Path Planning and Object Detection for a Robotic Manipulator

Alaina Herkelman

May 2, 2015

Contents

Chapter 1

Kinematics

The manipulator used was a four degree of freedom (DOF) arm with a gripper attached as the end effector. All joints were revolute and implemented by servo motors. A drawing of the arm is shown in Figure 1. To properly control the manipulator the forward and inverse kinematics were derived.

1.1 Forward Kinematics

Forward kinematics is finding the position and orientation of a manipulators end effector given the position of all of its joints. For this manipulator the end effector position was calculated in terms of the angles of the first four joints of the manipulator θ_1 - θ_4. The gripper was considered only as a length as it has no effect on the arm's overall position or orientation. An manipulator simplified as a drawing of its joints can be seen in Figure 2.

The coordinate frames of each of the joints were found using the Devavit-Hartenberg (D-H) convention. The resulting coordinate frames and DH parameters are also drawn in Figure 2. The D-H table, which contains each of the D-H parameters for calculating the transformation matrix of each joint is shown in Table 1. The matrix for each joint was then calculated using the formula in equation 1 where A_i is the transformation matrix between the coordinate frame $o_i x_i y_i z_i$ and $o_{i-1} x_{i-1} y_{i-1} z_{i-1}$ where $o_0 x_0 y_0 z_0$ is the origin coordinate frame. The transformation matrix T_4^0, to convert from the coordinate frame of the end effector to the coordinate frame of the origin is found by multiplying all of the A matrices together. So $T_4^0 = A_1 * A_2 * A_3 * A_4$. The fourth column of this matrix represents the the 3-Dimensional (3D) coordinates of the manipulator's end effector with respect to the origin coordinate frame. R_{14} is the x position R_{24} is the y position and R_{34} is the z position where R = $AF : jsdlfjkasdlfkjasldf;kj$. This column then returns the equations for x, y, and z in terms of the angles θ_1 - θ_4. As these equations are very long they were calculated in terms of these matrices each time the solution for inverse kinematics was needed.

1.2 Inverse Kinematics

Inverse kinematics of a manipulator are a way to calculated the position of each of the joints of a manipulator in terms of end effector position. The inverse kinematics of this manipulator were simple enough to be calculated analytically to formulate equations for each of the angles.

The last three angles of the manipulator θ_2, θ_3, and θ_4 all lie within the same plane as each other. This plane's location in 3D space is determined by θ_1 which only moves in the x-y plane. The project of the arm onto the x-y plane is shown in Figure 3 where x_c and y_c are the desired x and y coordinates of the end effector. From this drawing it is easy to see that θ_1 can be calculated as the inverse tangent of $\frac{(x_c)}{y_c}$ or $\theta_1 = Atan2(x_c, y_c)$ where Atan2 is the inverse tangent function taking signs into consideration.

The remaining three angles can be projected into the plane formed by the z-axis and the plane determined by θ_1 in the x-y axis. This is seen in Figure 4 where r = $\sqrt{x^2 + y^2} = l_1$ which is the length of the projected manipulator on the x-y plane. Two equations can be found for z_c and r using geometric properties on the projected image. Since only two equations can be found and there are three unknown variables the desired orientation of the end effector θ_c must be specified. The resulting three equations are shown in equation 2.

These equations were solved for θ_2, θ_3, and θ_4. These calculations are shown in Appendix A. The final results for the inverse kinematics; θ_1, θ_2, θ_3, and θ_4 in terms of x_c, y_c, z_c, andθ_c are shown in equation 4.

It also becomes helpful to have a way to calculate in the increase kinematics is possible. If the calculation of θ_3 is an imaginary number than there is no real solution to the inverse kinematics. Inverse cos returns an imaginary number when taking the inverse of a number greater than one. Therefore if the denominator is bigger than the numerator than the resulting θ_3 will be imaginary. So if $(z_c - l_4 sin(\theta_c))^2 + (r - l_4 cos(\theta_c))^2 - l_2^2 - l_3^2) < 2l_2l_3$ than the desired x,y,z coordinates are outside of the workspace of the manipulator.

The physical constraints of the manipulator must also be taken into account. Due to the way the arm is constructed the each joints has a limit of the angles it can go to. The are as follows; $-90^o \leq \theta_1 \leq 90^o$, $-50^o \leq \theta_2 \leq 120^o$, $-55^o \leq \theta_3 \leq 120^o$, $-110^o \leq \theta_4 \leq 90^o$. These constraints were found by finding the angle at which the joints cannot turn farther without colliding with another part of the arm. A sample of the manipulator workspace taken at all possible angles with an increment of 5^o is shown in Figure 5.

Chapter 2

Hardware

The manipulator and control system are set up as seen in the diagram in Figure 6. Camera data is read in and processed by the computer, the results are sent to the micro controller, which sends PWM signals to the servo motors through a power and control circuit.

The camera used was a Creative Senz 3D depth sensing and color camera. The computer code was run on Linux using bot C++ and Python. The micro controller used was a small lab built autopilot. The processor on the autopilot was a TM4C controller. The TM4C was used to control the servos. This code was originally designed on an RM48 processor but switched over to make future integration with a mobile platform robot easier. The manipulator itself is constructed from Hitec servos and the associated brackets. A list of all components can be found in Table 2.

The power/control circuit is mostly a 6-volt regulator with connections between the servos and micro controller. The layout is shown in Figure 7. The table of connections is in Table 3. An enable pin and jumper were used so that the servos only receive power when an enable signal, GPIO of 3.3v, is sent from the micro controller. The jumper can also be removed to stop powering the servos in the case power needed to be shut off quickly.

The last major design component for the hardware was the manipulator's gripper. It was built from a combination of a Hitec servo, existing brackets, and 3D printed pieces. The design concept was a fixed thumb with an opening and closing finder controlled by the servo. All parts were modeled in Solidworks and the thumb and finger pieces printed. Rubber was glued onto the 3D printed pieces to increase the friction when picking up objects. The assembly of the gripper opened and closed around a cylindrical object is shown in Figure 5. The object and similar objects were also 3D printed and used as the goal and obstacles for the arm.

Chapter 3

Image Processing

To locate objects in the manipulator workspace computer vision methods were used. The Creative Senz 3D depth sensing camera was used to gather depth and color data of the workspace. Open CV was then used to locate objects in the read images. The depth data could than be used to actually calculate the xyz position of the found image location. This was the general method for locating objects in the manipulator's workspace.

3.1 Creative Senz 3D

The Creative Senz 3D camera produces three different outputs which is sends using USB; sound, depth data, and color images. Each of these can be read in by the computer by using an interrupt system. To make the camera Linux compatible the DepthSense Software Development Kit (SDK) was used. This software was designed for use with SoftKinetics's 3D cameras but currently works with the Creative Senz 3D. The SDK for the Creative camera was not used as it is a Window's only program. Linux compatability was important for future work to be able to put code to interface with the camera could by run on Odroid which runs a Linux operating system.

3.2 Depthsense SDK

The DepthSense SDK libraries provide classes and functions for handling the camera dataa as well as sample code for reading in the data. The main function for gathering data initializes the color and depth nodes than waits for new data of each of those to be recieved. The nodes for color and depth can be configured to return the desired types of data. For example the different depth data types used in this project were the UV Map, depthMap, and VerticesFloatingPoint and the color data was ColorMap. The colorMap data stores the Red Green Blue (RGB) values for each pixel in the color image. The depthMap and verticiesFloatingPoint stores the xyz coordinates of each depth point as calculated in the SDK. The UVMap data is used to convert between the depth image and color image. This is necessary because the depth image is not taken at the exact same place on the camera as the color image and the two images are not the same size. Therefore to find the depth at a color pixel coordinate or the color of a point on the depth map on a way to convert between them is necessary.

The v member of the uv map refers to the row, or height, of the color image and the v refers to the column or width of the color image. The conversion algorithm between color pixel coordinates to the uv coordinate is shown in algorithm 1.

Reading in the color image and depth image as well as converting between locations in the two is all that is necessary from the Depthsense SDK software. This code makes up the main part of the camera code. The main function calls the functions necessary to set up the nodes and then waits for data. Functions exist called OnNewDepthSample and OnNewColorSample which run every time new data of this type is received. It is in these functions where data was read into variables to be used for further analysis.

3.3 Finding Depth Data at a Desired Location

Once an object is located in the color image the xyz coordinates must be found. To do this the depth map is searched through and using the uv mapping technique discussed above the row and column of the color image is found for each depth point. If the row and column matches the xy position of the found object than that depth position in xyz coordinates, found by using the verticiesFloatingPoint depth data, is stored as the real world position of the image coordinates. For more consistent readings of depth an average of the points around the found object was used rather than just the center point. This is because there are some holes in the depth data, where it reads a default 0,0,0 coordinate, particularly when an object starts approaching the lower limit of range on the camera. The average was a way to avoid no data being returned in these situations. The area used was $\pm$ a third of the found objects bounding circle radius. This gives enough points to create a safe average while guaranteeing the area is always within the desired object regardless of size. The found xyz position was also put through a low pass filter as the coordinates had a fair amount of noise. The filter used was a 3rd order Butterworth Filter whose equation is shown in equation 3. The derivation of this formula can be seen in Appendix 2. The algorithm for the camera coordinate to world coordinate position is shown in algorithm 2.

The final step in getting a usable xyz position is to convert from the camera coordinate frame to the base frame of the manipulator. The camera's coordinate frame is at the center point between the depth and color camera's and the manipulators coordinate frame is at the first joint of the manipulator. These frames are shown in Figure 3. The conversion between these frames is found by the equations 2-4. Where distance S is the distance between x_1 and x_0 measured along the z_0 axis and the distance x is the difference between z_0 and z_1 measured along the x_0 axis.

3.3.1 Point Cloud Library

The Point Cloud Library (PCL) is a library used for interpreting and using point clouds. Point clouds are a set of points in 3D space. These can refer to a pure depth image or also contain color information. PCL has a number of functions and examples for object recognition and tracking. While able to import the depth data from the camera into point cloud form and view using PCL's viewing functions the PCL functions for object recognition did not turn out to be reliable for this project. The alternative looked at was Open Source Computer Vision (OpenCV) which more reliably found objects with the camera data.

3.3.2 Open CV

OpenCV is another library with image processing functions which works across different computer languages and operating systems. The DepthSense SDK was written in C++ so the C++ version of Open CV functions were utilized in this project. Open CV has a number of different functions such as color detection, finding contours, shape fitting, and image filtering. Many of these functions were used for object detection in this project.

The first version of object detection was writted based on color dection. First the color image was save as a Mat, or matrix type, which is how Open CV saves images. The image was then converted from an RGB to HSV (Hue, Saturation, and Value). HSV has an advantage over RGB for color detection as it is easy to cover a range of saturations for a particular color, or hue, by only changing the saturation value rather than recalculating the RGB values for a slight change in saturation. The HSV range of the color was saved and compared to the pixels in the color image. A matrix of the same size was saved with a one if the pixel was in range and a zero if it was not. The result is a binary image, which is viewed as a black and white image, white for a value of one and black for a value of zero. This image was then put through filtering functions to remove holes and small objects caused by noise in the color image and the result is the object found in the image of that color.

The contours of the binary image were then found. The contours are the edges found in the image. Assuming only one object of the desired color is in the image all found contours were merged. The open CV function to find the minimum enclosing circle around a contour was used to return the center point and radius of the found object. An example of this code step by step is shown in Figures 1-5.

The code for this object location program is shown in Appendix C. While this code worked consistently it did have limitations. As stated above it relies on the assumption that only one object of the desired color is present in the camera image. If there is more than one the program will return a center point somewhere in between with a radius big enough to contain all objects of that color. The program also find the minimum enclosing circle of the object regardless of its shape. This means there is no way to determine the difference between a circle and any other shape in this code. It was therefore necessary to create a more well rounded algorithm for object detection.

For example to track an object of the color blue the HSV values wre found to be: H = 100 - 130, S = 100-255, V = 49-255. The original image is shown in Figure 1. The thresholded image before filtering, after finding blue, is shown in Figure 2 and after filtering in Figure 3. The found contours of the image are shown in Figure 4 and finally the minimum enclosing circle drawn onto the image along with the found center point is shown in Figure 5. This program ran fast enough to track an object in real time and calculate 3D coordinates. Viewing the results on screen has no noticeable delay.

Chapter 4

GMM Image Code

The image code written in Open CV was not just written for use with object detection for the manipulator. The first Open CV code for color detection was used in a program to calculate the ground truth, path traveled, by a mobile platform from an overhead video. The first object recognition code works for this purpose because the environment for these tests was controlled meaning the robot was the only object of the particular color in the video. The ground truth is important to have so that it can be compared to results of path reconstruction based on sensors on the robot.

For this situation a red rectangle was placed on top of the platform and video taken from above the test area using a GoPro camera. Loading the video file into the open CV program after. The video was analyzed one frame at a time to locate the robot and record the its center position. The center of the rectangle was positioned on the robot in such a way that the center of the rectangle was also the center of the robot.

Unlike with the color tracking using the depth camera the pixel coordinate of the desired location is not enough to get a estimate of the real world xyz position of the point as there is no depth data. It was therefore necessary to find another way to accurately measure the world coordinates of the robots center. The first method tried for this was using a least squares method to derive a linear equation to convert between pixel and world coordinates. The world coordinates were only calculated as an xy point as the robot traveled across a smooth surface and the z value was constant. A set of about 20 points were taken from the image at points where the world coordinates were known. Since the floor of the test area was tiled with tiles of a known size the corners were used as the known points. The conversion was setup as y = Ax+B where y is the vector of real world xy coordinates, x is the pixel uv values, and A and B were matrices found using least squares. The matrix $\tilde{A}$ which equals $[A|B]$ was solved by $YX(XX^T)^{-1}$ where XX^T is an invertible matrix and Y and X are full of known conversion points. These calculations were done to find A and B in Matlab and then put into the path finding code to convert to world coordinates.

This method had a number of problems; mainly inaccuracy and manual work. The linear system calculated did not provide a good estimate for converting to world coordinates. At times the result was eight inches from the actual xy coordinate. The desired accuracy range was to be with two inches of the actual value. Each time a new data was acquired the A and B matrices had to be recalculated. This is because the camera's position changes enough each time it was mounted above the test area to make the old matrices unusable. This meant that for each new run points would need to be found by hand to resolve for the A B matrices before being able to run the program.

To address these issues the first step was to mount the GoPro in such a way that its positioning in the test area was always the same, or at very least close enough to the same to produce a handleable error. The mount that comes with the GoPro has multiple bendable joints so even if mounted at the same location the camera is not likely to be in the same orientation. A more stationary mount for the GoPro was designed in Solidworks. The design was to have a case to hold the GoPro which attached to a clip that fit around the overhead light in the test area. The point at which the clip attached to the light was marked on the light and allowed the camera to be mounted at the same height and angle for all test runs in the environment. Figure 5 shows the Solidworks model for this mount which was then 3D printed and utilized to hold the camera.

With the camera in a fixed location a more accurate way to convert the pixel coordinates to world coordinates was developed. To account for the changes in position caused by the height of the mobile platform about thirty-five data points were gathered to convert the center of an objects pixel coordinates to the actual world coordinates. The object used for this was the same height was the robot to create the same offset. This offset can be seen in Figure 4. The bottle was placed at the corner of four tiles yet since the camera is not directly above it the object's center does not appear to be at the corner. The first version of the code did not account for this distortion and added to its inaccuracy.

An equation was made to generate the x and y world coordinates, The x-coordinate equation was linear and only dependent on the corresponding u pixel value. The y coordinate equation was a function of both the u and v pixel coordinates. This is because the lines found for the tiles were not perpendicular as the horizontal lines sloped slightly upwards making the y calculations a function of both x and y. A lookup table was generated for each tile corder in the test area based on these equations. Figure 3 shows the generated points, in black, compared to those hand gathered, in green. The lookup table could then be implemented in the main path locating program to convert to world coordinates. The algorithm for this is shown in Algorithm 1.

The results of this program were more accurate, within about 2 inches of the actual location, which was the desired accuracy. It was also no longer necessary to recalculate formulas for each new test in the environment. An example of the generated path is shown in Figure 2. There is some jitters in the found path due to the color detection not creating the ideal data. Low pass filtering does improve the path making it smother, but there is still some inaccuracy.

Chapter 5

Image Processing

The method selected for path planning was a graph search algorithm using A star (A*). The algorithm was first written and tested in Matlab and the rewritten to ne implemented on the real manipulator in python.

5.1 Graph Search

The graph search method of path planning works by finding a minimal path between a start and end position given a set of nodes. Each of these nodes represents a different state of the object the path planning is done for. For example for the manipulator in this project a node is a set of angles at a given orientation of the arm. The connection cost between each of these nodes must also be known, often calculated by distance. Nodes can either be calculated as the search algorithm is running or generated before searching. The latter becomes impractical for problems with a high number states and typically nodes are calculated while searching. Their are many different algorithms based on this method.

5.2 Dijkstra's Algorithm

Dijkstra's algorithm is a search method which is guaranteed to find the optional path. To do this it starts at the start node then looks at all of the neighbors. It saves all of the neighbors of the start node and records the cost to get there. It then looks at each of the neighbors neighbors. and calculates the cost to reach there. It repeats this process for all found nodes until the goal is found. If at any point a node is found to have a cheaper cost when reached through a different set of neighbors the new lower cost is saved as the cost to node. When the goal is found the cost to all nodes from the start to end have their costs known. The optimal path can then be found by traveling the least cost paths between the nodes from start to end. While this is guaranteed to find the least cost path between start and end this algorithm also looks at a large amount of unnecessary nodes as it looks at neighbors in all directions. This algorithm is accurate but inefficient.

5.3 Greedy Algorithm

On the opposite end of the efficiency versus accuracy spectrum is the Greedy Algorithm. It works by beginning at the start node and looking at the neighboring nodes. It then moves to the neighbor with the least cost and throws out all of the others. It keeps doing this until the end node is reached. This method creates the optimal path in the optimal amount of time if there are no obstacles and the optimal path is a straight line between the start and end node. As this is often not the case the greedy algorithm is not a practical search algorithm.

5.4 A*

The A* search algorithm is sort of a combination between the Dijkstra search algorithm and the greedy algorithm. It follows the same steps as Dijkstra's algorithm where is saves all of the neighbors of the node it is currently looking at. instead of looking at all of these node the search algorithm only looks at the node with the lowest cost each iteration until the goal is reached. Rather than the greedy algorithm A* does not throw away the other nodes so it can keep looking for the least cost. Rather than get stuck on obstacles A* can calculate a path around them. The total cost between nodes can be an equation tuned to work for the specific situation it is being used for. The A* algorithm provides a balance between accuracy and time as it only looks at nodes it views as potential candidates for reaching the goal quickly. If tuned correctly A* will return the optimal path.

5.5 Path Planning for the Manipulator

The first step of the path planning code was to find a way to discretize the workspace of the manipulator. The angle apace, $\theta_1, \theta_2, \theta_3$, and θ_4, rather than the alternative of using the x,y,z,θ_c position coordinates. This choice was made so that each part of the found path would be guaranteed to be in the workspace of the manipulator. By changing the step between the manipulators rather than the position the trajectory will be smooth since each angle will only be changed at most by a step size. Using the xyz position can results in large jumps between the angles to reach a small change in end effector position because the configuration changes between them. To calculate the neighbors of the current node the angles are by a maximum of a set step size.

The obstacles were entered into the path planning algorithm by keeping track of the xyz position of the obstacle. No further definition was necessary. Rather than calculate the arm nodes which would cause a collision with the obstacle the position of it was used as a parameter of the total cost function for each node. The closer the position of the end effector to the obstacle the higher the cost to go to that position. This acts as a potential field pushing the arm away from the obstacles. With proper tuning the arm is able to avoid the obstacles while traveling to the goal. The obstacle cost was taken as 1/distance between the end effector position and the obstacle center. This cost was found and summed for each obstacle, multiplied by a tuning parameter and added to the total cost.

The A* algorithm was implemented to calculated the path of the arm. The total cost of each node was calculated by A* distance between start node and current node + B*distance between current node and goal node + C* 1/sum of distance between current node and each obstacle where A, B, and C are wights adjusted for best results. The algorithm for this is shown in algorithm 1. It calculates the optimal path between a start position and end position while avoiding all inputted obstacles. Since the goal of this program is to pick up an object the path had to be separated into two different pieces. The first considered the goal object as an obstacle to avoid colliding with it before picking it up and traveling to a point a about two inches away from the goal. At this point the gripper was opened and a new path calculated from the current position to the goal where the gripper was then closed and the object picked up.

Chapter 6

Modeling

All models of the robotic manipulator were Matlab based. The first models were made simply as tests for the kinematics calculations. Given a set of angles the manipulator was drawn as a line between each of the joint positions. Each of these positions is calculated using the A matrices calculated for the joints as described in the Kinematic's section. So the first joint is calculated by the last column of A. Location of the second joint is located at A_1A_2's fourth column and so on. The origin, (0,0,0) is selected as the origin of the manipulator, the first joint. The result of this is shown in Figure 1. With this setup it was possible to test the actions of the manipulator based on different functions.

The inverse kinematics was tested using this model before testing on the real manipulator. The inverse kinematics equations along with the joint limits were programmed into a matlab function. The program could draw the resulting arm configuration for any desired end effector position in the workspace. For cases was not solvable the arm simply did not move, this was determined using the inverse kinematics feasibility test described in Chapter 2. The physical constraints were also programmed into the code to stop the arm for traveling outside of the physical angle limitations. Testing this code confirmed that the workspace of the manipulator is convex. This means that from one existing point in the workspace to other point in the workspace then the line between them is also in the workspace. This fact was use to create a trajectory from a starting position to a desired end position by calculating the linear path between them.

The next model was made to test the path planning code. In matlab obstacles were drawn on the 3D graph with the centers at the inputted obstacle value. The path planning algorithm then computed the path for the manipulator which was drawn through each configuration in the path. An example is shown in Figure 2. This model showed the algorithm to be able to consistently make it to the desired obstacle location with a smooth trajectory. The matlab simulation did not include any modeling of the gripper simply showing the path to the obstacle.

A more realistic model of the manipulator was created in the Virtual Robo Experimentation Platform (V-REP). This software includes physics engines which could more accurately simulated the manipulator at he goal objects reactions. First the servo and brackets 3D models were downloaded and the arm was assembled in Solidworks to create an accurate model of the manipulator. The 3D model was imported into V-REP and each servo set as a joint. Obstacles were created a similar size, shape, and weight as the real obstacles 3D printed for this project. The same matlab code was used to control the joint angles in V-REP. The simulation was then run and used to pick up objects.

The V-REP simulation was not as reliable as the matlab simulation and a few code changes needed to be made to avoid the open gripper hitting the obstacles. There were also some problems with the objects flying out of the gripper. This seemed to mostly be a result of the physics engine and after a few adjustments to the object materials the simulation successfully picked up the object. An image of this simulation is shown in Figure 4. The results of this simulation made it seem as though the path planning algorithm was likely to work with the real manipulator but would likely need some tuning to successfully grab objects.