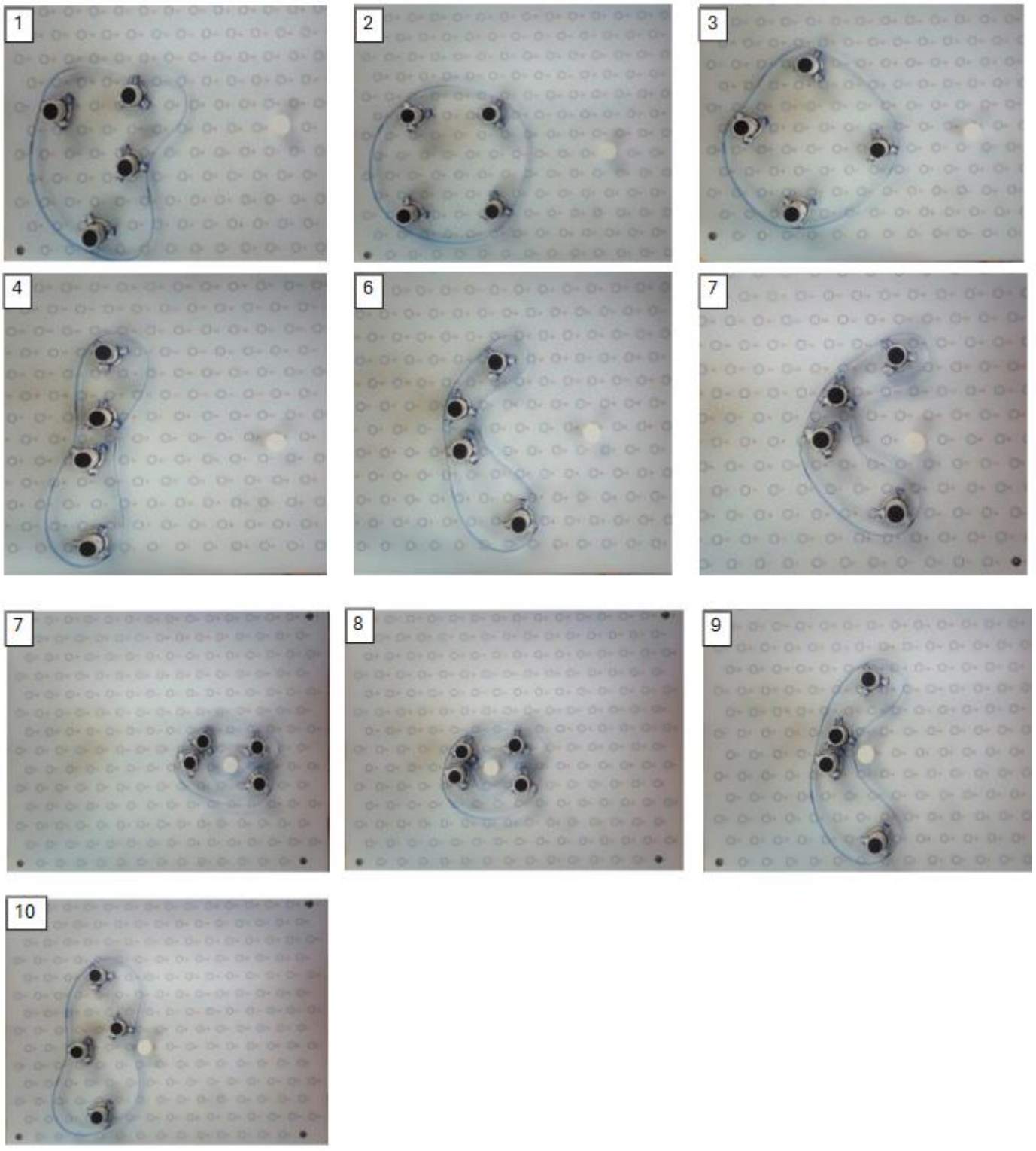

Figure 3 Experimental demonstration of bots moving and grabbing object through shape change

Additional Details

The project required conducting numerous experiments and demonstrations to thoroughly test how well the system could handle challenges and remain reliable. Some issues that popped up often were easy to spot and fix because we'd seen them before. But there were also occasional errors that were harder to figure out because they didn't happen regularly. This meant we had to spend a lot of time digging into the details to understand why they happened. We made sure that each part of the code worked correctly before moving on to the next, since everything was interconnected. This careful and time-consuming approach was crucial for building a robust system that could perform well under different circumstances.

Future work:

In future advancements, the camera system will evolve to independently identify obstacles using advanced image processing techniques. It will intelligently select and adapt its shape to efficiently interact with these obstacles within limited workspace boundaries. This will involve continuous refinement of shape-changing mechanisms and algorithms to ensure precise and adaptive maneuvering capabilities in dynamic environments.

Conclusion:

In this research, we successfully developed an autonomous control system for ferromagnetic bots using a 2D solenoid grid. By combining camera-based tracking, Arduino control, and Python algorithms, we replaced the manual activation of solenoids with an automated, vision-guided system. This improvement not only simplifies the control process but also enhances the accuracy and reliability of tasks such as shape-changing, navigation, and object manipulation within a 2D space.

We achieved several key milestones, including the accurate calibration of the camera system to track robot positions, the implementation of adaptive algorithms for solenoid activation, and the resolution of technical challenges through systematic troubleshooting. Solutions like recalibrating camera settings, optimizing solenoid activation thresholds, and refining movement algorithms were crucial in creating a robust and efficient system. This project demonstrates the importance of flexibility and persistence in developing advanced robotic systems.

The success of our project opens the door to further research and applications in areas that require precise, untethered control of small-scale bots. The methods and solutions we developed can be adapted and extended to other environments and tasks, potentially impacting various fields. Future work might focus on scaling the system for larger grids, improving the vision-based algorithms, and exploring new applications in medicine, industry, and research.

In conclusion, this research shows the potential of autonomous magnetic control systems to transform small-scale robotics, providing new capabilities for precision and coordination in complex tasks. The

insights and innovations from this project set a strong foundation for continued exploration and development in this exciting area of robotics.

References:

1. Bhat S, Ananthasuresh GK. Actuation of ferrobots in a plane for independent and swarm motion using a grid of electromagnets. In2023 International Conference on Manipulation, Automation and Robotics at Small Scales (MARSS) 2023 Oct 9 (pp. 1-6). IEEE.

Appendix:

Camera Calibration(camcalib)

1. Binary Symbol Board:

- A board featuring binary symbols is used for calibration.

2. Alignment with Solenoids:

- The board is aligned precisely with solenoids placed on top of a solenoid grid.

3. Circular Rings and Binary Symbols:

- Each solenoid is marked with a circular ring and a corresponding binary symbol inside it.

4. Image Capture:

- An overhead camera captures an image of the board, identifying the positions of the circular rings and the binary symbols within them.

5. Position Comparison:

- Controllers compare the detected solenoid positions in the image with their known grid positions.

6. Distortion Model:

- A camera distortion model, represented as a ratio of polynomials with unknown coefficients, is used to account for distortion.

7. Coefficient Estimation:

- The unknown coefficients of the distortion model are estimated by comparing the measured positions from the image with the known solenoid positions.

8. Error Correction:

- The estimated distortion model is utilized to measure the position of bots relative to the solenoids, correcting for any positional errors.

9. Model Parameter Storage:

- The camcalib process saves the estimated model parameters into a file, enabling other codes that require position measurements to access the calibrated model

Position measurement function(pos_mes_fun)

1. Initialize and Import Libraries:

- Import necessary Python libraries such as NumPy, OpenCV, and others.

2. Load Pre-calibrated Camera Parameters:

- Load camera calibration parameters (`capa`) from a saved file using pickle.

3. Main Processing Loop:

- Loop to capture frames from a video feed or image source.

4. Image Processing:

Trim and Pre-process Frame:

- Crop the frame to a specific region of interest (ROI).
- Apply contrast enhancement using CLAHE (Contrast Limited Adaptive Histogram Equalization).
- Convert the image to grayscale.

5. Object Detection:

Convert to Binary Image:

- Apply a threshold to the grayscale image to create a binary image, emphasizing object edges.

Noise Reduction:

- Apply median blur to the binary image to reduce noise and smooth edges.

Contour Detection:

- Find contours in the processed binary image using OpenCV's contour detection functions.

Filter Contours:

- Filter contours based on size and circularity metrics (variance of radius).

Identify Objects:

- For each valid contour (likely a circle), compute its centroid and check specific pixel values to determine its identity.

6. 3D Mapping:

- Use the pre-calibrated camera parameters (`capa`) to map the detected objects' centroids from image coordinates to 3D world coordinates.

7. Output Results:

- Store or output the 3D coordinates of detected objects along with their identities (if applicable).

8. Error Handling:

- Include error handling for edge cases such as no objects detected or issues with contour processing.

9. Display or Use Results:

- Optionally display processed images or results for verification or integration into a larger robotic control system.

10. Loop Continuation:

- Repeat the process for each new frame or input until the end of the video feed or image sequence.