58-COMPL-numpy

December 3, 2017

El objeto de esta hoja es mostrar las mejoras en eficiencia , enormes, que se obtienen al utilizar listas y matrices de numpy en lugar de las habituales de Sage. Esto se debe a que el código de numpy está muy optimizado.

Comenzamos generando dos listas de cien millones de números aleatorios entre cero y uno, calculando el cuadrado de cada elemento y sumando los resultados, primero en la forma normal en Sage y luego usando numpy:

```
In [1]: def cuad(x):
            return x*x
        def suma_cuadrados():
            L = [random() for muda in xsrange(10^7)]
            return sum(map(cuad,L))
```

```
In [2]: %time suma_cuadrados()
```

```
CPU times: user 5.63 s, sys: 288 ms, total: 5.92 s
Wall time: 5.86 s
```

```
Out[2]: 3335970.069522956
```

```
In [3]: import numpy as np
        np.random.seed([763829])
        %time A = np.random.rand(10000000)
        %time sum(A*A)
```

```
CPU times: user 108 ms, sys: 12 ms, total: 120 ms
Wall time: 122 ms
CPU times: user 8 ms, sys: 32 ms, total: 40 ms
Wall time: 39.9 ms
```

```
Out[3]: 3332849.8862028415
```

Tomamos ahora 10^8 números aleatorios todavía con tiempos muy bajos:

```
In [4]: import numpy as np
        np.random.seed([763829])
        %time A = np.random.rand(100000000)
        %time sum(A*A)
```

```
CPU times: user 984 ms, sys: 216 ms, total: 1.2 s
Wall time: 1.2 s
CPU times: user 104 ms, sys: 284 ms, total: 388 ms
Wall time: 386 ms
```

```
Out[4]: 33331946.820565887
```

Una vez generada la lista del tipo numpy C, podemos aplicar una función, como el logaritmo, a todos los elementos de la lista mediante $log(C)$:

```
In [5]: import numpy as np
        np.random.seed([763829])
        C = np.random.rand(100000000)
        print sum(log(C))
        print prod(log(C))
```

```
-100009700.132
0.0
```

Pasamos ahora de listas de numpy a matrices de numpy:

```
In [6]: A = np.random.rand(2,2) #Una matriz 2x2 con entradas numeros aleatorios
        print A
        B = A.dot(A) #B es su cuadrado. El producto de A por B es A.dot(B)
        print B
```

```
[[ 0.54789441  0.72054232]
 [ 0.61546796  0.41284761]]
[[ 0.743659    0.69225528]
 [ 0.59130593  0.61391386]]
```

Queremos elevar una matriz de numpy a un exponente, y para eso usamos la función recursiva general para elevar a exponentes en anillos:

```
In [7]: def potencia(A,n):
            m = A.shape[0]
            if n==0:
                return np.identity(m)
            elif n%2 == 0:
                B = potencia(A,n/2)
                return B.dot(B)
            else:
                B = potencia(A,(n-1)/2)
                return A.dot(B.dot(B))
```

```
In [8]: potencia(A,100)
```

```
Out[8]: array([[ 630922.60685662,  616939.23943193],
               [ 526972.98289444,  515293.48882875]])
```

La notación A^n no sirve para matrices de numpy, porque para ellas $A * A$ es el producto elemento a elemento, que no es igual al producto de matrices. Es por eso que definimos la función potencia usando como producro de matrices $A.dot(A)$.

```
In [9]: A^(100)
```

```
Out[9]: array([[  7.40776321e-27,   5.83377081e-15],
               [  8.32808228e-22,   3.79295049e-39]])
```

```
In [10]: C = matrix(RR,[[ 0.79881494,0.96851941],[0.90952275,0.89096117]])
```

```
In [11]: C^(100)
```

```
Out[11]: [6.77077042398315e24 7.33831108240865e24]
         [6.89130316554811e24 7.46894713973031e24]
```

```
In [12]: import numpy as np
         D = np.random.rand(500,500)
```

```
In [13]: %time E = D.dot(D)
```

```
CPU times: user 24 ms, sys: 4 ms, total: 28 ms
Wall time: 26.7 ms
```

```
In [14]: %time E_inv = np.linalg.inv(E)
```

```
CPU times: user 40 ms, sys: 0 ns, total: 40 ms
Wall time: 38.1 ms
```

Repetimos ahora los cálculos usando una matriz similar pero de Sage, con las operaciones habituales de Sage. Obsérvese la diferencia espectacular en los tiempos, debido a que los códigos que utiliza numpy están muy optimizados.

```
In [15]: F = matrix(RR,500,500,[random() for muda in srange(250000)])
```

Su cuadrado:

```
In [16]: %time G = F*F
```

```
CPU times: user 1min 4s, sys: 80 ms, total: 1min 4s
Wall time: 1min 4s
```

La inversa del cuadrado de F:

```
In [17]: %time G1=G.inverse()
```

```
CPU times: user 1min 17s, sys: 16 ms, total: 1min 17s
Wall time: 1min 17s
```

```
In [ ]:
```