Polynomial Interpolation on Points from Point Cloud (Final Project Numerical MATH5610)

Nicole Forrester, Shing Hei Ho, Tanner Watts

Project Overview

Cutting is one of the most common and essential tasks in surgery. Valuable autonomous surgical assistants need to be able to cut tissue precisely. The current state of the art in autonomous robotic surgery uses point clouds to represent the tissues the robot cuts and manipulates. Point clouds mean the robotic control algorithm must predict points when planning a cut path; however, cutting must be smooth and cannot be represented just by points. We implement two different interpolation methods to generate the smooth cut path from predicted points. We also use Scipy to test our implantation versus their interpolation methods and a few new ones.

For our open source libraries we used scipy for different interpolation methods, numpy for matrix arithmetic, and matplotlib to plot our data.

Package Setup

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
# imports
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math
import os
import pickle
```

In [ ]:
```
# mount the dataset (the drive)
from google.colab import drive
drive.mount('/content/drive/')
```

```
Mounted at /content/drive/
```

In [ ]:
```
# set the data directory location
DATA_DIREC = "/content/drive/MyDrive/Splines_Numerical/data/"
```

Our cubic spline implementation

In [ ]:
```
def cubic_spline(xi, yi):
    """
    Computes the cubic spline coefficients for a set of data points.
```

```
    Args:
        xi: A list of x-coordinates.
        yi: A list of y-coordinates.

    Returns:
        A tuple containing the x-coordinates, coefficients a, b, c, and d.
    """
    ### solve for ai, bi ci, di for i=0,...,n-1 ###
    ai = yi[:] # 0 ... n
    hi = np.diff(xi) # 0 ... n-1

    diag1 = np.diag(np.insert(hi[1:],0,0), k=1)
    diag0 = np.insert(2*(hi[0:-1]-hi[1:]),0,1)
    diag0 = np.insert(diag0,-1,1)
    diag0 = np.diag(diag0, k=0)
    diagNeg1 = np.diag(np.insert(hi[:-1], -1,0), k=-1)

    A = diag0 + diag1 + diagNeg1
    b = 3/hi[1:]*(ai[2:] - ai[1:-1]) - 3/hi[0:-1]*(ai[1:-1] - ai[0:-2])
    b = np.insert(b,0,0)
    b = np.insert(b,0,-1)

    ci = np.matmul(np.linalg.inv(A),b) #np.linalg.solve(A,b)
    di = 1/3*1 /hi * (np.diff(ci))
    bi = 1/hi*(np.diff(ai)) - hi/3*(2*ci[:-1]+ci[1:])

    return ai[:-1], bi, ci[:-1], di

def eval_cubic_spline(xi,ai,bi,ci,di,eval_x):
    n = len(xi)
    print("num nodes: ", n)
    eval_y = []
    for i,x in enumerate(eval_x):
        if i %10==0:
            print(f"evaluating interp node {i}")
        mask = xi>=x
        interval_idx = np.nonzero(mask)[0][0] - 1
        if interval_idx < 0:
            interval_idx+=1
        y = ai[interval_idx] + bi[interval_idx] * (x - xi[interval_idx]) + ci[interval
        eval_y.append(y)
    return eval_y
```

Our Newton polynomial interpolation implementation

In [ ]:
```
def newton_interp_poly(xi, yi):
  c = np.array(yi) #y_0...y_n
  n = len(yi)-1
  for k in range(1,n+1,1):
    d = xi[k] - xi[:k] #[x_k - x_0, ..., x_k - x_(k-1)]
    u = eval_newton_interp_poly(c[:k], xi[:k], xi[k])
    c[k] = (yi[k] - u)/np.prod(d)
  return c # vector of coeffs

def eval_newton_interp_poly(c, xi, x):
  '''
  xi are the interpolation nodes
```

```
  x is a point for evaluation
  c[i] is the leading coefficient of the poly interp x_0 ... x_i
  d[i] = x - xi[i]
  return the value of the interp poly at point x
  '''
  n = len(c)-1
  u = c[n]
  d = x - xi

  for k in range(n-1,-1,-1):
    u = u*d[k] + c[k]
  return u
```

Utilities

In [ ]:
```
def plot_2D_points(xs, ys_list, title, x_label, y_label):
    fig, ax = plt.subplots()
    for idx in range(len(ys_list)):
        ax.scatter(xs, ys_list[idx], s=8)
    ax.legend()
    ax.set_title(title)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    plt.show()
```

In [ ]:
```
def plot_3D_points(xs, ys, zs, title):
    fig = plt.figure(figsize = (10, 7))
    ax = plt.axes(projection ="3d")
    ax.scatter(xs, ys, zs, c=[0,0,1], s=8)
    ax.legend()
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    ax.view_init(azim=45, elev=30)
    plt.show()
```

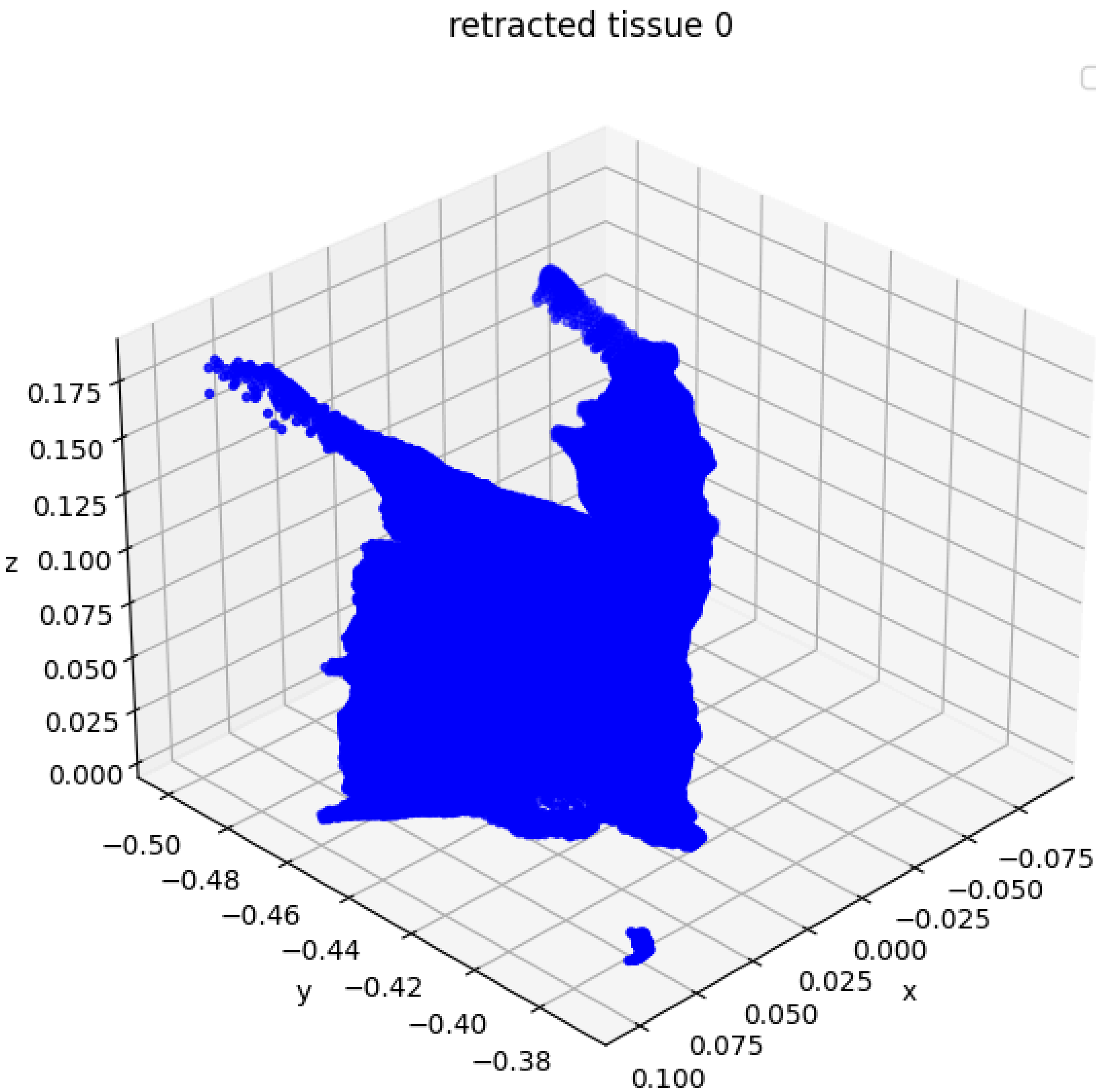
In [ ]:
```
def plot_3D_points_with_line(xs, ys, zs, title, line_x, line_y):
    fig = plt.figure(figsize = (10, 7))
    ax = plt.axes(projection ="3d")
    ax.scatter(xs, ys, zs, c="grey", s=0.01)
    ax.legend()
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    ax.view_init(azim=45, elev=30)
    ele = 25
    azm = 90
    ax.view_init(elev=ele, azim=azm)
    ax.plot(line_x, line_y, np.zeros(len(line_x)) + 0.01, color= "crimson")
    plt.show()
```

Data setup

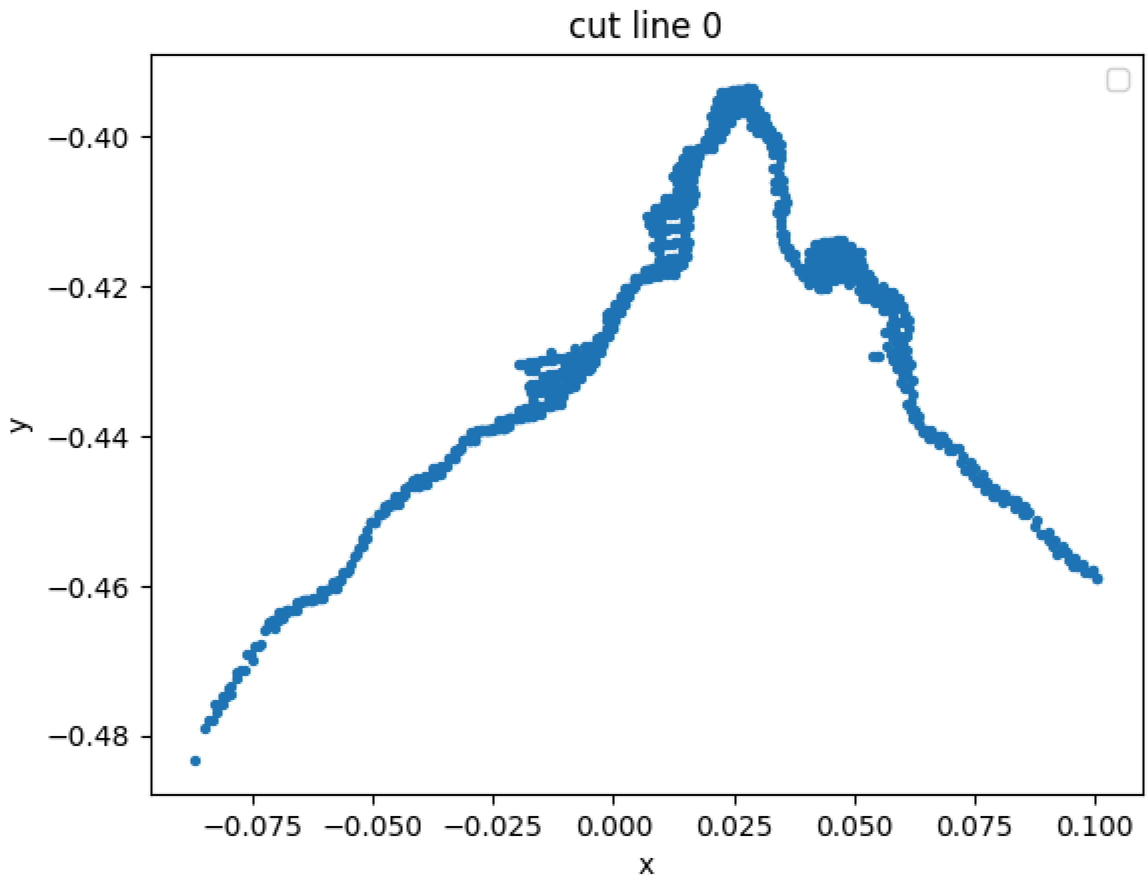
For each retracted tissue, we extact 3D partial-view point clouds, obtain the points at the bottom of the retracted tissue and plot their xy coordinates.

In [ ]:
```
pc_paths = [os.path.join(DATA_DIREC, "data_2.pickle"), os.path.join(DATA_DIREC, "data_
pcs = []
for pc_path in pc_paths:
  with open(pc_path, 'rb') as handle:
      data = pickle.load(handle)
  pc = np.array(data["pc"])[:,:3]
  pcs.append(pc)

pcs_filtered = []
for i in range(len(pcs)):
  pc = pcs[i]
  max_z = 0.008
  max_y = -0.379
  maskz = pc[:,2]<=max_z
  masky = pc[:,1]<=max_y
  mask = maskz & masky
  idx = np.nonzero(mask)[0]
  pc_filtered = pc[idx,:]
  pcs_filtered.append(pc_filtered)

  xs = pc_filtered[:,0]
  ys = pc_filtered[:,1]

  plot_3D_points(pc[:,0], pc[:,1], pc[:,2], title=f"retracted tissue {i}")

  plot_2D_points(xs, ys_list=[ys], title=f"cut line {i}", x_label="x", y_label="y")
```

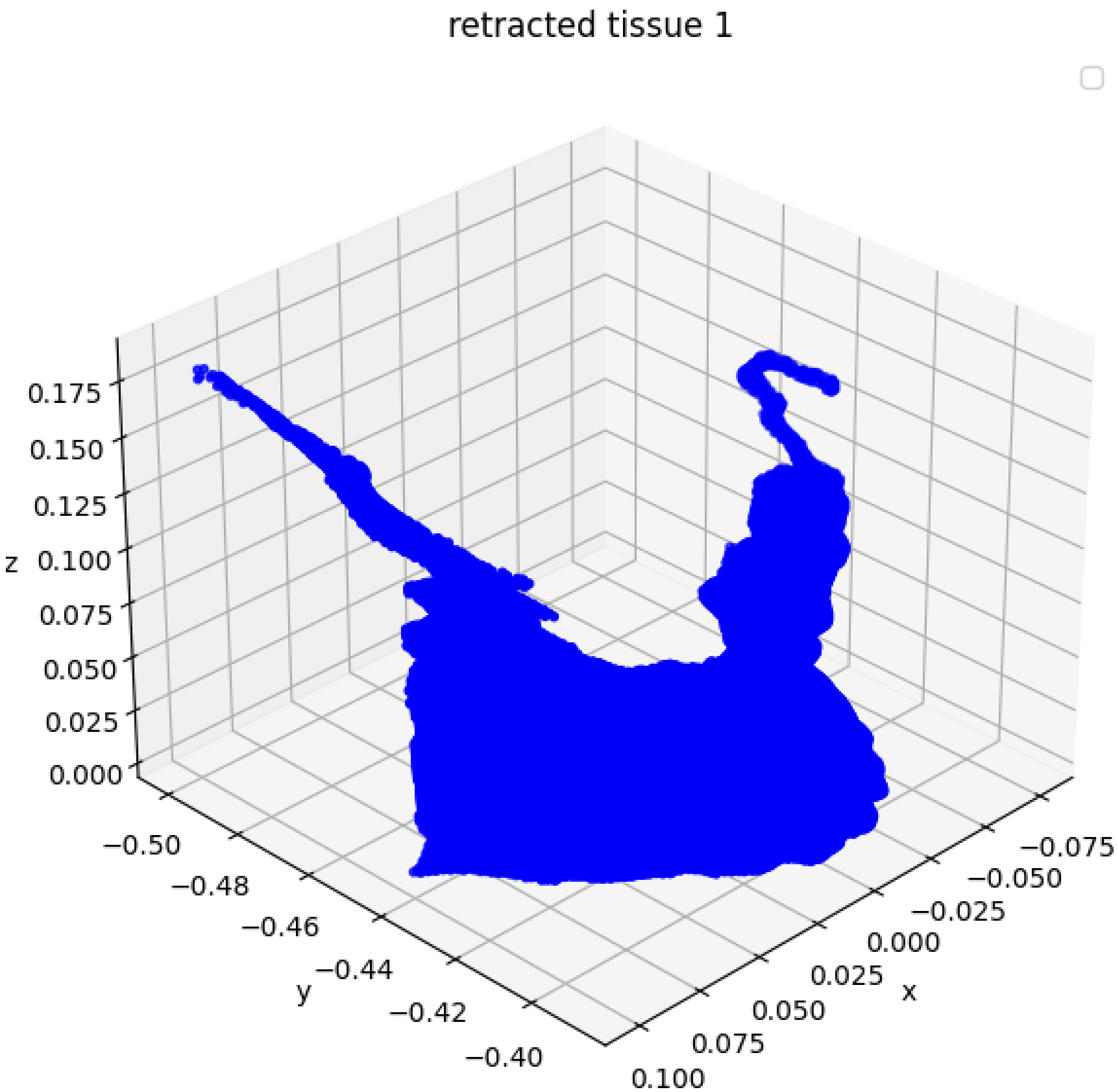
```
<ipython-input-8-6ac84deb4c4c>:4: UserWarning: *c* argument looks like a single numer
ic RGB or RGBA sequence, which should be avoided as value-mapping will have precedenc
e in case its length matches with *x* & *y*.  Please use the *color* keyword-argument
or provide a 2D array with a single row if you intend to specify the same RGB or RGBA
value for all points.
  ax.scatter(xs, ys, zs, c=[0,0,1], s=8)
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```



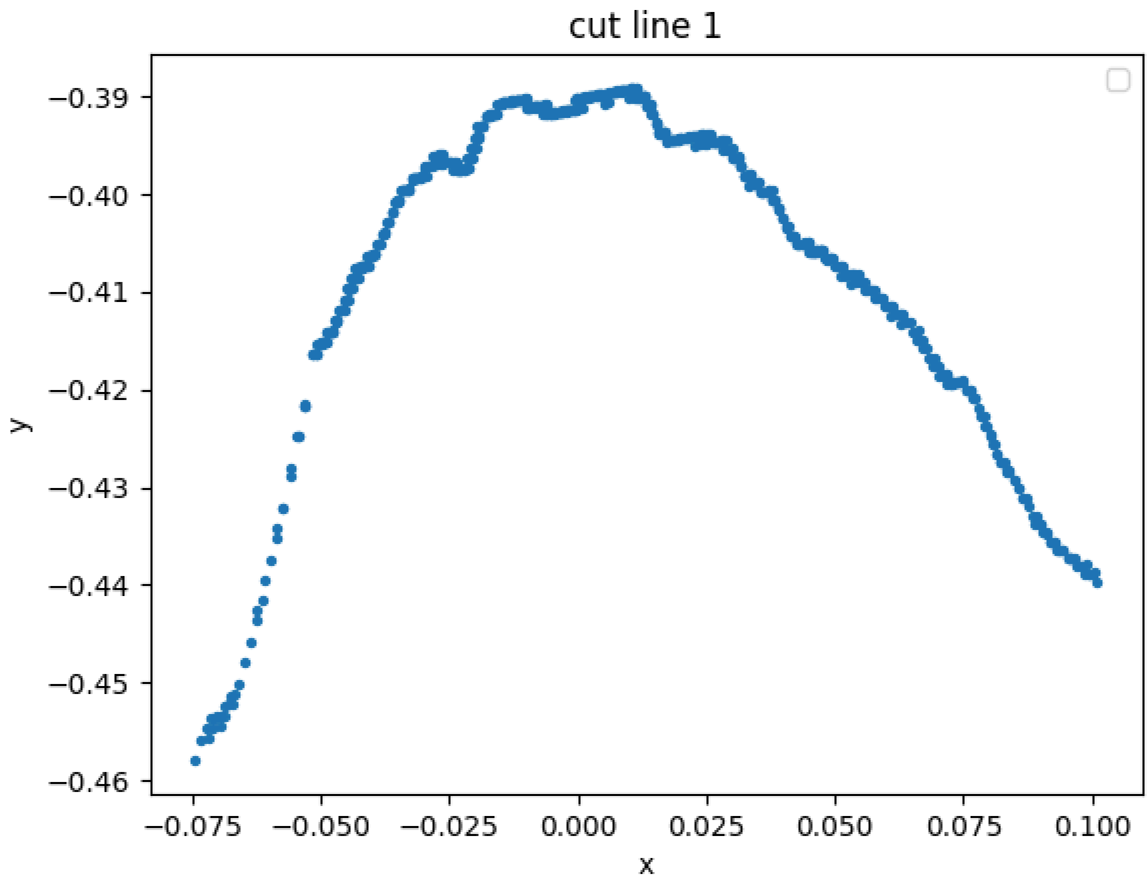

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```




```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```

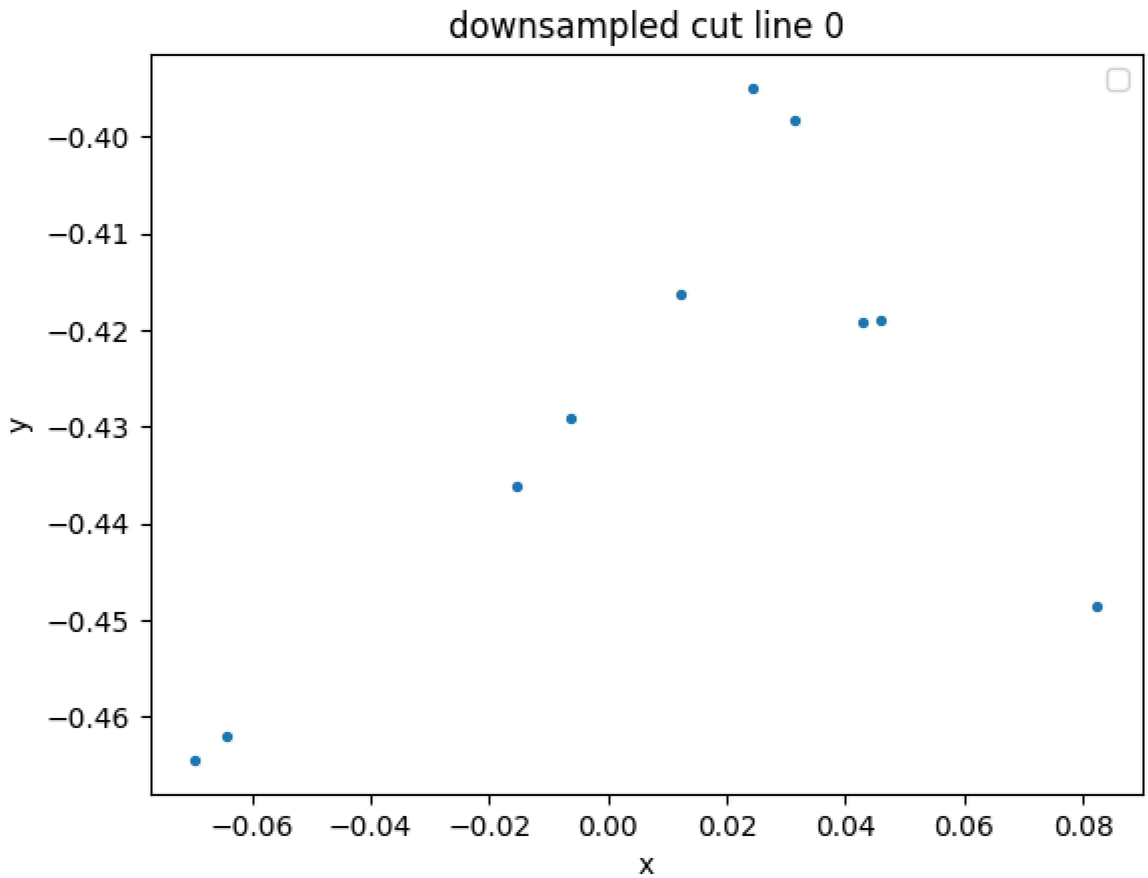



```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```

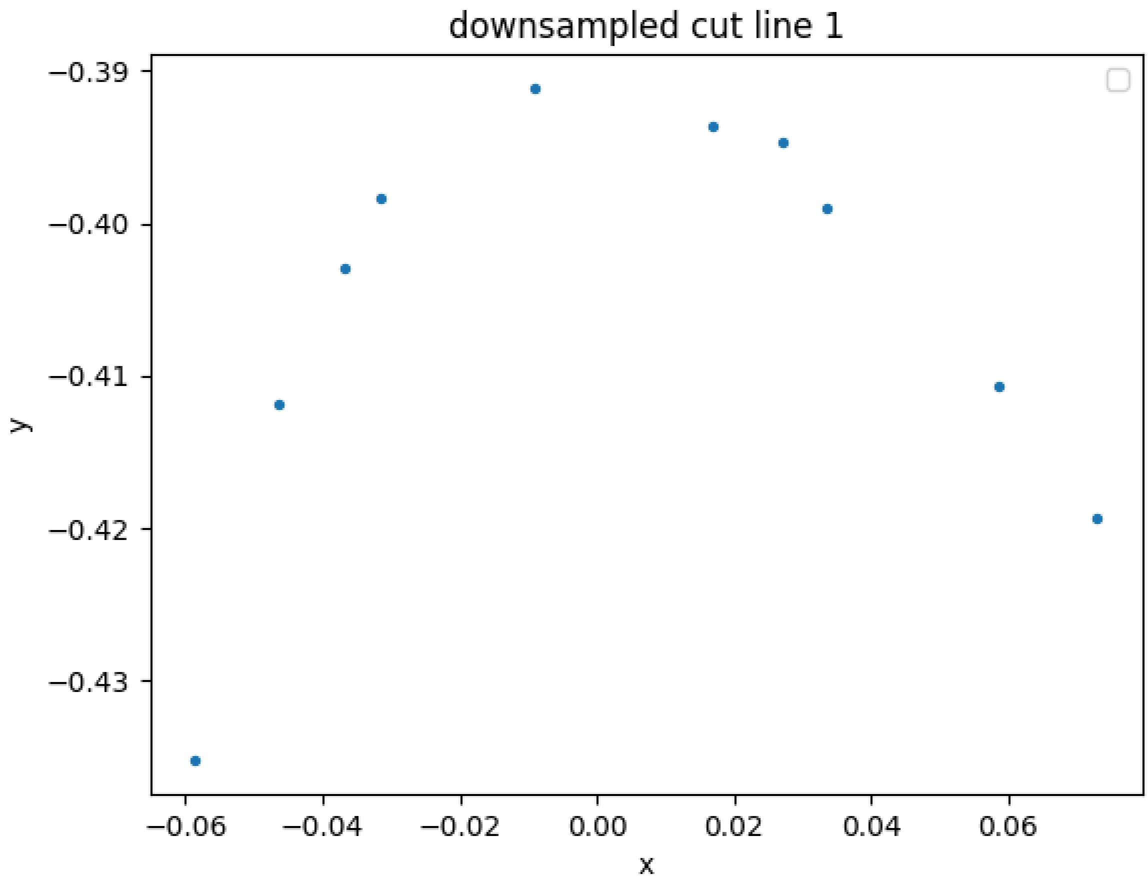


In [ ]:
```
# randomly sample points on each cut line for interpolation later
##### tune the number of interp points
num_interp_pts = 10
np.random.seed(2021)
pcs_downsampled = []
for i, pc_filtered in enumerate(pcs_filtered):
  rand_idxs = np.random.randint(low=0, high=len(pc_filtered), size=(num_interp_pts,))
  pc_downsampled = pc_filtered[rand_idxs,:]
  _,unique_idx = np.unique(pc_downsampled[:,1], return_index=True)
  pc_downsampled = pc_downsampled[unique_idx,:]
  pcs_downsampled.append(pc_downsampled)
  xs = pc_downsampled[:,0]
  ys = pc_downsampled[:,1]
  plot_2D_points(xs, ys_list=[ys], title=f"downsampled cut line {i}", x_label="x", y_l
  print(f"{i}--- num pts after downsample and remove duplicate: ", len(pc_downsampled)
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```




```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
0--- num pts after downsample and remove duplicate:  10
```




```
1--- num pts after downsample and remove duplicate:  10
```

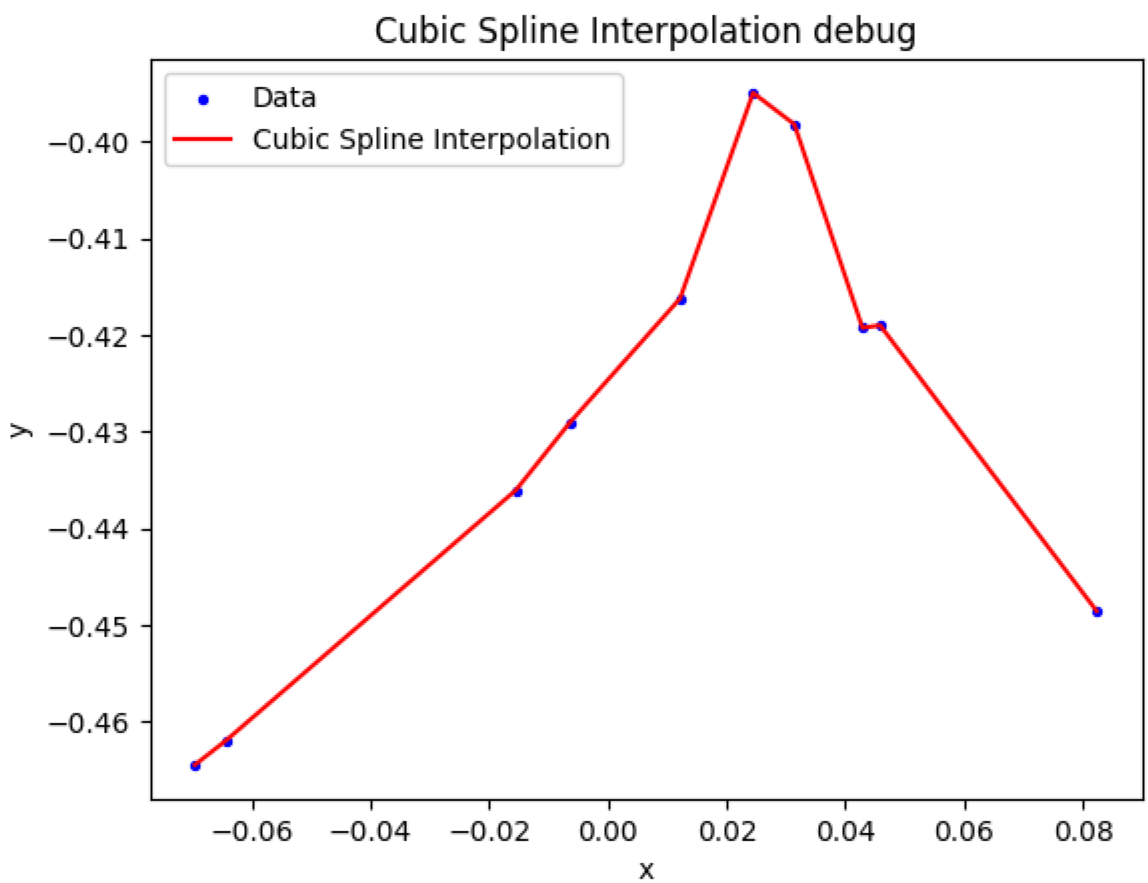
Debugging: make sure our implementatioin passes through all interpolation points

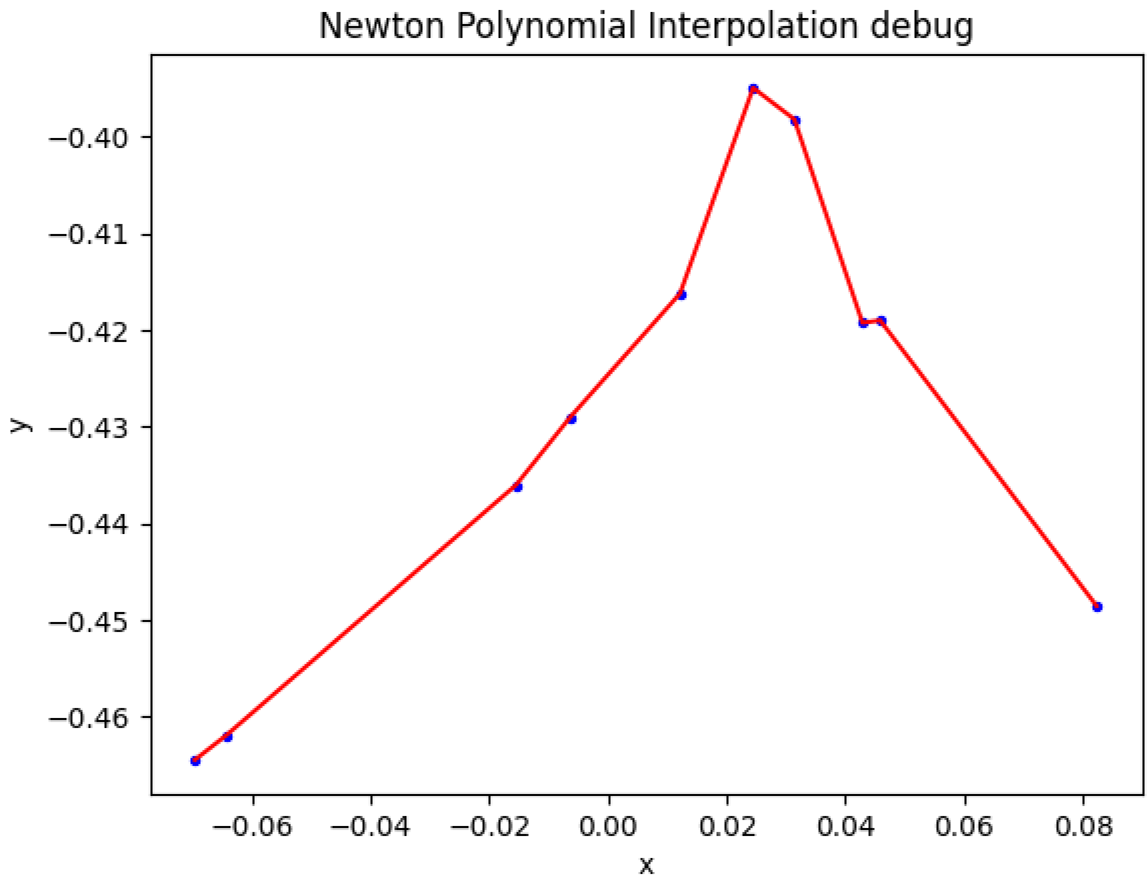
Here we are debugging our cubic spline interpolation to make sure the approximate line goes through all of our data points.

In [ ]:
```
pc_downsampled = pcs_downsampled[0]
xi = pc_downsampled[:,0].astype(float)
yi = pc_downsampled[:,1].astype(float)
sorted_x_idx = np.argsort(xi)
xi = xi[sorted_x_idx]
yi = yi[sorted_x_idx]

x_min = xi.min()
x_max = xi.max()
num_test_pts = 100
x_test = xi

######################### Cubic Spline Interpolation ########################

ai,bi,ci,di = cubic_spline(xi, yi)
y_test =  eval_cubic_spline(xi,ai,bi,ci,di,x_test)

plt.scatter(xi,yi, label='Data', color='blue', s=8)
plt.plot(x_test, y_test, label='Cubic Spline Interpolation', color='red')

plt.xlabel('x')
```

```
plt.ylabel('y')
plt.title('Cubic Spline Interpolation debug')

plt.legend()
plt.show()

############### Newton polynomial interpolation #############

c = newton_interp_poly(xi, yi)

y_test_newton = []
for x in x_test:
  y =  eval_newton_interp_poly(c,xi,x)
  y_test_newton.append(y)
ys = np.array(ys)

plt.scatter(xi, yi, label='Data', color='blue', s=8)
plt.plot(x_test, y_test_newton, label='Newton Poly Interpolation', color='red')

plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Newton Polynomial Interpolation debug')

plt.show()
```

```
num nodes:  10
evaluating interp node 0
```





Full comparison on "Cut Line 0"

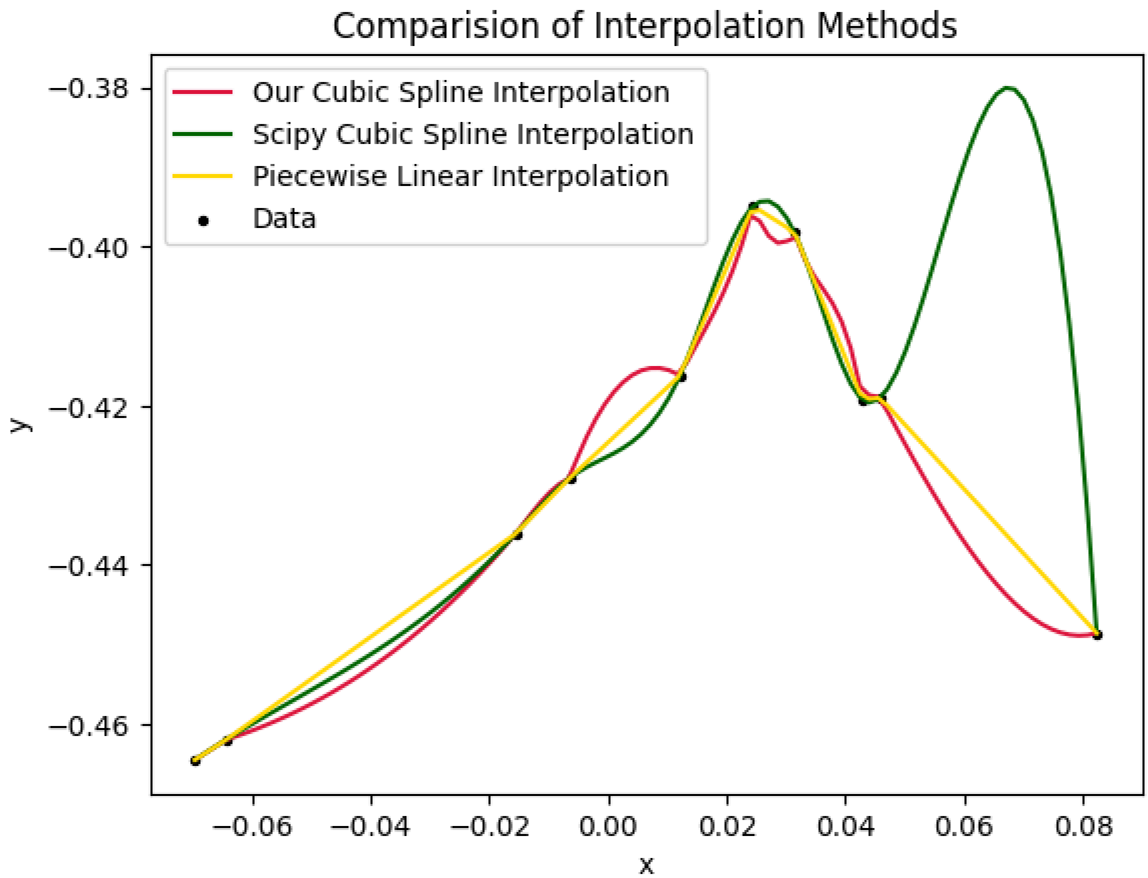
To compare multiple interpolation method we run interpolation on the same cut line with 100 test points and plot the results.

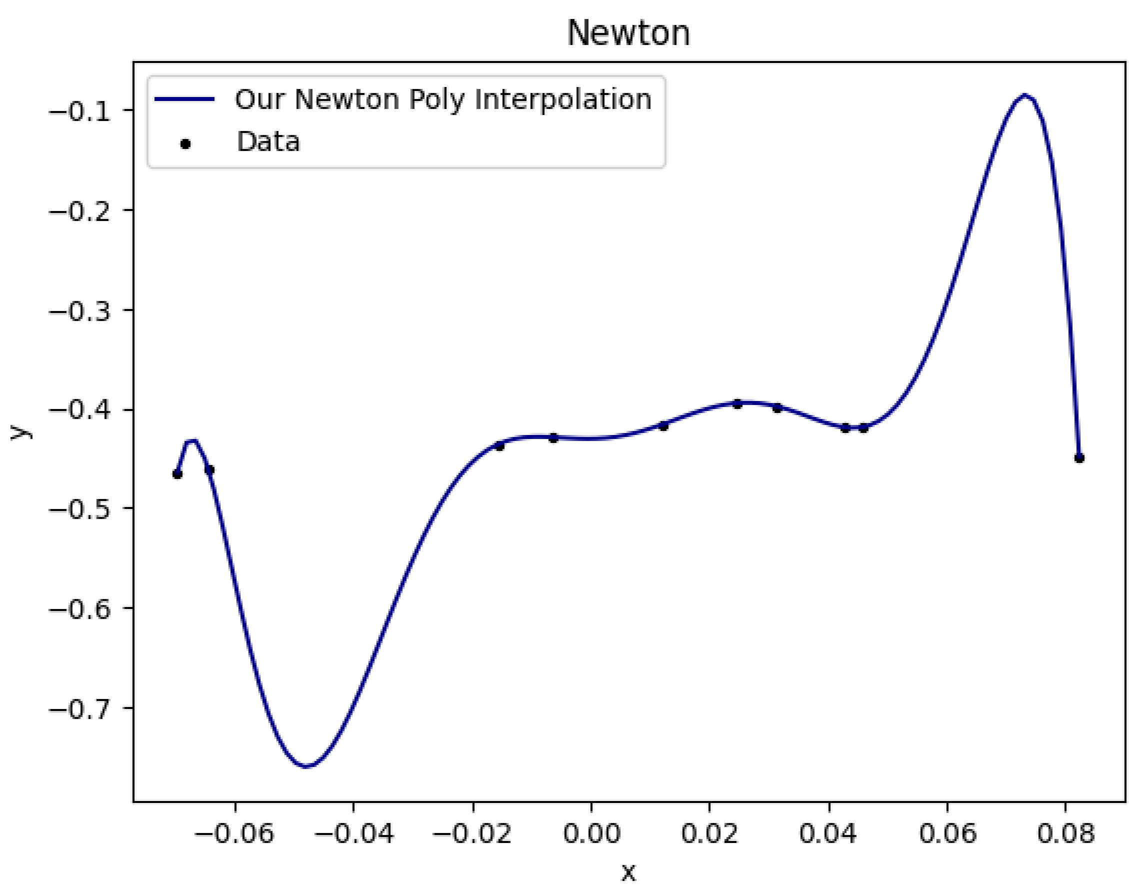
In [ ]:
```
pc_downsampled = pcs_downsampled[0]
xi = pc_downsampled[:,0].astype(float)
yi = pc_downsampled[:,1].astype(float)
sorted_x_idx = np.argsort(xi)
xi = xi[sorted_x_idx]
yi = yi[sorted_x_idx]

x_min = xi.min()
x_max = xi.max()
num_test_pts = 100
x_test = np.linspace(x_min, x_max, num_test_pts)
```

In [ ]:
```
######################### Cubic Spline Interpolation ########################

ai,bi,ci,di = cubic_spline(xi, yi)
y_test_Ospline =  eval_cubic_spline(xi,ai,bi,ci,di,x_test)


plt.plot(x_test, y_test_Ospline, label='Our Cubic Spline Interpolation', color='crimsc

################### scipy cubic spline ####################
from scipy.interpolate import CubicSpline
```

```
cs = CubicSpline(xi, yi)
y_test_spline = cs(x_test)

plt.plot(x_test, y_test_spline, label='Scipy Cubic Spline Interpolation', color='darkg


################# Piecewise Linear interpolation ##########################
xint = np.linspace(x_min, x_max, 300)
y_test_linear = np.interp(x_test, xi, yi)

plt.plot(x_test, y_test_linear, label='Piecewise Linear Interpolation', color='gold')


############### Newton polynomial interpolation #############

c = newton_interp_poly(xi, yi)

y_test = []
for x in x_test:
  y =  eval_newton_interp_poly(c,xi,x)
  y_test.append(y)
ys = np.array(ys)

#plt.plot(x_test, y_test, label='Our Newton Poly Interpolation', color='darkblue')

plt.scatter(xi,yi, label='Data', color='black', s=8)

plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Comparision of Interpolation Methods')

plt.legend()
plt.show()


plt.plot(x_test, y_test, label='Our Newton Poly Interpolation', color='darkblue')
plt.scatter(xi,yi, label='Data', color='black', s=8)

plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Newton')

plt.legend()
plt.show()

best_x_tissue_1 = x_test
best_y_tissue_1 = y_test_Ospline
```

```
num nodes:  10
evaluating interp node 0
evaluating interp node 10
evaluating interp node 20
evaluating interp node 30
evaluating interp node 40
evaluating interp node 50
evaluating interp node 60
evaluating interp node 70
evaluating interp node 80
evaluating interp node 90
```





Full comparison on "Cut Line 1"

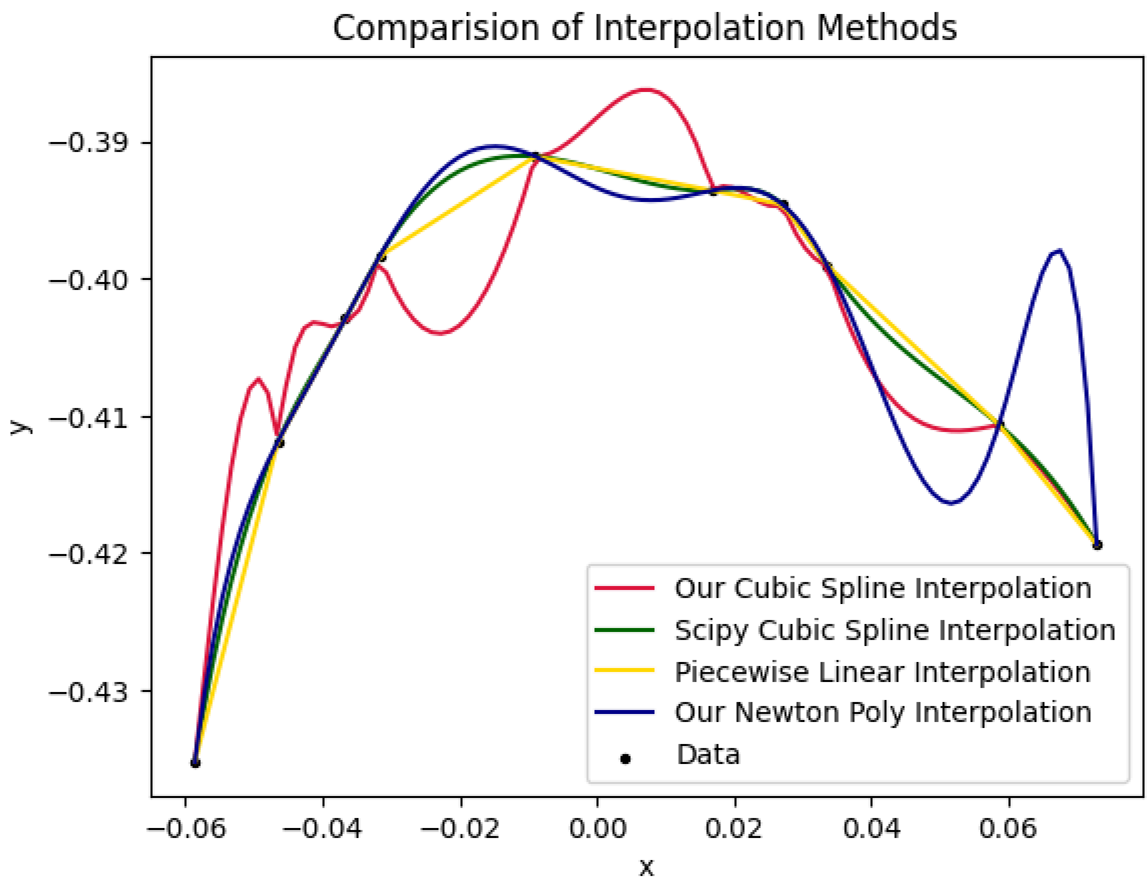
We run the same experiment for a new cut line.

In [ ]:
```
pc_downsampled = pcs_downsampled[1]
xi = pc_downsampled[:,0]
yi = pc_downsampled[:,1]
sorted_x_idx = np.argsort(xi)
xi = xi[sorted_x_idx]
yi = yi[sorted_x_idx]

x_min = xi.min()
x_max = xi.max()
num_test_pts = 100
x_test = np.linspace(x_min, x_max, num_test_pts)
```

In [ ]:
```
######################### Cubic Spline Interpolation ########################

ai,bi,ci,di = cubic_spline(xi, yi)
y_test_Ospline =  eval_cubic_spline(xi,ai,bi,ci,di,x_test)


plt.plot(x_test, y_test_Ospline, label='Our Cubic Spline Interpolation', color='crimsc

################### scipy cubic spline ####################
from scipy.interpolate import CubicSpline

cs = CubicSpline(xi, yi)
y_test_spline = cs(x_test)

plt.plot(x_test, y_test_spline, label='Scipy Cubic Spline Interpolation', color='darkg


################# Piecewise Linear interpolation ##########################
xint = np.linspace(x_min, x_max, 300)
y_test_linear = np.interp(x_test, xi, yi)

plt.plot(x_test, y_test_linear, label='Piecewise Linear Interpolation', color='gold')


############### Newton polynomial interpolation #############

c = newton_interp_poly(xi, yi)

y_test = []
for x in x_test:
  y =  eval_newton_interp_poly(c,xi,x)
  y_test.append(y)
ys = np.array(ys)

plt.plot(x_test, y_test, label='Our Newton Poly Interpolation', color='darkblue')

plt.scatter(xi,yi, label='Data', color='black', s=8)

plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Comparision of Interpolation Methods')

plt.legend()
plt.show()
```

```
best_x_tissue_2 = x_test
best_y_tissue_2 = y_test_spline
```

```
num nodes:  10
evaluating interp node 0
evaluating interp node 10
evaluating interp node 20
evaluating interp node 30
evaluating interp node 40
evaluating interp node 50
evaluating interp node 60
evaluating interp node 70
evaluating interp node 80
evaluating interp node 90
```



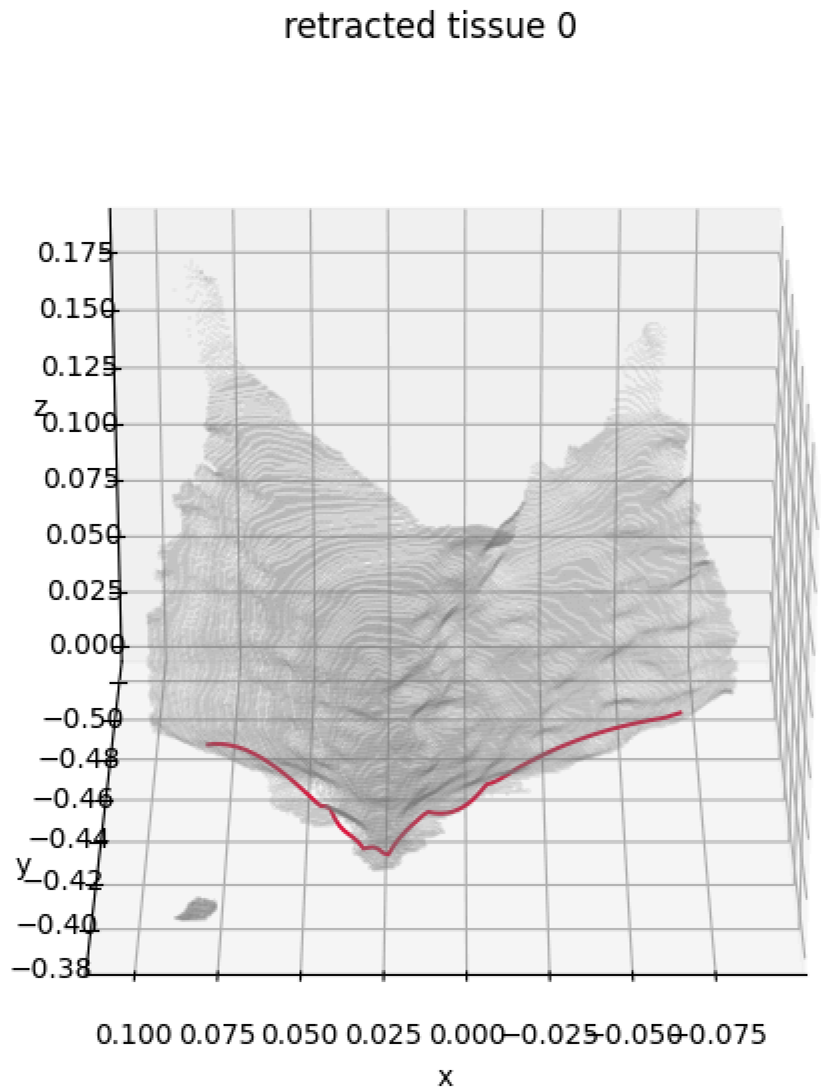
Conclusions

Precision and smoothness are the two most important factors in cutting. Based on the graphs, Scipy implementation of cubic spline interpolation is the smoothest for tissue 1 but not for tissue 0. For tissue 0, the best path was generated from our cubic spline implementation. For real-world applications, running this code will allow us to decide the best interpolation method for the given tissue qualitatively. If we had more time for this project, we would implement a method to compare paths quantitatively; that way the path can be chosen autonomously.

In [ ]:
```
# how our interpolation looks on the actual tissue.

pc_paths = [os.path.join(DATA_DIREC, "data_2.pickle"), os.path.join(DATA_DIREC, "data_
```

```
pcs = []
for pc_path in pc_paths:
  with open(pc_path, 'rb') as handle:
      data = pickle.load(handle)
  pc = np.array(data["pc"])[:,:3]
  pcs.append(pc)

pcs_filtered = []
bests = [(best_x_tissue_1, best_y_tissue_1), (best_x_tissue_2, best_y_tissue_2)]
for i in range(len(pcs)):
  pc = pcs[i]
  max_z = 0.008
  max_y = -0.379
  maskz = pc[:,2]<=max_z
  masky = pc[:,1]<=max_y
  mask = maskz & masky
  idx = np.nonzero(mask)[0]
  pc_filtered = pc[idx,:]
  pcs_filtered.append(pc_filtered)

  xs = pc_filtered[:,0]
  ys = pc_filtered[:,1]

  plot_3D_points_with_line(pc[:,0], pc[:,1], pc[:,2], f"retracted tissue {i}", bests[i
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```




```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that a
rtists whose label start with an underscore are ignored when legend() is called with
no argument.
```

retracted tissue 1