ME 135 Final Report

TEAM: High CaliBEAR

MEMBERS: Eric Zhang, Nick Entress, Eric Eom, Connie Yu

PROJECT: Motorized Etch-a-Sketch

Album of more pictures and videos

Background

A classic toy from the 1960s, the etch-a-sketch is essentially a two-dimensional plotter with two knobs that control horizontal and vertical movements of the magnetic stylus. Creating perfect masterpieces would normally require excellent motor skills, but the motorized etch-a-sketch (affectionately named the Sketchy Etcher) created by Team High CaliBEAR can provide all the precision you've ever wanted with none of the effort - at least aside from 10 weeks of programming difficulties on the development side.

This motorized etch-a-sketch has an accompanying GUI that allows the user to draw with a mouse (or use fingers on a touchscreen). Downstream processing in the GUI reads this user-controlled input and translates it onto the etch-a-sketch screen, using two stepper motors to replace the knobs turning the drawing shafts. To limit user error, the GUI ceases to record additional data from the user if the mouse moves too quickly or out of the etch-a-sketch screen boundaries. Returning to the tip of the drawing path (and apologizing for not following directions) allows the user to continue from where the cursor was dropped. To reset the screen, the user simply uses the Manual Reset panel buttons (or alternatively, the keyboard left and down arrow keys) to jog back to the bottom left corner of the etch-a-sketch. The drawing can then be reset with the Reset Graph button of the Drawing Settings Panel and the entire etch-a-sketch can be reset by swinging the carriage backward. For full explanations of the machine's possibilities, the user can click the Help button.

Some may see this as an example of lazy automation, but this exercise of compensating for the standard etch-a-sketch's many flaws has applications in real-world projects. Consider fully automated robotic surgical systems (similar to da Vinci robots by Intuitive Surgical): while an etch-a-sketch is incapable of assisting in cardiac surgeries, both machines have user-controlled components that can easily lead to mistakes. It would be simple to accidentally run forceps into a patient's ribs, or continue turning a knob past the etch-a-sketch's borders, but adding feedback control and restricting the possibilities would decrease the frequency of user error. Like robotic surgery, operating an etch-a-sketch is certainly an exercise in patience and hand-eye-coordination, but automation of these tasks would be helpful for lowering chances of mistakes and missteps, especially in applications as sensitive to error as surgeries.

Technical Difficulties

Predicted Difficulties

At the proposal stage, our main predicted difficulty was creating an algorithm for matching the user's path on the computer screen to the etch-a-sketch screen. There would likely be problems with matching the pixels of lines to the same length scale on the etch-a-sketch, in addition to translating the user-controlled path into precise motor control for the knobs. Another difficulty predicted by the team was the challenge of translating designs into g-code-like instructions for the stepper motors. While plenty of research was put into how scripts like StippleGen analyze pictures using concepts from weighted Voronoi stippling and the classic traveling salesman problem, this feature was ultimately deemed infeasible to finish within the project timeframe, and it was exchanged for the seemingly-simpler alternative of having a type-to-etch-a-sketch feature: essentially, the user would type into a textbox and the etch-a-sketch would print the word(s) onto its screen.

Actual Challenges

Motors

Actual technical challenges began with motor usage, even before programming. Stepper motors were chosen for their high torque range and comparatively lower cost (as compared to continuous rotation servos with encoders). The lack of encoders meant that corrections and adjustments for missed steps would need to be programmatically implemented, but this issue was largely addressed by experimentally testing motor limits and limiting the control range so that user inputs would never result in exceeding the motor capabilities (discussed later). Motor overheating was also an issue because the motors acted as 12V * 0.3A = 4W heaters when no energy was dissipated by shaft movement. To reduce the motors' temperatures, current was decreased as much as possible while retaining the appropriate torque range, and aluminum was used for the beam fixing the motors to the etch-a-sketch carriage to act as a makeshift heat sink.

Alphabet Library

A functionality that the team attempted to implement but never came to fruition is the alphabet library for a type-to-etch-a-sketch function. The purpose of this section was to allow the user to input words into the GUI to be printed onto the etch-a-sketch screen. Each letter would consist of a series of directional commands that would eventually form the desired character. Problems for this included the backlash of the motors whenever the direction of

the drawing vector changed, causing an inaccuracy with lines not going as far as expected and meeting at the proper junctions.

Figure 1: Attempted letter "A" with timing issue

Curvature for letters such as "B" and "S" also proved to be difficult. Strategy to simply integrate curves involved dividing curvature letters into full circle, half circle, and straight line components. This allowed for easy translation to polar equations and then to x and y equations. As our main real-time feature made use of slope components to draw, derivatives of these equations are then taken and the outputs of these are fed to the drawing program. However, complications arose with Etch-A-Sketch backlash as each change in direction in the small letters and curves resulted in inaccurate movement. Had backlash resistance been implemented, time would have been needed to calibrate and adjust the drawing of each letter. Thus, this feature was never fully finished due to time constraints.

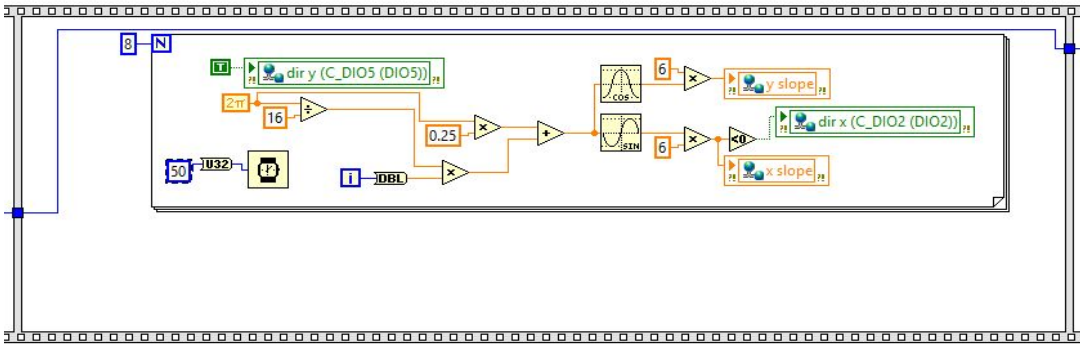


Figure 2: Code sequence for the curve of the letter "C".

If anything were to change, a more simplistic organized way of indicating direction and length of vectors in respect to the steps and rotations of the motor would have assisted greatly

in facilitating the string of commands for each individual line. The photo below shows a snippet of the code sequence for the letter “A”.

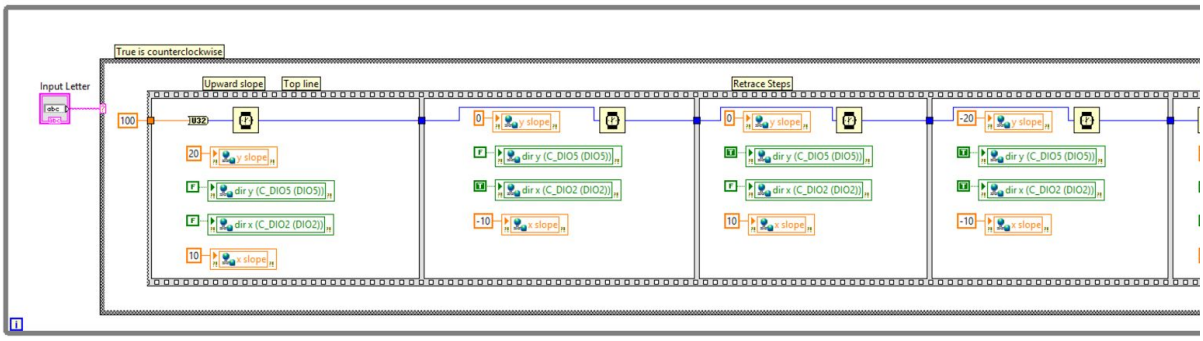
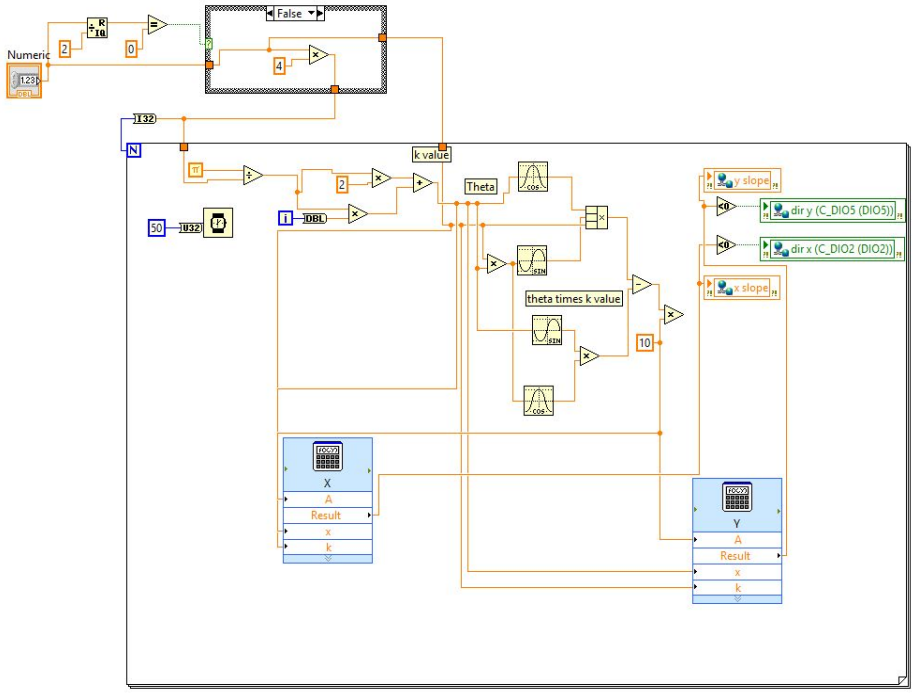


Figure 3: Portion of code sequence for the letter “A”.

Spirograph

Another attempt at a hardcoded demonstration took the form of a spirograph. The idea was to generate a spirograph pattern to demonstrate the project’s ability to draw curves. Similar to curve generation in letters, the spirograph is based on polar rose equations translated to x y coordinates. Derivatives of these equations are taken and used to produce a drawing. Again, due to backlash issues and time constraints, the proposal never came to fruition.



=

Figure 4: Code for Spirograph

Technical Challenges with Free-form Drawing

Drawing

Our first major challenge was taking the position of the mouse relative to the drawing window. In order to simply draw on the GUI as if it were paper, we took the position of the cursor relative to a defined image box. We then converted these screen pixel coordinates into XY coordinates and built these coordinates into a two dimensional array to feed to an XY graph. This graph continually updated to display the path of the cursor as the VI ran. However, the array logged any position on the screen that the mouse ever held since the time the program began running, even if it was outside the image drawing box. To prevent this, we added boundary conditions that prevented the position from being logged to the array if the cursor was outside of the box. We also gave it a condition that it would only log data when the mouse button was held down. We then scaled the graph and placed it over the image box to create the illusion that the user was drawing directly onto the graph.

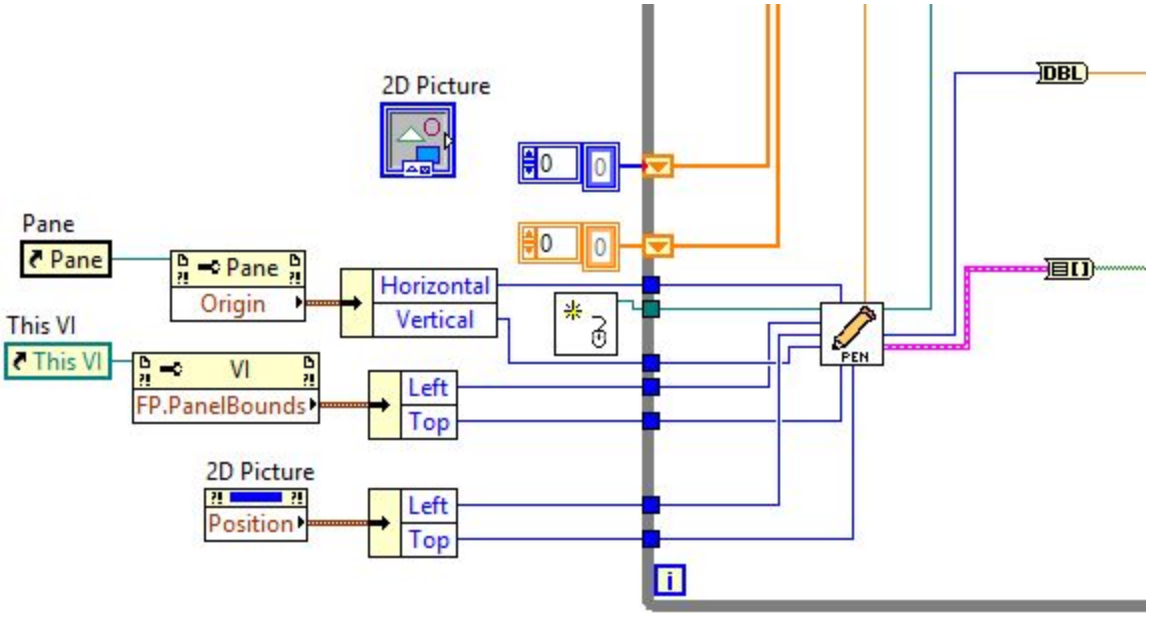


Figure 4. Snippet of code used to produce cursor coordinates

Generating Slopes for the Etch-a-Sketch

The stepper motors used in the etch-a-sketch did not have a system of feedback to determine the amount which they had turned, nor was there an easy way to determine the true position of the drawing without using complex imaging. In order to send the stepper motors commands to control their position, we sent the motor controlling program on the myRIO x and y changes over a fixed time interval. To do this, we used a feedback loop to compute the difference between the current graph coordinate and the graph coordinate from the previous sample. This produced X and Y slopes that could be converted to number of steps for the motors to run through in the same time interval as the sampling rate. These steps were then sent to the myRIO through global variables. One issue here was that the user on the computer could potentially draw much more quickly than the stepper motors could move. We considered several solutions to this problem, ultimately deciding to prevent position logging when the slope calculated was greater than the rate the motors could move. The user would then have to return to the position where logging ceased to continue drawing.

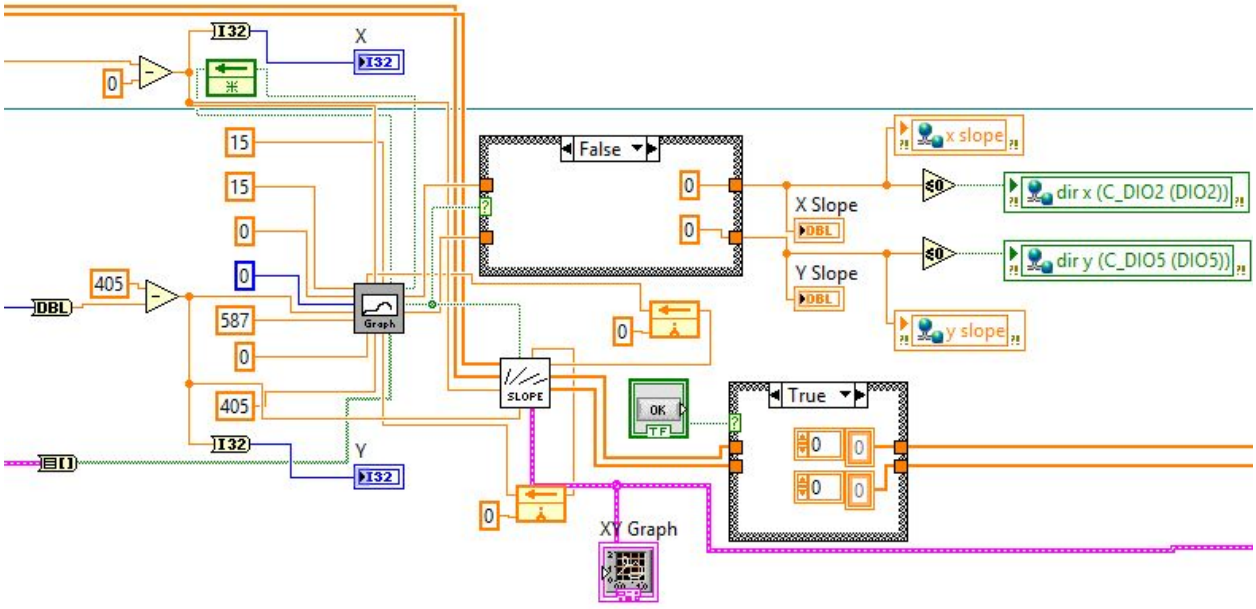


Figure 5. Code for producing graph and step commands

Controlling Motors from the myRIO

To control the motors, we had to send digital pulses to the motor controllers corresponding to each step. We did this by taking the number of steps sent by the computer program and passing them through a timed loop that sent a pulse to the controllers in each iteration. One challenge that we initially faced here was that, by sending pulses at a fixed rate, we could only get horizontal, vertical, or 45 degree lines. For example if the command we sent was x = 5 and y = 25, the cursor would travel 45 degrees for 5 steps, then vertically for 20 steps. To account for this, we changed the step rate to the motors based on the number of steps. In controlling the motors this way, we faced a variety of challenges in getting the motors to run properly. Most of these errors were due to simple mistakes buried in the code or due to our inexperience using stepper motors.

One issue that took us a while to resolve was that the motors would only run in the clockwise direction, and would get stuck attempting to run in the reverse direction. We eventually discovered that this was because the timing was controlled by the slope output from the global variables. When the slope was negative, we were sending a negative time signal to the loop, which caused the motors to malfunction. This was easily fixed by adding an absolute value function to the timing. Another challenge we faced was attempting to set the myRIO and computer programs such that they could be started and stopped simultaneously to limit the number of steps necessary on initial startup. To do this, we found instructions online for using application references and source distributions to start RIO VIs programmatically. In the end, we

were unable to get this to function properly and we had to manually start the motor control program from its front panel.

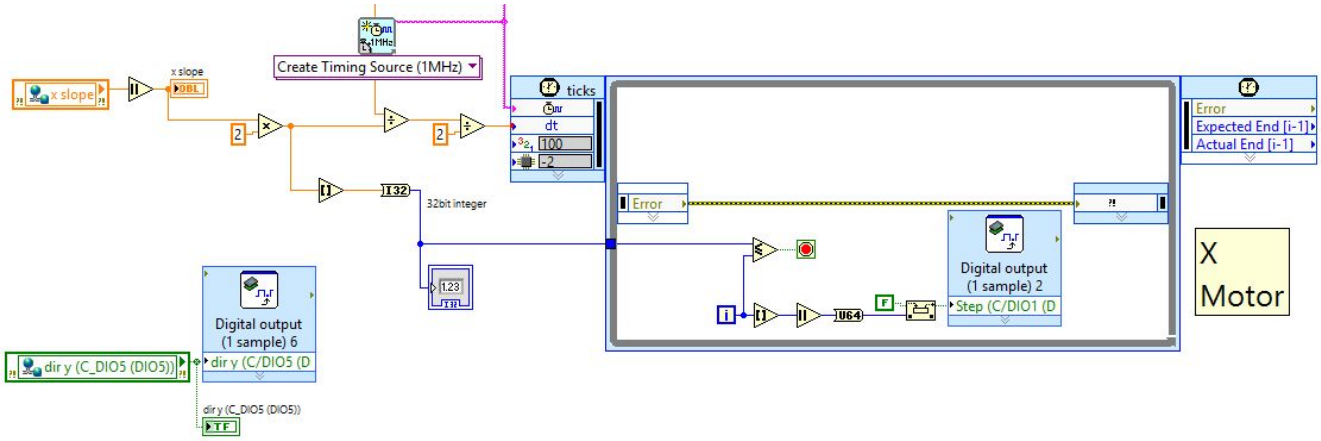


Figure 6. X segment of motor control program running on the myRIO

Note: Engineering journal is in the form of an Excel sheet (schedule of actions completed from 2/19-5/4), and photos + videos are accessible in a shared album