The theory behind the QR algortihm and its different variations implemented in Python

This algorithm is solely based on the QR factorisation of matrices in $\mathbb{R}^{mxn}$ and it is one of the most modern methods for approximating the eigenvalues of a matrix. Throughout this method, although it is not necessary, we assume that the matrix that we are trying to approximate its eigenvalues is invertible so its QR decomposition is unique, if we also require the diagonal elements of the upper triangular matrix R such as $A = QR$ where Q is unitary (orthogonal if the matrix that we start with is real), that is $Q^* = Q^{-1}$, **to be positive**. Below we see the one practical way that we saw in our notes on how to find the QR factorisation of our starting matrix (the Gram-Schmidt way is not studied here due to its numerical instability - see Linear Algebra II notes on how that method works and the QR Decomposition using Givens Rotations is impractical and outshined by the Householder method discussed below), and then we see the 3 variations of the QR algorithm that we studies in our lecture notes.

But like we said, its important to note that the requirement that A is invertible is NOT essential - any matrix (even non square and even singular) can assume a QR decomposition - its just not going to be unique, up to the matrix R!

QR decomposition using Householder reflections

A Householder reflection is a matrix $P \in \mathbb{C}^{nxn}$ where: $P = I_n - 2\frac{\vec{v}\vec{v}^*}{\vec{v}^*\vec{v}}$ and $\vec{v} \in \mathbb{C}^n$.

It is known that P is hermitian ($P^* = P$) and unitary ($P^* = P^{-1}$) so its an **involution** - $P^2 = I_n$.

Now it is possible for every $\vec{x} \in \mathbb{C}^n$ to find a Householder transformation $P = P(\vec{x})$ such as, if $\vec{x} = (x_1, \ldots, x_n)^T$ then:

$$P\vec{x} = P\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_{j-1} \\ x_j \\ \vdots \\ x_n \end{pmatrix} \tag{1}$$

$$= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{k-1} \\ -sgn(x_k)\alpha \\ 0 \\ \vdots \\ 0 \\ x_{j+1} \\ \vdots \\ x_n \end{pmatrix} \tag{2}$$

where $\alpha = ||\vec{y}||_2$ (2-norm in $\mathbb{C}^n$ (or $\mathbb{R}^n$)) where $\vec{y} = \begin{pmatrix} x_k \\ x_{k+1} \\ \vdots \\ x_j \end{pmatrix} \in \mathbb{C}^{j-k+1}$

Now we proved in our notes that this matrix is none other than:

[P=I_n-2\frac{\vec{v}\vec{v}^*}{\vec{v}^*\vec{v}}]

for $\vec{v} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ x_k + sgn(x_k)\alpha \\ x_{k+1} \\ \vdots \\ x_j \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{C}^n$

Now this is essential in our methodology for the QR decomposition because we can use these Householder transformations in the following way:

$$\begin{bmatrix} * & \cdots & * \\ \vdots & A & \vdots \\ * & \cdots & * \end{bmatrix} \xrightarrow{H_1 A} \begin{bmatrix} * & * & \cdots & * \\ 0 & * & \cdots & * \\ \vdots & \vdots & & \vdots \\ 0 & * & \cdots & * \end{bmatrix} \xrightarrow{H_2 H_1 A} \begin{bmatrix} * & * & \cdots & \cdots & * \\ 0 & \times & * & \cdots & * \\ \vdots & 0 & * & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & * & \cdots & * \end{bmatrix} \quad \text{κλπ}$$

Στο τέλος θα έχουμε

$$H_n H_{n-1} \cdots H_2 H_1 A = R \qquad \text{άνω τριγωνικός}$$

$$A = QR \, , Q = H_1 H_2 \cdots H_{n-1} H_n$$

Essentially H_i is a Householder matrix, that transforms the i-th column in the manner discussed above, of the product $H_{i-1}A$, $\forall i = 1, \ldots, n$ - this works due to the definition of matrix multiplication where if C and D matrices (where for simplicity both are nxn square matrices), then if $\vec{d}_1, \ldots, \vec{d}_n$ the n columns of matrix D:

[CD =

$$\begin{pmatrix} \uparrow & \ldots & \uparrow \\ C\vec{d}_1 & \ldots & C\vec{d}_n \\ \downarrow & \ldots & \downarrow \end{pmatrix}$$

]

Although there is already a QR decomposition function in Python under the numpy module - `numpy.linalg.qr` , we create a function here for educational purposes, using the above methodology:

In [1]:

```
import numpy as np
import sympy
from functools import reduce
sympy.init_printing(use_latex="mathjax")

def qr_householder(A):
    """A function that calculates the QR decomposition of the input matrix A
    and outputs the resulting matrices using sympy and also returns them as np.arrays"""
    try:
        if np.linalg.det(A) == 0:
            print("For our implementation A has to be an invertible matrix, please try again.")
            return None
    except numpy.linalg.LinAlgError:
        print("For our implementation A has to be a square matrix, please try again.")
        return None
    n = A.shape[0]
    R = A; H_list = []
    for i in range(n-1):
        norm_col = np.linalg.norm(R[i:,i])
        first_el = np.array([R[i,i]+norm_col]) # or np.array([R[i,i]-norm_col]) -
                                                # doesn't make a difference
        if i==0:
            v = np.concatenate((first_el,R[i+1:,i]))
        else:
            v = np.concatenate((np.zeros((i)),first_el,R[i+1:,i]))
        v.shape = (n,1)
        H = np.eye(n)-2*(v @ np.transpose(v))/np.linalg.norm(v)**2
```

```
        H_list.append(H)
        R = H @ R
    Q = reduce(lambda X,Y: X @ Y, H_list)
    return [Q, R]
```

In [2]:

```
# Testing to see if everything is okay
A = np.array([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
Q, R = qr_householder(A)
```

In [3]:

```
sympy.Matrix(Q)
```

Out[3]:

$$\begin{bmatrix} -0.857142857142857 & 0.394285714285714 & 0.331428571428571 \\ -0.428571428571429 & -0.902857142857143 & -0.0342857142857143 \\ 0.285714285714286 & -0.171428571428571 & 0.942857142857143 \end{bmatrix}$$

In [4]:

```
sympy.Matrix(R) # note that it is indeed, to the accuracy of the
                # epsilon of the machine, upper triangular
```

Out[4]:

$$\begin{bmatrix} -14.0 & -21.0 & 14.0 \\ -7.7835174203336 \cdot 10^{-16} & -175.0 & 70.0 \\ -5.33590265989075 \cdot 10^{-16} & 0.0 & -35.0 \end{bmatrix}$$

In [5]:

```
sympy.Matrix(Q)*sympy.Matrix(R) # see that we got back A
```

Out[5]:

$$\begin{bmatrix} 12.0 & -51.0 & 4.0 \\ 6.0 & 167.0 & -68.0 \\ -4.0 & 24.0 & -41.0 \end{bmatrix}$$

With this out of the way, its time to see the 3 variations of the QR algorithm that we saw in the class - for more theory see corresponding lectures in "Numerical Linear Algebra" from Trefethen and Bau. We will not get into the details of each variation of the algorithm - for why these methods work intuitively (or proven rigorously), see the lecture notes and the handwritten complementary theory and observations accompanying this chapter for this information.

NOTE:

In all the variations below, we can accelerate their order of convergece (the ratio of convergence stays the same - the ratio of 2 consecutive eigenvalues) by first "*relaxing*" A with *Householder matrices* and bringing it in its upper triangular Hessenberg form (this is because the QR algorithm *PRESERVES* upper triangular Hesseberg matrices) and then using consecutive *deflations* of the resulting matrices to decrease the number of eigenvalues we need to find, each time we "pinpoint" an eigenvalue, thus decreasing the dimensions of the problem each time by 1.

Simple QR algorithm (withour shifts)

In [6]:

```
def simple_qr(A, tol, maxiter):
    """Uses the simple no-shift QR algorithm to approximate
```

```python
    the eigenvalues of the input matrix"""
    A_new=A
    n = 0
    while np.linalg.norm(
        np.tril(A_new)-np.diag(np.diag(A_new)), ord=2) >= tol and n<maxiter:
        # we can use whatever norm we want due to the equivelance of operator
        # norms in vector spaces of finite dimensions
        Q, R = qr_householder(A_new)
        A_new = R @ Q
        n += 1
    return [np.diag(A_new), n]
```

In [7]:

```python
# Testing to see if everything is okay
A = np.array([[8, 7, 7], [5, 8, 4], [2, 0, 8]])
eigvals, iterations = simple_qr(A, 1E-16, 100)
```

In [8]:

```python
print(f"""Number of iterations it took to converge to the
          specified tolerace: {iterations}""")
```

```
Number of iterations it took to converge to the
          specified tolerace: 45
```

In [9]:

```python
# Approximation using the simple QR algorithm
sympy.Matrix(eigvals)
```

Out[9]:

$$\begin{bmatrix} 15.5135354931502 \\ 6.82394694026641 \\ 1.66251756658343 \end{bmatrix}$$

In [10]:

```python
# "Exact" eigvalues of matrix A, using the np.linalg.eig() function
sympy.Matrix(np.linalg.eig(A)[0])
```

Out[10]:

$$\begin{bmatrix} 15.5135354931502 \\ 1.66251756658343 \\ 6.82394694026639 \end{bmatrix}$$

QR algorithm with simple shifts

In [11]:

```python
def simple_shift_qr(A, tol, maxiter):
    """Uses the simple shift QR algorithm to approximate the
    eigenvalues of the input matrix"""
    n = A.shape[0]
    sigma=A[n-1,n-1]
    niter = 0
    while np.linalg.norm(
        np.tril(A-sigma*np.eye(n,n))-np.diag(np.diag(A-sigma*np.eye(n,n))), ord=2) \
        >= tol and niter<maxiter:
        # we can use whatever norm we want due to the equivelance of operator norms in
        # vector spaces of finite dimensions (frobenius norm, 2/1/inf-norm)
        Q, R = qr_householder(A-sigma*np.eye(n,n))
        A = R @ Q + sigma*np.eye(n,n)
        sigma=A[n-1,n-1]
        niter += 1
    return [np.diag(A), n]
```

In [12]:

```python
# Testing to see if everything is okay
A = np.array([[8, 7, 7], [5, 8, 4], [2, 0, 8]])
```

```python
eigvals, iterations = simple_shift_qr(A, 1E-20, 100)
```

In [13]:

```python
print(f"""Number of iterations it took to converge to the
        specified tolerace: {iterations}""")
```

```
Number of iterations it took to converge to the
        specified tolerace: 3
```

In [14]:

```python
# Approximation using the simple-shift QR algorithm
sympy.Matrix(eigvals)
```

Out[14]:

$$\begin{bmatrix} 15.5135354931502 \\ 1.66251756657081 \\ 6.823946940279 \end{bmatrix}$$

In [15]:

```python
# "Exact" eigvalues of matrix A, using the np.linalg.eig() function
sympy.Matrix(np.linalg.eig(A)[0])
```

Out[15]:

$$\begin{bmatrix} 15.5135354931502 \\ 1.66251756658343 \\ 6.82394694026639 \end{bmatrix}$$

QR algorithm with Wilkinson shifts

In [16]:

```python
def wilkinson_shift_qr(A, tol, maxiter):
    """Uses the Wilkinson shift QR algorithm to approximate the
    eigenvalues of the input matrix"""
    n = A.shape[0]
    sub_matrix = A[n-2:n,n-2:n]
    char_pol = [1, -np.trace(sub_matrix), np.linalg.det(sub_matrix)]
    sigma=max(map(lambda x: abs(x), numpy.roots(char_pol)))
    niter = 0
    while np.linalg.norm(
        np.tril(A-sigma*np.eye(n,n))-np.diag(np.diag(A-sigma*np.eye(n,n))), ord=2) \
        >= tol and niter<maxiter:
        # we can use whatever norm we want due to the equivelance of operator norms in
        # vector spaces of finite dimensions (frobenius norm, 2/1/inf-norm)
        Q, R = qr_householder(A-sigma*np.eye(n,n))
        A = R @ Q + sigma*np.eye(n,n)
        sub_matrix = A[n-2:n,n-2:n]
        char_pol = [1, -np.trace(sub_matrix), np.linalg.det(sub_matrix)]
        sigma=max(map(lambda x: abs(x), numpy.roots(char_pol)))
        niter += 1
    return [np.diag(A), n]
```

In [17]:

```python
# Testing to see if everything is okay
A = np.array([[8, 7, 7], [5, 8, 4], [2, 0, 8]])
eigvals, iterations = simple_shift_qr(A, 1E-20, 100)
```

In [18]:

```python
print(f"""Number of iterations it took to converge to the
        specified tolerace: {iterations}""")
```

```
Number of iterations it took to converge to the
        specified tolerace: 3
```

In [19]:

```python
# Approximation using the simple-shift QR algorithm
```

```python
sympy.Matrix(eigvals)
```

Out[19]:

$$\begin{bmatrix} 15.5135354931502 \\ 1.66251756657081 \\ 6.823946940279 \end{bmatrix}$$

In [20]:

```python
# "Exact" eigvalues of matrix A, using the np.linalg.eig() function
sympy.Matrix(np.linalg.eig(A)[0])
```

Out[20]:

$$\begin{bmatrix} 15.5135354931502 \\ 1.66251756658343 \\ 6.82394694026639 \end{bmatrix}$$