Improving your Python programs with NumPy and SciPy

Pawel Pomorski ppomorsk@sharcnet.ca

April 11, 2018

Outline

Talk aimed at those who know some Python but have not yet used NumPy.

Even those who know NumPy might learn some interesting new tricks.

Introduction

Python is excellent for prototyping.

However, “vanilla” Python is not good for numerically intensive work, particularly when you have explicit loops.

This seminar will show how you can use NumPy and SciPy to avoid these problems.

Your Python code will still not run as fast as compiled language code would. There are ways around that, come to Summer School to find out more.

Loop over arrays in “vanilla” Python

```
import time

N=10000000

numlist1=[]
numlist2=[]

for i in range(N):
    numlist1.append(i)
    numlist2.append(i)

numlist3=[]
start = time.time()
for i in range(N):
    numlist3.append(numlist1[i]+numlist2[i])
end = time.time()
print("time ",end-start)
```

Time (on graham cluster): 3.23 s

Same but not counting time to allocate list

```
import time

N=10000000

numlist1=[]
numlist2=[]

for i in range(N):
    numlist1.append(i)
    numlist2.append(i)

numlist3=N*[0,]

start = time.time()
for i in range(N):
    numlist3[i]=numlist1[i]+numlist2[i]
end = time.time()
print("time ",end-start)
```

Time (on graham cluster): 3.13 s

Map function

```
>>> items = [1, 2, 3, 4, 5]
>>> def sqr(x): return x ** 2
...
>>> list(map(sqr, items))
[1, 4, 9, 16, 25]
>>> def add(x,y): return x+y
...
>>> list(map(add, items, items))
[2, 4, 6, 8, 10]
```

Using map function, avoiding loop

```
from operator import add
import time

N=10000000

numlist1=[]
numlist2=[]

for i in range(N):
    numlist1.append(i)
    numlist2.append(i)

start = time.time()
numlist4=list(map(add,numlist1,numlist2))
end = time.time()
print("time ",end-start)
```

Time (on graham cluster): 1.26 s

NumPy and SciPy

NumPy and SciPy are not part of default Python, but there are easy ways to install them.

They contain routines written in highly efficient compiled languages.

On Compute Canada cluster, you can access them by loading environment modules:

module load python27-scipy-stack/2017a

or, if you are using Python 3

module load python35-scipy-stack/2017a

On your laptop it can be conveniently installed using Anaconda

https://www.anaconda.com/download

Quick overview of NumPy

```
>>> import numpy
>>> a=numpy.array([0.0,1.0,2.0,3.0,4.0,5.0])
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> a=numpy.arange(6.0)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> type(a)
<type 'numpy.ndarray'>
>>> a[3]
3.0
>>> b=a+a
>>> b
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> a*a
array([  0.,   1.,   4.,   9.,  16.,  25.])
>>> def f(x):
...     return x*x
...
>>> f(a)
array([  0.,   1.,   4.,   9.,  16.,  25.])
```

Precision in Python

Integer precision in python essentially infinite:

```
>>> print type(65535)
<type 'int'>
>>> print type(65536*65536)
<type 'int'>
>>> print type(65536*65536*65536)
<type 'int'>
>>> print type(65536*65536*65536*65536)
<type 'long'>
```

Default is 32 bit but Python will automatically switch to “long” which can take up to computer memory. Useful if operating with huge numbers, but that is an unusual case.

Using NumPy with explicit loop (the BAD way to do this)

```
import numpy
import time

N=10000000

numpylist1=numpy.arange(N,dtype="int64")
numpylist2=numpy.arange(N,dtype="int64")
numpylist3=numpy.empty(N,dtype="int64")

start = time.time()
for i in range(N):
    numpylist3[i]=numpylist1[i]+numpylist2[i]
end = time.time()
print("time ",end-start)
```

Time (on graham cluster): 5.18 s

Using NumPy correctly

```
import numpy
import time

N=10000000

numpylist1=numpy.arange(N,dtype="int64")
numpylist2=numpy.arange(N,dtype="int64")

start = time.time()
numpylist3=numpylist1+numpylist2
end = time.time()
print("time ",end-start)
```

Time (on graham cluster): 0.0406 s

Overview of results

Using explicit loop in Python lists: 3.13 s

Using map over Python lists: 1.26 s

Using explicit loop over NumPy array: 5.18 s

Using NumPy array operations: 0.0406 s

Explicit loop in NumPy is over 100 times slower than array operation!!

Python programmer should try to avoid explicit loops!

Getting rid of loops

We can see that we should definitely use NumPy arrays, and we should avoid loops.

This requires a somewhat new way of thinking if you have experience only with traditional programming languages or default Python.

You need to learn how to take problems which you would traditionally solve using a loop and figure out a way to do without it.

Basic loops which we have seen already are relatively easy to replace, but what if your loop contains if statements?

What if your loop performs a reduction?

Count number of array members satisfying condition - with loop

```
import numpy
import time

N=10000000

numpy.random.seed(1)
x = numpy.random.rand(N)

limit=0.2

num_less=0

start = time.time()
for i in range(N):
    if(x[i]<limit):
        num_less+=1

end = time.time()

print(num_less)
print("time ",end-start)
```

```
Time (on graham cluster): 2.804 s
```

Count number of array members satisfying condition - without loop

```
import numpy
import time

N=10000000

numpy.random.seed(1)
x = numpy.random.rand(N)

limit=0.2

start = time.time()
x_bool=x<limit
x_filtered=x[x_bool]

print(len(x_filtered))
end = time.time()
print("time ",end-start)
print(len(x[x<limit]))
```

Time (on graham cluster): 0.0739 s

38 times faster than loop.

Modify members of array satisfying condition - with loop

```
import numpy
import time

N=10000000

numpy.random.seed(1)
x = numpy.random.rand(N)

limit=0.2

start = time.time()
for i in range(N):
    if(x[i]<limit):
        x[i]=0

end = time.time()

print("time ",end-start)
```

Time (on graham cluster): 2.941 s

Modify members of array satisfying condition - without loop

```
import numpy
import time

N=10000000

numpy.random.seed(1)
x = numpy.random.rand(N)

limit=0.2

start = time.time()
x=-(x<limit)*x
end = time.time()
print("time ",end-start)

Time (on graham cluster): 0.0533 s

55 times faster than explicit loop.
```

Modify members of array satisfying two conditions

```
import numpy
import time

N=10000000

numpy.random.seed(1)
x = numpy.random.rand(N)

lower_limit=0.2
upper_limit=0.8

x=-(x<lower_limit)*x
x=-(x>upper_limit)*x

# cannot be done in one line
# x=-(x<lower_limit or x>upper_limit )*x # will not work

mask=numpy.logical_or(x<lower_limit,x>upper_limit)
x=-(mask)*x
```

SciPy

Some of the real power of NumPy and SciPy are that they make available a wide range of powerful algorithms which you can use in your Python code with just a few lines of code.

SciPy contains a large variety of routines relevant to many areas of mathematics, science, and engineering

See: https://docs.scipy.org/doc/scipy/reference/

Finding nearest neighbour in set

Consider set of points in two dimensions

A=((x_0, y_0),(x_1, y_1),...,(x_{N-1}, y_{N-1}))

Now pick a new point (xt,yt) and find the point in A which is closest to it.

Using explicit loop

```
import numpy
import time

N=10000000

numpy.random.seed(1)
xx = numpy.random.rand(N)
yy = numpy.random.rand(N)

joined = numpy.dstack((xx,yy))[0]  # has shape N by 2

pts = numpy.array([[0.5, 0.5], ]) # test point(s)

dmin=numpy.inf

start = time.time()
for i in range(N):
    d2=(joined[i][0]-pts[0][0])**2 + (joined[i][1]-pts[0][1])**2
    if(numpy.sqrt(d2)<dmin):
        dmin=numpy.sqrt(d2)
        imin=i
end = time.time()
print("time ",end-start)

Time (on graham cluster): 27.5 s
```

Without explicit loop

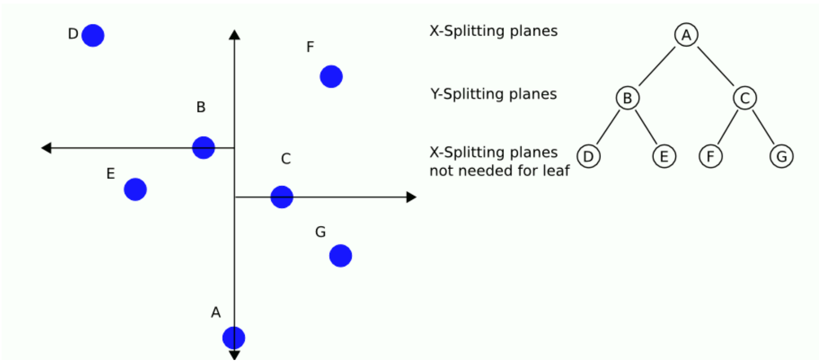
```
import numpy
import time

N=10000000

numpy.random.seed(1)
xx = numpy.random.rand(N)
yy = numpy.random.rand(N)

joined = numpy.dstack((xx,yy))[0]  # has shape N by 2

pts = numpy.array([[0.5, 0.5], ]) # test point(s)

start = time.time()

d=numpy.sqrt( (joined[:,0]-pts[0][0])**2 + (joined[:,1]-pts[0][1])**2 )

dmin = numpy.amin(d)
imin = numpy.argmin(d)
end = time.time()

print("time ",end-start)
```

Time (on graham cluster): 0.566 s

48 times faster than explicit loop.

k-d tree algorithm

Tree is a fast way to look for neighbours



k-d tree in Scipy

```
from scipy import spatial
help(spatial.cKDTree)

Help on class cKDTree in module scipy.spatial.ckdtree:

class cKDTree(__builtin__.object)
 |  cKDTree(data, int leafsize=10)
 |
 |  kd-tree for quick nearest-neighbor lookup
 |
 |  This class provides an index into a set of k-dimensional points
 |  which can be used to rapidly look up the nearest neighbors of any
 |  point.
 |
 |  The algorithm used is described in Maneewongvatana and Mount 1999.
 |  The general idea is that the kd-tree is a binary trie, each of whose
 |  nodes represents an axis-aligned hyperrectangle. Each node specifies
 |  an axis and splits the set of points based on whether their coordinate
 |  along that axis is greater than or less than a particular value.
 |
 |  During construction, the axis and splitting point are chosen by the
 |  "sliding midpoint" rule, which ensures that the cells do not all
 |  become long and thin.
 |
 |  The tree can be queried for the r closest neighbors of any given point
 |  ...
```

k-d tree in Scipy

```
...
class cKDTree(__builtin__.object)
 |  cKDTree(data, leafsize=16, compact_nodes=True, copy_data=False,
 |          balanced_tree=True)
...
 |  Parameters
 |  ----------
 |  data : array_like, shape (n,m)
 |      The n data points of dimension m to be indexed. This array is
 |      not copied unless this is necessary to produce a contiguous
 |      array of doubles, and so modifying this data will result in
 |      bogus results. The data are also copied if the kd-tree is built
 |      with copy_data=True.
...
 |  Methods defined here:
...
```

k-d tree in Scipy

```
...
 |  query(...)
 |      query(self, x, k=1, eps=0, p=2, distance_upper_bound=np.inf, n_jobs=1)
 |
 |      Query the kd-tree for nearest neighbors
 |
 |      Parameters
 |      ----------
 |      x : array_like, last dimension self.m
 |          An array of points to query.
...
 |      Returns
 |      -------
 |      d : array of floats
 |          The distances to the nearest neighbors.
 |          If x has shape tuple+(self.m,), then d has shape tuple+(len(k),).
 |          When k == 1, the last dimension of the output is squeezed.
 |          Missing neighbors are indicated with infinite distances.
 |      i : ndarray of ints
 |          The locations of the neighbors in self.data.
...
```

Using k-d tree

```
import numpy
from scipy import spatial
import time

N=10000000

numpy.random.seed(1)
xx = numpy.random.rand(N)
yy = numpy.random.rand(N)

joined = numpy.dstack((xx,yy))[0]  # has shape N by 2

start = time.time()
tree = spatial.cKDTree(joined)
end = time.time()
print("time to construct tree",end-start)

pts = numpy.array([[0.5, 0.5], ]) # test point(s)

start = time.time()
a,b=tree.query(pts)
end = time.time()
print("time to find closest point via tree ",end-start)
```

Time to construct tree: 12.9 s

Time to find closest point via tree 0.000637 s

Timing summary

Using explicit loop: 27.5 s

Using array operations: 0.566 s

Using k-d tree (tree construction included): 12.9 s

Using k-d tree (not including tree construction): 0.000637 s

These are results for just one test point.

If you had many test points (as would usually be the case in practice), the cost of constructing the tree would become relatively insignificant as the tree has to be constructed only once.

In the case of large number of test points performance using k-d tree would approach speedup of 1000 over standard array operations approach.

Conclusions

Use NumPy to access numerically efficient array routines.

Avoid explicit loops (even with NumPy)

Use SciPy to take advantage of powerful algorithms which will speed up your code.