Group: Ella Noomen, Florian Braun, Jacob McCalla

In [ ]:

In this exercise, we explore the numerical implications of two variations of the Gram-Schmidt algorithm for orthonormalising a set of vectors. Theorem 1.1.2 from the lecture slides describes this problem as follows: "$x_1, \ldots, x_k \in \mathbb{C}^n$ are linearly independent, then the Gram-Schmidt orthogonalization generates an orthonormal set $v1, \ldots, v_k \in \mathbb{C}^n$". The two variations of the Gram-Schmidt orthogonalisation are both iterative, and mathematically equivalent, but the critical difference lies in which set of vectors we use when computing the projection at each iteration. Function2 is known as the modified Gram-Schmidt algorithm.

In [2]:
```
import numpy as np
```

In [3]:
```
# This is the function on the LHS of Algorithm 1.1.3 on the lecture slides
def function1(X):
    v = np.zeros(X.shape)
    for j in range(X.shape[0]):
        y = np.zeros(X.shape[1])
        for i in range(j):
            r = np.inner(X[j], v[i])
            y += r * v[i]
        v[j] = (X[j] - y)/np.linalg.norm(X[j] - y)
    return v
```

In [4]:
```
# This is the function on the RHS of Algorithm 1.1.3 on the lecture slides
def function2(X):
    v = np.zeros(X.shape)
    for j in range(X.shape[0]):
        w = X[j]
        for i in range(j):
            r = np.inner(w, v[i])
            w = w - r * v[i]
        v[j] = (w)/np.linalg.norm(w)
    return v
```

In [5]:
```
# This function returns a Hilbert matrix of size nxn
def GenerateHilbert(n):
    H = np.zeros((n,n))
    for i in range(1, n+1):
        h = [1/(i + j) for j in range(n)]
        H[i-1] = h
    return H
```

In [6]:
```
n = 1000
```

In [7]:
```
H = GenerateHilbert(n)
```

In [8]:
```
R1 = function1(H)
```

In [9]:
```
R2 = function2(H)
```

In [10]:
```
R3 = function1(R1)
```

In [11]:
```
R4 = function1(R2)
```

Since $R_1^T R_1$ and $R_2^T R_2$ are supposed to be equal to the identity matrix, the sum of all off diagonal entry should be zero, therefore $G_k := R_k^T R_k - \mathbb{I}$, $\Sigma_k = \sum_{i=1}^n \sum_{j=1}^n |g_{ij}^k|$ has to be zero. Due to numerical / rounding errors, this does not hold for the computed matrices above. Large Σ_k indicates, that the according matrix R_k is not orthogonal.

In [12]:
```
np.sum(abs(np.matmul(R1.T, R1) - np.identity(n)))
```

Out[12]:
```
503893.94485053775
```

In [13]:
```
np.sum(abs(np.matmul(R2.T, R2) - np.identity(n)))
```

Out[13]:
```
4127.781037977058
```

As we can see, the value when using function2 is significantly smaller than when using function1, which shows that this function returns a more orthogonal matrix, and hence function2 yields better results for the problem. This is because due to the fact that the orthogonalization inside the inner loop is not directly based on the according vector x_j. Rather, the numerical errors are taken into account during the process. This results in a "more orthogonal" matrix, which is the aim of the algorithm.

In [14]:
```
np.sum(abs(np.matmul(R3.T, R3) - np.identity(n)))
```

Out[14]:
```
355194.0701877724
```

In [15]:
```
np.sum(abs(np.matmul(R4.T, R4) - np.identity(n)))
```

Out[15]:
```
3.5640844958381046e-07
```

Finally, matrices R3 and R4 are the result of reorthogonalising matrices R1 and R2. In this context, reorthogonalising means applying the function again to the output of running the algorithm the first time. As we can see, after reorthogonalisation, the values for $\sum_{i=1}^n \sum_{j=1}^n |g_{ij}^k|$ are (significantly) lower, suggesting that it is fruitful to apply to algorithm once more. In fact, in the case of R4, which was achieved by running with R2 as input, the sum is very close to 0, where 0 means the matrix is orthogonal.