3D Vessel B-Spline Project Report

Team Members: Bryan Ellis, Adam Macgregor, Javed Sahadat

INTRODUCTION

For our project, we were tasked with implementing a 3D B-Spline curve that represents a vessel within the brain. We were given 3 sets of data: a smooth path that represents the path of the desired vessel, a diameter value for the smooth path data, and MRI data to help us create visualizations of the brain.

METHODS

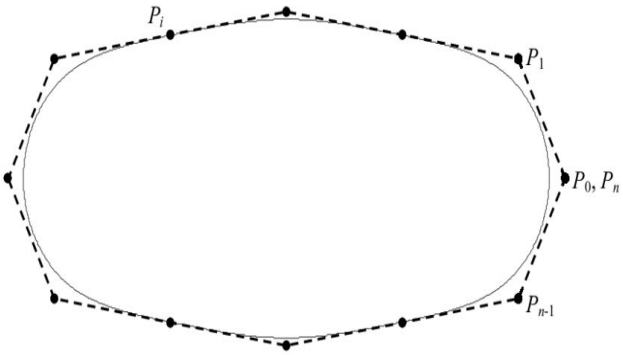
B-Splines

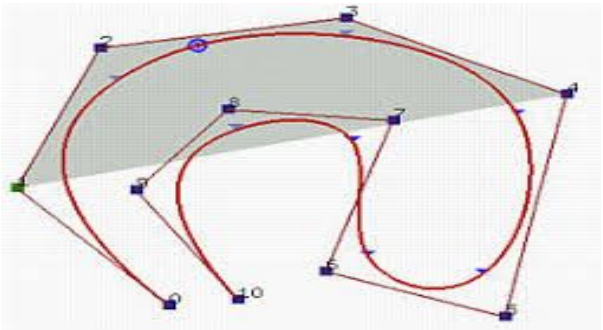
B-Splines are a combination of Bezier curve defined by a set of control points. B-Splines domains are also subdivided into points called knots. B-Splines are made of control points, knots and knot coefficients which have to satisfy the condition of continuity.

Open B-Spline Curve

B-Splines offer two important advantages over a bezier curve, which is: the domain is subdivided into knots, and the basis function will be nonzero for the whole interval. B-Spline curves will require more input information than the Bezier curve, namely *n* [(number of points) +1 control points], and *supply n* {(number of points) + p (degree) + 2 knots]. A point which a knot belongs to is defined as a knot point and they cut a B-Spline curve into segments. Control points, knots, and degree of a B-Spline curve are important because changing any of these values can drastically change the shape of the curve.

There are mainly two types of B-Spline curves which are called open curves and closed curves. Closed curves are generated by wrapping the control points or knot vectors. Open curves have the property where the first and last legs are not tangent to any knot or control point. Examples of both are shown below:





Algorithm 1 - Splines

Our project used an approach of B-Spline using weighted/basis functions for the control points. The weight functions are used to determine the total weights each control point gives. The value of u can be assigned any weight and will change the way the curve is evaluated. Since these equations are used for uniform B-Spline curve certain requirements are met such as positional continuity, which means each segment's end point is the starting for the new segment. The second constraint is tangential continuity which means there are no sudden changes in slope going from one segment to another. Lastly curvature continuity no polarity changes from segment to segment.

$$B0(u) = \frac{(1-u)^3}{6}$$

$$B1(u) = \frac{3u^3 - 6u^2 + 4}{6}$$

$$B2(u) = \frac{-3u^3 + 3u^2 + 3u + 1}{6}$$

$$B3(u) = \frac{u^3}{6}$$

Algorithm 2 - Knots

For a B-Spline curve the number of knots needed to be generated would be m +1 and m is number of points + degree + 1. The letter u in this algorithm is the number of knots. This algorithm was modified to work with 1739 points. The first degree+1 points were padded with zeros and the last degree+1 points were also padded by repeating. These knots well be uniformly spaced with algorithm 2.

$$\begin{aligned} u_0 &= u_1 = \cdots = u_p = 0 \\ u_{j+p} &= \frac{j}{n-p+1} \quad \text{for } j = 1, 2, \ldots, n-p \\ u_{m-p} &= u_{m-p+1} = \cdots = u_m = 1 \end{aligned}$$

Algorithm 3

For the evaluation of the B-Spline the Cox-De Boor's algorithm was used. The idea of this algorithm is from splines have local support meaning all the polynomials are positive in the finite domain and zero everywhere else. This algorithm evaluates in O(p^2) + O(p) time. The Cox-De Boor algorithm does directly evalue the B-Spline but it does return a value when multiplied with the control point evaluates the B-Spline. The letter t will be the knots, p is the degree, i is the index and x is position.

$$B_{i,0}(x) := \begin{cases} 1 & \text{if} \quad t_i \le x < t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{i,p}(x) := \frac{x - t_i}{t_{i+p} - t_i} B_{i,p-1}(x) + \frac{t_{i+p+1} - x}{t_{i+p+1} - t_{i+1}} B_{i+1,p-1}(x).$$

$$\alpha_{i,r} = \frac{x - t_i}{t_{i+1+p-r} - t_i}.$$

Alpha is this case is the ratio used in algorithm 3 which let you break down the control points in image 1 picture below to eventually to image 2.

Image 1

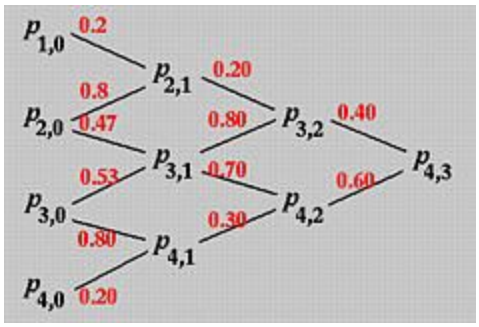
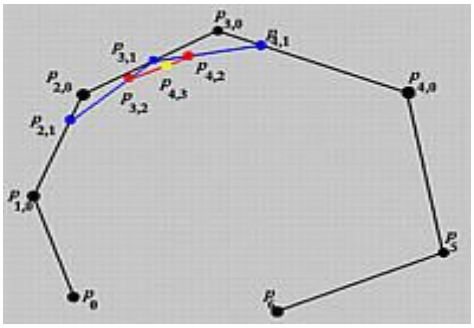


Image 2



IMPLEMENTATION

To implement the 3 algorithms above and display our project, we used Python.

Program summary:

- Initial inputs:
    - Large MRI data file that represent 2D MRI images.
    - Smooth path data file that represents a smooth path of a vessel in the brain.
    - Diameter data file that represents the diameter at each point of the vessel.
- Program steps:
    - Import data.
    - Find the control points using the smooth path data.
    - Calculate a B-Spline for using the control points.
    - Calculate the percentage error of our spline.
    - Receive user input to specify desired output.
- Output:
    - 3D graph of the vessel transposed against the 2D MRI image(s) that the user selects.

Functions

Control points function:

```
# Algorithm to find the control points
# coord is the array of values, u is the weight and index is the location in array
def find_c_points(coord, u, index):
    p1 = (((1 - u) ** 3) / 6) * coord[index]
    p2 = (((3 * u ** 3) - (6 * u ** 2) + 4) / 6) * coord[index + 1]
    p3 = (((-3 * u ** 3) + (3 * u ** 2) + (3 * u + 1)) / 6) * coord[index + 2]
    p4 = (u ** 3 / 6) * coord[index + 4]
    p5 = p1 + p2 + p3 + p4
    return p5
```

Knot function:

```
# degree is taken from the function and num is the number of control points
def find_knots(degree, num):

    # This find uniformly spaced knots with padding at the end
    # The first loop adds 0 to the first 0-degree of knots, the send loops is the
uniform space and the last loop
    # is the padding for the last four knots.
    # algorithm similar to
(https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/INT-APP/PARA-knot-generation.html)
    m = num + degree + 1
    knot = []
    k = 0
    # Padding the first p=degree values
    for a in range(degree):
        knot.append(0)
    for i in range(degree + 1, num + 1):
        knot.append(k)
        k = k + 1
    # m is defined as number of control points + degree + 1
    # Padding for the last m+1 values = number of control points-3+1 to control points
+1
    for b in range(m - degree, m + 1):
        knot.append(num - degree)
    return knot
```

Cox-De Boor's function:

```
# x_t is the query range and is defined as element belonging to (ti, ti+k)
# t_i is the index of the knot
# knot is passing the knot vector
# degree is the degree of the function
def cox_de_boors(x_t, t_i, degree, knot):
    # Implemented (https://en.wikipedia.org/wiki/De_Boor%27s_algorithm) under the local
support section
    # Local Support
```

```
    if degree == 0:
        if knot[t_i] <= x_t < knot[t_i + 1]:
            return 1
        else:
            return 0
    # Check DeBoors recursion for 0 Denominator (A/B * Recursion + C/D * Recursion)
    equation_a = 0
    equation_b = 0
    b = knot[t_i + degree] - knot[t_i]
    d = knot[t_i + degree + 1] - knot[t_i + 1]
    if b > 0:
        # if the denominator is not zero we follow the Cox-deBoors formula
        equation_a = ((x_t - knot[t_i]) / b) * cox_de_boors(x_t, t_i, degree - 1, knot)
    if d > 0:
        # if the denominator is not zero we follow the Cox-deBoors formula
        equation_b = ((knot[t_i + degree + 1] - x_t) / d) * cox_de_boors(x_t, t_i + 1,
degree - 1, knot)
    # Finally we combine both equations or add 0 to one of the equation or return 0
    final_eq = equation_a + equation_b
    return final_eq
```

B-Splines function:

```
def b_spline(c_points, n=100, degree=3, periodic=False):
    # Create a range of u values
    num_of_c_points = len(c_points)

    # function to find knots
    knots = find_knots(degree, num_of_c_points)

    # Calculate query range
    u = np.linspace(periodic, (num_of_c_points - degree), n)

    # Sample the curve at each u value
    samples = np.zeros((n, 3))
    for i in range(n):
        if not periodic:
        # returns the cox-De Boors recursion for the control point and saves them in
segment data
        # The return of recursion is multiplies by control point to get the value of
the spline
        # Final the multiplication is save in segment data
            if u[i] == num_of_c_points - degree:
                samples[i] = np.array(c_points[-1])
            else:
                for k in range(num_of_c_points):
                    samples[i] += cox_de_boors(u[i], k, degree, knots) * c_points[k]
    return samples
```

Error Checking:

```
# This is the function for calculate the RMS as the professor said during the last day
of class
def percent_error():
    p = round((len(x_axis)/len(x)))
    de = 0
    i = 0
    for j in range(0, len(x_axis), p):
        de = (x[i]-x_axis[j])**2 + (y[i]-y_axis[j])**2 + (z[i] - z_axis[j])**2 + de
        i = i+1
    de1 = math.sqrt(de/len(x))
    return de1
```

Plotting

To plot our 3D B-Spline alone, we used the matplotlib library to plot a 3D render of our spline as a line only, not using any diameter data. Next we used the mayavi library to combine the diameter data and the spline data. Finally we used both the mayavi and tvkt libraries to convert our image data into data that is usable with our 3D vessel plot.

Libraries Used

numpy

- Uses:
    - Used numpy arrays to store data.

matplotlib

- Uses:
    - Used 3D plot to show our 3D spline data without the diameter data
    - Used to read our MRI images to data

mayavi

- Uses:
    - To combine 3D Spline data and diameter data to represent the final vessel.
    - Included tvkt which is used to convert the color data of our MRI images to readable data for the mayavi library.

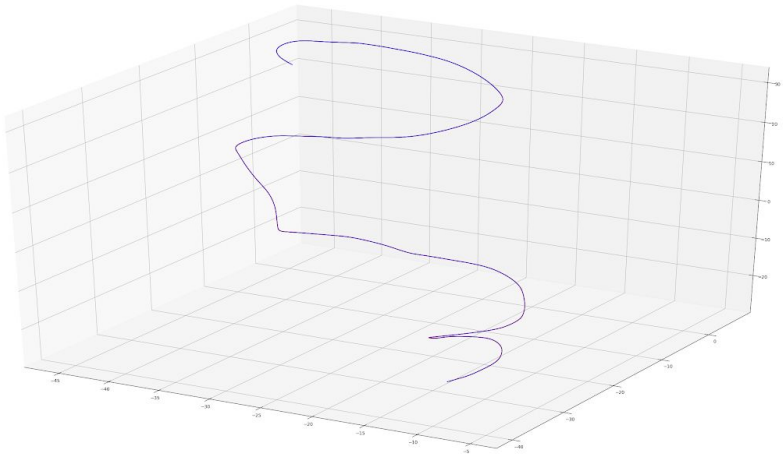
RESULTS

Percentage Error

When using 1740 data points for our spline:

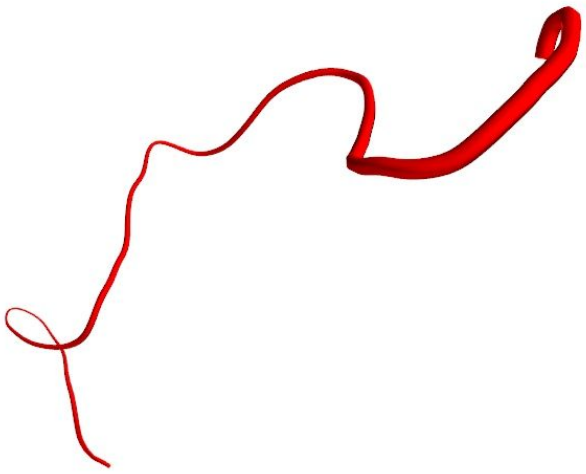
Percentage Error: 0.16718807139561817

Spline Output

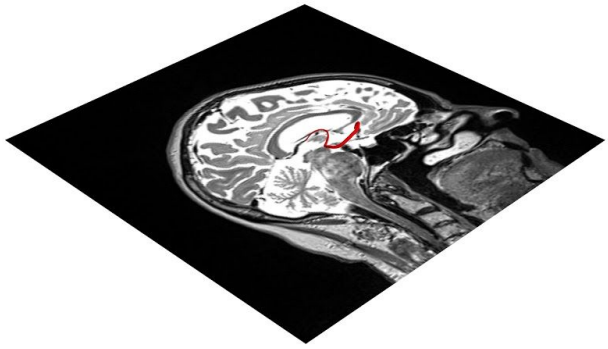


Vessel Images

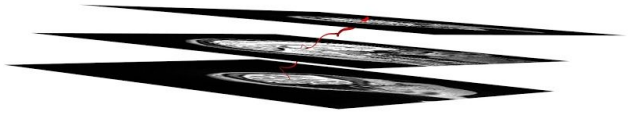
Vessel output alone:



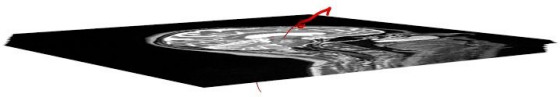
Single slice with vessel:



Multiple slices with vessel:



Slice range with vessel:



CONCLUSIONS

In conclusion, we were able to successfully create a three dimensional basis splice that represents the vessel data that was given. While we were successful in creating a valid spline, we did have some shortcomings. For one, our program takes ~30 seconds to run, we would like it to be more efficient if we had the time. Also, we were unable to discover a way to represent the whole MRI data in 3D form at once. We believe the delivered alternative is a sound alternative.

REFERENCES

1. University of Regina webpage on B-Spline curves- http://www2.cs.uregina.ca/~anima/408/Notes/Interpolation/UniformBSpline.htm
2. Knot generator - https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/INT-APP/PARA-knot-generation.html
3. Cox-De Boor's algorithm - https://en.wikipedia.org/wiki/De_Boor%27s_algorithm
4. Cox-De Boor's algorithm - https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html