Assignment 4 - Solving a finite element system

December 15, 2022

SID: 18072676

Assignment link: https://tbetcke.github.io/hpc_lecture_notes/2022-assignment_4.html

[1]:
```
import time
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

import scipy.sparse.linalg as sp
from scipy.sparse import coo_matrix
import pyamg
from scikits import umfpack
```

1 Part 1: creating the matrix and vector

Write a function that takes N as an input and returns the matrix A and the vector b. The matrix should be stored using an appropriate sparse format - you may use Scipy for this, and do not need to implement your own format.

We take the points p_0, …, p_(N-1)^2-1 to be ordered as: all elements of row 1 come first (in order), then all elements of row 2, etc. That is, for example, p_0 is located at (1, 1) in the mesh, p_5 at (1, 6), p_(N-1)+3 at (2, 4), etc.

[2]:
```
# ******* COMPUTING VECTOR b *******

def g(x, y):
    """
    Computes g(x, y) as defined in the assignemnt
    """
    if type(x) != np.ndarray:
        if x == 0.0:
            return np.sin(4*y)
        if x == 1.0:
            return np.sin(3+4*y)

    if type(y) != np.ndarray:
        if y == 0.0:
```

```
            return np.sin(3*x)
        if y == 1.0:
            return np.sin(3*x+4)

    raise ValueError("At least one of x, y must be 0.0 or 1.0")

def index_to_pos(i, row_length):
    """
    Provided index i of element of mesh p_i (for p_i in set p_0, p_1, ...,␣
 ↪p_(N-1)^2-1),
    returns the corresponding row and column indices of that element in the NxN␣
 ↪mesh

    Note: in this exercise, we always have row_length = N-1
    """
    row_index = i % row_length
    col_index = i // row_length

    # shifting by 1 since boundary is not included
    # in the set p_0, p_1, etc.
    row_index += 1
    col_index += 1

    return row_index, col_index

def b_j(j, N_one, c2, c3, h):
    """
    Gives the element b_j of vector b

    N_one, c1, c2, c3 are pre-computed constants:
    - N_one = N-1
    - c2 = (3+(h*k)**2)/9
    - c3 = (12+(h*k)**2)/36
    """
    # Get the row and column of p_j in the mesh
    row, col = index_to_pos(j, N_one)

    two_h = 2*h

    if row==1 and col==1:
        return c3*(g(0,0)+g(two_h,0)+g(0,two_h)) + c2*(g(h,0)+g(0,h))

    if row==N_one and col==1:
        return c3*(g(1,0)+g(1,two_h)+g(1-two_h,0)) + c2*(g(1-h,0)+g(1,h))

    if row==1 and col==N_one:
        return c3*(g(0,1)+g(two_h,1)+g(0,1-two_h)) + c2*(g(h,1)+g(0,1-h))
```

```
    if row==N_one and col==N_one:
        return c3*(g(1,1)+g(1-two_h,1)+g(1,1-two_h)) + c2*(g(1-h,1)+g(1,1-h))

    list_ = [1, N_one]
    cj_col = col*h
    cj_row = row*h

    if row==1 and not col in list_:
        return c3*(g(0,cj_col+h)+g(0,cj_col-h)) + c2*g(0,cj_col)

    if row==N_one and not col in list_:
        return c3*(g(1,cj_col+h)+g(1,cj_col-h)) + c2*g(1,cj_col)

    if not row in list_ and col==1:
        return c3*(g(cj_row+h,0)+g(cj_row-h,0)) + c2*g(cj_row,0)

    if not row in list_ and col==N_one:
        return c3*(g(cj_row+h,1)+g(cj_row-h,1)) + c2*g(cj_row,1)

    return 0.0

def generate_b(N):
    """
    Generates vector b for a mesh with side-length = N squares
    """
    k = 5

    # pre-computing some constants to avoid duplicate calculations

    h = 1/N
    N_one = N - 1
    N_one_squared = N_one**2

    hk_squared = (h*k)**2
    c2 = (3+hk_squared)/9
    c3 = (12+hk_squared)/36

    b = np.zeros(N_one_squared)

    for i in range(N_one_squared):
        b[i] = b_j(i, N_one, c2, c3, h)

    return b
```

Notes describing the thinking process behind the construction of matrix A

Matrix A represents the interaction between each point of the mesh (it excludes the mesh points located on the boundary as they are fixed). Each row / column describes the interaction of one single point on the mesh with all other points (note: this implies A is symmetric about its leading diagonal, since interaction of point p_i with point p_j is the same as the interaction of point p_j with point p_i)

For clarity, we reiterate our definition of the ordering of the mesh points in A: We take the points p_0, …, p_(N-1)^2-1 to be ordered as: all elements of row 1 come first (in order), then all elements of row 2, etc. That is, for example, p_0 is located at (1, 1) in the mesh, p_5 at (1, 6), p_(N-1)+3 at (2, 4), etc.

From the definition of A, we know that each point of the mesh only interacts with itself and its neighbours (including diagonally adjacent neighbours). We start by considering the case where all points have all their neighbours (including those on the boundary and in corners). In this case, for a row k in A, we have 9 non-zero elements which form a particular pattern. Now, if we consider the next point p_{i+1}) (i.e. row k+1 in A), all of its neighbours will also be the points that follow each of the neighbours of p_i (this is also the case at boundaries because of the way in which we defined the ordering of the mesh points in A above). Hence, the pattern at row k+1 will be the same as that at row k but shifted by 1 to the right. If we extrapolate this reasoning, then the pattern formed by the non-zero elements should form 9 diagonals, each corresponding to a different interaction (e.g. 1 will be for the point with itself (main leading diagonal), one for the interaction with the upper left point, one with upper point, etc. for all neighbours).

Given the definition for the ordering of mesh points in A, we have that, for a point p_i, its neighbours will be: - upper right: $p_{i-(N-1)+1}$ - upper: $p_{i-(N-1)}$ - upper left: $p_{i-(N-1)-1}$ - left: p_{i-1} - lower left: $p_{i+(N-1)-1}$ - lower: $p_{i+(N-1)}$ - lower right: $p_{i+(N-1)+1}$ - right: p_{i+1}

Now, the truth of the situation is that all points don't have all of their neighbours, only the inner points do. Hence, only these inner points will have a row with 9 non-zero elements, the others will have: - 4 if they are a point in a corner - 6 if they are a point on a boundary (but not a corner)

Now, since the points of the boundary condition are not included in matrix A, their interaction is already not encoded in the matrix (and cannot be!), so we don't have to worry about them. The interaction that are encoded in fully filled non-zero diagonals and that shouldn't be are for points which correspond to adjacent rows / columns of A (i.e. adjacent in the set $\{p_0, p_1, ...p_{(N-1)^2-1}\}$), but aren't adjacent in the mesh: that is, the leftmost and the rightmost points of the mesh.

- Leftmost points are the set $\{p_0, p_{N-1}, p_{2(N-1)}, \cdots, p_{(N-2)(N-1)}\}$
- Rightmost points are the set $\{p_{(N-1)-1}, p_{2(N-1)-1}, \cdots, p_{(N-1)^2-1}\}$

They both occur with period N-1 (so we expect all holes in the non-zero diagonals to occur at such frequency too).

Leftmost points are missing their 3 right-hand neighbours (upper right, middle right and lower right), while rightmost points are missing their 3 left-hand neighbours (upper left, middle left and lower left).

[3]:
```
# ******* COMPUTING MATRIX A IN COO FORMAT *******

def diag_indices(size, k):
    """
```

```
    Returns the indices in n-D array of the
    elements on the k-th diagonal.

    - size: the side-length of the array

    note: the array must be square (a.k.a size must be the same for all␣
 ↪dimensions)

    (adapted from StackOverflow)
    """
    i, j = np.diag_indices(size)

    if k < 0:
        return i[-k:], j[:k]
    elif k > 0:
        return i[:-k], j[k:]
    else:
        return i, j

def holed_array(arr, hole_start, hole_period):
    """
    Returns the same array but with elements periodically removed

    - val: the value with which the array is filled
    - size: the length of the array
    - hole_start: the start index for the holes
    - hole_period: period for the holes
    """
    # Adding holes
    hole_indices = np.arange(hole_start, arr.shape[0]-hole_start, hole_period,␣
 ↪dtype=int)
    arr[hole_indices] = 0.0

    return np.delete(arr, hole_indices)

def diag_to_coo(matsize, offset, val):
    """
    Returns the columns, rows and data corresponding
    to a leading diagonal offseted by "offset",
    in a 2D array of size "matsize", and filled
    with value "val"
    """
    rows, cols = diag_indices(matsize, offset)
    data = np.full(len(rows), val)

    return rows, cols, data
```

```python
def diag_to_coo_with_holes(matsize, offset, val, hole_start, hole_period):
    """
    Performs "diag_to_coo", but with elements peridodically
    removed.

    - hole_start: index in the returned diagonal where elements should start␣
 ↪being removed
    - hole_period: the period of removal of elements
    """
    rows, cols, data = diag_to_coo(matsize, offset, val)

    holed_indices = np.arange(len(rows), dtype=int)
    holed_indices = holed_array(holed_indices, hole_start, hole_period)
    rows = rows[holed_indices]
    cols = cols[holed_indices]
    data = data[holed_indices]

    return rows, cols, data

def generate_A(N):
    """
    Generates matrix A in coo format for a mesh with side-length = N squares
    """

    k = 5
    h = 1/N

    # pre-computing some constants to avoid duplicate calculations
    N_one = N - 1
    N_one_squared = N_one**2
    hk_squared = (h*k)**2

    all_rows = []
    all_cols = []
    all_data = []


    # *** Same element in mesh (leading diagonal of A) ***
    offset = 0
    val = (24-4*hk_squared)/9
    rows, cols, data = diag_to_coo(N_one_squared, offset, val)
    all_rows.extend(rows)
    all_cols.extend(cols)
    all_data.extend(data)

    # *** vertically / horizontally adjacent elements in mesh ***
```

```
    # * horizontally adjacent elements *
    offset = 1
    val = -(3+hk_squared)/9
    hole_start = N_one-1
    rows, cols, data = diag_to_coo_with_holes(N_one_squared, offset, val,␣
↪hole_start, N_one)
        # filling upper half diagonal
    all_rows.extend(rows)
    all_cols.extend(cols)
    all_data.extend(data)
        # filling lower half diagonal
    all_rows.extend(cols)
    all_cols.extend(rows)
    all_data.extend(data)

    # * vertically adjacent elements *
    offset = N_one
    val = -(3+hk_squared)/9
    rows, cols, data = diag_to_coo(N_one_squared, offset, val)
        # filling upper half diagonal
    all_rows.extend(rows)
    all_cols.extend(cols)
    all_data.extend(data)
        # filling lower half diagonal
    all_rows.extend(cols)
    all_cols.extend(rows)
    all_data.extend(data)


    # *** diagonally adjacent elements in mesh ***

    # * lower left elements in mesh *
    offset = (N_one-1)
    val = -(12+hk_squared)/36
    hole_start = 0
    rows, cols, data = diag_to_coo_with_holes(N_one_squared, offset, val,␣
↪hole_start, N_one)
        # filling upper half diagonal
    all_rows.extend(rows)
    all_cols.extend(cols)
    all_data.extend(data)
        # filling lower half diagonal
    all_rows.extend(cols)
    all_cols.extend(rows)
    all_data.extend(data)

    # * lower right elements in mesh *
```

```
    offset = (N_one+1)
    val = -(12+hk_squared)/36
    hole_start = N_one-1
    rows, cols, data = diag_to_coo_with_holes(N_one_squared, offset, val,␣
 ↪hole_start, N_one)
        # filling upper half diagonal
    all_rows.extend(rows)
    all_cols.extend(cols)
    all_data.extend(data)
        # filling lower half diagonal
    all_rows.extend(cols)
    all_cols.extend(rows)
    all_data.extend(data)

    # we generate our final result in csr format as many solvers
    # were found to show a warning message if that wasn't the case.
    return coo_matrix((all_data, (all_rows, all_cols))).tocsr()
```

[4]:
```
def generate_Ab(N):
    """
    Returns matrix A and vector b for a mesh with side-length = N squares
    """
    A = generate_A(N)
    b = generate_b(N)

    return A, b
```

[5]:
```
# True values of A and b for N = 2, 3, 4; as taken from assignment

# A and b for N=2
A_2 = np.array([
    [-0.1111111111111116],
])
b_2 = np.array([0.2699980311833446])

# A and b for N=3
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],
])
```

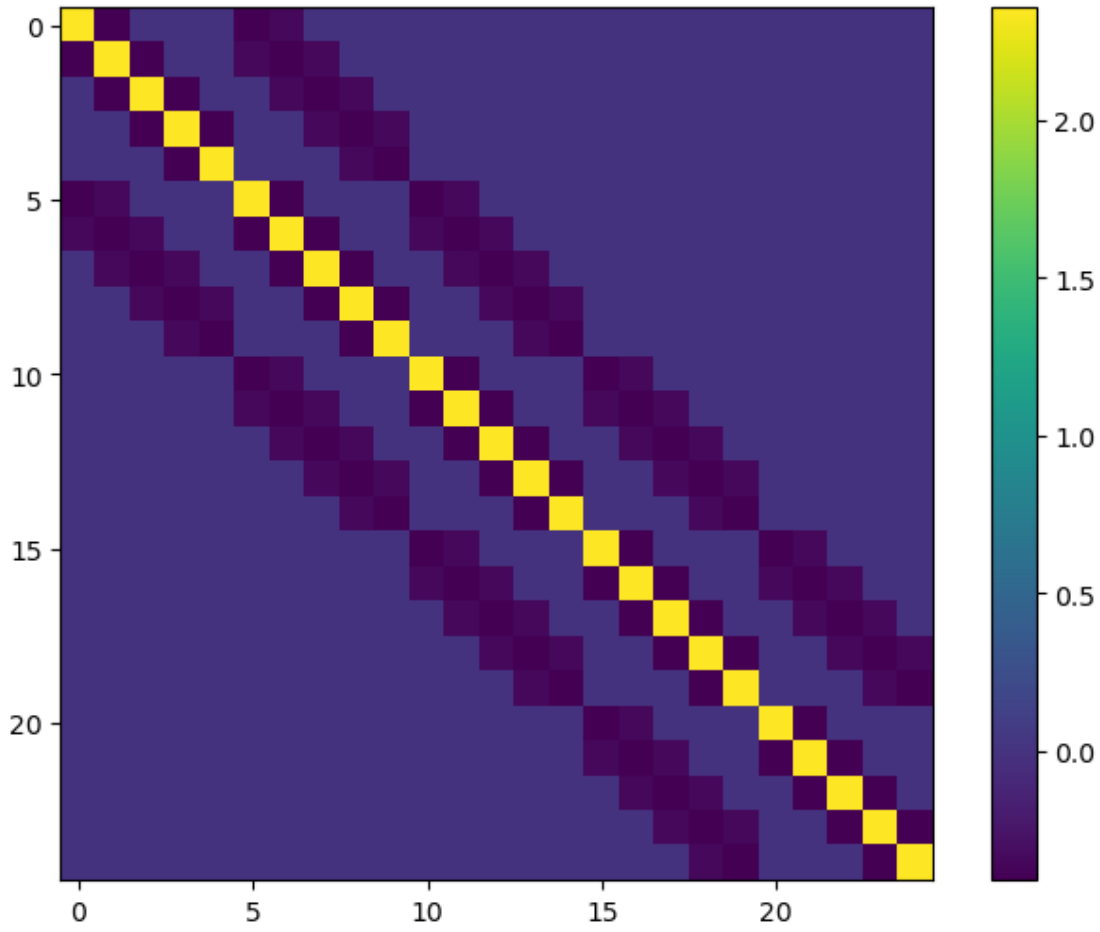
```
b_3 = np.array([1.7251323007221917, 0.15334285313223067, -0.34843455260733003,␣
 ↪-1.0558651156722307])

# A and b for N=4
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])

# A and b for N=5
A_5 = np.array([
    [2.222222222222222, -0.4444444444444444, 0.0, 0.0, -0.4444444444444444, -0.
 ↪3611111111111111, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
    [-0.4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
 ↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
 ↪0, 0.0, 0.0, 0.0, 0.0, 0.0],
    [0.0, -0.4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
 ↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
 ↪0, 0.0, 0.0, 0.0, 0.0],
    [0.0, 0.0, -0.4444444444444444, 2.222222222222222, 0.0, 0.0, -0.
 ↪3611111111111111, -0.4444444444444444, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
 ↪0],
    [-0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 2.222222222222222, -0.
 ↪4444444444444444, 0.0, 0.0, -0.4444444444444444, -0.3611111111111111, 0.0, 0.
 ↪0, 0.0, 0.0, 0.0, 0.0],
```

```
    [-0.3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, -0.
 ↪4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
 ↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
 ↪0, 0.0],
    [0.0, -0.3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0,␣
 ↪-0.4444444444444444, 2.222222222222222, -0.4444444444444444, 0.0, -0.
 ↪3611111111111111, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 0.0, 0.
 ↪0],
    [0.0, 0.0, -0.3611111111111111, -0.4444444444444444, 0.0, 0.0, -0.
 ↪4444444444444444, 2.222222222222222, 0.0, 0.0, -0.3611111111111111, -0.
 ↪4444444444444444, 0.0, 0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, -0.4444444444444444, -0.3611111111111111, 0.0, 0.0, 2.
 ↪222222222222222, -0.4444444444444444, 0.0, 0.0, -0.4444444444444444, -0.
 ↪3611111111111111, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
 ↪3611111111111111, 0.0, -0.4444444444444444, 2.222222222222222, -0.
 ↪4444444444444444, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
 ↪3611111111111111, 0.0],
    [0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
 ↪3611111111111111, 0.0, -0.4444444444444444, 2.222222222222222, -0.
 ↪4444444444444444, 0.0, -0.3611111111111111, -0.4444444444444444, -0.
 ↪3611111111111111],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.4444444444444444, 0.
 ↪0, 0.0, -0.4444444444444444, 2.222222222222222, 0.0, 0.0, -0.
 ↪3611111111111111, -0.4444444444444444],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.4444444444444444, -0.
 ↪3611111111111111, 0.0, 0.0, 2.222222222222222, -0.4444444444444444, 0.0, 0.
 ↪0],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.
 ↪4444444444444444, -0.3611111111111111, 0.0, -0.4444444444444444, 2.
 ↪222222222222222, -0.4444444444444444, 0.0],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.
 ↪4444444444444444, -0.3611111111111111, 0.0, -0.4444444444444444, 2.
 ↪222222222222222, -0.4444444444444444],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.3611111111111111, -0.
 ↪4444444444444444, 0.0, 0.0, -0.4444444444444444, 2.222222222222222],
])
b_5 = np.array([1.2673039440507343, 0.9698054647507671, 1.0133080988552785, 0.
 ↪07206335813040798, 0.9472174493756345, 0.0, 0.0, -0.9416429716282946, 0.
 ↪6400834406610956, 0.0, 0.0, -0.7322882523543968, -0.8159823324771336, -0.
 ↪9192523853093425, -0.48342793699793585, -0.19471066818706848])
```

[6]:
```
# Checking proper functioning of "generate_Ab"

N = 5
A, b = generate_Ab(N)
```

```
assert np.allclose(A_5, A.toarray())
assert np.allclose(b_5, b)

N = 4
A, b = generate_Ab(N)
assert np.allclose(A_4, A.toarray())
assert np.allclose(b_4, b)

N = 3
A, b = generate_Ab(N)
assert np.allclose(A_3, A.toarray())
assert np.allclose(b_3, b)

N = 2
A, b = generate_Ab(N)
assert np.allclose(A_2, A.toarray())
```

[7]:
```
N = 6
A, b = generate_Ab(N)

plt.figure(figsize=(8, 6))
plt.imshow(A.toarray())
plt.colorbar()
plt.show()
```



2 Part 2: solving the system

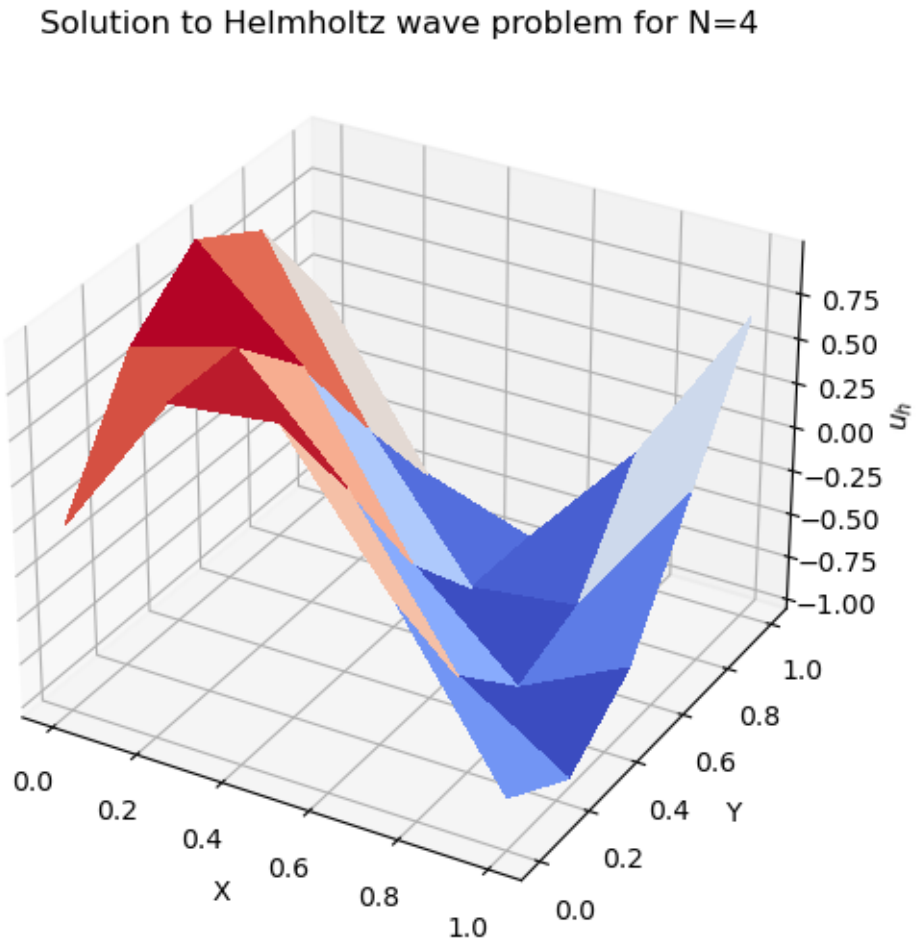
Using any matrix-vector solver, solve the matrix-vector problem for $N = 4$, $N = 8$ and $N = 16$, and plot the approximate solutions to the Helmholtz problem.

[8]:
```
def get_u(x, N):
    """
    Adds the outer boundary to a matrix of the inner values
    for u, the solution to the wave problem.

    x: a 1D array of size (N-1)^2 - 1, which is the solution
    to the Helmholtz wave problem

    returns u: a 2D array of shape (N+1, N+1)
    """
    u = np.zeros((N+1, N+1))
```

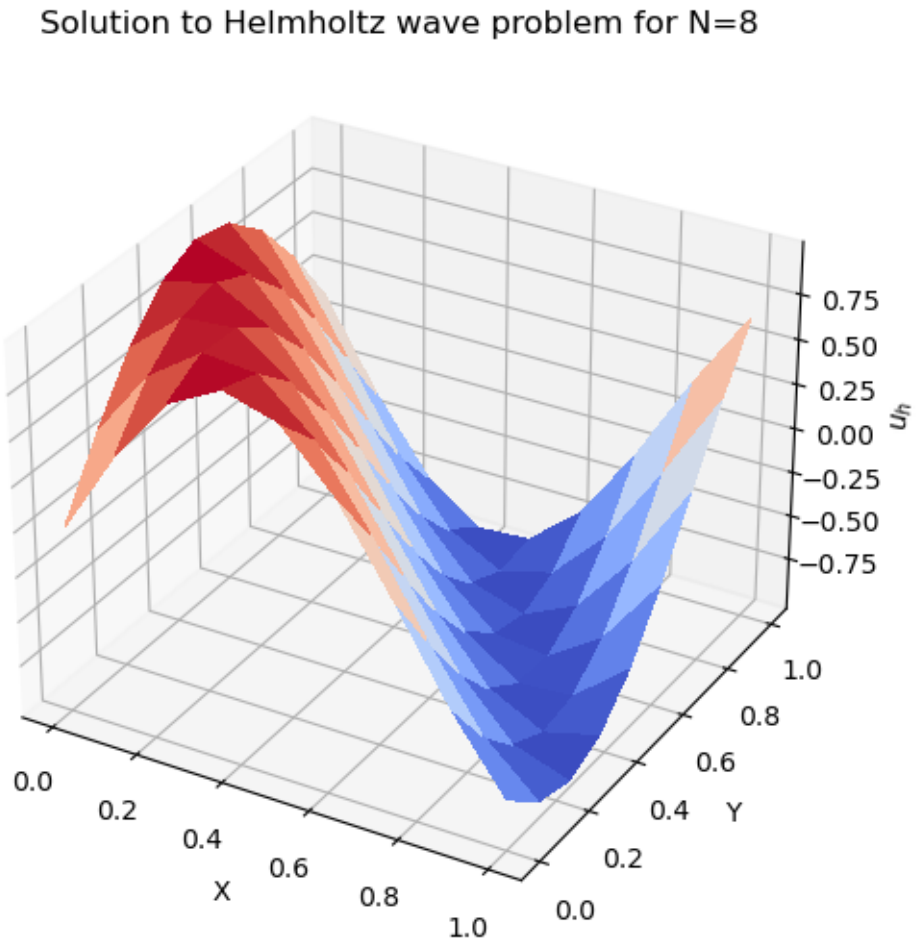
```
    # reshaping to format of the mesh
    u_inner = x.reshape((N-1, N-1), order="F")

    # Adding the fixed boundary conditions
    ticks = np.linspace(0, 1, N+1)
    u[0, :] = g(0.0, ticks)
    u[N, :] = g(1.0, ticks)
    u[:, 0] = g(ticks, 0.0)
    u[:, N] = g(ticks, 1.0)
    u[1:N, 1:N] = u_inner

    return u
```

[9]:
```
def plot_solution(x, N):
    """
    Plots a solution to the Helmholtz wave problem
    """
    # Getting u from x
    u = get_u(x, N)

    # generating the mesh of positions
    ticks = np.linspace(0, 1, N+1)
    X, Y = np.meshgrid(ticks, ticks)

    # Plotting
    fig = plt.figure(figsize=(6, 6))
    ax = fig.add_subplot(projection='3d')
    surf = ax.plot_surface(X, Y, u, antialiased=False, cmap=cm.coolwarm)
    ax.set_title(f"Solution to Helmholtz wave problem for N={N}")
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("$u_h$")
    plt.show()
```

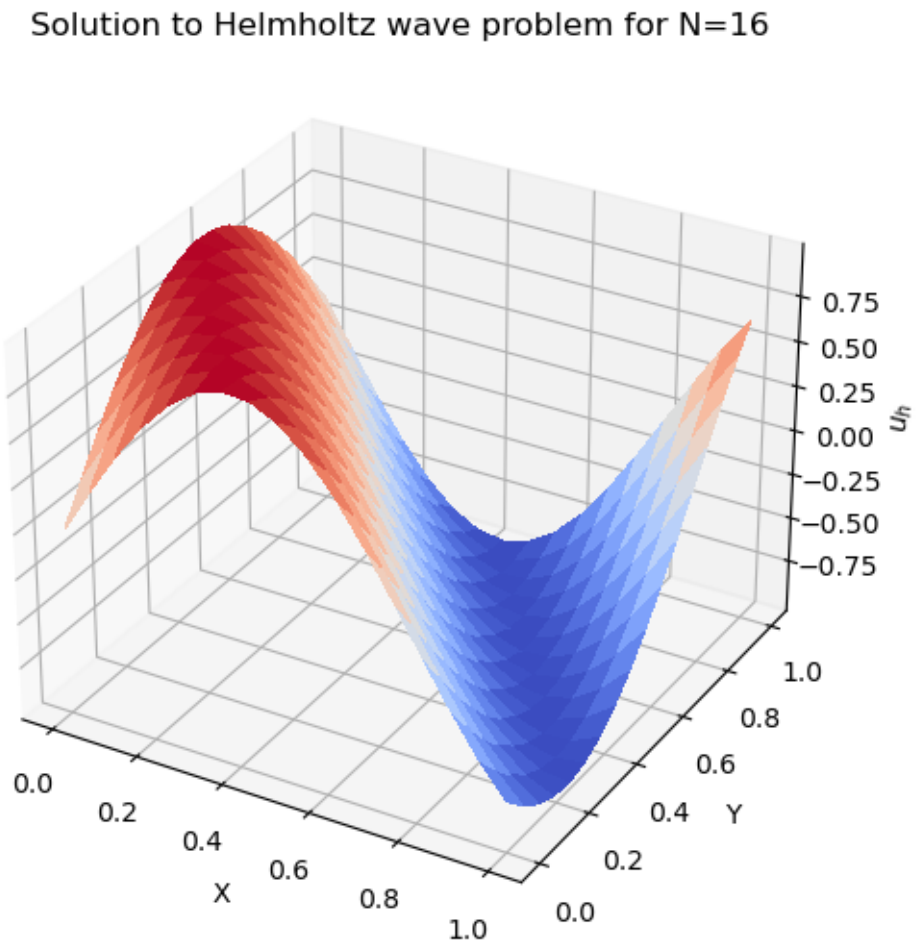
[51]:
```
N = 4

# Generating A and b
A, b = generate_Ab(N)

# Solving for x
x = sp.spsolve(A, b)

plot_solution(x, N)
```



[11]:
```
N = 8

# Generating A and b
A, b = generate_Ab(N)

# Solving for x
x = sp.spsolve(A, b)

plot_solution(x, N)
```



[12]:
```
N = 16

# Generating A and b
A, b = generate_Ab(N)

# Solving for x
x = sp.spsolve(A, b)

plot_solution(x, N)
```



3 Part 3: comparing solvers and preconditioners

In this section, your task is to evaluate the performance of various matrix-vector solvers. To do this, **solve the matrix-vector problem with small to medium sized value of N using a range of different solvers of your choice, measuring factors you deem to be important for your evaluation.** These factors should include the time taken by the solver, and may additionally include many other things such as the number of iterations taken by an iterative solver, or the size of the residual after each iteration. **Make a set of plots that show the measurements you have made and allow you to compare the solvers.**

You should compare at least five matrix-vector solvers: at least two of these should be iterative solvers, and at least one should be a direct solver. You can use solvers from the Scipy library. (You may optionally use additional solvers from other linear algebra libraries such as PETSc, but you do not need to do this to achieve high marks. You should use solvers from these libraries and do not need to implement your own solvers.) For two of the iterative solvers you have chosen to use, **repeat the comparisons with three different choices of preconditioner.**

Based on your experiments, **pick a solver** (and a preconditioner if it improves the solver) that

you think is most appropriate to solve this matrix-vector problem. **Explain, making use of the data from your experiments, why this is the best solver for this problem.**

3.1 3.1 Comparing the solvers

We start by comparing 5 different solvers. I picked them to be: * gmres (iterative) * cg (iterative) * qmr (iterative) * gcrotmk (iterative) * umfpack (direct)

We will

3.1.1 3.1.1 Computing the performance

To compare the time taken for different Ns, the most sensible thing to do is to compare the time it takes to reach a given tolerance. Hence, I shall start with a broad comparison of all solvers with a fairly low tolerance (because, for example, gmres solver takes

[13]:
```
N_store = [20, 40, 60, 80, 100, 120, 140, 160, 180, 200]
tol = 10**-10
maxiter = None
```

[14]:
```
# **** Evaluating GMRES ****

perfs_gmres = []

# reduced number of Ns as GMRES on its own is very inefficient
for N in [20, 40, 60, 80, 100]:
    print(N)
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    sp.gmres(A, b, callback=lambda res: residuals.append(res), tol=tol,␣
 ↪maxiter=maxiter)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    # recording performance in a dictionary for code cleanliness
    perf = {
        "name": "gmres",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }
```

```
    perfs_gmres.append(perf)
```

```
20
40
60
80
100
```

[15]:
```
# **** Evaluating CG ****

perfs_cg = []

callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))


for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    sp.cg(A, b, callback=callback, tol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    # recording performance in a dictionary for code cleanliness
    perf = {
        "name": "cg",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_cg.append(perf)
```

[16]:
```
# **** Evaluating qmr ****

perfs_qmr = []

callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))
```

```
for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    sp.qmr(A, b, callback=callback, tol=tol, maxiter=maxiter)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    # recording performance in a dictionary for code cleanliness
    perf = {
        "name": "qmr",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_qmr.append(perf)
```

[17]:
```
# **** Evaluating gcrotmk ****

perfs_gcrotmk = []

callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))


for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    sp.gcrotmk(A, b, callback=callback, tol=tol, atol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    # recording performance in a dictionary for code cleanliness
    perf = {
```

```
        "name": "gcrotmk",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_gcrotmk.append(perf)
```

[18]:
```
# **** Evaluating umfpack  ****

perfs_umfpack = []

for N in N_store:
    A, b = generate_Ab(N)

    start = time.time()
    x = umfpack.spsolve(A, b)
    end = time.time()

    time_taken = end - start

    # umfpack is a direct solver, hence we can only obtain 1 residual
    residual = np.linalg.norm(b - A @ x) / np.linalg.norm(b)

    # recording performance in a dictionary for code cleanliness
    perf = {
        "name": "umfpack",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "res": residual
    }

    perfs_umfpack.append(perf)
```

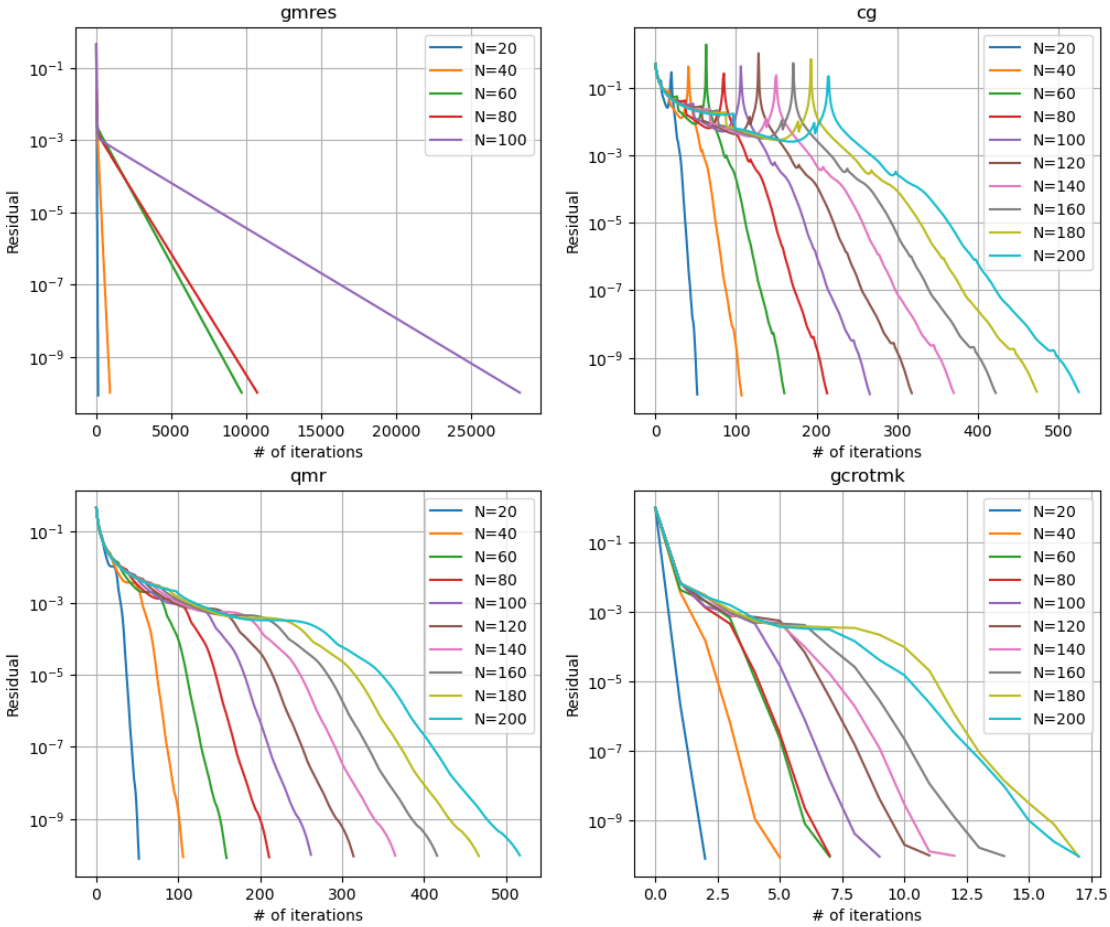
3.1.2 3.1.2 Comparing the performance

Here, we do a first performance comparison for all 5 solvers. We compare computing time, residual over iteration, and number of iterations

[19]:
```
# Defining a plotting function for code cleanliness

def plot_res(ax, perfs):
    """
    Generic plotter of residual against iteration for an iterative solver,
```

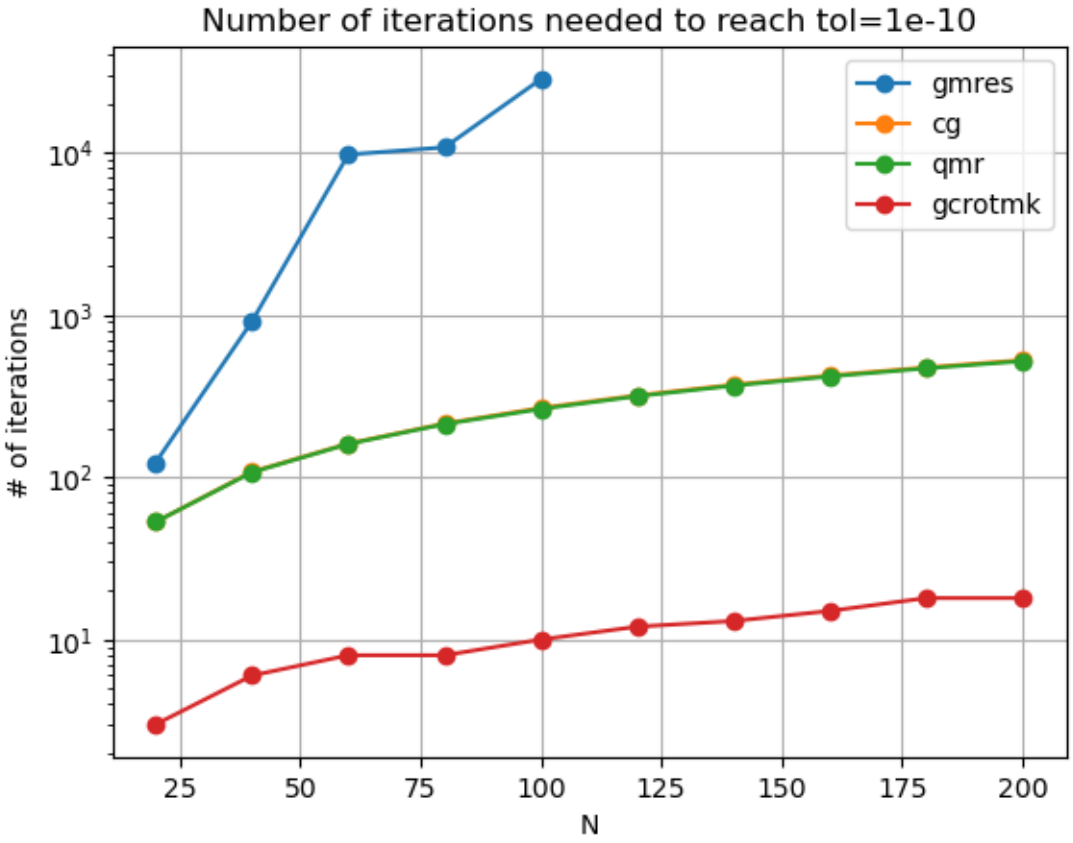
```
    for multiple values of N

    ax is matplotlib's Axes class

    perfs is an array of dictionaries of at least properties:
        {
            "tol": the tolerance
            "time": the time taken
            "iters": the number of iterations taken
            "res": an array of residuals of length "iters"
        }
    """
    for perf in perfs:
        ax.plot(np.arange(perf["iters"]), perf["res"], label=f'N={perf["N"]}')

    ax.grid()
    ax.set_xlabel("# of iterations")
    ax.set_ylabel("Residual")
    ax.set_yscale("log")
    ax.legend()
    ax.set_title(f"{perf['name']}")
```

[20]:
```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 10))

plot_res(ax1, perfs_gmres)
plot_res(ax2, perfs_cg)
plot_res(ax3, perfs_qmr)
plot_res(ax4, perfs_gcrotmk)

plt.suptitle(f"Evolution of the residual across iterative solvers (tol =␣
 ↪{perfs_gmres[0]['tol']})")
plt.show()
```



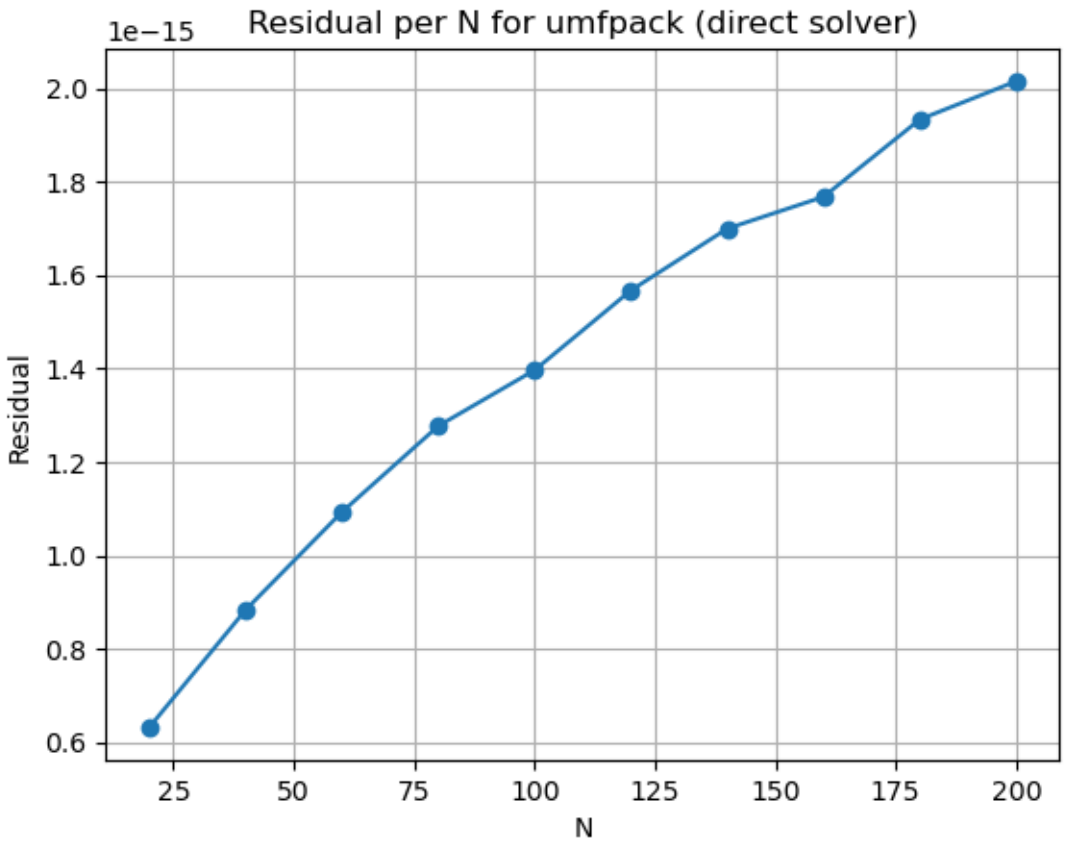
[21]:
```python
# Comparing the number of iterations for iterative solvers

# only iterative solvers
all_perfs = [perfs_gmres, perfs_cg, perfs_qmr, perfs_gcrotmk]

for perfs in all_perfs:
    iters = [perf["iters"] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]
    plt.plot(Ns, iters, "-o", label=perfs[0]["name"])

plt.grid()
plt.xlabel("N")
plt.ylabel(f"# of iterations")
plt.title(f"Number of iterations needed to reach tol={perfs_gmres[0]['tol']}")
plt.yscale("log")
```

```
plt.legend()
plt.show()
```

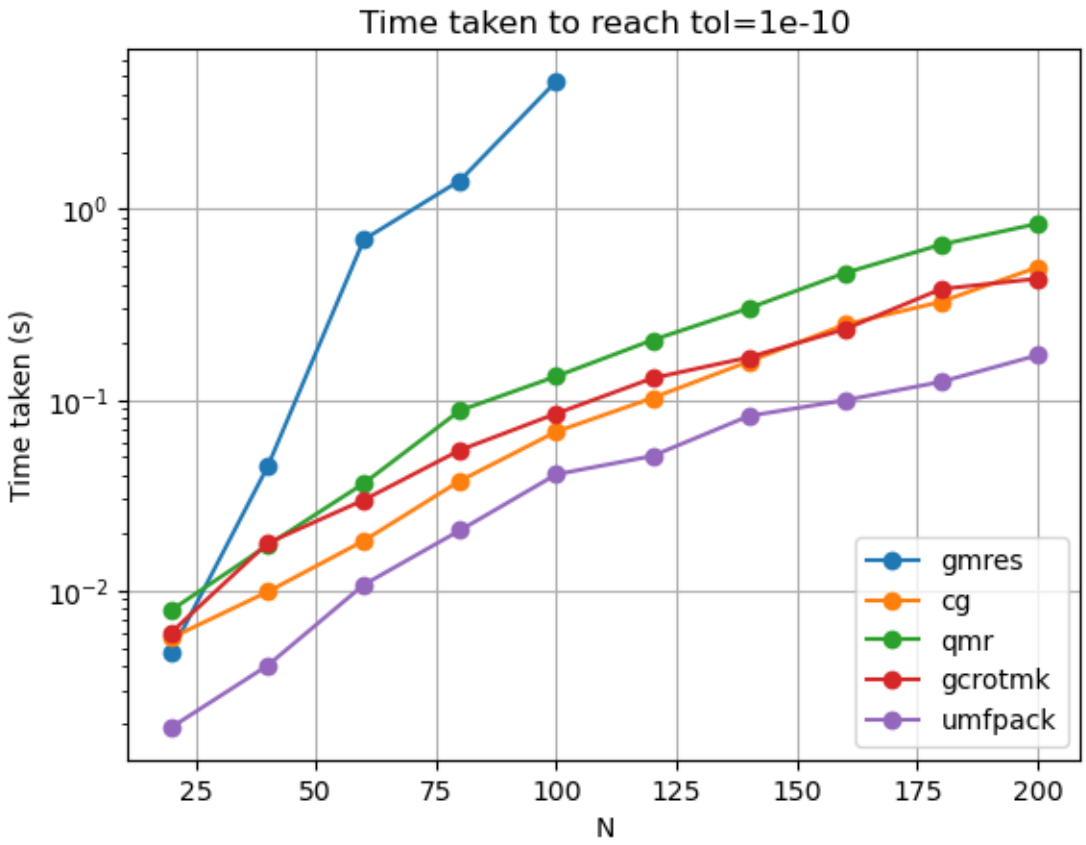


Notes: - gmres is very inefficient. It is already at over 25000 iterations for N=100 (hence why larger Ns weren't computed) - We can see cg seems clearly unstable (peaks at a certain point for every N) but always managed to reduce its tolerance below the threshold for all tested N so far. - Though qmr and cg follow completely different patterns for residual over iteration, they somehow manage to have an almost identical curve for the number of iterations over N (this corresponds to the end of each curve on the residual vs iteration plots)! - gcrotmk requires very few iterations, and that number seems to scale very well with increasing N.

[22]:
```
# Plotting for the direct solver. As it isn't iterative, it contains just 1␣
↪residual
plt.plot([p["N"] for p in perfs_umfpack], [p["res"] for p in perfs_umfpack],␣
↪"-o")
plt.grid()
plt.ylabel("Residual")
plt.xlabel("N")
plt.title("Residual per N for umfpack (direct solver)")
plt.show()
```



[23]:
```python
# Comparing the time taken for all solvers

all_perfs = [perfs_gmres, perfs_cg, perfs_qmr, perfs_gcrotmk, perfs_umfpack]

for perfs in all_perfs:
    times = [perf["time"] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]

    plt.plot(Ns, times, "-o", label=perfs[0]["name"])

plt.grid()
plt.xlabel("N")
plt.ylabel(f"Time taken (s)")
plt.title(f"Time taken to reach tol={perfs_gmres[0]['tol']}")
plt.yscale("log")
plt.legend()
plt.show()
```



All of umfpack, gcrotmk, qmr & cq seems to scale quite similarly as N increases. However, umfpack lies consistently below the other three. Moreover, if we look at umpack's graph of residual over N, we see that it achieves a (final) residual in the range 10^-15 < res < 10^-16, which is ~ 10^5 times more precise than the tolerance reached for gcrotmk, qmr & cq. Overall, umfpack beats all other tested solvers by a margin in both precision and time taken. Therefore, it is, so far, the best solver.

We now turn to boosting performance using pre-conditioners.

3.2 3.2 Adding preconditioners

I chose to test the following preconditioners: - Incomplete LU (ILU) - Analytical Multigrid (AMG) - Sparse Approximate Inverse (SPAI)

To use on the following 2 iterative solvers: - gmres - cg

[24]:
```
# Taken from lecture notes

def spai(A, m):
    """Perform m step of the SPAI iteration."""
    from scipy.sparse import identity
    from scipy.sparse import diags
```

```
    from scipy.sparse.linalg import onenormest

    n = A.shape[0]

    ident = identity(n, format='csr')
    alpha = 2 / onenormest(A @ A.T)
    M = alpha * A

    for index in range(m):
        C = A @ M
        G = ident - C
        AG = A @ G
        trace = (G.T @ AG).diagonal().sum()
        alpha = trace / np.linalg.norm(AG.data)**2
        M = M + alpha * G

    return M
```

3.2.1 3.2.1 Computing performance

[25]:
```
tol = 10**-10
maxiter = None
```

[26]:
```
# **** Evaluating GMRES w/ ILU as pre-conditioner ****

perfs_gmres_ilu = []

for N in N_store:
    print(N)
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    M = sp.LinearOperator(matvec=sp.spilu(A.tocsc(), fill_factor=20,␣
 ↪drop_rule='dynamic').solve,
                          shape=A.shape,
                          dtype=np.float64)
    sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res), tol=tol,␣
 ↪maxiter=maxiter)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "gmres w/ ILU",
```

```
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_gmres_ilu.append(perf)
```

```
20
40
60
80
100
120
140
160
180
200
```

[27]:
```
# **** Evaluating GMRES w/ AMG as pre-conditioner ****

perfs_gmres_amg = []

for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res), tol=tol,␣
 ↪maxiter=maxiter)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "gmres w/ AMG",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }
```

```
    perfs_gmres_amg.append(perf)
```

[28]:
```
# **** Evaluating GMRES w/ SPAI as pre-conditioner ****

perfs_gmres_spai = []

# Doing for fewer values of N as it is quite long to form SPAI
for N in [20, 40, 60, 80, 100, 120]:
    print("N", N)
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    M = spai(A, 4)
    sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res), tol=tol,␣
 ↪maxiter=maxiter)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "gmres w/ SPAI",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_gmres_spai.append(perf)
```

```
N 20
N 40
N 60
N 80
N 100
N 120
```

[29]:
```
# **** Evaluating CG w/ ILU as pre-conditioner ****

perfs_cg_ilu = []

callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))
```

```
for N in [20, 40, 60, 80, 100, 120, 140]:
    print(N)
    residuals = []

    A, b = generate_Ab(N)

    # not including this format conversion in the timing as that would confer␣
 ↪an unfair advantage
    # to the solvers that are already in the right format
    A_csc = A.tocsc()

    start = time.time()
    M = sp.LinearOperator(matvec=sp.spilu(A_csc, fill_factor=20,␣
 ↪drop_rule='dynamic').solve,
                          shape=A.shape,
                          dtype=np.float64)
    sp.cg(A, b, M=M, callback=callback, tol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "cg w/ ILU",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_cg_ilu.append(perf)
```

```
20
40
60
80
100
120
140
```

[30]:
```
# **** Evaluating CG w/ AMG as pre-conditioner ****

perfs_cg_amg = []
```

```
callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))


for N in N_store:
    print(N)
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.cg(A, b, M=M, callback=callback, tol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "cg w/ AMG",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_cg_amg.append(perf)
```

```
20
40
60
80
100
120
140
160
180
200
```

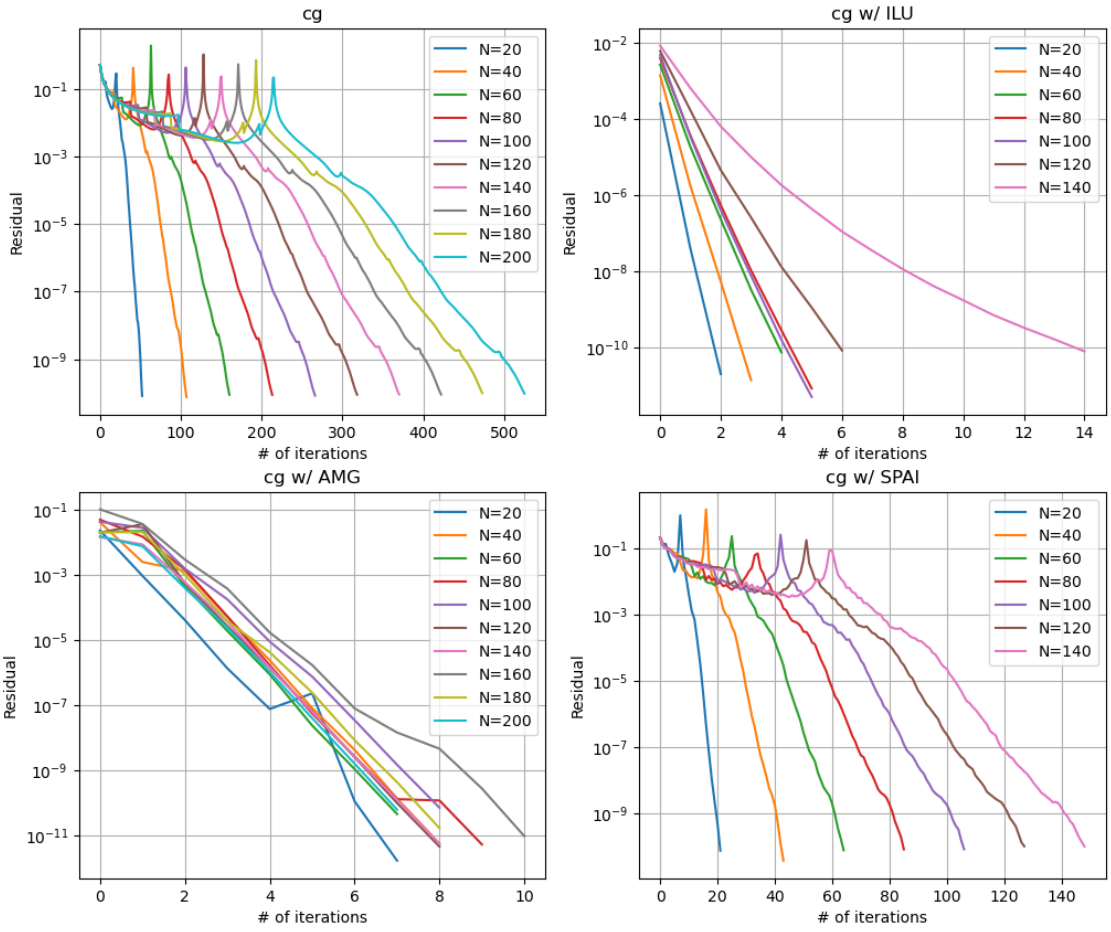
[31]:
```
# **** Evaluating CG w/ SPAI as pre-conditioner ****

perfs_cg_spai = []

# Doing for fewer values of N as it is quite long to form SPAI
for N in [20, 40, 60, 80, 100, 120, 140]:
```

```
    print("N", N)
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    M = spai(A, 4)
    sp.cg(A, b, M=M, callback=callback, tol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "cg w/ SPAI",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_cg_spai.append(perf)
```

```
N 20
N 40
N 60
N 80
N 100
N 120
N 140
```

3.2.2 3.2.2 Comparing preconditioners for CG

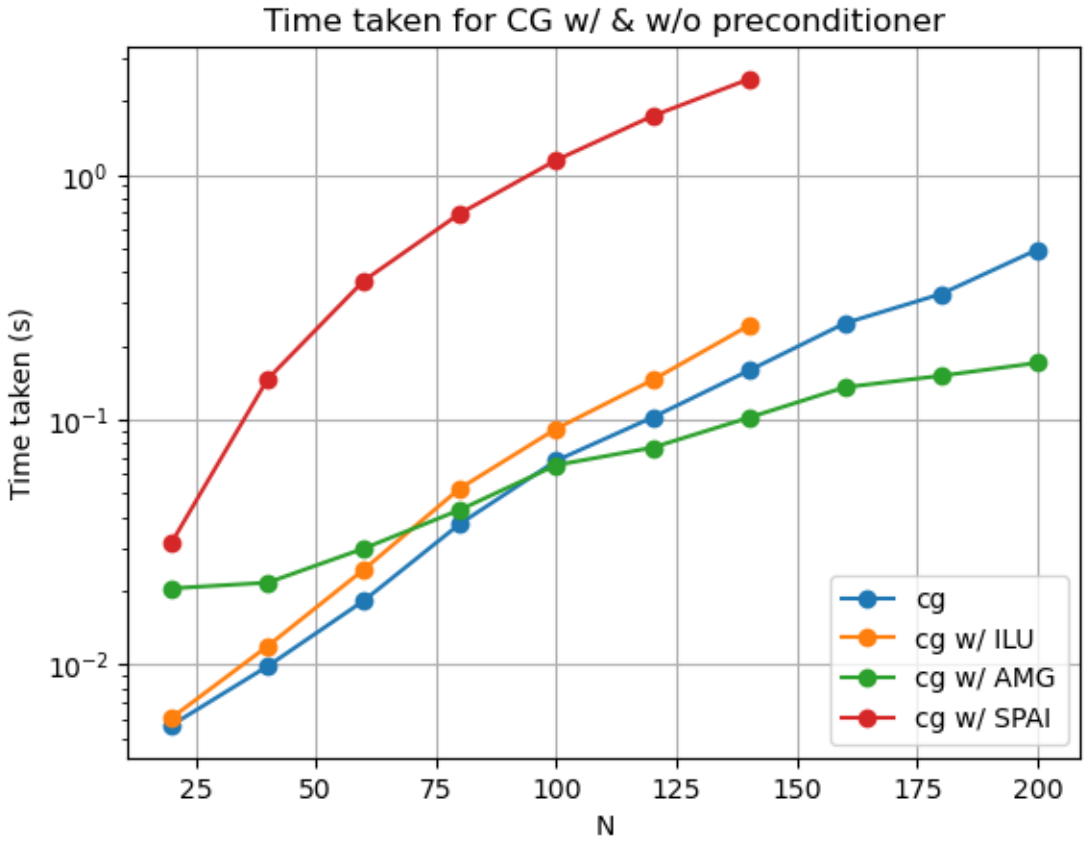
[32]:
```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 10))

plot_res(ax1, perfs_cg)
plot_res(ax2, perfs_cg_ilu)
plot_res(ax3, perfs_cg_amg)
plot_res(ax4, perfs_cg_spai)

plt.suptitle(f"Residual evolution for CG w/ preconditioners (tol =␣
  ↪{perfs_cg[0]['tol']})")
plt.show()
```



Note: We see that AMG and ILU have achieved to remove the bump in the residual curves. I take this to mean that it has increased the stability of the cg solve. They have both also reduced enormously the number of iterations needed. SPAI also had some positive impact on the number of iteration, but much less important than the other two, and no notable impact on the stability.

[33]:
```
# Comparing across preconditioners

all_perfs = [perfs_cg, perfs_cg_ilu, perfs_cg_amg, perfs_cg_spai]

for perfs in all_perfs:
    times = [perf["time"] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]

    plt.plot(Ns, times, "-o", label=perfs[0]["name"])

plt.grid()
```

```
plt.xlabel("N")
plt.ylabel(f"Time taken (s)")
plt.title(f"Time taken for CG w/ & w/o preconditioner")
plt.yscale("log")
plt.legend(loc="lower right")
plt.show()
```



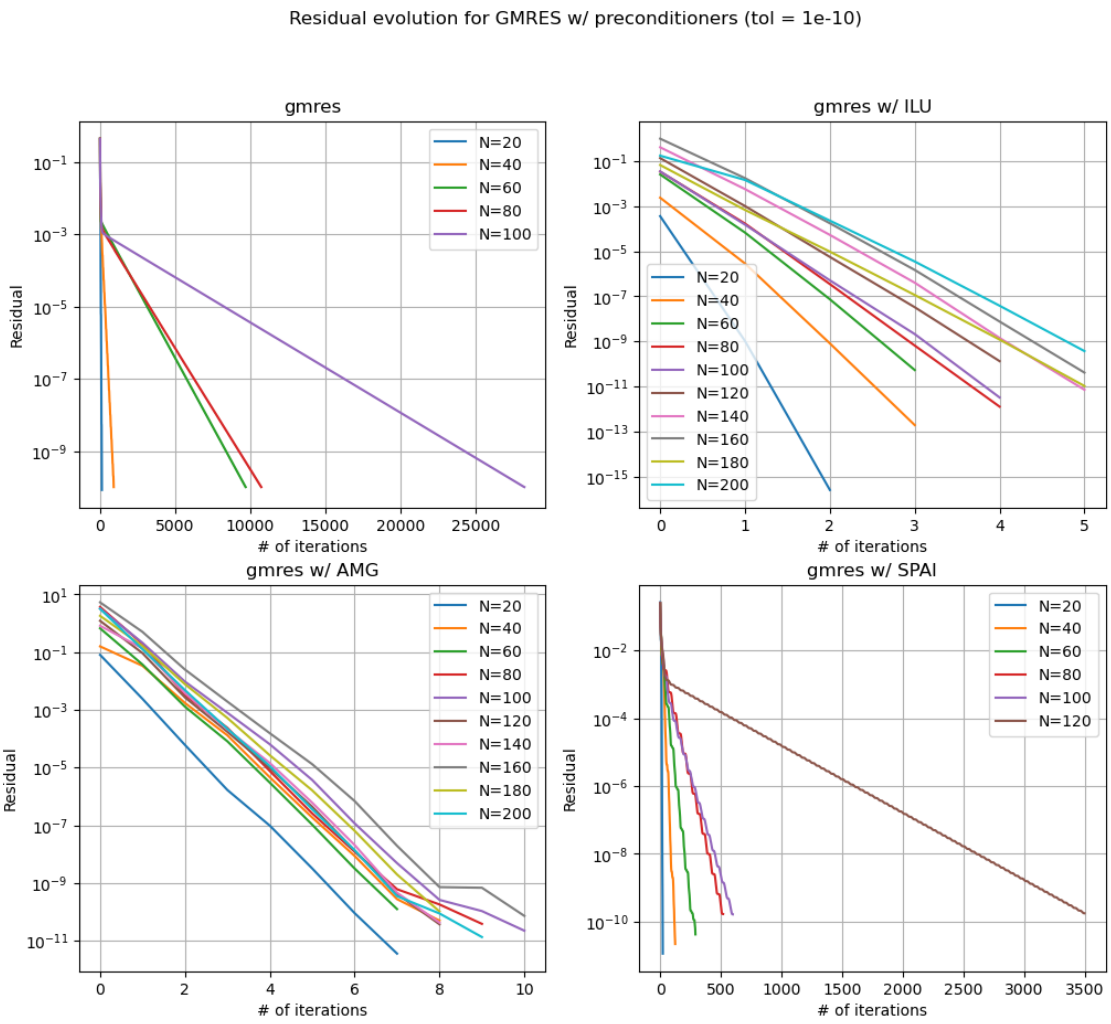
Note: Though cg w/ AMG starts with a larger time taken than both cg w/ ILU and cg only, we can see it scales better with N and undertakes them. If we extrapolate that result, we can presume that the larger the N, the better cg w/ AMG will be compared to the other three.

Hence, the winner for CG across the tested preconditioners is cg w/ AMG.

3.2.3 3.2.3 Comparing preconditioners for gmres

[34]:
```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 10))

plot_res(ax1, perfs_gmres)
plot_res(ax2, perfs_gmres_ilu)
plot_res(ax3, perfs_gmres_amg)
plot_res(ax4, perfs_gmres_spai)
```

```
plt.suptitle(f"Residual evolution for GMRES w/ preconditioners (tol =␣
  ↪{perfs_cg[0]['tol']})")
plt.show()
```

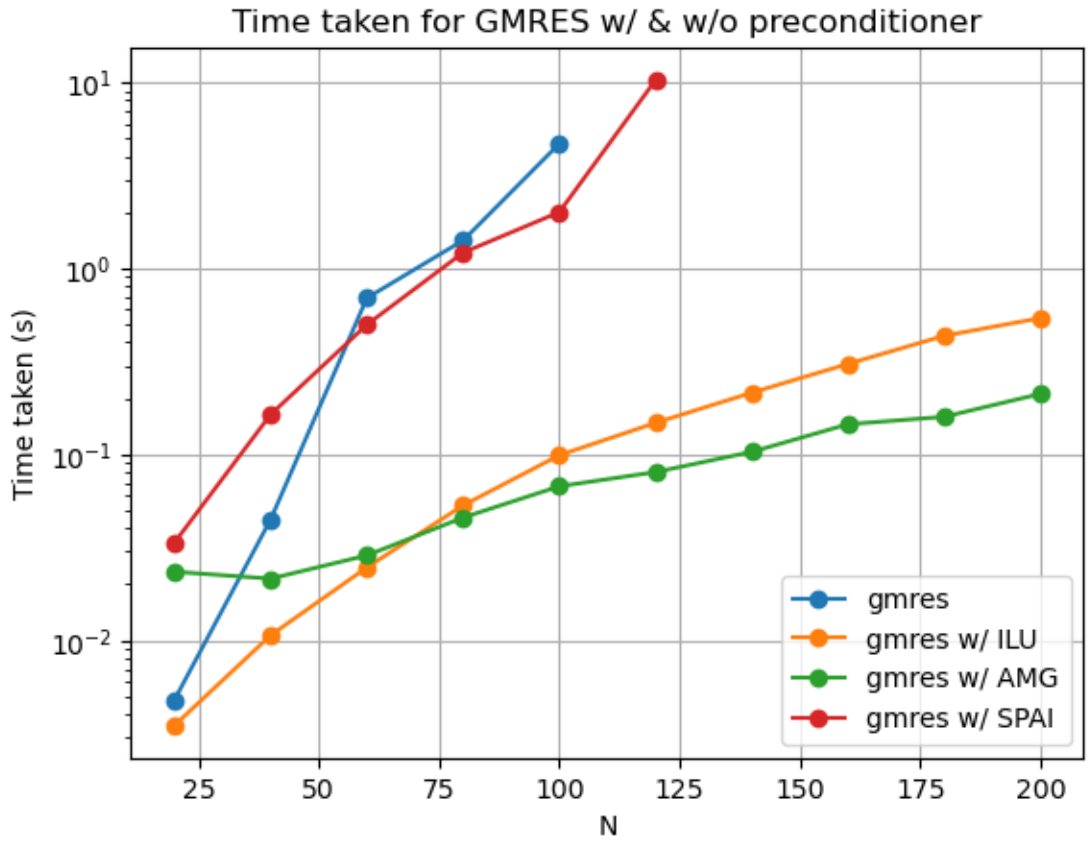


Note: Here, we see a similar trend than observed for cg: the SPAI preconditioner has had some impact, but the shape of the plot still looks very similar to what it looks like without preconditioner; moreover, that impact is negligible when you consider the plots for AMG & ILU.

Though ILU takes a lower number of iterations than AMG, we can see that its slope increases slightly with increasing N. On the other hand, the slope remains seemingly the same for AMG. This seems to indicate that the number of iterations scales better for AMG than for ILU (though they are both an extremely good improvement over gmres without any preconditioners).

However, the real evalution of the performance lies in the time taken, which we will now look at.

[35]:
```
# Comparing across preconditioners

all_perfs = [perfs_gmres, perfs_gmres_ilu, perfs_gmres_amg, perfs_gmres_spai]

for perfs in all_perfs:
    times = [perf["time"] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]

    plt.plot(Ns, times, "-o", label=perfs[0]["name"])

plt.grid()
plt.xlabel("N")
plt.ylabel(f"Time taken (s)")
plt.title(f"Time taken for GMRES w/ & w/o preconditioner")
plt.yscale("log")
plt.legend(loc="lower right")
plt.show()
```



Note: Though ILU performs better for smallest N, the slope with AMG is smaller (a.k.a. it scales better with N), such that AMG becomes better (a.k.a. faster) from around N=60 onwards. We can

extrapolate this trend and speculate AMG will keep getting better over the other preconditioners as N increases. On the other hand, gmres only & gmres w/ SPAI scale quite similarly, and much worse than ILU and AMG.

Hence, the winner for GMRES across the tested preconditioners is GMRES w/ AMG.

3.3 3.3 Final comparison of the best solvers

The best solvers we have found (but haven't yet compared) are: * GMRES w/ AMG * CG w/ AMG * umfpack

Note: It is important to note that the iterative solvers have to compute the residual at each iteration, while direct solver only compute it once and outside of the timing. This induces a biais in the evaluation of the performance in favour of direct solvers.

Hence, for this round of assessment, we first test whether evaluating the residuals makes a difference in performance for gmres w/ amg & cg w/ amg. If it does, then we know that their evaluation does induce a bais.

[36]:
```
# We make the tolerance higher for the iterative solvers to get closer to the␣
 ↪base tolerance of umfpack
# (to make a stronger case), and increase the range of N to get more␣
 ↪information on the time complexity.

tol = 10**-13
N_store = [50, 100, 150, 200, 250, 300]
```

3.3.1 3.3.1 Making sure residual callback doesn't affect performance recording

[37]:
```
# **** Evaluating GMRES w/ AMG as pre-conditioner with residuals computed ****

perfs_gmres_amg = []

for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res), tol=tol,␣
 ↪maxiter=maxiter)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
```

```
        "name": "gmres w/ AMG (w/ comp. res.)",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_gmres_amg.append(perf)
```

[38]:
```
# **** Evaluating GMRES w/ AMG as pre-conditioner without residuals computed␣
 ↪****

perfs_gmres_amg2 = []

for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.gmres(A, b, M=M, tol=tol, maxiter=maxiter)
    end = time.time()

    time_taken = end - start

    perf = {
        "name": "gmres w/ AMG ((w/o comp. res.))",
        "N": N,
        "tol": tol,
        "time": time_taken,
    }

    perfs_gmres_amg2.append(perf)
```

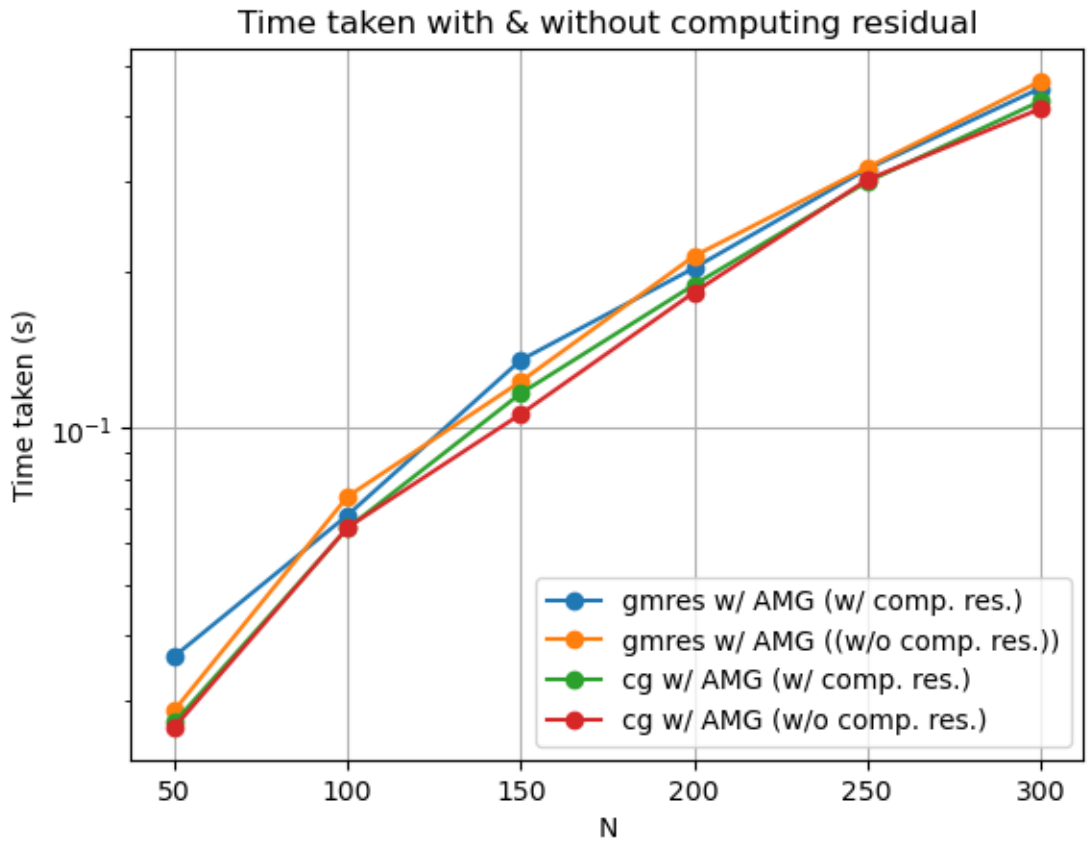
[39]:
```
# **** Evaluating CG w/ AMG as pre-conditioner with residuals computed ****

perfs_cg_amg = []

callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))


for N in N_store:
    residuals = []
```

```
    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.cg(A, b, M=M, callback=callback, tol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "cg w/ AMG (w/ comp. res.)",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
        "res": residuals
    }

    perfs_cg_amg.append(perf)
```

[40]:
```
# **** Evaluating CG w/ AMG as pre-conditioner without residuals computed ****

perfs_cg_amg2 = []

for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.cg(A, b, M=M, tol=tol)
    end = time.time()

    time_taken = end - start
    iters_taken = len(residuals)

    perf = {
        "name": "cg w/ AMG (w/o comp. res.)",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "iters": iters_taken,
```

```
        "res": residuals
    }

    perfs_cg_amg2.append(perf)
```

[41]:
```
# Comparing whether callback makes a difference

all_perfs = [perfs_gmres_amg, perfs_gmres_amg2, perfs_cg_amg, perfs_cg_amg2]

for perfs in all_perfs:
    times = [perf["time"] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]

    plt.plot(Ns, times, "-o", label=perfs[0]["name"])

plt.grid()
plt.xlabel("N")
plt.ylabel(f"Time taken (s)")
plt.title(f"Time taken with & without computing residual")
plt.yscale("log")
plt.legend(loc="lower right")
plt.show()
```



It appears like computing the residual or not makes very little difference to the computing time. This makes sense as the output at each iteration x has size that scales with N, not N^2 (like A does for example), hence it never gets very large.

3.3.2 3.3.2 Making the final comparison

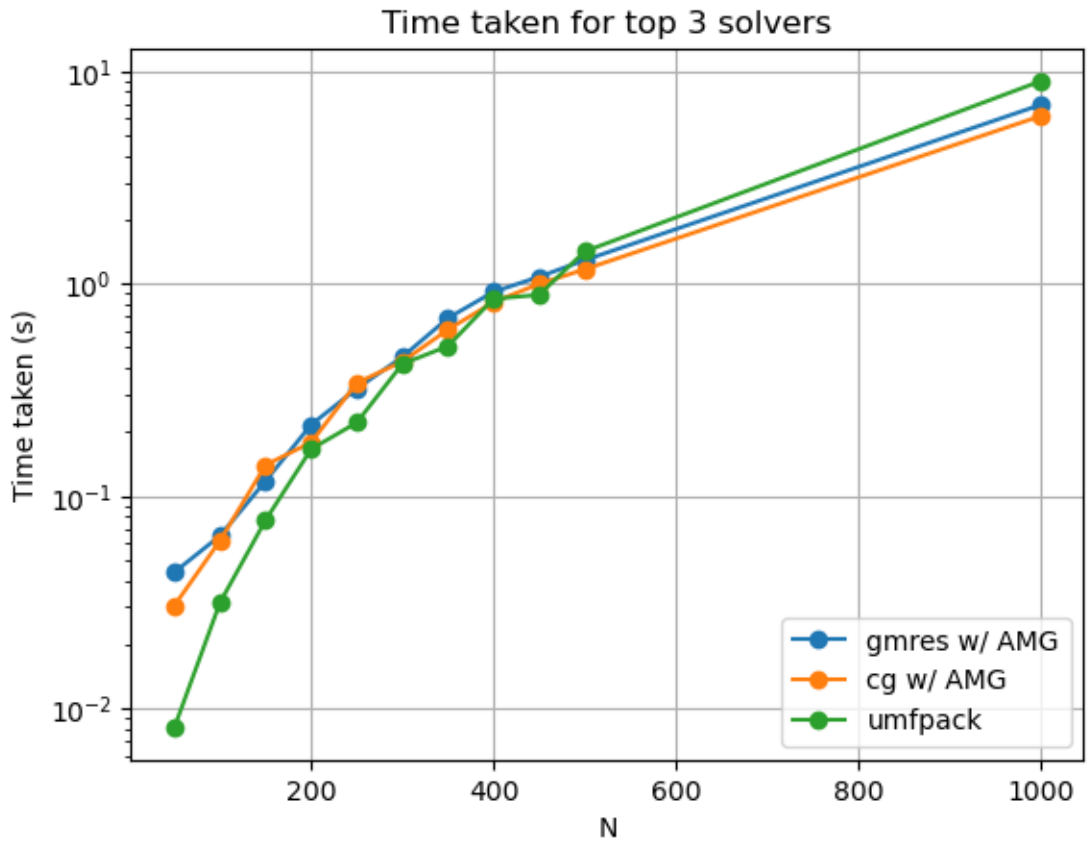
[42]:
```
tol = 10**-13
N_store = [50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 1000]
```

[43]:
```
# **** Re-evaluating all 3 best solvers (w/ computing residual) for the range␣
 ↪of N ****

perfs_umfpack = []

for N in N_store:
    A, b = generate_Ab(N)

    start = time.time()
    x = umfpack.spsolve(A, b)
    end = time.time()

    time_taken = end - start
    residual = np.linalg.norm(b - A @ x) / np.linalg.norm(b)

    # recording performance in a dictionary for code cleanliness
    perf = {
        "name": "umfpack",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "res": [residual] # as array here for cleaner code in plotting
    }

    perfs_umfpack.append(perf)


perfs_cg_amg = []
callback = lambda x: residuals.append(np.linalg.norm(b - A @ x) / np.linalg.
 ↪norm(b))

for N in N_store:
    residuals = []

    A, b = generate_Ab(N)
```

```
    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.cg(A, b, M=M, callback=callback, tol=tol)
    end = time.time()

    time_taken = end - start

    perf = {
        "name": "cg w/ AMG",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "res": residuals
    }

    perfs_cg_amg.append(perf)


perfs_gmres_amg = []

for N in N_store:
    residuals = []

    A, b = generate_Ab(N)

    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res), tol=tol,␣
 ↪maxiter=maxiter)
    end = time.time()

    time_taken = end - start

    perf = {
        "name": "gmres w/ AMG",
        "N": N,
        "tol": tol,
        "time": time_taken,
        "res": residuals
    }

    perfs_gmres_amg.append(perf)
```

[44]:
```
all_perfs = [perfs_gmres_amg, perfs_cg_amg, perfs_umfpack]
```

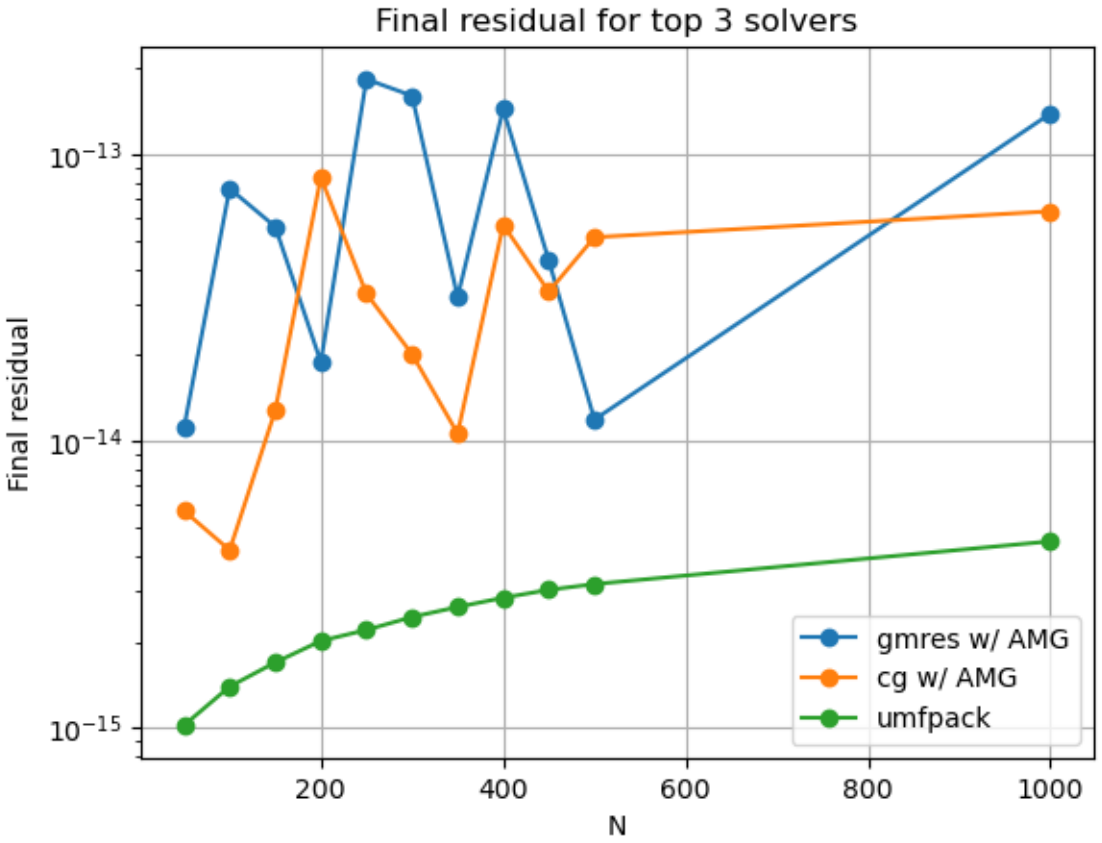
```
for perfs in all_perfs:
    times = [perf["time"] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]

    plt.plot(Ns, times, "-o", label=perfs[0]["name"])

plt.grid()
plt.xlabel("N")
plt.ylabel(f"Time taken (s)")
plt.title(f"Time taken for top 3 solvers")
plt.yscale("log")
plt.legend(loc="lower right")
plt.show()
```



[45]:
```
perfs_gmres_amg[0]["res"][-1]
```

[45]: 1.1183760264910362e-14

[46]:
```
all_perfs = [perfs_gmres_amg, perfs_cg_amg, perfs_umfpack]

for perfs in all_perfs:
    final_res = [perf["res"][-1] for perf in perfs]
    Ns = [perf["N"] for perf in perfs]

    plt.plot(Ns, final_res, "-o", label=perfs[0]["name"])

plt.grid()
plt.xlabel("N")
plt.ylabel(f"Final residual")
plt.title(f"Final residual for top 3 solvers")
plt.yscale("log")
plt.legend(loc="lower right")
plt.show()
```



Final note: All 3 solvers give us remarkably similar performances. The umfpack solver starts out (low N) with much better performance, but scales very slightly less well than gmres & cg with AMG as preconditioner, and ends up being overtaken by them. We can extrapolate this trend and presume that umfpack is slightly less good than the two other for very large N.

Though umfpack consistently gives the best final residual, it is only the case because we have pre-set the acceptable tolerance for the iterative solvers as tol=10**-13. It is reasonable to presume that a final residual as good as umfpack could be achieved.

Now, iterative solvers are more versatile, as the highest precision is not always needed (and hence direct solvers may waste some computational power in trying to get the most exact solution up to computer precision).

Hence, I decide to go with one of the iterative solvers: gmres w/ AMG. (they are both essentially equivalent in terms of performance; cg showed some signs of instability but those seemed to disappear when AMG was added as its preconditioner).

4 Part 4: Increasing N

In this section, you are going to use the solver you picked in part 3 to compute the solution for larger values of N.

The problem we have been solving in this assignment has the exact solution $u_{\text{exact}} = \sin(3x + 4y)$. A measure of the error of an approximate solution can be computed using

$$\sum_{i=0}^{N^2-1} h^2 \left| u_{\text{exact}}(\mathbf{m}_i) - u_h(\mathbf{m}_i) \right|,$$

where m_i is the midpoint of the ith square in the finite element mesh: the value of u_h at this midpoint will be the mean of the values at the four corners of the square. For points on the boundary, we set $u_h = g$ and so combine evaluations of g and values in the solution vector to compute some of the values in this sum.

For a range of values of N from small to large, **compute the solution to the matrix-vector problem. Measure the time taken to compute this solution, and compute the error of the solution. Make plots showing the time taken and error as N is increased.**

Using your plots, **estimate the complexity of the solver you are using** (ie is it $\mathcal{O}(N)$? Is it $\mathcal{O}(N^2)$?), and **estimate the order of convergence of your solution** (your error should decrease like $\mathcal{O}(N^{-\alpha})$ for some $\mathcal{O}(N^{-\alpha})$). Briefly (1-2 sentences) **comment on how you have made these estimates of the complexity and order.**

[47]:
```
def u_exact(x, y):
    """
    Computes the exact solution for u
    """
    return np.sin(3*x+4*y)

def u_midpoints(u):
    """
    Computes the value of u at each of the midpoints, provided the value of
    u at each of the grid points,

    Takes as input an array of shape (N+1, N+1) and returns one of shape (N, N)
```

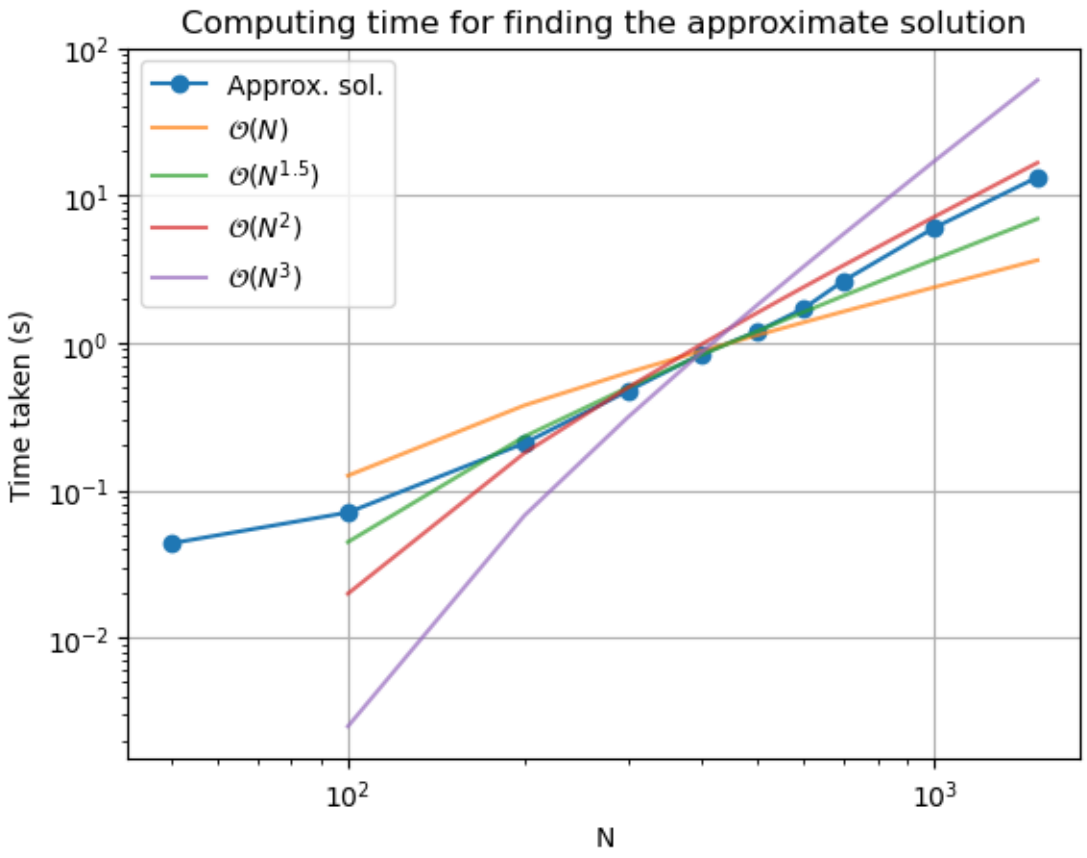
```
    """

    # The variable name indicates the location of the points
    # relative to the square they will contribute to
    upper_left = u[:-1, :-1]
    lower_left = u[1:, :-1]
    upper_right = u[:-1, 1:]
    lower_right = u[1:, 1:]

    # midpoint values for each square as the mean of the value at the 4 corners
    u_midpoints = np.mean(np.array([upper_left, lower_left, upper_right,␣
 ↪lower_right]), axis=0)

    return u_midpoints

def u_error(u_h, N):
    """
    Computes the error of approximate solution u_h according to the definition␣
 ↪given
    """
    h = 1/N

    # Computing the value of u_h at the midpoints
    u_h_midpoints = u_midpoints(u_h)

    # value of the coordinate x (or equivalently y) at each midpoint
    x_midpoints = np.arange(h/2, 1, h)

    X, Y = np.meshgrid(x_midpoints, x_midpoints, indexing="ij")

    # exact value of u at each midpoint
    u_ex = u_exact(X, Y)

    u_err = np.sum(h**2 * np.abs(u_h_midpoints - u_ex))

    return u_err

def best_solver(N):
    """
    Computes and returns u_h for the best solver (GMRES with AMG as␣
 ↪preconditioner)
    (picked at the end of part 3)
    """
    A, b = generate_Ab(N)

    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
```

```
    x, _ = sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res),␣
 ↪tol=tol, maxiter=maxiter)

    return get_u(x, N)
```

[48]:
```
# Computing the error and time taken for the best solver

N_store = np.array([50, 100, 200, 300, 400, 500, 600, 700, 1000, 1500])

tol = 10**-10

time_store = []
err_store = []

for N in N_store:
    A, b = generate_Ab(N)

    # Computing approximate solution
    start = time.time()
    ml = pyamg.smoothed_aggregation_solver(A.tocsr())
    M = ml.aspreconditioner()
    x,_ = sp.gmres(A, b, M=M, callback=lambda res: residuals.append(res),␣
 ↪tol=tol, maxiter=maxiter)
    end = time.time()

    # Computing error
    u_h = get_u(x, N)
    u_err = u_error(u_h, N)

    time_store.append(end-start)
    err_store.append(u_err)
```

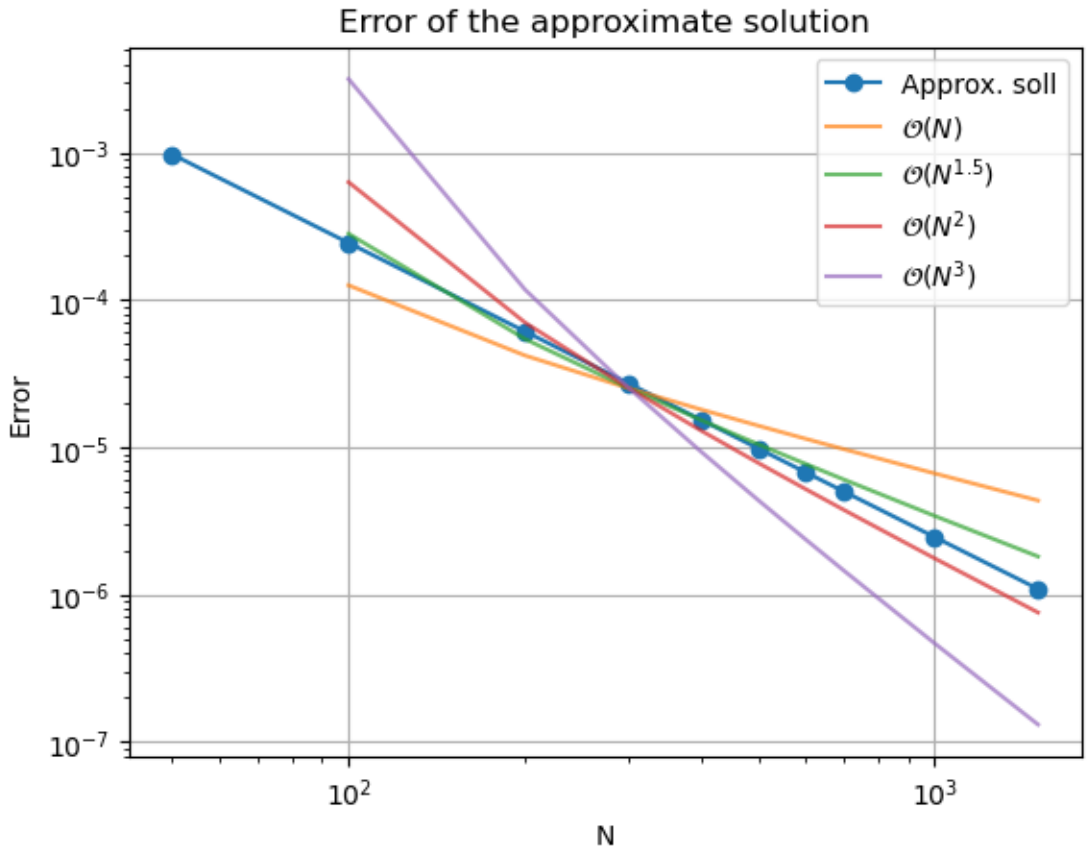
[49]:
```
x = N_store[1:]
y = N_store[1:]-N_store[0]

# the fits are multiplied by a constant factor. On a log plot, this is␣
 ↪equivalent to shifting the line
# up or down, and doesn't alter the slope of the line

plt.plot(N_store, time_store, "-o", label="Approx. sol.")
plt.plot(x, (10**-2.6)*y, label="$\mathcal{O}(N)$", alpha=0.7)
plt.plot(x, (10**-3.9)*y**1.5, label="$\mathcal{O}(N^{1.5})$", alpha=0.7)
plt.plot(x, (10**-5.1)*y**2, label="$\mathcal{O}(N^2)$", alpha=0.7)
plt.plot(x, (10**-7.7)*y**3, label="$\mathcal{O}(N^3)$", alpha=0.7)
plt.grid()
```

```
plt.xlabel("N")
plt.ylabel("Time taken (s)")
plt.title("Computing time for finding the approximate solution")
plt.yscale("log")
plt.xscale("log")
plt.legend()
plt.show()
```



[50]:
```
x = N_store[1:]
y = N_store[1:]-N_store[0]

# the fits are multiplied by a constant factor. On a log plot, this is␣
 ↪equivalent to shifting the line
# up or down, and doesn't alter the slope of the line

plt.plot(N_store, err_store, "-o", label="Approx. soll")
plt.plot(x, (10**-2.2)*1/y, label="$\mathcal{O}(N)$", alpha=0.7)
plt.plot(x, (10**-1)*1/y**1.5, label="$\mathcal{O}(N^{1.5})$", alpha=0.7)
plt.plot(x, (10**0.2)*1/y**2, label="$\mathcal{O}(N^2)$", alpha=0.7)
```

```
plt.plot(x, (10**2.6)*1/y**3, label="$\mathcal{O}(N^3)$", alpha=0.7)
plt.grid()
plt.xlabel("N")
plt.ylabel("Error")
plt.legend()
plt.title("Error of the approximate solution")
plt.yscale("log")
plt.xscale("log")
plt.show()
```



Comment Using the error as our example (but everything still applies for time taken), we approximate the relationship with N as $error = A * N^{\alpha}$ where A is a constant, and taking the logarithm on both sides, we arrive at relationship $log(error) = \alpha log(N) + log(A)$, which is the equation for a straight line.

Hence, making both of our axes logarithmic when plotting error/time against N, we do trial and error for different lines of best fit for a set of values of α (and guessing the corresponding A a.k.a the y-axis intercept), we take our best guess for α as that whose slope matches the plot most closely. We end up with estimates to time complexity $\sim \mathcal{O}(N^{1.5})$ and error $\sim \mathcal{O}(N^{-1.5})$.

5 Part 5: parallelisation

In this section, we will consider how your solution method could be parallelised; you do not need, however, to implement a parallel version of your solution method.

Comment on how your solution method could be parallelised. Which parts (if any) would be trivial to parallelise? Which parts (if any) would be difficult to parallelise? By how much would you expect parallelisation to speed up your solution method?

If in part 4 you used a solver that we have not studied in lectures, you can discuss different solvers in parts 4 and 5.

Comment:

Generating A & b:

- Computing the elements of b is trivial to parallelise. They only require a thread each, and a priori no shared memory as their value only depends on their position in the array.
- For A, we only need to compute the non-zero elements, which all lie on diagonals of the matrix. Each of these elements can also be computed on a separate thread with no shared memory needed for the same reason as for the elements of b.
- Hence both the computations of A and of b should be trivial to parallelise. The size of the diagonals of A scales like $O(N^2)$, while that of b like $O(N)$. If there are enough threads to compute all elements of the matrix/vector at once, then the complexity drops to $O(1)$. However, if for example $threads\_count < N^2$, then the computation for A will need to be done in batches instead. The # of batches will be given by $N^2/thread\_count$. The time complexity would then still scale like N^2, but it would be shifted down, and should remain a small number for not too large sizes of N (e.g. not 10^30).

Preconditioner (AMG):

- The Analytical Multigrid method has not been studied in the course (it has only been mentioned). As suggested in the prompt, I choose to discuss gmres with no preconditioner instead.

Solver (GMRES):

- GMRES being an iterative solver, it looks on first approach like it might be a good idea to attempt to paralelise the iterative loop. However, since the next iteration needs the output of the previous iteration, the iterative looping is forced to be serial, and hence cannot be parallelised.
- GMRES is built on the Arnoldi iteration (that is, it needs to compute a Krylov subspace made of orthogonal vectors) and a least-squares problem. The most computationally expensive tasks of this algorithm seem to be the matrix vector products (scales proportionally to the outer dimensions of the product).
- Parallelising a matvec can be tricky because each term of the matrix and of the vector are used to compute multiple of the terms of the final product. For example, for matvec of an NxN matrix with an N vector, each term of the matrix and of the vector appear once in each term of the product, and are therefore needed N times. A possible implementation would be for each thread to compute one element of the resulting vector, and to store in each thread's local memory exactly the data it needs: one row of the matrix and the whole vector: this corresponds to $2N$ entries i.e. space complexity $O(N)$. Each thread would then have time

complexity $O(N)$, and provided the *threads_count* is not too small compared to N (i.e. only few batches are needed), matvec would also scale as $O(N)$ time complexity.

- Now, if the matvec concerns a matrix like A (sparse, with predictable location of the non-zero elements, and with the number of non-zero elements scaling with matrix side length N (as is the case for A)), then the time complexity in each thread scales like $O(1)$, and provided also that the # of threads is not too small compared to N, the entire matvec also scales like $O(1)$.
- These implementations would boost massively the performance of gmres. Indeed, a lot of its matrix vector products concern A (as the Krylov subspace must be computed, which is made of elements $\{b, Ab, A^2b, ...\}$)

[ ]: