This is the coursework 1 of Computational linear algebra Luçjano Muhametaj CID 02168294 (GITHUB : https://github.com/Imperial-MATH96023/clacourse-2021-lm1621)

Excercise 1

a) Part 1

We can check easily that C is 1000x100. We can compute the QR factorisation to obtain Q and R using cla utils.
Now if we compute the norm of the rows of R we can easily observe that the norm of the first three rows is $\sim 10^2$, while the norm of the other ones is $\sim 10^{-9}$. We can see that the columns of C are a approximately a linear combination of the first three columns of Q (In other words we are saying that only the first three columns of Q are really 'relevant' (C lives in the span of these three columns)). In fact taken $C = QR$ we will have

$$C_{ij} = Q_{ik}R_{kj} = Q_{i1}R_{1j} + Q_{i2}R_{2j} + Q_{i3}R_{3j} + O(10^{-9}) \tag{1}$$

So considering that we have to do with time series we can probably say that from the forth measurement (column) all the columns are going to be almost linearly dependent to the first three columns.

b) Part 2

It is enough to take a 'compressed' Q_{com} and a 'compressed' R_{com} matrices (we take the first three columns of Q and the first three columns of R)). Then we can compute a product and obtain a compressed version of C, which will be called C_{com}
The code for this exercise can be found on `ex1.py`. I have also provided an automatic test `testex1.py`

c) Part 3

Taken $C = QR$, we can make the exercise more obvious by applying a permutation to R in order to reorder its columns from the one with the highest diagonal value (in norm) to the one with the lowest. (In other words $|r_{11}| > |r_{22}| > |r_{33}| > ...$) Then we can compute a compression considering only the columns with relevant $|r_{ii}|$.

Excercise 2

a) Part 1

We want to construct a 12 degree polynomial interpolating the given points with the values $f_i = 1$ when $i = 0$ and $i = 50$ and $f_i = 0$ otherwise. We can do it using a QR factorisation of the Vandermonde matrix (that we can implement using the builtin numpy function). In particular we want to find the coefficients of this polynomial, which will be stored in β (called B in the code). To find this β we have to solve $R\beta = Q^T y$. In particular we can compute different QR factorisations to obtain the final result.
For interpolation using GS-classical run `cw1.2.2.py`, using GS-modified `cw1.2.3.py`, using Householder `cw1.2.4.py` and using the numpy built in function `cw1.2.1.py`. In any case all the codes for these algorithms are presentet as functions in the file `ex2.py`.

b) Part 2

We can solve the point a) using GS modified, GS classical, Householder and also using the numpy factorisation function. What we can see is that using GS modified the difference is really small and the coefficients we get are even really close to the ones we get using the numpy builtin qr factorisation. (In particulare we get these coefficients: [9.41937158e-01, -5.93211603e+01, 1.23068470e+03, -1.18303373e+04, 5.84557018e+04, -1.35379152e+05, -1.25373712e+04, 9.42390600e+05, -2.67375407e+06, 3.85446609e+06, -3.17819513e+06, 1.42614680e+06, -2.70935308e+05]). On the other hand if we use GS classical we can easily see that the difference is much bigger. This is a consequence of the instability of GS classical. In fact the columns of Q (obtained with GS classical) are not quite orthogonal. We can see this for example by running the product Q*Q and seeing that it is not so close to I.

Excercise 3

a) Part 1

We need reduced QR factorisation. The inefficient exercise is the one implementing the householder QR factorisation. In fact that algorithm implements many implicit multiplications. In particular for n steps we are multiplying a mxm matrix with an $mx(m+n)$ matrix. In particular the first one is the matrix Q_i for each step and the second one is the extended matrix we obtained 'appending' I to the matrix A (we can write as (A I)).

b) Part 2

To do this I can just store the v vectors in the lower triangular complementary matrix of R (in these places R is zero). In fact v vectors are decreasing in size at each step. However for each column k of the matrix we have to insert a (m-k)-long vector and for each column we have only m-k-1 spaces left for column. So we have to adapt the vectors v rescaling them in order to make the element of the diagonal of R relevant for the v vectors. To do this we can just add to the loop of the householder function some lines which for each steps do $A_{k:m,k} = \frac{A_{k,k}}{v_1} v$.

c) Part 3

We can just add to the loop of the householder function some lines which for each steps do $A_{k:m,k} = \frac{A_{k,k}}{v_1} v$. In this way we have R-v which contains in its 'upper triangular part' R and in the 'lower triangular' one the vectors v.
For the code check function `householderVR` in `ex3.py`
We can check that this works with the test `testex3.py`

d) Part 4

We want to write an algorithm for computing Q^*b using a provided R-v array, without explicitly forming Q.
We can just use the implicit multiplication by Q*, that we have when we use householder. In particular we extract the vectors v contained in R-v with a loop by implementing (using the slice notation) $v = R - v_{k:,k}$ (then we also normalise v) and then we can transform b in Q*b in place with $b_{k:} = b_{k:} - 2v(v^* b_{k:})$.

For the code see function `multiplicationQstarb.py` in `ex3.py`.
We can check that this works with the test `testex3.py`

e) Part 5

To solve this point let's observe that for what we have seen in chapter 2.7 this problem is equivalent to $\hat{R}x = \hat{Q}^*b$. Given R-v we can obtain $\hat{Q}^*b$ starting from the algorithm we wrote in the section before (and observing that since now we need to work on the reduced form we can just take the first n elements of Q^*b). Then to obtain x we can proceed in various way to solve the triangular system (in the code we will use the builtin scipy function). For the code see function `solvels` in `ex3.py`.
We can check that this works with the test `testex3.py`

Excercise 4

a) Part 1

We can reformulate the problem as finding the stationary point of the following:

$$\phi(x, \lambda) = f(x) + \lambda g(x) \tag{2}$$

which we can rewrite in this case as:

$$\phi(x, \lambda) = \|Ax - b\|^2 + \lambda(\|x\|^2 - 1) \tag{3}$$

b) Part 2

Assuming to know the correct value of λ we can compute

$$\frac{\partial \phi}{\partial x} = 2A^TAx - 2A^Tb + 2\lambda x \tag{4}$$

For $\frac{\partial \phi}{\partial x} = 0$ we can then find x:

$$x = (A^TA + \lambda I)^{-1}A^Tb. \tag{5}$$

c) Part 3

Now using the QR factorisation we can write:

$$x = (R^TQ^TQR + \lambda I)^{-1}R^TQ^Tb \tag{6}$$

and then

$$x = (R^TR + \lambda I)^{-1}R^TQ^Tb \tag{7}$$

Now we can observe that using the Woodbury matrix identity we can write this expression in a different way. We see that:

$$(\lambda I + R^TR)^{-1} = \frac{1}{\lambda}I - \frac{1}{\lambda^2}R^T(I + \frac{1}{\lambda}RR^T)^{-1}R \tag{8}$$

Then:

$$x = \frac{1}{\lambda}(I - \frac{1}{\lambda}R^T(I + \frac{1}{\lambda}RR^T)^{-1}R)R^TQ^Tb \tag{9}$$

Now we can call $c = R^T Q^T b$ and we will have:

$$x = \frac{c}{\lambda} - \frac{1}{\lambda^2} R^T (I + \frac{1}{\lambda} RR^T)^{-1} (Rc). \tag{10}$$

Now we can see that this notation brings some advantages for starting matrices A with dimension mxn such that n is much bigger than m. In fact we will have that the dimension of RR^T, which is mxm is going to be 'smaller' than the dimension of $R^T R$, which is nxn. So that in these circumstances working with $(I + \frac{1}{\lambda} RR^T)^{-1}$ is much more efficient than working with $(R^T R + \lambda I)^{-1}$ (we can divide the second expression to see they are really similar except from the fact that in the first one we have RR^T and in the second one $R^T R$). (We are practically saying that for $n >> m$ we can use Woodbury identity to invert easily) Under these circumstances we can write an algorithm based on the following steps: we write

$$(I + \frac{1}{\lambda} RR^T)^{-1} (Rc) = y \tag{11}$$

which will give us

$$(I + \frac{1}{\lambda} RR^T) y = Rc \tag{12}$$

Then we can use householder to find y and put it in the expression of x that we found using the Woodbury identity.
Otherwise if we don't have these conditions it's better to use the first expression of x that we found in point b of this excercise.
The code for this algorithm is in the function `solvewoodbury` of `ex4.py`
We can check that this works with the test `testex4.py`

d) Part 4

We can elaborate a binary search algorithm to find λ such that $\|x\| = 1$. We start from two points λ_1 and λ_2 and we check if there exist a solution to the problem (for example we can do it by checking if $(\lambda_1 - 1)(\lambda_2 - 1) < 1$). Then we just check the value of x for $\lambda^* = \frac{\lambda_1 + \lambda 2}{2}$ and check if it's close enough to 1 (we can decide *at priori* how much we want it near to 1). If yes we have found ours solution, otherwise I can repeat all this but substituting one of the starting point with λ^* depending from the value of x we got (smaller or bigger than 1). Then we go on this way until we find our solution.
In any case we can solve this problem both by using a recursive function and by using a loop. In the code we assume to have a solution in $[\lambda_1, \lambda_2]$.
This problem can be seen as a sort of root finding problem for $f(x) = \|x\| - 1$, and there are a lot of possible approaches that we can implement such as the newton method. Although the coursework requires a binary search which is the one that can be found in `ex4.py` in the function `findlambdaloop`. (In the code I have also added a prototype `findlambda` of a recursion version algorithm).
We can check that this works with the test `testex4.py`

e) Part 5

Given A an mxn for an algorithm based on recursion we can observe that even for big m and n the algorithm works (but the time increases with the dimension). On the other hand for the loop based code if we take big m or n the process takes too long.