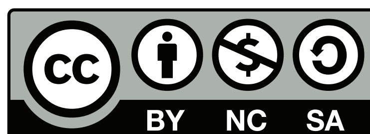
Algorithms for Interpolation

Tamas Kis | tamas.a.kis@outlook.com

Tamas Kis
https://tamaskis.github.io

Copyright © 2022 Tamas Kis.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



Contents

Contents iii

List of Algorithms iv

Preface v

I Interpolation of Data

1 Basic Mechanics 2
- 1.1 Sampling Continuous Functions Over Grids . . . 2
  - 1.1.1 Extension to Univariate, Vector-Valued Functions . . . 3
- 1.2 Interpreting Data as a Sampled Function . . . 3
- 1.3 Monotonicity of the Node Vector . . . 4
- 1.4 Finding the Query Interval . . . 4
  - 1.4.1 Special Case #1: Query Point at or Below Lower Bound of Node Vector . . . 6
  - 1.4.2 Special Case #2: Query Point at or Above Upper Bound of Node Vector . . . 6
  - 1.4.3 Algorithm . . . 6
- 1.5 Out of Bounds Queries and Extrapolation . . . 8
- 1.6 Horner's Method for Evaluating Polynomials . . . 10
  - 1.6.1 Extension to Univariate, Vector-Valued Polynomials . . . 11

2 Linear Interpolation 12
- 2.1 Linear Interpolation . . . 12

II Appendices

A Test Cases 16
- A.1 `find_interval` . . . 16
- A.2 `out_of_bounds` . . . 17
- A.3 `horners_method` . . . 17
- A.4 `interp1d_linear` . . . 18

References 20

List of Algorithms

Utilities

Algorithm 1	`find_interval`	Finds the interval containing a query point	7
Algorithm 2	`out_of_bounds`	Out of bounds handling for interpolation routines	9
Algorithm 3	`horners_method_scalar`	Horner's method for evaluating a univariate, scalar-valued polynomial	10
Algorithm 4	`horners_method`	Horner's method for evaluating a univariate polynomial	11

Linear Interpolation

Algorithm 5	`interp1d_linear`	Linear interpolation of univariate data	13

Preface

THE purpose of this text is to present commonly used interpolation routines in a standardized, easy-to-implement format. All algorithms presented in this text use 1-based indexing, and are implemented in the *Interpolation Toolbox* for MATLAB[1]. For programming languages using 0-based indexing (such as Python and C++), these algorithms require some slight adjustments.

[1] https://tamaskis.github.io/Interpolation_Toolbox-MATLAB/

PART I

Interpolation of Data

1

Basic Mechanics

1.1 Sampling Continuous Functions Over Grids

Consider the univariate, scalar-valued, continuous function $y = f(x)$ ($f : \mathbb{R} \to \mathbb{R}$), shown below in Fig. 1.1.

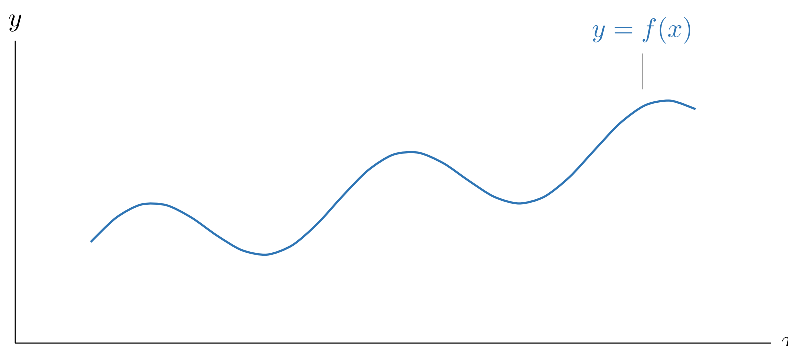


Figure 1.1: Continuous function.

Let a **node** be a single, discrete value of the independent variable x. Now, consider $N+1$ nodes $x_1, ..., x_{N+1}$, which we can store in the **node vector**[1] $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$.

$$\boxed{\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_{N+1} \end{bmatrix}} \tag{1.1}$$

If we sample $f(x)$ at every node x_i in the node vector $\mathbf{x}$, then each node x_i will have a corresponding function evaluation $y_i = f(x_i)$. See Fig. 1.2 below for a visual representation[2]. Like the nodes, we can store the function evaluations in a vector $\mathbf{y} \in \mathbb{R}^{1\times(N+1)}$, which we refer to as the **value vector**.

$$\boxed{\mathbf{y} = \begin{bmatrix} y_1 & \cdots & y_{N+1} \end{bmatrix} = \begin{bmatrix} f(x_1) & \cdots & f(x_{N+1}) \end{bmatrix}} \tag{1.2}$$

The row vectors $\mathbf{x}$ and $\mathbf{y}$ essentially form a set of data. Thus, by sampling the function over a grid, we can transition

---

[1] Also commonly referred to as a **grid** or **mesh**.
[2] While in this visualization the nodes are uniformly spaced, in general this is not the case. None of the methods presented in this text require a uniform grid.

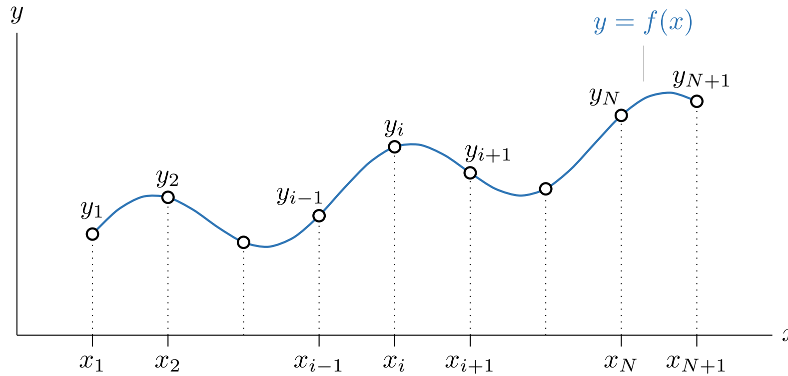


Figure 1.2: Sampling a continuous function over a grid.

from dealing with the function as a continuous function to treating it like a set of data.

$$y = f(x) \quad \xrightarrow{\text{sample}} \quad \mathbf{y} \text{ vs. } \mathbf{x}$$

1.1.1 Extension to Univariate, Vector-Valued Functions

Now, consider the univariate, vector-valued, continuous function $\mathbf{y} = \mathbf{f}(x)$ ($\mathbf{f} : \mathbb{R} \to \mathbb{R}^m$). If we sample this function at $N + 1$ points, then we still have the same node vector given by Eq. (1.1). However, instead of a value *vector*, we now have the **value matrix** $[\mathbf{y}] \in \mathbb{R}^{m\times(N+1)}$. Each column of $[\mathbf{y}]$ is a distinct value of $\mathbf{y}$.

$$\boxed{[\mathbf{y}] = \begin{bmatrix}\mathbf{y}_1 & \cdots & \mathbf{y}_{N+1}\end{bmatrix} = \begin{bmatrix}\mathbf{f}(x_1) & \cdots & \mathbf{f}(x_{N+1})\end{bmatrix}} \tag{1.3}$$

Again, $\mathbf{x}$ and $[\mathbf{y}]$ essentially form a set of data.

$$\mathbf{y} = \mathbf{f}(x) \quad \xrightarrow{\text{sample}} \quad [\mathbf{y}] \text{ vs. } \mathbf{x}$$

1.2 Interpreting Data as a Sampled Function

In the real world, functional relationships are all around us. When we collect data, there is some independent variable x for which we can measure a vector of values $\mathbf{y}$. By taking measurements at multiple values of x, and recording the corresponding values of $\mathbf{y}$, we built the data set

$$[\mathbf{y}] \text{ vs. } \mathbf{x}$$

where $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$ and $[\mathbf{y}] \in \mathbb{R}^{m\times(N+1)}$ store the values of the independent and dependent variables, respectively, and where the number of data points if $N + 1$.

In the previous section, we took a continuous function, sampled it, and said it could be treated like a set of data. Here, we take the converse viewpoint; we have a set of data, but we consider it as a sample from an underlying *unknown* function $\mathbf{f}(x)$. If we knew $\mathbf{f}(x)$, we could immediately find a value $\mathbf{y}$ for any point x.

In this case, while there is some underlying function $\mathbf{f}(x)$ that maps values of x to values of $\mathbf{y}$, we do not explicitly know it. However, we still want to evaluate it at other points, even though we can only represent it as a set of data. The interpolation methods presented in this text will allow us to (approximately) evaluate $\mathbf{f}(x)$ at any point x without ever explicitly knowing $\mathbf{f}(x)$.

1.3 Monotonicity of the Node Vector

Recall the node vector, $\mathbf{x} \in \mathbb{R}^{N+1}$. The algorithm used for finding the query interval (see Algorithm 1 in the next section) requires that the elements of the node vector be **monotonically increasing**. Mathematically, this means that

$$\boxed{x_{i+1} > x_i \; \forall \; i \in [1, N]} \tag{1.4}$$

For example, consider the node vector

$$\mathbf{x} = \begin{bmatrix} 1 & 4 & 5 & 8 & 20 & 21 \end{bmatrix}$$

This node vector is monotonically increasing. However, the node vector

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 3 & 4 & 5 \end{bmatrix}$$

is *not* a monotonically increasing vector since two of its elements are identical.

1.4 Finding the Query Interval

Consider the node vector $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$. A visual representation of the node vector is shown in Fig. 1.3 below. Now,

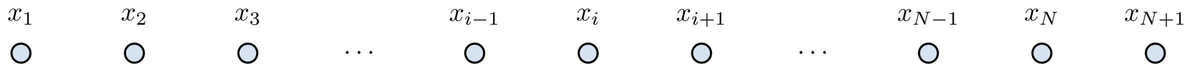


Figure 1.3: Node vector.

consider a **query point**, $x_q \in \mathbb{R}$. We want to find the **lower index**, l, and **upper index**, u, corresponding to the points x_l and x_u that form the **query interval** containing x_q.

$$\boxed{x_l \leq x_q < x_u, \qquad \text{i.e.} \quad x_q \in [l, u)} \tag{1.5}$$

Note that for any query interval, we always have

$$\boxed{u = l + 1} \tag{1.6}$$

As an example for finding l and u, see Example 1.4.1 below.

Example 1.4.1: Finding the lower and upper indices of a query interval.

Consider the node vector

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

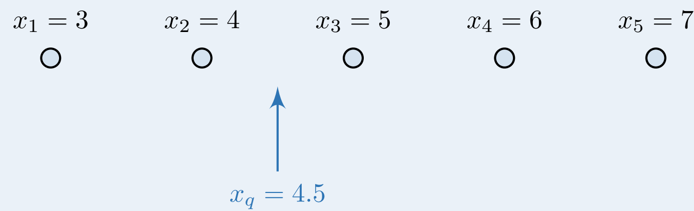
and the query point

$$x_q = 4.5$$

Find the lower and upper indices (l and u, respectively) defining the query interval for x_q.

■ ***SOLUTION***

First, let's draw a picture.



The query interval is clearly $[4, 5)$. In terms of the variables of $\mathbf{x}$, the query interval is $[x_2, x_3)$, since $x_2 = 4$ and $x_3 = 5$. Therefore, the lower and upper indices corresponding to the query interval for this query point are

$$\boxed{\begin{array}{l} l = 2 \\ u = 3 \end{array}}$$

> Note that since the query interval uses an inclusive lower bound and exclusive upper bound, if a query point is equal to the ith node, then the query interval is $[x_i, x_{i+1})$ and the lower and upper indices are $l = i$ and $u = i + 1$, respectively. This is shown in Example 1.4.2 below.

Example 1.4.2: Querying a node.

Consider the node vector

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

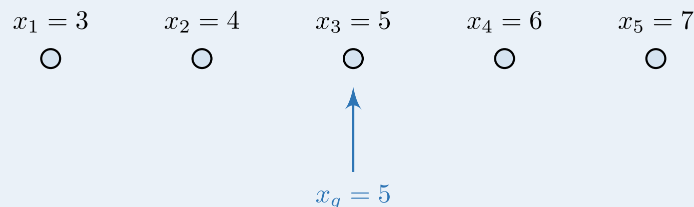
and the query point

$$x_q = 5$$

Find the lower and upper indices (l and u, respectively) defining the query interval for x_q.

■ ***SOLUTION***

First, let's draw a picture.



Since the lower bound of the query interval is defined to be inclusive, in this case the query interval is $[5, 6)$. In terms of the variables of $\mathbf{x}$, the query interval is $[x_3, x_4)$, since $x_3 = 5$ and $x_4 = 6$. Therefore, the lower and upper indices corresponding to the query interval for this query point are

$$\boxed{\begin{array}{l} l = 3 \\ u = 4 \end{array}}$$

1.4.1 Special Case #1: Query Point at or Below Lower Bound of Node Vector

Consider the case where the query point is at or below the lower bound of the node vector, i.e.

$$x_q \leq x_1$$

This case is illustrated in Fig. 1.4 below. For this special case, we manually define the query interval to be $[x_1, x_2)$

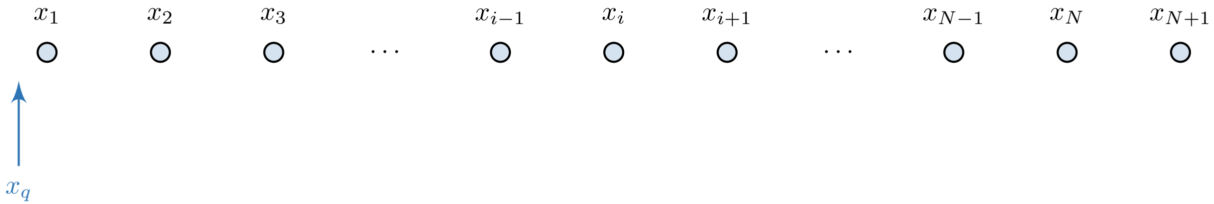


Figure 1.4: Query point at or below lower bound of node vector.

with corresponding lower and upper indices

$$\boxed{\begin{aligned} l &= 1 \\ u &= 2 \end{aligned}} \tag{1.7}$$

1.4.2 Special Case #2: Query Point at or Above Upper Bound of Node Vector

Consider the case where the query point is at or above the upper bound of the node vector, i.e.

$$x_q \geq x_{n+1}$$

This case is illustrated in Fig. 1.5 below.

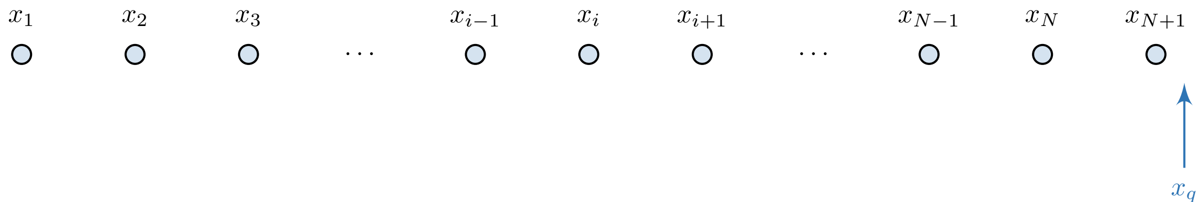


Figure 1.5: Query point at or above bound of node vector.

For this special case, we manually define the query interval to be $[x_N, x_{N+1})$ with corresponding lower and upper indices

$$\boxed{\begin{aligned} l &= N \\ u &= N+1 \end{aligned}} \tag{1.8}$$

1.4.3 Algorithm

The default method for algorithmically determining the interval containing a query point is the binary search algorithm; the basics of this algorithm for this specific problem are outlined in [1]. Algorithm 1 below is largely based on the binary search algorithm[3], but we also include the special cases discussed in Sections 1.4.1 and 1.4.2.

[3] In the case of a uniform grid, we could immediately get the lower index as

$$l = \left\lfloor \frac{x_q - x_1}{x_2 - x_1} \right\rfloor + 1$$

Algorithm 1: `find_interval`
Finds the interval containing a query point.

Inputs:

- $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$ - node vector (independent variable data)
- $x_q \in \mathbb{R}$ - query point

Procedure:

1. Determine the number of subintervals, N, given that $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$.
2. Case #1: Query point at or below lower boundary of node vector.

   if $x_q \leq x_1$
   $l = 1$
   $u = 2$

3. Case #2: Query point at or above upper boundary of node vector.

   else if $x_q \geq x_1$
   $l = N$
   $u = N + 1$

4. Case #3: Query point contained within the node vector. *Note that this procedure assumes 1-based indexing.*

   else

   (a) Initialize the lower and upper indices.

   $$l = 1$$
   $$u = N + 1$$

   (b) Binary search procedure.

   while $l < r$

   i. Midpoint index.

   $$m = \left\lfloor \frac{l+r}{2} \right\rfloor$$

   ii. Discard either the lower or upper half of the search space.

   if $x_m < x_q$
   $l = m + 1$
   else
   $u = m$
   end

However, the general overhead of the `find_interval` algorithm largely outweighs any performance boost that this direct equation would provide. Thus, it is not worth it to add additional logic to implement this case for uniform node vectors.

end

(c) Decrement lower index by 1 unless $x_l = x_q$.

if $x_l \neq x_q$
  $l = l - 1$
end

(d) Updates upper index based on value of lower index.

$u = l + 1$

5. Return the result.

return l, u

Outputs:

- $l \in \mathbb{Z}$ - lower index of interval containing query point
- $u \in \mathbb{Z}$ - upper index of interval containing query point

Note:

- The node vector, `x`, must be monotonically increasing (see Section 1.3).
- If the query point is extremely close to one of the nodes, the subinterval returned might be inaccurate. For example, in MATLAB, $(3 - \varepsilon) < 3$ evaluates as `false`, so if 3 were a node, this algorithm would return the wrong interval for $3 - \varepsilon$.

Test Cases:

- See Appendix A.1.

1.5 Out of Bounds Queries and Extrapolation

In Sections 1.4.1 and 1.4.2, we defined how to find the query interval when the query point was outside the bounds of the node vector. We refer to such queries as **out of bounds queries**; see Figs. 1.4 and 1.5 for illustrations of such queries.

Extrapolation is when we use an *interpolation* method to return a value for a query point outside the bounds of the node vector. In Section 1.4, we introduced a standardized way for determining the query interval for out of bounds queries (and have incorporated this in Algorithm 1 for finding the query interval). All interpolation routines in this text are set up such that when Algorithm 1 is used to find the query interval for an out of bounds query, the appropriate query interval is returned for use with extrapolation.

In general, there are four potential ways we handle out of bounds queries in interpolation routines. These four methods are listed below (note that the first is just extrapolation, which we just discussed):

1. **Extrapolation.** By default, all the algorithms we introduce in this text automatically perform extrapolation without the need for any additional logical; the query interval returned by Algorithm 1 will automatically be the query interval needed for extrapolation in the case of out of bounds queries.
2. **Hold the endpoint value.** Instead of extrapolating, one option is to hold the value at the nearest endpoint; for example, if a query point is below the lower bound of the node vector, we assume the corresponding value is the value at the lower bound of the node vector.
3. **Error.** Often, if a query point is outside the bounds of the node vector, this hints at a larger issue (i.e. we should not be getting out of bounds query points in the first place). In such cases, it is useful for an interpolation routine

to raise an error.

4. **Default value.** Sometimes, we just want to use a default value if the query point is outside the bounds of the node vector.

As mentioned, extrapolation is performed by default for all interpolation routines in this text without needing any additional logic. However, the remaining three methods require that we implement additional logic in each interpolation routine. Since this logic is the same with any interpolation routine, we formalize it as Algorithm 2 below.

Algorithm 2: `out_of_bounds`
Out of bounds handling for interpolation routines.

Inputs:

- $x_q \in \mathbb{R}$ - query point
- $x_l \in \mathbb{R}$ - lower bound of node vector
- $x_u \in \mathbb{R}$ - upper bound of node vector
- $\mathbf{y}_l \in \mathbb{R}^m$ - lower bound of value matrix
- $\mathbf{y}_u \in \mathbb{R}^m$ - upper bound of value matrix
- `method` - specifies what should occur if a query point is outside the bounds of the node vector; there are three options for what can be input:
    1. `'hold'` – holds the value from the endpoint
    2. `'error'` – raises an error
    3. provide a default value that is used any time the query point is out of bounds

Procedure:

1. Raise an error.

   if `method` = `'error'`
       `error`(Query point is out of bounds.)

2. Default the value.

   else if (`method` is a number) **or** (`method` is `NaN`)
       $\mathbf{y}_q =$ `method`

3. Hold the value from the relevant endpoint.

   else if `method` = `'hold'`
       if $x_q < x_l$
           $\mathbf{y}_q = \mathbf{y}_l$
       else
           $\mathbf{y}_q = \mathbf{y}_u$
       end

4. Raise an error for an invalid method.

   else
       `error`(Invalid out of bounds handling method specified.)
   end

5. Return the result.

   return $\mathbf{y}_q$

Outputs:

- $\mathbf{y}_q \in \mathbb{R}^m$ - manually set value for $\mathbf{y}$ at the query point

Test Cases:

- See Appendix A.1.

1.6 Horner's Method for Evaluating Polynomials

Consider the nth degree, univariate, scalar-valued polynomial $p_n : \mathbb{R} \to \mathbb{R}$.

$$\boxed{p_n(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n} \tag{1.9}$$

To efficiently evaluate this polynomial at the evaluation point $x_0 \in \mathbb{R}$, we can use Horner's method. Horner's method makes use of the **coefficient vector**, $\mathbf{c} \in \mathbb{R}^{1\times(n+1)}$, which is a *row* vector storing the coefficients of the polynomial [2, p. 95].

$$\boxed{\mathbf{c} = \begin{bmatrix} c_0 & \cdots & c_n \end{bmatrix}} \tag{1.10}$$

Note that if we are using 1-based indexing, then the ith element of $\mathbf{c}$ corresponds to the $(i-1)th$ coefficient.

Algorithm 3: `horners_method_scalar`
Horner's method for evaluating a univariate, scalar-valued polynomial.

Inputs:

- $x \in \mathbb{R}$ - evaluation point
- $\mathbf{c} \in \mathbb{R}^{1\times(n+1)}$ - coefficient vector

Procedure:

1. Determine the degree of the polynomial, n, given that $\mathbf{c} \in \mathbb{R}^{1\times(n+1)}$.
2. Initialize Horner's algorithm.

   $y = c_n$

3. Recursively compute $p_n(x)$.

   for $i = n-1$ **to** 0 **by** -1
   | $y = xy + c_i$
   end

4. Evaluation of $p_n(x)$.

   return y

Outputs:

- $y \in \mathbb{R}$ - evaluation of $p_n(x)$

1.6.1 Extension to Univariate, Vector-Valued Polynomials

Consider the nth degree, univariate, vector-valued polynomial $\mathbf{p}_n : \mathbb{R} \to \mathbb{R}^m$.

$$\boxed{\mathbf{p}_n(x) = \mathbf{c}_0 + \mathbf{c}_1 x + \mathbf{c}_2 x^2 + \cdots + \mathbf{c}_n x^n} \tag{1.11}$$

Now, the coefficients are no longer scalars, but rather vectors, i.e. $\mathbf{c}_i \in \mathbb{R}^m$. To efficiently evaluate this polynomial at an arbitrary point $x \in \mathbb{R}$, we can extend Horner's method to the vector-valued case. First, we now have a **coefficient *matrix*** $\mathbf{C} \in \mathbb{R}^{m \times (n+1)}$, where the ith column stores the $(i-1)th$ coefficient *vector*.

$$\boxed{\mathbf{C} = \begin{bmatrix} \mathbf{c}_0 & \cdots & \mathbf{c}_n \end{bmatrix}} \tag{1.12}$$

Note that we write the vector-valued version of Horner's method in terms of column vectors of $\mathbf{C}$ instead of the coefficient vectors $\mathbf{c}_i$.

Algorithm 4: `horners_method`

Horner's method for evaluating a univariate polynomial.

Inputs:

- $x \in \mathbb{R}$ - evaluation point
- $\mathbf{C} \in \mathbb{R}^{m \times (n+1)}$ - coefficient matrix

Procedure:

1. Determine the degree of the polynomial, n, given that $\mathbf{C} \in \mathbb{R}^{m \times (n+1)}$.
2. Initialize Horner's algorithm.

   $$\mathbf{y} = \mathbf{C}_{:,n+1}$$

3. Recursively compute $\mathbf{p}_n(x)$.

   for $i = n$ **to** 1 **by** -1

   $\quad \mathbf{y} = x\mathbf{y} + \mathbf{C}_{:,i}$

   end

4. Evaluation of $\mathbf{p}_n(x)$.

   return y

Outputs:

- $\mathbf{y} \in \mathbb{R}^m$ - evaluation of $\mathbf{p}_n(x)$

Test Cases:

- See Appendix A.3.

2

Linear Interpolation

2.1 Linear Interpolation

Consider an unknown, univariate, scalar-valued function $y = f(x)$ ($f : \mathbb{R} \to \mathbb{R}$). We want to approximate $y_q = f(x_q)$, where x_q is our query point. We do not know $f(x)$ specifically, but have a value vector $\mathbf{y}$ (vector of dependent variable values) corresponding to the node vector $\mathbf{x}$ (vector of independent variable values). Using Algorithm 1, we can find the lower and upper indices (l and r, respectively), corresponding to the query interval for x_q. With the indices l and r, we can find the two points, (x_l, y_l) and (x_u, y_u), between which to interpolate. The value, y_q, corresponding to the query point, x_q, obtained via linear interpolation can be found by constructing a line connecting the lower and upper endpoints and evaluating the equation of that line at the query point, x_q. First, the line passing through (x_l, y_l) and (x_u, y_u) has slope

$$m = \frac{y_u - y_l}{x_u - x_l}$$

A point on this line is (x_l, y_l). Using point-slope form, we can find an equation for this line.

$$y - y_l = m(x - xl) = \left(\frac{y_u - y_l}{x_u - x_l}\right)(x - x_l) \quad \to \quad y = y_l + \left(\frac{y_u - y_l}{x_u - x_l}\right)(x - x_l) \tag{2.1}$$

Evaluating at the query point, $x = x_q$, we get [3]

$$y_q = y_l + \left(\frac{y_u - y_l}{x_u - x_l}\right)(x_q - x_l) = y_l + \left(\frac{x_q - x_l}{x_u - x_l}\right)(y_u - y_l) = \left[1 - \frac{x_q - x_l}{x_u - x_l}\right] y_l + \left(\frac{x_q - x_l}{x_u - x_l}\right) y_u$$

Let

$$\boxed{p = \frac{x_q - x_l}{x_u - x_l}} \tag{2.2}$$

represent the **interpolation proportion**, or the proportion of the distance that x_q is from the x_l with respect to the width of the query interval. Then

$$y_q = (1 - p) y_l + p y_u$$

We can easily generalize this to the vector-valued case where the dependent variable is vector-valued. In this case, we still have the node vector $\mathbf{x} \in \mathbb{R}^{1 \times (N+1)}$, but instead of the value vector we now have the value matrix $[\mathbf{y}] \in \mathbb{R}^{m \times (N+1)}$. The linear interpolation becomes

$$\boxed{\mathbf{y}_q = (1 - p)\mathbf{y}_l + p\mathbf{y}_u} \tag{2.3}$$

where $\mathbf{y}_l \in \mathbb{R}^m$ and $\mathbf{y}_u \in \mathbb{R}^m$ are the lth and rth columns of $[\mathbf{y}]$, respectively, and where the interpolation proportion, p, is still given by Eq. (2.2).

In general, we can be given a query *vector*, $\mathbf{x}_q \in \mathbb{R}^{1\times q}$ that contains q query points where we want to find a value for $\mathbf{y}$. We can just loop over these query points, and return a value for $\mathbf{y}$ in each of them. We store these interpolated results as the matrix $[\mathbf{y}]_q \in \mathbb{R}^{m\times q}$. Algorithm 5 below formalizes a calculation procedure for performing linear interpolation.

Algorithm 5: `interp1_linear`

Linear interpolation of univariate, vector-valued data.

Inputs:

- $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$ - node vector (independent variable data)
- $[\mathbf{y}] \in \mathbb{R}^{m\times(N+1)}$ - value matrix (dependent variable data)
- $\mathbf{x}_q \in \mathbb{R}^{1\times q}$ - query vector (i.e. where to interpolate)
- `bounds` - **(OPTIONAL)** specifies what should occur if a query point is outside the bounds of the node vector; there are four options for what can be input (defaults to `'extrapolate'`):
    1. `'extrapolate'` – performs extrapolation
    2. `'hold'` – holds the value from the endpoint
    3. `'error'` – raises an error
    4. provide a default value that is used any time the query point is out of bounds

Procedure:

1. Default `bounds` to `extrapolate` if not input.
2. Determine if extrapolation will be performed.

$$\texttt{extrapolation} = (\texttt{bounds == 'extrapolate'})$$

3. Determine the number of subintervals, N, given that $\mathbf{x} \in \mathbb{R}^{1\times(N+1)}$.
4. Determine the number of query points, q, given that $\mathbf{x}_q \in \mathbb{R}^q$.
5. Determine the dimension of the dependent variable, m, given that $[\mathbf{y}] \in \mathbb{R}^{m\times(N+1)}$.
6. Preallocate the matrix $[\mathbf{y}_q] \in \mathbb{R}^{m\times q}$ to store the interpolated values.
7. Perform linear interpolation at each query point.

for $i = 1$ **to** q

(a) Find the lower and upper indices defining the query interval (Algorithm 1).

$$[l, u] = \texttt{find_interval}(\mathbf{x}, x_{q,i})$$

(b) Interpolation proportion for the ith query point.

$$p = \frac{x_{q,i} - x_l}{x_u - x_l}$$

(c) Linear interpolation for ith query point.

$$[\mathbf{y}_q]_{:,i} = (1 - p)[\mathbf{y}]_{:,l} + p[\mathbf{y}]_{:,u}$$

(d) Handle the edge case where x_q is out of bounds and we're not extrapolating (Algorithm 2).

if (**not** `extrapolate`) **and** $((x_{q,i} < x_1)$ **or** $(x_{q,i} > x_{N+1}))$

$$[\mathbf{y}_q]_{:,i} = \texttt{out_of_bounds}(x_{q,i}, x_1, x_{N+1}, y_1, y_{N+1}, \texttt{bounds})$$

end

end

8. Interpolated values of $\mathbf{y}$ at the query point(s).

return $[\mathbf{y}_q]$

Outputs:

- $[\mathbf{y}_q] \in \mathbb{R}^{m \times q}$ - interpolated values of $\mathbf{y}$ at the query point(s)

Note:

- The node vector, $\mathbf{x}$, must be monotonically increasing (see Section 1.3).

Test Cases:

- See Appendix A.4.

PART II

Appendices

A

Test Cases

A.1 `find_interval`

Two Nodes

Node Vector: $\mathbf{x} = \begin{bmatrix} 1 & 2 \end{bmatrix}$

x_q	**1-Based Indexing** l	u	**0-Based Indexing** l	u
0	1	2	0	1
0.5	1	2	0	1
1	1	2	0	1
1.5	1	2	0	1
2	1	2	0	1
2.5	1	2	0	1

Odd Number of Nodes

Node Vector: $\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$

x_q	**1-Based Indexing** l	u	**0-Based Indexing** l	u
1.5	1	2	0	1
2.5	2	3	1	2
3.5	3	4	2	3
4.5	4	5	3	4
1	1	2	0	1
2	2	3	1	2
3	3	4	2	3
4	4	5	3	4
5	4	5	3	4

0	1	2	0	1
6	4	5	3	4

Even Number of Nodes

Node Vector: $\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$

	1-Based Indexing		**0-Based Indexing**	
x_q	l	u	l	u
1.5	1	2	0	1
2.5	2	3	1	2
3.5	3	4	2	3
4.5	4	5	3	4
5.5	5	6	3	4
1	1	2	0	1
2	2	3	1	2
3	3	4	2	3
4	4	5	3	4
5	5	6	4	5
6	5	6	4	5
0	1	2	0	1
7	5	6	4	5

A.2 out_of_bounds

x_q	x_l	x_u	$\mathbf{y}_l$	$\mathbf{y}_u$	method	$\mathbf{y}_q$
0	1	2	3	4	'hold'	3
3	1	2	3	4	'hold'	4
0	1	2	3	4	'error'	N/A *(function should raise error)*
3	1	2	3	4	'error'	N/A *(function should raise error)*
0	1	2	3	4	NaN	NaN
3	1	2	3	4	NaN	NaN
0	1	2	3	4	0	0
3	1	2	3	4	0	0

A.3 horners_method

Scalar-Valued Test Cases

x	$\mathbf{c} = \begin{bmatrix} c_0 & \cdots & c_n \end{bmatrix}$	$y = \sum_{i=0}^{n} c_i x^i$
5	1	1
5	$\begin{bmatrix} 1 & 2 \end{bmatrix}$	11
5	$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$	86
5	$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$	586

Vector-Valued Test Cases

x	$\mathbf{C} = \begin{bmatrix} \mathbf{c}_0 & \cdots & \mathbf{c}_n \end{bmatrix}$	$\mathbf{y} = \sum_{i=0}^{n} \mathbf{c}_i x^i$
5	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$
5	$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$	$\begin{bmatrix} 11 \\ 22 \end{bmatrix}$
5	$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix}$	$\begin{bmatrix} 86 \\ 172 \end{bmatrix}$
5	$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \end{bmatrix}$	$\begin{bmatrix} 586 \\ 1172 \end{bmatrix}$

A.4 `interp1d_linear`

Two Nodes

Node Vector: $\mathbf{x} = \begin{bmatrix} 1 & 2 \end{bmatrix}$

Value Vector: $\mathbf{y} = \begin{bmatrix} 1 & 2 \end{bmatrix}$

$\mathbf{x}_q$	`method`	$[\mathbf{y}_q]$
1	N/A	1
1.25	N/A	1.25
1.5	N/A	1.5
1.75	N/A	1.75
2	N/A	2
0	`'extrapolate'`	0
0.25	`'extrapolate'`	0.25
0.5	`'extrapolate'`	0.5
0.75	`'extrapolate'`	0.75
2.25	`'extrapolate'`	2.25
2.5	`'extrapolate'`	2.5
2.75	`'extrapolate'`	2.75
0.5	`'hold'`	1
0.75	`'hold'`	1

2.25	'hold'	2
5	'hold'	2
0.5	'error'	N/A *(function should raise error)*
0.75	'error'	N/A *(function should raise error)*
2.25	'error'	N/A *(function should raise error)*
5	'error'	N/A *(function should raise error)*

Three Nodes

Node Vector: $\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

Value Vector: $\mathbf{y} = \begin{bmatrix} 1 & 2 & 4 \end{bmatrix}$

method = 'extrapolate'

$\mathbf{x}_q$	$[\mathbf{y}_q]$
0	0
0.25	0.25
0.5	0.5
0.75	0.75
1	1
1.25	1.25
1.5	1.5
1.75	1.75
2	2
2.25	2.5
2.5	3
2.75	3.5
3	4
3.25	4.5
3.5	5
3.75	5.5
4	6

Vector-Valued Case

Node Vector: $\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

Value Vector: $\mathbf{y} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix}$

method = 'extrapolate'

$\mathbf{x}_q$	$[\mathbf{y}_q]$
0	$(0,0)^T$
0.25	$(0.25,0.5)^T$
0.5	$(0.5,1)^T$
0.75	$(0.75,1.5)^T$
1	$(1,2)^T$

1.25	$(1.25, 2.5)^T$
1.5	$(1.5, 3)^T$
1.75	$(1.75, 3.5)^T$
2	$(2, 4)^T$
2.25	$(2.25, 4.5)^T$
2.5	$(2.5, 5)^T$
2.75	$(2.75, 5.5)^T$
3	$(3, 6)^T$
3.25	$(3.25, 6.5)^T$
3.5	$(3.5, 7)^T$
3.75	$(3.75, 7.5)^T$
4	$(4, 8)^T$

Bibliography

[1] *Binary search algorithm: Procedure for finding the leftmost element*. Wikipedia. Accessed: April 5, 2022. URL: https://en.wikipedia.org/wiki/Binary_search_algorithm#Procedure_for_finding_the_leftmost_element.

[2] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. 9th ed. Boston, MA: Brooks/Cole, Cengage Learning, 2011.

[3] *Linear interpolation*. Wikipedia. Accessed: April 5, 2022. URL: https://en.wikipedia.org/wiki/Linear_interpolation.