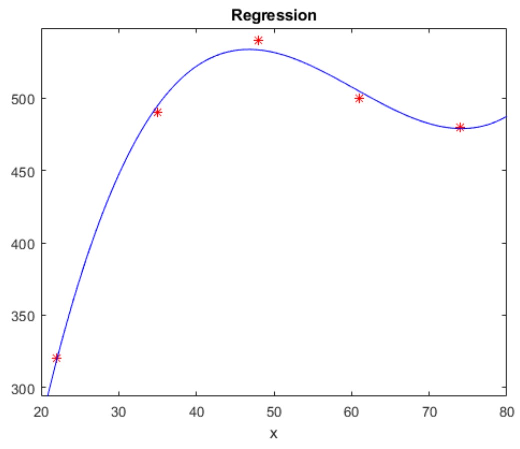
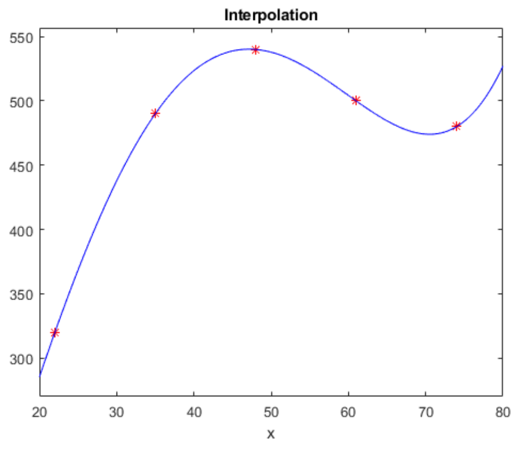
# INTERPOLATION

In the previous chapter, we dealt with regression and function fitting, when we determined the best fitting line, parabola, exponential function, etc. for the given points. In the case of regression, we already have knowledge about the function (or model) of the physical phenomenon examined and we would like to determine the parameters of the function from our measured values. The fitted function usually does not go through the measured points, but ideally it goes as close as possible to them.

Interpolation is a mathematical relationship that perfectly represents the given data points, returns the measurement values at these points, and adequately describes the behavior of the function between the points. Graphically represented, the function passes through the measurement points.





## GLOBAL INTERPOLATION WITH A SINGLE POLYNOMIAL

The general form of an algebraic polynomial:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

The coefficients $a_n, a_{n-1}, \cdots, a_1, a_0$ are real numbers, and $n$ is a non-negative integer, the degree of the polynomial. A polynomial of the first degree is a linear function whose graph is a straight line. The graph of a second degree (quadratic) polynomial is a parabola. Polynomials of a higher degree are some kind of curves, the higher the degree, the more 'bends' and inflection points it can have, the more complicated its graph can be.

If we have a dataset consisting of *n* points, this can be approximated with polynomials of different degrees, up to degree (*n-1*). In the case of fitting a polynomial of a lower degree, we can speak of regression, and in case of a polynomial of degree (n-1) we get interpolation. In the latter case, the polynomial goes through all points. This will be the case of polynomial interpolation. This is a global interpolation, because we fit a single function to all data. Let's look at an example!

The power of wind turbines varies with the wind speed. In an experiment, we measured power at different wind speeds, see table below!

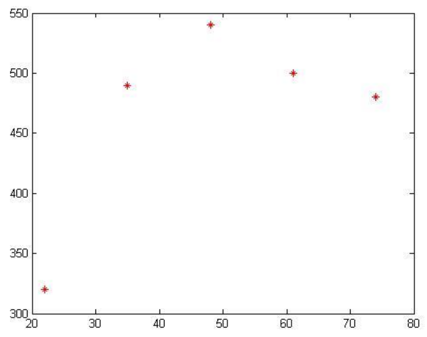
| Wind speed [km/h] | 22 | 35 | 48 | 61 | 74 |
|---|---|---|---|---|---|
| Power [W] | 320 | 490 | 540 | 500 | 480 |

We are looking for answers to the following questions:

1) What is the power of the wind turbine in the case of 42 and 68 km/h winds?

2) At what wind speed will the power be 400 W?

The questions can be answered using interpolation or regression. A fourth degree interpolation polynomial can be fitted to the 5 measurements. Load the windturbine.txt file!



```
>  % windturbine
>  clear all; close all; clc;
>  data = load('windturbine.txt')
>  v = data(:,1) % wind speed
>  p = data(:,2) % power
>  figure(1)
>  plot(v,p,'r*')
```

The coefficients of the fourth degree interpolation polynomial $f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ can be calculated by solving a system of linear equations, in the same way as we already learned from regression, by writing 5 equations for the five points. In contrast to regression, here we have exactly as many measurements as unknowns, so the matrix of coefficients will be square. This coefficient matrix *A* is also called the Vandermonde matrix.

$$\begin{aligned} y_1 &= a_4x_1^4 + a_3x_1^3 + a_2x_1^2 + a_1x_1 + a_0 \\ y_2 &= a_4x_2^4 + a_3x_2^3 + a_2x_2^2 + a_1x_2 + a_0 \\ y_3 &= a_4x_2^4 + a_3x_2^3 + a_2x_2^2 + a_1x_2 + a_0 \\ y_4 &= a_4x_3^4 + a_3x_3^3 + a_2x_3^2 + a_1x_3 + a_0 \\ y_5 &= a_4x_4^4 + a_3x_4^3 + a_2x_4^2 + a_1x_4 + a_0 \end{aligned} \quad \rightarrow A = \begin{pmatrix} {x_1}^4 & {x_1}^3 & {x_1}^2 & x_1 & 1 \\ {x_2}^4 & {x_2}^3 & {x_2}^2 & x_2 & 1 \\ {x_3}^4 & {x_3}^3 & {x_3}^2 & x_3 & 1 \\ {x_4}^4 & {x_4}^3 & {x_4}^2 & x_4 & 1 \\ {x_5}^4 & {x_5}^3 & {x_5}^2 & x_5 & 1 \end{pmatrix}; b = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix}$$

Of course, we can also use the built-in Matlab functions **polyfit** and **polyval** here.

```
>  a = polyfit(v,p,4) %  0.0001   -0.0171    0.5627   12.0190  -62.0517
```

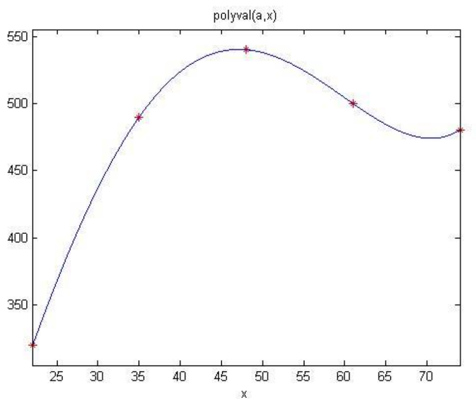
Using **polyfit**, we have to define the degree of the polynomial we want to fit and we will get the coefficients of the polynomial from the highest degree term back to the constant term: $a_4 = 0.0001$; $a_3 = -0.0171$; $a_2 = 0.5627$; $a_1 = 12.0190$; $a_0 = -62.0517$.

The first question can be easily answered after this, using the fitted polynomial. The **polyval** command can calculate the value of the polynomial at any place from the specified coefficients, e.g. the power at the 42 km/h in question:

```
>  polyval(a,42) % 531.7853
```

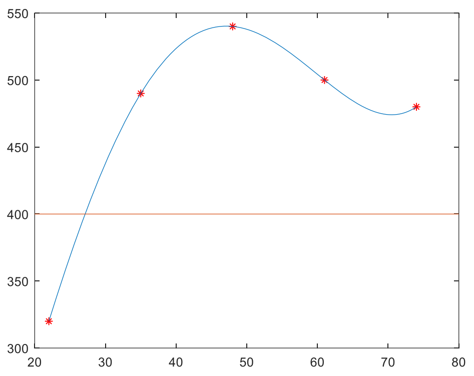
Or we can also define a function using the calculated coefficients and the **polyval** command, and then we can plot the function!

```
>  fp = @(x) polyval(a,x)
>  fp(68) % 476.5008
>  hold on;
>  fplot(fp,[min(v) max(v)])
```



However, in order to answer the second question, at what wind speed the power will be 400 W, we have to solve a non-linear equation! The nonlinear equation to be solved must be equal to zero: $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 - 400 = 0.$ Let's plot the line y=400 in the figure to chose initial value from the figure!

```
>  plot(xlim,[400,400])
>  h = @(x) fp(x)-400
>  x400 = fzero(h,30) % 27.1296
```



So, at a wind of 42 km/h, the power is 532 W, and at 68 km/h, 477 W. We get a power of 400 W at a wind of 27 km/h. Let's look at the result using regression, not interpolation! Let's rewrite the degree of the polynomial in the **polyfit** command from 4 to 3, 2, 1, and check the result!

To fit an nth degree polynomial written in standard form: $f(x) = a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$ requires a system of equations consisting of (n+1) linear equations. To write the equations, the coordinates of all available points must be substituted into the general form. In the case of fitting higher degree polynomials, our coefficient matrix will mostly be ill-conditioned, as it will contain very large and comparably small numbers as well and the solution will be numerically unstable. Let's look at the condition number of the coefficient matrix of our previous example! For this, we need to define the coefficient matrix, which we could do in the same way as we did with regression methods, but Matlab has a command for generating the Vandermonde matrix, so let's use it now!

```
>  A = [v.^4 v.^3 v.^2 v.^1 v.^0] % coeff. matrix traditionally
>  A = vander(v) % other solution: Vandermonde matrix
>  cond(A) %  1.5378e+09
```

This number is already very large (of the order of $10^9$), even though we only fitted a polynomial of degree 4, that is, not a very high degree. As a check, let's look at solving the system of linear equations using the Vandermonde matrix to see if it gives the same result as **polyfit**!

```
>  x = A\p % 0.0001; -0.0171; 0.5627; 12.0190; -62.0517
```

## LAGRANGE AND NEWTON INTERPOLATION POLYNOMIALS

Only one interpolation polynomial can be written through given points. However, this polynomial can be given in several forms. In addition to the general form, let's briefly look at two others, the Lagrange and Newton interpolation polynomials! In many cases, it is more appropriate to use these instead of the standard definition, as there is no need to solve an ill-conditioned task.

## LAGRANGE INTERPOLATION POLYNOMIAL

This type of polynomial can be written from the coordinates of the points without any calculations or solving a system of equations. It can be defined for *n* points in the following form:

$$f(x) = \sum_{i=1}^{n} y_i L_i(x) = \sum_{i=1}^{n} y_i \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)}$$

where $L_i(x) = \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{(x-x_j)}{(x_i-x_j)}$ are called Lagrangian functions. Writing the equation for two points:

$$f(x) = \frac{(x - x_2)}{(x_1 - x_2)} y_1 + \frac{(x - x_1)}{(x_2 - x_1)} y_2$$

Writing the equation for three points:

$$f(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} y_1 + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} y_2 + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} y_3$$

- It can be seen that the interpolation polynomial can be defined using the coordinates of the points without any prior calculations.
- It is difficult to work with this type of interpolation polynomial, because for each point to be interpolated, we have to rewrite the entire equation for that point. It is not enough to just substitute the coefficients, as in the general form, since all the multipliers in the numerators have x.
- If our point set is expanded with a new point, then all the Lagrange functions must be recalculated, in this it differs from the Newton form, where only the new terms must be calculated in the case of new points.

## NEWTON'S INTERPOLATION POLYNOMIAL[1]

The Newton's polynomial can be expressed recursively using the so-called divided differences. General form of the Newton's polynomial:

$$f(x) = a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2) + \cdots + a_n(x - x_1)(x - x_2)\cdots(x - x_{n-1})$$

The coefficients for two points are as follows:

$$a_1 = y_1;\ a_2 = \frac{y_2 - y_1}{x_2 - x_1}$$

Let's define the first-order divided differences as follows:

$$f[x_{i+1}, x_i] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

This is actually the slope of a line and equals the coefficient $a_2$ for two points.

[1] Supplementary material for home study

The second-order divided difference $f[x_3, x_2, x_1]$, can be written for three points, which is the difference of the two first-order divided differences divided by $(x_3 - x_1)$, this will be the coefficient $a_3$ in the Newton's polynomial (the first two coefficients are the same as before).

$$a_3 = f[x_3, x_2, x_1] = \frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1} = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1}$$

Similarly, the third-order divided difference can be written for four points by dividing the difference of two second-order divided differences by $(x_4 - x_1)$, and so on. The divided differences is a recursive division process. In general, the k-order divided difference

$$f[x_{i+k}, \cdots, x_i] = \frac{f[x_{i+k}, \cdots, x_{i+1}] - f[x_{i+k-1}, \cdots, x_i]}{x_{i+k} - x_i}, (k = 1,2, \cdots, n) \acute{e}s\ (i = 0, \cdots, n-k).$$

- It can be seen that in this case also, the interpolation polynomial can be defined using the coordinates of the points without any prior calculations.
- Working with this polynomial is no longer so difficult, once the $a_1 \cdots a_n$coefficients have been determined, they can be used for the interpolation of any point.
- If a new point is added to our set of points, it is not necessary to recalculate all coefficients, only the new term. In this way, the set can be easily expanded with a new point, and the points do not have to be in order.

## LOCAL INTERPOLATION

The following dataset represents the characteristic curve of a water reservoir. Given a certain water level (*H*) in the reservoir in cm, using the characteristic curve, we can find the volume (*V*) of the water in the reservoir in $10^6$ m$^3$ and the area of the water surface (*F*) in km$^2$. The dataset is located in the reservoir.txt file.

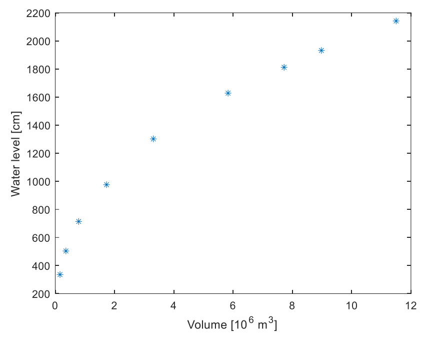
| Water level H [cm] | Volume V [$10^6$ m$^3$] | Surface F [km$^2$] |
|---|---|---|
| 336 | 0.16 | 0.05 |
| 504 | 0.36 | 0.09 |
| 714 | 0.79 | 0.19 |
| 976 | 1.73 | 0.37 |
| 1302 | 3.31 | 0.62 |
| 1628 | 5.83 | 0.90 |
| 1812 | 7.72 | 1.05 |
| 1932 | 8.98 | 1.16 |
| 2142 | 11.50 | 1.27 |

Plot the water level vs volume curve and use an interpolation function to find the following:

1) The volume corresponding to a water level of 15 m,
2) The water level corresponding to a volume of 11 million m$^3$?

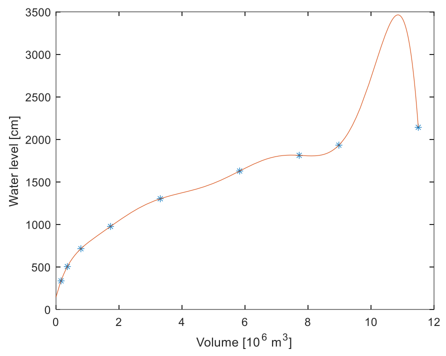
We can answer the questions with the help of interpolation. We plot the characteristic curve describing the volume, as usual, with the volume on the horizontal axis and the water level on the vertical axis. Load the data for the characteristic curves from the reservoir.txt file!

```
> clc; clear all; close all;
> data = load('reservoir.txt')
> H = data(:,1); % cm
> V = data(:,2); % million m^3
> F = data(:,3); % km^2
>
> figure(1)
> plot(V,H,'*'); hold on;
> ylabel(' Water level [cm]')
> xlabel(' Volume [10^6 m^3]')
```



In the case of interpolation, we can fit a polynomial of degree (*n-1*) to *n* points. Let's see what this means in the present case. Now we have 9 data, we can fit a global polynomial of the 8th degree to it!

```
> n = length(V) % 9
> a1 = polyfit(V,H,n-1)
> p1 = @(x) polyval(a1,x)
> fplot(p1,[0,max(V)])
```



What happened? Can the above curve be used for interpolation? Obviously not. If we were to use this curve for the section between 9-12 million $m^3$, we would get an incorrect answer.

This is a case of overfitting, when the high-degree polynomial perfectly reproduces the values of the known points, but begins to oscillate between them (especially at the edges). Between the measured points, the behavior of the curve differs significantly from the trend of the data, so it cannot be used reliably for interpolation and extrapolation. This oscillation phenomenon is called the Runge phenomenon. If there are few points, when the degree of the polynomial is low, the global interpolation polynomials are mostly suitable. In the case of many points, when the degree of the global interpolation polynomial is high, another solution must be sought!

If we query the condition number of the Vandermonde matrix, we get a very large number (of the order of $10^{10}$), which means an ill-conditioned matrix

```
> A = vander(V);
> cond(A) % 2.7260e+10
```

This is the case where fitting a global interpolation polynomial does not give good results. In general, for many data, only lower order regression polynomials or local interpolation such as spline interpolation can be used

## SPLINE INTERPOLATION

If we want to perform interpolation between many points, a better result can be achieved by using several polynomials of low degree than with one global polynomial of high degree. Each low-degree polynomial is valid only for a specific section, interval, between two or more points. In general, all polynomials have the same degree, they differ only in the coefficients.

The simplest interpolation is linear interpolation, when first degree polynomials (straight lines) are used to connect the points. Using second degree (quadratic) or third degree (cubic) polynomials, the points are connected by curves. This is a polynomial interpolation using different parameters for each section, which is called spline interpolation. This is actually a kind of local interpolation, since only the surrounding points are taken into account for each section.

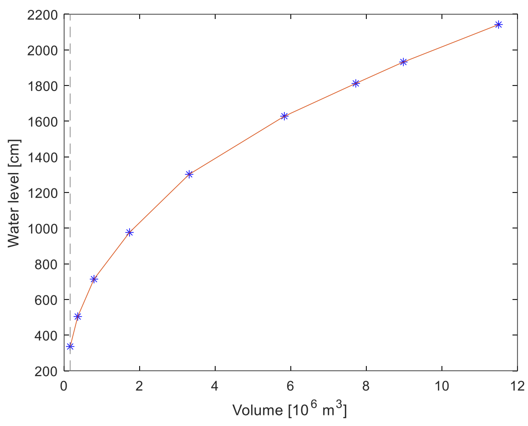
In the case of the nth-degree spline, nth-degree polynomial sections are connected to each other. The simplest case is to fit linear functions (first degree polynomials) between the given points. This is Euler's broken lines method. This results in a continuous curve, with a discontinuous derivative. However, it is often necessary that not only the fitted function should be continuous at the vertices, but also its derivatives. A solution to this can be, for example, the application of a quadratic spline, where the derivatives of the function at the vertices must be the same on both the right and left sides. This is the quadratic (second order) spline.

## LINEAR (SPLINE) INTERPOLATION

Let's fit linear functions between the points! For a given *n* point, there are (*n-1*) sections for which we have to define (*n-1*) straight lines. The easiest way to do this is with the help of the Lagrange polynomial written for two points (equation of a line for two points):

$$f_i(x) = \frac{(x - x_{i+1})}{(x_i - x_{i+1})} y_1 + \frac{(x - x_i)}{(x_{i+1} - x_i)} y_2$$

In Matlab, the **interp1** function can be used in general for spline interpolation. With this command, we can specify in a parameter what kind of spline interpolation we want to use, if we do not specify anything, the default is linear interpolation. Possible interpolation methods for **interp1**: '**linear**' - default, '**nearest**' - nearest neighbor interpolation, '**pchip**' - Piecewise Cubic Hermite Interpolating Polynomial (first order), '**spline**' - cubic second-order spline interpolation.



```
>  figure(2);
>  plot(V,H,'b*');hold on;
>  ylabel('Water level [cm]');
>  xlabel('Volume [10^6 m^3]');
>  sp = @(x) interp1(V,H,x)
>  fplot(sp,[0,max(V)])
```

In the case of linear interpolation, our function will be continuous, but there will be breaks in the slopes at the vertices. This is a continuous curve, with a discontinuous derivative (in general).  If we want to get a smoother function, a spline of a higher degree must be used.

## QUADRATIC SPLINE INTERPOLATION[2]

In the case of a quadratic (second order) spline, quadratic polynomials ($y = a \cdot x^2 + b \cdot x + c$) are fitted between the given points so that the first derivatives of the function on the right and left sides are the same at the connection points.

A quadratic polynomial has 3 unknown coefficients. For *n* points, we have to define (*n-1*) quadratic polynomials for the (*n-1*) sections, i.e. we will have 3*(n-1)=3n-3 unknowns. Let's write the known conditions in the form of equations to determine the parameters!

- All polynomials must pass through the endpoints of the segments, the vertices. Two equations can be written for each section in the following form: $f_i(x) = a_i \cdot x^2 + b_i \cdot x + c_i$, that is, we can write a total of *2*(n-1)=2n -2* equations.
- At the intermediate (*n-2*) point, the derivatives must also match ($f'(x) = 2a_i x + b_i$). So (*n-2*) equations can be written in the following form: $2\,a_{i-1}x_i + b_{i-1} = 2\,a_i x_i + b_i$
- The above two conditions only define (*3n-4*) equations, and we have (*3n-3*) unknowns. One more condition must be specified. There are several options here, for example, the value of the second derivative should be 0 at the first (or the last) point: $f_1''(x) = 2a_1 = 0$, so $a_1 = 0$. This actually means connecting the first two points (or the last two points) with a straight line

To fit quadratic splines, we need to define (*3n-4*) equation. The use of cubic, second-order splines is more widespread, partly because the second derivatives (curvature) are also the same in that case and therefore the function is smoother. And partly because a form can be derived where only (*n-2*) equations have to be solved. Matlab's built-in functions include only cubic splines, not quadratic splines.

## SECOND ORDER CUBIC SPLINE INTERPOLATION

Now let's look at the most commonly used spline, the cubic, second-order spline interpolation! In this case, we fit cubic (third degree) polynomials between the given points. Each polynomial can be written in the form:

$$y = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$$

At the vertices, the first and second order derivatives of the adjacent splines are set to be equal. For a given *n* point, we have (*n-1*) sections, and we have to compute a spline for each section. Each cubic polynomial has 4 unknown coefficients, which results in *4*(n-1)=4n-4* unknowns. We can use the following equations determined by the constraints:

- Each spline has to go through the starting and the end point of the section. This gives us *2*(n-1)=2n-2* equations in the following form: $y_i = a_i \cdot x^3 + b_i \cdot x^2 + c_i \cdot x + d_i$

[2] Supplementary material for home study

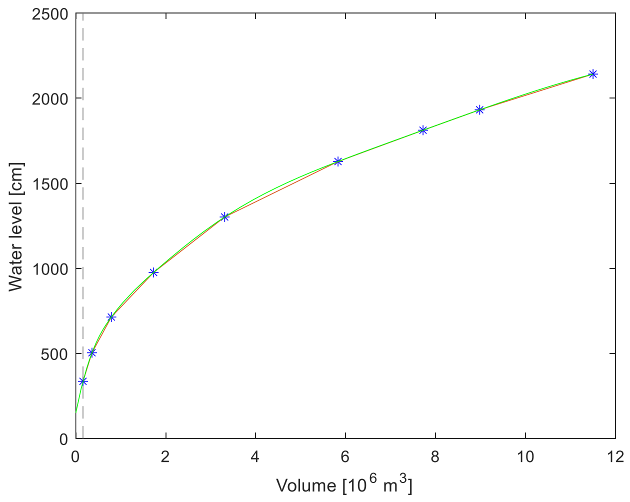
- There are *(n-2)* central points (not counting the starting and the end point). At each of these points, the first order differentials ($f'(x) = 3ax^2 + 2bx + c$ ) of the adjacent splines are set equal: $3\, a_{i-1}x_i{}^2 + 2b_{i-1}x_i + c_{i-1} = 3\, a_i x_i{}^2 + 2b_i x_i + c_i$
- At each of the central points, the second order differentials ($f''(x) = 6ax + 2b$) are also set equal: $6\, a_{i-1}x_i + 2b_{i-1} = 6\, a_i x_i + 2b_i$

These give us (*4n-6*) equations to solve for (*4n-4*) unknowns, so we still have to set two constraints, which can be done in several ways. One possibility is to set the second order differentials at the end and the starting points to zero, in this case, we get the so-called natural cubic spline. (that is, the first and last segments are straight). Another method used by Matlab's built-in function is the "not-a-knot" solution. This means that the third derivatives in the second and penultimate points are set equal as well, that is, we fit one polynomial to the first 3 and the last 3 points (the second point from the front and back is not a real node).

Note: Another form of the cubic second-order spline can be derived, where it is enough to solve a system of equations consisting of (*n-2*) linear equations, instead of (*4n-4*) equations. This derivation starts from the second derivatives and uses the Lagrange interpolation form. See e.g.: Amos Gilat, Vish Subramaniam (2011): Numerical Methods, An Introduction with Applications Using MATLAB, John Wiley & Sons

In Matlab, let’s fit a cubic spline to the characteristic curve of the reservoir! For this, we can use the **interp1** command again, only now the method should be '**spline**', instead of the default 'linear'!

```
> sp2 = @(x) interp1(V,H,x,'spline')
> fplot(sp2,[0,max(V)])
> sp2 = @(x) interp1(V,H,x,'spline')
> fplot(sp2,[0,max(V)] ,'g')
```



We can see that this resulted in a smoother function. As discussed earlier, Matlab does not use the natural spline, but the not-a-knot condition. This means that the third derivatives in the second and penultimate points are also the same. Since we are talking about third degree polynomials, in these cases all the coefficients of the polynomials will be the same. So, we actually fit a polynomial to the first 3 and another one for the last 3 points. The name of the command also comes from this fact, that the second and penultimate points are not real nodes.

Since the cubic, second-order spline interpolation is the most commonly used spline, Matlab has a separate command ('**spline**') for this, which does the same thing as the **interp1** command with the '**spline**' method.

```
>  sp2 = @(x) spline(V,H,x)
```

Let's return to the original question and, based on the fitted spline, determine how much water level belongs to a volume of 11 million $m^3$, and how much water volume belongs to a 15 m (1500 cm) water level!

```
>  % The water level corresponding to a volume of 11 million m^3:
>  H15 = sp2(11) % 2106 cm
>  % The volume corresponding to a water level of 15 m:
>  f = @(x) sp2(x)-1500
>  V1500 = fzero(f,5) % 4.6699 million m^3
```

## FIRST ORDER CUBIC SPLINE INTERPOLATION

The **interp1** command implements another cubic spline interpolation method in Matlab, this is the '**pchip**' method (piecewise cubic Hermite interpolation polynomial). This is a cubic first-order interpolation, where we also fit third-degree polynomials to each section, but at the vertices, only the first order derivatives of the adjacent splines are set to be equal, the second derivatives not (that's why we call it first-order spline). This is also called cubic Hermite interpolation, where to replace the missing conditions, Matlab determines the derivatives at the vertices from numerical differentiation. Let's see when this method can be useful!

Let us now take an example related to geodesy. To calculate the Earth's gravitational field, the changes in the Earth's density must be known. The Earth's density (ρ) varies with its radius (R) approximately as follows:

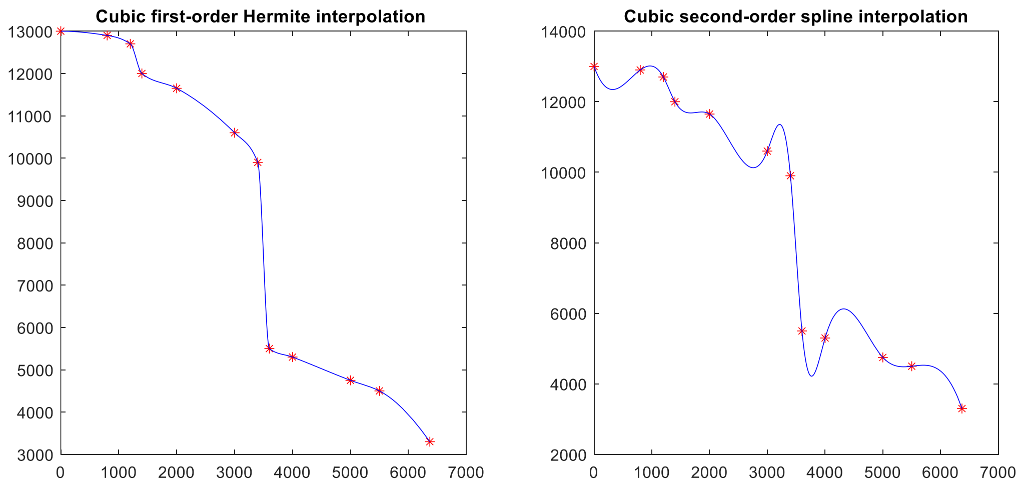
| Radius [km] | 0 | 800 | 1200 | 1400 | 2000 | 3000 | 3400 | 3600 | 4000 | 5000 | 5500 | 6370 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Density [kg/$m^3$] | 13000 | 12900 | 12700 | 12000 | 11650 | 10600 | 9900 | 5500 | 5300 | 4750 | 4500 | 3300 |

1) Let's fit a cubic spline curve to the radius-density values, and calculate the density of the Earth at a radius of 3200 km, based on the interpolation curve!
2) At what radius will the density value be exactly 4000 kg/$m^3$?

To solve this, load the earth_density.txt file, then fit a cubic second-order and a cubic first-order spline to the points! Let's see which one fits better!

```
>  data = load('earth_density.txt')
>  r = data(:,1) % radius in km
>  ro = data(:,2) % demsity in kg/m^3
>  format shortG
>
>  % cubic first-order Hermite interpolation
>  fsp1 = @(x) interp1(r,ro,x,'pchip')
>  fsp1(3200) % 10361
>  figure(3);subplot(1,2,1);
>  plot(r,ro,'r*'); hold on;
>  fplot(fsp1,[0,max(r)],'b');
>  title('Cubic first-order Hermite interpolation ')
>
```

```
>  % cubic second-order spline interpolation
>  fsp2 = @(x) spline(r,ro,x)
>  fsp2(3200) %  11350
>  subplot(1,2,2)
>  plot(r,ro,'r*'); hold on;
>  fplot(fsp2,[0,max(r)],'b');
>  title('Cubic second-order spline interpolation')
```



The two methods gave large differences in the density values at 3200 km, in the first case the result was 10361 kg/m$^3$, and in the second 11350 kg/m$^3$.

Based on the figure, it is clear that now the cubic first-order spline ('pchip') gives a better fit than the cubic second-order spline ('spline'), in the latter we experience a large oscillation. The first method is now better, as there are strong breaks and jumps in the data. For smooth data, the second-order spline normally gives better results.

We can answer now for the second question also, using fitted cubic first-order spline. At what radius will the density value be exactly 4000 kg/m$^3$?

```
>  % At what radius will the density value be exactly 4000 kg/m^3?
>  f = @(x) fsp1(x)-4000
>  R4000 = fzero(f,5500) % 5958.9 km
```

So the value of the density will be 4000 kg/m$^3$ at a radius of about 6000 km.

## INTERPOLATION OF PARAMETRIC CURVES

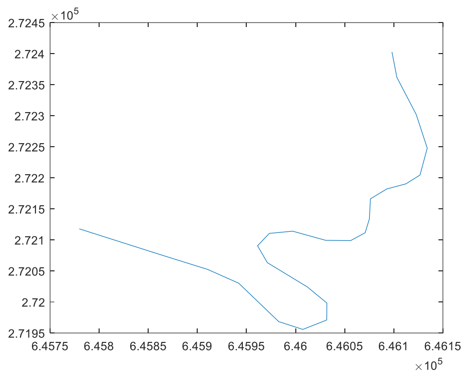
Most of the curves, that are used to model some data in practice, cannot be described using a single mathematical function, as there exist values of the independent variable (x, for example) where the curve has two distinct "function values". For example, when modelling a path or road using measured data points along the path, if it turns back on itself or loops, it cannot be modelled by a single function. In these cases, we use parametric curves consisting of independent functions for the different coordinates, where the parameter can be the number of points or the arc length along the curve.

As an example, let's see the following problem. With used a GPS to measure, the road axis of the nearly one kilometer section of a serpentine road leading to Visegrád Castle. The GPS measured and saved data (with a certain frequency) as we went along the path resulting in a data matrix containing the x and y coordinates of the points.

- Let's plot the results of the survey! Then fit a cubic, second order spline to the points and determine the coordinates of the point 500 m from the beginning of the path!

Let's load the data from the file path.txt, and plot the road axis!

```
> %% Parametric interpolation
> clc; close all; clear all; format
  longG;
> adat = load('path.txt')
> x = adat(:,1); y = adat(:,2);
> figure(1)
> plot(x,y)
```



In the figure, we can see that there are places where a given x coordinate has more than one y value. We would not be able to fit a curve to it with traditional function interpolation methods. However, the task can be solved in parametric form. We have to choose a parameter that has a single value for each point, this can be the number of the point or the distance from the starting point.

Now let this parameter be the arc length of the curve, which we will approximate with the distance between the measured points. As the path cannot be modelled by a single function, we have to use a different spline interpolation for the x and the y coordinates. A cubic spline applied to a section will look like this:

$$x(t) = a_1 + a_2 \cdot t + a_3 \cdot t^2 + a_4 \cdot t^3$$

$$y(t) = b_1 + b_2 \cdot t + b_3 \cdot t^2 + b_4 \cdot t^3$$

We first have to calculate the distances between the points ($\Delta$x,$\Delta$y), to approximate the arc length along the curve. To do this, we can use the built-in **diff** command that calculates the differences between points in a vector. If we have *n* number of points, we get back a vector containing *n-1* values. From this we can calculate distances between the points using the Pythagorean theorem (The first distance is 0, as the first point is to 0 distance from itself). If we cumulatively sum up the distances using the **cumsum** function, we get the distance of each point from the starting point.

```
> %  Parameter: the arc length of the curve, approximated by distances
> % x,y coordinate differences between the points:
> dx = diff(x)
> dy = diff(y)
> % distances:
> d0 = [0; sqrt(dx.^2+dy.^2)]
> % cumulatively sum up the distances:
> d = cumsum(d0)
```

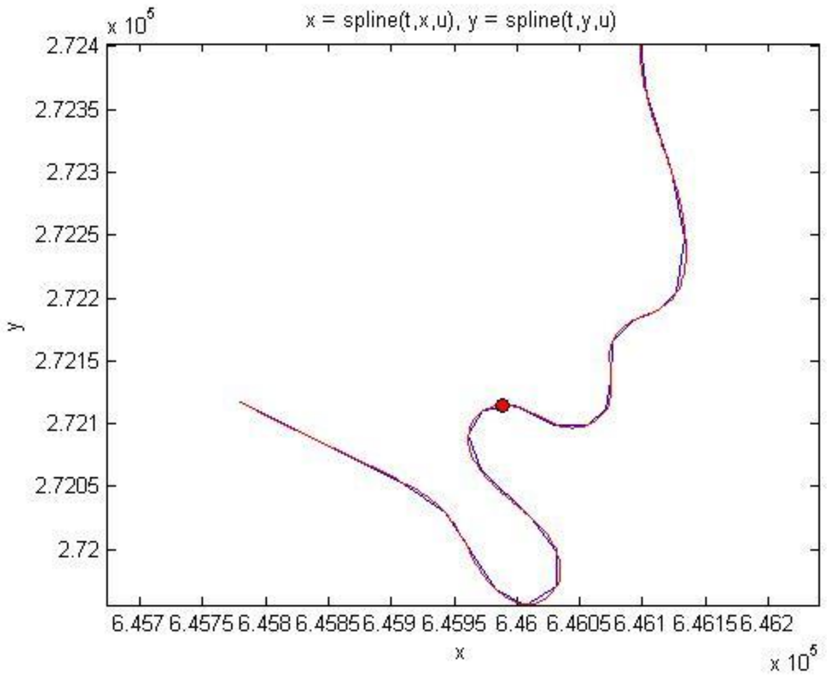
We can use the built in spline command now to define the parametric spline interpolations using the distances (d) as our independent variable.

```
> % parametric cubic splines for x and y
> xt=@(u) spline(d,x,u);
> yt=@(u) spline(d,y,u);
> % Plot the splines separately,
> % x and y coordinate depending on distance
> figure(2); fplot(xt,[0,1000]);
> figure(3); fplot(yt,[0,1000]);
> % plot the parametric curve: x as a function of y (parametrically)
> figure(1); hold on;
> hold on;
> fplot(xt,yt,[0,d(end)],'r');
```

What will be the coordinates of the point 500 m from the beginning of the road?

```
> % coordinates of the point 500 m from the beginning of the road
> x500 = xt(500) % 645989.003262212
> y500 = yt(500) % 272114.837059394
> format short;
> plot(x500,y500,'ko','MarkerFaceColor','r')
```

The coordinates of the asked points will be: 645989.00 m and 272114.84 m.



## NEW FUNCTIONS USED IN THE CHAPTER

| | | |
|---|---|---|
| vander | - | Vandermonde matrix |
| interp1 (method: linear, nearest, spline, pchip) | - | 1-D interpolation (method: linear, nearest neighbor, piecewise cubic spline interpolation, piecewise cubic Hermite interpolation) |
| spline | - | 1-D cubic spline interpolation |