Part 1: creating the matrix and vector

```
!pip install petsc4py pyamg
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting petsc4py
  Downloading petsc4py-3.18.2.tar.gz (2.5 MB)
     |████████████████████████████████| 2.5 MB 5.3 MB/s
Collecting pyamg
  Downloading pyamg-4.2.3-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (1.7 MB)
     |████████████████████████████████| 1.7 MB 52.8 MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (from petsc4py) (1.21.6)
Collecting petsc<3.19,>=3.18
  Downloading petsc-3.18.2.tar.gz (18.4 MB)
     |████████████████████████████████| 18.4 MB 681 kB/s
Collecting mpi4py>=1.2.2
  Downloading mpi4py-3.1.4.tar.gz (2.5 MB)
     |████████████████████████████████| 2.5 MB 53.2 MB/s
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
Requirement already satisfied: scipy>=0.12.0 in /usr/local/lib/python3.8/dist-packages (from pyamg) (1.7.3)
Building wheels for collected packages: petsc4py, petsc, mpi4py
  Building wheel for petsc4py (setup.py) ... error
  ERROR: Failed building wheel for petsc4py
  Running setup.py clean for petsc4py
  Building wheel for petsc (setup.py) ... error
  ERROR: Failed building wheel for petsc
  Running setup.py clean for petsc
  Building wheel for mpi4py (PEP 517) ... done
  Created wheel for mpi4py: filename=mpi4py-3.1.4-cp38-cp38-linux_x86_64.whl size=4438492 sha256=7ff82730d027493f47607ca06735c2d4bc83c46956f5f1
  Stored in directory: /root/.cache/pip/wheels/f3/35/48/0b9a7076995eea5ea64a7e4bc3f0f342f453080795276264e7
Successfully built mpi4py
Failed to build petsc4py petsc
Installing collected packages: mpi4py, petsc, pyamg, petsc4py
    Running setup.py install for petsc ... done
  DEPRECATION: petsc was installed using the legacy 'setup.py install' method, because a wheel could not be built for it. A possible replacemen
    Running setup.py install for petsc4py ... done
  DEPRECATION: petsc4py was installed using the legacy 'setup.py install' method, because a wheel could not be built for it. A possible replace
Successfully installed mpi4py-3.1.4 petsc-3.18.2 petsc4py-3.18.2 pyamg-4.2.3
```

```
import numpy as np
import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from scipy.sparse.linalg import spsolve
from scipy.sparse import coo_matrix, linalg
from petsc4py import PETSc
import time
import random
import sympy

%matplotlib inline
```

```
def g(x, y):
    if (x == 0):
        return np.sin(4*y)

    if (y == 0):
        return np.sin(3*x)

    if (x == 1):
        return np.sin(3 + (4*y))

    if (y == 1):
        return np.sin(3*x + 4)
```

```
def create_mat_vec(N):
    if N <= 0:
        raise Exception('Invalid input')

    h = 1 / N
    k = 5

    size = (N-1)**2

    ij = (24 - 4 * (h**2) * (k**2)) / 9   # i=j
    hv = (-3 - (h**2) * (k**2)) / 9   # horizontally or vertically adjacent
    da = (-12 - (h**2) * (k**2)) / 36   # diagonally adjacent
```

```
    data = []
    row = []
    col = []

    for i in range(size):
        for j in range(size):
            if (i == j):
                data += [ij]
                row += [i]
                col += [j]
            elif (abs(i-j)==N-1 or (abs(i-j)==1 and (min(i,j)+1)%(N-1)!=0)):
                data += [hv]
                row += [i]
                col += [j]
            elif ((abs(i-j)==N or abs(i-j)==N-2) and abs((i//(N-1)+1)-(j//(N-1)+1))==1):
                data += [da]
                row += [i]
                col += [j]
            else:
                pass

    A = coo_matrix((data, (row, col)), (size, size)).tocsr()


    param_a = (12 + (h**2) * (k**2)) / 36
    param_b = (3 + (h**2) * (k**2)) / 9

    b = np.zeros((size, 1))

    for j in range(size):
        g_1 = 0
        g_2 = 0

        if (j == 0):   # (h, h)
            g_1 = g(0,0) + g(2*h,0) + g(0,2*h)
            g_2 = g(h,0) + g(0,h)
        elif (j == N-2):   # (1-h, h)
            g_1 = g(1,0) + g(1,2*h) + g(1-2*h,0)
            g_2 = g(1-h,0) + g(1,h)
        elif (j == (N-2)*(N-1)):   # (h, 1-h)
            g_1 = g(0,1) + g(2*h,1) + g(0,1-2*h)
            g_2 = g(h,1) + g(0,1-h)
        elif (j == (N-1)*(N-1)-1):   # (1-h, 1-h)
            g_1 = g(1,1) + g(1-2*h,1) + g(1,1-2*h)
            g_2 = g(1-h,1) + g(1,1-h)

        elif (j%(N-1) == 0 and (j != (N-2)*(N-1) and j != 0)):   # (h, c)
            c = ((j // (N-1)) + 1) * h
            g_1 = g(0,c+h) + g(0,c-h)
            g_2 = g(0,c)
        elif ((j+1)%(N-1) == 0 and (j != N-2 and j != (N-1)*(N-1)-1)):   # (1-h, c)
            c = ((j+1) // (N-1)) * h
            g_1 = g(1,c+h) + g(1,c-h)
            g_2 = g(1,c)
        elif (j < N-1 and (j != N-2 and j != 0)):   # (c, h)
            c = (j + 1) * h
            g_1 = g(c+h,0) + g(c-h,0)
            g_2 = g(c,0)
        elif (j > (N-2)*(N-1)-1 and (j != (N-2)*(N-1) and j != (N-1)*(N-1)-1)):   # (c, 1-h)
            c = (j - (N-2)*(N-1) + 1) * h
            g_1 = g(c+h,1) + g(c-h,1)
            g_2 = g(c,1)

        b[j] = param_a * g_1 + param_b * g_2


    return A, b
```

```
N = 3
A, b = create_mat_vec(N)
A.todense(), b
```

```
    (matrix([[ 1.43209877, -0.64197531, -0.64197531, -0.41049383],
             [-0.64197531,  1.43209877, -0.41049383, -0.64197531],
             [-0.64197531, -0.41049383,  1.43209877, -0.64197531],
             [-0.41049383, -0.64197531, -0.64197531,  1.43209877]]),
     array([[ 1.7251323 ],
            [ 0.15334285],
            [-0.34843455],
            [-1.05586512]]))
```

```
A_3 = np.array([
        [1.4320987654320987, -0.6419753086419753, -0.6419753086419753, -0.4104938271604938],
        [-0.6419753086419753, 1.4320987654320987, -0.4104938271604938, -0.6419753086419753],
        [-0.6419753086419753, -0.4104938271604938, 1.4320987654320987, -0.6419753086419753],
        [-0.4104938271604938, -0.6419753086419753, -0.6419753086419753, 1.4320987654320987],
])
b_3 = np.array([[1.7251323007221917], [0.15334285313223067],
                [-0.34843455260733003], [-1.0558651156722307]])
```

```
assert (np.around(A.todense(), 10) == np.around(A_3, 10)).all()
assert (np.around(b, 10) == np.around(b_3, 10)).all()
```

```
N = 4
A, b = create_mat_vec(N)
A.todense(), b
```

```
(matrix([[ 1.97222222, -0.50694444,  0.        , -0.50694444, -0.37673611,
           0.        ,  0.        ,  0.        ,  0.        ],
         [-0.50694444,  1.97222222, -0.50694444, -0.37673611, -0.50694444,
          -0.37673611,  0.        ,  0.        ,  0.        ],
         [ 0.        , -0.50694444,  1.97222222,  0.        , -0.37673611,
          -0.50694444,  0.        ,  0.        ,  0.        ],
         [-0.50694444, -0.37673611,  0.        ,  1.97222222, -0.50694444,
           0.        , -0.50694444, -0.37673611,  0.        ],
         [-0.37673611, -0.50694444, -0.37673611, -0.50694444,  1.97222222,
          -0.50694444, -0.37673611, -0.50694444, -0.37673611],
         [ 0.        , -0.37673611, -0.50694444,  0.        , -0.50694444,
           1.97222222,  0.        , -0.37673611, -0.50694444],
         [ 0.        ,  0.        ,  0.        , -0.50694444, -0.37673611,
           0.        ,  1.97222222, -0.50694444,  0.        ],
         [ 0.        ,  0.        ,  0.        , -0.37673611, -0.50694444,
          -0.37673611, -0.50694444,  1.97222222, -0.50694444],
         [ 0.        ,  0.        ,  0.        ,  0.        , -0.37673611,
          -0.50694444,  0.        , -0.50694444,  1.97222222]]),
 array([[ 1.49048958],
        [ 1.05560075],
        [ 0.07847905],
        [ 0.83114079],
        [ 0.        ],
        [-0.87650207],
        [-0.64339809],
        [-0.74663924],
        [-0.5380215 ]]))
```

```
A_4 = np.array([
        [1.972222222222222, -0.5069444444444444, 0.0, -0.5069444444444444,
          -0.3767361111111111, 0.0, 0.0, 0.0, 0.0],
        [-0.5069444444444444, 1.972222222222222, -0.5069444444444444, -0.3767361111111111,
          -0.5069444444444444, -0.3767361111111111, 0.0, 0.0, 0.0],
        [0.0, -0.5069444444444444, 1.972222222222222, 0.0,
          -0.3767361111111111, -0.5069444444444444, 0.0, 0.0, 0.0],
        [-0.5069444444444444, -0.3767361111111111, 0.0, 1.972222222222222,
          -0.5069444444444444, 0.0, -0.5069444444444444, -0.3767361111111111, 0.0],
        [-0.3767361111111111, -0.5069444444444444, -0.3767361111111111, -0.5069444444444444,
          1.972222222222222, -0.5069444444444444, -0.3767361111111111, -0.5069444444444444, -0.3767361111111111],
        [0.0, -0.3767361111111111, -0.5069444444444444, 0.0, -0.5069444444444444,
          1.972222222222222, 0.0, -0.3767361111111111, -0.5069444444444444],
        [0.0, 0.0, 0.0, -0.5069444444444444, -0.3767361111111111, 0.0,
          1.972222222222222, -0.5069444444444444, 0.0],
        [0.0, 0.0, 0.0, -0.3767361111111111, -0.5069444444444444, -0.3767361111111111,
          -0.5069444444444444, 1.972222222222222, -0.5069444444444444],
        [0.0, 0.0, 0.0, 0.0, -0.3767361111111111, -0.5069444444444444, 0.0,
          -0.5069444444444444, 1.972222222222222],
])
b_4 = np.array([[1.4904895819530766], [1.055600747809247], [0.07847904705126368],
        [0.8311407883427149], [0.0], [-0.8765020708205272], [-0.6433980946818605],
        [-0.7466392365712349], [-0.538021498324083]])
```

```
assert (np.around(A.todense(), 10) == np.around(A_4, 10)).all()
assert (np.around(b, 10) == np.around(b_4, 10)).all()
```

Part 2: solving the system

```
# when N = 4

N = 4
h = 1 / N
A, b = create_mat_vec(N)
```

```
u = spsolve(A, b)
u
```

```
array([ 0.8456696 ,  0.45257486, -0.17716394,  0.25153569, -0.47668947,
       -0.94237126, -0.61312343, -1.01341685, -0.86657757])
```

```
sol = np.flip(u.reshape((N-1, N-1)), 0)
sol
```

```
array([[-0.61312343, -1.01341685, -0.86657757],
       [ 0.25153569, -0.47668947, -0.94237126],
       [ 0.8456696 ,  0.45257486, -0.17716394]])
```

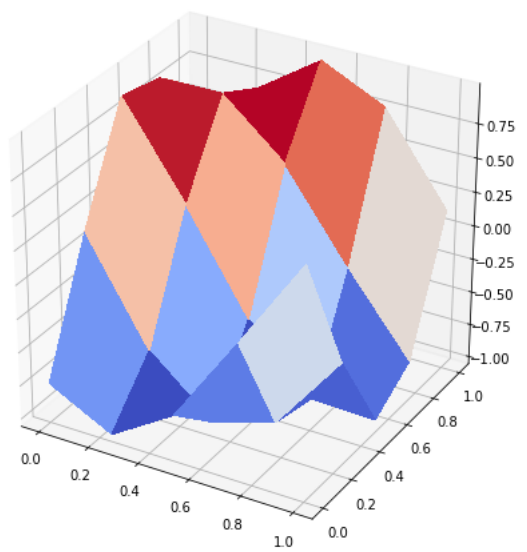
```
Z = np.pad(sol, ((1,1), (1,1)), 'constant')
Z
```

```
array([[ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        , -0.61312343, -1.01341685, -0.86657757,  0.        ],
       [ 0.        ,  0.25153569, -0.47668947, -0.94237126,  0.        ],
       [ 0.        ,  0.8456696 ,  0.45257486, -0.17716394,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ]])
```

```
for i in range(Z.shape[0]-1, -1, -1):
    x = 0
    y = (N - i) * h
    Z[i][0] = g(x, y)

    x = 1
    Z[i][-1] = g(x, y)

for i in range(1, Z.shape[1]-1):
    x = i * h
    y = 0
    Z[-1][i] = g(x, y)

    y = 1
    Z[0][i] = g(x, y)

Z
```

```
array([[-0.7568025 , -0.99929279, -0.70554033, -0.03317922,  0.6569866 ],
       [ 0.14112001, -0.61312343, -1.01341685, -0.86657757, -0.2794155 ],
       [ 0.90929743,  0.25153569, -0.47668947, -0.94237126, -0.95892427],
       [ 0.84147098,  0.8456696 ,  0.45257486, -0.17716394, -0.7568025 ],
       [ 0.        ,  0.68163876,  0.99749499,  0.7780732 ,  0.14112001]])
```

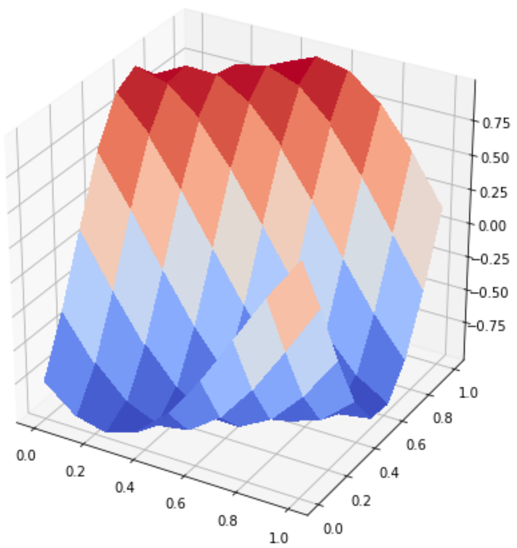
```
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
ticks= np.linspace(0, 1, N+1)
X, Y = np.meshgrid(ticks, ticks)
surf = ax.plot_surface(X, Y, Z, antialiased=False, cmap=cm.coolwarm)
```




```
# when N = 8

N = 8
h = 1 / N
A, b = create_mat_vec(N)
```

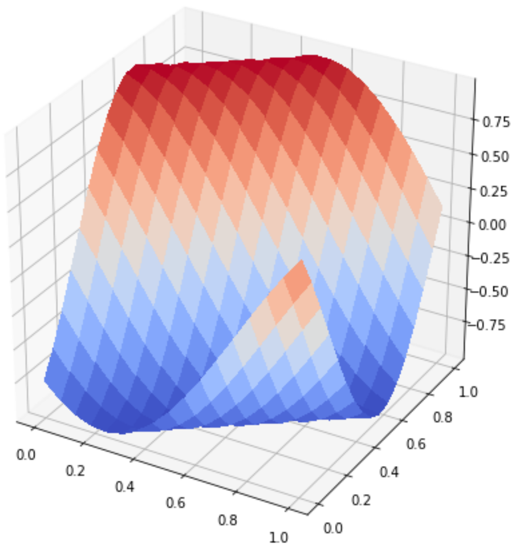
```
u = spsolve(A, b)

sol = np.flip(u.reshape((N-1, N-1)), 0)

Z = np.pad(sol, ((1,1), (1,1)), 'constant')

for i in range(Z.shape[0]-1, -1, -1):
    x = 0
    y = (N - i) * h
    Z[i][0] = g(x, y)

    x = 1
    Z[i][-1] = g(x, y)

for i in range(1, Z.shape[1]-1):
    x = i * h
    y = 0
    Z[-1][i] = g(x, y)

    y = 1
    Z[0][i] = g(x, y)
```

```
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
ticks= np.linspace(0, 1, N+1)
X, Y = np.meshgrid(ticks, ticks)
surf = ax.plot_surface(X, Y, Z, antialiased=False, cmap=cm.coolwarm)
```




```
# when N = 16

N = 16
h = 1 / N
A, b = create_mat_vec(N)
u = spsolve(A, b)

sol = np.flip(u.reshape((N-1, N-1)), 0)

Z = np.pad(sol, ((1,1), (1,1)), 'constant')

for i in range(Z.shape[0]-1, -1, -1):
    x = 0
    y = (N - i) * h
    Z[i][0] = g(x, y)

    x = 1
    Z[i][-1] = g(x, y)

for i in range(1, Z.shape[1]-1):
    x = i * h
    y = 0
    Z[-1][i] = g(x, y)

    y = 1
    Z[0][i] = g(x, y)
```

```
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
ticks= np.linspace(0, 1, N+1)
X, Y = np.meshgrid(ticks, ticks)
surf = ax.plot_surface(X, Y, Z, antialiased=False, cmap=cm.coolwarm)
```



Part 3: comparing solvers and preconditioners

In this section, I will compare 5 matrix-vector solvers:

- iterative solvers: BiCGSTAB (bcgs), Generalized Minimal Residual (gmres), and Conjugate Gradient (cg)
  - with three preconditioner: SOR (sor), Incomplete LU (ilu), and Additive Schwarz (asm)
- direct solvers: LU, and Cholesky

Moreover, the comparison will include the following three aspects:
a). time taken by the solver
b). number of iterations taken by an iterative solver, and
c). the size of the residual after each iteration.

```
def generate_mat(n):
    nnz = 3 * np.ones(n, dtype=np.int32)
    nnz[0] = nnz[-1] = 2

    A = PETSc.Mat()
    A.createAIJ([n, n], nnz=nnz)

    for i in range(n):
        A.setValue(i, i, 3)
    for i in range(n - 1):
        A.setValue(i, i + 1, -1)
        A.setValue(i + 1, i, -1)

    A.assemble()

    b = A.createVecLeft()
    b.array[:] = 1.0

    x = A.createVecRight()

    return A, b, x
```

```
list_N = np.linspace(10, 100000, 30, dtype=np.int32)
```

Compare iterative solvers with different preconditioner

```
# bcgs

iterationNum_bcgs = []
time_bcgs = []
```

```
for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('bcgs')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('none')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_bcgs.append(ksp.getIterationNumber())
    time_bcgs.append(run_time)

residualSize_bcgs = ksp.getConvergenceHistory()
```

```
# bcgs with sor

iterationNum_bcgs_sor = []
time_bcgs_sor = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('bcgs')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('sor')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_bcgs_sor.append(ksp.getIterationNumber())
    time_bcgs_sor.append(run_time)

residualSize_bcgs_sor = ksp.getConvergenceHistory()
```

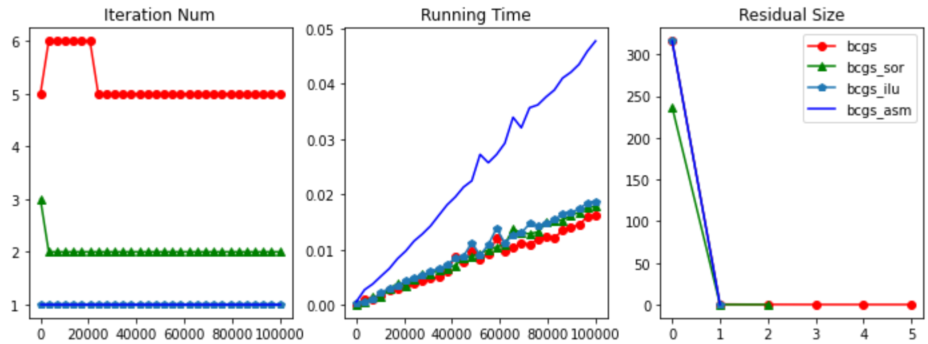
```
# bcgs with ilu

iterationNum_bcgs_ilu = []
time_bcgs_ilu = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('bcgs')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('ilu')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_bcgs_ilu.append(ksp.getIterationNumber())
    time_bcgs_ilu.append(run_time)

residualSize_bcgs_ilu = ksp.getConvergenceHistory()
```

```
# bcgs with asm

iterationNum_bcgs_asm = []
time_bcgs_asm = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('bcgs')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('asm')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
```

```
    run_time = time_end - time_start

    iterationNum_bcgs_asm.append(ksp.getIterationNumber())
    time_bcgs_asm.append(run_time)

residualSize_bcgs_asm = ksp.getConvergenceHistory()
```

```
plt.figure(figsize=(12,4))

ax = plt.subplot(1, 3, 1)
ax.plot(list_N, iterationNum_bcgs, "ro-")
ax.plot(list_N, iterationNum_bcgs_sor, "g^-")
ax.plot(list_N, iterationNum_bcgs_ilu, "p-")
ax.plot(list_N, iterationNum_bcgs_asm, "b")
ax.set_title("Iteration Num")

ax = plt.subplot(1, 3, 2)
ax.plot(list_N, time_bcgs, "ro-")
ax.plot(list_N, time_bcgs_sor, "g^-")
ax.plot(list_N, time_bcgs_ilu, "p-")
ax.plot(list_N, time_bcgs_asm, "b")
ax.set_title("Running Time")

ax = plt.subplot(1, 3, 3)
ax.plot(range(iterationNum_bcgs[-1]+1), residualSize_bcgs, "ro-")
ax.plot(range(iterationNum_bcgs_sor[-1]+1), residualSize_bcgs_sor, "g^-")
ax.plot(range(iterationNum_bcgs_ilu[-1]+1), residualSize_bcgs_ilu, "p-")
ax.plot(range(iterationNum_bcgs_asm[-1]+1), residualSize_bcgs_asm, "b")
ax.set_title("Residual Size")
plt.legend(['bcgs', 'bcgs_sor', 'bcgs_ilu', 'bcgs_asm'])
```

```
<matplotlib.legend.Legend at 0x7fc8c66c9430>
```




```
# CG

iterationNum_cg = []
time_cg = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('cg')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('none')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_cg.append(ksp.getIterationNumber())
    time_cg.append(run_time)

residualSize_cg = ksp.getConvergenceHistory()
```

```
# cg with sor

iterationNum_cg_sor = []
time_cg_sor = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('cg')
```

```
    ksp.setConvergenceHistory()
    ksp.getPC().setType('sor')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_cg_sor.append(ksp.getIterationNumber())
    time_cg_sor.append(run_time)

residualSize_cg_sor = ksp.getConvergenceHistory()
```

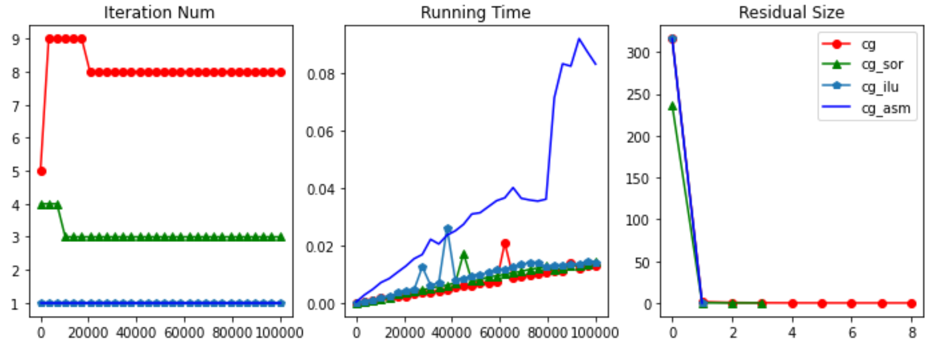
```
# cg with ilu

iterationNum_cg_ilu = []
time_cg_ilu = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('cg')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('ilu')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_cg_ilu.append(ksp.getIterationNumber())
    time_cg_ilu.append(run_time)

residualSize_cg_ilu = ksp.getConvergenceHistory()
```

```
# cg with asm

iterationNum_cg_asm = []
time_cg_asm = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('cg')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('asm')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_cg_asm.append(ksp.getIterationNumber())
    time_cg_asm.append(run_time)

residualSize_cg_asm = ksp.getConvergenceHistory()
```

```
plt.figure(figsize=(12,4))

ax = plt.subplot(1, 3, 1)
ax.plot(list_N, iterationNum_cg, "ro-")
ax.plot(list_N, iterationNum_cg_sor, "g^-")
ax.plot(list_N, iterationNum_cg_ilu, "p-")
ax.plot(list_N, iterationNum_cg_asm, "b")
ax.set_title("Iteration Num")

ax = plt.subplot(1, 3, 2)
ax.plot(list_N, time_cg, "ro-")
ax.plot(list_N, time_cg_sor, "g^-")
ax.plot(list_N, time_cg_ilu, "p-")
ax.plot(list_N, time_cg_asm, "b")
ax.set_title("Running Time")

ax = plt.subplot(1, 3, 3)
ax.plot(range(iterationNum_cg[-1]+1), residualSize_cg, "ro-")
ax.plot(range(iterationNum_cg_sor[-1]+1), residualSize_cg_sor, "g^-")
ax.plot(range(iterationNum_cg_ilu[-1]+1), residualSize_cg_ilu, "p-")
ax.plot(range(iterationNum_cg_asm[-1]+1), residualSize_cg_asm, "b")
```

```
ax.set_title("Residual  Size")
plt.legend(['cg',  'cg_sor',  'cg_ilu',  'cg_asm'])
```

<matplotlib.legend.Legend at 0x7fc8c657dfd0>




```
#  gmres

iterationNum_gmres  =  []
time_gmres  =  []

for  n  in  list_N:
    A,  b,  x  =  generate_mat(n)
    ksp  =  PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('gmres')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('none')

    time_start  =  time.time()
    ksp.solve(b,  x)
    time_end  =  time.time()
    run_time  =  time_end  -  time_start

    iterationNum_gmres.append(ksp.getIterationNumber())
    time_gmres.append(run_time)

residualSize_gmres  =  ksp.getConvergenceHistory()
```

```
#  gmres  with  sor

iterationNum_gmres_sor  =  []
time_gmres_sor  =  []

for  n  in  list_N:
    A,  b,  x  =  generate_mat(n)
    ksp  =  PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('gmres')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('sor')

    time_start  =  time.time()
    ksp.solve(b,  x)
    time_end  =  time.time()
    run_time  =  time_end  -  time_start

    iterationNum_gmres_sor.append(ksp.getIterationNumber())
    time_gmres_sor.append(run_time)

residualSize_gmres_sor  =  ksp.getConvergenceHistory()
```

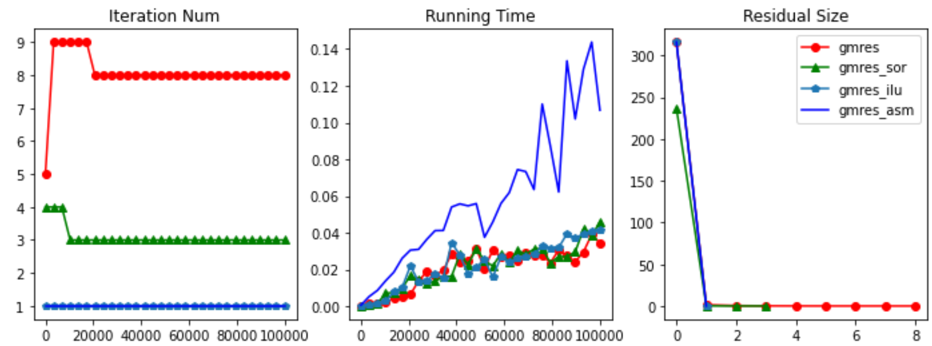
```
#  gmres  with  ilu

iterationNum_gmres_ilu  =  []
time_gmres_ilu  =  []

for  n  in  list_N:
    A,  b,  x  =  generate_mat(n)
    ksp  =  PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('gmres')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('ilu')

    time_start  =  time.time()
    ksp.solve(b,  x)
    time_end  =  time.time()
    run_time  =  time_end  -  time_start
```

```
    iterationNum_gmres_ilu.append(ksp.getIterationNumber())
    time_gmres_ilu.append(run_time)

residualSize_gmres_ilu = ksp.getConvergenceHistory()
```

```
# gmres with asm

iterationNum_gmres_asm = []
time_gmres_asm = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('gmres')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('asm')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    iterationNum_gmres_asm.append(ksp.getIterationNumber())
    time_gmres_asm.append(run_time)

residualSize_gmres_asm = ksp.getConvergenceHistory()
```

```
plt.figure(figsize=(12,4))

ax = plt.subplot(1, 3, 1)
ax.plot(list_N, iterationNum_gmres, "ro-")
ax.plot(list_N, iterationNum_gmres_sor, "g^-")
ax.plot(list_N, iterationNum_gmres_ilu, "p-")
ax.plot(list_N, iterationNum_gmres_asm, "b")
ax.set_title("Iteration Num")

ax = plt.subplot(1, 3, 2)
ax.plot(list_N, time_gmres, "ro-")
ax.plot(list_N, time_gmres_sor, "g^-")
ax.plot(list_N, time_gmres_ilu, "p-")
ax.plot(list_N, time_gmres_asm, "b")
ax.set_title("Running Time")

ax = plt.subplot(1, 3, 3)
ax.plot(range(iterationNum_gmres[-1]+1), residualSize_gmres, "ro-")
ax.plot(range(iterationNum_gmres_sor[-1]+1), residualSize_gmres_sor, "g^-")
ax.plot(range(iterationNum_gmres_ilu[-1]+1), residualSize_gmres_ilu, "p-")
ax.plot(range(iterationNum_gmres_asm[-1]+1), residualSize_gmres_asm, "b")
ax.set_title("Residual Size")
plt.legend(['gmres', 'gmres_sor', 'gmres_ilu', 'gmres_asm'])
```

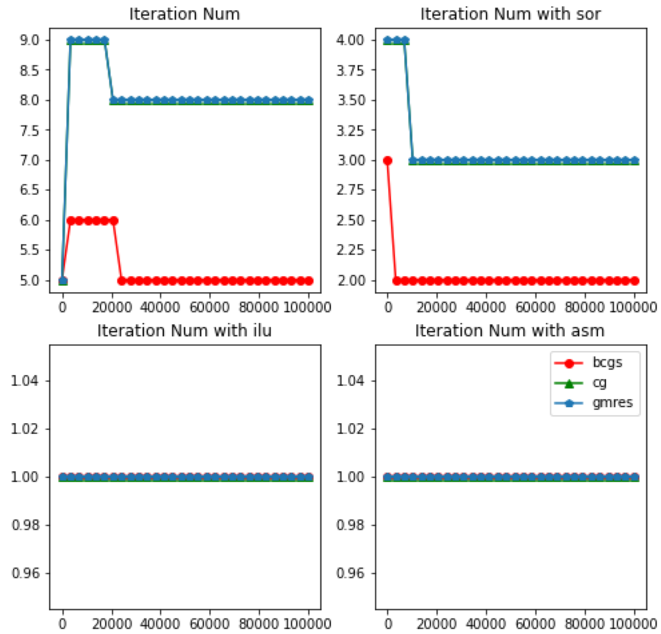
```
<matplotlib.legend.Legend at 0x7fc8c6436b20>
```




```
# compare Iteration Num

plt.figure(figsize=(8,8))

ax = plt.subplot(2, 2, 1)
ax.plot(list_N, iterationNum_bcgs, "ro-")
ax.plot(list_N, iterationNum_cg, "g^-")
ax.plot(list_N, iterationNum_gmres, "p-")
ax.set_title("Iteration Num")\

ax = plt.subplot(2, 2, 2)
```

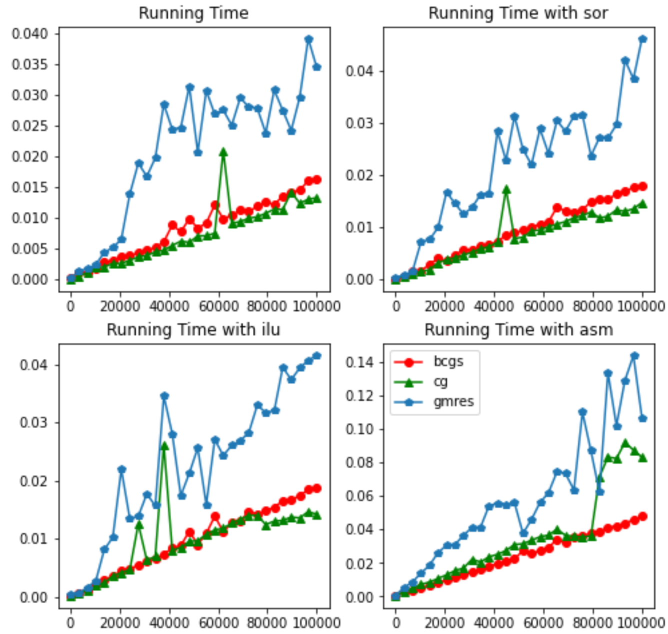
```
ax.plot(list_N, iterationNum_bcgs_sor, "ro-")
ax.plot(list_N, iterationNum_cg_sor, "g^-")
ax.plot(list_N, iterationNum_gmres_sor, "p-")
ax.set_title("Iteration Num with sor")

ax = plt.subplot(2, 2, 3)
ax.plot(list_N, iterationNum_bcgs_ilu, "ro-")
ax.plot(list_N, iterationNum_cg_ilu, "g^-")
ax.plot(list_N, iterationNum_gmres_ilu, "p-")
ax.set_title("Iteration Num with ilu")

ax = plt.subplot(2, 2, 4)
ax.plot(list_N, iterationNum_bcgs_asm, "ro-")
ax.plot(list_N, iterationNum_cg_asm, "g^-")
ax.plot(list_N, iterationNum_gmres_asm, "p-")
ax.set_title("Iteration Num with asm")

plt.legend(['bcgs', 'cg', 'gmres'])
```

<matplotlib.legend.Legend at 0x7fc8c5d14ca0>



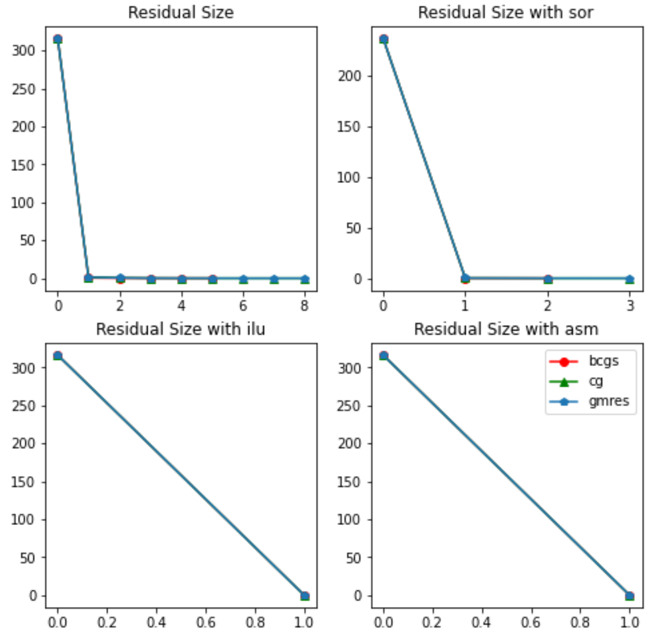

```
# compare Running Time

plt.figure(figsize=(8,8))

ax = plt.subplot(2, 2, 1)
ax.plot(list_N, time_bcgs, "ro-")
ax.plot(list_N, time_cg, "g^-")
ax.plot(list_N, time_gmres, "p-")
ax.set_title("Running Time")

ax = plt.subplot(2, 2, 2)
ax.plot(list_N, time_bcgs_sor, "ro-")
ax.plot(list_N, time_cg_sor, "g^-")
ax.plot(list_N, time_gmres_sor, "p-")
ax.set_title("Running Time with sor")

ax = plt.subplot(2, 2, 3)
ax.plot(list_N, time_bcgs_ilu, "ro-")
ax.plot(list_N, time_cg_ilu, "g^-")
ax.plot(list_N, time_gmres_ilu, "p-")
ax.set_title("Running Time with ilu")

ax = plt.subplot(2, 2, 4)
ax.plot(list_N, time_bcgs_asm, "ro-")
ax.plot(list_N, time_cg_asm, "g^-")
ax.plot(list_N, time_gmres_asm, "p-")
ax.set_title("Running Time with asm")

plt.legend(['bcgs', 'cg', 'gmres'])
```

<matplotlib.legend.Legend at 0x7fc8c5be7e20>




```
# compare Residual Size

plt.figure(figsize=(8,8))

ax = plt.subplot(2, 2, 1)
ax.plot(range(iterationNum_bcgs[-1]+1), residualSize_bcgs, "ro-")
ax.plot(range(iterationNum_cg[-1]+1), residualSize_cg, "g^-")
ax.plot(range(iterationNum_gmres[-1]+1), residualSize_gmres, "p-")
ax.set_title("Residual Size")

ax = plt.subplot(2, 2, 2)
ax.plot(range(iterationNum_bcgs_sor[-1]+1), residualSize_bcgs_sor, "ro-")
ax.plot(range(iterationNum_cg_sor[-1]+1), residualSize_cg_sor, "g^-")
ax.plot(range(iterationNum_gmres_sor[-1]+1), residualSize_gmres_sor, "p-")
ax.set_title("Residual Size with sor")

ax = plt.subplot(2, 2, 3)
ax.plot(range(iterationNum_bcgs_ilu[-1]+1), residualSize_bcgs_ilu, "ro-")
ax.plot(range(iterationNum_cg_ilu[-1]+1), residualSize_cg_ilu, "g^-")
ax.plot(range(iterationNum_gmres_ilu[-1]+1), residualSize_gmres_ilu, "p-")
ax.set_title("Residual Size with ilu")

ax = plt.subplot(2, 2, 4)
ax.plot(range(iterationNum_bcgs_asm[-1]+1), residualSize_bcgs_asm, "ro-")
ax.plot(range(iterationNum_cg_asm[-1]+1), residualSize_cg_asm, "g^-")
ax.plot(range(iterationNum_gmres_asm[-1]+1), residualSize_gmres_asm, "p-")
ax.set_title("Residual Size with asm")

plt.legend(['bcgs', 'cg', 'gmres'])
```

<matplotlib.legend.Legend at 0x7fc8c3612970>



According to the images above, we can see that, for each iterative solver, the iteration number of Incomplete LU is the least, and its running time is also better than others; only residual size is larger at the beginning, but in the end it will also get a result that is not inferior to the others.
What's more, for preconditioner of Incomplete LU, which solver is used has little effect on running time and residual size, but the running time is shortest compared to other solvers' when solver CG is used.
Therefore, the solver of CG with preconditioner Incomplete LU showed the best comprehensive performance. Then we will compare it to direct solvers.

Compareison and pick the best solver

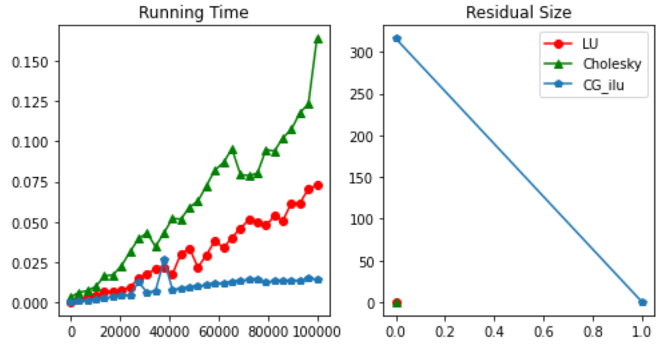
```
# Direct solver LU

time_lu = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType("preonly")
    ksp.setConvergenceHistory()
    ksp.getPC().setType("lu")

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    time_lu.append(run_time)

residual = A * x - b
residualSize_lu = residual.norm() / b.norm()
```

```
# Direct solver Cholesky

time_cholesky = []

for n in list_N:
    A, b, x = generate_mat(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType("preonly")
    ksp.setConvergenceHistory()
    ksp.getPC().setType("cholesky")

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    time_cholesky.append(run_time)
```

```
residual = A * x - b
residualSize_cholesky = residual.norm() / b.norm()
```

```
plt.figure(figsize=(8,4))

ax = plt.subplot(1, 2, 1)
ax.plot(list_N, time_lu, "ro-")
ax.plot(list_N, time_cholesky, "g^-")
ax.plot(list_N, time_cg_ilu, "p-")
ax.set_title("Running Time")

ax = plt.subplot(1, 2, 2)
ax.plot(range(1), residualSize_lu, "ro-")
ax.plot(range(1), residualSize_cholesky, "g^-")
ax.plot(range(iterationNum_cg_ilu[-1]+1), residualSize_cg_ilu, "p-")
ax.set_title("Residual Size")
plt.legend(['LU', 'Cholesky', 'CG_ilu'])
```

```
<matplotlib.legend.Legend at 0x7fc8c2d81c70>
```



According to the comparison above, I think the solver of CG with preconditioner Incomplete LU is the most appropriate to solve this matrix-vector problem. Because their all can get similar final result, but the running time of CG_ilu does not increase significantly as N increases. Therefore, I will pick solver of CG with preconditioner Incomplete LU.

Part 4: increasing N

```
def generate_matrix(N):
    A_, b_ = create_mat_vec(N)
    A_ = A_.todense()

    n = A_.shape[0]
    nnz = 3 * np.ones(n, dtype=np.int32)
    nnz[0] = nnz[-1] = 2

    A = PETSc.Mat()
    A.createAIJ([n, n], nnz=nnz)
    A.setOption(PETSc.Mat.Option.NEW_NONZERO_LOCATION_ERR, False)

    for i in range(A_.shape[0]):
        for j in range(A_.shape[1]):
            A.setValue(i, j, A_[i,j])
    A.assemble()

    b = A.createVecLeft()
    for i in range(b_.shape[0]):
        b.array[i] = b_[i]

    x = A.createVecRight()

    return A, b, x
```

```
n = 10
A, b, x = generate_matrix(n)
ksp = PETSc.KSP().create()
ksp.setOperators(A)
ksp.setType('cg')
ksp.setConvergenceHistory()
ksp.getPC().setType('ilu')

ksp.solve(b, x)
u_1 = ksp.getSolution()[:]
```

```
A_, b_ = create_mat_vec(n)
u_2 = spsolve(A_, b_)
```

```
# check if get right result
assert (np.round(u_1,10) == np.round(u_2,10)).all()
```

```
def h_u(mat, N):
    h = 1 / N

    mat = np.flip(mat.reshape((n-1, n-1)), 0)
    mat = np.pad(mat, ((1,1), (1,1)), 'constant')

    for i in range(mat.shape[0]-1, -1, -1):
        x = 0
        y = (n - i) * h
        mat[i][0] = g(x, y)

        x = 1
        mat[i][-1] = g(x, y)

    for i in range(1, mat.shape[1]-1):
        x = i * h
        y = 0
        mat[-1][i] = g(x, y)

        y = 1
        mat[0][i] = g(x, y)

    return mat
```

```
def exact_u(N):
    h = 1 / N

    mat = np.zeros((N+1, N+1))

    for i in range(mat.shape[0]-1, -1, -1):
        for j in range(mat.shape[1]):
            x = j * h
            y = (N - i) * h
            mat[i][j] = np.sin(3 * x + 4 * y)

    return mat
```

```
def get_error(u_exact, u_h, N):
    e = 0.0
    size = N ** 2
    h = 1 / N

    for i in range(size):
        r = i // N
        l = i - (r * N)
        u_exact_m = np.mean([u_exact[r][l],u_exact[r][l+1],u_exact[r+1][l],u_exact[r+1][l+1]])
        u_h_m = np.mean([u_h[r][l],u_h[r][l+1],u_h[r+1][l],u_h[r+1][l+1]])
        e += (h**2) * abs(u_exact_m - u_h_m)

    return e
```

```
list_N = [3, 4, 5, 10, 15, 27, 39, 51]
```

```
time_list = []
error_list = []

for n in list_N:
    A, b, x = generate_matrix(n)
    ksp = PETSc.KSP().create()
    ksp.setOperators(A)
    ksp.setType('cg')
    ksp.setConvergenceHistory()
    ksp.getPC().setType('ilu')

    time_start = time.time()
    ksp.solve(b, x)
    time_end = time.time()
    run_time = time_end - time_start

    u_h = ksp.getSolution()[:]
    u_h = h_u(u_h, n)
    u_exact = exact_u(n)
```

```
    error = get_error(u_exact, u_h, n)

    error_list.append(error)
    time_list.append(run_time)
```

```
list_N
```

    [3, 4, 5, 10, 15, 27, 39, 51]

```
error_list
```

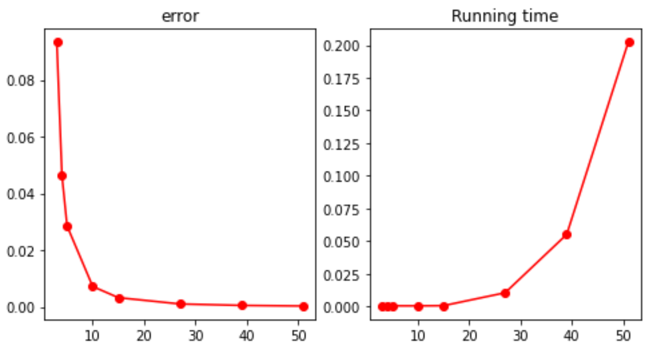
    [0.093303481498384,
     0.046355487125902956,
     0.028471845351189573,
     0.007149662920574294,
     0.003215703362153086,
     0.0010000111295221424,
     0.0004802273989314084,
     0.00028103136728383486]

```
time_list
```

    [0.0005681514739990234,
     0.0003781318664550781,
     0.0005180835723876953,
     0.0004868507385253906,
     0.000629425048828125,
     0.010500431060791016,
     0.05514883995056152,
     0.20235443115234375]

```
plt.figure(figsize=(8,4))

ax = plt.subplot(1, 2, 1)
plt.plot(list_N, error_list, "ro-")
ax.set_title("error")

ax = plt.subplot(1, 2, 2)
plt.plot(list_N, time_list, "ro-")
ax.set_title("Running time")
```

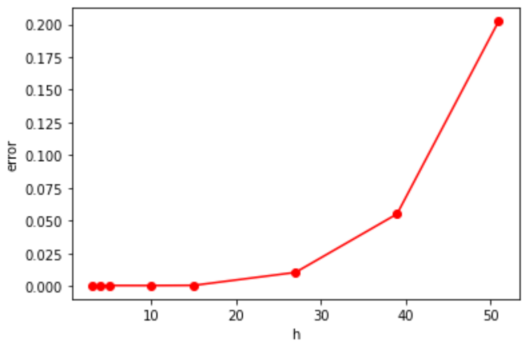
    Text(0.5, 1.0, 'Running time')



Estimate the complexity of the solver

```
plt.plot(list_N, time_list, "ro-")
plt.xlabel("h")
plt.ylabel("error")
```

    Text(0, 0.5, 'error')



Since the figure above is a parabola, I guess the time complexity is $O(n^2)$, and the equation for the parabola is $a \times x^2 + b = y$.

```
a = sympy.Symbol('a')
b = sympy.Symbol("b")
```

```
sol = []

for i in range(1, len(list_N)):
    result = sympy.solve([a*(list_N[0]**2)+b-time_list[0],
               a*(list_N[i]**2)+b-time_list[i]],a,b)
    sol.append(result)

sol
```

```
[{a: -2.71456582205636e-5, b: 0.000812462397984095},
 {a: -3.12924385070800e-6, b: 0.000596314668655395},
 {a: -8.93414675534418e-7, b: 0.000576192206078833},
 {a: 2.83673957542139e-7, b: 0.000565598408381144},
 {a: 1.37948327594333e-5, b: 0.000443997979164123},
 {a: 3.60983389395254e-5, b: 0.000243266423543294},
 {a: 7.78496449376331e-5, b: -0.000132495330439675}]
```

We can get solution for $a \times x^2 + b = y$. Therefore, the time complexity for this solver is $O(n^2)$.

Estimate the order of convergence

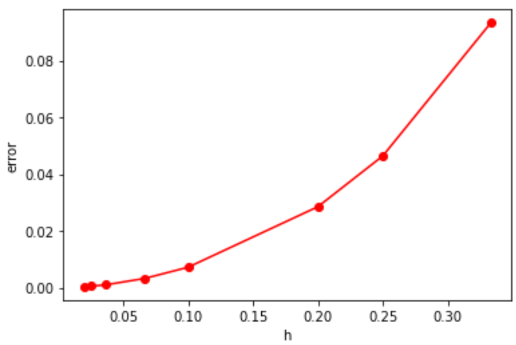
```
h_list = np.divide(1.0, list_N)
h_list
```

```
array([0.33333333, 0.25      , 0.2       , 0.1       , 0.06666667,
       0.03703704, 0.02564103, 0.01960784])
```

Since the relationship between h and $error$ is $error(h) = Ch^p$, where h is the interval size, C is a constant, and p is the order of convergence. So, we will draw a $h - error$ diagram to reveal the order of convergence p.

```
plt.plot(h_list, error_list, "ro-")
plt.xlabel("h")
plt.ylabel("error")
```

```
Text(0, 0.5, 'error')
```

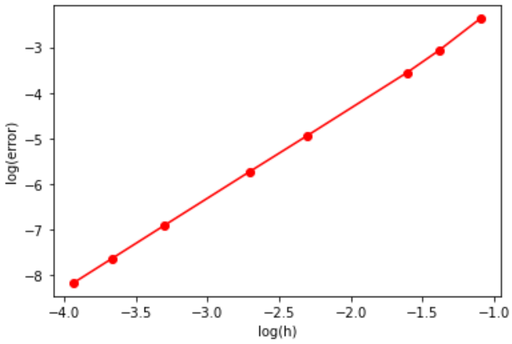


The figure above gives a parabola, so the order of convergence p is seem to be 2. However, we still need to draw a $log(h) - log(error)$ diagram to get the exact p value.

Because $error = Ch^p$, then we can deduce that $log(error) = log(C) + p\ log(h)$, and the $log(h) - log(error)$ diagram will produce a straight line with slope p and offset $logC$.

```
plt.plot(np.log(h_list), np.log(error_list), "ro-")
plt.xlabel("log(h)")
plt.ylabel("log(error)")
```

Text(0, 0.5, 'log(error)')




```
a = sympy.Symbol('a')
b = sympy.Symbol("b")
```

```
result = sympy.solve([a*np.log(h_list)[0]+b-np.log(error_list)[0],
          a*np.log(h_list)[-1]+b-np.log(error_list)[-1]],a,b)
result[a]
```

2.04896197561845

Thus, we can get the exact value for the order of convergence p, which is 2.049. Therefore, the order of convergence of the solution is 2.

Part 5: parallelisation

The algorithm of Conjugate Gradients is shown as below:

$$d_0 = r_0 = b - Ax_0$$
$$\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i}$$
$$x_{i+1} = x_i + \alpha_i d_i$$
$$r_{i+1} = r_i - \alpha_i A d_i$$
$$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$$
$$d_{i+1} = r_{i+1} + \beta_{i+1} d_i$$

```
# example code of Conjugate Gradients

def conjgrad(A, b, x):
    # source code from https://gist.github.com/glederrey/7fe6e03bbc85c81ed60f3585eea2e073

    r = b - np.dot(A, x)
    p = r
    rsold = np.dot(np.transpose(r), r)

    for i in range(len(b)):
        Ap = np.dot(A, p)
        alpha = rsold / np.dot(np.transpose(p), Ap)
        x = x + np.dot(alpha, p)
        r = r - np.dot(alpha, Ap)
        rsnew = np.dot(np.transpose(r), r)
        if np.sqrt(rsnew) < 1e-8:
            break
        p = r + (rsnew/rsold)*p
        rsold = rsnew
    return x
```

According to the algorithm, we can know that it is difficult to parallelise the loop, because the calculation of variables must be operated step by step. For example, to calculate the variable x_{i+1}, we must calculate the x_i firstly.
However, the updating for variables in each loop can be parallelised. For instance, we can calculate variable β_{i+1}, and x_{i+1} at the same time. This parallelization has a negligible increase in speed.

The algorithm of Incomplete LU Decomposition is shown as below (https://core.ac.uk/download/pdf/82542937.pdf):

```
do i = 2, n
    do k = 1, i - 1
        lik = a(i, k)/a(k, k)
            do j = k + 1, n
                a(i, j) = a(i, j) - lik * a(k, j)
            end
        a(i, k) = lik
    end
end
```

According to the pseudocode above, we can see that the time complexity is $O(n^3)$.
For the incomplete LU, we can parallelise the operations in the loop of

```
do j = k + 1, n
    ...
end
```

because conduct the calculation of "$a(i, j) = a(i, j) - lik * a(k, j)$" have no requirements for previous values. What's more, the time complexity of this operations is $O(n)$. Thus, after parallelise, the time complexity of whole algorithm will be $O(n^2)$.
Additionally, the rest of this algorithm cannot be parallelized, because they need to calculate previous values in advanced. For example, many calculations need to get variable "lik" first.
Therefore, this parallelise will greatly improve the running speed.