Introduction

The Lynxmotion L6 (figure 1) is a 4 DOF robotic arm, suitable for experimentation upon equation of forward and inverse kinematics as well as path planning and torque control. This study focuses upon deriving and implementing algorithms that resolve the position and orientation of the effector, produce joint angles out of a given position of the effector in 3D space and plan a “smooth” path to the desired position within a given amount of time.

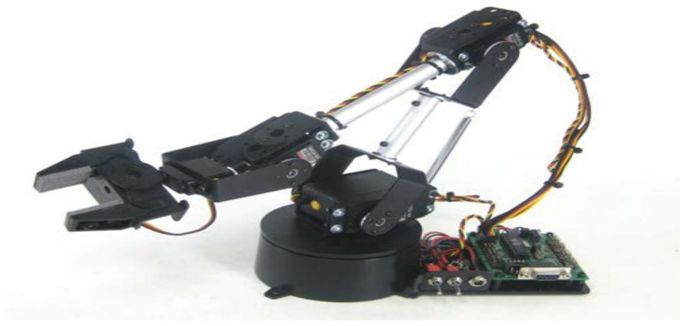


Figure 1. The Lynxmotion L6 4 DOF robotic arm.

Forward Kinematics

Aside the obvious joint motion, the wrist of the robot is also rotating about the axis defined by the previous two joints, thus it should also correspond to a frame. A suitable assignment of a set of frames for the entire arm is shown in figure 2.

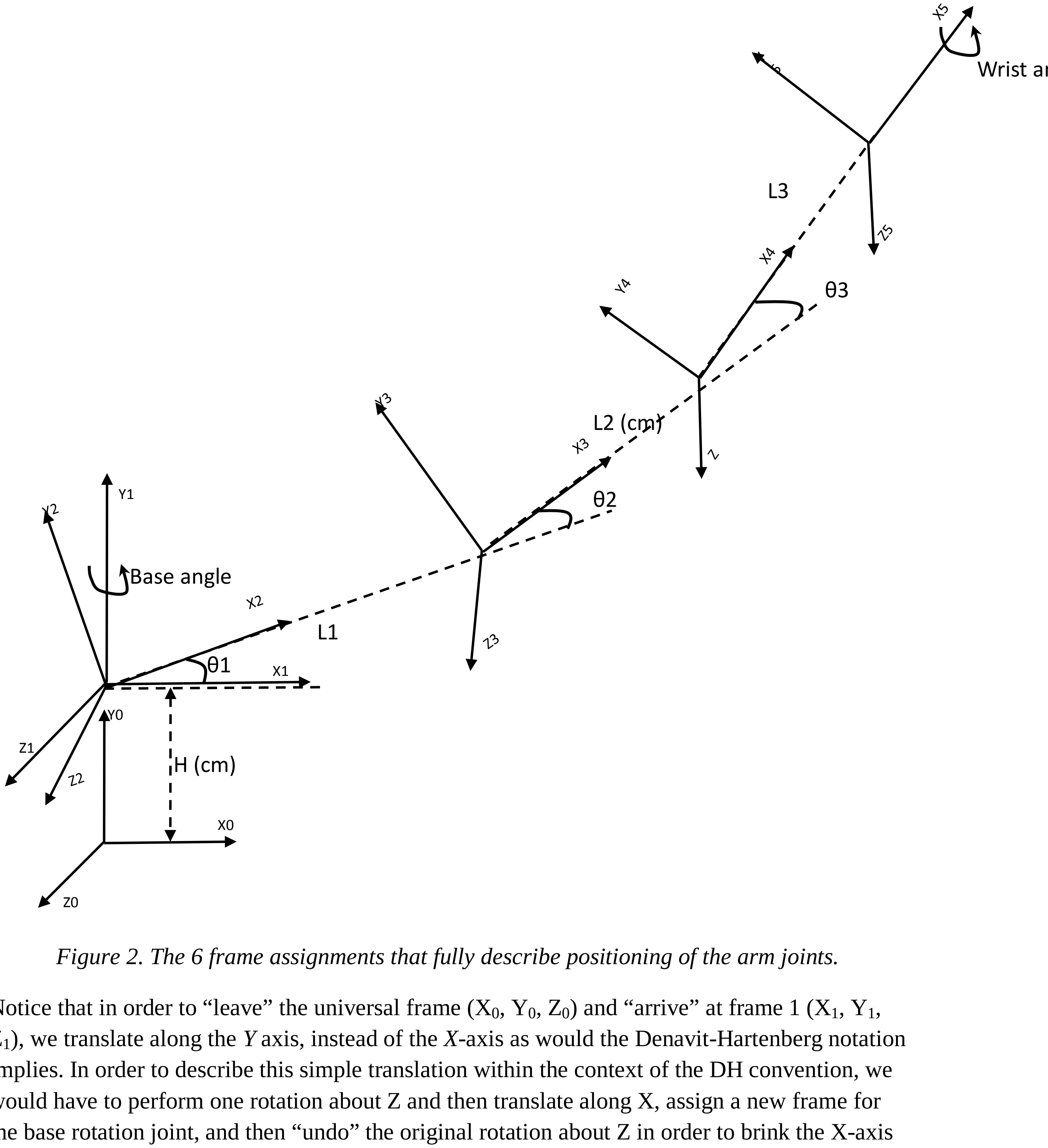


Figure 2. The 6 frame assignments that fully describe positioning of the arm joints.

Notice that in order to "leave" the universal frame (X_0, Y_0, Z_0) and "arrive" at frame 1 (X_1, Y_1, Z_1), we translate along the *Y* axis, instead of the *X*-axis as would the Denavit-Hartenberg notation implies. In order to describe this simple translation within the context of the DH convention, we would have to perform one rotation about Z and then translate along X, assign a new frame for the base rotation joint, and then "undo" the original rotation about Z in order to brink the X-axis back to its original orientation. It's a rather tedious and unnecessary work just to describe a simple translation about Y. However, although frame assignments follow the design of figure 2, table 1 illustrates the DH parameters that fully describe the arm's geometry.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	H	0	90^o
2	0	0	0	Base Angle
3	0	0	0	-90^o
4	0	0	0	θ_1
5	0	L_1	0	θ_2
6	0	L_2	0	θ_3
7	Wrist Angle	L_3	0	0

Table 1. The DH parameters for the lynxmotion arm.

Notice that now we are forced to use 8 frames in total instead of 6. Although table 1 describes the arm geometry in a rather coherent and clear manner for the human, consider that a program doesn't really distinguish a multiplication of a rotation matrix about Y from one about X or Z (nor does it gets "confused" by consecutive translations along different axes). In the end, it's just the same amount of calculations. However, if we use a set of 8 links instead of the minimum (6), the program would have to perform 2 extra matrix multiplications (anyway leading to the same result) just to indulge our own need to simplify notations. Conclusively, the problem is not complex enough to justify the use of 8 links, thereby the number of links used to represent the arm in the program will be 6.

Representing Links in C

A link is implemented as a class named *ArmLink*, which is nothing more than a "loose" implementation of a DH table entry. The term "loose" refers to the fact that the class offers the appropriate methods to rotate and translate not only along X and about Z, X axes, but along and about all X, Y, Z axes. Specifically, class *ArmLink* contains a 4x4 transformation matrix *T*, which is being updated according to actions (rotations or translations) performed on the specific link by simply multiplying with the appropriate matrices from the left. The *ArmLink* class also contains three variables, *anglex*, *angley* and anglez that store angles of rotations along X, Y and Z from the link's initial orientation. The properties of the *ArmLink* class are defined as follows:

```
class ArmLink
    {
        // transformation matrix with respect to previous link
        // (if there is one)
        public double[][] T;
        // current angles about x, y and z
        public double anglex, angley, anglez;
```

The existence of the three angle properties is somehow redundant, since the transformation matrix of the link contains all the information needed to acquire the position and orientation of a joint in space. However, they are used to avoid extra calculations when positioning the servos to the corresponding angle of the joint; they are also used to display the current angle of each joint on the application window.

Matrix Multiplication

Class *ArmLink* contains a static method (used by several other classes in the program) that performs matrix multiplication, function *mulMatrices()*. The function is implemented as follows:

```
public static double[][] mulMatrices(double[][] A, double[][] B, int rowsA,
int colsA, int colsB, ref int rowsC, ref int colsC)
        {
            int i, j;
            int k;
            double linecolprod;
            rowsC = rowsA;
            colsC = colsB;
            double[][] C = new double[rowsA][];
            for (i = 0; i < rowsC; i++)
                C[i] = new double[colsC];

            for (i = 0; i < rowsC; i++)
                for (j = 0; j < colsC; j++)
                {
                    linecolprod = 0;
                    for (k = 0; k < colsA; k++)
                        linecolprod += A[i][k] * B[k][j];
                    C[i][j] = linecolprod;
                }
            return C;
        }
```

The function returns a new array containing the resulting matrix.

Rotations and Translations

Class *ArmLink* contains static methods for performing translations and rotations along and about X,Y, Z. These functions simply receive the appropriate parameters (either an angle or a set of 3 translations) and create a new link with a new transformation matrix, produced by the multiplication of the rotation/translation *R* with the original link transfer matrix:

$$T_{New\ Link} = R \cdot T_{Old\ Link}$$

The idea behind this, is that every time a link rotates, the original object gets “disposed” of and eventually replaced by the one that the rotation/translation function returns. This is a nice approach, since, instead of having to update every detail of the old object, we just copy it and update its transfer matrix, leaving the old one to be “relieved” by the garbage collector of C#.

Static method *translateXYZ()* implements translation along the X, Y, Z as follows:

```
public static ArmLink translateXYZ(ArmLink link, double Dx, double Dy,
                                   double Dz) {
 int i, rows=0, cols=0;
 double[][] Tmatrix;
 Tmatrix = new double[4][];
 for (i = 0; i < 4; i++)
     Tmatrix[i] = new double[4];
 // assigning values to the translation matrix
 Tmatrix[0][0] = 1;
 Tmatrix[0][1] = Tmatrix[0][2] = 0;
 Tmatrix[0][2] = 0;
 Tmatrix[0][3] = Dx;
 Tmatrix[1][0] = 0;
 Tmatrix[1][1] = 1;
 Tmatrix[1][2] = 0;
 Tmatrix[1][3] = Dy;
 Tmatrix[2][0] = 0;
 Tmatrix[2][1] = 0;
 Tmatrix[2][2] = 1;
 Tmatrix[2][3] = Dz;
 Tmatrix[3][0] = Tmatrix[3][1] = Tmatrix[3][2] = 0;
 Tmatrix[3][3] = 1;
 // Translation matrix filled
 ArmLink newlink = new ArmLink();
 newlink.T = Tmatrix;

 return newlink;
}
```

Note that the translation function is used only once to place the links in 3D space. Since all links are revolute, the translation operation is called only once during the creation of the arm.

In quite a similar manner, methods *rotateX(), rotateY()* and *rotateZ()* perform rotations about X, Y and Z correspondingly on an *ArmLink* object. For simplicity, only the implementation of *rotateZ()* is shown below (see Appendices for more code):

```
public static ArmLink rotateZ(ArmLink link, double angle) {
   int i, rows=0, cols=0;
   double[][] Rz = new double[4][];

   for (i = 0; i < 4; i++)
        Rz[i] = new double[4];
   // assigning values to the rotation matrix
   Rz[0][0] = Math.Cos(angle);
   Rz[0][1] = -Math.Sin(angle);
   Rz[0][2] = Rz[0][3] = 0;
   Rz[1][0] = Math.Sin(angle);
   Rz[1][1] = Math.Cos(angle);
   Rz[1][2] = Rz[1][3] = 0;
   Rz[2][0] = Rz[2][1] = 0;
   Rz[2][2] = 1;
   Rz[2][3] = 0;
   Rz[3][0] = Rz[3][1] = Rz[3][2] = 0;
   Rz[3][3] = 1;
   // Rotation matrix about Z filled
   double totalangle = (link.anglez + angle >= 2 * Math.PI) ?
            link.anglez + angle - 2 * Math.PI : link.anglez + angle;
   ArmLink newlink = new ArmLink(link.anglex, link.angley, totalangle);
   newlink.T = mulMatrices(link.T,Rz, 4, 4, 4, ref rows, ref cols);

   return newlink;
 }
```

Assembling the Arm

We may now think or implement a robotic arm practically as an array of links, each one having a transformation matrix expressed in terms of the frame of the previous link in the array. Class *RobotArm* is essentially composed of an ordered set of links. In fact, the class contains a static method that creates an *ArmLink* object based on the design of the *Lynxmotion L6* robot as illustrated in figure 3.Specifically:

1. Link #1 is a frame translated along Y by 6.7 cm from the universal axis (Link #0).
2. Link #2 is a frame coinciding with Link #1.
3. Link #3 is a frame translated along X by 15 cm from Link #2.
4. Link #4 is a frame translated along X by 17.5 cm from Link #3.
5. Link #5 is a frame translated along X by 3 cm from Link #4.
6. Link #6 is a frame translated along X by 4 cm from Link #5.

Link #1 rotates about Y, whereas Links #2, #3, #4 about Z. Link #5 rotates about X. Although not necessary, link 6 was added simply to follow the physical parameters of the L6 arm (the rotation about X due to a servo exactly 4 cm away from the grip). Thus, Link #6 is actually the frame that corresponds to the effector.

Static method *lynxmotionL6()* creates a new *ArmLink* object with the above specifications:

```
public static RobotArm lynxmotionL6() {

     RobotArm arm = new RobotArm(7);
     arm.links[1] = ArmLink.translateXYZ(arm.links[0], 0, 6.7, 0);
     arm.links[2] = ArmLink.translateXYZ(arm.links[1], 0, 0, 0);
     arm.links[3] = ArmLink.translateXYZ(arm.links[2], 15, 0, 0);
     arm.links[4] = ArmLink.translateXYZ(arm.links[3], 17.5, 0, 0);
     arm.links[5] = ArmLink.translateXYZ(arm.links[4], 3, 0, 0);
     arm.links[6] = ArmLink.translateXYZ(arm.links[5], 4, 0, 0);

     return arm;
 }
```

It is rather clear that upon creation, the all links have zero rotation (only translations along the appropriate axes) which justifies the initial arm layout shown in figure 3.

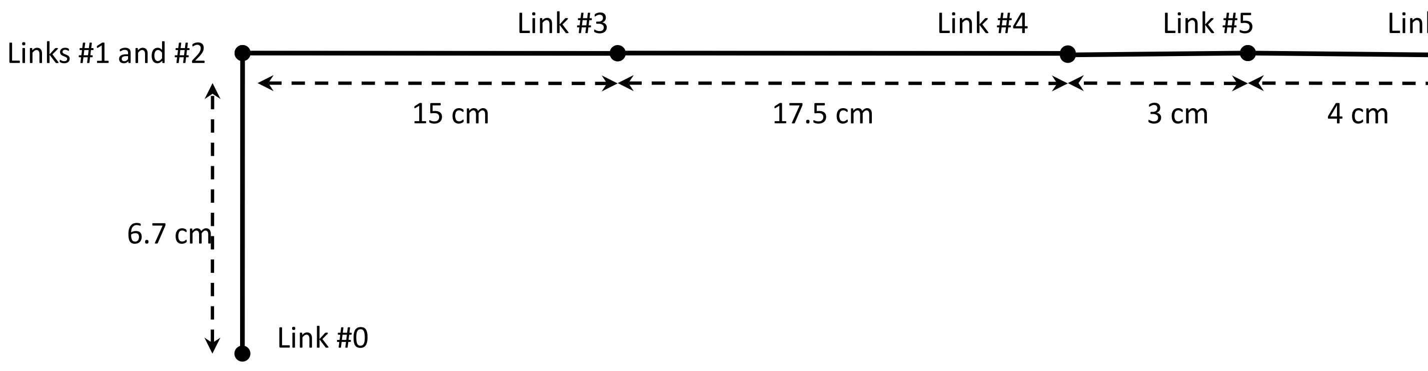


Figure 3. Initial Positioning of the links.

Creating a 3D impression on a 2D canvas

Most equations involved in forward and inverse kinematics where initially tested on a virtual version of the robot. In order to produce a 3D drawing of the arm on a 2D surface, simple equations that project a 3D point onto a 3D plane were used (Solanki Paresh, *A short discussion on mapping 3D objects to a 2D display)*. Specifically, let *(Px,Py,Pz)* be the coordinates of a point in 3D space. Consider that X and Y form a virtual screen through which the human eye observes the given point (figure 4). Given that the eye is located in (Ex, Ey, Ez) in 3D space we may calculate the coordinates of the point on the screen (Sx, Sy) as the cross-section of the beam that connects the human eye with the given point and the virtual screen defined by axes X and Y (figure 5). By using simple trigonometry on similar triangles as shown in figure 5, we may deduct the equations for coordinates *Sx* and *Sy*:

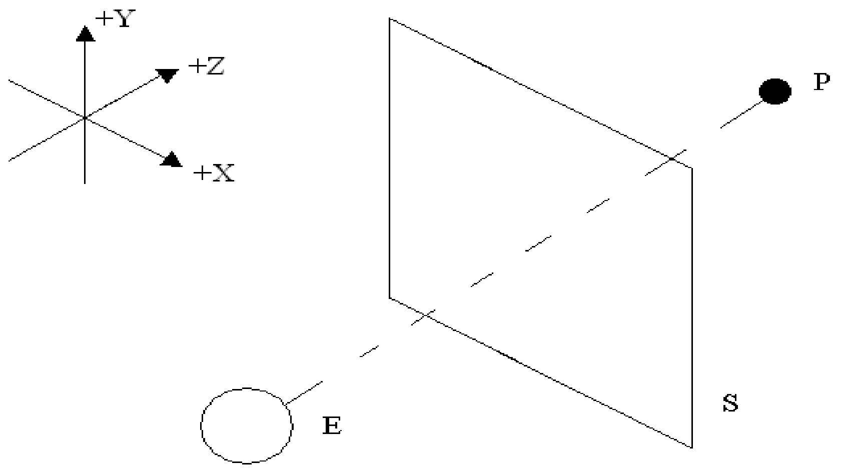


Figure 4. A virtual screen S and the position of the eye (E) looking at a point (P).

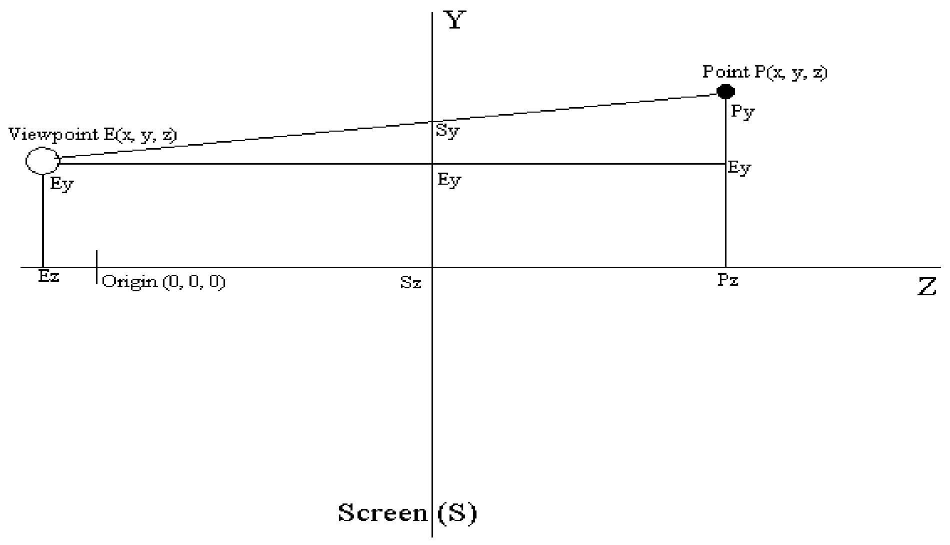


Figure 5. Similar triangles formed by the beam that connects the eye to the point.

Following a few calculations, we get that:

$$S_y = \frac{(P_y - E_y) \cdot (S_z - E_z)}{P_z - E_z} + E_y \quad (1)$$

$$S_x = \frac{(P_x - E_x) \cdot (S_z - E_z)}{P_z - E_z} + E_x \quad (2)$$

Practically, to avoid extra calculations, we assume a 10 cm distance of the eye from the screen and a 5cm offset on the x axis. The drawing canvas is scaled at 25 pixels/cm to provide a relatively realistic depiction of the entire arm on the panel.

Drawing Cylinders

The virtual arm (figure 6) is composed of 4 cylinders, all translated and rotated appropriately using the corresponding link transformations. The cylinders are originally placed at the beginning of the universal frame and along the X or Y axis. By multiplying their transformation matrices, they interestingly fall into place and eventually, form the 3D virtual model of the arm.

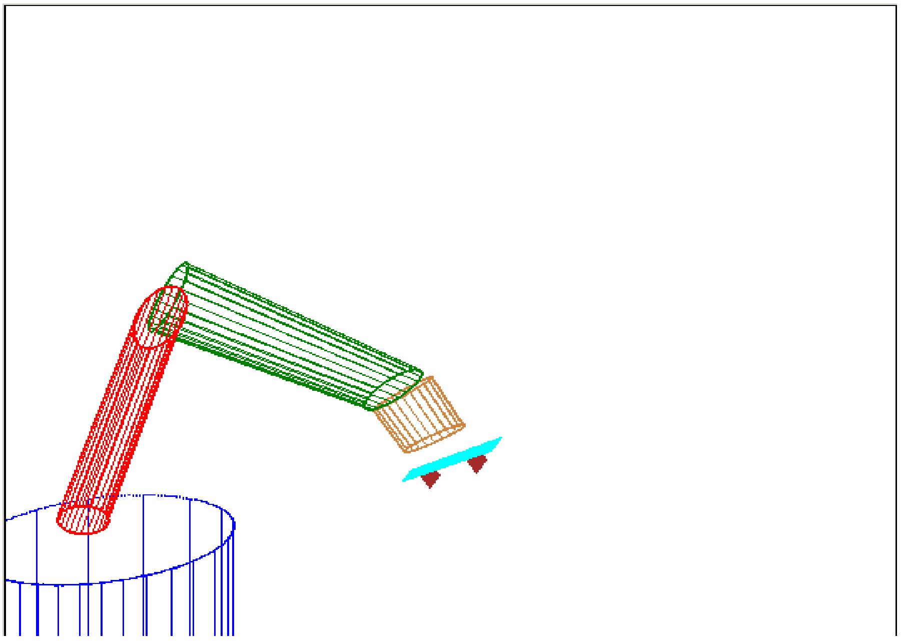


Figure 6. A 3D drawing of the arm using 4 cylinders according to link transformations.

The translation and rotation of the cylinders according to the corresponding transformation matrices of the arm's links in order to produce the arm's 3D model, turned out to be a good test for the validity of the forward kinematics equations

Solving Forward Kinematics

Finding Coordinates of the Effector

In order to derive the coordinates of a given link, let i, we simply multiply the transformation matrices of the links in backwards order, starting from link #*i*, all the way to link #1. The result is the multiplied from the left to a $(0, 0, 0, 1)^T$ vector.

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = T_1 \cdot T_2 \cdots T_i \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad (3)$$

Method *linkPosition()* in class *RobotArm* computes the vector $(X, Y, Z, 1)^T$:

```
public double[][] linkPosition(int linkindex) {
 int i, rows=0, cols=0;
 double[][] zeropoint;
 zeropoint = new double[4][];
 for (i=0; i<4; i++)
   zeropoint[i] = new double[1];
 zeropoint[0][0] = 0;
 zeropoint[1][0] = 0;
 zeropoint[2][0] = 0;
 zeropoint[3][0] = 1;
 zeropoint = ArmLink.mulMatrices(links[linkindex].T, zeropoint,
                                  4, 4, 1, ref rows, ref cols);
 for (i = linkindex - 1; i > 0;  i--)
          zeropoint = ArmLink.mulMatrices(links[i].T, zeropoint,
                                           4, 4, 1, ref rows, ref cols);

 return zeropoint;
}
```

Calculating the Euler Angles (yaw, pitch, roll)

In quite a similar fashion, we compute the product R=$T_1T_2\ldots T_i$ and apply the known formulas (Craig, *Introduction to Robotics Mechanics and Control)*:

$$A_Y = atan2\left(\sqrt{{r_{31}}^2 + {r_{32}}^2}, r_{33}\right) \quad (4)$$

$$A_Z = atan2\left(\frac{r_{23}}{\sin A_Y}, \frac{r_{13}}{\sin A_Y}\right) \quad (5)$$

$$A_X = atan2\left(\frac{r_{32}}{\sin A_Y}, \frac{-r_{31}}{\sin A_Y}\right) \quad (6)$$

where, $R=[r_{ij}]$, i, j=1, 2, 3, 4. Function *linkAngles()* in class *RobotArm* computes the 3 angles found with equations (4), (5) and (6):

```
public EulerParams linkAngles(int linkindex) {
       int i, j, rowsc = 0, colsc = 0;
       double[] angles = new double[3];

       double[][] R = new double[4][];
       for (i = 0; i < 4; i++)
       {
              R[i] = new double[4];
              for (j = 0; j < 4; j++)
              R[i][j] = links[linkindex].T[i][j];
       }

       for (i = linkindex - 1; i > 0; i--)
              R = ArmLink.mulMatrices(links[i].T, R, 4, 4, 4, ref rowsc, ref colsc);

      EulerParams p = new EulerParams();
      p.Yaw = Math.Atan2(-R[2][0], Math.Sqrt(Math.Pow(R[0][0], 2) +
                                   Math.Pow(R[1][0],2))); // yaw (about y)
      p.Pitch = Math.Atan2(R[1][0] / Math.Cos(angles[1]),
                            R[0][0] / Math.Cos(angles[1]));    // pitch (about x)
      p.Roll = Math.Atan2(R[2][1] / Math.Cos(angles[1]),
                           R[2][2] / Math.Cos(angles[1]));    // roll (about z)

     return p;
      }
```

Solving Inverse Kinematics

The idea behind solving inverse kinematics is to find a 2D plane on which the majority of the links are fixed and reduce the problem into solving inverse kinematics for a planar manipulator composed of these joints. Thereafter, if the number of joints of the planar manipulator is greater than 2, then, given certain loose criteria and rules, we again fix a number of joints until the problem eventually diminishes into solving inverse kinematics for a manipulator with 2 joints.

Eliminating the base rotation axis

The only joint that rotates about Y is the base of the arm (actually, the wrist also rotates about X, but it does not affect the coordinates of the effector, thus we may disregard this link for the Inverse Kinematics problem). The rest of the joints form a planar manipulator. Given the coordinates of a desired location in 3D space (x, y, z), we may do a little of "reverse engineering" by trying to find the base rotation angle that turned the arm in a way such as the effector went from 0 to z in the Z axis. The angle can be found as shown in figure 7. Simply put, we are trying to find by what angle do we have to rotate about Y in order for the given point to "touch" the X-Y plane.

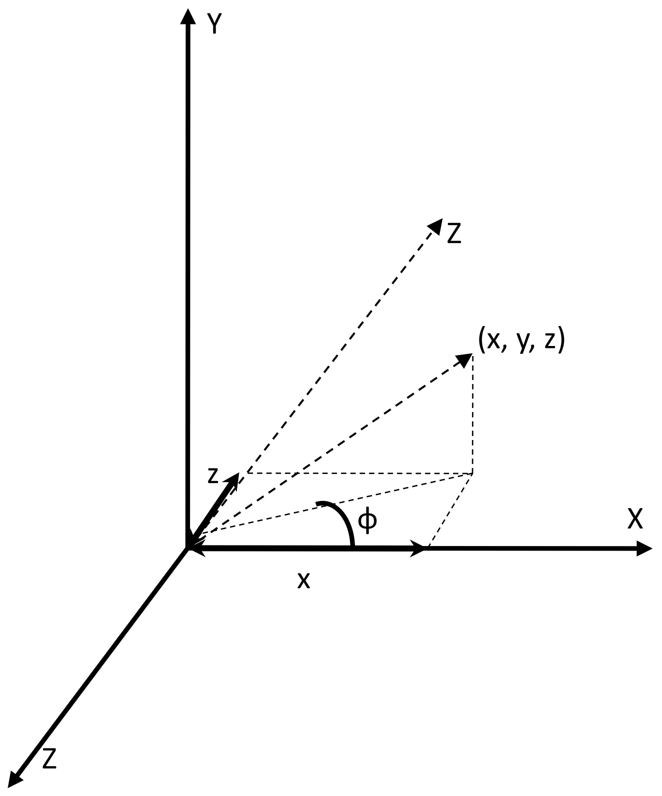


Figure 7. Rotation angle φ (about Y) of a point away from the X-Y plane.

We can easily see that in order to get to (x, y, z) by rotation about the Y axis, the angle that "brings" us there is:

$$\varphi = atan2(z, x) \quad (7)$$

Now the next step is to take rotate the point back to the X-Y plane by a $-\varphi$ angle. Let x_1 and y_1 be the coordinates following a rotation by $-\varphi$ and also take-away the offset of the base in Y (6.7 cm), we get a brand new point of interest:

$$x_1 = x \cdot \cos(-\varphi) + z \cdot \sin(-\varphi) \qquad (8)$$

$$y_1 = y - 6.7 \qquad (9)$$

We have now reduced the problem into solving inverse kinematics for the planar manipulator of figure 8 given a desired point in 2D space *(x_1, y_1)*.

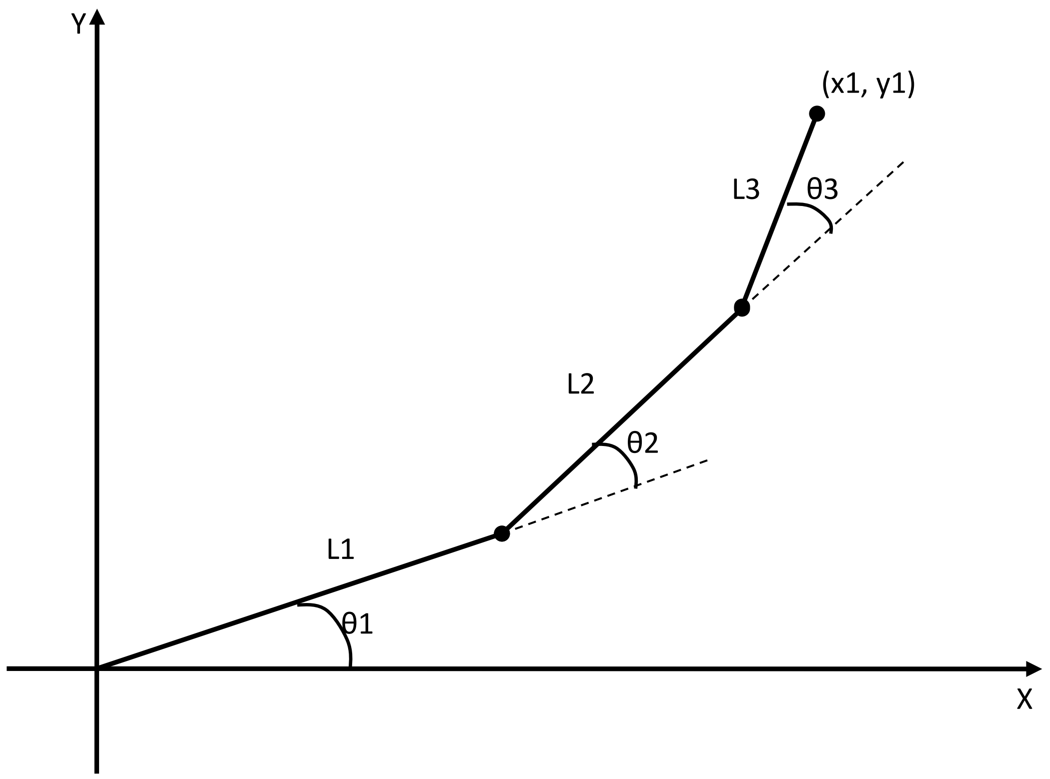


Figure 8. Reduced IK problem for a 3-link planar manipulator.

It is obvious that the problem has infinite solutions within a certain interval for θ_1. We now have to determine a value for θ_1 and further reduce the problem into a 2-link planar manipulator inverse kinematics problem.

Fixing angle θ_1

If that $\theta_3=0$, we may consider the problem of solving inverse kinematics for the manipulator of figure 9, in which segments L_2 and L_3 are connected as single straight line.

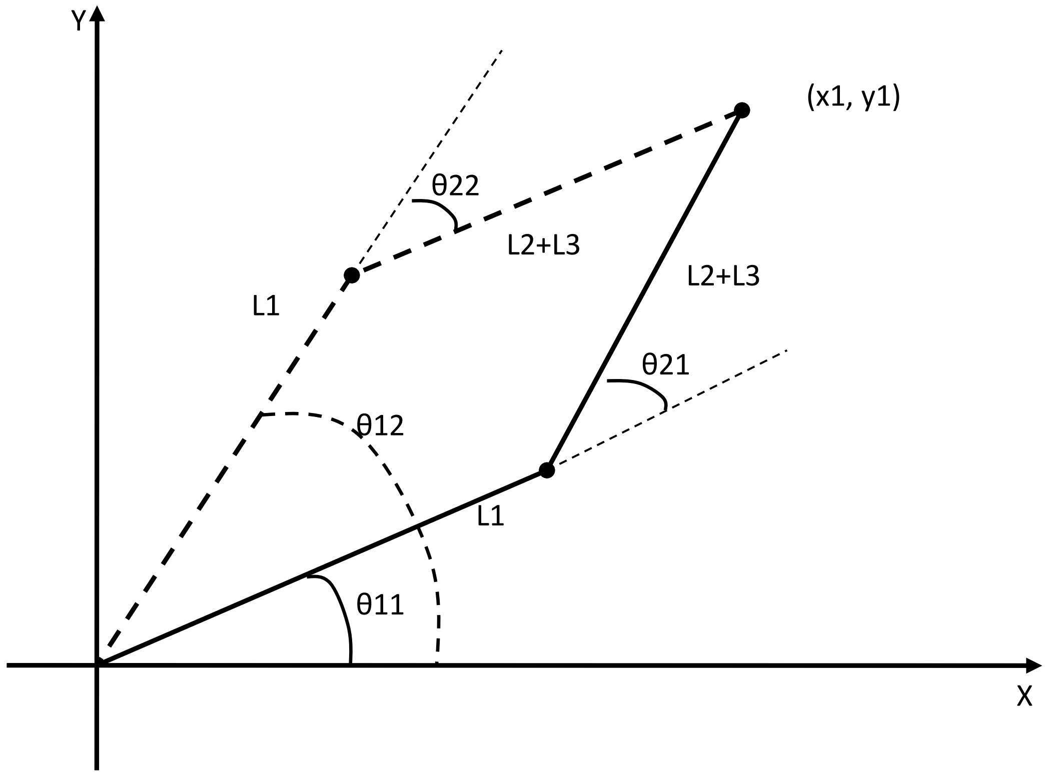


Figure 9. Assuming \a 2-link manipulator by joining L2 and L3 in a straight line.

Solving inverse kinematics for the problem of figure 9, will give us two solutions which in fact, contain the *minimum* and *maximum* values for θ_1 (θ_{11} and θ_{12}).

$$\theta_{2(1,2)} = \arccos\left(\frac{{x_1}^2 + {y_1}^2 - {L_1}^2 - {L_2}^2}{2L_1L_2}\right) \quad (10)$$

$$\theta_{1(1,2)} = arcsin\left(\frac{y_1\left(l_1 + l_2\cos\left(\theta_{1(1,2)}\right)\right) - x_2 l_2 \sin\left(\theta_{2(1,2)}\right)}{{x_1}^2 + {y_1}^2}\right) \quad (11)$$

Equation (11), following a few calculations will give us the minimum and maximum values for θ1. Since now we have an interval, $[\theta_{11}, \theta_{12}]$ in for which we get infinite pairs of solutions, we might be able to choose the closest value of θ_1 in this interval to the current angle.

Once again, after we have chosen the closest value of θ_1 in $[\theta_{11}, \theta_{12}]$ to the current position of the joint, the problem eventually diminishes into solving inverse kinematics for a 2-link planar manipulator (θ_2, θ_3). Inverse kinematics are done with method *solveIK()* in class *RobotArm.* It is a rather extensive function, thereby omitted from the text; however, it is enlisted in the appendices for viewing.

Trajectory Planning

The program drives the manipulator to a destination point by initially solving inverse kinematics and thereby dividing the total motion of each joint by a desired time in which the movement of the manipulator should conclude. Let *T* (in seconds) be the total time of motion to the destination point, Sbase, Sθ_1, Sθ_2, Sθ_3 the initial angles of the joints, Ebase, Eθ_1, Eθ_2, Eθ_3 the angles upon reaching the destination, then each angle advances by a certain amount given by the following:

$$\Delta base = \frac{Ebase - Sbase}{T} \qquad (12)$$

$$\Delta\theta_1 = \frac{E\theta_1 - S\theta_1}{T} \qquad (13)$$

$$\Delta\theta_2 = \frac{E\theta_2 - S\theta_2}{T} \qquad (14)$$

$$\Delta\theta_3 = \frac{E\theta_3 - S\theta_3}{T} \qquad (15)$$

From equations (12), (13), (14), (15) we easily conclude that all joints perform a linear rotation towards destination.

A trajectory is represented as an array of elementary moves inside class *Trajectory*. Each move involves a vector of steps $(\Delta base, \Delta\theta_1, \Delta\theta_2, \Delta\theta_3)^T$ corresponding to each of the joints. A move is performed in a predetermined time interval, defined as a fraction (1/10 is used in the program) of the total motion time (fixed at 10 seconds for every trajectory in the program). A trajectory is created by function static function *planTrajectory()* in class *RobotArm.*

```
public static Trajectory planTrajectory(RobotArm arm, double Ebase,
                  double Eth1,double Eth2, double Eth3, int time) {
 double Sbase = arm.links[1].angley;
 double Sth1 = arm.links[2].anglez;
 double Sth2 = arm.links[3].anglez;
 double Sth3 = arm.links[4].anglez;

 Trajectory t = new Trajectory(Sbase, Sth1, Sth2, Sth3, Ebase, Eth1,
                               Eth2, Eth3, time);
 double baseangle = Sbase;
 double th1 = Sth1;
 double th2 = Sth2;
 double th3 = Sth3;

 while ((Math.Abs(baseangle-Ebase)>0.01)||(Math.Abs(th1-
       Eth1)>0.01)||(Math.Abs(th2-Eth2)>0.01)||(Math.Abs(th3-Eth3)>0.01)) {
          t.len++;
          if (Math.Abs(baseangle - Ebase) > 0.01)
          {
                t.moves[t.len - 1].Dbase = t.stepbase;
```

```
                    baseangle += t.stepbase;
              }
               else t.moves[t.len - 1].Dbase = 0;

         if (Math.Abs(th1 - Eth1) > 0.01)
         {
                  t.moves[t.len - 1].Dth1 = t.stepth1;
                  th1 += t.stepth1;
         }
         else t.moves[t.len - 1].Dth1 = 0;

         if (Math.Abs(th2 - Eth2) > 0.01)
         {
               t.moves[t.len - 1].Dth2 = t.stepth2;
               th2 += t.stepth2;
         }
         else t.moves[t.len - 1].Dth2 = 0;

         if (Math.Abs(th3 - Eth3) > 0.01)
         {
               t.moves[t.len - 1].Dth3 = t.stepth3;
               th3 += t.stepth3;
         }
         else t.moves[t.len - 1].Dth3 = 0;
 }

return t;
}
```

Drawing a sine wave

As a demo trajectory, the arm may draw a sin wave on a plane defined by *X-Y* plane (x=25 cm), using *y=20+3sin(z)* (cm), for *z in [-1, 12]*. The motion is divided in 40 steps, each performed in 1 sec. For ever value of *y*, inverse kinematics is solved and a new move is added to the trajectory list (obviously, the angles don't increase linearly, due to the *sin* in the equation that produces *y*, however the movement is smooth due to the number of elementary steps*)*. Static method *demosinWave()* produces the trajectory:

```
public Trajectory demoSinwave() {
   double baseangle = 0, th1 = 0, th2 = 0, th3 = 0;
   double z, Dz, x, y;
   RobotArm.solveIK(ref baseangle, ref th1, ref th2, ref th3, 25, 20,
                        10,  links[2].anglez,links[3].anglez,
                        links[4].anglez);
   Trajectory t = RobotArm.planTrajectory(this, baseangle, th1, th2,
                                          th3, 10);
   Dz = 13*0.025;
   x = 25;
   for (z = 12; z > -1; z -= Dz) {
      y = 20 + 3 * Math.Sin(z);
      double newbaseangle = 0, newth1 = 0, newth2 = 0, newth3 = 0;
      Boolean solved = RobotArm.solveIK(ref newbaseangle, ref newth1,
                                        ref newth2, ref newth3, x, y,
                                        z, th1, th2, th3);
      // appending trajectory
      t.len++;
      t.moves[t.len - 1].Dbase = newbaseangle - baseangle;
      t.moves[t.len - 1].Dth1 = newth1 - th1;
      t.moves[t.len - 1].Dth2 = newth2 - th2;
      t.moves[t.len - 1].Dth3 = newth3 - th3;

      baseangle = newbaseangle;
      th1 = newth1;
      th2 = newth2;
      th3 = newth3;
   }
 return t;
}
```

Aligning Servos to the Virtual Arm

The lynxmotion arm servos position themselves through serial commands within a range of 300 to 1500 and sometimes exceede the upper or lower limit by 200-300 (practically we may position them at 100 or 1900 but these position may prove impractical in certain occasions). Given the joint that the servo is mounted, the program translates an actual angle in rads, into a position in the range of 300 to 1500. Take for example the base joint. A value of 300 corresponds to 0 degrees and a value of 1500 corresponds to -90 degrees (-π/2). We may then construct a linear relationship between the actual angle of the joint and the position of the servo as:

$$Servo\ Position = -\frac{2400}{\pi} angle + 300 \quad (16)$$

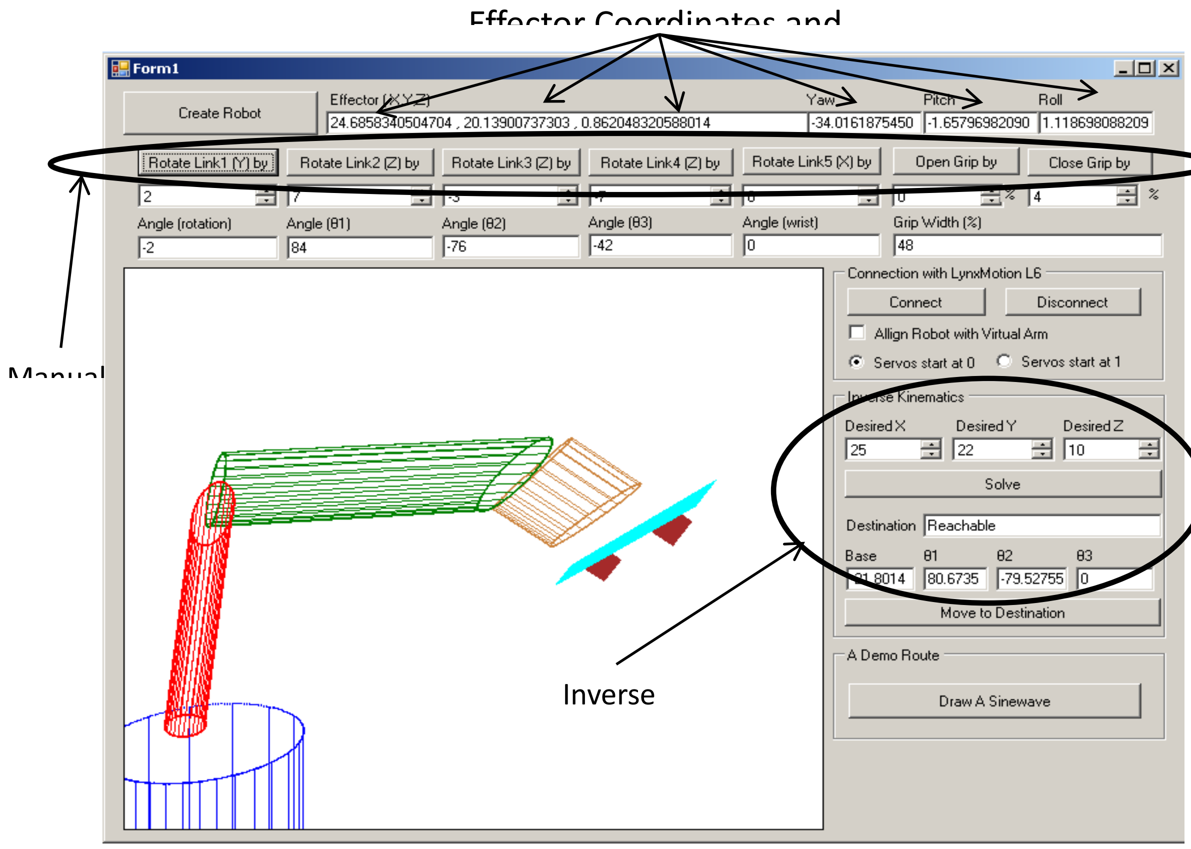
In quite a similar fashion, we assign servo positions for every joint.

Using the program

Prior to using the arm, just hit “Create” to create a new virtual lynxmotion robot (figure 10). The robot should appear in the fashion described in figure 3 (all joint angles set to 0). You may move each joint (and servo) independently by using the buttons and numeric controls located above the virtual robot drawing. If you wish to connect a real lynxmotion robot, just choose the first servo channel (some robots start at channel 0 and others at channel 1) and hit “Connect”. If you wish to align and simultaneously move both the virtual arm and the robot, just check *“Align Robot with Virtual Arm”*. In many cases, I found that the robot becomes a bit “fidgety” or “week” at first, but eventually following a few moves of several joints, it looks nicely aligned.

If you wish to watch the demo trajectory, hit *“Draw a Sinewave”*. If you wish to solve inverse kinematics for a point in space that you have in mind, just type the (x,y,z) coordinates and press “Solve”. If the solution a solution is found, then, you should see a

"Reachable" message in the textbox and the joint angles that take the arm to that position below, otherwise "Unreachable" and "N/A". Since equations have been solved, the destination is stored and you may press "Move to Destination" to see the robot move to the selected point using a trajectory of 10 steps in 10 seconds (1 sec/step).



Effector (X,Y,Z)	Yaw	Pitch	Roll
24.6858340504704 , 20.13900737303 , 0.862048320588014	-34.0161875450	-1.65796982090	1.118698088209

Rotate Link1 (Y) by	Rotate Link2 (Z) by	Rotate Link3 (Z) by	Rotate Link4 (Z) by	Rotate Link5 (X) by	Open Grip by	Close Grip by
2	7	-3	-7	0	0 %	4 %

Angle (rotation)	Angle (θ1)	Angle (θ2)	Angle (θ3)	Angle (wrist)	Grip Width (%)
-2	84	-76	-42	0	48

☐ Allign Robot with Virtual Arm

[✓ Servos start at 0] Servos start at 1

Desired X	Desired Y	Desired Z
25	22	10

Destination: Reachable

Base	θ1	θ2	θ3
21.8014	80.6735	-79.52755	0

Figure 10. A program performing forward/inverse kinematics and trajectory planning.

Conclusions-Improvements

With a little "polishing" of the classes in the program, we may optimize code to resolve kinematics for every manipulator given a set of DH parameters. Specifically, it may be possible to construct a recursive algorithm that resolves inverse kinematics following a "Rubik's cube" approach to reduce the problem into solving equations for a planar manipulator in conjunction with a set of predicates to fix the free solutions that may occur. In addition, we may use recursion to produce a valid sequence of positions in 3D against constraints (such as solid objects within the workspace).