PRÁCTICA 0:
Python para cálculo matricial: Introducción a Numpy, Matplotlib y Sympy

1. Introducción

Python es un lenguaje de programación de propósito general cada vez más extendido entre la comunidad de desarrolladores. Cuenta con potentes bibliotecas como `numpy` o `sympy` que permiten realizar fácilmente operaciones de cálculo vectorial y cálculo simbólico. Además, al ser un lenguaje interpretado, es posible usarlo de manera interactiva de forma similar a programas de cálculo científico como Mathemathica o Matlab.

Existen multitud de editores que permiten escribir código Python de manera amigable. Sin embargo, para usarlo de manera interactiva es recomendable usar un editor diseñado específicamente para este fin. Spyder[1] cumple este requisito y es uno de los editor más usados por la comunidad científica. Su interfaz es muy similar a la de la Matlab y permite, desde una misma ventana, editar archivos de código Python, ejecutar comandos de manera interactiva y analizar las variables declaradas. La interfaz de Spyder incorpora una terminal de IPython, el intérprete interactivo de Python con funcionalidades extendidas como el resaltado de sintaxis o los comandos mágicos[2].

[1] Más información en www.spyder-ide.org

[2] Comandos destinados a controlar el intérprete. Comienzan con el símbolo %

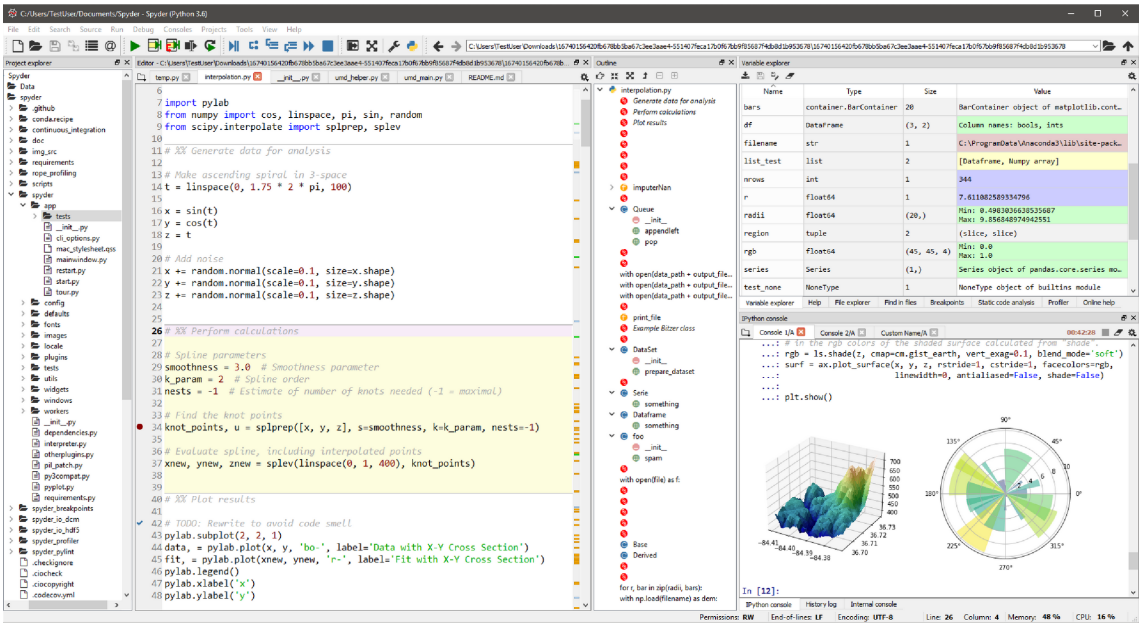

Figura 1: Ventana principal de Spyder

En las prácticas de esta asignatura vamos trabajar con Python de manera interactiva por lo que se recomienda usar Spyder como entorno de desarrollo. Además haremos uso de las bibliotecas `numpy`, `matplotlib` y `sympy` para trabajar con matrices y variables simbólicas. Tanto Spyder como las bibliotecas mencionadas forman parte de Anaconda[3], una distribución de Python enfocada a tareas de cálculo científico. Además pueden instalarse mediante el gestor de paquetes `pip`, que viene incluido en la distribución estándar de Python. Todos estos programas y bibliotecas, incluyendo la distribución Anaconda en su conjunto, son gratuitos y de código abierto.

2. Vectores y matrices con numpy

La biblioteca `numpy` nos proporciona las herramientas necesarias para trabajar con vectores y matrices cómodamente en Python, como lo haríamos en programas específicos de cálculo matricial como Matlab. Define, entre otras, la clase `array`, que nos permite crear arrays (vectores o matrices) a partir de listas de Python. Por ejemplo:

```
import numpy as np

# Define un objeto vector
vector = np.array([1, 3.1416, 40, 0, 2, 5])

# Define una matriz de 3 filas 2 columnas
matriz = np.array([[1, 2], [3, 4], [5, 6]])
```

Esa es la manera más directa para crear arrays, pero no es la única. `numpy` proporciona algunas funciones para definir arrays con valores concretos:

- `arange(ini=0, fin, inc=1)` → Devuelve un vector cuyos elementos empiezan con un valor `ini` y terminan con un valor `fin`, con incrementos de `inc`.
- `zeros(dim)` → Devuelve un array de dimensión `dim` (si es un escalar se considera un vector, si es una tupla de números entonces son las dimensiones del array), con todas sus entradas iguales a 0.
- `ones(dim):` → Similar a `zeros` pero con todas sus entradas iguales a 1.
- `eye(x, y=none)` → Si `y` vale `None`, devuelve una matriz identidad de dimensión $x \times x$. En otro caso, devuelve una matriz diagonal rectangular de dimensión $x \times y$.
- `zeros_like(x)` → Define un array de la misma dimensión que `x` con todas sus entradas iguales a 0.
- `ones_like(x)` → Define un array de la misma dimensión que `x` con todas sus entradas iguales a 1.

[3]Más información y descargas en www.anaconda.com/distribution/

- `linspace(ini, fin, num)` → Devuelve un vector de `num` elementos, cuyos elementos empiezan con un valor `ini` y van creciendo de manera uniforme hasta terminar con un valor `fin`.

- `random.rand(dim1, dim2, ...)` → Devuelve un array de dimensiones `dim1` × `dim2` × ... con números aleatorios generados por una distribución uniforme entre 0 y 1. Existen funciones similares que generan números enteros (`random.randint`) o que usan una distribución normal (`random.randn`).

Una vez definido, podemos comprobar la dimensión de un array consultando su propiedad `shape`, que contiene una tupla con tantos elementos como dimensiones tenga el array:

```
m = np.ones((4, 5))
print(m)
-> [[1. 1. 1. 1. 1.]
    [1. 1. 1. 1. 1.]
    [1. 1. 1. 1. 1.]
    [1. 1. 1. 1. 1.]]
m.shape
-> (4, 5)
```

Ejercicio 1

Defina los siguientes arrays en Python:

1. Un vector con 9 elementos, todos ellos iguales 101.
2. Una matriz de dimensión 4×6 con valores aleatorios de acuerdo a una distribución normal con media cero y varianza unitaria.
3. Un vector de 10 elementos con valores aleatorios de números enteros entre 4 y 100
4. Una matriz diagonal de dimensión 5×5 cuyos elementos de la diagonal sean (1, 2, 3, 4, 5).

La clase `array` implementa los operadores necesarios para operar con estos objetos:

```
# Define 2 arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Suma de 2 arrays elemento a elemento
# (deben tener la misma dimensión)
a + b
-> array([5, 7, 9])

# Multiplición de 2 arrays elemento a elemento
# (deben tener la misma dimensión)
a * b
```

```
-> array([4, 10, 18])

# Multiplicación de un array por un escalar
a * 10
-> array([10, 20, 30])

# Producto vectorial de 2 arrays
# (las dimensiones deben ser compatibles)
a @ b
-> 32

# En versiones de Python anteriores a la 3.5 el producto
# vectorial se obtiene mediante la función dot
np.dot(a, b)
-> 32

# Potencia
a ** 2
-> array([1, 4, 9])
```

El operador `[]` nos permite acceder a los elementos de un array. Debemos incluir tantos argumentos como dimensiones tenga el array y tener en cuenta que los arrays, al igual que otros elementos en Python, se indexan comenzando en 0. Además es posible usar el operador : junto con el operador `[]` para acceder a un rango de elementos de array. Esto nos permite definir arrays con un subconjunto de los elementos de otros arrays. Observe los siguientes ejemplos:

```
# Define una matriz 3x3
m = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accede al elemento de la primera fila y segunda columna
m[0,1]
-> 2

# Crea un vector con la fila central de m
m[1,:]
-> array([4, 5, 6])

# Crea una matriz 2x2 con elementos situados en las dos
# últimas filas y dos últimas columnas de m
m[1:3, 1:3]
-> array([[5, 6],
          [8, 9]])
```

Ejercicio 2

Tras leer el siguiente bloque de código, trate de inferir el valor de los arrays `a`, `b`, `c`, `d`, `e`, `f`, `g` y `h`. Compruebe el resultado ejecutando el código.

```
a = np.arange(100)
b = a[:20]
c = a[20:]
d = a[-1:-10:-1]
e = a[10:11]
f = a[::-1]
g = a[a % 5 == 0]
h = f[[1, 15, 60]]
```

La biblioteca `numpy` cuenta además con funciones para realizar operaciones básicas sobre arrays, como `sum` o `mean`, que realizan respectivamente la suma o la media de los elementos de un array. En caso de arrays multidimensionales, estas funciones permiten también obtener la suma o la media sólo en una dimensión. Otras funciones interesantes son `transpose`, `reshape` o `flatten` cuyo funcionamiento puede observarse en los siguientes ejemplos:

```
# Define una matriz 2x3
m = np.array([[1, 2, 3], [4, 5, 6]])

# Calcula la suma y la media de m
np.sum(m)
-> 21
np.mean(m)
-> 3.5

# Calcula la suma de todas las filas de m
np.sum(m, axis=0)
-> array([5, 7, 9])

# Calcula la traspuesta de m
m.transpose()
-> array([[1, 4],
          [2, 5],
          [3, 6]])

# Convierte m en un array de dimensión 1x6
m.reshape((1,6))
-> array([[1, 2, 3, 4, 5, 6]])

# Esto último también puede hacerse con la función flatten
m.flatten()
-> array([[1, 2, 3, 4, 5, 6]])
```

También disponemos de funciones trigonométricas, como `sin`, `cos` o `tan`, y logarítmicas, como `log` o `log10` que se aplican a todos los elementos (uno a uno) de un array, como si se tratara de operaciones vectoriales.

Ejercicio 3

Genere una matriz de dimensión 100×5 cuyos valores sean los siguientes:

- 1ª columna: Valores entre -1 y 1, equiespaciados.
- 2ª columna: El seno de los valores de la 1ª columna.
- 3ª columna: El valor de la función logística de los valores de la 1ª columna, la cual es $g(x) = \frac{1}{1+\exp(-x)}$.
- 4ª columna: 1 si el valor de $\sin(x) > 0$ y -1 en otro caso, donde x son los valores de la primera columna.
- 5ª columna: Valores aleatorios de acuerdo a una distribución gaussiana con media 1 y varianza 0.5.

A continuación, extraiga un vector de la matriz anterior con todos los valores de la función logística, cuando el valor absoluto del seno de x es menor a 0.49 (puede usar la función `where` de `numpy`. Por último, convierta este vector en una matriz de 5 columnas y las filas que sean necesarias.

3. Gráficas con Matplotlib

La biblioteca `matplotlib` nos permite realizar representaciones gráficas en Python y está preparada para trabajar con arrays de `numpy`. Las gráficas generadas pueden mostrarse en una ventana independiente o en la ventana de comandos de IPython. Por defecto, Spyder las mostrará de una forma u otra según lo hayamos especificado en la configuración del programa. Además, podemos cambiar el modo mediante el comando mágico `%matplotlib`. Así, si ejecutamos `%matplotlib inline` las gráficas se mostrarán en la ventana de comandos, mientras que `%matplotlib auto` hace que se muestren en una ventana independiente. Este último modo tiene la ventaja de ser interactivo, es decir, la gráfica se va actualizando con cada método o función que ejecutamos. El siguiente ejemplo muestra cómo podemos usar `matplotlib` para representar una curva. El resultado se muestra en la figura 2.

```
import matplotlib.pyplot as plt
import numpy as np

# Define x como un vector con valores entre -pi y pi
# e y como el seno de los valores de x
x = np.linspace(-np.pi, np.pi, 1000)
y = np.sin(x)

# Construye la gráfica
plt.plot(x, y)                  # Representa y respecto de x
plt.xlabel("Ángulo")            # Pone título al eje x
plt.ylabel("Seno")              # Pone título al eje y
plt.title("Función seno")       # Pone título a la figura
plt.axis([-np.pi,np.pi,-2,2])   # Ajusta los ejes
```

También es posible usar `matplotlib` con una sintaxis orientada a objetos, en la que en lugar de usar funciones del módulo `pyplot`, definimos objetos que representan la figura o los ejes cartesianos, y dibujamos usando métodos de esos objetos. El siguiente ejemplo da lugar al mismo resultado que el ejemplo anterior (figura 2):

```
# Define x como un vector con valores entre -pi y pi.
x = np.linspace(-np.pi, np.pi, 1000)

# Define y como el seno de x
y = np.sin(x)

# Construye la gráfica
fig, ax = plt.subplots()
ax.plot(x, y)                   # Representa y respecto de x
ax.set_xlabel("Ángulo")         # Pone título al eje x
ax.set_ylabel("Seno")           # Pone título al eje y
ax.set_title("Función seno")    # Pone título a la figura
ax.axis([-np.pi,np.pi,-2,2])    # Ajusta los ejes
```

Si queremos representar varias curvas en una misma gráfica podemos hacerlo con sucesivas llamadas al método `plot`, como se muestra a continuación:

```
# Define x como un vector con valores entre -pi y pi.
x = np.linspace(-np.pi, np.pi, 1000)

# Representa las curvas asociando una etiqueta a cada una
fig, ax = plt.subplots()
ax.plot(x, np.sin(x), label='Seno')
ax.plot(x, 1/(1 + np.exp(-x)), label=u"Logística")
ax.plot(x, (0.2 * x * x) - 0.5, label=r'$0.2 x^2 - 0.5$')

# Añade títulos
plt.title("Tres funciones trigonométricas")
plt.xlabel(r"$\theta$ (rad)")
plt.ylabel("Magnitud")

# Añade la leyenda
plt.legend(loc=0)
```

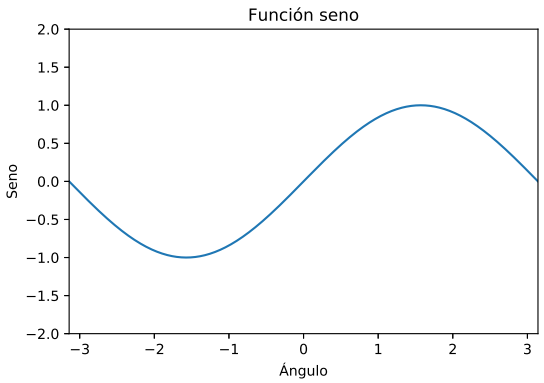

Figura 2: Representación de una curva `matplotlib`

Observe cómo, usando cadenas *Raw* y el símbolo $, podemos incluir texto como lo haríamos en el entorno matemático de Latex. El resultado se muestra en la figura 3.

También es posible mostrar diferentes curvas en una misma ventana pero en diferentes ejes cartesianos, como se hace en el siguiente ejemplo (figura 4):

```
# Calcula las proyecciones en 'x' e 'y'
x = np.linspace(0, 5, 1000)
y1 = np.exp(-0.2 * x) * np.cos(2 * np.pi * x)
y2 = np.cos(2 * np.pi * x)
y3 = np.exp(0.2 * x) * np.cos(2 * np.pi * x)

# Representa las curvas
f, (ax1, ax2, ax3) = plt.subplots(3, 1)
ax1.plot(x, y1)
ax1.set_title('Estable subamortiguado')
ax2.plot(x, y2)
ax2.set_title('Criticamente estable')
ax3.plot(x, y3)
ax3.set_title('Inestable')
```

Ejercicio 4

Genere una figura en la que se muestren 2 ejes cartesianos uno al lado del otro, es decir, formando una malla de 1 fila y 2 columnas. En el primero de ellos, represente 3 curvas: $\sin(2\pi x)$, $\sin(4\pi x)$ y $\sin(8\pi x)$, donde x es un vector de 1000 números uniformemente distribuidos entre 0 y 1. En el segundo eje cartesiano, represente la función $e^{-t}\cos(2\pi t)$ para $t \in [0, 5]$ usando una línea punteada de color rojo. Etiquete los ejes de forma apropiada y añada una legenda en ambos ejes cartesianos.

Además de la función `plot`, `matplotlib` dispone de otras funciones para realizar gráficos específicos, como gráficos de barras (`bar` y `barh`), histogramas (`hist`) o gráficos tipo tarta (`pie`).

También es posible hacer gráficos en 3 dimensiones, aunque para ello necesitamos una extensión de `matplotlib` que permite añadir ejes cartesianos en 3 dimensiones a

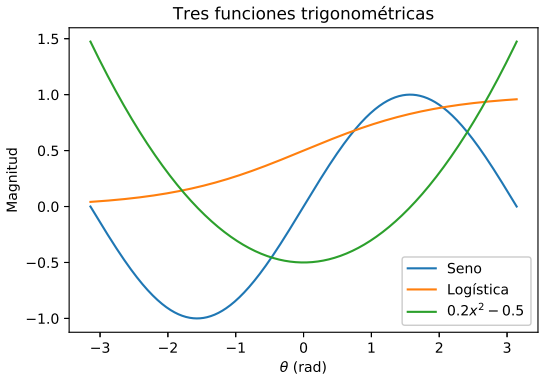

Figura 3: Representación de 2 curvas sobre los mismos ejes cartesianos.

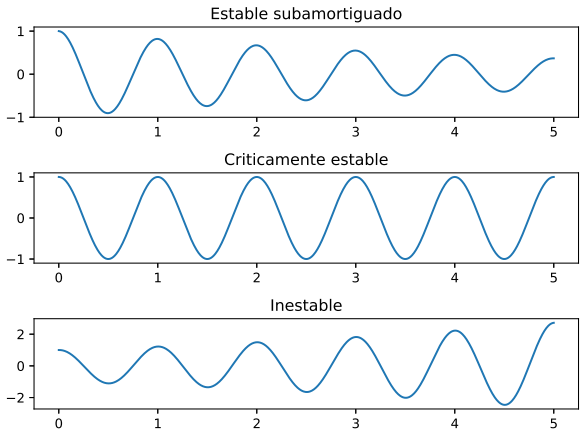

Figura 4: Representación de 3 curvas en una misma figura usando distintos ejes cartesianos.

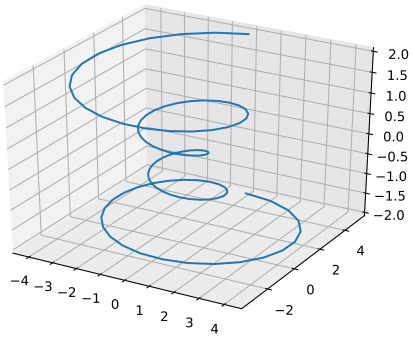

Figura 5: Representación de 1 curva en el espacio.

nuestras figuras. Observe el siguiente ejemplo donde se representa una curva en el espacio. El resultado se muestra en la figura 5.

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z)
```

4. Cálculo simbólico con sympy

La biblioteca `sympy` permite realizar operaciones de cálculo simbólico en Python. Para ello debemos declarar las variables como símbolos, usando la función `symbols`, y operar con ellas como si se tratara de variables normales.

```
import sympy as sym

alpha = sym.symbols('alpha')
r = sym.sin(alpha)**2 + sym.cos(alpha)**2

print(r)
-> sin(alpha)**2 + cos(alpha)**2
print(r.simplify())
-> 1
```

Observe el uso del método `simplify` en el ejemplo anterior. Este método resulta muy útil para simplificar una expresión que contenga variables simbólicas y a menudo se usa junto al método `nsimplify`, que permite fijar el nivel de tolerancia para considerar nulos valores muy próximos a 0.

Otro método interesante es `subs`, que nos permite sustituir una variable simbólica por otra expresión (o un valor concreto). El siguiente ejemplo muestra como usar `subs` y `nsimplify`:

```
alpha, beta = sym.symbols('alpha, beta')
r = sym.sin(alpha) * sym.cos(beta) + sym.cos(alpha) * sym.sin(beta)

print(r.simplify())
-> sin(alpha + beta)

r = r.subs(alpha, np.pi/2)
print(r.simplify())
-> 6.12323399573677e-17*sin(beta) + 1.0*cos(beta)
print(r.nsimplify(tolerance=1e-10))
-> cos(beta)
```