MAT4110 – Oblig 1

Jonas Gahr Sturtzel Lunde (`jonassl`)

September 25, 2019

Exercise 1

From lecture notes 4, we know that the least squared solution to the linear system

$$A\boldsymbol{x} = \boldsymbol{b}$$

can be solved by QR factorization of A. We have that

$$||A\boldsymbol{x} - \boldsymbol{b}||^2 = ||R\boldsymbol{x} - Q_T\boldsymbol{b}||^2$$

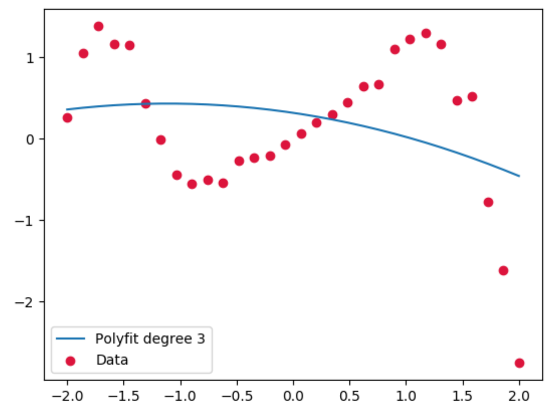
Further, let R_1 be the $m \times m$ upper triagonal half of R, and $\boldsymbol{c}_1$ be the upper m elements of $\boldsymbol{c}$, where $\boldsymbol{c} = Q^T\boldsymbol{b}$. The lecture notes show that the least squared problem is equivalent to

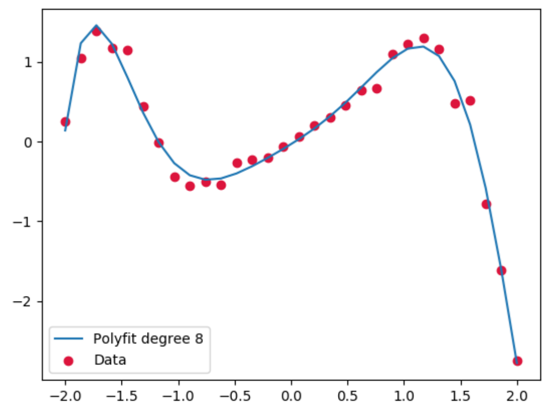
$$R_1\boldsymbol{x} = \boldsymbol{c}_1$$

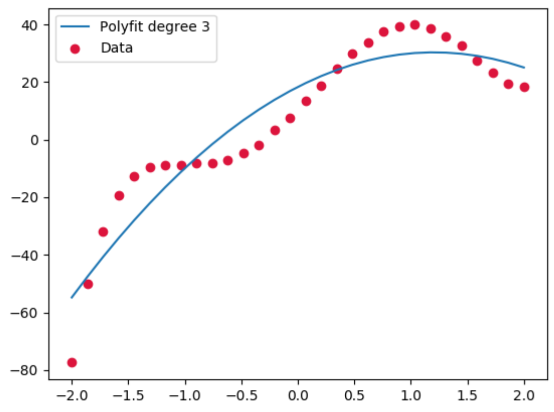
where R_1 is upper triagonal.

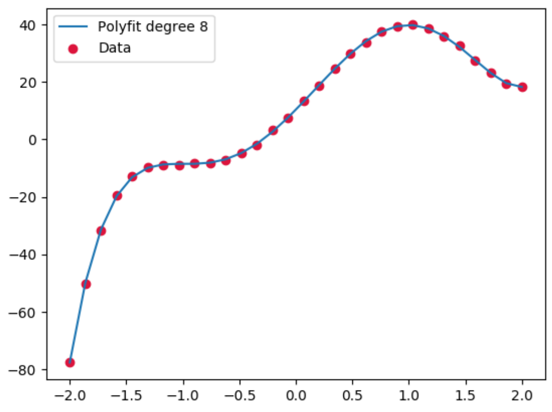
This has been implemented in Python, in the file QR.py, and run in the file oblig1.py. To avoid confusion with the x-axis named $\boldsymbol{x}$, the coefficient vector is named $\boldsymbol{\beta}$.

Below we see the results for the first dataset at the top, and the second below. m=3 is shown in the left, and m=8 on the right.









Exercise 2

The normal equation should solve the least squared problem for some $\boldsymbol{x}$, and is given by

$$A^T A\boldsymbol{x} = A^T\boldsymbol{b}$$

Recognizing that $B = A^T A$ is a symmetric, positive definite matrix, gives us the QR factorization $RR^T = B$. Inserting gives

$$RR^T\boldsymbol{x} = A^T\boldsymbol{b}$$

A^T and $\boldsymbol{b}$ are known, and multiplying them gives

$$RR^T\boldsymbol{x} = \boldsymbol{w}$$

where $\boldsymbol{w} = A^T\boldsymbol{b}$. This can be split into the two equations

$$R\boldsymbol{z} = \boldsymbol{w}$$

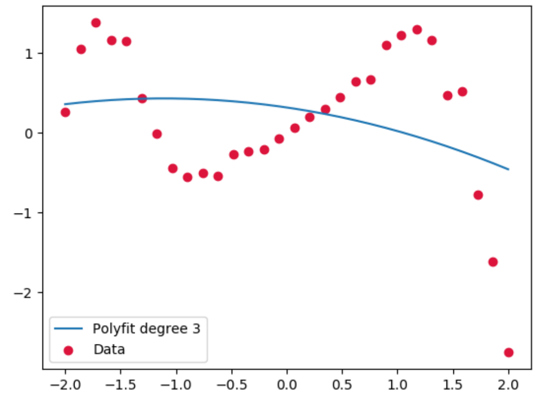
and, having solved for z using backward substitution,

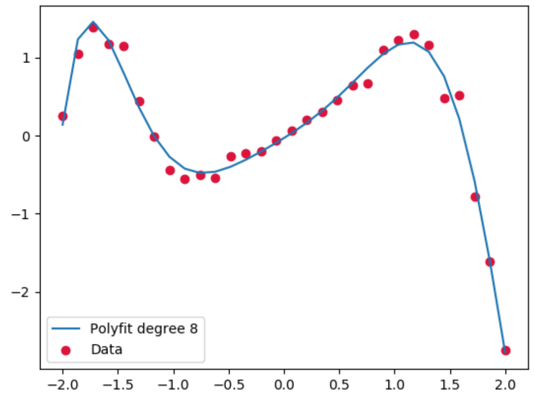
$$R^T\boldsymbol{x} = \boldsymbol{z}$$

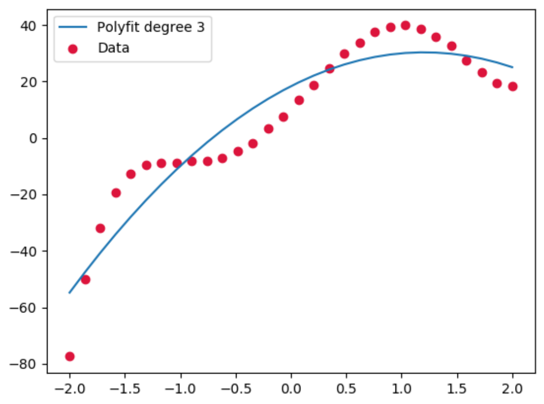
which can be solved using forward substitution.

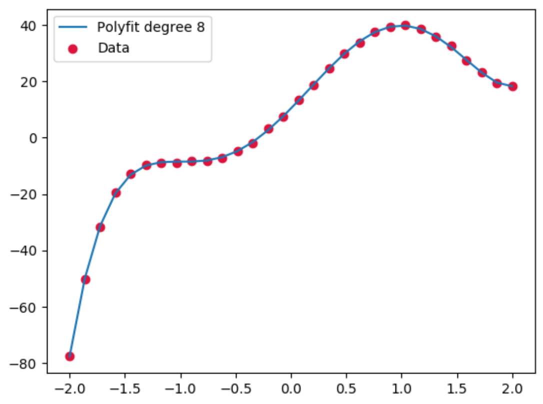
This was implemented in Python, in the file Choleksy.py, and run in the oblig1.py file.

Below we see the results for the first dataset at the top, and the second below. m=3 is shown in the left, and m=8 on the right. They look identical to the ones we got from the QR factorization.







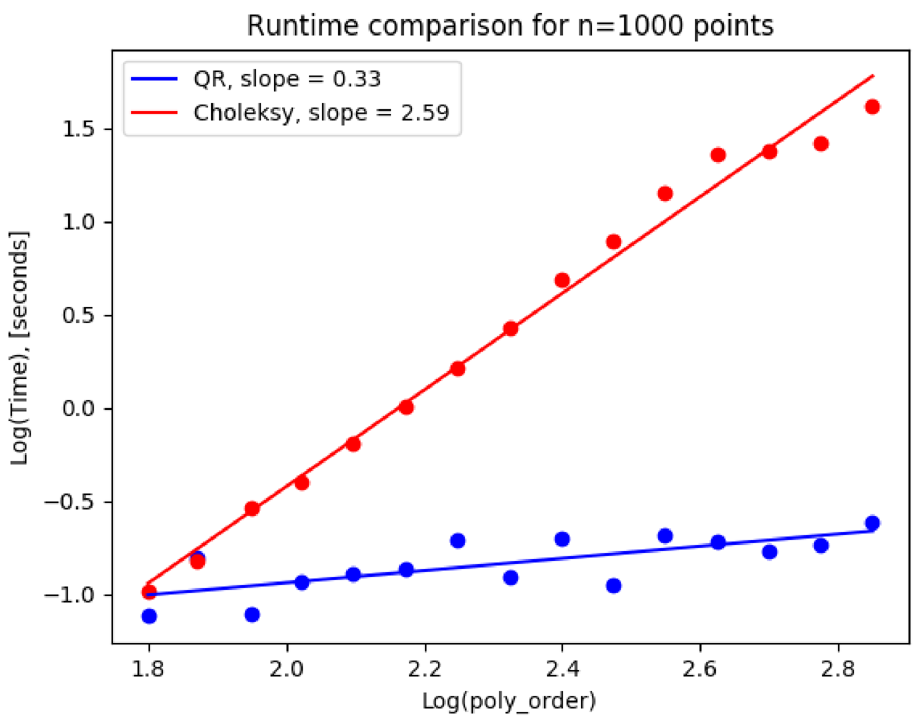


Exercise 3

Cholesky factorization involves taking the square root of quantities in A. This makes the algorithm especially sensitive to ill-conditioned matrices, as the square root has a very high loss of precision, compared to other operation.

I couldn't find any good sources on the behavior of QR on ill-conditioned matrices, but it uses Gram-Schmidt, which doesn't contain square roots, so I would imagine it plays more nicely.

The QR solver also seems to scale vastly better, in time, for larger polynomial degrees. Holding the number of points constant at $n = 1000$, and increasing the polynomial order, we see from the plot below that Choleksy is much slower for large m, seemingly scaling as $m^{2.59}$ to QRs $m^{0.33}$ Granted, this doesn't look at the scaling with increased number of points.



Appendix: Code

Listing 1: oblig1.py

```
import time
import numpy as np
import matplotlib.pyplot as plt
from tools import solve_lower_triangular, solve_upper_triangular
from Cholesky import least_squares_cholesky
from QR import least_squares_QR
np.random.seed(42)

### Setting up the data.
n = 30
start = -2
stop = 2
x = np.linspace(start, stop, n)
eps = 1

r1 = np.random.uniform(0, 1, n)*eps
r2 = np.random.uniform(0, 1, n)*eps

y1 = x*(np.cos(r1 + 0.5*x**3) + np.sin(0.5*x**3))
y2 = 4*x**5 - 5*x**4 - 20*x**3 + 10*x**2 + 40*x + 10 + r2

## Running the two solvers for m=3 and m=8, and plotting the solutions.
for m in [3, 8]:
    A = np.zeros((n, m))
    for i in range(m):
        A[:,i] = x**i

    for y_nr in range(2):
        y = [y1, y2][y_nr]

        beta = least_squares_cholesky(A, y)
        plt.scatter(x, y, c="crimson", label="Data")
        result = A@beta
        plt.plot(x, result, label=f"Polyfit␣degree␣{m}")
        plt.legend()
        plt.savefig(f"../plots/cholesky_y{y_nr}_m={m}.png")
        plt.clf()

        beta = least_squares_QR(A, y)
        plt.scatter(x, y, c="crimson", label="Data")
        result = A@beta
        plt.plot(x, result, label=f"Polyfit␣degree␣{m}")
        plt.legend()
        plt.savefig(f"../plots/QR_y{y_nr}_m={m}.png")
        plt.clf()



# Doing a timing of the two solvers for n=1000 over different ms.
n = 1000
start = -2
stop = 2
x = np.linspace(start, stop, n)
eps = 1
r = np.random.uniform(0, 1, n)*eps
y = 4*x**5 - 5*x**4 - 20*x**3 + 10*x**2 + 40*x + 10 + r

time_QR = []
time_cholesky = []
```

```
error_QR = []
error_cholesky = []

m_list = np.logspace(1.8, 2.85, 15, dtype=int)
for m in m_list:
    print(m)
    A = np.zeros((n, m))
    for i in range(m):
        A[:, i] = x**i

    t0 = time.time()
    least_squares_cholesky(A, y)
    time_cholesky.append(time.time() - t0)

    t0 = time.time()
    least_squares_QR(A, y)
    time_QR.append(time.time() - t0)


time_QR = np.log10(np.array(time_QR))
time_cholesky = np.log10(np.array(time_cholesky))
m_log = np.log10(np.array(m_list))
plt.plot(m_log, time_QR, "bo")
plt.plot(m_log, time_cholesky, "ro")
c1, c0 = np.polyfit(m_log, time_QR, 1)
plt.plot(m_log, c0 + m_log*c1, label=f"QR,␣slope␣=␣{c1:.2f}", c="b")
c1, c0 = np.polyfit(m_log, time_cholesky, 1)
plt.plot(m_log, c0 + m_log*c1, label=f"Choleksy,␣slope␣=␣{c1:.2f}", c="r")
plt.legend()
plt.title("Runtime␣comparison␣for␣n=1000␣points")
plt.ylabel("Log(Time),␣[seconds]")
plt.xlabel("Log(poly_order)")
plt.savefig("../plots/timings.png")
```

Listing 2: QR.py

```
import numpy as np
from tools import solve_lower_triangular, solve_upper_triangular


def least_squares_QR(A, y):
    # Solves the least squares problem Ax = b, for the x that minimizes ||Ax - b||_2, usin
    m = A.shape[1]
    Q, R = np.linalg.qr(A, mode="complete")  # mode=complete to get complete QR, without r
    R1 = R[:m, :m]
    c = Q.T@y
    c1 = c[:m]
    n2 = c1.size
    x = solve_upper_triangular(R1, c1)
    return x
```

Listing 3: cholesky.py

```
import numpy as np
import matplotlib.pyplot as plt
from tools import solve_lower_triangular, solve_upper_triangular
np.random.seed(42)


def cholesky(A):
    # Performs a Cholesky factorization A = L*L^T of a matrix A.
    n = A.shape[0]
    L = np.array([[0.0] * n for i in range(n)])
    for i in range(n):
        for k in range(i+1):
            tmp_sum = sum(L[i][j] * L[k][j] for j in range(k))

            if (i == k):
                L[i][k] = np.sqrt(A[i][i] - tmp_sum)
            else:
                L[i][k] = (1.0 / L[k][k] * (A[i][k] - tmp_sum))
    return L


def least_squares_cholesky(A, y):
    # Solves the least squares problem Ax = b, for the x that minimizes ||Ax - b||_2, usin
    R = cholesky(A.T@A)
    RT = R.T
    w = A.T@y
    z = solve_lower_triangular(R, w)
    x = solve_upper_triangular(RT, z)
    return x
```

Listing 4: tools.py

```
import numpy as np

def solve_upper_triangular(A, y):
    ## Solves the matrix equation Ax = y for an upper triangular A using backward substitu
    N = A.shape[0]
    x = np.zeros(N)
    x[N-1] = y[N-1]/A[N-1,N-1]
    for i in range(N-1, -1, -1):
        tmp = y[i]
        for j in range(N-1, i, -1):
            tmp -= x[j]*A[i,j]
        x[i] = tmp/A[i,i]
    return x

def solve_lower_triangular(A, y):
    ## Solves the matrix equation Ax = y for a lower triangular A using forward substituti
    N = A.shape[0]
    x = np.zeros(N)
    x[0] = y[0]/A[0,0]
    for i in range(1, N, 1):
        tmp = y[i]
        for j in range(0, i, 1):
            tmp -= x[j]*A[i,j]
        x[i] = tmp/A[i,i]
    return x
```