



1997

Mapping Robotic Movement to a Three-Dimensional Coordinate System

Craig A. Materick '97
Illinois Wesleyan University

Follow this and additional works at: https://digitalcommons.iwu.edu/cs_honproj

Recommended Citation

Materick '97, Craig A., "Mapping Robotic Movement to a Three-Dimensional Coordinate System" (1997). *Honors Projects*. 12.

https://digitalcommons.iwu.edu/cs_honproj/12

This Article is protected by copyright and/or related rights. It has been brought to you by Digital Commons @ IWU with permission from the rights-holder(s). You are free to use this material in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This material has been accepted for inclusion by faculty at Illinois Wesleyan University. For more information, please contact digitalcommons@iwu.edu.

©Copyright is owned by the author of this document.

**Mapping Robotic Movement to a
Three-Dimensional Coordinate System**

Craig A. Materick and Dr. Lon Shapiro*
Illinois Wesleyan University
Department of Computer Science

Table of Contents

Abstract.....	1
Historical Background.....	3
The Industrial Revolution.....	4
Robotic Tasks.....	5
Welding.....	5
Material Handling.....	6
Machine Loading and Unloading.....	6
Spray Finishing.....	6
Machining.....	6
Assembly.....	6
Inspection.....	7
Remote Manipulation.....	7
Joint and Arm Designs.....	8
Robot Arm Kinematics.....	9
The Direct Kinematics Problem.....	10
The Inverse Kinematics Problem.....	11
Choosing The Method.....	12
Breakdown of Shelley's Arm Movement.....	13

Mathematical Model of Shelley's Arm Control.....	15
Problems and Solutions.....	20
Joint Over-rotation.....	20
Symmetric Rotation.....	22
k Coordinate of End-effector Equal To Zero.....	22
Servo 2 Rotation Calculations.....	22
Claw Offset.....	23
Future Projects.....	24
Full Three-Dimensional Implementation.....	24
Vision Processing.....	24
Kinematic Calibration.....	25
Trajectory Planning.....	25
Touch Sensitivity.....	25
Conclusion.....	26
References.....	27
Figure A (Robotic Arm Joints).....	29
Figure B (Robotic Arms).....	30
Figure C (Shelley General Arm Layout).....	31
Figure D (Kinematics Block Diagram).....	32
Figure E (Two-Dimensional Top View).....	33

Figure F (Minimum Arm Reach).....	34
Figure G (Minimum Intersection y Value).....	35
Program Appendix.....	after text page 36
- includes Sample Input and Output	

Abstract

Title: Mapping Robotic Movement to a Three-Dimensional Coordinate System

The Illinois Wesleyan Intelligence Network on Knowledge (I.W.I.N.K.) is a project to design and implement an artificial "person" named Shelley. Robotics, networking, and artificial intelligence will be the main topics of the preliminary work. For my research honors project, I designed the three-dimensional coordinate system in which the robotic arms move and interact with objects. The arms we have constructed are based on an arrangement of six servos, each of which rotate approximately 185 degrees. The program takes in data about the location of an object in three-dimensional coordinates and moves each of the six motors in the arm to arrive at that point. The mathematics involved is based on intersecting circles using the following equation:

$$(x-h)^2 + (y-k)^2 = r^2$$

(Assuming the center of the circle is (h,k) and the radius is r)

Included in this work is a look at robotic arm developments through history, from Leonardo da Vinci through the Industrial Revolution, and beyond. Also discussed are the various joint and arm designs developed during these years of research and some robotics projects which employ these different designs. Next, we will investigate the various methods of control developed by other robotic arm research projects and apply one particular method to control Shelley, as briefly outlined above. Finally, we will highlight problems we faced

during the implementation of this program, the solutions to these problems, and various ideas about future research possibilities.

Historical Background

In 1818, Mary Shelley penned the novel, *Frankenstein*, and the concept of recreating humans through technology was popularized. In fact, Mary Shelley subtitled the book *The Modern Prometheus* after the Titan in Greek mythology who fashioned the first human beings out of clay.[2] Currently, Mary Shelley's name is once again associated with the recreation of humans through technological means. Shelley is the name of the robot Illinois Wesleyan University students and faculty are undertaking to create.

For centuries, humans have been interested in recreating themselves through technology. Some of the earliest signs can be seen in cave paintings and the sculptures of ancient Greece and Rome. Robotics can be seen as a logical extension of sculpting and other art forms. The artisans used whatever technology was available to them. Depending on the definition one considers, something as ordinary as a washing machine could be considered a robot. Webster's dictionary defines robot as "an automatic device that performs functions ordinarily ascribed to human beings." [3] However, most studies in the history of robotics focus on projects that fit the more precise definition used by the Robot Institute of America: "A robot is a reprogrammable multi-functional manipulator designed to move materials, parts, tools, or specialized devices, through variable programmed motions for the performance of a variety of tasks." [3] The first real signs of development in robotics can be seen in the work of Leonardo da Vinci. Although he is notorious for his incomplete works [5], he made detailed studies into the anatomy of human arms and applied

this knowledge to several projects. Since his main area of interest was flight, most of his sketches and models used this research to produce articulated wing structures designed to allow man to fly. These devices do not exactly fit the second strict robot definition given above, but they are commonly considered the first true beginnings of modern robotics. The technology expanded in small steps for many years until the Industrial Revolution.

The Industrial Revolution

The Industrial Revolution provided both the need for and the technology to produce robotics. These robots were classified as industrial robots which are defined as “a general purpose, computer-controlled manipulator consisting of several rigid links connected by revolute or prismatic joints. One end of the chain is attached to a supporting base, while the other end is free and equipped with a tool to manipulate objects or perform assembly tasks.”[3] This definition is somewhat restrictive because it implies that true industrial robots must have some level of intelligence. To slightly broaden this definition, human-controlled robots are often included. Industrial robots are normally used to perform relatively simple, repetitive tasks which have been programmed in by a human user. Early work can be traced to the period immediately following World War II.[3] During the late 1940s, research programs were started at the Oak Ridge and Argonne National Laboratories to develop remotely controlled mechanical manipulators for handling radioactive materials. These systems were master-slave configurations. In other words, they were designed to reproduce hand and arm motions made by a human operator in a

remote location. The master manipulator was guided by the user through a sequence of movements, while the slave manipulator duplicated the master as closely as possible. New, more sophisticated robotics were developed by men like Joseph Engelberger and George Devol known as the “Father” and “Grandfather of industrial robotics” respectively.[2] Devol is credited with receiving the first industrial robot patent in 1950 and for designing a playback system for teaching machine tools to remember their motions.[2] When applied to robots, this technology allowed users to simply program robot motion instead of directly controlling it. Engelberger is known for combining Devol’s ideas from above with his own business savvy to lead Unimation Inc., the first industrial robotics company, to the top of its field.[2]

Robotic Tasks

During the Industrial Revolution, robots were developed to perform many tasks including welding, material handling, machine loading and unloading, spray finishing, machining, assembly, inspection, and remote manipulation.[2]

Welding

Welding comes in two forms, spot and arc, and is the largest industrial robotic application. Accuracy, heat resistance, and speed are the keys to the success of welding robots. Current models like the German-made KUKA and others are used for welding in automobile plants around the world. They provide higher quality cars produced faster than any human assembly line could. Specialized units like the KUKA 200[2] are designed to

spot-weld extremely curved surfaces, and this ability lends to an ever increasing degree of flexibility in tasks.

Material Handling

Material handling is the second largest industrial robot application. A common material-handling job is the grouping and/or removing of parts as they come down a conveyor.

Machine Loading and Unloading

Similar to the material handling application is machine loading and unloading in which robots pick up and transfer parts to and from machines. These robots are mainly used in foundries to remove parts from casts.

Spray Finishing

Spray finishing includes the application of paints and other decorative and protective coatings. The benefits of using robotics for this task are even coverage, speed, and lower costs due to reduced fresh air requirements.

Machining

Machining includes cutting, grinding, polishing, drilling, sanding, buffing, deflashing, and deburring. This application requires extremely accurate, often complex robotics, but the benefits are increased speed, uniformity, and output accuracy.

Assembly

The assembly application includes means of combining parts other than welding

including fitting together parts, and holding them together with nuts, bolts, screws, and bonding. Eighty-five percent of all manual labor expended in U.S. industry can be classified as assembly.[2] Many of these tedious and repetitive positions can and are being “manned” by robots. For this project, we will not go into the various ethical questions that arise over replacing humans with robots.

The first industrial robot developed especially for assembly was the PUMA (programmable universal machine for assembly) by Unimation, Inc.[2] Various models in the PUMA line are still used for tasks ranging from small-appliance assembly, electrical component insertion, and wire harness wrapping. One of the most popular robotic assembly tasks is the manufacture of printed circuit boards for computers and many other electrical components. A robot can produce such detailed products far more rapidly and efficiently than a human could.

Inspection

Inspection robots use either tactile or visual data to determine the quality of a product. They are often associated with assembly robotics to ensure that the work is being done to the specified tolerances.

Remote Manipulation

Finally, remote manipulators are another popular application of industrial robots. Often, they utilize the master/slave convention as previously described. Examples can be seen in underwater exploration and the space program. The benefits of such robots are

extraordinary. If manufactured correctly, they can withstand extreme and dangerous conditions, and allow humans to “go” places we would not normally have access to like distant planets and the depths of the ocean. Although Shelley is an educational robot, and not an industrial robot, it is important to look at industrial robot development. Without the Industrial Revolution and the advances in industrial robotics, we would not be able to assemble a multi-functional, dextrous manipulator like Shelley.

Joint and Arm Designs

From all of the robotics research over the years, many joint and arm designs have been developed. Each joint design was created to fulfill a specific task. Figure A contains images of the various joints. They can be split into two basic categories. The sliding group includes the cylindrical, prismatic, and planar joints which all involve one surface sliding on or through another. The other joints allow only for rotation. The screw joint is a hybrid of the two categories. It allows controlled sliding. In other words, the central portion slides through the outer ring, but the sliding is controlled by the rotation of the central cylinder. In comparison, the prismatic inner cylinder simply slides without rotation, and the inner cylinder of the cylindrical joint slides with or without rotation. Various combinations of these joints can be used to produce different types of arms. Four different variations can be found in Figure B. These four motion-defined categories are: cartesian coordinate, cylindrical coordinate, spherical coordinate, and revolute or articulated coordinate. The defining factor for the categories is the number and type of axes.

Cartesian arms have three linear axes. IBM's RS-1 model is an example of this design. The Versatran 600 robot from Prad is a cylindrical coordinate arm and has one rotary and two linear axes. Spherical coordinate arms have one linear and two rotary axes while revolute coordinate arms have three rotational axes. The 2000B and the PUMA from Unimation Inc.[3] are examples of these two models respectively. Shelley is composed of revolute joints. From Figure A, one can see that the basic design of the revolute joint allows only rotation in two directions, say clockwise and counter-clockwise, around a fixed point. This rotation value, in degrees, radians, or servo steps, is the data we need to keep track of when calculating Shelley's arm movements. Although Shelley's arm design is not identical to the one pictured in Figure B, it is a type of revolute or articulated arm. A sample of Shelley's arm design can be found in Figure C.

Robot Arm Kinematics

Now that we have looked at some historic developments and types of arms, we can investigate different possible solutions to the arm control problem we attempted to solve for Shelley's arms. Controlling the final position of a robotic arm, not the path of motion, is a specialized kinematic problem. Robot arm kinematics deals with the analytical study of the geometry of a robot arm with respect to a fixed coordinate system without regard to the forces that cause the motion. It deals with the relation between the joint rotations and the position and orientation of the end-effector.[3] Applied to Shelley's layout in Figure C, kinematics studies the rotation of servos 1 through 6 and the resulting position of the claw.

There were two ways to look at this problem. First, if we looked at each servo and link individually, starting with 1 and working towards 6, we could tell where the claw would finish. On the other hand, we could work backwards by assigning a location for the claw and then determining the rotational settings required of the servos, working from 6 to 1, to deliver the claw to that point. These two approaches are the basis of the two problems in the study of kinematics. These two fundamental problems are designed to answer the following questions of specific interest in robot arm kinematics[3]:

1. For a given manipulator, given the joint angle vector and the geometric link parameters, where n is the number of degrees of freedom, what is the position and orientation of the end-effector of the manipulator with respect to a reference coordinate system?
2. Given a desired position and orientation of the end-effector of the manipulator and the geometric link parameters with respect to a reference coordinate system, can the manipulator reach the desired position and orientation? And if it can, how many different manipulator configurations will satisfy the same condition?

The first question outlines the direct kinematics problem while the second refers to the inverse kinematics problem. The block diagram in Figure D indicates the relationship between these two problems.

The Direct Kinematics Problem

The direct method looks at the properties of each joint and link starting at the base

to determine the position of the end-effector. Each joint-link pair constitutes one degree of freedom. In Shelley's case, she had essentially six joint-link pairs, resulting in six degrees of freedom. Since the links may rotate and/or translate with respect to a reference coordinate system, the total spatial displacement of the end-effector is due to the angular rotations and linear translations of the joint-link pairs. One of the most popular methods to solve the direct kinematics problem is a generalized and systematic approach utilizing matrix algebra to describe and represent the spatial geometry of the joint-link pairs of the robot arm with respect to a fixed reference coordinate system. This solution is referred to as the Denavit and Hartenberg method[3]. Essentially, this method uses a homogeneous transformation matrix to describe the spatial relationship between any two adjacent links. When chain-multiplied together, this set of matrices produces one matrix that represents the spatial geometry of the entire arm in terms of an attached coordinate system. In other words, location of the end effector is given in terms of the special arm coordinate system, not the cartesian world coordinate system, or a further homogeneous transformation matrix must be found to relate the two coordinate systems.

The Inverse Kinematics Problem

The inverse kinematics problem works essentially opposite to the direct problem. The claw is positioned as needed, and then we work backwards through the joints, or servos, to determine the required rotational settings of each one. All calculations are done based on the world cartesian coordinate system instead of an arm-attached reference frame.

Once again, matrix algebra may be used to solve the inverse problem in much the same way as it is used to solve the direct problem. The only differences would be that the matrices are chain-multiplied in reverse order, and, since the matrices for each joint-link pair in the inverse solution are given in terms of the world coordinate system and not an attached coordinate system, there is no need to calculate an extra homogeneous transformation matrix to convert between the two coordinate systems.

A more straight-forward approach, known as the geometric approach, to the inverse kinematics problem is also available. In this method, the reach, or work envelope, of each joint-link pair is studied and represented with a geometric shape. The intersection points of these geometric work envelopes are calculated starting at the end-effector and working back towards the base. One intersection point between each joint-link pair's work envelope and that of the next joint-link pair in the chain is chosen to represent a joint in the robotic arm. If all of the work envelopes intersect without requiring more rotation from any servo than it can provide, a solution is found. From the many intersection points, we can calculate the rotational settings of the various servos. In Shelley's case, since the joints are all revolute (Figure A), the geometric shapes representing the joint-link pair work envelopes are all simple circles centered at a servo, or the end-effector coordinates. (Refer to Figure E for an example of two joint-link combinations from Shelley's arms).

Choosing The Method

For our project, we were looking for the most straight-forward, least complicated

solution to controlling Shelley's arms. We first decided that since we wanted to input the end-effector coordinates in terms of a world coordinate system and have the program output the required rotational settings of the servos, the inverse kinematics problem was the best approach because the direct kinematic approach solves a completely different problem where the servo settings are known and the end-effector position is the calculated value. As previously noted, the work envelopes of the joint-link pairs in Shelley's arms could be represented by simple circles. For this reason, the geometric approach made more sense than using the far more complicated matrix algebra approach. As is the goal with most projects of this type, we wanted to keep things as simple as possible to provide a simple platform on which to build with future research.

Breakdown of Shelley's Arm Movement

Shelley's arms, manufactured by Robix/Advanced Design, Inc. (<http://www.robix.com>), contain six revolute servos, each of which rotates approximately 180 degrees. (Refer to Figure C for a picture of one of Shelley's arms). The rotation properties of the servos and the lengths of the links connecting them determine the overall work volume of the arm. The problem we were attempting to solve was: if we specify a location in three-dimensional space, how do we determine the rotational settings for each servo such that the end-effector reaches that location? To complicate this, there was the added fact that Shelley's vision capabilities were, and are, non-existent. This meant that the technique used to calculate the arm movements needed to be very precise because she could

not make the fine adjustments a typical human being would make when reaching for an object.[7] We reached a point where we knew which technique we wanted to employ, the geometric solution to the inverse kinematics problem. It was time to break down Shelley's arm movement and apply the technique.

The first thing that we decided when studying Shelley's general arm design was that two servos (1 and 2) controlled the horizontal movement and two (3 and 4) controlled the vertical movement. Servos 5 and 6 controlled the rotation and grasp of the claw, respectively. The layouts of the two servos in each dimension were similar. This meant that we could break up the three-dimensional movement into vertical and horizontal portions to make the task easier. Figure E shows a top view of the horizontal portion of the control scheme that was the focus of this project.

The next step was to break down this horizontal movement into the rotation of two joints connected to two links. Figure D shows the two-dimensional system, pulled from Figure C, that we were working with. The following description focuses on servos 1 and 2 and segments r_1 and r_2 from Figure C. We started off with five known values and one value provided by the user. The known values were the location of the base $(0,0)$, the length of both segments or links (r_1 and r_2), and the rotation limitations of each of the two joints (servos 1 and 2). We decided that the rotation of the servos would be measured in positive and negative rotation compared to a centered zero value because the Robix kits and driver software were designed to specify the rotation in this manner. The coordinates of

the end-effector (h,k) would be input by the user. For this two-dimensional representation, which provides only the horizontal movement of the arm, the end of link r_2 furthest from the base was deemed the end-effector. This particular point was used because, in a full three-dimensional implementation of this project, the far end of r_2 would intersect with the vertical system defined by servos 3 and 4.

Mathematical Model of Shelley's Arm Control

One end of r_1 was attached to the base while one end of r_2 was always connected to the end-effector coordinates. Since we were employing the geometric method, the range of available reach for each link was represented by a circle as in Figure E. One circle was centered at the base and had a radius equal to the length of r_1 . Assuming the base was at $(0,0)$ in our two-dimensional system, the equation for this circle was:

$$x^2 + y^2 = r_1^2 \quad \{\text{EQ 1}\}$$

The assumption that the base was at $(0,0)$ simplified several calculations, but it also cut down on the flexibility of the program to a certain extent because each arm would have needed its own coordinate system. Since we were only working with one arm for this project, this assumption was fine, but future projects to integrate both arms should allow for base of either arm to be at any coordinates. The second circle was centered on the end-effector coordinates (h,k) and had a radius equal to the length of r_2 . The equation of this circle was:

$$(x - h)^2 + (y - k)^2 = r_2^2 \quad \{\text{EQ 2}\}$$

Servo 1 and its associated angle ($A1$) were located at the base while servo 2 and its angle ($A2$) were located at the intersection point (x,y) between the two circles. (Refer to Figure E for details). The next step was to find this intersection point

We first had to make sure that the circles did intersect. We decided that if the end-effector coordinates were neither beyond the maximum reach of the two links nor closer than some minimum reach, then the two circles would intersect. The maximum reach was just the length of both segments, r_1 and r_2 , added together, but the minimum reach required some more calculations. With servo 1 set to any value, the minimum reach was based on the maximum rotation of servo 2 as shown in Figure F. The equations involved were as follows:

$$\alpha = \text{servo 2 max rotation} \quad \{\text{EQ 3}\}$$

$$\theta = \pi - \alpha \quad \{\text{EQ 4}\}$$

$$B = r_2 \sin \theta \quad \{\text{EQ 5}\}$$

$$C = r_2 \cos \theta \quad \{\text{EQ 6}\}$$

$$A = r_1 - C \quad \{\text{EQ 7}\}$$

$$\text{Distance} = (A^2 + B^2)^{1/2} \quad \{\text{EQ 8}\}$$

The resulting code for this calculation was:

```
#define MINREACH sqrt(pow(r2*sin(PI-SERVO2MAXR),2) +
pow(r1-(r2*cos(PI-SERVO2MAXR)),2))
```

{CODE 1}

Now that we had these minimum and maximum distance values, we knew that if the end-effector coordinates were neither too far nor too close to the base, then there would be at

least one intersection point between the two circles. For this project, we were not concerned about which of the two intersection points, if there were two as in Figure E, was a better choice based on the current position of the arm because we only wished to use whichever intersection led to a viable solution. A viable solution meant one which did not require excess rotation from either servo. Ideally, the program would decide which intersection point required the least arm movement from the current position, but this was beyond the scope of this project.

The next step was to determine an intersection point (x,y) . To accomplish this, we simply solved both previous circle equations, {EQ 1} and {EQ 2}, simultaneously for x and y . By subtracting {EQ 2} from {EQ 1} and solving for y we obtained:

$$y = -(h/k)x + ((r_1^2 - r_2^2 + h^2 + k^2) / 2k) \quad \text{\{EQ 9\}}$$

by multiplying through on the quadratic terms and solving for y . Next, we substituted this y value into the first circle equation {EQ 1}:

$$x^2 + [-h/k + ((r_1^2 - r_2^2 + h^2 + k^2) / 2k)] = r_1^2 \quad \text{\{EQ 10\}}$$

Then, we set this equal to zero and simplified to determine the A, B, and C coefficients of the x terms to solve for x using the quadratic formula:

$$\begin{matrix} A & B & C \\ ((h^2/k^2) + 1)x^2 - [(h(r_1^2 - r_2^2 + h^2 + k^2)) / k^2]x + ((r_1^2 - r_2^2 + h^2 + k^2) / 2k) - r_1^2 = 0 & \text{\{EQ 11\}} \end{matrix}$$

These coefficients for the various x terms were then plugged into the quadratic equation and a solution was found for x .

```
intrsectn.x = (-b - sqrt(pow(b,2) - (4*a*c))) / (2*a);    {CODE 2}
```

The variables a , b , and c were previously defined as the x coefficients from {EQ 11}. This is where the existence of two intersection points caused a problem. Depending on whether we added or subtracted in the quadratic equation solution, we got two different answers in many cases. We chose to subtract first, and then to add only if the resulting value would require a greater rotation than the servo's capabilities. The reasoning behind this was that for the arm with which we were working, Shelley's left, negative rotation of servo 1 translated into counter-clockwise rotation. Counter-clockwise rotation for that arm was away from Shelley's head; and it made sense to try to avoid obscuring her "vision" or making contact with the head. Once an appropriate value for x was found, we used it to solve for y in {EQ 9}.

```
intrsectn.y = (pow(r1,2) - pow(r2,2) - (2*intrsectn.x*ee.x) +  
              pow(ee.x,2) + pow(ee.y,2)) / (2*ee.x);    {CODE 3}
```

In the code example, $ee.x$ and $ee.y$ are equivalent to h and k , respectively. This gave us the coordinates of our intersection point.

Once we obtained the coordinates of the intersection point (x,y) , the calculation of the values for the angles, $A1$ and $A2$, were the next step. From the layout in Figure E and the knowledge that the sine of an angle in a right triangle is equal to the opposite side over the hypotenuse, we could calculate that:

$$A1 = \sin^{-1}(x/r_1) \quad \text{{EQ 12}}$$

```
theangle = asin(i.x/r1);    {CODE 4}
```

This gave us the angle at which servo 1 needed to be set. Servo 2 was a little more complicated. We needed to find the distance (L in Figure E) from the end-effector coordinates to the “line” created by r_1 . If a line is defined by the equation:

$$Ax + By = C \quad \{\text{EQ 13}\}$$

then the distance from a point (x,y) to the line would be [6]:

$$|Ax + By - C| / (A^2 + B^2)^{1/2} \quad \{\text{EQ 14}\}$$

This is based on drawing a line from the point perpendicular to the known line, and then measuring the distance, using the basic distance formula, from the original point to the point where the two lines intersect. Using the point-slope line equation, we defined the “line” created by r_1 as:

$$y = \text{slope} * x \quad \{\text{EQ 15}\}$$

The b variable from the basic point-slope equation is zero because we assumed that the base was at $(0,0)$. Also as a result of this assumption, the *slope* was equal to y/x . We converted {EQ 15} into the form of {EQ 13} and found the following values to substitute in {EQ 14}:

$$A = -\text{slope} \quad \{\text{EQ 16}\}$$

$$B = 1 \quad \{\text{EQ 17}\}$$

$$C = 1 \quad \{\text{EQ 18}\}$$

We plugged these values into the distance equation and obtained a value for the distance L shown in Figure E.


```
dist = fabs((-slope*ee.x) + ee.y) / sqrt(pow(slope,2) + 1); {CODE 5}
```

Next, we solved for the angle(A_2) of servo 2 once again using the rules of right triangles and the newly found L value:

$$A_2 = \sin^{-1}(L / r_2) \quad \{\text{EQ 19}\}$$

```
theangle = asin(dist/r2); {CODE 6}
```

At this stage, we had both rotational settings, in radians, for the servos in question.

The last task was to convert from radians into Robix step values. We first manually measured the maximum rotation of the servos in both radians and Robix step values. Next, we found the percentage of maximum radian rotation we had just calculated for each servo. Then, we multiplied this percentage by the maximum Robix step value that corresponded to the maximum radian value for each servo. This gave us the rotational settings for servos 1 and 2 in Robix steps which we then input into the Robix control program to actually move the arm.

Problems and Solutions

Joint Over-rotation

Along the way, we encountered some minor problems we had to solve. One conflict arose when an intersection point was determined {EQ 10 and EQ 11}. We needed to make sure that both servos could rotate to the extent that the solution demanded. Comparing the solution to the maximum rotation value was the way to keep track of this for servo 2. Servo 1 was a different situation. To avoid problems which arose regarding the over-

rotation of servo 1, we devised a method to check the y coordinate of the intersection point to make sure that it was not less than some minimum value that r_1 could reach based on the rotation of servo 1. The schematic for this can be found in Figure G. The equations to calculate this minimum y intersection value were:

$$\alpha = \text{max rotation of servo 1} \quad \{\text{EQ 20}\}$$

$$\theta = \alpha - (\pi / 2) \quad \{\text{EQ 21}\}$$

$$\text{min. intersection} = -(r_1 \sin \theta) \quad \{\text{EQ 22}\}$$

```
#define MININSCTNY -(r1*sin(SERVO1MAXR-(PI/2)))
```

{CODE 7}

This technique still worked even if the maximum rotation of servo 1 was less than $\pi/2$ because the negative sign in {EQ 22} then resulted in a positive minimum y intersection value which was appropriate for the problem because r_1 was still in the first quadrant of the graph where both x and y are positive. This minimum y intersection value applied regardless of positive or negative rotation of servo 1 because we adjusted the servos to rotate the same amount on either side of the straight, zero position. If the rotation demanded of servo 1 was not within range, we first tried the alternate intersection point (adding when solving the quadratic equation instead of subtracting), and ended the program only if the new rotation value was still out of range. If the rotation value calculated for the servo 2 was out of range, the program would immediately stop and print an error message to the screen.

Symmetrical Rotation

Another problem presented itself because we used the maximum rotation of either servo for some calculations {EQ3 and EQ20}. To allow for simpler calculations, we had to ensure that each servo could rotate the same number of degrees both directions from the zero centered point. To do this, we had to pay close attention to the rotation of the servos during installation.

k Coordinate of End-effector Equal To Zero

When calculating the intersection point {EQ 10 and EQ 11}, we decided that we needed to check if the k value of the end-effector coordinates (h,k) was zero. The equations we used to solve for the intersection used k in the denominator. If k was zero, we simply switched the solving order and solved for y first using the quadratic equation so that h would be in the denominator instead of k . We then solved for x in terms of y . (Refer to functions `GetIntersection1` and `GetIntersection2`, pages 3 to 5 in the Program Appendix, to see the two orders of calculating and the system to check for $k = 0$. We knew that h and k could not both be zero because that would produce an error message that the end-effector was too close to the base.

Servo 2 Rotation Calculations

When working with the angle for servo 2 {EQ 13 - EQ 19} we ran into additional problems. The first problem was that it was possible to have an undefined slope for the "line" created by extending r_1 . This happened when the x value of the intersection point

was zero. If it was equal to zero the distance (L) we wanted to calculate was simply the absolute value of the h coordinate of the end-effector, and the x value on the "line" when inserting k into the equation {EQ 15} for the line was just the x coordinate of the intersection. Refer to function **GetServo2Angle** (Program Appendix pages 4 and 5) to see the execution of this check. This data was used in the solution to the following problem. It was difficult to tell if the rotation of servo 2 should be positive or negative. This was because the straight, zero position was in line with r_1 and not the reference coordinate system. An example of this is in Figure E. To solve this problem, we plugged the k value of the end-effector into the previously described point-slope equation {EQ 15} for the "line" created by r_1 and solved for x . If this calculated value was less than the h value of the end-effector, then the rotation was positive. Otherwise, it was negative. This routine can also be found in **GetServo2Angle** (Program Appendix pages 4 and 5).

Claw Offset

The final problem we found was that the claw is offset to the left from the line of r_2 . This problem was caused because the actual rotating part on the servos is not centered. We had to decide if the claw was going to be offset to the right or the left during construction based on the positioning of servos 5 and 6. The choice between an offset to the left or right was completely arbitrary with one no better than the other. (Refer to Figure C for details). We figured out the compensating angle to by measuring the offset (opposite side) and the length of r_2 (hypotenuse) and using the sine function as it applies to right triangles. This

value is assigned to the constant, **CORRECT**, in the program and is added to the rotation of servo 2 to make it rotate slightly clockwise.

Future Projects

After we solved all of the various problems, the program performed correctly, and the project was a success. (Refer to the test runs in the Program Appendix). However, there is still a great deal of work left for future researchers.

Full Three-Dimensional Implementation

The first project should probably be to turn this two-dimensional program into a full three-dimensional implementation. This can be achieved by combining two versions of the program located in the Program Appendix: one for each of the horizontal and vertical movements. If you look at Figure C, one set of equations would apply to servos 1 and 2 and another set to servos 3 and 4. Starting at the claw, one would work backward through the servos, towards the base, to get a solution. Servos 3 and 4 would directly relate to servos 1 and 2 respectively as they are used in the current implementation. The intersection point between the circles around the end of link r_2 and servo 3 would occur when the circle from servo 3 reaches the height of r_2 .

Vision Processing

Another important project would be vision processing. This would allow Shelley to actually view her environment and "intelligently" select objects to manipulate. Ideally, she would be able to triangulate distance with her stereoscopic vision in the same manner as a

human being. This would allow her to move her arm to a specific object location without requiring coordinates from the user. The coordinate system would be built into her programming.

Kinematic Calibration

Vision processing would lead to other capabilities such as kinematic calibration[1]. In this technique, the robot self-calibrates its arms by moving to all extremes, viewing the results and taking optical measurements, and storing the data for calculations in the movement program. Shelley would be able to calculate, on her own, the lengths of the links and the rotational limitations of the servos in her arms. This would be particularly useful because the servo rotation values tend to change due to usage. At this point, the human operator must take all the detailed measurements.

Trajectory Planning

Another capability would be trajectory planning[3]. This is where a spline path is created to represent the movement of the arm from the current position to the desired position. The eyes would be able to scan this path and determine if any objects are obstructing the movement of the arm. Then the path could be modified if necessary.

Touch Sensitivity

Along the same sensory input line, having touch sensitivity would also be a tremendous asset. Shelley would be able to grasp objects without having to worry about their dimensions. She could simply squeeze until a desired pressure between the claw and

the object was achieved. This capability could also keep her from damaging any delicate items she is manipulating and from attempting to pick up an item which is too heavy for her arms to handle. A heavy object would just slide out of the claw if she were squeezing it with the correct intensity.

Conclusion

The program resulting from this project is an important step in the development of Shelley as an autonomous “being.” The program functions properly, and it may be used, as is, to do simple manipulation which only requires two-dimensional coordinates. An example of this is tic-tac-toe. Placing marbles on a tic-tac-toe board requires only two-dimensional position coordinates if the arm is positioned such that the vertical movement of the claw is the same for every spot. In other words, the claw drops in exactly the same manner to lower a ball into any of the nine positions on the board. In fact, Shelley defeated Illinois State's robot, IRIS, at a tic-tac-toe competition on Saturday, April 12 during the research conference. She also performed admirably the following Sunday against human competitors at Family Day.

Without too much work, a future researcher could transform the program into a full three-dimensional implementation. The goal behind the general techniques I utilized was to provide a flexible platform for easy modifications in the future, and I feel I have achieved this through both the mathematical model I have outlined and the program which implements it.

References

- 1.) An, Chae H., et al, Model-Based Control of A Robot Manipulator, The MIT Press, Cambridge, Mass., 1988.
- 2.) Asimov, Isaac, and Frenkel, Karen A., Robots: Machines in Man's Image, Harmony Books, New York, NY, 1985.
- 3.) Fu, K. S., et al, Robotics: Control, Sensing, Vision, and Intelligence, McGraw-Hill Book Company, New York, NY, 1987.
- 4.) Korein, James U., A Geometric Investigation of Reach, The MIT Press, Cambridge, Mass., 1985.
- 5.) Rosheim, Mark E., Robot Evolution: The Development of Anthrobotics, John Wiley & Sons, Inc., New York, NY, 1994.
- 6.) Thomas, George B., and Finney, Ross L., Elements of Calculus and Analytic Geometry, Addison-Wesley, Menlo Park, CA, 1989.
- 7.) Wickens, Christopher D., Engineering Psychology and Human Performance, Harper Collins, New York, NY, 1992.

Special Thanks to:

Dr. Lon Shapiro for assistance on some of the robotic control ideas and paper content

Dr. Keith Coates for assistance with some of the mathematical concepts

Dr. Susan Anderson-Freed for assistance with paper content

Dr. Lawrence Stout for participating in my defense

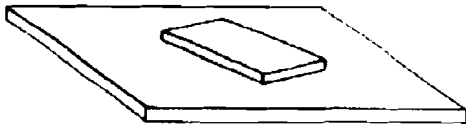
Karen Anderson for proof-reading

Figure A

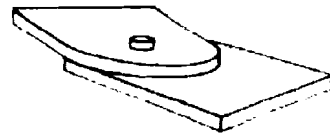
Six Different Robotic Arm Joint Designs

[3]

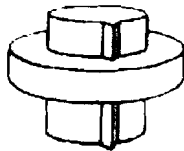
Planar



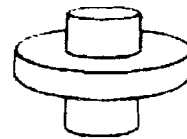
Revolute



Prismatic



Cylindrical



Screw



Spherical

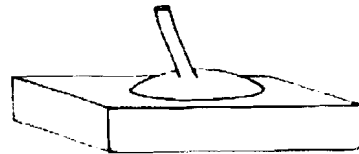


Figure B

Four Different Robotic Arm Designs

[3]

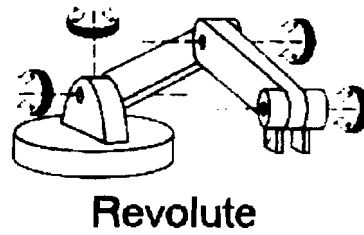
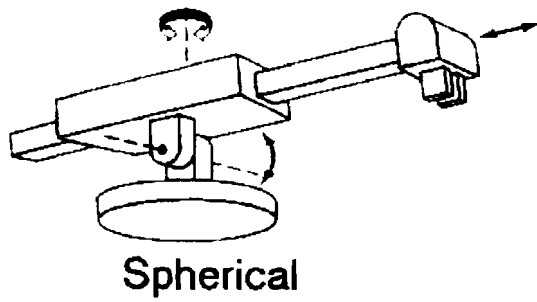
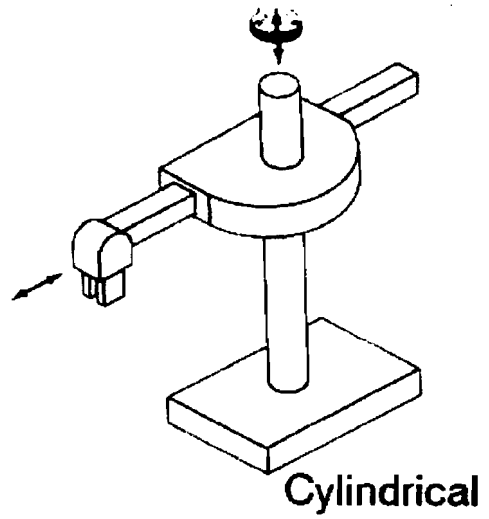
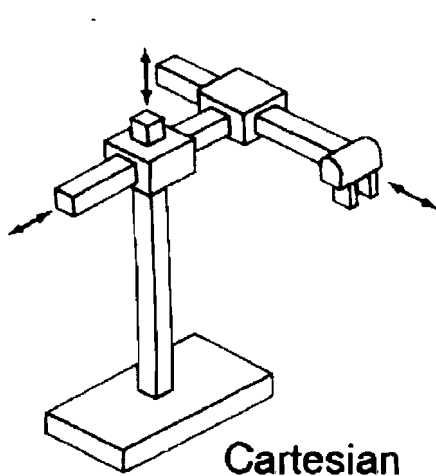


Figure C

Shelley's General Arm Layout

Robix User's Manual (<http://www.robix.com>)

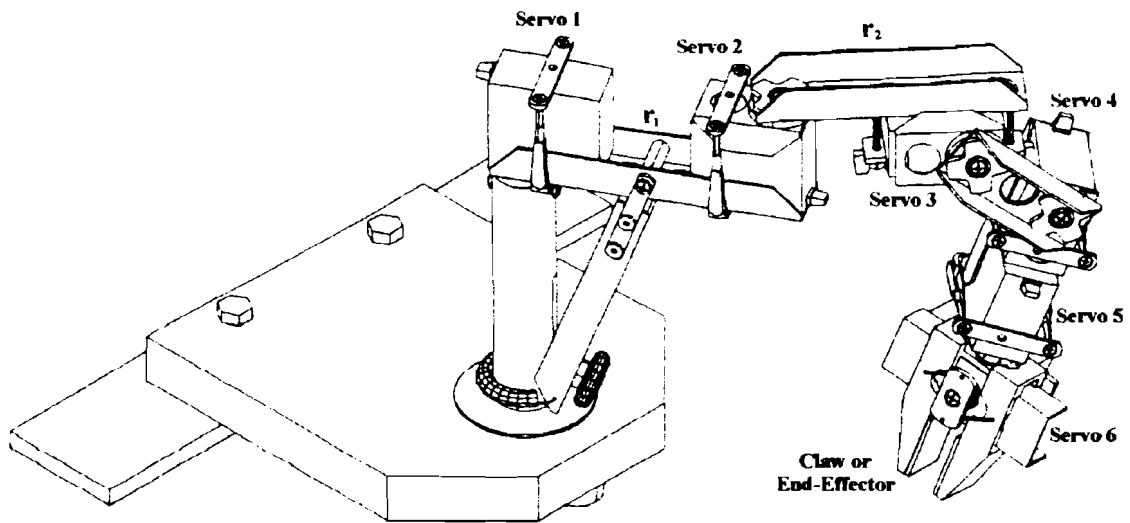


Figure D

Arm Kinematics Block Diagram

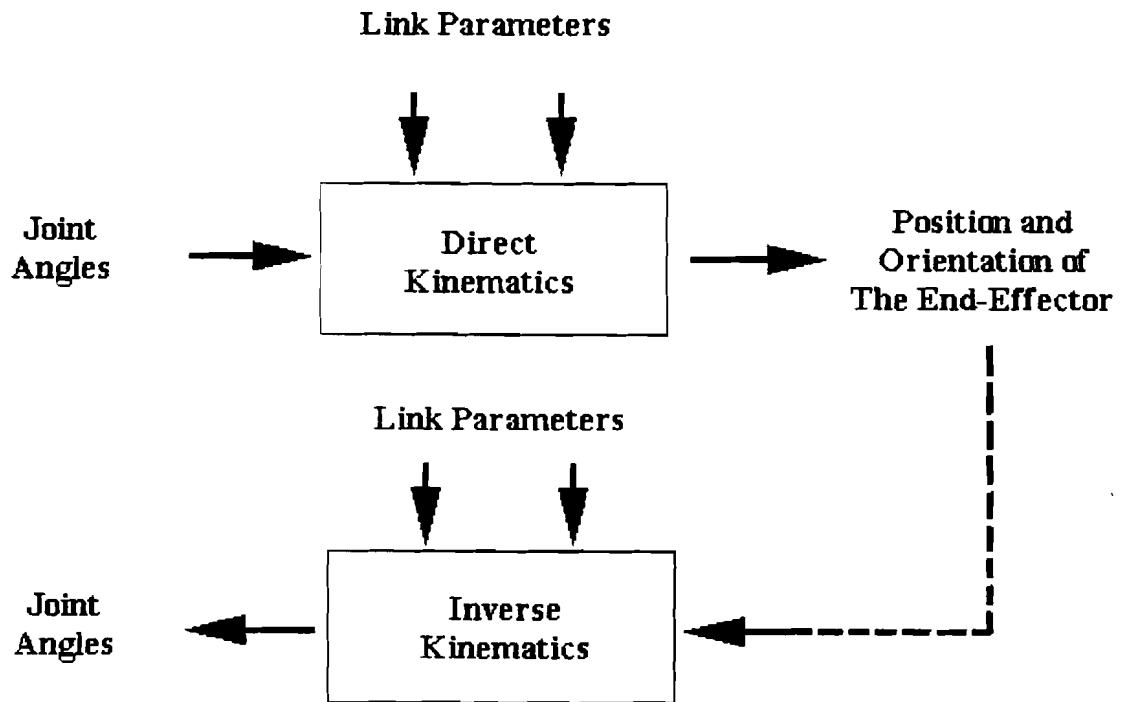
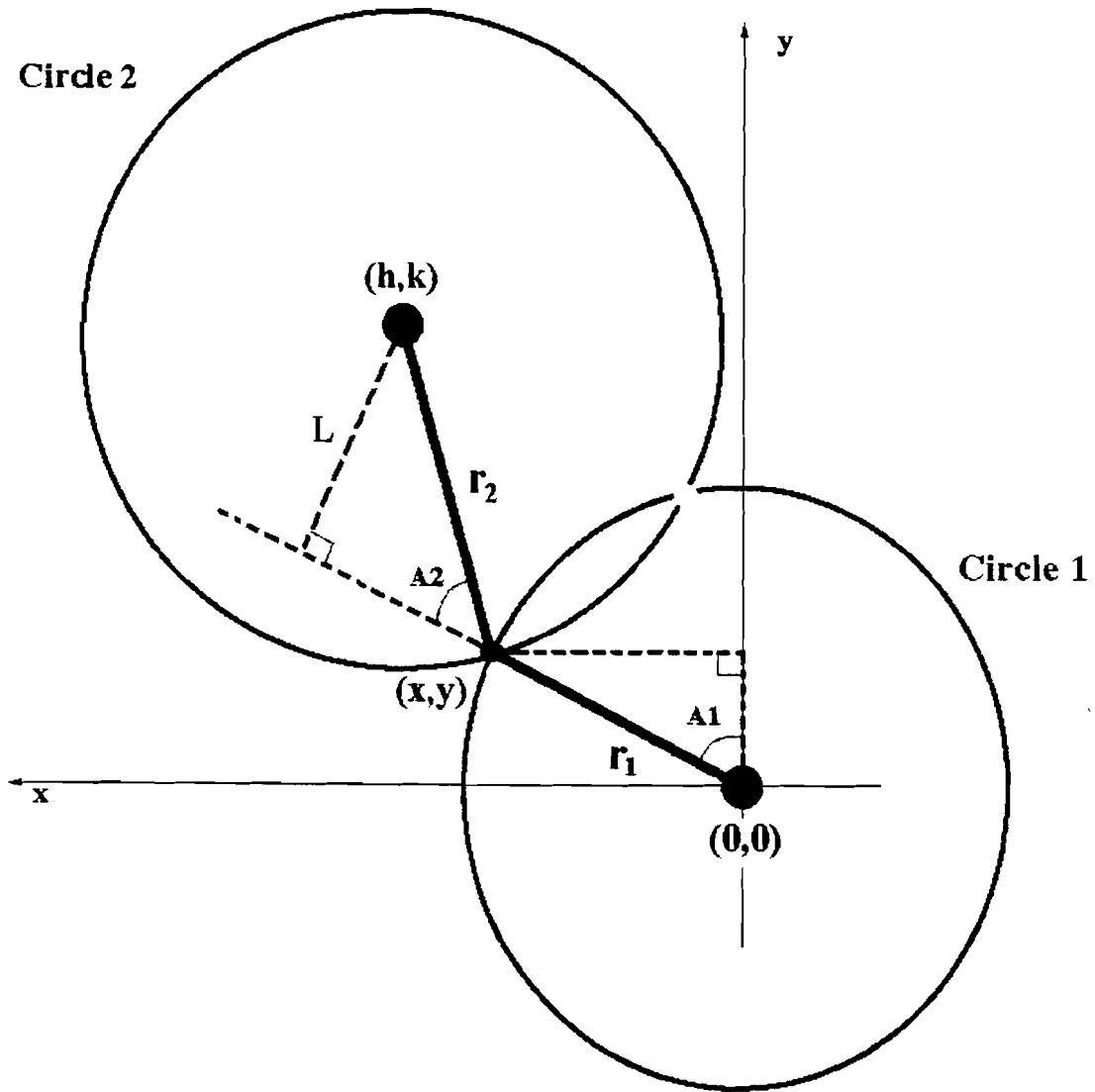


Figure E

Two-Dimensional Horizontal Breakdown Top View

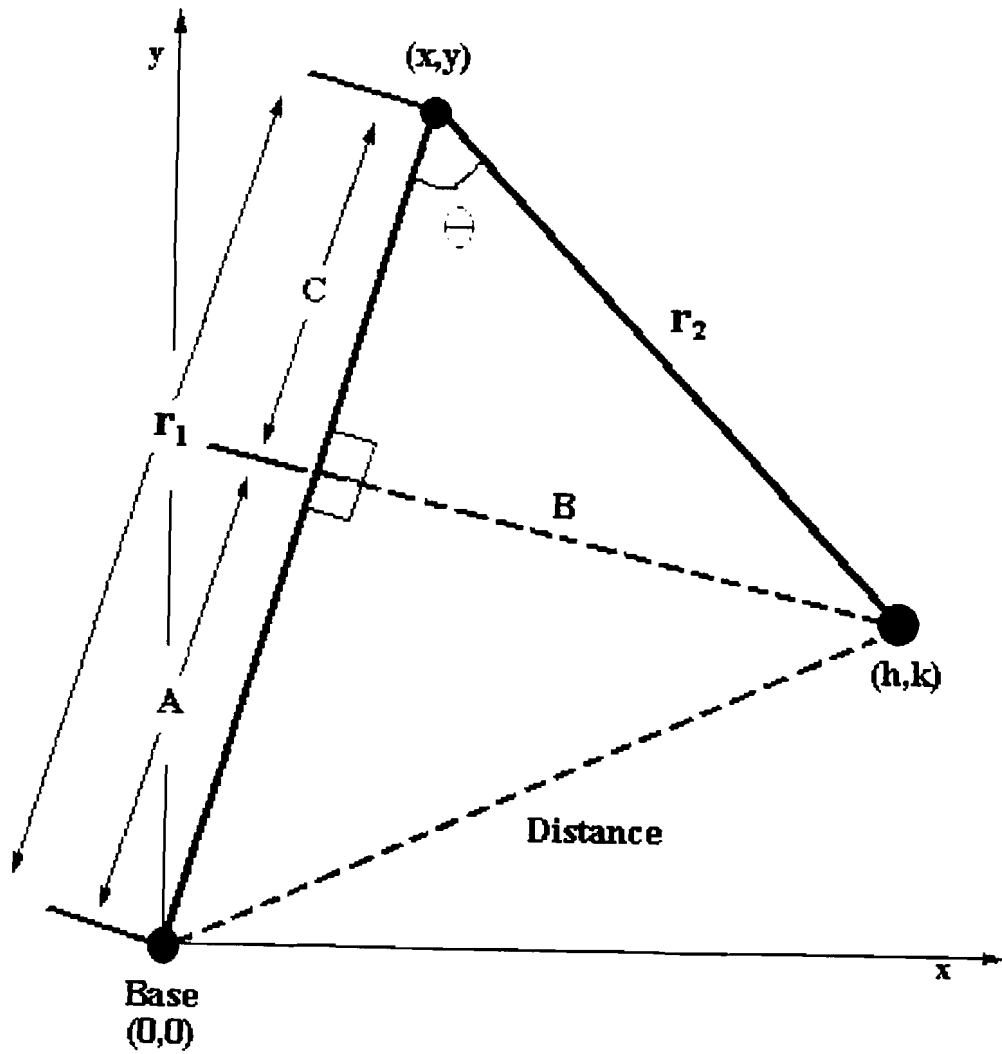


$(0,0)$ =Base; (h,k) =end-effector; (x,y) =primary intersection point

r_1 =Link 1; r_2 =Link 2

Figure F

Minimum Arm Reach Schematic

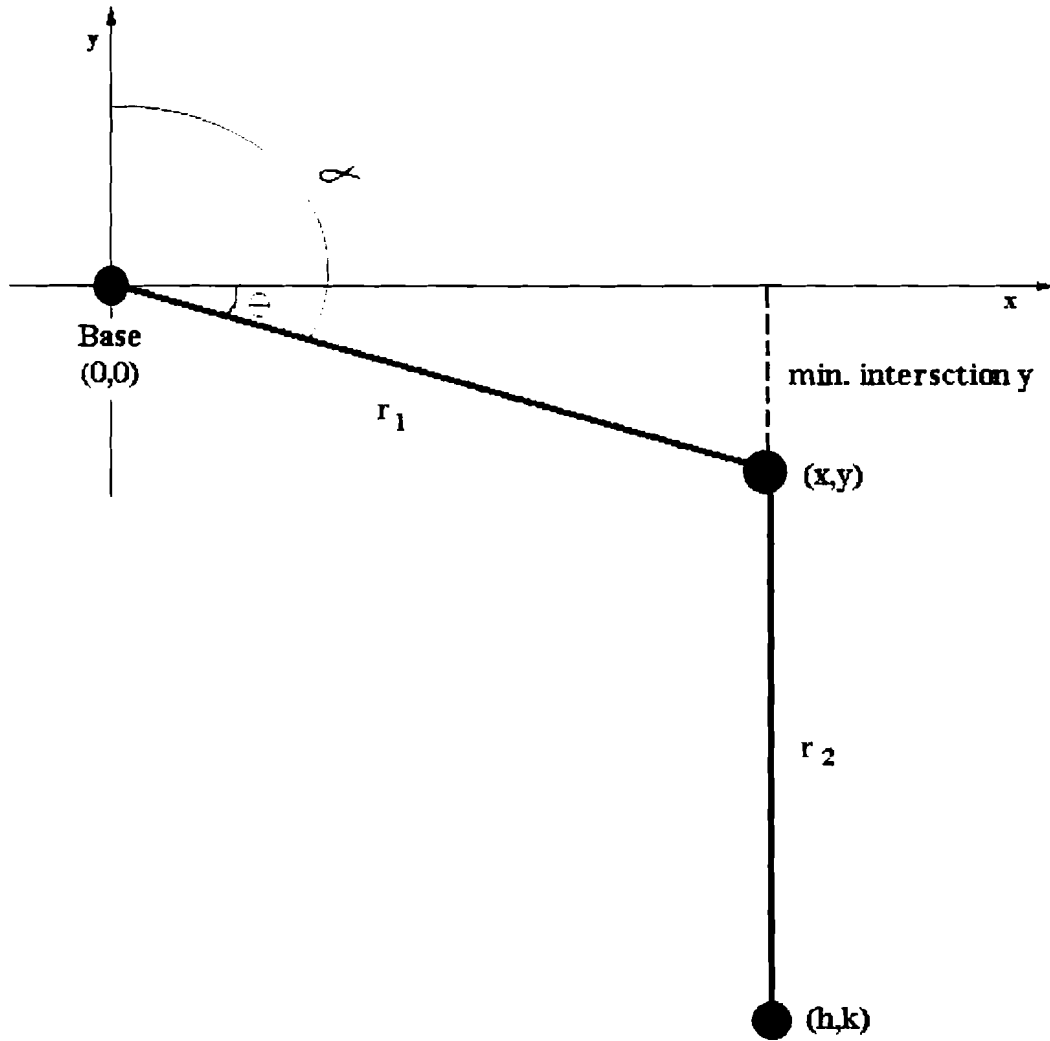


$(0,0)$ =Base; (h,k) =end-effector; (x,y) =primary intersection point

r_1 =Link 1; r_2 =Link 2

Figure G

Minimum Intersection y Value Schematic



$(0,0)$ =Base; (h,k) =end-effector; (x,y) =primary intersection point

r_1 =Link 1; r_2 =Link 2

**Program
Appendix**

```
/* filename: tttarms2.c */

/* Craig Materick
   CS 499 - Research Honors
   Spring 1997
   2D simulation of Shelley's arm movement

The CODE # markers work in conjunction with the research honors paper
*/

#include <math.h>
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>

/* so we can work in radians */
#define PI 3.141592653589793

/* radian constants */
/* min and max rotation assumed to be equidistant from 0 */
#define SERVO1MAXR (PI/2.0)
#define SERVO2MAXR (PI/2.0)

/* robix step constants */
/* max rotation */
#define SERVO1MAXS 1400
#define SERVO2MAXS 1400

/* segment length constants */
#define r1 10.0
#define r2 13.0

/* angle to correct for offset of the claw */
#define CORRECT ((4.965*PI)/180.0)

/* Min reach constant */
/*****CODE 1*****/
#define MINREACH sqrt(pow(r2*sin(PI-SERVO2MAXR),2) + pow(r1-(r2*cos(PI-SERVO2MAXR)),2))

/* prevents r1 from over-rotating */
/*****CODE 7*****/
#define MININSCTNY -(r1*sin(SERVO1MAXR-(PI/2)))

/* type to hold 3D floating point point coordinates */
typedef struct {
    float x, y, z;
} threeDPTType;

/* function prototypes */
float Distance(threeDPTType, threeDPTType);
/* Input : 2 points (2D coordinates for this program)
   Output: the floating point distance between the points
*/
threeDPTType GetIntersection1(threeDPTType);
/* Input : the coordinates of the end-effector as a point variable
   Output: the coordinates of the intersection of the two circles
```

```
    Uses subtraction in the solution of the quadratic equation
    Assumes that there is at least one intersection point
*/
threeDPTType GetIntersection2(threeDPTType);
/* Input : the coordinates of the end-effector as a point variable
   Output: the coordinates of the intersection of the two circles
   Uses addition in the solution of the quadratic equation
   Assumes that there is at least one intersection point
*/
float GetServo1Angle(threeDPTType);
/* Input : the coordinates of the intersection point
   Output: the angle of servo 1 in radians
   Assumes the base is at (0,0,0)
*/
float GetServo2Angle(threeDPTType, threeDPTType);
/* Input : the coordinates of the intersection point
   Output: the angle of servo 2 in radians
   Assumes the base is at (0,0,0)
*/
int AngleToStepServo1(float);
/* Input : the rotation of servo 1 in radians
   Output: the rotation of servo 1 in Robix steps
   assumes proper constants have been defined to compare the two
*/
int AngleToStepServo2(float);
/* Input : the rotation of servo 2 in radians
   Output: the rotation of servo 2 in Robix steps
   assumes proper constants have been defined to compare the two
*/

int main()
{
    threeDPTType base, eeffector, intersect;
    float servo1angle, servo2angle, eedist;
    int step1, step2;

/* Where, in the coordinate system is the base at? */
/* can be changed with future development */
    base.x = 0;
    base.y = 0;
    base.z = 0;

/* Object Location input */
    printf("X coordinate of end-effector: ");
    scanf("%f", &eeffector.x);
    printf("Y coordinate of end-effector: ");
    scanf("%f", &eeffector.y);

/* find the distance from the end-effector to the base */
    eedist = Distance(eeffector, base);

/* check if it is too close or far away */
    if (eedist < MINREACH)
        printf("\nThe end-effector coordinates are too close to the base (0,0)\n");
    else if (eedist > (r1+r2))
        printf("\nThe end-effector coordinates are too far from the base (0,0)\n");
}
```

```

else
{
    intersect = GetIntersection1(eeffector);

/* check if we are asking for too much rotation from servo 1 */
/* if so, try another intersection point */
    if (intersect.y < MININSCTNY)
        intersect = GetIntersection2(eeffector);

/* check if the new angle is OK */
/* if not, the program is over */
    if (intersect.y < MININSCTNY)
        printf("\nEnd-effector coordinates not in work envelope(servo 1)\n");

/* if the rotation is fine, we continue onward */
else
{
    servolangle = GetServo1Angle(intersect);
    printf("Intersection X: %f\n", intersect.x);
    printf("Intersection Y: %f\n", intersect.y);
    printf("servo 1 angle: %f\n", servolangle);
    step1 = AngleToStepServo1(servolangle);
    printf("servo 1 steps: %d\n", step1);

    servo2angle = GetServo2Angle(intersect, eeffector);
/* check if we are asking for too much rotation from servo 2 */
    if (fabs(servo2angle) > SERVO2MAXR)
        printf("\nEnd-effector coordinates not in work envelope(servo 2)\n");

/* if the rotation is fine, we continue onward */
else
{
    printf("servo 2 angle: %f\n", servo2angle);
    step2 = AngleToStepServo2(servo2angle);
    printf("servo 2 steps: %d\n\n", step2);
} /* else servo 2 OK */
} /* else servo 1 OK*/
} /* else distance OK*/
} /* main */

float Distance (threeDPTYPE p1, threeDPTYPE p2)
{ /* finds the distance between two points using basic distance equation*/
    float d;
    d=sqrt((pow(p1.y-p2.y,2))+(pow(p1.x-p2.x,2)));
    return d;
} /* Distance */

threeDPTYPE GetIntersection1 (threeDPTYPE ee)
{ /* returns the point of intersection using "-" */
/* when solving the quadratic equation */
    threeDPTYPE intrsctn;
    float a, b, c;
/* if "k" equals zero */
    if (ee.y == 0)
    {
/* find a, b, and c for the quadratic equation */

```

```

    a = 1;
    b = 0;
    c = (pow((pow(r1,2) - pow(r2,2) + pow(ee.x,2) +
        pow(ee.y,2)) / (2*ee.x),2)) - pow(r1,2);
/* solve for y using the quadratic equation */
    intrsctn.y = (-b - sqrt(pow(b,2) - (4*a*c))) / (2*a);
/* solve for x using y */
    intrsctn.x = (pow(r1,2) - pow(r2,2) - (2*intrsctn.y*ee.y) +
        pow(ee.x,2) + pow(ee.y,2)) / (2*ee.x);
    intrsctn.z = 0;
} /* if */

/* if "k" doesn't equal zero */
else
{
/* find a, b, and c for the quadratic equation */
    a = (pow(ee.x,2) / pow(ee.y,2)) + 1;
    b = -((ee.x * (pow(r1,2) - pow(r2,2) + pow(ee.x,2) +
        pow(ee.y,2))) / pow(ee.y,2));
    c = (pow((pow(r1,2) - pow(r2,2) + pow(ee.x,2) +
        pow(ee.y,2)) / (2*ee.y),2)) - pow(r1,2);
/* solve for x using the quadratic equation */
/*****CODE 2*****/
    intrsctn.x = (-b - sqrt(pow(b,2) - (4*a*c))) / (2*a);
/* solve for y using x */
/*****CODE 3*****/
    intrsctn.y = (pow(r1,2) - pow(r2,2) - (2*intrsctn.x*ee.x) +
        pow(ee.x,2) + pow(ee.y,2)) / (2*ee.y);
    intrsctn.z = 0;
} /* else */
return intrsctn;
} /* GetIntersection1 */

```

```

threeDPTType GetIntersection2 (threeDPTType ee)
{ /* returns the point of intersection using "+" */
/* when solving the quadratic equation */
    threeDPTType intrsctn;
    float a, b, c;
/* if "k" equals zero */
    if (ee.y == 0)
    {
/* find a, b, and c for the quadratic equation */
        a = 1;
        b = 0;
        c = (pow((pow(r1,2) - pow(r2,2) + pow(ee.x,2) +
            pow(ee.y,2)) / (2*ee.x),2)) - pow(r1,2);
/* solve for y using the quadratic equation */
        intrsctn.y = (-b + sqrt(pow(b,2) - (4*a*c))) / (2*a);
/* solve for x using y */
        intrsctn.x = (pow(r1,2) - pow(r2,2) - (2*intrsctn.y*ee.y) +
            pow(ee.x,2) + pow(ee.y,2)) / (2*ee.x);
        intrsctn.z = 0;
    } /* if */

/* if "k" doesn't equal zero */
else

```

```

{
/* find a, b, and c for the quadratic equation */
a = (pow(ee.x,2) / pow(ee.y,2)) + 1;
b = -((ee.x * (pow(r1,2) - pow(r2,2) + pow(ee.x,2) +
pow(ee.y,2))) / pow(ee.y,2));
c = (pow((pow(r1,2) - pow(r2,2) + pow(ee.x,2) +
pow(ee.y,2)) / (2*ee.y),2)) - pow(r1,2);
/* solve for x using the quadratic equation */
intrsctn.x = (-b + sqrt(pow(b,2) - (4*a*c))) / (2*a);
/* solve for y using x */
intrsctn.y = (pow(r1,2) - pow(r2,2) - (2*intrsctn.x*ee.y) +
pow(ee.x,2) + pow(ee.y,2)) / (2*ee.y);
intrsctn.z = 0;
} /* else */
return intrsctn;
} /* GetIntersection2 */

float GetServo1Angle(threeDPType i)
{ /* returns the angle of servo 1 in radians based on the intersection point */
float theangle;
/*****CODE 4*****/
theangle = asin(i.x/r1);
return theangle;
} /* GetServo1Angle */

int AngleToStepServo1(float angle)
{ /* converts the calculated angle to robix steps */
int steps;
float temp;
/* find percent rotation */
temp = angle / SERVO1MAXR;
steps = floor((temp*SERVO1MAXS) + 0.5);
return steps;
} /* AngleToStepServo1 */

float GetServo2Angle(threeDPType i, threeDPType o)
{ /* returns the angle of servo 2 in radians based on the intersection point */
float slope, dist, theangle, tempx;
/* if segment 1 has an undefined slope */
if (i.x == 0)
{
dist = fabs(o.x);
tempx = i.x; /* the x value on the segment 1 "line" at the y value
of the end-effector */
} /* if */
else
{
/* get the slope of segment one, currently assumes the base is at (0,0,0) */
slope = i.y / i.x;
/* find the distance using the special distance equation */
/*****CODE 5*****/
dist = fabs((-slope*o.x) + o.y) / sqrt(pow(slope,2) + 1);
/* the x value on the segment 1 "line" at the y value of the end-effector */
tempx = o.y / slope;
} /* else */
/*****CODE 6*****/

```

```
theangle = asin(dist/r2);

/* now we have to determine if it's positive or negative rotation*/
if (temp > 0.0)
    theangle *= -1;
/* correct for the offset of the claw */
theangle += CORRECT;
return theangle;
} /* GetServo2Angle */

int AngleToStepServo2(float angle)
{ /* converts the calculated angle to robix steps */
    int steps;
    float temp;
/* find percent rotation */
    temp = angle / SERVO2MAXR;
    steps = floor((temp*SERVO2MAXS) + 0.5);
    return steps;
} /* AngleToStepServo2 */
```

Sample Input and Output

This sample data is intended to show that the mathematical calculations work.
***In all the tests, the angle of servo 2 has the claw offset corection added to it.

Test One

Input:

X coordinate of end-effector: 0

Y coordinate of end-effector: 23

Output:

Intersection X: 0.000000

Intersection Y: 10.000000

servo 1 angle: 0.000000

servo 1 steps: 0

servo 2 angle: 0.086656

servo 2 steps: 77

Test Two

Input:

X coordinate of end-effector: 23

Y coordinate of end-effector: 0

Output:

Intersection X: 10.000000

Intersection Y: 0.000000

servo 1 angle: 1.570796

servo 1 steps: 1400

servo 2 angle: 0.086656

servo 2 steps: 77

Test Three

Input:

X coordinate of end-effector: -7

Y coordinate of end-effector: 16

Output:

Intersection X: -9.462180

Intersection Y: 3.235296

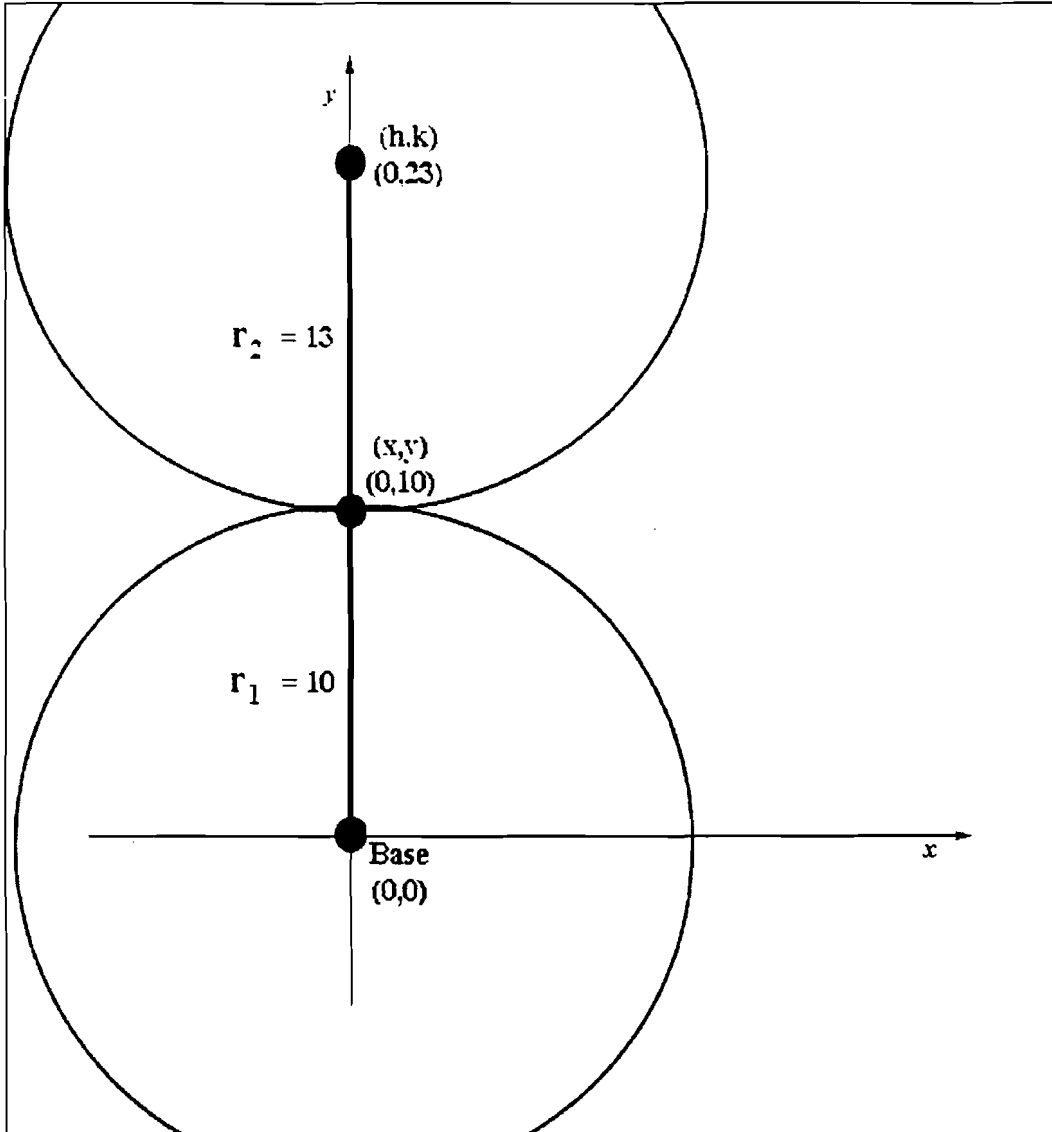
servo 1 angle: -1.241339

servo 1 steps: -1106

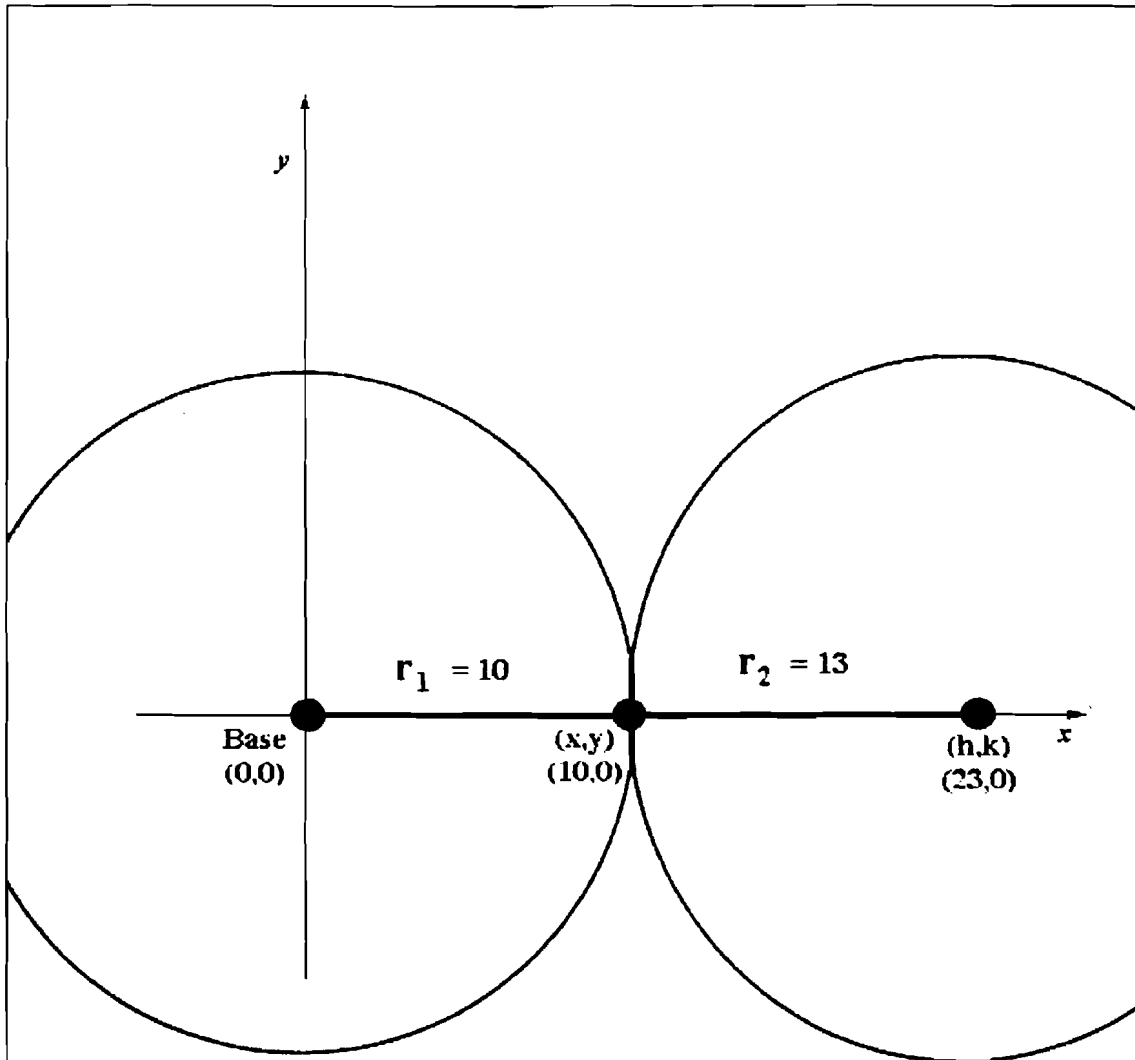
servo 2 angle: 1.518544

servo 2 steps: 1353

Test One



Test Two



Test Three

