CMPUT 501 Fall 2021 Group Report

by

Chen Jiang, Faezeh Haghverd, Javier Sales-Ortiz, Jeramy Luo,
Kerrick Johnstonbaugh, Sait Akturk

Master of Science

Department of Computing Science

University of Alberta

Abstract

In this report, we present and discuss the technical details for a ground-up Uncalibrated Visual Servoing (UVS) implementation using Python 3 and ROS. Our implemented UVS routine enables the smooth execution of reaching motions with an RGBD camera and a WAM manipulator, and it is easy to support types of robot arm manipulators. We hope this implementation can be a good start-up point for our future research combining vision and robotics.

Contents

1 Introduction **1**

2 Implementation Details **3**
2.1 Design . . . 3
2.2 Vision Implementations . . . 4
2.2.1 RGBD Vision and Visual Tasks . . . 4
2.2.2 Visual Tracking . . . 5
2.3 Control Implementations . . . 7
2.3.1 Visual-motor Jacobian . . . 9
2.3.2 Trajectory and Waypoints Planning . . . 9
2.3.3 Solvers . . . 10

3 Methods **11**
3.1 Simulation . . . 11
3.2 Visual Servoing Error . . . 12
3.3 Registration-based Tracking . . . 13
3.4 Uncalibrated Visual Servoing with RGBD Vision . . . 14
3.4.1 Visual-motor Control . . . 14
3.5 Trajectory Planning . . . 16
3.6 Solvers . . . 19
3.7 Dynamic Visual Servoing . . . 20
3.7.1 Data-driven Inverse Dynamics Modelling . . . 22
3.8 Functionality and Structure of libbarrett . . . 23

4 Case Studies **26**
4.1 Simulation . . . 26
4.2 Uncalibrated Visual Servoing Control . . . 27
4.2.1 Point-to-Point Tasks . . . 27
4.3 Data-driven Inverse Dynamics Modelling . . . 31

5 Conclusion **32**

References **34**

List of Tables

4.1 Error values (NRMSE) for predicted joint torques in single-joint and multi-joint dataset 31

List of Figures

2.1 Design flow of our UVS routine. 4
2.2 Vision setup for robot manipulation. 5

3.1 Waypoints sampled from Bézier Curve as seen from camera . . 17
3.2 Curve interpolations using different parameters. 18
3.3 Schematic neural network model 23
3.4 System organization in libbarrett. 24

4.1 Visualization of our simulated 3-axis arm manipulator with a PinHole and a CCD camera model. 27
4.2 Configuration of the WAM in the 2DOF uncalibrated visual servoing experiments. The two active joints are marked J2 and J4. 28
4.3 Configuration of the WAM in the 3DOF uncalibrated visual servoing experiments. The three active joints are marked J1, J2 and J4. 29

Chapter 1

Introduction

In this project, we examined how to build an uncalibrated visual servoing (UVS) routine from the ground up using a WAM arm robot and RGB-D vision for tracking. To accomplish this, we created the robot arm simulation with two virtual cameras for UVS with integrating what we learned during the course which is the analytical implementation of kinematics and inverse kinematics, the numerical solutions for inverse kinematics, and multiple-view geometry. After testing simulations with WAM robotic arm configuration, 2D UVS with point-to-point case implemented using ROS and Python3. The implementation consists of five submodules which are arm, camera, tracker, control, and interface modules. Initially, the Lucas-Kanade image tracker was used for getting the error which is the pixel distance between the desired location and current location. To move the robotic arm in 2D, two of the joints of the robotic arm are used. For the 3D version of the same task, MTF [3] based trackers were used to deal with complex changes due to 3D movements and the third joint of the robotic arm was utilized to achieve spherical motion. The task error was determined by combining 3D error using depth supervision provided by Intel Realsense RGBD camera and the 2D error mentioned above. The robotic arm control changed from positional-based control to velocity-based control to get smoother movement between trajectory waypoints. The trajectory planning was configured to fit curved spline using Bezier curves for better adjustment of the velocity changes from the initial point to midpoint and midpoint to the target point. For testing and visualization, the waypoints

are drawn on top of the camera user interface during execution.

Chapter 2

Implementation Details

In this chapter, we outline the implementation details of our UVS routine. First, we specify our overall routine design. Next, we present the details of our routine procedure, starting from vision to control modules.

2.1 Design

The overall UVS routine design is interpreted in Figure 2.1. ROS is used as the base communication platform between WAM robot node and the camera nodes being hosted on a master computer. Five specialized modules are categorized to handle interaction with real-time environment:

- **Arm**: Module to control robot arm through ROS. A dedicated module is built on top of libbarrett to support WAM robot arm position and velocity control.
- **Camera**: Module to acquire visual inputs from real-time camera stream. Wrapper modules are implemented to support Intel Realsense RGB-D cameras and standard webcam.
- **Tracker**: Module to support real-time visual tracking. Trackers from MTF [3] and OpenCV are wrapped inside to support real-time registration-based tracking.
- **Control**: Module to apply visual-motor control. Linear solving routine and inverse kinematics with visual servoing are implemented here.

- **Interface**: Module to host task types of interface for servoing. GUI communication and visualization routines are implemented here to handle various visual tasks (e.g. point-to-point tasks).

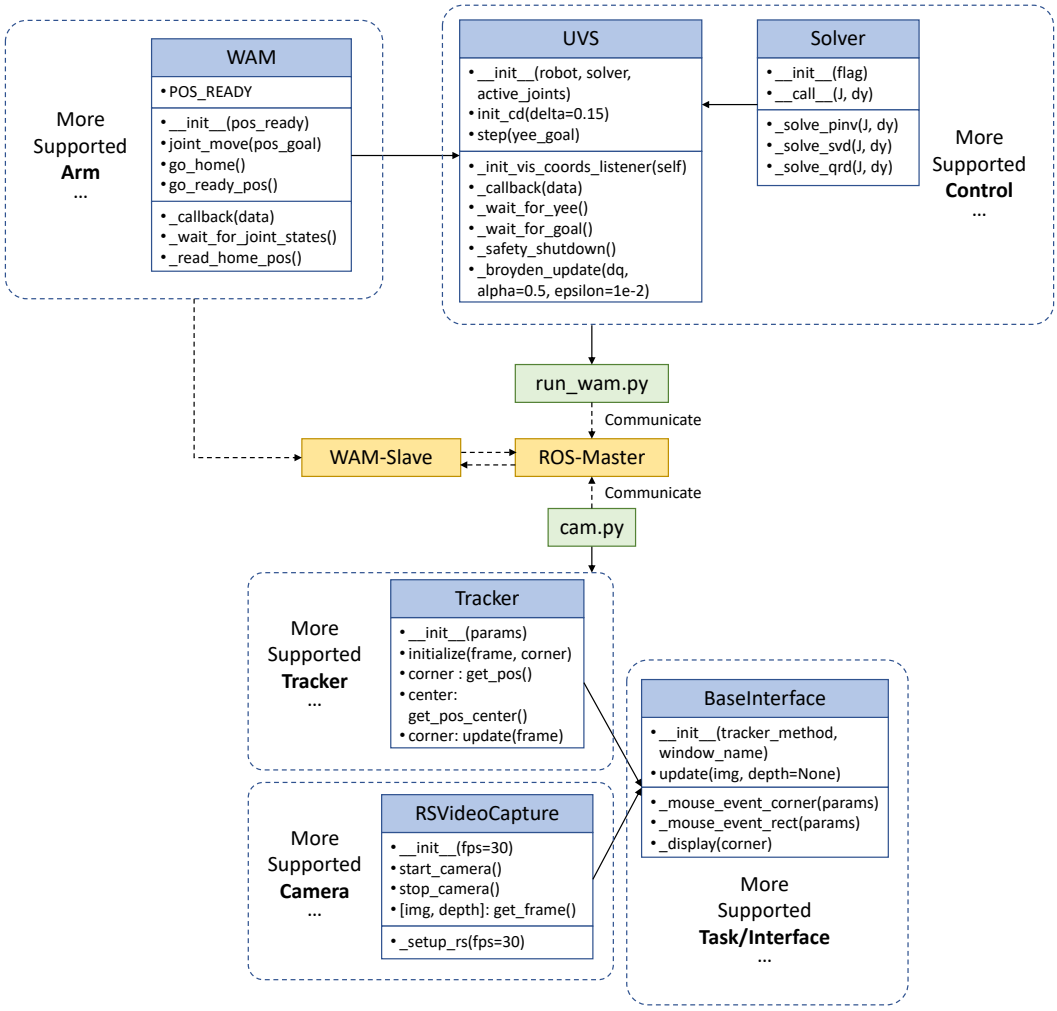

Figure 2.1: Design flow of our UVS routine.

2.2 Vision Implementations

2.2.1 RGBD Vision and Visual Tasks

The vision setup for our UVS routine is presented in Figure 2.2. A Intel Realsense Lidar D515 camera is setup to observe the scene where robotic

actions can possibly take place. RGB and depth camera frames are retrieved in 640×480 resolution with 30 FPS.

Figure 2.2: Vision setup for robot manipulation.

To specify a visual task for robotic control, a simple GUI interface is constructed using OpenCV. The user first specifies the type of visual tasks (e.g. point-to-point task, etc) to run by demand. By pressing "c" (corners) or "p" key, the user can then click on the image, and define the region or point to track with a dedicated visual tracker. The tracker records and publishes coordinates of the tracked regions into ROS communication.

2.2.2 Visual Tracking

Tracking the objects in the image domain, one of the most important tasks for visual-based robotic and mobile applications in particular visual servoing, SLAM, augmented reality Applications(AR). Image tracking systems could be summed under two main types which are based on online learning and detection-based trackers(OLT) and registration-based trackers(RBT). The usage of the tracker types is selected based on the task needs such that although OLT is used for long-term tracking where the robustness to appearance changes is more important than precision, in robotic applications, tracking precision and speed is the main consideration. The precision and speed requirement of

robotic and mobile-based tasks stems from the fact that the battery and hardware are limited to achieve mobility and agileness. Thus, in this project, the modular tracking framework(MTF) is utilized to get high precision and fast-tracking performance using RBTs. The framework divided tracking into three separate submodules which are state space model, appearance model, search method to create an extendable and efficient implementation of registration-based trackers. Multiple registration-based tracking implementation is available and detailed results could be found in the thesis paper of the author [3].

MTF is implemented in C++, thus requiring compilation for each machine with different configurations. Although the detailed description of the compilation process is written, the required third-party libraries and their versions are either outdated or no longer maintained, thus the compilation of the framework was required multiple attempts to solve issues. Another problem with third-party libraries is multiple versions of the third-party libraries cause problems with other programs and libraries in the machine. Therefore, the initial compilation was done for Docker which uses the Linux kernel of the machine thus creating machine-specific optimized binaries, isolated from the operating system thus could be used with multiple different versions of the same libraries without any compatibility issues or side-effects, creating lightweight images in terms of memory and disk space, portable configuration file could be used for different machines, thus shortens compilation configuration times from days to few minutes and runs nearly identical with native performance. However, the other team members did not have time to learn new tools, thus we decided to compile the framework for the machine natively, however, due to compatibility issues with OpenCV installed with ROS, the framework was compiled under the workspace folder portably.

MTF has many different registration based trackers that could be utilized, If no config file is provided MTF uses the ESM search method with SSD appearance model and homograph state-space model, however, according to our observations, the affine and homography transformations could not be tracked by the default tracker with desired precision. Thus, we experi-

mented with composite search methods, in particular, PFFC(Particle Filter + Forward Compositional Lucas Kanade) and LMES(Least Median of Squares + Efficient Second-Order Minimization) [3]. Based on our observations, the composite search methods obtained better precision and robustness to fast motions and illumination changes. The different state-space models were experimented with and observed that state-space models with low degrees of freedom reduce the precision of composite search models, thus, the homography state-space model with 8 DoF is chosen as the default state-space model to achieve the required precision for the task. One of the drawbacks with all experimented registration based trackers noted that the tracking precision and robustness reduced drastically with occlusion of the tracked image or losing track due to 3D rotation that the tracked template is completely lost for a moment, which is expected because most of the registration based trackers uses initial or recent patch templates and once the tracker loses the tracking information, the tracking is lost indefinitely. Additionally, parallel, hierarchical, and cascade tracking were examined, although cascade tracking hierarchical tracking with multiple state-space models increases the robustness of the tracking, the tracking speed was reduced substantially. In conclusion, we used the LMES composite search model with homography space state model and normalized cross-correlation(NCC) as appearance model is used.

2.3 Control Implementations

In this section we assume visual trackers have been initialized such that camera coordinates (u, v) and depth (d) information is available in real-time for both the robot end-effector and the target position/object. The general process flow for generating point-to-point movements via uncalibrated visual servoing is the same for both position and velocity control. First we will discuss the generic process, and then we will go into further detail about the implementation differences in position and velocity point-to-point UVS.

The generic process always begins with an initial visual Jacobian estimation procedure. Throughout this report a central difference method is used

to initialize the Jacobian (see section 3.4.1). The next step is calculating waypoints in visual coordinates between the robot end-effector and the target position or object (see section 3.5). The reaching process is executed through a series of smaller sub-reaches to each waypoint, in sequence. Each sub-reaching movement moves the end-effector a short distance in image space. To calculate the change in joint positions or the corresponding joint velocity commands, we assume the robot end-effector position in camera coordinates is approximately linear in the joint angles of the robot for very small movements (i.e. we make an assumption of "local linearity"). After solving the resulting linear system (see section 2.3.3) the robot is sent a command to execute the sub-reach (either through position or velocity control) and then the Jacobian estimate is updated (typically through Broyden's method, see section 3.4.1).

Position Control

In position control mode the sub-reach movements discussed above are executed by sending joint position commands to the robot. For the WAM, this is done in practice through the JointMove ROS service defined in the wam node in the open source Barrett ROS package (barrett-ros-pkg). The barrett-ros-pkg leverages the open source libbarrett library. The JointMove ROS service uses the moveTo method of the Wam class in libbarrett. The Wam.moveTo method commands the WAM robot to follow a trapezoidal joint velocity profile to move from the initial joint positions of the robot to the joint positions commanded in the call to the JointMove ROS service.

Velocity Control

In velocity control mode, the user must define the desired duration, T, in seconds, of sub-reach movements between waypoints. The sub-reaching movements are executed by solving for the desired change in joint angles, dq, and commanding joint velocities of dq/T for a duration of T seconds. At the end of each sub-reach (after T seconds), the Jacobian estimate is updated, and new joint velocity commands are computed and immediately sent to the robot. For safety, joint velocity commands are clipped; if the norm of the joint velocity

vector exceeds some threshold value, the joint velocity vector is rescaled to have a safe norm.

Joint velocity commands to the WAM are sent by publishing to the jnt_vel_cmd ROS topic. The wam node subscribes to this topic, executing the corresponding callback function any time a new message is published.

Both the jnt_vel_cmd callback function and the libbarrett Wam.moveTo function make use of the trackReferenceSignal member function of the Wam class defined in libbarrett. The trackReferenceSignal function connects the input of the supervisory controller system to either a stream of joint position or joint velocity commands, and the supervisory controller handles each appropriately (see section 3.8).

2.3.1 Visual-motor Jacobian

Before the manipulator begins its task, we first aim to have an initial approximation of the visual-motor Jacobian. To do this, the robot performs a Jacobian initialization routine, attributed as orthogonal exploratory motions [4]. The routine moves each of its joints one at a time, and approximate columns of the Jacobian by central difference.

Once the task has started and the visual coordinates for the end-effector and the target are known, the Jacobian is updated using Broyden's method in real time. The only case where the Jacobian is not updated is if the option of using "convergence steps" is set. This option is designed specifically for our safe experimental purpose, so that the end-effector can reach the target with greater precision once it is already close to it. This works by disabling the Jacobian updates and taking a sequence of steps towards the target position. See section 3.5 for details about the "convergence steps".

2.3.2 Trajectory and Waypoints Planning

Since the visual servoing system is modelled as linear, the Jacobian is only accurate in the vicinity of the position of the end-effector. This means that if we want to move the manipulator to a target point further away, we should

take intermediate steps. While generating these we have many options as to which properties we want the waypoints to have.

The simplest option is to generate a straight line in the image space between the starting and goal positions, and position the waypoints at equidistant intervals. We used this option for some of our initial experiments. However, in to accelerate as the end-effector departs form its initial position and decelerate as it approaches the target position, we can vary the distance between points. That is, points near the start and end positions are closer together, and points in the middle of the line are sampled further apart.

Additionally, we can specify other desired characteristics about the trajectory our end-effector follows. We experimented with having the end-effector leave its starting position and arrive to its final position by following a vertical trajectory. The horizontal movements in width and depth are performed at the intermediate steps. This curve is described by a Bezier curve, which its parameters are discussed more deeply in the methods section.

2.3.3 Solvers

Solver acts as the controller to find the required update in configuration for moving towared desired point based on the waypoints(dy) and the updated Jacobean (J). Overall, it mainly involves inverse Jacobian calculation. So as for solvers, we here programmed three alternatives, pseudoinverse, QR decomposition-based inversion, and singular value decomposition (SVD), which will be further explained in section 3.6.

Chapter 3

Methods

In order to successfully perform Uncalibrated Visual Servoing with a RGBD camera, some key-enabling methods need to be developed. In this chapter, we describe and formulate those key-enabling methods, including (1) Simulation, (2) Visual servoing error, (3) Registration-based tracking, (4) Uncalibrated visual servoing with RGBD Vision, and (5) Trajectory planning.

3.1 Simulation

To better understand the fundamentals of numerical methods with respect to robotics and vision, a simulation was conducted in which a 3-axis robot was first modeled using homogeneous transformation. After developing a suitable model, subsequent additions and modifications were added to further explore the different theories and methods involved in robotics and vision.

First, a simple forward kinematics model was created with a robot with dimensions modelled after a WAM research robot. With this model, angles could be defined for the joints of the robot and the position of the joints and end effector was returned. Inverse kinematics were then implemented with both analytical derivatives and numerical methods being applied. This modified model was able to return joint angles when given a goal point.

Lastly, the simulated 3D robot was projected onto a 2D image plane. With this image plane, inverse kinematics tests were repeated. This final model was a suitable representation of a simple visual servoing system that could be referenced during development of the visual servoing project with the WAM

research robot. This camera projection was unique to the simulation and will be described in this section.

The homogeneous coordinates of the robot simulation were mapped from a Euclidean 3D space into an Euclidean 2D space (central projection) using a pinhole camera model. This can be represented by the following equation where f is defined as the focal length of the camera.

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.1}$$

Two cameras were used during the simulation, and their positions and rotations were accounted for using the following equation in which R is a 3 x 3 rotation matrix that represents the orientation of the camera and C represents the coordinates of the camera in world coordinates:

$$X_{cam} = \begin{bmatrix} R & -R\tilde{C} \\ 0 & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.2}$$

With these two equations, one can map the 3D coordinates of the simulated robot onto two 2D image planes, each representing one camera. With these image planes, the previously described inverse kinematics tests could be repeated, completing our visual servoing simulation model.

3.2 Visual Servoing Error

To perform a positioning task through visual servoing with eye-to-hand camera, an minimizable error over visual features should be defined over camera inputs. The most simple definition is done by using visual coordinates. Specifically, let $\boldsymbol{y_t} = \{(x_1, y_1), ..., (x_n, y_n)\}$ be defined as a vector that holds the value of n visual coordinates tracked from a set of points at time t. The visual servoing error $\boldsymbol{e}$ can be expressed as:

$$\boldsymbol{e} = \boldsymbol{y_t} - \boldsymbol{y}^* \tag{3.3}$$

where $\boldsymbol{y}^*$ defines the desired target value of visual coordinates when the robot end-effector has reached its goal. The error reaches 0 when the visual coordinates of $\boldsymbol{y_t}$ perfectly collide with the visual coordinates of $\boldsymbol{y}*$. For point-to-point task, we set $n = 1$ and return the center of mass for a tracked region.

Depth Visual Features

A rotational positioning task can be achieved by incorporating both image feature and depth feature. More specifically, for a depth camera, an RGB image frame $I(x, y, t)$ along with a depth image frame $Z(x, y, t)$ can be retrieved at any given time t, with (x, y) being the visual coordinates. For n visual coordinates being tracked with $\{(x_1, y_1), ..., (x_n, y_n)\}$, we retrieve the corresponding n depth feature values as $\boldsymbol{z_t} = \{z_1, z_2, ..., z_n\}$. The final visual servoing error $\boldsymbol{e}$ is expressed as the concatenation of both image and depth error:

$$\boldsymbol{e} = [\boldsymbol{y_t}; \boldsymbol{z_t}] - [\boldsymbol{y}^*; \boldsymbol{z}^*] \tag{3.4}$$

3.3 Registration-based Tracking

Registration-based trackers (RBT) track the objects by warping the sequence of the image patches to align with the template. Unlike, online learning and detection-based trackers (OLT), RBTs could work on higher degrees of freedom which enables higher precision. The reasoning for why using higher degree freedom (DoF) enables better precise tracking is that complicated motions such as 3D rotations and translation could be tracked more accurately using higher DoF due to fact that higher level transforms could better simulate the projectivity, which converts relative change between the objects and the camera in the 3D world into 2D images. A recent study related to registration based trackers called modular tracking framework (MTF) subdivides the tracking task into three submodules which are search method (SM), appearance model (AM) and state-space model (SSM). The study explains the general flow of the registration based trackers as for each frame I_t optimal parameters calculated by the search method using appearance model f and search state model of

w. The search state model(w) warps the initial points on the grid x_0 using optimal parameters and send the warped points to $\mathbf{w}(x_0, p_s t)$ to the appearance model. Then, the appearance model retrieves the pixel values at these warped points and uses $p_a t$ to compute the warped patch's resemblance to the template, which it sends back to the search model. Lastly, this resemblance is used by the search method to determine the parameters p_{t+1} that maximise it for the next frame I_{t+1} [3].

$$p_t = \underset{p_s, p_a}{\operatorname{argmax}} f(I_0(\boldsymbol{x_0}), I_t(w(\boldsymbol{x_0}, p_s)), p_a) \tag{3.5}$$

3.4 Uncalibrated Visual Servoing with RGBD Vision

3.4.1 Visual-motor Control

We aim find Δq by solving the linear system:

$$J\Delta q = \boldsymbol{e}$$

where:

- $\boldsymbol{e} = \boldsymbol{y^*} - \boldsymbol{y_{ee}}$: The image error, where $\boldsymbol{y_{ee}}$ is the coordinate of the robot end effector position, and $\boldsymbol{y^*}$ can take the value of either a waypoint or the final goal position.
- Δq: Is the change in the angle of the robot joints.

Thus, the joint angle update will follow the operation:

$$q_{t+1} = q_t + \Delta q$$

For these joint updates to be succesful in completing the task, we need to initialize and update the Jacobian.

Jacobian Initialization

The manipulator starts its task at a known configuration of joint angles. Before the Jacobian Initialization, a tracker must be placed at the position of the end effector. This position takes the name of $\boldsymbol{y_{ee}}$ for $\boldsymbol{y}$ of **e**nd-**e**ffector.

Then, each joint is moved one at a time in the positive and negative directions. The movement starts at the most distal joints (furthest away from the base) and continues towards the more proximal ones (closer to the origin).

Then these positions are recorded, and a first approximation of the Jacobian is given by filling the matrix column by column:

$$\boldsymbol{jac}_j = \frac{\boldsymbol{y_{ee,+delta}} - \boldsymbol{y_{ee,-delta}}}{2 * \delta};\ j \in [1, m] \tag{3.6}$$

where:

- $\boldsymbol{y_{ee,+delta}}$: the position of the end effector after positive joint movement
- $\boldsymbol{y_{ee,-delta}}$: the position of the end effector after negative joint movement
- $delta$: radians the joint moves (0.15 rad $\approx 8.6°$)
- j: the column of the Jacobian being filled $j \in$ [joints being used in WAM]
- m: the number of joints being used in WAM

And the whole Jacobian is formed by horizontally concatenating:

$$J = [jac_1, jac_2, ...jac_m] \tag{3.7}$$

Jacobian Updates

The Jacobian is updated after every movement step using Broyden's method, except for the condition described in section 3.5. Broyden's method is implemented with an α to scale the size of the updates.

Broyden's update is performed by the following operation:

$$J_t = J_{t-1} + \alpha * \frac{(\Delta y_{ee} - J_t \Delta q)\Delta q^T}{\Delta q^T \Delta q + \epsilon} \tag{3.8}$$

where:

- α: Scaling factor of the update, for our experiments $\alpha = 0.5$
- $\Delta q = q_t - q_{t-1}$: Change in joint positions of the manipulator.
- ϵ: Constant factor added to the denominator to prevent division by zero ($\epsilon = 0.01$)

The key advantage of using Broyden's method instead of other approximation methods such as Newton's is that we do not need to specify the kinematic equations of the WAM in order to approximate the Jacobian.

3.5 Trajectory Planning

Because the Jacobian is only accurate in the vicinty of the position of the end-effector, points between the manipulator's position and the goal (waypoints) are generated in order for the robot to take intermediate steps. In our experiments, we used the following methods for trajectory generation.

Equidistant points on straight line

This is the simplest method of generating waypoints. The only parameter that is used is the number of steps. The process followed to generate these waypoints is simple.

- Define $pos_{start} = (x_s, y_s, z_s)$ and $pos_{end} = (x_e, y_e, z_e)$
- Define a linear space $t = [0, ..., 1]$, where $|t|$ = number of steps.
- For the x coordinates of the waypoints $x_i = (x_e - x_s) * t_i + x_s$
- Similar for y and z coordinates.
- Waypoint$_i = (x_i, y_i, z_i)$

Thus, the manipulator can start by approaching Waypoint$_2$ (the first waypoint equals the starting position). Once a step in that waypoint's direction is taken, the target position moves to the next waypoint. This causes the end-effector to approach the goal (final) position through intermediate steps.

Bézier curve

While the previous method is simple and effective, we can obtain useful properties when using more complex trajectory planning.

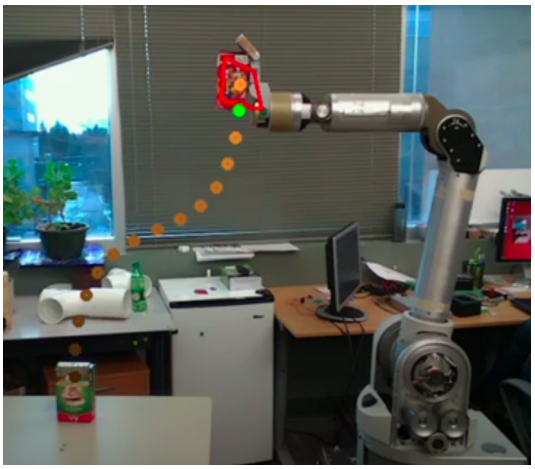

Figure 3.1: Waypoints sampled from Bézier Curve as seen from camera

As a flexible and customizable curve for interpolating between the start and end positions we chose a cubic Bézier curve as seen on Figure 3.1. Note how the end effector leaves its starting position vertically and reaches the goal vertically as well. This is useful when we care about the angle the manipulator approaches to grasp certain objects. Additionally, the color of the points indicates their depth, with lighter being shallower and darker being deeper.

For a brief qualitative description, Cubic Bézier curves [1] interpolate between four points with the line beginning and ending at the first and last points. The algorithm we designed calculates the intermediate points to give us our desired properties. These intermediate points "pull" the line into their respective direction. Here we focus on two parameters: how intensely the curve turns, and how uniformly the points are sampled.

The intermediate points in Figure 3.2 are shown in red. The plots 1. and 3. show the trajectory turning drastically. This is useful when the end point needs to be approached vertically from a greater distance. The curves in plots 2. and 4. are closer to a straight line. This would be the case when the constraint of approaching vertically can be relaxed, and more importance is placed on minimizing the distance travelled.

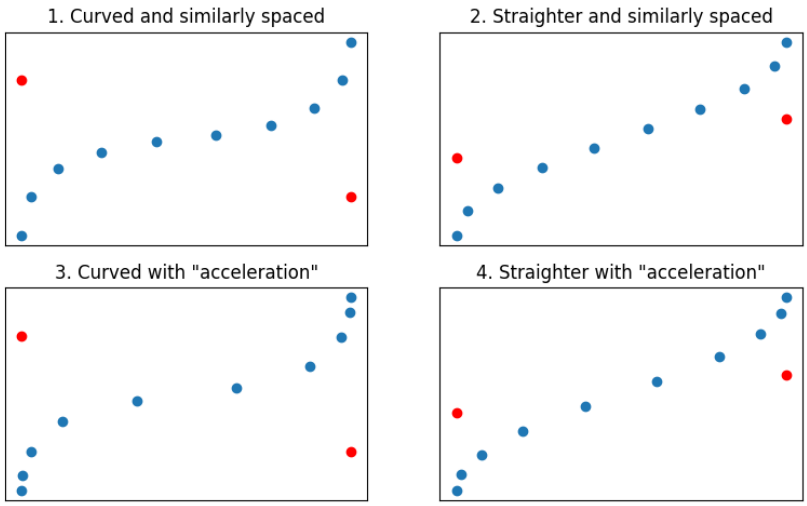

Figure 3.2: Curve interpolations using different parameters.

The curve in plots 1. and 2. show the points being sampled at approximately uniform distances. This maintains the end-effector at a relatively constant speed throughout the trajectory. In contrast, plots 3. and 4. show a clustering of points near the end points while less are sampled from the middle of the trajectory. This has the effect of the initial and final movements of the end-effector being performed slower in comparison to the middle points. Hence, we can indirectly introduce acceleration in the trajectory with the spacing of the waypoints.

The last parameter, not pictured in Figure 3.2, is the number of points in the trajectory. For our experiments we chose 15 steps.

By using Bézier Curves for trajectory planning we found that we could have greater control in specifying how the manipulator completes its task.

Convergence Steps

To finally reach the target point we have included the option to take a set of "convergence steps". In these convergence steps we fix the Jacobian (i.e. we keep the latest Jacobian estimate and stop updating it). Each convergence step is intended to move the end-effector to the point halfway between the initial ee position and the target position. Changes in joint positions (or joint velocity commands) are calculated by solving Jdq = dy for dq where dy is

1/2(y_ee + y_target). Geometric series 1/2 + 1/4 + ... will converge with the ee eventually reaching the target. In practice we stop taking full steps and updating the Jacobian once the norm of the difference between the end-effector and the target is below some threshold. Then we begin the convergence steps and once again cease motion when the distance between the ee and the target is small enough (below some smaller threshold).

3.6 Solvers

Solver is responsible for solving the 3.9 equation, finding the command Δq based on the current activated waypoint and the updated Jacobian (J).

$$\boldsymbol{e} = J\Delta q \tag{3.9}$$

In this regard, we used pseudo-inverse which sets the Δq value to:

$$\Delta q = J^{\dagger}\boldsymbol{e} \tag{3.10}$$

This method solves for all J matrices- square, non-square or not full rank. However, it has stability problems near singularities neighbourhood, which means that for configurations near singularity you can achieve large Δq for small changes in $\boldsymbol{e}$ which makes the problem ill-conditioned in that neighbourhood.

However, Jacobian matrix inversion for high-dof problems, worst case 6 or more degrees of freedom, is computationally intensive. For that case, we provided two numerical tools to compute the J inversion. First, in QR-decomposition approach, J is decomposed to:

$$J = QR \tag{3.11}$$

Where Q is a unitary matrix and R is an upper triangular matrix. Using properties that these matrices offer, we have the 3.10 equation solved as:

$$R\Delta q = Q^{H}\boldsymbol{e} \tag{3.12}$$

Where Q^H is the Hermitian transpose of Q, and next step is solving the resulted equation by back substitution to find Δq.

Second programmed alternative for solver is based on SVD, where J is decomposed to

$$J = U\Sigma V^* \tag{3.13}$$

and for the $m \times n$ J matrix, U is an $m \times m$ unitary matrix, Σ is an $m \times n$ diagonal matrix, and V is an $n \times n$ unitary matrix,and V^* is the conjugate transpose of V. The pseudo-inverse is calculated based on inverting singular value decomposed J, and so the 3.10 equation is solved as

$$\Delta q = V\Sigma^{-1}U^T \boldsymbol{e} \tag{3.14}$$

.

3.7 Dynamic Visual Servoing

Different control approaches can be utilized in visual servoing, kinematics-based, as we used so far, or dynamics-based, as Oliva et al. explored in their work [2]. Discussing the use of dynamic visual servoing, in this section, we talk about model-based control algorithms and how to extract the manipulator direct/inverse dynamics modelling.

Approaching robotic control systems, centralized control algorithms allow taking advantage of the knowledge of manipulator dynamics, partially or completely, to generate compensating torques for nonlinear terms. By compensating system nonlinearity, these kinds of algorithms allow for designing more efficient controllers, with more accurate and reliable performance. In dynamic visual servoing, visual features play role of the measurement process and input affecting the system dynamics [5]. Oliva et al. integrated the visual feedback with an impedance controller to shape a dynamic visual servo control system which can interact with the environment [2].

Generally speaking, direct dynamics modelling of a robotic arm involves computing joints' acceleration given the joint angle(q), velocity ($\dot{q}$) and torque

(τ); while modelling inverse dynamics involves computing the required command torques in each joint given a particular desired joint configuration, i.e. joints' angle, angular velocity and angular acceleration ($\ddot{q}$). Noteworthy, desired torque in each joint depends on the joint states of all robot joints.

The early days of robotic research approached dynamic modelling problems using rigid body analytical modelling tools, i.e. Lagrange formulation and Newton-Euler method. Lagrange formulation is a variational approach based on the kinetic and potential energy of each link. Newton-Euler method, on the other side, relies on applying Newton's second law of motion on joints to have force balance in each. The dynamic modelling of a robotic arm usually is formulated based on Lagrange formulation as:

$$\tau = M(\theta)\ddot{\theta} + C(\theta, \dot{\theta}) + G(\theta) \tag{3.15}$$

M is the $n \times n$ mass matrix, C is the $n \times 1$ vector containing velocity product terms, and G is the $n \times 1$ vector of gravity terms. Overall, this equation looks like Newton's second law of motion (f = "ma" term + gravity term), except that the "ma" term depends on not only the joint acceleration but also the products of joint velocities. This is due to the fact that the mass of the moving parts is not constant when it comes to the dynamic behaviour of robotic systems.

However, deriving the manipulator's inverse dynamics analytically is not considered efficient as it does not accurately consider all nonlinear system behaviours, especially frictional and backlash responses. Moreover, these common rigid body modelling tools lead to significantly tedious and computationally heavy calculations for robotic arms with high degrees of freedom (DOF), like the 7-DOF Barrett WAM arm. Furthermore, modelling the rotor inertia effects is considered to be of utmost importance and challenging at the same time. To the best of our knowledge, the literature review showed that people mostly do not consider rotor inertia effects in manipulators' dynamic modelling.

To overcome the mentioned limitations of classical modelling approaches, data-driven learning-based algorithms have been explored recently. This ap-

proach involves learning the unknown nonlinear inverse dynamics for the robot based on input data derived from the robot while tracking optimized trajectories which excite the dynamic system. Yilmaz et al. deployed a neural network to predict the inverse dynamics model of the 6-degree-of-freedom da Vinci Research Kit [6]. They assumed that the manipulator's dynamic always depends on the current joint configuration. We replicated their approach for inverse dynamic identification of our current 7-Dof WAM arm.

3.7.1 Data-driven Inverse Dynamics Modelling

The inverse dynamics problem is formulated here to find the lumped relationship between joints' states, i.e. position and velocity, and the inverse dynamics torque for the 7 Dof WAM arm:

$$\hat{\tau} = H(\dot{q}, \ddot{q}) \tag{3.16}$$

Where $\dot{q}$ and $\ddot{q}$ are 7-dimensional vectors of current joint angles and current joint angular velocities, respectively, is a 7-dimensional vector of the estimated joint torques, and H is the lumped function to estimate the joint torques based on $\dot{q}$ and $\ddot{q}$.

Data recording includes three steps: excitation trajectory generation, performing the experiments by robot tracking assigned trajectory, and finally recording and collecting data samples. Based on joint limits, the desired trajectory is generated by a normally distributed series of joint position commands through a ROSPY topic. Next, the generated trajectories are published as commands to the robot through ROS, where a joint state listener function collects the published joint states by the WAM controller computer. Joint state includes time, joint position, velocity, and torque (effort). All listened states are collected as a JSON data set. The size of the Gaussian distributions for generating desired joint positions is set to 1000, with zero-mean and standard deviations based on each joint angle limit. The robot controller computer has a frequency of about 400 Hz for data collection while the robot paves tracking the trajectory of these 1000 generated desired positions. Two datasets were

collected here, first with only one joint moving and then multi-joint moving.

A one-layer neural network is used here as a black-box model to identify the lumped function H. As mentioned earlier, Yilmaz et al. work on deploying NN to estimate da Vinci Research Kit's inverse dynamics model is replicated here with some minor modifications. The NN is modelled as depicted in (figure):

- One input layer of 14 neurons for the angular position(q) and velocity (dq) of all joints
- One hidden layer of 100 neurons
- Hyperbolic tangent activation functions were used for each layer
- One output layer of 7 neurons for inferred joint torques

We train the neural network with backpropagation in Pytorch, using the mean squared loss function for both the training criteria and also the validation loop.

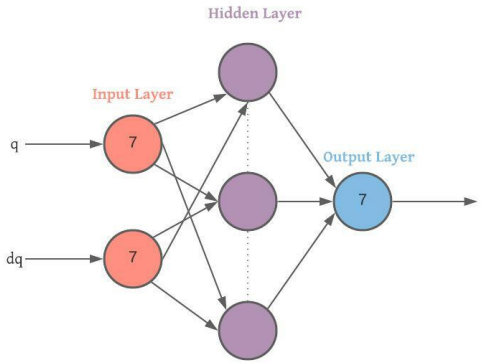

Figure 3.3: Schematic neural network model

3.8 Functionality and Structure of libbarrett

Libbarrett is an open source real-time control library for the WAM written in C++ and maintained by Barrett Technology. The purpose of this section is to provide some tools to the reader to understand the structure of libbarrett and how the high level position or velocity commands used in our experiments are

processed by libbarrett and translated to motor torque commands.

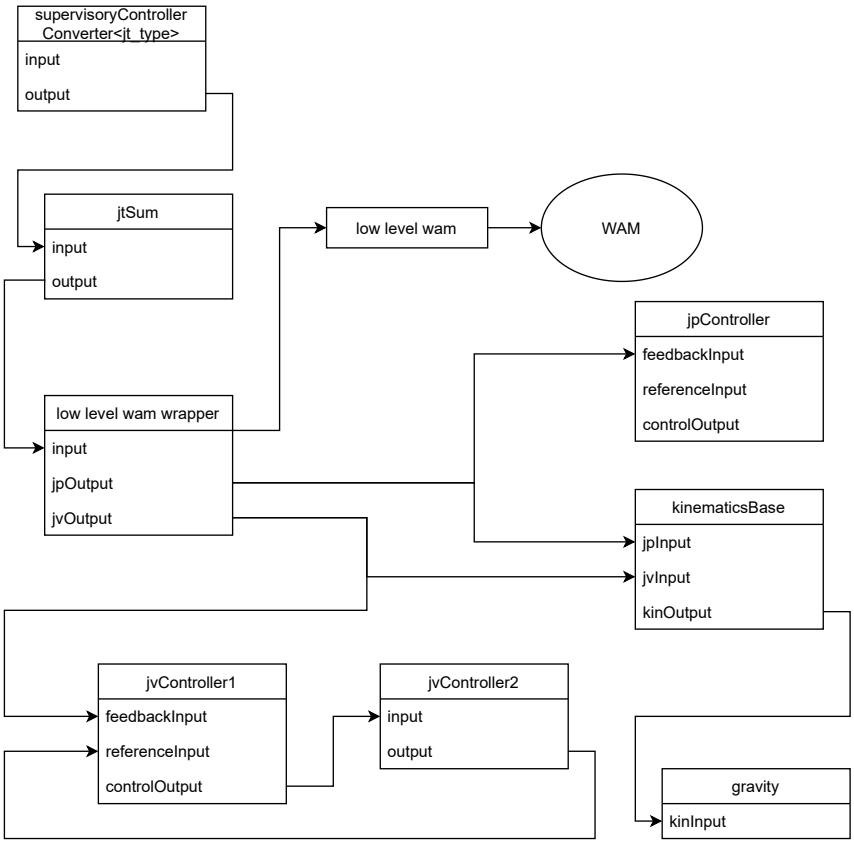

Figure 3.4: System organization in libbarrett.

The first thing to understand, especially for the reader who is not particularly familiar with large C++ packages, is that the implementation details of most classes are found in files with names that end in "-inl.h". Where some C++ projects might have a class header file "foo.h" with implementation details in the corresponding "foo.c" file, libbarrett typically does not include implementation details in files with a ".c" or ".cpp" extension. Take for example the systems::Wam class. In the "libbarrett/include/barrett/systems/" folder (note: not in the "libbarrett/src/" branch of the directory tree), we have the "wam.h" header file. If we want to see the actual constructor for the systems::Wam class, or the implementation of any of the member functions, we have to look one folder deeper, in the "libbarrett/include/barrett/systems/detail/" directory, and open the "wam-inl.h" file. "wam-inl.h" is included at the bot-

tom of the "wam.h" header file, and there is no "wam.c" or "wam.cpp" file whatsoever.

Another feature of libbarrett that deserves some attention and explanation are "systems". Here we briefly discuss the functionality and structure of systems in libbarrett. However, we refer the reader who requires a more thorough understanding to the "ex0_systems_intro.cpp" file in the examples folder of libbarrett. The purpose of systems, according to the authors of this file, is to give programmers access to low level control of the WAM without "getting bogged down in the details of real time programming". Systems can be thought of as similar to blocks in a block diagram. The output of one system can be connected to the input of one or more different systems. In this way systems can be chained together or used to provide a stream of information to several other systems. Each system implements the operate() function, which is where the data processing/calculations of the system take place. Instead of calling the operate() function directly, the systems::ExecutionManager handles calling the appropriate operate() functions of each connected system.

The systems::Wam class is instantiated whenever we interact with the WAM. The systems::Wam has many other subsystems that are initialized and connected in its constructor. We have mapped out the network formed by a subset of these subsystems in figure 3.4. The names of each subsystem are relatively self explanatory, and in general the behavior of each of these systems is as one would expect. There are many more subsystems in the systems::Wam class (many of which deal with control of the tool orientation) that we have omitted for clarity and because they were not as relevant to our experiments. We encourage the curious reader to explore the operate() function implementation in each of these subsystems as necessary.

Chapter 4

Case Studies

In this chapter, we discuss what we learned while developing and experimentally testing our UVS routine. We study multiple case scenarios where methods of dynamics, visual tracking and uncalibrated visual servoing are feasible.

4.1 Simulation

The 3-DOF controls using uncalibrated visual servoing were first performed with a simulated 3-axis robot. Figure 4.1 presents a visualization of our arm simulator with views from two virtual camera models. Our simulation code is open-sourced.

By applying camera translation and rotation, we setup the two simulated camera models side-by-side, forming a simple scenario stereo vision. For each simulated camera, the z-axis were pointing in the direction of the 3-axis robot. The visualized camera views were implemented by the commonly applied right-handed world coordinate system of computer vision.

To return visual visual errors used for uncalibrated visual servoing control, a simple point tracker is hooked onto the camera views to return visual errors from the simulated camera views.

Compared to basic servoing where perfect 3D coordinates are accessible, and are directly applied in the motor-control system, our simulated uncalibrated visual servoing model shows little difference in convergence when performing basic reaching motion. The key to maintain good convergence is to stably setup and translate the simulated stereo cameras, so that issues of visual

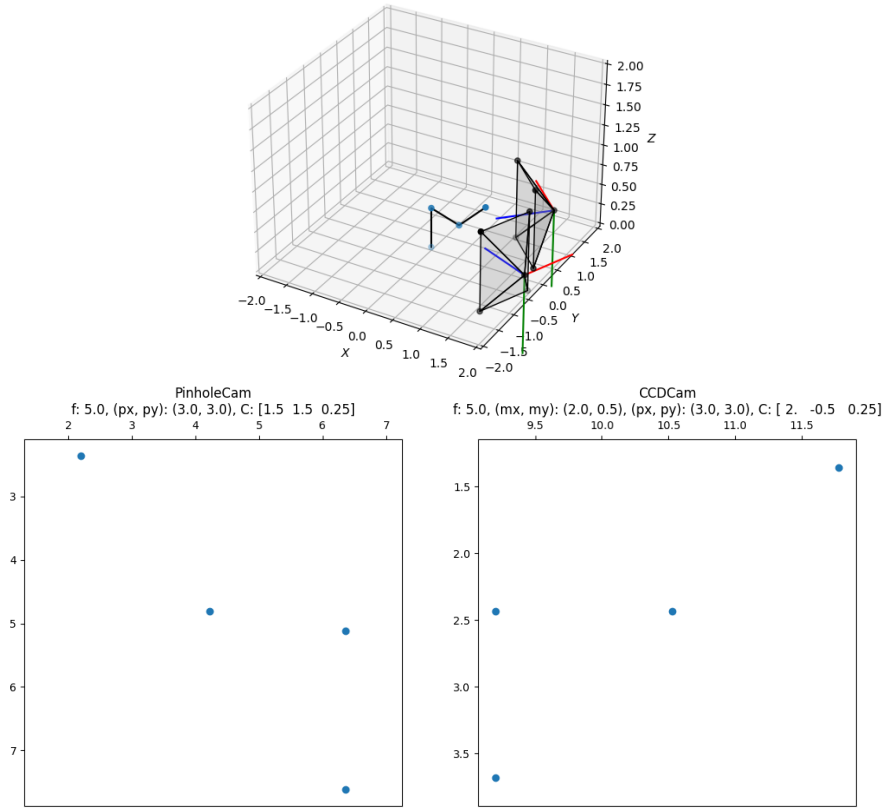

Figure 4.1: Visualization of our simulated 3-axis arm manipulator with a PinHole and a CCD camera model.

ambiguity can be addressed. To validate this, we setup a comparison scenario, where only one simulated camera is used. In this one-simulated camera experiment, it was much harder to converge for the uncalibrated visual servoing control, due to the fact that one 2D view is fundamentally ambiguous visually. And the fact that only one pair of reference 2D end effector coordinate is used, forming a 2×3 visual-motor Jacobian and rendering the linear system under-determined thus more unstable for convergence.

4.2 Uncalibrated Visual Servoing Control

4.2.1 Point-to-Point Tasks

2-DOF Control

We release our implemented UVS routine in GitHub.The 2-DOF control experiments were the first uncalibrated visual servoing experiments we performed

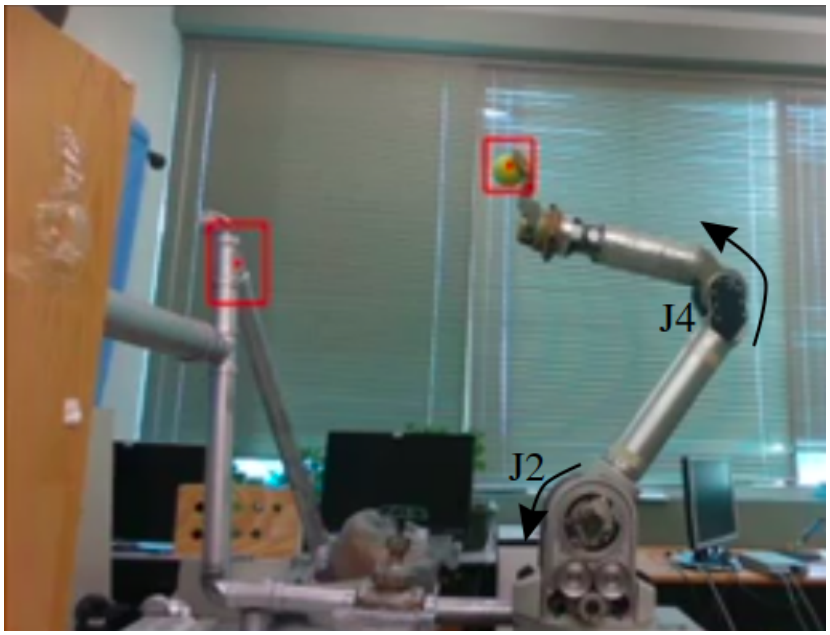

Figure 4.2: Configuration of the WAM in the 2DOF uncalibrated visual servoing experiments. The two active joints are marked J2 and J4.

with the physical WAM. As seen in figure 4.2, the WAM performed a point-to-point reaching maneuver, starting from an upright position and reaching toward a mock-industrial pipe setup. The WAM was constrained to only use joints 2 and 4, which can be thought of as the shoulder and elbow joints of the robot, respectively. The WAM held a tennis ball that was tracked with a Lucas Kanade tracker. The target was initially also tracked with a Lucas Kanade tracker, but since the target was generally stationary, we began specifying the target with a static point in camera coordinates.

Problems that appeared in our experiments were initially difficult to understand, because we didn't have the waypoint visualization at this point. We relied on printing the Jacobian at each step during stepwise execution of the sub-reach movements to diagnose and understand the issues we were facing. The main issue that would occur could be described as instability of the system, where a series of small sub-reach movements were followed by very large, undesired movements. These larger swinging movements sometimes required us to use the emergency to avoid collision (usually with the nearby table).

The simple solution we first implemented to address this problem was simply to include a damping factor in the Broyden update to the Jacobian. Using damped updates significantly increased the robustness of the system to the described instability.

3-DOF Control

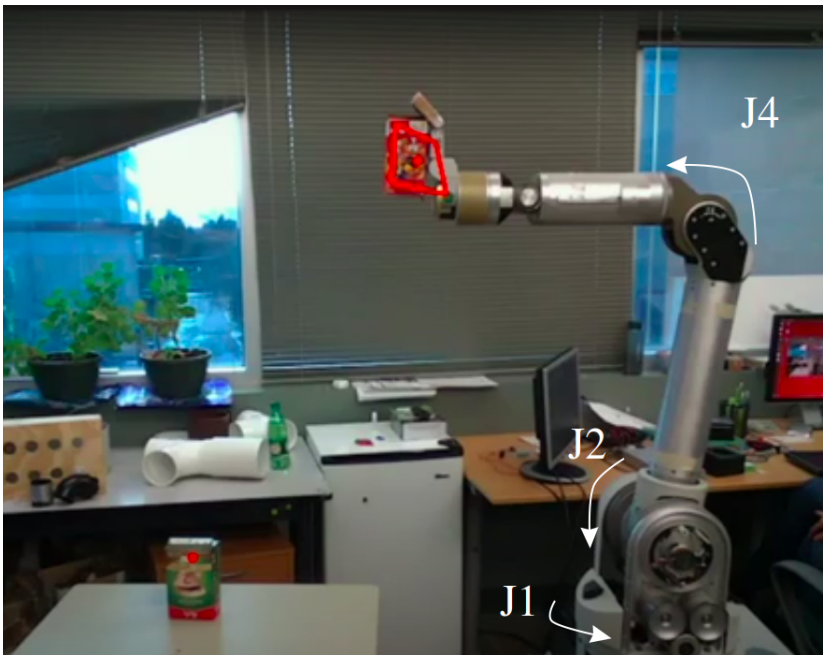

Figure 4.3: Configuration of the WAM in the 3DOF uncalibrated visual servoing experiments. The three active joints are marked J1, J2 and J4.

The 3-DOF control experiments began with initial experiments in which we used position control to move between waypoints linearly interspaced between the initial end-effector position and the goal position. We continually upgraded the system throughout the set of these experiments, adding MTF tracking, velocity control, spline trajectory planning, and waypoint visualization. In most experiments the end-effector position was tracked by tracking a small box affixed to the hand of the WAM. As in the 2-DOF experiments, we generally specified the goal position as a static point in image coordinates (by clicking the point in the video feed of the camera).

The three joints used in the 3-DOF experiments were joints 1, 2 and 4. Ac-

tuating joints 2 and 4 enabled the WAM to move in a plane, while actuating joint 1 rotated that plane. Actuation of joints 1, 2 and 4 allowed the WAM to reach in a 3-dimensional spherical work space.

The center of the MTF tracker was used to determine the image space coordinates of the robot end-effector. The depth of the same point was read to get a 3DOF position measurement.

A lesson we quickly learned in the 3-DOF control experiments was that it is crucial to ensure the tracked point of the end-effector moves when each active joint moves in isolation. We initially used the same initial configuration that we used in the 2-DOF experiments, with the hand of the robot positioned above the shoulder joints of the robot (see figure 4.2). When joint 1 was actuated during the central difference based Jacobian estimation, the object held in the hand of the robot rotated, but the center of the tracker remained at almost the exact same position relative to the camera (u, v image coordinates and depth remained approximately constant). For this reason the column of the initial Jacobian estimate that corresponded to the change in end-effector position with respect to motion in the first joint was close to a zero vector. This caused issues in the initial steps of visual servoing, as the initial Jacobian was nearly singular.

We were able to solve the problem of a singular initial Jacobian by simply changing the initial configuration of the WAM. We chose a configuration where the hand was not directly above the shoulder joint, such that isolated movement of each individual joint did cause a significant change in the tracked position of the end-effector. In the new initial configuration, the first link of the robot was approximately vertical, with an angle of approximately 80 degrees between the first and second link (see fig 4.3).

Dataset	first attempt	second attempt	third attempt	Avg
single-joint	2.97	3.01	3.13	3.03
multi-joint	2.27	1.91	1.93	2.03

Table 4.1: Error values (NRMSE) for predicted joint torques in single-joint and multi-joint dataset

4.3 Data-driven Inverse Dynamics Modelling

The normalized root mean square errors (NRMSE) between estimated and measured torques for each joint are calculated and considered as the accuracy validation measure of the trained inverse dynamic function. Each dataset was trained three times. 4.1 represents the results for different attempts, along with the associated average of each dataset.

As expected, nonlinearities increase with having movement in multiple joints, compared to one-moving-joint trajectory, where, as a consequence, performance error is lower.the showed error higher error is due to smaller training set. No significant difference was observed between the obtained results and the results of the Yilmaz et al.

Chapter 5

Conclusion

In this report, we investigate the ground-up construction of an uncalibrated visual servoing(UVS) routine with a WAM arm robot and RGBD vision for tracking. WAM robotic arm kinematics and inverse kinematics were tested in a simulation environment using configurations provided by the WAM robot, we learned to implement analytical kinematics and inverse kinematics and some numerical solutions for obtaining inverse kinematics and initial jacobian configuration. The simulation was extended by adding two virtual cameras to simulate point-to-point visual servoing tasks using image jacobians.

After completing tests with the simulation, the group outlined the implementation details for a real WAM arm robot using Python3 and ROS. We used Libbarret to control the WAM arm robot and one Intel RealSense RGB-D camera to capture the depth information without requiring a secondary camera for the task. Our implementation consists of five main modules which are arm module which controls the WAM arm robot using Libbarrett and ROS, the camera module manages image stream from the camera, the tracker module implements the image trackers using MTF [3] and OpenCV based trackers, the control module implements numerical solvers, kinematics and inverse kinematics to control WAM robotic arm and lastly interface module to handle various visual tasks.

The first implementation was done for the 2D UVS task with the point-to-point scenario where the target and the robotic arm share the same 2D plane and the task aim to use one of the simplest image trackers which is the

Lucas-Kanade tracker and only two joints of the robotic arm which are J2 and J4 to reach the determined target without depth supervision. According to our experiment results, the undesired large movements were observed during experiment execution, and to address this problem the damping factor was introduced for the Broyden method to increase the robustness of the system.

After solving the issues related 2D UVS task with the point-to-point scenario, a 3D version of the same task was experimented with using depth supervision, after selecting the target point and initial point for the robotic arm, the initial Jacobian was calculated using the central difference method, the linear trajectory between two points divided into multiple waypoints for Jacobian updates. The third joint activated which is the J1 joint of the robotic arm for 3D movement, we changed our image tracker to MTF based trackers that use a composite search method called LMES with an appearance model NCC and homography as the tracker's state-space model to improve tracking precision during execution which was essential to deal with complex changes due to 3D rotation and translation. The initial robotic arm control was based on positional control that we observed that the movement of the robotic arm was losing the momentum between the waypoints due to that the duration of the task was increasing. The velocity control was used to tackle the issue with determining the desired duration of the trajectories between two points and thus the velocity of the robotic arm was controlled. However, the linear trajectory between the robotic arm and the target point was determined for positional control and it has to be adjusted the trajectory for velocity control where the robotic arm starts with low velocity, increases its velocity until mid waypoint, and decreases its velocity while reaching the target point to avoid the collision. Therefore, the linear trajectory changed to fit a spline configured using the Bezier curve. For testing purposes, we also implemented the visualization of waypoints using depth supervision to check whether the robotic arm position and the waypoint position matches during the execution.

References

[1] R. H. Bartels, J. C. Beatty, and B. A. Barsky, "An introduction to splines for use in computer graphics and geometric modelling," in Morgan Kaufmann, 1998, ch. 10 "Bézier Curves".

[2] A. A. Oliva, P. R. Giordano, and F. Chaumette, "A general visual-impedance framework for effectively combining vision and force sensing in feature space," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4441–4448, 2021. DOI: 10.1109/LRA.2021.3068911.

[3] A. Singh, "Modular tracking framework: A unified approach to registration based tracking," M.S. thesis, University of Alberta, 2018.

[4] H. Sutanto, R. Sharma, and V. Varma, "The role of exploratory movement in visual servoing without calibration," *Robotics and autonomous systems*, vol. 23, no. 3, pp. 153–169, 1998.

[5] L. Weiss, *Dynamic visual servo control of robots: An adaptive image-based approach, technical report*, 1984.

[6] N. Yilmaz, J. Y. Wu, P. Kazanzides, and U. Tumerdem, "Neural network based inverse dynamics identification and external force estimation on the da vinci research kit," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 1387–1393.