Brandon Swatek - zadanie numeryczne nr 5

1. Zadania numeryczne

5. Macierz $\mathbf{A} \in \mathbb{R}^{64\times64}$ ma strukturę:

$$\begin{bmatrix} 5 & 2 & 1 & 1 & 2 & 1 & 1 & \dots \\ 2 & 5 & 2 & 1 & 1 & 2 & 1 & \dots \\ 1 & 2 & 5 & 2 & 1 & 1 & 2 & \dots \\ 1 & 1 & 2 & 5 & 2 & 1 & 1 & \dots \\ 2 & 1 & 1 & 2 & 5 & 2 & 1 & \dots \\ 1 & 2 & 1 & 1 & 2 & 5 & 2 & \dots \\ 1 & 1 & 2 & 1 & 1 & 2 & 5 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} \tag{1}$$

Za pomocą odpowiedniego algorytmu rozwiązać układ równań $\mathbf{Ax} = \mathbf{e}$ gdzie $\mathbf{e}$ jest wektorem którego wszystkie składowe są równe 1 ($e_i = 1, i = 1, .., 64$).

Do rozwiązania zadania numerycznego wybieram metodę iteracyjną - prewarunkowaną metodę gradientów sprzężonych.

In [1]:
```
import numpy as np


def preconditioned_conjugate_gradients(matrix_a, vector_x, vector_b, it
erations):
    r = vector_b - np.dot(matrix_a, vector_x)
    matrix_m = preconditioner(matrix_a)
    z = thomas(matrix_m.diagonal(-1), matrix_m.diagonal(), matrix_m.dia
gonal(1), r)
    p = z
    rsold = np.dot(r.T, z)
    for k in range(iterations):
        Ap = np.dot(matrix_a, p)
        alpha = float(rsold / np.dot(np.transpose(p), Ap))
        vector_x = vector_x + np.dot(alpha, p)
        r = r - np.dot(alpha, Ap)
        z = thomas(matrix_m.diagonal(-1), matrix_m.diagonal(), matrix_
m.diagonal(1), r)
        rsnew = np.dot(np.transpose(r), z)
        p = z + (rsnew / rsold) * p
        rsold = rsnew
    return vector_x
```

Metoda oryginalna ma złożoność $O(kN^2)$, sens prewarunkowania polega na przyspieszeniu działania metody. Obliczanie dodatkowego wektora w każdej pętli zwiększa złożoność, jednak już po mniejszej zadanej liczbie iteracji k można uzyskać satysfakcjonujący wynik.

In [2]:
```
def preconditioner(matrix_a):
    n = len(matrix_a[0])
    jacobi = np.zeros((n, n))
    diag = np.diagonal(matrix_a)
    for i in range(n):
        for j in range(n):
            if i == j:
                jacobi[i][j] = diag[i]
    return jacobi


def thomas(a, b, c, d):
    nf = len(d)
    ac, bc, cc, dc = map(np.array, (a, b, c, d))
    for it in range(1, nf):
        mc = ac[it - 1] / bc[it - 1]
        bc[it] = bc[it] - mc * cc[it - 1]
    dc[it] = dc[it] - mc * dc[it - 1]
    xc = bc
    xc[-1] = dc[-1] / bc[-1]
    for il in range(nf - 2, -1, -1):
        xc[il] = (dc[il] - cc[il] * xc[il + 1]) / bc[il]
    return xc
```

Właściwy algorytm wymaga też metod pomocniczych, jak wyżej. Funkcja preconditioner zwraca macierz umożliwiającą prewarunkowanie, (Jacobi preconditioner), którą pobieram jako diagonalę macierzy wejściowej. Algorytm Thomasa, zastępuje w moim kodzie konieczność obliczania macierzy odwrotnej.

In [3]:
```
def create_a(n):
    matrix_a = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i == j:
                matrix_a[i][j] = 5
            elif (i - j) == 1 or (j - i) == 1 or (j - i) == 4 or (i -
j) == 4:
                matrix_a[i][j] = 2
            else:
                matrix_a[i][j] = 1
    return matrix_a


e = np.ones(64)
x = np.zeros(64)
A = create_a(64)
print(np.linalg.solve(A, e))
print(preconditioned_conjugate_gradients(A, x, e, 12))
```

```
[0.0211007  0.01422044 0.0159436  0.0176289  0.00998803 0.01468522
 0.01298754 0.01216405 0.01524432 0.01267438 0.01386826 0.01409431
 0.01280751 0.01411594 0.01338202 0.01339421 0.01392666 0.01328359
 0.01370478 0.01363143 0.01341933 0.01372954 0.01349511 0.01356723
 0.01364633 0.0134951  0.01362928 0.01356947 0.01354428 0.01362573
 0.01353448 0.01359261 0.01359261 0.01353448 0.01362573 0.01354428
 0.01356947 0.01362928 0.0134951  0.01364633 0.01356723 0.01349511
 0.01372954 0.01341933 0.01363143 0.01370478 0.01328359 0.01392666
 0.01339421 0.01338202 0.01411594 0.01280751 0.01409431 0.01386826
 0.01267438 0.01524432 0.01216405 0.01298754 0.01468522 0.00998803
 0.0176289  0.0159436  0.01422044 0.0211007 ]
[0.02110003 0.01422131 0.01594421 0.01762858 0.00998806 0.01468497
 0.01298661 0.01216313 0.01524488 0.01267565 0.01386872 0.01409388
 0.01280931 0.01411353 0.01338396 0.01339236 0.01392331 0.01328502
 0.01370464 0.01363246 0.01342544 0.01372755 0.01349781 0.01356497
 0.01363924 0.01350624 0.01360883 0.01357783 0.01355269 0.01359957
 0.01356693 0.0135827  0.0135827  0.01356693 0.01359957 0.01355269
 0.01357783 0.01360883 0.01350624 0.01363924 0.01356497 0.01349781
 0.01372755 0.01342544 0.01363246 0.01370464 0.01328502 0.01392331
 0.01339236 0.01338396 0.01411353 0.01280931 0.01409388 0.01386872
 0.01267565 0.01524488 0.01216313 0.01298661 0.01468497 0.00998806
 0.01762858 0.01594421 0.01422131 0.02110003]
```

Dla porównania zamieszczam też wyniki uzyskane przy pomocy numpy.linalg.solve