Review of Visual Servoing Fundamentals

Matthew Ebert;
ECE 490; Supervisor: Dr. Capson
dept. of Electrical and Computer Engineering
University of Victoria

Abstract –This report reviews the fundamentals of visual servoing (VS). VS is the direct application of computer vision within a control loop to generate a feedback signal. For mono-camera set ups, two configurations are possible for a VS system: eye-in-hand where the camera is mounted directly to the controlled object; eye-to-hand where the camera is mounted externally. Captured images with digital cameras create certain errors when the 3D scene is focused through a lens and projected onto a 2D sensor. Camera calibration can account for this error but must be completed for devices individually. Two primary methods of VS are prevalent in literature and practice. Image based visual servoing using the 2D image plane to create a control feature without needing to transform the pixel coordinates to real units. This method has shown robustness but can have slower convergence rates due to the computation of Jacobian matrices and non-linear control paths. Position based visual servoing calculates 3D poses of objects and the camera with respect to a reference frame. Once the poses have been calculated conventional control techniques can handle to rest of the system. This method is prone to errors due to its high dependence on camera calibration but allows for more direct control paths. Two implementations of VS were reviewed in this report. A UAM drone found an IBVS to be an optimal solution to airborne manipulation. An industrial assembly robot found PBVS to have the fastest convergence time and greatest reliability. The field of VS is expanding rapidly with the improvements to computer hardware and vision techniques. With the broad application range of VS, many systems may start to integrate this technology.

Key words –visual servoing, camera, pose, control theory, computer vision

I. INTRODUCTION

Visual servoing (VS) is the practice of using visual feedback within control loops to guide and position robotics or other controllable devices. A typical system involves one or more cameras to identify visual features in an environment and calculate an error function from the current and desired state of these features.

A. Background

With current image capture technology, small and high-quality cameras can be implemented in a wide variety of scenarios. This ability gives VS a broad scope of application. Some current applications of VS include autonomous vehicles, drones, and industrial production [1]. The limits on these systems come from two primary factors: computer hardware requirements and image scene limitations.

Computer Vision (CV) involves very computational costly operations such as neural networks and convolution. To achieve the speeds needed for a control system, the image analysis portion of the process must be simple or powerful computer hardware must be available. Additionally, the image factors that plague computer vision including occlusion, distortion, and background noise have a large impact on the object detection and tracking required for VS. These factors are mitigated by either robust CV techniques, which often come with processing or memory costs, or by altering the environment itself.

A distinction should be noted between VS and so called 'dynamic look and move' (DLM) systems. DLM systems use vision as an external input to a control loop for stability and offline* correction. DLM does not require online CV. VS directly integrates with the control loop in the feedback system and thus must operate in an online capacity.

B. Visual Servoing Process

VS follows a similar control scheme to conventional control systems. For closed loop VS, first a desired position/state is defined and fed to the system. Visual sensors (cameras) record the current state of the system. The images are processed to detect visual feature which are combined to create a set of control parameters. These parameters are mathematically compared with the desired parameters to produce an error signal. A control signal is generated from the error signal which is fed to the actuators of the system. The system responds and the camera's record this response and repeats the cycle until the target positioned has been reached.

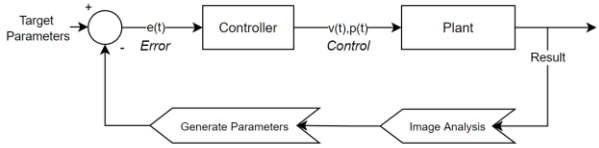

Figure 1: Example VS control loop.

Within VS systems, two configurations are defined regarding camera positioning (Figure 2).

1. **Eye-in-hand**: The camera is attached to the controlled object, such as a robotic arm, and is itself moved towards the target as the system responds to control input.
2. **Eye-to-hand**: The camera is fixed externally to the system and observes both the end effector and the target

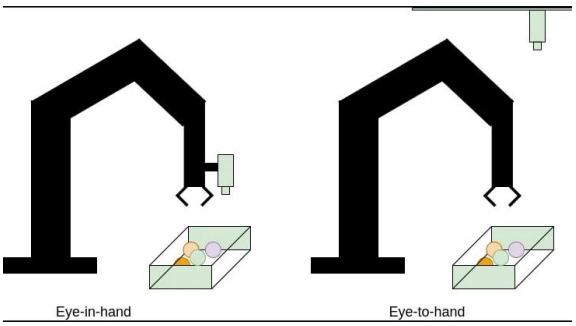

Figure 2: Illustration of eye-in-hand and eye-to-hand VS configurations [2].

The eye-in-hand method does not require the system itself to be in view. Consequently, the camera may sometimes be abstracted as a 'floating' element to make the design of the system more intuitive. However, the motion

*Online process: an online process refers to a calculation or control signal which is computed and used during operation. An offline process occurs outside operation time or is not essential within the control loop.

of the camera during control can create inaccuracies within the visual system which might decrease the performance [3]. The eye-to-hand system does not suffer from errors due to motion but does require a clear view of both the systems end effector and the target location. This constraint can be limiting in some applications. While neither configuration can be deemed superior, different applications may be better suited to one or the other.

Two primary distinct types of VS have been implemented and discussed in literature. These are defined as Image based visual servoing (IBVS) and position/pose based visual servoing (PBVS). Both will be discussed in later sections.

II. Camera Calibration

Modern cameras and optics are key to the accuracy of VS systems. Regardless of the method employed, certain knowledge from the 3D world is required. However, to capture a digital image, certain errors necessarily occur (based on implemented camera technology) when projecting a 3D world into a 2D image. These projection errors are not necessarily uniform between two different devices. Thus, cameras must be calibrated individually to find their specific correction parameters.

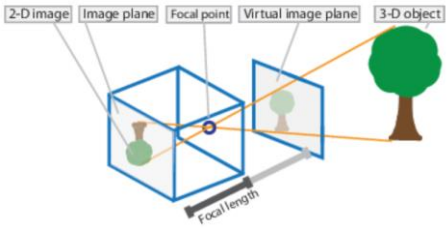

Figure 3: Pinhole camera diagram for capturing an image [4].

Camera calibration involves two sets of camera parameters: extrinsic and intrinsic. We may combine these parameters into transformation matrices to convert from different coordinate frames (Figure 4).

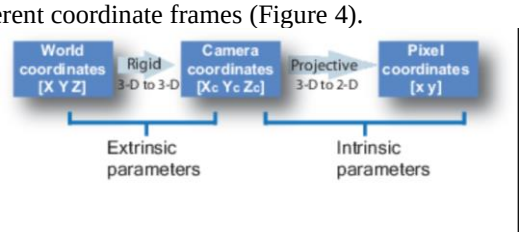

Figure 4: Image transformation from world coordinates to pixel coordinates [4].

This transformation takes the form

$$w\,[x\ y\ 1] = [X\ Y\ Z\ 1]P,$$

where

$$P = \begin{bmatrix} R \\ t \end{bmatrix} K.$$

X, Y, and Z are the world points, x and y are the image points, w is the scale factor, and P is defined as the camera matrix [4].

A. Extrinsic Paramters

The extrinsic parameters account for the camera's rotation and translation with respect to the image. The extrinsic parameter transformation matrix can convert between the world coordinates of an object and the corresponding camera coordinates. The rotation and translation portion of the matrix are preformed around the optical center (principal point) of the camera.

Figure 5: Extrinsic camera calibration example. Transformation matrix has rotation and translation components [4].

Extrinsic transformation can be used to find absolute distances and orientations of objects such as a robot end effector or work piece.

B. Intrinsic Parameters

Intrinsic camera calibration seeks to convert between the pixel units of the camera and real-world units such as meters on the image plane. Intrinsic calibration depends exclusively on camera optic properties such as focal length, the optical center, and skew. The intrinsic transformation matrix, K, takes the form

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

The coefficients are defined in Figure 6.

$[c_x \quad c_y]$ – Optical center (the principal point), in pixels.

$(f_x, \quad f_y)$ – Focal length in pixels.
$f_x = F/p_x$
$f_y = F/p_y$
F – Focal length in world units, typically expressed in millimeters.
$(p_x, \quad p_y)$ – Size of the pixel in world units.

s – Skew coefficient, which is non-zero if the image axes are not perpendicular.
$s = f_x \tan \alpha$

Figure 6: Intrinsic camera coefficient definitions [4].

C. Lens Distortion

Additional camera calibration is required for lens distortion (Figure 7). An ideal pin hole camera does not have a lens, but most real cameras do. Both radial and tangential distortion can appear in the image. Radial distortion is caused by the shape of the lens and tangential distortion occurs when the lens and camera sensor are not collinear.

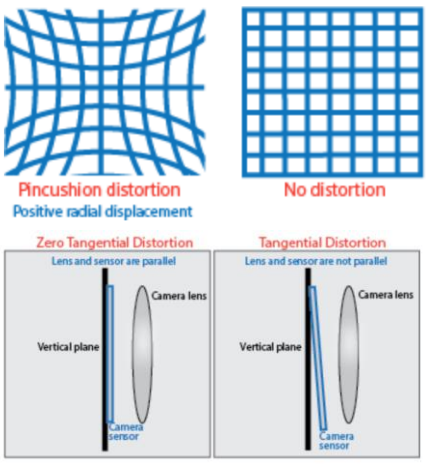

Figure 7: Positive radial image distortion (bottom left) and tangential distortion (bottom right) [4].

To correct for this type of distortion, normalized image coordinates are multiplied with weighted lens and tangential distortion coefficients.

For radial distortion, the correction equation takes the form

$$x_{\text{distorted}} = x(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6)$$

$$y_{\text{distorted}} = y(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6),$$

where the k values are additional intrinsic camera parameters and

$$r^2 = x^2 + y^2.$$

The variable x and y are normalized image coordinates having no units.

For tangential distortion, the correction equation takes the form using the same values of x, y, and r

$$x_{\text{distorted}} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{\text{distorted}} = y + [p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x * y]$$

The p values are intrinsic camera parameters [4].

For an example of the camera calibration process, See Appendix A.

III. Visual Servoing

The aim of a VS task is to minimize error between a set of parameters **(s**(m(t)**, a))** and the desired set of parameters **(s*)** [5] with the equation

$$\boldsymbol{e}(t) = \boldsymbol{s}(\boldsymbol{m}(t), \boldsymbol{a}) - \boldsymbol{s}^*.$$

The parameters of **s** vary depending on methodology and will be discussed later in this section. A velocity controller can be easily developed using the transformation Jacobian of these parameters of the form

$$\dot{\boldsymbol{s}} = \boldsymbol{L}_e \boldsymbol{v}_c$$

where:

- $\dot{\boldsymbol{s}}$ is the time varying parameters
- $\boldsymbol{L}_e$ is the feature Jacobian
- $\boldsymbol{v}_c$ is the angular (ω) and linear velocity (v) of the camera or object of interest

Substituting this value into the error function we find

$$\dot{\boldsymbol{e}} = \boldsymbol{L}_e \boldsymbol{v}_c$$

This equation alters based on the two perspective configurations of VS systems: eye-in-hand and eye-to-hand

In actuality, the difference in mathematics and coordinate transformation is minimal for both perspectives. The only differences appear in the implementation [5].

By finding the inverse of the feature Jacobian (if it exists) or an approximation, we may define a proportional velocity controller as a function of the error, a proportional gain, and inverse Jacobian [5].

$$\boldsymbol{v}_c = -\lambda \boldsymbol{L}_e^{-1} \boldsymbol{e} \quad \textbf{or} \quad \boldsymbol{v}_c = -\lambda \widehat{\boldsymbol{L}^+}_e \boldsymbol{e},$$

$\widehat{\boldsymbol{L}^+}_e$ is the approximation of the pseudo inverse of Le.

It is worth noting that if the number of features in **s (**i.e. the size of s) equals 6, the feature Jacobian (L) is invertible [5].

Regarding the parameter set and feature Jacobian, these values vary based on the type of servoing (IBVS/PBVS).

A. Image Based Visual Servoing

IBVS seeks a control method which resides in the 2D image plane where no transformation to real world units is necessary. This means the parameters of interest are the pixel coordinates of objects and points define by the cameras coordinate frame. Thus, we define **s** as

$$\boldsymbol{s} = \boldsymbol{x} = [x, y],$$

where

$$\begin{cases} x = X/Z = (u - c_u)/f\alpha \\ y = Y/Z = (v - c_v)/f, \end{cases}$$

The values of u and v are the pixel coordinates of the point of interest (PoI); c, f, and α are the intrinsic camera parameters [5]. The derivative of the 2D image coordinates gives

$$\begin{cases} \dot{x} = -v_x/Z + xv_z/Z + xy\omega_x - (1 + x^2)\omega_y + y\omega_z \\ \dot{y} = -v_y/Z + yv_z/Z + (1 + y^2)\omega_x - xy\omega_y - x\omega_z \end{cases}$$

[5]

where the v's and ω's are the relative 3D spatial linear and angular velocities respectively.

The feature Jacobian may be developed as

$$\mathbf{L}_{\mathbf{x}} = \begin{bmatrix} \frac{-1}{Z} & 0 & \frac{x}{Z} & xy & -(1 + x^2) & y \\ 0 & \frac{-1}{Z} & \frac{y}{Z} & 1 + y^2 & -xy & -x \end{bmatrix}$$

[5]

Inserting these equations into the error equation gives us our IBVS control function

$$\dot{\boldsymbol{x}} = \boldsymbol{L}_x \boldsymbol{v}_c$$

and

$$\mathbf{v}_c = -\lambda \mathbf{L}_{e^*}^{+} (\mathbf{s} - \mathbf{s}^*)$$

B. *Positioned Based Visual Servoing*

PBVS uses calculated poses of objects and the camera to generate a control function based on 3D coordinates. In this case, **s** is comprised of pose data of either the camera or an object relative to the chosen reference frame.

Chaumette et al. develops this definition for an eye-in-hand system to produce the following control function [5].

$$\mathbf{v}_c = -\lambda \widehat{\mathbf{L}_\mathbf{e}^{-1}} \mathbf{e},$$

$$\widehat{\mathbf{L}_\mathbf{e}^{-1}} = \begin{bmatrix} -\mathbf{I}_3 & [{}^c\mathbf{t}_o]_\times \mathbf{L}_{\theta\mathbf{u}}^{-1} \\ \mathbf{0} & \mathbf{L}_{\theta\mathbf{u}}^{-1} \end{bmatrix},$$

$$\begin{cases} \boldsymbol{v}_c &= -\lambda(({}^{c^*}\mathbf{t}_o - {}^c\mathbf{t}_o) + [{}^c\mathbf{t}_o]_\times \theta\mathbf{u}) \\ \boldsymbol{\omega}_c &= -\lambda\theta\mathbf{u}, \end{cases}$$

In this scheme, s is defined as

$$\mathbf{s} = ({}^c\mathbf{t}_o, \theta\mathbf{u})$$

$\boldsymbol{{}^c t_0}$ is the coordinate vector of the reference frame relative to the camera frame and $\boldsymbol{\theta u}$ gives the angle parameterization for rotation.

1) *Pose Estimation*

PBVS require accurate pose estimation of object to convert from pixel data to meaningful position information.

The pose of an object defines it position and orientation to a reference frame. In a 3D environment, poses contain 6 parameters which define translation and rotation. With a calibrated camera, we may obtain the pose from analyzing key points or comparing template models on the image plane. Both mono and stereo camera systems can determine the pose of an object. Stereo vision has the advantage of multiple view angles to help calculating distance and requires fewer points. Mono vision pose estimation will be considered for the remainder of the report.

For determining object pose from a single RGB image, two methods are prevalent in literature: Key point and CNN based approaches [6] [7] [8]. Both these approaches consist of two stages: detecting an object with key points or models, then solving a Perspective-n-Point (PnP) problem. PnP refers to solving a 3D objects position and orientation from a set of 2D points. Solving the PnP problem involves minimizing certain function to infer the objects pose. To avoid ambiguous results with a mono camera system, at least 4 key points are required to fully define the objects position in 3D space [9] [5].

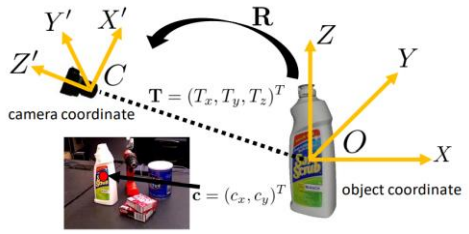

Figure 8: Diagram representing the conversion between object and camera coordinate systems [8].

If the exact 3D model of the object is available, we may use existing PnP algorithms to solve the pose of an object. Occlusion and false detection can cause inaccuracies with this method because the key point prediction lacks robustness with these factors [7].

A defining difference between PBVS and IBVS is found in the way the sensor is treated. PBVS treats the camera as 3D sensor, while IBVS treats it as 2D [5]. The advantage of PBVS is the ability to compare the size of objects to known models and compute direct paths for error correction. This has been shown to increase accuracy and convergence of the system [10]. However, PBVS requires a highly accurate estimation of camera parameters and the desired objectives.

One other issue appears using PBVS regarding the camera frame. No limits within the control system restrict the camera to keep the target object in frame. During motion, there is the potential for the camera or system to leave the frame and prevent further tracking [5]. IBVS work directly within the image plane which effectively removes this danger.

IBVS requires no calibration. Each instance of PBVS must be specifically calibrated for the camera in use. IBVS has shown superior robustness due to it slightly reduced dependence on accurate parameter estimation (in this case, the distance from the image plane Z). However, projecting a 3D environment into a 2D plane prevents a direct trajectory for error correction. Additionally, IBVS requires the calculation of the inverse Jacobian, which has a high processing cost and can increase convergence times [10]. With the abstraction of the environment in the image plane, IBVS is highly non-linear. This can cause problems within control schemes including converging to local minimums and not global minimums regarding the error function. Problems may also occur when mapping the control function to actuators especially at crossing points which are known to produce singularities [11].

Several sources have shown preference to IBVS over PBVS on grounds of superior robustness and simplicity. However, this has not been conclusively demonstrated [10].

IV. Example Applications of Visual Servo

The following section reviews and summarizes the finding of several VS systems.

A. *Image Dynamics-Based Visual Servo Control for Unmanned Aerial Manipulator With a Virtual Camera [3]*

In their report, Lai N. et al., propose and test a visual servo system for an unmanned aerial vehicle. This vehicle consists of a UAV (drone) and a grabber like manipulator. The system they proposed uses computer vision to locate, approach and interact with a target object. For the VS system they conclude a IBVS eye-to-hand system will produce the best results for the following reasons

- IBVS does not depend on camera calibration and accurate conversion from image space to real space
- Eye-to-hand configuration could introduce additional error due to the disturbance from the motion of the UAV portion and manipulator portion.
- Testing indicated that image-based VS showed favorable performance to other approaches

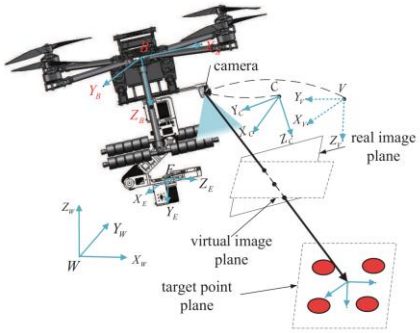

Figure 9: VS system diagram

Lai et al. used a static target object as the reference point for the UAM IBVS control. Using the dots as parameters, they defined the desired UAM image pose and created an error function based on the current pose. All calculation were completed in the image plane without any need to convert to physical units.

They implemented a 'sliding mode control' function to deal with the dynamics of the non-linear system.
The manipulator was also controlled with IBVS. The depth of the arm was estimated using forward kinematics for use in the feature Jacobian which allowed for accurate position control. The model was simulated and tested in a constrained environment to produce satisfactory results.

B. Comparing Position- and Image-Based Visual Servoing for Robotic Assembly of Large Structures [10]

Peng et al. use both PBVS and IBVS with an industrial assembly robot (Figure 3) [10].

The task of the system was to detect a target panel and align it according to a desired state. The authors found that in their case PBVS proved more accurate and faster than the IBVS model. These results differ from other literature. They attribute this discrepancy to the IBVS inferring the desired image from the target image. This inference required the model to use camera parameters, creating a dependency which IBVS usually avoids. They predict that if the desired image is known a priori, IBVS will produce superior results.

However, in their case, IBVS and PBVS were comparable.

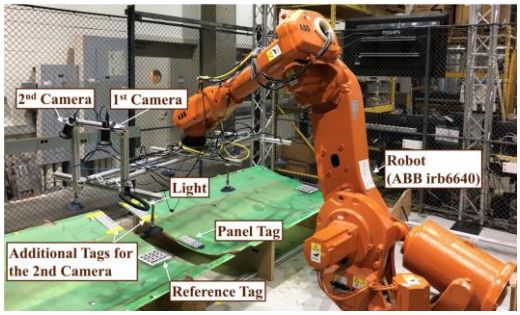

Figure 10: Test set up for the robot assembly station used to test PBVS and IBVS control [10].

V. CONCLUSION

Visual servoing provides a control method with an incredibly versatile scope. Given an appropriate camera and sufficient computing power, VS can provide precise control in a variable environment where conventional sensor fails to fully capture the system and its state.

The camera itself requires certain software corrections to cope with errors cause by projection from a 3D environment to a 2D sensor and distortion from the camera's optics. These corrections vary for each individual device and are the result of some significant errors within VS systems.

Image based visual servoing and position/pose based visual servoing comprise the main methods currently used in the field of VS. IBVS uses the 2D image plane to measure the error between a current image and the desired image. The error signal is produced entirely with pixel data without the need to convert to real units. The pixel values are transformed into a control signal via a feature Jacobian. Calculating the Jacobian requires the object planes distance from the lens to be known. If this value varies the cost of calculating the Jacobian can become large. PBVS uses image analysis to compute poses and reference frames for objects and key features. Once the location is known, direct control can be applied from the positional and velocity data of the system. PBVS depends heavily on the intrinsic and extrinsic camera parameters, so any error inherent in these sets carries to the control system. Thus, PBVS is consider less robust than IBVS in most cases.

Some effective VS system include a UAM (unmanned arial manipulator) and an industrial assembly robot [3] [10]. The arial drone showed the best performance with an IBVS eye-to-hand system which used a static target plane to orient itself. The industrial robot acquired the best performance with a stereo PBVS system which oriented labeled panels to a desired location.

Several examples have demonstrated the accuracy and viability of VS systems and they are becoming more numerous with the development of autonomous vehicles and systems. While VS systems suffer the problems of computer vision including processing power, object detection, tracking, and distortion, the ever-increasing performance of computer hardware and CV techniques may allow VS to be a commonly integrated system in the field of automation and control.

VI. REFERENCES

[1] M. N. Favorskaya and L. C. Jain, Computer Vision in Control Systems-4: Real Life Applications, Springer, 2015.

[2] A. Kumar, "Control Automation," 15 May 2020. [Online]. Available: https://control.com/technical-articles/an-overview-of-visual-servoing-for-robot-manipulators/. [Accessed 19 July 2022].

[3] N. Lai, Y. Chen, J. Liang and e. al, "Image Dynamics-Based Visual Servo Control or Unmanned Aerial Manipulatorl With a Virtual Camera," *IEEE Robotics & Automation Society,* 2022.

[4] MathWorks, "Camera Calibration," MathWorks, 2022. [Online]. Available: https://www.mathworks.com/help/vision/ug/camera-calibration.html. [Accessed q12 Jul 2022].

[5] F. Chaumette and S. Hutchinson, "Visual Servo Control Part 1: Basic Approaches," *IEEE Robotics & Automation Magazine,* pp. 82-90, December 2006.

[6] Z. Moritz, B. Simon and B. Sven, "6D Object Pose Estimation using Keypoints and Part Affinity Fields," in *24th RoboCup International Symposium*, University of Bonn, Germany, 2021.

[7] G. Pavlakos, X. Zhou, A. Chan, K. Derpanis and K. Daniilidis, "6-DoF object from Semantic Keypoints," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[8] Y. Xiang, T. Schmidt and e. al., "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes," *In: Robotics: Science and Systems (RSS),* 2018.

[9] Visual Servoing Platform, "Pose estimation from points," 12 Jul 2022. [Online]. Available: https://visp-doc.inria.fr/doxygen/visp-daily/tutorial-pose-estimation.html#tutorial_pose_est_image. [Accessed 12 Jul 2022].

[10] Y. Peng, D. Jivani and R. Radke, "Comparing Position- and Image-Based Visual Servoing for Robotic Assembly of Large Structures," Department of Electrical, Computer, and Systems Engineering,Rensselaer Polytechnic Institute, New York.

[11] G. Palmieri, M. Palpacelli and M. Battistelli, "A Comparison between Position-Based and Image-Based Dynamic Visual Servoings in the Control of a Translating Parallel Manipulator," *Journal of Robotics,* vol. 2012, p. 11, 2012.

[12] OpenCV team, "OpenCV," OpenCV team, 2022. [Online]. Available: https://opencv.org/. [Accessed 13 Jul 2022].

[13] Visual Servoing Platform , "Visual Servoing Platform," 13 Jul 2022. [Online]. Available: https://visp-doc.inria.fr/doxygen/visp-daily/index.html. [Accessed 13 Jul 2022].

APPENDIX A: LAB 2 – CAMERA CALIBRATION, OBJECT POSE ESTIMATION, AND VISUAL SERVO SIMULATION

A. *Introduction*

This laboratory seeks to implement certain visual servo techniques. Two open-source libraries were used for this purpose.

1. OpenCV: An open-source computer vision library [12].
2. VISP (Visual Servoing Platform): An open-source VS library [13].

B. *Apparatus*

All software tasks were completed in the C++ language. The CMake tool was used to build, compile, and run the created programs. A Logitech C920 HD camera was used for all image input. The camera was secured on a tower to be horizontal with a black tabletop.

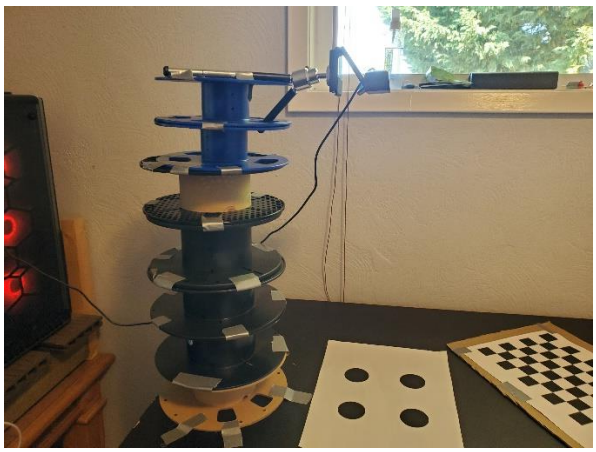

Figure 11: Camera apparatus for computing pose of 4Dot object and tracking the golf ball

C. *Camera Calibration*

Intrinsic camera calibration is the process of estimating the physical camera parameters including focal length, skew, distortion, and the optical center of the camera. The VISP library has an example project for basic camera calibration. The example uses OpenCV tools in tandem with a chessboard to derive the camera parameters.

1) *Procedure*

The Logitech C920 HD camera, which was used for all image recording in this laboratory, was connected to a PC. It was placed to view a well-lit area. The camera was calibrated with the following steps on the PC.

1. Take several photos of the chessboard piece. Ensure the images capture the chessboard at different angles and at different locations on the screen. Try to place the chessboard near the edges of the camera frame in some of the images to help determine distortion.
2. Save the images to the same folder as the VISP camera calibration example executable.
3. Run the VISP camera calibration executable.
4. Save the calculated camera parameters in an .xml file.

The VISP program uses the following equations to convert between meters and pixels as well as correct for distortion.

(u, v) are pixel coordinates which correspond to (x, y) in normalized space.

Conversion	**Equations**
Pixels to Meters	$x = (u - u_0) * (1 + k_{du} r^2)/p_x$ $y = (v - v_0) * (1 + k_{du} r^2)/p_y$ $r^2 = ((u - u_0)/p_x)^2 + ((v - v_0)/p_y)^2$
Meters to Pixels	$u = u_0 + x p_x (1 + k_{ud} r^2)$ $v = v_0 + y p_y (1 + k_{ud} r^2)$ $r^2 = x^2 + y^2$
Constants	• (u_0, v_0) are the coordinates of the principal point in pixel; • (p_x, p_y) are the ratio between the focal length and the size of a pixel; • (k_{ud}, k_{du}) are the parameters used to correct the distortion. k_{ud} is the distortion

2) Results

Following the steps above, the camera parameters were successfully generated. First, 10 images were taken of the chessboard.

Figure 12: Sample calibration image with chessboard

Then, the calibration executable was run. The program iterated through the supplied images and located the chessboard features (square intersections).

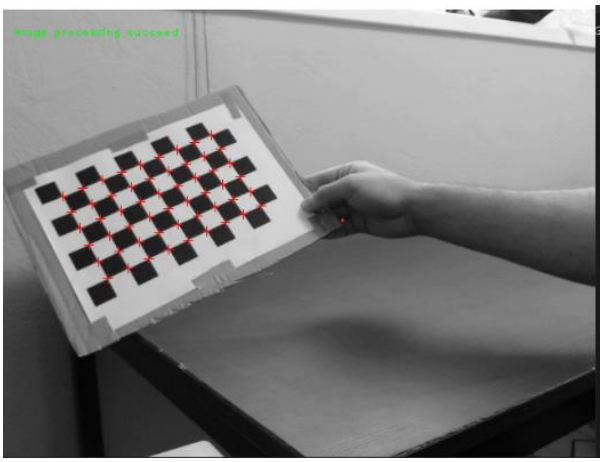

Figure 13: Chessboard feature points; Generated by VISP.

All the sample images were compiled together to make a camera profile.

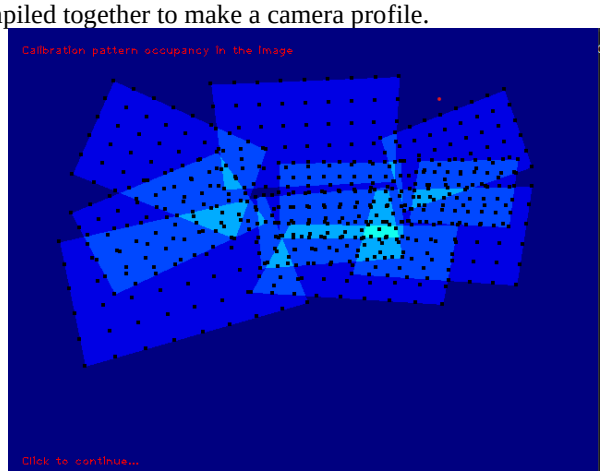

Figure 14: Camera calibration pattern; Generated by VISP.

Next, the known chessboard model was used to project the 'actual' image on to the virtual image plane.

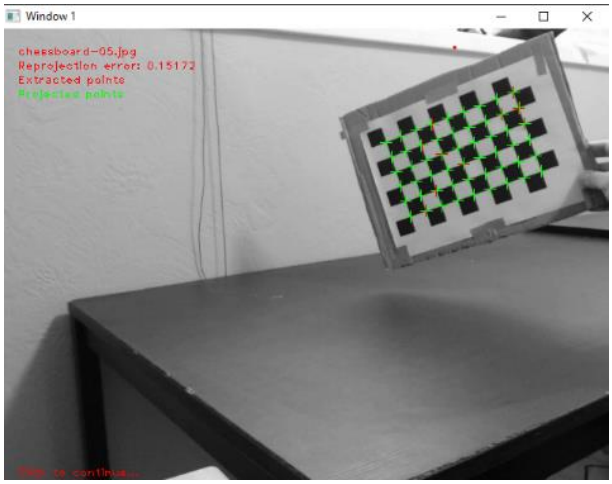

Figure 15: The projected points on the chessboard model onto the image plane. Generated by VISP.

Finally, distortion was corrected for using the lines on the chessboard. In the case of the Logitech camera, almost no distortion is present. Consequently, the correction is minimal or insignificant.

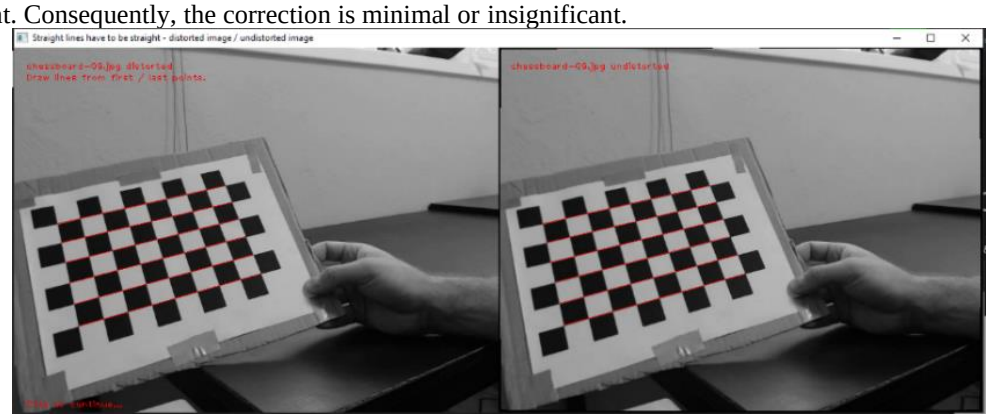

Figure 16: Undistorting lines on the chessboard. Generated by VISP.

The programed outputted the camera parameters to an .xml final for future use. The output consists of 2 data sets: one without distortion correction (ideal pinhole camera) and one with. For our camera, the differences between these two data sets is minimal since the Logitech camera does not have a large lens.

```
<name>Camera</name>
<!--Size of the image on which camera calibration was performed-->
<image_width>640</image_width>
<image_height>480</image_height>
<!--Intrinsic camera parameters computed for each projection model-->
<model>
  <!--Projection model type-->
  <type>perspectiveProjWithoutDistortion</type>
  <!--Pixel ratio-->
  <px>622.69564730044863</px>
  <py>619.30910237045339</py>
  <!--Principal point-->
  <u0>315.63755332018906</u0>
  <v0>234.14308871478232</v0>
</model>
<model>
  <!--Projection model type-->
  <type>perspectiveProjWithDistortion</type>
  <!--Pixel ratio-->
  <px>639.42973823391799</px>
  <py>638.4307649315756</py>
  <!--Principal point-->
  <u0>313.56806909633349</u0>
  <v0>230.45109392563688</v0>
  <!--Undistorted to distorted distortion parameter-->
  <kud>0.032706156952937791</kud>
  <!--Distorted to undistorted distortion parameter-->
  <kdu>-0.032839204349518696</kdu>
```

Figure 17: Intrinsic camera parameters of Logitech camera. Generated by VISP.

D. Object Pose Estimation

Using the calibrated camera parameters, object pose estimation was performed on a sample image.

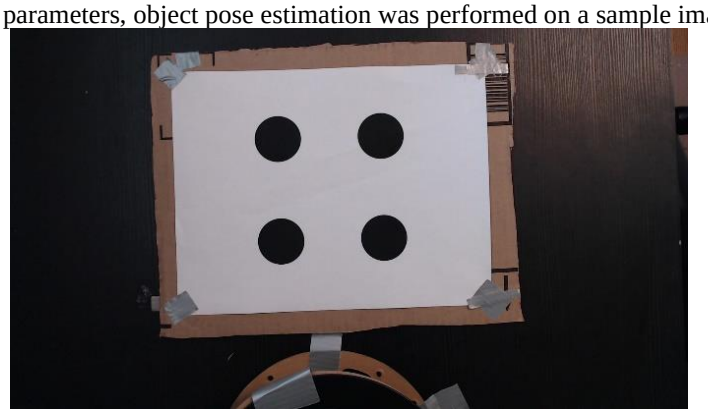

Figure 18: Sample 4-dot image used for pose estimation.

An example function from VISP was modified to find the pose of the sample image.

The program has 3 parts

1. Initialization
2. Tracking
3. Pose Computation

The initialization process involves selecting the key points on the object (in this case the 4 circles). These are selected with mouse clicks and binary blob detection. Once all the blobs have been initialized, the software maintains the location of the dots by using a simple blob tracking method. Next, the pose of the object is calculated. The pixel coordinate are converted into meters with the camera parameters then the PnP problem solved from the known model of the object. The results of the program are shown below.

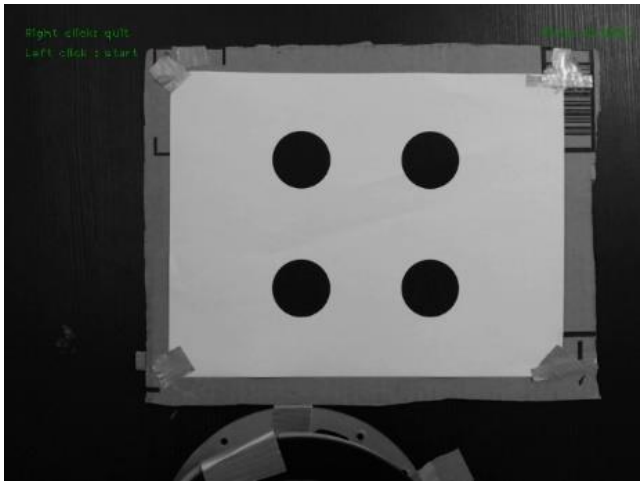

Figure 19: Initializing the program

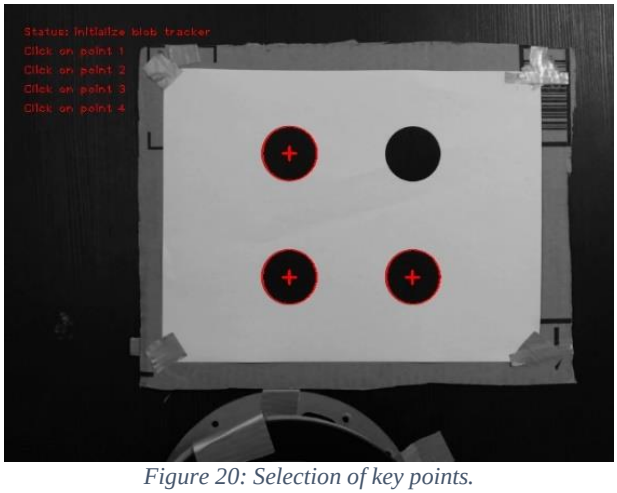

Figure 20: Selection of key points.

The rotation and translation of the object are measure with respect to the camera's reference frame. The orders of the components are

Translation: $(x, y, z) \rightarrow (Green, Red, Blue)$
Rotation tu: $(\theta_z, \theta_y, \theta_x) \rightarrow (Blue, Red, Green)$

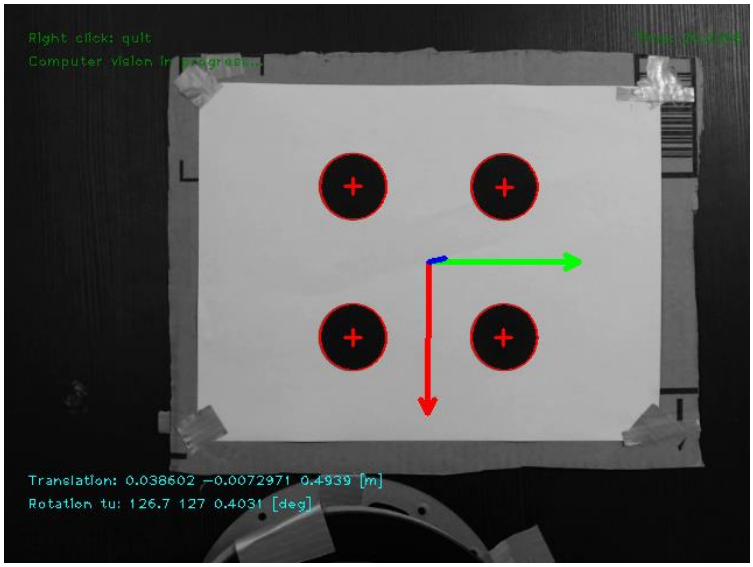

Figure 21: First pose estimation.

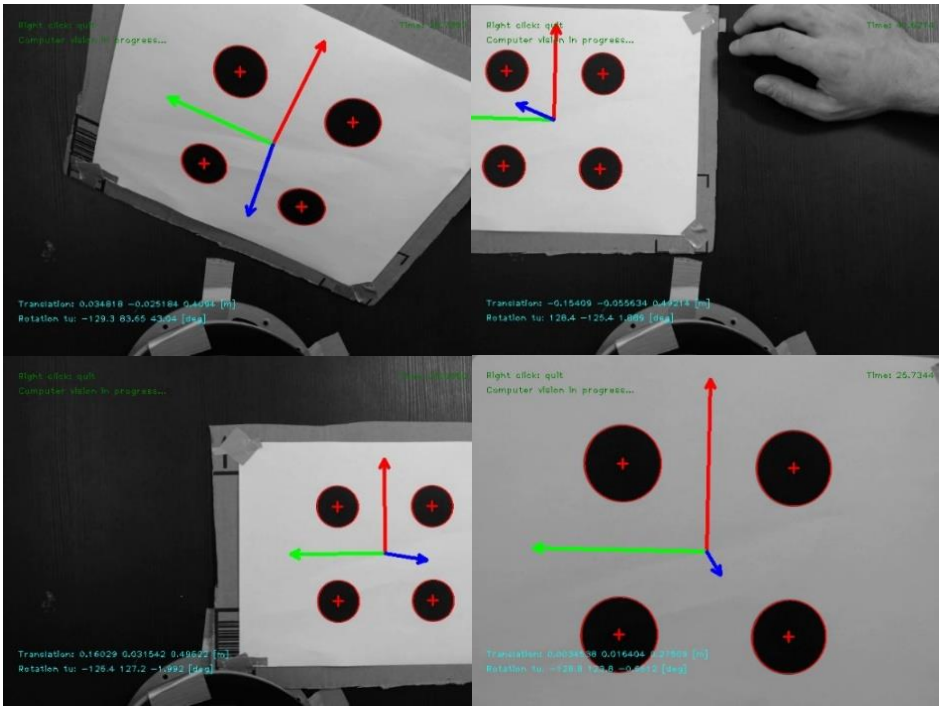

Figure 22: Pose estimation of 4Dot object

The code for this program is listed below.

```
//This function estimates the pose of the 4 Dot object
//Created by: ViSP
//Modified by: Matthew Ebert   Date: 2022-07-10
#include "pose_helper.h"
#include "main.hpp"

int poseestimator()
{

  try
```

```
{
  //File locations and settings
  std::string opt_intrinsic_file = "F:/DEV/VISP_Testing/include/camera.xml";
  std::string opt_camera_name = "Camera";
  double opt_square_width = 0.1;
  int opt_device = 0;

  vpImage<unsigned char> I;
 // cv::Mat text;

  // Load the parameters of our camera
  vpCameraParameters cam(840, 840, I.getWidth() / 2, I.getHeight() / 2); // Default parameters
  vpXmlParserCamera parser;
  if (!opt_intrinsic_file.empty() && !opt_camera_name.empty())
  {
    std::cout << "Intrinsic file: " << opt_intrinsic_file << std::endl;
    std::cout << "Camera name   : " << opt_camera_name << std::endl;
    if (parser.parse(cam, opt_intrinsic_file, opt_camera_name, vpCameraParameters::perspectiveProjWithDistortion) ==
        vpXmlParserCamera::SEQUENCE_OK)
    {
      std::cout << "Succeed to read camera parameters from xml file" << std::endl;
    }
    else
    {
      std::cout << "Unable to read camera parameters from xml file" << std::endl;
    }
  }

  std::cout << "Use OpenCV grabber on device " << opt_device << std::endl;
  cv::VideoCapture g(opt_device); // Open the default camera
  if (!g.isOpened())
  { // Check if we succeeded
    std::cout << "Failed to open the camera" << std::endl;
    return -1;
  }
  cv::Mat frame;
  g >> frame; // get a new frame from camera
  vpImageConvert::convert(frame, I);

  std::cout << "Square width  : " << opt_square_width << std::endl;
  std::cout << cam << std::endl;

  // The pose container
  vpHomogeneousMatrix cMo;

  std::vector<vpDot2> dot(4);
  std::vector<vpPoint> point;   // 3D coordinates of the points
  std::vector<vpImagePoint> ip; // 2D coordinates of the points in pixels
  double L = opt_square_width / 2.;
  point.push_back(vpPoint(-L, -L, 0));
  point.push_back(vpPoint(L, -L, 0));
  point.push_back(vpPoint(L, L, 0));
  point.push_back(vpPoint(-L, L, 0));
  vpDisplayOpenCV d(I);
  bool quit = false;
  bool apply_cv = false; // apply computer vision
  bool init_cv = true;   // initialize tracking and pose computation

  while (!quit)
  {
    double t_begin = vpTime::measureTimeMs();
    // Image Acquisition
    g >> frame;
    vpImageConvert::convert(frame, I);

    vpDisplay::display(I);
    if (apply_cv)
    {
      try
      {
        ip = track(I, dot, init_cv); //Track the dots on the object
        computePose(point, ip, cam, init_cv, cMo); //call compute pose function
        vpDisplay::displayFrame(I, cMo, cam, opt_square_width, vpColor::none, 3); //Show the capture image frame
        if (init_cv)
          init_cv = false; // turn off the computer vision initialisation specific stuff

        { // Display estimated pose in [m] and [deg]
          vpPoseVector pose(cMo);
          std::stringstream ss;
          ss << "Translation: " << std::setprecision(5) << pose[0] << " " << pose[1] << " " << pose[2] << " [m]";
          vpDisplay::displayText(I, 400, 20, ss.str(), vpColor::cyan);
          ss.str(""); // erase ss
          ss << "Rotation tu: " << std::setprecision(4) << vpMath::deg(pose[3]) << " " << vpMath::deg(pose[4]) << " "
             << vpMath::deg(pose[5]) << " [deg]";
          vpDisplay::displayText(I, 420, 20, ss.str(), vpColor::cyan);
          //cv::Point pos(100,100);
          //cv::putText(text, ss.str(), pos,cv::FONT_HERSHEY_PLAIN, 1, (0,255,0), 1, cv::LINE_AA);
          //cv::imshow("Pose", text);

        }
      }
      catch (...)
      {
        std::cout << "Computer vision failure." << std::endl;
        apply_cv = false;
        init_cv = true;
      }
    }
    vpDisplay::displayText(I, 20, 20, "Right click: quit", vpColor::darkGreen);
    if (apply_cv)
    {
```

```
        vpDisplay::displayText(I, 40, 20, "Computer vision in progress...", vpColor::darkGreen);
      }
      else
      {
        vpDisplay::displayText(I, 40, 20, "Left click : start", vpColor::darkGreen);
      }
      vpMouseButton::vpMouseButtonType button;
      if (vpDisplay::getClick(I, button, false))
      {
        if (button == vpMouseButton::button3)
        {
          quit = true;
        }
        else if (button == vpMouseButton::button1)
        {
          apply_cv = true;
        }
      }
      {
        std::stringstream ss;
        ss << "Time: " << vpTime::measureTimeMs() - t_begin << " ms";
        vpDisplay::displayText(I, 20, I.getWidth() - 100, ss.str(), vpColor::darkGreen);
      }
      vpDisplay::flush(I);
    }
  }
  catch (const vpException &e)
  {
    std::cout << "Catch an exception: " << e.getMessage() << std::endl;
  }

  return 0;
}
```

```
#include <visp3/core/vpDisplay.h>
#include <visp3/core/vpPixelMeterConversion.h>
#include <visp3/vision/vpPose.h>

#include "pose_helper.h"

void computePose(std::vector<vpPoint> &point, const std::vector<vpImagePoint> &ip, const vpCameraParameters &cam,
                 bool init, vpHomogeneousMatrix &cMo)
{
//Compute the pose of the 4 Dot object
  vpPose pose;
  double x = 0, y = 0;
  for (unsigned int i = 0; i < point.size(); i++) {
    vpPixelMeterConversion::convertPoint(cam, ip[i], x, y); // Convert the pixel points into metric values
    point[i].set_x(x);
    point[i].set_y(y);
    pose.addPoint(point[i]);
  }

  if (init == true) {// For the first pose calculation
    vpHomogeneousMatrix cMo_dem;
    vpHomogeneousMatrix cMo_lag;
    pose.computePose(vpPose::DEMENTHON, cMo_dem); // Use 2 PnP algorithms to compute the pose
    pose.computePose(vpPose::LAGRANGE, cMo_lag);  // using non initialized linear approaches
    double residual_dem = pose.computeResidual(cMo_dem);
    double residual_lag = pose.computeResidual(cMo_lag);
    if (residual_dem < residual_lag) // Take the better result
      cMo = cMo_dem;
    else
      cMo = cMo_lag;
  }
  pose.computePose(vpPose::VIRTUAL_VS, cMo); // Update the pose based on previous pose
}


std::vector<vpImagePoint> track(vpImage<unsigned char> &I, std::vector<vpDot2> &dot, bool init)
{
  try {
    double distance_same_blob = 10.; // 2 blobs are declared same if their distance is less than this value
    std::vector<vpImagePoint> ip(dot.size());
    if (init) {
     // Set up the blobs with mouse clicks and begin tracking
      vpDisplay::flush(I);
      for (unsigned int i = 0; i < dot.size(); i++) {
        dot[i].setGraphics(true);
        dot[i].setGraphicsThickness(2);
        std::stringstream ss;
        ss << "Click on point " << i + 1;
        vpDisplay::displayText(I, 20, 20, "Status: initialize blob tracker", vpColor::red);
        vpDisplay::displayText(I, 40 + i * 20, 20, ss.str(), vpColor::red);
        vpDisplay::flush(I);
        dot[i].initTracking(I);
        vpDisplay::flush(I);
      }
    } else {
      for (unsigned int i = 0; i < dot.size(); i++) {
        dot[i].track(I); // Track each dot using a blob tracker for the 4Dot object
      }
    }
    for (unsigned int i = 0; i < dot.size(); i++) {
      ip[i] = dot[i].getCog();
      // Compare distances between all the dots to check if some of them are not the same
    }
    for (unsigned int i = 0; i < ip.size(); i++) {
      for (unsigned int j = i + 1; j < ip.size(); j++) {
        if (vpImagePoint::distance(ip[i], ip[j]) < distance_same_blob) {
          std::cout << "Traking lost: 2 blobs are the same" << std::endl;
```

```
          throw(vpException(vpException::fatalError, "Tracking lost: 2 blobs are the same"));
        }
      }
    }

    return ip;
  } catch (...) {
    std::cout << "Traking lost" << std::endl;
    throw(vpException(vpException::fatalError, "Tracking lost"));
  }
}
```

E. IBVS Simulation

The VISP software has built in simulators which can graphically demonstrate VS systems. An example code provided by VISP was run and modified to experiment with IBVS control. The program is listed below.

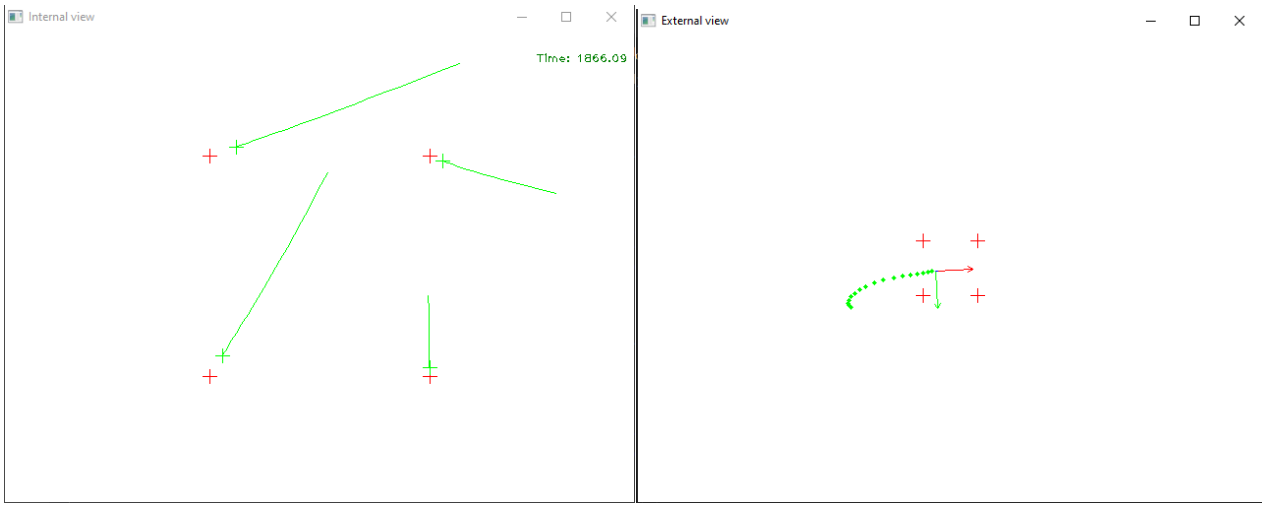

Figure 23: IBVS simulation output. Generated by VISP

The value of lambda was adjusted to create different responses of the simulation. The results follow.

Table 1: Settling time of IBVS simulation with proportional control

No.	Lambda	Settling Time (ms)
1	0.5	32859
2	1	18268.6
3	5	4783.62
4	30	2042.15
5	45	4864.16
6	49	-- (unstable)

The proportional control decreased settling time with an increasing lambda to a minimum settling time of 2042.2 ms at $\lambda = 30$. Past this point, oscillations extended the settling time until the system became unstable at $\lambda = 50$. The implemented code is shown below.

```
//This simulate a IBVS system in which an eye-in-hand camera is positioned to face a square face on.
//Created by: ViSP
//Modified by: Matthew Ebert   Date: 2022-07-12


#include "main.hpp"

void display_trajectory(const vpImage<unsigned char> &I, std::vector<vpPoint> &point, const vpHomogeneousMatrix &cMo,
                        const vpCameraParameters &cam);

void display_trajectory(const vpImage<unsigned char> &I, std::vector<vpPoint> &point, const vpHomogeneousMatrix &cMo,
                        const vpCameraParameters &cam)
{
  static std::vector<vpImagePoint> traj[4];
  vpImagePoint cog;
  for (unsigned int i = 0; i < 4; i++)
  {
    // Project the point at the given camera position
```

```
    point[i].project(cMo);
    vpMeterPixelConversion::convertPoint(cam, point[i].get_x(), point[i].get_y(), cog);
    traj[i].push_back(cog);
  }
  for (unsigned int i = 0; i < 4; i++)
  {
    for (unsigned int j = 1; j < traj[i].size(); j++)
    {
      vpDisplay::displayLine(I, traj[i][j - 1], traj[i][j], vpColor::green);
    }
  }
}

void IBVS()
{
  try
  {
    vpHomogeneousMatrix cdMo(0, 0, 0.75, 0, 0, 0);
    vpHomogeneousMatrix cMo(0.15, -0.1, 1., vpMath::rad(10), vpMath::rad(-10), vpMath::rad(50));

    std::vector<vpPoint> point;
    point.push_back(vpPoint(-0.1, -0.1, 0));
    point.push_back(vpPoint(0.1, -0.1, 0));
    point.push_back(vpPoint(0.1, 0.1, 0));
    point.push_back(vpPoint(-0.1, 0.1, 0));

    vpServo task;
    task.setServo(vpServo::EYEINHAND_CAMERA);
    task.setInteractionMatrixType(vpServo::CURRENT);
    task.setLambda(1); // SET PROPORTIONAL CONTROL TO CHANGE RATE OF CONVERGENCE

    vpFeaturePoint p[4], pd[4];
    for (unsigned int i = 0; i < 4; i++)
    {
      point[i].track(cdMo);
      vpFeatureBuilder::create(pd[i], point[i]);
      point[i].track(cMo);
      vpFeatureBuilder::create(p[i], point[i]);
      task.addFeature(p[i], pd[i]);
    }

    vpHomogeneousMatrix wMc, wMo;
    vpSimulatorCamera robot;
    robot.setSamplingTime(0.040);
    robot.getPosition(wMc);
    wMo = wMc * cMo;

    vpImage<unsigned char> Iint(480, 640, 255);
    vpImage<unsigned char> Iext(480, 640, 255);
#if defined(VISP_HAVE_X11)
    vpDisplayX displayInt(Iint, 0, 0, "Internal view");
    vpDisplayX displayExt(Iext, 670, 0, "External view");
#elif defined(VISP_HAVE_GDI)
    vpDisplayGDI displayInt(Iint, 0, 0, "Internal view");
    vpDisplayGDI displayExt(Iext, 670, 0, "External view");
#elif defined(VISP_HAVE_OPENCV)
    vpDisplayOpenCV displayInt(Iint, 0, 0, "Internal view");
    vpDisplayOpenCV displayExt(Iext, 670, 0, "External view");
#else
    std::cout << "No image viewer is available..." << std::endl;
#endif

#if defined(VISP_HAVE_DISPLAY)
    vpProjectionDisplay externalview;
    for (unsigned int i = 0; i < 4; i++)
      externalview.insert(point[i]);
#endif
    vpCameraParameters cam(840, 840, Iint.getWidth() / 2, Iint.getHeight() / 2);
    vpHomogeneousMatrix cextMo(0, 0, 3, 0, 0, 0);
    double t_begin = vpTime::measureTimeMs();
    while (1)
    {

      robot.getPosition(wMc);
      cMo = wMc.inverse() * wMo;
      for (unsigned int i = 0; i < 4; i++)
      {
        point[i].track(cMo);
        vpFeatureBuilder::create(p[i], point[i]);
        // std::cout << "point "<< i<< " " << point[i].get_x();
        // std::cout << "    p "<< i << " " << p[i].get_x() << std::endl;
      }
      vpColVector v = task.computeControlLaw();
      robot.setVelocity(vpRobot::CAMERA_FRAME, v);

      vpDisplay::display(Iint);
      vpDisplay::display(Iext);
      display_trajectory(Iint, point, cMo, cam);

      vpServoDisplay::display(task, cam, Iint, vpColor::green, vpColor::red);
#if defined(VISP_HAVE_DISPLAY)
      externalview.display(Iext, cextMo, cMo, cam, vpColor::red, true);
#endif
      {
        std::stringstream ss;
        ss << "Time: " << vpTime::measureTimeMs() - t_begin << " ms";
        vpDisplay::displayText(Iint, 20, Iint.getWidth() - 100, ss.str(), vpColor::darkGreen);
      }
      vpDisplay::flush(Iint);
      vpDisplay::flush(Iext);
```

```
      // A click to exit
      if (vpDisplay::getClick(Iint, false) || vpDisplay::getClick(Iext, false) ||
            (v[0] < 0.0005 && v[1] < 0.0005 && v[2] < 0.0005 && v[3] < 0.0005 && v[4] < 0.0005))
      {

        std::cout << "Done \n"
                  << "Total time: " << vpTime::measureTimeMs() - t_begin;
        break;
      }

      vpTime::wait(robot.getSamplingTime() * 1000);
    }
  }
  catch (const vpException &e)
  {
    std::cout << "Catch an exception: " << e << std::endl;
  }
}
```

F. Golf Ball Tracking and Relative Measurements

Finally, a program was written to calculate the image and world coordinates of a golf ball. A simple mathematical formula using ratios was implemented to approximate the golf balls distance from the camera lens. The method uses the pixel radius of the golf ball at the table and the distance from the table to the lens to approximate the golf ball height with

$$Z_{camera} = \frac{R_{table}}{R} * Z_{table},$$

where R is the current radius of the ball. The images below show the results.

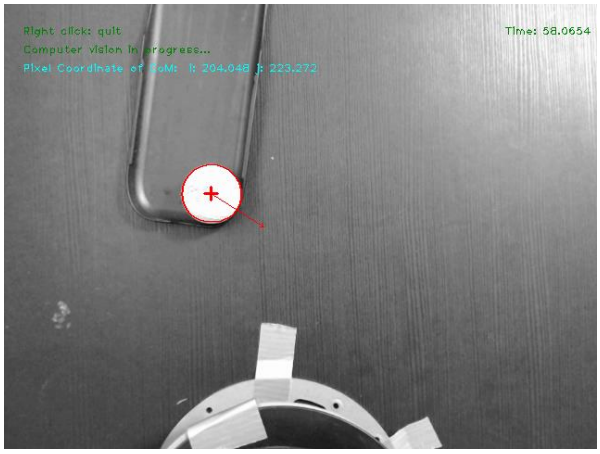

Figure 24: Program tracking golf ball and recording pixel coordinates.

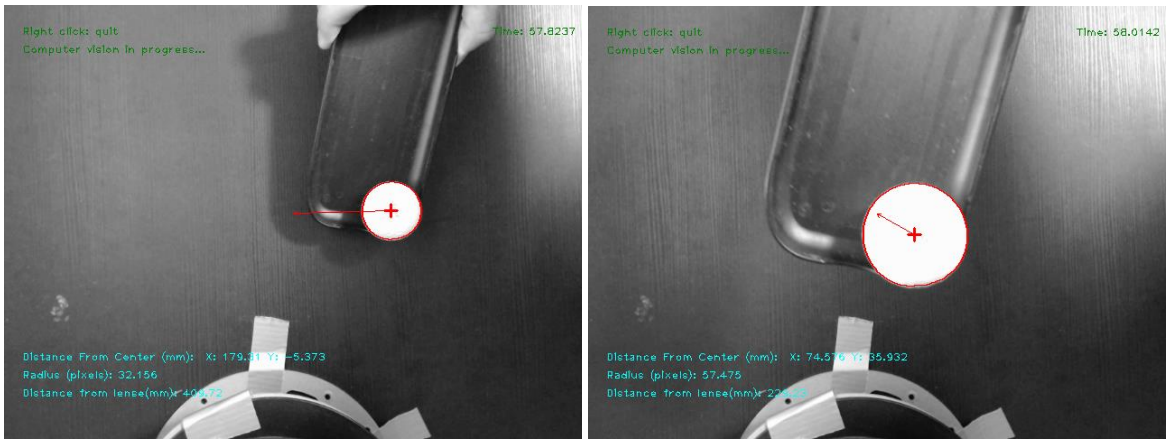

Figure 25: Tracking position of the golf ball in mm. The reference frame is located at the center of the camera's sensor. The position, pixel radius, and approximated distance from the lens is shown. The red arrow points towards the origin of the reference frame.

The program has two sections. First, the golf ball object is initialized with a mouse click. Second, blob tracking outlines the ball and tracks its movement from frame to frame. Finally, the pixel coordinates are transformed into real units and the area the blob analyzed to find the radius. The results are posted on the screen in blue.

The implemented program is listed below.

```
//This function tracks a golf ball and returns its pixel coordinate,
//distance from optical center in mm, and the approximate distance from the lens
//Created by: Matthew Ebert Date: 2022-07-12
//Inspire by code created by ViSP for 4Dot pose estimation


#include "main.hpp"

// FUNCTIONS
void findGlobalCoordinates(vpPoint &point, vpImagePoint &ip, const vpCameraParameters &cam, bool init, vpHomogeneousMatrix &cMo);

vpImagePoint track(vpImage<unsigned char> &I, vpDot2 &dot, bool init);
#define VISP_HAVE_OPENCV

// Cosntants
const std::string opt_intrinsic_file = "F:/DEV/VISP_Testing/include/camera.xml";
const std::string opt_camera_name = "Camera";
const int opt_device = 0;

int IPVSGolf()
{
  try
  {

    vpImage<unsigned char> I;

    // Set up camera parameters
    vpCameraParameters cam(840, 840, I.getWidth() / 2, I.getHeight() / 2); // Default parameters
    vpXmlParserCamera parser;
    if (!opt_intrinsic_file.empty() && !opt_camera_name.empty())
    {
      std::cout << "Intrinsic file: " << opt_intrinsic_file << std::endl;
      std::cout << "Camera name   : " << opt_camera_name << std::endl;
      if (parser.parse(cam, opt_intrinsic_file, opt_camera_name, vpCameraParameters::perspectiveProjWithDistortion) ==
          vpXmlParserCamera::SEQUENCE_OK)
      {
        std::cout << "Succefully Imported Camera Paramters" << std::endl;
      }
      else
      {
        std::cout << "Unable to read camera parameters from xml file" << std::endl;
        throw(vpException(vpException::fatalError,
                "Could not import camera parameters from : %s", opt_intrinsic_file));
      }
    }//if

    //Open Camera Devices
    std::cout << "Use OpenCV grabber on device " << opt_device << std::endl;
    cv::VideoCapture g(opt_device);
    if (!g.isOpened())
    {
      std::cout << "Failed to open the camera" << std::endl;
      throw(vpException(vpException::fatalError, "Could not open Camera: %d", opt_device));

    }
    //Read initial frame
    cv::Mat frame;
    g >> frame;
    //Convert to VP image
    vpImageConvert::convert(frame, I);

    // The pose container
    vpHomogeneousMatrix cMo;
    vpDot2 dot;       //finds golf ball as circle on black background
    vpPoint point;   // 3D coordinates of the points
    vpPoint spoint;
    vpImagePoint ip; // 2D coordinates of the points in pixels
    vpImagePoint ips;
    vpPoint r;
    vpDisplayOpenCV d(I);
    bool quit = false;
    bool apply_cv = false; // apply computer vision
    bool init_cv = true;   // initialize tracking and pose computation

    //set objective position
    ips.set_ij(230, 320);

    while (!quit)
    {
      double t_begin = vpTime::measureTimeMs();
      // Image Acquisition
      g >> frame;
      vpImageConvert::convert(frame, I);

      vpDisplay::display(I);
      if (apply_cv)
      {
        try
        {

          ip = track(I, dot, init_cv); //Find image coordinates of CoM of the golfball
          findGlobalCoordinates(point, ip, cam, init_cv, cMo);
          double radius = sqrt(dot.getArea()/M_PI);//find radius from dot area
          double z = 250 * 52.7/radius; //calculate distance from radius
```

```
          //vpPixelMeterConversion::convertPoint(cam, ip, prad, temp);

          vpDisplay::displayFrame(I, cMo, cam, 0.3, vpColor::none, 3);
          if (init_cv)
            init_cv = false; // turn off the computer vision initialisation
          {
            std::stringstream ss;
            //ss << "Pixel Coordinate of CoM: " << " i: " << ip.get_i() << " j: " << ip.get_j();
           // vpDisplay::displayText(I, 400, 20, ss.str(), vpColor::red);
            ss << "Distance From Center (mm): " << std::setprecision(5) << " X: " << point.get_x() * (1000) << " Y: " << point.get_y() * 1000;
            vpDisplay::displayText(I, 380, 20, ss.str(), vpColor::cyan);
            ss.str("");
            ss << std::setprecision(5) << "Radius (pixels): " << radius;
            vpDisplay::displayText(I, 400, 20, ss.str(), vpColor::cyan);
            ss.str("");
            ss << std::setprecision(5) << "Distance from lense(mm): " << z ;
            vpDisplay::displayText(I, 420, 20, ss.str(), vpColor::cyan);
            vpDisplay::displayArrow(I, ip, ips, vpColor::red, 4U, 2U,1U);

          }
        }
        catch (...)
        {
          std::cout << "Computer vision failure." << std::endl;
          apply_cv = false;
          init_cv = true;
        }
      }
      vpDisplay::displayText(I, 20, 20, "Right click: quit", vpColor::darkGreen);
      if (apply_cv)
      {
        vpDisplay::displayText(I, 40, 20, "Computer vision in progress...", vpColor::darkGreen);
      }
      else
      {
        vpDisplay::displayText(I, 40, 20, "Left click : start", vpColor::darkGreen);
      }
      vpMouseButton::vpMouseButtonType button;
      if (vpDisplay::getClick(I, button, false))
      {
        if (button == vpMouseButton::button3)
        {
          quit = true;
        }
        else if (button == vpMouseButton::button1)
        {
          apply_cv = true;
        }
      }
      {
        std::stringstream ss;
        ss << "Time: " << vpTime::measureTimeMs() - t_begin << " ms";
        vpDisplay::displayText(I, 20, I.getWidth() - 100, ss.str(), vpColor::darkGreen);
      }
      vpDisplay::flush(I);
    }
  }
  catch (const vpException &e)
  {
    std::cout << "Catch an exception: " << e.getMessage() << std::endl;
  }

  return 0;
}

vpImagePoint track(vpImage<unsigned char> &I, vpDot2 &dot, bool init)
{
  try
  {
    vpImagePoint ip;
    if (init)
    {
      vpDisplay::flush(I);

      dot.setGraphics(true);
      dot.setGraphicsThickness(2);
      std::stringstream ss;
      ss << "Click on point ";
      vpDisplay::displayText(I, 20, 20, "Status: initialize blob tracker", vpColor::red);
      vpDisplay::displayText(I, 40, 20, ss.str(), vpColor::red);
      vpDisplay::flush(I);
      dot.initTracking(I);
      vpDisplay::flush(I);
    }
    else
    {
      dot.track(I);
    }

    ip = dot.getCog();
    return ip;
  }
  catch (...)
  {
    std::cout << "Traking lost" << std::endl;
    throw(vpException(vpException::fatalError, "Tracking lost"));
  }
}

void findGlobalCoordinates(vpPoint &point, vpImagePoint &ip, const vpCameraParameters &cam,
                  bool init, vpHomogeneousMatrix &cMo)
```

```
{
  vpPose pose;
  double x = 0, y = 0;
  vpPixelMeterConversion::convertPoint(cam, ip, x, y);
  point.set_x(x);
  point.set_y(y);
}
```

G. Conclusion

The objective of this laboratory was to experiment with and implement various visual servoing techniques. The intrinsic parameters of a camera were found using an ViSP calibration method. These parameters were used to implement an object pose estimating program which successfully tracked and analyzed a live video feed. An IBVS simulation was tested for settling time and convergence. It was found the system had the fastest settling time with a proportional gain of 30 and became unstable when the gain reached 49. Finally, a program was written which track a golf ball on a horizontal surface. The program measures the pixel coordinates, distance from the optical center in mm, and approximated the golf ball's distance from the lens using it a ratio of pixel radii. Future test should implement a physical control system to verify the IBVS simulation results.