HPSC - Assignment 1

NGUYEN T. Hoang - SID: 15M54097

Spring 2016, W831 Mon-Thu. Period 1-2

Due date: 2016/05/10

Problem

Measure the convergence rate of FDM `step09.py`, given the error function:

$$\text{error} = \sqrt{\sum_{i,j=1}^{nx,ny} \frac{(p_{exact} - p_{approx})^2}{p_{exact}^2}}$$

The exact solution is available from BEM `step02.py`:

$$p_{exact} = \frac{x}{4} - 4 \sum_{n=odd}^{\infty} \frac{1}{(nx)^2 \sinh 2n\pi} \sinh n\pi x \cos n\pi y$$

The source code and jupyter notebook for this assignment can be found at:
`https://github.com/gear/HPSC/tree/master/hw`

Answer

Using the given code for FDM and exact solution in the lectures, I extract the boundary points from the solution of FDM and compare with the exact solution.

Extracting boundary points In this assignment, I rewrite `step09.py` of FDM as a function named `fdm` (file: `assign1.py`). The parameters of this function is:

- `nx`: x-axis resolution.
- `ny`: y-axis resolution.
- `nit`: number of time step.
- `draw`: (boolean) plot the data.

`fmd`'s output is a nx-by-ny numpy array with the final values of the solution of 2D Laplace's equation for the given number of time step `nit`. Function `get_border` is used to generate a 1-D array border from the 2D output. To match it with the exact result output of the function `exact`, the extracting order is given as follow:

Listing 1: Get border solution from 2D FDM

```python
# Extracted from assign1.py
...
def get_border(a):
    size = a.shape
    length = size[0]
    size = 2*(size[0] + size[1])
    ret = np.zeros(size)
    ret[0:length] = a[:,0]
    ret[length:2*length] = a[length-1,:]
    temp = a[:,length-1]
    ret[2*length:3*length] = temp[::-1]
    temp = a[0,:]
    ret[3*length:] = temp[::-1]
    return ret[::-1]
...
```

Calculating error The first problem I have with the given error function is the fact that it might contain zero division when p_{exact} is zero. Besides, the second problem is about error term normalization and hence can be difficult to comprehence the result. The solutions for these problems:

- Zero division: Introduce a tolerance parameter of small value. If p_{exact} is smaller than this value, we use this tolerance value instead of real value of p_{exact}. The function named `error` in `assign1.py` implements this solution.

- Normalization: Each term inside the square root of the given error function is a square of the relative error of a data point. It is sufficient to divide each of these terms to the total number of data point in the sense that each error contributes a small portion to the overall error. In addition to the division, we can also introduce a different way to compute relative error. Instead of dividing to p_{exact}, we can divide the difference to $(p_{approx} + p_{exact})$. With this scheme, we do not have to use the extra tolerance variable. The function named `error_rel` implements the new relative error and the normalization scheme. The error functions are re-defined as follow:

$$\text{error} = \sqrt{\frac{1}{n}\sum_{i,j=1}^{nx,ny} \frac{(p_{exact} - p_{approx})^2}{p_{exact}^2}}$$

$$\text{error\_rel} = \sqrt{\frac{1}{n}\sum_{i,j=1}^{nx,ny} \frac{(p_{exact} - p_{approx})^2}{(p_{exact} + p_{approx})^2}}$$

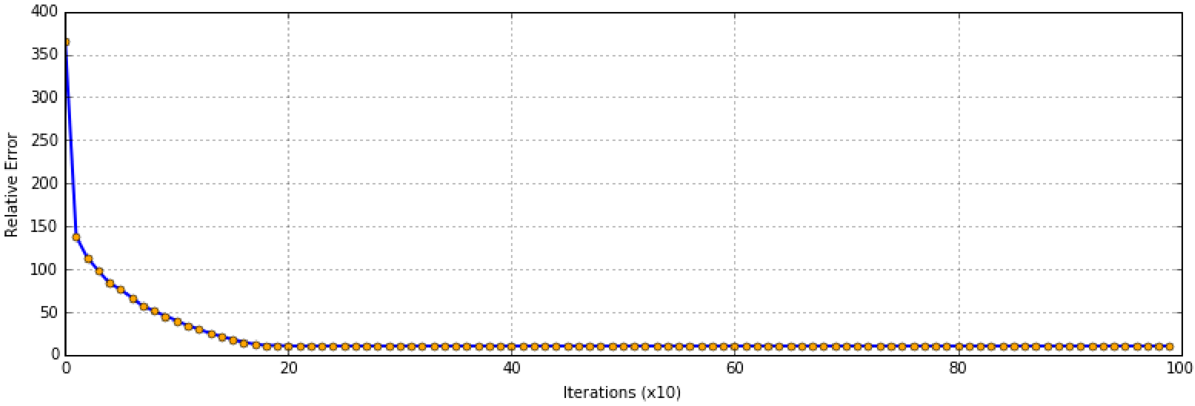
Plotting error In this assignment, I choose to compute 128 boundary points for the exact solution, therefore the FDM solution has the shape of (32,32). To observe the convergence rate, a 100-elements array storing FDM solutions by number of iteration ranging from 0 to 990 with step of 10 is generated.

Listing 2: Error plot

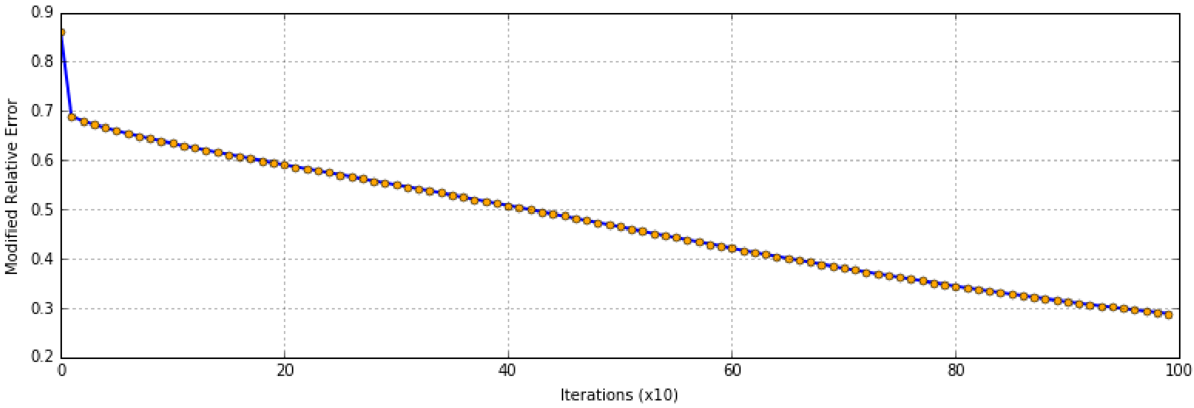
```
import matplotlib.pyplot as plt
import assign1 as a
import numpy as np

_,_,exact = a.exact(128)
fdm = [a.get_border(a.fdm(32,32,i*10)) for i in range(100)]
error_rel = [a.error_rel(exact, fdm[i]) for i in range(100)]
errors = [a.error(exact, fdm[i]) for i in range(100)]

fig1 = plt.figure(figsize=(13,4), dpi=100)
ax = fig1.gca()
ax.plot(error_rel, '-o', ms=5, lw=2, alpha=1, mfc='orange')
ax.grid()
plt.ylabel('Modified␣Relative␣Error')
plt.xlabel('Iterations')
plt.show()

```

```
    fig2 = plt.figure(figsize=(13,4), dpi=100)
    ax = fig2.gca()
    ax.plot(errors, '-o', ms=5, alpha=1, mfc='orange')
    ax.grid()
    plt.ylabel('Relative Error')
    plt.xlabel('Iterations')
    plt.show()
```

The result plots:



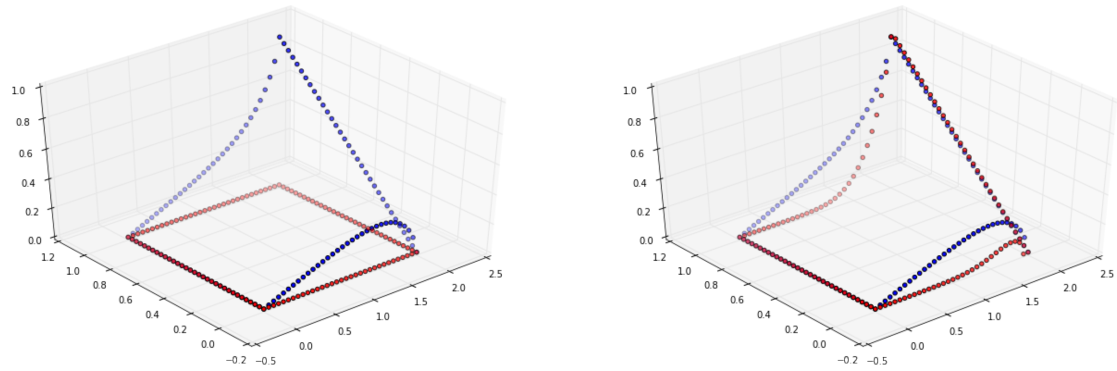
(a) Error computed with given error function.



(b) Error computed with modified error function.

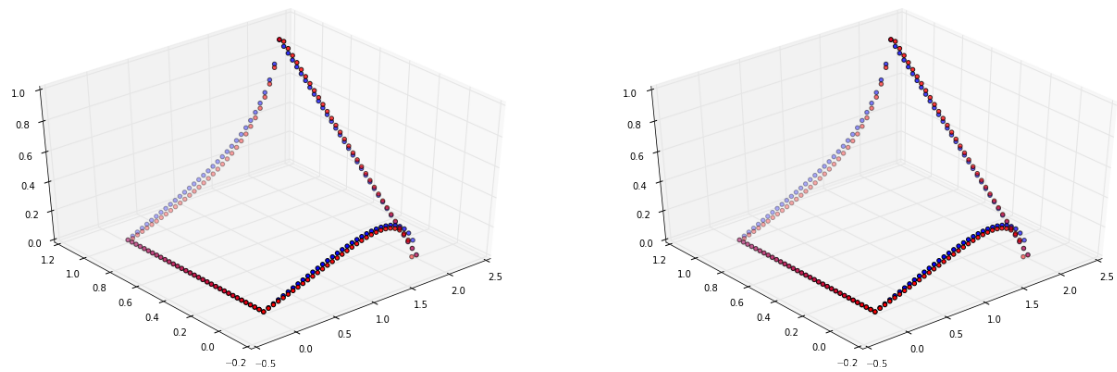
Figure 1: Convergence of error plot.

As we can see in Figgure 1(a), the error rate is converged to a small value starting from 200 iterations. The difference between 200 iterations and 990 iterations is unclear. On the other hand, in Figure 1(b) - the modified error function, we can see the same type of decrease from 0 to 200 iterations. However, the error rate of scheme is bounded by 1, and we can clearly see the convergence of higher iterations. Figure 2 demonstrates the result boundary points at 0 interation and 200 iterations. Figure 3 demonstrates boundary points at 990 iterations and 2000 iterations.



(a) At 0 iteration. e=364.64, er=0.86. (b) At 200 iterations. e=10.50, er=0.59.

Figure 2: Scatter plot of boundary points. e: error; er: error_rel



(a) At 990 iterations. e=10.50, er=0.29 (b) At 2000 iterations. e=10.40, er=0.18

Figure 3: Scatter plot of boundary points. e: error; er: error_rel