Università
Ca'Foscari
Venezia

Tracking Visual Markers For Maglev Platforms

Alessandra Dal Bello

Relatore:
Prof. Tommaso Grigoletto

Correlatore:
Prof. Damiano Varagnolo

Dipartimento di Scienze Molecolari e Nanosistemi
Università Ca' Foscari di Venezia
Anno Accademico 2023/2024

Abstract

This Bachelor thesis is part of a collaborative project with the Norwegian University of Science and Technology (NTNU), University of Padua and Ca' Foscari University of Venice aimed at developing a magnetic levitation (maglev) platform intended for university teaching. The main objective of this thesis is to estimate the three dimensional position of a levitating magnet in a maglev platform, using visual markers and computer vision algorithms. This information can be then be used to refine pose estimation, helping to improve the control of the magnet.

This work leverages the OpenCV library, a widely recognized industry-wide resource for computer vision applications. Accurate marker selection and camera calibration are critical to ensure reliable trajectory reproduction while minimizing errors.

This thesis represents a first step in the use of computer vision for maglev control as techniques and results obtained here could extend to several areas, including integration into a mobile device application, thus allowing a wider audience to explore electromagnetic fields through video recording. Furthermore, the same methodologies can be adapted to different physical experiments in many scientific contexts.

Contents

List of Figures

Listings

Introduction

Magnetic levitation is an advanced technology that uses magnetic fields to keep objects suspended without physical contact. This technology has applications in various fields, such as transport, scientific research and robotics.

The project in which my thesis is embedded was initiated in 2021 by four students from Norwegian University Of Science and Technology, and has continued over the years under the supervision of Professor Damiano Varagnolo. The project involved numerous students, both internal and external, who contributed to the development of various parts of the project.

The main objective of the project is to make the study of electromagnetic fields accessible to anyone, thanks to the creation of an electromagnetic platform that, by implementing an advanced centralising system, is able to keep a magnet in balance, thus allowing its movements to be observed and studied.

The platform consists of eight magnets arranged in pairs in a circular form and four solenoids also positioned in a circle within the configuration of the magnets. The whole is mounted on a breadboard, which provides the power supply, and is topped by a plexiglass panel, above which the magnet to be studied is placed.

This thesis focuses on the analysis of the magnet's movements. By using ArUco markers to 'mark' the magnet, its trajectory can be recognised and traced. The data collected through video recordings were processed using a Python code in order to visualise the magnet's movements.

Tracking the magnet's trajectory enables further study of the interactions between the magnet itself and the MagLev platform.

Model Description

Introduction to the model

The reference model used in this thesis is commonly applied in the study of small suspended objects. It is a complementary model consisting of a magnet that "levitates" above a base of solenoids and fixed magnets. The following illustration represents this model:

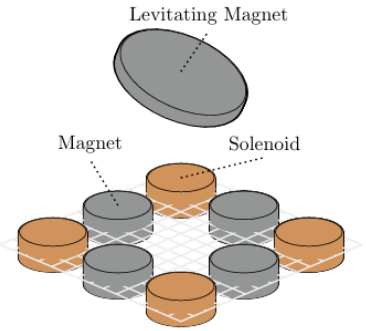

Figure 1: Illustration of the magnetic levitation system [6]

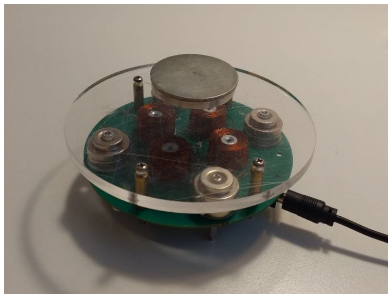

Figure 2: MagLev platform with the levitating magnet

The model shown consists of four fixed magnets, four controllable electromagnets, an on-board controller and a Hall sensor.

The Hall sensor is designed as a magnetic field detector, but in this project it is used as a position sensor. This is because in the presence of the levitating magnet, a magnetic field is detected; therefore, depending on where it is picked up, an approximation of position can be obtained. One of the problems with the model is precisely this, that an accurate estimate of the magnet's position cannot be obtained with the Hall sensor. This thesis aims to solve this problem by using ArUco markers to 'mark' the magnet in a balanced position.

This model is interpreted as a **planar magnetic motor**, widely used in low-speed,

high-precision translation devices. In such systems, the main challenge lies in manipulating and controlling the magnetic fields and forces acting on the levitating magnet.

Two models have been developed to simulate the magnetic field and control the system:

- The **first model** is based on a semi-analytical solution of the magnetic field generated by ideal solenoids [6], providing high accuracy but with a high computational cost.
- The **second model** uses a discretization approach that significantly reduces computation time by up to 98%, while maintaining good accuracy [6].

Chapter 1

Experimental Setup

This chapter describes the theoretical background necessary for understanding and carrying out the project. In particular, it deals with the pinhole camera model, image distortion, camera calibration and ArUco markers.

1.1 Pinhole Camera Model

In computer vision, a lot of methods illustrate how an optical camera functions, but this project employs the pinhole camera model. In this model, each light ray passes through the pinhole and reaches the image plane. The size of the image is determined by the distance between the image plane and the pinhole (or the camera center), which is defined as the focal length f.

This model is depicted in Figure 1.2, where the plane (u, v) represents the pixel location, F_c indicates the origin of the ray, and $P = (X_w, Y_w, Z_w)$ is the point being re-projected. In Figure 1.2, it is evident that the center of the imager chip does not coincide with the optical axis. To account for this misalignment, two parameters, c_x and c_y, must be introduced to describe the displacement of the coordinate center on the image plane relative to the optical axis. These parameters are essential for constructing an accurate projection of a point **P** in the real world, with coordinates (X_w, Y_w, Z_w), onto the image plane at a specific pixel position (u,v) [2].

$$u = f_x \frac{X_c}{Z_c} + c_x$$

$$v = f_y \frac{Y_c}{Z_c} + c_y$$

In the overlook formula, there are two new parameters, f_x and f_y, which represent two distinct focal lengths. The units of measurement used are pixels for points

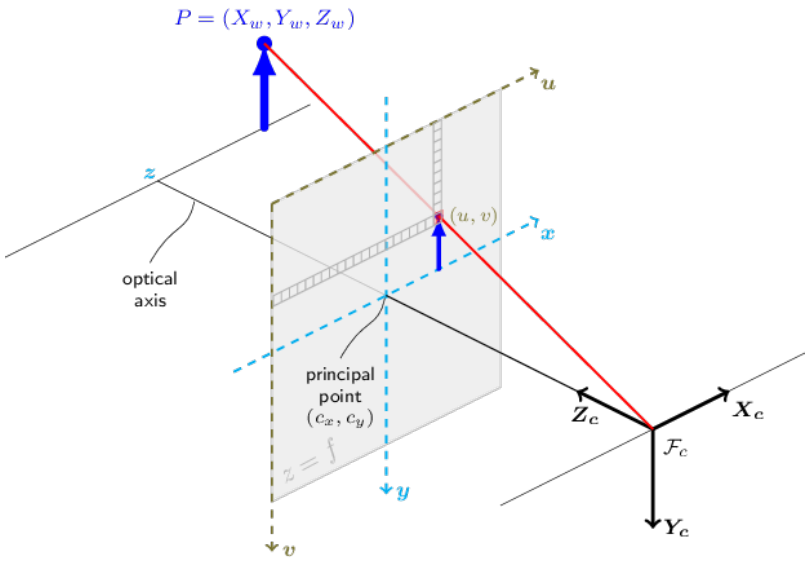

Figure 1.1: Pinhole camera model scheme [2]

in the digital image plane, while points in the image plane are represented in real-world units (e.g., centimeters). To make these two different units of measurement work simultaneously, two additional parameters, k and l, are introduced, which have units of $\frac{pixel}{cm}$ corresponding to the conversion that will be applied to the focal lengths.

The pinhole camera model projects a point from the real world, denoted as P_w, onto the image plane through a perspective transformation, resulting in the corresponding pixel p. The key difference between these two points is that P_w is a 3D point, while p is a 2D point. This creates an algebraic issue since their dimensions do not match. To address this, homogeneous vectors and transformations are used. The relationship between these points is described by the following equation:

$$s \cdot p = A[R|t]P_w$$

where s is a scaling factor, p is the point on the image plane, and A is the intrinsic matrix (I will explain the intrinsic parameters later). The matrices R and t represent the rotation and translation that define the transformation from the world coordinate system to the camera coordinate system. Specifically, A is the following matrix [2]:

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

By substituting all the relevant data into the equation above

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$$

with

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = [R|t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

1.2 Intrinsic and Extrinsic Camera Parameters

Intrinsic parameters are those variables closely related to the camera's structure, such as focal length and optical centre. Also influencing image capture is the distortion parameter that relates to lens parameters, there are two types of distortion: tangential and radial. The latter is further divided into two cases: barrel distortion (monotonically decreasing) and dot distortion (monotonically increasing).

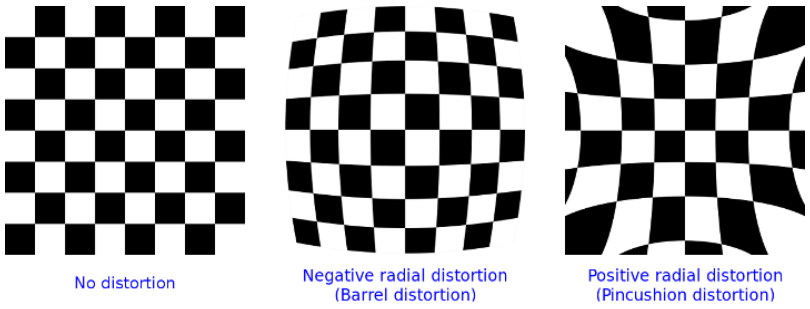

Figure 1.2: Example of distortion [2]

Distortion can be corrected using additional parameters.

Extrinsic parameters describe the position of the camera with respect to a point in space, in fact algebraically they correspond to the translation and rotation vectors that translate the coordinates of a 3D point to a given reference system, thus enabling a correct transformation of points from the real to the digital world.

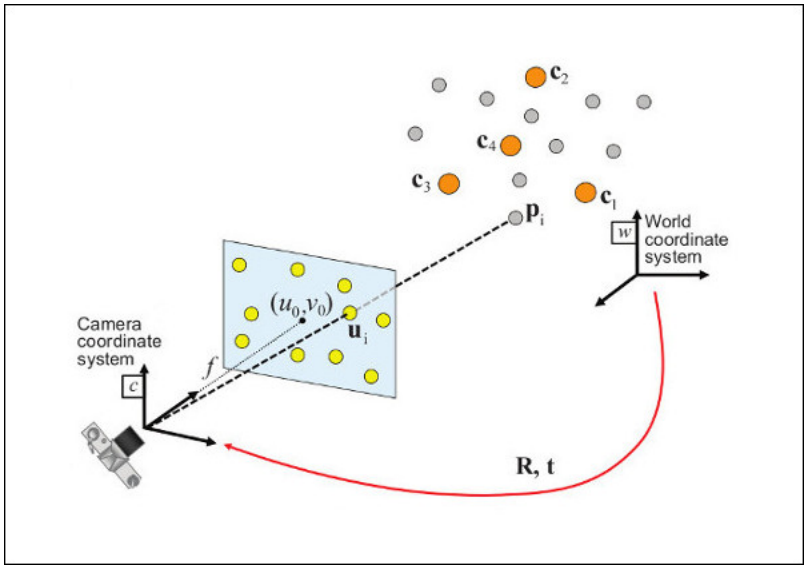

Figure 1.3: Relation between camera and real world system
[4]

1.3 Camera Calibration

The previous section described the types of parameters used to estimate the position and orientation of an object. Intrinsic and extrinsic parameters are unknown, and must be obtained through the process of camera calibration. Camera calibration techniques are very important, as they correct the distortion of the image due to the lens. Calibration techniques are based on sets of points in the real world, the relative coordinates of which are known, together with the corresponding coordinates in the image plane [7]. There are many different algorithms that analyse the collected point sets and search for their correlation, but the two used by the OpenCV library are based on Zhang's calibration methods.
After calibration, the code will output the re-projection index, i.e. a parameter indicating how accurate the calibration is; they attempt to obtain values on the scale of a few hundredths.

In robotics and computer vision, the most commonly used method for camera calibration with lens distortion is to use a planar rectangular chessboard. The chessboard consists of a regular pattern of black and white squares, making it easy for software to recognise specific points, which are already known (e.g. the edges of the chessboard). Knowing these points in the real-world coordinate system and knowing the coordinates in the image, it is possible to solve the distortion coordinate equation

using the distortion coefficients [1].

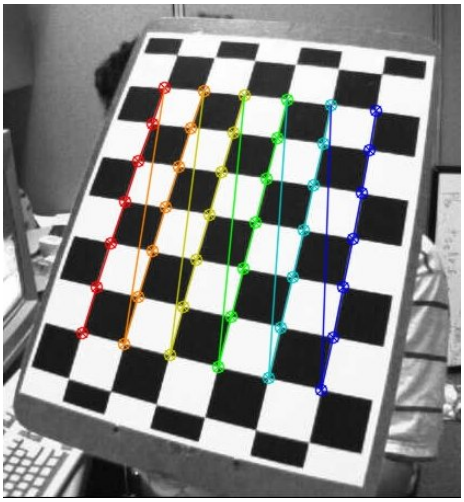

Figure 1.4: Example of calibration pattern [1]

1.4 Rotation and Translation of a Reference System

In computer vision, real-world information is taken through the analysis of videos and images captured by video cameras, smartphones or cameras. As explained in the section on the *Pinhole Camera Model*, real-world points undergo a series of algebraic processing to determine the correct correspondence in the digital plane. However, so far it has not been specified to which reference system the collected points belong and to what the coordinates that can be visualised by means of code, in this case written in Python, refer. In this thesis, ArUco markers were used to identify objects of interest, such as the moving magnet and the MagLev platform. The analysis of the collected data was carried out with the OpenCV library. The 3D camera reference system consists of the z-axis which is oriented outwards from the camera, the x-axis which is positively oriented to the right and the y-axis which is positively oriented downwards.

The object of the study was the movement of the magnet relative to the fixed marker near the platform, which required a change in the reference system. To describe the relative pose of the moving magnet, i.e. the combination of position and orientation, it is necessary to consider two components: translation and rotation. Translation represents the distance between the origins of two coordinate systems, while rotation can be described by the angles between the corresponding axes of

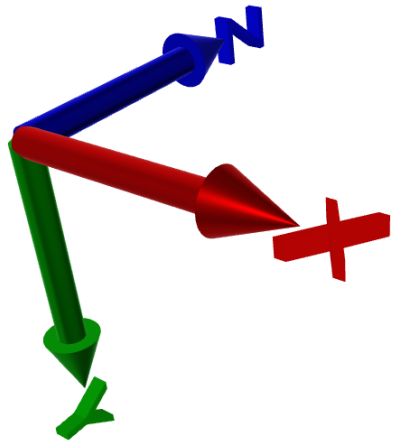

Figure 1.5: Orientation of camera axes

the two coordinate systems. Together, these two components describe the change of coordinates [8].
Rotation and translation in three dimensions are complex topics, and there are various mathematical methods to apply them. Chapter 2 will explain how these concepts have been used in this thesis, while this section provides the essential theoretical basis.
In three-dimensional space, the z-axis is orthogonal to the x- and y-axes, and its orientation is often described by the right-hand rule. The basis vectors are unit vectors parallel to the axes, such that [8]

$$\hat{\mathbf{z}} = \hat{\mathbf{x}} \times \hat{\mathbf{y}}$$
$$\hat{\mathbf{x}} = \hat{\mathbf{y}} \times \hat{\mathbf{z}}$$
$$\hat{\mathbf{y}} = \hat{\mathbf{z}} \times \hat{\mathbf{x}}$$

In the figure below, we can see how a system B (in red) was first translated by a vector t_b and then rotated by an angle. As can be seen, the two systems have different orientations, and the aim is to express the coordinates of system B with respect to the orientation of system A using a series of rotations.
Each base consists of three elements, which form the columns of an orthogonal matrix 3×3. The relationship between system A and system B is expressed by the following equation [8].

$$\begin{pmatrix} {}^{A}p_x \\ {}^{A}p_y \\ {}^{A}p_z \end{pmatrix} = {}^{A}\mathbf{R}_B \begin{pmatrix} {}^{B}p_x \\ {}^{B}p_y \\ {}^{B}p_z \end{pmatrix}$$

Rotations about the individual axes are described by matrices [8]:

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & \end{pmatrix}$$

To rotate the system, the Rodrigues matrix was used, which is directly linked to the OpenCV library and offers greater precision than the Euler angle method. The Rodrigues matrix aroud a fixed axis specified by the unit vector $\hat{\boldsymbol{\omega}} = (\omega_x, \omega_y, \omega_z) \in \mathbb{R}^3$ looks like [5]:

$$\mathbf{R}_{\hat{\omega}}(\theta) = e^{\hat{\omega}\theta} = \mathbf{I} + \hat{\boldsymbol{\omega}}\sin\theta + \hat{\boldsymbol{\omega}}^2(1-\cos\theta)$$

$$= \begin{pmatrix} \cos\theta + \omega_x^2(1-\cos\theta) & \omega_x\omega_y(1-\cos\theta) - \omega_z\sin\theta & \omega_y\sin\theta + \omega_x\omega_z(1-\cos\theta) \\ \omega_z\sin\theta + \omega_x\omega_y(1-\cos\theta) & \cos\theta + \omega_y^2(1-\cos\theta) & -\omega_x\sin\theta + \omega_y\omega_z(1-\cos\theta) \\ -\omega_y\sin\theta + \omega_x\omega_z(1-\cos\theta) & \omega_x\sin\theta + \omega_y\omega_z(1-\cos\theta) & \cos\theta + \omega_z^2(1-\cos\theta) \end{pmatrix}$$

where $\mathbf{I}$ represent the identity matrix and $\hat{\boldsymbol{\omega}}$ denotes the anti-symmetric matrix with entries [5]:

$$\hat{\boldsymbol{\omega}} = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}.$$

Note that $\hat{\boldsymbol{\omega}}\hat{\boldsymbol{\omega}} = 0$ [5].

The translation process is explained in the next chapter in the section *Data Elaboration*.

1.5 ArUco Marker

An ArUco marker is a specially designed square symbol with a thick black border and an inner grid of black and white squares, forming a binary code that gives the marker a unique identifier (ID). The black border helps the marker be quickly recognized in images, while the binary code not only identifies the marker but also allows for error detection and correction. The marker's size determines the number of squares in the inner grid. For example, a 4x4 marker has 16 squares (or bits). Markers might appear rotated in an image, so the detection system needs to figure out the correct orientation to accurately identify each corner of the marker, which is also done using the binary code. A dictionary of markers is a collection of different markers used for a particular application, essentially a list of the binary codes for each marker in that set. The key features of a dictionary are its size (the number of markers it contains) and the size of each marker (the number of bits in its grid). The ArUco module includes several predefined dictionaries with different numbers of markers and marker sizes. While it might seem logical to think that the marker ID is the decimal number obtained by converting the binary code, this isn't practical for larger markers due to the high number of bits involved. Instead, a marker's ID is simply its position within the dictionary, starting from 0. For example, the first five markers in a dictionary have IDs 0, 1, 2, 3, and 4 [3].

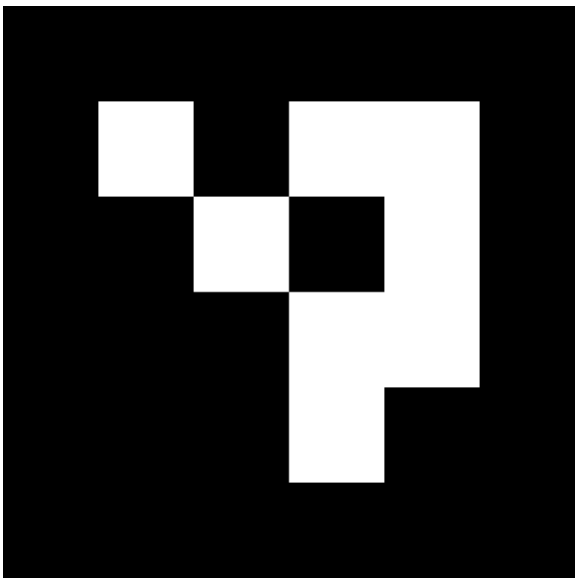

Figure 1.6: Marker ID 0, dictionary 4x4

As you move into the detection phase, it's important to understand that marker

identification involves determining the position of the four corners and the marker's ID. During the marker identification phase, the system analyzes all square shapes that could potentially be markers. This process continues until the actual marker is identified, using thresholds that help filter out incorrect shapes. Once a potential marker is identified, the system moves to the verification phase, where it examines the binary code within the marker. The image is then standardized and thresholded using Otsu's method to differentiate between black and white bits. The image is divided into cells according to the marker's size and border, and the number of black or white pixels in each cell is counted to determine whether it represents a black or white bit. Finally, the bits are analyzed to verify if the marker is part of the specified dictionary, with error correction techniques applied if necessary [3]. After detecting the markers, the next step is to estimate the camera's pose relative to the markers. This phase, known as position estimation, requires that the camera is properly calibrated. Calibration is crucial because the marker's position will be determined within the camera's reference system. To achieve accurate pose estimation, you need to know your camera's calibration parameters, including the camera matrix and distortion coefficients. These parameters are essential for correcting any optical distortions and for accurately mapping the marker's position [3].

1.5.1 ArUco Marker Dictionaries

Previously, the structure of the ArUco markers was explained and briefly how they are grouped. This subsection, on the other hand, focuses on the explanation of dictionaries and marker generation.

Dictionaries are groups of markers used for specific applications, classified according to the number of bits they contain. It is important to emphasize that each dictionary is unique and distinct from the others [3].

Dictionaries provide two fundamental pieces of information:

- **The size of the dictionary**, which indicates the number of markers it contains [3].
- **The size of the marker**, which represents the number of bits it contains [3].

The above two pieces of information are very important for the correct use of ArUco markers, the symbolism used has the following meaning: `DICT_4X4_50`. Here, `DICT` indicates that it is a dictionary, `4x4` represents the size of the marker (16 bits), and `50` indicates that the dictionary contains 50 distinct markers.

There are different types of dictionaries, such as `DICT_5x5_100` or `DICT_6x6_50`. In general, marker sizes range from 16 bits (4x4) to 49 bits (7x7), and dictionaries can contain 50, 100, 250, or 1000 markers.

The choice of the dictionary to be used is not random, but must be adapted to the specific requirements of the project. The size of the dictionary depends on the number of markers required; for example, in this thesis, two are needed: one for the magnet and one for the MagLev platform, making a 50-marker dictionary sufficient.

The size of the marker, on the other hand, varies depending on several factors: the amount of information to be collected, the resolution of the video, the distance between the marker and the camera, and the size at which the marker is printed. In general, if the video is shot close up and not much information is needed, 16 bits are sufficient. However, if the footage is shot from a greater distance or in low light, and more detailed information is to be gathered, it is preferable to use 25-bit markers, as is the case in this thesis.

In addition to the correct selection of the 'digital' size of the marker, it is also essential to carefully choose its 'physical' size, i.e. the size it will take once printed. As will be explained in the section on pose estimation and marker recognition, if the marker is too small it may not be identified correctly. Another important precaution at this stage is to leave a large white border around the marker; otherwise, even in this scenario, the marker may not be detected properly.

Chapter 2

Data Collection and Manipulation

2.1 Camera Calibration: Correcting Distortion Errors

The previous chapter introduced the concept of image distortion and a brief overview of camera calibration techniques. This section focuses instead on using the OpenCV library, in Python, to calibrate the camera and minimise distortion errors in the context of the project. As already pointed out, this is a crucial step in image and video processing on the computer; not doing it correctly means working with inaccurate and therefore unreliable data.

As it is written in the OpenCV documentation,calibration must be performed on a set of 10-20 images of a chessboard. It is important that the images are all taken at the same distance and under homogeneous lighting conditions to ensure a more accurate calibration.

Before calibration, the code must recognise the chessboard and its edges, which is handled by the following lines of code:

```
# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*9, 3), np.float32)
objp[:, :2] = np.mgrid[0:9, 0:6].T.reshape(-1, 2)# creates a grid of 3D points

# Arrays to store object points and image points from all the images.
objpoints = []  # 3d point in real world space
imgpoints = []  # 2d points in image plane

images = glob.glob(r'C:\Users\ale10\OneDrive - unive.it\Desktop\images7\photo_2024-*
    ')

# Variable to maintain image size
img_shape = None

for fname in images:
```

```
    img = cv.imread(fname)
    if img is None:
        print(f"Error loading image: {fname}")
        continue

    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Finding the corners of the chessboard
    ret, corners = cv.findChessboardCorners(gray, (9, 6),None)

    # If found, add object points and image points
    if ret==True:
        objpoints.append(objp)

        corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)

        imgpoints.append(corners2)

        # Draw and display angles
        cv.drawChessboardCorners(img, (9, 6), corners2, ret)
        cv.imshow('img', img)
        cv.waitKey(500)

        # Maintain image size
        img_shape = gray.shape[::-1]
    else:
        print(f"Chessboard corners not found in: {fname}")

cv.destroyAllWindows()
```

Source Code 2.1: Chessboard recognition.

This operation collects the object and image points required for the actual calibration of the camera.

```
if img_shape is not None:

    # Calibrate the camera using all points collected
    ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints,
        img_shape, None, None)
```

Source Code 2.2: Camera Calibration.

The results obtained include the camera matrix, distortion coefficients, rotation and translation parameters, etc. [1]. To estimate the quality of the calibration, the re-projection error is calculated:

```
    mean_error = 0
    for i in range(len(objpoints)):
        imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, dist
            )
        error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2) / len(imgpoints2)
        mean_error += error

    print("Total error: {}".format(mean_error / len(objpoints)))
else:
    print("No valid image found for calibration.")
```

Source Code 2.3: Calculation of re-projection error.

With this data, distortion can be removed from images. The documentation proposes two methods to do this; the following was used in this thesis:

```
# Process each image again to distort it and save it
for idx, fname in enumerate(images):

    img = cv.imread(fname)

    cv.imshow('image',img)
    cv.waitKey(500)
    cv.destroyAllWindows()

    h, w = img.shape[:2]

    # Calculate new optimised camera matrix
    newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w, h),1, (w, h))

    # Correct distortion
    dst = cv.undistort(img, mtx, dist, None, newcameramtx)

    # Crop the image using the ROI
    x, y, w, h = roi
    dst = dst[y:y+h, x:x+w]

    cv.imshow('image',dst)
    cv.waitKey(500)
    cv.destroyAllWindows()


    # Save the undistorted image
    output_filename = os.path.join(output_dir, f'calibresult_{idx}.png')
    cv.imwrite(output_filename, dst)
```

Source Code 2.4: Distortion error correction.

All data are saved in a 'npz' file. In summary, the results obtained at this stage are:

Figure 2.1: Before distortion removing

Figure 2.2: After distortion removing

Figure 2.2 has not been cropped to show how the image is 'folded' during correction.

The calibration matrix calculated from the calibration parameters is :

```
[[897.97047864   0.         623.81677773]
 [  0.         894.94753804 456.85205433]
 [  0.           0.           1.        ]]
```

The distortion coefficients are:

```
[[ 1.11431662e-01 -9.60364353e-01  6.96994697e-03 -2.37430981e-03
   3.04437072e+00]]
```

2.2 ArUco Marker Identification: Computer Vision Application

To achieve accurate trajectory tracking, it is essential that the algorithm correctly recognises the markers in the video. Incorrect identification can result in the 'digital' trajectory not corresponding to the path actually taken, due to a number of issues that will be discussed in this section. The code for the identification of ArUco Markers, as well as for the calibration of the camera, is provided by the official OpenCV documentation [1].
The Python code used is as follows:

```python
import cv2 as cv
import numpy as np
from cv2 import aruco
import matplotlib.pyplot as plt

dvecs = []

# Import data calibration
calib_data_path = r'C:\Users\ale10\OneDrive - unive.it\Desktop\images7\calibrated\
    calibration_data.npz'
calib_data = np.load(calib_data_path)

cam_mat = calib_data["camMatrix"]
dist_coef = calib_data["distCoef"]

# Parameters for ArUco Marker
dictionary = aruco.getPredefinedDictionary(aruco.DICT_5X5_50)
parameters = cv.aruco.DetectorParameters_create()

# Size of the marker (centimeter)
markerSizeInCM = 5.0

# Defines the 3D points of the marker in the object's co-ordinate system
marker_corners_3d = np.array([
    [-markerSizeInCM/2, markerSizeInCM/2, 0],
    [markerSizeInCM/2, markerSizeInCM/2, 0],
    [markerSizeInCM/2, -markerSizeInCM/2, 0],
    [-markerSizeInCM/2, -markerSizeInCM/2, 0]
], dtype=np.float32)
```

```
29
30 # Open the video file
31 cap = cv.VideoCapture(r'C:\Users\ale10\OneDrive - unive.it\Desktop\images\magnete29.
       mp4')
32
33 while True:
34     ret, frame = cap.read()
35     if not ret:
36         break
37
38     # Correct frame distortion
39     frame_undistorted = cv.undistort(frame, cam_mat, dist_coef)
40     frame_undistorted = frame
41
42     # Detects ArUco markers in the frame
43     corners, ids, rejected = aruco.detectMarkers(frame_undistorted, dictionary,
           parameters=parameters)
44
45     # If there are markers detected, estimate their pose
46     if ids is not None:
47         rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(corners, markerSizeInCM,
               cam_mat, dist_coef)
48
49         # Draw markers and their axes
50         frame_markers = aruco.drawDetectedMarkers(frame_undistorted.copy(), corners,
                ids)
51         for i in range(len(ids)):
52             rvec, tvec = rvecs[i][0], tvecs[i][0]
53
54             # Use solvePnP to get a more accurate pose
55             success, rvec, tvec = cv.solvePnP(marker_corners_3d, corners[i], cam_mat
                   , dist_coef, rvec, tvec, useExtrinsicGuess=True,flags=cv.
                   SOLVEPNP_AP3P)
56
57             # Draw the axes of the marker
58             cv.drawFrameAxes(frame_markers, cam_mat, dist_coef, rvec, tvec, 10)
```

Source Code 2.5: Markers identification.

The two main problems encountered at this stage are:

- **Intermittent marker recognition;**
- **Flipping of the z-axis in some situations;**

Non-constant marker recognition can be caused by several combinations of problems. Firstly, as explained in the previous sections, it is essential to use the correct dictionary. For example, the use of a `DICT_4X4`, which encodes a limited amount of information, combined with filming from a long distance, leads to poor and inaccurate data collection, causing incorrect marker recognition. In addition, camera resolution and lighting conditions affect identification: low light, as well as excessive light, reduces the visibility of the marker. Filming must be done from the right angle; standing perpendicular to the marker is not recommended. It is also essential that

the marker is always clearly visible: if it is obscured, it will not be recognised. Finally, the print size of the marker and the white border around it are very important; too small a size or the absence of a white border can cause numerous recognition problems.

The problem of the z-axis flipping is very common among ArUco markers and can lead to coordinates being collected in the wrong half-plane, causing confusion in data collection (as the marker does not remain flipped constantly, but rotates randomly). Currently, there is no definitive solution; however, many users opt for algebraic matrix corrections, reversing the z-axis with the y-axis. This solution, however, only works in certain specific cases, which are difficult to identify. In this project, the problem was handled by printing large markers (5 cm), with 1 cm of white border, and taking close-up shots with good lighting.

These precautions, although they may seem trivial, have led to a significant increase in tracking accuracy, as identification occurs correctly and the z-axis is not affected.

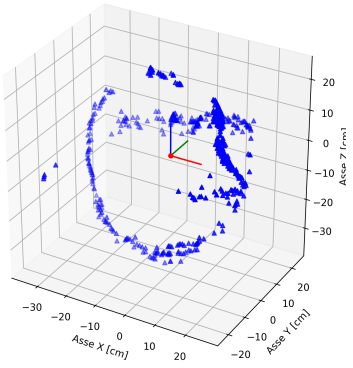

Figure 2.3: Incorrect marker identification

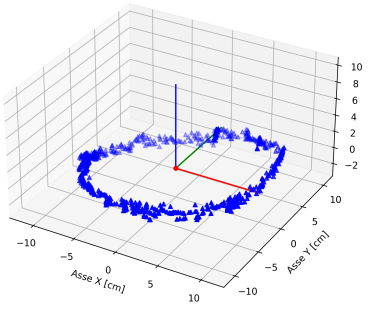

Figure 2.4: Correct marker identification

The images above illustrate what has been described above. Both trajectories are the result of tracing a circular motion, in which a fixed marker is placed in the center of the circle, while another marker is run around the perimeter of the circle.

In Figure 2.3, a `DICT_5X5_50` marker with a 2 cm side and a white border of a few millimeters was used. The shots were not taken accurately; at one point, in order for the marker to follow the circumference, the central marker was obscured, causing the reference system to be lost for a few frames. This introduced complications to the algorithm during the transformation from the camera's reference system to that of the fixed marker, also causing the z-axis to continuously flip.

In Figure 2.4, however, the same type of `DICT_5X5_50` marker was used, but with

a 5 cm side, more than double the size of the previous one, and a 1 cm white border. Video footage was taken more carefully, avoiding obscuring the central marker during movement. This made it possible to obtain more stable data (no z-axis flips), thus being more reliable and faithful to the initial track. As can be seen, the quality of the result is significantly higher in the second case.

2.3 Data Elaboration

The previous section explained how to identify one or more markers in a video or image. In script 2.5, on line 47, the `estimatePoseSingleMarkers` method from the ArUco library is used, which returns two vectors: **`rvecs`** and **`tvecs`**.

`rvecs` is a 3D rotation vector that defines the orientation of the marker, indicating the angles of rotation with respect to the axes. Although not done in this project, this vector can be converted into a rotation matrix using Rodrigues' formula.

`tvecs`, on the other hand, represents the translation vector indicating the position of the marker in the coordinates x, y and z. The unit of measurement in which it is expressed is the same as that used in defining the size of the marker (in this case centimetres, but pixels can also be used).

```
    # If there are markers detected, estimate their pose
    if ids is not None:
        rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(corners, markerSizeInCM,
            cam_mat, dist_coef)

        # Draw markers and their axes
        frame_markers = aruco.drawDetectedMarkers(frame_undistorted.copy(), corners,
             ids)
        for i in range(len(ids)):
            rvec, tvec = rvecs[i][0], tvecs[i][0]

            # Use solvePnP to get a more accurate pose
            success, rvec, tvec = cv.solvePnP(marker_corners_3d, corners[i], cam_mat
                , dist_coef, rvec, tvec, useExtrinsicGuess=True,flags=cv.
                SOLVEPNP_AP3P)

            # Draw the axes of the marker
            cv.drawFrameAxes(frame_markers, cam_mat, dist_coef, rvec, tvec, 10)

            # Based on the identified ID I append the detected position to the
                associated marker list
            # in this case it could have been =1 or =2 because I printed two markers
                 with that ID
            if ids[i] == 1:
                mvec = tvec

            elif ids[i] == 2:
                fvec = tvec
```

```
23                  R, _ = cv.Rodrigues(rvec)
24
25      else:
26          frame_markers = frame_undistorted.copy()
27
28      dvec = mvec-fvec
29      distance = np.linalg.norm(dvec)
30      text_position = (10, 30)
31      cv.putText(frame_markers, f" Relative distance: {distance:.2f} cm",
            text_position, cv.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
```

Source Code 2.6: Drawing of axes.

In the previous lines of the code (to the lines 18 and 21), movement and translation vectors are assigned based on the recognition of the marker ID. This process takes advantage of the fact that the marker ID is already known, since it has been generated and printed previously. Thanks to this, it is possible to use the ID recognised by the code to perform a conditional check with the `if` instruction. During the analysis, each frame is processed and the data for the two markers are separated. Next, the movement of the moving marker with respect to the reference system of the fixed marker is expressed. The procedure is illustrated in the diagram below:

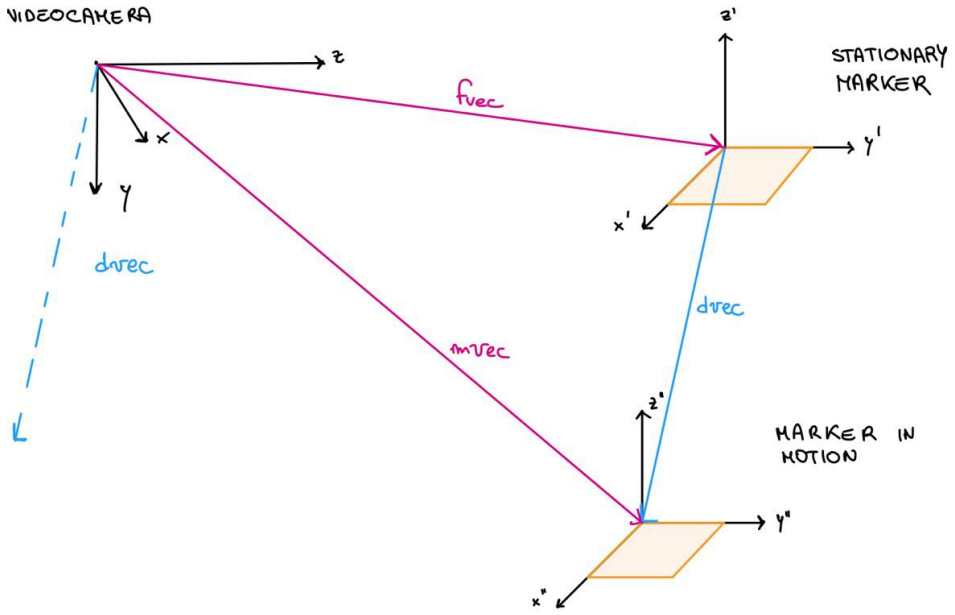

Figure 2.5: Scheme of the reference system

The image shows a spatial diagram in which the camera observes two markers: one fixed and one moving (placed on top of a magnet). The camera has its own

reference system, while initially the two markers are defined with respect to the camera's reference system.

In detail, the video camera is positioned at the top left and has a Cartesian coordinate system with axes x, y, and z. Two observation vectors point from the camera towards the fixed marker (**fvec**) and the moving marker (**mvec**) respectively. The stationary marker has its own coordinate system, labelled as x', y', z', while the moving marker has its own system x'', y'', z''.

Initially, the positions of the markers are measured with respect to the reference system of the camera. However, the aim is to analyse the movement of the magnet with respect to the platform. Since placing the fixed marker directly on the platform would have caused an obstruction by the moving marker, it was decided to place the fixed marker at the foot of the platform, moving it slightly towards the observer. It is therefore necessary to perform a change of co-ordinates to express the position of the magnet in relation to the fixed marker.

As shown in Figure 2.5, the vector **dvec** (in blue) represents the distance between the two markers and is calculated as:

$$\texttt{dvec} = \texttt{mvec} - \texttt{fvec}$$

To express this distance in the fixed marker reference system, the rotation matrix calculated in the Source Code 2.6 at line 23 is used:

$$\texttt{d'vec} = R \cdot \texttt{dvec}$$

The vector `d'vec` is now expressed in the fixed marker reference system, labelled x', y', z'.

Chapter 3

Results

The images below show two frames of the videos taken of the magnet in motion. The frame in Figure 3.1 belongs to the original video before computer processing. Figure 3.2 is from the post-processing of the original video, in which the marker recognition (green border around the black square with the respective ID) and the Cartesian axes x (in red), y (in green) and z (in blue) can be seen. In addition, the relative distance between the moving magnet and the stationary marker can be read at the top left.

The three-dimensional trajectory of the displacements performed by the magnet

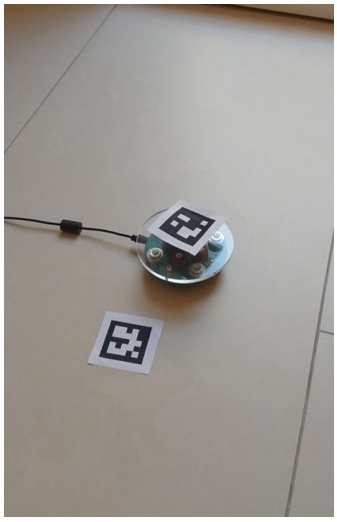

Figure 3.1: Frame from the original video

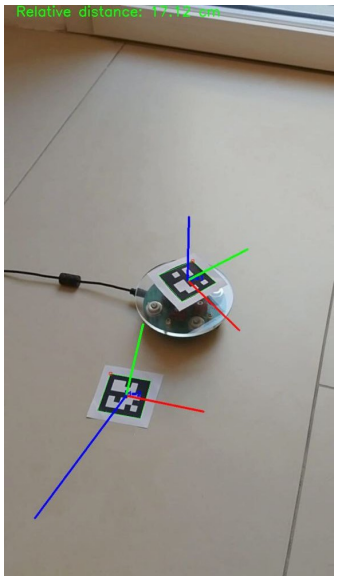

Figure 3.2: Frame post processing

can be seen from the following plot:

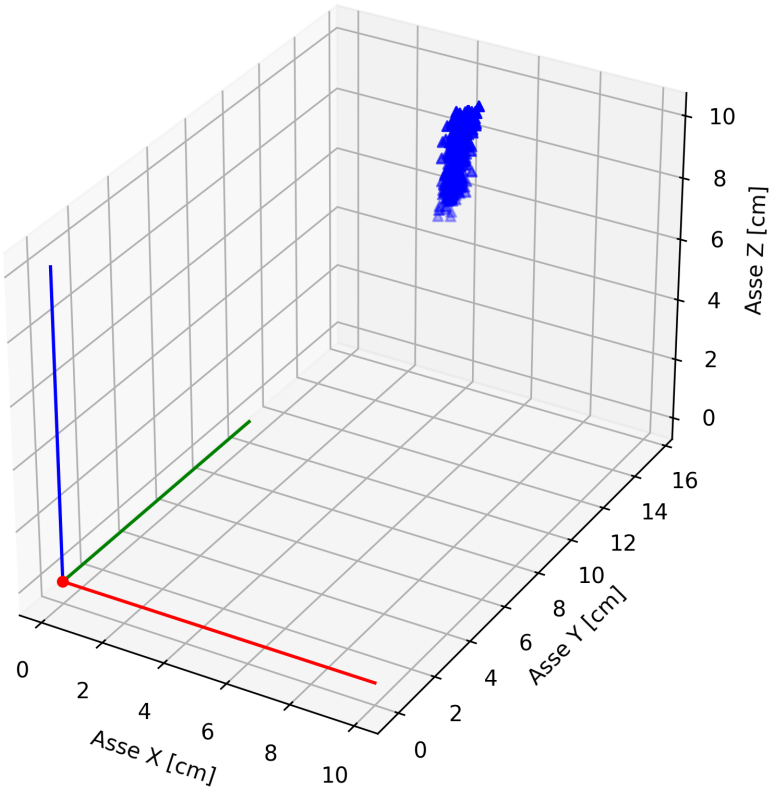

Figure 3.3: Three-dimensional trajectory of the magnet

As can been seen in Figure 3.3, the trajectory is not clearly legible. This is because the magnet makes very small variations, and although they are recognised by the software, the coordinates of the positions are somewhat confused. The problem could be related to the type of marker and the presence of noise and outliers.

Conclusion

The aim of this thesis as to be able to trace the three-dimensional trajectory of the magnet's movements. To achieve this, the magnet was marked with ArUco markers and the videos were taken by a mobile phone.

In the use of this type of marker, critical points were found in the accuracy of the pose estimation. The analysis showed that for every small trajectories the displacement variations are not detected correctly but inaccurately and noisily, while for large movements this criticality did not arise (can be seen very well by comparing Figure 2.4 with Figure 3.3).

In the light of this problem, it would be interesting in the future to try to replicate the project but with a more robust type of marker or with camera devices with a higher resolution.

To overcome this problem, one possible solution is to use more reference markers (defined in the text as 'stationary' markers) to try to decrease the estimation error by averaging the movements detected.

To improve the quality of the data collected, a filtering operation (e.g. with Kallman filters) could also be carried out to eliminate noise and outliers, thus making the trajectory cleaner and more readable.

This project provides a solid base from which future students can build. One aspect that I find it might be interesting to analyse the estimation of the magnet's rotation, as it could provide important information for the dynamic modelling of the magnet.

Bibliography

[1] Camera calibration. https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html.

[2] Camera calibration and 3d reconstruction. https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html.

[3] Markers and dictionaries. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html.

[4] Perspective-n-point. https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html.

[5] Rodirgues' rotation formula. https://mathworld.wolfram.com/RodriguesRotationFormula.html.

[6] Hans ALvar Engmark and Kiet Tuan Hoang. Modeling and control of a magnetic letivation platform. *IFAC PapersOnLine*, 56(2):7276–7281, 2023.

[7] P.Corke. *Robotics, Vision and Control: Fundamental Algorithms in Python (3^a ed.)*. Springer, 2023, Ch.13.

[8] P.Corke. *Robotics, Vision and Control: Fundamental Algorithms in Python (3^a ed.)*. Springer, 2023, Ch.2.

Acknowledgements

I dedicate this achievement to my parents, my brother, my grandmother and you Davide who are my number one fan.

I also dedicate this achievement to you Paola who is no longer with us.

I dedicate this achievement to all my friends who have supported me over these three years and who have managed to make this difficult journey a little easier. To my university friends.

A thank you also to me.