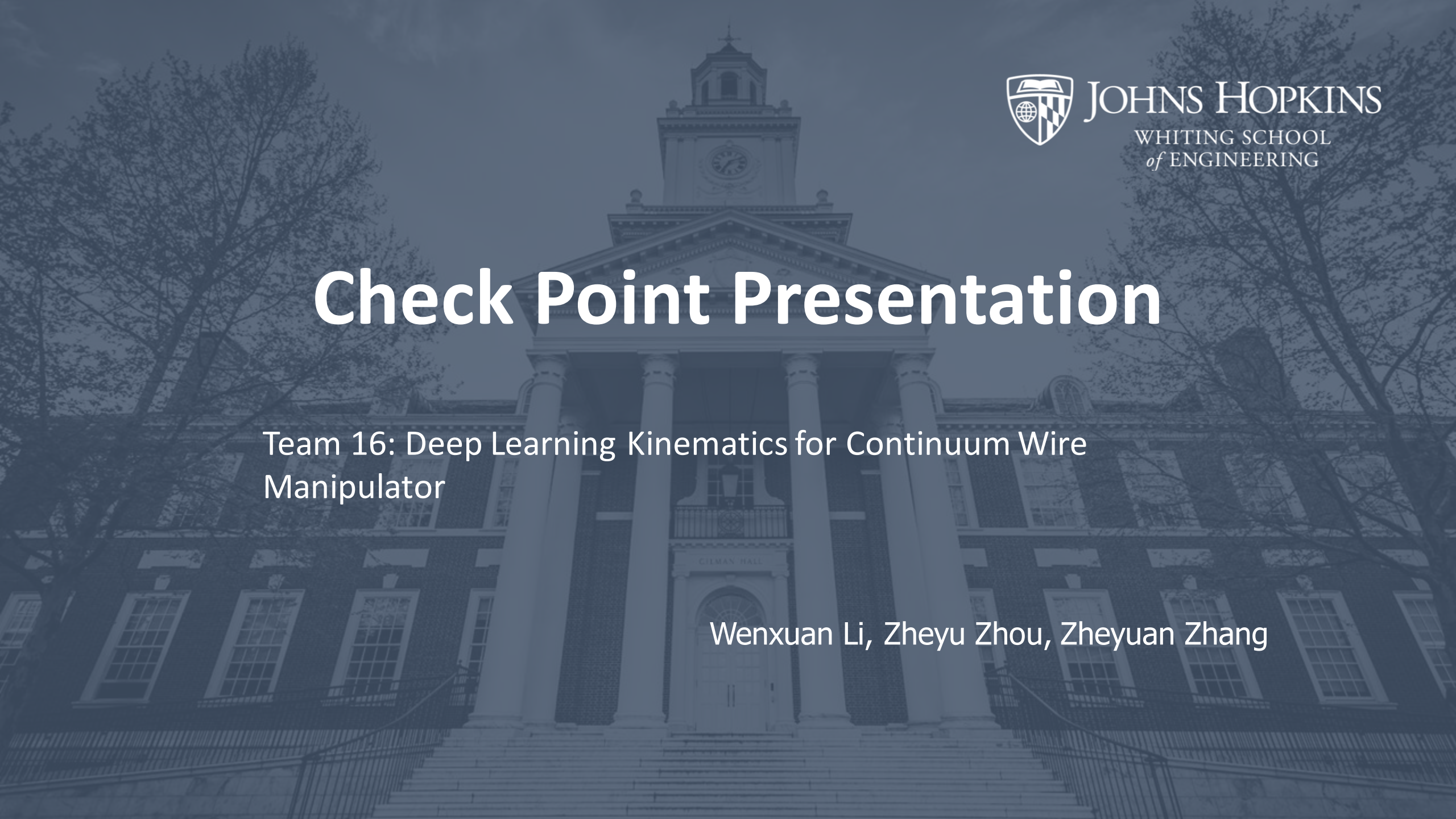
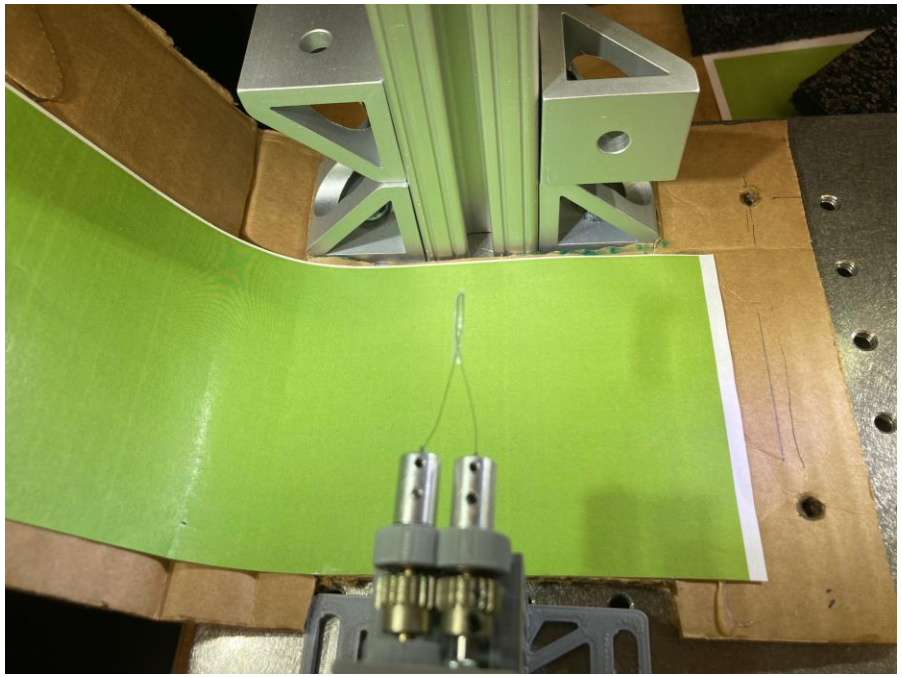


Check Point Presentation

Team 16: Deep Learning Kinematics for Continuum Wire Manipulator

Wenxuan Li, Zheyu Zhou, Zheyuan Zhang

Project Summary



- The project goal is to develop a Deep Learning based kinematics model for the novel continuum wire manipulator for future surgery implementation.

Documentation

- Equipment list
- Test station setup guide
- Software environment requirement
- Background reading
- Data collection procedure
- CAD
- Code

Data Collection Procedure

In this file, we will outline the steps to create a data collection procedure that tests the motor encoder numbers from -6000 to 6000, corresponding to angles from -119.06 degrees to 119.06 degrees. The PID control will control the motor and traverse all positions, while a detection program will loop to monitor the motor's traversal of all positions. When the motor reaches an incremental position, the camera will receive a shooting command, and the output will be stored using ROS. The result will include the motor angle and a set of point clouds representing the shape of the soft robot, which can be transformed into a corresponding voxel using a function.

Set up PID control for the motor:

Implement PID control to precisely control the motor position.

Set the motor to traverse all positions from -6000 to 6000 (or -119.06 degrees to 119.06 degrees).

Develop the detection program.

Create a loop in the detection program to monitor the motor's traversal of all positions.

When the motor reaches an incremental position, send a shooting command to the camera.

Configure the camera to capture images.

Set up the camera to receive shooting commands and capture images upon receiving a command.

Use the ROS framework to output and store the captured images.

Process the captured images.

Implement a detection algorithm to obtain a set of point clouds representing the shape of the soft robot.

Save the motor angle and the set of point clouds as the result.

Convert point clouds to voxels.

Create a function that takes the set of point clouds and the motor angle as input and outputs the corresponding voxel representation of the soft robot's shape.

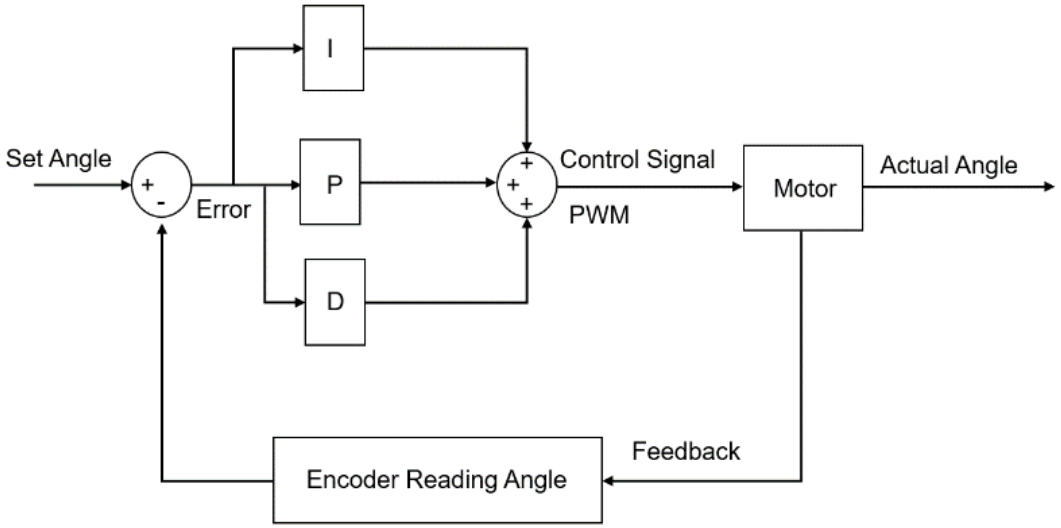
By following these steps, you will create a data collection procedure that tests the motor encoder numbers from -6000 to 6000 and captures images of the soft robot's shape at each incremental position. The output will include the motor angle and a set of point clouds representing the soft robot's shape, which can be transformed into a voxel representation using a function.

Technical Progress — Testbed setup

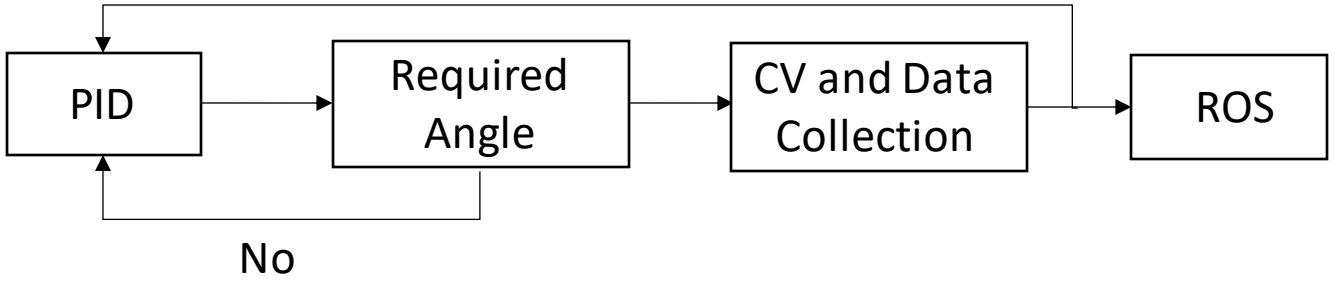
- Camera Pole and Holder
- Motor Housing and Set up.
- Green Curtain
- Lighting

Technical Progress — Motor Control

PID Logic

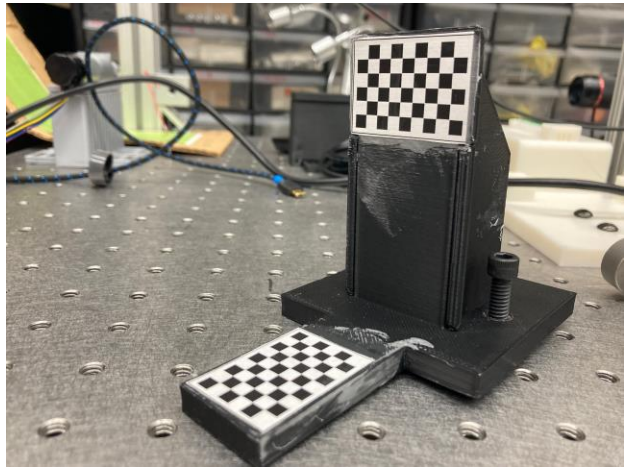


Data collection Process



Technical Progress — Calibration

- Intrinsic



Intrinsic.py

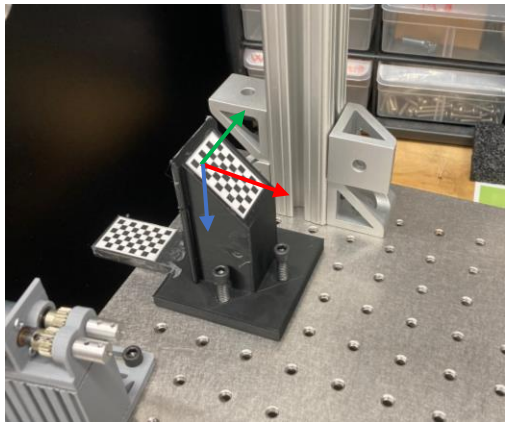
```
# which_cam = '_l'

# # video capture
# capture = cv2.VideoCapture(0)
# capture.open(4)

which_cam = '_r'

# video capture
capture = cv2.VideoCapture(1)
capture.open(6)

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((5*8,3), np.float32)
objp[:,:2] = np.mgrid[0:8,0:5].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

# Num of data collected
num_data = 20


while True:
    ret, frame = capture.read()
    frame = cv2.resize(frame, (640, 360), interpolation=cv2.INTER_CUBIC)
    frame = frame[25:, :]
    cv2.imshow("frame", frame) # display left
    ret_cb, corners = cv2.findChessboardCorners(frame, (8,5), None)
    if ret_cb == True:
        gray_masked = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        corners = cv2.cornerSubPix(gray_masked, corners, (15, 15), (-1, -1), criteria)

        if len(objpoints) < num_data:
            objpoints.append(objp)
            imgpoints.append(corners)
        elif ret_cb != True:
            print(ret_cb, 'not found corners yet')

    if len(objpoints) == num_data:
        # calibrate
        # print(len(objpoints[0]), objpoints[0], type(objpoints), 'objpoints 3D')
        # print(len(imgpoints[0]), imgpoints[0], type(imgpoints[0]),'imgpoints 2D')
        ret_cali, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray_masked.shape[::-1],None,None)
        if ret_cali != True:
            h,  w = frame.shape[:2]
            newcameramtx, roi = cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))

            # undistort
            undst = cv2.undistort(frame, mtx, dist, None, newcameramtx)
            print(undst.shape[:2], 'shape after undistort')
            print(mtx, dist,newcameramtx, 'output')
            np.save('mtx'+which_cam+'.npy',mtx)
            np.save('dist'+which_cam+'.npy',dist)
            np.save('newcameramtx'+which_cam+'.npy',newcameramtx)


            mean_error = 0
            for i in range(len(objpoints)):
                imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx, dist)
                error = cv2.norm(imgpoints[i], imgpoints2, cv2.NORM_L2)/len(imgpoints2)
                mean_error += error
                print( "total error: {}".format(mean_error/len(objpoints)) )

            break
    if cv2.waitKey(100) & 0xff == ord('q') or ret != True: # press q to quit
            break
capture.release()  # release cam left
cv2.destroyAllWindows() # close windows
```

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_o \\ Y_o \\ Z_o \end{bmatrix}$$

(OpenCV, (n.d.))

Technical Progress — Calibration

- Extrinsic



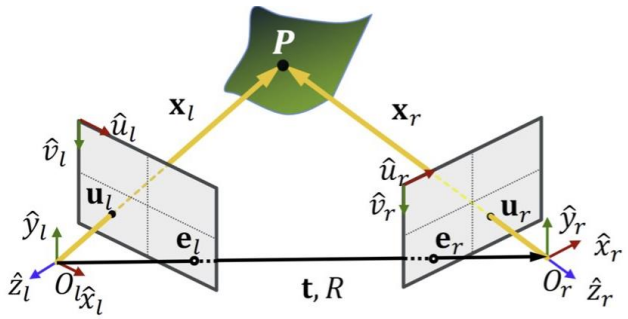
Extrinsic.py

```
import numpy as np
import cv2

# Calibrate Camera Seperately (Use Intrinsic.py)
objp_l = np.load('objp_l.npy')
mtx_l = np.load('mtx_l.npy')
dist_l = np.load('dist_l.npy')
newcameramtx_l = np.load('newcameramtx_l.npy')

objp_r = np.load('objp_r.npy')
mtx_r = np.load('mtx_r.npy')
dist_r = np.load('dist_r.npy')
newcameramtx_r = np.load('newcameramtx_r.npy')

capture_l = cv2.VideoCapture(0)
capture_l.open(4)

capture_r = cv2.VideoCapture(1)
capture_r.open(6)

while True:
    ret_l, frame_l = capture_l.read()
    frame_l = cv2.resize(frame_l, (640, 360), interpolation=cv2.INTER_CUBIC)
    frame_l = frame_l[25:, :]
    # frame_l = cv2.undistort(frame_l, mtx_l, dist_l, None, newcameramtx_l)
    ret_cb_l, corners_l = cv2.findChessboardCorners(frame_l, (8,5), None)

    ret_r, frame_r = capture_r.read()
    frame_r = cv2.resize(frame_r, (640, 360), interpolation=cv2.INTER_CUBIC)
    frame_r = frame_r[25:, :]
    # frame_r = cv2.undistort(frame_r, mtx_r, dist_r, None, newcameramtx_r)
    ret_cb_r, corners_r = cv2.findChessboardCorners(frame_r, (8,5), None)

    cv2.imshow('frame_l', frame_l)
    cv2.imshow('frame_r', frame_r)
    print(ret_cb_l, ret_cb_r, 'find or not')
    if ret_cb_l == True and ret_cb_r == True:
        rtval_l, rvec_l, tvec_l = cv2.solvePnP(objp_l, corners_l, newcameramtx_l, dist_l)
        R3x3_l, Jacobian_l  = cv2.Rodrigues(rvec_l)

        rtval_r, rvec_r, tvec_r = cv2.solvePnP(objp_r, corners_r, newcameramtx_r, dist_r)
        R3x3_r, Jacobian_r  = cv2.Rodrigues(rvec_r)

        R3x3_l2r = R3x3_r.dot(R3x3_l.T)
        tvec_l2r = R3x3_r.dot(-R3x3_l.T.dot(tvec_l)) + tvec_r
        M3x4_l2r = np.concatenate((R3x3_l2r, tvec_l2r), axis=1)

        np.save('R3x3_l', R3x3_l)
        np.save('tvec_l', tvec_l)

        np.save('R3x3_r', R3x3_r)
        np.save('tvec_r', tvec_r)

        np.save('R3x3_l2r', R3x3_l2r)
        np.save('tvec_l2r', tvec_l2r)
        np.save('M3x4_l2r', M3x4_l2r)
        if rtval_l == True and rtval_r == True:
            break
    if cv2.waitKey(100) & 0xff == ord('q') or ret_l != True or ret_r != True: # press q to quit
        break
```

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix} = \mathbf{K}\,{}^{c}\mathbf{M}_o \begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix}$$

$${}^{c_2}\mathbf{M}_{c_1} = {}^{c_2}\mathbf{M}_o \cdot {}^{o}\mathbf{M}_{c_1} = {}^{c_2}\mathbf{M}_o \cdot ({}^{c_1}\mathbf{M}_o)^{-1} = \begin{bmatrix} {}^{c_2}\mathbf{R}_o & {}^{c_2}\mathbf{t}_o \\ 0_{3\times1} & 1 \end{bmatrix} \cdot \begin{bmatrix} {}^{c_1}\mathbf{R}_o^T & -{}^{c_1}\mathbf{R}_o^T \cdot {}^{c_1}\mathbf{t}_o \\ 0_{1\times3} & 1 \end{bmatrix}$$

(OpenCV, (n.d.))

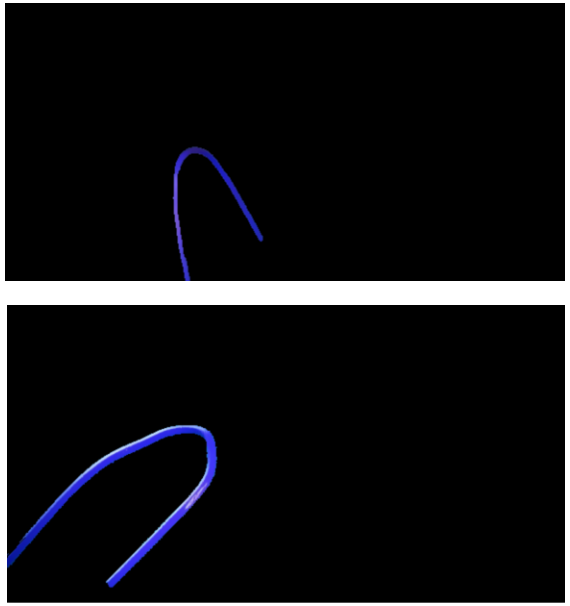


(Jones, 2023)

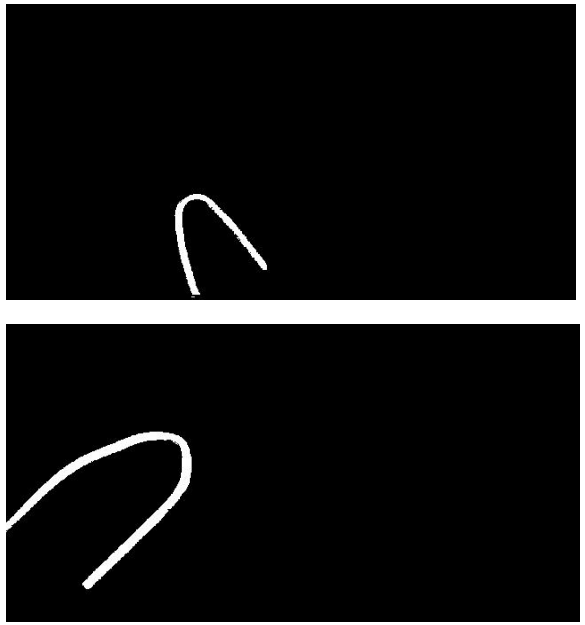
Technical Progress — Image Processing

- Filtering

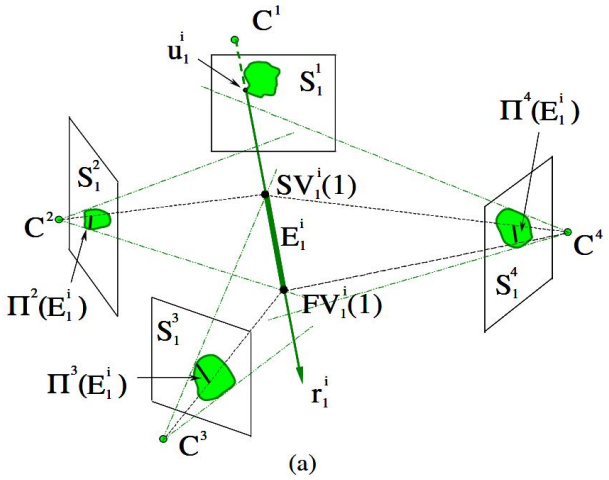
Bilateral Filter

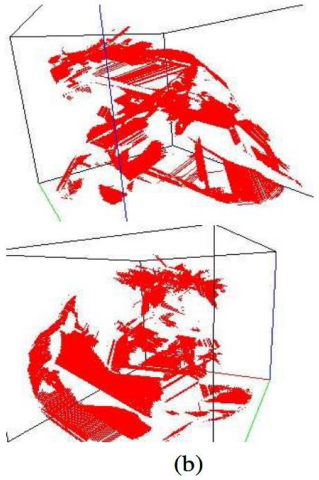


Chroma Key Mask



Technical Progress — Shape from Silhouette





(Kong-man, 2023)

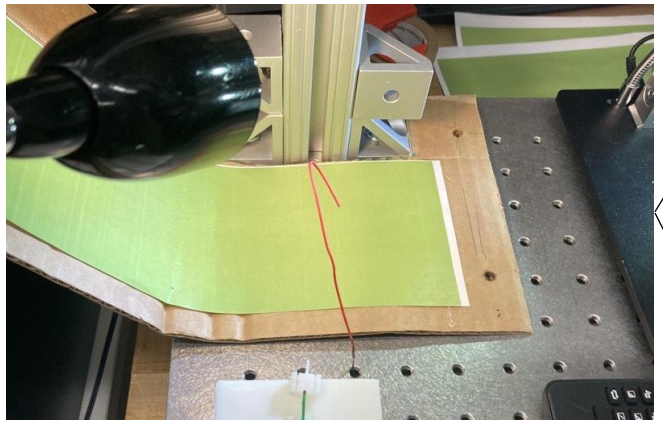
SFS.py

```python
import numpy as np
# import matplotlib.pyplot as plt
# from mpl_toolkits.mplot3d import Axes3D

def uvtoXYZ(img, E_oc_inv):
    XYZ_all = []
    print(img.shape[0], img.shape[1])
    for h in range(img.shape[0]):
            for w in range(img.shape[1]):
                # print(img[h,w])
                if img[h,w] == 255:
                    # wire_pixel_l.append((i,j))
                    uv11 = np.array([
                        [w],
                        [h],
                        [1],
                        [1]
                    ])
                    XYZ1 = np.dot(E_oc_inv,uv11)
                    XYZ = XYZ1.T[0][0:3]
                    XYZ_all.append(XYZ)
    return XYZ_all

def findintersection(set_l, set_r, tvec_l, tvec_r):
    tvec_l = tvec_l.T[0] / np.linalg.norm(tvec_l.T[0])
    tvec_r = tvec_r.T[0] / np.linalg.norm(tvec_r.T[0])

    cross_check = np.cross(tvec_l, tvec_r)
    intersect_points = []
    # intersect_points_ = []
    if (np.linalg.norm(cross_check) != 0):
        for XYZ_l in set_l:
            for XYZ_r in set_r:
                # Find the point of intersection
                coeff_matrix = np.vstack((tvec_l, -tvec_r)).T
                const_matrix = XYZ_r - XYZ_l
                t1, t2 = np.linalg.lstsq(coeff_matrix, const_matrix, rcond=None)[0]
                # intersect = XYZ_l + tvec_l * t
                intersect = XYZ_l + tvec_l * t1

                # intersect_ = XYZ_r + tvec_r * t2
                # print(t1, tvec_l, '1')
                # print(t2, tvec_r, '2')
                # print(intersect, intersect_, intersect-intersect_)
                # intersect = t
                intersect_points.append(intersect)
                # intersect_points_.append(intersect_)

                print(len(intersect_points), len(set_l)*len(set_r))
    else:
        print("The two vectors are parallel and do not intersect.")
        return 0

    return np.array(intersect_points)

def pointcloud2voxel(intersect_points, resolution):
    # Determine the bounding box of the point cloud
    min_coords = np.min(intersect_points, axis=0)
    max_coords = np.max(intersect_points, axis=0)
    bounding_box_size = max_coords - min_coords
    print(bounding_box_size)
    # Calculate the number of voxels needed in each dimension
    voxel_counts = np.ceil(bounding_box_size / resolution).astype(int)
    print(voxel_counts, 'voxel counts')
    # Create an empty voxel grid
    voxel_grid = np.zeros(voxel_counts)
    print(np.shape(voxel_grid), 'grid shape')
    # Determine which voxels are occupied by points
    for point in intersect_points:
        # Calculate the index of the voxel that contains the point
        voxel_index = np.floor((point - min_coords) / resolution).astype(int)

        # Mark the voxel as occupied
        voxel_grid[tuple(voxel_index)] = 1

    return voxel_grid
```

z

y

Technical Progress — Shape from Silhouette

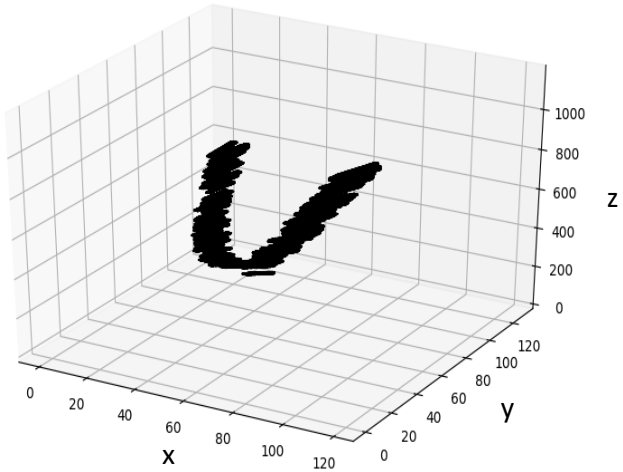


Test Wire

Binary Top View

Binary Side View

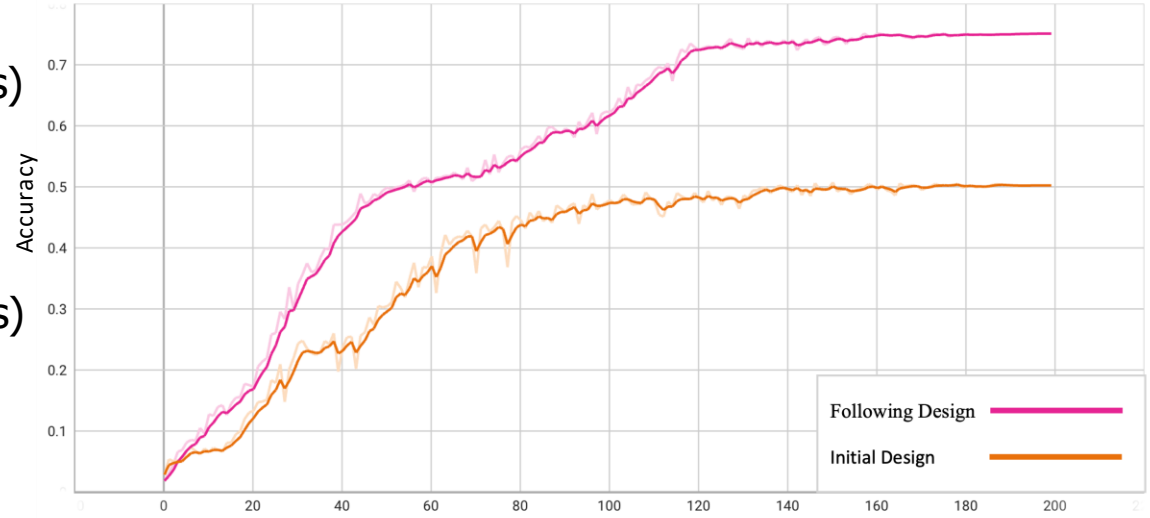
Voxels



Technical Progress — Deep Learning Model Design

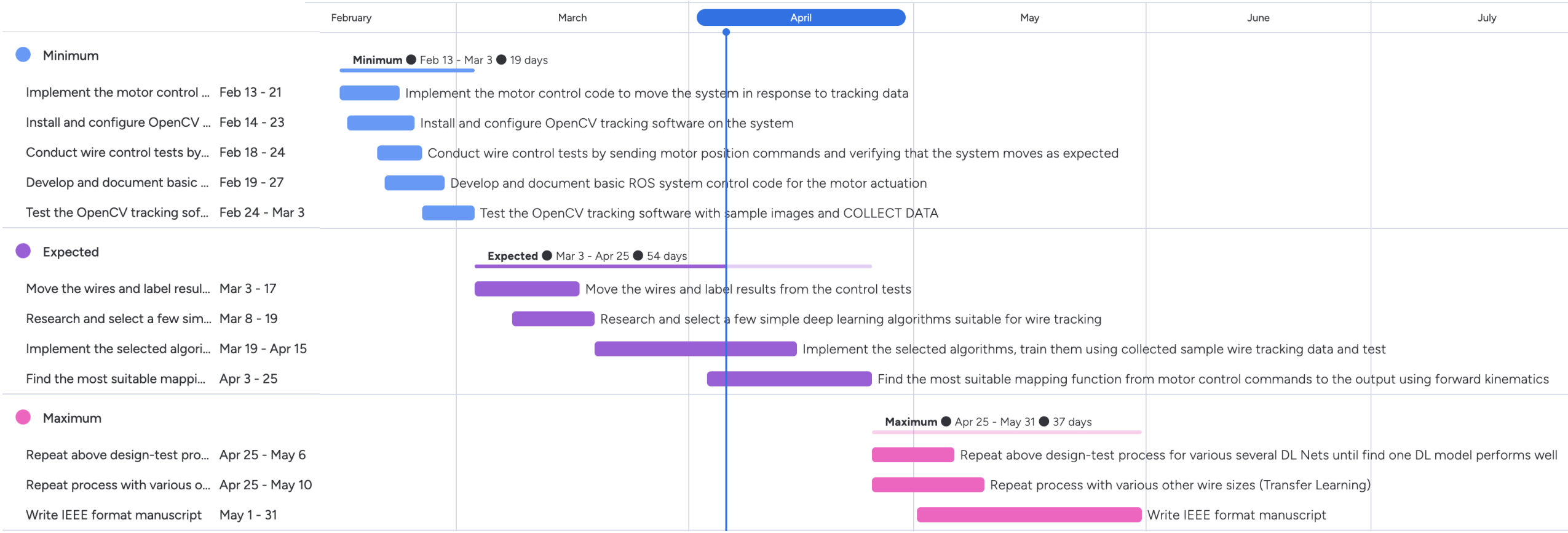
Implementation – Train – Validation:

- ☑ Initial Design:
  - Fully connected feed-forward neural network (2 hidden layers)
  - AdamW Optimizer & ReLU activation function
- ☑ Following Design 1:
  - Fully connected feed-forward neural network (3 hidden layers)
  - AdamW Optimizer & ReLU activation function
- ☑ Following Design 2:
  - Fully connected feed-forward neural network (3 hidden layers)
  - Vanilla gradient descent & ReLU activation function
- 🤖👦💻👦💻 Following Design 3:
  - PointNet Encoder + Fully connected feed-forward neural network (3 hidden layers)
  - AdamW Optimizer & ReLU activation function



Learning Curve (Initial Design vs. Following Design)

Management— Timeline



Management— Milestone & Deliverables

Deliverables	Milestone	Subtasks	Status
Minimum	Milestone 1	OpenCV Setup: create environment, image capturing from two cameras, processing image and shape carving, and 3D data output	✅
		Motor Control Test: motor angle control with PID, ROS system setup and encoder data collection and image data matching	✅
		Hardware Setup: nitinol wire manufacturing, motor mounting, and camera mounting and CV auxiliary parts	✅
		Data Collection: create datasets for Deep Learning and transfer learning method	✅
Expected	Milestone 2	Move the wires and label results from the control tests	✅
		Research and select a few simple deep learning algorithms suitable for wire tracking	✅
		Implement the selected algorithms, train them using collected sample wire tracking data and test after	🤖👦💻👦💻
Maximum	Milestone 3	Repeat above design-test process for various several DL Nets until find the most suitable mapping function from motor control commands to the output using forward kinematics	🤖👦💻👦💻
		Repeat process with various other wire sizes (Transfer Learning to various other wire sizes)	❌
		Write IEEE format manuscript	❌

Next Steps

1. Finish repeating "Implementation – Train – Validation" process for various several DL Nets until we find the most suitable mapping function from motor control to the output using forward kinematics
2. Repeat above train-test process with various other wire sizes ->Transfer Learning
3. Write IEEE format manuscript

Reference

1. Jones, C. (2023, March 9). *Uncalibrated Binocular Stereo* [Lecture]. Johns Hopkins. https://jhu.instructure.com/courses/35057/files/6581057?module_item_id=2693902
2. Kong-man, C., Baker, S. and Kanade, T. (2005). Shape-from-silhouette across time part I: Theory and algorithms. International Journal of Computer Vision, vol. 62, no. 3, pp. 221–247.
3. OpenCV (n.d.). *Basic concepts of the homography explained with code*. https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

(Kuntz, Sethi, Webster III, & Alterovitz, 2020)