Brad Pitney

12/16/2013

ECE 578

Perkowski

Homework #4: Kinematics Modeling of the Wrist Joints of the Countess Quanta Robot

# Introduction

The right arm of the Countess Quanta robot is controlled by six servo motors. Two ‘wrist joint’ servos at the end of the arm control the rotation and angle of the right hand. In this project, the kinematics of this wrist is modeled to calculate the location of the tip of the middle finger of the hand in Cartesian coordinates, for a specified set of servo angles. Inverse kinematics is used to find the required servo angles to place the fingertip at the specified Cartesian coordinates. A C# program was created to apply these calculations and provide a graphical representation of the robot end effector. A linear path is modeled with this software, to simulate an instrument strumming motion made with the hand.

# Problem Description

The wrist joints of the right arm are composed of two servos, numbered Servo 1 and Servo 0. Servo 1 controls wrist rotation and is the second to last servo from the end of the arm. With the hand in a neutral position, the rotation axis of Servo 1 extends down the length of the hand’s middle finger. Changing the position of Servo 1 allows for rotating the palm of the hand towards or away from the robot’s body. Servo 0 is mounted at the end of the arm, so the orientation of Servo 0 depends on position of Servo 1.

Servo 0 controls the side-to-side bend in the wrist, which affects the direction the fingertips are pointing. The rotation axis of Servo 0 extends out from the back of the robot’s hand, and is perpendicular to the rotation axis of Servo 1. The figures below show the physical range of Servo 0, and its effect on the robot’s hand.

 

# Design Process

To model the wrist kinematics, I started by drawing pictures of the physical system. To restrict the scope of this project and focus on the effects of the two wrist servos, I simplified some properties of this system. These simplifications would need to be reviewed if this model were to be applied to the actual robot.

* The distance from Servo 0 to the tip of the robot’s middle finger is 6.5 inches. I use this length, but disregard the other dimensions of the finger.
* I ignore the other fingers, for the purpose of this model.
* I deal with the servo positions as angles (in degrees) rather than motor coordinates.
* I ignore the physical range limitations of the servo motors and assume that Servo 1 can rotate 360 degrees and Servo 0 can bend in a range of 0 to 180 degrees.

The model focuses on the rotation of the two servos and the 3D Cartesian position of the tip of the middle finger. The rotation of Servo 1 is used to orient Servo 0, which is then used to bend the wrist and move the fingertip off of the rotation axis. The figure below shows this 3D model while viewed from both the top and the side.



In this model, the robot’s wrist is oriented upwards, with the rotation axis of Servo 1 following the Z axis. The left image is the top-down view showing angle **a1** as the rotation of Servo 1. Servo 1’s rotation is measured counterclockwise, with the positive X axis representing 0 degrees. The top-down view shows the X and Y axes, so the XY plane represents a horizontal surface that the robot hand extends up from. The image on the right shows a side view of the device, with **a0** representing the angle of Servo 0, with respect to the positive D axis. The D axis is set parallel to the vector shown in the left image, so the displayed **a0** angle is always independent of the **a1** rotation. The right image also shows the Z axis, with represents the ‘up/down’ direction is this framework. The distance **L** represents the distance from the Servo 0 rotation axis to the tip of the middle finger. The large circles in each image represent the move range of the fingertip (i.e. along the surface of a half-sphere with radius L).

The forward kinematics for this system can be calculated using simple trigonometry. For this, we assume that length **L** and angles **a0** and **a1** are known. With values **L** and **a0** we can calculate the distances **z** and **d** from the right image. Once distance **d** is known, this can be used with **a1** to calculate the **x** and **y** coordinates on the left image. It can be shown that the following equations are true:

The inverse kinematics for this system can be calculated with similar trigonometry. We assume that values **L**, **x**, and **y** are known. The left image shows that angle **a1** and distance **d** can be calculated from **x** and **y**. In the right image, distances **d** and **L** can be used to calculate angle **a0** and the **z** coordinate. This leads to the following equations:

Note that in this system, the **z** coordinate cannot be set independently of the values of **x**, **y**, and **L**. This is due to the system restricting the fingertip location to positions along the surface of a half-sphere extending from the XY plane. Along this surface, each **xy** coordinate returns a single **z** coordinate, so this value can always be calculated from **x**, **y**, and **L**.

Once the model was created and the kinematics equations were defined, I created a ‘WristKinematics’ program in C# to automate these calculations and provide a visual representation of the specified state. This program displays a form with text boxes for the rotation (angle a1), bend (angle a0), length, X, Y, and Z values. Two buttons are provided to perform either the forward or inverse kinematics calculations with the specified values. When one of these calculations is performed, the calculated values are entered in their textboxes and the visual display is updated. The display shows the top-down view and side view of the model, for the last calculated values.

# Results

The figure below shows example output from a forward kinematics calculation.



In this example, the Rotation angle (**a1**) has been set to 45 degrees and the Bend angle (**a0**) has been set to 60 degrees. With the finger length set to 6.5 inches, the software calculates that the fingertip will be placed 2.3 inches away from the rotation axis in both the **x** and **y** directions, and 5.63 inches in the **z** direction. The display gives a visual representation of this state.

The figure below shows example output after running an inverse kinematics calculation.



In this example, the **x** and **y** coordinates for the fingertip have been specified: 4 inches in the **x** direction and 2.5 inches in the **y** direction. The software calculates the fingertip to be 4.47 inches in the z direction. To reach these XYZ coordinates with the fingertip, it determines that the Rotation angle (**a1**) should be set to 32.01degrees and the Bend angle (**a0**) should be set to 43.47 degrees. The display reflects these angles.

One application of this modeling is in determining the robot motions required for moving the end effector along a path. In the case of the Countess Quanta robot, the right hand is used in playing an attached harp instrument, so a path might be defined to move the fingers across the strings. In this case, we would want to move the finger along a linear path, in order to strum the strings of the instrument without colliding with the surface of the instrument itself. To simulate this scenario, we can specify a line in the XY plane to represent the desired strumming path, and use the software to calculate the servo angles required to implement this path.

As an example, suppose that the instrument is placed such that we want to move the fingertip linearly in the XY plane, following the line described by:

Essentially, we want to instruct the robot to move the fingertip from some starting point on this line, say XY coordinate (-6, -2.2), to an ending point at perhaps (4, 4.8). We want the robot to move in such a way that the path of the fingertip appears linear when viewed from above. To do this, we can break the move into small increments and calculate the servo angles required to reach each increment. The figure below shows overlapped screenshots from my WristKinematics program, while calculating angles at each 1 inch **x** increment.



The following table shows the calculated values for the specified **x** and **y** coordinates along this line. The **a1** and **a0** columns show the required servo angles to place the fingertip at each of these **xy** coordinates.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **y** | **z** | **a1** | **a0** |
| -6 | -2.2 | 1.19 | 200.14 | 10.53 |
| -5 | -1.5 | 3.87 | 196.70 | 36.57 |
| -4 | -0.8 | 5.06 | 191.31 | 51.13 |
| -3 | -0.1 | 5.77 | 181.91 | 62.50 |
| -2 | 0.6 | 6.16 | 163.30 | 71.26 |
| -1 | 1.3 | 6.29 | 127.57 | 75.38 |
| 0 | 2 | 6.18 | 90.00 | 72.08 |
| 1 | 2.7 | 5.83 | 69.68 | 63.71 |
| 2 | 3.4 | 5.17 | 59.53 | 52.64 |
| 3 | 4.1 | 4.05 | 53.81 | 38.59 |
| 4 | 4.8 | 1.79 | 50.19 | 16.00 |

These servo angles could then be converted to servo motor positions and sent to Servo 0 and Servo 1 in the robot motor control software. To maintain constant speed of the fingertip throughout the move, the velocity setting of each servo would need to be adjusted for each increment so that both servos complete their increment at the same time.

# Conclusion

The kinematics of the wrist of the right arm of the Countess Quanta robot was modeled by analyzing the last two servos at the end of the arm. These servos were modeled and used to create equations for the forward and inverse kinematics calculations. The ‘WristKinematics’ C# program was created to implement these calculations and to generate a visual representation of this model. Examples of the forward and inverse kinematics calculations were presented. An application of this model was examined by simulating a linear motion path and calculating the required servo coordinates to navigate the robot fingertip along this path.

# Source Code

// Brad Pitney

// ECE 578

// Fall 2013

// The WristKinematics project models the wrist joint of the right arm of the

// Countess Quanta robot. It calculates the forward and inverse kinematics of

// the joint servos, and creates a top-down and side view display of this joint.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

namespace WristKinematics

{

 /// <summary>

 /// Interaction logic for MainWindow.xaml

 /// </summary>

 public partial class MainWindow : Window

 {

 public MainWindow()

 {

 InitializeComponent();

 }

 Line aboveLine, sideLine;

 double aboveCenterTop, aboveCenterLeft, sideCenterTop, sideCenterLeft;

 // This method initializes the display of the model.

 private void WindowLoaded(object sender, RoutedEventArgs e)

 {

 aboveCenterTop = Canvas.GetTop(aboveCircle) + aboveCircle.Height / 2;

 aboveCenterLeft = Canvas.GetLeft(aboveCircle) + aboveCircle.Width / 2;

 sideCenterTop = Canvas.GetTop(sideCircle) + sideCircle.Height / 2;

 sideCenterLeft = Canvas.GetLeft(sideCircle) + sideCircle.Width / 2;

 // Create axis lines.

 Line aboveXAxis = new Line();

 aboveXAxis.Stroke = System.Windows.Media.Brushes.Gray;

 aboveXAxis.StrokeThickness = 1;

 aboveXAxis.X1 = aboveCenterLeft - aboveCircle.Width / 2 - 30;

 aboveXAxis.Y1 = aboveCenterTop;

 aboveXAxis.X2 = aboveCenterLeft + aboveCircle.Width / 2 + 30;

 aboveXAxis.Y2 = aboveCenterTop;

 DisplayCanvas.Children.Add(aboveXAxis);

 Line aboveYAxis = new Line();

 aboveYAxis.Stroke = System.Windows.Media.Brushes.Gray;

 aboveYAxis.StrokeThickness = 1;

 aboveYAxis.X1 = aboveCenterLeft;

 aboveYAxis.Y1 = aboveCenterTop - aboveCircle.Height / 2 - 30;

 aboveYAxis.X2 = aboveCenterLeft;

 aboveYAxis.Y2 = aboveCenterTop + aboveCircle.Height / 2 + 30;

 DisplayCanvas.Children.Add(aboveYAxis);

 Line sideXAxis = new Line();

 sideXAxis.Stroke = System.Windows.Media.Brushes.Gray;

 sideXAxis.StrokeThickness = 1;

 sideXAxis.X1 = sideCenterLeft - sideCircle.Width / 2 - 30;

 sideXAxis.Y1 = sideCenterTop;

 sideXAxis.X2 = sideCenterLeft + sideCircle.Width / 2 + 30;

 sideXAxis.Y2 = sideCenterTop;

 DisplayCanvas.Children.Add(sideXAxis);

 Line sideYAxis = new Line();

 sideYAxis.Stroke = System.Windows.Media.Brushes.Gray;

 sideYAxis.StrokeThickness = 1;

 sideYAxis.X1 = sideCenterLeft;

 sideYAxis.Y1 = sideCenterTop - sideCircle.Height / 2 - 30;

 sideYAxis.X2 = sideCenterLeft;

 sideYAxis.Y2 = sideCenterTop + sideCircle.Height / 2 + 30;

 DisplayCanvas.Children.Add(sideYAxis);

 // Create lines in the model, and move servo and fingertip markers to initial positions.

 Canvas.SetTop(aboveCenterMark, aboveCenterTop - aboveCenterMark.Height / 2);

 Canvas.SetLeft(aboveCenterMark, aboveCenterLeft - aboveCenterMark.Width / 2);

 Canvas.SetTop(sideCenterMark, sideCenterTop - sideCenterMark.Height / 2);

 Canvas.SetLeft(sideCenterMark, sideCenterLeft - sideCenterMark.Width / 2);

 Line sideLineDown = new Line();

 sideLineDown.Stroke = System.Windows.Media.Brushes.Blue;

 sideLineDown.StrokeThickness = 2.5;

 sideLineDown.X1 = sideCenterLeft;

 sideLineDown.Y1 = sideCenterTop;

 sideLineDown.X2 = sideCenterLeft;

 sideLineDown.Y2 = sideCenterTop + 150;

 DisplayCanvas.Children.Add(sideLineDown);

 aboveLine = new Line();

 aboveLine.Stroke = System.Windows.Media.Brushes.Blue;

 aboveLine.StrokeThickness = 2.5;

 aboveLine.X1 = aboveCenterLeft;

 aboveLine.Y1 = aboveCenterTop;

 aboveLine.X2 = aboveCenterLeft - 50;

 aboveLine.Y2 = aboveCenterTop - 50;

 DisplayCanvas.Children.Add(aboveLine);

 sideLine = new Line();

 sideLine.Stroke = System.Windows.Media.Brushes.Blue;

 sideLine.StrokeThickness = 2.5;

 sideLine.X1 = sideCenterLeft;

 sideLine.Y1 = sideCenterTop;

 sideLine.X2 = sideCenterLeft - 50;

 sideLine.Y2 = sideCenterTop - 50;

 DisplayCanvas.Children.Add(sideLine);

 Canvas.SetTop(aboveEndMark, aboveCenterTop - aboveEndMark.Height / 2 - 50);

 Canvas.SetLeft(aboveEndMark, aboveCenterLeft - aboveEndMark.Height / 2 - 50);

 Canvas.SetTop(sideEndMark, sideCenterTop - sideEndMark.Height / 2 - 50);

 Canvas.SetLeft(sideEndMark, sideCenterLeft - sideEndMark.Height / 2 - 50);

 // Create divider lines.

 Line divider = new Line();

 divider.Stroke = System.Windows.Media.Brushes.Black;

 divider.StrokeThickness = 1;

 divider.X1 = DisplayCanvas.ActualWidth / 2;

 divider.Y1 = 0;

 divider.X2 = DisplayCanvas.ActualWidth / 2;

 divider.Y2 = 300;

 DisplayCanvas.Children.Add(divider);

 Line divider2 = new Line();

 divider2.Stroke = System.Windows.Media.Brushes.Black;

 divider2.StrokeThickness = 1;

 divider2.X1 = 0;

 divider2.Y1 = 300;

 divider2.X2 = DisplayCanvas.ActualWidth;

 divider2.Y2 = 300;

 DisplayCanvas.Children.Add(divider2);

 }

 // This method takes the latest coordinates and updates the visual model.

 private void UpdateDisplay(double x, double y, double z, double d, double length)

 {

 // Scale the coordinates to the display.

 double xScaled = (x / length) \* (aboveCircle.Width / 2);

 double yScaled = (y / length) \* (aboveCircle.Height / 2);

 double zScaled = (z / length) \* (sideCircle.Height / 2);

 double dScaled = (d / length) \* (sideCircle.Width / 2);

 // Update the line segments.

 aboveLine.X2 = aboveCenterLeft + xScaled;

 aboveLine.Y2 = aboveCenterTop - yScaled;

 sideLine.X2 = sideCenterLeft + dScaled;

 sideLine.Y2 = sideCenterTop - zScaled;

 // Update the circle markers.

 Canvas.SetTop(aboveEndMark, aboveCenterTop - aboveEndMark.Height / 2 - yScaled);

 Canvas.SetLeft(aboveEndMark, aboveCenterLeft - aboveEndMark.Height / 2 + xScaled);

 Canvas.SetTop(sideEndMark, sideCenterTop - sideEndMark.Height / 2 - zScaled);

 Canvas.SetLeft(sideEndMark, sideCenterLeft - sideEndMark.Height / 2 + dScaled);

 }

 // This method performs the forward kinematics calculations.

 private void toCartesianButton\_Click(object sender, RoutedEventArgs e)

 {

 double rotationAngle = GetRadians(Convert.ToDouble(rotationTextbox.Text));

 double bendAngle = GetRadians(Convert.ToDouble(bendTextbox.Text));

 double length = Convert.ToDouble(lengthTextbox.Text);

 // Calculate x, y, z, d from the provided bendAngle and rotationAngle.

 double x = length \* Math.Cos(bendAngle) \* Math.Cos(rotationAngle);

 double y = length \* Math.Cos(bendAngle) \* Math.Sin(rotationAngle);

 double z = length \* Math.Sin(bendAngle);

 double d = length \* Math.Cos(bendAngle);

 xTextbox.Text = x.ToString("F2");

 yTextbox.Text = y.ToString("F2");

 zTextbox.Text = z.ToString("F2");

 UpdateDisplay(x, y, z, d, length);

 }

 // This method performs the inverse kinematics calculations.

 private void toAnglesButton\_Click(object sender, RoutedEventArgs e)

 {

 double x = Convert.ToDouble(xTextbox.Text);

 double y = Convert.ToDouble(yTextbox.Text);

 double length = Convert.ToDouble(lengthTextbox.Text);

 // Calculate z, d, rotationAngle, and bendAngle from the provided x and y coordinates.

 double rotationAngle = Math.Atan(y / x);

 double d = y / Math.Sin(rotationAngle);

 double bendAngle = Math.Acos( d / length );

 double z = length \* Math.Sin(bendAngle);

 // If bendAngle calculation exceeds 90 degrees, mirror this and adjust the

 // rotation angle.

 if (bendAngle > Math.PI / 2)

 {

 bendAngle = Math.PI - bendAngle;

 rotationAngle = rotationAngle + Math.PI;

 }

 // Keep rotationAngle between 0 and 360 degrees.

 if (rotationAngle < 0)

 {

 rotationAngle += 2 \* Math.PI;

 }

 rotationTextbox.Text = GetDegrees(rotationAngle).ToString("F2");

 bendTextbox.Text = GetDegrees(bendAngle).ToString("F2");

 zTextbox.Text = z.ToString("F2");

 UpdateDisplay(x, y, z, Math.Abs(d), length);

 }

 private double GetRadians(double degrees)

 {

 return (degrees \* Math.PI / 180);

 }

 private double GetDegrees(double radians)

 {

 return (radians \* 180 / Math.PI);

 }

 }

}