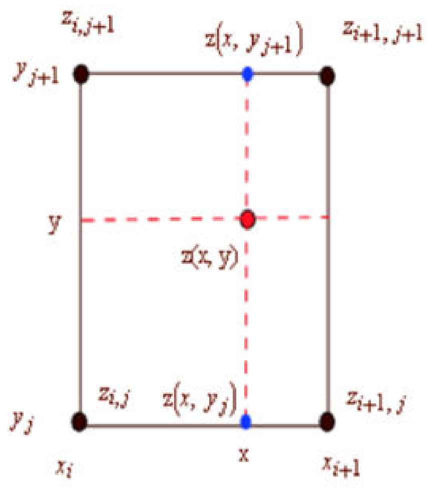
2D Interpolation and regression

In the previous practicals, we looked at regression models, as well as global and local one dimensional interpolation of data, such as using a polynomial, linear sections or splines. In many cases, our problem is two dimensional in nature, so we cannot use one dimensional regression or interpolation, but luckily, many previously studied methods can be extended into higher dimensions as well.

2D interplation of data given on a regular grid

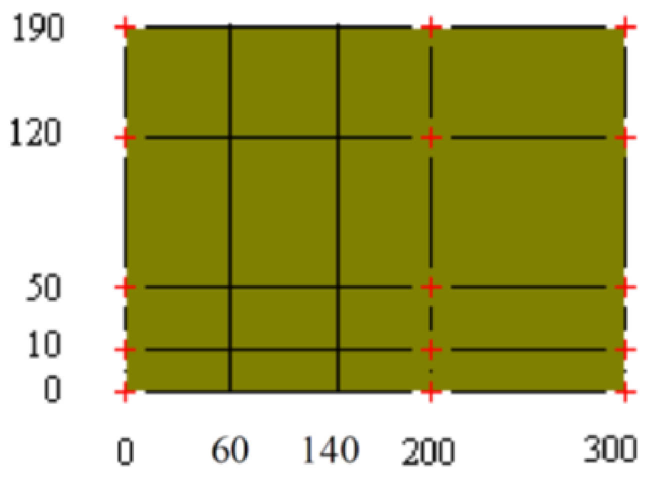
If we have data points that are located in a regular grid, the simplest form of interpolation we can use the so-called bilinear interpolation. The interpolation incorporates three computational steps:

1. We linearly interpolate along $y = y_j = \text{const.}$ and calculate $z(x, y_j)$.
2. We linearly interpolate along $y = y_{j+1} = \text{const.}$ and calculate $z(x, y_{j+1})$.
3. We linearly interpolate along x, between $z(x, y_j)$ and $z(x, y_{j+1})$ and find the value $z(x, y)$.



If we introduce more points to the interpolation process, we can use higher order polynomials as well (such as the bicubic interpolation).

As an example let's look at terrain heights measured in a grid. The layout of the measurements, with their x and y coordinates, is given in the following figure:



The matrix containing the measurements are located in the **`terrain.txt`** file. Let's load them:

```
clear all; close all;
x = [0, 60, 140, 200, 300];
y = [190, 120, 50, 10, 0];
Z = load('terrain.txt');
```

The built-in function calculating the interpolation works on matrices, so we first have to create a regular grid from the vectors of the x and y coordinates. We can use the built-in **`meshgrid`** command for this.

```
[X, Y] = meshgrid(x, y)
```

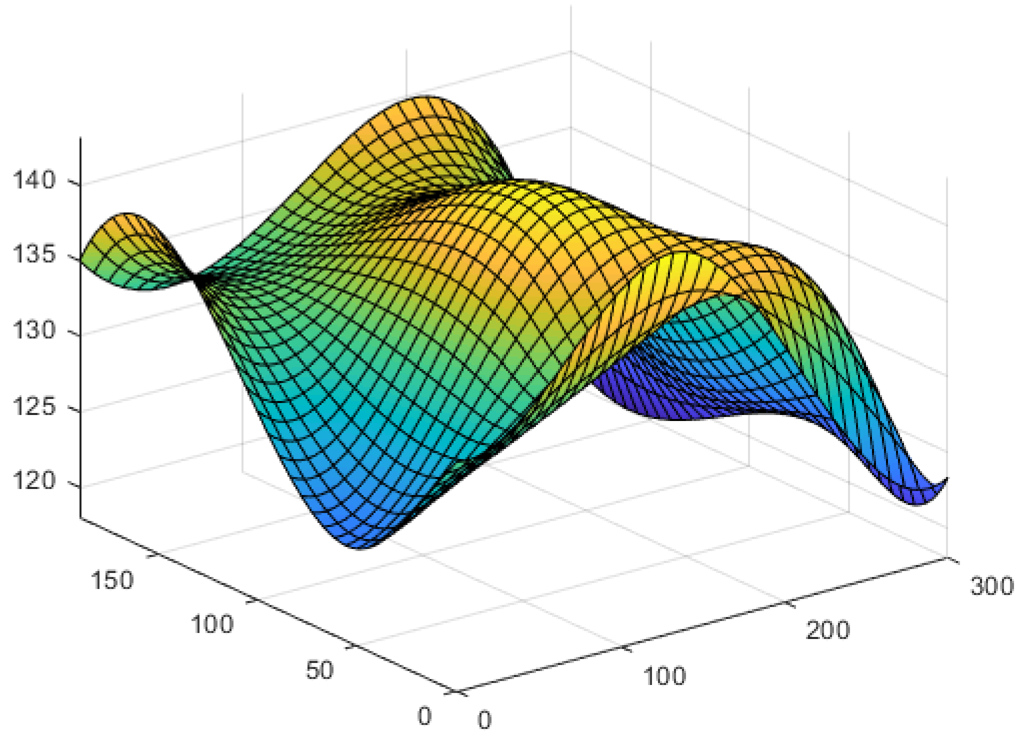
```
X = 5×5
     0    60   140   200   300
     0    60   140   200   300
     0    60   140   200   300
     0    60   140   200   300
     0    60   140   200   300
Y = 5×5
   190   190   190   190   190
   120   120   120   120   120
    50    50    50    50    50
    10    10    10    10    10
     0     0     0     0     0
```

Now, using the X, Y and Z matrices, we have 3 dimensional coordinates for each terrain height. For two dimensional interpolation, we can use the **`interp2`** command, similarly to the **`interp1`** from the previous practical. The **`interp2`** command can only work with data given in a regular grid (the grid points don't necessarily have to be equidistant). We specify different methods for the interpolation:

- `nearest` - nearest neighbor interpolation,
- `linear` - 2D linear interpolation (bilinear),
- `spline` - cubic spline,
- `cubic` - 2D cubic interpolation (bicubic).

```
F = @(u, v) interp2(X, Y, Z, u, v, 'spline');
figure(1);
```

```
fsurf(F, [min(x), max(x), min(y), max(y)]);
```



Using our interpolation, we can answer the following questions:

1. What is the height in (100, 100)?
2. What does an West-East section in this point look like?
3. How much cut and fill is needed if we want to level the surface at 135 meters?

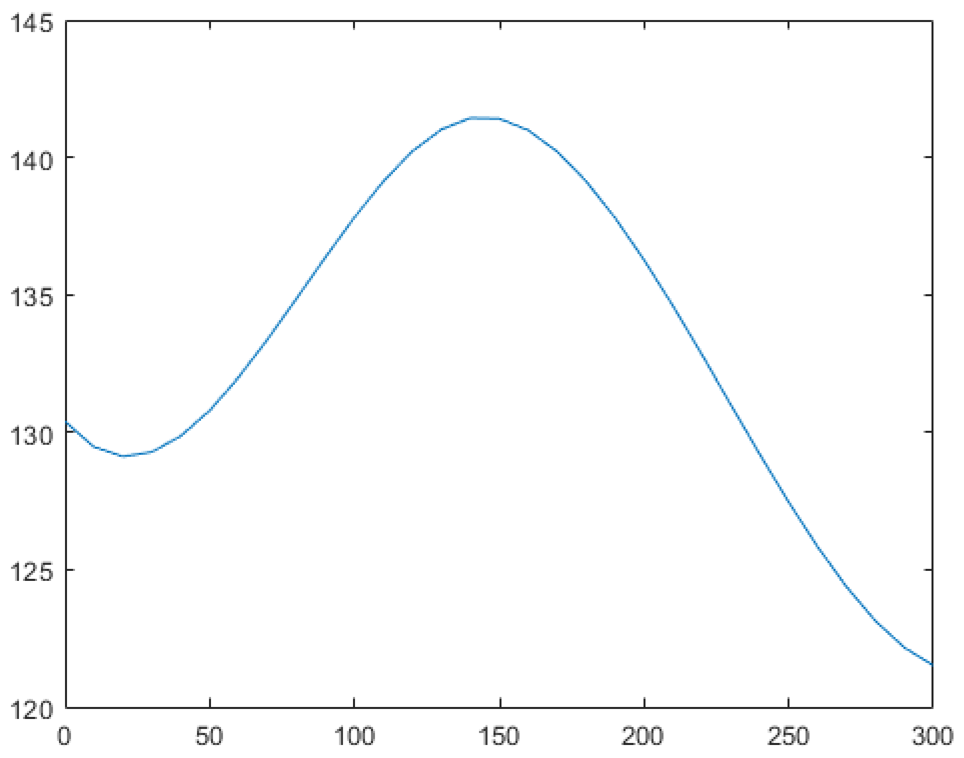
The answer to the first question can be found by a simple query using the interpolation:

```
F(100, 100)
```

```
ans =
        137.807868245719
```

The W-E section at (100, 100) can be drawn by creating two vectors corresponding to the coordinates in the section. The x coordinates go from 0 to 300 with some resolution (let's say, 10 meters) and the y values are a constant 100:

```
xs = 0:10:300; % x coordinates at every 10 meters from 0 to 300
ys = 100*ones(1, length(xs)); % y coordinates are always 100
zs = F(xs, ys); % terrain height from the interpolation
figure(2);
plot(xs, zs);
```



To answer the third question, the amount of cut and fill can be calculated using 2D integration (covered in subsequent practicals). If we calculate the volume between the terrain and a 0 reference level and then subtract it from the volume under the horizontal surface at 135 m, we get the total amount of cut and fill needed. The volume under the terrain can be calculated using the following integral:

$$V_T = \int_0^{190} \int_0^{300} F(x, y)\, \mathrm{dx}\, \mathrm{dy}$$

```
VT = integral2(F, 0, 300, 0, 190)
```

```
VT =
          7614016.55348964
```

The volume under the level surface at 135 m:

```
VS = 135 * 300 * 190
```

```
VS =
     7695000
```

The sum of the cut and the fill volumes:

```
V = VS - VT
```

```
V =
          80983.4465103578
```

2D regression of data given on an irregular grid

When our data is not located on a regular grid, interpolating it becomes somewhat harder. In case of polynomial regression however, data location can be arbitrary, although, as the degree of the polynomial gets higher, we see similar oscillation as in the one dimensional case (Runge's phenomenon). Moreover, in the two dimensional case, the number of coefficients grows very rapidly as the degree goes up, which means that higher degree polynomials require a lot more data points.

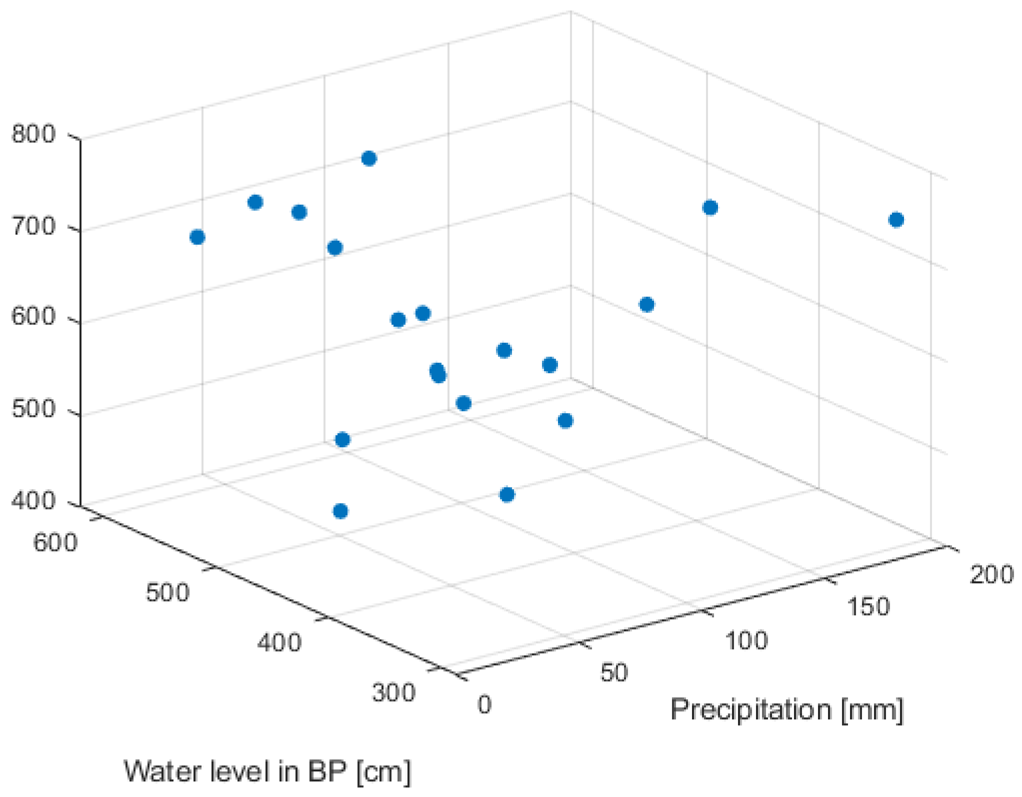
Let's look at the following example. The catchment area (the source of water) of the section of the Danube in Hungary is located in Austria and Bavaria. Depending on the precipitation in these areas, and the water level of the Danube in Budapest at the same time, we can forecast the height of the flood wave at Budapest caused by the precipitation.

Our data matrix located in **`flood.txt`** contains 3 columns:

1. The precipitation in the catchment area that caused flood waves on the Danube, in mm.
2. The water level of the Danube at Budapest during the precipitation, in cm.
3. The height of the flood wave at Budapest caused by the precipitation in cm.

Let's load the data and visualize it:

```
clear all; close all;
data = load('flood.txt');
x = data(:, 1);  % precipitation in the catchment area [mm]
y = data(:, 2); % water level in BP during the precipitation [cm]
z = data(:, 3); % forecast flood wave height in BP [cm]
figure(1);
scatter3(x, y, z, 'filled'); % 3D scatter plot
xlabel('Precipitation [mm]');
ylabel('Water level in BP [cm]')
```



The simplest regression polynomial we can use is a regression plane:

$$z(x, y) = c_1 + c_2 x + c_3 y$$

The generic form of a more complicated, 2D quadratic regression polynomial:

$$z(x, y) = c_1 + c_2 x + c_3 y + c_4 x^2 + c_5 xy + c_6 y^2$$

Our task is to approximate the height of the flood wave in Budapest, if the precipitation in the catchment area was 100 mm and the water level in Budapest at that time was 400 cm. In order to do this, we can use a 2D quadratic polynomial to model the data and query the value of the model at the given values.

The regression polynomial can be written in matrix form (where ***n*** is the number of data points):

$$A = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1y_1 & y_1^2 \\ 1 & x_2 & y_2 & x_2^2 & x_2y_2 & y_2^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & y_n & x_n^2 & x_ny_n & y_n^2 \end{bmatrix} \qquad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_6 \end{bmatrix} \qquad b = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_6 \end{bmatrix}$$

$$\boldsymbol{A} \cdot \boldsymbol{c} = \boldsymbol{b}$$

In MATLAB:

```
n = length(x); % number of data points
A = [ones(n, 1), x, y, x.^2, x.*y, y.^2]
```

```
A = 19×6
          1          58         405        3364       23490      164025
          1          52         450        2704       23400      202500
          1         133         350       17689       46550      122500
          1         179         285       32041       51015       81225
          1          98         330        9604       32340      108900
          1          72         400        5184       28800      160000
          1          72         550        5184       39600      302500
          1          43         480        1849       20640      230400
          1          62         450        3844       27900      202500
          1          67         610        4489       40870      372100
     ⋮
```

```
c = A\z
```

```
c = 6×1
         -1339.90488568953
          17.1883973493066
          4.95723840977286
       -0.0372951053113031
       -0.0179199642155538
      -0.00333877203214026
```

We can also solve the fitting of the regression model using the built-in **`regress`** function. The function can give us back the 95% confidence intervals the residual values at the points as well:

```
[c2, int95, res] = regress(z, A)
```

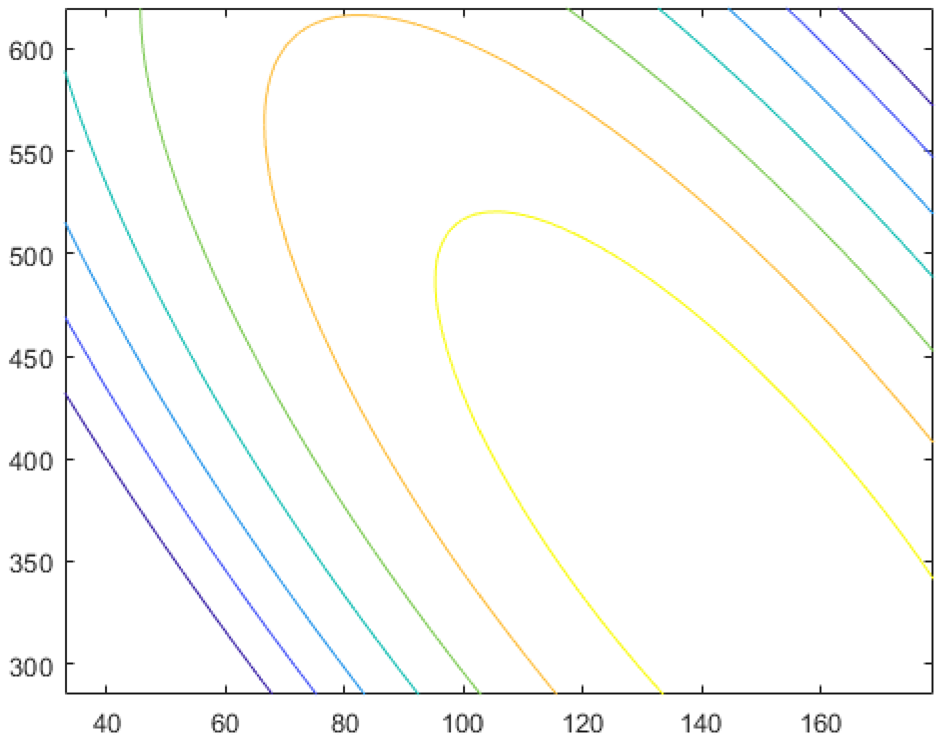
```
c2 = 6×1
         -1339.90488568954
          17.1883973493066
          4.95723840977287
       -0.0372951053113031
       -0.0179199642155538
      -0.00333877203214028
int95 = 6×2
         -3269.18333399781          589.373562618741
          2.34025664610191          32.0365380525113
         -1.31208695454193          11.2265637740877
       -0.0702443913017601       -0.0043458193208462
       -0.0397719281370144       0.00393199970590674
      -0.00860910072975285        0.0019315566654723
res = 19×1
          19.3390597341336
          71.6254030419311
          1.70162083411185
         0.720015424926714
          30.8593786732861
          3.08123311450356
          -41.193900243456
         -50.5908496868752
          22.8976885737432
         -3.46942828393469
     ⋮
```

We can define the regression surface using the coefficients:

```
f = @(x, y) c(1) + c(2)*x + c(3)*y + c(4)*x.^2 + c(5)*x.*y + c(6)*y.^2;
```

Visualizing the model using a contour plot first:

```
figure(2);
cont = fcontour(f, [min(x), max(x), min(y), max(y)]);
cont.LevelList = 450:50:800; % contour lines between 450 and 800
```



3D visualization:

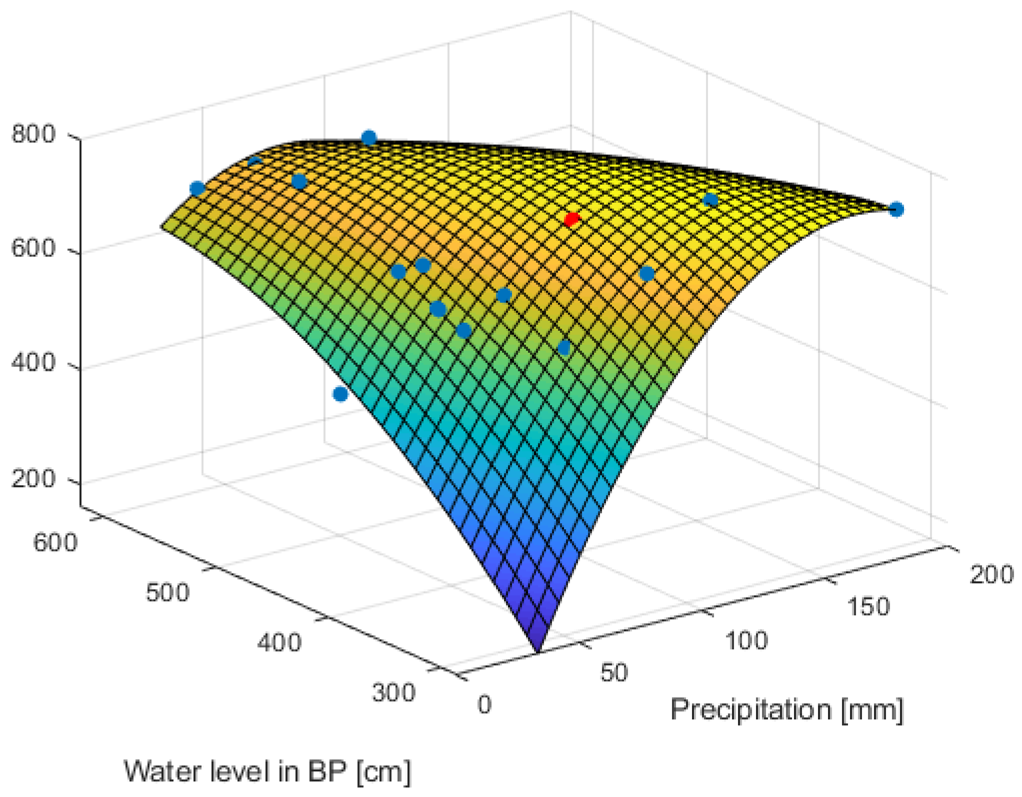
```
figure(1); hold on;
fsurf(f, [min(x), max(x), min(y), max(y)])
```

Forecasting the height of the flood wave at 100 mm of precipitation and 400 cm of water level:

```
xfw = 100;
yfw = 400;
flood_wave = f(xfw, yfw)
```

```
flood_wave =
         737.877066272642
```

```
plot3(xfw, yfw, flood_wave, 'ro', 'MarkerFaceColor', 'r');
```



2D interpolation of data on an irregular grid

When interpolating data on an irregular grid, we have multiple methods to choose from:

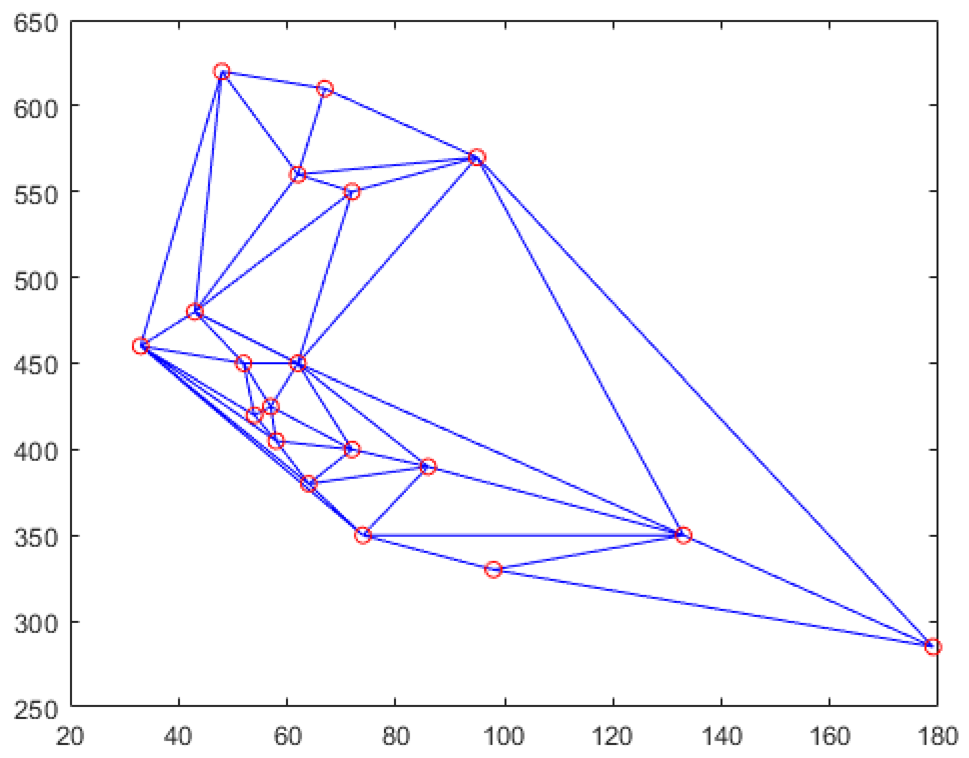
- the nearest neighbor method, when the interpolated point gets the value of the closest data point,
- kriging, when the distance from data point gets taken into account with some weighting,
- radial basis functions, when multiple basis functions are fitted to the data points,
- division of the irregular grid with simpler elements.

In MATLAB, the built-in **`griddata`** function uses the last method, the so-called Delaunay triangulation. The method creates triangles between the data points in such a way that if we drew the circumcircle for each triangle, no data points would be inside the circles. When choosing the `linear` option for the **`griddata`** command, the surface of the triangles are used for the interpolation. The **`griddata`** command can also use other types of interpolations, such as the `nearest` (nearest neighbor), or the `cubic` (bicubic spline).

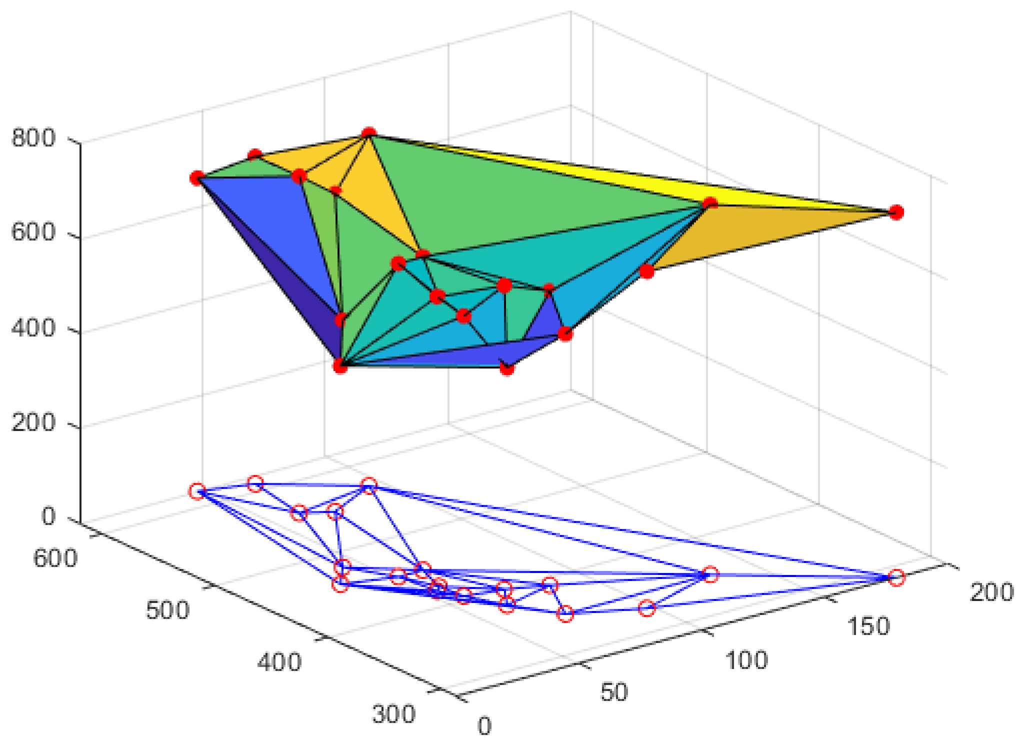
The following two figures show the Delaunay triangulation of the data points from the previous example.

```
tri = delaunay(x, y); % creates a Delaunay triangulation from the data points

figure();
triplot(tri, x, y); hold on; % plot of the triangles
plot(x, y, 'ro');
```



```
figure();
trisurf(tri, x, y, z); % 3D plot of the triangles
hold on;
scatter3(x, y, z, 'ro', 'filled')
plot(x, y, 'ro');
triplot(tri, x, y);
```



How much is the predicted height of the flood wave if the precipitation was 100 mm and the water level was 400 cm, using linear and cubic interpolation?

In order to answer the question, we first have to define the interpolating functions:

```
zlin = @(u, v) griddata(x, y, z, u, v, 'linear');
zcub = @(u, v) griddata(x, y, z, u, v, 'cubic');
```

Using the linear interpolation, the answer is:

```
flood_lin = zlin(100, 400)
```

```
flood_lin =
         724.737732656514
```

From the cubic interpolation, we get:

```
flood_cubic = zcub(100, 400)
```

```
flood_cubic =
         735.393860996604
```

Using the regression polynomial, the answer is ~738 cm.

In order to better see the difference between the two types of interpolations, we can plot the corresponding surfaces. First, we create 1000 points between the smallest and largest x and y values and then we use these vectors to create a mesh grid.

```
xv = linspace(min(x), max(x), 1000);
yv = linspace(min(y), max(y), 1000);
[xq, yq] = meshgrid(xv, yv);
```

We can query the value of the interpolation at each of these points. By default, the **`griddata`** function does not extrapolate, so if our query point is out of the convex perimeter of the dataset, we get a NaN (not a number) value.

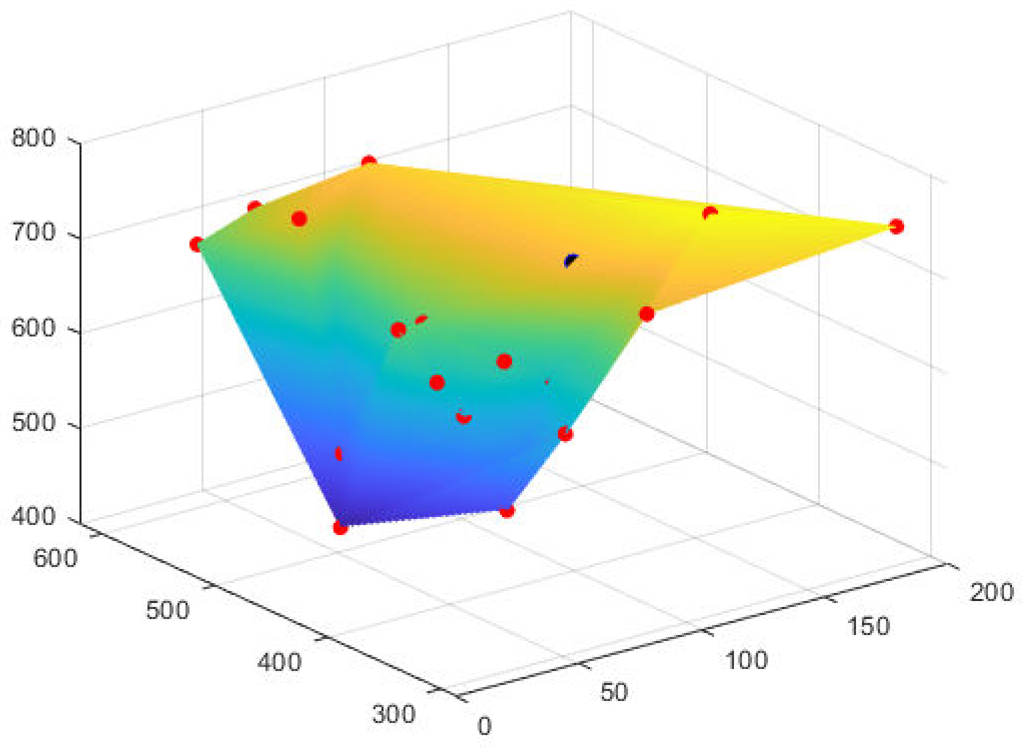
```
zql = zlin(xq, yq)
```

```
zql = 1000×1000
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN ···
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
       ⋮
```

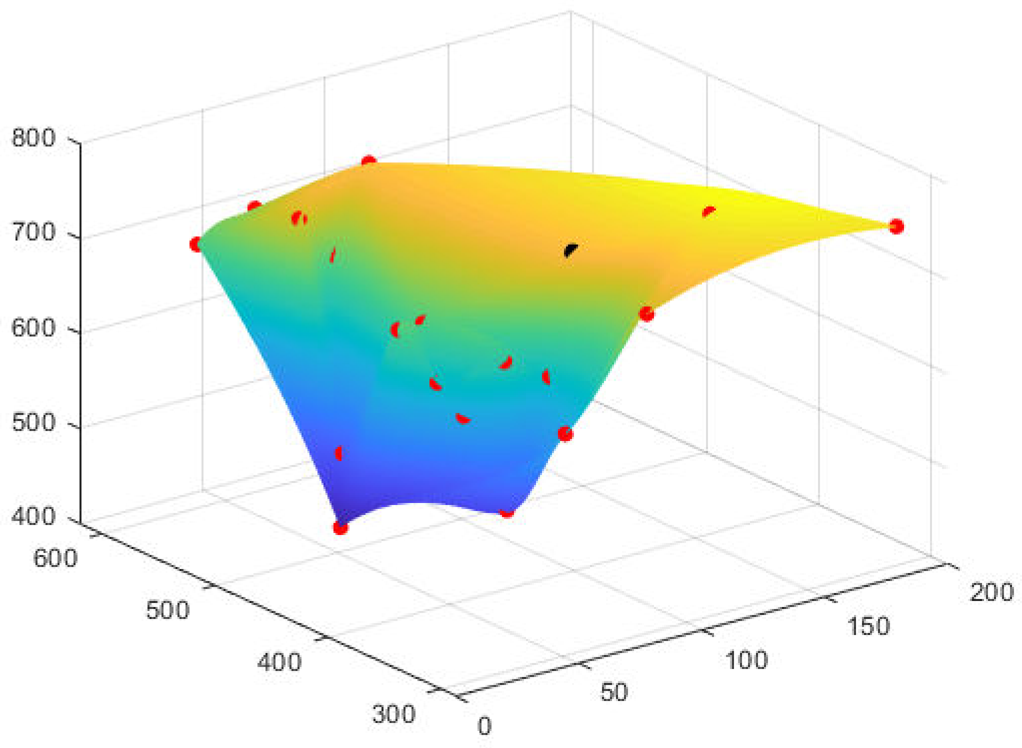
```
zqc = zcub(xq, yq);
```

Now, using the new variables, we can plot the interpolating surfaces. In case of the linear interpolation, we get the same surface as in the Delaunay triangulation plot:

```
% Linear interpolation
figure();
scatter3(x, y, z, 'ro', 'filled');
hold on;
mesh(xq, yq, zql);
plot3(100, 400, flood_lin, 'bo', 'MarkerFaceColor', 'k');
```



```
% Cubic interpolation
figure();
scatter3(x, y, z, 'ro', 'filled');
hold on;
mesh(xq, yq, zqc);
plot3(100, 400, flood_cubic, 'ko', 'MarkerFaceColor', 'k');
```


800
700
600
500
400
600
500
400
300
0
50
100
150
200