NumPy

Apuntes versión 1.0

Víctor Araya Sánchez

2017

Objetivos de este apunte

Al terminar de leer, hacer los ejercicios propuestos de este apunte y trabajar en clases serás capaz de:

- Identificar herramientas útiles del módulo NumPy para el apoyo a la solución de problemas.
- Utilizar herramientas matriciales del módulo NumPy en la definición de fórmulas y resolución de problemas acotados.

1. Introducción

A diferencia de lo que hemos hecho con los apuntes anteriores, en esta ocasión no partiremos recurriendo a la RAE. Para los casos pasados era importante partir desde el lenguaje natural y sobre el construir los conceptos con los que trabajaríamos.

En el caso de este apunte, y el que sigue, trabajaremos con herramientas específicas de Python para la manipulación de información propia del ámbito STEM[1]. En particular, ahora trabajaremos con un módulo llamado NumPy, el cual añade funcionalidades matemáticas más específicas de lo que hace el módulo math.

Es importante explicitar que este documento no es un apunte sobre matemáticas, ni particularmente sobre Álgebra Lineal, sino sobre herramientas específicas para trabajar en ese ámbito.

Para encontrar información más detalladas sobre NumPy y sus potencialidades te sugerimos visitar su página oficial, y la documentación asociada:
https://docs.scipy.org/doc/numpy/reference/routines.html.

2. NumPy en general

NumPy es un módulo bastante completo que se desarrolló pensando principalmente en el trabajo con matrices y vectores. Es un módulo de código abierto y gratuito, que no forma parte de la librería estándar de Python, es decir, requiere instalación adicional.

Es regular que se utilice en conjunto con otros módulos como `mathplotlib` para realizar operaciones que comúnmente se trabajan en ambientes de programación científica como Matlab, SciLab y Octave.

[1] https://es.wikipedia.org/wiki/STEM

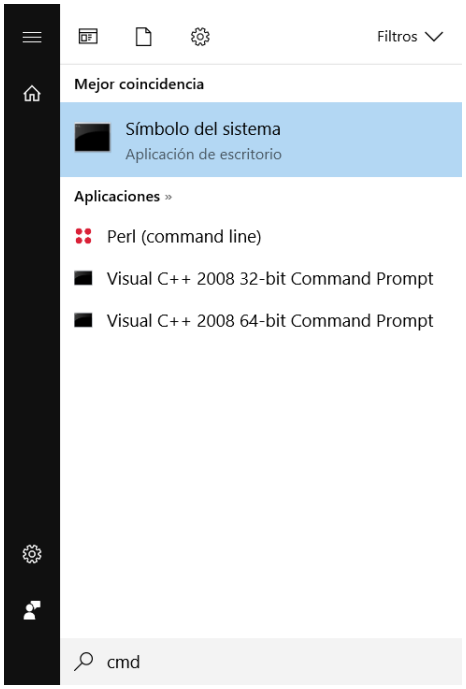
2.1. Instalación

Windows

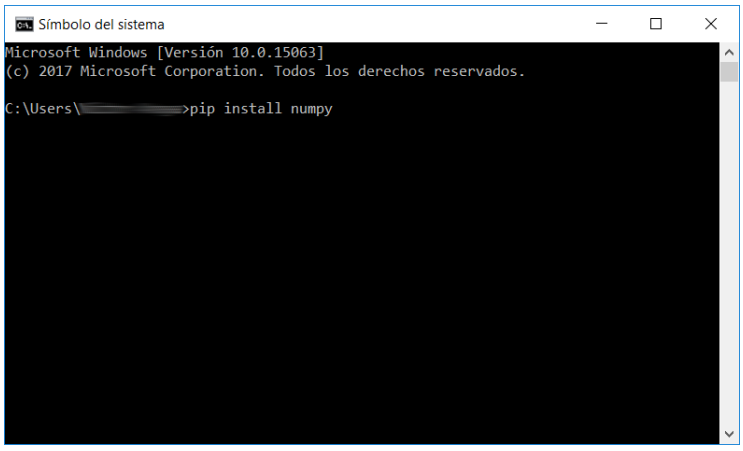
Para instalar NumPy en Windows existen varias alternativas. Aquí exploraremos el uso del instalador por defecto de Python llamado pip.

Recurrir al instalador de módulos propio de Python[2]:

- Se debe abrir la línea de comandos o Símbolos del Sistema (Regularmente abriendo el menú Windows, escribiendo cmd y luego enter):



- Iniciada la línea de comandos escribe `pip install numpy`:



[2]Es necesario agregar Python al path de las variables de entorno de Windows, salvo que ya lo hicieras al momento de instalarlo. Puedes tener información adicional en:
https://silvercorp.wordpress.com/2012/05/27/pasos-de-instalacion-de-python-en-windows/

- Luego de eso se descargará e instalará automáticamente NumPy en tu equipo.

Linux

Para instalar NumPy en Linux, en sistemas basados en Debian, sigue el procedimiento de instalación regular:

- Abre una terminal.
- Escribe `sudo apt-get install python-numpy`.
- uego de eso se descargará e instalará automáticamente NumPy en tu equipo.

Otros sistemas operativos

Para instalar NumPy en otros sistemas operativos como MacOS X revisa:
https://docs.python.org/2.7/installing/index.html

2.2. Características Generales y Herramientas

La forma tradicional de importar NumPy es utilizando la instrucción:

```
import numpy as np
```

Por esto, para los ejemplos que sigan se utilizara `np` como referencia a NumPy.

Dentro de las distintas herramientas que incorpora NumPy se destacan algunas de uso frecuente que comentaremos a continuación.

NumPy trabaja con arreglos, o `arrays`, que equivalen a agrupaciones de datos ordenados en filas y columnas. Estos arreglos pueden ser interpretados como matrices o vectores según la forma o dimensiones de estos.

La forma estándar de crear un arreglo es:

- `array()` – que permite crear vectores y matrices dependiendo del argumento que se le dé. Esta es una de las funciones más utilizadas y la veremos con mayor detalle más adelante.

Junto a esta función existen otras que trabajan en forma más específica en la creación de vectores y matrices como:

- `identity(n)` – Genera una matriz cuadrada llena de ceros, exceptuando la diagonal principal, que se llena con unos. Es decir, genera una matriz identidad de dimensión `n * n`.

```
>>> np.identity(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

- `ones(shape)` – Genera un vector o matriz llena de unos de tamaño `shape`. `shape` tiene forma de par ordenado `(a, b)`, donde `a` es el número de filas y `b` el de columnas.

```
>>> np.ones((3,2))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

- `zeros(shape)` – Genera un vector o matriz llena de unos de tamaño `shape`.

```
>>> np.zeros((1,5))
array([[ 0.,  0.,  0.,  0.,  0.]])
```

Como notarás, por defecto, NumPy supone que estamos trabajando con números de punto flotante (`floats`), por ese motivo incluye un punto al final de cada cifra. Si quisieras trabajar exclusivamente con enteros, por ejemplo, deberías usar el argumento opcional `dtype=np.int` como sigue:

```
>>> np.ones((2,5), dtype=np.int)
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

Obteniendo, por ejemplo, una matriz `2 * 5` exclusiva de unos en formato `int`.

Actividad

Existen diversas funciones que sirven para manipular arreglos. Investiga y describe para qué sirven, como se utilizan y qué argumentos requieren:

- `reshape()`
- `transpose()`
- `rot90()`

NumPy incorpora distintas funciones de tipo matemático, como trigonométricas, hiperbólicas, de redondeo, exponenciales y logarítmicas, entre otras. En esta misma línea se incorporan funciones propias del álgebra lineal y resolución de sistemas de ecuaciones. Veremos algunas de estas más adelante.

Finalmente, NumPy incorpora funciones para generación de números aleatorios y rangos. Este último punto es particularmente interesante, dado que los rangos que permite generar pueden tener pasos no enteros:

- `np.arange([start,]stop[,step])` – Genera un `array`[3] con números que parten en `start` y llegan hasta `stop`, sin incluirlo, avanzando a pasos de `step`, que en este caso pueden ser

[3] Esto es interesante dado que los `array` son un nuevo objeto iterable, tal como los `strings`, `lists`, `tuples` y otros. Por lo tanto pueden recorrerse y manipularse de igual forma.

`floats`. Tanto `start` como `step` son opcionales, no obstante, si se quiere explicitar un `step` es necesario dar el valor de `start`. El valor por defecto de `start` es cero y el de `step` es 1.

```
>>> np.arange(15,16.6,0.2)
array([ 15. ,  15.2,  15.4,  15.6,  15.8,  16. ,  16.2,  16.4])
```

Actividad

¿Qué hace la función linspace()? ¿qué argumentos requiere?

Ejercicio Propuesto

Modifica el script que encuentra los puntos de una elipse, bajo su definición de lugar geométrico, para superar el problema de las distancias no enteras.

3. Arreglos

Para crear o definir vectores y matrices basta con ocupar la función `array(contenido)`. El argumento `contenido` debe ser una lista[4] organizada de forma coherente. Por ejemplo, si quieres crear un vector de 1 fila y 3 columnas (`1 * 3`) basta con crear un `array` con un objeto iterable coherente con eso:

```
>>> np.array([1,2,3])
array([1, 2, 3])
```

Ahora, si la intención es crear una matriz de 3 filas y 1 columna (`3 * 1`) basta con dar el argumento de esa forma:

```
>>> np.array([[4],[5],[6]])
array([[4],
       [5],
       [6]])
```

Como notaras, las listas que se dan de argumento, si están anidadas dentro de otra lista, corresponden a una fila dentro de la matriz. Si la lista es única la interpretación más directa es la de un

[4] En estricto rigor pueden ser otros objetos iterables, no obstante sugerimos que trabajes con listas a efectos del curso.

vector.

Para crear una matriz de `n * m` basta con dar los argumentos de forma correcta, por ejemplo para una matriz de `3 * 5`:

```
>>> np.array([[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3]])
array([[1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3]])
```

Obteniendo una matriz de 3 filas y 5 columnas.

Par acceder a la información dentro de un `array` basta con hacer el mismo procedimiento que para las listas:

```
>>> matriz = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
>>> matriz[0]
array([1, 2, 3, 4, 5])
>>> matriz[1][0]
6
```

De igual forma, si quisieras modificar un arreglo lo puedes hacer tal como si fuese una lista:

```
>>> matriz[1][0] = 20
>>> matriz
array([[ 1,  2,  3,  4,  5],
       [20,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

Actividad

Define un arreglo cualquiera y asígnalo a una variable llamada `matriz1`. Luego, define otra variable de la siguiente forma: `matriz2 = matriz1`.

Reemplaza cualquier valor de la `matriz1`, ¿qué pasa con la `matriz2`? ¿Por qué ocurre eso?

Investiga cómo evitar que ocurra.

4. Operatoria

Hasta ahora hemos definido arreglos, no obstante, salvo el orden visual de su representación no hemos visto ventajas sobre una lista de listas. Ahora comenzaremos a operar con ellos y notaremos

cuáles son las reales ventajas de este tipo de dato.

Supongamos que tenemos dos listas que representa vectores:

```
>>> vector1 = [10,2,0]
>>> vector2 = [0,4,15]
```

Ahora necesitamos sumar dichos vectores. Si sumáramos de forma directa obtendríamos lo que sigue:

```
>>> vector1 + vector2
[10, 2, 0, 0, 4, 15]
```

Dado que la suma entre dos listas equivale a concatenarlas o unirlas una seguida de la otra, equivalente a lo que ocurre con los `string`.

Actividad

Define una función que sume vectores de igual longitud, pero independiente de ella, sin recurrir a NumPy.

Tras la actividad notarás que es un gasto de tiempo y energía importante operar con vectores definiendo nuestras propias funciones. Por ahora, imagina como sería el procedimiento de la multiplicación entre matrices.

NumPy opera con estos elementos sin problema. Para ello definiremos los vectores como `array` y los operaremos normalmente:

```
>>> vector1 = np.array(vector1)
>>> vector2 = np.array(vector2)
>>> vector1 + vector2
array([10,  6, 15])
```

Este sí es el resultado que esperábamos, y lo obtuvimos de forma rápida y directa.

Podríamos asumir que para la multiplicación es lo mismo, por lo que operaremos una matriz y un vector, en forma de `array`:

```
>>> matriz = np.array([[1,1,1,1,1],[2,2,2,2,2],[3,3,3,3,3]])
>>> vector = np.array([4,4,4,4,4])
```

Bastaría entonces con multiplicar la matriz con el vector para obtener el resultado esperado:

```
>>> matriz*vector
array([[ 4,  4,  4,  4,  4],
       [ 8,  8,  8,  8,  8],
       [12, 12, 12, 12, 12]])
```

Pero no basta con eso, la multiplicación sigue la misma lógica que la suma, multiplica elemento por elemento en la medida que coinciden en posición.

No obstante, NumPy tiene la solución a dicho problema:

```
>>> matriz.dot(vector)
array([20, 40, 60])
```

El método `.dot()` opera la multiplicación de matrices con vectores o matrices con matrices tal como se espera regularmente.

Finalmente, como comentamos arriba, NumPy incluye herramientas propias del Álgebra Lineal, para lo cual es necesario trabajar con un sub-módulo llamado `linalg`. Por ejemplo, supongamos que necesitas calcular la matriz inversa de una matriz dada:

```
>>> matriz = np.array([[2,1],[5,3]])
```

Basta con usar la función inv de linagl como sigue:

```
>>> np.linalg.inv(matriz)
array([[ 3., -1.],
       [-5.,  2.]])
```

Y para comprobar basta que la matriz por su inversa sea igual a la identidad:

```
>>> matriz.dot(np.linalg.inv(matriz))
array([[  1.00000000e+00,   2.22044605e-16],
       [  0.00000000e+00,   1.00000000e+00]])
```

Es importante notar el cálculo que realiza NumPy no es exacto, sino que una aproximación numérica. Por esto es que el segundo término de la matriz resultante `2.22044605e-16` es muy pequeño, cercano a cero, y no exactamente cero.

Ejemplo

Tomaremos un ejemplo de la literatura y lo solucionaremos con el uso de Python y NumPy:

"Una estructura de hormigón está deteriorada en un 25 %, debido un proceso de corrosión. Se envuelve la estructura en una malla de titanio a la que se aplica un microvoltaje que invierte el proceso químico de corrosión, logrando que mensualmente se recupere el 40 % de la zona deteriorada, aunque se sigue deteriorando mensualmente un 20 % de la zona sana. ¿Cuál será la situación a los 3 meses? ¿Y a los 10 meses? ¿Y al cabo de mucho tiempo?

Si consideramos tantos por uno, al cabo de un mes se ha recuperado 0.25 x 0.4 = 0.1 de la zona deteriorada, pero la zona sana se ha reducido a 0.75 x 0.8 = 0.6. En total, la zona sana al cabo de un mes sería: 0.1 + 0.6 = 0.7. Si procedemos del mismo modo con la zona deteriorada, resulta: 0.25 x 0.6 + 0.75 x 0.2 = 0.15 + 0.15 = 0.3." (Almedia, Márquez & Franco, 2001)

Es claro que hacer este procedimiento manualmente tomaría mucho tiempo, además de no permitir hacer estimaciones certeras para una cantidad de ciclos elevada. Por esto usaremos una interpretación matricial del problema, en donde consideraremos lo que sigue:

Sea d_k = tanto por uno deteriorado en el mes k.

Sea s_k = tanto por uno sano en el mes k.

La situación sugiere una ecuación en diferencias:

$$d_{k+1} = 0.6 \times d_k + 0.2 \times s_k$$

$$s_{k+1} = 0.4 \times d_k + 0.8 \times s_k$$

que matricialmente será:

$$v_{k+1} = \begin{pmatrix} 0.6 & 0.2 \\ 0.4 & 0.8 \end{pmatrix} \cdot v_k$$

donde

$$v_k = \begin{pmatrix} d_k \\ s_k \end{pmatrix}$$

La explicación de esta interpretación matemática, en detalle, está en el Almedia, Márquez & Franco (2001).

A continuación, veremos una posible solución a lo solicitado utilizando un script en Python.

```
#!/usr/bin/env python
#-*- coding: cp1252 -*-

# Deterioro y mejora

import numpy as np

def estimarProximaCondicion(oxidado, sano):
```

```
    # Se define el vector de condición actual
    actual = np.array([oxidado, sano])

    # Se define la matriz de transición, que explicita cómo
    # cambia en el tiempo la situación:
    # - La fila 0 describe el deterioro
    # - La fila 1 describe lo sano
    # - La columna 0 se estima en función del deterioro del paso
        anterior
    # - La columna 1 se estima en función de lo sano anterior
    matrizTransicion = np.array([[0.6, 0.2], [0.4, 0.8]])

    # Se calcula cómo será la situación al pasar 1 mes
    proximo = matrizTransicion.dot(actual)

    return proximo

def estimarVariosCiclos(oxidado, sano, ciclos):
    i = 1
    while i <= ciclos:
        siguiente = estimarProximaCondicion(oxidado, sano)

        # Se redefine el valor de lo oxidado y lo sano en función
            de
        # la nueva estimación
        oxidado = siguiente[0]
        sano = siguiente[1]

        i += 1
    return siguiente


valoresFinales = estimarVariosCiclos(.25, .75, 100)

print valoresFinales
```

Actividad

Agrega los comentarios propios de las buenas prácticas al script presentado.

¿Cuánto tiempo le toma al sistema equilibrarse? ¿Qué pasaría si las condiciones iniciales fuera distintas?

Construye una función que estime la cantidad de ciclos necesarios para llegar al equilibrio.

Ejercicio Propuesto

Retomemos la multiplicación de matrices, pero ahora no solo lo imaginaremos.

Construye una función en Python que multiplique apropiadamente dos matrices o una matriz y un vector, sin recurrir al método `.dot()`, independiente de la cantidad de elementos que tengan las matrices.

Sugerencia: Comienza con matrices de `2 * 2` y luego de `3 * 3`. Cuando lo consigas, busca generalizar el proceso.

5. Referencias

Almedia, P., Márquez, I., & Franco, J. (2001). Una Aplicación del Cálculo Matricial a un Problema de Ingeniería. Divulgaciones Matemáticas, 9(2), 197-205. Recuperado de: https://www.emis.de/journals/DM/vol9-2.htm