MAT4110 - oblig 1

Halvard Sutterud

September 2018

Exercise 1

The QR-factorization decomposes a $n \times m$ matrix A (with $n \geq m$)by

$$A = QR, \tag{1}$$

where $Q \in \mathbb{R}^{n \times n}$ is orthogonal and $R \in \mathbb{R}^{n \times m}$ is a matrix

$$R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}, \tag{2}$$

where $R_1 \in$ is upper triangular. The linear problem can then be written as

$$Ax = b \Rightarrow Rx = Q^{-1}b = Q^T b \tag{3}$$

since Q is orthogonal. As R has $n - m$ rows consisting of zeros, these will not contribute, and we simplify the problem further

$$R_1 x = c_1. \tag{4}$$

Here c_1 comes from the decomposition $Q^T b = [c_1, c_2]$, where c_1 has length m and c_2 has length $n - m$. As R_1 is upper triangular, we can solve eq. (4) by simple backwards substitution,

$$\begin{aligned} x_m &= \frac{c_m}{R_{m,m}} \\ x_i &= \frac{c_i - \sum_{j=i+1}^{m} R_{i,j} x_j}{R_{i,i}} \quad \text{for} \quad i < m. \end{aligned}$$

This is implemented for our design matrix X in the function `ex1` in the python code, as can be seen in listing 1.

Exercise 2

The cholesky factorization is a factorization of a symmetric $n \times n$ matrix $A = LDL^T$, where L is lower triangular, and D is diagonal. In the case that A is positive definite (which it is in our case, as $A = X^T X$), this can be rewritten

$$A = RR^T, \tag{5}$$

if one writes $R = LD^{1/2}$, with $D^{1/2}$ as a matrix with the square root of the diagonal elements of D along the diagonal, i.e. $(D^{1/2})_{i,i} = \sqrt{D_{i,i}}$. We can then rewrite

This is implemented in the function `ex2` in the python code, as can be seen in listing 1.

Exercise 3

The condition number when solving a linear system $Ax = b$ is given in the lecture notes as the ratio of the largest singular value and the smallest,

$$K_2(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} \tag{6}$$

where K_2 indicates that we are using the 2-norm. In the QR-algorithm, we solve the matrix equation $R_1 x = c_1$, with $c_1 = Q_1 b$, and R_1 and Q_1 as the so-called economic matrices of the QR-factorization (as provided by numpy). Applying the SVD-tranformation and calculating the K2-conditioning, we get $K_2(R_1) = 58.95$ (for the given seed).

In the case of the cholesky solver, the problem solved is

$$RR^T x = A^T b \tag{7}$$

.

We solve it by two separate problems, $R^T x = y$ and $Ry = A^T b$. However, it can be shown that (left as an exercise to the reader)

$$K_2(RR^T) = K_2(R)K_2(R^T) = K_2(R)^2, \tag{8}$$

which is good, because the conditioning should be independent of the way we solve the problem (eq. (8)). Applying this to the given seed, we get $K_2(R)^2 = (58.95)^2 = 3475.16$. We observe that the conditioning of the original problem can be increased by increasing the number of times we solve a linear system.

Listing 1: The file `oblig1.py`

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
import argparse
from textwrap import wrap

def data1(plot = True):
    n = 30
    start = -2
    stop = 2
    x = np.linspace(start,stop,n)
    eps = 1
    np.random.random(1)
    r = np.random.random(n) * eps
    y = x*(np.cos(r+0.5*x**3)+np.sin(0.5*x**3))
    if plot:
        plt.plot(x,y,'o')
        plt.show()
    return x,y

def data2(plot = True):
    n = 30
    start = -2
    stop = 2
    x = np.linspace(start,stop,n)
    eps = 1
    # rng(1)
    r = np.random.random(n) * eps
    y = 4*x**5 - 5*x**4 - 20*x**3 + 10*x**2 + 40*x + 10 + r
    if plot:
        plt.plot(x,y,'o')
```

```
        plt.show()
    return x,y

def QR_solve(A,b):
    """Solves Ax = b with QR-factorization"""

    # with mode = 'economic', R==R1
    Q,R = linalg.qr(A, mode = 'economic')
    M = R.shape[1]
    R1 = R
    c = Q.T @ b
    c1, c2 = c[:M], c[M:]

    x = backward(R1,c1)
    return x

def backward(U,b):
    M = U.shape[0]
    x = np.zeros(M)
    x[-1] = b[-1] / U[-1,-1]
    for i in range(2,M+1):
        x[-i] = (b[-i] - np.sum(U[-i,-i+1:]*x[-i+1:])) / U[-i,-i]
    return x

def forward(L,b):
    M = L.shape[0]
    x = np.zeros(M)
    print(L.shape, b.shape)

    x[0] = b[0]/L[0,0]
    for i in range(1,M):
        x[i] = (b[i] - np.sum(L[i,:i-1]*x[:i-1])) / L[i,i]
    return x


def ex1(args):
    print('ex1')
    x,y = data1(plot=False)
    deg = 6
    X = np.array([x**i for i in range(deg+1)]).T

    beta = QR_solve(X,y)
    plt.plot(x,y,'o', label='data')
    plt.plot(x,X@beta,'-', label = 'order {}'.format(deg))
    plt.legend()
    plt.show()

def ex2(args):
    print('ex2')
    x,y = data1(plot=False)
    deg = 5
    X = np.array([x**i for i in range(deg+1)]).T
    B = X.T @ X
    beta = cholesky_solve(X, y)
    plt.plot(x,y,'o', label='data')
    plt.plot(x,X@beta,'-', label = 'order {}'.format(deg))
    plt.legend()
    plt.show()
```

```python
def cholesky_solve(A,b):
    """Solves Ax=b through normal equations A.T Ax = A.T b, using cholesky
    factorization, solving R y = A.T b with forward sub and then R.T x = y
        """
    # Solve
    # Solve R^T x = y
    # remember, R is lower diag
    R = cholesky(A.T@A)
    print(A.shape)
    x = backward(R.T, forward(R, A.T @ b))
    return x



def cholesky(A, RR = True):
    Ak = A.copy()
    n = A.shape[0]
    L = np.zeros(Ak.shape)
    D = np.zeros(Ak.shape[0])
    for k in range(n):
        L[:,k] = Ak[:,k]
        D[k]   = Ak[k,k]
        L[:,k] = L[:,k]/D[k]
        Ak -= D[k]*np.outer(L[:,k], L[:,k])
    if RR:
        R = L  * np.sqrt(D)
        return R
    else:
        return L, np.diag(D)

def K2_condition(A):
    '''Calculates the K-2 condition of solving a linear system Ax = b using
    the singular values of A'''

    svd = linalg.svd(A)
    sing_vals = svd[1]
    return (np.max(sing_vals) / np.min(sing_vals))

def ex3(args):
    x,y = data1(plot=False)
    deg = 5
    X = np.array([x**i for i in range(deg+1)]).T

    Q,R = linalg.qr(X, mode = 'economic')
    #svd = linalg.svd()
    Rchol = cholesky(X.T@X)
    print(K2_condition(Rchol))
    print(K2_condition(Rchol.T))
    print(K2_condition(Rchol.T)**2)
    print(K2_condition(Rchol @ Rchol.T))

    print(K2_condition(X))

    print(R.shape)
    # K2_condition(R)
```

```
def main(args):
    """Either runs all parts or just one"""
    parts = {1:ex1, 2: ex2, 3:ex3}
    if args.part == 0:
        ex1(args)
        ex2(args)
        ex3(args)
    else:
        parts[args.part](args)

def get_args():
    parser = argparse.ArgumentParser()

    parser.add_argument('-p','--part', type=int,default=0,
                        choices = [0,1,2,3])
    return parser.parse_args()

if __name__=='__main__':
    args = get_args()
    main(args)
```