PROBLEM 1 - Jacobi

Gisela Martí Guerrero, ICC 2023

Python program that computes the effective Hamiltonian by Jacobi diagonalization method applied to the Hilbert matrix $N = 10$.

In [1]:
```
import numpy as np
from pandas import *
```

We start by defyining two functions that are later used in the main Jacobi function.

1. The first function "max_element" finds the maximum absolute element in a 2D array a and returns the value of the maximum element along with its indices ip and iq. The function uses nested loops to iterate over the elements of the lower triangular part of the array (excluding the main diagonal). It then checks if the absolute value of the current element $a[i, j]$ is greater than or equal to the current maximum element. If true, it updates updates the "max_el" array to the absolute value of $a[i, j]$, and also updates the indices ip and iq to the values of i and j.

In [2]:
```
def maximum_element(a):
    max_el = 0.0
    n = len(a)
    for i in range(n - 1):
        for j in range(i + 1, n):
            if abs(a[i,j]) >= max_el:
                max_el = abs(a[i,j])
                ip = i
                iq = j
    return max_el, ip, iq
```

2. The second function "rotation" performs a rotation operation on a matrix a to make the element $a[ip, iq]$ become zero. It also updates a transformation matrix v accordingly. The function checks a condition involving the absolute value of $a[ip, iq]$ and h. If $abs(a[ip, iq]) < abs(h) \cdot 1.0^{-36}$, set $t = \frac{a[ip,iq]}{h}$. Otherwise, compute a rotation to zero out $a[ip, iq]$. τ is calculated using $sin = t \cdot cos$ and $cos = \frac{1}{\sqrt{t^2+1}}$, and the matrix a is updated to perferom the rotation. The elements in the matrix are updated for three cases: $1 < i < p$, $p < i < q$, and $n > i > q$.

In [3]:
```
def rotation(a, v, ip, iq):
    n = len(a)
    h = a[iq,iq] - a[ip,ip]
    if abs(a[ip,iq]) < abs(h)*1.0e-36: # Machine epsilon ~ 1e-36
        t = a[ip,iq]/h
    else:
        theta = h/(2.0*a[ip,iq])
        t = 1.0/(abs(theta) + np.sqrt(1.0 + theta**2))
        if theta < 0.0:
            t = -t
    c = 1.0/np.sqrt(1.0 + t**2)
    s = t*c
    tau = s/(1.0 + c)
    temp = a[ip,iq] #h
    a[ip,iq] = 0.0
    a[ip,ip] = a[ip,ip] - t*temp
    a[iq,iq] = a[iq,iq] + t*temp
    for i in range(ip):       # Case of rotations 1 < i < p
        temp = a[i,ip]
        a[i,ip] = temp - s*(a[i,iq] + tau*temp)
        a[i,iq] = a[i,iq] + s*(temp - tau*a[i,iq])
    for i in range(ip+1,iq):  # Case of rotations p < i < q
        temp = a[ip,i]
        a[ip,i] = temp - s*(a[i,iq] + tau*a[ip,i])
        a[i,iq] = a[i,iq] + s*(temp - tau*a[i,iq])
    for i in range(iq+1,n):   # Case of n > i > q
        temp = a[ip,i]
        a[ip,i] = temp - s*(a[iq,i] + tau*temp)
        a[iq,i] = a[iq,i] + s*(temp - tau*a[iq,i])
    for i in range(n):
        temp = v[i,ip]
        v[i,ip] = temp - s*(v[i,iq] + tau*v[i,ip])
        v[i,iq] = v[i,iq] + s*(temp - tau*v[i,iq])
```

We define the Jacobi function, which implements Jacobi's method for solving the eigenvalue problem. The function iteratively applies Jacobi rotations to diagonalize the matrix a. A limit on the number of rotations is set with "max_rot" to avoid infinite loops in case the method doesn't converge. The transformation matrix is initilized in v as the identity matrix, which will be updated during the Jacobi rotations.

Within the main iteration loop, the maximum off-diagonal element and its indices are found with the "max_element" function. If the maximum off-diagonal element is below the tolerance, the function returns the eigenvalues (diagonal elements of a) and the transformation matrix v. Then the rotation is applied to diagonalize the matrix a and update the transformation matrix v.

In [4]:
```
def jacobi(a):
    n = len(a)
    max_rot = 10*(n**2)
    v = np.identity(n)*1.0
    for i in range(max_rot):
        max_a,ip,iq = maximum_element(a)
        if max_a < tolerance:
            return np.diagonal(a),v
        rotation(a,v,ip,iq)
```

The Hilbert matrix is now defined in the following function, taking N as an input parameter

In [5]:
```
def hilbert(n):
    H_matrix = np.array([[1.0/(i+j-1) for j in range(1, n+1)] for i in range(1, n+1)])
    return H_matrix
```

The original matrix is divided into "PAP" and "QAQ". "PAP" corrresponds to the top-left quadrant of the matrix, and "QAQ" to the bottom-right quadrant of the matrix.

In [6]:
```
# Define the tolerance
tolerance = 1.0e-9

# Hilbert matrix
N = 10
HM = hilbert(N)

# PAP and QAQ quadrants
PAP = HM[:int(N/2),:int(N/2)]
QAQ = HM[int(N/2):,int(N/2):]
```

In [7]:
```
print("Input matrix:")
print(DataFrame(HM))
print("")
print("PAP:")
print(DataFrame(PAP))
print("")
print("QAQ:")
print(DataFrame(QAQ))
```

```
Input matrix:
          0         1         2         3         4         5         6  \
0  1.000000  0.500000  0.333333  0.250000  0.200000  0.166667  0.142857
1  0.500000  0.333333  0.250000  0.200000  0.166667  0.142857  0.125000
2  0.333333  0.250000  0.200000  0.166667  0.142857  0.125000  0.111111
3  0.250000  0.200000  0.166667  0.142857  0.125000  0.111111  0.100000
4  0.200000  0.166667  0.142857  0.125000  0.111111  0.100000  0.090909
5  0.166667  0.142857  0.125000  0.111111  0.100000  0.090909  0.083333
6  0.142857  0.125000  0.111111  0.100000  0.090909  0.083333  0.076923
7  0.125000  0.111111  0.100000  0.090909  0.083333  0.076923  0.071429
8  0.111111  0.100000  0.090909  0.083333  0.076923  0.071429  0.066667
9  0.100000  0.090909  0.083333  0.076923  0.071429  0.066667  0.062500

          7         8         9
0  0.125000  0.111111  0.100000
1  0.111111  0.100000  0.090909
2  0.100000  0.090909  0.083333
3  0.090909  0.083333  0.076923
4  0.083333  0.076923  0.071429
5  0.076923  0.071429  0.066667
6  0.071429  0.066667  0.062500
7  0.066667  0.062500  0.058824
8  0.062500  0.058824  0.055556
9  0.058824  0.055556  0.052632

PAP:
          0         1         2         3         4
0  1.000000  0.500000  0.333333  0.250000  0.200000
1  0.500000  0.333333  0.250000  0.200000  0.166667
2  0.333333  0.250000  0.200000  0.166667  0.142857
3  0.250000  0.200000  0.166667  0.142857  0.125000
4  0.200000  0.166667  0.142857  0.125000  0.111111

QAQ:
          0         1         2         3         4
0  0.090909  0.083333  0.076923  0.071429  0.066667
1  0.083333  0.076923  0.071429  0.066667  0.062500
2  0.076923  0.071429  0.066667  0.062500  0.058824
3  0.071429  0.066667  0.062500  0.058824  0.055556
4  0.066667  0.062500  0.058824  0.055556  0.052632
```

The Jacobi method is aplied to find the eigenvalues and eigenvector of the matrix. It then checks if computed eigenvalues and eigenvectors satisfy the eigenvalue problem:

$$HM \cdot v = eigenval \cdot v \cdot HM \cdot v = eigenval \cdot v$$

In [8]:
```
# Obtain the eigenvalues and the transformation matrix
eigenval, v = jacobi(HM.copy())

# Check if the eigenvalue problem is satisfied
print("Is the eigenvalue problem satisfied?: ",np.all(np.isclose((HM@v),(eigenval*v))))
print("")
print("Eigenvalues:")
print(DataFrame(eigenval))
print("")
print("Transformation matrix:")
print(DataFrame(v))
```

```
Is the eigenvalue problem satisfied?:  True

Eigenvalues:
              0
0  1.751920e+00
1  3.429295e-01
2  2.530891e-03
3  4.729689e-06
4  1.287496e-04
5  7.471500e-10
6  1.222677e-09
7  3.574182e-02
8  2.004035e-10
9  1.228968e-07

Transformation matrix:
          0         1         2         3         4         5         6  \
0  0.699515 -0.637641  0.105516 -0.006675 -0.029313 -0.000126 -0.000134
1  0.425999  0.070437 -0.576272  0.127507  0.322879  0.005959  0.005871
2  0.316977  0.233557  0.201934 -0.497400 -0.583114 -0.068125 -0.059183
3  0.255523  0.282109  0.412044  0.410990 -0.169553  0.313677  0.214869
4  0.215278  0.293657  0.351322  0.363287  0.229493 -0.672249 -0.236619
5  0.186578  0.290982  0.195433 -0.048748  0.374836  0.638650 -0.283607
6  0.164947  0.282498  0.019399 -0.343218  0.309362 -0.144124  0.763411
7  0.147992  0.271744 -0.147212 -0.352917  0.112144 -0.069885 -0.288594
8  0.134310  0.260324 -0.294209 -0.075756 -0.154211 -0.077595 -0.323533
9  0.123017  0.248988 -0.419514  0.430041 -0.448206  0.073901  0.207647

          7         8         9
0  0.303404  0.000043  0.001247
1 -0.599560 -0.001695 -0.037685
2 -0.378571  0.014558  0.255145
3 -0.144314 -0.037406 -0.566476
4  0.027497 -0.004708  0.225085
5  0.146917  0.087269  0.426542
6  0.229139  0.148950  0.029143
7  0.285632 -0.661374 -0.371865
8  0.324264  0.689726 -0.326419
9  0.350361 -0.235421  0.366471
```

The effective Hamiltonian matrix H_{eff} is now computed by transforming the original matrix with the eigenvectors obtained from the Jacobi method, $v^T \cdot HM \cdot v$.

In [9]:
```
# Effective Hamiltonian matrix
eff_H = v.T@HM@v
print("Effective Hamiltonian:")
print(DataFrame(eff_H))
```

```
Effective Hamiltonian:
              0             1             2             3             4  \
0  1.751920e+00 -5.689893e-16 -8.562595e-15 -2.885886e-13  1.920366e-10
1 -4.267420e-16  3.429295e-01  1.863162e-13 -8.264223e-15  6.260508e-10
2 -8.592953e-15  1.863275e-13  2.530891e-03 -2.031144e-15 -7.565021e-16
3 -2.885614e-13 -8.260392e-15 -2.027332e-15  4.729689e-06  8.261826e-11
4  1.920366e-10  6.260508e-10 -7.628616e-16  8.261827e-11  1.287496e-04
5 -1.863866e-18  1.853979e-13 -2.879189e-19 -7.147236e-18 -8.669652e-12
6  4.847879e-10  1.003477e-12  5.682503e-16  8.486005e-14  2.446813e-11
7  1.732741e-11 -8.640938e-12  4.336809e-18  6.524295e-15  7.856874e-10
8 -3.203350e-11 -5.476152e-13 -2.755421e-14 -7.760132e-14 -6.152927e-12
9  5.233481e-10 -2.384651e-12 -2.032800e-15  1.490884e-10  1.092110e-10

              5             6             7             8             9
0  3.469447e-18  4.847879e-10  1.732745e-11 -3.203348e-11  5.233480e-10
1  1.854081e-13  1.003485e-12 -8.640956e-12 -5.476071e-13 -2.384669e-12
2 -9.242824e-18  5.585539e-16 -3.848918e-18 -2.755234e-14 -2.023880e-15
3  5.569586e-18  8.485709e-14  6.533697e-15 -7.761042e-14  1.490884e-10
4 -8.669652e-12  2.446813e-11  7.856874e-10 -6.152932e-12  1.092110e-10
5  7.471500e-10  9.436124e-10  1.655852e-14 -3.639752e-10 -2.229056e-11
6  9.436124e-10  1.222677e-09 -7.381042e-14 -4.862699e-10  3.676109e-11
7  1.654764e-14 -7.383330e-14  3.574182e-02 -7.650933e-13 -2.394456e-13
8 -3.639752e-10 -4.862699e-10 -7.650872e-13  2.004035e-10 -3.566914e-14
9 -2.229056e-11  3.676108e-11 -2.394644e-13 -3.567329e-14  1.228968e-07
```