PixelBot: Robot-Generated Light Vector Drawings

UC Berkeley EECS149 Fall 2014 Final Project
John Wilkey, Alec Guertin, Chester Chu

1 Introduction

The goal of the PixelBot project is to design and implement a robotic system that can take a graphic from user input and replicate a scaled version of the drawing on a flat surface. The current implementation accomplishes this task through the use of a Pololu m3pi (see Figure 1) with an onboard LED that flashes on and off as the robot moves to different positions. The user can use a camera with a long exposure to capture the sequence of flashes and produce a light drawing. The following sections explore the modeling, design and analysis of this system.

2 Modeling and Design

2.1 Basic Modeling and Platform

The initial model for our system included 3 main sections: user input, frame detection and drawing. In the user input stage, the goal was to allow the user to draw a picture in a computer application. The data from this picture would then be used as the input to our robotic system. In the frame detection stage, the system would use sensors to detect a frame for the drawing as defined by the user. Finally, in the drawing stage, the system would draw the input scaled to a size proportional to the detected frame. Correct behavior of the system would be defined as accurately recreating the user's drawing in the framed area.
We chose to implement our system using the Pololu m3pi, a small robot with an Atmel Atmega AVR microcontroller receiving serial commands from an MBED LPC1768 acting as a high level controller. The m3pi is a small circular robot, approximately 8.8 cm in diameter, with two wheels controlled by motors and a caster wheel. The m3pi is also mounted with an array of five reflective sensors, making it ideal for an out-of-the-box system for frame detection.
After choosing this platform, we were able to expand our model to include more detail. In our new model, we developed a state machine (see Figure 9) to govern the m3pi's behavior. This state machine represents the second and third stages of our three-stage model. We were also able to make design decisions on how the m3pi would interact with its sensors and actuators to accomplish these tasks.

2.2 Design Overview

The first step in the process is interpreting user input. We implemented the user interface for drawing using Turtle graphics in Python. A portion of the code for this application is adapted from `http://svn.python.org/projects/python/trunk/Demo/tkinter/guido/paint.py`. This application allows the user to make basic drawings with their computer mouse. We then take the PostScript data from this drawing and load it onto the m3pi. The PostScript data includes the dimensions of the canvas and a series of "moveto" and "lineto" instructions. Each "moveto" line specifies to draw a line from the current coordinates on the canvas to the new x- and y- coordinates specified (assuming a grid with the origin at the bottom left corner of the frame). A "lineto" instruction specifies to draw a line from the current coordinates to the new coordinates. We chose to use PostScript since it offered a convenient way to represent directions for drawing vectors with little overhead. For our purposes, we only support the most basic "moveto", "lineto" and dimensions instructions.
The m3pi's first task is to scale the drawing from the dimensions of the user drawing to a frame on the drawing surface. We utilized the m3pi's onboard reflective sensors to detect the edge of a frame marked with non-reflective electrical tape on a reflective surface. We use these sensors to locate the corners of the frame and measure the distance between them. From this measurement we can calculate the necessary scaling factor for the drawing. The m3pi API provides methods for calibrating the reflective sensors to the tape line and tracking its position relative to the robot (left or right) as well as controlling the wheel motors.
After establishing the size and location of the frame, the m3pi must draw the user's graphic. In our design, we use the PostScript instructions to specify the robot's movements and actions accordingly, following the same movements as the user's mouse when drawing.

2.3 State Machine and Algorithms

The robot's behavior is modeled by a state machine that expands the ideas from the second and third stages of the basic model above. The states and transitions of this FSM include:

- LINE_CAL: LINE_CAL is the initial state for the robot. Here, we assume that the robot starts on the line and facing the bottom right corner of the frame. In this state the robot sets up state variables and calibrates sensors to the frame's tape line. This state should only be reached once at the beginning of the program and transition to the FIND_CORNER state under normal conditions. If the robot cannot cali-

brate the sensors to find a line, the FSM transitions to the ERROR state.

- FIND_CORNER: This state specifies that the robot should follow the frame until finding a corner. The robot drives forward and corrects its movements to stay over the line. A boolean state variable specifies whether to search for the bottom left or right corner of the frame. The robot exits this state after the reflective sensors detect the appropriate corner or can no longer detect the line. The first time this state is run, the robot looks for the bottom right corner and the second time looks for the bottom left corner and measures how long it travels in order to determine the frame width. This state transitions to the REV_TURN state after successfully detecting a corner. The robot transitions to the ERROR state if the sensors can no longer detect the frame.

- REV_TURN: Here, the robot turns 180 degrees, pivoting on the frame line to face opposite corner. The state machine transitions back to FIND_CORNER if only one frame corner has been found, or DRAW_INIT if both corners have been found.

- DRAW_INIT: In this state, the robot prepares for the drawing phase of the model. The m3pi reads in the PostScript file, prepares state variables including the scaling factor and localizes itself at point (0,0) and an angle of 0 degrees in the corner of the frame. This state should only be run once after finding both bottom corners of the frame. This state transitions to the PARSE state under normal conditions and the ERROR state if an error is detected when opening the PostScript instruction file.

- PARSE: This state is for interpreting the next instruction for the robot. In this state, the robot reads in more data from the file and sets state variables according to the next unparsed PostScript instruction. After parsing, this state transitions into the TURN state upon reading a valid instruction, the FINISHED state if there are no more instructions, or the ERROR state if there is a parsing error.

- TURN: At this point the robot calculates the angle necessary to turn to face the next coordinate specified in the PARSE state. We can calculate this angle using the state variables for the robots current coordinates, desired coordinates and net angle turned since DRAW_INIT (an angle of 0 degrees specifying the vector parallel to the x-axis). The robot then turns the remaining angle necessary. This state branches to the MOVE or DRAW state based on the "moveto" or "lineto" instruction received from parsing.

- MOVE: This state calculates the Euclidean distance between the m3pi's current position and its desired position and then runs the motors to move that distance. This state should always follow the TURN state and transition to the PARSE state.

- DRAW: This state operates similarly to the MOVE state, but also draws as it moves (enables the LED).

- FINISH: In FINISH, the m3pi stops and alerts the user that it is finished drawing with the onscreen message "Done". This state is only entered when no more PostScript instructions remain. With proper behavior, the m3pi remains in this state until the user deactivates the robot.

- ERROR: This state alerts the user that an error occurred by printing a message to the robot's screen with the state that the error occurred in. The ERROR state can be reached when another state cannot properly execute its logic or detects an inability to follow correct defined behavior. Some examples of situations that lead to this state include: an inability to detect the tape line in the FIND_CORNER state, a parsing error in the PARSE state or encountering a bad file descriptor in the DRAW_INIT state.

Our final implementation of the code for this system is designed using this state machine logic. The algorithm runs a while loop that in each iteration executes the logic for the current state and then updates the state based on state variables. For more details, please see the code repository in the "Resource Links" section.

3 Error and Analysis

3.1 Physical vs. Model

There are several sources of error with the physical robot that causes its behavior to deviate from that of our model. We therefore must calibrate the robot to account for its inherent inaccuracies and biases. The functions that control the robot's movements can be treated as an affine model where we compensate for movement biases by calibrating the robot along a series of known paths and correct for these biases. Moreover, we can measure the sensitivity factor empirically to determine what factors to multiply our input angle by in our turn commands to convert the m3pi's turn API calls into turns by angle and calls to our forward() and backward() functions into measures of distance, something not available in the m3pi API. To that end, we determined the sensitivity metric as our first calibration, using a square box and repeatedly circling it, the intent being to both calibrate for consistent 90 degree turns and to gain an understanding of the rate at which buildup error would become problematic. In order to properly calibrate this model, however, we had to first identify the error and then instrument that error.

3.2 Motor Precision and Weight Distribution

A principle source of error in our robot, as we came to realize, was the robot's reliance on reliable realtime behavior to control its motions. Specifically, the m3pi APIs motor control functions are predicated on being able to precisely control the duration of the motor's spinning. It was found, however, that there are inherent inconsistencies in our chosen platform with regard to these functions that likely stem from inconsistencies in timing from one moment to the next on the RTOS running on the MBED.

One major source of error in our physical implementation was inaccuracy caused by motor control and weight distribution. We witnessed slight discrepancies between the robot's movements and the intended behavior as specified by our model. The robot's turn angles were often inaccurate and the m3pi drifted to the right when driving forward. These two inconsistencies, resulted in problems with build-up error. After several instructions, the small error produced with each movement would compound, and the robot would end up a significant distance away from its intended location.

We discovered that one source of right bias in straight driving was the uneven distribution of the robot's weight. The mbed controller, which is offset toward the right side of the robot, was enough to throw off the weight of the robot and cause it to veer slightly right when it should have been driving straight. How such a small difference in weight could cause a noticeable bias was something that was unfortunately unexpected. While we were able to compensate for it by introducing a reverse correcting angle at periodic intervals, it was a less-than-ideal solution as our model had not considered that tracking straight would not actually go straight.

Next, the precision and consistency of the robot's motors provided another difficulty. We found that the m3pi often did not turn the full angle specified. What's more, the degree to which it deviated from its intended angle differed inconsistently with the specific angle it was asked to turn. This made the affine model of inaccuracy more difficult to apply since there was no longer a constant bias term but one that was itself a function of the input.

To mitigate these concerns, we modeled the error by measuring the error in our instructions and calibrating our robot to counteract it. We instructed the robot to turn 45, 90 and 180 degrees and measured the angles across multiple trials. We then calculated average deviation between the m3pi's intended turn angle and the actual angle turned. From these results, we could see that much of the error in turning was from a bias that was consistent between the trials for all three angles. We used this calibration to correct the turn instruction in our code and make the angles more accurate. The results of our trials can be found in the graphs in Figure 2, Figure 3, Figure 4. Depending on the angle, we add a small correcting factor to counteract the bias term while leaving the sensitivity term unchanged.

Next, we decided to rectify the issues with forward motion introduced by the robot's uneven weight distribution. We positioned the robot on a grid and instructed it to drive straight for a specified distance. Using the average distance it deviated from a straight line, the trial data of which can be found in Figure 5 and Figure 6, we modified the forward function to correct for this movement bias by overdriving the right wheel and inducing a compensating turn at 1 second intervals. The values for these compensations was determined empirically.

Finally, the above tests were performed on two candidate surfaces: paper and linoleum tiles. We noted that the tile surface consistently performed better in both absolute accuracy and test/retest consistency. This led to us opting to use tile as our surface.

3.3 Rocking and Sliding

Another major source of error that was realized was slippage caused by abrupt changes in velocity and/or direction. When attempting to drive short distances or make sudden stops, the robot would often rock forward, causing its wheels to noticeably slip. Through empirical measurements, it was determined that the robot could not reproduce arcs in succession whose endpoints were within 20 'pixels' of each other. To this end, we employed a smoothing filter on our coordinate generating program to filter these jagged lines into a smoother shape. The smoothing algorithm looks at the input instructions and examines points associated with draw instructions in groups of three. The algorithm will calculate the length of the line between point 1 and 2, and then the length of the line between 2 and 3. Then, based on the length of these lines and the angle between them, the algorithm will determine whether to replace the two lines with a single line between points 1 and 3. The net result of this smoothing is less jerking motion which translates to less wheel slippage and allows the robot to maintain a more accurate sense of orientation. This, unfortunately reduces overall resolution in the drawing, but greatly improved the robot's accuracy and so was deemed a reasonable tradeoff.

3.4 Mitigations That Didn't Work

There were a number of mitigation techniques we employed that failed to reduce the robot's inaccuracy. We initially tried to compensate for the forward drift by adding ballast to balance the robot's weight distribution. This, however, proved less effective as we learned that the additional weight–while having addressed the drift–led to less consistency between trials. This, also, led us to realize that the additional weight and actuating force generated by the solenoid–as our original design had called for–would induce too much inaccuracy in the robot's movements to produce meaningful results, and was primary reason for us abandoning that idea. Next, we attempted to deal with compounding buildup error by having the robot relocalize to the origin after each line segment. This, also, proved

ineffective and led to worse results overall. It was found that the movements needed to return to the origin and back were themselves a source of error that served to only make the drawings worse.

4 Final Results and Next Steps

We were able to produce several images with reasonable accuracy. For examples of images drawn by the robot, see Figures 7 and 8. Unfortunately, despite our added mitigations to the errors we found, the robot still shows some build-up error. As a result the robot is limited in the complexity of the drawings that it can reproduce. However, we have explored modifications to our design that may improve the robot's accuracy and the overall user experience.

A principle problem we encountered during development and one that we were unable to completely mitigate was the problem of buildup error and motor inconsistencies. Specifically, we were never able to address the occasional turn that would end up completely inaccurate, nor were we able to compensate entirely for surface irregularities and voltage and temperature changes in the robot. In the end, it became clear that this project would benefit tremendously from a way to allow the robot to receive constant feedback of its absolute position from known fixed points. To that end, we believe the most prudent next step in this project would be implementing a form of closed loop control, such as infrared guidance beacons similar to what was demonstrated in one of the first lectures of this class with the flying robots. It has become clear to us that without a way of being able to update the robot of its position, relying on the robot's motor motions exclusively will never be accurate enough to avoid build up error.

Next, for this project, we initially wished to create a robot that could draw images using a pen controlled by a solenoid. However, after discovering the great effect that changes in weight had on accuracy, we decided to eliminate the mechanical component and have the robot draw using an LED. We would hope to spend more time calibrating our system to work with physical drawings. However, due to the limitations of the hardware, this will most likely require building on another robot platform.

We also have the future goal of making the robot interact with the user in real time. The user drawing application could update the robot via WiFi/bluetooth with new instructions as the user continues to draw. In this case, we could add a STANDBY state to our state machine. The robot would return from this state to the PARSE state upon receiving new input and continue drawing rather than going to the FINISH state. When coupled with adding the solenoid, a user could use the robot to play a game of tic-tac-toe with chalk on the sidewalk or draw a picture in stages.

5 Conclusion

This project afforded us an opportunity to apply our understanding of modeling a system with finite state machines and translating that model into a physical system. This project proved quite rewarding despite a number of obstacles we faced in our implementation related to the inherent differences between the physical system and our model. Paramount in these obstacles was our need for precision control of the robot which required an ability to precisely control timing of the system and account for noise from both the environment and the robot itself. To this end, we turned to modeling noise by first identifying where noise was manifesting itself then measuring it and calibrating our robot's motions to counteract it. In this way, our initial design used a modal FSM model that controls the high-level logic of our project while the more detailed calibrations and other nuances induced by the physical system were modeled as an affine model using empirically-determined values and compensating for inaccuracies and biases. Taken together, this technique allowed us use a rather inaccurate and inconsistent robot to make relatively precision patterns. Nevertheless, the inconsistencies that remained in the final project allowed us to appreciate the difficulties in reliable realtime behavior as a large portion of the innacuracies that remain are almost surely timing related as the m3pi's API is largely dependent on timing. If expanded to include a form of closed-loop guidance from a fixed frame of reference as alluded to in our "Next Steps" section, the resulting drawings can undoubtedly be made far more accurate.

6 Resource Links

m3pi Line Follower Starter Code:
`http://developer.mbed.org/cookbook/m3pi-LineFollowing`
Python Drawing Program Starter Code:
`http://svn.python.org/projects/python/trunk/Demo/tkinter/guido/paint.py`
m3pi API:
`http://developer.mbed.org/cookbook/m3pi`
Code Repositories:
`https://bitbucket.org/ee149tabot/final-project`
`https://developer.mbed.org/teams/EE149/code/FinalProject/`
Video Demo:
`https://www.youtube.com/watch?v=A5haKARs_QE&feature=youtu.be`

7 Appendix

Figures and graphs referenced in earlier sections.

Figure 1: Hardware: m3pi & MBED LPC1768

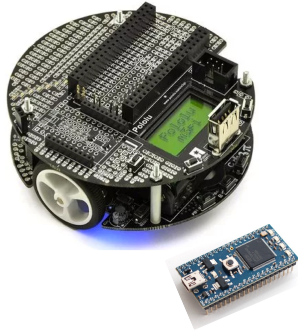

Figure 2: 180 Degree Error

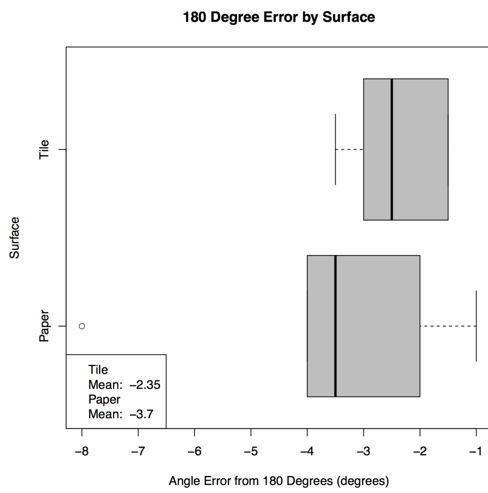

Figure 3: 90 Degree Error

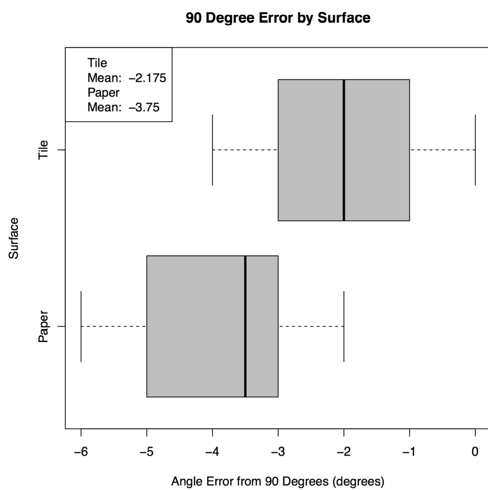

Figure 4: 45 Degree Error

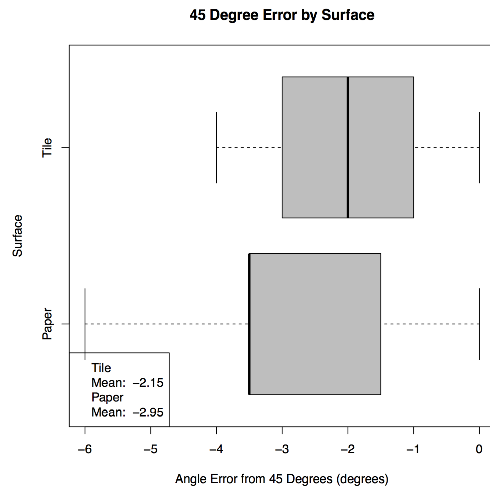

Figure 5: 1 Second Drive Error

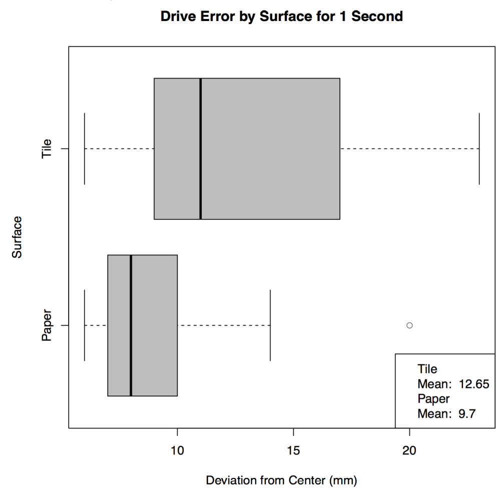

Figure 6: 3 Second Drive Error

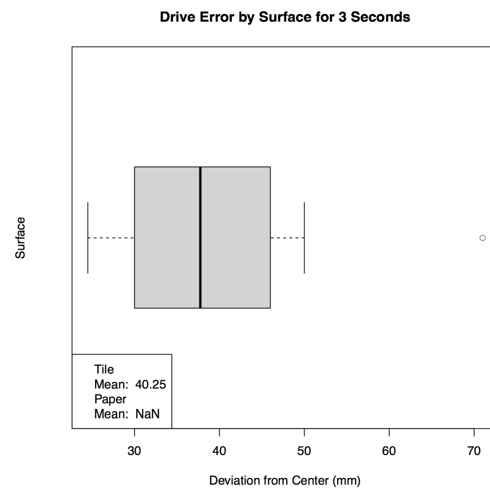

Figure 7: Demo: 'Cal' logo light drawing

Figure 8: Demo: Smiley face

Figure 10: Calibration Model

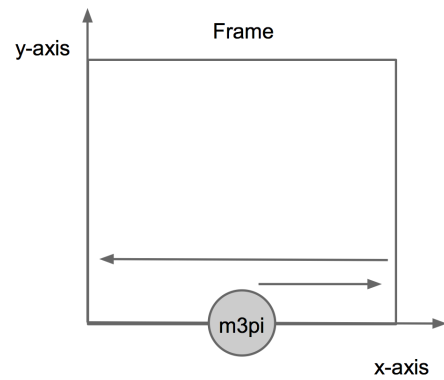

Figure 9: FSM Model of Physical System