YOST LABS
630 Second Street
Portsmouth Ohio 45662, USA

Phone: 740.876.4936
info@yostlabs.com
yostlabs.com

AN2013.01

Calculating Angles Between Two Yost Labs 3-Space Sensor™ Devices on a Human Body

1. Introduction

This application note provides the mathematics and reference source code for calculating the angle between two YOST LABS 3-Space Sensor devices. This is especially useful for organic motion-capture, bio-mechanics studies, range-of-motion studies, sports studies, and ergonomics studies since it is possible to extract human joint-angles from body-worn sensors.

The YOST LABS 3-Space Sensor is a miniature, high-precision, high-reliability, Attitude and Heading Reference System (AHRS) / Inertial Measurement Unit (IMU) in a single low-cost end-use-ready device. The Attitude and Heading Reference System (AHRS) / Inertial Measurement Unit (IMU) uses triaxial gyroscope, accelerometer, and compass sensors in conjunction with advanced processing and on-board quaternion-based Kalman filtering algorithms to determine orientation relative to an absolute reference in real-time.

Orientation can also be returned relative to a designated reference orientation. This makes 3-Space Sensor placement and alignment easier since the devices can make use of arbitrarily defined zero-identity orientations which makes perfect physical alignment unnecessary and reduces the difficulty in extracting desired output angles.

The 3-Space Sensor devices can return orientation in a number of formats, including as forward and down vectors, thus making it simple to calculate the angle between two of these devices. However, many surfaces, such as those of the human body, may not be flat or smooth, and, thus, we must be able to compensate for the possibility of imperfect sensor placement and alignment. We can use the devices' quaternion orientation output and quaternion operations to account for the human body's irregularities and obtain more accurate forward and down vectors.

The Python language source code listed within this document is in three parts and contains cross-references to the equations used. These listings contain all the code needed to return the angle or angles (in radians and

degrees) between two 3-Space Sensor devices for a variety of possible joint configurations.

For convenience, this document assumes the use of two 3-Space Sensor Wireless 2.4GHz DSSS devices that are being communicated via USB. Since calculating different joint angles requires different sensor placement, this application note uses two possible sensor placement configurations. In the first configuration, the devices are mounted on the right upper-arm and fore-arm and are positioned to be lined up with the LED lights towards the shoulder. In the second configuration, one device is mounted in the middle of the back with the LED light up towards the head and the other device is mounted on the right upper-arm with the LED light towards the shoulder. Additionally, prior to running the code, both configurations require the sensors to be properly mounted and the person to be in the standard T-pose, arms straight out to the side of the body and the palms facing forward. All vectors used are unit vectors and all quaternions used are unit quaternions.

3-Space Sensor Commands Used

There are four 3-Space Sensor commands used in the Python source code.

- **Command 0x00**: This command gets the filtered tared orientation of the device as a quaternion.
- **Command 0x06**: This command gets the filtered orientation of the device as a quaternion.
- **Command 0x0C**: This command gets the filtered orientation of the device as two vectors, where the first vector refers to North and the second refers to Gravity. These vectors are given in the device's reference frame and not the global reference frame.
- **Command 0x61**: This command sets the tare orientation of the device to be the same as the supplied orientation, which should be passed as a quaternion.

2. Types of Joints in the Human Body

There are five types of joints in the human body. 3-Space Sensor devices can be used to detect the motions of and extract angles from each of these joint types.

2.1 Hinge Joint

A hinge joint acts much like a hinge on a door. They allow for back and forth movement around the axis of the joints, but do not allow side to side or lateral movements. Hinge joint examples in the human body are the elbows, knees, and the middle and end joints of the fingers and toes.

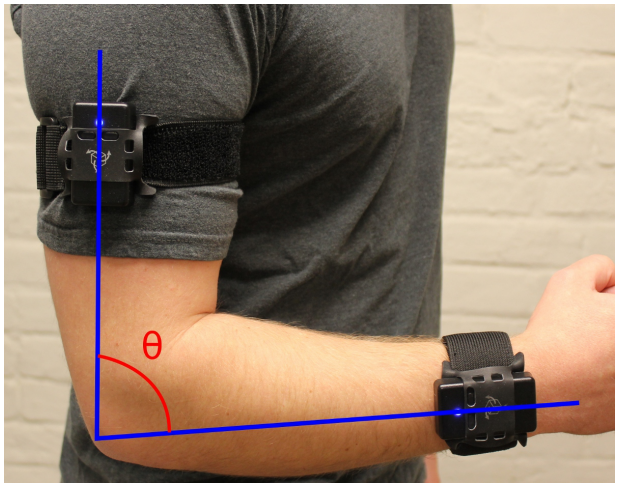

Figure 1 – Typical Hinge Joint

2.2 Ball and Socket Joint

A ball and socket joint (or spheroidal joint) is a joint in which a ball-shaped surface of one bone is connected to a corresponding socket-shaped recess of another bone. This configuration allows for movement around multiple axes in almost any direction. Ball and socket joint examples in the human body are the hip joints and the shoulder joints.

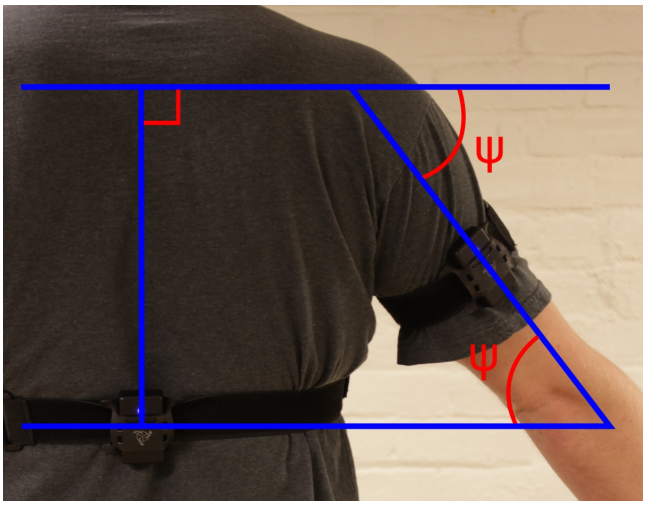

Figure 2 – Typical Ball and Socket Joint

2.3 Ellipsoid Joint

An ellipsoid joint (or condyloid joint) allows for angular, bending movements but with limited rotation. So it is similar to the movements of a ball and socket joint with lesser magnitude. Ellipsoid joint examples in the human body are the wrist joints and ankle joints.

2.4 Pivot Joint

A pivot joint allows for rotation around a single axis. Pivot joint examples in the human body are the neck and forearms.

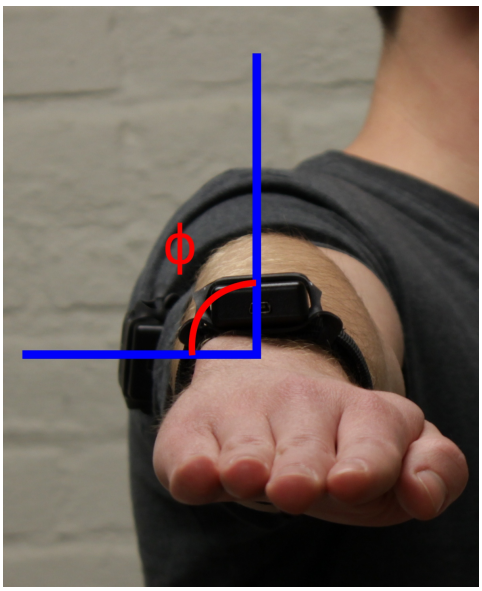

Figure 3 – Typical Pivot Joint

2.5 Saddle Joint

A saddle joint allows for the same movements as the ellipsoid joint. A saddle joint example in the human body is the carpometacarpal joint of the thumbs.

3. Mathematical and Algorithmic Foundations

3.1 Algorithms for Calculating the Angle Between Two Vectors

Dot (Inner) Product

Using the properties of vectors, the dot product can be used to calculate the angle between two vectors.

$$v_0 = (ix_0, jy_0, kz_0) \quad \textit{Eqn. 1}$$

$$v_1 = (ix_1, jy_1, kz_1) \quad \textit{Eqn. 2}$$

$$v_0 \cdot v_1 = x_0x_1 + y_0y_1 + z_0z_1 \quad \textit{Eqn. 3}$$

$$= \| v_0 \| \| v_1 \| \cos(\theta) \quad \textit{Eqn. 4}$$

Now using some algebra we can combine ***Equation 3*** and ***Equation 4*** into an equation defining θ.

$$\| v_0 \| \| v_1 \| \cos(\theta) = x_0x_1 + y_0y_1 + z_0z_1 \quad \textit{Eqn. 5}$$

$$\theta = \arccos\left(\frac{x_0x_1 + y_0y_1 + z_0z_1}{\| v_0 \| \| v_1 \|}\right) \quad \textit{Eqn. 6}$$

We can simplify ***Equation 6*** knowing that the lengths of the vectors are 1.

$$\theta = \arccos x_0x_1 + y_0y_1 + z_0z_1 \quad \textit{Eqn. 7}$$

So θ represents the angle between these two vectors. This algorithm can be applied to the forward and down

vectors received from the 3-Space Sensor devices to calculate the angle between them.

Cross (Outer) Product

Using the properties of vectors, the cross product can be used to calculate a vector perpendicular to two vectors and the angle between the two vectors.

$$v_0 \times v_1 = (i(y_0 z_1 - z_0 y_1), j(z_0 x_1 - x_0 z_1), k(x_0 y_1 - y_0 x_1)) \quad \textit{Eqn. 8}$$

$$= \| v_0 \| \| v_1 \| \sin(\theta) n \quad \textit{Eqn. 9}$$

Where ***n*** is a unit vector perpendicular to v_0 and v_1, and where θ is the angle between them. To calculate θ, remember that the vectors are unit vectors so the lengths of the vectors are 1. Take note however, that the cross vector may not have a length of 1 due to θ.

$$\| v_0 \times v_1 \| = \| \| v_0 \| \| v_1 \| \sin(\theta) n \| \quad \textit{Eqn. 10}$$

$$= \| v_0 \| \| v_1 \| \sin(\theta) \| n \| \quad \textit{Eqn. 11}$$

$$\sin(\theta) = \| v_0 \times v_1 \| \quad \textit{Eqn. 12}$$

$$\theta = \arcsin \| v_0 \times v_1 \| \quad \textit{Eqn. 13}$$

So θ represents the angle between these two vectors. This algorithm can be applied to the forward and down vectors received from the 3-Space Sensor devices to calculate the angle between them.

3.2 Algorithms for Quaternion Operations

A quaternion, ***q***, is a fourth dimensional vector that can be interpreted as a third dimensional rotation.

$$q = (ix, jy, kz, w) \quad \textit{Eqn. 14}$$

$$= u \sin(\theta/2), \cos(\theta/2) \quad \textit{Eqn. 15}$$

$$= (v, w) \quad \textit{Eqn. 16}$$

For ***Equation 15***, ***u*** is a vector defined as:

$$u = (i, j, k) \quad \textit{Eqn. 17}$$

Quaternion Conjugate and Inverse

Finding the conjugate of a quaternion, ***q'***, is easily done by negating the imaginary numbers or the vector part of the quaternion. And since all quaternions in this application note are unit quaternions, the conjugate of a quaternion is equal to the inverse of the quaternion, q^{-1}.

$$q' = q^{-1} = (-ix, -jy, -kz, w) \quad \textit{Eqn. 18}$$

Quaternion Multiplication

$$q_0 = (v_0, w_0) \quad \textit{Eqn. 19}$$

$$q_1 = (v_1, w_1) \quad \textit{Eqn. 20}$$

$$q_0 q_1 = (w_0 v_1 + w_1 v_0 + v_0 \times v_1, w_0 + w_1 - v_0 \cdot v_1) \quad \textit{Eqn. 21}$$

Quaternion Vector Multiplication

To rotate a vector by a quaternion, we must use pure quaternions, ***p***, which are quaternions with its real part as 0 and conjugacy. So any vector can be made into a pure quaternion by putting the vector in the vector part

and 0 in the real part of a quaternion. The return value of the conjugacy is a pure quaternion, which can be interpreted as a vector by ignoring the real part.

$$p = (v, 0) \quad \textit{Eqn. 22}$$

$$v_r = qpq' \quad \textit{Eqn. 23}$$

3.3 Compensation for the Human Body

As mentioned earlier the human body is not flat nor smooth, and makes perfect placement and alignment of 3-Space Sensor devices difficult. Thus, a method of correction is needed to accommodate imperfect sensor alignment issues. To do this we are going to use the device's North and Gravity vectors, specifically the Gravity vector. The Gravity vector denotes the direction in which the device thinks gravity is pulling on it. If the human body was flat and smooth, the Gravity vector would line up with one of the Cartesian coordinate axes of the device's reference frame when in the starting position. So in order to fix this we must calculate the Gravity vector's offset from the Cartesian coordinate axis we denote as the true gravity vector and correct for it.

$$\theta = \arccos(g_d \cdot g) \quad \textit{Eqn. 24}$$

$$a = g_d \times g \quad \textit{Eqn. 25}$$

$$q_o = a\sin(-\theta/2), \cos(-\theta/2) \quad \textit{Eqn. 26}$$

Where $\boldsymbol{g_d}$ is the Gravity vector from the device and $\boldsymbol{g}$ is the gravity vector we want, $\boldsymbol{a}$ is a unit vector that denotes the axis of rotation from $\boldsymbol{g_d}$ to $\boldsymbol{g}$, and $\boldsymbol{q_o}$ is the rotational offset as a quaternion.

Knowing this, all we need to do is offset $\boldsymbol{g_d}$ so it is lined up with the $\mathbf{g}$. This is done by multiplying the filtered orientation of the device by $\boldsymbol{q_o}$ and set the result as the tare orientation of the device. Note that we must retain this offset so that we can post-multiply the orientation of the device by it later.

3.4 Calculating the Vectors of a 3-Space Sensor Device

The following are the vectors that will be used to calculate the angles of the joints and the methods used to calculate these vectors.

Forward Vector

To calculate the forward vector of a device, we are going to use the filtered tared orientation of the device and a vector in the device's reference frame that will be denoted as the forward vector. Remember the rotational offset of the device must be applied to the filtered tared orientation before calculating the forward vector of the device.

$$v_F = q_t q_o v \quad \textit{Eqn. 27}$$

Where $\boldsymbol{q_t}$ is the filtered tared orientation of the device, $\boldsymbol{v_F}$ is the device's forward vector in global-space, and $\boldsymbol{v}$ is the unrotated forward vector in the sensor's space.

Down Vector

To calculate the down vector of a device, we are going to use the filtered tared orientation of the device and a vector in the device's reference frame that will be denoted as the down vector. Remember the rotational offset of the device must be applied to the filtered tared orientation before calculating the down vector of the device.

$$v_D = q_t q_o v \quad \textit{Eqn. 28}$$

Where q_t is the filtered tared orientation of the device, v_D is the device's down vector in global-space, and v is the unrotated down vector in the sensor's space.

Up Vector

To calculate the up vector of a device, a simple negation of the calculated down-vector is all that is necessary.

$$v_U = -v_D \quad \textit{Eqn. 29}$$

Right Vector

To calculate the right vector of a device, we are going to use the forward and down vectors of the device and perform the cross product on them.

$$v_R = v_F \times v_D \quad \textit{Eqn. 30}$$

Where v_R is the right vector of the device. Also, note that the forward and down vectors already have the compensation applied to them so v_R will also already compensated and that the 3-Space Sensor uses a left-handed space by default.

3.5 Methods for Calculating the Angle(s) of the Joints

Calculating the Angle of a Hinge Joint

This section will discuss how to calculate the angle of the hinge joint of the right arm using the first setup of the 3-Space Sensor devices. We are going to use the forward vector of the devices, where v_{F0} is the forward vector of the first device and v_{F1} is the forward vector of the second device. After calculating the forward vectors, we will also need to calculate the up vector from the device that is being used as the reference device, in this case the first device. This vector, v_{U0}, will help in determining the sign of the angle.

Now using the forward vectors from the devices and the up vector we can calculate the angle between the devices.

$$\theta = \arccos(v_{F1} \cdot v_{F0}) \quad \textit{Eqn. 31}$$

$$a = v_{F1} \times v_{F0} \quad \textit{Eqn. 32}$$

$$\theta = copysign(\theta, (v_{U0} \cdot a)) \quad \textit{Eqn. 33}$$

Take note that arccos will always return a positive value, so we must use the dot product of a and v_{U0} to calculate the sign of θ. The function *copysign*, is a function that returns the first parameter with the same sign as the second parameter, so using the dot product is perfect because it ranges from -1 to 1.

Calculating the Angle of a Pivot Joint

This section will discuss how to calculate the angle of the pivot joint of the right arm using the first setup of the 3-Space Sensor devices. We are going to use the down vector of the devices, where v_{D0} is the down vector of the first device and v_{D1} is the down vector of the second device. After calculating the down vectors, we will also need to calculate the forward vector from the device that is being used as the reference device, in this case the

first device. This vector, v_{F0}, will help in determining the sign of the angle.

Now using the down vectors from the devices and the forward vector we can calculate the angle between the devices.

$$\theta = \arccos(v_{D1} \cdot v_{D0}) \quad \textit{Eqn. 34}$$

$$a = v_{D1} \times v_{D0} \quad \textit{Eqn. 35}$$

$$\theta = copysign(\theta, (v_{F0} \cdot a)) \quad \textit{Eqn. 36}$$

Take note that arccos will always return a positive value, so we must use the dot product of a and v_{F0} to calculate the sign of θ. The function *copysign*, is a function that returns the first parameter with the same sign as the second parameter. Using the dot product is perfect because it ranges from -1 to 1.

Calculating the Pitch, Yaw, and Roll Angles of Multi-Axis Joints

This section will discuss how to calculate the pitch, yaw, and roll angles of a multi-axis joint such as the ball and socket joint of the right arm using the second setup of the 3-Space Sensor devices. We will be using the method described in [1] and the forward, down, and right vectors of the devices to derive these angles. Where v_{F0}, v_{D0}, and v_{R0} are respectively the forward, down, and right vectors of the first device which will be the reference device and v_{F1}, v_{D1}, v_{R1} are respectively the forward, down, and right vectors of the second device. The method takes the orientation of the arm and performs rotations in a precise order to move the arm back to its initial orientation to derive the pitch, yaw, and roll angles.

Before we start deriving the angles, we must transform v_{F1} to be on the same transverse plane as v_{F0}. To do this we will use the right vector of each device and calculate a quaternion that will rotate v_{R1} to line up with v_{R0}.

$$\theta = \arccos(v_{R1} \cdot v_{R0}) \quad \textit{Eqn. 37}$$

$$a = v_{R1} \times v_{R0} \quad \textit{Eqn. 38}$$

$$q_{Ro} = a\sin(\theta/2), \cos(\theta/2) \quad \textit{Eqn. 39}$$

The quaternion q_{Ro}, is the rotational offset for the right vectors. Now we can start decomposing the angles.

The first step in decomposing the angles is to undo any rotations on the vertical axis (this will be the yaw angle). So we will be using the forward vector of each device to calculate the yaw angle. However, we must first apply the offset q_{Ro} to v_{F1}.

$$v_{TF1} = q_{Ro} v_{F1} \quad \textit{Eqn. 40}$$

The vector v_{TF1}, is the transformed vector of v_{F1}. Now calculate the angle between v_{TF1} and v_{F0}.

$$\theta_{yaw} = \arccos(v_{TF1} \cdot v_{F0}) \quad \textit{Eqn. 41}$$

The angle θ_{yaw}, is the yaw angle of the joint. Now we must undo this rotation by calculating a quaternion and transforming v_{F1} again, but this time to be on the same frontal plane as v_{F0}. We must also transform v_{D1} to be used for a later calculation.

$$a = v_{TF1} \times v_{F0} \quad \textit{Eqn. 42}$$

$$q_\theta = a\sin(\theta_{yaw}/2), \cos(\theta_{yaw}/2) \quad \textit{Eqn. 43}$$

$$v_{TF1} = q_\theta v_{F1} \quad \textit{Eqn. 44}$$

$$v_{TD1} = q_\theta v_{D1} \quad \textit{Eqn. 45}$$

The quaternion q_θ, is the rotational offset of θ_{yaw}. Now take note that arccos will always return a positive value, so we must use the dot product of a and v_{R0} to calculate the sign of θ_{yaw}.

$$\theta_{yaw} = copysign(\theta_{yaw}, (a \cdot v_{R0})) \quad \textit{Eqn. 46}$$

The function *copysign*, is a function that returns the first parameter with the same sign as the second parameter, so using the dot product is perfect because it ranges from -1 to 1. Next we must undo any rotations on the frontal horizontal axis (this will be the pitch angle). So we will be using v_{TF1} and v_{F0} to calculate the pitch angle.

$$\varphi_{pitch} = \arccos(v_{TF1} \cdot v_{F0}) \quad \textit{Eqn. 47}$$

The angle ϕ_{pitch}, is the pitch angle of the joint. Now we must undo this rotation by calculating a quaternion and transforming v_{TD1} again, but this time so it will be on the same sagittal plane as v_{D0}.

$$a = v_{TF1} \times v_{F0} \quad \textit{Eqn. 48}$$

$$q_\varphi = a\sin(\varphi_{pitch}/2), \cos(\varphi_{pitch}/2) \quad \textit{Eqn. 49}$$

$$v_{TD1} = q_\varphi v_{TD1} \quad \textit{Eqn. 50}$$

The quaternion q_ϕ is the rotational offset of ϕ_{pitch}. Now we must use the dot product of a and v_{D0} to calculate the sign of ϕ_{pitch}.

$$\varphi_{pitch} = copysign(\varphi_{picth}, (a \cdot v_{D0})) \quad \textit{Eqn. 51}$$

Finally calculate any rotations on the sagittal horizontal axis (this will be the roll angle). We will be using v_{TD1} and v_{D0} to calculate the roll angle.

$$\psi_{roll} = \arccos(v_{TD1} \cdot v_{D0}) \quad \textit{Eqn. 52}$$

The angle ψ_{roll}, is the roll angle of the joint. Now we must calculate the sign of ψ_{roll} using the cross product of v_{TD1} and v_{D0}, a, and the dot product of a and v_{F0}.

$$a = v_{TD1} \times v_{D0} \quad \textit{Eqn. 53}$$

$$\psi_{roll} = copysign(\psi_{roll}, (a \cdot v_{F0})) \quad \textit{Eqn. 54}$$

The planes and axes mentioned are described in [2]. And as mentioned in [1], there are two positions in which the so called “gimbal-lock” will occur. Those positions are when the arm is straight up or straight down, making θ_{yaw} and ψ_{roll} ambiguous.

4. Software Implementation

The reference Python code in this documentation is written in Python 2.7, uses the internal libraries math and struct, and the external library, serial, which is provided by PySerial 2.6. A custom library, threespace, has functions for calculating the angles and performing vector ans quaternion operations.

Communication with the 3-Space Sensor devices is done with PySerial's Serial class. It takes the COM port name the device was given by the computer when plugged in to connect to it for writing and reading data to and from the device.

4.1 Connecting to a 3-Space Sensor Device Python Source Code

To connect a 3-Space Sensor devices, you need to know the COM port name the device is on. This can be found in the Device Manager or in the 3-Space Sensor Software Suite. This COM port name is used for creating a Serial class object to communicate to the 3-Space Sensor device.

```
def openPort(com_port):
    try:
        serial_port = serial.Serial(com_port, timeout=0.1, writeTimeout=0.1, baudrate=115200)
        return serial_port
    except Exception as ex:
        print "Failed to create a serial port on port:", com_port
        raise ex
```

4.2 Communicating with a 3-Space Sensor Device Python Source Code

The 3-Space Sensor device has a list of commands for getting and setting data. These commands must be sent to the 3-Space Sensor device through a command packet that must be created, and the serial port must read the full amount of bytes returned from the 3-Space Sensor device. The command packet is comprised of a header byte, the command byte(s), input data byte(s), and a checksum byte. The construction of these command packets and the amount of bytes to be read can be found in the 3-Space Sensor device's User Manual.

```
def commandWriteRead(serial_port, command, byte_size=0, data_format='', input_data=[]):
    """ Writes and reads data to and from a serial port given.

        Args:
            serial_port: A Serial object that is communicating with a 3-Space Sensor device.
            command: A char string of one of the 3-Space Sensor device's commands.
            byte_size: The number of bytes to read from the serial port.
            data_format: The format for which struct is to pack or unpack data.
            input_data: Data to be sent to the 3-Space Sensor device.
    """
    data_str = ''
    if len(input_data) > 0:
        data_str = struct.pack(data_format, *input_data)
        command_data = START_BYTE + command + data_str + createCheckSum(command + data_str)

    try:
        serial_port.write(command_data)
    except Exception as ex:
        print "There was an error writing command to the port", serial_port.name
        raise ex

    if byte_size > 0:
        try:
            data_str = serial_port.read(byte_size)
        except Exception as ex:
```

```
            print "There was an error reading from the port", serial_port.name
            raise ex

        output_data = list(struct.unpack(data_format, data_str))
        return output_data

    return None
```

4.3 Angle Calculation Python Source Code

The angle calculation function is listed below and returns the angle in radians. It uses the methods stated in the sections *Calculating the Angle of a Hinge Joint* and *Calculating the Angle of a Hinge Joint*.

```
def calculateAngle(vec0, vec1, vec2=None):
    """ Calculates the angle between the two given vectors using the dot product.

        Args:
            vec0: A unit vector.
            vec1: A unit vector.
            vec2: A unit vector perpendicular to vec0 and vec1.
    """
    ## The max and min is used to account for possible floating point error
    dot_product = max(min(vectorDot(vec0, vec1), 1.0), -1.0)
    angle = math.acos(dot_product)

    if vec2 is not None:
        axis = vectorNormalize(vectorCross(vec0, vec1))
        angle = math.copysign(angle, vectorDot(vec2, axis))

    return angle
```

4.4 Pitch, Yaw, and Roll Calculation Python Source Code

The function below is used to calculate the pitch, yaw, and roll angles in radians. It uses the method stated in section *Calculating the Pitch, Yaw, and Roll Angles of Multi-Axis Joints*.

```
def calculatePitchYawRoll(forward0, down0, forward1, down1):
    """ Calculates the pitch, yaw, and roll angles using the forward and down vectors calculated from
        two 3-Space Sensor devices.

        Args:
            forward0: A unit vector that denotes the forward vector of the first 3-Space Sensor device.
            down0: A unit vector that denotes the down vector of the first 3-Space Sensor device.
            forward1: A unit vector that denotes the forward vector of the second 3-Space Sensor
            device.
            down1: A unit vector that denotes the down vector of the second 3-Space Sensor device.
    """
    ## Assumes the devices' axis directions are default (XYZ) and are positioned or has had its
    ## orientation manipulated so that the Right axis is up

    ## First, calculate the right vector for both devices using the forward and down vectors
    right0 = vectorNormalize(vectorCross(forward0, down0))
    right1 = vectorNormalize(vectorCross(forward1, down1))

    ## Second, calculate the angle between the right vectors and a vector perpendicular to them
    angle = calculateAngle(right1, right0)
    axis = vectorNormalize(vectorCross(right1, right0))

    ## Third, create a quaternion using the calculated axis and angle that will be used to
    ## transform the forward vector of the second device so that it is on the same horizontal
    ## plane as the forward vector of the first device
    quat = createQuaternion(axis, angle)
    transformed_forward1 = vectorNormalize(quaternionVectorMultiplication(quat, forward1))

    ## Fourth, calculate the angle between the transformed forward vector and the forward vector
    ## of the first device
```

```
    ## This angle is the yaw
    yaw = calculateAngle(transformed_forward1, forward0)

    ## Fifth, calculate a vector perpendicular to the transformed forward vector and the forward vector
    ## of the first device
    axis = vectorNormalize(vectorCross(transformed_forward1, forward0))

    ## Sixth, create a quaternion using the calculated axis and yaw angle that will be used to
    ## transform the forward vector of the second device so that it is on the same vertical plane
    ## as the forward vector of the first device and to transform the down vector of the second
    ## device to be used in a later calculation
    quat = createQuaternion(axis, yaw)
    transformed_forward1 = vectorNormalize(quaternionVectorMultiplication(quat, forward1))
    transformed_down1 = vectorNormalize(quaternionVectorMultiplication(quat, down1))

    ## Set the sign of yaw using the axis calculated and the right vector of the first device
    yaw = math.copysign(yaw, vectorDot(axis, right0))

    ## Seventh, calculate the angle between the transformed forward vector and the forward vector of
    ## the first device
    ## This angle is the pitch
    pitch = calculateAngle(transformed_forward1, forward0)

    ## Eighth, calculate a vector perpendicular to the transformed forward vector
    ## and the forward vector of the first device
    axis = vectorNormalize(vectorCross(transformed_forward1, forward0))

    ## Ninth, create a quaternion using the calculated axis and pitch angle that will
    ## be used to transform the transformed down vector so that it is on the same vertical plane
    ## as the down vector of the first device
    quat = createQuaternion(axis, pitch)
    transformed_down1 = vectorNormalize(quaternionVectorMultiplication(quat, transformed_down1))

    ## Set the sign of pitch using the axis calculated and the down vector of the first device
    pitch = math.copysign(pitch, vectorDot(axis, down0))

    ## Tenth, calculate the angle between the transformed down vector and the down vector of the first
    ## device
    ## This angle is the roll
    roll = calculateAngle(transformed_down1, down0)
    axis = vectorNormalize(vectorCross(transformed_down1, down0))

    ## Set the sign of roll using the axis calculated and the forward vector of the first device
    roll = math.copysign(roll, vectorDot(axis, forward0))

    return [pitch, yaw, roll]
```

4.5 Compensation Offset Python Source Code

The function below is used to calculate the offset of the 3-Space Sensor device on the human body. The function returns the offset orientation as a quaternion.

```
def offsetQuaternion(serial_port, gravity=[-1.0, 0.0, 0.0], init_offset=None):
    """ Calculates the offset of the 3-Space Sensor device on the human body.

        Args:
            serial_port: A Serial object that is communicating with a 3-Space Sensor device.
            gravity: A unit vector that denotes the gravity direction the 3-Space Sensor device should
            be reading.
            init_offset: A unit quaternion the denotes a rotational offset for a 3-Space Sensor device.
    """
    ## First, find what the device reads as the gravity direction using the read North Gravity command
    ## The command returns 6 floats, the first 3 make the North vector and the last 3 make the Gravity
    ## vector
    north_gravity = commandWriteRead(serial_port, READ_NORTH_GRAVITY_COMMAND,

byte_size=24, data_format='>ffffff')
```

```
    sensor_gravity = north_gravity[3:]

    ## Second, read the current filtered orientation as a quaternion from the device using the read
    ## Filtered Quaternion command
    filt_data = commandWriteRead(serial_port, READ_FILT_QUAT_COMMAND, byte_size=16,

data_format='>ffff')

    ## Third, using the gravity vector given and the Gravity vector from the device,
    ## calculate the angle between them and a vector perpendicular to them
    angle = calculateAngle(sensor_gravity, gravity)
    axis = vectorNormalize(vectorCross(sensor_gravity, gravity))

    ## Fourth, create a quaternion using the calculated axis and angle that will be used to offset the
    ## filtered quaternion of the device so the gravity vectors will line up
    ## Also apply the initial offset if any
    offset = createQuaternion(axis, -angle)
    if init_offset is not None:
        offset = quaternionMultiplication(offset, init_offset)
    tare_data = quaternionMultiplication(filt_data, offset)

    ## Fifth, set the offset filtered quaternion as the tare orientation for the device using the set
Tare
    ## Quaternion command
    commandWriteRead(serial_port, SET_TARE_QUAT_COMMAND, data_format='>ffff',

input_data=tare_data)

    ## The calculated offset quaternion is returned because it needs to be applied
    ## to the filtered tared quaternion received from the device
    return offset
```

4.6 Full Python Source Code

Full Source Code of the Custom Library

```
#!/usr/bin/env python2.7

##
## Script Name: threespace.py
##
## Application Note: Calculating Angles Between Two YOST LABS 3-Space Sensor Devices using Two Vectors
##                   on a Human Body
##
## Description: Helper functions for calculating the angles between two YOST LABS 3-Space Sensor
devices in
##              Python 2.7
##
## Website: ww.yostlabs.com
##
## Copyright: Copyright (C) 2017 Yost Labs, Inc.
## Permission is hereby granted, free of charge, to any person obtaining a copy of this software
## associated documentation files (the "Software"), to deal in the Software without, including
## without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
## and/or sell copies of the Software, and to permit persons to whom the Software is furnished
## to do so, subject to the following conditions:
##     The above copyright notice and this permission notice shall be
##     included in all copies or substantial portions of the Software.
##     THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
##     BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
##     NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
##     DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
##     OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
##

import serial
import struct
```

```
import math

## Static Variables ##
START_BYTE = chr(0xf7)
READ_FILT_TARED_QUAT_COMMAND = chr(0x00)
READ_FILT_QUAT_COMMAND = chr(0x06)
READ_NORTH_GRAVITY_COMMAND = chr(0x0c)
SET_TARE_QUAT_COMMAND = chr(0x61)

def vectorCross(vec0, vec1):
    """ Performs the cross product on the two given vectors.

        Args:
            vec0: A unit vector.
            vec1: A unit vector.
    """
    x0, y0, z0 = vec0
    x1, y1, z1 = vec1

    return [y0 * z1 - z0 * y1, z0 * x1 - x0 * z1, x0 * y1 - y0 * x1]

def vectorDot(vec0, vec1):
    """ Performs the dot product on the two given vectors.

        Args:
            vec0: A unit vector.
            vec1: A unit vector.
    """
    x0, y0, z0 = vec0
    x1, y1, z1 = vec1

    return x0 * x1 + y0 * y1 + z0 * z1

def vectorLength(vec):
    """ Calculates the length of a vector given.

        Args:
            vec: A vector.
    """
    return (vectorDot(vec, vec) ** 0.5)

def vectorNormalize(vec):
    """ Normalizes the vector given.

        Args:
            vec: A vector.
    """
    length = vectorLength(vec)
    x, y, z = vec

    return [x / length, y / length, z / length]

def createQuaternion(vec, angle):
    """ Creates a quaternion from an axis and an angle.

        Args:
            vec: A unit vector.
            angle: An angle in radians.

    """
    ## Quaternions represent half the angle in complex space so the angle must be halved
    x, y, z = vec
    tmp_quat = [0.0] * 4
    tmp_quat[0] = x * math.sin(angle / 2.0)
    tmp_quat[1] = y * math.sin(angle / 2.0)
    tmp_quat[2] = z * math.sin(angle / 2.0)
    tmp_quat[3] = math.cos(angle / 2.0)
```

```
    ## Normalize the quaternion
    qx, qy, qz, qw = tmp_quat
    length = (qx * qx + qy * qy + qz * qz + qw * qw) ** 0.5

    tmp_quat[0] /= length
    tmp_quat[1] /= length
    tmp_quat[2] /= length
    tmp_quat[3] /= length

    return tmp_quat

def quaternionMultiplication(quat0, quat1):
    """ Performs quaternion multiplication on the two given quaternions.

        Args:
            quat0: A unit quaternion.
            quat1: A unit quaternion.
    """
    x0, y0, z0, w0 = quat0
    x1, y1, z1, w1 = quat1

    x_cross, y_cross, z_cross = vectorCross([x0, y0, z0], [x1, y1, z1])
    w_new = w0 * w1 - vectorDot([x0, y0, z0], [x1, y1, z1])
    x_new = x1 * w0 + x0 * w1 + x_cross
    y_new = y1 * w0 + y0 * w1 + y_cross
    z_new = z1 * w0 + z0 * w1 + z_cross

    return [x_new, y_new, z_new, w_new]

def quaternionVectorMultiplication(quat, vec):
    """ Rotates the given vector by the given quaternion.

        Args:
            quat: A unit quaternion.
            vec: A unit vector.
    """
    ## Procedure: quat * vec_quat * -quat
    qx, qy, qz, qw = quat
    vx, vy, vz = vec
    vw = 0.0
    neg_qx = -qx
    neg_qy = -qy
    neg_qz = -qz
    neg_qw = qw

    ## First Multiply
    x_cross, y_cross, z_cross = vectorCross([qx, qy, qz], vec)
    w_new = qw * vw - vectorDot([qx, qy, qz], vec)
    x_new = vx * qw + qx * vw + x_cross
    y_new = vy * qw + qy * vw + y_cross
    z_new = vz * qw + qz * vw + z_cross

    ## Second Multiply
    x_cross, y_cross, z_cross = vectorCross([x_new, y_new, z_new], [neg_qx, neg_qy, neg_qz])
    w = w_new * neg_qw - vectorDot([x_new, y_new, z_new], [neg_qx, neg_qy, neg_qz])
    x = neg_qx * w_new + x_new * neg_qw + x_cross
    y = neg_qy * w_new + y_new * neg_qw + y_cross
    z = neg_qz * w_new + z_new * neg_qw + z_cross

    return [x, y, z]

def calculateAngle(vec0, vec1, vec2=None):
    """ Calculates the angle between the two given vectors using the dot product.

        Args:
            vec0: A unit vector.
```

```
            vec1: A unit vector.
            vec2: A unit vector perpendicular to vec0 and vec1.
    """
    ## The max and min is used to account for possible floating point error
    dot_product = max(min(vectorDot(vec0, vec1), 1.0), -1.0)
    angle = math.acos(dot_product)

    if vec2 is not None:
        axis = vectorNormalize(vectorCross(vec0, vec1))
        angle = math.copysign(angle, vectorDot(vec2, axis))

    return angle

def calculatePitchYawRoll(forward0, down0, forward1, down1):
    """ Calculates the pitch, yaw, and roll angles using the forward and down vectors calculated from
        two 3-Space Sensor devices.

        Args:
            forward0: A unit vector that denotes the forward vector of the first 3-Space Sensor device.
            down0: A unit vector that denotes the down vector of the first 3-Space Sensor device.
            forward1: A unit vector that denotes the forward vector of the second 3-Space Sensor
            device.
            down1: A unit vector that denotes the down vector of the second 3-Space Sensor device.
    """
    ## Assumes the devices' axis directions are default (XYZ) and are positioned or has had its
    ## orientation manipulated so that the Right axis is up

    ## First, calculate the right vector for both devices using the forward and down vectors
    right0 = vectorNormalize(vectorCross(forward0, down0))
    right1 = vectorNormalize(vectorCross(forward1, down1))

    ## Second, calculate the angle between the right vectors and a vector perpendicular to them
    angle = calculateAngle(right1, right0)
    axis = vectorNormalize(vectorCross(right1, right0))

    ## Third, create a quaternion using the calculated axis and angle that will be used
    ## to transform the forward vector of the second device so that it is on the same horizontal
    ## plane as the forward vector of the first device
    quat = createQuaternion(axis, angle)
    transformed_forward1 = vectorNormalize(quaternionVectorMultiplication(quat, forward1))

    ## Fourth, calculate the angle between the transformed forward vector and the forward vector
    ## of the first device
    ## This angle is the yaw
    yaw = calculateAngle(transformed_forward1, forward0)

    ## Fifth, calculate a vector perpendicular to the transformed forward vector and the forward vector
    ## of the first device
    axis = vectorNormalize(vectorCross(transformed_forward1, forward0))

    ## Sixth, create a quaternion using the calculated axis and yaw angle that will be used
    ## to transform the forward vector of the second device so that it is on the same vertical
    ## plane as the forward vector of the first device and to transform the down vector of the
    ## second device to be used in a later calculation
    quat = createQuaternion(axis, yaw)
    transformed_forward1 = vectorNormalize(quaternionVectorMultiplication(quat, forward1))
    transformed_down1 = vectorNormalize(quaternionVectorMultiplication(quat, down1))

    ## Set the sign of yaw using the axis calculated and the right vector of the first device
    yaw = math.copysign(yaw, vectorDot(axis, right0))

    ## Seventh, calculate the angle between the transformed forward vector and the forward vector of
    ## the first device
    ## This angle is the pitch
    pitch = calculateAngle(transformed_forward1, forward0)

    ## Eighth, calculate a vector perpendicular to the transformed forward vector
```

```
    ## and the forward vector of the first device
    axis = vectorNormalize(vectorCross(transformed_forward1, forward0))

    ## Ninth, create a quaternion using the calculated axis and pitch angle that
    ## will be used to transform the transformed down vector so that it is on the same
    ## vertical plane as the down vector of the first device
    quat = createQuaternion(axis, pitch)
    transformed_down1 = vectorNormalize(quaternionVectorMultiplication(quat, transformed_down1))

    ## Set the sign of pitch using the axis calculated and the down vector of the first device
    pitch = math.copysign(pitch, vectorDot(axis, down0))

    ## Tenth, calculate the angle between the transformed down vector and the down vector
    ## of the first device
    ## This angle is the roll
    roll = calculateAngle(transformed_down1, down0)
    axis = vectorNormalize(vectorCross(transformed_down1, down0))

    ## Set the sign of roll using the axis calculated and the forward vector of the first device
    roll = math.copysign(roll, vectorDot(axis, forward0))

    return [pitch, yaw, roll]

def createCheckSum(char_data):
    """ Calculates the checksum for the given data.

        Args:
            char_data: A string of data.
    """
    checksum = 0
    for byte in char_data:
        checksum += ord(byte)
    return chr(checksum % 256)

def offsetQuaternion(serial_port, gravity=[-1.0, 0.0, 0.0], init_offset=None):
    """ Calculates the offset of the 3-Space Sensor device on the human body.

        Args:
            serial_port: A Serial object that is communicating with a 3-Space Sensor device.
            gravity: A unit vector that denotes the gravity direction the 3-Space Sensor device should
            be reading.
            init_offset: A unit quaternion the denotes a rotational offset for a 3-Space Sensor device.
    """
    ## First, find what the device reads as the gravity direction using the read North Gravity command
    ## The command returns 6 floats, the first 3 make the North vector and the last 3 make the Gravity
    ## vector
    north_gravity = commandWriteRead(serial_port, READ_NORTH_GRAVITY_COMMAND,
        byte_size=24, data_format='>ffffff')
    sensor_gravity = north_gravity[3:]

    ## Second, read the current filtered orientation as a quaternion from the device using the read
    ## Filtered Quaternion command
    filt_data = commandWriteRead(serial_port, READ_FILT_QUAT_COMMAND, byte_size=16,data_format='>ffff')

    ## Third, using the gravity vector given and the Gravity vector from the device,
    ## calculate the angle between them and a vector perpendicular to them
    angle = calculateAngle(sensor_gravity, gravity)
    axis = vectorNormalize(vectorCross(sensor_gravity, gravity))

    ## Fourth, create a quaternion using the calculated axis and angle that will be used to offset the
    ## filtered quaternion of the device so the gravity vectors will line up
    ## Also apply the initial offset if any
    offset = createQuaternion(axis, -angle)
    if init_offset is not None:
        offset = quaternionMultiplication(offset, init_offset)
    tare_data = quaternionMultiplication(filt_data, offset)
```

```
    ## Fifth, set the offset filtered quaternion as the tare orientation for the device
    ## using the set Tare Quaternion command
    commandWriteRead(serial_port, SET_TARE_QUAT_COMMAND, data_format='>ffff', input_data=tare_data)

    ## The calculated offset quaternion is returned because it needs to be applied
    ## to the filtered tared quaternion received from the device
    return offset

def openPort(com_port):
    try:
        serial_port = serial.Serial(com_port, timeout=0.1, writeTimeout=0.1, baudrate=115200)
        return serial_port
    except Exception as ex:
        print "Failed to create a serial port on port:", com_port
        raise ex

def closePort(serial_port):
    try:
        serial_port.close()
    except Exception as ex:
        print "Failed to close the port:", serial_port.name
        raise ex

def commandWriteRead(serial_port, command, byte_size=0, data_format='', input_data=[]):
    """ Writes and reads data to and from a serial port given.

        Args:
            serial_port: A Serial object that is communicating with a 3-Space Sensor device.
            command: A char string of one of the 3-Space Sensor device's commands.
            byte_size: The number of bytes to read from the serial port.
            data_format: The format for which struct is to pack or unpack data.
            input_data: Data to be sent to the 3-Space Sensor device.
    """
    data_str = ''
    if len(input_data) > 0:
        data_str = struct.pack(data_format, *input_data)
        command_data = START_BYTE + command + data_str + createCheckSum(command + data_str)

    try:
        serial_port.write(command_data)
    except Exception as ex:
        print "There was an error writing command to the port", serial_port.name
        raise ex

    if byte_size > 0:
        try:
            data_str = serial_port.read(byte_size)
        except Exception as ex:
            print "There was an error reading from the port", serial_port.name
            raise ex

        output_data = list(struct.unpack(data_format, data_str))
        return output_data

    return None

def calculateDeviceVector(serial_port, vec, offset):
    """ Calculates a vector in a 3-Space Sensor device's reference frame.

        Args:
            serial_port: A Serial object that is communicating with a 3-Space Sensor device.
            vec: A unit vector.
            offset: A unit quaternion that denotes the offset of the 3-Space Sensor device.
    """
    ## Get the filtered tared orientation of the device
    data = commandWriteRead(serial_port, READ_FILT_TARED_QUAT_COMMAND, byte_size=16,
```

```
data_format='>ffff')

    ## Apply the offset for the device
    quat = quaternionMultiplication(data, offset)

    ## Calculate a vector for the device with its orientation
    vector = quaternionVectorMultiplication(quat, vec)

    return vector
```

4.7 Full Source Code for Calculating the Angle of a Hinge Joint

```
#!/usr/bin/env python2.7

##
## Script Name: hinge.py
##
## Application Note: Calculating Angles Between Two YOST LABS 3-Space Sensor Devices using Two Vectors
##                   on a Human Body
##
## Description: Calculates the hinge angle between two YOST LABS 3-Space Sensor devices in Python 2.7
##
## Website: ww.yostlabs.com
##
## Copyright: Copyright (C) 2017 Yost Labs, Inc.
## Permission is hereby granted, free of charge, to any person obtaining a copy of this software
## associated documentation files (the "Software"), to deal in the Software without, including
## without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
## and/or sell copies of the Software, and to permit persons to whom the Software is furnished
## to do so, subject to the following conditions:
##     The above copyright notice and this permission notice shall be
##     included in all copies or substantial portions of the Software.
##     THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
##     BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
##     NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
##     DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
##     OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
##

import time
import threespace

## Change the COM port names as needed
## First device
serial_port0 = threespace.openPort("COM10")

## Second device
serial_port1 = threespace.openPort("COM11")

print "To quit, hold down \"Ctrl\" key and press \"C\" key"
print "---------------------------------------------------"
print "Get into the starting position."

time.sleep(3)
print "Please hold for 10 seconds to compensate for device positioning."
time.sleep(5)

## Calculate the rotational offset of the compensation for the the first device
offset0 = threespace.offsetQuaternion(serial_port0)

## Calculate the rotational offset of the compensation for the the second device
offset1 = threespace.offsetQuaternion(serial_port1)

time.sleep(2)
```

```
while True:
    ## Calculate the forward vector of the first device
    ## The initial forward vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading up the arm
    forward0 = threespace.calculateDeviceVector(serial_port0, [0.0, 0.0, 1.0], offset0)

    ## Calculate the forward vector of the second device
    ## The initial forward vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading up the arm
    forward1 = threespace.calculateDeviceVector(serial_port1, [0.0, 0.0, 1.0], offset1)

    ## Calculate a vector perpendicular to the forward vectors and parallel to the axis
    ## of rotation to use for determining the sign of the angle
    ## Using the first device's orientation will give the best results
    ## The initial vector to use depends on the initial forward vector
    up0 = threespace.calculateDeviceVector(serial_port0, [0.0, 1.0, 0.0], offset0)

    ## Calculate the angle between the two devices
    angle = threespace.calculateAngle(forward1, forward0, up0)
    ## Print as radians and degrees
    print "Hinge"
    print "Radians: %0.4f\tDegrees: %0.4f" % (angle, threespace.math.degrees(angle))
    print "============================================="

## Close the serial ports
threespace.closePort(serial_port0)
threespace.closePort(serial_port1)
```

4.8 Full Source Code for Calculating the Angle of a Pivot Joint

```
#!/usr/bin/env python2.7

##
## Script Name: pivot.py
##
## Application Note: Calculating Angles Between Two YOST LABS 3-Space Sensor Devices using Two Vectors
##                   on a Human Body
##
## Description: Calculates the pivot angle between two YOST LABS 3-Space Sensor devices in Python 2.7
##
## Website: ww.yostlabs.com
##
## Copyright: Copyright (C) 2017 Yost Labs, Inc.
## Permission is hereby granted, free of charge, to any person obtaining a copy of this software
## associated documentation files (the "Software"), to deal in the Software without, including
## without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
## and/or sell copies of the Software, and to permit persons to whom the Software is furnished
## to do so, subject to the following conditions:
##     The above copyright notice and this permission notice shall be
##     included in all copies or substantial portions of the Software.
##     THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
##     BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
##     NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
##     DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
##     OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
##

import time
import threespace

## Change the COM port names as needed
## First device
serial_port0 = threespace.openPort("COM10")

## Second device
```

```
serial_port1 = threespace.openPort("COM11")

print "To quit, hold down \"Ctrl\" key and press \"C\" key"
print "-------------------------------------------------"
print "Get into the starting position."

time.sleep(3)
print "Please hold for 10 seconds to compensate for device positioning."
time.sleep(5)

## Calculate the rotational offset of the compensation for the the first device
offset0 = threespace.offsetQuaternion(serial_port0)

## Calculate the rotational offset of the compensation for the the second device
offset1 = threespace.offsetQuaternion(serial_port1)

time.sleep(2)

while True:
    ## Calculate the down vector of the first device with its orientation
    ## The initial down vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading into the arm
    down0 = threespace.calculateDeviceVector(serial_port0, [0.0, -1.0, 0.0], offset0)

    ## Calculate the down vector of the second device with its orientation
    ## The initial down vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading into the arm
    down1 = threespace.calculateDeviceVector(serial_port1, [0.0, -1.0, 0.0], offset1)

    ## Calculate a vector perpendicular to the down vectors and parallel to the axis of rotation
    ## to use for determining the sign of the angle
    ## Using the first device's orientation will give the best results
    ## The initial vector to use depends on the initial down vector
    forward0 = threespace.calculateDeviceVector(serial_port0, [0.0, 0.0, 1.0], offset0)

    ## Calculate the angle between the two devices
    angle = threespace.calculateAngle(down1, down0, forward0)
    ## Print as radians and degrees
    print "Pivot"
    print "Radians: %0.4f\tDegrees: %0.4f" % (angle, threespace.math.degrees(angle))
    print "============================================="

## Close the serial ports
threespace.closePort(serial_port0)
threespace.closePort(serial_port1)
```

4.9 Full Source Code for Calculating the Pitch, Yaw, and Roll Angles of a Multi-Axis Joint

```
!/usr/bin/env python2.7

##
## Script Name: multi_axis.py
##
## Application Note: Calculating Angles Between Two YOST LABS 3-Space Sensor Devices using Two Vectors
##                   on a Human Body
##
## Description: Calculates the pitch, yaw, and roll angles between two YOST LABS 3-Space Sensor devices
in
##              Python 2.7
##
## Website: ww.yostlabs.com
##
## Copyright: Copyright (C) 2017 Yost Labs, Inc.
## Permission is hereby granted, free of charge, to any person obtaining a copy of this software
## associated documentation files (the "Software"), to deal in the Software without, including
```

```
## without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
## and/or sell copies of the Software, and to permit persons to whom the Software is furnished
## to do so, subject to the following conditions:
##     The above copyright notice and this permission notice shall be
##     included in all copies or substantial portions of the Software.
##     THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
##     BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
##     NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
##     DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
##     OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
##


import time
import threespace

## Change the COM port names as needed
## First device
serial_port0 = threespace.openPort("COM10")

## Second device
serial_port1 = threespace.openPort("COM11")

print "To quit, hold down \"Ctrl\" key and press \"C\" key"
print "----------------------------------------------------"
print "Get into the starting position."

time.sleep(3)
print "Please hold for 10 seconds to compensate for device positioning."
time.sleep(5)

## Create a quaternion to be used to orient the first device to the same orientational space
## as the second device
quat = threespace.createQuaternion([0.0, 1.0, 0.0], -threespace.math.pi / 2.0)

## Calculate the rotational offset of the compensation for the the first device
offset0 = threespace.offsetQuaternion(serial_port0, [0.0, 0.0, -1.0], quat)

## Calculate the rotational offset of the compensation for the the second device
offset1 = threespace.offsetQuaternion(serial_port1)

time.sleep(2)

while True:
    ## Calculate the forward and down vectors of the first device with its orientation
    ## The initial forward vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading to the left of the body
    forward0 = threespace.calculateDeviceVector(serial_port0, [0.0, 0.0, 1.0], offset0)
    ## The initial down vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading into the body
    down0 = threespace.calculateDeviceVector(serial_port0, [0.0, -1.0, 0.0], offset0)

    ## Calculate the forward and down vectors of the second device with its orientation
    ## The initial forward vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading up the arm
    forward1 = threespace.calculateDeviceVector(serial_port1, [0.0, 0.0, 1.0], offset1)
    ## The initial down vector to use depends on the orientation and axis direction of the device
    ## The resultant vector must be heading into the arm
    down1 = threespace.calculateDeviceVector(serial_port1, [0.0, -1.0, 0.0], offset1)

    ## Calculate the Pitch Yaw and Roll between the two devices
    angle_list = threespace.calculatePitchYawRoll(forward0, down0, forward1, down1)
    ## Print as radians and degrees
    print "Pitch"
    print "Radians: %0.4f\tDegrees: %0.4f" % (angle_list[0], threespace.math.degrees(angle_list[0]))
    print "============================================="
    print "Yaw"
    print "Radians: %0.4f\tDegrees: %0.4f" % (angle_list[1], threespace.math.degrees(angle_list[1]))
```

```
    print "============================================="
    print "Roll"
    print "Radians: %0.4f\tDegrees: %0.4f" % (angle_list[2], threespace.math.degrees(angle_list[2]))
    print "============================================="

## Close the serial ports
threespace.closePort(serial_port0)
threespace.closePort(serial_port1)
```

5. References

1. Doorenbosch, Harlaar, and Veeger. The globe system: An unambiguous description of shoulder positions in daily life movements
   http://www.rehab.research.va.gov/jour/03/40/2/PDF/doorenbosch.pdf
2. Understanding Exercise – Planes, Axes and Movement
   http://www.todaysfitnesstrainer.com/understanding-exercise-planes-axes-movement/
3. The Joints
   http://www.shockfamily.net/skeleton/JOINTS.HTML
4. Five Different Types of Joints
   http://www.livestrong.com/article/115889-five-different-types-joints/
5. Open, Sesamoid: Types of Joints
   http://science.howstuffworks.com/life/human-biology/bone11.htm

YOST LABS
630 Second Street
Portsmouth Ohio 45662, USA

Phone: 740.876.4936
info@yostlabs.com
yostlabs.com

Made in USA. Patents:
8498827, 8682610,
9255799, 9354058.
Additional patents pending.

About Yost Labs, Inc. We are a fast growing private company based in historic Portsmouth, Ohio. With over a decade of experience in low-latency inertial sensor innovation, we enable motion tracking in many of today's and tomorrow's most exciting products. We make virtual reality interactive. We stabilize drones and navigate autonomous cars. We measure human motion for athletic performance and rehabilitation. We are dedicated to supporting you and your team—providing expert advice and integration consulting for the world's fastest inertial motion sensor technology.

Yost Labs' innovation has been recognized with numerous patents with additional patents pending. Our customers and value-added resellers include the US Navy, US Air Force, NASA, US Army Corps of Engineers and over 1,000 leading technology firms and academic institutions around the world.