project3_demo1

December 3, 2024

1 Finite Difference method for solving discrete Laplace Equation

1.0.1 Exercise 1: solve the 4x4 linear equation A.x = b

`4*u[i,j] - u[i-1,j] - u[i+1,j] - u[i,j-1] - u[i,j+1] = 0`

Dervie and solve the linear system A.x =b

REF: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve.html#scipy.linalg.solve

[1]:
```
import numpy as np
from scipy import linalg
from scipy.linalg import solve
```

[59]:
```
A = np.array([
    [4, -1, -1, 0],
    [-1, 4, 0, -1],
    [-1, 0, 4, -1],
    [0, -1, -1, 4]
])

b = np.array([0, 0, 1, 1])

x = solve(A, b)

print(x)
```

```
[0.125 0.125 0.375 0.375]
```

1.0.2 Exercise 2: arbitrary size of the matrix.

If we want our solve could sovle an arbitrary size of the system NxN in 2D.
First, we need to generate the left-hand matrix.
The left-hand matrix contains two components: One is an diagnoal matrix with only three banded values.
The other component conatins negative identity matrice.

we could use the `dia_matrix` in `scipy.sparse` and `np.identity()` for these components.

REF: https://docs.scipy.org/doc/scipy/reference/sparse.html

[2]:
```python
import numpy as np
from scipy.sparse import dia_array  # if dia_array is not able, use dia_matrix
from scipy.sparse import dia_matrix
from numba import jit, njit, prange
```

Part 1:

Write a function to generate the matrxi A with arbitrary size N.
The shpae of the matrix A is (N^2, N^2).

Hints: depedning on your implmentation, you might want to use `numba` to speed it up.

We could decompose the matrix into several (N,N) submatrix and initialize the diagonal matrixs and the offset terms seperately.

[3]:
```python
def generate_the_laplace_matrix_with_size(N=4):
    """
    assume sqrt(N) is an integer.

    """
    nsq = N*N
    A   = np.zeros((nsq,nsq))

    # TODO
    for i in range(N):
        for j in range(N):
            index = i * N + j
            A[index, index] = 4
            if j > 0:
                A[index, index - 1] = -1
            if j < N - 1:
                A[index, index + 1] = -1
            if i > 0:
                A[index, index - N] = -1
            if i < N - 1:
                A[index, index + N] = -1

    return A
```

[4]:
```python
N = 4
A = generate_the_laplace_matrix_with_size(N)
print(A)
```

```
[[ 4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  4.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.  0.  0.  0.  0.]
```

```
 [ 0.  0.  0. -1.  0.  0. -1.  4.  0.  0.  0. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  0.  0.  0.  4. -1.  0.  0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.  0.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0.  4. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0. -1.  4.]]
```

Part2:

The right hand side of the lienar equation is a vecotr.
generate a vecotr is simple with `np.array()`

[4]:
```python
def generate_the_rhs_vector_with_size(N=4):
    b = np.zeros(N*N)
    #TODO
    b[-N:] = 1   # Setting the boundary conditions for the first few points
    return b
```

[6]:
```python
b = generate_the_rhs_vector_with_size(N=N)
print(b)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
```

Part 3:

Once we have the linear problem `A x = b`, we could solve the system with `scipy.linalg.solve`

REF: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve.html#scipy.linalg.solve

[5]:
```python
from scipy import linalg
```

[8]:
```python
x = linalg.solve(A, b)
```

Part 4:

Once we have the solution, we should convert the solution vector to the finite difference grids `u[i,j]`.

[6]:
```python
def convert_solution(x):
    usize = np.sqrt(len(x))
    u = x.reshape(int(usize),int(usize)).transpose()
    return u
```

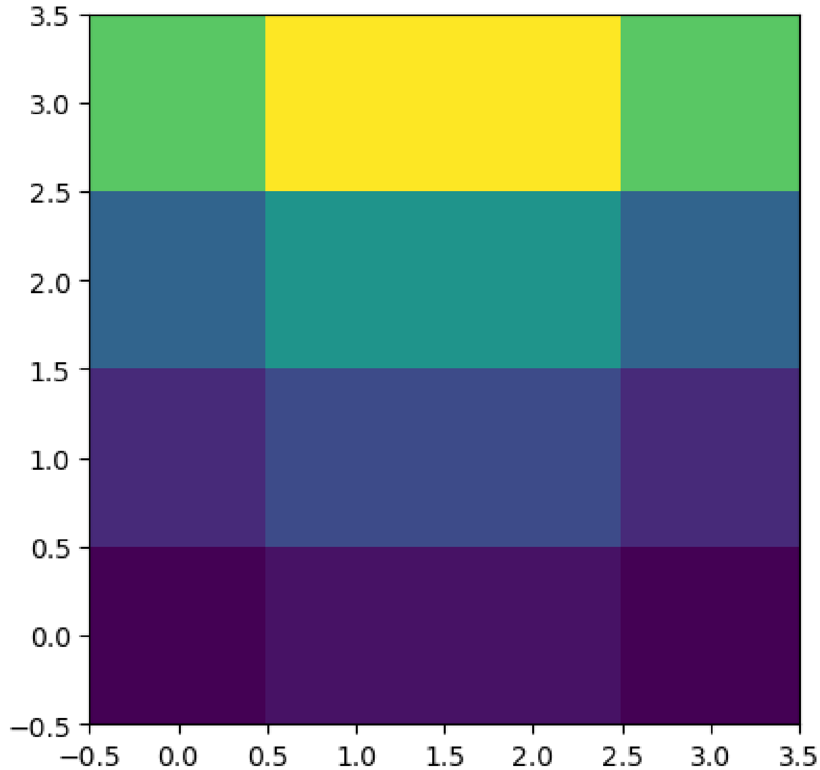
Part 5:

Now, let's visualize the solution with `matplotlib`

[7]:
```python
import matplotlib.pyplot as plt
```

[11]:
```python
u = convert_solution(x)
plt.imshow(u.T,origin="lower")
```

[11]: <matplotlib.image.AxesImage at 0x21b672181d0>



Part 6:

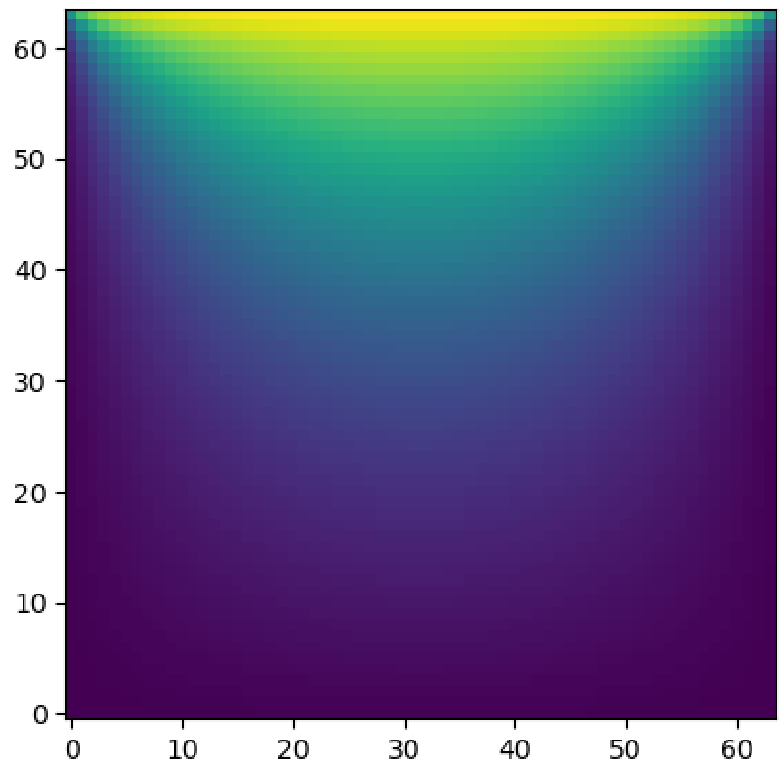
Now we have tested our code, we could write a "solver" function to wrap all necessary codes. This solver function could be either in the notebook or in a seperate python file.

[8]:
```python
def solve_laplace(N=16):
    A = generate_the_laplace_matrix_with_size(N=N)
    b = generate_the_rhs_vector_with_size(N=N)
    x = linalg.solve(A,b)
    u = convert_solution(x)
    return u
```

[13]:
```python
u = solve_laplace(N=64)
```

[14]:
```python
plt.imshow(u.T,origin="lower")
```

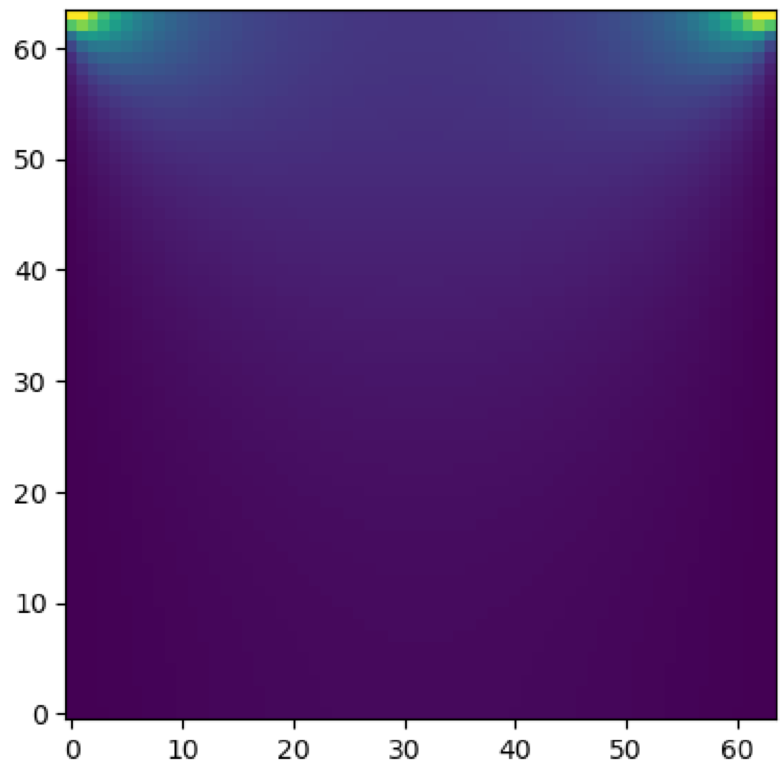
[14]: <matplotlib.image.AxesImage at 0x21b07186e90>



[15]:
```
field = np.gradient(u)
```

[16]:
```
plt.imshow(field[1].T,origin="lower")
```

[16]: <matplotlib.image.AxesImage at 0x21b672083d0>



Part 7:

The buttole neck of this solver is in the `linalg.solve()`.
Let's measure the performance of `linalg.solve()` with different resolutions.

[9]:
```
import time
```

[13]:
```
resolutions = np.array([8,16,32,64,128])
times       = np.zeros(len(resolutions))

for i, N in enumerate(resolutions):
    print("Measuring resolution N = ", N)
    A = generate_the_laplace_matrix_with_size(N)
    b = generate_the_rhs_vector_with_size(N)
    t1 = time.time()
    x  = linalg.solve(A,b)
    t2 = time.time()
    times[i] = (t2-t1)
```

```
Measuring resolution N =  8
Measuring resolution N =  16
Measuring resolution N =  32
Measuring resolution N =  64
```

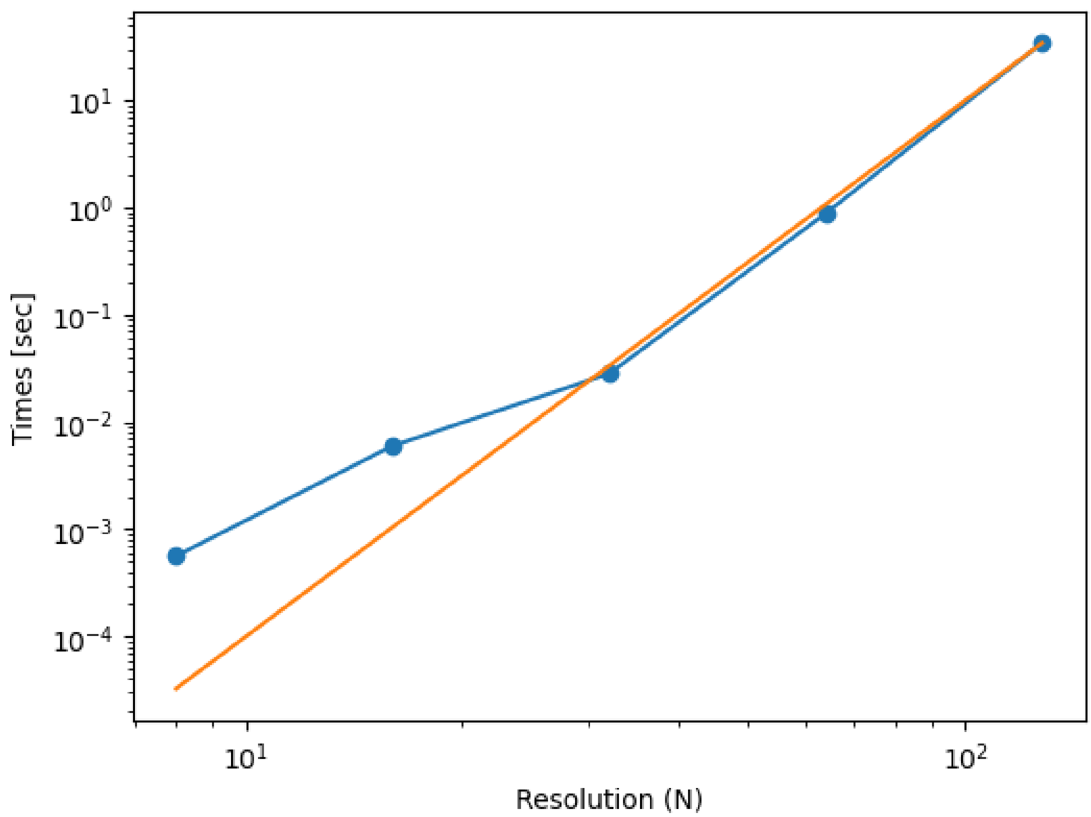
```
Measuring resolution N =  128
```

Part 8:

Plot Computing time vs N in log-log scale.

[12]:
```python
plt.figure(1)
plt.plot(resolutions,times,'-o')
plt.plot(resolutions,times[-1]*resolutions**5/resolutions[-1]**5)
plt.xscale('log')
plt.yscale('log')
plt.xlabel("Resolution (N)")
plt.ylabel("Times [sec]")
```

[12]:
```
Text(0, 0.5, 'Times [sec]')
```



You could see that the perofmance is propotional to `N^5`!
This is becasue the size of the matrix A is proportional to `N^2` and
the calculation time of `linalg.solve(M,b)` is proportial to the cubic of the size of M, N^3,
giving `N^5` at the end.

The `scipy.linalg.solve` is robust, but since the matrix A is a sparse matrix, we could use special method to solve it.

REF: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html

[10]:
```
from scipy.sparse import csc_matrix
import scipy.sparse.linalg as splinalg
```

[14]:
```
N = 16
A = generate_the_laplace_matrix_with_size(N)
A = csc_matrix(A)
b = generate_the_rhs_vector_with_size(N)
x = splinalg.spsolve(A,b)
```

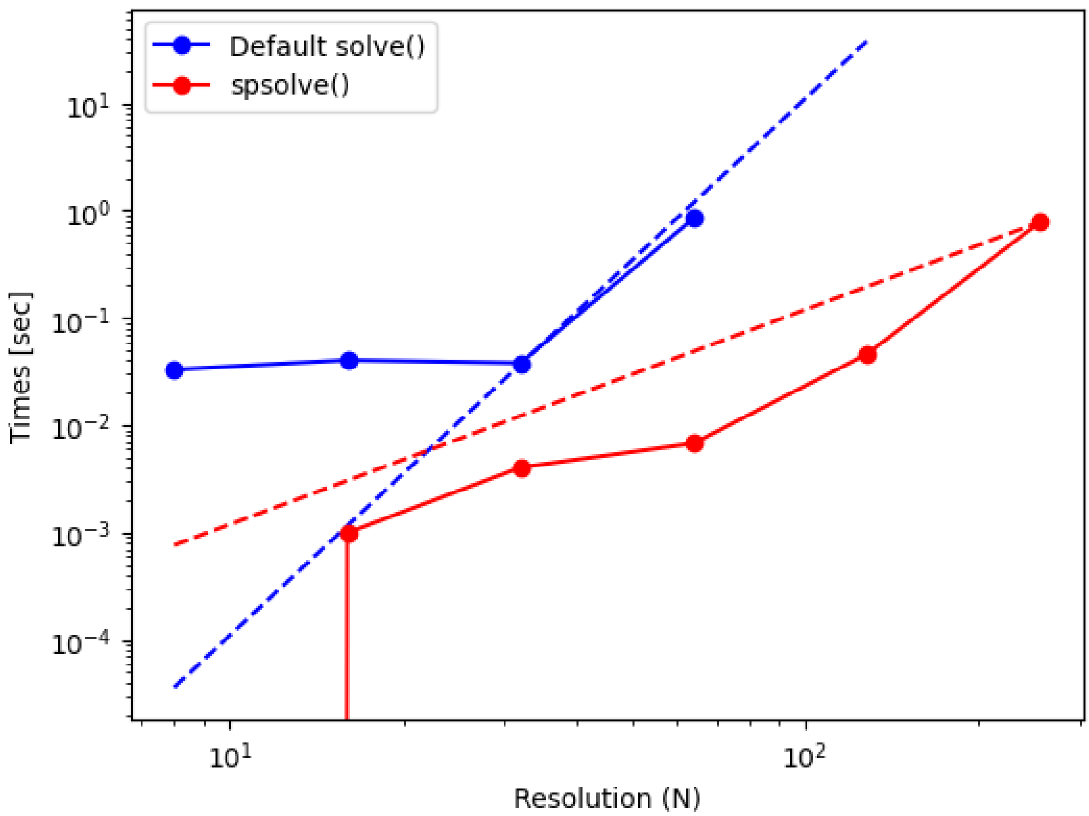
Let's measure the performance again with `spsolve()`

[16]:
```
resolutions = np.array([8,16,32,64,128,256])
times_sp    = np.zeros(len(resolutions))

for i, N in enumerate(resolutions):
    print("Measuring resolution N = ", N)
    A = generate_the_laplace_matrix_with_size(N)
    A = csc_matrix(A)
    b = generate_the_rhs_vector_with_size(N)
    t1 = time.time()
    x  = splinalg.spsolve(A,b)
    t2 = time.time()
    times_sp[i] = (t2-t1)
```

```
Measuring resolution N =  8
Measuring resolution N =  16
Measuring resolution N =  32
Measuring resolution N =  64
Measuring resolution N =  128
Measuring resolution N =  256
```

[17]:
```
resolutions1 = np.array([8,16,32,64,128])

plt.figure(1)
plt.plot(resolutions1[:-1],times[:-1],'b-o',label="Default solve()")
plt.plot(resolutions1,times[-1]*resolutions1**5/resolutions1[-1]**5,'b--')
plt.plot(resolutions,times_sp,'r-o', label="spsolve()")
plt.plot(resolutions,times_sp[-1]*resolutions**2/resolutions[-1]**2,'r--')
plt.xscale('log')
plt.yscale('log')
plt.legend(loc='upper left')
plt.xlabel("Resolution (N)")
plt.ylabel("Times [sec]")
```

[17]:
```
Text(0, 0.5, 'Times [sec]')
```



You got several oreders of speedup if you know the matrix is a sparse matrix.