Stephen Mylabathula
Zach Chavis
Luze Yang
Joshua Palm

Connect the Dots with Baxter

ROAD TO ROBOTICS

The focus of our project was to enable the Baxter robot to detect drawn dots on a surface, and then with its arm connect each dot together with a marker such that all dots are connected. One reason we chose this project was for its employment of critical course concepts, including understanding the ROS environment, inverse kinematics, conversion between coordinate frames, gathering data from sensors, and motion planning. Another was the applications toward real-world robotics. A possible application of this project is to manipulate a robotic arm in an assembly line which inspects parts using computer vision and, upon finding defects on the surface, marks them for further human inspection. A robotic arm can also be used to finely print a circuit board, using some form of vision to maintain high accuracy of the product. A final application is in artistic rendering, in which a robotic arm and vision system can create or recreate a work of art while retaining stroke and texture information normally lost when printing art [2].

Our initial outline of the project was to have the Baxter robot use some form of computer vision strategy to identify dots we drew on a piece of paper. Upon locating all dots, the Baxter robot would grip the pen, and draw from dot to dot. Although modest, this outline would prove to have many difficulties in implementation.

We began our project by studying ROS together for a couple weeks, developing a basic understanding of nodes, topics, messages, publishing and subscribing, and programming in C++. Once we felt comfortable with the basics, we were all eager to jump into programming on an actual robot. Following Rethink's tutorials, we set up a computer to communicate with Baxter and set up the environment to run ROS code on the machine.

The first major roadblock on the project came from our unfamiliarity with ROS and MoveIt. We followed online tutorials for getting MoveIt (an inverse kinematics library) to work on Baxter, using the RViz utility to visualize the robot and get a basic idea of the capabilities of Baxter. This involved setting a start and goal position, planning a path between the two, and executing the path. Joint positions were being read by Rviz as the Baxter robot was visualized correctly on the display, even reacting to how we moved the joints by hand in real time, but we couldn't get the robot to plan. The robot turned out to be sensitive to initial parameters, and the solution to our planning problem involved letting the program know if each of Baxter's limbs had a gripper on the end effector. However, we immediately ran into another hurdle. When trying to execute this path, we were receiving a timeout error, which stated that there was a problem in which the joint trajectories may or may not have been published. We spent an excruciating two

weeks or group meetings trying to resolve this issue. There were leads suggesting this was a problem with the joints being published to a dead connection, so we investigated the complex rqt graph that ROS generated, only to conclude that we didn't think there was a problem with the joint trajectory server node. We investigated other demos which seemed to work fine, but anything involving MoveIt IK seemed to break. We couldn't have had any less of an idea about the problem or how to go about solving it, as every tutorial we found seemed to have RViz working on Baxter out of the box. It was then brought to our attention that the timeout error was due to a severe desynchronization between the clock on the system in which we planned, and the system clock on the Baxter unit. Upon syncing the clocks together, the gears began to turn, literally, and Baxter executed all planned paths flawlessly via Rviz.

Our next course of action was to test the visual capabilities of the robot. We used some ROS services to enable the cameras, and to display a live feed of the camera onto the screen. We had a rough idea of what the Baxter robot was capable of seeing, and moved into analyzing still images for dots offline. Since we had no prior computer vision experience, we relied on tutorials to develop a visual strategy. The idea was that if we could find the centers of all shapes in the image, we could easily have the center points of each dot. Our first method involved a contour detection strategy, in which the image ran through a low-pass filter, a canny edge detection filter, a contour detector, and code to analyze the contours and return their centers. Unfortunately, when testing this strategy on a live camera feed, the noise from the camera and the complexity of the scene made this method seem highly impractical and specific.

With some basic kinematic control and computer vision in place, we pushed on to refining these methods and piecing things together into a single program. Originally all of our code was in C++, but we couldn't figure out how to programmatically employ the MoveIt libraries in ROS. After much frustration, we decided to give Python a try, as it was easier to rapidly run Python commands until something worked. We began to follow a MoveIt commander tutorial, using Python utilities to try and plan paths for the robot. Some basic movement was possible right away, by moving just one joint at a time manually through the command line. But when trying to get a more complex motion model working, frustration ensued, as joint messages were broken, publishing was not working, and no paths could be planned. Again, we spent much time learning to use ROS debug tools and analyzing the rqt graphs, until finally we noticed that critical nodes were not subscribed to the topics we were publishing. Once this problem was patched, we could finally use the forward kinematics of Baxter programmatically. The inverse problem, however, was still an issue we couldn't resolve.

Soon, our group found a Baxter-specific Python implementation of Inverse Kinematics, completely set up and easy to use. This proved very helpful, and we felt like we were finally back to making progress. We experimented with picking up objects on a table (without computer vision), and getting a feel for the Baxter robot's coordinate system. Our computer vision strategy, regardless of its state of functioning, was to return a point in an image taken by the camera. We began to work out on paper how to do the visual servoing with Baxter, transforming a point in an image into Baxter's coordinate frame. It was at this point when Professor Sattar

brought a webpage to our attention, hosted on Rethink, involving a worked example of the visual servoing [1]. The example involved locating golf balls with Baxter's camera, locating a tray for the balls, and picking and placing each ball in the tray. The Python code was very readable and easy to work with, and the complex portions of the code were explained on the webpage. We concluded that some simple reworking of these tools could lead us to the results we wanted for our project. We finally had a working piece of Python code which could enable the cameras, take pictures, move the manipulators, and transform between the camera and Baxter, all in one place.

The worked example used a hough circle computer vision strategy, which we figured we could leverage to locate dots as well. After tuning the parameters, we were finding all sorts of circles via Baxter's camera. However, when trying to locate drawn dots, we realized the hough circle function was great at perfect circles, but subpar with imperfect ones. We also tried to give Baxter a pen and have it draw on a surface, and immediately realized two very big problems: Baxter's motion plan and accuracy were not predictable, and the writing utensil created friction which affected the completion of the plan. Because of these two problems, if Baxter's plan from one point to another went too low, the friction of the pen on the paper would cause Baxter to halt, and the plan would not complete. Our immediate partial solution to this was to reduce the friction of the utensil. A whiteboard was brought in, and whiteboard markers were used. This worked much better than previous equipment, but still negatively affected the connecting of the dots due to the unpredictable planning.

A new computer vision strategy was again needed, as dots were not being located by Baxter. Upon some reading, we found the blob detection algorithm in OpenCV. This algorithm finds like collections of pixels and groups them together as one object. This worked almost perfectly, and Baxter was able to locate drawn dots. However, since we switched to the whiteboard, Baxter had difficulty locating dots with the glare of the ceiling lights on the whiteboard. Our only way to overcome this was to move the whiteboard to an area free of glare, even resorting to lighting the board ourselves. This problem is material specific, and had we gotten paper to work, we wouldn't have run into this issue.

There was still the issue of Baxter's motion plan. We spent some time trying to figure out how to set limitations on the plan, but did not receive much help. Without restricting the plan, we had little means of programatically solving this issue, other than having many small dots clustered, which is difficult due to Baxter's accuracy, or interpolating imaginary dots between the points, which is an issue since we have to pause between each dot. Our group came up with a brilliant strategy: to fabricate a marker holster that allowed the marker to give, while maintaining contact with the board if Baxter moved slightly up or down in a plan. The holster was 3D printed, and designed to fit a standard whiteboard marker (see image 1). A spring was placed into the holster behind the marker, and given to Baxter. We ran some tests of drawing lines across the whiteboard, and things worked very well. The only parameters we had to play with now were drawing height and spring tension.

Image 1: 3D-printed marker holster

The project was nearing completion. There were still a few issues that remained. Taking an image with the camera that had the grippers attached meant that the grippers slightly obstructed the view of the dots. This was solved by using Baxter's left limb (which was empty) to take the picture, while the right limb did all the drawing. Another issue was that even though we reduced the friction of the marker considerably, if the motion was along the axis perpendicular to that which the grippers spanned, the marker would shift in the grippers, and a successful drawing was no longer guaranteed. Our quick fix to this problem was to sort the dots based on their Y location in the image. Since the grippers opened along the X axis in image space, more lines close to parallel with the X axis were drawn, and the marker was rarely affected. But this can easily fail, if all the dots were located in a line along the Y axis. To fix this, we had to rotate Baxter's cuff. By rotating Baxter's cuff based on the direction of the next dot, the line could always be drawn parallel to the span of the grippers, and the marker wouldn't slide out of the grippers. We also tried to dynamically assign the height of the table via the IR sensor, but the reading was inaccurate and usually caused the marker to be too low, forcing us to use a hardcoded value for the table.

CODE OVERVIEW

The final code for this project was written in Python, implementing Numpy, OpenCV, and various IK packages including MoveIt and Baxter services. The code is an open feedback system divided into four main parts: setup, image processing, coordinate frame transformation, and IK. The code is divided into three main parts: the RobotMotion class, the RobotVision class, and the main method. The RobotMotion class is used for controlling Baxter's movements, as well as using the IR sensor for grabbing the distance from Baxter's end effector to the surface of the table. The methods in the RobotVision class involve all camera control and vision processing, including opening the cameras and locating the dots.

ALGORITHM OVERVIEW

The first part is the initial setup. The right and left limb cameras are enabled, and while the right limb is moved to the side and the grippers are calibrated, Baxter's left limb is moved over the board and pointed straight down. The IR sensor is used to grab the distance from the initial position to the table, which is later used to transform the positions of each dot into Baxter's frame. If the IR sensor doesn't detect the whiteboard on the table due to it being out of range, the algorithm defaults to the known height of the table. We found that the IR sensor data was somewhat unreliable, and sometimes causes Baxter to push the marker too hard into the whiteboard (as seen in our project demo).

The camera feed begins, and blob detection is run on each frame until the specified number of dots is found. Each dot's x and y coordinate in the image is saved into an array in whatever order the blob algorithm returned them in.

Each dot is then pre-processed. They are sorted according to the Y location in the image, and then they are transformed from image space to Baxter's coordinates. To do this, a seemingly simple equation is used. Dots are transformed to Baxter's coordinates according to this equation:

$$B = (Pp - Cp) * cc * d + Bp \quad [1]$$

Where B is Baxter's coordinates, Pp is the pixels coordinates, Cp is the center pixel coordinates, Bp is Baxter's pose, cc is the camera calibration factor, and d is the distance from the end effector to the whiteboard. Pp-Cp gives the pixel coordinates relative to the center of the image. The image can be thought of as a plane that exists some distance away from the origin of the camera projection, and the actual point in space lies on a line between the center of projection and through the pixel on the projection plane. Using cc and d, that plane can be shifted such that each pixel represents a more familiar metric, which ends up being cc = 2.5 millimeters per pixel at one meter. We finally add this to Baxter's current pose to obtain where the coordinate exists in Baxter's frame. This method is limited, as it assumes that every point in image space is planar.

Finally, when going from dot to dot, we must account for two things: the offset of the grippers to the camera, and the rotation of the cuff from the current position to the next dot. By adding an offset to the transformed point in Baxter's frame, we can account for the grippers not being in the center of the image, thus making the each point now represent the location in Baxter's frame in which the grippers must go. Then, the rotation between each point must be calculated, which ends up being a simple atan2 of the change in Y and the change in X of the current pose to the next pose. When rotating the cuff, we also need to rotate the offset, which is a simple matrix transformation of the offset around Z. Finally, we assume that X is to the right and Y is out; however, in Baxter's frame, X is out and Y is to the left, meaning we must add 90 degrees to the final angle. The IK method in RobotMotion is called, passing in the pose of the next dot after processing, and the motion plan moves Baxter's manipulator (and the marker) from one dot to another.

RESULTS

With all these key features in place, the Baxter robot was successfully able to locate and connect a given number of dots most of the time. We still had issues with the motion plan in the end, and sometimes the IK solver thinks the best plan of action for a simple cuff rotation involves bending the entire arm, which destroys the connected graph. There are still some computer vision issues as well, such as glare, dim lighting, and rarely, false positives. However, we are very pleased with what we accomplished, and feel we have achieved what we set out to do earlier in the semester.

FUTURE WORK

There are many possible extensions to this project if we, or future groups, were to move forward with the progress we've made. There could be more investigation into preventing the motion plan from doing anything weird, such as curving the lines while drawing between the dots. Force feedback could be implemented to keep the pen from being pushed down, eliminating the need of the spring-loaded marker holster. A closed-loop system could be implemented in which some form of sensing helps keep the marker on the correct path, and improves the accuracy of connecting the dots while Baxter draws. The number of shapes could be stochastically determined without prior knowledge of how many dots there are. An order to the dots could be implemented, or there could be multiple systems of dots on the board, perhaps differentiated by shape or color. The computer vision could be further improved to allow for object recognition of things other than dots, perhaps a brush and an inkwell. And lastly, a very interesting idea we came up with for expansion of the project, is to have Baxter drawing the dots with one hand, while simultaneously solving the problem with the other, avoiding any collisions between each arm.

CONCLUSION

Our goal was to implement algorithms on an actual robot to accomplish the task of connecting the dots. We believed this project was the culmination of all topics discussed in the course, and that it would be a very cool project to demonstrate and expand upon in the future. There were many challenges and learning curves we had to overcome, including the ROS environment, computer vision, and the (un)reliability of sensors for vision and motion estimation. However, we believe this project was an invaluable learning experience for robot development, as challenges like these are unavoidable in the field or robotics.

BIBLIOGRAPHY

[1] Pugh, Vyvyan, and Horatio Coles-Abell. “Visual Servoing.” *Worked Example: Visual Servoing*, Active Robots Ltd, 2014, sdk.rethinkrobotics.com/wiki/Worked_Example_Visual_Servoing.

[2] Tresset, Patrick, and Frederic Fol Leymarie. “Portrait Drawing by Paul the Robot.”*Computers & Graphics*, vol. 37, no. 5, 2013, pp. 348–363., doc.gold.ac.uk/~ma701pt/patricktresset/wp-content/uploads/2015/03/computerandgraphicstresset.pdf.