taskB_DONE

October 3, 2019

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
%load_ext autoreload
%autoreload 2
```

1 Data Generation

[2]:
```
np.random.seed(10)
p, q = (np.random.rand(i, 2) for i in (4, 5))
p_big, q_big = (np.random.rand(i, 80) for i in (100, 120))

print(p, "\n\n", q)
```

```
[[0.77132064 0.02075195]
 [0.63364823 0.74880388]
 [0.49850701 0.22479665]
 [0.19806286 0.76053071]]

 [[0.16911084 0.08833981]
 [0.68535982 0.95339335]
 [0.00394827 0.51219226]
 [0.81262096 0.61252607]
 [0.72175532 0.29187607]]
```

2 Solution

[67]:
```
def naive(p, q):
    ''' fill your code in here...
    '''
    d = np.zeros((p.shape[0], q.shape[0]))
    for i in range(d.shape[0]):
        for j in range(d.shape[1]):
            d[i,j] = np.sqrt((p[i,0]-q[j,0])**2 + (p[i,1] - q[j,1])**2)
    return d
```

2.0.1 Use matching indices

Instead of iterating through indices, one can use them directly to parallelize the operations with Numpy.

[28]:
```
rows, cols = np.indices((p.shape[0], q.shape[0]))
print(rows.ravel(), end='\n\n')
print(cols.ravel())
# Notice that all possible combinations are present.
```

```
[0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3]

[0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4]
```

[29]:
```
print(p[rows.ravel()], end='\n\n')
print(q[cols.ravel()])
```

```
[[0.77132064 0.02075195]
 [0.77132064 0.02075195]
 [0.77132064 0.02075195]
 [0.77132064 0.02075195]
 [0.77132064 0.02075195]
 [0.63364823 0.74880388]
 [0.63364823 0.74880388]
 [0.63364823 0.74880388]
 [0.63364823 0.74880388]
 [0.63364823 0.74880388]
 [0.49850701 0.22479665]
 [0.49850701 0.22479665]
 [0.49850701 0.22479665]
 [0.49850701 0.22479665]
 [0.49850701 0.22479665]
 [0.19806286 0.76053071]
 [0.19806286 0.76053071]
 [0.19806286 0.76053071]
 [0.19806286 0.76053071]
 [0.19806286 0.76053071]]

[[0.16911084 0.08833981]
 [0.68535982 0.95339335]
 [0.00394827 0.51219226]
 [0.81262096 0.61252607]
 [0.72175532 0.29187607]
 [0.16911084 0.08833981]
 [0.68535982 0.95339335]
 [0.00394827 0.51219226]
 [0.81262096 0.61252607]
 [0.72175532 0.29187607]
```

```
 [0.16911084 0.08833981]
 [0.68535982 0.95339335]
 [0.00394827 0.51219226]
 [0.81262096 0.61252607]
 [0.72175532 0.29187607]
 [0.16911084 0.08833981]
 [0.68535982 0.95339335]
 [0.00394827 0.51219226]
 [0.81262096 0.61252607]
 [0.72175532 0.29187607]]
```

[72]:
```
def with_indices(p, q):
    ''' fill your code in here...
    '''
    rows, cols = np.indices((p.shape[0], q.shape[0]))
    p_expand = p[rows.ravel()]
    q_expand = q[cols.ravel()]
    d = np.sqrt(np.sum((p_expand - q_expand)**2, axis=1)).reshape(
        (p.shape[0], q.shape[0]))
    return d
```

[73]:
```
# Test results
print(naive(p,q))
print(with_indices(p,q))
```

```
[[0.60599073 0.93659449 0.91124856 0.59321356 0.27561751]
 [0.80746999 0.21102354 0.67268649 0.22495084 0.46534491]
 [0.35654215 0.75217493 0.57200052 0.49900068 0.23310825]
 [0.67281411 0.52407472 0.31520226 0.63212897 0.70277376]]
[[0.60599073 0.93659449 0.91124856 0.59321356 0.27561751]
 [0.80746999 0.21102354 0.67268649 0.22495084 0.46534491]
 [0.35654215 0.75217493 0.57200052 0.49900068 0.23310825]
 [0.67281411 0.52407472 0.31520226 0.63212897 0.70277376]]
```

2.0.2 Use a library

`scipy` is the equivalent of matlab toolboxes and have a lot to offer. Actually the pairwise computation is part of the library through the `spatial` module.

[74]:
```
from scipy.spatial.distance import cdist

def scipy_version(p, q):
    return cdist(p, q)
```

2.0.3 Numpy Magic

[75]:
```
def tensor_broadcasting(p, q):
    return np.sqrt(np.sum((p[:,np.newaxis,:]-q[np.newaxis,:,:])**2, axis=2))
```

3 Compare methods

[76]:
```
methods = [naive, with_indices, scipy_version, tensor_broadcasting]
timers = []
results = []
for f in methods:
    r = %timeit -o f(p_big, q_big)
    timers.append(r)
    results.append(f(p, q))
```

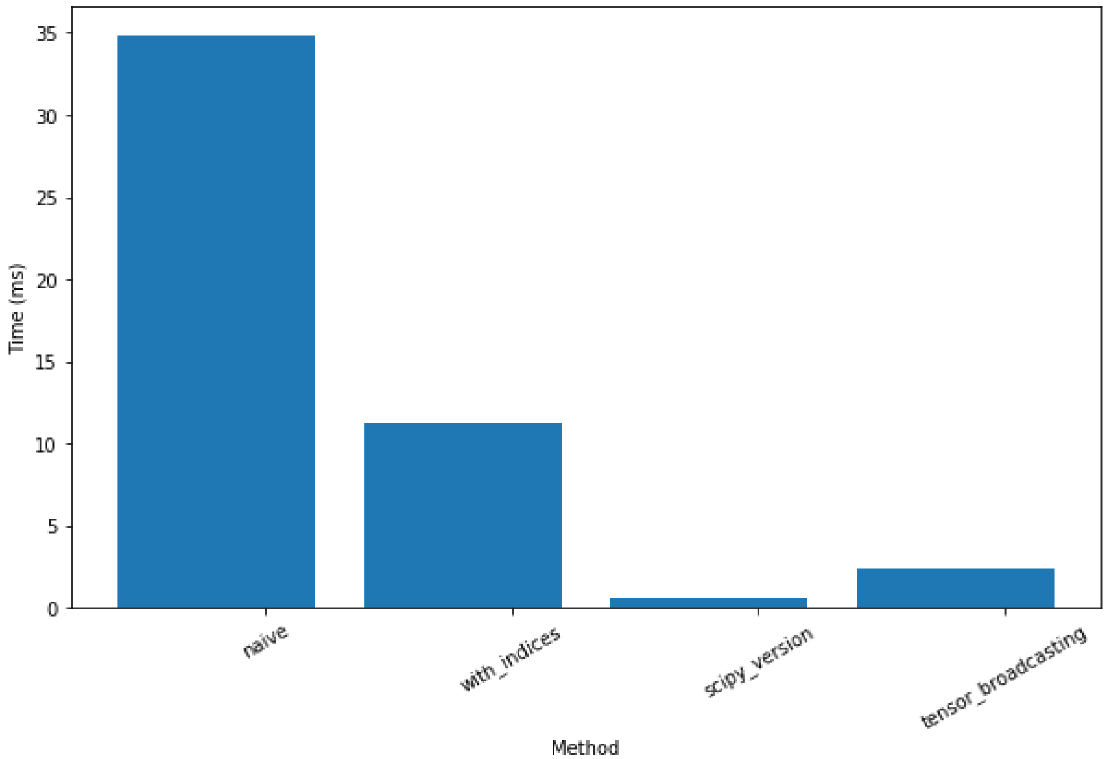
```
36.1 ms ś 1.28 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
12 ms ś 681 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
623 ţs ś 13.1 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
2.45 ms ś 57.6 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

[77]:
```
for r in results:
    print(r)
```

```
[[0.60599073 0.93659449 0.91124856 0.59321356 0.27561751]
 [0.80746999 0.21102354 0.67268649 0.22495084 0.46534491]
 [0.35654215 0.75217493 0.57200052 0.49900068 0.23310825]
 [0.67281411 0.52407472 0.31520226 0.63212897 0.70277376]]
[[0.60599073 0.93659449 0.91124856 0.59321356 0.27561751]
 [0.80746999 0.21102354 0.67268649 0.22495084 0.46534491]
 [0.35654215 0.75217493 0.57200052 0.49900068 0.23310825]
 [0.67281411 0.52407472 0.31520226 0.63212897 0.70277376]]
[[0.60599073 0.93659449 0.91124856 0.59321356 0.27561751]
 [0.80746999 0.21102354 0.67268649 0.22495084 0.46534491]
 [0.35654215 0.75217493 0.57200052 0.49900068 0.23310825]
 [0.67281411 0.52407472 0.31520226 0.63212897 0.70277376]]
[[0.60599073 0.93659449 0.91124856 0.59321356 0.27561751]
 [0.80746999 0.21102354 0.67268649 0.22495084 0.46534491]
 [0.35654215 0.75217493 0.57200052 0.49900068 0.23310825]
 [0.67281411 0.52407472 0.31520226 0.63212897 0.70277376]]
```

[78]:
```
plt.figure(figsize=(10,6))
plt.bar(np.arange(len(methods)), [r.best*1000 for r in timers], log=False)  #␣
↪Set log to True for logarithmic scale
```

```
plt.xticks(np.arange(len(methods))+0.2, [f.__name__ for f in methods],
 ↪rotation=30)
plt.xlabel('Method')
plt.ylabel('Time (ms)')
plt.show()
```



[ ]: