Michael EngstromECE 578 – Embedded Robotics – Fall 2012 Prof. Perkowski

Homework 2 - Inverse Kinematics

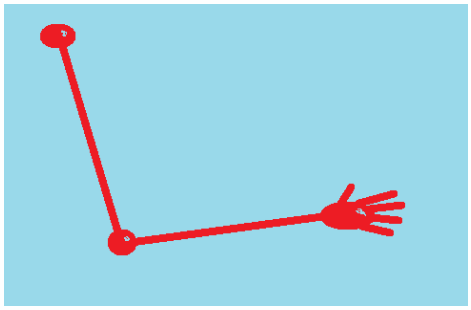
for

A Robot Arm with Hand to reach a target

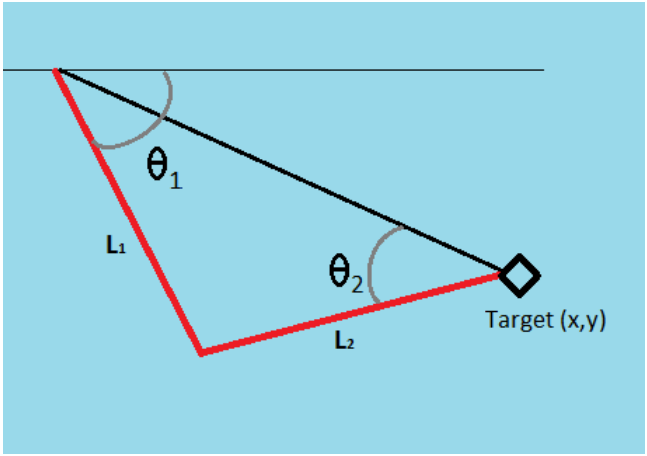
One of the tasks my team wants for our Robot Arm with Hand is to be able to shake hands with someone standing in front of it. Our plan is to place a proximity sensor on the hand and use a search method to discover objects in front of it. When the search is complete, we have arrived at our target (a human hand extended to shake). Homework 1 used a Law of Cosines approach to calculate the angles needed for a 2 DOF robot arm. This homework builds on these results and compares a random search approach to finding the angles for placing the end of the robot arm at the target.

Law of Cosines Inverse Kinematics

As show in Homework 1, by simplifying the model to 2 DOF (shoulder, elbow) we are able to bypass some of the trickier math involving transforms and translations and use the Law of Cosines with a simple triangle to find the angles needed to place the hand at the target. Here is an example of the Arm:



And here is the geometric representation with a desired target:



From the Law of Cosines we have:

$$c^2 = a^2 + b^2 - 2ab\cos(c)$$

Inserting our values and solving for the desired angles gives us:

$$\theta_2 = \arccos\left[\frac{x^2 + y^2 - {L_1}^2 - {L_2}^2}{2L_1L_2}\right]$$

And

$$\theta_1 = \arcsin\left[\frac{L_2\sin(\theta_2)}{\sqrt{x^2 + y^2}}\right] + \arctan 2\left(\frac{x}{y}\right)$$

Using the above angle formulas and some input values, a simple c++ program was written to calculate the output angles needed to place the hand at the target. Below is an example of the program running. Note that the angle calculations are converted from radians to degrees by multiplying by ($\frac{180}{\pi}$¿. When the function is integrated into our control program the degrees will be converted to the pulse width necessary to move the servos to the desired angles.

Random Angle Method

The Law of Cosines Inverse Kinematic method will be compared with a Random Angle method. This idea is to use a random angle change to reduce the error between where the robot is and the target. If the new angles do not move the hand closer to the target they are discarded and new random angles are generated.

Algorithm for Random Angle method:

- While (hand > (some close value to target) –OR– (count is too high))
    - Increment count
    - Move servo 1 small random distance
    - Move servo 2 small random distance
    - If hand is farther from goal
        - Discard new angle values
    - Else
        - Copy new angle values

The problem with the above algorithm is there are possible positions where one servo would need to move away from the target to allow another servo to move. This is an example of local minima prohibiting the realization of a goal. I had to

simplify the model and create a mathematical method (such as a computer program) to input the values and output the solution.

This is the code for the above program:

```
#include <iostream>
#include <string>
#include <math.h>
#include "StdAfx.h"
using namespace std;

//#define RAD_CONVERT = 57.296 //(180 / pi)

// Three points in space: base, elbow, hand
int _points_x [3] = {0, 0, 0};
int _points_y [3] = {0, 500, 1000};

// Random points in space: base, elbow, hand for randSearch algorithm
int _points_x_rand [3] = {0, 0, 0};
int _points_y_rand [3] = {0, 500, 1000};

//private var _distance:Number = 100;
int _distance = 1000;    //millimeters, assume 3ft arm reach and (upper
arm)=(lower arm)=500 mm
int _distance_rand = 1000;
//private var _lastPoint:int = 2;
int _lastPoint = 2;
int randIter = 0;

// Give a starting target in millimeters
float targetX = 250;
float targetY = 250;

//arm angles start straight, ie hanging down
float _theta1 = 0;
float _theta2 = 0;

//Random Angle variables
//arm angles start straight, ie hanging down
float _theta1_rand = 0;
float _theta2_rand = 0;

float RAD_CONVERT = (180 / 3.14159);

void randSearch()
{
            // Compute difference between start and end points
      float dx = (targetX - _points_x_rand[0]);
      float dy = (targetY - _points_y_rand[0]);
      cout << "Difference of (x, y): " << "(" << dx << ", " << dy << ")" << endl;
      // Compute distance between start and end points
      float dist = sqrt(dx*dx + dy*dy);
      cout << "Distance from base to target: " << dist << endl;
      // Compute angle between start and end points
      //float theta = atan2(dy,dx);
      float theta = float(rand() % 180 + 1);
      float temp = (int(theta * RAD_CONVERT)) % 180;
      cout << "Angle between base and target: " << temp << endl;
      // Clamp the distance
```

```
	float totalLength = _distance_rand * 2;
	//if( dist < totalLength ) {
		// Calculate first angle:
http://en.wikipedia.org/wiki/Dot_product#Geometric_interpretation
		_theta1_rand = (int(acos( dist / totalLength ) + theta)) % 180;
		dx = dx - _distance_rand * cos( _theta1_rand );
		dy = dy - _distance_rand * sin( _theta1_rand );
		// Calculate second angle from first angle and segment
		_theta2_rand = atan2(dy, dx);
	//} else {
		// If the distance is greater than arm length, arm is straight
	//	_theta1_rand = _theta2_rand = int(theta);
	//}
}
void solve()
{
	// Compute difference between start and end points
	float dx = (targetX - _points_x[0]);
	float dy = (targetY - _points_y[0]);
	cout << "Difference of (x, y): " << "(" << dx << ", " << dy << ")" << endl;
	// Compute distance between start and end points
	float dist = sqrt(dx*dx + dy*dy);
	cout << "Distance from base to target: " << dist << endl;
	// Compute angle between start and end points
	float theta = atan2(dy,dx);
	float temp = (theta * RAD_CONVERT);
	cout << "Angle between base and target: " << temp << endl;
	// Clamp the distance
	float totalLength = _distance * 2;
	if( dist < totalLength ) {
		// Calculate first angle:
http://en.wikipedia.org/wiki/Dot_product#Geometric_interpretation
		_theta1 = acos( dist / totalLength ) + theta;
		dx = dx - _distance * cos( _theta1 );
		dy = dy - _distance * sin( _theta1 );
		// Calculate second angle from first angle and segment
		_theta2 = atan2(dy, dx);
	} else {
		// If the distance is greater than arm length, arm is straight
		_theta1 = _theta2 = int(theta);
	}


	//cout << "Angle between base and target: " << temp << endl;
/*		// Compute positions from angles
	_points[1].x = _points[0].x + Math.cos( _theta1 ) * _distance;
	_points[1].y = _points[0].y + Math.sin( _theta1 ) * _distance;
	_points[2].x = _points[1].x + Math.cos( _theta2 ) * _distance;
	_points[2].y = _points[1].y + Math.sin( _theta2 ) * _distance;
}
*/
}

int main()
{
	int xVal, yVal;

	cout << "Your initial target coords are (" << targetX << ", " << targetY <<
") in mm" << endl;
	cout << "Enter the X-coordinate (mm) for the target:" << endl;
	cin >> xVal;
	cout << "Enter the Y-coordinate (mm) for the target:" << endl;
```

```
	cin >> yVal;

	targetX = float(xVal);
	targetY = float(yVal);

	cout << "Your target coordinates are (" << xVal << ", " << yVal << ") in
mm" << endl;
	cout << "Your float target coords are (" << targetX << ", " << targetY <<
") in mm" << endl;

	solve();
	float temp1 = (_theta1 * RAD_CONVERT);
	float temp2 = (_theta2 * RAD_CONVERT);
	cout << "Theta 1 is " << temp1 << endl << "Theta 2 is " << temp2 << endl;

	// Try random angle measurement, see how many iterations until at
previously calculated best angles
	cout << "Random angle search: " << endl;
	randSearch();
	float temp1_rand = (_theta1_rand * RAD_CONVERT);
	float temp2_rand = (_theta2_rand * RAD_CONVERT);
	cout << "Theta 1 rand is " << temp1_rand << endl << "Theta 2 rand is " <<
temp2_rand << endl;

	cout << "It took " << randIter << " iterations to match calculated IK
values" << endl;
}
```

Conclusions:

In this homework I have compared the concepts of inverse kinematics and created a function to solve a 2 DOF robot arm problem. It is definitely more efficient to calculate the values of the angles directly. The calculations from Homework 1 worked much better, however they are limited to 2 DOF in the current implementation. Other methods such as Jacobian, or translation and transformation, are scalable and preferred for multiple DOFs.

References:

Braunl, Thomas, "EMBEDDED ROBOTICS. Mobile Robot Design and Applications with Embedded Systems" Springer, 2008

Luger, George, “Artificial Intelligence” Addison-Wesley, 2008

www.lorenzonuvoletta.com/tag/inverse-kinematics

Michael EngstromECE 578 – Embedded Robotics – Fall 2012 Prof. Perkowski

http://generalrobotics.org/ppp/Kinematics_final.ppt

http://freespace.virgin.net/hugo.elias/models/m_ik2.htm