profiling_handson_with_solution

June 2, 2017

1 Profiling and Optimization Hands-On

```
In [1]: import numpy as np
        from math import sin, cos
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1.1 Task 1: which is faster?

Given a large 2D array, we will explore different ways to create the array and to calculate its mean. Determine which one is fastest, using the `%timeit` notebook function.

1.1.1 setup: define our 2D array:

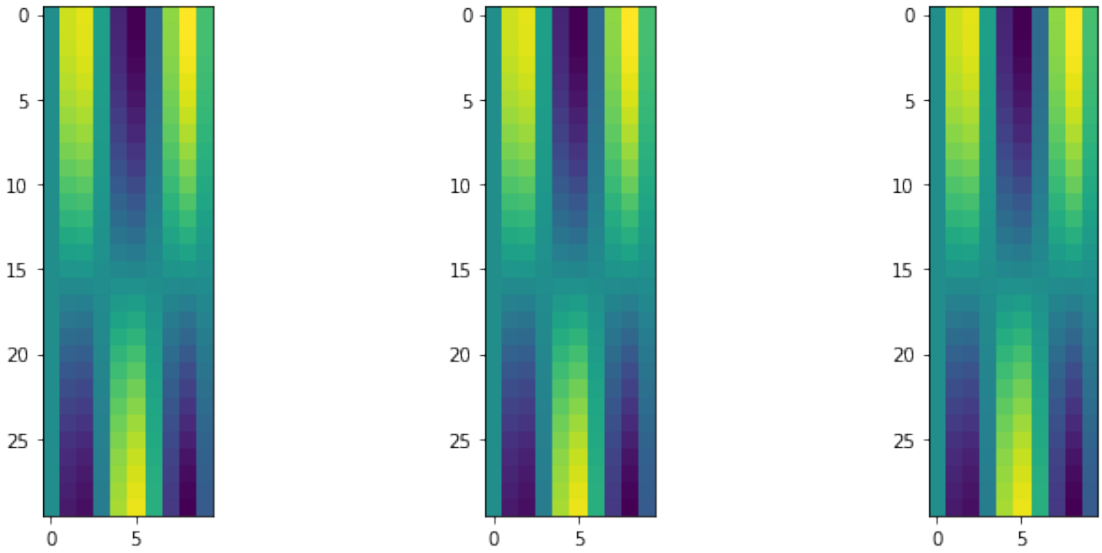
We'll make some dummy test data that looks like:
$M_{ij} = \sin(i)\cos(0.1j)$
and we will construct this array in multiple ways.

```
In [2]: def create_array_loop(N,M):
            arr = []
            for y in range(M):
                row = []
                for x in range(N):
                    row.append(sin(x)*cos(0.1*y))
                arr.append(row)
            return arr

        def create_array_list(N,M):
            """a 2D array using a list-comprehension"""
            return [[sin(x)*cos(0.1*y) for x in range(N)] for y in range(M)]

        def create_array_np(N,M):
            """ a 2D array using numpy"""
            X,Y = np.meshgrid(np.arange(N), np.arange(M))
            return np.sin(X)*np.cos(0.1*Y)
```

Let's first just plot the arrays, to see if they are the same:

```
In [3]: N=10; M=30  # our array dimensions
        plt.figure(figsize=(12,5))
        plt.subplot(1,3,1)
        plt.imshow( create_array_loop(N,M))
        plt.subplot(1,3,2)
        plt.imshow( create_array_list(N,M))
        plt.subplot(1,3,3)
        plt.imshow( create_array_np(N,M))
```

```
Out[3]: <matplotlib.image.AxesImage at 0x114a08438>
```



1.1.2 Task: determine which array creation routine is fastest

And make a plot of the speed of each! Does the result change much when the array size becomes larger? Try much larger sizes for N and M

Hint: use the `%timeit -o` magic function to have `%timeit` return results (see the timeit help)

1.1.3 SOLUTION:

```
In [4]: N=3000; M=1000
```

```
In [5]: res_loop = %timeit -o create_array_loop(N,M)
```

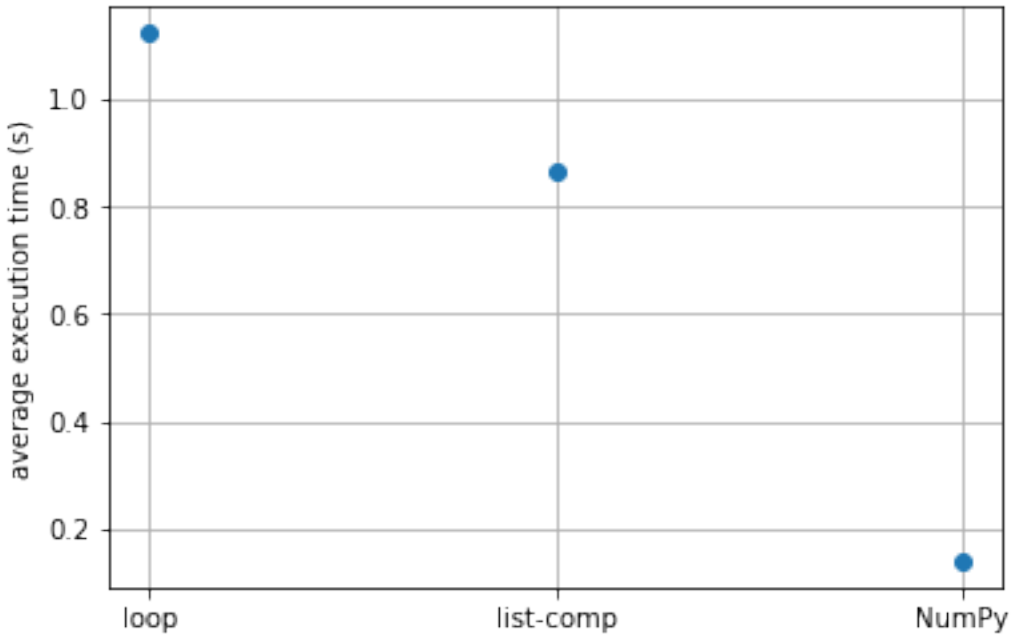
```
1.12 s ś 47.7 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

```
In [6]: res_list = %timeit -o create_array_list(N,M)
```

```
867 ms ś 8.14 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

```
In [7]: res_np    = %timeit -o create_array_np(N,M)

140 ms ś 769 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

```
In [8]: plt.plot([res_loop.average, res_list.average, res_np.average], "o")
        plt.xticks( np.arange(3), ['loop','list-comp','NumPy'])
        plt.ylabel("average execution time (s)")
        plt.grid()
```



1.2 Task 2: determine the fastest way to find the mean of our array

note that `create_array_list()` and `create_array_loop` both return a list-of-lists, while `create_array_np` returns a 2D numpy array. There are multiple ways to compute the mean of these arrays. See again which is fastest!

try at least:

1. using the built-in python `sum` function and either a for-loop or list-comprehension
2. using pure numpy (e.g. `array.mean()`)
3. other ways you can think of!

1.2.1 SOLUTION

```
In [9]: N=100; M=100
        a_list = create_array_list(N,M)
        a_np = create_array_np(N,M)
```

```
In [10]: sum([sum(x) for x in a_list])/(N*M)

Out[10]: -0.00017124964340864182

In [11]: a_np.mean()

Out[11]: -0.00017124964340864226

In [12]: %timeit a_np.mean()

11.6 ţs ś 161 ns per loop (mean ś std. dev. of 7 runs, 100000 loops each)


In [13]: %timeit sum([sum(x) for x in a_list])/(N*M)

78.7 ţs ś 803 ns per loop (mean ś std. dev. of 7 runs, 10000 loops each)
```

1.3 Task 3: Speed up a real-world problem!

Here your task is to speed up an algorithm for finding the solution to the *Heat Equation* in 2D using a finite-difference method, given an initial temperature distribution.

the heat equation is defined as:

$$\frac{du}{dt} = \alpha \nabla^2 u$$

Where u is the temperature. This can be approximated simply by iterating in time and approximating the spatial gradient using neighboring array elements. For time-step k of width Δt and spatial width Δx:

$$\frac{u_{ij}^{k+1} - u_{ij}^{k}}{\Delta t} = \frac{\alpha}{(\Delta x)^2}\left(u_{i,j-1}^{k} + u_{i-1,j}^{k} - 4u_{i,j}^{k} + u_{i+1,j}^{k} + u_{i,j+1}^{k}\right)$$

Below we give a naïeve way to solve this, using for-loops (which are not ideal in python). See if you can speed this up by either:

1. re-writing the code to use numpy to get rid of the spatial loops
2. using cython or numba to compile the function (may need to experiment also with some of their compile options)

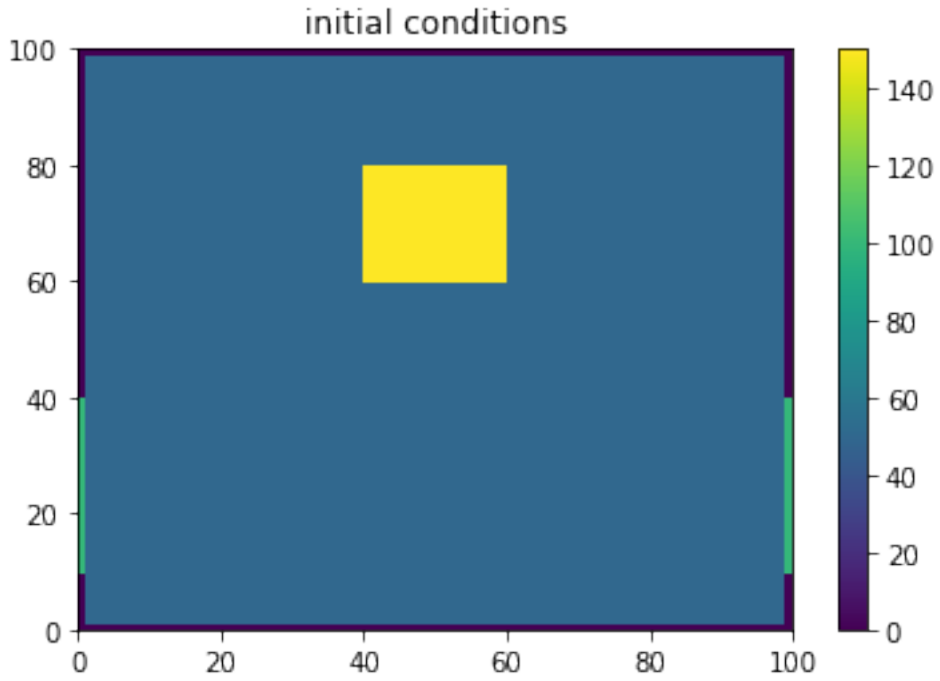
You should also try to see what the memory usage is! (hint: use the memory_profiler module). Is there a memory leak?

the setup: set up the initial condtions (defining the temperature at the boundary, as well as some hot-spots that are initially at a particular temperature)

```
In [14]: N=100; M=100  # define the spatial grid
         grid = np.zeros(shape=(N,M))
         grid[10:40,:] = 100 # a hot-spot on the border
         grid[1:-1,1:-1] = 50 # some initial temperature in the middle
```

```
grid[60:80, 40:60] = 150 # a hot-spot initially heated at the start, that will cool dou

plt.pcolormesh(grid)
plt.colorbar()
plt.title("initial conditions")
```

Out[14]: <matplotlib.text.Text at 0x118f81630>

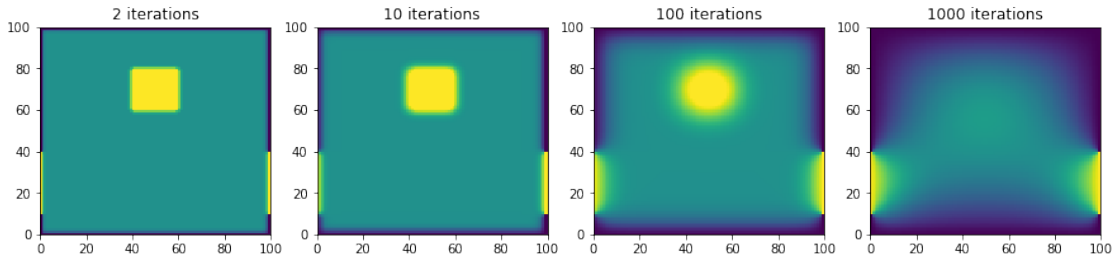


In [15]:
```
def solve_heat_equation_loops(init_cond, iterations=100, delta_x=1.0, alpha=1.0):

    delta_t = delta_x**2/(4*alpha)
    prev = np.copy(init_cond)
    cur = np.copy(init_cond)
    N,M = init_cond.shape

    for k in range(iterations):
        for i in range(1,N-1):
            for j in range(1,M-1):
                cur[i,j] = prev[i,j] + alpha*delta_t/delta_x**2 * (
                    prev[i,j-1] + prev[i-1,j] - 4*prev[i,j] + prev[i,j+1] + prev[i+1,j]
                )

        prev,cur = cur,prev  #swap pointers

    return prev
```

We'll also define a convenience function to test the results (you can use this same plotter with your own solver)

```
In [16]: def plot_heat_equation(solver, iters=(2,10,100,1000)):

             fig, axes = plt.subplots(1,len(iters), figsize=(15,3))

             for ii, iterations in enumerate(iters):
                 result = solver(init_cond=grid, iterations=iterations)
                 axes[ii].pcolormesh(result, vmin=0, vmax=100)
                 axes[ii].set_title("{} iterations".format(iterations))
```

```
In [17]: plot_heat_equation(solver=solve_heat_equation_loops)
```



Note that our code is quite slow...

1.3.1 Your turn!

Write an improved verson

- how much faster is your version on average?
- how much memory does it use on average? Is it more than the loop version?
- which line is the slowest line?

(hint: if done right, you should get a factor of about 100 speed increase)

```
In [18]: def my_heat_equation_solver(init_cond, iterations=100, delta_x=1.0, alpha=1.0):
             ## your code here
             return init_cond # replace with real return value
```

```
In [19]: #plot_heat_equation(solver=my_heat_equation_solver)
```

1.3.2 SOLUTION

there are many ways to achieve this...

```
In [20]: results = {}
         r = %timeit -o solve_heat_equation_loops(grid, iterations=50)
         results['loop'] = r
```

```
882 ms ś 15.8 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

using Numba:

```
In [21]: from numba import jit
         solve_heat_equation_numba = jit(solve_heat_equation_loops)

In [22]: r = %timeit -o solve_heat_equation_numba(grid, iterations=50)
         results['numba'] = r
```

```
362 ţs ś 10.9 ţs per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

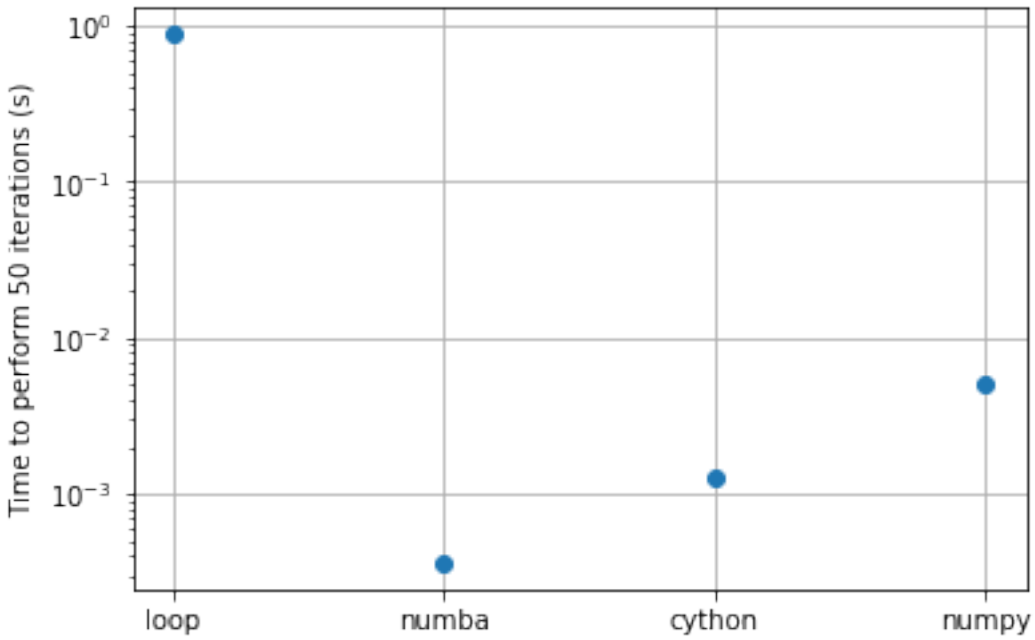
using cython:

```
In [23]: %load_ext cython

In [24]: %%cython

         cimport numpy as cnp
         import numpy as np

         def solve_heat_equation_cython(init_cond, int iterations=100, double delta_x=1.0, doubl

             cdef int i,j,k, N, M
             cdef float delta_t
             cdef cnp.ndarray[double, mode="c", ndim=2] prev, cur  # this seems to give the bigg

             delta_t = delta_x**2/(4*alpha)
             prev = np.copy(init_cond)
             cur = np.copy(init_cond)
             N,M = init_cond.shape

             for k in range(iterations):
                 for i in range(1,N-1):
                     for j in range(1,M-1):
                         cur[i,j] = prev[i,j] + alpha*delta_t/delta_x**2 * (
                             prev[i,j-1] + prev[i-1,j] - 4*prev[i,j] + prev[i,j+1] + prev[i+1,j]
                         )

                 prev,cur = cur,prev

             return prev


In [25]: r = %timeit -o solve_heat_equation_cython(grid, iterations=50)
         results['cython'] = r
```

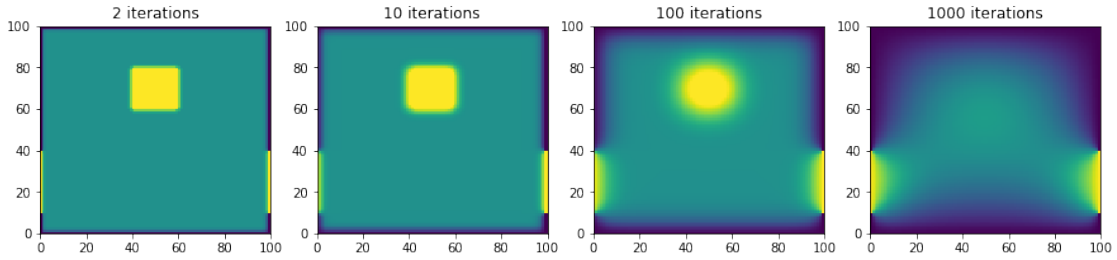
```
1.29 ms ś 15.8 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

with numpy using range-slicing `array[start:end,start:end]` we can get rid of the inner for-loops and turn that part into vector operations

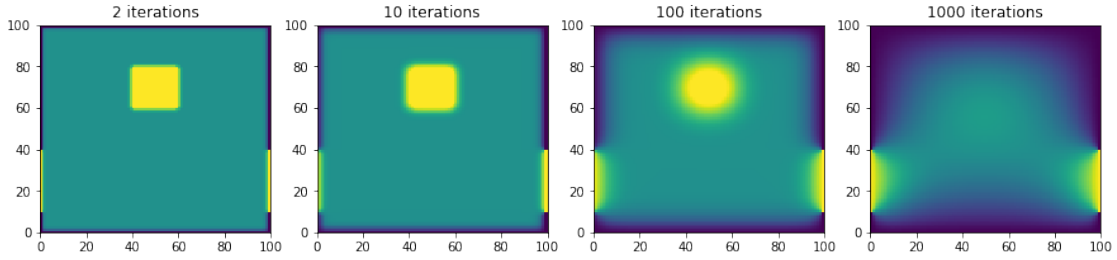
```
In [26]: def solve_heat_equation_numpy(init_cond,  iterations=100,  delta_x=1.0, alpha=1.0):

             delta_t = delta_x**2/(4*alpha)
             prev = np.copy(init_cond)
             cur = np.copy(init_cond)

             # define some slices to make it easier to type
             # just avoids too many things like prev[1:-1,1:-1])
             z = slice(1,-1) # zero
             p = slice(2,None) # plus 1
             m = slice(0,-2) # minus 1

             for k in range(iterations):
                 cur[z,z] = (
                     prev[z,z] + alpha*delta_t/delta_x**2 * (
                         prev[z,m] + prev[m,z] - 4.0*prev[z,z] + prev[z,p] + prev[p,z]
                     )
                 )
                 prev,cur = cur,prev # swap the pointers

             return prev # since we swapped, prev is the most recent

In [27]: r = %timeit -o solve_heat_equation_numpy(grid, iterations=50)
         results['numpy'] = r

5 ms ś 44.7 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)


In [28]: res = [results[x].average for x in results]
         names = [x for x in results]

         plt.semilogy()
         plt.plot(res, "o")
         plt.xticks(np.arange(len(res)), names)
         plt.ylabel("Time to perform 50 iterations (s)")
         plt.grid()
```



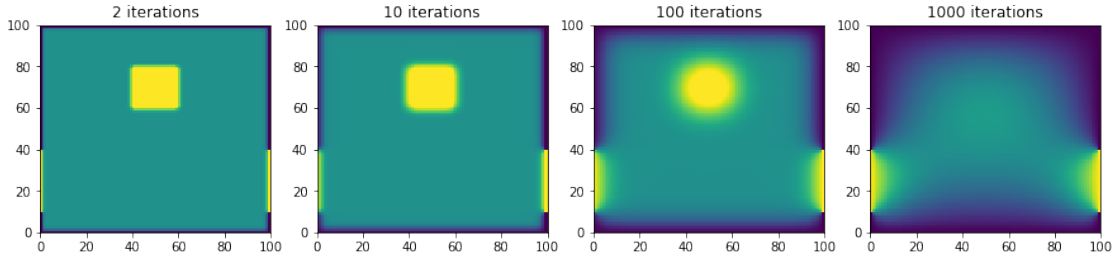
In [29]: `plot_heat_equation(solver=solve_heat_equation_numba)`



In [30]: `plot_heat_equation(solver=solve_heat_equation_cython)`



```
In [31]: plot_heat_equation(solver=solve_heat_equation_numpy)
```



Lets look at memory usage:

```
In [32]: %load_ext memory_profiler

In [33]: %memit solve_heat_equation_numpy(grid, 5000)

peak memory: 170.14 MiB, increment: 0.01 MiB

In [34]: %memit solve_heat_equation_loops(grid, 5000)

peak memory: 170.14 MiB, increment: 0.00 MiB

In [ ]:
```