EE-543: Vision and Learning Based Control

Project Report

Group 3

IBVS, PBVS, Q-Learning

Kushagra Agrawal	**B20296**
Madhur Jajoo	**B20211**
Vanshaj Nathani	**B20237**

1 IBVS

Image-based visual servoing (IBVS) is a technique used in robotics and computer vision to control the motion of a robot based on visual information obtained from images. It involves using camera images to guide the robot's movements and achieve a desired task or goal.

1.1 Overview

1.1.1 Image Acquisition

The robot captures images of the scene or environment using one or more cameras. These images serve as the visual input for the servoing process.

1.1.2 Image Processing

The acquired images are pre-processed to extract relevant information. This may involve operations such as filtering, image enhancement, feature extraction, or object detection. Here we have used Aruco Marker. then we extract the features i.e. corners of the marker, these features are reference points for the robot.

1.1.3 Error Computation

The visual error is computed by comparing the current image features with the desired image features. Various techniques can be used to measure the discrepancy, such as Euclidean distance, geometric transformations, or statistical methods.

1.1.4 Control law

A control law or algorithm is designed to translate the visual error into control signals for the robot. This control law determines how the robot should adjust its motion to minimize the visual error. Common control laws include proportional-derivative (PD) control, inverse kinematics, or optimization-based methods.

1.1.5 Robot Motion

The computed control signals are used to update the robot's pose, position, or joint values. The robot moves or adjusts its configuration based on the control commands to minimize the visual error and reach the desired state.
The above process is done iteratively in a loop.

1.2 CODE Explaination

1.2.1 getJacobian Function

This function computes the Jacobian matrix for a given pixel coordinates (u, v) and focal length (f) and depth (z).
The Jacobian matrix represents the partial derivatives of the image coordinates with respect to the camera coordinates.

1.2.2 imageConversion Function

It converts an array of RGBA pixel values into a NumPy array representing an image.
It creates a new image from the pixel values using the 'PIL' library and returns it as a NumPy array.

1.2.3 servoing Function

This function performs ArUco marker detection on a given frame using the OpenCV library.
It converts the frame to grayscale and uses the aruco.detectMarkers function to detect ArUco markers in the frame.
If markers are detected, it extracts the corner points and calculates the center point.
It visualizes the detected markers and returns the points.

1.2.4 robotControl Function

This function calculates the control velocity for the robot based on the difference between the required position (requiredPos) and the detected points (points).
It computes the error vector, calculates the root mean square (RMS) error, and stores the RMS error and timestamp.
It computes the Jacobian matrix using the getJacobian function and performs velocity computation using the MP-pseudoinverse of the Jacobian and the error vector.
It returns the velocity vector.

1.2.5 main()

It initializes the PyBullet physics simulation environment and loads the URDF models.
It sets up the camera view and captures images from the simulation using p.getCameraImage.
It converts the captured image to a NumPy array and performs ArUco marker detection using servoing.
If markers are detected, it computes the control velocity using robotControl and updates the robot's position and orientation accordingly.
It also updates the plot of the root mean square (RMS) error.
The simulation runs in a loop until the program is terminated.
Calling main Function starts the simulation

2 PBVS

It is also a technique used in robotics and computer vision to control the orentiation and motion of a robot based on visual information obtained from images, Unlike image-based visual servoing (IBVS), which directly operates on image features, PBVS operates on the 3D position and orientation of objects in the scene.

2.1 Overview

2.1.1 Image Acquisition

The robot captures images of the scene or environment using one or more cameras. These images serve as the visual input for the servoing process.

2.1.2 Camera Callibration

The camera parameters, such as intrinsic and extrinsic parameters, need to be calibrated to accurately relate the 2D image coordinates to the 3D world coordinates.

2.1.3 3D Scene Reconstruction

Using the acquired images and camera calibration, a 3D reconstruction of the scene or objects of interest is performed. This involves estimating the 3D positions and orientations of features or objects from the image correspondences.

2.1.4 Error Computation

The visual error is computed by comparing the current 3D positions and orientations with the desired ones. This error can be computed using different techniques, such as geometric transformations, point cloud registration, or pose estimation algorithms.

2.1.5 Control Law

A control law or algorithm is designed to translate the visual error into control signals for the robot. This control law determines how the robot should adjust its position and orientation to minimize the visual error.

2.1.6 Robot Motion

The computed control signals are used to update the robot's position and orientation. The robot moves or adjusts its configuration based on the control commands to minimize the visual error and reach the desired position and orientation.

The above process is done iteratively in a loop.

2.2 Code Explaination

2.2.1 getJacobian Function

This function computes the Jacobian matrix for a given pixel coordinates (u, v) and focal length (f) and depth (z).
The Jacobian matrix represents the partial derivatives of the image coordinates with respect to the camera coordinates.

2.2.2 imageConversion Function

It converts an array of RGBA pixel values into a NumPy array representing an image.
It creates a new image from the pixel values using the 'PIL' library and returns it as a NumPy array.

2.2.3 servoing Function

This function performs ArUco marker detection on a given frame using the OpenCV library.
It converts the frame to grayscale and uses the aruco.detectMarkers function to detect ArUco markers in the frame.
If markers are detected, it extracts the corner points and calculates the center point.
It visualizes the detected markers and returns the points.

2.2.4 getpv function

This function takes the sorted corner coordinates of markers as input and calculates the cross product of two vectors formed from the corner coordinates.

2.2.5 transformAxes function

This function takes a transformation matrix and a set of coordinates as input. It applies the transformation to the coordinates and returns the transformed coordinates.

2.2.6 robotControl

This function takes the sorted corner coordinates of markers, focal length (default: 1), and a mapping array 'k' as input. It calculates the 3D positions of the markers based on the provided mapping array and returns the new 3D positions of the markers.

2.2.7 main()

It initializes the PyBullet physics simulation environment and loads the URDF models.
Captures an image using the simulated camera in the environment. and converts it to numpy array
Performs marker detection and obtains the sorted corner coordinates.
Calls the robotControl function to calculate the new 3D positions of the markers.
Calculates the desired velocity for the robot based on the new marker positions and the current robot position.
Applies the desired velocity to the robot joints to control its movement.
Calling main Function starts the simulation.

3 Q-Leaerning

3.1 Overview

Q-learning is a reinforcement learning algorithm that aims to find an optimal action-selection policy for an agent in a Markov decision process (MDP). It is a model-free algorithm, meaning it does not require prior knowledge or a model of the environment.

3.1.1 State and Action

The environment is modeled as a series of discrete states, and the agent can take actions to transition between states.

3.2 Q-Table

Q-learning maintains a Q-table that represents the expected cumulative reward (called the Q-value) for each state-action pair. Initially, the Q-table is initialized with arbitrary values.

3.2.1 Exploration vs. Exploitation

The agent balances exploration and exploitation to learn the optimal policy. During exploration, the agent takes random actions to gather information about the environment. During exploitation, the agent chooses the action with the highest Q-value for the current state.

3.2.2 Reward and update

After taking an action, the agent receives a reward from the environment. The Q-value for the current state-action pair is updated using the following formula:$Q(s_t,a_t) = Q(s_t,a_t) + \alpha * (R + \gamma * Q(s_{t-1},a_{t-1})$
Here: $Q(s_t,a_t) is Q - value of state action pair at time t$
$\alpha = Learning rate$
R: immediate reward
γ =Discount Factor

3.2.3 Iterative Update

The agent interacts with the environment, updating the Q-table based on the observed rewards and updating its policy. The process continues until the agent converges to an optimal policy.

3.2.4 Convergence

Q-learning guarantees convergence to the optimal policy under certain conditions, such as exploring all state-action pairs infinitely and a sufficiently small learning rate.

3.3 Code Explanation

We define the class move to represent the different directions as numbers (UP is 0, Down is 1, Left is 2, Right is 3)
We define Class Q-learning which have the Q-Learning algorithm. We initialize Learning rate, Discount factor,probability(ε) Q-table with the shape specified.

3.3.1 Update function

It updates the q-value for the given state-action pair in the table.

3.3.2 _get_temporal_difference function

it calculates the temporal difference between the current Q-value and the updated Q-value.

3.3.3 _get_temporal_difference_target Function

it calculates the target Q-value for the next state based on the maximum Q-value.

3.3.4 get_best_action function

It returns the best action to take for a given state based on the Q-values.

3.3.5 main()

It sets up the problem environment, initializes the Q-learning agent, and runs the Q-learning algorithm. The agent chooses actions either randomly (with a probability of ε) or based on the best Q-value. The next position is updated accordingly.
If the next position is outside the grid boundaries, the agent receives a negative reward, and the episode ends. Otherwise, if the agent reaches the goal position, it receives a positive reward, and the episode ends.
After each action, the agent updates the Q-values based on the observed reward and the temporal difference.
The loop continues until the maximum number of actions is reached or the agent reaches the goal.
The code at the end gives the optimal path based on the q-value and the q-matrix in action.txt file
The file final.py gives the robot simulation of the q-learning based on this action.txt file.