Marine Data Literacy Course - Practical 4

From data formats to practical use of water column depth data

Aleksandra Cupiał, Wojciech Brodziński and Gabriela Gic-Grusza (University of Gdansk)

Friday 3rd December 2021, 4:00pm-7:00pm

Contents:

1 Introduction

This instruction sheet is meant to describe and list the structured activities, instructions and expected outputs for the practical exercise. It also serves to support the students to follow and execute the various steps of the exercise.

It is suggested that these instructions are kept open and available during the practical session such that instructions and pieces of code can be used directly without rewriting.

The students will be asked to answer four MC questions which are based on outputs from the practical session. The tutors delivering the practical will guide you to prepare your answers at the appropriate stages of your practical, so that you will be able to submit your answers at the end of the session.

1.1 Targets of the practical

Bathymetry is the study of the depth of oceans or lakes and is the underwater equivalent to hypsometry or topography. Bathymetric (or hydrographic) charts are typically produced to support safety of surface or sub-surface navigation. Usually these charts show seafloor relief or terrain as contour lines (called depth contours or isobaths) as well as selected depths (soundings) and typically also provide surface navigational information. Bathymetric information is also crucial for modelling (e.g. wave modelling), especially in the coastal zone and construction of offshore infrastructure like wind farms.

The Leibniz Institute for Baltic Sea Research Warnemünde (IOW) is a non-university marine research institute that provides bathymetry data for Baltic Sea area. For more information see https://www.io-warnemuende.de/topography-of-the-baltic-sea.html (https://www.io-warnemuende.de/topography-of-the-baltic-sea.html) The goal of this practical session is to:

- learn how to manipulate different data formats, typical for bathymetric data (dat, txt, csv)
- understand how the selection of subsets of data influences the final analysis
- learn the basic aspects of spatial data interpolation and discover how different methods of interpolation impact on data accuracy of the gridded output

In this practical the students will learn how to:

- Download bathymetry data in different formats
- Present bathymetric data as a map with and without land mask
- Manipulate input array and present simple bathymetric profile
- Open and save data text file
- Use the most common interpolation methods (Nearest neighbor, Linear Interpolation etc.) on the bathymetric profile data
- Assess interpolation errors

Before the practical, the students are expected to:

1. Download and install Anaconda Individual Edition from https://www.anaconda.com/products/individual; to run Python code on your local machine using Jupyter Notebook OR use a Google Account to log into Collaboratory by visiting https://colab.research.google.com (https://colab.research.google.com) to run the code on as remote server.
2. Download a copy of the files that were specifically used for this presentation from the course website.

IMPORTANT: These installations need to be done ahead of the practical session so that their functionality can be tested BEFORE the session.

1.2 Datasets used

Description provided by the IOW website.

The data is organized in two sets and includes a digitized topography of the Baltic Sea. Land heights and water depths have been calculated for two regular spherical grids from available data.

Data set "iowtopo2" covers the whole Baltic Sea from 9° to 31° East and from 53°30’ to 66° North by (660 x 750) grid cells. The resolution is 2 minutes with respect to longitude, and 1 minute to latitude. This is approximately 1 nautical mile, or 2 km, respectively. The region of the Belt Sea from 9° to 15°10’ East and from 53°30’ to 56°30’ North is comprised within data set "iowtopo1" with a twofold higher resolution (1 minute in longitude and 0.5 minutes in latitude corresponding to approx. 1 km).

The data specify a representative average of the water depth or the land height of each grid cell, counted by negative and positive values in meters. Some statistical parameters allow a rough estimate of the reliability of the data. Since a common average of land heights and water depths lead to rather unsatisfying results with respect to the gridded shoreline, a landmask is proposed in both data sets.

Data are provided in two formats. NetCDF files (.nc) are self-describing binaries which may be visualized and processed by tools like Ferret, Grads or Matlab. Alternatively rather big ascii files (.dat) are given which start with two header lines and contain the following data:

- **xlon**: the geographic longitude of the grid cell centre
- **ylat**: the corresponding geographical latitude
- **z_topo**: land height/water depth, composite of z_water, z_land and the proposed landmask
- **z_water**: average of all water depths allocated from original data to this grid cell
- **z_land**: average of land heights allocated to this grid cell from edcdaac.usgs.gov/gtopo30/gtopo30.html data
- **z_min**: minimal value of the original data
- **z_max**: maximal value of the original data
- **z_stdev**: standard deviation of original data from averages z_water/z_land
- **z_near**: datum lying nearest to the centre of this grid cell
- **d-near**: distance of above mentioned data point from centre of grid cell
- **n_wet**:
  - $>$ 0: number of original water depths allovated to this grid cell
  - $<$ 0: number of neighbors interpolated to fill this empty cell
- **n_dry**:
  - $>$ 0: number of original land heights allocated to this cell
  - $<$ 0: number of iterations to find direct neighbours for interpolation
- **landmask**: proposed "naturally loking" landmask (land=0, water=1)
- **flag**: flag indicating a pure data average (0), or an interpolated/masked land height (+1) or water depth (-1)

2 Initial data manipulation

2.1 Initializing the Python Environment

Importing the required libraries in Python.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.mlab as ml
from scipy import interpolate
from scipy.interpolate import interp1d
```

2.2 Reading the model .dat file

In [2]:
```
df = pd.read_table('iowtopo1_rev02.dat', sep="\s+") #'iowtopo2_rev03.dat'
```

In [3]:
```
df.head()
```

Out[3]:

	x_lon	y_lat	z_topo	z_water(m)	z_land(m)	z_min(m)	z_max(m)	z_stdev(m)	z_near(m
0	9.00833	53.50417	20.5	0.0	20.5	20.0	21.0	0.7	21.
1	9.02500	53.50417	20.0	0.0	20.0	20.0	20.0	0.0	20.
2	9.04167	53.50417	22.0	0.0	22.0	21.0	23.0	1.4	21.
3	9.05833	53.50417	11.5	0.0	11.5	10.0	13.0	2.1	10.
4	9.07500	53.50417	6.0	0.0	6.0	6.0	6.0	0.0	6.

2.3 Basic dataset statistics

Calculating basic statictics based on the given dataset

In [4]:
```
df[["z_topo","z_water(m)","z_land(m)"]].describe()
```

Out[4]:

	z_topo	z_water(m)	z_land(m)
count	133200.000000	133200.000000	133200.000000
mean	12.667331	-9.296549	21.923123
std	40.911709	14.456380	32.521916
min	-93.800000	-93.800000	0.000000
25%	-15.900000	-15.800000	0.000000
50%	5.000000	0.000000	5.000000
75%	35.000000	0.000000	35.000000
max	220.500000	0.000000	220.500000

2.4 Lat Long data visualization

Defining data bounding box

In [5]:
```
BBox = (df.x_lon.min(), df.x_lon.max(), df.y_lat.min(), df.y_lat.max())
print(BBox)
```

```
(9.00833, 15.15833, 53.504169999999995, 56.495830000000005)
```

Reading open street map within the bounding box

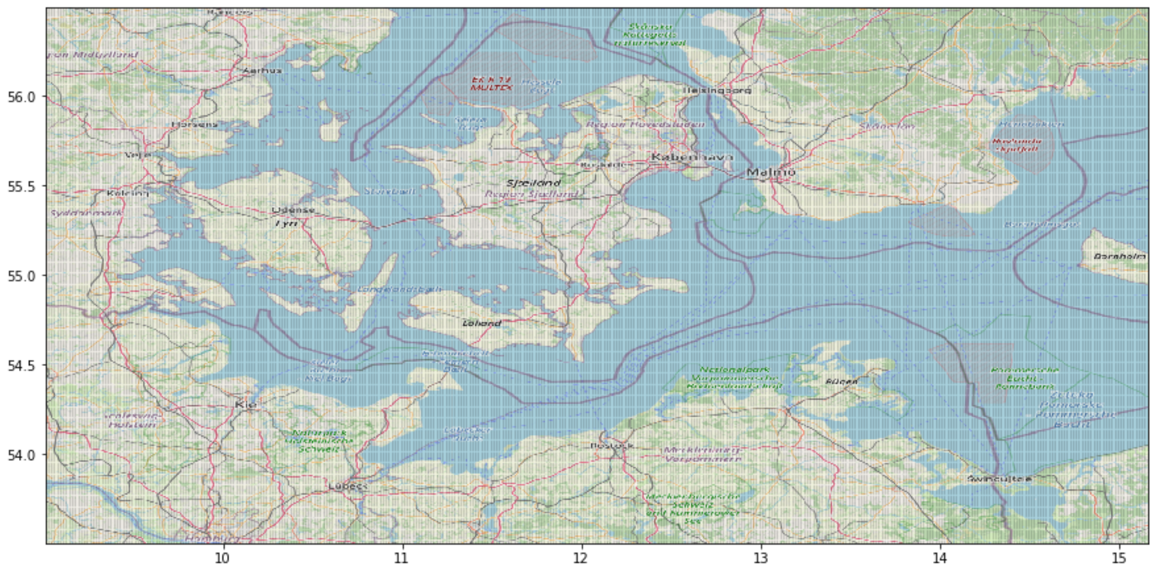
1. Go to https://www.openstreetmap.org/#map=5/51.500/-0.100 (https://www.openstreetmap.org/#map=5/51.500/-0.100)
2. Select "Export" button in the top left Menu
3. Isert BBox points
4. Select "Share" in the right menu
5. Save .png file in in the working folder

In [8]:
```
map1 = plt.imread('map1.png')
```

Plotting data grid using openstreet map as a basemap (we are showing here only the location of grid points without the depth value in these points)

In [16]:
```
fig, ax = plt.subplots(figsize = (15,14.5))
ax.scatter(df.x_lon, df.y_lat, zorder=1, alpha= 0.2, c='k', s=0.1)
ax.set_title(' ')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
ax.imshow(map1, zorder=0, extent = BBox)
```

Out[16]: `<matplotlib.image.AxesImage at 0xbb02308>`



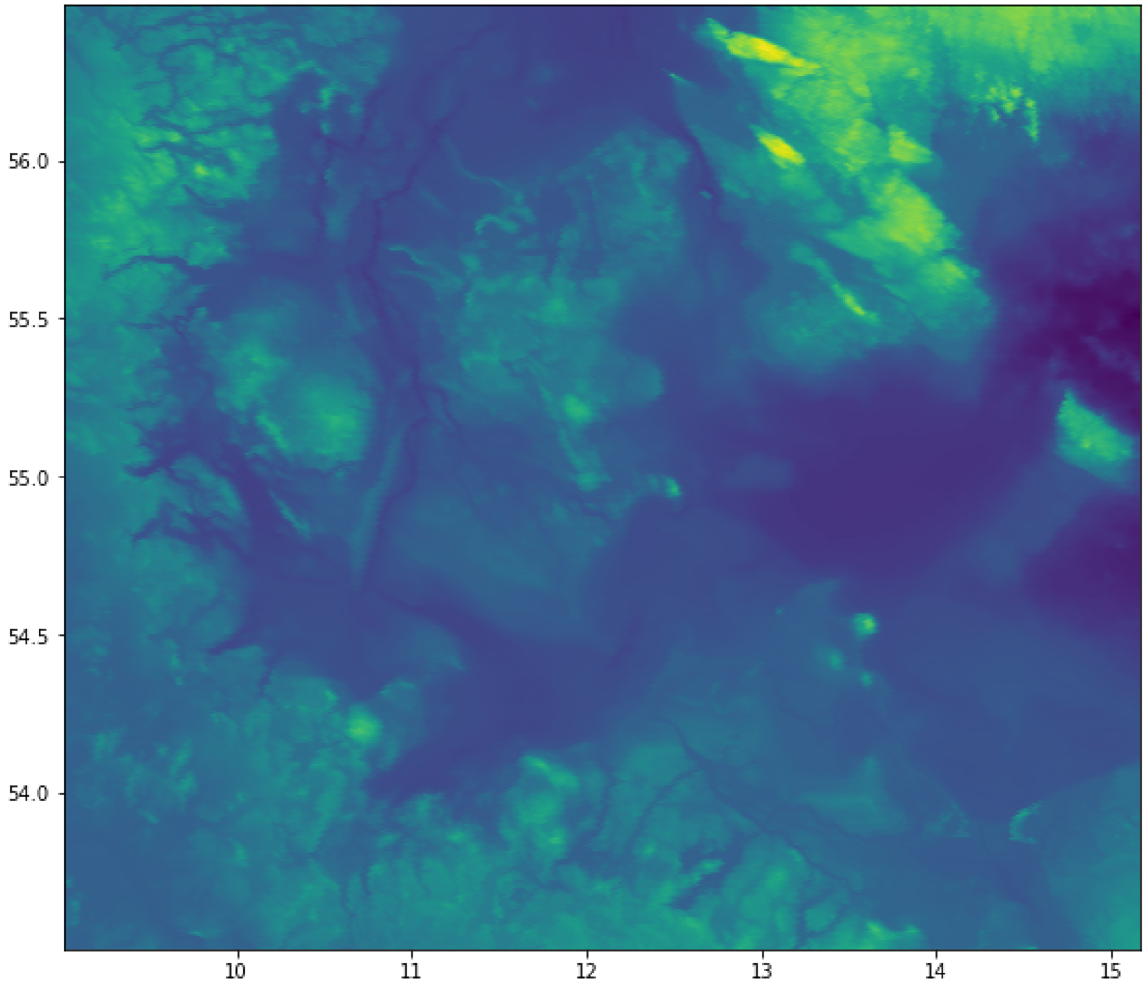
Plotting actual latitude, longitude and depth data values.

In [17]:
```
x1 = df.x_lon
y1 = df.y_lat
z1 = df.z_topo
print(z1)
```

```
0          20.5
1          20.0
2          22.0
3          11.5
4           6.0
          ...
133195    142.0
133196    125.0
133197    124.0
133198    131.0
133199    138.5
Name: z_topo, Length: 133200, dtype: float64
```

In [18]:
```
fig, ax = plt.subplots(figsize = (10,9))
ax.set_title(' ')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
plt.scatter(x=x1,y=y1,c=z1)
```

Out[18]: <matplotlib.collections.PathCollection at 0xbfe7dc8>



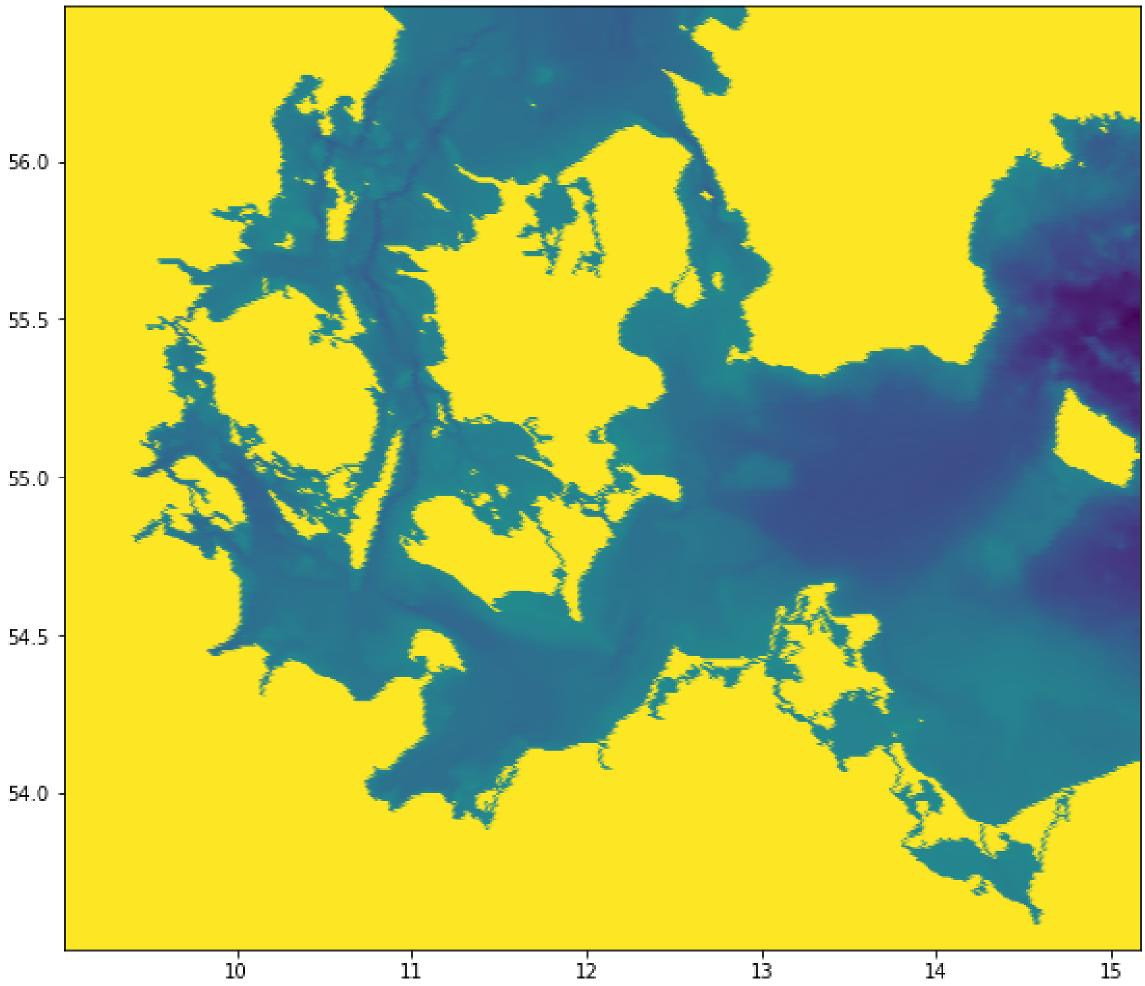
Change all positive values (terrain, not wet) to one value, to create mask (land)

In [19]:
```
z2=z1
z2[z2 > 0] = 100
print(z2)
```

```
0         100.0
1         100.0
2         100.0
3         100.0
4         100.0
          ...
133195    100.0
133196    100.0
133197    100.0
133198    100.0
133199    100.0
Name: z_topo, Length: 133200, dtype: float64
```

In [20]:
```
fig, ax = plt.subplots(figsize = (10,9))
ax.set_title(' ')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
plt.scatter(x=x1,y=y1,c=z2)
```

Out[20]: <matplotlib.collections.PathCollection at 0xc057048>



2.5 Excercise 1

Perform the same calculations using iowtopo2_rev03.dat input file.

To see analyzed area on openstreet map go to https://www.openstreetmap.org/ (https://www.openstreetmap.org/) and enter BBox values.

Quiz Question 1:
What is the mean water depth in the area captured in iowtopo2_rev03.dat?

1. 48.880079
2. -17.450946
3. -219.316708
4. 145.776708

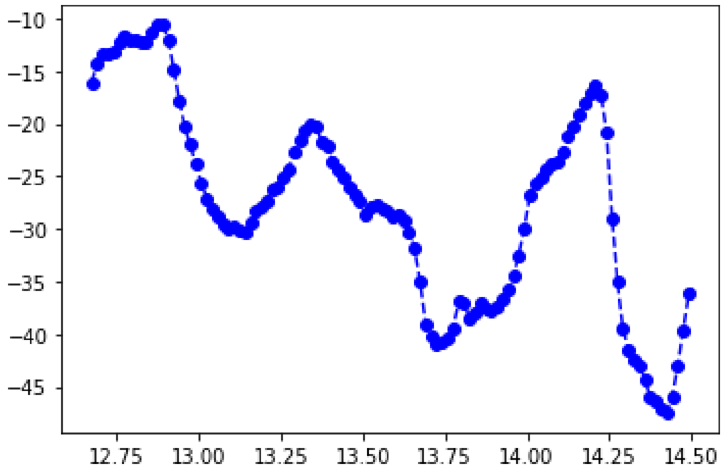
3 Preparing bathymetric profiles

3.1 Simple array manipulation

Let's select some data points from the source array. We will take wet point along y_lat equal to 55.29583° N and x_lat in range from 12.67500° E to 14.49167° E, which includes only wet points (according to the map above). To do that we need to select appropriate rows from the data array.

In [21]:
```
df = pd.read_table('iowtopo1_rev02.dat', sep="\s+") #'iowtopo2_rev03.dat'
xx=(df.x_lon[79770:79880])
zz=(df.z_topo[79770:79880])
plt.plot(xx, zz,  '--bo')
```

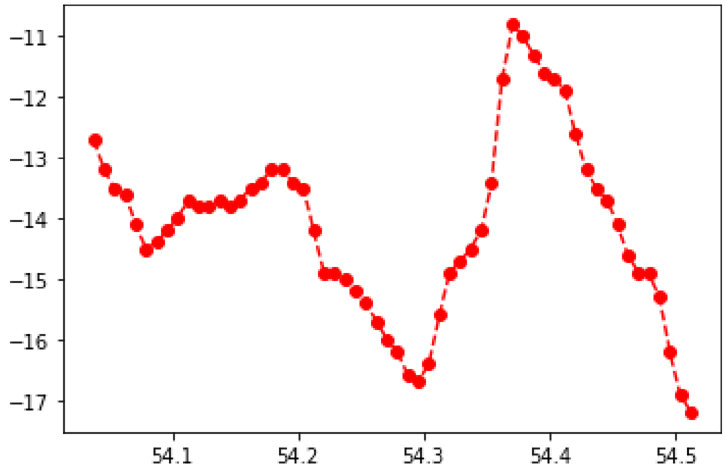
Out[21]:
```
[<matplotlib.lines.Line2D at 0xc0b93c8>]
```



We can also select data along selected x_lat. Note that input grid y_lat value changes every 370 rows.

```
In [22]: yy=(df.y_lat[23990:45450:370])
         zz=(df.z_topo[23990:45450:370])
         plt.plot(yy, zz,  '--ro')
```

```
Out[22]: [<matplotlib.lines.Line2D at 0xcaf6f08>]
```



3.2 Save data file

Save data stored as 1-D arrays in text file including 2 columns.

```
In [23]: np.savetxt('batyprof.csv', [p for p in zip(yy, zz)], delimiter=',', fmt='%s')
```

3.3 Excercise 2

Open txt file (example_profile.txt) with example depth profile data. Use similar function as we've used to open .dat file:

```
df = pd.read_table('iowtopo1_rev02.dat', sep="\s+")
```

Then create new 1-D arrays containing x (distance along the profile) and d (depth) using similar function as we had here:

```
 x1 = df.x_lon
y1 = df.y_lat
```

Now plot the profile.

Quiz Question 2:
What kind of cross-shore depth profile is stored in example_profile.txt?
1. multi-bar
2. gentle slope
3. steep slope
4. one bar

4 Basic concepts of interpolation

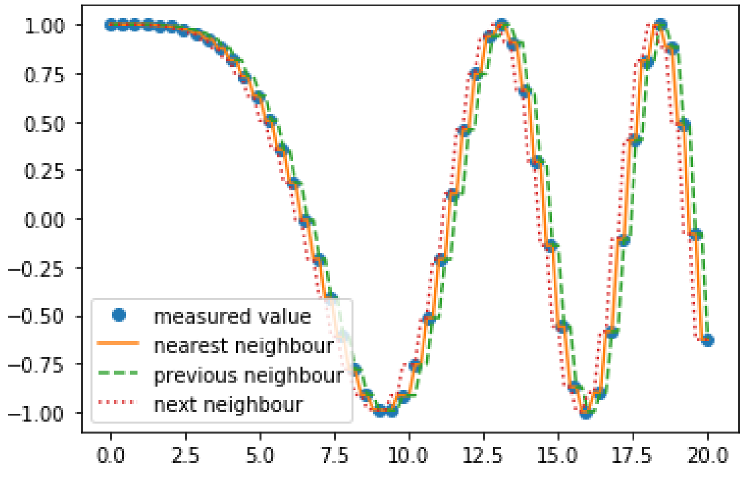
Interpolation is the problem of approximating the value of a function for a non-given point in some space when given the value of that function in points around (neighboring) that point. There are several interpolation techniques, some of them very sophisticated and each has its advantages and disadvantages. Here, we will learn about 3 most used techniques in bathymetric data: the nearest neighbor, linear interpolation, cubic spline interpolation and see an example of spatial interpolation that uses Delaunay triangulation.

4.1 1-D interpolation

The following examples will be presented for the one-dimesional data.

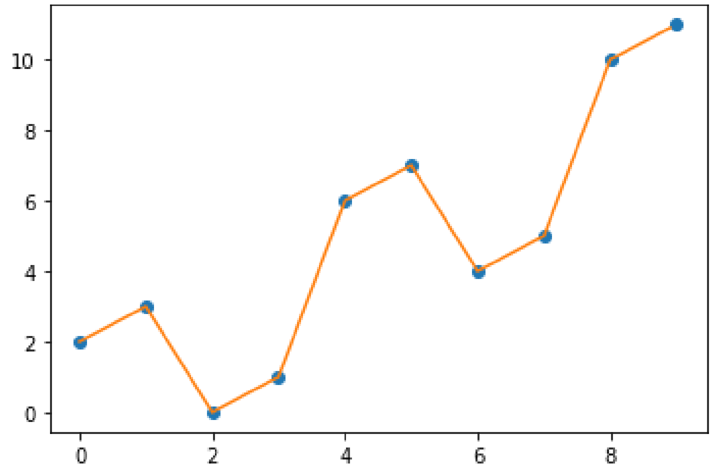
The **nearest neighbor** algorithm selects the value of the nearest point and does not consider the values of neighboring points at all, yielding a piecewise-constant interpolant. The algorithm is very simple to implement and is commonly used (usually along with mipmapping) in real-time 3D rendering to select color values for a textured surface.

In [24]:
```
x = np.linspace(0, 20)
y = np.cos(-x**2/27.0)
f1 = interp1d(x, y, kind='nearest')
f2 = interp1d(x, y, kind='previous')
f3 = interp1d(x, y, kind='next')
xnew = np.linspace(0, 20, num=101)
import matplotlib.pyplot as plt
plt.plot(x, y, 'o')
plt.plot(xnew, f1(xnew), '-', xnew, f2(xnew), '--', xnew, f3(xnew), ':')
plt.legend(['measured value', 'nearest neighbour', 'previous neighbour', 'nex
t neighbour'], loc='best')
plt.show()
```



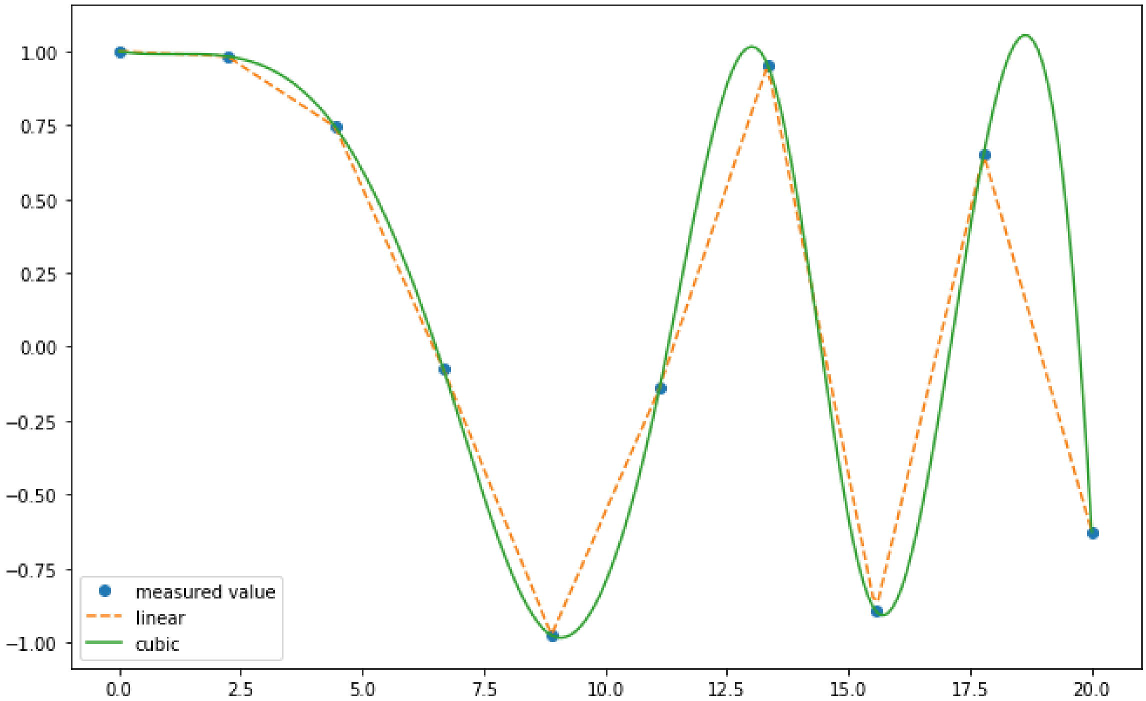
Linear interpolation is a method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data points.

In [25]:
```
X = np.arange(0,10) #X, Y - Example measured data
Y = X^2
#f = interpolate.interp1d(X, Y)
f = interp1d(X, Y,kind='linear')
Xnew = np.arange(0, 9, 0.1)
Ynew = f(Xnew)    # use interpolation function returned by `interp1d`
plt.plot(X, Y, 'o', Xnew, Ynew, '-')
plt.show()
```



In **cubic spline** interpolation method the interpolating function takes the form of a piecewise polynomial of 3rd order. Specifically, we assume that each pair of neighbouring data points (x_i, y_i) and (x_{i+1}, y_{i+1}) is joined by a cubic polynomial, which means that for n points the interpolant is built from $n-1$ cubic functions. Additionally, we want each polynomial to join with its neighbours as smoothly as possible, therefore we constrain the interpolating function to have continuous first and second derivatives at the data points.
In an example below we will compare the results of linear and cubic spline interpolation for a given data set.

In [26]:
```
x = np.linspace(0, 20, num=10)
y = np.cos(-x**2/27.0)
f1 = interp1d(x, y, kind='linear')
f2 = interp1d(x, y, kind='cubic')
xnew = np.linspace(0, 20, num=201)
plt.figure(figsize = (11,7))
plt.plot(x, y, 'o')
plt.plot(xnew, f1(xnew), '--', xnew, f2(xnew), '-')
plt.legend(['measured value', 'linear', 'cubic'])
plt.show()
```



4.2 Excercise 3

Quiz question 3:
Assume you have obtained measurements data set (x,y) described by the following expressions in Python: x = np.linspace(0, 6, num=7), y = np.exp(x/2)*np.cos(2*x). Calculate the interpolated value y_{int} for x_{int}=5.30 using the nearest neighbour (NN), linear and cubic spline methods.
(Hint: you can print the output with print() function, e.g. print(f(x)) will print the value of function f for a given x value).

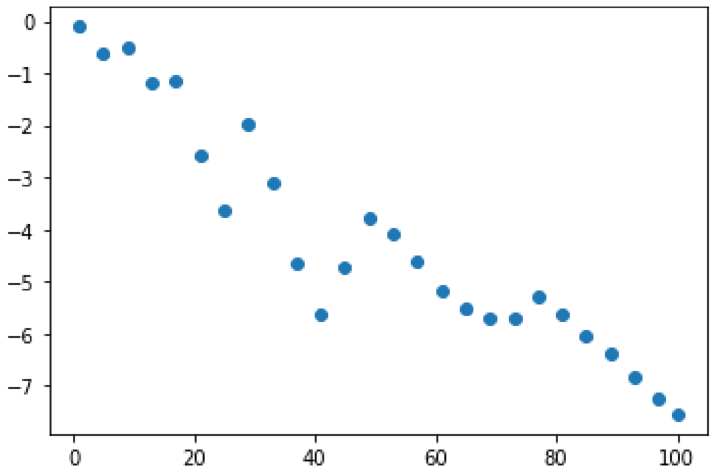
1. NN -10.22; Linear -8.00; Cubic -2.07
2. NN -10.22; Linear -2.07; Cubic -8.00
3. NN -8.00; Linear -2.07; Cubic -10.22
4. NN -2.07; Linear -10.22; Cubic -8.00

4.3 Excercise 4

Let us take a look at depth point measurements. Open txt file (sample_data.txt) with example depth profile data. Use similar function as in the exercise 2. Plot the data points:

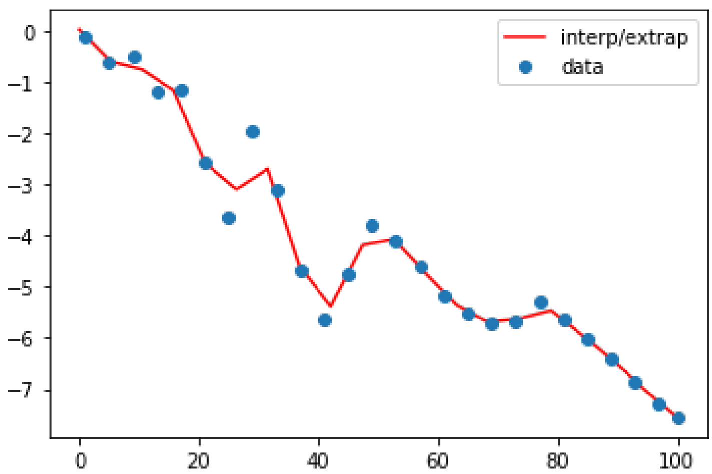
In [27]:
```
sd = pd.read_table('sample_data.txt', sep="\s+")
X = sd.x
D = sd.d
plt.plot(X, D, 'o')
```

Out[27]: [<matplotlib.lines.Line2D at 0x11078d88>]



Now interpolate the data using interp1d function, as in the example above. Plot the results.
Remember that the default interpolation method is linear.

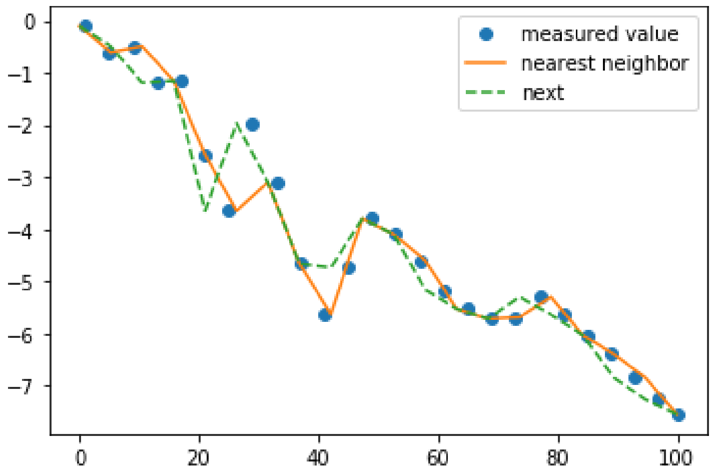
In [28]:
```
X_new = np.linspace(0,100,num=20)
intfunc = interpolate.interp1d(X,D,fill_value="extrapolate")
f = intfunc(X_new)
plt.plot(X_new,f,'r', label='interp/extrap')
plt.plot(X,D, 'o', label='data')
plt.legend()
plt.show()
```



Prepare appropriate lines using nearest and next neighbour alghoritm. Plot the results.
Hint: look at the cell below. What can you write instead of **??** marks?

In [ ]:
```
f1 = interp1d(X, D, ??,fill_value="extrapolate")
f2 = interp1d(X, D, ??,fill_value="extrapolate")
plt.plot(X, D, 'o')
??
```

In [29]:
```
f1 = interp1d(X, D, kind='nearest',fill_value="extrapolate")
f2 = interp1d(X, D, kind='next',fill_value="extrapolate")
plt.plot(X, D, 'o')
X_new = np.linspace(0,100,num=20)
plt.plot(X_new,f1(X_new),'-',X_new,f2(X_new),'--')
plt.legend(['measured value','nearest neighbor','next'],loc='best')
plt.show()
```



Quiz Question 4:
Based on the results from Exercise 4, select the true statement.

1. Significant difference between neighbor values does not influence results of interpolation, when different interpolation methods are used;
2. There is no difference between the three algorithms (nearest neighbor, next neighbor, linear) results;
3. Linear interpolation algorithm is the best one to interpolate depth profile data;
4. There are differences between the results of linear and next neighbor interpolation algorithms.

4.4 2-D interpolation

Examples refer to one dimensional data. As one can see, interpolation is a commonly used technique to create continuous surface from discrete points. However, a lot of real-world phenomena are continuous - elevations, density, temperatures etc. If we wanted to model these surfaces for analysis, it is impossible to take measurements throughout the surface. Hence, the field measurements are taken at various points along the surface and the intermediate values are inferred by interpolation.

Let us take a look at depth point measurements:

In [30]:
```
ed = pd.read_table('example_2d.txt', sep="\s+")
ed.head()
```

Out[30]:

	x_lon	y_lat	d
0	17.96495	54.87318	-21.26429
1	18.10300	54.91549	-24.99855
2	17.97012	54.95810	-23.30563
3	18.01966	54.88771	-22.44334
4	18.16236	54.92071	-24.22124

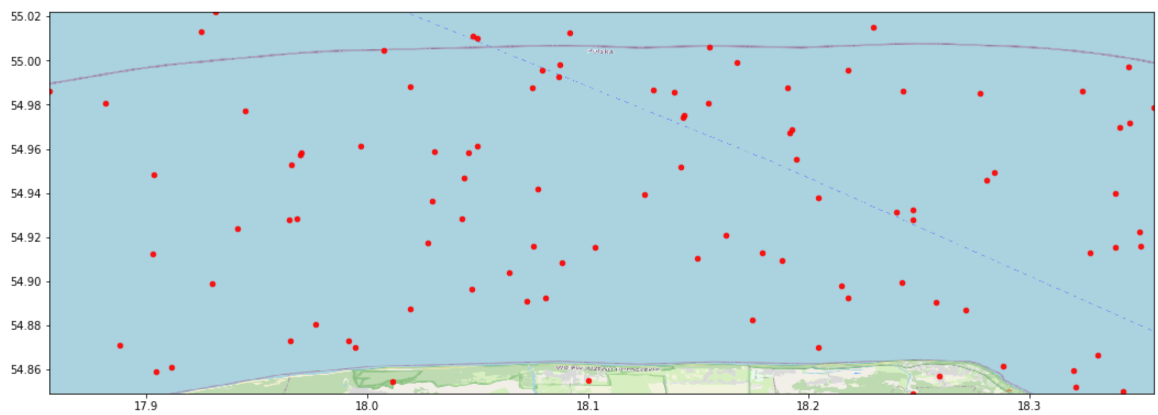
In [31]:
```
BBox = (ed.x_lon.min(), ed.x_lon.max(), ed.y_lat.min(), ed.y_lat.max())
print(BBox)
```

```
(17.85582, 18.35652, 54.8492, 55.022119999999994)
```

We use the same functions as previously to visualize the data.

In [32]:
```
map2 = plt.imread('map2.png')
fig, ax = plt.subplots(figsize = (18,17))
ax.scatter(ed.x_lon, ed.y_lat, zorder=1, alpha= 0.9, c='r', s=20)
ax.set_title(' ')
ax.set_xlim(BBox[0],BBox[1])
ax.set_ylim(BBox[2],BBox[3])
ax.imshow(map2, zorder=0, extent = BBox)
```

Out[32]:
```
<matplotlib.image.AxesImage at 0xc04ed08>
```



We will use matlib functions `tripcolor` and `tricontourf` to show simple method to create surface from irregular points based on unstructured triangular grid. It is important to note, that the triangulation itself is not an interpolation method.

In the default setting, both these functions use Delaunay triangulation method. For given set of points, a set of triangles is generated. Each triangle is given by the indices of the three points that make up the triangle, ordered in either a clockwise or anticlockwise manner. Function `tripcolor` conducts triangulation and fills these triangles with an average of z value from the original set of points. Function `tricontourf` conducts triangulation as well and generates set of contour lines which are the filled. Function `tricontour` leaves contours unfilled.

Data preparation.

In [33]:
```
x = ed.x_lon
y = ed.y_lat
d = ed.d
```

In [34]:
```
f, ax = plt.subplots(1,2, sharex=True, sharey=True, figsize = (16,6))
ax[0].tripcolor(x,y,d)
ax[1].tricontourf(x,y,d, 20) # choose 20 contour levels, just to show how goo
d its interpolation is
ax[0].plot(x,y, 'ko ')
ax[1].plot(x,y, 'ko ')
```

Out[34]: [<matplotlib.lines.Line2D at 0x185e4e88>]