Summer Projects
Science and Technology Council
IIT Kanpur

DUM-E

Documentation

Date of Submission:

July 18, 2020

Contents

1 Acknowledgements

We are immensely grateful to Robotics Club, IIT Kanpur and Science & Technology Council, IIT Kanpur for providing us with the resources and motivation for this Summer Project. We thank our mentors, and the Coordinators of Robotics Club: Arunim Joarder, Ramyata Pate, Prakash Choudhary, and Suman Singha for their guidance, constant support and invaluable inputs, without which we would not have been able to do this project in a timely manner.

2 Introduction

2.1 Problem Statement

Our aim is to create a virtual simulation of a robotic arm which can detect, and mimic the motion of a reference arm. We aim to achieve this using open source resources, including the models of the Microsoft Kinect depth sensing camera, OpenCV (for computer vision), Gazebo (simulation platform), and ROS (robot operating system). DUM-E involves the following major tasks

1. **Handling input data from Microsoft Kinect** (References- Gazebo tutorials)
   This task can be divided into 2 goals-
   (a) Detection using tags and markers
   (b) Detection using ML algorithms to identify human arms and its various joints

2. **Calculation of Joint Angles** (References- Inverse Kinematics)
   This is done by using Inverse Kinematics to calculate joint angles, given information about joint positions.

   We used python to do the necessary calculations for IK. The inverse kinematics problem can be solved in a very intuitive way if you use Homogeneous Coordinates and Transformations as a way of representing the position of each point of the robot. For getting this done we worked with matrices in python, and therefore we used the numpy library for all calculations. To install `numpy` for python3 enter the following commands into the terminal:

   ```
   $ sudo apt install python3-pip
   $ pip3 install numpy
   ```

2.2 Brief Approach

A depth sensing camera (Microsoft Kinect) is used for detection of the arm movements. We import 2 robotic arms and the depth sensing camera in Gazebo. The first robotic arm (say A) representing the human arm is sent its required final pose using a publisher (controlled by us). Data input from the camera is used to detect the position of joints on the robotic arm A. Using the data input from the camera we get the desired pose for the second robotic arm (say B). Applying inverse kinematics to the given data, we get the angles required to get robotic arm B to desired position.

2.3 Impact

It is hard to fully express the overall impact of robotic arms. In one way or the other robotic arms are omnipresent. This particular robotic arm which can replicate a human hand will find applications in the sectors wherever positioning of a human harm is necessary but hazardous. Some such sectors are-

- Space exploration
- Surgeries in Medical field
- Material Handling Operations(Welding and Molding)
- Military Operations-

- Bomb disarmament and Disposal
- Demining

3 ROS Setup

3.1 Linux

(This one is the preferred OS)
Most of the people work with Windows OS. You dont need to boot the Linux OS entirely. There are 2 ways to operate Ubuntu from inside Windows. These will allow to quickly come in and out of the Ubuntu shell without slowing down your Windows host:

- **Docker image based on Ubuntu with ROS pre-installed**
  - For the Docker method go to ROS through Docker.
- **WSL (Windows Subsytstem for Linux)**
  While both the methods are applicable the WSL method is much more preferred beacuse Docker is not much integrated with the Windows platform which is hosting it.
  - For the WSL method(preferred) go to ROS through WSL.

The above link will guide you to the process for installing Ubuntu and ROS on your Ubuntu shell. If you are using Ubuntu shell for the first time, don't worry just implement the steps exactly. If you want to learn more about Shell commands and language check out More about Ubuntu.

3.2 Windows

ROS can also work on Windows. Follow the below link for the proper setup:
ROS Installation on Windows.

3.3 Tutorials for ROS

The roswiki website is like your constant companion throughout the project. If you ever get stuck, roswiki has everything to sort out the problems. Here is a link for tutorials that would help you in getting familiarized with ROS and its concepts: ROS Tutorials.

They are divided into 3 sections depending upon the your knowlege of ROS. A pre-requisite for ROS is a basic knowledge of either C++ or Python 2(or Python 3). These tutorials only give a basic introduction to the ROS filesystem and ROS concepts, just enough so that you can start your work. For more information, you can watch the lecture series on ROS by ETH Zurich: Video Lecture Series on ROS.

If somehow you are stuck with ROS software and just want to solve some temporary problem, this online version of ROS can temporarily help you: Online ROS Development Studio. **Note that this online ROS studio has less computation power**.

4 ROS basics (1)

4.1 Workspace

To put it in simple words, workspace is just a directory inside your ROS filesystem. It contains all the packages. What are packages? A software in ROS is made of packages. Packages can contain nodes,libraries,datasets and much more. After all the initialization is done and you have a catkin workspace, you will find 3 folders inside it. These 3 folders are generally present inside any legitimate workspace:

- `build`: This is where Cmake and Catkin are invoked to build the packages in the `src` directory
- `devel`: This is where build targets are stored
- `src`: Contains all your packages

For detailed information, go to Catkin Workspace.

4.2 CMakeLists.txt

The `CMakeLists.txt` is the file that contains information that is given to the Cmake build system to build the packages. It is present in every package. For further details about the structure of `CMakeLists.txt` and it functions, check out CMakeLists.

4.3 XML file

So what is an XML file? To put it in simple words, XML files are files that contain information in a very specific manner. XML files, like HTML, contain tags. So for example:

```
<book>
    <name id="4"> Harry Potter and the Goblet of Fire </name>
    <author> J.K Rowling </author>
    <publisher> Bloomberg </publisher>
</book>
```

The above is a very simple XML file. It contains information about a book. These tags are not universal. They are created by the programmer. Yes, you guessed it right!!! We need to know that this XML file contains data about a book before we can parse and extract data.

For more information you can refer to Xml Introduction.

Similarly the `package.xml` file contains metadata(i.e information about the data) like authors, version, dependencies and so on.
For more details, go to Package.xml.

4.4 Subscribers and Publishers

Subscribers and publishers are nodes. "Node" is the ROS term for an executable that is connected to the ROS network. In this project, we created the publisher ("talker") node which publishes the value of angles that we want the joints to go to using the `trajectory_msgs/JointTrajectory` message on to the topic `/custom/arm_controller/command`.
For more information you can refer to Subscribers and publishers

4.5 Launch Files

ROS launch is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the `.launch` extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.
The roslaunch package contains the roslaunch tools, which reads the roslaunch `.launch`/XML format. It also contains a variety of other support tools to help you use these files.
Many ROS packages come with "launch files", which you can run with:

```
$ roslaunch package_name file.launch
```

These launch files usually bring up a set of nodes for the package that provide some aggregate functionality.
References- Launch files

4.6 ROS Parameters

Parameters are named using the normal ROS naming convention. This means that ROS parameters have a hierarchy that matches the namespaces used for topics and nodes. This hierarchy is meant to protect parameter names from colliding.
The ROS Parameter Server can store strings, integers, floats, booleans, lists, dictionaries, iso8601 dates, and base64-encoded data. Dictionaries must have string keys.
References- ROS Parameters

4.7 rqt plugins

Rqt is a handy debugging tool to check the connections of nodes and topics in our ROS environment. We used the

```
$ rosrun rqt_graph rqt_graph
```

command to check these connections. For more info refer- rqt

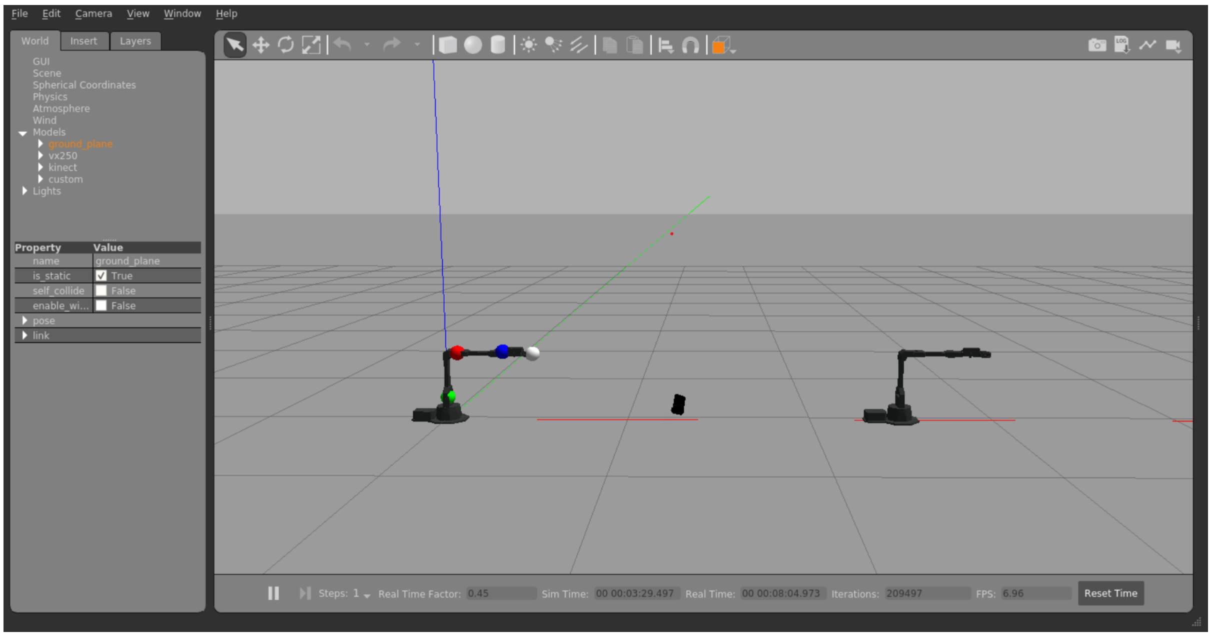
5 Simulation Environment

5.1 Gazebo Robot Simulator (2)

Gazebo is a open source 3D robot simulator. Gazebo integrated the ODE physics engine, OpenGL rendering, and support code for sensor simulation and actuator control. In this project we use a Gazebo world for our robotic simulation. Our world contains the two robotic arms corresponding to a human arm and the controlled robot arm, and also a depth sensing camera. We use the camera for joint detection of the robot corresponding to the human arm.

Gazebo tutorials:-

- Installation
- Building a World
- Building a Robot



Property	Value
name	ground_plane
is_static	✓ True
self_collide	False
enable_wi...	False

Steps: 1 Real Time Factor: 0.45 Sim Time: 00 00:03:29.497 Real Time: 00 00:08:04.973 Iterations: 209497 FPS: 6.96

Figure 1: Our Gazebo world

5.2 Interbotix Ros Arms

For our Gazebo simulation environment we used the interbotix-ros-arms (5) package which works along with a robotic manipulation library Modern Robotics(6). This package provides descriptions and urdf files for many different robotic arms from ViperX and WidowX series.

- **interbotix_description**: contains the meshes and URDFs (including accurate inertial models for the robot-arm links) for the arms and turrets
- **interbotix_gazebo**: contains the config files necessary to launch a robot arm (not turret) model in Gazebo, including tuned PID gains for ros_control

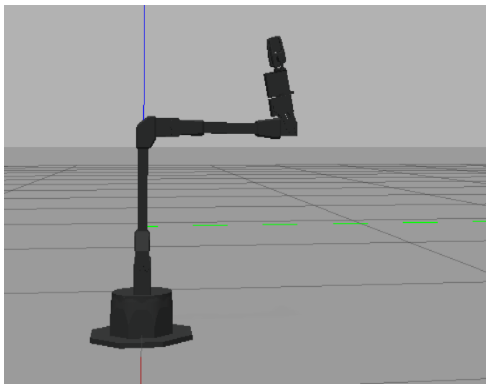
The first robotic arm contains markers attached to its joints for detection and image processing through our camera.

5.2.1 Adding markers to robot joints

For markers on the joints we have added a spherical fixed link of a specified color. To do this we made changes to the '`.urdf.xacro`' file of the robot (also an XML type file). The code snippet for adding the spherical markers is written below:

```
<link name="$(arg robot_name)/marker">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <sphere radius="0.04"/>
    </geometry>
    <material name="red"/>
  </visual>
</link>

<joint name="marker_joint" type="fixed">
  <parent link="$(arg robot_name)/forearm_link"/>
  <child link="$(arg robot_name)/marker"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
</joint>
```


(a)

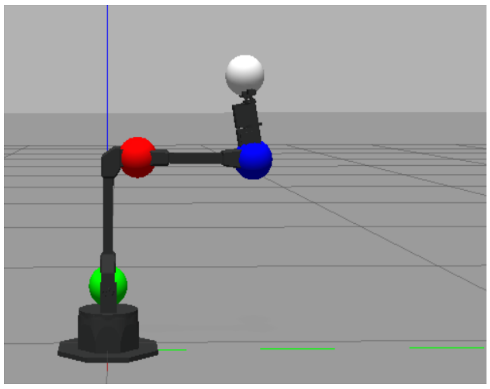

(b)

Figure 2: (a) Default arm in Package (b) Customised arm with Markers

5.3 Openni Kinect

Kinect is a line of motion sensing input devices produced by Microsoft and first released in 2010. The device incorporates RGB cameras, infrared projectors and detectors that mapped depth through either structured light or time of flight calculations. We use the standard openni kinect plugin in our Gazebo world
In our project, we use both the RGB as well as depth image from the camera for our processing.

5.4 Joint Trajectory Controller

joint_trajectory_controller is a ROS Controller for executing joint-space trajectories on a group of joints. Trajectories are specified as a set of waypoints to be reached at specific time instants,

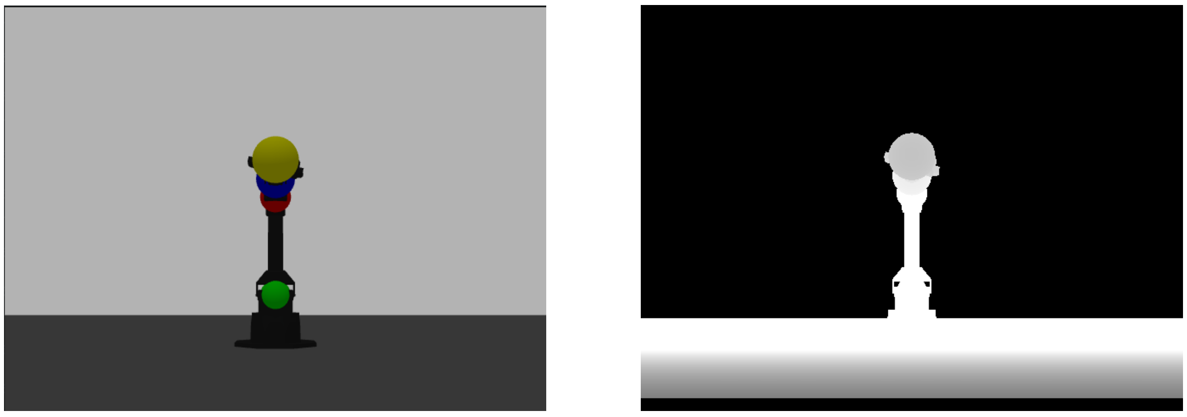

(a) (b)

Figure 3: (a) Color image output from Kinect (b) Depth image output from Kinect

which the controller attempts to execute as well as the mechanism allows. Waypoints consist of positions, and optionally velocities and accelerations. For our robot arm, we provide joint angles for the final position. These final positions are used by the joint_trajectory_controller for moving the arm to the final position.

6 ROS Architecture

Rqt graph is a very easy and efficient way to figure out how the nodes are reacting with each other, which nodes are involved in the whole process, who is subscribing to which topic and who is publishing to a particular topic.
In the ellipses are the names of the nodes. The topic through which a particular pair of nodes are interacting is written on the arrow.
The type of a topic is defined by the type of the message that is being published into it. A publisher node publishes a particular type of message and a subscriber node extracts what is written on the topic. The arrows denote the flow of messages. An arrow from a node to a topic means that the node is publishing and similarly an arrow from a topic to a node means that the node is subscribing.

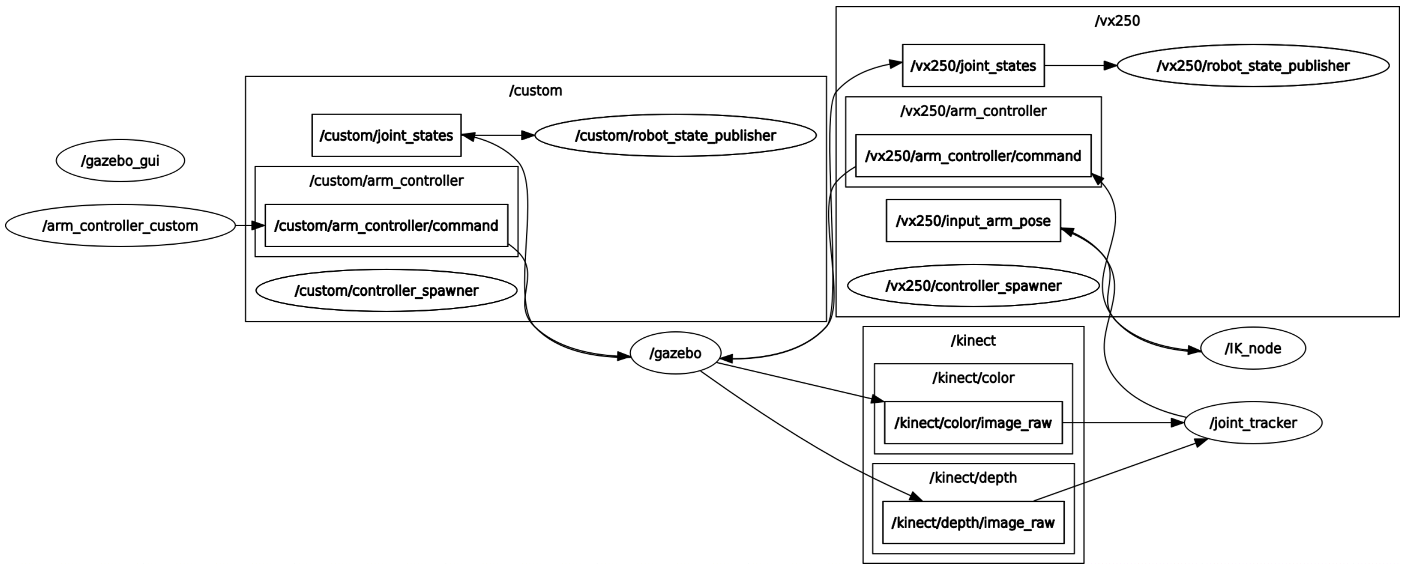


Figure 4: rqt Graph

A brief description of each node is given below,

- `/gazebo_gui`
  No topics are published or subscribed to yet. Everything is running in the Gazebo environment. Both the arms and the camera are in the world file of gazebo.

- `/arm_controller_custom`
  Topics published:
  - `/custom/arm_controller/command`

  We feed the angles into the reference arm.The arm moves so as to obtain that pose.

- `/gazebo`
  Topics subscribed:
  - `/custom/joint_states`
  - `/custom/arm_controller/command`

  Topics published:
  - `/kinect/color/image_raw`
  - `/kinect/depth/image_raw`

What happens here is that the Kinect camera observes the refernce arm through markers and provides the depth coordinates.All the coordinates of the joints are then sent to the `/joint_tracker` node which publishes the coordinates into the `/vx250/input_arm_pose`.

- `/IK_node`
  Topics published:
  - `/vx250/arm_controller/command`

  Topics subscribed:
  - `/vx250/input_arm_pose`

  This node calculates the joint angles using 3D coordinates of the joints and Inverse Kinematics.

- `/joint_tracker`
  Topics published:
  - `/vx250/arm_controller/command`

  Topics subscribed:
  - `/kinect/color/image_raw`
  - `/kinect/depth/image_raw`

  This node employs the Kalman Filter on the joint positions to send joint angles to the arm.

- `/custom/robot_state_publisher`
  Topics subscribed:
  - `/custom/joint_states`

  This node publishes the robot's state for RViz.

- `/vx250/robot_state_publisher`
  Topics subscribed:
  - `/vx250/joint_states`

  This node publishes the robot's state for RViz.

- `/custom/controller_spawner`

7 Perception

7.1 Joint Detection

To localise the spatial position of the reference robotic arm, we used our depth camera to detect the three-dimensional position of the joints of the arm. We placed differently colored spherical markers on the joints of the robotic arm and used color detection techniques to decipher their position in space. A step-by-step breakdown of the process implemented to achieve this is given below:

- **Convert the BGR image inputted from the Kinect Camera to HSV**: We do this because it is more efficient to detect color in an HSV image than in a BGR image.
- **Threshold the HSV Image to Detect the Different Colors**(7): We threshold the HSV values for the image to detect the colored markers. Once detected, a mask is applied on the colored markers in the image.
- **The Center of the Colored Portion of the Image is Calculated**: This is the location of the joint. We calculate the center of the colored portion by finding contours of the mask, and then calculate its centroid.
- **Find the 3D Position of These Points**(8): Once we have the two dimensional information of the location of the different joints of the reference arm, we find the depth of those point using the depth data provided by the Kinect Camera.

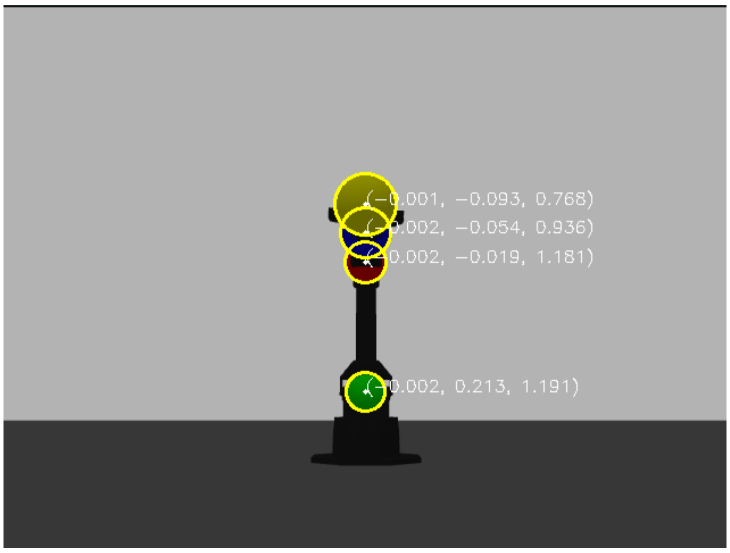


Figure 5: Image after detecting joints

7.2 State Estimation

One major limitation of the above method (other than its reliance on colored markers for tracking) is its reliance on all the markers to be visible at all times. This is obviously highly impractical. The visibility of any particular marker can get obscured by other markers or the links of the robotic arm itself. Hence, when we are unable to detect the colored markers it is desirable to estimate their position on the basis of their previous movement patterns. This is

what 'State Estimation' refers to in this context.
We can use the Kalman Filtering technique (this is also known as linear quadratic estimation) to implement this state estimation computation. Kalman filtering is a recursive algorithm which uses data (containing statistical inaccuracies) and makes a prediction. The algorithm has two steps. These are the prediction step and the updating step. The prediction step predicts the state variables with their uncertainties. These predictions are updated in the updating step by observing the variables in the next measurement. This process continues recursively, generally yielding more accurate predictions over the longer the process runs.
A good resource to learn more about the Kalman Filter and its applications is:

- Kalman Filter Lesson 1
- Kalman Filter Lesson 2

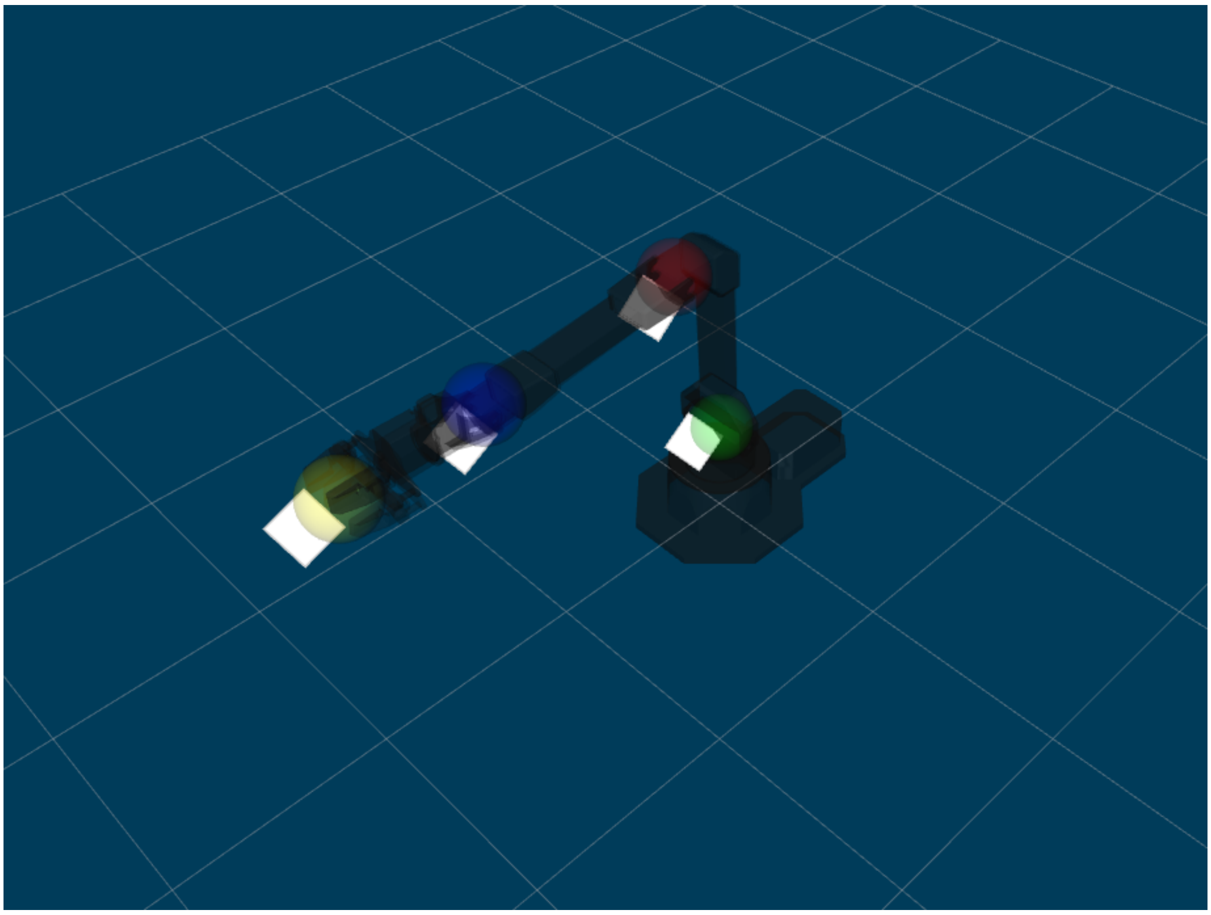


Figure 6: RViz model to check model behaviour.
The white squares represent current estimate of marker position

7.3 Inverse Kinematics (3)

Once we have obtained the position of the joints in three dimensional space, we need to obtain the variable joint parameters. This is important because we need to communicate these parameters to our primary robotic arm to achieve the desired movement. Calculating these variable joint parameters is called inverse kinematics.
The parameters we have employed in this project are the angles between successive links. These

parameters describe the spatial position of the arm entirely, and also are obtainable just from the spatial position of the joints: which we have already detected as described above.
To obtain the desired angles from the positions we switching through frames using transformation matrices.

8 Future Aspects

8.1 OpenPose - for Human Arm Detection(4)

OpenPose is a Realtime Multi-person 2D pose estimation and joint detection algorithm that uses Convolutional Neural Networks and is built on the caffe framework. OpenPose represents the first real-time multi-person system to jointly detect human body, hand, facial, and foot keypoints (in total 135 keypoints) on single images.
Our future goal is to include this library on top of our model to facilitate joint detection of human arms and subsequently make the hardware act according to that. We hope to use this along with ROS and extract relevant arm information

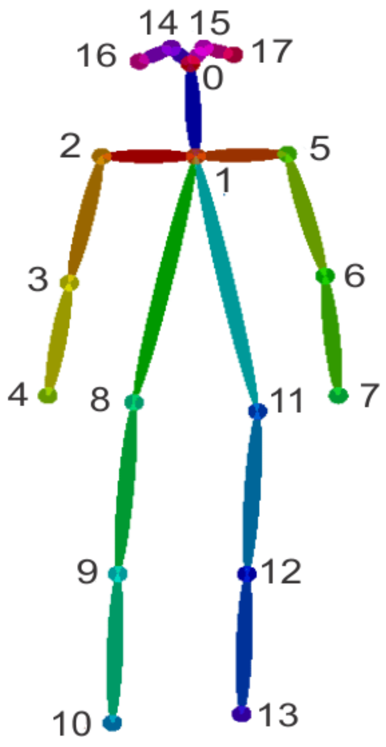


Figure 7: OpenPose 18 Keypoint Detection

8.2 Hardware Manufacturing

The last step of this project is to manufacture a robotic arm which can mimic a human arm. This robotic arm will be designed similar to the interbotix-ros arm which we have used in our simulation. We have used 4 degrees of freedoms of the arm-

- 1 at the Wrist
- 1 at the Elbow
- 2 at the Shoulder

The designed arm will have an extra degree of freedom for the gripper which will be controlled by detecting the motion of 2 fingers.

Bibliography

[1] What is ros? Open Source Robotics Foundation (OSRF), Roswiki.

[2] Gazebo robot simulator. Open Source Robotics Foundation (OSRF), GazeboSim website.

[3] Inverse kinematics. IEEE Explore website, Online Motion Generation.

[4] Github repository 1. Carnegie Mellon University, Github Repository-Perceptual Computing Lab/openpose, OpenPose.

[5] Github repository 2. Interbotix Labs,Github Repository-interbotix ros arms, Interbotix ROS Arms.

[6] Github repository 3. The Neuroscience and Robotics Lab,Northwestern University,Github Repository - NxRLab/ModernRobotics, Modern Robotics.

[7] Opencv video. Adrian Rosebrock-Ball Tracking with OpenCV, Ball Tracking.

[8] Camera projection. PennState College of Engineering-Computer Science Engineering, Lecture.