**Technical Guide for NAO Kinect**

By: Jill Mercer and Brady Rainey

Group 3

**Table of Contents:**

1. Setup
2. Running Program
3. Program Code
   1. Java 4 Kinect (kinectviewerapp)
   2. KinectTracking.java
4. Issues
5. Citations

**I - Setup**:

In order to run the KinectTracking.java the computer running the application must have installed:

1. Kinect for Windows SDK
2. Kinect for Windows Developer Toolkit
3. A Kinect plugged into the computer

The package kinectviewerapp must have also been imported into the project, a copy of the kinectviewer app as well as J4K jars can be found in the documentation folder and can be imported into eclipse by “importing an existing project into workspace”.

For the uses of the assigned project version 1.8 was used, depending on the type of Kinect a newer version may be necessary.

J4K library as well as the NAO libraries, access it from the gui and connect to the robot, mirroring is started once program launches and end when loop finishes or user hits the back button on robots head. (NAO confirms mirroring has started)

**II - Running the Program:**

Access the KinectTracking.java file in src/edu/sru/thangiah/nao/demo/ and run

A GUI will pop up that will allow the user to scan for connected robots.

Once connected, run the Kinect Tracking application.

A window will be launched if Kinect is connected showing skeleton frameworks if one is available.

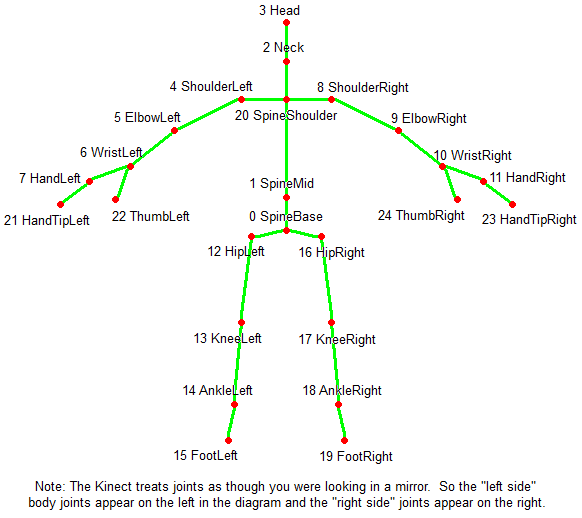
In order to show the numbers the skeleton is reading in, when the GUI showing the skeleton is launched, hit the “show console” button and then make the window full screen. If the number console does not show up, relaunch the application. The user **MUST** ensure that a skeleton has been identified prior to starting the program on the NAO, otherwise it will not run.

Run the program by touching the front part of the NAO’s head, it will mimic the movements until the user hits the head in the back.

**III** – **Program Code:**

1. **Java 4 Kinect:**

The Java 4 Kinect library and package are used to acquire all skeleton data. The kinectviewerapp is located under: /[NAOFrameworkv1.0](https://github.com/samthangiah/NAO-Kinect/tree/master/NAOFramework%20v%201.0)/[NAOHumanoid](https://github.com/samthangiah/NAO-Kinect/tree/master/NAOFramework%20v%201.0/NAOHumanoid)/[src](https://github.com/samthangiah/NAO-Kinect/tree/master/NAOFramework%20v%201.0/NAOHumanoid/src)/[edu](https://github.com/samthangiah/NAO-Kinect/tree/master/NAOFramework%20v%201.0/NAOHumanoid/src/edu)/[sru](https://github.com/samthangiah/NAO-Kinect/tree/master/NAOFramework%20v%201.0/NAOHumanoid/src/edu/sru)/[thangiah](https://github.com/samthangiah/NAO-Kinect/tree/master/NAOFramework%20v%201.0/NAOHumanoid/src/edu/sru/thangiah)/**nao**/. The skeleton class was used to obtain all X,Y and Z coordinates of joints. All j4k documentation for what classes contain what methods are included in the documentation folder.

The main object used was the skeleton class. This allowed access to all X,Y and Z coordinates and was updated frame by frame. The currentSkel variable is what tracks the skeleton real time where the x,y and z coordinates are accessed from. At the time of the project the skeleton class contained an array of 25 joints. For the uses of the Kinect 1.8 only the first 20 were useable the last five (all hand joints) are accessible by the Kinect 2.0 model. As illustrated:

**2.0 features**

**2.0 features**

**Figure 1**

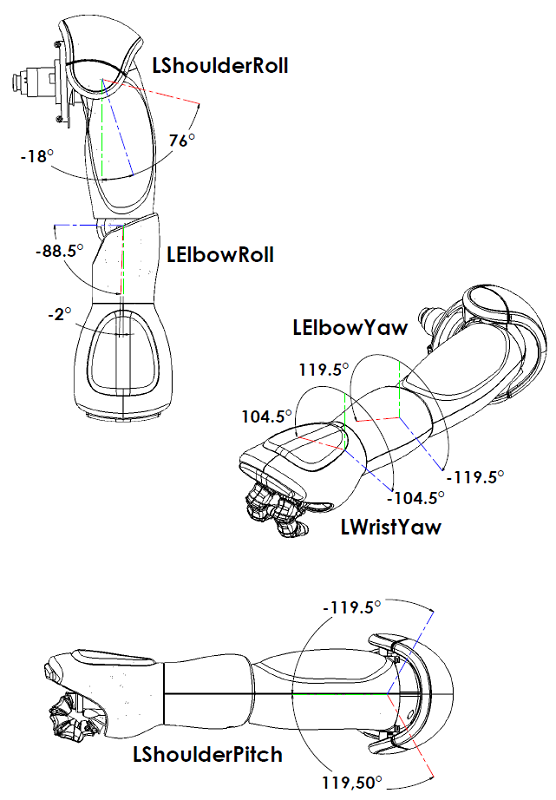
Other difference between Kinect 2.0 and Kinect 1.8 include being able to track up to 6 skeletons as opposed to 2 (Kinect 1.8) and having more functionally in relation to depth and inferred detection (Kinect 2.0).

The skeleton array that is located in ViewerPanel3D holds up to six skeletons. Limiting this to only one causes an error as the loop is constantly checking to see if there are more skeletons added to the screen. Only one skeleton is accounted for in the program, if two people are in view of the Kinect the robot tries to mimic both at once.

All changes were made in the ViewerPanel3D.java file. Anything printed out in the code will go to the console on the GUI.

1. **KinectTracking.java**

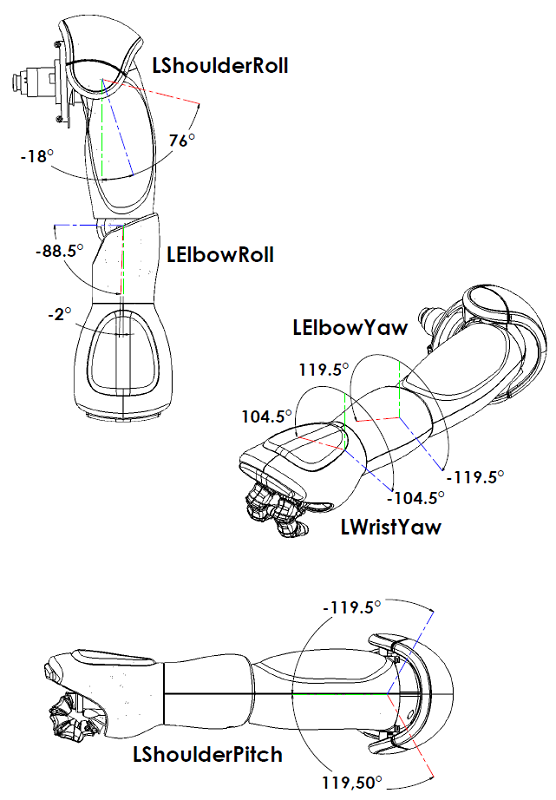
This class takes the currentSkel variable and calculates the angles needed to transfer over to the NAO robot. There were two methods that were used to get the angles for different joints, slope angle and vector angle.

**Slope angle uses the distance formula (y2-y1) / (x2-x1) and then gets the arc tan of that to determine the pitch of the shoulder:

**Figure 2**

This method works when the z axis is not a factor (vertical movements), since while raising the arm the only points changing are the y and x joint positions of the shoulder and elbow.

The other method (getVectorAngle), involves getting the 3 joints where the middle joint designates the angle to calculate. For example, in elbow roll joints the shoulder, elbow, and wrist joints are used to find the angle of the elbow. Since the mid joint is the elbow it will find what the elbow roll angle will be.

By taking these three joints the method calculates the distance between the mid joint and the first outer joint i.e. the difference between shoulder and elbow using X, Y, and Z. The same is done for the third joint and mid joint. Once this is calculated the magnitude of the first difference and the second differences is found, i.e. the magnitude of shoulder to elbow and the magnitude of elbow to wrist. Finally, calculate the dot product and then find the angle between by taking arc cosine of (dotProduct/(magnitude1 \* magnitude2)).

**Figure 3**

The NAO receives the data by sending the various joint information from the getVectorAngle methods to the prebuilt motion.setAngles() function from the NAO. A key point is that the NAO takes radian values not degrees, all methods used to find the angles are already in radians. To gain functionality of the right arm, the angle of the left was negated.

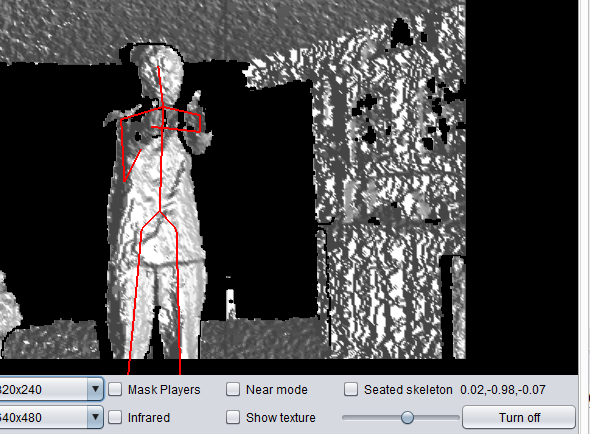
**IV – Issues:**

There were a couple of issues in relation to the robot and console. As for the NAO, balance was a major issue. When mimicking movements, the method to setAngles requires a speed, if the speed exceeded 30% there was a chance the robot would fall over. All speeds are set at 20% to prevent major issues when the arms are above the head (the most frequent state that caused misbalance).

Mirroring the legs also caused a problem. The legs (although having a pitch and roll) cannot be mimicked as far as could be tested, as lifting one leg made the robot go into an unbalanced state. This lack of balance again caused the robot to fall, setting the NAO balance function disallowed the lifting of the leg entirely as it prevented the NAO from going into this unbalanced, one legged, state. Future work could include voice commands that move the robot forward instead of the user lifting the legs themselves through the Kinect.

The NAO also has some unpredicted joint movements. There are two main reasons that cause this, NAO’s collision control and the Kinect skeleton estimation. The NAO has built in collision control mechanisms that prevents the robot from colliding with itself, i.e. touching its head or putting the arms together. If the user tries to force the robot to touch its hands together, the NAO tries to correct the collision and results in moving the arms sporadically. An in-depth explanation of this method can be found in the NAO documentation found at <http://doc.aldebaran.com/1-14/naoqi/motion/reflexes-collision-avoidance.html>.

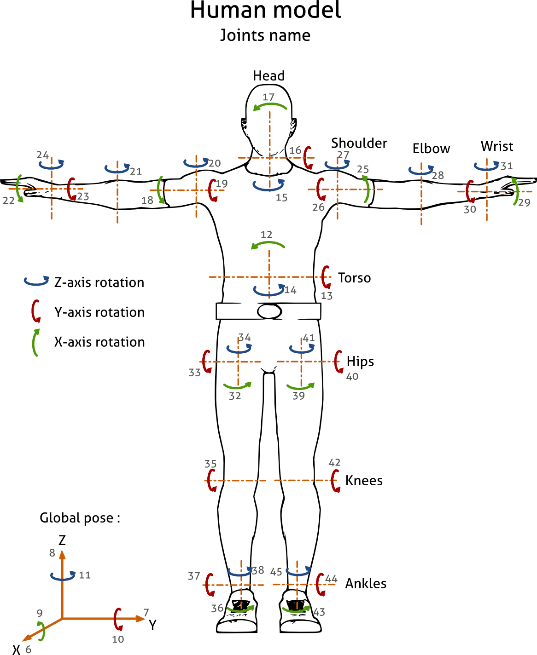
**Figure 4**

The Kinect skeleton estimation tries to calculate where the user’s limbs are if held directly at the camera. Because the Kinect is only seeing a single point, it is forced to estimate where the rest of the joints are located.

As can be noted in the figure 4, although arms are held out toward camera and should produce a straight line, the result is a curve in both arms. This is due to the Kinect guessing at joint positions and these joints fluctuate as the Kinect tries to adjust. Should this minor joint continually move around (which happens when the Kinect tries to adjust) the NAO robot limb itself will continually move, even if the user is stationary. This results in the robot moving its arm completely vertical or horizontal and then coming back to the state in which the user is at.

In relation to the console that displays data when using the KinectViewerApp, it only appears if the “Show Console” button is clicked and then the window is brought to full screen. This must be done prior to making the window full screen otherwise the console will not show. There is also currently no way to get rid of the console once it has been launched. A fix for this is unknown, there are many functions kept from the user when using the J4K libraries so modifying data from the source is tricky. One approach would be to redo the Kinect app entirely, however for the uses of the project at hand, the “Show Console” button worked.

The last area of issue involves angle math difficulties and correction angles. Finding the correct angles for the NAO was the meat of the project, the vector math function seemed to prove trustworthy so long as the user is aware what the function is capable of. As mentioned, the slope angle is useful for vertical movements, all lateral movements require getVectorAngle. For instance, future work involving the NAO’s hand would likely require vector math (between thumb, wrist, and finger), whereas a bend at the waist would use slope math.

Since the Kinect skeleton and the NAO are not one-to-one in terms of measuring angles a “correction angle” needed to be applied to the NAO. For uses of the project, that required testing to see what angle the kinectViewer was reading and relating that the NAO documentation. For example, a standard T position (as noted in figure 5) to determine the angle of elbow roll resulted in 180 degrees on the Kinect. This same angle on the NAO needed to be 0 therefore 180 degrees was the “correction angle” to be subtracted. This also needed to be done for shoulder roll and likely anything that uses the vector angle function. The slope angle was one-to-one and did not require a correction angle.

**Figure 5**

Although this worked for the project at hand, dealing with the NAO limits this correction angle only to this one robot. Any other applications of this program will require their own testing to determine what the correction angle would be.

**V – Citations:**

A. Barmpoutis. "Tensor Body: Real-time Reconstruction of the Human Body and Avatar Synthesis from RGB-D", *IEEE Transactions on Cybernetics, Special issue on Computer Vision for RGB-D Sensors: Kinect and Its Applications*, October 2013, Vol. 43(5), pp. 1347-1356.

Barmpoutis, Angelos. "J4K Java Library." *Java for Kinect (J4K) Library*. University of Florida, Digital Worlds Institute, n.d. Web. 03 Dec. 2016.

Hoover, Richard. "Adventures in Motion Capture: Using Kinect Data (Part 1)."*Adventures in Motion Capture: Using Kinect Data (Part 1)*. SeaLeft Studios, 1 July 2016. Web. 03 Dec. 2016.

"Human Posture." *Human Posture — The MORSE Simulator Documentation*. LAAS-CNRS/ONERA, 2015. Web. 03 Dec. 2016.

"NAO Software 1.14.5 Documentation." *Joints — NAO Software 1.14.5 Documentation*. Aldebaran Robotics, 27 June 2013. Web. 03 Dec. 2016.