600.112: Introduction to Programming for Scientists and Engineers

Assignment 8: Performance with NumPy*

Peter H. Fröhlich  Joanne Selinski
phf@cs.jhu.edu  joanne@cs.jhu.edu

Due Date: 2014/11/11 before 10:00pm!

Introduction

The eighth assignment for 600.112: Introduction to Programming for Scientists and Engineers is all about the Numpy library and how to use it to speed up scientific computations.

There are two things to do: First you’ll re-write the program for two-dimensional heat transfer from Assignment 6 using Numpy. Second you’ll write a program to play CONWAY’s Game of Life, again using Numpy.

There are detailed submission instructions on Piazza which you should follow to the letter! You can lose points if you create more work than necessary for the graders by not following the instructions.

1 NumPy Heat Transfer (30%)

In Assignment 6 you wrote a program to simulate the transfer of heat in a metal plate of negligible thickness. As a reminder, Figure 1 shows some of the images we got from that program for various distributions of heat sources and cooling agents.

For this problem, you’ll re-write the program from Assignment 6 but this time using the Numpy library for scientific computations. That is, your new program should produce the exact same results as the previous program (as far as possible with floating point numbers), but it should be faster because Numpy allows you to move many computations from Python down closer to the machine itself.

*Disclaimer: This is not a course in physics or biology or epidemiology or even mathematics. Our exposition of the science behind the projects cuts corners whenever we can do so without lying outright. We are not trying to teach you anything but computer science!
Recall the overall structure of the program you wrote for Assignment 6. It first reads a description of the heat transfer problem to solve from a file plate.dat in the format shown in Figure 2. The first line gives the size of the plate in both dimensions, the following lines give the boundary conditions: A capital “C” stands for a “cooling agent” which has a temperature of $25^\circ C$ whereas a capital “H” stands for a “heat source” which has a temperature of $100^\circ C$. A period “.” simply means that the temperature of that cell is not fixed but computed as part of the simulation. The program then proceeds to repeatedly average the temperatures around each cell to compute the new temperature of a cell according to the difference equation we derived:

$$T_{i,j} = \frac{1}{4}(T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1})$$

At the edges and corners of the plate where we have only 2 or 3 neighbors instead of 4, the formula is adjusted accordingly. This part of the program takes the most time: We repeatedly “run across” the rows and columns of the plate to produce new values for all computed cells, and all of the computation is carried out “cell-by-cell.” So only the actual arithmetic is performed directly by the machine (fast on average), most of the remaining code (loops, access of matrix elements) is performed by PYTHON (slow on average). Therefore this is the part where we can improve performance the most if we are able to express our solution in terms of matrix operations provided by NUMPY: each of those operations is fast (on average) and deals with all cells of the matrix on the machine level without involving PYTHON again. The last step of the program draws the resulting steady-state heat distribution using matplotlib and we are done.

We need to rethink the way we perform averaging to get away from the idea of doing it “cell-by-cell.” Instead of dealing with a single cell and it’s neighbors, we want to deal with all cells at the same time. Obviously we have to still worry about the edges and corners of the plate where the averaging computation needs to proceed differently than in the center, but let’s focus on the center first. If you consider a cell $(i, j)$ for which we want to compute a new temperature, we first need to sum up the temperatures of the cells around it: $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$. The key insight is to realize that instead of looking at the left $(i, j-1)$ neighbor, we could simply “shift the matrix” one column to the right and use the cell $(i, j)$ of the shifted matrix. Of course the same is true for the other neighbors: Each of which can equally well be found at position $(i, j)$ in some appropriately shifted matrix. As soon as we “line up” the elements correctly, we can use NUMPY’s element-wise matrix addition operation to express the averaging step as follows:

$$\begin{bmatrix} [N] = \frac{1}{4}\left(\uparrow [O] + \downarrow [O] + \leftarrow [O] + \rightarrow [O]\right)\end{bmatrix}$$

Here $[N]$ is the “new” matrix of (averaged) temperatures while $[O]$ is the “old” matrix of temperatures. Obviously this is not yet correct for edges and corners: First we should only add 2 or 3 cells there instead of 4, second we should divide by 2 or 3 instead of 4 to get the correct average. But let’s focus on how we would shift the matrix first; luckily NUMPY provides a function roll that works as demonstrated in Figure 3: We can “shift” a matrix by a certain num-
Figure 3 The `numpy.roll` function in action.

```python
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> numpy.roll(a, 1, 0)
array([[7, 8, 9],
       [1, 2, 3],
       [4, 5, 6]])
>>> numpy.roll(a, -1, 0)
array([[4, 5, 6],
       [7, 8, 9],
       [1, 2, 3]])
>>> numpy.roll(a, 1, 1)
array([[3, 1, 2],
       [6, 4, 5],
       [9, 7, 8]])
>>> numpy.roll(a, -1, 1)
array([[2, 3, 1],
       [5, 6, 4],
       [8, 9, 7]])
```

The `numpy.roll` function in action.

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```

Using these functions, we can shift our matrix appropriately and add the shifted matrices together to compute the sums we need in each new cell: for cells in the center of the matrix, the four shifted matrices will have the correct neighbors at position \((i, j)\); for cells on the edges of the matrix, three of the four shifted matrices will have the correct neighbors while one will have a zero; for cells in the corners of the matrix, two of the four shifted matrices will have the correct neighbors while two will have zeros. However, to compute the averages we still have to divide by the appropriate number of neighbors. Luckily NUMPY allows us to perform element-wise division of a matrix, so if we set up a matrix to be 2 in the corners, 3 on the edges, and 4 everywhere else, we can divide the matrix of sums by that matrix to compute the proper averages. And building that matrix is straightforward as well:

```python
def make_divisor(size):
    base = N.ones((size, size))
    div = up(base) + down(base) + left(←base) + right(base)
    return div
```
Figure 4 The `numpy.where` function in action.

```python
>>> a
array([[False,  True,  True],
       [ True, False,  True],
       [ True,  True, False]], dtype=bool)
>>> b
array([[ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.]]
      dtype=float32)
>>> c
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]]
      dtype=float32)
>>> where(a, b, c)
array([[ 0.,  4.,  4.],
       [ 4.,  0.,  4.],
       [ 4.,  4.,  0.]]
      dtype=float32)
```

of two matrices based on a third boolean matrix; if, for example, all elements of `cond` are `True`, then `numpy.where(cond, a, b)` is just going to be the matrix `a`; if they are all false, it’s going to be the matrix `b` instead. In the solution to Assignment 6 you should have already used such a matrix to keep track of where the boundary conditions are, but again you had to process it one element at a time; now we can use NUMPY to distinguish between boundary conditions and computed cells in one go. So provided we have three matrices, one with the current temperatures, one with the boundary conditions marked by `True`, and one with the appropriate divisors, our averaging step now reads as follows:

```python
def average(old, clamped, divisor):
    sums = up(old) + down(old) + left(old) + right(old)
    averages = sums / div
    new = N.where(clamped, old, averages)
    return new
```

That was simple, wasn’t it? No more neighbors to find, no more values to add up and average, it’s all taken care of by NUMPY. The rest of the program is mostly concerned with input and output just like before, the only difference is that we return three matrices from `read_config` so we can avoid having to recompute the divisor matrix again and again in each averaging step. You may want to look closely at the way we compare matrices now, it’s also using NUMPY of course.

```python
EPSILON = 0.0001
HOT = 100.0
COLD = 25.0
...
def matrices_close_enough(a, b):
    d = a - b
    return max(abs(d.min()), abs(d.max())) <= EPSILON
def set_cell(values, bounds, i, j, char):
    assert char in [".", "H", "C"]
    if char == "H":
        values[i,j] = HOT
        bounds[i,j] = True
    elif char == "C":
        values[i,j] = COLD
        bounds[i,j] = True
    else:
        values[i,j] = 0.0
        bounds[i,j] = False
def read_config(name):
    data = open(name)
    line = data.readline()
    size = int(line)
    assert size > 0
    values = N.zeros((size, size))
    bounds = N.zeros((size, size), dtype=bool)
    i = 0
    for line in data:
        line = line.strip()
        assert len(line) == size
        j = 0
        for char in line:
            set_cell(values, bounds, i, j, char)
```

600.112: Introduction to Programming for Scientists and Engineers Fall 2014
\[ j = j + 1 \]
\[ i = i + 1 \]
```
data.close()
return values, bounds, make_divisor(size)
```
```
def main():
current, bounds, div = read_config("plate.dat")

next = average(current, bounds, div)
while not matrices_close_enough(next, current):
current = next
next = average(current, bounds, div)

P.imshow(current, origin="upper", vmin=COLD, vmax=HOT)
P.axis("off")
P.tight_layout()
P.colorbar(ticks=range(int(COLD-25), int(HOT+25+1), 25)).set_label("temperature")
P.show()
```
```
if __name__ == "__main__":
    main()
```

2 NumPy Game of Life (70%)

The next simulation has slightly more complex rules than the previous one, but in the end there’s even less code to write. CONWAY’s Game of Life is played on a two-dimensional grid of some given size. Each grid is a “cell” which can be either alive or dead. A given starting configuration of cells “evolves” from one step to the next as follows:

- If the cell is alive but has less than two alive neighbors, it dies from loneliness.
- If the cell is alive and has two or three alive neighbors, it stays alive.
- If the cell is alive and has more than three alive neighbors, it dies from overcrowding.
- If the cell is dead and has exactly three alive neighbors, it is reanimated and becomes alive again.

In CONWAY’s game, all eight cells around a given cell are considered its neighbors; edges and corners of the grid obviously have fewer than eight possible neighbors.

For this problem, you’ll write a program that simulates the game of life as outlined above using the NumPy library. This requires that you approach the problem from a matrix perspective: Don’t start by thinking about individual cells too much, think about how you can apply the rules to all cells by using matrix operations. Please call your program life.py and nothing else. Figure 5 illustrates the format your input file life.dat will have, while Figures 6 and 7 illustrate the format of the output your program will generate. Note that certain starting configurations don’t ever change, certain starting configurations never stop changing, and some starting configurations eventually reach a state in which they won’t change again. Your program should always run the simulation until the “steady state” has been reached, but that means you will run forever for some inputs; that’s okay though, in a graphical version of this you’d just close the window, and in our textual version you’ll just have to stop the program manually once you see a loop.
Figure 6 Output for the `life.py` program (part 1).

```
. . . . . .
. . . * . *
. . * * * .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```

Figure 7 Output for the `life.py` program (part 2).

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
```