Problems week 4

Magnus Chr. Hvidtfeldt

Technical University of Denmark, Lyngby, DK,

s255792@dtu.dk

1 Datafitting

1.1 Et datafittingsproblem

Vi henter vores givne vektorer med scipy, og bruger basisfunktionen

$$F(x) = c_0 + c_1 x + c_2 sin(x) + c_3 cos(x) + c_4 sin(2x) + c_5 cos(2x)$$

Dermed benytter vi normalligningen $A^T Ax = A^T b$ således hvorved vi kan aflæse vores A til at værende vores basisfunktioner. Vi har

$$A = \begin{bmatrix} g_0(x_0) & \dots & g_n(x_0) \\ g_0(x_1) & \dots & g_n(x_1) \\ \vdots & \ddots & \vdots \\ g_0(x_m) & \dots & g_n(x_m) \end{bmatrix}$$

Hvor g_i er vores basisfunktion og x er vores data. I denne opgave har vi dermed basisfunktionerne

$$A = \begin{bmatrix} | & | & | & | & | & | \\ \cos(2x) & \sin(2x) & \cos(x) & \sin(x) & x & 1 \\ | & | & | & | & | & | \end{bmatrix}$$

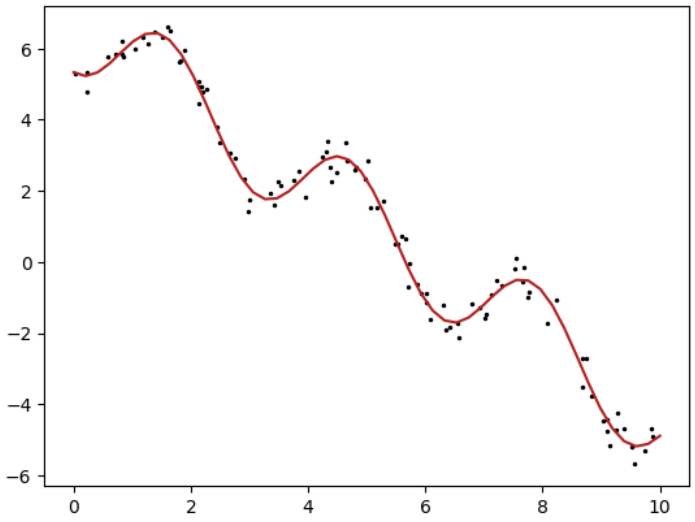
Det implementerer vi således

```
from scipy import io
from sympy import *
import numpy as np
init_printing()
FitData = io.loadmat('FitData.mat')
x = np.ravel(FitData['x'])
y = np.ravel(FitData['y'])
A = np.array([np.cos(2*x), np.sin(2*x), np.cos(x), np.sin(x), x,
np.ones(100)]).T
c = np.linalg.solve(A.T @ A, A.T @ y)

def F(x):
    return c[5] + c[4]*x + c[3]*np.sin(x) + c[2]*np.cos(x) +
    c[1]*np.sin(2*x) + c[0]*np.cos(2*x)
```

Vi plotter dermed funktionen for at se hhv. punkterne og vores fitting således

```
import matplotlib.pyplot as plt
xx = np.linspace(0, 10, 50)
```

```
plt.plot(x, y, '.k', markersize=3)
plt.plot(xx, F(xx), color="firebrick")
```



Vi kan se at vi har fittet funktionen rimelig godt. Tilmed tjekker vi også vores funktion

```
F(4.17)
```

```
np.float64(2.741935172162666)
```

Som var det vi forventede. Det er ikke en god ide at bruge basisfunktionen $F(x) = c_0 + c_1x + c_2sin^2(x) + c_3cos^2(x)$, det er grundet kravene til basisfunktionerne. Den er ikke velegnet til at beskrive data da funktionerne $sin^2(x) + cos^2(x) = 1$ dermed er de lineært afhængige.

1.2 Trigonometrisk fit

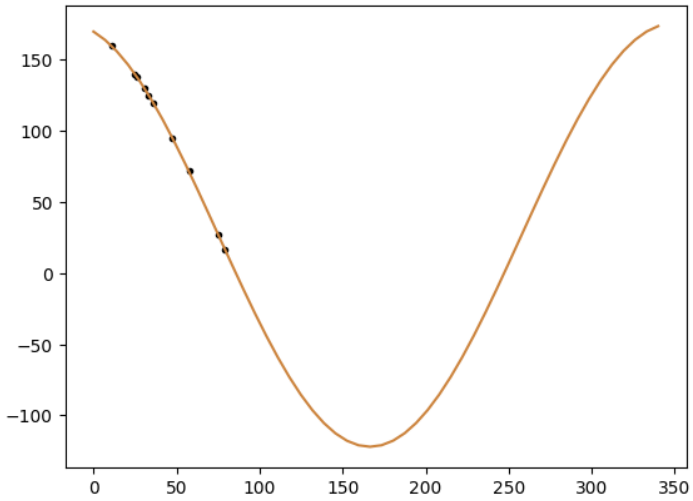
Vi øver brugen af et fit til at forudsige opførslen af solopgangen. Vi er givet basisfunktionen $F(x) = c_0 + c_1sin(\omega x) + c_2cos(\omega x)$, hvor $\omega = 2\pi/365$. Således har vi at perioden for F er 365 dage. Vi beregner mindste kvadrat således

```
x = np.array([11, 25, 26, 31, 33, 36, 47, 58, 75, 79])
y = np.array([160, 140, 138, 130, 125, 120, 95, 72, 27, 17])
A = np.array([np.cos((2*np.pi)/365 * x), np.sin((2*np.pi)/365 * x),
np.ones(10)]).T
```

Nu kan vi plotte vores solopgangs graf, hvor x er nr. dag fra 1. januar, og y er tidspunktet for solopgang i minutter efter kl. 6:00.

```
sol = np.linalg.solve(A.T @ A, A.T @ y)
def Fn(x):
    return sol[0]*np.cos((2*np.pi)/365 * x) +
    sol[1]*np.sin((2*np.pi)/365 * x) + sol[2]
```

```
xx = np.linspace(0, 340, 50)
plt.plot(x, y, '.k')
plt.plot(xx, Fn(xx), '-', color="peru")
```



Jeg har sat plottet til at vise vores forudsigelse i fremtiden. Vi forudsiger f.eks. at solen står tidligst op ved at minimerer funktionen således

```
min = 999
s = 0
for i in range(80, 366):
    if Fn(i) < min:
        min = Fn(i)
        s = i
min, s
```

```
(-122.0005506985046, 166)
```

```
6 + min/60
```

```
np.float64(3.9666574883582566)
```

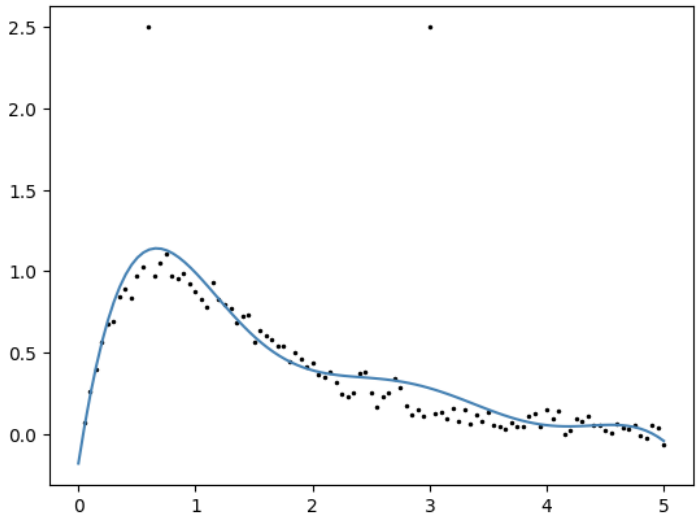
Dvs. omkring kl. 4 står solen op tidligst. Det passer da nogenlunde med hvad jeg forventede. Det er desuden d. 10 juni den har målt til den tidligste dag, hvilket ikke er helt korrekt, men en god approximation.

1.3 Håndtering af "grovfejl" i data

Af og til kan man opleve at der er grove fejl i datasæt, og det gør, at ens linear squares approximering bliver haltet meget til siden, på grund af et par få grove fejl (f.eks. pga. instrumentet eller andet). Her er en metode hvorpå man kan fjerne sådanne fejl.

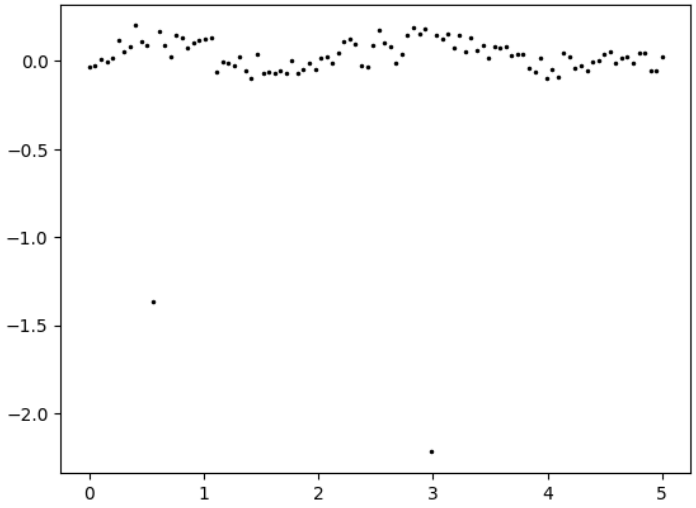
Først fitter vi et polynomium af grad 8 således

```
Grovdata = io.loadmat('Grovdata')
x = np.ravel(Grovdata['x'])
y = np.ravel(Grovdata['y'])

c = np.polyfit(x, y, 8)
xx = np.linspace(0, 5, 100)
def F(x):
    return np.polyval(c, x)

plt.plot(x, y, '.k',markersize=3)
plt.plot(xx, F(xx), '-', color="steelblue")
```



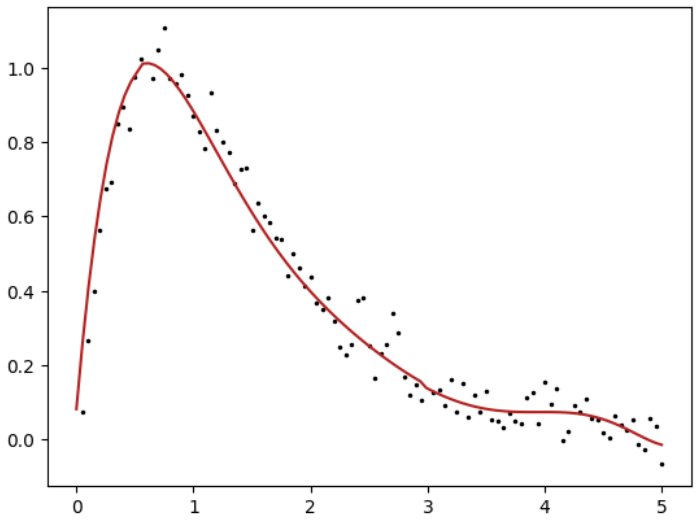
Her kan vi se at der specifikt er 2 punkter som er grove, og som gør at vores approximering er højere end forventet. Dette vil vi gerne fjerne. Først plottes fejlene således

```
err = np.polyval(c, x) - y
plt.plot(xx, err, '.k', markersize=3)
```



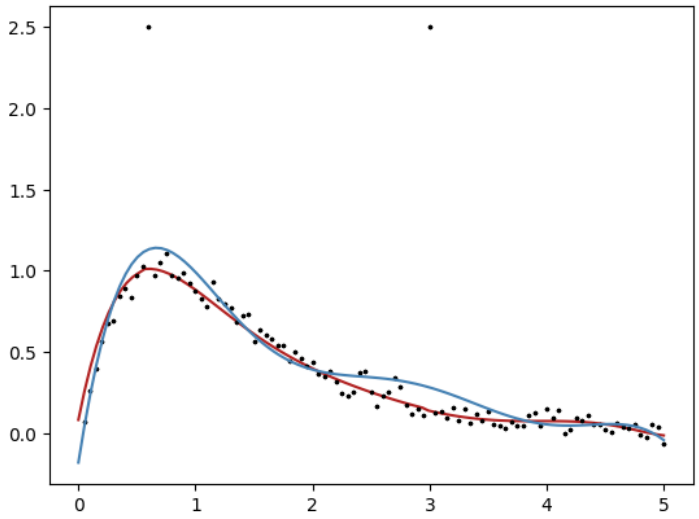
Nu kan vi fjerne fejlene vha. følgende metode. Beregn først gennemsnittet μ for fejlene. Dernæst findes de index n for hvilke der gælder, at $|e_n| > 5\mu$. Derefter kan vi gå gennem de fejl og fjerne dets index fra vores andre fejl.

```
me = np.mean(abs(err))
N = np.where(abs(err) > 5*me)[0]
x, y = np.delete(x,N), np.delete(y,N)
p = np.polyfit(x, y, 8)
fit = np.polyval(p, x)
e = np.polyval(p, x) - y
xx = np.linspace(0, 5, 98)
plt.plot(x, y, '.k', markersize=3)
plt.plot(xx, fit, '-', color="firebrick")
```



Hermed har vi også fittet polynomium af grad 8 til de resterende data, med de fjernede grovfejl. Vi kan se at det nye fit er bedre end det gamle således

```
xx = np.linspace(0, 5, 98)
plt.plot(x, y, '.k', markersize=3)
plt.plot(xx, fit, '-', color="firebrick")

plt.plot(x, y, '.k',markersize=3)
plt.plot(xx, F(xx), '-', color="steelblue")
```



Her kan vi se at det nye fit (rødt), passer bedre end det blå. Vi kan se at det blå går opad hvor de har de grovfejl.

1.4 Datafitting med to forskellige baser for andengradspolynomier

Vi har basisfunktionen $F_A(x) = a_0 + a_1x + a_2x^2$. Vi finder koefficientmatricen A og beregner konditionstallet således

```
x = np.linspace(0, 1, 200)
A = np.array([x**2, x, np.ones(200)]).T
condA = np.linalg.cond(A.T @ A)
condA
```

```
np.float64(514.1654995067415)
```

Vi kan se at konditionstallet er meget højt.

Derfor tjekkes Bernstein basis om den er bedre at bruge til dette. Vi tjekker om Bernstein-basen er lineært uafhængig.

Vi kan omformere dem til standard basis således

$$[B_0^2]_{\mathcal{S}_2} = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}, \quad [B_1^2]_{\mathcal{S}_2} = \begin{pmatrix} 0 \\ 2 \\ -2 \end{pmatrix}, \quad [B_2^2]_{\mathcal{S}_2} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Hermed indsættes i python for RREF

```
b0 = Matrix([1, -2, 1])
b1 = Matrix([0, 2, -2])
b2 = Matrix([0, 0, 1])
B = Matrix.hstack(b0, b1, b2)
display(B.rref(pivots=False))
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Dermed kan vi se at Bernstein basis er lineært uafhængig.

Nu kan vi dermed bruge Bernstein basis som matrix B og beregne konditionstallet således

```
B = np.array([x**2, 2*x*(1-x), (x-1)**2]).T
condB = np.linalg.cond(B.T @ B)
condB
```

```
np.float64(9.802995981006045)
```

Givet at det er konditionstallene som betyder mest for den numeriske beregning, må vi konkludere at det er Bernstein-basen B som er bedst at bruge, da det har den mindste konditionstal og dermed mindste relative fejl.

Vi ser på graferne, at B matricen er mere fleksibel, i den forstand at den har kurvet som minder om x^2 andengrads kurver, mens A ikke har det i samme kapacitet.

2 Algoritme til bestemmelse af et nulpunkt for en funktion af én variabel

2.1 Implementering og brug af Newtons algoritme

Vi skriver her en Python-funktion der implementerer Newtons algoritme baseret på lærebogen. Vi implementerer ikke stopkriterierne, men udfører nmax iterationer i stedet.

```
def Newton(f, df, x0, nmax=5):
    x = x0
    X = [x0]
    for n in range(1, nmax+1):
        fx = f(x)
        fp = df(x)
        x = x - fx/fp
        X.append(x)
    return X
```

Vi bruger funktionen på et eksempel i bogen s. 128 med funktionen $f(x) = ((x-2)x + 1)x - 3$ således

```
x = symbols('x')
f = lambda x : ((x-2)*x + 1)*x-3
df = lambda x : (3*x-4)*x+1
```

```
x0 = 3
nmax = 5
xn = Newton(f,df,x0,nmax)
np.set_printoptions(suppress=True, precision=15)
xn
```

```
[3, 2.4375, 2.2130327163151096, 2.175554938721488, 2.174560100666446, 2.174559
4102933126]
```

Vi har fået de samme resultater som i bogen side 128 med minimal afvigelse i de sidste cifre.