Assignment 3

Vilhelm Roxling and Axel Goteman

September 24, 2014

Task 1

In A.1 the code for our householder method in the orthogonalization class. The variable both is determining wether or not to compute the matrix Q in the factorization.

It works for full rank matrices, rectangular and square up to a size of at least 1000. As opposed to the gramschmidt method it seems stable. When compared to scipy.linalg's qr method it is significantly slower for small matrices, but the relative time difference actually decreases with larger matrices. This indicates the methods should have almost the same time complexity, although our method uses some significantly slower operations.

To optimize the method we did the following:

In the first for-loop the upper triangular matrix is constructed, and the vectors v are found. When multiplying Q_i and R, we utilized the fact that this operation can be written as $Q_iR = (Q_iR_1Q_iR_2 \dots Q_iR_n)$, and that multiplication with Q_i only changes a part of R or the vectors R_i. We also used that a vector-matrix multiplication with a householder matrix can be done with a scalar product.

When or if the Q matrix is constructed similar properties makes similar optimizations possible. We use the fact that $Q = QI = (Qe_1Qe_2 \dots Qe_m)$ (if producing full factorization) and that $Q = Q_1Q_2 \dots Q_n$. First we used the method qdot that makes a multiplication of Q with an arbitrary vector, but then realized that a lot of operations could be skipped if we avoid multiplications where only zeros are involved. This is made in the last, rather unreadable for-loop.

Task 2

The code for the rotation method in the orthogonalization class can be seen in A.2 and is called givens. It works fine for full rank matrices, square as well as rectangular. This method we invented on our own. The idea we used is described below:

The goal is the same as in the householder case, we want to multiply A from the left with a matrix that makes a certain column of A having zeros below the main diagonal. But now we want to accomplish this using rotation matrices instead. We started with the fact that in two dimensions we rotate the vector $v = (v_x, v_y)$ to the x-axis by multiplying it with the matrix

$$J = \begin{pmatrix} cos\theta & sin\theta \\ sin\theta & -cos\theta \end{pmatrix} = \begin{pmatrix} \frac{v_x}{r} & \frac{v_y}{r} \\ \frac{v_y}{r} & -\frac{v_x}{r} \end{pmatrix} \tag{1}$$

where θ is the angle between the x-axis and v and $r = \sqrt{v_x^2 + v_y^2}$ is the length of v. This would change v to $v' = (r, o)$. Now by letting G_1^1 be the identity matrix with J at the lower right corner and multiplying G_1^1 with A from the left, v would become the last two elements of the first column of A. Now by instead placing a new J one step up along the diagonal in G_2^1, v becomes the second and third to last elements, $v = (A_{0,m-2}, r)$, where r is the length of the previous "v". Using this repeatedly we can obtain the previously mentioned goal. With each rotation all the columns are affected, but only two rows. To achieve the same result for the next column, the same method is used, only one step less. This wont affect the first column which is zero. This is then repeated until the matrix is upper triangular. Note that all the G_i^k are orthogonal matrices.

Implementing this in the naive way would results in a ridiculous number of matrix multiplications, making it near impossible to use for large matrices. The time complexity would be $o(n^5)$ for a square matrix. Fortunately some optimizations are possible:

To begin with, as mentioned, multiplication with G_i^k only affects rows i and $i + 1$. Thus a multiplication of $o(n^3)$ can be reduced to the $o(n)$ multiplication of a $2x2$ matrix (J) and a $2xn$ matrix. When creating R we now that the columns 1 to n will be zeroed at the affected rows. Thus the $2xn$ matrix can be reduced to a $2xn - i$ matrix.

The method is then reduced $o(n^3)$. Compared to the householder, it has the same time complexity, but is 2-3 times slower. Mostly because of some slow operations (square root and semi-large matrix multiplications).

A Appendix

A.1 Householder

```
def householder(self, both = True):
        R = self.A.copy()
        self.v_list = []

        for i in range(self.n):
            a = R[i][i:]
            e = zeros(self.m-i)
            e[0] = np.linalg.norm(a)
            v = np.sign(a[0])*e + a
            v = v/np.linalg.norm(v)
            self.v_list.append(v)
            R[i,i:] = -sign(a[0])*e
            for j in range(1,self.n-i):
                R[i+j,i:] -= 2*v*np.dot(v, R[i+j,i:])
        if not both:
            return R, self.v_list

        Q = zeros((self.m, self.m))
        I = eye(self.m)

        for e, i in zip(I[:], range(self.m)):
            if self.n-1-i >= 0:
                k = self.n-1-i
                e[i:] -= 2*self.v_list[i][0]*self.v_list[i]
            else:
                k = 0
                e[self.n-1:] -= 2*self.v_list[-1][i-(self.n-1)]*self.v_list[-1]

            for j in range(k+1, self.n):
                v = self.v_list[self.n-1-j]
                e[self.n-1-j:] -= 2*np.dot(v, e[self.n-1-j:])*v

            Q[i] = e

        return Q.T, R.T

    def qdot(self, b):

        for i in range(self.n):
            v = self.v_list[self.n-1-i]
            b[self.n-1-i:] -= 2 * np.dot(v, b[self.n-1-i:])*v

        return b
```

A.2 Given's

```
    def givens(self, both = True):

        R = self.A.copy()
        Q = eye(self.m)

        for i in range(self.n):

            if i == self.m-1:
                break

            x1 = R[i, -2]
            x2 = R[i, -1]
            r = np.sqrt(x1**2+x2**2)
            c = x1/r
            s = x2/r
            J = np.array([[c, s], [s, -c]])
            R[:,-2:] = np.dot(R[:,-2:], J.T)
            if both:
```

```
            Q[-2:,:] = np.dot(J, Q[-2:,:])

        for j in range(3, self.m-i+1):
            x1 = R[i, -j]
            x2 = r
            r = sqrt(x1**2+x2**2)
            c = x1/r
            s = x2/r
            J = np.array([[c, s], [s, -c]])
            R[i:,-j:-j+2] = np.dot(R[i:,-j:-j+2], J.T)
            if both:
                Q[-j:-j+2,:] = np.dot(J,Q[-j:-j+2,:])

    if not both:
        return R.T

    return Q.T, R.T
```