Instructions to setting up the

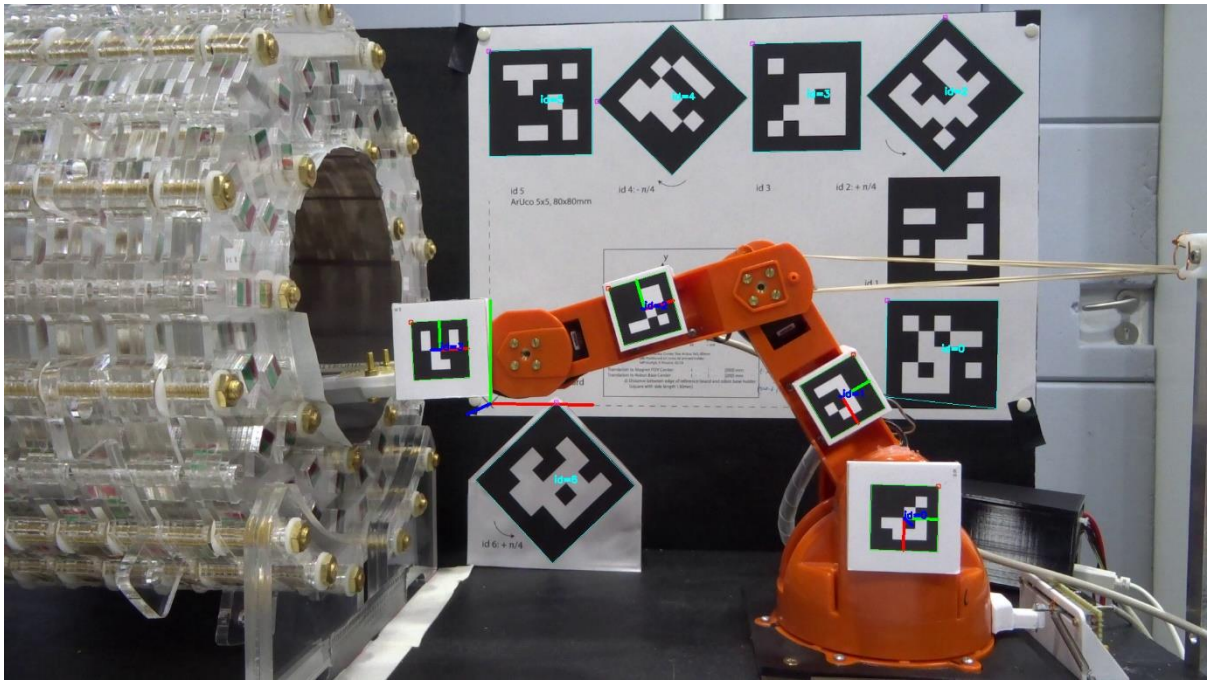
Motion Tracking

Project

Easy Scalable, Low-Cost Open Source Magnetic Field Detection System for Evaluating Low-Field MRI Magnets using a Motion Tracked Robot

Pavel Povolni

October 2024



Content

1 Preamble

These instructions describe the Motion Tracking of the robot.
Further information can be found in the publication, as well as in the Appendix and Supplementary Information referenced there.
Software used:

- Motion Tracking: Python 3.9 and openCV 4.5.5
- Mechanical design (CAD): Autodesk Inventor 2022 (Commercial). All parts exported as .Steps
- Video Cutting Program: Vegas Pro 14 (any other free tool can also be used, e.g. OpenShot)

If you have any further questions, please contact us!

2 Hardware

The ArUco markers, which are then automatically detected by the software, are essential for Motion Tracking.

2.1 ArUco Marker

You can find the ArUco markers used at

...\4_MotionTracking\41_Hardware\411_ArUcoCodes

In principle, there is nothing to stop you from generating new ArUco markers or installing additional markers (e.g. to increase accuracy). You can generate ArUco markers using the openCV library in Python. For us it was more practical to generate the markers using this tool: https://chev.me/arucogen/
and to arrange the exported svg files that can be downloaded there in a image-program (e.g. Inkscape).

2.1.1 Reference Board

The reference board consists of 2 parts (the 2 PDF files in the folder ...\Reference Board).
The board "Reference_Board_Final.pdf" (part1) is printed out on normal DinA3-paper. Make sure that the file is not scaled. This document is glued to the back of the Reference System (see 2_Robot_manual/Chapter 2.4).
The paper is aligned with the edges of the Reference System (see Figure 1).

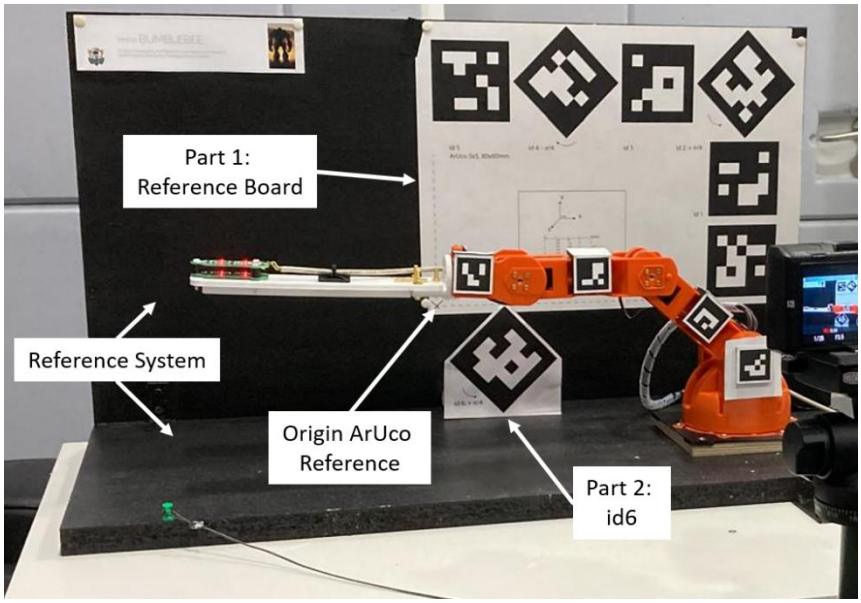


Figure 1: Setup of the environment + mounting of the reference board

The second part is the marker id6 ("Reference_Board_Final_id6.pdf"). A printout on DinA4 is sufficient here and you cut it out at the markings. This marker is mounted on an additional holder so that the Motion Tracking is more precise thanks to this additional depth information.
This holder consists of 2 parts (see Figure 2), both of which are 3D printed. Part 2 is firmly screwed into the Reference System, where it is touching the rear part of the Reference System. The horizontal position is determined using the auxiliary part. The support is aligned with the edge of the DinA3 sheet. This precise positioning is very important because this positioning is stored in the Motion Tracking evaluation.

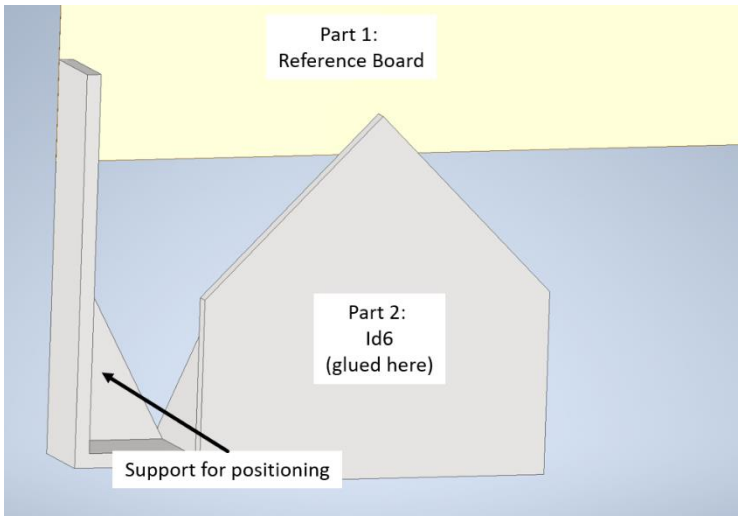


Figure 2Mounting id6

2.1.2 Robot Marker

The Robot Markers are arranged on a sheet of A4 paper. After printing (normal paper), the markers are cut out at the markings and attached to the additional holders on the robot (see Ch 2.2, 2_Robot_manual.pdf).
Our holders are 3D printed from PETG. We have had good experiences with the glue "UHU stic". Any other similar glue should work the same way.

2.2 Checkered Board for Calibration

You can find more details on the calibration itself in Ch 3.
A known chessboard pattern is required for calibration. The checkerboard pattern (9x13 boxes, 30x30mm) is available in

...\4_MotionTracking\41_Hardware\412_CheckeredBoard

Print out the board on a DinA3 sheet (no scaling) and glue it on a solid surface (e.g. on a wooden board or a plastic plate) so that it cannot bend.

2.3 Assembly of the reference board / distances from one another

see details in chapter 2.4 in Robot_manual.pdf

2.4 Setup Camera

We used a Sony HX60 as our camera (aperture f3.5). The optical & digital zoom were set to minimum, so the optical distortions are minimal. Any other comparable digital camera can be used in exactly the same way. It doesn't have to be an expensive one. The main thing is that it runs reliably during the measurement.
We carried out the Motion Tracking in post processing. The entire video was therefore saved on the SD card and only processed afterwards. In principle, Motion Tracking can also be reprogrammed so that the camera is read out live via USB. A camera that supports this is therefore recommended, but can be really expensive if it's an "industrial one".
An interface between OpenCV and the project libgphoto2 (through the API "VideoCaptureAPI" using CAP_GPHOTO2) should to be possible, whereby libgphoto2 supports a variety of cameras using the PTP or MTP protocol: http://www.gphoto.org/proj/libgphoto2/support.php
(This interface was not tested, we used the webcam Logitech C920s for experiments with live connection or post processing in the presented experiment).

The camera was positioned in such a way that all markers covered as much of the image plane as possible (see Figure 3). The camera axis should be approximately in the middle (blue line) of the setup so that the inevitable distortions (caused by the imperfect camera optics) are evenly distributed or minimized.

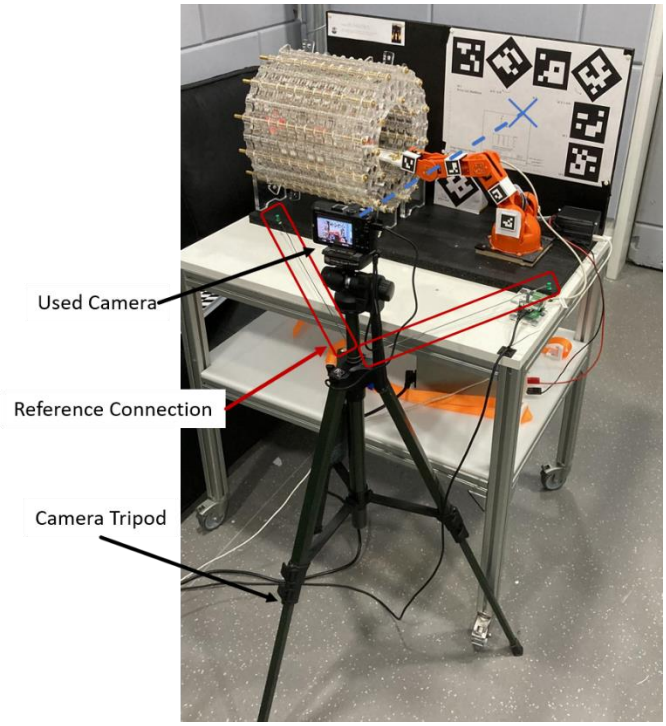


Figure 3: Camera Setup

The camera should not be moved after calibration (see next chapter). To ensure this, we have connected the tripod to the Reference System using cords. This allows us to ensure that the camera is reliably stable during the measurement.
However, this connection was not stable enough for experiments on different days (the laboratory is used by several groups). The calibration process is automated as much as possible so that the distortion factors can be calculated quickly after a new calibration the next day.
In a follow-up development, it would make sense to design a camera holder that is permanently connected to the Reference System.

The camera was operated in FullHD mode at 25fps. Higher fps are not necessary (as the measurements are static). A higher resolution would be good for higher accuracy, but also means higher data volumes and therefore longer post processing.
Further information on calibration are given in the following chapter.

3 Calibration

Motion Tracking uses a camera that is inevitably not perfect. The camera optics therefore distort the image (e.g. fisheye effect etc.). If this distortion is known, the recorded image can be corrected, which significantly improves the accuracy of Motion Tracking itself.
You can find good instructions and an explanation of the calculation in the OpenCV tutorial:
https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

The basic calibration procedure is as follows: The camera is fixed, a video with the checkered board is recorded, which is then processed using Python scripts (using openCV) and the distortion factors are calculated.

3.1 Video Recording

We have removed the magnet from the Reference System in order to fill as large an area of the camera as possible with the Checkered Board.
For calibration, the Checkered Board is swiveled and moved in front of the camera (see Figure 4). Make sure that all boxes are in the picture at all times (and nothing is cut off) and that they are not covered by the robot.

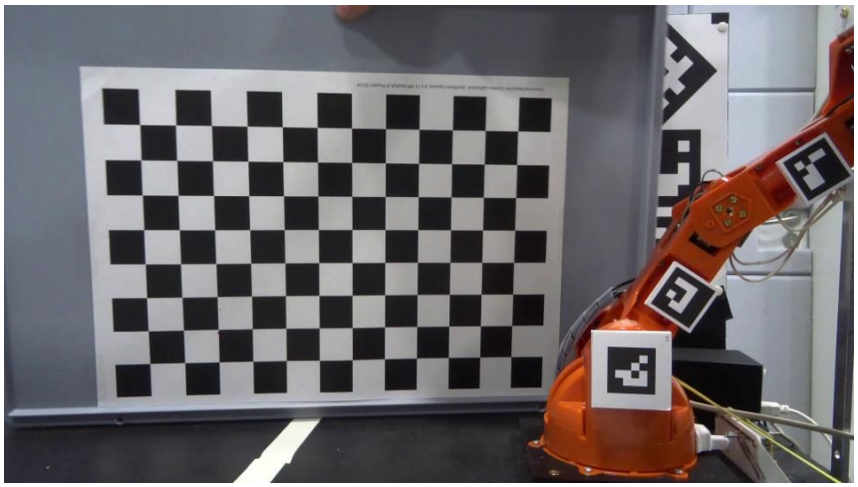


Figure 4: Checkered Board for Calibration

A video length of approx. 30 seconds has proven to be good.
Our camera saves the video file in a Sony format. But the Python script expects an .mp4 format. Therefore, the camera file must be converted to mp4, e.g. using the Video Cut program. An example video is available at:

...\4_MotionTracking\42_Software\0_calibration\calibration_video.mp4

3.2 Calibration in Post Processing

The entire Motion Tracking is based on an evaluation of the video files using Python scripts and the free openCV library. It is extremely important that the correct software versions are installed,

otherwise the scripts will not work. A Conda environment is recommended. The most important versions are openCV 4.5.5 and Python 3.9.18. Details can be found in the following folder (including the YML file for installing the correct versions):

...\4_MotionTracking\42_Software\Environment

The calibration software can be found here:

... \4_MotionTracking\42_Software\0_calibration

3.2.1 Check the structure

It makes sense to check the position of the camera shortly before calibration and mapping to ensure that all markers can be detected. It could be, for example, that the lighting in the lab is not sufficient for reliable detection (this was a problem for us), the position of the camera is poor, or the camera itself is not working reliably (aperture, focus, etc.).

Proceed as follows for this check:

- Everything is set up the way you want it for mapping
- Start the camera and record a short video of the Robot Marker and Reference Marker (10-15s is enough). Take care, that all Robot Makers are clearly visible and not covered by something.
- Convert the video to .mp4 format if the camera does not save it that way
- Start the script "3_Main_Help_Detect_RobotMarker_VideoFile.py". Before that, change the variable "video_file" to the location of your video
- The script now analyses the video frame by frame, tries to detect ArUco markers and displays all detected markers in an additional window (example see Figure 5).
- It is important that all Robot Markers are recognized. If only a single Robot Marker is missing, the mapping will no longer work. The majority of the Reference Board Markers should be recognized to increase accuracy. In the example frame, id0 is not recognized because the robot's rubber band blocks the view. This is fine, as the other 6 markers are sufficient to create the reference coordinate system. In our tests, up to 2 missing markers were still ok.

Figure 5: An exemplary frame. All Robot Markers are recognized, as are most of the Reference Markers. A detected ArUco marker is labeled with a coordinate system, frame and id number.

3.2.2 Preparation

Images are used in calibration. The recorded video is therefore split into individual photos and saved.

- *Data handling*

  The calibration video is saved in the following folder with the name "calibration_video.mp4":

  ...\4_MotionTracking\42_Software\0_calibration

The extracted photos are saved in the following folder:

...\4_MotionTracking\42_Software\0_calibration\calibration_result\img_calibration

The splitting script deletes the entire contents of this folder and then saves the new images in this folder

- *Split images*

The script "0_images_extraction.py" loads the video and extracts the photos. You don't have to change anything in the script. Just start it.

Only the "num_img" parameter could be changed if necessary. This is used to set the number of exported images (in equidistant time points). The value "300" was quite good (trade off between duration of calibration and accuracy of calculation).

3.2.3 Calibration

The script "1_camera_calibration.py" evaluates the photos and calculates the calibration factors. The software is commented in case something needs to be changed. But actually you don't need any adjustments (if you use the same Checkered Board).

Simply start the software and wait until all images have been processed and the calibration factors have been calculated.
The calculated factors are saved in the "cam_dist.csv" and "cam_mtx.csv" files. In addition, the so-called "reprojection error" is calculated, which is a measure of the accuracy of the camera-based Motion Tracking. This error is only displayed in the console (and not saved). It is worth making a note of this error to evaluate your Motion Tracking.

4 Motion Tracking during Mapping

Motion Tracking is part of the mapping of the magnet. Details on the mapping itself (control of the robot and sensor, evaluation, etc.) are listed in the instructions "4_Mapping_manual". This section describes the Motion Tracking during the experiment, as well as the evaluation of the recorded images.

4.1 Video Recording

4.1.1 During Measurement

The camera records all of the robot's movements throughout the experiment. This caused problems with our camera:
Typical (amateur and medium professional) cameras have a 30 minute recording limit due to customs regulations: https://www.recordinglimits.com/faq/
Due to this problem, the mapping control stops shortly before reaching this time limit and asks the user to stop and restart the recording. If you are using a different camera or the customs regulations are different from ours, you can skip this stop/start.

There is also a maximum file size that a single video file can have. This again depends on your hardware and your SD card. In our case, the camera automatically started a new video file after approx. 17 minutes (30 minutes become 17+13 minutes).
In our case, the mapping takes about 3 hours, which results in 12 individual video files with different lengths (17min, 12min + leftovers).

Smartphones don’t have a 30min limit, but may be limited in remaining memory. In case you have a high quality webcam, you can use the free Software “OBS Studio” on your PC to record the video. In experiments we didn’t had a time limit with this setup.
If you are using DSLR cameras with high resolution (e.g. 4K) you may have problems with overheating. So, here you can use the implemented break to cool your camera.

4.1.2 Prepare Video Files for following Motion Tracking

A continuous video in a single file (mp4 format) is required for automated Motion Tracking.
We manually cut the individual video files into a single video in a cutting program (commercial) and rendered it as mp4 (length approx. 2h50min, size 20Gb). We have also had good experiences with shorter recordings using the free software OpenShot (www.openshot.org). During the 3h experiment, the software did not seem to be so stable, but this could also be due to our PC hardware. It is worth a try.
The two files resulting from the file size limit (in our case the 17min limit) can simply be attached directly to each other in the cut program and would not have to be cut. This results in 30min clips (6 in our case)

The 30-minute limit is more problematic (in the following we just use the term “30-minute” limit, which may come from any limit like overtemperature, memory etc. and may be any other time duration).

Manual control of the camera results in different offset durations for each video file at the beginning (movement of the robot does not start immediately when the camera restarts) and end (user does not switch the camera off immediately as soon as the script reports it).
The following precautions have been taken to simplify this: The mapping script always waits exactly 20 seconds until the next movement of the robot. The start of this time period is the moment at which a new position is sent to the robot (for details, see the next instructions).

With this knowledge, the 30-minute clips can be cut in such a way that they can then simply be joined together in the cutting program.

Step 1: Start of the 30min- SubClip
Typically, a few seconds elapse between the manual camera restart and the continuation of the mapping (in the mapping script, an "Enter" in the command window is sufficient).
The start of the robot's movement is sought in this time period. The last frame in which the robot **has not yet** moved becomes the new start of the SubClip (cut point 1).

Step 2: End of the 30min-SubClips
The mapping script reports whether the 30-minute limit has been reached shortly before the next robot position is moved, whereby the limit is restricted to 28 minutes to ensure that all positions are mapped reliably. If the limit is reached, the experiment pauses and the user is informed by a message in the command window and beep signals.
As the measurement runs completely automatically, it is not always necessary for the user to stop the camera directly with the pause. If the user is attentive and pays attention, it is easy to find the end of the SubClip (approx. end of video minus a few seconds). If this is not the case, the last few minutes of the SubClip must be searched for.
For the end, search for the last movement of the robot. Again, look for the last frame before the movement begins (see step 1). The end of the SubClip is exactly 20 seconds (500 frames at 25fps) later.

In our setup, the shoulder motor was relieved via a suspension to increase the positioning radius. This suspension had to be retensioned during the measurement. Since retensioning takes some time, it also makes sense to restart the camera during this time. The cutting of the subclip is analogous to the 30min limit

Render the video at the end with the following settings:

- no color grading
- FullHD (1920*1080px)
- 25fps, PAL
- mp4 format
- audio track can be deactivated
- rendering good/optimal (no quality reduction of the image)
- color space standard (image is converted into gray image in the analysis).

The bit rate was variable for us (average 12Mbit/s, maximum 18Mbit/s, standard setting of our cutting program). We do not investigated whether this is important. Depending on the performance of your PC hardware (especially CPU power) and the length of the experiment, rendering can take up to several hours.

To be honest, this step is tedious and time-consuming and should be better solved. If you want to continue Motion Tracking in post processing, it makes sense to automate this step and use a Python script based on the MoviePy library, for example:

https://pypi.org/project/moviepy/

If possible, the reboot of the camera should be automated so that the merging of the files can be made easier (e.g. control of the camera via USB, Powercycle or similar).

4.2 Analysis in Post Processing

After the video has been prepared, the evaluation of the Motion Tracking can be started using the main script "2_Main_ArUco_Detection_PostProcessing.py".

The math used here (4x4 affine transformations) is described in the paper and appendix. If there are any questions, please let us know and we can describe the math in a new version of this manual.

The script analyzes the video and detects the actual angles of the robot joints for each position. It also detects the base angle, although this is replaced by the sensor value in the later evaluation of the mapping. This software has been programmed in an object-oriented manner and is presented below. Details should become clear from the detailed comments.

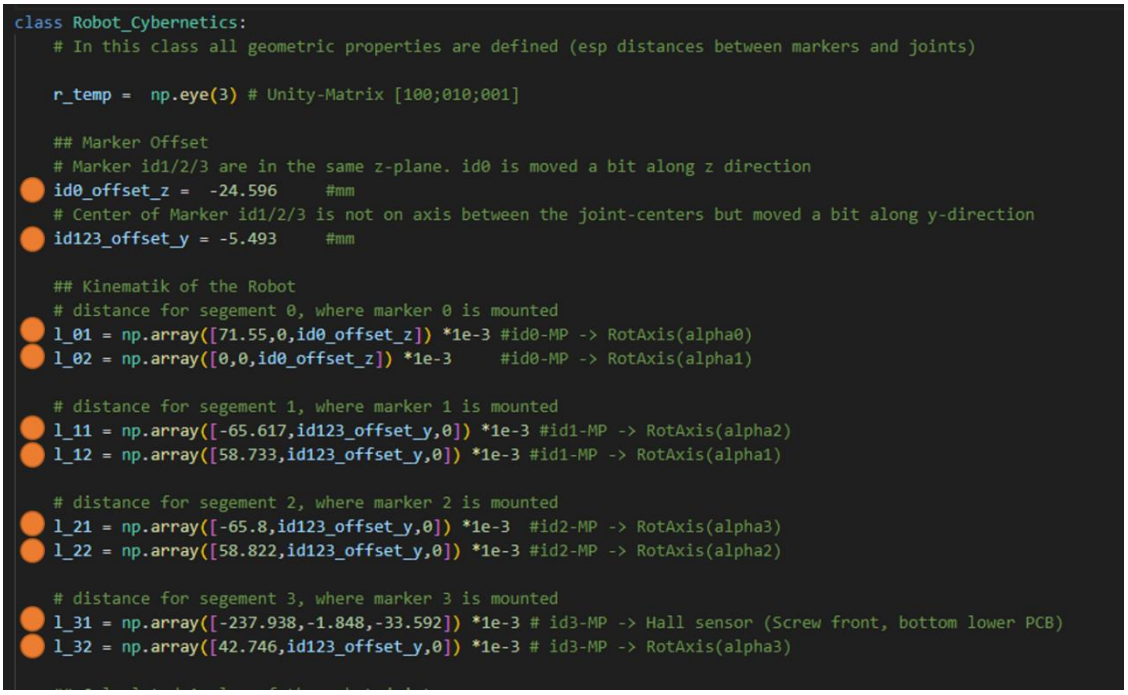
4.2.1 Preparation Software: Update SetUp File

First of all, the parameters of the software must match your robot and setup. If your setup is identical to ours, you don't need to change anything.
However, if you use a different robot or change the ArUco markers (position, size, library used), you must adjust the Motion Tracking.
These parameters are all stored in the class definition in the "MotionTracking_Classes.py" file and can be adapted to your setup:

- **Reference Board**
  Class "ReferenceBoard" (determination of the origin of the Reference System)
    - ID_List: All IDs of the markers that are installed on the Refrence board.
    - Marker_length: Side length of the ArUco markers used
    - ArucoDict: Library from which the markers come (in our case 5x5)

  Class “MarkerReferenceBoard" (definition of the marker itself)
    - mkX_T: Translation of each individual marker relative to the origin of the Reference System (X=1/2/3/4/5/6)
    - mkX_rot_angle: Rotation of each individual marker relative to the origin of the Reference System (X=1/2/3/4/5/6)
- **Robot Marker**
  Class "Robot_ArucoMarker" (all markers that are mounted on the robot)
    - ID_List: All IDs of the markers that are mounted on the robot.
    - Marker_length: Side length of the ArUco markers used
    - ArucoDict: Library from which the markers come (in our case 4x4)

  Class “MarkerOnRobot" (definition of the marker itself)
    - No change necessary
- **Robot kinematics**
  Class "Robot_Cybernetics" (mechanical description of the robot and the position of the markers on the robot)
    - In Chapter 1.2.3 of "2_Robot_manual" it was described how the exact position of the markers on the robot can be determined. These values are required in this class, see Figure 6



```
class Robot_Cybernetics:
    # In this class all geometric properties are defined (esp distances between markers and joints)

    r_temp =  np.eye(3) # Unity-Matrix [100;010;001]

    ## Marker Offset
    # Marker id1/2/3 are in the same z-plane. id0 is moved a bit along z direction
    id0_offset_z =  -24.596     #mm
    # Center of Marker id1/2/3 is not on axis between the joint-centers but moved a bit along y-direction
    id123_offset_y = -5.493     #mm

    ## Kinematik of the Robot
    # distance for segement 0, where marker 0 is mounted
    l_01 = np.array([71.55,0,id0_offset_z]) *1e-3 #id0-MP -> RotAxis(alpha0)
    l_02 = np.array([0,0,id0_offset_z]) *1e-3     #id0-MP -> RotAxis(alpha1)

    # distance for segement 1, where marker 1 is mounted
    l_11 = np.array([-65.617,id123_offset_y,0]) *1e-3 #id1-MP -> RotAxis(alpha2)
    l_12 = np.array([58.733,id123_offset_y,0]) *1e-3 #id1-MP -> RotAxis(alpha1)

    # distance for segement 2, where marker 2 is mounted
    l_21 = np.array([-65.8,id123_offset_y,0]) *1e-3  #id2-MP -> RotAxis(alpha3)
    l_22 = np.array([58.822,id123_offset_y,0]) *1e-3 #id2-MP -> RotAxis(alpha2)

    # distance for segement 3, where marker 3 is mounted
    l_31 = np.array([-237.938,-1.848,-33.592]) *1e-3 # id3-MP -> Hall sensor (Screw front, bottom lower PCB)
    l_32 = np.array([42.746,id123_offset_y,0]) *1e-3 # id3-MP -> RotAxis(alpha3)
```

Figure 6: Customizable parameters in the class "Robot_Cybernetics". The orange circles show the parameters that can be adapted to your robot

4.2.2 Adapt Experiment Parameter

After adapting the software to the setup and robot, the software still needs to know about the experiment that has been carried out.

The following parameters are adjusted in the "2_Main_ArUco_Detection_PostProcessing.py" script for this purpose:

- count_pos: Number of different positions that were approached during the experiment
- video_file: Storage location of the rendered video
- can_dist/cam_mtx: Correction factors that have been calculated in the camera calibration. The storage location corresponds to the same location in which the calibration scripts also store these tables. Make sure that these files also match the experiment itself (e.g. if you repeat the analysis later)
- Avg_Frames: Number of frames over which the position is to be averaged (normal average to increase accuracy). A value between 100 (=4s) and 200 (=8s) was good
- MT_Time_Move: Dead time in which the robot moves to the new position, stops and has swung out of any subsequent shaking. This time must be the same as in the mapping script
- MT_Time_Meas: Duration in which the sensor carries out its measurement. This duration is always guaranteed (see Ch 4.1.2). This time must be the same as in the mapping script

During the analysis, the result of the Motion Tracking is saved in a csv table in the “\2_Save_Folder” folder. Before you start the analysis, delete old contents of this folder (or rename it). Otherwise the new values of this table will be appended at the end.
For debugging, an image with drawn (detected & averaged ArUco markers) is saved in the "\3_Images_MotionTracking" folder. Nothing is overwritten. But it makes sense to clean up the folder before the analysis to keep an overview.

4.2.3 Run Analysis

Once everything has been set up and prepared, execute the script "2_Main_ArUco_Detection_PostProcessing.py".

Motion Tracking then runs automatically. Depending on which PC hardware you use (hard disk/CPU) and how many frames are averaged, it will take a while until the script is finished. If you want the analysis to be faster (e.g. for debugging), you can reduce the number of frames averaged.
The progress is displayed in the console.

During the analysis, a frame is displayed on which the detected markers (averaged position) are plotted. This frame is also saved.
If you notice any errors (especially if the drawn marker does not match the "real" marker), the script should be aborted.
Problems we had at this point, which caused mispositioned ArUco marker in the detection:

- Lightning of the experiment too poor. Either increase the brightness of the video (e.g. in the cutting program), or repeat the experiment.
  But before repeating it, use Chapter 3.2.1 and check the setup.
- The parameters “MT_Time_Move”/ ”MT_Time_Meas” did not matched the experiment, so that the roboter movement was part of the analysis and falsified the result.

5 Outlook

In our setup, Motion Tracking was carried out entirely in post-processing, which is quite time-consuming. However, this gives you complete control and makes debugging much easier.

A live analysis of the position during the mapping itself would be an advantage and should be easy to integrate. For that it is important to establish a connection to the camera via openCV (it worked for us with a FullHD webcam. However, the distortion of the image was horrible, so it made little sense).

A good integration of the Python scripts (used in Motion Tracking) into the Matlab scripts (used in the Mapping) should be checked.
We conducted our first experiments using a software localhost:
A localhost is opened in the Matlab script, to which the Python script connects via the "socket" library. The result of the Motion Tracking is then transferred via this line.

It would probably make more sense to translate the Matlab scripts into Python. This should not be a major problem as no special toolboxes are used.