A Data Science and Visualization Primer

Just a Day at the Office – Smoothing/Denoising, Baseline Corrections and Peak Detection

assoc. Prof. Dr. Thomas Hofer
Department of Theoretical Chemistry
Email: T.Hofer@uibk.ac.at

Stefanie Kröll, MSc
Department of Theoretical Chemistry
Email: Stefanie.Kroell@uibk.ac.at

Welcome to the Third Exercise Block

In this slide set, we will outline the basis of smoothing data series to reduce noise and carry out baseline corrections.

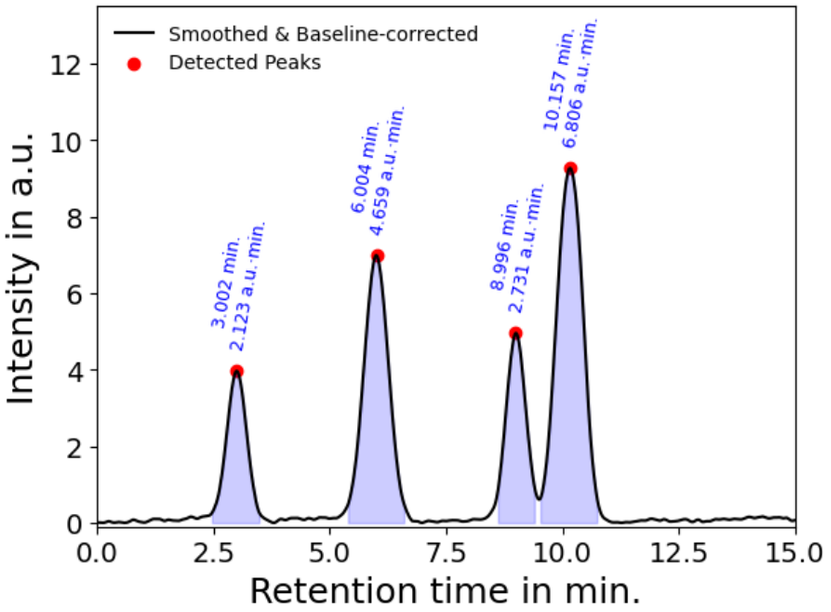
This steps are difficult or nearly impossible using standard spreadsheet software and are brilliant examples in which Scipy offers a flexible and powerful alternative.

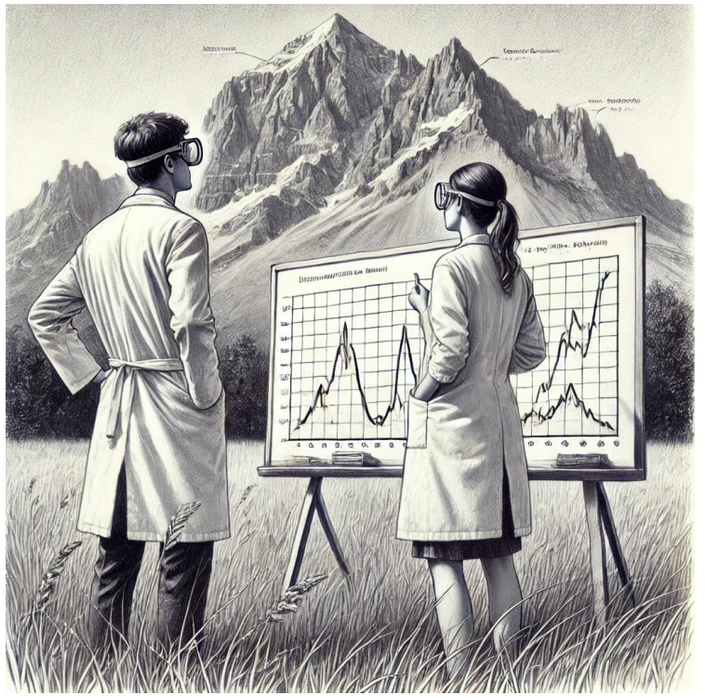
After cleaning the data we can then apply the highly versatile peak detection algorithms.

These kind of analysis methods can be relevant when depicting X-ray diffraction patterns, spectrograms from various techniques (IR/Raman, UV/Vis, X-ray, NMR, ...), chromatograms/electropherograms and many more.

If done right, Scipy and Numpy also offer creative ways for automated peak integration.

While setting up the analysis for a single plot can be a bit tedious and time consuming, the methods really shine in case the process can be automated for a larger number of samples.

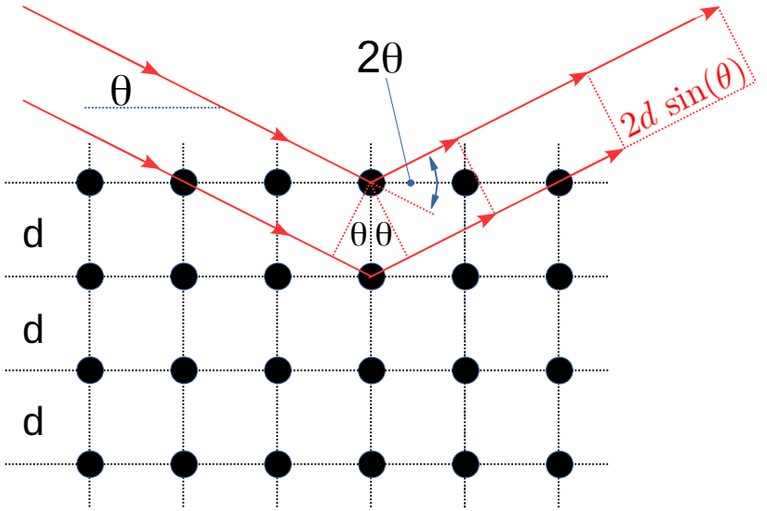




Exercise C.1 – Peak Detection in X-ray Diffraction (XRD) Patterns

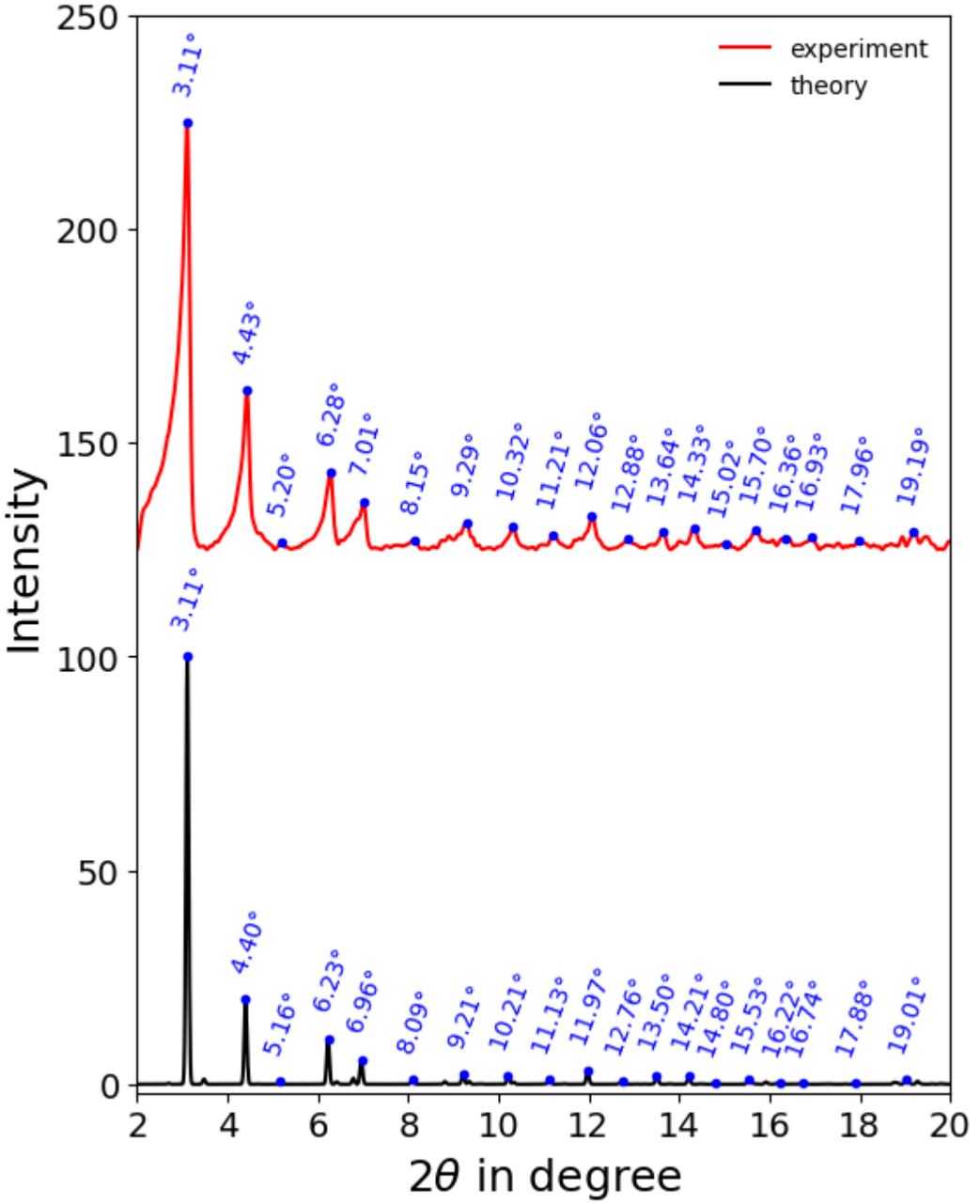
A key method in chemical sciences to characterise crystallized compounds is X-ray diffraction.

XRD patterns are obtained by measuring the intensity I of the reflexes observed at certain angles 2θ after irradiating the sample with X-ray beams of constant wave length:



Typically, the experimental data are compared to a calculated reference taken from chemical structure data bases.

Let's start by importing the respective files.



Loading the Files and Setting Up the Data (*low difficulty*)

Similar as in Exercises A and B we upload the data files to colab. Alternatively, providing access *via* google drive is possible.

```
from google.colab import files

uploaded = files.upload()
```

In the next cell we load the required python modules (again NumPy and Matplotlib).

We load the data from the files into the arrays `xrd_exp` and `xrd_theo` using again `np.loadtext()`.

As before we extract the data into distinct arrays.

In this example we use `x_exp` and `x_theo` to store the angle data, the respective intensities are are stored in `y_exp` and `y_theo`.

```
import numpy as np
import matplotlib.pyplot as plt

xrd_exp = np.loadtxt('XRD_experimental.dat')
xrd_theo = np.loadtxt('XRD_theory.dat')

x_exp  = xrd_exp[:, 0]
y_exp  = xrd_exp[:, 1]

x_theo = xrd_theo[:, 0]
y_theo = xrd_theo[:, 1]
```

ProTip: Storing data in separate array does not require any additional memory but makes working with data sets that much easier.

Always Double-Check Your Workflow (… or It Will Be Wrong)

Did we read the data correctly ... ?

It is always a good idea to have a look at the data before starting any analysis by generating a quick plot for orientation (no title, axis label, *etc.* required for now).

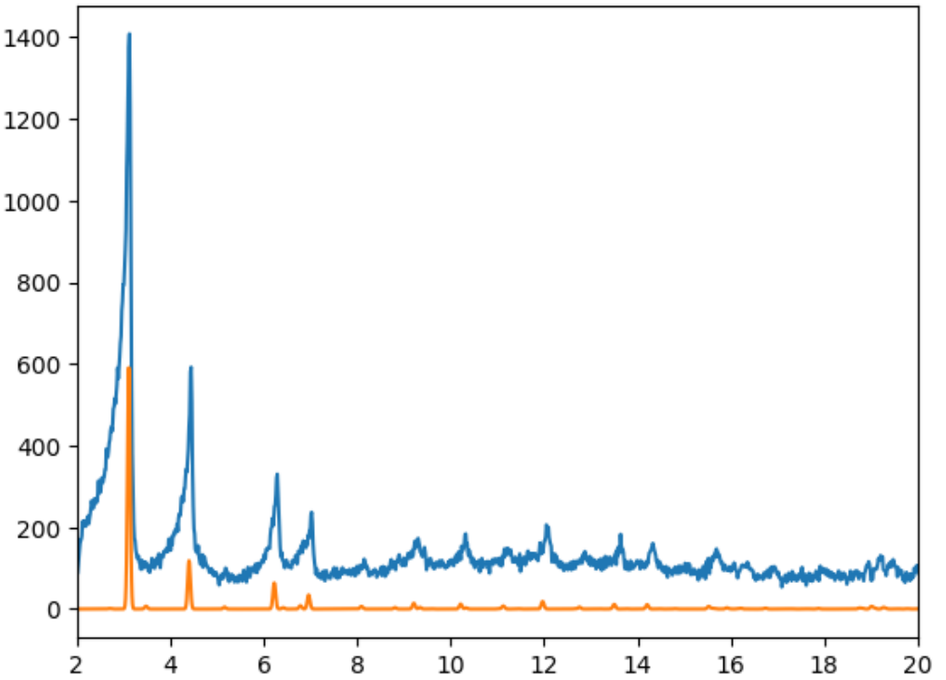
As before we use the `plt.plot()` function of to show both data sets in a single graph.

Without specifying any colors, the defaults are applied.

We can see that the experimental data (shown in blue) has a lot of noise and the baseline has a notable offset of approx. 100 a.u.

Using modules form Scipy smoothing (SM) and baseline correction (BC) are straightforward.

```
plt.plot(x_exp,  y_exp)
plt.plot(x_theo, y_theo)

xmin=2.0
xmax=20.0

plt.xlim(xmin, xmax)

plt.show()
```



Data Denoising and Peak Detection (*low/intermediate difficulty*)

Using the command `gaussian_filtered1d()` we can achieve a denoising using just a single line, with `sigma` steering the degree of denoising.

The smoothed y-values are stored in `y_smooth`.

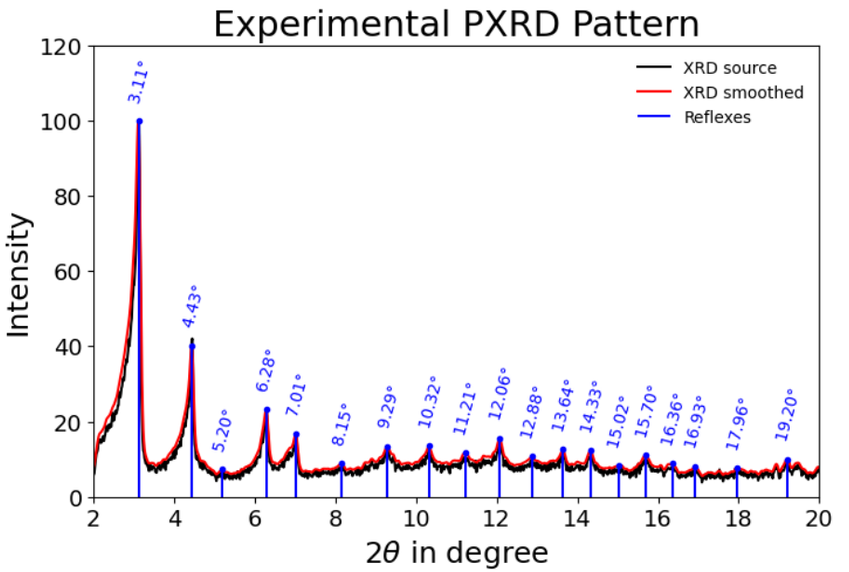
Since the smoothing influences the height of the data, we have so separately normalize the `y_exp` and `y_smooth` to a value of 100 using `np.max()`.

The command `find_peaks()` identifies maxima:

- The output array `peaks` contains the indices of the found maxima.
- The dictionary `info` contains further data, which we don't need in this exercise.

To access the actual maxima we have to use the array `peak` in the source data, `x_exp[peaks]` and `y_smooth[peaks]`.

```
from scipy.ndimage import gaussian_filter1d
from scipy.signal import find_peaks

y_smooth = gaussian_filter1d(y_exp, sigma = 2)

y_smooth = 100 * y_smooth / np.max(y_smooth)
y_exp    = 100 * y_exp    / np.max(y_exp)

peaks, info = find_peaks(y_smooth, distance = 35,
                         prominence = 0.75)
[...]

plt.plot(x_exp, y_exp, color='k', label="XRD source")

plt.plot(x_exp, y_smooth, color='r',
         label="XRD smoothed")

plt.plot(x_exp[peaks], y_smooth[peaks], "o",
         color = "b", markersize = 3)
```



Data Denoising and Peak Detection (*low/intermediate difficulty*)

A helpful visualization is the addition of *drop lines*, vertical lines between zero and the data points.

In this example we draw the drop lines at the position of the maxima `x_exp[peaks]` and within the range of `ymin = 0` to `ymax = y_smooth[peaks]`.

Add labels using the `plt.text()` command:

- Loop over all identified peaks

  `for p in range(len(peaks)):`

- Only consider maxima in the range 2 to 20°

  `if x_exp[peaks[p]] < xmax:`

- Write the text at the peak position using several space characters for offset:

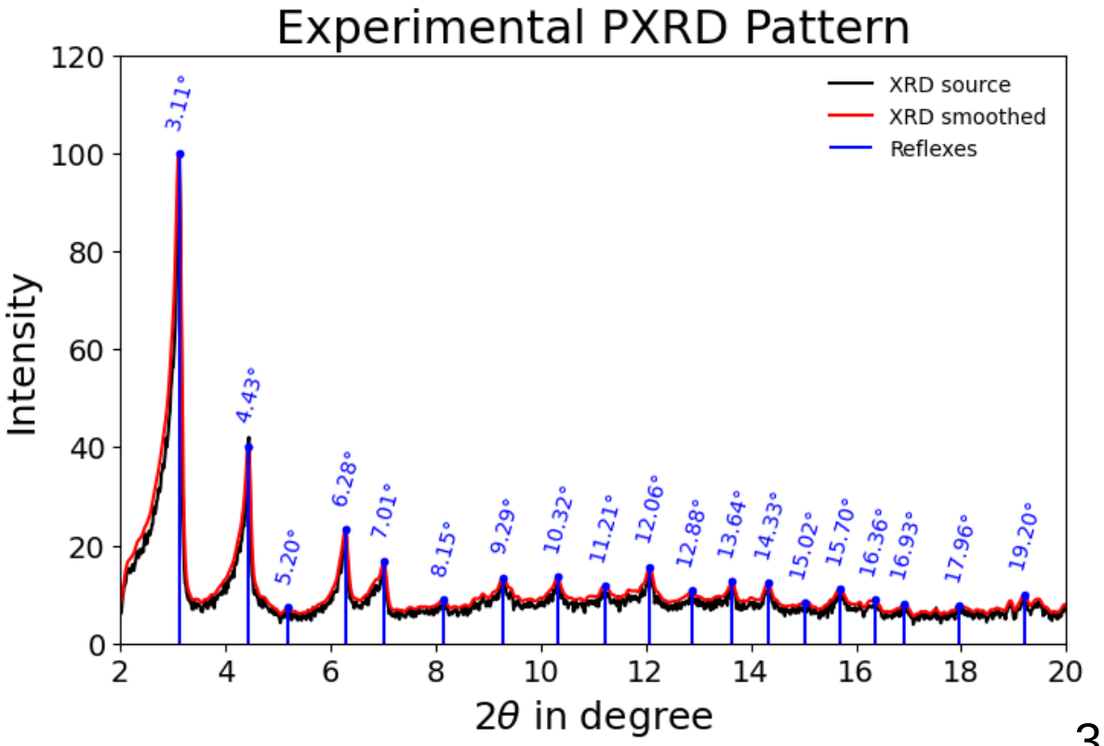
  `f"    {x_exp[peaks[p]]:.2f}°"`

- Write the text at a slight angle (optional)

  `rotation = 75`

- Fine tune the position using horziontal alignment: `ha = 'left'`, `'right'` or `'center'`

```
plt.vlines(x_exp[peaks], ymin = 0,
           ymax = y_smooth[peaks], color='b',
           label = "Reflexes" )

for p in range(len(peaks)):
  if x_exp[peaks[p]] < xmax:
    plt.text(x_exp[peaks[p]],
             y_smooth[peaks[p]] + 0.05,
             f"    {x_exp[peaks[p]]:.2f}°",
             fontsize = 10, ha = 'center',
             color = 'blue', rotation = 75)
```



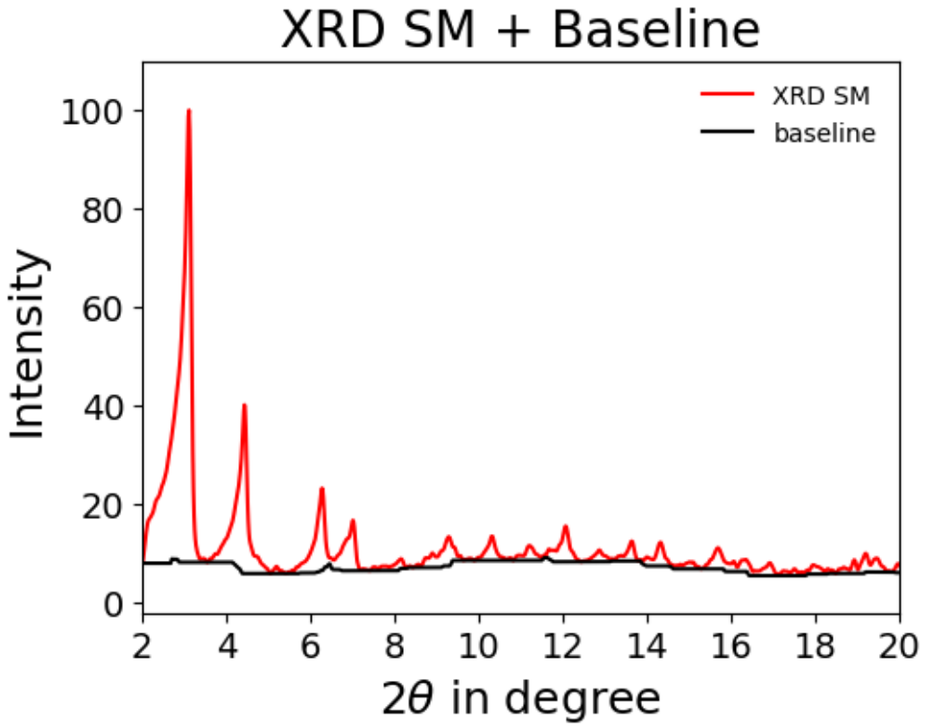
Baseline correction (*low difficulty*)

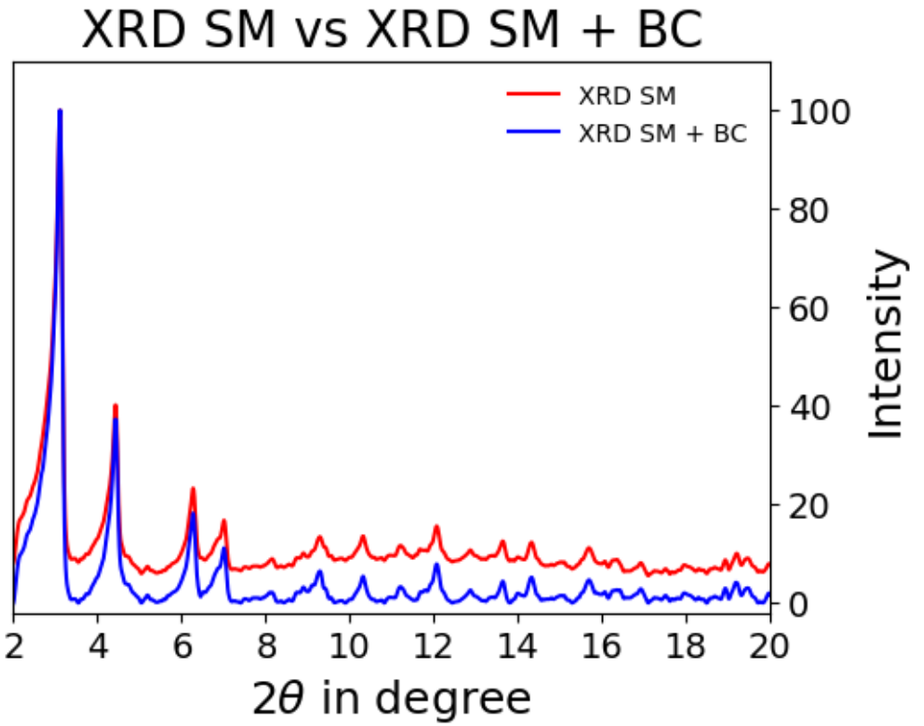
Using only the single command `minimum_filter1d()` a `baseline` is obtained. The parameter `size` is used to balance the baseline detection.

Next, the baseline is subtracted from `y_smooth`.

The resulting smoothed (SM) & baseline-corrected (BC) data `y_corr` is renormalized to 100 again.

```
from scipy.ndimage import minimum_filter1d
baseline = minimum_filter1d(y_smooth, size = 90)
y_corr = y_smooth - baseline
y_corr = 100 * y_corr / np.max(y_corr)
```





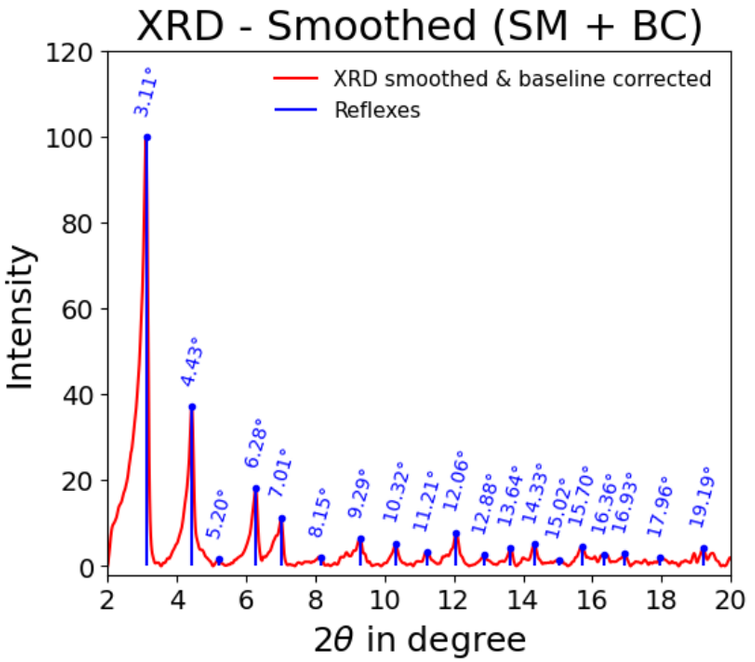
Correlation Plots of Detected Reflexes (*low difficulty*)

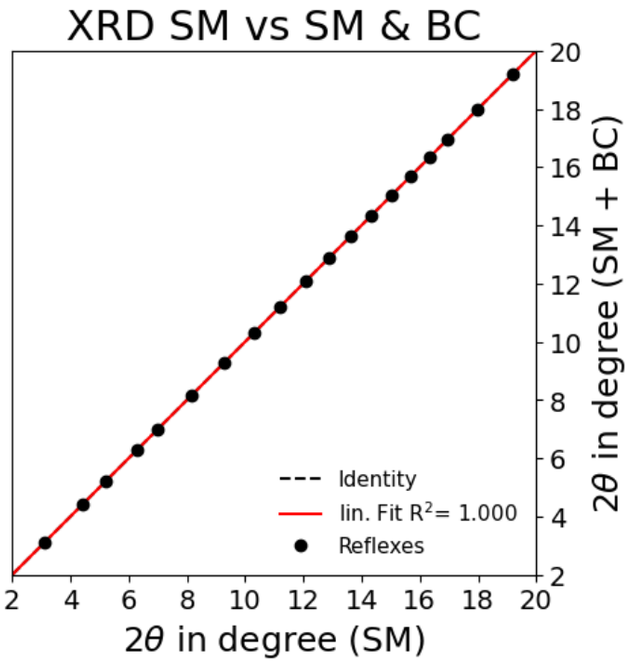
Comparing the detected 2θ values before and after the baseline correction is a perfect example for a correlation plot.

After applying the peak detection to the baseline-corrected `y_corr`, we employ linear regression to assess the reflexes using the range setting `[0:19]`.

<u>Careful:</u> Make sure that the same reflexes are identified in both data sets `y_smooth` and `y_corr`.

```
from scipy.stats import linregress

peaks_corr, info_corr = find_peaks(y_corr,
                                   prominence = 0.5,
                                   distance = 35)

reg = linregress(x_exp[peaks[0:19]],
                 x_exp[peaks_corr[0:19]])

[…]

ax2.set_aspect('equal')

ax2.plot(x_exp[peaks[0:19]], x_exp[peaks_corr[0:19]],
         marker = 'o', linestyle = ' ', color = 'k',
         label = "Reflexes")
```





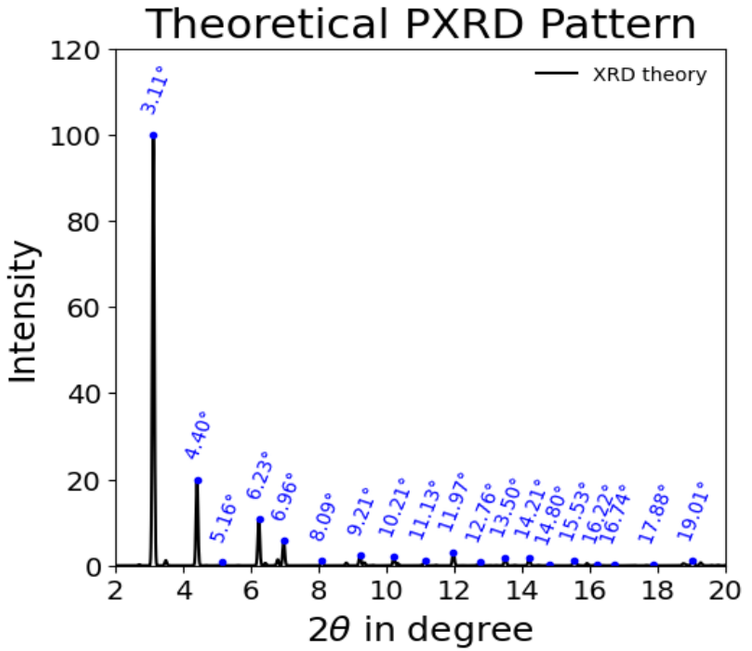
Comparison – Experiment vs Theory (*low difficulty*)

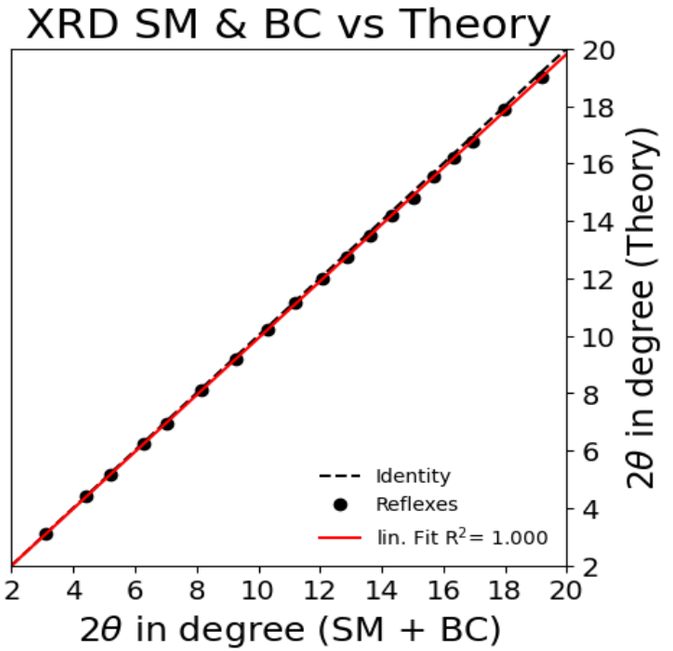
Working with the theoretical data is much easier since neither denoising nor a baseline correction is required.

After normalization of `y_theo` to 100, we again employ linear regression in the range `[0:19]`.

Careful: Make sure that the same reflexes are identified in both data sets `y_corr` and `y_theo`.

```
y_theo = 100 * y_theo/ np.max(y_theo)

peaks_theo, info_theo = find_peaks(y_theo,
                                   prominence = 0.1,
                                   distance = 50)

reg2 = linregress(x_exp[peaks[0:19]],
                  x_theo[peaks_theo[0:19]])
[…]

ax2.set_aspect('equal')

ax2.plot(x_exp[peaks[0:19]], x_theo[peaks_theo[0:19]],
         marker = 'o', linestyle = ' ', color = 'k',
         label = "Reflexes")
```

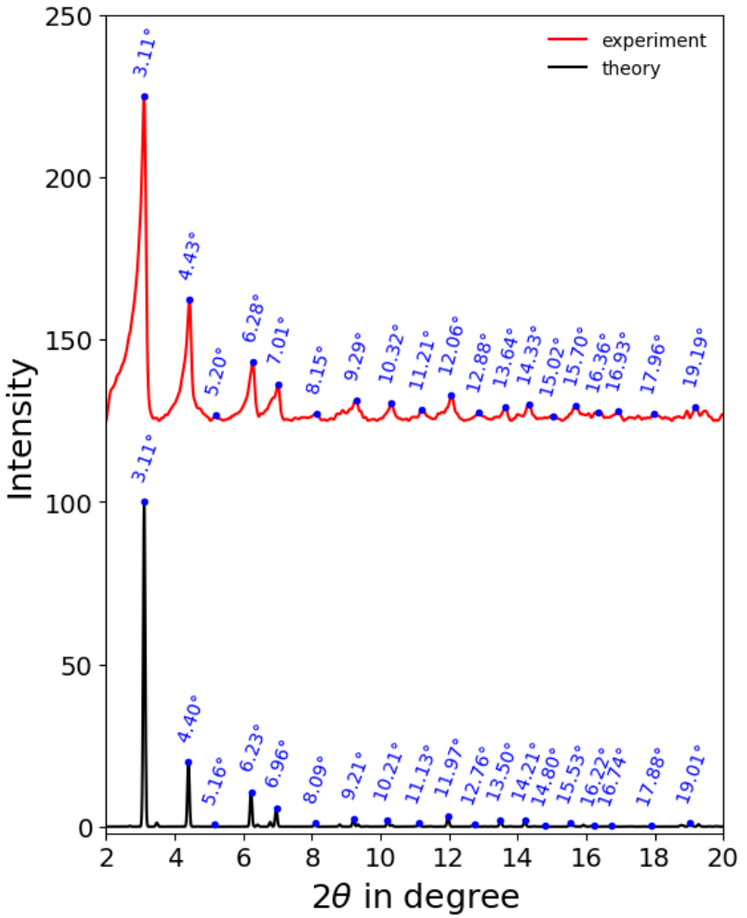




Final Image for Publication

Now that we verified the data, we can join the plots into a single graph.

To make the experimental data appear above the theoretical one, we simply define a suitable `offset` value.




```
# Plot the experimental XRD at an offset
offset = 125

plt.plot(x_exp, y_corr + offset, color='r', label="experiment")

plt.plot(x_exp[peaks_corr], y_corr[peaks_corr] + offset, "o",
         color = "b", markersize = 3)

for i in range(len(peaks_corr)):
  if x_exp[peaks_corr[i]] < xmax:
    plt.text(x_exp[peaks_corr[i]],
             y_corr[peaks_corr[i]] + offset,
             f"    {x_exp[peaks_corr[i]]:.2f}°", fontsize=10,
             ha='center', color='b', rotation=75)

# Plot the experimental XRD at zero level (no offset required)
plt.plot(x_theo, y_theo, color='k', label="theory")

plt.plot(x_theo[peaks_theo[0:19]], y_theo[peaks_theo[0:19]],
         "o", color = "b", markersize = 3)

for i in range(len(peaks_theo[0:19])):
  if x_theo[peaks_theo[i]] < xmax:
    plt.text(x_theo[peaks_theo[i]],
             y_theo[peaks_theo[i]],
             f"    {x_theo[peaks_theo[i]]:.2f}°",
             fontsize = 10, ha = 'center',
             color = 'b', rotation = 75)
```

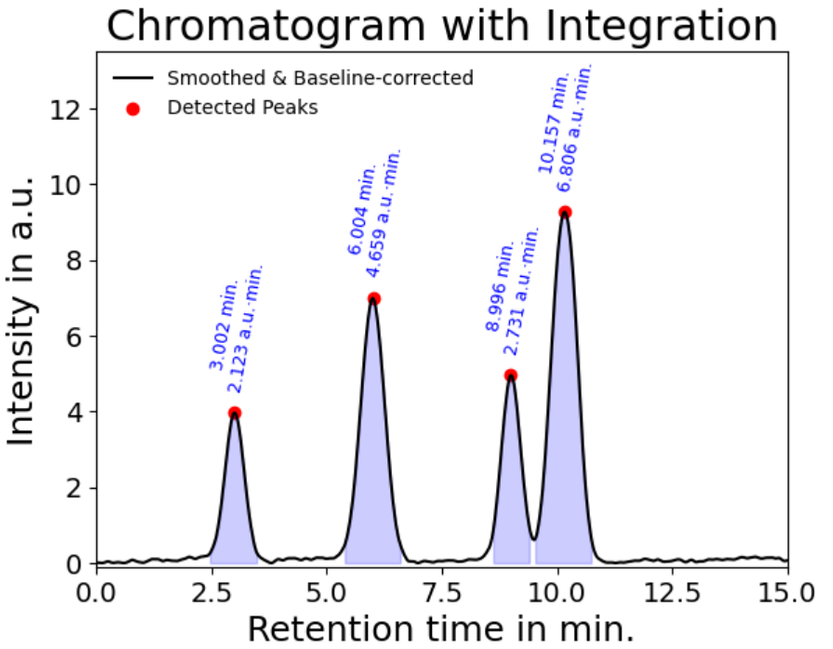
Exercise C.2 – Integration by Peak (*advanced difficulty*)

In many applications (*e.g.* chromatography, electrophoresis time series of repeated peaks are obtained.

This data is oftentimes subject to noise and baseline drifts, making the determination of the area challenging.

By using the capabilities of Scipy's `find_peaks()` command, the peak detection and integration can be fully automated.

Here, we work again with a csv-file, so we set `delimiter = ','`. Also, this file contains an extra headline, that we can ignore using `skiprows = 1`.




```
Time (min),Intensity
0.0,0.0570612997219531
0.01000667111407605,0.09731879960898993
0.0200133422281521,-0.03213017120692467
0.0300200133422281 54,-0.059722672783663625
0.0400266844563042,-0.10651371843296462
0.05003335557038025,-0.08003938521937651
[...]
```

`Time (min),Intensity` ← headline (has to be skipped)

```
from google.colab import files

uploaded = files.upload()
```

```
import numpy as np

data = np.loadtxt("Chromatogram.csv",
                  delimiter=',',
                  skiprows = 1)

t = data[:,0]
I = data[:,1]
```

Denoising and Baseline Correction (*low difficulty*)

Proceed as in the previous example:

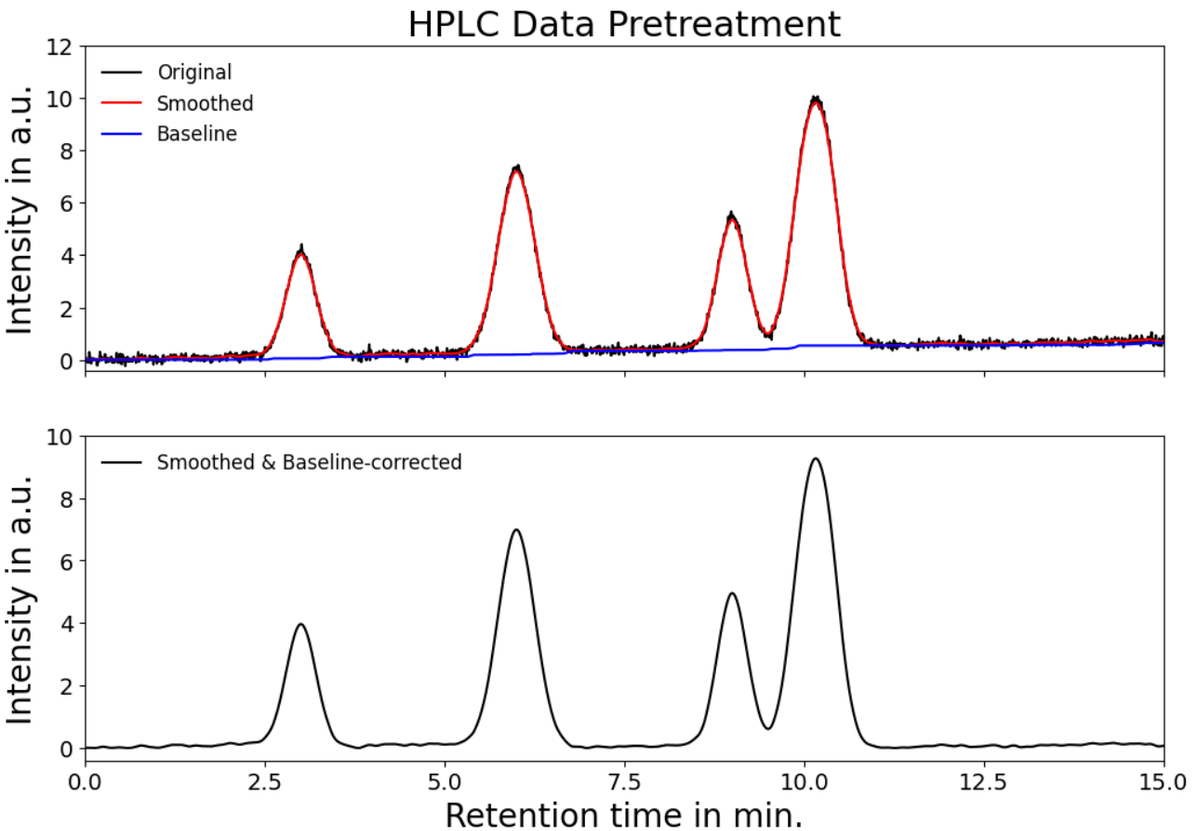
- `gaussian_filtered1d()` for smoothing/denoising. Experiment with the `sigma` value to find the best setting.
- `minimum_filter1d()` calculates the baseline correction. Experiment with the `size` parameter to find the best compromise.

Careful: We need to properly adjust the parameters `sigma` and `size` in an way to find the ideal settings.

Careful: Always plot the data or you will miss potential errors!

This is an ideal example for `plt.subplot()`. We show the original and denoised chromatogram together with the baseline on top, and the smoothed and baseline-corrected data on the bottom.

```
from scipy.ndimage import gaussian_filter1d
from scipy.ndimage import minimum_filter1d
import matplotlib.pyplot as plt

I_smooth = gaussian_filter1d(I, sigma = 5)

baseline = minimum_filter1d(I_smooth, size = 300)
I_corr = I_smooth – baseline

[...]
```



Peak Detection (*intermediate difficulty*)

Now we can simply apply find peaks again.

This time we make use of the `info` object. It is a python dictionary containing additional information about the identified peaks:

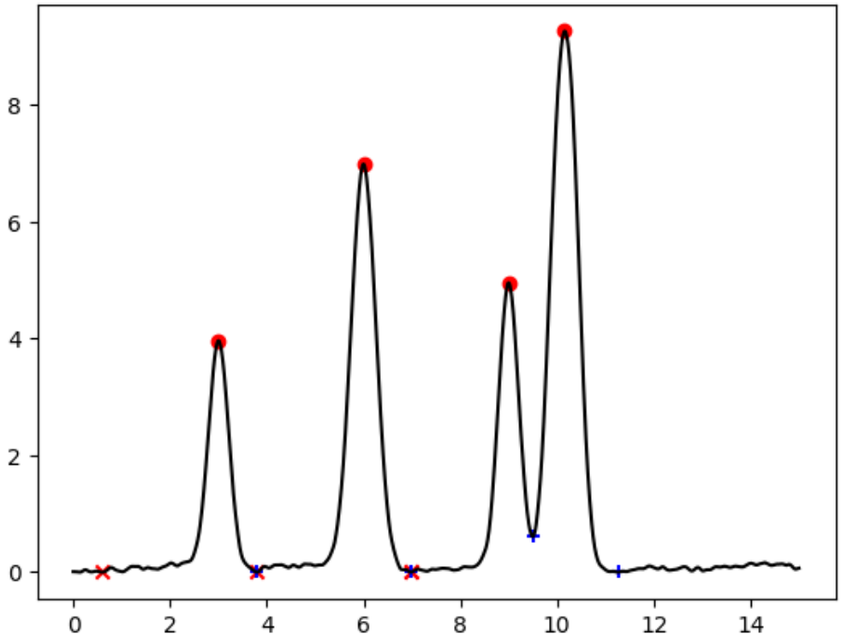
```
print(info)

{'prominences': array([3.964, 6.985, 4.345, 9.263]),
 'left_bases': array([ 59, 380, 697, 697]),
 'right_bases': array([ 380,  697,  949, 1125])}
```

- Prominence: How much a peak stands out.
- Left base: Start position of the peak as index.
- Right base: End position of the peak as index.

While the peak identification is straightforward, the default for the base detection is not particularly good, especially when we have difficult data such as the merged peaks 3 and 4.

Luckily, there is a way to fine-tune the base detection.

```
from scipy.signal import find_peaks

peaks, info = find_peaks(I_corr, prominence=0.20)

plt.plot(t, I_corr, color='k')
plt.scatter(t[peaks], I_corr[peaks], color='red')

left_bases  = info['left_bases']
right_bases = info['right_bases']

plt.scatter(t[left_bases], I_corr[left_bases],
            marker = 'x', color='red')

plt.scatter(t[right_bases], I_corr[right_bases],
            marker = '+', color='blue')
```



Peak Detection (*intermediate difficulty*)

The module `peak_widths()` provides a more sensitive way to extract data about the peaks, *i.e.*:

- Widths
- Left-bases
- Peak Heights
- Right bases

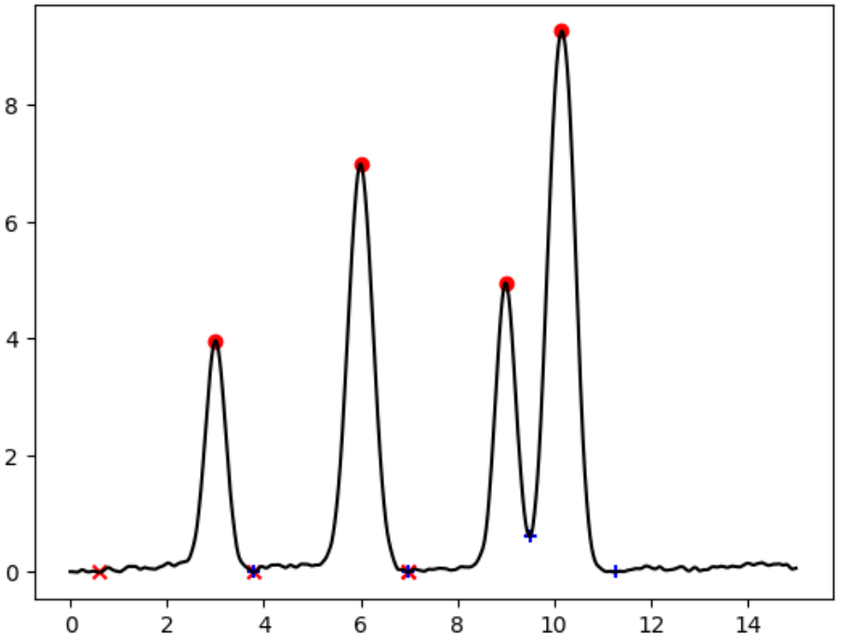
Careful: The abbreviation `ips` means index position and are interpolated values (non-integer).

The values for `left_ips` and `right_ips` have to be converted to integer values when plotting using the command `.astype(int)`.

To properly separate the last peaks, the setting `rel_height=0.93` is adequate. If this value is set higher, the two peaks would be detect as one.

This is not ideal for the peak 1 and 2. In a more advanced analysis, `peak_widths()` could be applied for instance two times, each with different settings for `rel_height`.

```
from scipy.signal import peak_widths

widths, peak_heights, left_ips, right_ips, =
peak_widths(I_corr, peaks, rel_height=0.93)

[…]

plt.scatter(t[left_ips.astype(int)],
            I_corr[left_ips.astype(int)],
            marker = 'x', color='red')

plt.scatter(t[right_ips.astype(int)],
            I_corr[right_ips.astype(int)],
            marker = '+', color='blue')
```



Final Image for Publication (*intermediate difficulty*)

To carry out the actual integration we use the Numpy module `np.trapezoid()`.

As an exercise we manually calculate the peak widths at the same time.

We want to store the area data for later use in an array. For this we need to:

- know how many peaks have been detected.

  ```
  n_peaks = len(peaks)
  ```

- prepare an array beforehand.

  ```
  area = np.zeros(n_peaks)
  ```

- loop over all peaks and calculate one by one

  ```
  for p in range(n_peaks):
    [...]
  ```

As before we need to convert the values for `left_ips` and `right_ips` to integer values using the command `.astype(int)`.

```
n_peaks = len(peaks)

area = np.zeros(n_peaks)

[...]

for p in range(n_peaks):
  left = left_ips[p].astype(int)
  right = right_ips[p].astype(int)
  width = t[right] - t[left]
  area[p] = np.trapezoid(I[left:right],
                         t[left:right])

[...]
```

```
-----------------------------------------------------
     Ret. Time / min.  Area / a.u.·min.  Width / min.
-----------------------------------------------------
 1 of 4:   3.002             2.123           1.011
 2 of 4:   6.004             4.659           1.201
 3 of 4:   8.996             2.731           0.781
 4 of 4:  10.157             6.806           1.211
-----------------------------------------------------
```

Final Image for Publication (*intermediate difficulty*)

Here, we highlight the integration area of each peak using `plt.fill_between()`. The `alpha` parameter steers the amount of translucency.

Also in this case we need to convert the values for `left_ips` and `right_ips` to integer values using the command `.astype(int)`.

As in the exercise before we plot the chromatogram labeling the individual peaks. This time we print two lines per peak by using the newline character `\n`.

While the integration for the peaks is not ideal (due to the setting of `rel_height=0.93` above), it should not have an impact if all samples area analyzed using identical settings.

Alternatively, a separate base detection for peaks 1 and 2 can be implemented.

```
plt.plot(t, I_corr, color='k',
         label = 'Smoothed & Baseline-corrected')

plt.scatter(t[peaks], I_corr[peaks],
            color='red', label = 'Detected Peaks')

for p in range(n_peaks):
  left = left_ips[p].astype(int)
  right = right_ips[p].astype(int)
  plt.fill_between(t, I_corr,
                   where=(t >= t[left]) & (t <= t[right]),
                   color='b', alpha=0.2)
  plt.text(t[peaks[p]], I_corr[peaks[p]], fontsize=9,
           f"    {t[peaks[p]]:.3f} min.\n
                 {area[p]:.3f} a.u.·min.",
           ha='center', color='blue', rotation=80)
```