Mapana Journal of Sciences
2021, Vol. 20, No. 1, 79-103
ISSN 0975-3303|https://doi.org/10.12723/mjs.56.5

Bernstein Polynomials: Properties and Applications to Bezier Curves, B-Splines, and Solution of Boundary Value Problems

M N Anandaram*

Abstract

Bernstein polynomials (also known as B-polys) have excellent properties allowing them to be used as basis functions in many applications in physics. This paper provides a brief tutorial description of their properties, and their use in obtaining B-polys, B-splines or Basis spline functions, Bezier curves, and ODE solution curves is computationally demonstrated. Also, an example is described showing their application for solving the fourth order BVP relating to the bending at the free end of a cantilever.

Keywords: Bernstein polynomials (B-polys), B-Splines, Bezier Curves, BVP solution

1. Introduction to Bernstein Basis Polynomials

Over a century ago, the Russian mathematician Sergei Natanovich Bernstein (1880-1968) discovered many properties and uses of Bernstein polynomials. They were originally used for constructing Bezier curves (see next section) in computer graphics, but their properties later enabled their use as basis functions for the solution of many types of differential equations (see [1] to [5], [8], [10] [11] and many more references therein).

*Professor of Physics (Retired), Bangalore University, Bangalore, India; mnanandaram@gmail.com

The binomial theorem describes the algebraic expansion of a binomial $\boldsymbol{x+y}$ by

$$(\boldsymbol{x+y})^{\boldsymbol{p}} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{p}} \binom{\boldsymbol{p}}{\boldsymbol{i}} \boldsymbol{x^i y^{p-i}} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{p}} \binom{\boldsymbol{p}}{\boldsymbol{p-i}} \boldsymbol{x^{p-i} y^i} \quad (1.1)$$

Here $\binom{\boldsymbol{p}}{\boldsymbol{i}}$is the binomial ("p choose i" or combinational, pCi) coefficient of each term and is given by

$$\binom{\boldsymbol{p}}{\boldsymbol{i}} = \boldsymbol{pCi} = \boldsymbol{C(p,i)} = \frac{\boldsymbol{p!}}{\boldsymbol{i!(p-i)!}} \quad (1.2)$$

It can be seen after defining $\boldsymbol{y \equiv (1-x)}$that,

$$\boldsymbol{1 = 1^p = \big(x + (1-x)\big)^p = \sum_{i=0}^{p} C(p,i) x^i (1-x)^{p-i} = \sum_{i=0}^{p} B(p,i,x)} \quad (1.3)$$

In (1.3), $\boldsymbol{B(p,i,x)}$ is known as the $\boldsymbol{i^{th}}$ Bernstein basis polynomial, or B-poly of degree p for any x in the domain range [0,1] such that $\boldsymbol{0 \leq x \leq 1}$and is defined by,

$$\boldsymbol{B(p,i,x) \equiv C(p,i) x^i (1-x)^{p-i}}, \qquad \boldsymbol{0 \leq i \leq p}, \qquad \boldsymbol{x \in [0,1]} \quad (1.4)$$

The above expression is the $\boldsymbol{i^{th}}$ B-polynomial and there exist polynomials such as $\boldsymbol{k = p+1}$ (where k is the order) for a B-polynomial of degree p (order k) with $\boldsymbol{i = 0, 1, 2, \ldots p}$, $(\boldsymbol{i = 1, 2, \ldots k})$. In (1.3), it is seen that all the B-polys are of unity weight, they are used in an unmodified manner as such and that the sum of all B-polys of any given degree equals unity. In many applications, a given function is to be approximated by B-polynomials in a generalized domain $[\boldsymbol{a}, \boldsymbol{b}]$of which$[\boldsymbol{a = 0}, \boldsymbol{b = 1}]$ is a special case. In that case, (1.3) takes the form

$$\boldsymbol{1 = ((x-a) + (b-x))^n (b-a)^{-p} = \sum_{i=0}^{p} C(p,i) (x-a)^i (b-x)^{p-i} (b-a)^{-p}} \quad (1.5)$$

Obviously, the definition (1.4) is now modified as,

$$\boldsymbol{B(p,i,a,b,x) \equiv C(p,i) (x-a)^i (b-x)^{p-i} / (b-a)^p}, \boldsymbol{x \in [a,b]} \quad (1.6)$$

Some of the properties ([2]) of the B-polynomials are outlined below.

They are symmetric for any x: $\boldsymbol{B(p,i,a,b,x) = B(p,p-i,a,b,1-x)}$.

Both (1.3) and (1.5) show that they form a partition of unity: $\sum_{i=0}^{n} \boldsymbol{B(n,i,a,b,x) = 1}$, that is, the sum of all (k=p+1) $\boldsymbol{p^{th}}$degree B-polynomials at any point x is unity over the entire domain. Eqn. (1.6) shows that the first ($\boldsymbol{0^{th}}$)basis polynomial of $\boldsymbol{p^{th}}$degree has a value of unity at the left domain limit and the last ($\boldsymbol{k^{th} = (p+1)^{th}}$) basis polynomial has a value of unity at the right domain limit, that is, $\boldsymbol{B(p,i=0,a,b,x=a) = 1}$, and, $\boldsymbol{B(p,i=p,a,b,x=b) = 1}$, respectively. In the domain interior $\boldsymbol{a < x < b}$ each of the B-polynomials has a unique local maximum at $\boldsymbol{x = i/p}$ with a value given by,

$$\boldsymbol{B(p,i,a,b,x=i/p) = C(p,i) i^{i} (p-i)^{p-i} p^{-p}} \qquad (1.7)$$

All B-polynomials are conveniently defined to vanish outside their domain boundary limits. Thus, the set of (p+1) $\boldsymbol{p^{th}}$degree B-polynomials defined on a domain interval forms a complete basis of continuous polynomials in terms of which any arbitrary function can be expanded by using an appropriate weighting factor for each B-polynomial (see 2.1). This will be demonstrated in the case of Bezier curve formation in the next section. Table 1.1 lists the expressions of all Bernstein basis polynomials up to the $\boldsymbol{5^{th}}$degree for $\boldsymbol{x \in [0,1]}$.

Table 1.1. Listing of all k=(p+1) Bernstein polynomials by degree *p*

$B(p,i,x)$ for$0 \le p \le 5$ and $0 \le i \le p$are provided below

p	i	C(p,i) = p!/i!(p-i)!	B(p, i, a= 0, b= 1, x)	Figure Ref.
0	0	1	B(0 ,0, x) = 1	NA
1	0	1	B(1,0,x) = $1-x$	NA
1	1	1	B(1,1,x) = x	
2	0	1	B(2,0,x) = $(1-x)^2$	Figure 2.1
2	1	2	B(2,1,x) = $2\,x\,(1-x)$	(left panel)
2	2	1	B(2,2,x) = x^2	
3	0	1	B(3,0,x) = $(1-x)^3$	Figure 2.2
3	1	3	B(3,1,x) = $3\,x\,(1-x)^2$	(left panel)
3	2	3	B(3,2,x) = $3\,x^2(1-x)$	
3	3	1	B(3,3,x) = x^3	
4	0	1	B(4,0,x) = $(1-x)^4$	Figure2.3
4	1	4	B(4,1,x) = $4\,x\,(1-x)^3$	(left panel)
4	2	6	B(4,2,x) = $6\,x^2\,(1-x)^2$	
4	3	4	B(4,3,x) = $4\,x^3(1-x)$	

4	4	1	B(4,4,x) = x^4	
5	0	1	B(5,0,x) = $(1-x)^5$	NA
5	1	5	B(5,1,x) = $5x\,(1-x)^4$	
5	2	10	B(5,2,x) = $10x^2\,(1-x)^3$	
5	3	10	B(5,3,x) = $10x^3(1-x)^2$	
5	4	5	B(5,4,x) = $5x^4(1-x)$	
5	5	1	B(5,5,x) = x^5	

The last column of Table 1.1 points to relevant figures which show all the B-polys on the left-side, whereas their right-sides show how the Bezier curves of the same degree can be created by the summation of all the individually and arbitrarily weighted Bernstein polynomials. This is discussed in the next section.

In the next section, the application of weighted Bernstein polynomials to the construction of Bezier curves will be taken up to be followed by their use to construct by piece-wise addition the more versatile spline curves (also known as B-spline curves).

2. Application of Bernstein Polynomials to the Bezier Curve Function

A Bezier curve function $\boldsymbol{P(x)}$ of degree p and order $\boldsymbol{k = p + 1}$ is defined to consist of the sum (also known as a blend) of $\boldsymbol{k}$ weighted Bernstein polynomials, $\boldsymbol{B(p, i, x)}$ of the same degree $\boldsymbol{p}$(1.4) and is given by,

$$\boldsymbol{P(x) = \sum_{i=0}^{p} w_i B(p,i,x) = \sum_{i=0}^{p} w_i\, C(p,i) x^i (1-x)^{p-i}\ ,\ x\ \in [0,1]} \tag{2.1}$$

Here, $\boldsymbol{w_i}$ is not only the coefficient or weight of the $\boldsymbol{i^{th}}$ B-polynomial but is also the $\boldsymbol{i^{th}}$ of the $(\boldsymbol{p+1})$ control points. A linear Bezier curve requires just two control points $(\boldsymbol{w_0, w_1})$and using the first degree Bernstein polynomial (see Table 1.1) is given by,

$$\boldsymbol{P(x) = \sum_{i=0}^{1} w_i B(p,i,x) = w_0 B(1,0,x) + w_1 B(1,1,x) = w_0(1-x) + w_1\, x} \tag{2.2}$$

A quadratic Bezier curve requires 3 control points and a 2nd degree Bernstein polynomial. It is given (see Table 1.1) by

$$\boldsymbol{P(x) = \sum_{i=0}^{2} w_i B(p, i, x) = w_0 B(2, 0, x) + w_1 B(2, 1, x) + w_2 B(2, 2, x)} \quad (2.3)$$

Substituting the polynomial expressions we obtain,

$$\boldsymbol{P(x) = w_0(1-x)^2 + w_1(2x(1-x)) + w_2 x^2,\ x \in [0, 1]} \quad (2.4)$$

Similarly, a cubic Bezier curve can be written using (2.1) and Table 1.1 as

$$\boldsymbol{P(x) = w_0(1-x)^3 + w_1(3x\,(1-x)^2) + w_2(3x^2(1-x)) + w_3 x^3,\ x \in [0, 1]} \quad (2.5)$$

Similarly, a quartic Bezier curve can be written using (2.1) and Table 1 as

$$\boldsymbol{P(x) = w_0(1-x)^4 + w_1(4x(1-x)^3) + w_2(6x^2(1-x)^2) + w_3(4x^3(1-x)) + w_4 x^4} \quad (2.6)$$

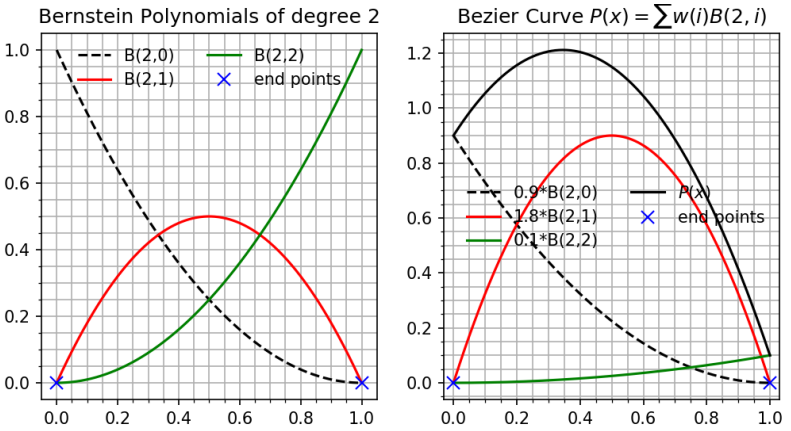

Figure 2.1: B-polys of degree 2 (left) are weighted at right to form Bezier curve P(x).

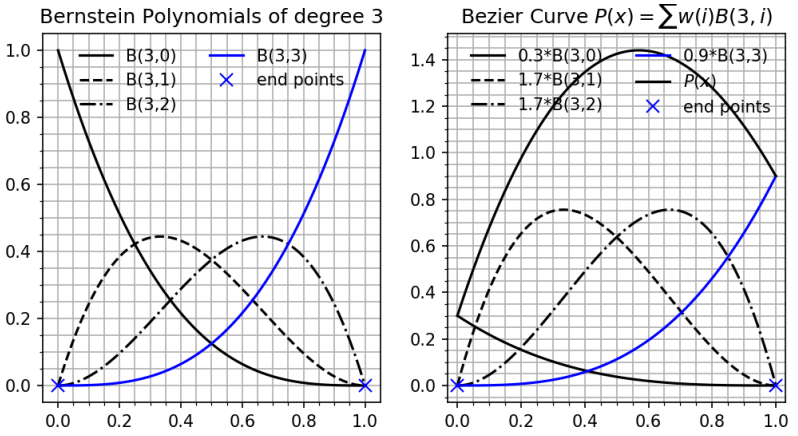

Figure 2.2: B-polys of degree 3 (left) are weighted at right to form Bezier curve P(x).

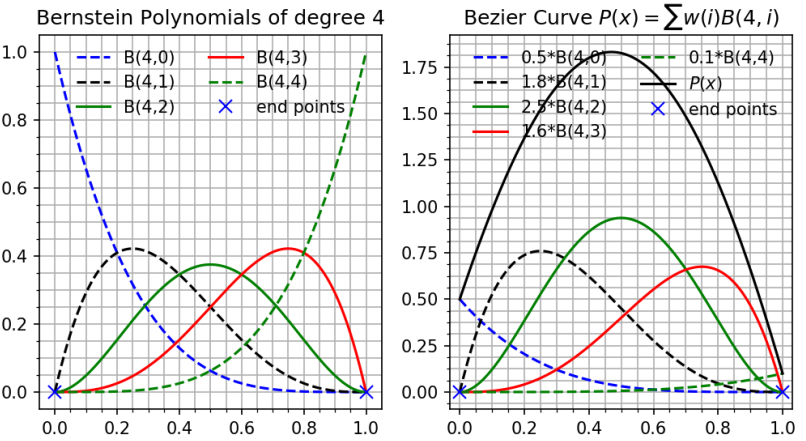

Figure 2.3: B-polys of degree 4 (left)are weighted at right to form Bezier curve P(x).

It is evident that the given number of weights (or control points) decide the order of the Bernstein polynomials used, and their member polynomials of degree (order - 1) span the entire domain of the Bezier curve. The role of those weights is to modify the shape of the Bezier curve as a whole, but they do not partition the domain into smaller intervals.

The main disadvantage of constructing Bezier curves as is that while their basis polynomial functions span the entire solution domain, a change of any one weight changes the shape of the entire curve, and also its degree becomes larger with an increase in the number of the

weights or control points. This problem can be addressed by dividing the solution domain into a number of smaller intervals, each of which can be spanned by using a basis spline or B-spline functions of a smaller degree. This will be discussed in the next section.

3. Splines and B-Splines

To draw smooth curves through data points, drafters used thin and flexible strips of wood, hard rubber, metal, or plastic called mechanical splines. Pins were placed at a judicious selection of points along a curve in a design to use a mechanical spline, and then the spline was bent so that it touched each of these pins. The spline interpolates the curve at these pins with this construction. It can be used to reproduce the curve in other drawings. The points where the pins are located are called knots. We can change the shape of the curve defined by the spline by adjusting the location of the knots.

A 'spline' is a function that is constructed piece-wise from polynomial functions. Earlier, spline was the name of a tool used by engineers to construct smooth shapes which had their desired properties. Drafters have been using a bendable strip known as "spline", which is fixed in position at a number of points that relaxes to form a smooth curve passing through those points. The malleability of the spline material combined with the constraint of **the control points (also known as knots)** caused the strip to take the shape that minimized the energy required for bending it between the fixed points, which resulted in the smoothest possible shape.

Now we can make use of a class of splines called **B-splines (also known as basis splines).** A B-spline function is the maximally differentiable interpolative basis function. The B-spline curve is a generalization of the Bezier curve (Note: a B-spline curve with **no interior knots** is a Bezier curve). The B-splines are defined by their order k, their domain limited by the two end-point knots and a number of interior knots **m** between the two end-point knots so that the total number of knots will be **m+2**. A knot is simply a point where splines on both sides meet with smooth continuity. The degree of the

B-spline polynomial is one less than the spline order **k** that is (**degree, p = k -1**).

3.1. B-spline knots

B-spline curves are composed of many polynomial pieces and are therefore more useful than Bezier curves. Consider **m+ 2** real values $\boldsymbol{t_i}$, always comprising of two end-points (end-knots) of physical domain boundary, $\boldsymbol{t_0}$ and $\boldsymbol{t_{m+1}}$ and $\boldsymbol{m}$ interior knots $\boldsymbol{t_1}$ to $\boldsymbol{t_m}$ between them such that $\boldsymbol{m \geq 0}$. These knots constitute a defined physical knot sequence given by $\{\boldsymbol{t_i}\}$ such that $\boldsymbol{t_0 \leq t_1 \leq \cdots\cdots \leq t_{m+1}}$**.** When the knots are equidistant, they are said to be uniform; otherwise, they are said to be non-uniform. Here only uniform knot vectors are considered. Thus, a knot vector of just two physical end-point knots could be $\boldsymbol{[0, 1]}$**.** Bezier curves (Sec.2) possess only two end-point knots, $\boldsymbol{t_0}(= \boldsymbol{a = 0})$ and $\boldsymbol{t_1}(= \boldsymbol{b = 1})$ but no interior knots$(\boldsymbol{m = 0})$ so that they are a limiting case of a B-spline which has no interior physical knots.

An example of a general open uniform knot vector with $\boldsymbol{m = 3}$ is $\boldsymbol{[0, 1, 2, 3, 4]}$ between the end knots $\boldsymbol{[0, 4]}$ and its **normalized** form $\boldsymbol{[0,\ 0.25, 0.50, 0.75, 1]}$ also has similar properties. A uniform knot vector with $\boldsymbol{m + 2}$ knots can be given by $[\boldsymbol{t_0, t_1, t_2, \cdots, t_m, t_{m+1}}]$ and it can be normalized by dividing through the last knot of that sequence. These knots (or points) are also referred to as **physical** knots as they lead to non-zero knot intervals.

It is necessary to enable the uniform knot vector to make use of B-spline basis functions of **a chosen order or degree** in each partition interval created by the internal knots between the two end-knots. This enablement is accomplished by introducing an equal number of repetitions (or multiplicity) of both end-knots. The repeated knots are called **ghost** knots (points).

3.2. The B-spline or Basis Spline function

Suppose we consider the uniform knot vector again with $\boldsymbol{m + 2}$ knots. This number will now be increased by repeating both the left and right end-point knots $\boldsymbol{p = k - 1}$ times in their respective places. This is defined as the augmented knot vector, or the open uniform knot vector, which now has $\boldsymbol{m + 2k}$ knots in all. The index of the repeated knots can be reset as follows:

$[\boldsymbol{t_0} = \cdots = \boldsymbol{t_0} = \boldsymbol{t_0},\ \boldsymbol{t_1}, \boldsymbol{t_2}, \cdots, \boldsymbol{t_m},\ \boldsymbol{t_{m+1}} = \boldsymbol{t_{m+1}} = \cdots = \boldsymbol{t_{m+1}}]$(3.2.1)

The repeated knots are also referred to as ghost knots since they can form only null or zero-valued knot intervals. In (3.2.2), the indexing of the augmented knot vector runs as $\boldsymbol{i} = \{\mathbf{0}, \mathbf{1} \ldots \boldsymbol{m} + \mathbf{2}\boldsymbol{k} - \mathbf{1}\}$. An example of an open uniform knot vector is $[\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}\ ]$, which has two repeated end-knots (or ghosts) but no internal knots. Another example of an open uniform vector is $[\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0.25}, \mathbf{0.5}, \mathbf{0.75}, \ \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$which has three repeats (ghosts) of end-knots. The multiplicity of the end-knots is then defined to be four. In this way, the augmentation is to be done, and this step also decides the order and degree (order – 1) of the B-spline basis function required. The method of finding expressions for the desired B-spline basis has to use the well-known Cox-deBoor recurrence formula, which is extensively used and also used in computer calculations. It is outlined below.

Recursive Definition of the B-spline (Basis Spline)

For each of either **uniform or** augmented knots $\boldsymbol{t_i}$ where $\boldsymbol{i}$ is a counter for the knot sequence such as $(\boldsymbol{i} = \mathbf{0}, \mathbf{1} \ldots \boldsymbol{m} + \mathbf{2}\boldsymbol{k} - \mathbf{1})$ a set of real-valued functions $\boldsymbol{N}(\boldsymbol{p}, \boldsymbol{i}, \boldsymbol{x})$ for $\boldsymbol{k} = \mathbf{0}, \mathbf{1}, \cdots, \boldsymbol{p}$ with $\boldsymbol{p}$ being the **degree** of the B-spline basis function is defined by the **Cox-deBoor recurrence** relation given below after the following the note on notation.

Important Note on notation: In Sec. 2, the notation $\boldsymbol{B}(\boldsymbol{p}, \boldsymbol{i}, \boldsymbol{x})$ was used for Bernstein Basis functions (aka, B-polys). This has been changed to the notation $\boldsymbol{N}(\boldsymbol{p}, \boldsymbol{i}, \boldsymbol{x})$ as it should refer to both B-polys basis functions which apply to knot vectors containing only one interval between its two end-knots and to the Basis spline functions which apply to all knot vectors with any number of internal knots. In the latter case, a shifting of B-polys has to be done in each interval bounded by each pair of internal knots. Both notations are interchangeably used in publications with the above remark in mind.

The deBoor recursive definition (see [4], [5], and [11])now follows:

For zeroth degree B-spline (p = 0):

$$\boldsymbol{N}(\mathbf{0}, \boldsymbol{i}, \boldsymbol{x}) = \mathbf{1}\boldsymbol{if}\boldsymbol{t_i} \le \boldsymbol{x} \le \boldsymbol{t_{i+1}}\boldsymbol{and}\ = \mathbf{0}\boldsymbol{elseif} \qquad (3.2.2a)$$

For higher degrees ($\boldsymbol{p} > 0$):

$$\boldsymbol{N(p,i,x) = \left[(x - t_i)/(t_{i+j} - t_i)\right]N(p-1,i,x) +}$$

$$\boldsymbol{+\left[(t_{i+j+1} - x)/(t_{i+j+1} - t_{i+1})\right]N(p-1,i+1,x)} \qquad (3.2.2b)$$

Expressions 3.2.2aand 3.2.2b specify how to construct a degree **p** (or, order **k**) basis function from B-spline functions of lower degree **p-1** (or, order **k-1**). A few properties of $\boldsymbol{N(p,i,x)}$ are similar to Bernstein basis polynomials, and three of them are listed below.

1. $\boldsymbol{N(p,i,x)}$ is a degree p (order k=p+1) polynomial in x, non-zero and non-negative.
2. At most non-zero k= p+1 degree basis functions for $\boldsymbol{x\epsilon}\ [\boldsymbol{t_i, t_{i+1}}]$ are:

$\boldsymbol{N(p,i-p,x)}$,$\boldsymbol{N(p,i-p+1,x)}$,$\boldsymbol{N(p,i-p+2,x)}$,..., etc. up to $\boldsymbol{N(p,i,x)}$.

3. Partition of unity: The sum of all $\boldsymbol{k = p+1}$ degree p basis functions as given above is 1.

3.3 Now, a few simple examples for a closed domain interval [0, 1] as end-knots but without internal knots are demonstrated below.

3.3.1. Consider the open uniform knot vector [**0 0 0 1 1 1**]**, of degree, p = 2 (order = 3)**. The six knots in it can form five intervals of type $[\boldsymbol{t_i},\ \boldsymbol{t_{i+1}}]$ out of which the four formed by **ghost (or repeated) knots** such as **[0, 0] and [1, 1] do not exist and hence do not give rise to non-zero basis functions. The remaining non-zero physical interval is the end-knot interval [0, 1] for which the B-spline basis functions $\boldsymbol{N(0,i,x)}$ can be calculated [8] by hand** using the Cox-deBoor recursive formula, and they are as listed in Table 2 below:

Table 2: Basis functions $N(p,i,x)$ of degree = 2 for knot vector [0 0 0 1 1 1]

N(0, 0, x) = 0	N(0,1, x) = 0	N(0,2, x) = 1	N(0,3, x) = 0
N(1,0, x) = 0	N(1, 1, x) = $(1-x)$	N(1,2, x) = x	N(1,3, x) = 0
N(2,0, x) = $(1-x)^2$	N(2,1, x) = $2x(1-x)$	N(2, 2, x) =x^2	N(2,3, x) = 0

The three non-zero of 2nd degree basis functions shown in the table above (bottom row) are seen to be the same as the **Bernstein polynomials (B-polys) of 2nd degree** in Table 1.1 (they can also be

directly computed using Eqn. (1.4)) and are shown graphed in Figure 3.3.1 below using the B-spline computing and plotting script (also see the left panel of Fig. 2.1).

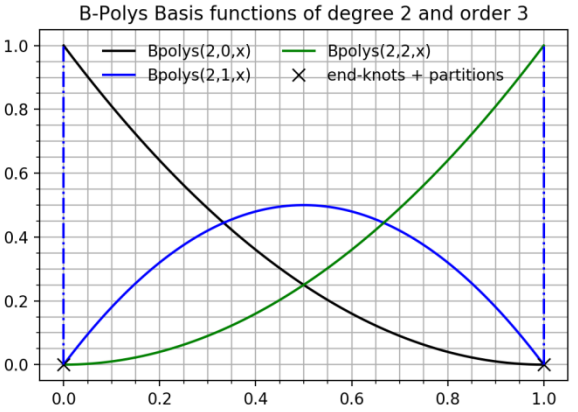

Figure 3.3.1: 2nd degree B-spline (B-polys) basis functions drawn in single closed interval for the open uniform knot vector [0, 0, 0, 0, 1, 1, 1, 1] with 3 ghosts (repeated knots).

3.3.2. Consider the open uniform knot vector **[0 0 0 0 1 1 1 1], of** degree, **p = 3 (order = 4)**. The degree wise basis functions $\boldsymbol{N(i,0,x)}$ to $\boldsymbol{N(i,3,x)}$ resulting from the deBoor recursive formula for the non-zero end-knot interval **[0, 1]** are as tabulated below:

Table 3: Basis functions $N(3,i,x)$ of degree = 3 for knot vector [0 0 0 0 1 1 1 1] in domain [0, 1] (Reference: Magoon [8])

N(0, 0, x) = 0	N(0, 1, x) = 0	N(0, 2, x) = 0	N(0, 3, x) = 1	N(0, 4, x)=0
N(1, 0, x) = 0	N(1, 1, x) = 0	N(1, 2, x) $=(1-x)$	N(1, 3, x) = x	N(1, 4, x)=0
N(2, 0, x) = 0	N(2, 1, x) = $(1-x)^2$	N(2, 2, x) $= 2x(1-x)$	N(2, 3, x) = x^2	N(2, 4, x)=0
N(3, 0, x)= $(1-x)^3$	N(3, 1, x) $=3x(1-x)^2$	N(3, 2, x) $=3x^2(1-x)$	N(3, 3, x)= x^3	N(3, 4, x)=0

All four B-spline basis functions in the 5th row above are plotted in Fig. 3.3.2.

The four non-zero basis functions of the 3rd degree shown in Table 3 above (5th row) are the same as the Bernstein polynomials of the 3rd degree in Table 1.1. These were computed by the author's Python

computing script and plotted in Figure 3.3.2 below (note: can also compute directly using Eqn. (1.4)) and are shown graphed in Figure 3.3.3 below (see also the left side panel of Figure 2.2). Many examples of similar hand calculations of such basis functions are given with all details in [8] and [9]. However, now the use of dedicated software such as SPLIPY has enabled calculations of all basis functions, including derivatives and their evaluations at all specified collocation points.

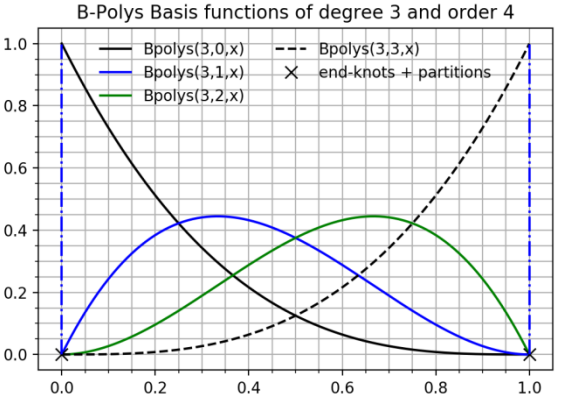

Figure 3.3.2: 3rd degree B-spline (B-polys) basis functions drawn in single closed interval for the open uniform knot vector [0, 0, 0, 0, 1, 1, 1, 1] with 3 ghosts (repeated knots).

3.3.3. Consider the open uniform knot vector **[0 0 0 0 0 1 1 1 1 1], of** degree, **p = 4 (order = 5)**. The five B-spline basis functions $\boldsymbol{N(i(=0,1,2,3,4),\ 4,\ x)}$ in the closed interval **[0, 1]** resulting from the Cox-deBoor recursive formula are found to be the same as the **4th degree Bernstein polynomials** listed in Table 1.1 above (Note: It can also be directly computed using Eqn. (1.4)) and by using the B-spline computing and plotting script are shown graphed in Figure 4.2.1 (in Sec.4.2) (also the left side panel of Figure 2.3).

It is evident from the above three examples (3.3.1 to 3.3.3) that Bernstein basis polynomials of the same degree can be directly used as B-spline basis functions when the open uniform (or augmented) knot vectors have any number of repeated end-knots but do **not** have any internal knot between the two end-knots. These examples suggest instances of augmented end-knot vectors, which can be used to approximate the solutions of simpler ODEs ([9], [10], [11]).

4.1 Application of B-polys to a 4th order ODE

The example of the 4th order DE considered here arises from the Euler-Bernoulli beam theory. The relationship between the deflection $\boldsymbol{y(x)}$ and the uniformly applied load $\boldsymbol{w(x)}$ at any point $\boldsymbol{x}$ of a beam is described by a fourth order differential equation given by

$$\boldsymbol{EIy^{iv}(x) = EI\,(d^4y(x)/dx^4) = w(x)\,,} \tag{4.1}$$

where $\boldsymbol{E}$ is the elastic modulus of the beam material assumed to be linear, I is the moment of inertia of the beam of uniform cross-section about its neutral axis so that their product $\boldsymbol{EI}$ is constant for a given beam of length $\boldsymbol{L}$ and $\boldsymbol{w(x)}$ is the weight per unit length that may depend on the nature of load distribution along the beam, but here it is assumed to be a constant. In this case, the total weight of the beam $\boldsymbol{w_0 = wL}$ will be the load that acts along the entire length, $\boldsymbol{L}$, of the beam downwards. In the examples considered here, this will be taken as the only load acting to cause a proportionate deflection along the length of the beam. Then (1.1) can be rewritten as,

$$\boldsymbol{y^{iv}(x) = -(w_0/EI) = -1} \tag{4.2}$$

As the second term $\boldsymbol{(w_0/EI)}$ is a constant for a given beam, it is normalised for convenience in this paper by setting $\boldsymbol{(w_0/EI) \equiv 1.}$ This expression constitutes the BVP of the problem, and its solution requires four boundary conditions (BCs). These **BCs** depend on the way the beam is set up for study and analysis. In this paper, the beam is set up as a cantilever of unit length, $\boldsymbol{L \equiv 1,}$ by having one end firmly clamped and keeping the other end free. The four BCs relevant to this cantilever set up are:

$$\boldsymbol{y(0) = 0\,;\;\; y'(0) = 0;\;\; y''(1) = 0,\;\text{and}\;\; y'''(1) = 0} \tag{4.3}$$

where the clamped end provides the first two BCs, and the free end provides the last two BCs, respectively. Using these Eqn. (4.2) is easily solved to yield the cantilever deflection profile and the maximum deflection at the free end as follows:

$$\boldsymbol{y(x) = -(1/24)\left(x^4 - 4x^3 + 6x^2\right)} \tag{4.11}$$

$$\boldsymbol{y_{max} = y(x = 1) = -0.125} \tag{4.12}$$

The computed solution, to be described next, will be compared with the exact expressions (4.11) and (4.12) later.

4.2 Algorithmic Steps of Computer Solution

We will now express the desired solution $\boldsymbol{y(x)}$ by adopting the same form as Eqn. (2.1) and rewrite it in terms of a $\mathbf{p}^{\text{th}}$ degree (order, $\boldsymbol{k = p + 1}$) bspline basis function $\boldsymbol{B(p, i, x)}$, which has $\boldsymbol{k = p + 1}$ basis functions (Sec. 2) counted by the second index $\boldsymbol{i}$ as follows:

$$\boldsymbol{y(x) = \sum_{i=0}^{p} w_i B(p, i, x) = \sum_{i=0}^{p} w_i\, C(p, i) x^i (1 - x)^{p-i}}\,, \quad \boldsymbol{x \in [0, 1]} \tag{4.13}$$

As many publications also use **order** instead of a degree, 4.13 can be recast as follows.

$$\boldsymbol{y(x) = \sum_{i=1}^{k} w_i B(k, i, x) = \sum_{i=1}^{k} w_i\, C(k, i) x^i (1 - x)^{k-i}}\,, \boldsymbol{x \in [0, 1]} \tag{4.14}$$

It is noticed that the index starts with unity in 4.14, which keeps the **number** of basis functions is unchanged as long as $\boldsymbol{k = p + 1.}$ This is convenient for computer calculations done in this paper, so hand calculation of both B-polys and bsplines basis functions was avoided. The Python software package **SPLIPY** [7] is used here for all orders (or degrees) of bspline basis function calculations and their evaluation at any or all **x** values in the domain of the solution space. It should be recalled here that when bspline basis functions are applied to the entire domain extent as a single interval, then they reduce to the Bernstein basis functions (B-polys) of the same order or degree (see Table 1.1). However, if the domain is subdivided into two or more intervals, then the B-polys are to be separately computed for each interval. This job is better left to the SPLIPY software, which does all necessary calculations using the same CarlDeBoor's algorithm [5] to [11]. This software can calculate B-polys basis functions of any order also for undivided domains with only two end-knots, and this will be used to solve the 4^{th} order ODE problem. The solution is solved in sequential steps outlined below.

Step 1. The problem statement. The 4^{th} order ODE for the cantilever is given by

$y^{iv}(x) = -1$ **with BCs** $y(0) = 0$; $y'(0) = 0$; $y''(1) = 0$, **and**

$y'''(1) = 0$; $x \in [0, 1]$ (4.15)

Since this ODE is of 4th order, the solution of the form (4.14) must contain at least 5th order b-spline basis functions in order that they can be smoothly differentiated four times. Hence we set $k = p + 1 = 5$ which means 5 B-polys of 4th degree are involved. They are listed in five rows starting from 14th row in Table 1.1 and Figure 3.3.3 shows their plot which is also suitable for solving Eqn.(4.15). The undetermined solution and its differentials now have the form $y(x) = \sum_{i=1}^{5} w_i B(5, i, x)$ where there are 5 unknown weights $\{w_i\}$ to be determined. It is these weights (aka, control points) which help to modify the attached basis functions at a few selected positions (aka, Greville coordinates; see below) in the domain and add them together to determine the shape of the solution curve (see Sec.2). The expanded solution and its all four derivatives in their compact form are given below.

$y(x) = w_1 B(5, 1, x) + w_2 B(5, 2, x) + w_3 B(5, 3, x) + w_4 B(5, 4, x) + w_5 B(5, 5, x)$ (4.16)

$y(x) = \sum_{i=1}^{5} w_i B(k, i, x)$ →BC: $y(x = 0) = 0$ **(4.16)**

$y'(x) = \sum_{i=1}^{5} w_i B'(k, i, x)$ →BC: $y'(x = 0) = 0$ (4.17)

$y''(x) = \sum_{i=1}^{5} w_i B''(k, i, x)$ →BC: $y''(x = 1) = 0$ (4.18)

$y'''(x) = \sum_{i=1}^{5} w_i B'''(k, i, x)$ →BC: $y'''(x = 1) = 0$ (4.19)

$y''''(x) = \sum_{i=1}^{5} w_i B''''(k, i, x)$ →BC: $y''''(x) = -1$ (4.20)

It is to be noted that weights are kept constant, only the $k = 5$ basis functions $B(k, i, x)$ are successively differentiated in the above expressions, and they can also be evaluated at the specified x values provided by the Greville averaging method (Step 2 below)by the SPLIPY [6] software. Since there are five equations all the BCs stated in (4.15) can be applied, and they are also shown to the right of the arrow against each equation above. When all the five expressions are expanded, a simultaneous equation is obtained as a five by six matrix. This can easily be solved by using the Numpy's linear algebra module to obtain the five Bezier control points (weights) w_i.

These five weights can then be immediately substituted in (4.16) to yield a general equation for the deflection profile.

We now require a set of five well-placed evaluation coordinates (aka, collocation points) along the domain width, that is, along the unit length of the cantilever. To find these points, we need to specify the knot vector for the problem, and this is done in the next step.

Step 2. Knot vector specification for the problem domain

Since the solution (4.16) has 5th order b-splines and there is just one interval between the end knots $[\mathbf{0}, \mathbf{1}]$**,** the appropriate open uniform knot vector is given by $[\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$**. T**he domain end knots **0** and **1** are each repeated $\boldsymbol{p = 4}$ times, which of course, corresponds to the degree (and order, $\boldsymbol{k = 5}$) of the b-splines. This knot vector specification was developed by Carl deBoor in order that the computing software identifies it and the needed 5 basis functions are calculated. Further, this knot vector is used to determine the evaluation coordinates for the solution by carrying out a sequential averaging of $\boldsymbol{p = 4}$ knots in the knot vector. This is known as the **Greville** averaging, and the coordinates so obtained are called **Greville abscissas**. First this process yields the values **(0+0+0+0)/4, (0+0+0+0)/4, (0+0+0+1)/4, (0+0+1+1)/4, (0+1+1+1)/4, (1+1+1+1)/4, and (1+1+1+1)/4** . The resulting values are **0.0, 0.0, 0.25, 0.50, 0.75, 1.0, and 1.0.** It is standard practice to omit one of the first and one of the last repeated values, and the remaining five values **{0.0, 0.25, 0.50, 0.75, 1.0]** constitute the Greville abscissas (aka, control points or Greville collocation coordinates, [8],[9]). By using these values, all the B-poly basis functions in (4.16) are evaluated and plotted. The SPLIPY {6] software can do these calculations too once the open knot vector and its order values are supplied to it. The following python function can be used to get the Greville abscissa values.

```
def   GrevilleAbsc(degree,  agknots):
    # GREVILLE Abcissa Control Points (Coordinates) computation
    agk = np.array(agknots); magk = len( agk ); Ngx = magk - degree
    Gx = np.zeros((Ngx+1))
    for  i  in  range (Ngx+1):
        for j in range(degr):      #while i < magk-ndegr+1
            Gx[i] += agk [ i + j ] / degree
    print("Full array Gx as obtained is:\n", Gx)
    # now omit the first and last repeated coordinates
    Gx = Gx[ 1 : -1 ] # <-- This is array of Greville abcissae values
    return Gx
```

The Greville abscissae method as outlined above is excellent and adequate for problems considered here and has been applied to solve several ODEs ([8],[9]). For more complicated problems, collocation points are found as nodes and weights supplied by applying the Gauss-Legendre quadrature method. These are described in [10], [11] and [12].

Step 3. Solution Evaluation and plotting.

This is the final step. The five Greville collocation points found in Step 2 for the given 5th order knot vector are [$\mathbf{0.0}$, $\mathbf{0.25}$, $\mathbf{0.5}$, $\mathbf{0.75}$, $\mathbf{1.0}$]**.** Of these, the first and the last are used to evaluate all the five basis functions in each of the five equations (4.16) to (4.20). Therefore, they form a square matrix of 5 rows and 5 columns, which may be labelled as **A.** The values on the right-hand side of the equations can be collected to form a column vector labelled as B. It is given by $[\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, -\mathbf{1}]^{\boldsymbol{T}}$**.** The unknown column vector of the five weights labelled as Y is given by $[\boldsymbol{w_1}, \boldsymbol{w_2}, \boldsymbol{w_3}, \boldsymbol{w_4}, \boldsymbol{w_5}]^{\boldsymbol{T}}$**.** Together these may be written as $\boldsymbol{AY} = \boldsymbol{b}$, and this matrix equation can be readily solved by the linear algebra module of numpy. These weights (or, control points) can now be used in Eqn. (4.16) along with the five Greville points to compute the Bezier curve, which defines the deflection profile of the cantilever. The RMS error of points on

this curve can be computed by comparing with the exact theoretical solution, and the error profile can also be plotted.

All these steps are programmed with explanatory comments and written in the **Python 3.7** script listed in the Appendix. The results are presented in the three graphs (Fig.4.2.1, 4.2.2, and 4.2.3), which are also provided with relevant details.

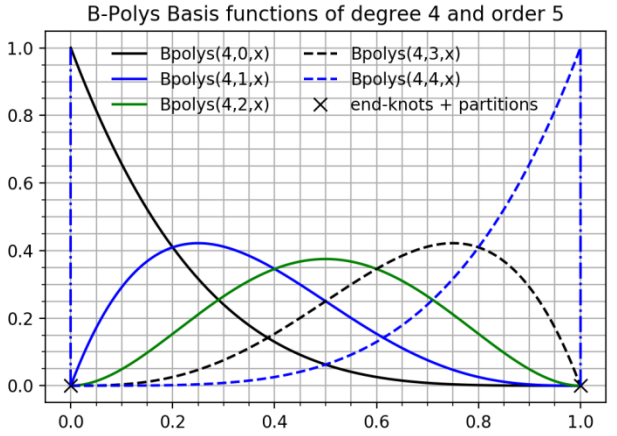

Figure 4.2.1: 4th degree Bernstein basis functions drawn in single closed interval for the open uniform knot vector $[0, 0, 0, 0, 0, 1, 1, 1, 1, 1]$ with 4 ghosts (repeated knots).

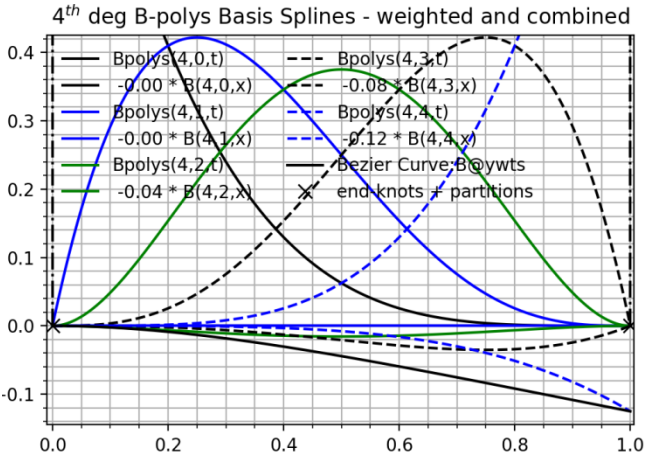

Figure 4.2.2. The lower part of all the 4th degree Bernstein basis functions drawn for the 5th order(4th degree)open knot vector $[0, 0, 0, 0, 0, 1, 1, 1, 1, 1]$are shown above the abscissa line, and their weighted

summation to form the Bezier curve is shown below the abscissa line. This Bezier curve at the bottom of this figure defines the computed cantilever deflection profile, which has its maximum value of -0.125 at the free end at the right.

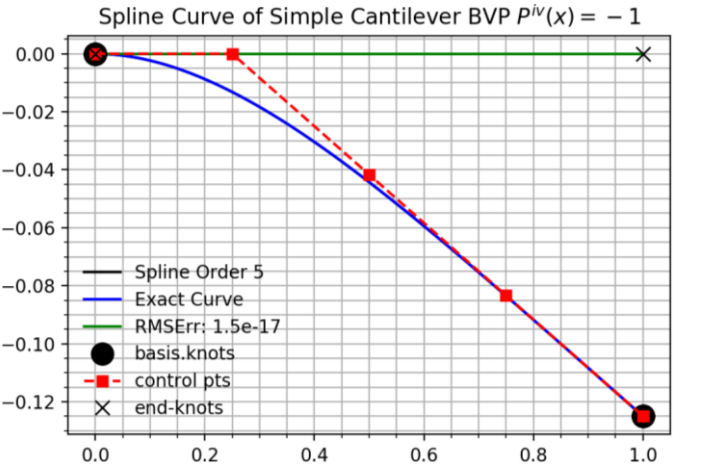

Figure 4.2.3. The computed cantilever deflection profile is drawn here as a Bezier curve with its five control points (weights). This profile coincides with the exact theoretical curve with an RMS error smaller than the machine precision. The maximum deflection of 0.125 at the free end (right edge of cantilever) also agrees with the calculated value.

5. Conclusion

The solution methodology of the 4th order cantilever BVP making use of Bernstein basis polynomials, basis spline functions, and construction of the Bezier curve as the solution profile of the problem under consideration highlights the role these basis functions can play as constituting an alternate method of solving various types of ODEs and BVPs occurring in many fields of physics. These basis functions have near ideal properties required for their use not only directly (as shown in Sec.4 above) but as Ritz variational basis functions too. Especially in quantum mechanics [12], where it rivals other routine methods. A good review of the many possible applications of B-splines in the field of atomic physics and quantum mechanics and their ease of use in making highly accurate computations as of 2001 is provided in [11]. Since then, the use of B-splines has become a vigorously active field of research worldwide.

References

[1] https://en.wikipedia.org/wiki/Bernstein_polynomial
[2] http://mae.engr.ucdavis.edu/~farouki/bernstein.pdf
[3] https://en.wikipedia.org/wiki/B-spline
[4] J.S. Racine,"A Primer on Regression Splines",(Chapter article), 2019. Pdf (downloadable from https://cran.r-project.org/ web/ packages/ crs/vignettes/spline_primer.pdf)
[5] https://en.wikipedia.org/wiki/De_Boor's_algorithm
[6] https://sintefmath.github.io/Splipy/index.html
[7] M. I. Bhatti and P. Bracken," Solutions of differential equations in a Bernstein polynomial basis", Journal of Computational and Applied Mathematics, Vol. 205(1), pp.272-280, 2007.
[8] J. Magoon, "Application of the b-spline collocation method to a geometrically non-linear beam problem" , MS Thesis, 2010, RIT, 2010. (Access from:http://scholarworks.rit.edu/theses)
[9] R. Jhaveri, "Design of passive suspension system with non-linear springs using b-spline collocation method", MS Thesis, RIT, 2011. (Accessfrom:http://scholarworks.rit.edu/theses)
[10] http://www.am.qub.ac.uk/users/h.vanderhart/Splinestop.htm
[11] H.Bachau etal., "Applications of *B*-splines in atomic and molecular physics", Rep. Prog. Phys. 64, 1815–1942, 2001.
[12] Johnson, "Lectures in Atomic Physics", University of Notre Dame, USA, 2006. (access this book from) https:// www3.nd.edu/ ~johnson/Publications/book.pdf

Appendix

Python 3.7 Script to solve to solve the 4th order-BVP ODE:
""" ********written by Mandyam N.Anandaram********

```
This BVP-01_FreeEndedCantilever.py problem script solves
Free Ended Cantilever Problem (left end clamped and right end free)
P'''' = -w_EI where w_EI = w/EI, and w = w0*L acting downwards
  (e.g., w0 = 15 kN/m, E = 2.0e05 MPa; I = 3.0e04 cm^4, L = 3.0 m)
 with 4 BCs: P(0) = 0; P'(0) = 0; P''(L) = 0; P'''(L) = 0 for 0 <= x <= L
Exact solution: P(x) = -(w_EI/24)*(x**4 - 4*L*x**3 + 6*L**2*x**2),
P(L) = -(w_EI * 3/24)*L**4  (Max deflection at x = L)
Unitless form: set L = 1 and w_EI = 1 to get  P'''' = -1 for 0 <= x <= 1
with exact normalized solution:
  P(x) = -(1.0/24)*(x**4 - 4*x**3 + 6*x**2);
```

```
 P(1) = -w_EI*L**4/8 ==> -1/8 = -0.125   (← max deflection at free-end )
and the BCs to be used now are:
P(0) = 0; P'(0) = 0; P''(1) = 0; P'''(1) = 0;
and the given data that  P''''(x) = -1 (normalized for 0 <= x <= 1)
"""                          ****************************
import splipy as splinpy   # ← SPLIPY 1.3.1 must be pre-installed
import numpy as np
from numpy.linalg import solve as npLA_solve
import matplotlib.pyplot as plt
def mainsolver( order,  augknotvec ) :
   ordr, agknots = order, np.array(augknotvec); agkmax = agknots[-1]
   degr = ordr-1 #degree = number of ghost knots, degree of B-polys etc
   #create B-spline basis function operator
   basis = splinpy.BSplineBasis(ordr, agknots)
   #if agkmax > 1: agknots = basis.reparam(agkmax) #normalize() #agknots)
   print("Basis Spline Function Parameters :")
   print("  Input data:  order =",ordr,"; degree =",degr)
   print("  Input augmented (open uniform) knots:",agknots)
   print("  Total number of knots: basis.__len__() = ",basis.__len__())
   print("  Starting point of knots: basis.start() = ",basis.start())
   print("  Ending point of knots: basis.end() = ",basis.end())
   print("  Physical/unique knots vector: basis.knot_spans() = ",
               basis.knot_spans(include_ghost_knots=False))
   print("  Number of internal knots within basis.knot_spans() = ",
                           basis.__len__() - 2*ordr )
   print("  Number of repeated end-knots (ghost knots) = degree =",degr)
   nbfs = basis.num_functions(); # Number of Basis funcs
   print("  Number of Basis Spline funcs: basis.num_functions() = ",nbfs)
   # create an array of nbfs controlpoints (= the number of basis funcs).
   # For 1D get weights/control points using Greville abcissa points
   # using grevx = basis.greville()  --- done above.
   # curve = splp.Curve(basis, controlpoints)
   grevx = np.array(basis.greville()) # compute/get greville eval points
   print("The",len(grevx),"Greville abcissa pts are:\n   ",grevx)
   # 201 uniformly spaced evaluation points on the domain (0,augkmax)
   t = np.linspace( 0,  agkmax,  201)
   # evaluate *all* basis functions on *all* points t. The
   # returned variable B is a matrix
   B = basis.evaluate(t, d = 0)
   # B.shape = (201,nbfs), 201 visualization points, nbfs basis functions
```

```
    print("B.shape =",  B.shape)
    degr = ordr-1;  augk = np.array(agknots[degr:-degr])  # omit ghost
knots
    # display the b-spline basis functions:
    plt.figure(figsize=(8,6),dpi=120)
    # plot the basis functions
    plt.plot(t, B, label="B(%d,i,x)"%nbfs)
    plt.plot(augk,augk*0.0 ,"kx", ms=8,label="knot partitions")
    for ia in range(len(augk)):  # ghosts not marked/repeated
          kx = augk[ia]
          plt.plot([kx,kx],[0.0,1.0] ,"b-.", ms=8)
    plt.grid(which="both");  plt.minorticks_on()
    plt.legend(loc=0,ncol=2,frameon=False)
    plt.title("Bernstein Basis functions of 4th Degree  and 5th Order")
    plt.show()

    # Here we compute needed derivatives upto 4th order required for the
    # BVP by evaluating them  at the Greville evaluation points which
    # were computed above from the augmented knot vector.
    # Note that for a 4th order ODE an augmented knot vector of
minimum
    # order 5 (degree 4) and above must be used
    w_EI = 1.0  # w_EI = W/EI = 1.0  assumed for the beam cantilever
    # basis.eval() for all greville coords is not required here:
    nwts = len(grevx) # note that nwts = nbfs, the number of basis funcs
    # Create an array ywts (= len(grevx) to hold control points to be
    # determined at all the corresponding Greville abcissae points
    #nwts = len(grevx) # note that nwts = nbfs, the number of basis funcs
    ywts = np.zeros((nwts,1),float);#print("y control points:",ywts)
    # To determine the nwts control points (weights) the following #nwts
    # equations are evaluated at only required/selected greville x-coords
    P0 = basis.evaluate(grevx[0],d=0)       # P (x = grevx[0]) = 0.0(value)
    dP0 = basis.evaluate(grevx[0],d=1)      # P'(x = grevx[0]) = 0.0
    d2P1 = basis.evaluate(grevx[-1],d=2)    # P'''(x = grevx[-1]) = 0.0
    d3P1 = basis.evaluate(grevx[-1],d=3)    # P'''(x = grevx[-1]) = 0.0
    d4P_gx2 = basis.evaluate(grevx[2],d=4)  # P''''(x = grevx[2]) = -1.0
    d4P_gx3 = basis.evaluate(grevx[3],d=4)  # P''''(x = grevx[3]) = -1.0
    #d4P_gxk = basis.evaluate(grevx[-1],d=4)  # P''''(x = grevx[-1]) = -1.0
    #Ax = See below
    # bx is the vector of all BCs and has len(bx) = nwts
    bx = np.zeros((nwts,1),float);print("bx:\n",bx)
    if nwts == 5:
```

```
        Ax = np.vstack((P0, dP0, d2P1, d3P1, d4P_gx2))
        bx[-1,0] = -1.0
    elif nwts == 6:
        Ax = np.vstack((P0, dP0, d2P1, d3P1, d4P_gx2, d4P_gx3))
        bx[-1,0] = -1.0; bx[-2,0] = -1.0;
    elif nwts == 7:
        d4P_gx4 = basis.evaluate(grevx[-1],d=4) # P''''(x = grevx[4]) = -1.0
        Ax = np.vstack((P0,dP0,d2P1,d3P1, d4P_gx2, d4P_gx3, d4P_gx4))
        bx[-1,0] = -1.0; bx[-2,0] = -1.0; bx[-3,0] = -1.0
    elif nwts == 8:
        d4P_gx4 = basis.evaluate(grevx[4],d=4) # P''''(x = grevx[4]) = -1.0
        d4P_gx5 = basis.evaluate(grevx[-1],d=4) # P''''(x = grevx[5]) = -1.0
        Ax =
np.vstack((P0,dP0,d2P1,d3P1,d4P_gx2,d4P_gx3,d4P_gx4,d4P_gx5))
        bx[-1,0] = -1.0;bx[-2,0] = -1.0; bx[-3,0] = -1.0; bx[-4,0] = -1.0
    else:
        print("solution works for 5th,6th,7th and 8th order knot vectors
only")

    print("Ax[]:\n",Ax);print("bx :\n",bx);# print("y control points:",ywts)
    # Solve for all the nwts control points using np.linalg.solve() below
    yx = npLA_solve(Ax,bx); #print("Ax[]:\n",Ax);print("bx :",bx);
    print("yx =",yx); print("Verify: bx = Ax @ yx =",Ax@yx)
    for ilen in range(len(yx)): ywts[ilen,0] = yx[ilen]
    print("Number of Curve weights (control pts) =",ywts.shape[0])
    print("Curve weights (control points) array:\n =",ywts)
    #   Bspline Curve
    # Curves are defined by associating a controlpoint to each basis
    # function. y(x) = \sum_{i=1}^n w_i. B{p, i, x} where w_i are the
    # controlpoints and B{p,i,x} are the basis functions. y(x) is the
    # parametric curve, and by letting each controlpoint be a vector of
    # length 1, we may create aplanar  curve.   It is necessary to create an
    # array of as many controlpoints as the number of basis functions.
    # For 1D curves Greville abcissa points are found from the augmented
    # knot vector using grevx = basis.greville()  and these are used to
    # determine the weights or control points from the problem matrix.
    # Finally the planar  curve is generated from the command :
    #         curve = splp.Curve(basis, controlpoints)
    # ywts = weights array = number of Greville abcissae (eval
coordinates).
    print(  "weights:",  ywts)
    #wts = np.array([0.0, 1.25, 1.0]); print("weights:",wts)
```

```
    clrs1 = ['b--','g--','m--','y--','r--','c--','k--','m-','k--','g--']
    clrs2 = ['b-.','g-.','m-.','y-.','r-.','c-','k-.','m-.','k-.','g-.']
    # display the spline curve by combining the weighted B-splines
    plt.figure(figsize=(8,6),dpi=120)
    # plot the basis functions, weighted basis funcs and blended curve
    for iw in range(ordr):
        plt.plot(t, B[:,iw],clrs1[iw],label="Bpolys(%d,%d,t)"%(degr,iw))
        plt.plot(t, B[:,iw]*ywts[iw,0],clrs2[iw], label=" %.2f *
B(%d,%d,t)"%(ywts[iw,0],degr,iw))
    plt.plot(t, B@ywts,"k",label="Curve:B@ywts")
    plt.plot(augk,augk*0.0 ,"kx", ms=8,label="knots")
    #for ia in range(len(augk)):  # ghosts not marked/repeated
    #    kx = augk[ia]; plt.plot([kx,kx],[0.0,1.0] ,"k-.", ms=8)
    plt.grid(which="both"); plt.minorticks_on()
    plt.legend(loc=0,ncol=2,frameon=False)
    plt.title('B-polys Basis splines  weighted and combined')
    plt.show()

    curve1d = splinpy.Curve(basis,ywts) # Construct spline curve from
ywts
    print("# access curve evaluation at greville abcissae points below:")
    for igr in grevx:
        print("curve1d(",igr,") = ", curve1d(igr) )
        #print("d(curveid.deriv(",igr,") )= ", curve1d.derivative(igr) )
        #print("dd(curveid.deriv(",igr,",d=2)) = ", curve1d.derivative(igr,
d=2) )
    print("# access spline curve controlpoints (= weights) below :")
    for i in range(ordr): print("curve1d[",i,"] =", curve1d[i] )
    # prints all the (0-indexed) polygon control points (= weights)
    print("Given knot vector is :",basis.knots)
    # Evaluate the curve at all visualization points of the domain.
    tx = np.linspace(0,agkmax,201) # spline evaluation points
    Pbspx = curve1d(tx); Pbspx = Pbspx[:,0];
    #print( "Px.shape =", Pbspx.shape)    #Px[:,0]--->Px[:]
    Pexact = w_EI / 24*tx*tx*(4.0*tx  - tx*tx  -  6.0)   #    recall that w_EI =
1
    print("Maximum sag at the free end = ",min(Pexact))  # -ve
    # get RMS error for the difference (Exact sol - Bspline sol)
    rmserr =  np.sqrt( ( (Pexact  -  Pbspx)**2 ).mean() )
    print( "RMS Error =", rmserr)
    # plot the curve itself
    plt.figure(figsize=(6,8),dpi=180)
```

```
    plt.plot(tx, Pbspx, "k",label="Spline Order %d"%ordr)
    plt.plot(tx, Pexact,"b",label="Exact Curve")
    plt.plot(tx, Pbspx-Pexact, "g",label="RMSErr: %.1e"%rmserr)
    # then evaluate the break-points (the knots)
    tbk = basis.knots; #print("basis knots, ti = ",tbk)
    xtbk = curve1d(tbk); #print("xtbk.shape",xtbk.shape)
    # Also plot the breakpoints as black dots
    plt.plot(tbk, xtbk, 'ko ',ms=12, label="basis.knots")
    plt.plot(grevx, curve1d.controlpoints,'rs--',label="control pts")
    plt.plot(augk,augk*0.0 ,"kx", ms=8,label="end-knots")
    plt.grid(which = “both” ); plt.minorticks_on()
    plt.legend(loc=0,frameon=False)
    plt.title("Spline Curve of Simple Cantilever BVP $P^{iv}(x)=-1$")
    plt.show()
if __name__ == "__main__":
    # Use  ONE of 5th, 6th 7th and 8th Order augmented knot vectors
(ONLY)
    order, augknots = 5, [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]; # 4th degree B-polys
    #order, augknots = 6, [0, 0, 0, 0, 0, 0,  1, 1, 1, 1, 1, 1]; # 5th degree B-polys
    #order, augknots = 7, [0, 0, 0, 0, 0, 0, 0,  1, 1, 1, 1, 1, 1, 1]; # sextic basls
    #order, augknots = 8, [0, 0, 0, 0, 0, 0, 0, 0,  1, 1, 1, 1, 1, 1, 1, 1];# septic
basis
    mainsolver( order,  augknots )  # ← computes  the  solution now
```

Conflict of Interest statement

No funding or any help has been received from any funding Agency or persons respectively. Hence there is no conflict of interest with any entity.

Author contribution statement

The work reported in this single-authored paper has been fully (100%) carried out by M.N. Anandaram.