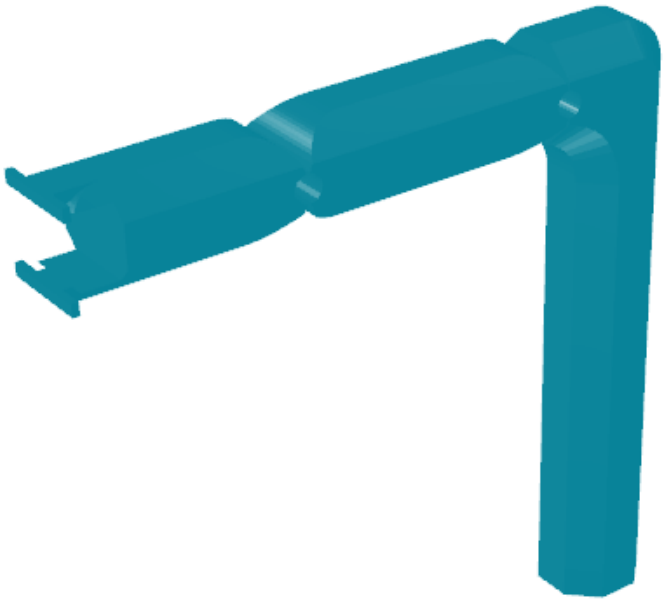


Analysis of 3R Anthropomorphic Robot Manipulator

YILDIZ TECHNICAL UNIVERSITY | MECHATRONICS ENGINEERING

Anıl ÖZTÜRK | MKT4141 Robot Engineering

Prof. Dr. Vasfi Emre ÖMÜRLÜ | HW 1 | 24.10.2017

Proof of Individuality: I, hereby, accept that I, myself, have prepared, written, solved, this homework based on my knowledge and research, although I would have gotten some conceptual help and studied the topics generally with my classmates, etc. Otherwise proven, I accept the grade given.

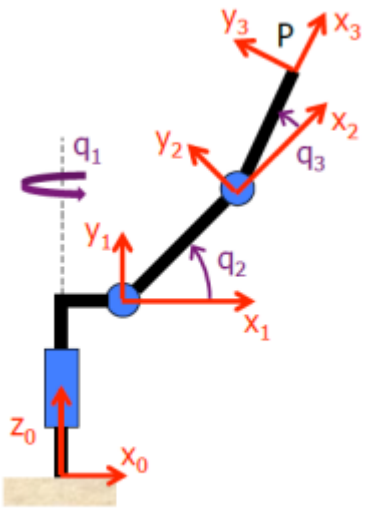
Signature:

DISCUSSION TOPICS

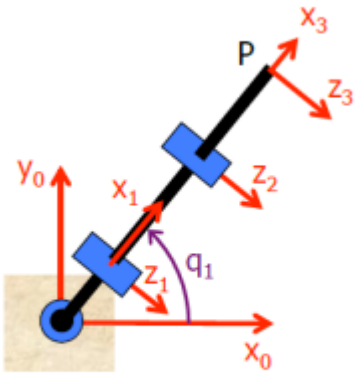
In this report, the following subjects are going to be reviewed:

- Visualization and discussion of 3D workspace of the robot manipulator
- Drawing 2D cross section plots of the 3D workspace
- Calculating volume of the 3D workspace
- Drawing a straight-line trajectory with 3D model of the robot manipulator

The robotic manipulator shown as below:



side view



top view

The end-effector's (x,y,z) position matrix is defined as:

$$\boldsymbol{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} \cos q_1 \left(a_1 + a_2 \cos q_2 + a_3 \cos(q_2 + q_3)\right) \\ \sin q_1 \left(a_1 + a_2 \cos q_2 + a_3 \cos(q_2 + q_3)\right) \\ d_1 + a_2 \sin q_2 + a_3 \sin(q_2 + q_3) \end{pmatrix} = \boldsymbol{f}(\boldsymbol{q}).$$

VISUALIZATION OF THE 3D WORKSPACE

The robot manipulator creates a workspace with reaching some points with its end-effector, in joints' angle range of course. In this part, every point that the robot manipulator can reach must be found and visualized on a 3D plot. The **ILNumerics** expansion package has been used for plotting the points on 3-axial plane.

We need to storage position values for every angle combinations of the joints can reach. Incrementing 10 degrees for loop is enough to recognize the shape of the plot. We have the joint limits as below:

$$q_1 \in [-150°, +150°], \quad q_2 \in [-140°, +140°], \quad q_3 \in [-100°, +100°].$$

We are going to increment 10 degrees each of them in one loop. From that information, we are going to need an array that has **e** elements.

$$\boldsymbol{e} = \frac{(150 - (-150) + 1)\, x\, (140 - (-140) + 1)\, x\, (100 - (-100) + 1)}{10} = \mathbf{18879}$$

```
float[,] position_matrix = new float[18879, 3];
int loopcounter = 0;
```

We defined the position matrix for every combination of angles and (x,y,z) positions. Then we defined the counter for our angle loop.

```
for (int joint1 = -150; joint1 < 151; joint1++)
{
    for (int joint2 = -140; joint2 < 141; joint2++)
    {
        for (int joint3 = -100; joint3 < 101; joint3++)
        {
                    (The array indexing algorithm)
                    joint3 += 9;
        }
        joint2 += 9;
    }
    joint1 += 9;
}
```

We constructed our for loop for every angle. And then incremented them by 9 separately for getting 10 angle increment ratio. Now we must construct the array indexing algorithm.

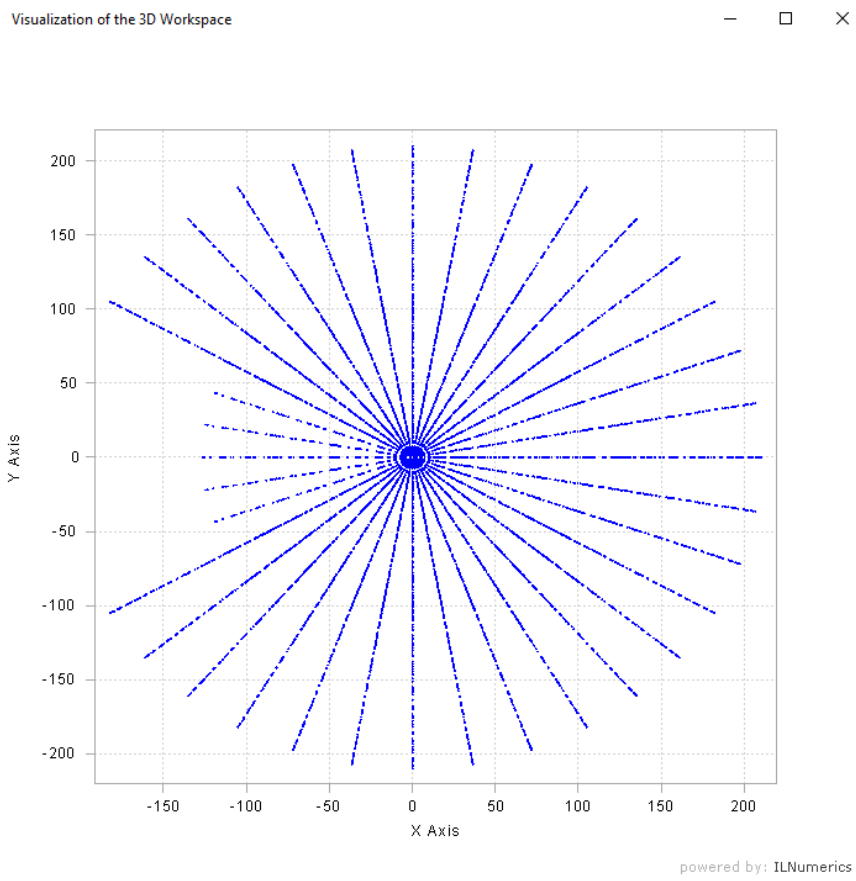
```
position_matrix[loopcounter,0] = (float)(Math.Cos(q1 * Math.PI / 180) * (30 + 100 * Math.Cos(q2
* Math.PI / 180) + 80 * Math.Cos((q2 + q3) * Math.PI / 180)));

position_matrix[loopcounter,1] = (float)(Math.Sin(q1 * Math.PI / 180) * (30 + 100 * Math.Cos(q2
* Math.PI / 180) + 80 * Math.Cos((q2 + q3) * Math.PI / 180)));

position_matrix[loopcounter,2] = (float)(200 + 100 * Math.Sin(q2 * Math.PI / 180) + 80 *
Math.Sin((q2 + q3) * Math.PI / 180));

loopcounter++;
```

This will be our array indexing algorithm. We are executing the forward kinematics equations to get values of position elements, then we are copying them into our position matrix.

```
        ILArray<float> position_matrix_ILArray = position_matrix;
        var workspacescene = new ILScene {
        new ILPlotCube(twoDMode: false) {

            new ILPoints {
                Positions = position_matrix_ILArray,
                Color = Color.FromArgb(0,0,255),
                Size = 1,
                         }
        }
        };
            var scene_delegate = workspacescene.First<ILPlotCube>().ScaleModes;
            scene_delegate.XAxisScale = AxisScale.Linear;
            scene_delegate.YAxisScale = AxisScale.Linear;
            scene_delegate.ZAxisScale = AxisScale.Linear;

            _3D_plot.Scene = workspacescene;
```

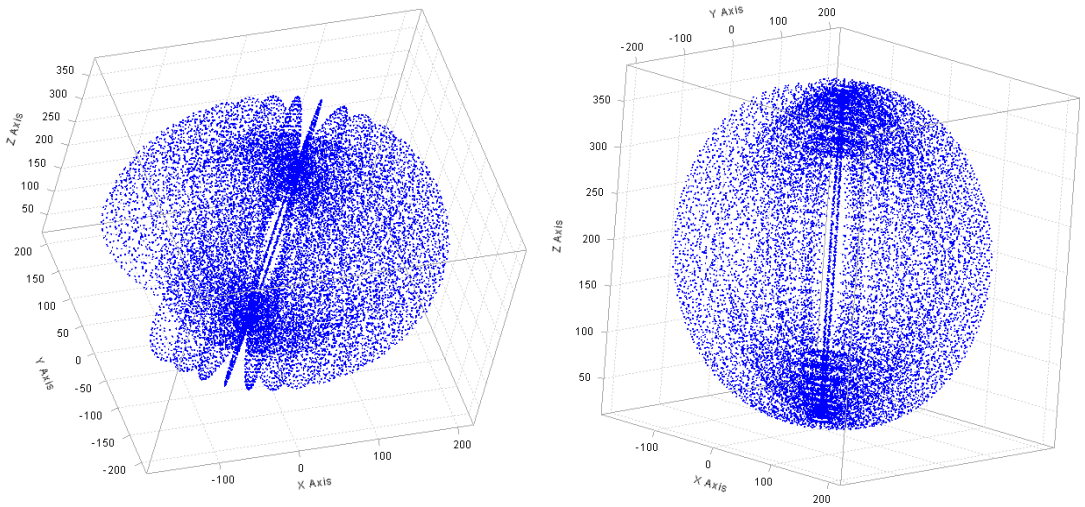
We transformed our position matrix into compatible for ***ILNumerics*** expansion. We created a new scene for ILN framework. We created a cubic plot area for getting 3D effect. We defined a point group in our cubic plot area and defined their properties. We got their position information from our position matrix, defined their color as ***RGB*** formatted value and set their size. The RGB value selected as ***(0,0,255)***, so it's blue.

We created a variable to show on the Windows form. We set their scene and points as we defined before. We set the scale rate as linear because we want to see the are on ***1:1*** ratio. The scale function has also Logarithmic option.



The program executes with this screen. Its default view is top-view. We are seeing X-Y plane from top in this screenshot. And we can clearly see the singularities of the robotic arm. For those singularities that we cannot see properly, at more inner side of the mechanism. We are going to be able to see them on 2D cross-section task. We can move and rotate the plot through the graphical interface with our mouse.

You can see more example screenshots below:



DRAWING 2D CROSS SECTIONS OF THE WORKSPACE

For this task, we are going to draw cross sectional workspace areas for specific z values. Z values determines the height of the plotted area. We split our workspace z values 10 equal pieces.

Our z values starts from **20**, and ends at **380**. Our z value cluster is going to be like:

$$z_{increment} = \frac{z_{max} - z_{min}}{10} = \frac{380 - 20}{10} = \mathbf{36}$$

a is the piece number. So our z values are going to be (from top to bottom):

$$z(a) = (36\ x\ (10 - a)) + 20$$

```
Bitmap DrawingVariable;
int crosssection_number = 10;

double xval, yval, zval, wanted_zval;

double tolerance = 1;

public CrossSection_Visualizer()
{
    InitializeComponent();

    DrawingVariable = new Bitmap(crosssection_plot.Size.Width, crosssection_plot.Size.Height);

    crosssection_plot.Image = DrawingVariable;
}
```

Firstly, we defined a ***Bitmap*** variable to execute drawing commands on. We set a cross section number that will be fill the a value requirement on our second equation. Then we set our ***Bitmap*** variable as new Bitmap with width and height same as our **pictureBox** element. The tolerance value will be explained later.

```
public void paint_cs()
{
        wanted_zval = (36 *(10-crosssection_number)) + 20;

        for (int joint1 = -150; joint1 < 151; joint1++)
        {
            for (int joint2 = -140; joint2 < 141; joint2++)
            {
                for (int joint3 = -101; joint3 < 101; joint3++)
                {
                    (The drawing algorithm)
                }
            }
        }
}
```

We created a function that paints cross sectional workspace area, created our angle combination loop as before and we defined our required z value variable.

```
xval = Math.Cos(joint1 * Math.PI / 180) * (30 + 100 * Math.Cos(joint2 * Math.PI / 180) + 80 *
Math.Cos((joint2 + joint3) * Math.PI / 180));

yval = Math.Sin(joint1 * Math.PI / 180) * (30 + 100 * Math.Cos(joint2 * Math.PI / 180) + 80 *
Math.Cos((joint2 + joint3) * Math.PI / 180));

zval = 200 + 100 * Math.Sin(joint2 * Math.PI / 180) + 80 * Math.Sin((joint2 + joint3) * Math.PI
/ 180);

if (Math.Abs(wanted_zval - zval) < tolerance)

       {

       Graphics graphics;
       Pen pen_object = new Pen(Color.Black, 5);
       Size pen_size = new Size(1, 1);
       graphics = Graphics.FromImage(DrawingVariable);


       Point coordinate = new Point(Convert.ToInt32(1.5*xval+360), Convert.ToInt32(360-
1.5*yval));

       Rectangle rectangle = new Rectangle(coordinate, pen_size);
       graphics.DrawRectangle(pen_object, rectangle);
       crosssection_plot.Image = DrawingVariable;
       graphics.Dispose();


       }
```

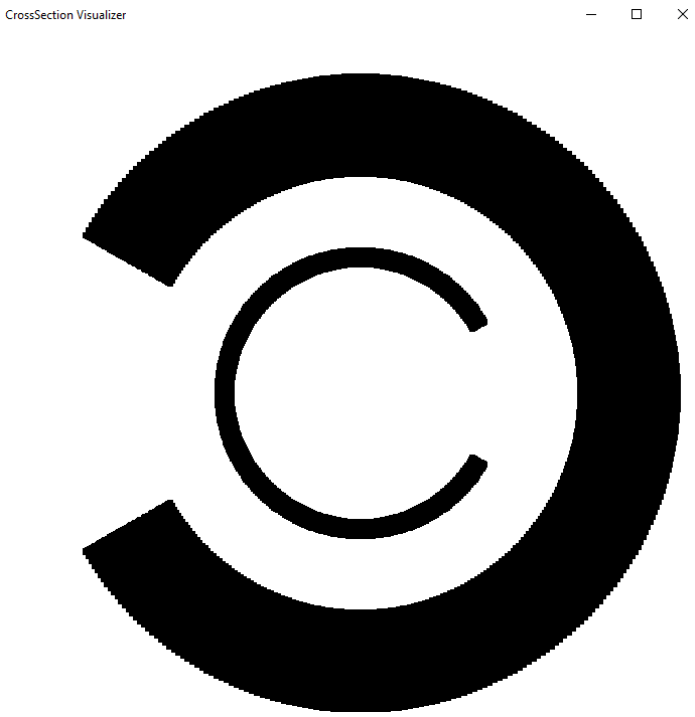
Our drawing algorithm calculates the coordinate as ***x,y,z*** values as before. But we are incrementing the angles by 1 in every loop. We can miss the z value by floating point. So we must add a tolerance value for let the ***(-1,+1)*** z valued ***x,y*** points into the cross section plot.

If our z value difference is lesser than our pre-defined tolerance value (in this code it's **1**), we are going to draw the points to the GUI window.

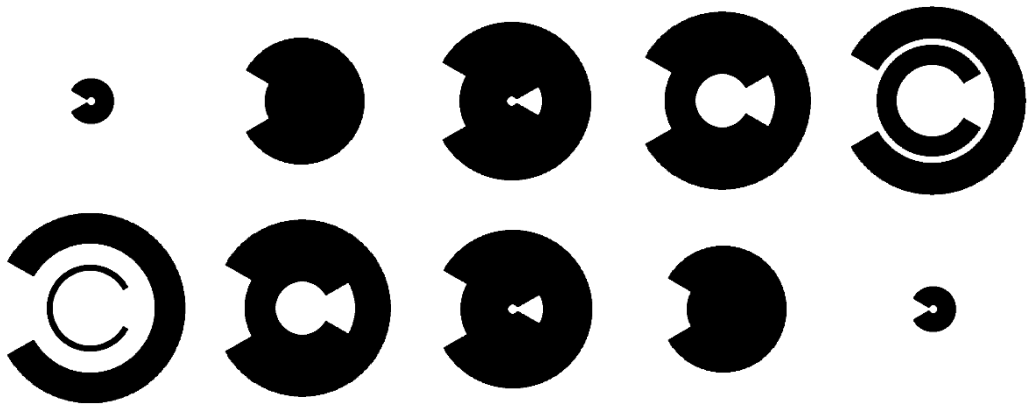
Firstly, we created a ***Graphics*** object that does the graphical jobs on our ***pictureBox*** object for us. Then we created a pen object with specific color and width. Then we defined a size value for a rectangle. Then we set our ***Graphics*** object equal to our ***pictureBox*** image. We set our coordinate value for current for-loop ***x,y,z*** values. We created a rectangle that has pre-defined coordinate values and size. Finally we drawed that rectangle to the ***Graphics*** object. Our ***DrawingVariable*** is changed. We set our ***pictureBox*** Image as ***DrawingVariable*** and disposed our ***Graphics*** object to get rid of memory leaks. Because our for loop will create a ***Graphics*** object every loop.

We gave an ***360px*** offset to our ***x,y*** coordinates, because the ***pictureBox*** object's coordinate system starts from ***top-left***.

The program will start with cross-sectioned area. In this case, our section number is 5. So our z-height is **200**. The black areas represent the area our robotic arm can reach.



The program will give us following cross-sections for ***z-heights*** **20,56,92…380** respectively as below:



The results are going to be symmetrically alike. Because our angle ranges are symmetrical on each joint.

CALCULATING THE VOLUME OF THE WORKSPACE

For this task, we are going to calculate the volume of the workspace. If the robot manipulator could access every point around it, the formula will be volume of the ellipsoid. But there are angle limits, so there are singularity areas in the workspace.

Since we are dealing with a robotic system, we have multiple variables and multiple derivatives. We must create a Jacobian form of the position definition. Let's remember the position matrix:

$$\boldsymbol{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} \cos q_1 \left(a_1 + a_2 \cos q_2 + a_3 \cos(q_2 + q_3)\right) \\ \sin q_1 \left(a_1 + a_2 \cos q_2 + a_3 \cos(q_2 + q_3)\right) \\ d_1 + a_2 \sin q_2 + a_3 \sin(q_2 + q_3) \end{pmatrix} = \boldsymbol{f}(\boldsymbol{q}).$$

We can clearly see we have three variable outputs and three derivative variables. We can form them as a 3x3 Jacobian matrix. The following form is the Jacobian form of our position equation.

$$J = \begin{bmatrix} \frac{\partial p_x}{\partial q_1} & \frac{\partial p_x}{\partial q_2} & \frac{\partial p_x}{\partial q_3} \\ \frac{\partial p_y}{\partial q_1} & \frac{\partial p_y}{\partial q_2} & \frac{\partial p_y}{\partial q_3} \\ \frac{\partial p_z}{\partial q_1} & \frac{\partial p_z}{\partial q_2} & \frac{\partial p_z}{\partial q_3} \end{bmatrix}$$

We need to get difference of volume from this Jacobian matrix. We must get the datas from this matrix as no repeating-variable form, so we can easily differentiate them. If we get the determinant of the Jacobian matrix, we get;

$$\frac{16 * cos(q1)^2 * sin(q2)^3 * sin(q3)^2}{25} + \frac{16 * sin(q1)^2 * sin(q2)^3 * sin(q3)^2}{25} - \frac{6 * cos(q1)^2 * cos(q2)^2 * sin(q3)}{25}$$

$$- \frac{4 * cos(q1)^2 * cos(q2)^3 * sin(q3)}{5} - \frac{6 * cos(q1)^2 * sin(q2)^2 * sin(q3)}{25} - \frac{6 * cos(q2)^2 * sin(q1)^2 * sin(q3)}{25}$$

$$- \frac{4 * cos(q2)^3 * sin(q1)^2 * sin(q3)}{5} - \frac{6 * sin(q1)^2 * sin(q2)^2 * sin(q3)}{25} - \frac{16 * cos(q1)^2 * cos(q2)^3 * cos(q3) * sin(q3)}{25}$$

$$- \frac{4 * cos(q1)^2 * cos(q2) * sin(q2)^2 * sin(q3)}{5} - \frac{16 * cos(q2)^3 * cos(q3) * sin(q1)^2 * sin(q3)}{25}$$

$$- \frac{4 * cos(q2) * sin(q1)^2 * sin(q2)^2 * sin(q3)}{5} + \frac{16 * cos(q1)^2 * cos(q2)^2 * sin(q2) * sin(q3)^2}{25}$$

$$+ \frac{16 * cos(q2)^2 * sin(q1)^2 * sin(q2) * sin(q3)^2}{25} - \frac{16 * cos(q1)^2 * cos(q2) * cos(q3) * sin(q2)^2 * sin(q3)}{25}$$

$$- \frac{16 * cos(q2) * cos(q3) * sin(q1)^2 * sin(q2)^2 * sin(q3)}{25}$$

Since this equation is too long to solve, we must simplify it.

```
J =  [diff(x, q1), diff(x, q2), diff(x, q3);
      diff(y, q1), diff(y, q2), diff(y, q3);
      diff(z, q1), diff(z, q2), diff(z, q3)];

disp(simplify(det(J)));
```

We can simplify the equation using ***MATLAB***. Then the determinant as follows;

$$\boldsymbol{Det(J)} = \frac{16 * sin(q2)}{25} - \frac{6 * sin(q3)}{25} - \frac{4 * cos(q2) * sin(q3)}{5} - \frac{16 * cos(q3)^2 * sin(q2)}{25} - \frac{16 * cos(q2) * cos(q3) * sin(q3)}{25}$$

Our determinant definition has no $q1$ values, so we can form our volume equation without derivating for it. We can define our volume as **integrative summation of our multiplied position difference (x,y,z).** One part of the formula as follows;

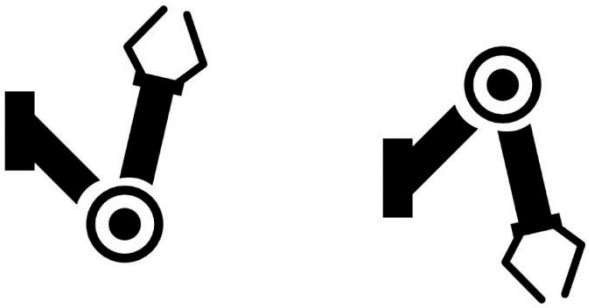
$$\int_{-100}^{100} \int_{-140}^{140} |\det(J)| dq_2 dq_3$$

We must take absolute of J, because if the position change towards to negative way, the differentiation could come negative, but we don't want to lose negative changes, we want to add them to our current volume. The integral limits are degree-type, so we must be careful while calculating trigonometric functions in determinant value.

We don't have a $q1$ in our integral, but we still have the changing of it. So we must add it to the function as a multiplier.

$$z_{change} = 150 - (-150) = 300$$

Also we must divide our result to 2. Because our end-effector can reach the target 2 different way.



If we don't divide the result, we will get twice of our volume.

Finally we can define our volume function as:

$$Volume = 150 \int_{-100}^{100} \int_{-140}^{140} |\det(J)| dq_2 dq_3$$

If we calculate it with ***dblquad()*** function using ***MATLAB***, we will get the volume.

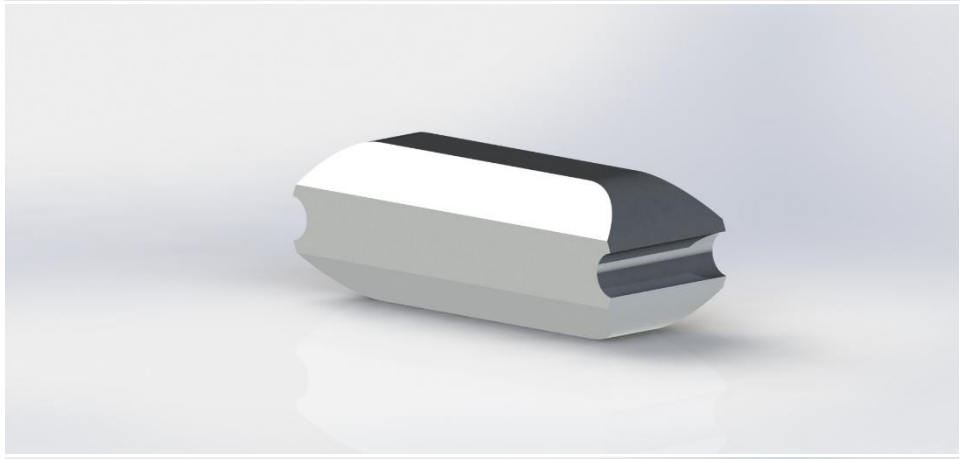
$$\underline{\boldsymbol{Volume} = 23.853\ m^3}$$

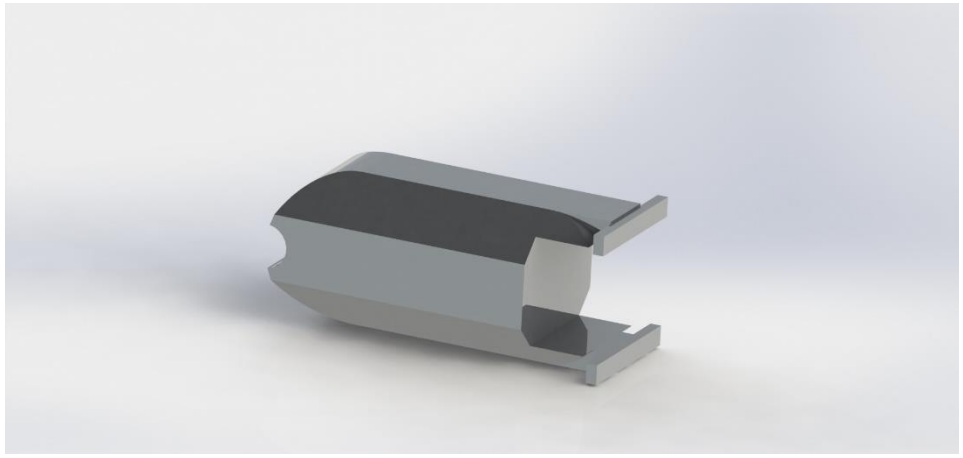
3D SIMULATION OF THE ROBOT MANIPULATOR

For this task, we are going to model the joints of our robotic manipulator. Then we implement them on a ***GUI*** based windows application written on ***C#***, simulate it to go from point A to point B as a **straight line**.

We are going to simulate the system using ***HelixToolkit*** expansion for C#. We are going to create our GUI based on ***WPF Form Application*** type.

We are going to model our robot arm's joints with ***SolidWorks*** first, so we can implement them to our application.





After modelling our joints, we can start coding with defining some of the variables.

```
double joint1, joint2, joint3;
double sensitivity = 10;
bool is_started = false;
double wanted_x;
double wanted_y;
double wanted_z;
double loop_x, loop_y, loop_z;
bool kinematics_end = false;
public static int interval = 10;
bool[] position_check = new bool[interval + 1];
double minimum_distance = 1000;
double[,] angles = new double[interval, 3];
double[,] positions = new double[interval, 3];

Model3DGroup ArmModel = new Model3DGroup();
Model3D arm_link1 = null;
Model3D arm_link2 = null;
Model3D arm_link3 = null;
Model3D positionmodel = null;
Model3D positionmodel2 = null;

ModelVisual3D RoboticArm = new ModelVisual3D();

private const string MODEL1 = "Joint1.stl";
private const string MODEL2 = "Joint2.stl";
private const string MODEL3 = "Joint3.stl";
private const string MODEL4 = "sphere.stl";

RotateTransform3D Rotation = new RotateTransform3D();

TranslateTransform3D Translation = new TranslateTransform3D();
```

We define;

- Variables for our joint angle values
- Stepping sensitivity for robotic arm's animation, angle incrementation sensitivity
- Variables to know if the program has fully loaded, if the kinematics process ended
- Variables for storing target x,y,z values
- Variables for storing temporary x,y,z values coming from the loop
- Arrays for checked position status, angles and positions
- 3D model definitions for joints and animation
- Variables to load model paths
- Variable for minimum distance value that updates every loop
- Delegate variables to carry rotation and translation transforms

```
            RoboticArm.Content = Load3D(MODEL1, MODEL2, MODEL3, MODEL4, MODEL4);
            viewer_3D.Children.Add(RoboticArm);

            DispatcherTimer timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromMilliseconds(100);
            timer.Tick += timer_Tick;
            timer.Start();
```

We defined our robotic arm's joint model and implemented the model group in our 3D animation space. We defined a timer that will provide us the animation part.

```
Transform3DGroup transform_joint1 = new Transform3DGroup();
            Transform3DGroup transform_joint2 = new Transform3DGroup();
            Transform3DGroup transform_joint3 = new Transform3DGroup();
            Transform3DGroup transform_position1 = new Transform3DGroup();
            Transform3DGroup transform_position2 = new Transform3DGroup();

            Translation = new TranslateTransform3D(Convert.ToDouble(coordinate_x.Text)-40, Convert.ToDouble(coordinate_y.Text)-30,
Convert.ToDouble(coordinate_z.Text)-10);

            transform_position1.Children.Add(Translation);

            Translation = new TranslateTransform3D(Convert.ToDouble(coordinate_end_x.Text)-40,
Convert.ToDouble(coordinate_end_y.Text)-30, Convert.ToDouble(coordinate_end_z.Text)-10);

            transform_position2.Children.Add(Translation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(0, 0, 1), 90), new Point3D(0, 0, 0));

            transform_position1.Children.Add(Rotation);
            transform_position2.Children.Add(Rotation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(1, 0, 0), 90), new Point3D(0, 0, 0));

            transform_joint1.Children.Add(Rotation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(0, 0, 1), joint1), new Point3D(17.5, -30, 0));

            transform_joint1.Children.Add(Rotation);

            Translation = new TranslateTransform3D(0, 182.5,0);

            transform_joint2.Children.Add(Translation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(0, 1, 0), 90), new Point3D(0, 0, 0));

            transform_joint2.Children.Add(Rotation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(1, 0, 0), joint2), new Point3D(0, 200, 0));

            transform_joint2.Children.Add(Rotation);
            transform_joint2.Children.Add(transform_joint1);

            Translation = new TranslateTransform3D(100, 0, 0);
            transform_joint3.Children.Add(Translation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(0, 0, 1), 0), new Point3D(0, 0, 0));
            transform_joint3.Children.Add(Rotation);

            Rotation = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(0, 0, 1), joint3), new Point3D(100, 17.5, 0));
            transform_joint3.Children.Add(Rotation);

            transform_joint3.Children.Add(transform_joint2);

            positionmodel.Transform = transform_position1;
            positionmodel2.Transform = transform_position2;
            arm_link1.Transform = transform_joint1;
            arm_link2.Transform = transform_joint2;
             arm_link3.Transform = transform_joint3;
```

We created a function that updates the models. It does all the position arrangements, including translation and rotation transforms with our model group. We defined some integer values for rotating and giving an offset to the models to get exactly same model at the start.

```
kinematics_end = false;

            for (int interval_loop = 0; interval_loop < interval; interval_loop++ )
            {
                positions[interval_loop, 0] = Convert.ToDouble(coordinate_x.Text) +
(Convert.ToDouble(coordinate_end_x.Text) - Convert.ToDouble(coordinate_x.Text)) * interval_loop
/ (interval-1);

                positions[interval_loop, 1] = Convert.ToDouble(coordinate_y.Text) +
(Convert.ToDouble(coordinate_end_y.Text) - Convert.ToDouble(coordinate_y.Text)) * interval_loop
/ (interval - 1);

                positions[interval_loop, 2] = Convert.ToDouble(coordinate_z.Text) +
(Convert.ToDouble(coordinate_end_z.Text) - Convert.ToDouble(coordinate_z.Text)) * interval_loop
/ (interval - 1);

                minimum_distance = 1000;
                wanted_x = positions[interval_loop, 0];
                wanted_y = positions[interval_loop, 1];
                wanted_z = positions[interval_loop, 2];

                            (The array indexing algorithm)

                }
            }

               kinematics_end = true;
```

We defined our function to calculate required angles to get wanted position. If we want to create straight line effect without drawing any splines; we must split the position difference in equal pieces and tell robot to go each of them respectively. This must decrease the spline effect. The more we split, better smoothness we get.

The user will define a split ratio called as interval variable. When the program receives the calculation order, it will split the position difference between start and end points to amount of interval. Then it will calculate the required angles to get these interval positions including starting and ending points.

We transfer that interval positions to our position array with separate x,y,z values as columns. We execute our array indexing algorithm that includes position distance checking and we let the program know the angle calculation process is over.

```
            for (int loop_joint1 = -150; loop_joint1 < 151; loop_joint1++) {
                for (int loop_joint2 =-140; loop_joint2 < 141; loop_joint2++) {
                    for (int loop_joint3 = -100; loop_joint3 < 101; loop_joint3++) {

                        loop_x = Math.Cos(loop_joint1 * Math.PI / 180) * (30 + 100 *
Math.Cos(loop_joint2 * Math.PI / 180) + 80 * Math.Cos((loop_joint2 + loop_joint3) * Math.PI /
180));
                        loop_y = Math.Sin(loop_joint1 * Math.PI / 180) * (30 + 100 *
Math.Cos(loop_joint2 * Math.PI / 180) + 80 * Math.Cos((loop_joint2 + loop_joint3) * Math.PI /
180));
                        loop_z = 200 + 100 * Math.Sin(loop_joint2 * Math.PI / 180) + 80 *
Math.Sin((loop_joint2 + loop_joint3) * Math.PI / 180);

                        if (Math.Sqrt(Math.Pow(wanted_x - loop_x, 2) + Math.Pow(wanted_y -
loop_y, 2) + Math.Pow(wanted_z - loop_z, 2)) < minimum_distance)
                        {
                            minimum_distance = (Math.Sqrt(Math.Pow(wanted_x - loop_x, 2) +
Math.Pow(wanted_y - loop_y, 2) + Math.Pow(wanted_z - loop_z, 2)));

                            angles[interval_loop, 0] = loop_joint1;
                            angles[interval_loop, 1] = loop_joint2;
                            angles[interval_loop, 2] = loop_joint3;

                        }
                    }
                    }
```

We created our array indexing algorithm as before. But we modified the algorithm for getting most precise position this time. The algorithm operates as follows:

- Every joint value returns a x,y,z position
- The algorithm compares returned position with wanted position
- If the difference between these two positions is lesser than ***mindist*** value, the new ***mindist*** value will be set as that and the new angle values will be set
- If the difference between these two positions is equal or more than ***mindist*** value, nothing will change

This way, we will get the angle values that provide us best position value with the least difference from the wanted position.

Applying this to the all interval positions, we will get a straight-line-like trajectory for our robot manipulator.

We completed our computational part. Now we must design our GUI. Since we created our application with ***WPF***, we must design it as ***XML*** form. The following **XAML** codes are enough to define our GUI.

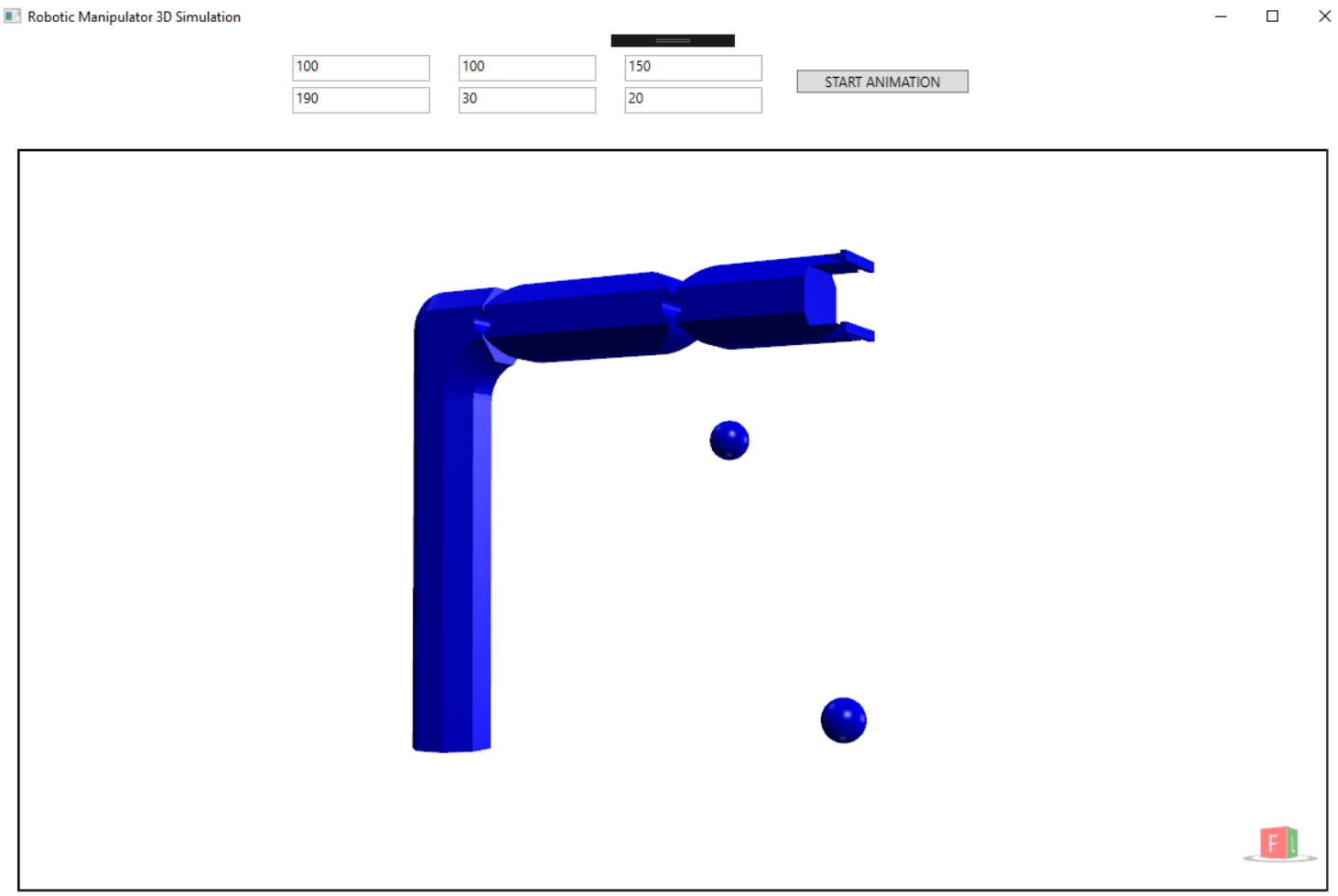
```
<Window x:Class="RobotEngineering_HW1_14067019.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:RobotEngineering_HW1_14067019"
        xmlns:helix="http://helix-toolkit.org/wpf"
        mc:Ignorable="d"

        Title="Robotic Manipulator 3D Simulation" Height="800" Width="1200">
    <Grid Margin="0,0,0.2,-268.2">

        <helix:HelixViewport3D x:Name="viewer_3D" ZoomExtentsWhenLoaded="true" Margin="20,100,20,281.4"
BorderBrush="Black" BorderThickness="2" Background="White">
            <helix:DefaultLights/>
        </helix:HelixViewport3D>

        <TextBox x:Name="coordinate_x" HorizontalAlignment="Left" Height="23" Margin="260,18,0,0"
TextWrapping="Wrap" Text="100" VerticalAlignment="Top" Width="120" TextChanged="coordinate_x_TextChanged"/>

        <TextBox x:Name="coordinate_y" HorizontalAlignment="Left" Height="23" Margin="405,18,0,0"
TextWrapping="Wrap" Text="100" VerticalAlignment="Top" Width="120" TextChanged="coordinate_y_TextChanged" />

        <TextBox x:Name="coordinate_z" HorizontalAlignment="Left" Height="23" Margin="550,18,0,0"
TextWrapping="Wrap" Text="150" VerticalAlignment="Top" Width="120" TextChanged="coordinate_z_TextChanged" />

        <Button x:Name="inversebuton" Content="START ANIMATION" HorizontalAlignment="Left" Margin="700,31,0,0"
VerticalAlignment="Top" Width="150" Click="Calculate_Angles" Height="20"/>

<TextBox x:Name="coordinate_end_x" HorizontalAlignment="Left" Height="23" Margin="260,46,0,0" TextWrapping="Wrap"
Text="190" VerticalAlignment="Top" Width="120" TextChanged="coordinate_end_x_TextChanged"/>

<TextBox x:Name="coordinate_end_y" HorizontalAlignment="Left" Height="23" Margin="405,46,0,0" TextWrapping="Wrap"
Text="30" VerticalAlignment="Top" Width="120" TextChanged="coordinate_end_y_TextChanged" />

<TextBox x:Name="coordinate_end_z" HorizontalAlignment="Left" Height="23" Margin="550,46,0,0" TextWrapping="Wrap"
Text="190" VerticalAlignment="Top" Width="120" TextChanged="coordinate_end_z_TextChanged" />
    </Grid>

</Window>
```

- We created our form windows with width and height properties.
- We created an 3D viewer object that comes from ***Helix Toolkit*** with given properties and position margin values.
- We created a light object to see our model properly and aesthetic.
- We defined the textboxes to store values of starting and ending positions.
- We created a button that triggers the angle calculation function.

Every textbox object has a function for changing of the text situation. This is for seeing the change of spheres that represents starting and ending positions instantly. We must create an if statement, else our program will crash. Because it will try to update model states before creating them.

```
private void coordinate_end_z_TextChanged(object sender, TextChangedEventArgs e)
{
    if (is_started==true)
    update_models();
}
```

The program's GUI is like below:



- First column, first and second rows represent starting and ending x values, respectively.
- Second column, first and second rows represent starting and ending y values, respectively.
- Third column, first and second rows represent starting and ending z values, respectively.
- Start Animation button starts the calculating function. When the calculation is complete, the animation will start between these 2 points.