E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Lab Assignment 10: Interpolation; Taylor Series
Version: revision03

Due date: Friday April 7th 2017 at 12 pm (noon).

General instructions, guidelines, and comments:

- For each question, you will have to write and submit one or more Matlab functions. We provide a number of test cases that you can use to test your function. The fact that your function works for all test cases provided does not guarantee that it will work for all possible test cases relevant to the question. It is your responsibility to test your function thoroughly, to ensure that it will also work in situations not covered by the test cases provided. During the grading process, your function will be evaluated on a number of test cases, some of which are provided here, some of which are not.
- Submit on bCourses one m-file for each function that you have to write. The name of each file must be the name of the corresponding function, with the suffix `.m` appended to it. For example, if the name of the function is `my_function`, the name of the file that you have to submit is `my_function.m`. **Carefully check the name of each file that you submit.** Do not submit any zip file. If you re-submit a file that you have already submitted, bCourses may rename the file by adding a number to the file's name (*e.g.,* rename `my_function.m` into `my_function-01.m`). This behavior is okay and should be handled seamlessly by our grading system. Do not rename the file yourself as a response to this behavior.
- A number of optional Matlab toolboxes can be installed alongside Matlab to give it more functionality. All the functions that you have to write to complete this assignment can, however, be implemented without the use of any optional Matlab toolboxes. We encourage you to not use optional toolboxes to complete this assignment. All functions of the Matlab base installation will be available to our grading system, but functions from optional toolboxes may not. If one of your function uses a function that is not available to our grading system, you will loose all points allocated to the corresponding part of this assignment. To guarantee that you are not using a Matlab function from an optional toolbox that is not available to our grading system, use one or both of the following methods:
  - ◇ Only use functions from the base installation of Matlab.
  - ◇ Make sure that your function works on the computers of the 1109 Etcheverry Hall computer lab. All the functions available on these computers will be available to our grading system.

- For this assignment, the required submissions are:
  - ◇ `my_linear_interpolation.m`
  - ◇ `my_lagrange.m`
  - ◇ `my_d_polynomial.m`
  - ◇ `my_splines_coefficients.m`
  - ◇ `my_splines_values.m`
  - ◇ `my_taylor_approx.m`

Built-in functions that you may not use

You may not use Matlab's built-in functions `interp1`, `interp2`, `interp3`, `interpft`, `interpn`, `spline`, `pchip`, `mkpp`, `ppval`, `polyval`, `polyvalm`, `polyfit`, and `polyder`. This rule applies to each question of this lab assignment.

Where to find information about the methods used in this lab?

Interpolation techniques were covered in lecture L25 (March 20$^{\text{th}}$ 2017). Chapter 14 of the textbook also covers this topic. Taylor series were covered in lecture L26 (March 22$^{\text{th}}$ 2017). Chapter 15 of the textbook also covers this topic.

1. Linear interpolation

In this question, you will write a function that performs linear interpolation. More precisely, write a function with the following header:

```
function [y_interp] = my_linear_interpolation(x_data, y_data, x_interp)
```

where:

- `x_data` and `y_data` are two $m \times 1$ arrays of class `double` that represent two-dimensional data. In other words, these column vectors represent a set of points of coordinates (`x_data(i)`, `y_data(i)`), $i = \{1, 2, \ldots, m\}$. You can assume that $m > 1$, that all the elements of `x_data` and `y_data` are different from `NaN`, `Inf`, and `-Inf`, and that the elements of `x_data` are different from each other and sorted in increasing order.
- `x_interp` is a $p \times q$ array of class `double` that represents x-values at which to calculate interpolated values. You can assume that each element of this array is different from `NaN`, `Inf`, and `-Inf`, and is within the range of values defined by `x_data`.
- `y_interp` is a $p \times q$ array of class `double` that represents the interpolated y-values calculated by linear interpolation (`y_interp(i,j)` is the interpolated value at $x =$`x_interp(i,j)`). For each element `x_interp(i,j)` of `x_interp`, the linear interpolation should be conducted between the two successive data points (in `x_data` and `y_data`) whose x-values bracket `x_interp(i,j)`.

Test cases:

```
>> y_interp = my_linear_interpolation([2; 4], [10; 20], [2.5, 3, 3.5])
y_interp =
   12.5000   15.0000   17.5000

>> x_data = transpose(1:10);
>> y_data = transpose(cos(1:10));
>> y_interp = my_linear_interpolation(x_data, y_data, 1.5:9.5)
y_interp =
  Columns 1 through 7
    0.0621   -0.7031   -0.8218   -0.1850    0.6219    0.8570    0.3042
  Columns 8 through 9
   -0.5283   -0.8751

>> y_interp = my_linear_interpolation(x_data, y_data, [2, 3, 4; 9, 8, 7])
y_interp =
   -0.4161   -0.9900   -0.6536
   -0.9111   -0.1455    0.7539
```

2. Lagrange polynomial

Consider a set of m data points of coordinates $(x_i, y_i), i = \{1, 2, \ldots, m\}$ such that:

- All the x_i's are different.
- The x_i's are ordered in increasing order.

The Lagrange polynomial corresponding to this data set is the polynomial of least degree that "passes through all the points" of the data set. This polynomial is the polynomial L such that:

$$L(x) = \sum_{i=1}^{m} y_i l_i(x) \tag{1}$$

where the l_i's, called the "Lagrange basis polynomials", are themselves polynomials, and are such that:

$$l_i(x_j) = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i \end{cases} \tag{2}$$

The Lagrange basis polynomials are given by:

$$l_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{m} \frac{x - x_j}{x_i - x_j} \tag{3}$$

The Lagrange basis polynomials l_i's and the resulting Lagrange polynomial are of degree $(m-1)$ or lower, and can therefore be written in the following form:

$$a_0 + a_1 x + \cdots + a_{m-2} x^{m-2} + a_{m-1} x^{m-1} \tag{4}$$

where x is the independent variable and the a_i's, $i = \{0, 1, ..., m-1\}$ are the coefficients of the polynomial.

In this question, you will write a function that calculates the coefficients of the Lagrange basis polynomials, and of the corresponding Lagrange polynomial itself. More precisely, write a function with the following header:

```
function [coefficients] = my_lagrange(x_data, y_data)
```

where:

- `x_data` and `y_data` are two $m \times 1$ arrays of class `double` that represent two-dimensional data. In other words, these column vectors represent a set of points of coordinates (`x_data(i)`, `y_data(i)`), $i = \{1, 2, \ldots, m\}$. You can assume that $m > 1$, that all the elements of `x_data` and `y_data` are different from `NaN`, `Inf`, and `-Inf`, and that the elements of `x_data` are different from each other and sorted in increasing order.
- `coefficients` is a $(m+1) \times m$ array of class `double` that represents the Lagrange basis polynomials and the Lagrange polynomial corresponding to the x- and y-data represented by `x_data` and `y_data`, respectively. Each row of `coefficients` represents the coefficients of a polynomial of degree $(m-1)$ or lower *i.e.* the coefficients $a_0, a_1, ..., a_{m-1}$ of this polynomial (see Equation 4), in this order. The first row of `coefficients` represents the polynomial l_1, the second row of `coefficients` represents the polynomial l_2, and so on. The last row of `coefficients` represents the Lagrange polynomial.

There exist multiple approaches to solve this question. One approach consists of "directly" applying Equations 1 and 3. This approach does not involve solving systems of linear algebraic equations. Another approach consists of solving systems of linear algebraic equations that reflect the fact that a Lagrange polynomial "passes through all the points" of the corresponding data set. If you decide to use this approach, you may wish to recognize that the Lagrange basis polynomial l_i is itself the Lagrange polynomial of the following data set:

$$(x_1, 0), (x_2, 0), \ldots, (x_{i-1}, 0), (x_i, 1), (x_{i+1}, 0), \ldots, (x_{m-1}, 0), (x_m, 0) \tag{5}$$

Test cases:

```
>> % Test case 1 (see Figure 1)
>> % Note: for this test case, you may get a slightly different value for
>> % coefficients(4,5), depending on the algorithm you used
>> % in your function. Your value of coefficients(4,5) should,
```

```
>> % in any case, be smaller in magnitude than 1e-15. In other words,
>> % your value of coefficients(4,5) should be such that:
>> % abs(coefficients(4,5)) < 1e-15
>> format shorte
>> x_data = [-4.5; -3; -2.4;   2;  4; 5.9];
>> y_data = [  -2; -3;    2; 7.2; -4; 1.6];
>> coefficients = my_lagrange(x_data, y_data)
coefficients =
   1.8776e-01  -3.1823e-02  -5.6067e-02   7.5470e-03   3.5912e-03  -5.5249e-04
  -1.8183e+00   5.1022e-01   4.4138e-01  -9.9554e-02  -1.7835e-02   3.5670e-03
   2.1637e+00  -7.8744e-01  -3.9949e-01   1.0985e-01   1.4941e-02  -3.3956e-03
   6.8553e-01   3.7891e-01  -4.1070e-02  -3.8471e-02  -8.8960e-19   8.9654e-04
  -2.6421e-01  -7.9984e-02   6.8851e-02   2.4125e-02  -1.3821e-03  -6.9107e-04
   4.5532e-02   1.0118e-02  -1.3596e-02  -3.4957e-03   6.8509e-04   1.7566e-04
   1.5472e+01   2.2403e-02  -2.6039e+00   1.2418e-01   8.2828e-02  -6.8868e-03

>> % Test case 2 (see Figure 2)
>> format shorte
>> x_data = [-121; -94; -20; 15; 41];
>> y_data = [71; -12; 10; 57; 107];
>> coefficients = my_lagrange(x_data, y_data)
coefficients =
   1.9244e-02  -5.8537e-04  -6.4729e-05   9.6536e-07   1.6644e-08
  -5.0621e-02   1.6600e-03   1.6534e-04  -2.8911e-06  -3.4013e-08
   4.3837e-01  -3.1630e-02  -3.1961e-06   9.9643e-06   6.2668e-08
   6.9139e-01   3.0775e-02  -4.4767e-04  -1.4381e-05  -7.4130e-08
  -9.8374e-02  -2.1997e-04   3.5026e-04   6.3426e-06   2.8830e-08
   3.5240e+01   1.3529e+00   5.3487e-03   6.1812e-05   1.0760e-06
```

Figures 1 and 2 show the data used for the test cases above, as well as the corresponding polynomials $y_i l_i$, $i = \{1, 2, \ldots, m\}$ and Lagrange polynomials L.

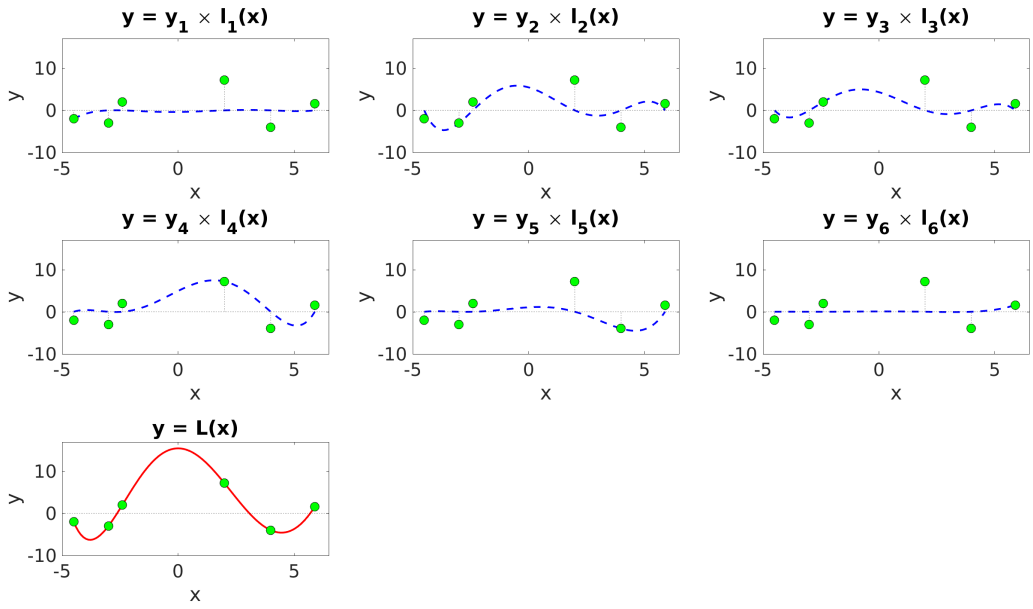

Figure 1: Original data (green dots), and corresponding polynomials $y_i l_i$, $i = \{1, 2, \ldots, m\}$ (blue dashed lines) and Lagrange polynomial L (red solid line) for test case 1.

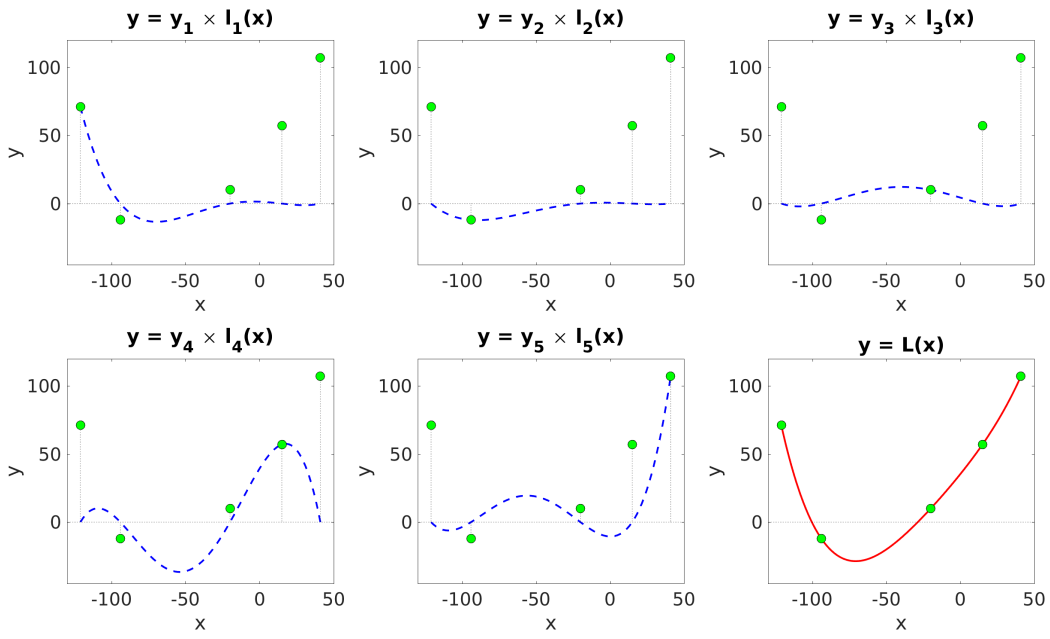

Figure 2: Original data (green dots), and corresponding polynomials $y_i l_i$, $i = \{1, 2, \ldots, m\}$ (blue dashed lines) and Lagrange polynomial L (red solid line) for test case 2.

3. Derivatives of polynomials

In this question, you will write a function that calculates the coefficients of the derivatives of real polynomials. Consider a real polynomial P of degree n or lower. This polynomial can be written as:

$$P(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_1 x + b_0 \tag{6}$$

where the b_i's, $i = \{0, 1, \ldots, n\}$ are real coefficients.

The derivative of this polynomial is the polynomial P' such that:

$$P'(x) = n b_n x^{n-1} + (n-1) b_{n-1} x^{n-2} + \cdots + 2 b_2 x + b_1 \tag{7}$$

This polynomial can be written in the form:

$$P'(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0 \tag{8}$$

where the c_i's, $i = \{0, 1, \ldots, n\}$ are real coefficients that can be calculated from the b_i's (note

that $c_n = 0$). By applying this reasoning multiple times, we can show that the polynomial P, and any of its derivatives (first, second, and higher derivatives), can be written in the form:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \tag{9}$$

In this question, you will write a function that calculates the coefficients of the k^{th} derivative of a polynomial, given the coefficients of this polynomial. More precisely, write a function with the following header:

```
function [coefficients_d] = my_d_polynomial(coefficients, k)
```

where:

- `coefficients` is a $1 \times (n+1)$ array of class `double` that represents the coefficients $a_n, a_{n-1}, \ldots, a_0$ (in this order) of a real polynomial P of degree n or lower, as given by Equation 9. You can assume that $n \geqslant 0$ and that all the elements of `coefficients` are different from `NaN`, `Inf`, and `-Inf`.
- `k` is a scalar of class `double` that represents an integer that is greater than or equal to `0`.
- `coefficients_d` is a $1 \times (n+1)$ array of class `double` that represents the coefficients $a_n, a_{n-1}, \ldots, a_0$ (in this order) of the polynomial $P^{(k)}$, as given by Equation 9. $P^{(k)}$ is the `k`$^{\text{th}}$ derivative of P. Note that $P^{(0)} = P$ (*i.e.* the 0^{th} derivative of P is P itself).

Test cases:

```
>> coefficients_d = my_d_polynomial([1, 1, 1, 1], 0)
coefficients_d =
     1     1     1     1

>> coefficients_d = my_d_polynomial([1, 1, 1, 1], 1)
coefficients_d =
     0     3     2     1

>> coefficients_d = my_d_polynomial([1, 1, 1, 1], 2)
coefficients_d =
     0     0     6     2

>> coefficients_d = my_d_polynomial([1, 1, 1, 1], 5)
coefficients_d =
     0     0     0     0

>> coefficients_d = my_d_polynomial([4, 0, 0, -3, 0, 0, 1, -2], 0)
coefficients_d =
     4     0     0    -3     0     0     1    -2

>> coefficients_d = my_d_polynomial([4, 0, 0, -3, 0, 0, 1, -2], 1)
coefficients_d =
     0    28     0     0   -12     0     0     1
```

```
>> coefficients_d = my_d_polynomial([4, 0, 0, -3, 0, 0, 1, -2], 2)
coefficients_d =
     0     0   168     0     0   -36     0     0
```

4. Cubic splines

Consider a set of m data points of coordinates $(x_i, y_i), i = \{1, 2, \ldots, m\}$ such that:

- All the x_i's are different.
- The x_i's are ordered in increasing order.

In this question, you will write a function that performs cubic spline interpolation on such data sets. For each interval $[x_i, x_{i+1}]$, $i = \{1, 2, \ldots, m-1\}$, you should determine the polynomial of degree three (the cubic spline S_i), such that for all $i = \{1, 2, \ldots, m-1\}$:

- The spline S_i "goes through" the points (x_i, y_i) and (x_{i+1}, y_{i+1});
- The first derivatives of the splines S_i and S_{i+1} are equal at $x = x_{i+1}$;
- The second derivatives of the splines S_i and S_{i+1} are equal at $x = x_{i+1}$; and
- The conditions described by your function's input parameter `condition` are satisfied (see below for details).

4.1. Coefficients of the splines

Write a function with the following header:

```
function [coefficients] = my_splines_coefficients(x_data, y_data, condition)
```

where:

- `x_data` and `y_data` are two $m \times 1$ arrays of class `double` that represent two-dimensional data. In other words, these column vectors represent a set of points of coordinates (`x_data(i)`, `y_data(i)`), $i = \{1, 2, \ldots, m\}$. You can assume that $m > 3$, that all the elements of `x_data` and `y_data` are different from `NaN`, `Inf`, and `-Inf`, and that the elements of `x_data` are different from each other and sorted in increasing order..
- `condition` is a 1×1 `struct` array (see below for details).
- `coefficients` is a $(m-1) \times 4$ array of class `double` that represents the coefficients of the cubic splines corresponding to the x- and y-data represented by `x_data` and `y_data`, respectively. In `coefficients`, each row represents one spline: row number `i` represents the spline between the points of coordinates (`x_data(i),y_data(i)`) and (`x_data(i+1),y_data(i+1)`). The four values in the row represent the coefficients a, b, c, and d of the corresponding spline, where the equation of the spline is $y = ax^3 + bx^2 + cx + d$.

The input parameter `condition` has a field named `'type'`, which can take one of three values:

- `'natural'`. In this case, the cubic splines determined by your function should be natural splines: the second derivative of the corresponding spline should be set to zero at the first and last point of the data set. In other words, $S_1''(x_1) = 0$ and $S_{m-1}''(x_m) = 0$.
- `'clamped'`. In this case, the cubic splines determined by your function should be clamped: the first derivative of the corresponding spline should be set to `condition.left` at the first point of the data set, and the first derivative of the corresponding spline should be set to `condition.right` at the last point of the data set. In other words, $S_1'(x_1) =$ `condition.left` and $S_{m-1}'(x_m) =$ `condition.right`. `condition.left` and `condition.right` are scalars of class `double` that are different from `NaN`, `Inf`, and `-Inf`.
- `'not-a-knot'`. In this case, the cubic splines determined by your function should satisfy the "not-a-knot" conditions: the third derivatives of the corresponding splines should match at the second point of the data set, and the third derivatives of the corresponding splines should match at the one-before-last point of the data set. In other words, $S_1'''(x_2) = S_2'''(x_2)$ and $S_{m-2}'''(x_{m-1}) = S_{m-1}'''(x_{m-1})$.

The fields `'left'` and `'right'` will be present in `condition` if and only if `condition.type` is `'clamped'`.

Test cases:

```
>> x_data = [-4.5; -3; -2.4;   2;  4; 5.9];
>> y_data = [  -2; -3;    2; 7.2; -4; 1.6];
>> condition = struct('type', 'natural');
>> coefficients = my_splines_coefficients(x_data, y_data, condition)
coefficients =
    1.4882   20.0912   86.3951  115.5470
   -4.7647  -36.1852  -82.4340  -53.2820
    0.0274   -1.6822    0.3731   12.9636
    0.8658   -6.7125   10.4337    6.2566
   -0.6450   11.4170  -62.0844  102.9473

>> condition = struct('type', 'not-a-knot');
>> coefficients = my_splines_coefficients(x_data, y_data, condition)
coefficients =
   -2.2109  -17.6020  -38.1667  -18.7755
   -2.2109  -17.6020  -38.1667  -18.7755
    0.0243   -1.5085    0.4578   12.1240
    0.6024   -4.9770    7.3948    7.4994
    0.6024   -4.9770    7.3948    7.4994

>> condition = struct('type', 'clamped', 'left', 0, 'right', 0);
>> coefficients = my_splines_coefficients(x_data, y_data, condition)
coefficients =
    2.9639   35.1225  136.0449  169.0577
   -5.7514  -43.3156  -99.2694  -66.2566
    0.0157   -1.7921    0.3870   13.4685
    1.0970   -8.2800   13.3628    4.8180
   -1.6936   25.2078 -120.5883  183.4195

>> x_data = [-121; -94; -20; 15; 41; 56; 65; 97];
```

```
>> y_data = [7; 12; 3; 5.7; -1; 2; 9; 2];
>> condition = struct('type', 'natural');
>> coefficients = my_splines_coefficients(x_data, y_data, condition)
coefficients =
   -0.0001   -0.0324   -3.6749 -121.0262
    0.0001    0.0113    0.4326    7.6734
   -0.0002   -0.0044    0.1186    5.5807
    0.0003   -0.0288    0.4845    3.7513
    0.0006   -0.0566    1.6241  -11.8231
   -0.0030    0.5389  -31.7230  610.6571
    0.0004   -0.1236   11.3398 -322.3706

>> condition = struct('type', 'not-a-knot');
>> coefficients = my_splines_coefficients(x_data, y_data, condition)
coefficients =
    0.0001    0.0102    0.4156    7.6788
    0.0001    0.0102    0.4156    7.6788
   -0.0002   -0.0048    0.1161    5.6824
    0.0004   -0.0301    0.4956    3.7851
    0.0004   -0.0278    0.4031    5.0491
   -0.0011    0.2173  -13.3211  261.2330
   -0.0011    0.2173  -13.3211  261.2330

>> condition = struct('type', 'clamped', 'left', 0, 'right', 0);
>> coefficients = my_splines_coefficients(x_data, y_data, condition)
coefficients =
   -0.0003   -0.0952   -9.6984 -310.6667
    0.0001    0.0129    0.4642    7.7601
   -0.0002   -0.0042    0.1229    5.4849
    0.0003   -0.0287    0.4913    3.6430
    0.0007   -0.0668    2.0531  -17.7014
   -0.0037    0.6557  -38.4097  737.6027
    0.0010   -0.2477   20.3140 -534.7427
```

Figure 3 shows the cubic splines corresponding to the two sets of data points used in the previous test cases.

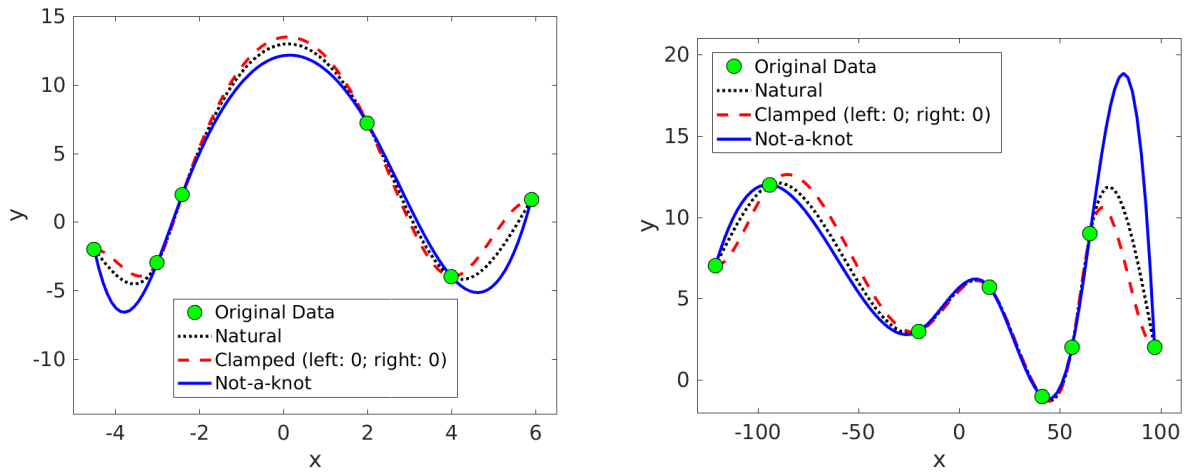

Figure 3: Original data (green dots), and corresponding cubic splines determined using different conditions. The left panel shows the data corresponding to the first data set used in the test cases above. The right panel shows the data corresponding to the second data set used in the test cases above.

4.2. Interpolate data

Write a function with the following header:

```
function [y_interp] = my_splines_values(x_data, y_data, condition, x_interp)
```

where:

- `x_data`, `y_data`, and `condition` represent the same quantities as in the function `my_splines_coefficients`.
- `x_interp` is a $p \times q$ array of class `double` that represents x-values at which to calculate interpolated values. You can assume that all the elements of this array are different from `NaN`, `Inf`, and `-Inf`, and are within the range of values defined by `x_data`.
- `y_interp` is a $p \times q$ array of class `double` that represents the interpolated y-values as determined by the cubic splines corresponding to `x_data`, `y_data`, and `condition` (`y_interp(i,j)` is the interpolated value at $x =$`x_interp(i,j)`)

Test cases:

```
>> x_data = [-4.5; -3; -2.4;   2;  4; 5.9];
>> y_data = [  -2; -3;    2; 7.2; -4; 1.6];
>> condition = struct('type', 'natural');
>> y_interp = my_splines_values(x_data, y_data, condition, -4:5)
y_interp =
  Columns 1 through 7
   -3.8216   -3.0000    5.2694   10.8809   12.9636   11.6819    7.2000
```

```
  Columns 8 through 10
    0.5206   -4.0000   -2.6781

>> condition = struct('type', 'not-a-knot');
>> y_interp = my_splines_values(x_data, y_data, condition, -4:5)
y_interp =
  Columns 1 through 7
   -6.2449   -3.0000    4.9798   10.1334   12.1240   11.0976    7.2000
  Columns 8 through 10
    1.1554   -4.0000   -4.6516

>> condition = struct('type', 'clamped', 'left', 0, 'right', 0);
>> y_interp = my_splines_values(x_data, y_data, condition, -4:5)
y_interp =
  Columns 1 through 7
  -2.8521   -3.0000    5.4004   11.2737   13.4685   12.0792    7.2000
  Columns 8 through 10
    0.0066   -4.0000   -1.0283

>> x_data = [-121; -94; -20; 15; 41; 56; 65; 97];
>> y_data = [7; 12; 3; 5.7; -1; 2; 9; 2];
>> condition = struct('type', 'natural');
>> x_interp = [-100, -50, 0; 10, 20, 30];
>> y_interp = my_splines_values(x_data, y_data, condition, x_interp)
y_interp =
   11.4294    5.9880    5.5807
    6.1272    4.6776    1.6817

>> condition = struct('type', 'not-a-knot');
>> y_interp = my_splines_values(x_data, y_data, condition, x_interp)
y_interp =
   11.8398    5.3758    5.6824
    6.1738    4.6236    1.5590

>> condition = struct('type', 'clamped', 'left', 0, 'right', 0);
>> y_interp = my_splines_values(x_data, y_data, condition, x_interp)
y_interp =
   10.8285    6.8517    5.4849
    6.0925    4.7067    1.7358
```

5. Taylor series approximation of a cubic polynomial

Consider a real-valued function that is C^∞ (*i.e.* infinitely differentiable) over some interval I, and a point $a \in I$. Then, for any value x of this interval I, the value of $f(x)$ can be calculated using the corresponding Taylor series:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots \tag{10}$$

$$= f(a) + \sum_{k=1}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k \tag{11}$$

where we will call a the "center point" of the Taylor series. One can use a truncated version of this Taylor series to approximate the function f over the interval I:

$$f(x) \approx f(a) + \sum_{k=1}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k \tag{12}$$

Consider a rechargeable battery, whose state of charge is measured by a number z that varies from 0 (battery is discharged) to 1 (battery is fully charged). In this question, we assume that the voltage V delivered by the battery is a cubic function of its state of charge:

$$V(z) = p_3 z^3 + p_2 z^2 + p_1 z + p_0 \tag{13}$$

In this question, you will write a function that measures how well V can be approximated using Taylor series. More precisely, write a function with the following header:

```
function [rmse] = my_taylor_approx(p, a, n)
```

where:

- `p` is a 1×4 array of class `double` that represents the coefficients p_3, p_2, p_1, and p_0 (in this order) of Equation 13. You can assume that all the elements of `p` are different from `NaN`, `Inf`, and `-Inf`.
- `a` is a scalar of class `double` that represents the center point to use for the Taylor series (*i.e.* a in Equation 12). You can assume that $\texttt{0} \leqslant \texttt{a} \leqslant \texttt{1}$.
- `n` is a scalar of class `double` that represents n in Equation 12. You can assume that `n` is either `0`, `1`, or `2`.
- `rmse` is a scalar of class `double` that represents the root mean squared error between the true value of the voltage, as given by Equation 13, and the corresponding Taylor series approximation, as given by equation 12 when applied to Equation 13.

To calculate `rmse`:

1. Create a vector of 100 equally-spaced values of z ranging from 0 to 1.
2. Calculate the values V_i, $i = \{1, 2, \ldots, 100\}$ of the voltage delivered by the battery at each of these points, as given by Equation 13.

3. Calculate the values $\hat{V}_i$, $i = \{1, 2, \ldots, 100\}$ of the voltage delivered by the battery at each of these points, as predicted by the Taylor series approximation of the function V, using `a` as the center point, and using a number of terms defined by `n`.
4. Calculate the root mean squared error as:

$$\sqrt{\frac{1}{m}\sum_{i=1}^{m}(V_i - \hat{V}_i)^2} \tag{14}$$

   where m is the number of points (here: 100) where V is calculated.

Test cases:

```
>> [rmse] = my_taylor_approx([1.7, -2.5, 1.6, 3.5], 0.5, 2)
rmse =
    0.0827

>> [rmse] = my_taylor_approx([1.7, -2.5, 1.6, 3.5], 0.3, 1)
rmse =
    0.0452

>> [rmse] = my_taylor_approx([1.7, -2.5, 1.6, 3.5], 0.5, 0)
rmse =
    0.1882
```