Problem 1

Use the Jacobi Method to solve the following system for $n = 100$. Use the stopping criterion that the infinity norm of the difference between the iterate and true solution is less than 10^{-6} or the number of iterations reaches n. The correct solution is $[1, \dots, 1]$, compute the right-hand side b using the np.dot function to multiply matrix and vector. Report the number of steps needed and the forward error (difference from the solution) and the backward error (the residual) in the infinity norm. The system is

$$\begin{bmatrix} 3 & -1 & & & \\ -1 & 3 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 3 & -1 \\ & & & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = b$$

MATH 5660 only:

(a) Same as above, but do not store the matrix A. This is possible since you know what its entries are and it has a special structure so you can just hardcode them and compute $b[i]$ and new $x[i]$ by formulas. $Make sure you get the same result as above.

(b) Change the diagonal entries of A to 2, compute new b, and run your code for $n = 100, 1000, 10000$.

In [1]:

```
import numpy as np

def jacobi(A, b, x0, eps=1e-6, max_iterations=None):
    d = np.diag(A)
    r = A - np.diag(d)
    x = x0.copy()
    k = 0

    while True:
        x_2 = (b - np.dot(r, x)) / d
        ferror = np.linalg.norm(x_2 - x, np.inf)
        berror = np.linalg.norm(b - np.dot(A, x_2), np.inf)

        if ferror < eps or (max_iterations is not None and k >= max_iterat
            break

        x = x_2
        k += 1

    return x, k, ferror, berror

n = 100
diag = 3 * np.ones(n)
offdiag = -np.ones(n - 1)
A = np.diag(diag) + np.diag(offdiag, k=-1) + np.diag(offdiag, k=1)

correct_solution = np.ones(n)

b = np.dot(A, correct_solution)

x0 = np.zeros(n)

x, iterations, ferror, berror = jacobi(A, b, x0)

print('A = \n', A)
print()
print('b (computed using np.dot(A, b)) = \n', b)
print()
print('x = \n', x)
print()
print('Iterations:', iterations)
print()
print('Forward error:', ferror)
print()
print('Backward error:', berror)
```

```
A =
 [[ 3. -1.  0. ...  0.  0.  0.]
 [-1.  3. -1. ...  0.  0.  0.]
 [ 0. -1.  3. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  3. -1.  0.]
 [ 0.  0.  0. ... -1.  3. -1.]
 [ 0.  0.  0. ...  0. -1.  3.]]

b (computed using np.dot(A, b)) =
 [2. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 2.]

x =
 [0.99999968 0.99999937 0.99999907 0.99999881 0.99999856 0.99999837
 0.99999818 0.99999806 0.99999794 0.99999787 0.9999978  0.99999776
 0.99999773 0.99999771 0.9999977  0.99999769 0.99999769 0.99999769
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768 0.99999768
 0.99999768 0.99999768 0.99999768 0.99999768 0.99999769 0.99999769
 0.99999769 0.9999977  0.99999771 0.99999773 0.99999776 0.9999978
 0.99999787 0.99999794 0.99999806 0.99999818 0.99999837 0.99999856
 0.99999881 0.99999907 0.99999937 0.99999968]

Iterations: 32

Forward error: 7.726460993229267e-07

Backward error: 1.5452818485917064e-06
```

Problem 2

Carry out the steps of Problem 1 with $n = 100$ for

(a) Gauss–Seidel Method and

(b) SOR with $\omega = 1.2$.

Which one converges faster - Jacobi, Gauss-Seidel, or SOR?

MATH 5660 only:

Carry out the steps as in Problem 1 part for MATH 5660 only, with the Gauss–Seidel Method

In [2]:
```
# GS Method

def Gauss_Seidel(A, b, x0, eps=1e-6, max_iterations=None):
    n = A.shape[0]
    x = x_0.copy()
    k = 0

    while True:
        for i in range(n):
            sum = 0.
            for j in range(n):
                if i != j:
                    sum += A[i,j] * x[j]
            x[i] = (b[i] - sum)/A[i,i]

        ferror = np.linalg.norm(x - correct_solution, np.inf)
        berror = np.linalg.norm(b - np.dot(A, x), np.inf)

        if ferror < eps or (max_iterations is not None and k >= max_iterat
            break

        k+= 1

    return x, k, ferror, berror

x_0 = np.zeros(n)

x, iterations, ferror, berror = Gauss_Seidel(A, b, x0)

#print('A = \n', A)
#print()
#print('x = \n', x)
print('With Gauss_Seidel Method:')
print()
print('Iterations:', iterations)
print()
print('Forward error:', ferror)
print()
print('Backward error:', berror)
```

```
With Gauss_Seidel Method:

Iterations: 19

Forward error: 9.5367431640625e-07

Backward error: 9.564705892861625e-07
```

In [3]:
```
# SOR method, omega = 1.2

def SOR(A, b, x0, omega, eps=1e-6, max_iterations=None):
    n = A.shape[0]
    x = x_0.copy()
    k = 0

    while k < n:
        x_2 = x.copy()
        for i in range(n):
            sum = 0
            for j in range(n):
                if i != j:
                    sum += A[i,j]*x_2[j]
            x_2[i] = omega*(b[i] - sum)/A[i,i] + (1.0 - omega) * x[i]
        ferror = np.linalg.norm(x_2 - correct_solution, np.inf)
        berror = np.linalg.norm(b - np.dot(A, x_2), np.inf)
        if ferror < eps or (max_iterations is not None and k >= max_iterat
            break
        x = x_2
        k += 1
    return x, k, ferror, berror

x_0 = np.zeros(n)

omega = 1.2

x, iterations, ferror, berror = SOR(A, b, x0, omega)

#print('A = \n', A)
#print()
#print('x = \n', x)
#print()
print('With SOR Method when \omega = 1.2')
print()
print('Iterations:', iterations)
print()
print('Forward error:', ferror)
print()
print('Backward error:', berror)
```

```
With SOR Method when \omega = 1.2

Iterations: 15

Forward error: 4.0626615649408393e-07

Backward error: 1.5515907683116836e-06
```

Problem 3

Solve the system $Hx = b$ by the Conjugate Gradient Method, where H is the $n \times n$ Hilbert matrix (see Wikipedia for definition) and b is the vector of all ones, for

(a) $n = 4$

(b) $n = 8$.

What can you say about the solutions?

In [4]:
```
import scipy as sp

n = 4
H = sp.linalg.hilbert(n)
b = np.ones(n)
x0 = np.zeros(n)

def ConjGrad(A, b, x0, eps):
    if not np.array_equal(A.T, A):
        print('Error: the matrix A is not symmetric!')
        return

    n = A.shape[0]
    r = np.zeros((n,n+1))
    u = np.zeros((n,n+1))
    r[:,0] = b-np.dot(A, x0)
    if np.linalg.norm(r) < eps:
        return x0
    u[:,0] = r[:,0]
    x = x0.copy()

    for k in range(n):
        a = np.dot(u[:,k], r[:,k])/np.dot(u[:,k], np.dot(A,u[:,k]))
        x += a*u[:,k]
        r[:,k+1] = b - np.dot(A, x)
        if np.linalg.norm(r) < eps:
            return x
        sum = np.zeros(n)
        for i in range(k+1):
            sum += np.dot(r[:,k+1], np.dot(A, u[:,i]))/np.dot(u[:,i], np.d
        u[:,k+1] = r[:,k+1] - sum

    return x

ConjGrad(H, b, x0, 10e-6)
```

Out[4]:
```
array([  -4.,   60., -180.,  140.])
```

In [5]:
```
n = 8
H = sp.linalg.hilbert(n)
b = np.ones(n)
x0 = np.ones(n)

ConjGrad(H, b, x0, 10e-6)
```

Out[5]:
```
array([-8.00000058e+00,  5.04000029e+02, -7.56000035e+03,  4.62000018e+0
4,
       -1.38600005e+05,  2.16216006e+05, -1.68168004e+05,  5.14800012e+0
4])
```

In [6]:
```
# checking answers

n_values = [4, 8]

for n in n_values:
    H = sp.linalg.hilbert(n)
    b = np.ones(n)
    x = sp.sparse.linalg.cg(H, b)

    print('-'*50)
    print('n =', n)
    print()
    print('Solution:', x)
```

```
--------------------------------------------------
n = 4

Solution: (array([  -4.,   60., -180.,  140.]), 0)
--------------------------------------------------
n = 8

Solution: (array([-7.99999699e+00,  5.03999798e+02, -7.55999759e+03,  4.
61999879e+04,
       -1.38599968e+05,  2.16215955e+05, -1.68167968e+05,  5.14799908e+0
4]), 0)
```

As we would expect from the inherent ill-conditioning of the Hilbert matrix, the greater the number of entires (n), the less accurate the answers given to us by the Conjugate Gradient Method become. Meaning that it is possible that the Conjugate Gradient method may become impractical when a Hilbert matrix becomes large enough.

To further illustrate this, we can compare the 2-norm condition numbers of several Hilbert matrices. Note how the rate of change increases slightly with each value of n (i.e. first step is scaled by a factor of ~19, second by ~27, third by ~29, fourth by ~31, etc.).

In [7]:
```
n_values = [1,2,3,4,5,6,7,8]

for n in n_values:
    H = sp.linalg.hilbert(n)
    cond_num = np.linalg.cond(H)

    print('-'*60)
    print('n =', n)
    print()
    print('Condition number for the Hilbert Matrix:', cond_num)
```

```
------------------------------------------------------------
n = 1

Condition number for the Hilbert Matrix: 1.0
------------------------------------------------------------
n = 2

Condition number for the Hilbert Matrix: 19.281470067903967
------------------------------------------------------------
n = 3

Condition number for the Hilbert Matrix: 524.0567775860627
------------------------------------------------------------
n = 4

Condition number for the Hilbert Matrix: 15513.738738929038
------------------------------------------------------------
n = 5

Condition number for the Hilbert Matrix: 476607.25024100044
------------------------------------------------------------
n = 6

Condition number for the Hilbert Matrix: 14951058.641453395
------------------------------------------------------------
n = 7

Condition number for the Hilbert Matrix: 475367356.9114392
------------------------------------------------------------
n = 8

Condition number for the Hilbert Matrix: 15257575566.627958
```

In [ ]:
```

```

MATH 5660 only:

Convergence of iterative methods like the Conjugate Gradient Method can be accelerated by the use of a technique called **preconditioning**. The convergence rates of iterative methods often depend, directly or indirectly, on the condition number of the coefficient matrix A. The idea of preconditioning is to reduce the effective condition number of the problem.

The preconditioned form of the $n \times n$ linear system $A\boldsymbol{x} = \boldsymbol{b}$ is

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b},$$

where M is an invertible $n \times n$ matrix called the **preconditioner**.

When A is a symmetric positive-definite $n \times n$ matrix, we will choose a symmetric positive-definite matrix M for use as a preconditioner. A particularly simple choice is the **Jacobi preconditioner** $M = D$, where D is the diagonal of A. The Preconditioned Conjugate Gradient Method is now easy to describe: Replace $A\boldsymbol{x} = \boldsymbol{b}$ with the preconditioned equation $M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}$, and replace the Euclidean inner product with $(\boldsymbol{v}, \boldsymbol{w})M$.

To convert Algorithm 2 in Section 3.3 to the preconditioned version, let $\boldsymbol{z}_k = M^{-1}\boldsymbol{b} - M^{-1}A\boldsymbol{x}_k = M^{-1}\boldsymbol{r}_k$. Then the algorithm is

```
1. Initialize $\boldsymbol{x}_0$ as any vector. Set $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$ and $\boldsymbol{u}_0 = \boldsymbol{z}_0 = M^{-1}\boldsymbol{r}_0$.
2. For $k = 0, 1, \ldots, n-1$:
    A. $a_k = \frac{\boldsymbol{r}_k^T \boldsymbol{z}_k}{\boldsymbol{u}_k^T A\boldsymbol{u}_k}$
    B. $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + a_k\boldsymbol{u}_k$
    C. $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - a_k A\boldsymbol{u}_k$
    D. if ($||\boldsymbol{r}_{k+1}|| < \epsilon$):
        a. break
    E. $\boldsymbol{z}_{k+1} = M^{-1}\boldsymbol{r}_{k+1}$
    F. $\boldsymbol{b}_k = \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{z}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{z}_k}$
    G. $\boldsymbol{u}_{k+1} = \boldsymbol{z}_{k+1} + \boldsymbol{b}_k\boldsymbol{u}_k$
3. return $\boldsymbol{x}_{k+1}$.
```

Now, consider the following problem.

Let A be the $n \times n$ matrix with $n = 1000$ and entries $A(i, i) = i$, $A(i, i+1) = A(i+1, i) = 1/2$, $A(i, i+2) = A(i+2, i) = 1/2$ for all i that fit within the matrix.

(a) Take a look at the nonzero structure of the matrix using plt.spy(A).

(b) Let $\boldsymbol{x}_e$ be the vector of n ones (exact solution). Set $\boldsymbol{b} = A\boldsymbol{x}_e$, and apply the Conjugate Gradient Method, without preconditioner, and with the Jacobi preconditioner. Compare errors (using 2-norm) of the two runs in a plot versus step number (using semilogy). (So you need to modify the conjugate gradient codes to keep track of and return the solutions of all steps.) Use eps = 1e-10.

The two methods may converge in different number of steps. Which one do you see is faster?