Data Fitting

Exercise 3

Comparison between Polynomial and Natural Cubic Spline Interpolation

Author
Cesare De Cal

Professor
Annie Cuyt

Assistant Professor
Ferre Knaepkens

November 5, 2019

1 Introduction

An imaginary chemistry experiment produces the following data set:

x_i	-1	-0.96	-0.86	-0.79	0.22	0.50	0.93
f_i	-1.000	-0.151	0.894	0.986	0.895	0.500	-0.306

The purpose of this exercise is to use these data points to compute and plot the interpolating polynomial together with the natural cubic spline.

As we've seen in class with the Runge's phenomenon, there may be an unsatisfactory oscillating behavior in high-degree polynomial interpolants (especially on equidistant grids). The natural cubic spline, however, which falls under piece-wise interpolation, will be stiffer, have less tendency to oscillate, and give a better insight into the chemistry experiment data.

2 Tools

The following programming language and libraries have been used in this exercise:

- Python 3.7
- SciPy

The SciPy `interpolate` sub-package was used to compute the interpolating polynomial and the natural cubic spline:

- `scipy.interpolate.lagrange(x, y)`
- `scipy.interpolate.CubicSpline(x, y, bc_type)`

The following NumPy methods of the SciPy environment have been used in this exercise:

- `numpy.array(object)`
- `numpy.linspace(start, stop, num)`
- `numpy.polynomial.polynomial.Polynomial(poly)`
- `numpy.setprint_options(precision)`

The following Matplotlib methods of the SciPy environment have been used in this exercise to plot:

- `matplotlib.pyplot.plot(x, y, formatting, label)`
- `matplotlib.pyplot.legend()`
- `matplotlib.pyplot.show()`
- `matplotlib.pyplot.figure(dpi)`
- `matplotlib.pyplot.ylim(miny, maxy)`
- `matplotlib.pyplot.xlabel(name)`
- `matplotlib.pyplot.ylabel(name)`

3 Computation and plotting

The exercise asks to compute the interpolating polynomial of the given data set. To do so, I first create two arrays in Python containing the data points using `np.array` and a linear space from -1 to 1 containing 1000 equidistantly-spaced points. These values are finally passed to the `lagrange` method which returns the *Lagrange* interpolating polynomial (`poly = lagrange(x, y)`).

The following are coefficients the interpolating polynomial retrieved with `Polynomial(poly).coef` in the Lagrange basis:

- 4.3649939020826437e-02
- 1.6076695441034385e+01
- -4.8402059220598259e-02
- -2.0100476671430044e+01
- 1.6880653327119008e-02
- 5.0294636999940936e+00
- -6.4460635283115466e-03

Here is the 6th degree Lagrange equation:

$$0.04365x^6 + 16.08x^5 - 0.0484x^4 - 20.1x^3 + 0.01688x^2 + 5.029x - 0.006446$$

In order to calculate the natural cubic spline I use the built-in SciPy method `CubicSpline(x, y, bc_type)` and use "Natural" for the `bc_type`. Here are the coefficients for the 6 equations:

- $x^3 : [-8.0654538905002710e+02, 1.4844899135604635e+02, 2.5130673577722442e+02, -7.2398801257531864e-01, 1.2138819568393451e+00, 5.0415317855085340e-01]$
- $x^2 : [-8.5265128291212022e-14, -9.6785446686003496e+01, -5.2250749279189527e+01, 5.2366523402757015e-01, -1.6700184440756460e+00, -6.5035760033060053e-01]$
- $x^1 : [2.2515472622480029e+01, 1.8644054755039885e+01, 3.7404351585205808e+00, 1.1953927535924665e-01, -1.0382774667893093e+00, -1.6879827592230572e+00]$
- $x^0 : [-1.0000000000000000e+00, -1.5100000000000000e-01, 8.9400000000000002e-01, 9.8599999999999999e-01, 8.9500000000000002e-01, 5.0000000000000000e-01]$

4 Observations

The natural cubic spline is a much more accurate representation than the *Lagrange* function, which appears inconsistent with several *ups and downs* and doesn't provide any insights into the experiment data.

Since the natural cubic spline is a type of piece-wise interpolation, the graph passes through the experiment data points with low-degree polynomials. This way, we only use low-degree polynomials and therefore eliminate the excessive oscillations that are present in the interpolating polynomial.