bb_bot: Teaching a robot how to dance using skeletal mapping and inverse kinematics

By Halishia Chugani, Dominique Gordon, and Bridget Jackson
hsc2137, dlg2157, bmj2125

Results Video: https://vimeo.com/267511886

Project Description:

The aim of this project was to create a robot that could mimic dance moves given human input. By the end of this project, we created one cohesive program that recorded human dance moves and mapped the skeletal joint locations from the human movement onto a robot to cause it to move in the exact same manner in simulation. Specifically, given human input of the desired dance moves through a kinect camera, our program captured consecutive RGB and depth images, which were then passed through an external skeletal joint detector called OpenPose to convert the images into consecutive skeletal joint values. The X,Y, Z coordinates of the joints were calculated using the depth image that corresponded to the RGB image taken at the same time. These real life coordinates were then converted to angles, which were mapped onto the Baxter simulation. Using MoveIt, we enabled the baxter to move according to these joint angles.

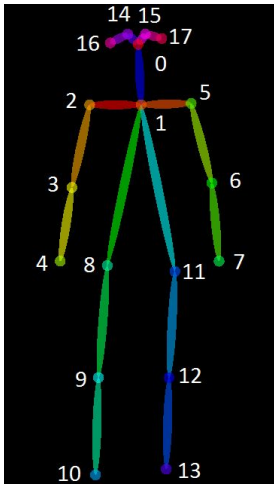
Related Previous Work:

A lot of previous studies have been conducted that involve robots mimicking human actions. Notable ones that influenced our project were:
2014: Markerless human–robot interface for dual robot manipulators using Kinect sensor
2015: A survey of Applications and Human Motion Recognition with Microsoft Kinect
2016: Inverse Kinematics Based Human Mimicking System using Skeletal Tracking Technology
2017: VNect: Real-time 3D Human Pose Estimation with a Single RGB Camera

Implementation Details:

Step 1: Capturing input from a Kinect and retrieving skeletal joints
To capture input from the Kinect 2, we used libfreenect and iai_kinect2, which together allowed us to access the camera info, depth image and RGB image. In order to find corresponding depth and RGB images, the files were named the timestamp from when the subscriber message was generated. From there, each RGB images was passed through OpenPose, which generated a json file that contained the skeletal joint information for the contents of the RGB image. The json file was also labeled with the RGB image's timestamp so that information from the corresponding depth image could be matched to the skeletal joint values later on. Below is an image of the joint positions and their corresponding labels according to OpenPose.

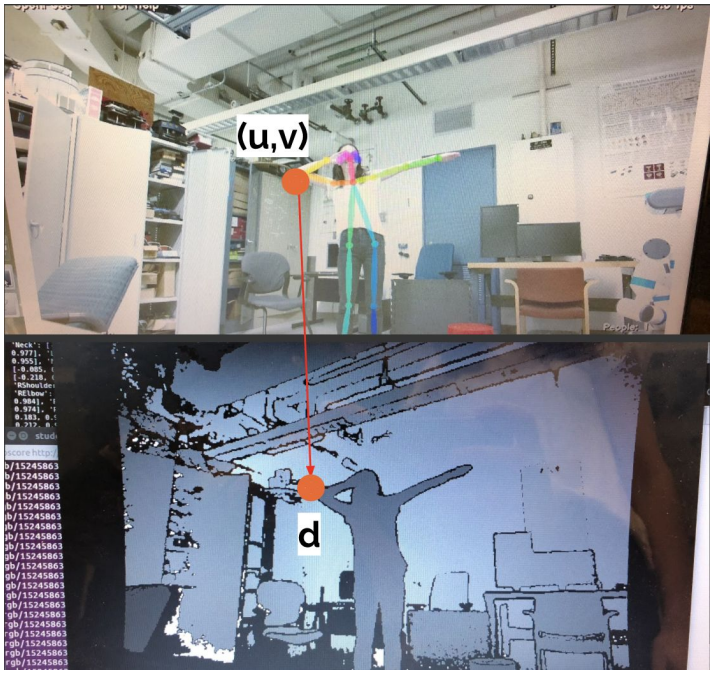
```
// POSE_COCO_BODY_PARTS {
//     {0,  "Nose"},
//     {1,  "Neck"},
//     {2,  "RShoulder"},
//     {3,  "RElbow"},
//     {4,  "RWrist"},
//     {5,  "LShoulder"},
//     {6,  "LElbow"},
//     {7,  "LWrist"},
//     {8,  "RHip"},
//     {9,  "RKnee"},
//     {10, "RAnkle"},
//     {11, "LHip"},
//     {12, "LKnee"},
//     {13, "LAnkle"},
//     {14, "REye"},
//     {15, "LEye"},
//     {16, "REar"},
//     {17, "LEar"},
//     {18, "Background"},
// }
```




```
{
    "0":[296.994,258.976,0.845918,238.996,365.027,0.189116],
    "1":[381.024,321.984,0.587007],
    "2":[313.996,314.97,0.377899],
    "3":[238.996,365.027,0.189116],
    "4":[283.015,332.986,0.665039],
    "5":[457.987,324.003,0.430488,283.015,332.986,0.665039],
    "6":[],
    "7":[],
    "8":[],
    "9":[],
    "10":[],
    "11":[],
    "12":[],
    "13":[],
    "14":[293.001,242.991,0.674305],
    "15":[314.978,241,0.797508],
    "16":[],
    "17":[369.007,235.964,0.88765]
}
```


Step 2: Converting skeletal joints to real world X, Y, Z coordinates

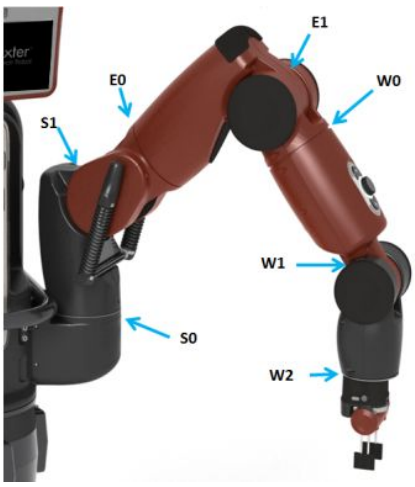
We designed our file naming conventions so that each RGB image, depth image, json skeletal joints file, and camera information file was named with a timestamp so that all of the filenames could be tied together to provide a cohesive set of information for each timeframe. Because the Baxter robot only enables upper body movements, we parsed through the json file to isolate the first 8 joint values that were pertinent to our project (Nose, Neck, RShoulder, RElbow, RWrist, LShoulder, LElbow, LWrist). Once the human skeletal joint points (u, v) were isolated, we mapped these 2D (u,v) values onto the depth image at the corresponding timestamp to obtain 3D (u,v,d) values: depth d = depthimage[u][v] (See below image). Using Camera Info for each image, along with the depth, we were able retrieve the real-world ray that we could multiply with <u,v,d> to get a real world <x,y,z>.



Step 3: X,Y,Z coordinates onto the Baxter simulation

Our Baxter simulation installation was streamlined thanks to another group's project, Baxter The Pool Wiz, in which they created a ./install.sh file that they kindly let us use. To map the x,y,z real world coordinates from the human skeletal joints on to the baxter, we first had to convert our X,Y,Z coordinates into radians. We realized from the skeletal tracking points we were only able to retrieve a total of two angles on each side: shoulder and elbow. This is because three points

were required to make an angle, so we used head, shoulder, elbow points to get the shoulder angle, and shoulder, elbow, wrist, to get the elbow angle. As seen in the image on the right, the Baxter contains seven joints. After some experimenting on Rviz and Gazebo, we learned that S1 on the Baxter corresponded to the shoulder angle we obtained from the human, and E1 on the baxter corresponded to the human elbow. Having assigned the human angles onto the Baxter, we were then able to determine the information necessary for the other joint angles which did not have a mapping in the skeletal joint values from OpenPose. The joints angles we manually determined were rotational joints S0 and E0. To have these rotational values correctly correspond to the movement the human performed, we created thresholds for how rotational joints should be configured depending on the values of the other joints. For instance, if the other joint angles we had recorded were within the threshold of a dab movement, then we would set the rotational joint values to orient S0 and E0 appropriately whilst using the retrieved joint angles for the rest of the arm. We did not capture joint angles for the wrist because none of the required dance moves required bending or rotation of the wrist (the wrist is always aligned straight with the forearm), as such we had to specify w0, w1 and w2 as zeroes on both the right and left sides.



Default rotational values:

1. Rollie
   Left_pose = [.7, Shoulder-pi, -3, Pi-elbow, 0,0,0,0,0]
   Right_pose = [-.7, Shoulder-pi, 3, Pi-elbow, 0,0,0,0,0]
2. Dab
   Left_pose = [.7, Shoulder-pi, -2, Pi-elbow, 0,0,0,0,0]
   Right_pose = [-.7, Shoulder-pi, 2, Pi-elbow, 0,0,0,0,0]
3. Pants
   Left_pose = [.7, Shoulder-pi, -1, Pi-elbow, 0,0,0,0,0]
   Right_pose = [-.7, Shoulder-pi, 1, Pi-elbow, 0,0,0,0,0]

Trials and Tribulations:

Mac Computer:

When we began this project, we initially ran Parallels in order to use Ubuntu on our Mac computers; however, once we got to the step that required us to use the Kinect, we discovered that the Kinect is not compatible with Parallels. We then installed Ubuntu using a flash-drive; however, upon installing graphics drivers while following an installation tutorial for OpenNI2, we were forced to reinstall Ubuntu as the graphics driver broke the operating system. Due to many similar issues, we ended up reinstalling Ubuntu about twelve times (not an exaggeration) before ultimately deciding our path on this project would be much easier using a laptop already running Ubuntu. David was kind enough to lend us a laptop from the lab.

OpenNI2, NiTE2.2:
OpenNI2 is an application that receives depth, RGB, and IR videos from a device (in our case, the Kinect). NiTE2.2 is a middleware that receives color and depth input from the hardware device and performs functions including skeleton joint tracking. One should be able to fully implementing 3D skeletal tracking using OpenNI2 and NiTE2.2. However, after many attempts to use OpenNI2 and NiTE2.2, we discovered that they are outdated and no longer capable of performing properly.

2D Skeletal tracking with OpenCV2:
After concluding that we could no longer use OpenNI2 and NiTE2.2 to accomplish 3D skeletal tracking, we implemented our own skeletal tracker from scratch. We used OpenCV2 to recognize skin tone regions using the webcam from our laptop (Image 1). We then contained those regions within rotated rectangles (Image 2). Using the midpoints of the shorter sides of the rectangles, we could isolate the length of the limbs (Image 3). We then improved our program to recognize different colors and wrapped our respective body parts in those colors (blue: upper arm, green: forearm, and skin-tone: hand). This allowed us to differentiate the body parts (Image 4). This 2D skeletal tracker worked completely as it used the line segments' endpoints as joint positions. However, once the input was changed from the webcam to the kinect (we would need the kinect to eventually map these 2D coordinates onto the depth image to get the 3D skeleton), the kinect was not able to consistently identify regions with our pre-set colors the way the webcam could. We solved this issue by implementing a clicker system where for each color the user wanted the program to recognize, the user would click on the desired color until satisfied with the area recognized and the line generated. We also added a slide-bar that allowed the user adjust the threshold of RGB values accepted for a given color (e.g. +-50 on B for RGB 100 100 100). Upon completing our 2D skeletal tracker with the kinect, we found OpenPose, a 2D skeletal tracker that worked without the color recognition where you had to wrap different body parts with different colors. This streamlined our process significantly, so we decided to use OpenPose for our project instead of our 2D skeletal tracker.

Image 1: OpenCV2 recognizes given color

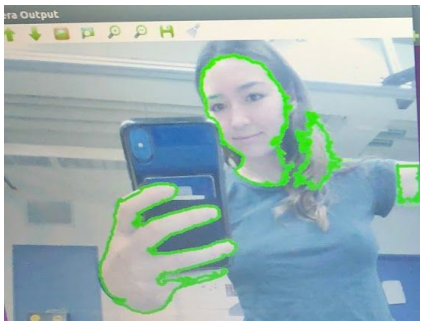


Image 2: Contained region within rectangles

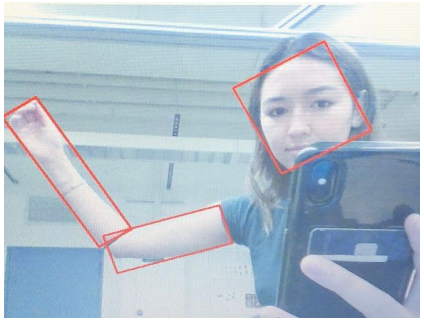


Image 3: Isolated line segments

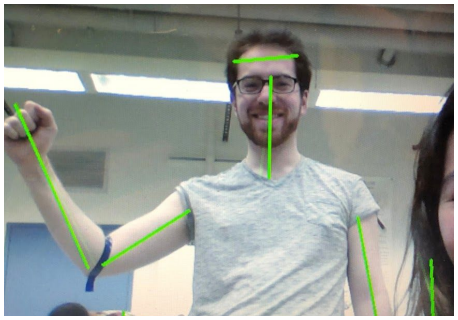
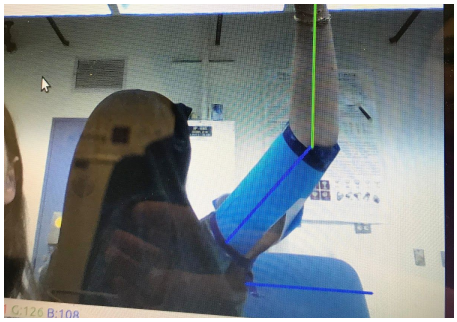


Image 4: Calibrated for other colors



Simulation to Robot:
Though the movements successfully performed on the simulation, when we tried to run our program on the real Baxter, we received a lot of errors claiming that our joint values were out of bounds. Upon looking up this maximum joint value limits for Baxter, this made sense as many of our values were larger; however, this was very odd because they had always been able to execute on the simulation. We had already put a lot of work into translating the 3D skeleton into joints that performed well on the simulation, and once we began to try to adjust the values so they would work on the real robot, it would break on the simulation, so we made the decision to just have our program work on the simulation and no longer pursue getting it to work on the real robot.

Real-time:
When we attempted to implement real-time processing we waited for OpenPose to finish processing an rgb image before it moved on to the next image. Unfortunately, OpenPose takes about 4 seconds to process each rgb image. This meant that too much information was lost whilst waiting for the processing to finish for real-time to have been feasible. Had we used a more powerful computer, it is likely that real-time simulation would have been effective due to that fact that the OpenPose output can be implemented immediately in simulation. However, even if we had used a different computer, OpenPose would indeed always cause some kind of delay and it was difficult to tell if this delay would have distorted the ability of the processing to the point that it wasn't actually in real-time.

Results:

All-in-all we were able to successfully make our simulated robot dance using skeletal mapping and inverse kinematics. There are times where the robot does not perform perfectly; however, through visually analysis one can see that it nearly mirrors the human moves that it is copying. To view our full results, we have created a video where you can see a side-to-side view of both the human dances and the skeletal mapping vs the robots ultimate movements. The video is provided in our submission titled "ResultsVideo.mp4" and can be found at the following link: https://vimeo.com/267511886

Division of work:

We're all best friends, so it made it very easy to work on this project together and each contribute a ton and contribute equally. We all worked on each aspect of this project; sometimes individually, sometimes all together, sometimes in pairs. At least one of us attended almost every single minute of office hours. The way we would typically work in order for us to easily pass off the work between us would be to make sure when we transitioned the work from one group of us to another we always had some overlap between the groups so that there was time to get the new group up to speed with the current state and give them time to ask any questions. Below is an example of a typical schedule of when and how much each of us would work on the project over a two day span. For example, on Monday, Dominique worked on the project until 6:30 so that there was a 30 minute overlap with when Bridget and Halishia began their work.

Sunday: (Office Hours 10-12)
10-6 Halishia
10-noon; 4-6; 9-12 Dominique
10-3; 9-12 Bridget

Monday: (Office Hours 10-11)
10-1; 6-8 Halishia
10-1; 4:30-6:30 Dominique
6-8 Bridget

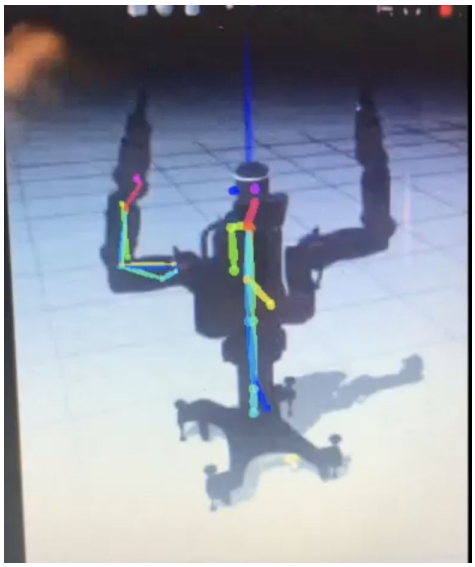
How we ran our program:

(This assumes all the necessary packages are already installed/built and all files are in the correct locations)

First terminal window:
./baxter.sh sim
roslaunch baxter_gazebo baxter_world.launch
Second terminal window:
./baxter.sh sim
rosrun baxter_tools enable_robot.py -e
rosrun baxter_interface joint_trajectory_action_server.py -l both
Third terminal window:
./baxter.sh sim
roslaunch baxter_moveit_config demo_baxter.launch right_electric_gripper:=true left_electric_gripper:=true
Fourth terminal window:
roslaunch kinect2_bridge kinect2_bridge.launch
Fifth terminal window:
python newest3d.py

Extras:

We reran the final video of the robot simulation back through openpose just to see what the output would be. Unfortunately, it wasn't able to recognize the figure very well; it in fact recognized 1-3 different different figures at various points. We have included a video of this in our submission titled "RobotThroughOpenPose.mov." Please note that the video is sped up. Below is a screenshot from the provided video.



References:

https://arxiv.org/abs/1705.01583
http://docs.ros.org/api/image_geometry/html/python/
https://engagedscholarship.csuohio.edu/cgi/viewcontent.cgi?referer=https://scholar.google.com/&httpsredir=1&article=1417&context=enece_facpub
https://github.com/CMU-Perceptual-Computing-Lab/openpose
https://github.com/code-iai/iai_kinect2
https://github.com/grawson/Baxter-The-Pool-Wiz
https://github.com/OpenKinect/libfreenect2
https://link.springer.com/article/10.1007/s10846-016-0384-6
https://opencv.org/
http://openni.ru/files/nite/index.html
http://www.rethinkrobotics.com/baxter/
https://www.sciencedirect.com/science/article/pii/S0736584513000628
http://sdk.rethinkrobotics.com/wiki/Hardware_Specifications
https://structure.io/openni