Interpolation

January 14, 2016

1 Interpolation

Interpolation is a method to construct new data points, knowing a set of data points. We will use it to convert units and provide the output result on a regular grid (which may be needed for further processing).

In this exercise we take the output of an azimuthal integration program (SPD) which provide the distance to the center and transform it into diffraction angle 2θ. We will evaluate the calculation time for various polynomial interpolation schemes and also the "correctness" of the interpolation.

1.1 Read the input data

Input data are created by the SPD progam and contain a 1D array of data (EDF files are usually 2D datasets), representing the diffraction intensity as function of the pixel coordinate.

```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

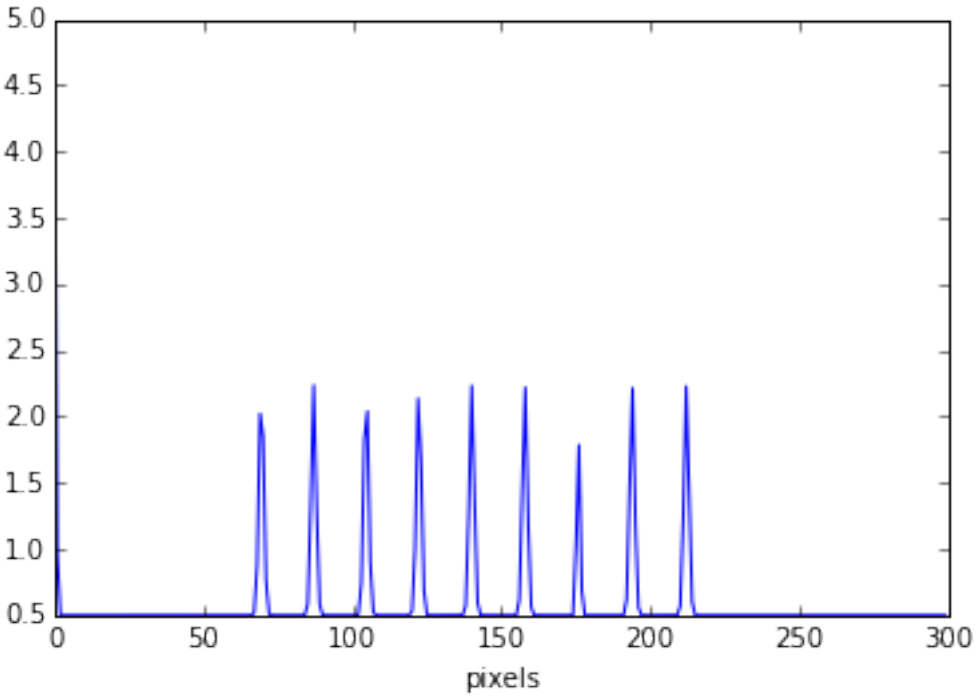
```
In [2]: import fabio
        img = fabio.open("moke-powder.edf")
        data = img.data.reshape(-1)
        for key, value in img.header.items():
            print("%20s:\t %s"%(key,value))
```

```
EDF_DataBlockID:         1.Image.Psd
      EDF_BinarySize:         1536
      EDF_HeaderSize:         8192
           ByteOrder:         LowByteFirst
            DataType:         FloatValue
               Dim_1:         300
               Dim_2:         1
               Dummy:         -5
              DDummy:         0.1
            Offset_1:         0 pixel
            Offset_2:         0 pixel
            Center_1:         0 pixel
            Center_2:         0 pixel
             BSize_1:         1
             BSize_2:         1
             PSize_1:         0.0001 m
             PSize_2:         6.28319 rad
      SampleDistance:         0.1 m
          WaveLength:         1e-10 m
  DetectorRotation_1:         0_deg
  DetectorRotation_2:         0_deg
```

```
  DetectorRotation_3:          0_deg
      ProjectionType:          Saxs
   RasterOrientation:          1
          AxisType_2:          Angle
           History-1:          saxs_mac -add 9 /home/kieffer/workspace/edna/tests/data/images/Moke-2th%%-
           History-2:          spd azim_int=1 azim_pass=1 azim_ext=-powder.edf azim_r0=0 azim_r_num=300 az
            HeaderID:          EH:000001:000000:000000
         Compression:          None
               Image:          1
     SaxsDataVersion:          2.40
                Size:          1536
```

```
In [3]: plot(data)
        xlabel("pixels")
```

```
Out[3]: <matplotlib.text.Text at 0x7f7789e2b3d0>
```



From the EDF header information, one can read the sample distance (SampleDistance: 0.1m) and the pixel size (PSize_1: 0.0001 m).

This is enough to define the pixel position in space. Pixel i being actually located at (i+0.5)*pixel_size. The diffraction angle is then:

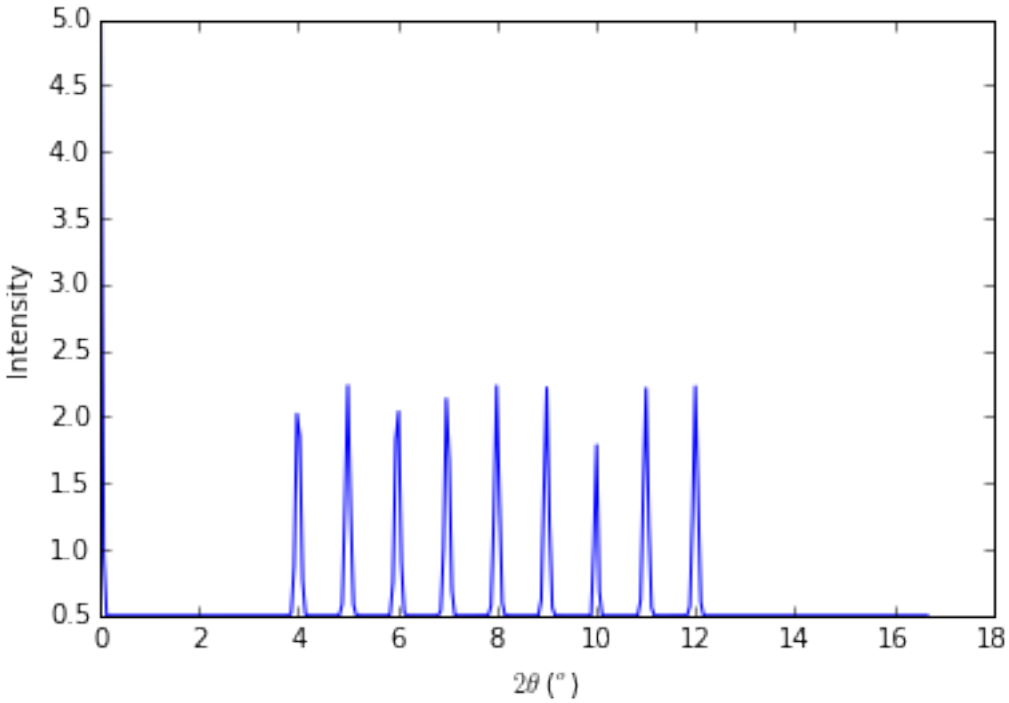
$tan(2\theta_i) = \frac{(i+0.5)*PixelSize}{SampleDistance}$

nota: it is often adviced to use *arctan2* which spreads from $-\pi$ to $+\pi$ while *arctan* is limited to $-\pi/2$ to $+\pi/2$

So one can directly use numpy functions to calculate 2θ:

```
In [4]: tth = numpy.rad2deg(numpy.arctan2(1e-4*(numpy.arange(data.size)+0.5), 0.1))
        plot(tth, data)
        xlabel(r"$2\theta$ ($^{o}$)")
        ylabel("Intensity")
```

Out[4]: <matplotlib.text.Text at 0x7f7789d49750>



These test data are actually triangular shaped peaks at full integer values of 2θ values in degree.

The interpolated values will be taken on a much denser grid, 2000 points instead of the 300 initial points. While this oversampling is not especially meaningful on the physical point of view it will help highlighting artifacts.

1.2 SciPy Interpolation

All 1d interpolation are available from *scipy.interpolate.interp1d*. One needs to provide the initial data_set as x and y and the kind on interpolator expected (often, the order of the polynomial used) and the fill value for "out_of_bound" data.

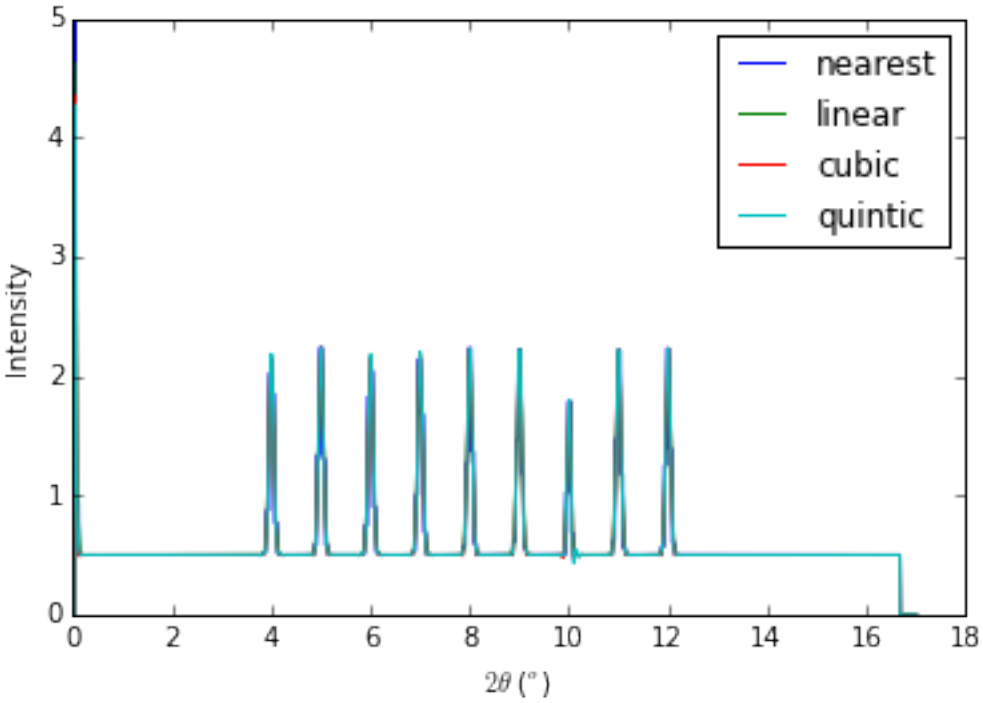
The retruned type is a *function* one may call on the data set of its choice.

```
In [5]: from scipy.interpolate import interp1d
        nearest = interp1d(tth, data, kind="nearest", fill_value=0, bounds_error=False)
        linear = interp1d(tth, data, kind="linear", fill_value=0, bounds_error=False)
        cubic = interp1d(tth, data, kind="cubic", fill_value=0, bounds_error=False)
        quintic = interp1d(tth, data, kind=5, fill_value=0, bounds_error=False)

In [6]: tth_dense = numpy.linspace(0, 17, 2000)

In [7]: plot(tth_dense, nearest(tth_dense), label="nearest")
        plot(tth_dense, linear(tth_dense), label="linear")
        plot(tth_dense, cubic(tth_dense), label="cubic")
        plot(tth_dense, quintic(tth_dense), label="quintic")
        xlabel(r"$2\theta$ ($^{o}$)")
        ylabel("Intensity")
        legend()
```

```
Out[7]: <matplotlib.legend.Legend at 0x7f77868c6f10>
```



1.3 Performance measurement of the various interpolation schemes

For performance measurement we will simply use the *%timeit* magic function of ipython

```
In [8]: %timeit nearest(tth_dense)
```

```
10000 loops, best of 3: 152 µs per loop
```

```
In [9]: %timeit linear(tth_dense)
```

```
1000 loops, best of 3: 255 µs per loop
```

```
In [10]: %timeit cubic(tth_dense)
```

```
1000 loops, best of 3: 318 µs per loop
```

```
In [11]: %timeit quintic(tth_dense)
```

```
1000 loops, best of 3: 423 µs per loop
```

The calculation time increases with the order of the polynomal fitted. With elder version of scipy, the difference was much more striking.

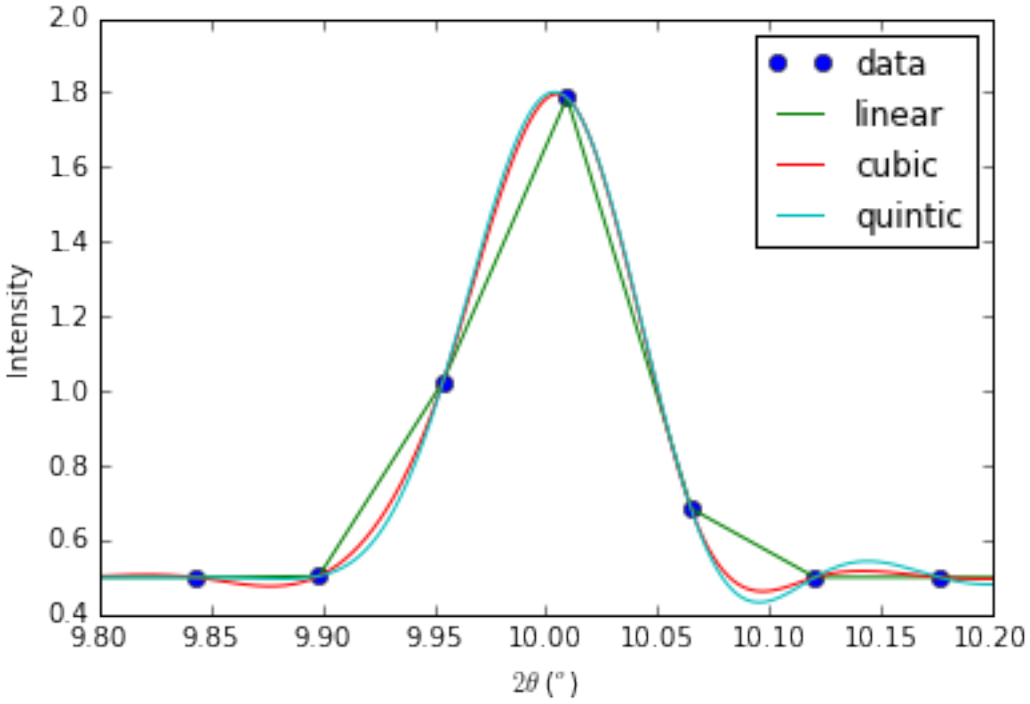
1.4 Validation of the results

Lets zoom in the region 9.8-10.2 and have a closer look at the curves & datapoints:

In [12]:
```
dmin, dmax = 9.8, 10.2

tth_dense = numpy.linspace(dmin, dmax, 500)
#plot(tth_dense, nearest(tth_dense), label="nearest")
mask = (tth <= dmax) & (tth >= dmin)
plot(tth[mask], data[mask], "o", label="data")
plot(tth_dense, linear(tth_dense), label="linear")
plot(tth_dense, cubic(tth_dense), label="cubic")
plot(tth_dense, quintic(tth_dense), label="quintic")
xlabel(r"$2\theta$ ($^{o}$)")
ylabel("Intensity")
legend()
```

Out[12]: <matplotlib.legend.Legend at 0x7f77867a4110>



Our input image had peak of triangular shape: those are peaks are continuous but discontinuous on their first derivative. This discontinuity introduces artifacts when using higher order interpolators (cubic, ...).

2 Conclusion

While cubic spline (and higher order) present smoother interpolated curves, they can introduce wobbles, hence negative intensities. The safest interpolation scheme remains the **linear interpolation** which garanties integrated intensity conservation but at the price of a broadening of the signal.