Object Tracking using Arm-Mounted Camera

RBE501 - Robot Dynamics Course Project

Worcester Polytechnic Institute

Bhushan Ashok Rane
Gaurang Manoj Salvi
Rutwik Rajesh Bonde
Yash Rajendra Patil

December 13, 2021

ABSTRACT

In this project, we have designed and implemented a object tracking system using robotic arm. This involves usage of 7 DOF Kinova Robot Manipulator with a stereo vision Intel RealSense Depth Camera mounted on the end-effector with eye-in-hand configuration. We try to keep the center of object at the center of camera. Using the camera feed, an object is detected in the camera frame and the center of object is estimated. The camera velocity in cartesian space is obtained as a product of the error distance between object center and camera center and the Image Jacobian. This is further converted in to joint velocities using Jacobian Inverse. This provides us the joint velocities required for tracking the object. Using the velocity controller, end-effector is moved in required direction and object is kept centered in the frame. Such kind of object tracking system, based on Visual servoing, is being used for cinematography, filming sports (for player tracking) and tracking and picking up a moving object on conveyer belt in manufacturing industries.

Introduction

Object tracking is the task of automatically identifying the objects in the real world and interpreting them as a set of trajectory and then tracking them as they move around in the environment. In this project, a robotic manipulator (with a camera in eye-in-hand configuration), with real-time visual tracking of 3D object travelling at unknown velocities in 2D space is presented. Computer Vision is integrated with Robot Control by understanding the dynamics of robot manipulators and their motion in 3D space. The goal of the project is to move the end effector of the robot in 3D space by setting arm joints into position corresponding to the real time tracking of target object.

Materials

- **Kinova Gen3 Robot**
  Kinova Gen 3 is a 7 degrees of freedom robotic manipulator with a continuous payload capacity of 4kg and reach of 902mm. It has a modular hardware, robust Kinova Kortex application programming interface, and a very versatile end-effector interface module. A 2D/3D vision module can be integrated with the robot optionally. Additionally, there is no constraint on rotation of any robot joint and infinite rotation on all robot joints can be obtained. Using Gen3 robot manipulator is advantageous for this project because of open-source Software Development Kit (SDK) and ROS Packages, availability of dynamic programming environments using C++ and Python, and easy integration with Gazebo simulation environment using MoveIt. A stereo vision camera has been mounted in eye-in-hand configuration, on the end-effector of Kinova Gen3 robot for our project usage.

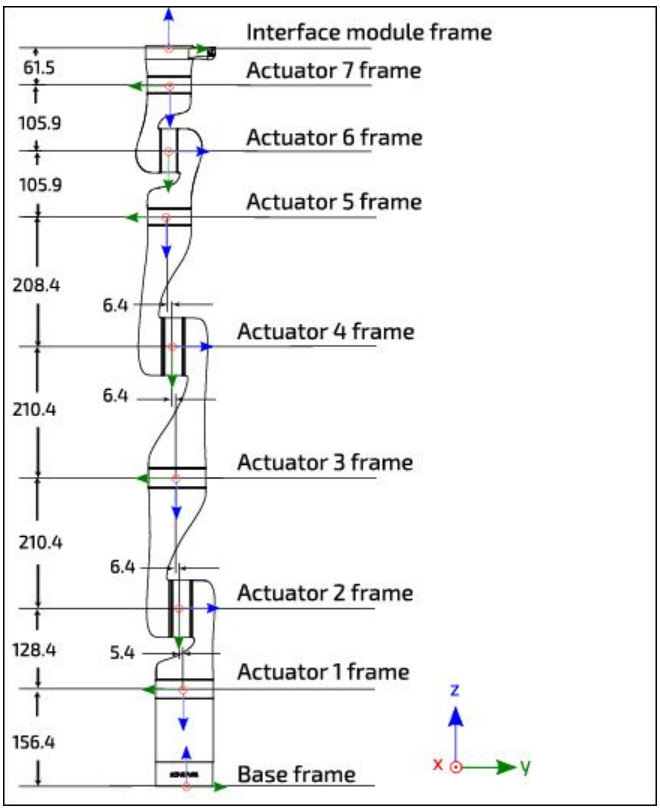

Figure 1: Kinova Gen 3

- **Robot Operating System (ROS)**
  ROS is an open-source middleware suite of software tools and libraries used for development of robot applications. Being open-source, ROS is widely used by researchers and industry. Also, it's modular architecture makes your code more reusable. It provides a set of utilities to the user to communicate over topics, services, action services allowing nodes to exchange information messages with each other. In this project, ROS Noetic on Ubuntu 20.04 will be used for development of all the software framework.
- **Gazebo**
  Gazebo is an open-source real-time 3D robotics simulator used to simulate the physics and visuals of realistic and functional robot models. You can evaluate your algorithms and test your robot in difficult and dangerous

scenarios without any harm to your robot. For robot manipulators, it is essential to initially test the robot trajectory for safety purposes before implementing the whole scenario on real robot. Gazebo is just a 3D simulator, while ROS serves as the interface for the robot. Combining both, results in a powerful robot simulator. In this work, Kinova Gen3 will be spawned in controlled environment and object tracking trajectories will be tested.

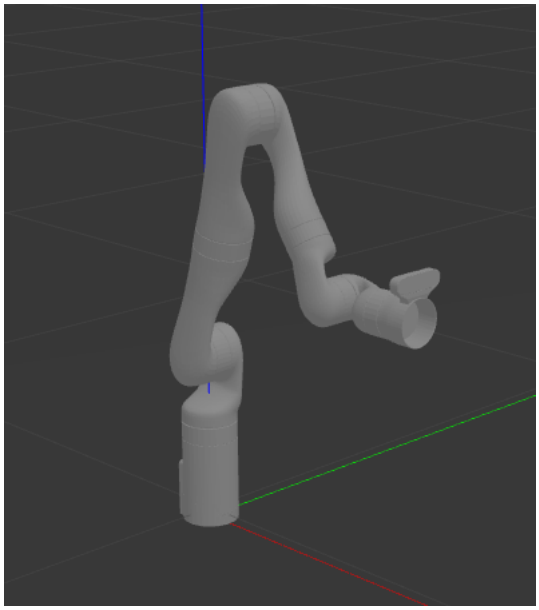

Figure 2: Kinova Gen 3 in Gazebo

Methods

1. **Colour Thresholding:**

   Use color thresholding to specify a color range and return a black and white image. All colors between the start and stop colors (inclusively) become white and the rest of the image pixels become black. The two colors are separated with a hyphen between them. Thresholding, by default, take place in the sRGB colorspace. To try out the algorithm and gauge it's accuracy and effectiveness, we did the blob tracking.Blob is a mass of uniform colour, i.e. we are used a coloured cube as a blob and perform colour thresholding to filter that colour. In colour thresholding, we specify a range within which the different shades of colour that we are looking for, lies. And all the pixels that lie within that range are assigned bright value and everything else is dark. Generally, colour thresholding is done in HSV (Hue-Saturation-Value) range because HSV is more robust to lighting changes and keeps giving consistent results.

   Hue-Saturation-Value Representation Model:
   Hue in HSV model represent the color portion of the model. It is expressed as a number from 0 to 360 degrees as follows:-

   - Red color falls between 0 and 60 degrees.
   - Yellow color falls between 61 and 120 degrees.
   - Green color falls between 121 and 180 degrees.
   - Cyan color falls between 181 and 240 degrees.
   - Blue color falls between 241 and 300 degrees.
   - Magenta color falls between 301 and 360 degrees.

   Saturation is the amount of gray in a particular color, from 0 to 100 percent. Reducing this component toward zero introduces more gray and produces a faded effect. Sometimes, saturation appears as a range from 0 to 1, where 0 is gray, and 1 is a primary color.

   Value works in conjunction with saturation and describes the brightness or intensity of the color, from 0 to 100 percent, where 0 is completely black, and 100 is the brightest and reveals the most color.

2. **Center Estimation of Blob**

   Once the blob has been detected or segmented from the environment, we need a point on the blob to track it, and the most intuitive point is the center of the blob. This will keep the center of the blob at the frame center. We estimate the center of the object by calculating Moments or just by averaging all the bright pixels after color thresholding. After the calculation, we obtain the center of the object in Pixel frame, i.e., in terms of number of pixels from the upper left corner of the image.

3. **Image Jacobian**

   Image Jacobian defines the relationship between the velocity of point in pixel frame to the camera velocity in world frame. Image Jacobian is 2 x 6 matrix. If the object moves forward, then by looking at two consecutive frames, we can get the position of center in both frames and the estimate the velocity of the object in pixel frame. This velocity can be then converted to the camera velocity by multiplying it with inverse of Image Jacobian. The image Jacobian is given by:-

   Pixel Velocity = Image Jacobian * Camera Velocity

$$\begin{bmatrix} \dot{f_u} \\ \dot{f_v} \end{bmatrix} = \begin{bmatrix} -\frac{\lambda}{z} & 0 & \frac{f_u}{z} & \frac{f_u . f_v}{\lambda} & -(\lambda + \frac{f_u^2}{\lambda}) & f_v \\ 0 & -\frac{\lambda}{z} & \frac{f_v}{z} & (\lambda + \frac{f_v^2}{\lambda}) & -\frac{f_u . f_v}{\lambda} & -f_u \end{bmatrix} \begin{bmatrix} v_c \\ w_c \end{bmatrix}$$

   $\dot{f_u}$ = Pixel velocity in X-direction
   $\dot{f_v}$ = Pixel velocity in Y-direction
   f_u = Pixel value of feature point in X-direction
   f_v = Pixel value of feature point in Y-direction
   λ = Focal length of camera
   z = Distance of camera from feature point
   v_c = Linear Component of Camera Twist
   w_c = Rotational Component of Camera Twist

4. **Jacobian Inverse**

   Once we get the velocity of camera from the inverse image jacobian equation, we essentially obtain the end-effector velocity. Now, this can be converted to joint velocity using Inverse Jacobian of the robot. By multiplying Inverse Jacobian with the end-effector velocity we will get the joint velocities which are required to move the end-effector in that particular direction.

Implementation

- **ROS Kortex**

  The Kinova Gen3 simulation package is provided by kinova named ros_kortex which has ROS packages to use the Kinova Gen3 Arm in simulation as well as interact with a real Kinova Gen3 Arm. The catch here was the URDF file of Kinova Gen3 Arm did not have a camera integrated in it. We had to add the camera by making a couple changes in URDF file and adding the snippet to spawn the camera on the end-effector on the arm. We added Intel Realsense Camera using the *intelrealsense* in file *libgazebo_ros_openni_kinect.so*

Figure 3: View from Camera in Home Position

- **Simulation Environment**
  In the simulation environment only 2 models are present, first the Kinova Gen3 Manipulator and other is the blue coloured cube which acts as the object to be tracked.

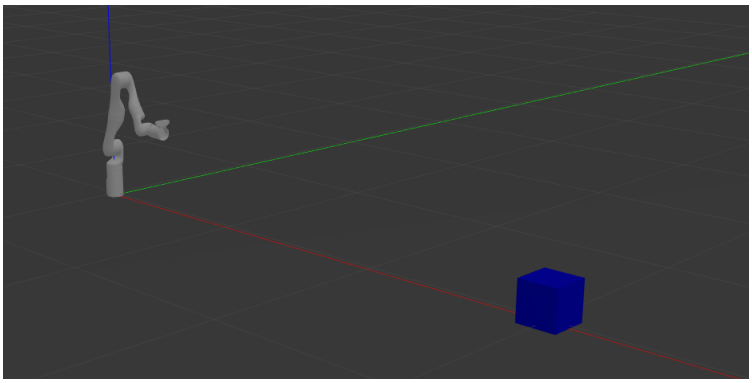

Figure 4: Simulation Environment

- **Color Thresholding**
  We are using color thresholding to get the center of the object which is used a feature point for visual servoing. We first convert the image from BGR colorspace to HSV colorspace for robust thresholding. Thresholding is carried out, essentially if the pixel value lies between a specified range then the pixel is assigned value 255 or else 0. This creates a binary Mask. Then to estimate the center of the segmented image, we can either use Moments or we can simply average the position of all bright pixels. As a result, we get the center of the blob which is tracked.

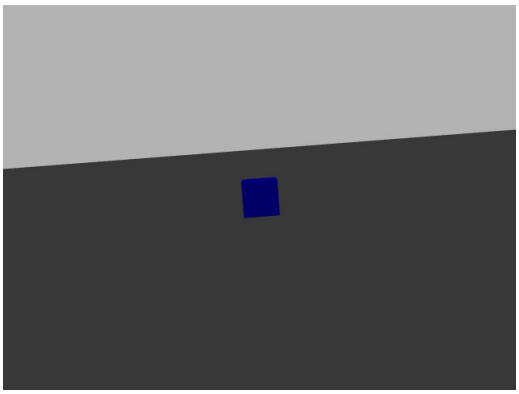

Figure 5: Original Image

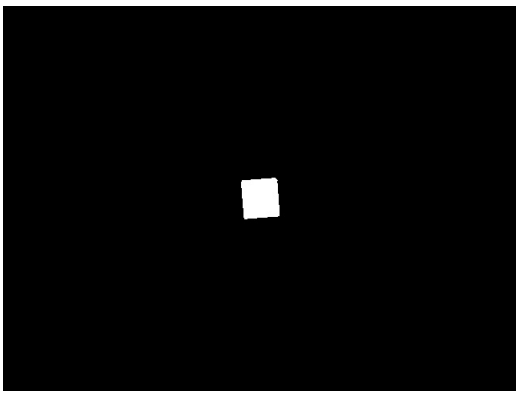

Figure 6: Mask Detection from Color Thresholding

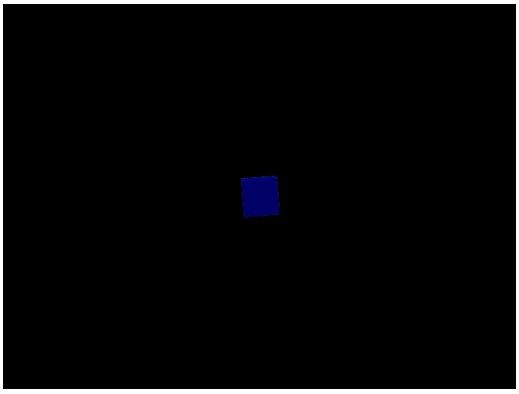

Figure 7: Segmented Object

- **Velocity Control**
  Now we have a current and a desired location of the feature point, desired being the center of the frame. The error is calculated and it is multiplied with a Gain for fast convergence. This error is then multiplied with the Inverse of Image Jacobian to obtain required Camera Velocity. Now this Camera Velocity is in Body frame, we convert it to Space frame. This is the required camera velocity but we can only control the joint velocities so we convert the camera velocity to joint velocities by multiplying it with inverse robot jacobian. This provides us with the joint velocities which are then applied to the robot. The *kortex_driver* which acts as an interface between the robot and ROS subscribes to a topic "/my_gen3/in/joint_velocity". When published relevant message with all 7 joint velocities to this topic the driver applies them in simulation. And this cycle continues on receiving each new image, it is thresholded, the center is estimated and required joint velocities are applied.

Results

With the above algorithm we were able to successfully track the object. It all came down to gain tuning for fast as well as smooth tracking. If the gain was kept too high (gain > 200) the robot would go berserk because the calculation gave insanely high joint velocities which would not even be possible in real life. For Moderately-high gains (gain > 100) the robot tends to overshoot and trace circles around the actual goal point. The time taken was considerable and also movement was a little shaky due to constant velocity changes. Moderate Gains (gains > 50) performed well in terms of smooth tracking but took some time to get to the point. For Low Gains (gains < 49) the tracking was very smooth, nearly free of shaking but took longer time. Figures 12, 13, 14, 15, and 16 represent the error in number of pixels on Y-axis and time required to track object in seconds on X-axis.

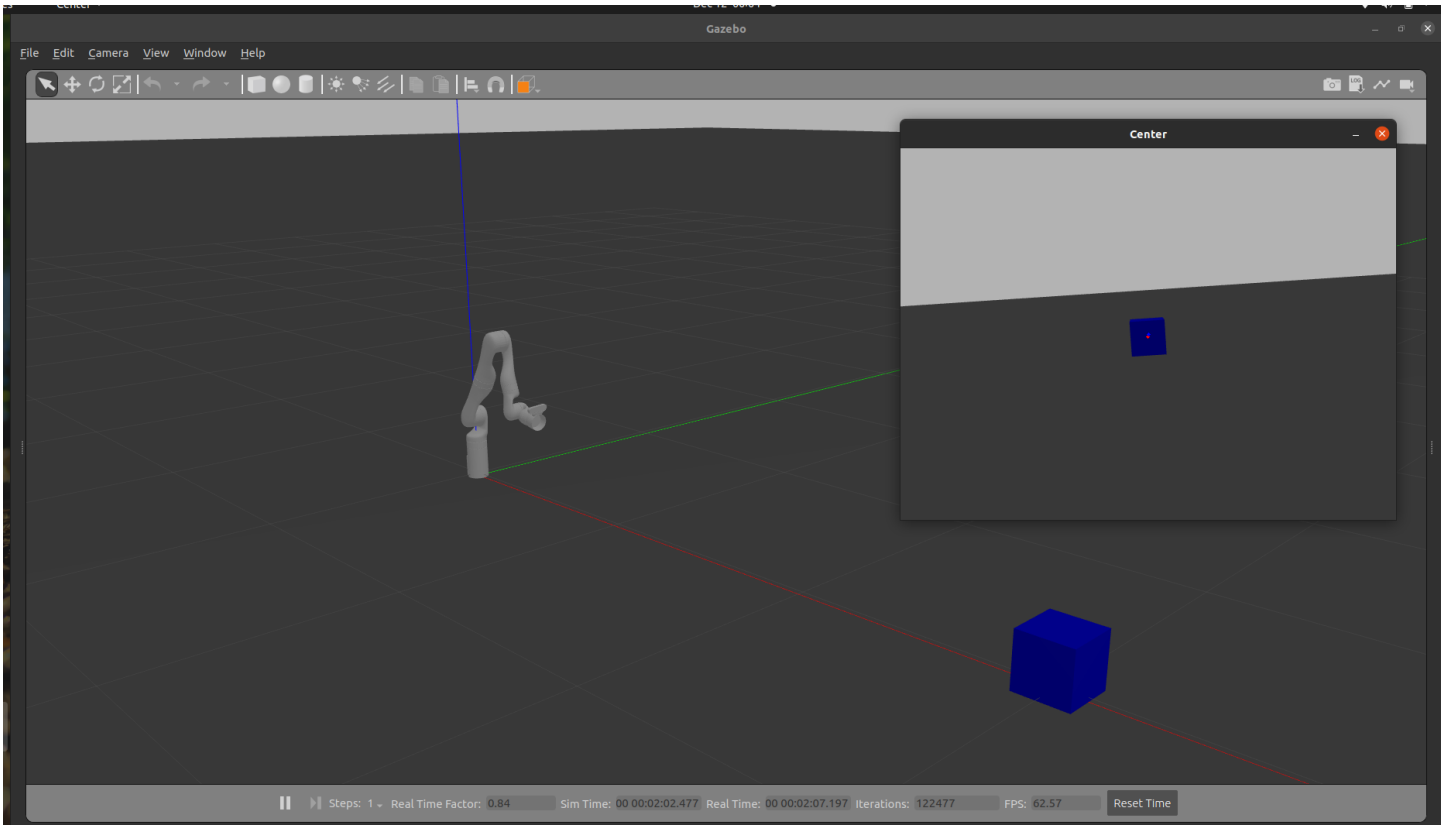

Figure 8: Performance Testing for initial position

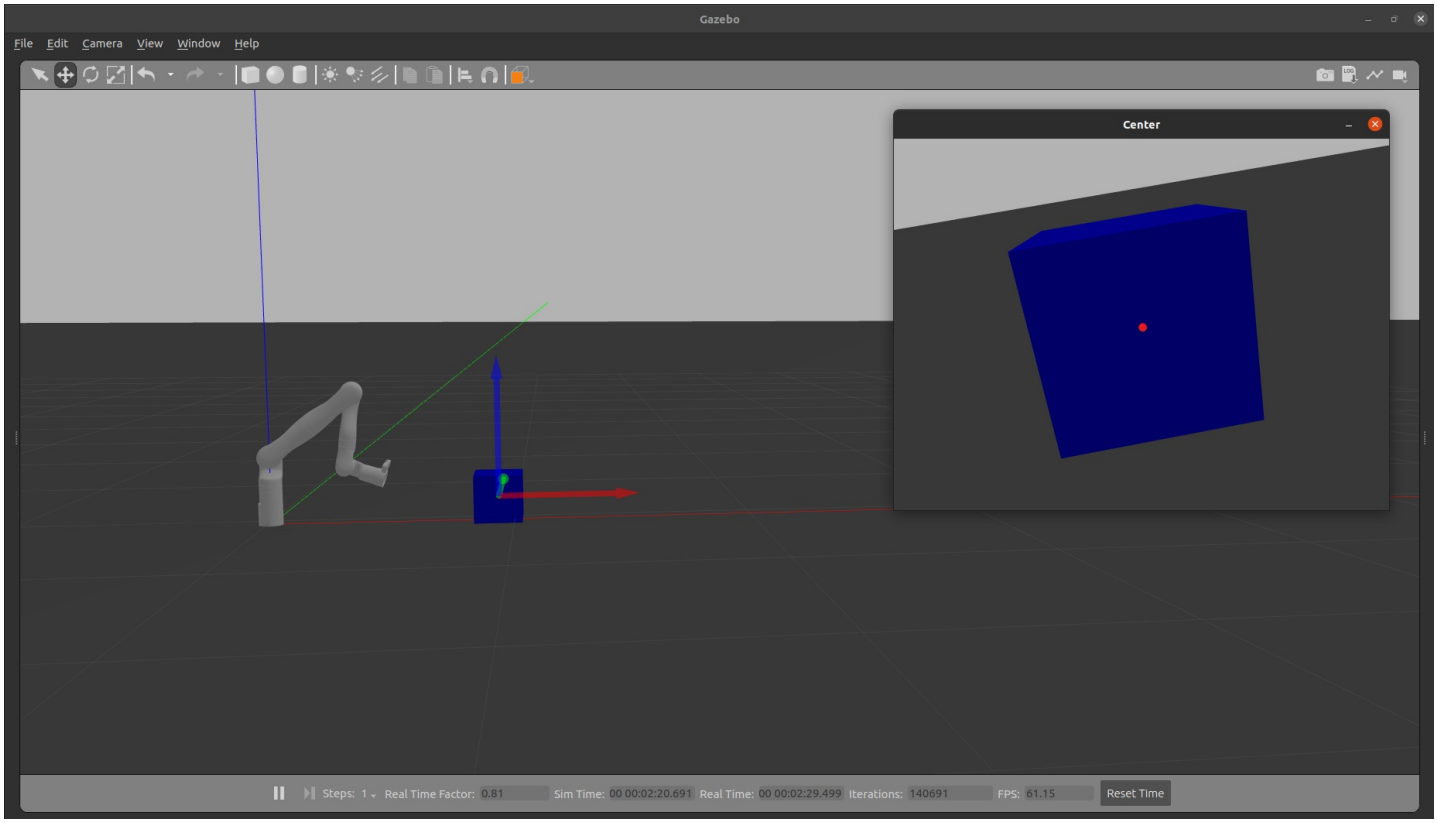

Figure 9: Performance Testing for position nearer to robot

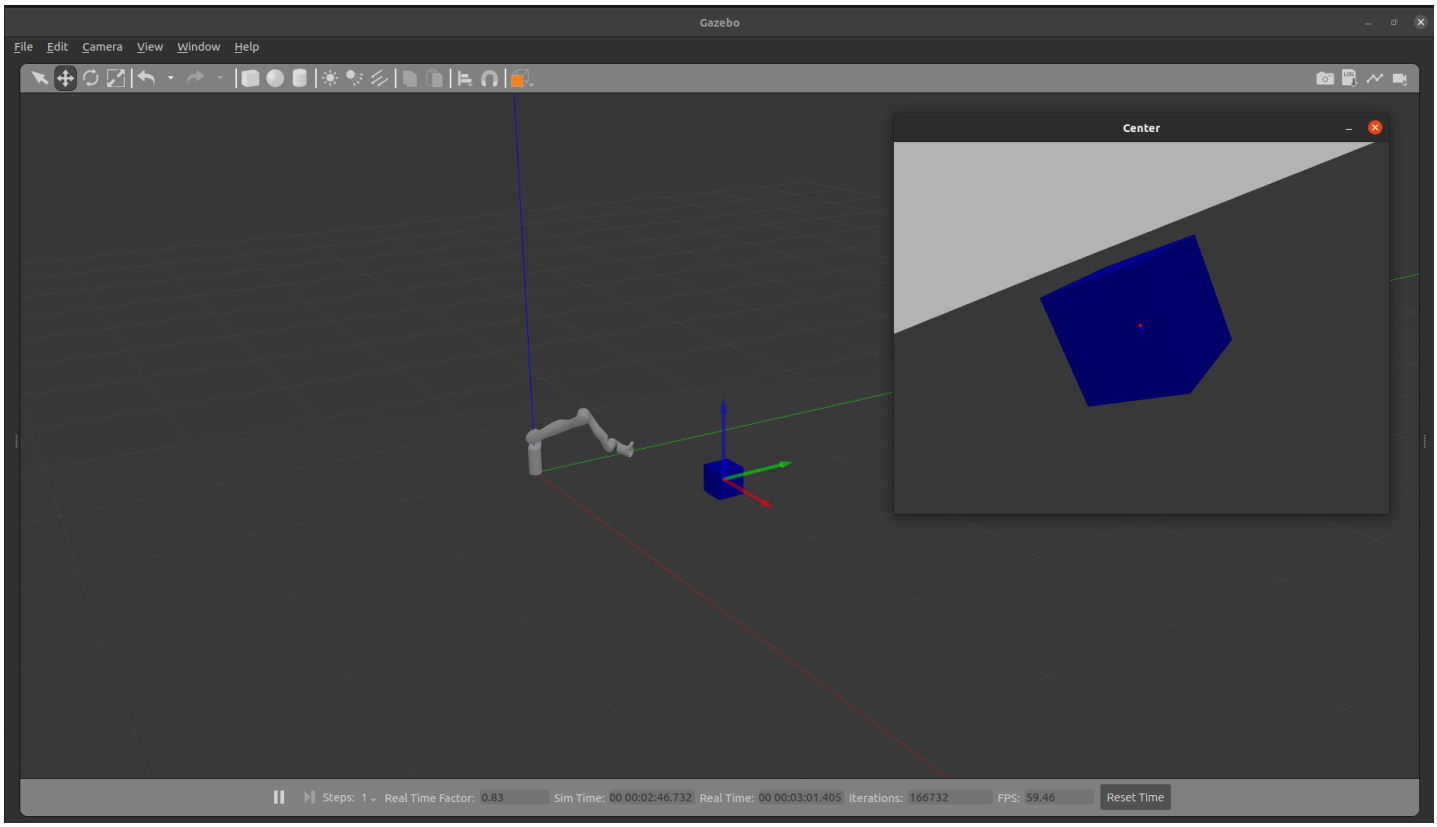

Figure 10: Performance Testing for arbitrary position

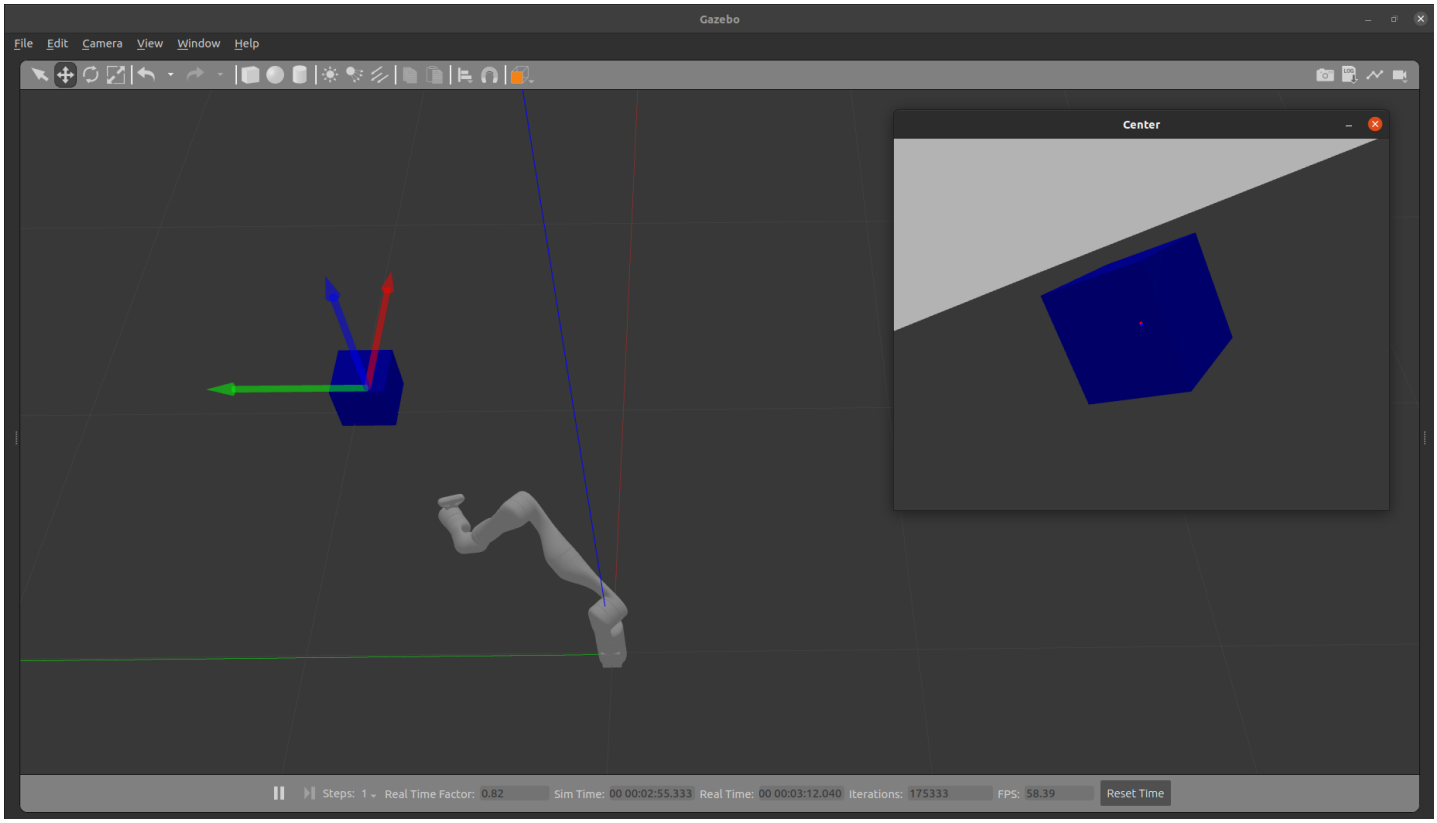

Figure 11: Performance Testing for arbitrary position (Isometric View)

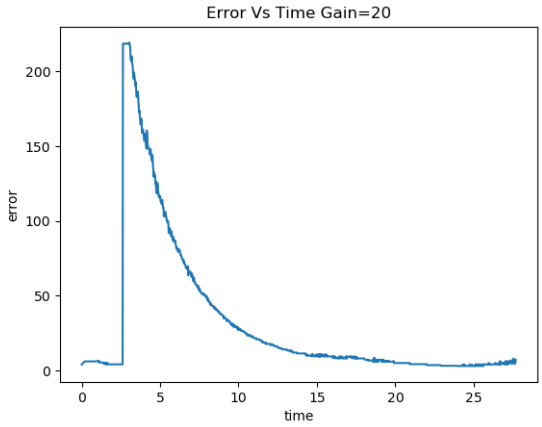

Figure 12: Error vs Time Plot for Gain = 20

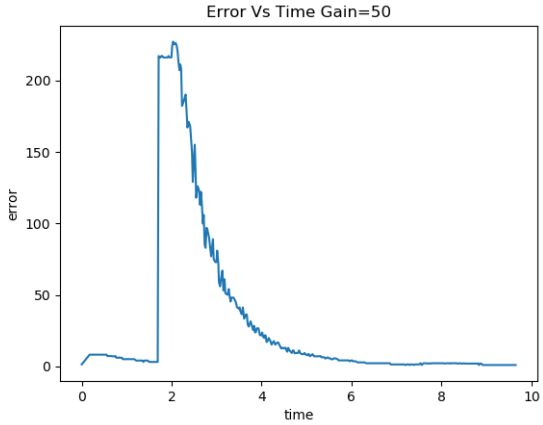

Figure 13: Error vs Time Plot for Gain = 50

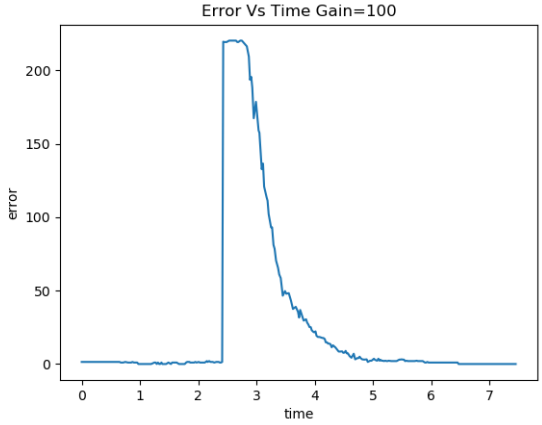

Figure 14: Error vs Time Plot for Gain = 100

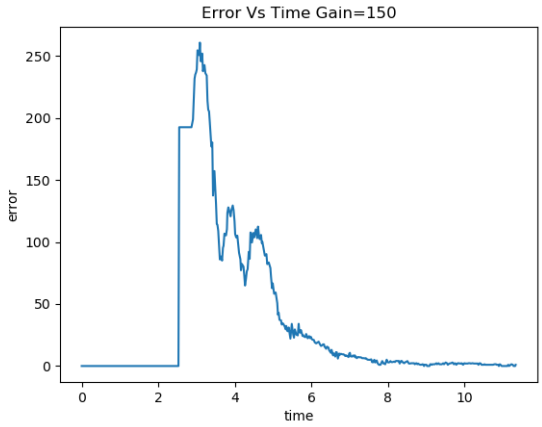

Figure 15: Error vs Time Plot for Gain = 150

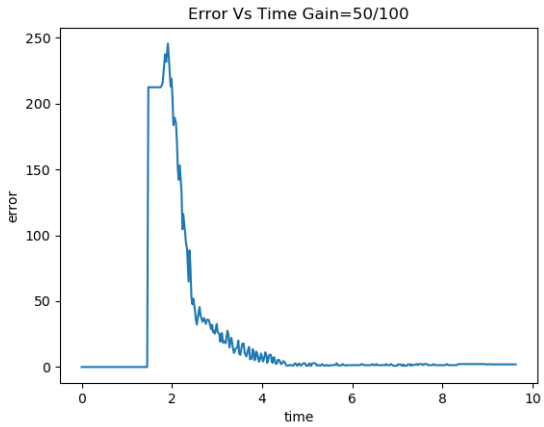

Figure 16: Error vs Time Plot for Gain = 50/100

Conclusion

To keep the moving object continuously focused in the real time video frame, it is very necessary to keep a track of the object while in motion. We describe a system to allow for real time object tracking in a defined work space using a robot manipulator. From the experiments, we found that gain tuning according to error between the camera center and object center, is of crucial importance for smooth motion of camera in real time following the object. We have followed a kind of approach which enables the robot to discover itself, the gain required for the robot to move smoothly according to detected error. Our results provide evidence that Visual servoing (vision-based robot control) is efficient way to track a moving object in real time.

References

1. W. E. Dixon, E. Zergeroglu, Y. Fang and D. M. Dawson, "Object tracking by a robot manipulator: a robust cooperative visual servoing approach," Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292), 2002, pp. 211-216 vol.1, doi: 10.1109/ROBOT.2002.1013363.
2. R. K. Megalingam, N. Katta, R. Geesala, P. K. Yadav and R. C. Rangaiah, "Keyboard-Based Control and Simulation of 6-DOF Robotic Arm Using ROS," 2018 4th International Conference on Computing Communication and Automation (ICCCA), 2018, pp. 1-5, doi: 10.1109/CCAA.2018.8777568.
3. M. Ito and M. Shibata, "Non-delayed visual tracking of hand-eye robot for a moving target object," 2009 ICCAS-SICE, 2009, pp. 4035-4040.
4. N. P. Papanikolopoulos, P. K. Khosla and T. Kanade, "Visual tracking of a moving target by a camera mounted on a robot: a combination of control and vision," in IEEE Transactions on Robotics and Automation, vol. 9, no. 1, pp. 14-35, Feb. 1993, doi: 10.1109/70.210792.
5. Gazebo Sim v11.0.0 *(http://gazebosim.org/)*
6. ROS Noetic *(http://wiki.ros.org/noetic)*
7. ros_kortex – ROS packages for KINOVA KORTEX robotic arms *(https://github.com/Kinovarobotics/ros_kortex)*

Authorship Table

Bhushan Ashok Rane	Worked on Joint Velocity Controller in ROS and it's integration with the manipulator
Gaurang Manoj Salvi	Worked on formulation of Inverse Jacobian to get desired joint velocities
Rutwik Rajesh Bonde	Worked on framing Image Jacobian to get camera velocity
Yash Rajendra Patil	Worked on integration of vision module with robot
Collaborative Effort	Literature Survey, Report Drafting and Creating Presentation