## Problems Encountered and Workarounds:

The first problem we encountered was being able to map trajectory from the IMU sensors accurately. Working off of various examples, homework assignments, and help from our TA, we were able to develop a Matlab function that will plot the trajectory of an IMU sensor.

The next problem encountered was ways to synchronize the sensor data, the pressure mat data and Kinect data so that together they can be used to reference the same activity along the time axis. The way to work around this issue is to have data from pressure mat and Kinect carry with it the UTC time stamp which describe a unique point in time that has only one instance. Data with the same UTC stamp should correspond to the exact same moment in history and therefore must be the same data describing only one and same instance of the sensor characteristics.

Another issue we ran into was that the pressure mat is extremely sensitive to any deformation in it. So the mat may conduct even though no one is touching it, due to wrinkles in the mat itself. The plan to overcome this, is to pin the mat flatly on some kind of rigid board, such as a corkboard. This would allow the board to be flat and not conduct when it shouldn’t be.

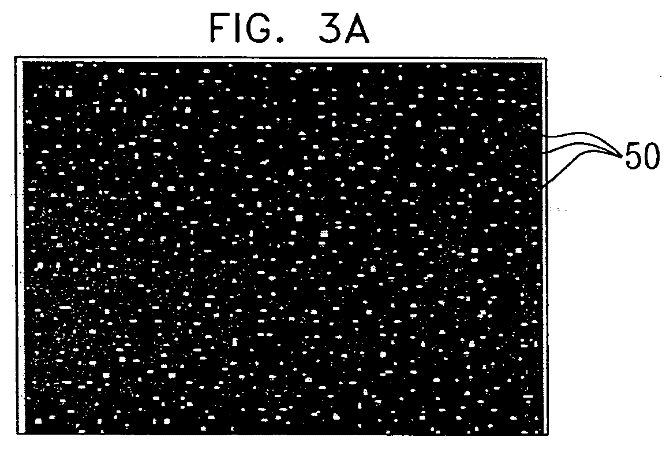
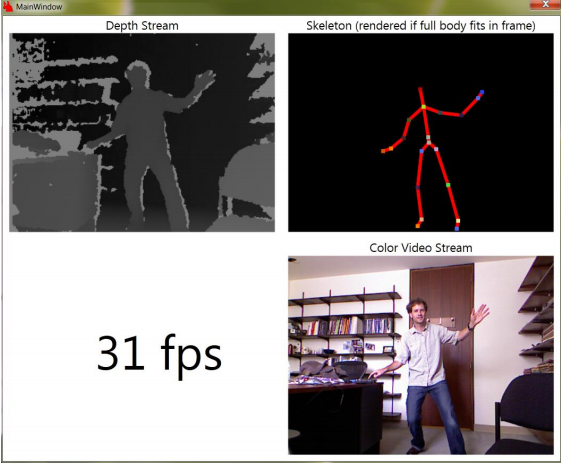
For a while we could not figure out how to connect every stage of the system together. We could get raw data from the IMU sensors, but then we would have to manually connect the sensors via Bluetooth to our computers and then manually start the functions in Matlab to process the data. We eventually settled on using Python. As stated before, Python can be used to automate all of these tasks. Python also allowed us to create our web interface.

## Theory, Software, and Hardware

In addition to the pressure mat and the IMU sensors providing 2D coordinates (XY) and raw data of 3D acceleration data points to be integrated as 3D coordinates (XYZ), the Kinect Sensor, inevitably became our final judge and ultimate standard for calibration due to its ability of directly gathering the 3D coordinates (XYZ) of specific major joints of body and its satisfactory resolution.

###### 

Components of Kinect Sensor (v2)

The first step is gathering the depth image through Light Coding. As shown above, the Kinect Sensor includes IR emitters and a Depth Sensor. The IR emitters project speckle pattern of IR laser light forwards as shown in the right picture. In this way, each coordinate of the 2D frame forwarding (YZ) as labeled by a specific pattern which could be received and recognized by the Depth Sensor. Meanwhile, the Kinect Sensor maintains the original pattern map projected. By comparing the projected pattern map and the received pattern map, the Kinect Sensor can utilize the algorithm of Stereo Triangulation to calculate the distance between each projected IR laser point and the Kinect Sensor. The resolution of the Kinect Sensor is satisfactory – 3mm spatial and 1cm depth at the distance of 2m.

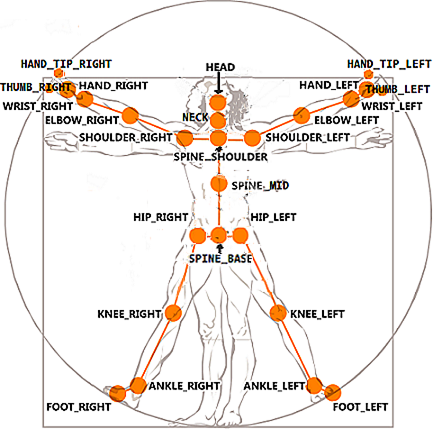
The second step is recognizing joints from the depth image. This step became feasible through training the algorithm through machine learning in the Microsoft lab. Starting with 100,000 depth images with known skeleton form a motion capture system, the algorithm had learnt a *randomized decision forest* mapping depth images to major body joints, outputs the 3D coordinates (XYZ) of joint position in meters.

To get any data points from the Kinect Sensor, we need to use the Kinect for Windows SDK v2. In addition, in order to show both RGB images and 3D coordinate points of selected joints on the screen, using OpenCV 2.4.10 would be the simplest way. To use these APIs properly, we choose VS2013 as the developing environment.

## Basic Design

The sample C++ code for acquiring data points from the Kinect Sensor is shown below:

|  |
| --- |
| for( int type = 0; type < JointType::JointType\_Count; type++ ){ |
| ColorSpacePoint colorSpacePoint = { 0 }; |
| pCoordinateMapper->MapCameraPointToColorSpace( joint[type].Position, &colorSpacePoint ); |
| int x = static\_cast<int>( colorSpacePoint.X ); |
| int y = static\_cast<int>( colorSpacePoint.Y ); |
| if( ( x >= 0 ) && ( x < width ) && ( y >= 0 ) && ( y < height ) ){ |
| cv::circle( bufferMat, cv::Point( x, y ), 5, static\_cast<cv::Scalar>( color[count] ), -1, CV\_AA );}} |

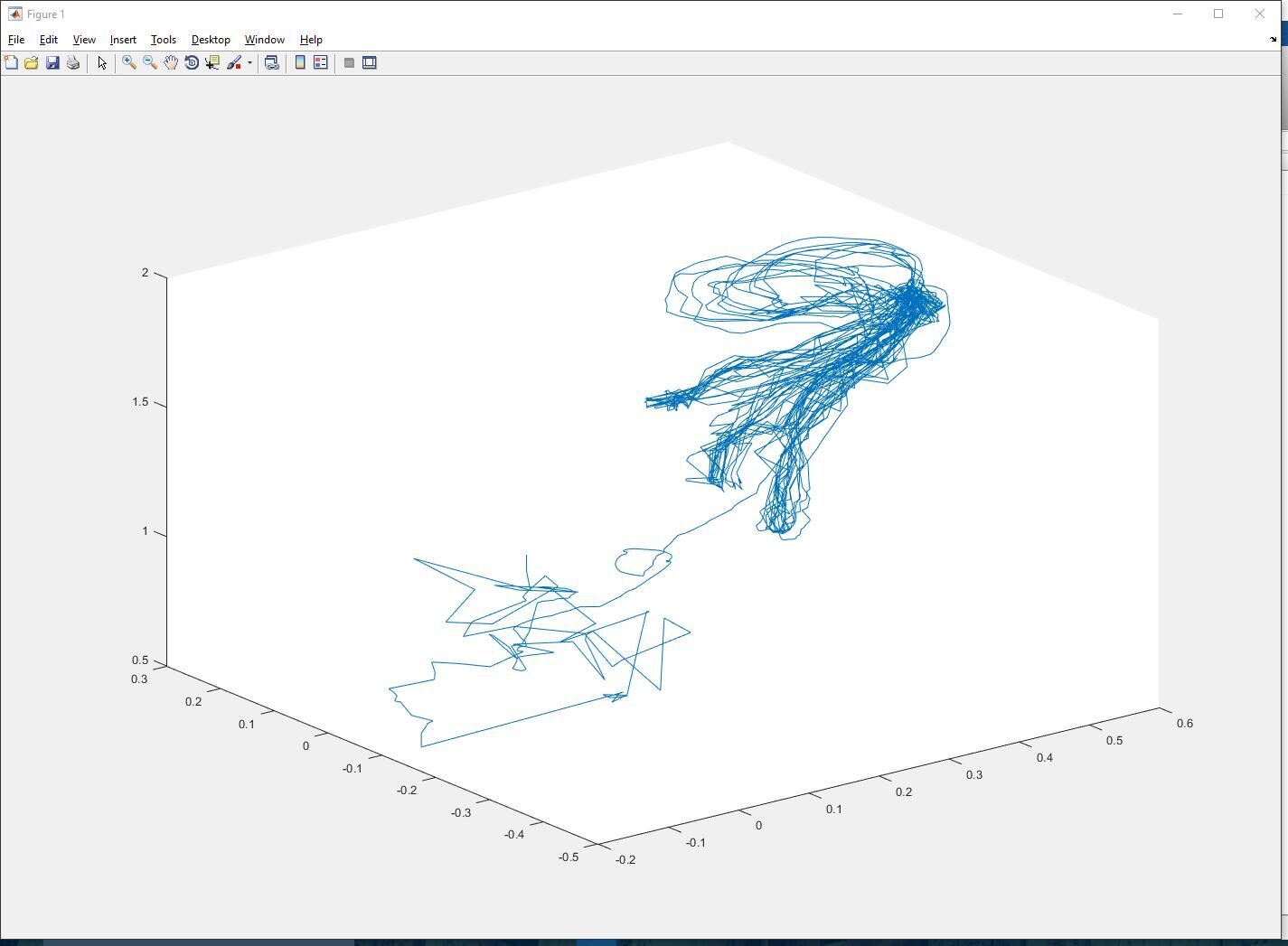
In the SDK, each joint was labeled as the JointType enumeration as shown in the figure below and table right.

To record the movement of human skeleton, we need 5 variables per (frame-joint): # of frame (which would be upgraded to UTC timestamp in the next quarter), # of joint as enumerated in JointType, and 3D coordinates (XYZ) in meters gathered from the variable joint[type].Position. In addition, to record the joints we are interested in only, we can limit the variable type to the joints we specified. In the data points shown in the later section of Experimental Verification, we choose to track the movement of joint #9 ElbowRight and joint #10 WristRight.

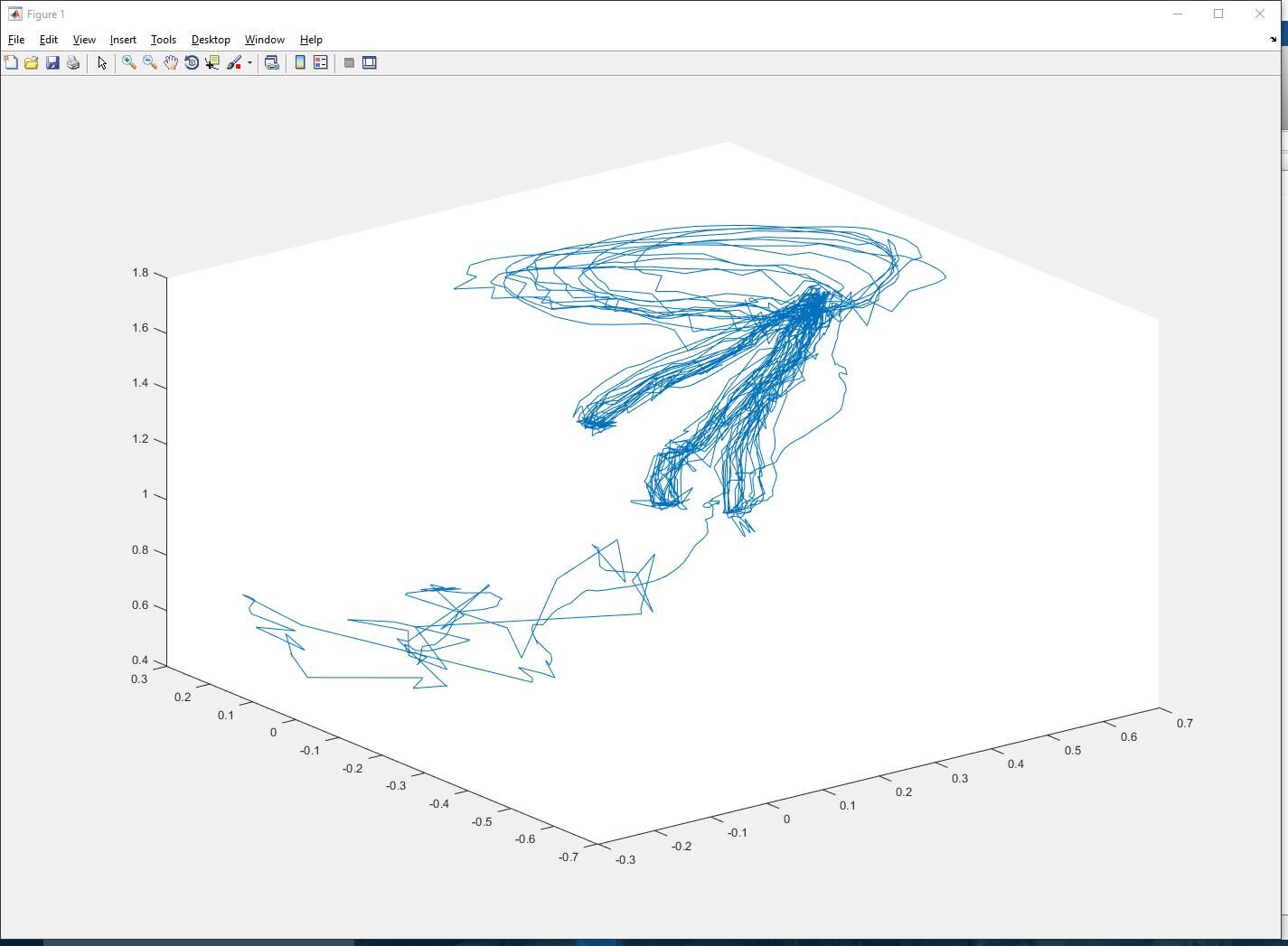
## Experimental Verification

After the coding, gathering 3D coordinate (XYZ) data points from the Kinect Sensor would be straightforward. Sample trajectory as the model moving is shown below, with comparison with data points collected by IMU.

Using the plot3 function in Matlab: plot3(xyz(:, 1), xyz(:, 2), xyz(:, 3))



Movements recorded for joint #9 ElbowRight



Movements recorded for joint #10 WristRight.

## Future plans

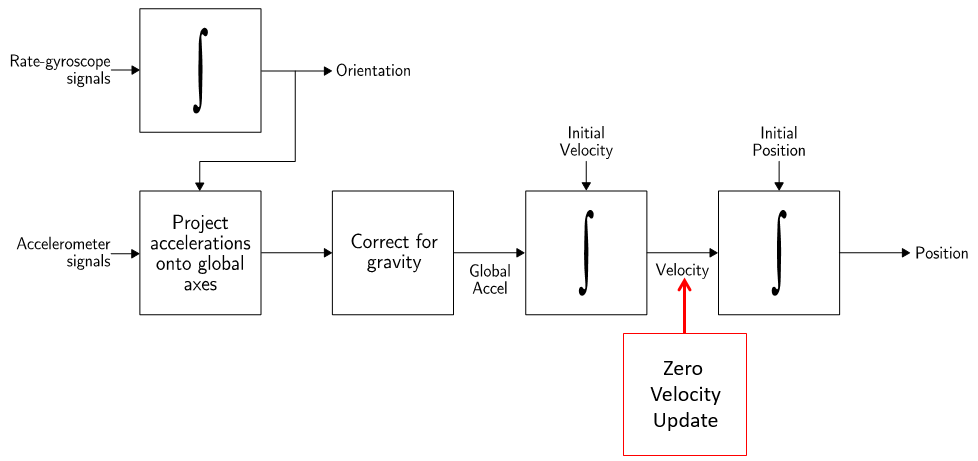
Calibration for the IMU sensor is the central task of the entire project. As the data point of each timestamp gathered by the IMU sensor has an error range provided by product specification, each data point we gathered is not a point, but a range of coordinates. By calibration, we can derive the algorithm learning which point within the error range is acceptable, and hopefully our product could achieve a commercial quality.

In next quarter, we planned to caliber the IMU sensor in two ways: using constraints, using absolute coordinates gathered by the Kinect Sensor and using zero velocity update.

Using constraints would be the most self-evident method for calibration. At first the data points collected by our IMU sensor is totally senseless. My shoulder is in the fourth floor, while my wrist is in the seventh floor. The length of my arm is a constant (with an error range indeed) and can become an absolute constraint for the 3D coordinates we collected from sensors.

The Kinect Sensor, directly gathering position data with a resolution of centimeters, is the ultimate judge of the accuracy of the pressure mat - IMU sensor combination in trajectory calculations to yield a performance parameter of ground truth data acquisition. However, to synchronize the data points gathered by different sensors, the Kinect Sensor is required to provide data points with a UTC time stamp.

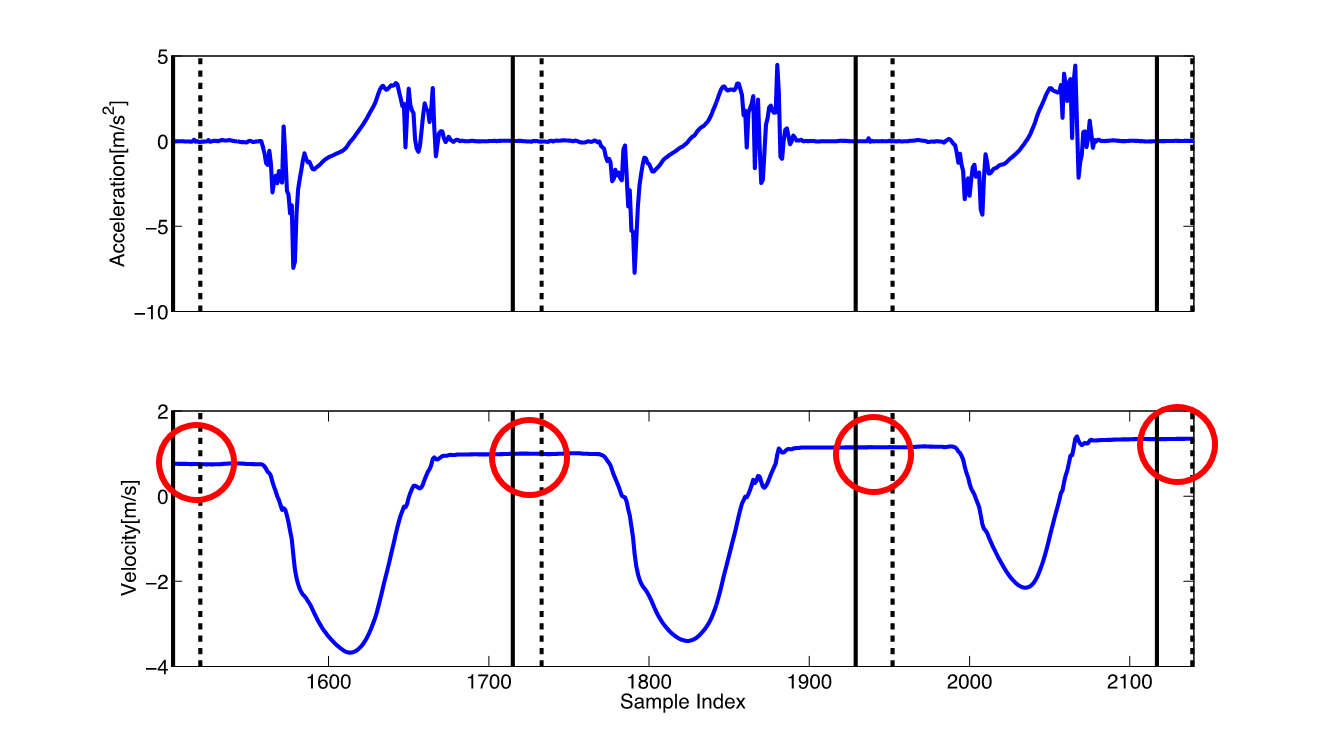
Furthermore, we could utilize the absolute coordinates acquired by the Kinect Sensor to calibrate the IMU sensor. We need to derive an algorithm for the IMU sensor learning from the absolute coordinates acquired by the Kinect Sensor, to train our sensor combination to become commercial quality. Deriving this algorithm could be the hardest part in next quarter, but it could be the most successful method of calibration.

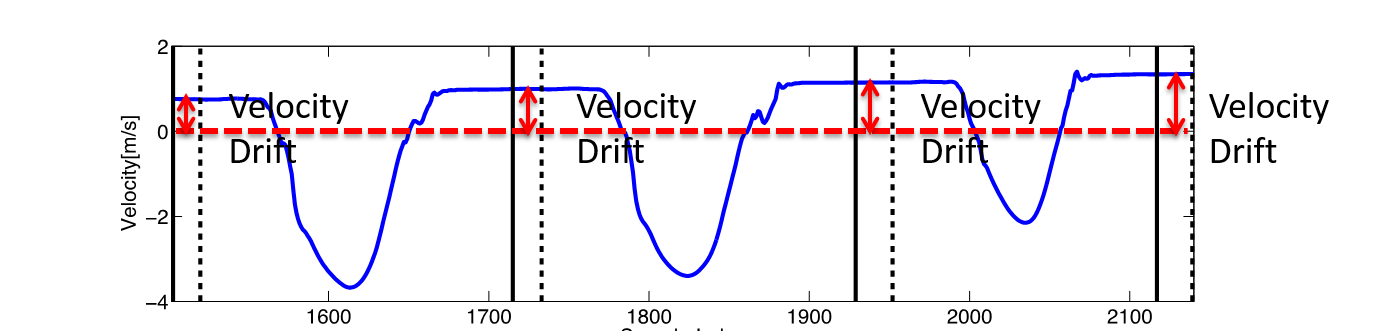


The additional step of Zero Velocity Update in integrating the coordinates

The third calibration method we are going to use is Zero Velocity Update. The IMU sensors are with errors constantly due to constant bias, temperature noise, etc. However, we can set the velocity to zero when foot is stationary, and smooth the velocity in between before being integrated for position. We need to detect whether it is at Zero Velocity by threshold the motion acceleration energy calculated from gathered data points from accelerometers and gyroscopes.

The projected results of the method of Zero Velocity is shown as below:





References:

------------------------------------------------------------------------------------------------------------------

[\*] Introduction for Kinect for Windows SDK v2 [Online, Traditional Chinese] available: <https://kheresy.wordpress.com/2014/12/29/kinect-for-windows-sdk-v2-basic/>

[\*] Depth-varying light fields for three dimensional sensing [Online, US Patent 2008/0106746 A1] available: <http://www.google.com/patents/US20080106746>

[\*] How does the Kinect work? [Online] available: <http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf>

[\*] Reference, Kinect for Windows SDK [Online] available: <https://msdn.microsoft.com/en-us/library/microsoft.kinect.jointtype.aspx>

[\*] Zero Velocity Update Tutorial Slides provided by our TA.