Week 2

Group Members: Group 3, Andrew Scerbo, Joey Lodato, Ronny Gattuso

Work Dates: 4/1 to 4/7

Joseph Lodato Work Log

This week I did a mix of the physical construction I was doing last week, while also starting on the software side of motor control. For the physical construction I continued work on the belt carriage and for the software side I made version 1 of our XY stepper motor controllers.

Physical construction work:

- For this side of the project the biggest thing that happened was the belts coming in from amazon. This means I had to start working on integrating them into the frame to move the Y axis along the X direction (confusing I know), then also move the Pen carriage along the Y direction. I have successfully got the first part of this working, although it does need refinements still. Currently the belt mounts I 3d printed last week are working ok and can move the Y axis bar around successfully. Although there is some issues with it still, the main issue is that it is hard to tension the belts successfully, I have a new belt mount that should fix this, although I have not tested them yet. If this new belt mount does work, I will add it to the other X motor and the Y motor once the carriage is designed and made.
- I also mounted the RPi and a breadboard with the motor controllers to the frame and wired together the 3 XY Stepper motors to the RPi and the motor controllers.

Code / Motor controllers work:

- This week I also worked on writing some code and getting the RPi setup. First, since the Pi is mounted to the frame, we will want to write the code on our laptops, rather than dragging over the whole metal frame and trying to get it on a desk to wrote code. So I set up a GitHub Repository that we all have access to. With this we can write code on our own

laptops and just do a `git pull` or `git fetch` when ever we want to try our code on on the RPi.

- With the GitHub set up I begin writing code to make a stepper motor class. This is so we can initialize as many stepper motors as we want to control and do it all in a single python file. Right now the code initializes 3 motors and moves each around a little bit so we can verify the code and the physical set up does work (which both do). Below is my full code for the first version of my motor controller:

```
import time
import gpiod

MICROSTEPPING_MODE = "FULL"  # Options: FULL, HALF, 1/4, 1/8
MICROSTEP_CONFIG = {
    "FULL": [0, 0],
    "HALF": [1, 0],
    "1/4": [0, 1],
    "1/8": [1, 1],
}

class StepperMotor:
    def __init__(self, chip_name, dir_pin, step_pin, ms1_pin, ms2_pin,
enable_pin, name="Motor"):
        self.chip = gpiod.Chip(chip_name)
        self.pins = [dir_pin, step_pin, ms1_pin, ms2_pin, enable_pin]
        self.lines = self.chip.get_lines(self.pins)
        self.lines.request(consumer="mp6500", type=gpiod.LINE_REQ_DIR_OUT)
        self.name = name
        self._set_microstepping(MICROSTEPPING_MODE)
        self.enable()

    def _set_microstepping(self, mode):
        ms_values = MICROSTEP_CONFIG[mode]
        self.microstep_values = ms_values

    def enable(self):
        self.lines.set_values([0, 0, *self.microstep_values, 0])

    def disable(self):
        self.lines.set_values([0, 0, 0, 0, 1])

    def set_direction(self, direction):
        values = self.lines.get_values()
        values[0] = direction
        self.lines.set_values(values)
```

```
    def pulse(self, delay=0.001):
        values = self.lines.get_values()
        values[1] = 1
        self.lines.set_values(values)
        time.sleep(delay)
        values[1] = 0
        self.lines.set_values(values)
        time.sleep(delay)

    def cleanup(self):
        self.disable()
        self.lines.release()
        self.chip.close()



def moveX(steps, direction, delay=0.001):
    # Move the two X motors together, they are coupled so they shouldnt get
desynced by this code
    motorX1.set_direction(direction)
    motorX2.set_direction(direction)

    for _ in range(steps):
        motorX1.pulse(delay)
        motorX2.pulse(delay)

def moveY(steps, direction, delay=0.001):
    #move only Y motor
    motorY.set_direction(direction)
    for _ in range(steps):
        motorY.pulse(delay)

def moveXY(x_steps, x_dir, y_steps, y_dir, delay=0.001):

    # Move both X and Y at the same time but doing different things

    motorX1.set_direction(x_dir)
    motorX2.set_direction(x_dir)
    motorY.set_direction(y_dir)

    max_steps = max(x_steps, y_steps)
    x_ratio = x_steps / max_steps if x_steps else 0
    y_ratio = y_steps / max_steps if y_steps else 0

    x_progress = 0.0
    y_progress = 0.0
```

```
    for _ in range(max_steps):
        if x_progress < 1.0:
            motorX1.pulse(delay)
            motorX2.pulse(delay)
            x_progress += x_ratio
        if y_progress < 1.0:
            motorY.pulse(delay)
            y_progress += y_ratio

if __name__ == '__main__':
    main()

# Initialize motors
motorX1 = StepperMotor("gpiochip4", 20, 21, 22, 23, 24, name="X1")
motorX2 = StepperMotor("gpiochip4", 25, 26, 27, 28, 29, name="X2")
motorY = StepperMotor("gpiochip4", 5, 6, 7, 8, 9, name="Y")

# if we use stepper motors for the Z axis we will want to disable
microstepping for them as we don need the precision
#      (We also are not using microstepping so that comment above might be
useless)
# it will be a slightly different initalization as I will make if it
diffferent class likely (idk)


def main():
    print("Starting motor test...")

    # Move motors in a sequence
    try:
        print("Moving X forward 100 steps")
        moveX(100, direction=1)
        time.sleep(1)

        print("Moving Y forward 50 steps")
        moveY(50, direction=1)
        time.sleep(1)

        print("Moving X back")
        moveX(100, direction=0)
        time.sleep(1)

        print("Moving Y back")
        moveY(50, direction=0)
        time.sleep(1)
```

```
    except KeyboardInterrupt:
        print("Motion interrupted by user")

    finally:
        motorX1.cleanup()
        motorX2.cleanup()
        motorY.cleanup()
```

Next Up:

- Next week I want to have the full physical construction done other than maybe motion in the Z axis (up and down). Ideally I will have the XY motion completely done from a physical side, meaning I can throw a lot of my energy at getting a good motor driver script going that will take in data from Andrew's image processing code. At some point I also need to figure out and design a system to raise and lower the drawing surface, this might happen next week but is likely to get worked on the week after.

Andrew Scerbo Work Log

Wednesday 4/2 --- Andrew Scerbo

Made my code with the proper dimensions of the camera and only displays the and records the coordinates in a square plane.

```
import cv2
import numpy as np
from skimage.morphology import skeletonize
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
import io
from PIL import Image

# --- Camera Setup ---
cap = cv2.VideoCapture(1)
if not cap.isOpened():
    print("Camera not available")
    exit()

# --- UI Settings ---
BUTTON_WIDTH = 150
DISPLAY_SIZE = 480  # Square display (480x480)
```

```
WINDOW_NAME = "Skeleton Camera"
snapshot_counter = 0

# --- Button Area (x1, y1, x2, y2) ---
button_coords = (DISPLAY_SIZE + 10, 200, DISPLAY_SIZE + BUTTON_WIDTH - 10,
280)
button_clicked = False

# --- Mouse Callback ---
def mouse_callback(event, x, y, flags, param):
    global button_clicked
    if event == cv2.EVENT_LBUTTONDOWN:
        x1, y1, x2, y2 = button_coords
        if x1 <= x <= x2 and y1 <= y <= y2:
            button_clicked = True

cv2.namedWindow(WINDOW_NAME, cv2.WND_PROP_FULLSCREEN)
cv2.setWindowProperty(WINDOW_NAME, cv2.WND_PROP_FULLSCREEN,
cv2.WINDOW_FULLSCREEN)
cv2.setMouseCallback(WINDOW_NAME, mouse_callback)

# --- Skeleton Plot Helper ---
def generate_skeleton_plot(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    _, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)
    binary_bool = binary > 0
    skeleton = skeletonize(binary_bool).astype(np.uint8) * 255

    contours, _ = cv2.findContours(skeleton, cv2.RETR_TREE,
cv2.CHAIN_APPROX_NONE)

    fig = Figure(figsize=(5, 5), dpi=100)
    ax = fig.add_subplot(1, 1, 1)

    for contour in contours:
        coords = contour.reshape(-1, 2)
        if len(coords) > 1:
            xs, ys = zip(*coords)
            ax.plot(xs, ys, marker='.', linestyle='-', linewidth=0.5)

    ax.set_title("Skeleton Contours")
    ax.invert_yaxis()
    ax.axis('equal')
    ax.grid(True)

    canvas = FigureCanvas(fig)
```

```
    buf = io.BytesIO()
    canvas.print_png(buf)
    buf.seek(0)
    img_pil = Image.open(buf).convert('RGB')
    img_np = np.array(img_pil)
    return cv2.cvtColor(img_np, cv2.COLOR_RGB2BGR)

# --- Main Loop ---
while True:
    ret, frame = cap.read()
    if not ret:
        continue

    # Crop center square
    h, w = frame.shape[:2]
    min_dim = min(h, w)
    cx, cy = w // 2, h // 2
    square_frame = frame[cy - min_dim//2:cy + min_dim//2, cx - min_dim//2:cx +
min_dim//2]
    square_frame = cv2.resize(square_frame, (DISPLAY_SIZE, DISPLAY_SIZE))

    # Create canvas: square display + button
    canvas_width = DISPLAY_SIZE + BUTTON_WIDTH
    canvas = np.ones((DISPLAY_SIZE, canvas_width, 3), dtype=np.uint8) * 50
    canvas[:, :DISPLAY_SIZE] = square_frame

    # Draw button
    x1, y1, x2, y2 = button_coords
    cv2.rectangle(canvas, (x1, y1), (x2, y2), (200, 200, 200), -1)
    cv2.putText(canvas, "Snapshot", (x1 + 10, y1 + 50),
cv2.FONT_HERSHEY_SIMPLEX,
                0.8, (0, 0, 0), 2)

    cv2.imshow(WINDOW_NAME, canvas)

    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        break

    if button_clicked:
        plot_img = generate_skeleton_plot(square_frame)
        filename = f"skeleton_plot_{snapshot_counter}.png"
        cv2.imwrite(filename, plot_img)
        snapshot_counter += 1

        # Show result briefly
```

```
        cv2.imshow("Skeleton Plot", plot_img)
        cv2.waitKey(3000)
        cv2.destroyWindow("Skeleton Plot")

        button_clicked = False

cap.release()
cv2.destroyAllWindows()
```

Monday 4/7 --- Andrew Scerbo

We got a mini monitor to display our program. I implemented a GUI so I program can display the camera and a button to display the coordinates. The coordinates will be implemented as mechanical movements soon but I am also left space so when we add different buttons for different processes that they are available.

```
import cv2
import numpy as np
from skimage.morphology import skeletonize
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
import io
from PIL import Image

# --- Camera Setup ---
cap = cv2.VideoCapture(1)
if not cap.isOpened():
    print("Camera not available")
    exit()

# --- UI Settings ---
BUTTON_WIDTH = 150
DISPLAY_SIZE = 480  # Square display (480x480)
WINDOW_NAME = "Skeleton Camera"

# --- Button Area (x1, y1, x2, y2) ---
button_coords = (DISPLAY_SIZE + 10, 200, DISPLAY_SIZE + BUTTON_WIDTH - 10,
280)
button_clicked = False

# --- Mouse Callback ---
def mouse_callback(event, x, y, flags, param):
    global button_clicked
    if event == cv2.EVENT_LBUTTONDOWN:
        x1, y1, x2, y2 = button_coords
```

```
        if x1 <= x <= x2 and y1 <= y <= y2:
            button_clicked = True

cv2.namedWindow(WINDOW_NAME, cv2.WND_PROP_FULLSCREEN)
cv2.setWindowProperty(WINDOW_NAME, cv2.WND_PROP_FULLSCREEN,
cv2.WINDOW_FULLSCREEN)
cv2.setMouseCallback(WINDOW_NAME, mouse_callback)

# --- Skeleton Plot Helper ---
def generate_skeleton_plot(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    _, binary = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)
    binary_bool = binary > 0
    skeleton = skeletonize(binary_bool).astype(np.uint8) * 255

    contours, _ = cv2.findContours(skeleton, cv2.RETR_TREE,
cv2.CHAIN_APPROX_NONE)

    fig = Figure(figsize=(5, 5), dpi=100)
    ax = fig.add_subplot(1, 1, 1)

    for contour in contours:
        coords = contour.reshape(-1, 2)
        if len(coords) > 1:
            xs, ys = zip(*coords)
            ax.plot(xs, ys, marker='.', linestyle='-', linewidth=0.5)

    ax.set_title("Skeleton Contours")
    ax.invert_yaxis()
    ax.axis('equal')
    ax.grid(True)

    canvas = FigureCanvas(fig)
    buf = io.BytesIO()
    canvas.print_png(buf)
    buf.seek(0)
    img_pil = Image.open(buf).convert('RGB')
    img_np = np.array(img_pil)
    return cv2.cvtColor(img_np, cv2.COLOR_RGB2BGR)

# --- Main Loop ---
while True:
    ret, frame = cap.read()
    if not ret:
        continue
```

```
    # Crop center square
    h, w = frame.shape[:2]
    min_dim = min(h, w)
    cx, cy = w // 2, h // 2
    square_frame = frame[cy - min_dim//2:cy + min_dim//2, cx - min_dim//2:cx +
min_dim//2]
    square_frame = cv2.resize(square_frame, (DISPLAY_SIZE, DISPLAY_SIZE))

    # Create canvas: square display + button
    canvas_width = DISPLAY_SIZE + BUTTON_WIDTH
    canvas = np.ones((DISPLAY_SIZE, canvas_width, 3), dtype=np.uint8) * 50
    canvas[:, :DISPLAY_SIZE] = square_frame

    # Draw button
    x1, y1, x2, y2 = button_coords
    cv2.rectangle(canvas, (x1, y1), (x2, y2), (200, 200, 200), -1)
    cv2.putText(canvas, "Snapshot", (x1 + 10, y1 + 50),
cv2.FONT_HERSHEY_SIMPLEX,
                0.8, (0, 0, 0), 2)

    cv2.imshow(WINDOW_NAME, canvas)

    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        break

    if button_clicked:
        plot_img = generate_skeleton_plot(square_frame)

        # Just show the result, no saving
        cv2.imshow("Skeleton Plot", plot_img)
        cv2.waitKey(3000)
        cv2.destroyWindow("Skeleton Plot")

        button_clicked = False

cap.release()
cv2.destroyAllWindows()
```

Ronny Gattuso Work Log