Grasping of moving objects by assistive robotic arm

Student: José Angel Melendez

Professors: Sofiane Achiche, Maxime Raison, Yann S.L. Kam Cio

1. Introduction

Many activities of daily living require proper control and usage of upper extremities. People with conditions such as spinal cord injuries and neuromuscular diseases are unable to perform properly in these activities. In order to assist patients with this conditions, robotic solutions have been implemented in the form of robotic arms controlled through a joystick.

Specifically, Kinova developed JACO (shown in Fig.1), a 6 DOF assistive robotic arm that can be placed on wheelchairs and can be controlled by users through a joystick. Studies have shown that its use is in fact efficient and represents a good "alternative for increasing the autonomy of individuals with upper extremity disabilities." [1]

However, the operation of such robots requires the user to be dexterous with the controller, and therefore may be too demanding for certain users, causing the object manipulation to be too slow. In this context, "a major current challenge is fast automated scene analysis using vision and artificial intelligence for object prehension by an assistive robot." [2]

Bousquet-Jette et al. (2017) describe in their paper a scene analysis technique for robotic object prehension using machine vision and a JACO robotic arm, in which a scene is scanned by a Kinect and further analyzed by AI algorithms to determine pose of objects, and calculating the prehension pose of the robot. The system requires on average 0.6 s to analyze an object in a scene.

The technique employed is effective for scene analysis and prehension because it is not a real time application, meaning that the scene is static and does not change from the time that it is evaluated to the time that the robot grasps a specific object. Being this said, grasping of moving objects is still a challenge to assistive robotics.

This project proposes a solution to the grasping of moving objects by a robotic arm, specifically with JACO, by using real time 3D object detection with a standard webcam, trajectory prediction, and robotic arm movement.

Figure 1. JACO arm.

2. Problem solution

To simplify the problem, the object to grasp used for this project is a yellow ping pong ball. Using an object of known size and shape allows us to use a simple webcam to obtain XYZ coordinates of an object, rather than using a stereo or 3D camera. Being this said, the challenge of the project can be stated as to achieve optimal time grasping of a moving ping pong ball by the JACO arm. The diagram shown in Fig. 2 represents the steps proposed to achieve the goal.

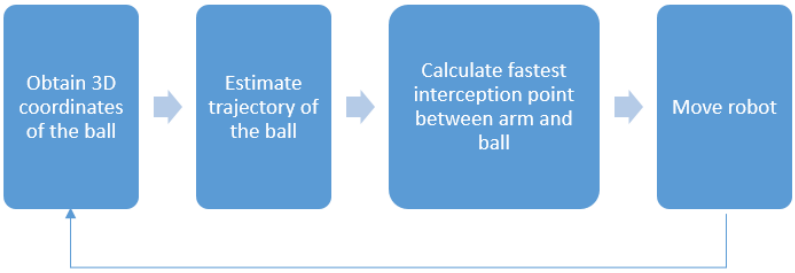

Figure 2. Algorithm diagram.

2.1 Obtain 3D coordinates of the ball

The mission of this part of the system is to obtain xyz coordinates of the ball. Several tracking methods like optical tracking, mean shift, cam shift, etc. were evaluated, but had major problems with drifting and run time, as they are not able to perform in real time (<40ms for 24fps) in an average PC.

Besides, a method had to be devised to obtain 3D coordinates of the ball from a standard webcam. As the object to be tracked has a fixed shape and dimensions, distance can be obtained from the perceived diameter of the ball. This being said, the algorithm for ball detection must also determine the diameter of the ball as accurately as possible.

There is an algorithm called “simple blob detector” [3], which extracts blobs from an image and obtains their size and shape, accordingly. The algorithm is described as follows:

1. Convert the source image to binary images by applying thresholding with several thresholds.
2. Extract connected components from every binary image and calculate their centers.
3. Group centers from several binary images by their coordinates. Close centers form one group.
4. From groups, estimate final centers of blobs and their radiuses.
5. Filter blobs according to circularity, area, inertia, convexity, etc.

Fig. 3 shows the effects of the simple blob detector over the picture of a flower. It can be seen that the algorithm detects two blobs with different diameters in each pass, and at the end it averages their radiuses and locations to obtain two blobs. After that, one can discriminate blobs by size or shape.

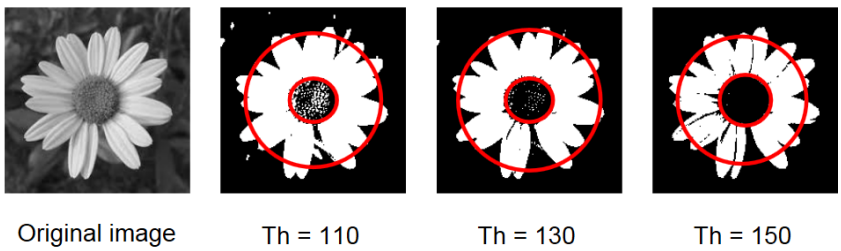

Figure 3. Simple blob detector algorithm example.

As it can be shown in Fig. 3, the algorithm is good at calculating position and diameter of an object because it does it in several thresholds and at the end averages the result. However, the perquisite of this algorithm is that the image must be a grayscale image, and that the target to detect must be in a significant grayscale range that is different to its background.

Being this shown, simply applying a grayscale transformation to an image of a ping pong ball will not have good results because the transformation takes in account the brightness of the pixels and not in particular the color, so many colors can have the same grayscale value and thus the algorithm may in occasions not perceive adequately the ball.

I devised an algorithm that performs a grayscale transformation to an image in a way that pixels which are closer to certain color are darker than those of different colors.

To explain this, one must understand the principle of color distance. Pixels have RGB values, which describe the amount of red, green and blue in the pixel, respectively. However, like RGB, different color scales exist (HSV, HSL, CMYK, Lab, Luv, XYZ, etc.), as well as equations that relate RGB scale to the different scales. Fig. 4 shows RGB, HSV and Luv colorspaces.

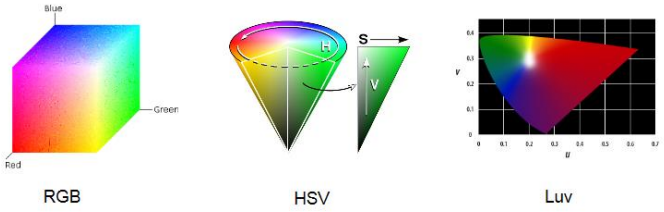

Figure 4. Colorspaces RGB, HSV and Luv.

In the Luv colorspace, every pixel has three values (L, u and v). The "L" value refers to the luminosity of a specific pixel, while u and v values can be graphed in a plane and can be viewed as coordinates. [4] The color difference between two colors can be calculated using the Euclidean distance of the (u, v) pairs. This means that if two colors are visually alike, they will have a small Euclidean distance, while different colors will have a large Euclidean distance.

I hereby propose an image transformation that takes as input a colored image and a target color, and yields as an output a grayscale image in which the darkest pixels correspond to pixels in the original image whose color is near to the target color. First every pixel is changed from RGB to Luv, which can be performed by OpenCV. [5] Then Eq. 1 is applied to every pixel to get a new image.

$$p(L,u,v) = \sqrt{u^2 + v^2} \tag{1}$$

By applying this transformation to every pixel in an RGB image, one can obtain a grayscale image. As it can be seen in Fig. 5, the pixels corresponding to the ball are clearly darker. This image can now be processed by the simple blob detector algorithm and yield proper xy pixel coordinates and pixel diameter of the ball.

Figure 5. Color distance grayscale transformation.

This algorithm runs in 10ms on an average computer on a 640x480 resolution image. This makes it feasible for use in real time applications. The system can now obtain pixel coordinates and diameter of the ball accurately under different lighting condition. This algorithm also discriminates objects by circularity, thresholds, area, inertia and convexity, and these parameters have been tuned to eliminate the most possible number of false positives. This video shows the algorithm running in real time.

Next, we most obtain real world x, y and z coordinates based on pixel location and diameter of the identified object. The coordinate frame p has pixel units and represents the output of the ball detection algorithm described above. The coordinate frame r has meter units and represents the real location of the ball in respect with the webcam, which is located in the origin (0, 0, 0) and facing in the direction of rz. Fig. 6 illustrates the stated coordinate frames.

Figure 6. Coordinate frames.

When the ball is detected by the algorithm, the ball diameter in pixels is calculated. With this information, the real diameter of the ball, and the focal length of the webcam, one can estimate the rz coordinate of the ball using Eq. 2. The focal length of the webcam can be obtained experimentally by measuring the real distance from the webcam to the ball.

$$rz = \frac{realDiameter \times focalLength}{pixelDiameter} \quad (2)$$

Finally, x and y coordinates can be obtained using Eq. 3 and 4, being width and height the resolution of the webcam used:

$$rx = \frac{rz \times (px - \frac{width}{2})}{focalLength} \quad (3)$$

$$ry = \frac{-rz \times (px - \frac{height}{2})}{focalLength} \quad (4)$$

This way, the first step of the system is complete, as it outputs the x, y and z coordinates of the ball. This video shows the execution of this algorithm and the visualization of x, y and z coordinates in meters, as well as a 3D visualization in RViz, which will be explained later in this report.

2.2 Estimate trajectory of the ball

In order to correctly estimate the trajectory of the ball, one must choose a model. As the project's aim is for the assistive robotic arm to grip a moving object, this object is either being handed to the arm, the wheelchair is moving towards or away the object, or at most being thrown to the arm. This restricts the trajectory of the object to being uniformly accelerated motion.

This being stated, the ball will be modeled as having uniformly accelerated motion. To further estimate the trajectory, one must know, in real time, what the velocity, and acceleration vectors of the ball are, according to the vector equations 5 and 6.

$$\boldsymbol{v}(t) = \frac{\boldsymbol{r}(t) - \boldsymbol{r}(t-dt)}{dt} \tag{5}$$

$$\boldsymbol{a}(t) = \frac{\boldsymbol{v}(t) - \boldsymbol{v}(t-dt)}{dt} \tag{6}$$

A low pass filter is applied to the values of velocity and acceleration in each iteration to filter out random motion and noise. This video shows the velocity and acceleration vectors graphically for the ball in real time. The filtered values are then input in Eq. 7, which can estimate the position of the ball at any given time t.

$$\boldsymbol{r}(t) = \boldsymbol{r}(t_0) + \boldsymbol{v}(t_0) * t + \frac{1}{2}\boldsymbol{a}(t_0)t^2 \tag{7}$$

2.3 Calculate fastest interception point between ball and robot

The next step in the algorithm is to calculate, according to the current position of the end effector of the robot and its maximum linear velocity, what is the fastest time in which it can intercept the trajectory of the ball. Fig. 7 helps visualize this.

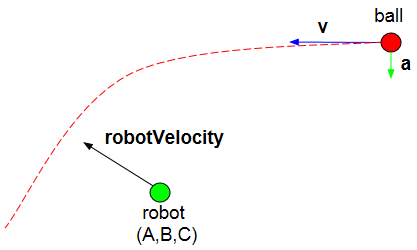

Figure 7. Visualization of predicted ball trajectory.

Instead of the robot running after the ball, which will make it impossible to catch it if the robot speed is slower than the ball (which will happen because assistive robots have low speed), the robot needs to intercept the ball in the minimum time possible.

The distance between the robot and the ball, given that the robot remains there, at any given time, is described by Eq. 8.

$$d(t) = |\boldsymbol{r}(t) - (A, B, C)| \tag{8}$$

Now, the time that will take the robot to move towards the ball starting in (A, B, C) and ending in the predicted ball position at time t, moving at top speed is given by Eq. 9.

$$l(t) = \frac{d(t)}{robotVelocity} \tag{9}$$

Being this said, if the robot starts moving at t0 towards a given predicted position of the ball r(t), it will be there in time l(t), and the ball will arrive there at t. Thus, when the robot is moving as fast as it can, it will intercept the ball when the condition in Eq. 10 is met.

$$i(t) = t - l(t) == 0 \tag{10}$$

Finally, using secant method for zero crossing, one can numerically compute time t when this will happen, and input this time to Eq. 11 to calculate the resulting interception coordinates.

$$\boldsymbol{r}(t) = \boldsymbol{r}(t_0) + \boldsymbol{v}(t_0) * t + \frac{1}{2}\boldsymbol{a}(t_0)t^2 \tag{11}$$

Each iteration, this interception point is calculated and the robot moves towards there. This video shows the algorithm working in 2D, and this other video shows it in 3D, with the robotVelocity parameter set to the one of the JACO arm.

An interesting tweak of this algorithm is that if the robotVelocity parameter is tuned to be smaller than the actual velocity of the robot, the robot performs interception tangential to the ball trajectory, which is very useful while trying to grasp objects. Fig. 8 further illustrates this concept.

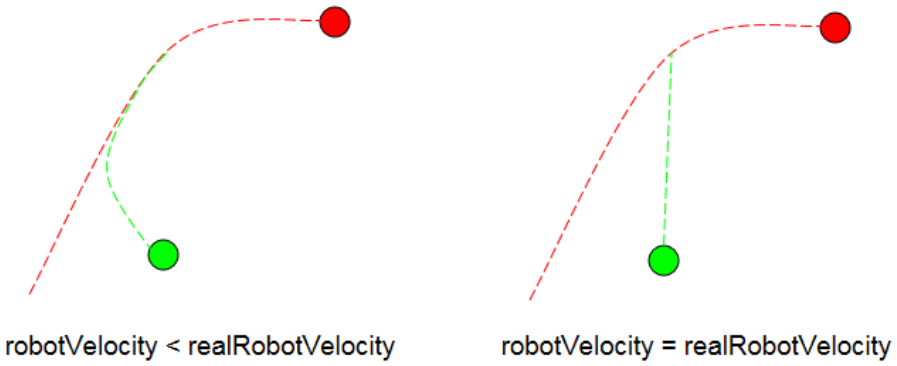

Figure 8. Different types of interception.

2.3 Robot movement

Once the interception point is calculated, the robot must move towards there. In the next iteration, the point will update and the robot needs to change its objective to that new point. Ideally, if the tracking of the ball was flawless, the interception point never changes and the robot only needs to calculate the interception point once. However, since there is noise in the measurements and ball trajectory is a mere estimation, as the act of handling a ball has no specific trajectory, the interception point needs to be calculated in each iteration.

Moving a robotic arm to specified coordinates is no trivial task, but MoveIt! provides the necessary framework and IK engines for it to seem as so. Every iteration, the robot is commanded a new target position, and fed to the IK engine. Once a solution is found, it starts moving towards there in the shortest route possible.

3. Solution implementation

The implementation of the solution and integration with the robot was done using the ROS environment. ROS (Robot Operating System) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. [6]

ROS allows the creation of nodes that communicate with each other using messages. Any node may publish a message to a topic, and any other node is able to subscribe to this topic to read messages as they are published.

RViz is the 3D visualizer of ROS for displaying sensor data and state information from ROS. [7] This is helpful for the project because we can visualize a virtual robot here without having one, as well as visualizing the ball in 3D in real time.

MoveIt! is a state of the art software for mobile manipulation, incorporating the latest advances in motion planning, manipulation, 3D perception, kinematics, control and navigation. It is a framework that can be integrated into ROS. [8]

The diagram in Fig. 9 illustrates the different nodes needed for this project and the interactions between them.

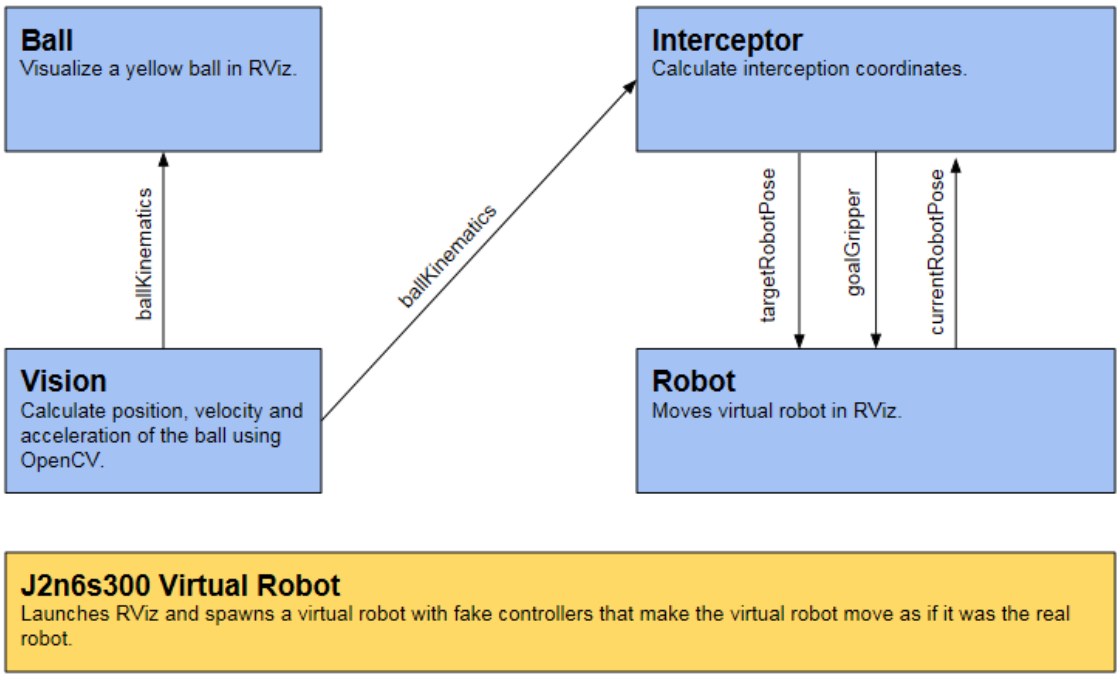

Figure 9. ROS nodes and message topics of the solution implementation.

Blue boxes represent nodes, arrows represent messages, and the yellow box represents a launch file given by Kinova Robotics, the creators of JACO arm. All nodes were coded in C++. This video shows the full implementation of the project running in the JACO arm.

3.1 Vision node

The code in this node grabs video from a webcam at a rate of 24 fps, and runs the algorithm described in this paper for position, velocity, and acceleration estimation of a ping pong ball. Once they are calculated, a window is created that displays in real time the image captured and, if a ball is detected, a red dot indicating the position of the ball and the XYZ coordinates of it. Finally, position, velocity and acceleration are filtered using a moving average filter and sent as a message to the *ballKinematics* topic.

This node contains parameters that can be tuned regarding the blob detection algorithm and the low pass filters.

3.2 Ball node

This program basically initializes a marker, which is a data type of RViz, with the shape and size of a yellow ping pong ball, and loops polling the *ballKinematics* topic. Whenever there is a message in the topic, the node updates the marker's position and publishes it to RViz, so that the ball can be visualized in 3D.

This node contains parameters that can be tuned regarding the appearance of the ball.

3.3 Interceptor node

The interceptor node polls the *ballKinematics* and *currentRobotPose* topics. Whenever the ball is in reach, the interception pose in which the robot will intercept the ball in optimal time is calculated using the algorithm described before in this paper. Then, a low pass moving average filter is applied to the interception pose and further sent as a message to the *targetRobotPose* topic. If the ball is already in the gripper of the robot, it sends a message to the *goalGripper* topic indicating to close the fingers.

The orientation of the robot was intended to be always tangential to the ball trajectory and facing the ball, but due to the fact that it is updated in real time, and to the kinematic characteristics of the robotic arm, for the robot to correct a few degrees of pose, the joints need to move a lot and deviates the robot from its path. However, this feature can be added to faster robots or robots with different configurations just uncommenting some lines. For now, the robot's end effector orientation is constant.

When the ball is detected out of reach of the robot, the robot moves as far as it can to reach it and wait for the ball to be in reach.

This node contains parameters that can be tuned to improve performance, such as the low pass filter window and the robotVelocity parameter, which represents the velocity used by the algorithm to calculate the interception point.

3.4 Robot node

The robot node starts by sending the robot to home position. It then polls the *goalRobotPose* and *goalGripper* topics. Every second, it updates the Inverse Kinematics engine to calculate a trajectory from the current pose to the goal pose, and commands MoveIt! to move the robot (either virtual or physical). It also publishes to the currentRobotPose topic the actual configuration of the robot.

Whenever the robot is commanded to close the fingers (which means it grabbed an object), the robot is commanded to move to the home position and open the gripper at the end.

This node contains parameters relating the home position and overall orientation of the end effector that can be tuned according to the positioning of the robot.

3.5 J2n6s300 virtual robot file

Running this file does various things. First, it runs RViz for visualization. Then, a virtual robot is spawned in the visualizer. Finally, the most important part, is that fake controller nodes are initialized. This generates listeners that will move the virtual robot as if it was a real robot, with real velocities and PID parameters. If one wants to run this project using a real robot in parallel or without the virtual robot, just needs to connect it and maybe change a few lines in the robot node. I am not sure as I had no access to the real robot, but I am certain that it is possible.

All the files needed for this program to run are contained in the kinova-ros folder, obtained from the official repository of Kinova, the creators of JACO arm. [9]

4. How to run project

Follow the installation manual to install all the dependencies and to download the project code. The installation manual is located here.

The workspace folder "melendez_ws" contains everything regarding this project. In it there is two folders, the "kinova-ros" and the "pingpong" folder. The first folder is read only and one must not modify anything in it. All my implementation is in the pingpong folder. Within the pingpong folder, the source code is in the "include" and "src" folders. Inside the "launch" directory, there is a file that runs all nodes and the launch file provided by Kinova. By launching our launch file, the project is run. To do so:

1. Make sure to have all the required libraries installed in your computer. I wrote a manual on how to install everything, and with tutorials of the components.
2. Move the "melendez_ws" folder to your home directory
3. Run the following commands on a terminal

```
$source ~/melendez_ws/devel/setup.bash

$roslaunch pingpong demo.launch
```

5. Conclusions and future work

A solution to gripping moving objects is proposed. As shown in the video, the arm effectively intercepts objects, grips them, and returns home. Also, a solution to real time 3D tracking of a ping pong ball has been proposed.

This solution is structured modularly in nodes using ROS. This is useful because if one wants to identify other type of objects, only modify vision node and make it send the object's position, velocity and acceleration as fast as your algorithm can, and other nodes will take care of the rest. If you want to command a different robot, only change robot node. And so on and so on.

This project will serve as a basis to further research, and possibly one day integrated to assistive robots. In the meantime, more research has to be conducted. It needs to be tested and debugged on a real JACO arm. The orientation of the end effector also needs work, in order to achieve tangential orientation to the ball's trajectory. Furthermore, the whole trajectory of the robot needs to be optimized because there are cases in which the IK engine yields a solution that involves the arm moving too much.

6. Bibliography

[1] V. Maheu, J. Frappier, P. Archambault and F. Routhier, "Evaluation of the JACO robotic arm," *IEEE International Conference on Rehabilitation Robotics*, 2011.

[2] C. Bousquet-Jette, S. Achiche, D. Beaini, Y. Law-Kam Cio, C. Leblond-Ménard and M. Raison, "Fast scene analysis using vision and artificial intelligence for object prehension by an assistive robot," *Elsevier*, 2017.

[3] OpenCV, "Simple Blob Detector," 2017. [Online]. Available: http://docs.opencv.org/master/d0/d7a/classcv_1_1SimpleBlobDetector.html.

[4] Adobe, "Technical Guides," 2000. [Online]. Available: http://dba.med.sc.edu/price/irf/Adobe_tg/models/cieluv.html.

[5] OpenCV, "Miscellaneous Image Transformations," 2017. [Online]. Available: http://docs.opencv.org/3.1.0/d7/d1b/group__imgproc__misc.html.

[6] "Robot Operating System," 2017. [Online]. Available: http://www.ros.org/.

[7] "ROS Visualization," 2017. [Online]. Available: http://wiki.ros.org/rviz.

[8] "MoveIt! Motion Planning Framework," 2017. [Online]. Available: http://moveit.ros.org/.

[9] "Kinova ROS," 2017. [Online]. Available: https://github.com/Kinovarobotics/kinova-ros.