MAT300/500 Programming Project
Spring 2023

Please submit all project parts on the Moodle page for MAT300 or MAT500. **Due dates are listed on the syllabus and the Moodle site.** You should include all neccessary files to recompile, and a working executable, and also a README file with instructions, all in a zipped folder (one file). You may use graphics frameworks that you have developed in other classes. Time-stamp determines the submit time, due by midnight on the due-date. Each part has a separate due date. You may incorporate all parts into the same framework with options to run them separately, or you may work out each part separately. **Late work will not be accepted.** Please note: Moodle submission due time is 0:00 hour of date one day after the posted due date. So, if the project is due at midnight on January 24, then the Moodle due date is 0:00 hour January 25. It is recommended that you submit the project by 10pm on the due date so that you are sure that it is not late, then you may resubmit until midnight.

The project consists of eight parts, each with a similar interface. For all parts except 1 and 5, you will need to have a window appear in which the user is able to click on points with the mouse. These points will be used either as control points or interpolation points which are used to render a parametric curve (roughly, these points either affect the shape of the curve or the curve is forced to pass through them.) Let's refer to these points, in either case, as *Input Points*. The algorithms used to plot the curves based on these Input Points will be summarized briefly before each part is due, giving enough detail for you to proceed with the projects. Later, we will go into more details, emphasizing the theory behind the algorithms. To illustrate the relative speed of the algorithms and evaluation methods, we will require that the Input Points always be movable with the mouse. This adds a dynamic aspect to the projects, and gives the student some feeling for the relative speed of the calculuations. For parts 1 and 5, the graph will be of a *function, not a parametric curve.* In part 1 the function will be a polynomial, and in part 5 it will be a spline, or piecewise polynomial.

Note: All parts of the project should run smoothly with up to $d = 20$ control points, or input points, or control coefficients.

User Interface Recommendations

I strongly recommend that you quickly choose a development option and become familiar with the basic UI elements such as moving a point with mouse drag, and some buttons and drop-down boxes. This should be done in the first week of the course if possible. Many students have done this series of projects with Visual Studio and Direct X and no other frameworks. Over the years, other sets of libraries and development environments have become popular, and students have used those. For example, C#, Unity, JavaScript, to name a few. My current favorite is JUCE, a great cross-platform set of libraries to do UI, graphics, and audio development in C++. I will make availabe a basic JUCE project which illustrates how to graph a function in the style of Audacity (for audio signals). Other sample code which uses C# or Javascript may also be made available.

A very important point to note is that none of the code samples should be considered as "templates", where you can follow some guidelines, fill in a few blocks of code, and be done. Rather, these samples are meant to illustrate how certain things can be done, which you might learn from and possibly re-use parts of it for your project. Please do NOT ask questions about the code samples which imply that you are trying to use them to understand the project assignment. The project assignment may change slightly from year to year, and your job is to understand the written requirements, ask questions, and work out your implementation, with or without the other code samples. My personal recommendation is to work from scratch, and only to consult various code samples in order to get ideas or to explain some particular functionality. This way you will learn the most and be most confident in your final work.

Part I. De Casteljau Algorithm for Polynomial Functions

In this part you will illustrate how the coefficients of Bernstein polynomials affect the shape of a polynomial graph. The starting point is a window that can accomodate a graph of a function for t-values on the interval $[0, 1]$, and y-values between -3 and 3. The relative scale of the axes should be stretched in the t-direction so that $[0, 1]$ has the same length as $[-3, 3]$ in the y-direction. The interface should also have a button which changes the degree, which should default to $d = 1$. The default graph should be the graph with all coefficients equal to 1, which simply gives output 1 for all t. On the line $y = 1$ should appear $d + 1$ equally-spaced dots, the first and last of which are on the lines $t = 0$ and $t = 1$, which are moveable up and down. These dots correspond to the coefficients $a_0, \ldots, a_d$ of the Bernstein polynomials of degree d. These dots should be movable *up and down only* with the mouse. As the dots are moved a polynomial function $p(t)$ is regraphed in the window in real time. A large number of t-values should be chosen so the graphs look smooth but the computations are still fast. The general polynomial is simply a sum of the Bernstein polynomials of degree d:

$$p(t) = p_{[a_0, a_1 \ldots a_d]}(t) = \sum_{i=0}^{d} a_i B_i^d(t).$$

There should also be a drop-down menu which allows the choice of NLI or BB-form. This means that the calculation of $p(t)$ will be done in one of two different ways: i) De Casteljau algorithm, or Nested Linear Interpolation (NLI), or ii) BB-form. In the NLI method, the control coefficients $a_0, \ldots, a_d$ are playing the role of what will be the control points in the next part. The NLI-form as a recursive function looks like

$$p(t) = p_{[a_0, a_1 \ldots a_d]}(t) = (1 - t)p_{[a_0, a_1 \ldots a_{d-1}]}(t) + tp_{[a_1 \ldots a_d]}(t),$$

where the subscripts indicate the collection of coefficients that define that particular polynomial. This can also be represented as an array of numbers for any particular t-value:

$$\begin{array}{ccccc}
a_0^0 & & & & \\
& a_0^1 & & & \\
a_1^0 & & a_0^2 & & \\
& a_1^1 & & a_0^3 & \\
a_2^0 & & a_1^2 & & \\
& a_2^1 & & & \\
a_3^0 & & & &
\end{array}$$

For instance, to obtain the value a_0^1 we add together the values to the upper and lower left of this one, with the coefficients $(1-t)$ on the upper and t on the lower, to obtain:

$$a_0^1 = (1-t)a_0^0 + ta_1^0.$$

The same pattern follows for all of the values, with last one giving:

$$a_0^3 = (1-t)a_0^2 + ta_1^2 = p(t).$$

Each such array corresponds to one t-value. Computing $p(t)$ by this method is called Nested Linear Interpolation.

The other option is to compute with the BB-form, which is the sum of Bernstein polynomials:

$$p(t) = p_{[a_0,a_1\ldots a_d]}(t) = \sum_{i=0}^{d} a_i B_i^d(t) = \sum_{i=0}^{d} a_i \binom{d}{i} (1-t)^{d-i} t^i.$$

Again, a large number of t-values should be chosen so that the graph looks smooth but the calculations can still be done in real time. The points on the graph which are computed should be joined by small line segments to complete the graph.

Note: the binomial coefficients can be defined with factorials by $\binom{d}{i} = \frac{d!}{(d-i)!i!}$, however they are much more efficiently computed with the Pascal recursion identity:

$$\binom{d}{i} = \binom{d-1}{i-1} + \binom{d-1}{i}.$$

Part II: De Casteljau Algorithm for Bezier Curves

In this part you will implement the De Casteljau Algorithm for Bezier Curves, also known as Nested Linear Interpolation. If the control points are labelled P_0, $P_1 \ldots P_d$ then we are computing the curve $\gamma(t) = \gamma_{[P_0,P_1\ldots P_d]}(t)$ for $0 \le t \le 1$. An appropriate approximation is used. For instance, a finite collection of points (on or close to the curve) is computed and then the piecewise linear path between these points is plotted on the screen. This finite set of points can be computed by choosing a finite set of t-values (between 0 and 1) and using either nested linear interpolation (NLI-form) or the Bernstein polynomials (BB-form) to compute the corresponding points $\gamma(t)$. A third method is mid-point subdivision. Here the finite set of points to connect are not all on the curve. They are either on the curve, or on a tangent line to the curve and very close to the point of contact. You are required to do all three of the methods in this part of the project. We will now describe in more detail each of the computation methods:

1. NLI-form: $\gamma(t) = \gamma_{[P_0,P_1\ldots P_d]}(t) = (1-t)\gamma_{[P_0,P_1\ldots P_{d-1}]}(t) + t\gamma_{[P_1\ldots P_d]}(t)$.

   The base case is $\gamma_{[P_0,P_1]}(t) = (1-t)P_0 + tP_1$. The next case is $\gamma_{[P_0,P_1,P_2]}(t) = (1-t)\gamma_{[P_0,P_1]}(t) + t_{[P_1,P_2]}(t)$. For each t value, the computation of $\gamma(t)$ can be visualized using the Bezier point array. The first stage Bezier points are the control points: $P_i = P_i^0$. The higher stages

$k = 1, \ldots, d$ are given by the recursion: $P_i^k = (1-t)P_i^{k-1} + tP_{i+1}^{k-1}$. Thus, to the right of any two consecutive Bezier points (meaning that their lower indices are consecutive) is the point which is obtained by linear interpolation along the line segment between the two points. The distance along the line segment is proportional to the value t. This value t stays the same for the entire array, so that a new array must be constructed for each new t value. Here is such an array of Bezier Points for a degree three Bezier curve:

$$\begin{array}{cccc} P_0^0 & & & \\ & P_0^1 & & \\ P_1^0 & & P_0^2 & \\ & P_1^1 & & P_0^3 \\ P_2^0 & & P_1^2 & \\ & P_2^1 & & \\ P_3^0 & & & \end{array}$$

The collection of line segements joining the control points together is called the control polyline. The collection of line segments joining each of the consecutive Bezier points for each stage is called the shells. For the array above, the polyline consists of three connected line segments, and the shells consist of this polyline plus the two line segments joining the consecutive stage one Bezier points and the one line segment joining the two stage two Bezier points. The point P_0^3 is a distinguished point on this last line segment, since it is equal to $\gamma(t)$.

2. BB-form $\gamma(t) = \gamma_{[P_0, P_1 \ldots P_d]}(t) = \sum_{i=0}^{d} B_i^d(t) P_i$. The $B_i^d(t) = \binom{d}{i}(1-t)^{d-i}t^i, \quad i = 0, \ldots, d.$

3. Midpoint Subdivision

   The first approximation to the curve $\gamma(t)$ is just the set of control points joined by line segments. Let's represent this approximation by:

   $$P_0^0 \longrightarrow P_1^0 \longrightarrow P_2^0 \longrightarrow P_3^0.$$

   The easiest way to describe the points used in the second approximation in mid-point subdivision is to use the Bezier point array for $\gamma(\frac{1}{2})$. (See the triangular array above, with $t = \frac{1}{2}$ implicit.) The points used in the second approximation are just the points on the upper and lower edges of this triangular array. This includes the middle point $\gamma(\frac{1}{2}) = P_0^3$. Thus the second approximation is the set of points $P_0^0, P_0^1, P_0^2, P_0^3, P_1^2, P_2^1, P_3^0$ joined by line segments. We can represent this approximation as:

   $$P_0^0 \longrightarrow P_0^1 \longrightarrow P_0^2 \longrightarrow P_0^3 \longrightarrow P_1^2 \longrightarrow P_2^1 \longrightarrow P_3^0.$$

   The process of going from the first to the second approximation is called the subdivision step.

   For the third approximation, we note that the upper and lower edges of the above triangle can each form a new set of control points for a Bezier curve. We now just repeat the subdivision step on each of these new sets of control points ($P_0^0, P_0^1, P_0^2, P_0^3$ and $P_0^3, P_1^2, P_2^1, P_3^0$) to obtain

the next approximation. If we relable these points as $P_0^0 = Q_0, P_0^1 = Q_1, P_0^2 = Q_2, P_0^3 = Q_3$ and $P_0^3 = R_0, P_1^2 = R_1, P_2^1 = R_2, P_3^0 = R_3$, (with $Q_3 = R_0$) then we can simply iterate the subdivision step for the Q_i and R_i. Thus the next approximation will be:

$$Q_0^0 \longrightarrow Q_0^1 \longrightarrow Q_0^2 \longrightarrow Q_0^3 \longrightarrow Q_1^2 \longrightarrow Q_2^1 \longrightarrow Q_3^0$$

$$R_0^0 \longrightarrow R_0^1 \longrightarrow R_0^2 \longrightarrow R_0^3 \longrightarrow R_1^2 \longrightarrow R_2^1 \longrightarrow R_3^0$$

which can be joined together (since $Q_0^3 = Q_3 = R_0 = R_0^0$) to obtain:

$$Q_0^0 \longrightarrow Q_0^1 \longrightarrow Q_0^2 \longrightarrow Q_0^3 \longrightarrow Q_1^2 \longrightarrow Q_2^1 \longrightarrow Q_3^0 \longrightarrow R_0^1 \longrightarrow R_0^2 \longrightarrow R_0^3 \longrightarrow R_1^2 \longrightarrow R_2^1 \longrightarrow R_3^0$$

If we let $k = 1$ for the first approximation, $k = 2$ for the second, and $k = 3$ for the third, then there are $2^{(k-1)} \cdot 3 + 1$ points to connect with line segments for each approximation. This pattern persists for all k.

In this project, you should determine a reasonable value of k in order to produce a nice curve but still allow for speed of moving control points.

Further specifications:

- Input: Use mouse input for control points.
- Output: Display the Bezier Curve together with the control points and the control polyline. Use a different color for the polyline.
- Make the control points movable with the mouse, so that the user can click on a point and move a control point to a new location. The curve should be re-rendered in real time as the control point moves.
- For the NLI computation option, display the nested linear pieces (or shells) and Bezier points for the default value $t = 1/2$ with other colors. This should toggle on or off as part of the display.
- make the t-value movable in the previous part so that any $0 \leq t \leq 1$ can be chosen so that the shells and Bezier points are displayed. (This feature is best implemented with a slider that can be grabbed with the mouse.)
- Use a drop-down menu (or other selection control) to implement the choice of three methods of evaluation to plot points: 1) Nested Linear Interpolation, 2) direct evaluation of Bernstein polynomials, 3) Midpoint Subdivision. The drop-down menu will give the choice of which method is being used. (Make this option to change the computation method while still keeping the same control points that have been clicked by the user.)

Part III - Interpolating Polynomials

In this part we do interpolation using polynomials.

- Input: Points clicked on screen.
- Output: Parametric Polynomial Curve passing through the points.

The interpolating polynomial function of the desired form is guaranteed to exist and is unique. It is possible to set up a linear system to solve for this function, using the unknown coefficients as the variables of the system. Each equation of the system corresponds to one interpolation point. Since a polynomial of degree $\leq d$ has $d+1$ free coefficients, it can be forced to go through (interpolate) $d+1$ points. It is simpler, however, to use the Newton form since it is more efficient than solving a linear system.

For the polynomial output curve, use the Newton form for each coordinate $x(t)$ and $y(t)$. The t_i-values can be simply $t_0 = 0, t_1 = 1, t_2 = 2, \ldots, t_d = d$. The basic interval on which to evaluate the curve is now $[0, d]$. The $x(0)$ value is just the x-coordinate of the first point clicked, and similarly for the y-value. The input points (not control points now, but they could be called interpolation points) clicked by the user are then $P_0, \ldots, P_d$. As points are clicked, two separate Newton forms are built for $x(t)$ and $y(t)$. No linear system is neccessary. To conform to standard interpolation notation, we pretend that a function called $g(t)$ has produced the data values to be interpolated, when in fact they are coming from the points clicked by the user. Let $P_i = (a_i, b_i)$ be an input point. To find the Newton form for $x(t)$ we will assume that g has the values: $g(i) = a_i$ (and for $y(t)$ $g(i) = b_i$) for $i = 0, \ldots, d$. The base case for divided differences is: $[a]g = g(a)$ for any a. The next case is $[a, b]g = \dfrac{[b]g - [a]g}{b - a}$, for any a, b, hence the name divided difference.

Newton form:

$$p(t) = [0]g + [0, 1]gt + [0, 1, 2]gt(t-1) + \cdots + [0, 1, 2, \ldots, d]g \cdot t(t-1)\cdots(t-d+1).$$

Recursion for Divided Difference coefficients:

$$[t_i, t_{i+1}, \ldots, t_{i+k}]g = \frac{[t_{i+1}, t_{i+2}, \ldots, t_{i+k}]g - [t_i, t_{i+1}, \ldots, t_{i+k-1}]g}{t_{i+k} - t_i}.$$

Note: The Newton form can be given with the input points in any order. The coefficients and the terms (basis polynomials) will be different, but the polynomial is actually identical (this can be checked by expanding it out and comparing.) This fact is useful for this project since the user may click on any point and move it around. The Newton form should quickly be recomputed without that point (being clicked) and then the new points encountered by dragging the mouse around will always be the last points in the new Newton forms for the full set of points. This way, in the process of dragging points, there is always just one new point, hence one additional term in the Newton form.

Part IV - Interpolating Cubic Splines

For the interpolating spline, a linear system is solved using the standard basis spline functions $1, t, t^2, t^3$, $(t-1)^3_+$, $(t-2)^3_+$, $(t-3)^3_+$, etc., for each coordinate $x(t)$ and $y(t)$. Again the t-values can be simply $0, 1, 2, 3, \ldots, k$. The $x(0)$ value is again just the x-coordinate of the first point clicked, and similarly for the y-value. Given n interpolation points, the vector space of cubic splines has dimension $n + 2$.

A spline function can be written in the form:

$$f(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + b_1 (t-1)_+^3 + b_2 (t-2)_+^3 + \cdots + b_{k-1}(t-(k-1))_+^3.$$

Here we define the basic shifted (or truncated) power function as:

$$(t-c)_+^d = \begin{cases} 0, & t < c \\ (t-c)^d, & t \geq c \end{cases}.$$

We count $k - 1 + 4 = k + 3$ free coefficients. If we let $n = k + 1$ be the number of interpolation points, then there are $n + 2$ free coefficients.

There are two free parameters (n+2 in all) that are not accounted for. These last two free coefficients are determined (in the case of the natural cubic spline) by setting $f''(0) = f''(k) = 0$.

Solving for the two splines $x(t)$ and $y(t)$ involves solving two $(n+2) \times (n+2)$ linear systems. Since the input points will accumulate, it is worth trying to use part of the solution to the linear system for k points in the solution to the linear system for $k + 1$ points. Another approach is to solve the linear systems as a preprocessing step and save the coefficient matrices to be used in the program.

Part V: De Boor Algorithm for spline functions

In this part you will plot a sum of B-spline functions similar to the plot of Bernstein polynomials in the very first project. In this case you will use a sequence of intervals $[0, 1, 2, 3, 4, \ldots, N]$.

For the case of degree $d = 1$ there will be a dot on the line $y = 1$ above each integer point in the list $1, 2, 3, 4, \ldots, N - 1$. These dots will again be the coefficients which are moveable. Here they represent the coefficients of the B-spline functions, which for degree $d = 1$ are hat functions labelled:

$$\mathcal{B}_0^1(t), \ldots, \mathcal{B}_{N-2}^1(t).$$

For the case degree $d = 2$ there will be a dot on the line $y = 1$ above each half-integer point in the list $1.5, 2.5, 3.5, \ldots, 1.5 + N - 3 = N - 1.5$, representing the coefficients of the quadratic B-spline functions

$$\mathcal{B}_0^2(t), \ldots, \mathcal{B}_{N-3}^2(t).$$

For the higher degree d cases there will be a dot on the line $y = 1$ above each integer or half-integer point in the list

$$\frac{d+1}{2}, \frac{d+1}{2} + 1, \frac{d+1}{2} + 2, \ldots, \frac{d+1}{2} + N - d - 1 = N - \frac{d+1}{2},$$

representing the coefficients of the B-spline functions

$$\mathcal{B}_0^d(t), \ldots, \mathcal{B}_{N-d-1}^d(t).$$

In order that there are sufficiently many intervals, you should set $N = d + 10$ as a default, but also make N adjustable with a user interface.

Along with the the dots representing coefficients of the B-spline functions, you should also draw the initial graph, which is simply a line segment of the graph $y = 1$ above the interval $[d, N - d]$. This is also the actual value of the sum:

$$\mathcal{B}_0^d(t) + \mathcal{B}_1^d(t) + \cdots + \mathcal{B}_{N-d-1}^d(t), \quad \text{for} \quad t \in [d, N-d].$$

Then, as the user clicks and drags one of the points on the line $y = 1$, we are changing the coefficient corresponding to that B-spline, and the sum now becomes:

$$f(t) = c_0\mathcal{B}_0^d(t) + c_1\mathcal{B}_1^d(t) + \cdots + c_{N-d-1}\mathcal{B}_{N-d-1}^d(t), \quad \text{for} \quad t \in [d, N-d]$$

where the values of the coefficients c_i are allowed to change.

You should implement three different methods to evaluate the above sum:

1. De Boor Algorithm
2. Divided Differences
3. Sum of Shifted Power Functions

In the first case, the De Boor Algorithm consists of Nested Linear Interpolation to compute the coefficients in the following steps: (Note: In this project, we have the simple assumption that $t_i = i$ for all i.)

- (Stage zero:) Set $c_i^{[0]} = c_i$ for $i = 0, \ldots, N - d - 1$.
- For $t \in [d, N - d)$ set $J = \lfloor t \rfloor$ (floor function), so that $t \in [J, J + 1)$.
- (Stage p:) For $p = 1, \ldots, d$, for $i = J - d + p, \ldots, J$: set

$$c_i^{[p]} = \frac{t - t_i}{t_{i+d-(p-1)} - t_i} c_i^{[p-1]} + \frac{t_{i+d-(p-1)} - t}{t_{i+d-(p-1)} - t_i} c_{i-1}^{[p-1]}$$

$$= \frac{t - i}{d - p + 1} c_i^{[p-1]} + \frac{i + d - p + 1 - t}{d - p + 1} c_{i-1}^{[p-1]}.$$

- $f(t) = c_J^{[d]}$.

In the second case, the B-spline functions are defined as divided differences:

Given a knot sequence $\mathbf{t}$ with $t_0 \leq t_1 \leq \ldots \leq t_N$, we define for any degree $d \geq 0$, and for $0 \leq i \leq N - d - 1$:

$$\mathcal{B}_i^d(t) = (-1)^{d+1}(t_{i+d+1} - t_i)[t_i, t_{i+1}, \ldots, t_{i+d+1}](t - x)_+^d$$

where $(t-x)_+^d$ is the left-sided shifted power function of x, with t constant, which is equal to $(t-x)^d$ for $x \leq t$, and is equal to zero for $x > t$.

In this case, again we assume $t_i = i$ which simplifies the formulas. The value is computed with the divided difference table.

In the third case, we use Cramer's Rule and a determinant to compute the above divided difference, and then rewrite it as a sum of shifted power functions:

$$\begin{aligned}
\mathcal{B}_i^d(t) &= (-1)^{d+1}(t_{i+d+1}-t_i)[t_i,\ldots,t_{i+d+1}](t-x)_+^d \\
&= (-1)^{d+1}(t_{i+d+1}-t_i)\frac{1}{D}\begin{vmatrix} 1 & t_i & t_i^2 & \cdots & t_i^d & (t-t_i)_+^d \\ 1 & t_{i+1} & t_{i+1}^2 & \cdots & t_{i+1}^d & (t-t_{i+1})_+^d \\ \vdots & \vdots & \cdots & \vdots & \vdots & \\ 1 & t_{i+d+1} & t_{i+d+1}^2 & \cdots & t_{i+d+1}^d & (t-t_{i+d+1})_+^d \end{vmatrix}
\end{aligned}$$

where D is the Vandermonde determinant:

$$D = \begin{vmatrix} 1 & t_i & t_i^2 & \cdots & t_i^d & t_i^{d+1} \\ 1 & t_{i+1} & t_{i+1}^2 & \cdots & t_{i+1}^d & t_{i+1}^{d+1} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \\ 1 & t_{i+d+1} & t_{i+d+1}^2 & \cdots & t_{i+d+1}^d & t_{i+d+1}^{d+1} \end{vmatrix}.$$

By performing a cofactor expansion of the previous determinant along the last column, we can write the B-spline as a sum:

$$\mathcal{B}_i^d(t) = a_i(t-t_i)_+^d + a_{i+1}(t-t_{i+1})_+^d + \cdots + a_{i+d+1}(t-t_{i+d+1})_+^d.$$

Note: each coefficient a_j is the result of a Vandermonde Determinant calculation, so can be written as a product of backward differences.

Part VI: De Boor Algorithm for Polynomial Curves

Implement The De Boor Algorithm for a parametric polynomial curve.

This project follows the basic input and graphical output of the second assignment (Part II) which used Bezier Curves.

The user will input points P_0, $P_1,\ldots,P_d$, and, just as in the first project, the program will generate the Bezier curve and control polyline. In addition to this, however, the program will now generate:

- the polar form $F[u_1,\ldots,u_d]$ obtained from the BB-form, and

- the knot sequence: $[0, 0, \ldots, 0, 1, 1, \ldots, 1]$ consisting of $d+1$ zeros and $d+1$ ones.

At this stage, the control points should be movable, just as in project I.

It is not necessary to display the polar form, but the default knot sequence should be displayed in a text box as a comma-delimited list.

Next, there should be a button which changes the knot sequence to a new default:

$$[t_0, t_1, t_2, t_3, \ldots, t_N] = [-\frac{d}{10}, -\frac{d-1}{10}, \ldots, -\frac{2}{10}, -\frac{1}{10}, 0, 1, 1+\frac{1}{10}, 1+\frac{2}{10}, \ldots, 1+\frac{d}{10}],$$

where the length of the knot sequence is: $N+1 = 2d+2$ which is twice the number of control points above. Also, since $N = 2d+1$, the number of B-splines for this knot sequence is $d+1 = N-d$.

When this button is clicked, there should be a new set of DeBoor control points $Q_0, \ldots, Q_d$, displayed on the screen, with a new polyline connecting them in order. Those points are obtained from the polar form simply by evaluating:

$$\begin{aligned}
Q_0 &= F[-\frac{d-1}{10}, -\frac{d-2}{10}, \ldots, -\frac{1}{10}, 0] = F[t_1, \ldots, t_d] \\
Q_1 &= F[-\frac{d-2}{10}, -\frac{d-3}{10}, \ldots, -\frac{1}{10}, 0, 1] = F[t_2, \ldots, t_{d+1}] \\
Q_2 &= F[-\frac{d-3}{10}, -\frac{d-4}{10}, \ldots, -\frac{1}{10}, 0, 1, 1+\frac{1}{10}] = F[t_3, \ldots, t_{d+2}] \\
&\vdots \\
Q_d &= F[1, 1+\frac{1}{10}, 1+\frac{2}{1}, \ldots, 1+\frac{d-1}{10}] = F[t_{d+1}, \ldots, t_{2d}]
\end{aligned}$$

Note: the first and last knot values t_0 and t_N are not used in the calculuation of the points $Q_0, \ldots, Q_d$, however, they are used in the definition of the B-splines $\mathcal{B}_0(t), \ldots, \mathcal{B}_d(t)$, and hence they will be used in the calculation of a B-spline sum.

At this stage, the new DeBoor control points should be movable, with the same click and drag interface as is used for the Bezier control points. However, a different algorithm is used in the case of these control points. It is still nested linear interpolation, but the weights of the points change.

First set $Q_i^{[0]} = Q_i$, for $i = 0, \ldots, d$.

Then, for $t \in [0, 1]$, for $p = 1, \ldots, d$, and for $i = p, \ldots, d$: set

$$Q_i^{[p]} = \frac{t - t_i}{t_{i+d-(p-1)} - t_i} Q_i^{[p-1]} + \frac{t_{i+d-(p-1)} - t}{t_{i+d-(p-1)} - t_i} Q_{i-1}^{[p-1]}.$$

Then $\gamma(t) = Q_d^{[d]}$.

Part VII: De Boor Algorithm for B-Spline Curves

Implement The De Boor Algorithm (or Recursive B-Spline Algorithm) for parametric curves.

This project follows the basic input and graphical output of the second assignment (Part II) which used Bezier Curves.

The Algorithm:

1. Knot sequence $\mathbf{t} = \{t_0, \ldots, t_N\}$, with $t_0 \leq t_1 \leq \ldots \leq t_N$, degree d, curve $\gamma(t)$ defined for $t \in [t_d, t_{N-d})$.
2. Input points: $P_0, \ldots, P_s$. These will define the control points of your curve, just as in the Bezier case. The subscript 's' will be determined by the user, by entering $s+1$ points. The following steps will compute points on the B-spline curve $\gamma(t) = \sum_{i=0}^{s} P_i \mathcal{B}_i^d(t)$ without explicity evaluating the B-spline functions $\mathcal{B}_i^d(t)$.
3. (Stage zero:) Set $P_i^{[0]} = P_i$ for $i = 0, \ldots, s$.
4. For $t \in [t_d, t_{N-d})$ find J so that $t \in [t_J, t_{J+1})$.
5. (Stage p:) For $p = 1, \ldots, d$, for $i = J-d+p, \ldots, J$: set
$$P_i^{[p]} = \frac{t - t_i}{t_{i+d-(p-1)} - t_i} P_i^{[p-1]} + \frac{t_{i+d-(p-1)} - t}{t_{i+d-(p-1)} - t_i} P_{i-1}^{[p-1]}.$$
6. $\gamma(t) = P_J^{[d]}$.
7. Your program will plot the curve with t-values in the interval $[t_d, t_{N-d})$ using the above algorithm for each t-value chosen.

Specifications:

1. Use default degree $d = 3$. Also, allow for this to be changed with your interface.
2. Use the simple default knot sequence $\mathbf{t}$ with $t_i = i$, and length $N+1$. The sequence $\mathbf{t}$ is then $t_0 = 0, \ldots, t_N = N$ with N determined by $s = N - d - 1$, ie. $N = s + d + 1$.
3. Allow knot sequence to be entered by the user. Choice of knot sequence needs to be consistent with the degree and number of control points. For example, if N is not equal to $s+d+1$, then an extension of knots can be added to the sequence, taking the next value in the standard sequence of whole numbers. This means that the user can enter a partial knot sequence, and the rest of the sequence is given some default values.
4. Example: (can be done with the knot sequence entered by user, or as a preset) "continuity" at end-points:

   Use the basic d-extension of the knot sequence $\{0, 1, 2, \ldots, k\}$ so that we get the form:
$$\{0, 0, 0, \ldots, 0, 1, 2, \ldots, k-1, k, k, \ldots, k\},$$

ie. the knot sequence with break points $\{1, 2, \ldots, k-1\}$, and end points 0 and k. The multiplicities are (including 0 and k):

$$\{d+1, 1, 1, \ldots, 1, 1, d+1\}.$$

(Then $N = k - 1 + 2(d+1) = k + 2d + 1$, and $s = k + d$.) This has the effect of making the curve satisfy: $\gamma(0) = P_0$ and $\gamma(1) = P_s$.

5. Implement a slider for t so that using the mouse, the t-values can be dragged back and forth and the point on the curve moves accordingly. This slider will now need to display the basic interval $[t_d, t_{N-d})$.

6. Render the shells, ie. successive linear segments, used for any particular t-value, and have these move according to the t-slider also. Remember: only $d+1$ control points are used in the calculation of any particular point on the curve.

Part VIII: 3D version of previous project II, III, IV, or VII.

This project should implement one of the above projects in 3D, with a user interface that allows for movement of control points in three dimensions. The viewpoint of the curve should also be moveable.