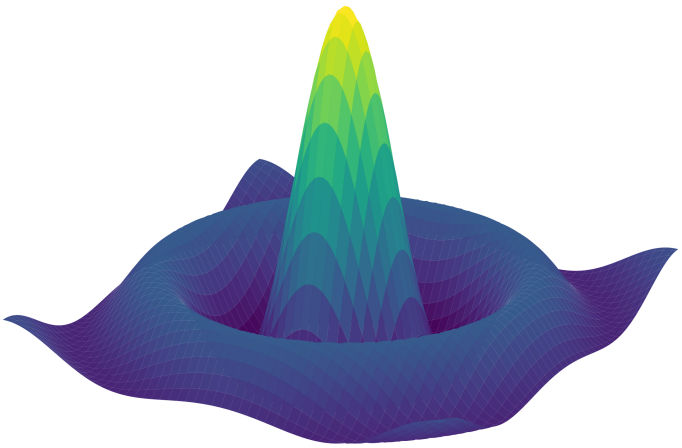
INTRODUCCIÓN A



Francisco Javier Aragón Artacho

Departamento de Matemáticas
Universidad de Alicante

Septiembre 2022

Contenido

1 Primeros pasos con Python

Durante esta guía, escrita para Python 3, supondremos siempre que se ha cargado el paquete `numpy` utilizando el alias `np`:

```
In :  import numpy as np
```

Comenzaremos viendo los operadores básicos en Python que nos permiten definir y hacer operaciones con escalares, vectores y matrices, así como con "cadenas de texto". Veremos cómo utilizar los operadores lógicos (con los cuales podremos hacer comparaciones lógicas de tipo `0/1` o *verdadero/falso*) y finalizaremos con unos cuantos comandos que pueden sernos útiles.

1.1 Operadores básicos

- $+$, $-$, $*$, $/$ son los operadores suma, resta, multiplicación y división, respectivamente. Con $(\ )$ agrupamos las operaciones.

  ```
  In :  8+5*(9-3)/2
  Out:  23.0
  ```

- El módulo (resto de la división) entre dos números puede calcularse con el operador $\%$:

  ```
  In :  7%3
  Out:  1
  ```

- El operador de potencia es $**$.

  ```
  In :  8**2
  Out:  64
  ```

- El número complejo `i` se escribe como `1j`.

  ```
  In :  1j**2
  Out:  (-1+0j)
  ```

- Para calcular la raíz cuadrada, podemos elevar el número a `0.5` o usar la función `np.sqrt`.

  ```
  In :  2**.5
  Out:  1.4142135623730951
  In :  np.sqrt(2)
  Out:  1.4142135623730951
  ```

  Si usamos `np.sqrt` para calcular la raíz cuadrada de un número real negativo, obtendremos un *Warning* y el resultado será `nan`:

  ```
  In :  np.sqrt(-1)
  __main__:1:  RuntimeWarning:  invalid value encountered in sqrt
  Out:  nan
  In :  np.sqrt(-1+0j)
  Out:  1j
  ```

- El valor absoluto se obtiene con la función `abs`.

  ```
  In :  abs(-3.5)
  Out:  3.5
  ```

- Las funciones trigonómetricas de `numpy` más usuales son `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan` (seno, coseno, tangente, arcoseno, arcocoseno, arcotangente).

  ```
  In :  np.cos(np.pi)
  Out:  -1.0
  ```

- Las funciones exponencial y logaritmo neperiano se obtienen con los comandos `exp` y `log` de `numpy`.

```
In :  np.exp(np.log(2))
Out:  2
```

1.2 Asignación de variables

- Las variables se definen de forma muy sencilla con el operador "=".

```
In :  a=2
In :  3*a
Out:  6
```

- Podemos definir varias variables a la vez:

```
In :  a,b,c=5,6,7
```

Esto sería equivalente a

```
In :  a=5;b=6;c=7
```

- Con "+ =" podemos sumarle al valor de una variable el de otra. Por ejemplo

```
In :  a=2;  a+=4;  a
Out:  6
```

sería equivalente a escribir

```
In :  a=2;  a=a+4
```

De igual manera podemos restarle a una variable otra, multiplicarla, dividirla o elevarla, con los operadores "− =", "∗ =", "/ =" y "∗∗ =".

1.3 Vectores y matrices

- Los vectores se definen con el comando `array` de `numpy`, entre corchetes y separados por comas. Podemos realizar cualquier operación con vectores, como multiplicar por un escalar:

```
In :  v=np.array([1,2,3]);2*v
Out:  array([2, 4, 6])
```

- Las matrices se crean fácilmente separando las filas con corchetes:

```
In :  A=np.array([[1,2,3],[4,5,6],[7,8,9]])
In :  A
Out:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Mediante `A[i,j]` obtenemos el elemento[1] de la fila `i` y la columna `j` de la matriz `A`.

```
In :  A[0,2]
Out:  3
```

Es posible cambiar de forma sencilla cualquier elemento de una matriz, o introducir un nuevo elemento:

[1] Recuerda que Python empieza a contar por el cero.

```
In :  A[1,1]=0; A
Out:
array([[1, 2, 3],
       [4, 0, 6],
       [7, 8, 9]])
```

- El operador ":" sirve para seleccionar todos los elementos de una fila/columna en una matriz.

  ```
  In :  A=np.array([[1,2,3],[4,5,6],[7,8,9]]); A[2,:]
  Out:  array([7, 8, 9])
  ```

  También podemos elegir elementos de una matriz (por ejemplo, una submatriz) especificando las filas y/o columnas.

  ```
  In :  A[1:3,[0,2]]
  Out:
  array([[4, 6],
         [7, 9]])
  ```

  Una opción bastante útil para seleccionar elementos de una matriz son las *máscaras*. Por ejemplo, podemos seleccionar aquellos elementos de la matriz `A` que sean mayores que `5` con el comando

  ```
  In :  A[A>5]
  Out:  array([6, 7, 8, 9])
  ```

- La propiedad `shape` nos devuelve las dimensiones de la matriz (en formato *tuple*).

  ```
  In :  np.array([[1,2,3],[4,5,6]]).shape
  Out:  (2,3)
  ```

- Es fácil trasponer una matriz usando `.T` o `.transpose()`.

  ```
  In :  np.array([[1,2,3],[4,5,6]]).T
  Out:
  array([[1, 4],
         [2, 5],
         [3, 6]])
  ```

- Podemos crear vectores de cualquier longitud cuyos elementos distan en una unidad mediante el comando `range`.

  ```
  In :  np.array(range(1,5))
  Out:  array([1, 2, 3, 4])
  ```

  También podemos crear rangos con cualquier incremento:

  ```
  In :  np.array(range(1,5,2))
  Out:  array([1, 3])
  ```

- Otra forma de crear vectores con valores equidistantes es mediante el comando `linspace` de `numpy`. Su sintaxis es "`linspace(Inicio,Fin,NúmeroDePuntos)`".

  ```
  In :  np.linspace(1,2,5)
  Out:  array([1.  , 1.25, 1.5 , 1.75, 2.  ])
  ```

- Se puede obtener el último elemento de una fila o columna de una matriz usando números negativos `end`.

  ```
  In :  A=np.array([[1,2,3],[4,5,6]]); A[:,-2]
  Out:  array([3, 6])
  In :  A[:,-2]
  Out:  array([2, 5])
  ```

1.3.1 Matrices especiales

- El comando `rand` del paquete `numpy.random` nos permite crear números aleatorios[2] distribuidos uniformemente en el intervalo $(0,1)$. Mediante `rand(m,n)` obtenemos una matriz de tamaño $\texttt{m} \times \texttt{n}$ con elementos formados por este tipo de números.

```
In :  np.random.rand(2,4)
Out:
array([[0.72567069, 0.14288355, 0.73298759, 0.76243007],
       [0.17627146, 0.19741388, 0.20293247, 0.52486388]])
```

- La matriz identidad se obtiene con el comando `eye` de `numpy`.

```
In :  np.eye(3)
Out:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

- Para obtener una matriz de ceros[3], usar el comando `zeros`.

```
In :  np.zeros([2,3])
Out:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- Con el comando `ones` obtenemos una matriz de unos.

```
In :  np.zeros([2,3])
Out:
array([[1., 1., 1.],
       [1., 1., 1.]])
```

- Con el comando `diag(v)` de `numpy`, Python nos devuelve una matriz con el vector `v` en la diagonal.

```
In :  np.diag([1,2,3])
Out:
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

  Escribiendo un segundo parámetro `diag(v,k)` obtenemos una matriz con el vector `v` posicionado en la `k`-ésima diagonal superior (o inferior, si $\texttt{k} < 0$).

```
In :  np.diag([1,2,3],-1)
Out:
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0]])
```

  Aplicado a una matriz, el comando `diag` devuelve su `k`-ésima diagonal.

```
In :  A=np.array([[1,2,3],[4,5,6],[7,8,9]]); np.diag(A)
Out:  array([1, 5, 9])
```

[2] En realidad se trata de números *pseudoaleatorios*.

[3] Es más eficiente crear una matriz de ceros y luego cambiar los valores de sus elementos que crear una matriz vacía e ir rellenando sus valores (cuando se agranda una matriz, internamente se le asigna una nueva posición en la memoria).

1.3.2 Operaciones con matrices y vectores

Es esencial tener siempre en cuenta que el paquete `numpy` ha sido diseñado para trabajar como Matlab/Octave, por lo que es necesario tener cuidado a la hora de utilizar operaciones como la multiplicación, la división o la potencia.

- Usaremos $*, /, +, -, **$ para realizar operaciones elemento a elemento.

```
In :  A=np.array([[1,2,3],[4,5,6],[7,8,9]]); 5*A
Out:
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])
In :  A*A
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
In :  A+1
array([[ 2,  3,  4],
       [ 5,  6,  7],
       [ 8,  9, 10]])
```

- Si queremos multiplicar dos matrices o una matriz por un vector, usaremos `@` o el comando `dot` de `numpy`:

```
In :  b=np.array([1,2,3]); A@b
Out:  array([14, 32, 50])
In :  A@A
Out:
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])
```

- Para calcular la potencia de una matriz, esto es, multiplicar la matriz por sí misma un número determinado de veces (por consiguiente la matriz deberá ser cuadrada), usaremos el comando `matrix_power` de `np.linalg`. Así `np.linalg.matrix_power(A,3)` = `A@A@A`.

```
In :  np.linalg.matrix_power(A,3)
Out:
array([[ 468,  576,  684],
       [1062, 1305, 1548],
       [1656, 2034, 2412]])
```

- Para calcular la inversa de una matriz usaremos `inv` del paquete `linalg`.

```
In :  A=array([[1,1,1],[0,1,0],[0,1,1]]); linalg.inv(A)
Out:
array([[ 1.,  0., -1.],
       [ 0.,  1., -0.],
       [ 0., -1.,  1.]])
```

- El determinante de una matriz se obtiene con `det`.

```
In :  A=array([[1,1,1],[0,1,0],[0,1,1]]); linalg.det(A)
Out:  1.0
```

- Para resolver un sistema `Ax = b`, usaremos `solve` de `linalg`. Python lo resuelve con un algoritmo mucho más preciso y rápido que si calculamos `linalg.inv(A)@b`.

```
In :  A=np.array([[1,2,1],[3,5,2],[3,2,1]]); b=np.array([1,2,3])
In :  linalg.solve(A,b)
Out:  array([ 1., -1., 2.])
```

- La función `tril` de `numpy` devuelve una matriz triangular inferior de una matriz dada. Introduciendo un segundo argumento `tril(A,k)` especificamos cuántas diagonales por encima o por debajo de la diagonal principal deben ser cero.

```
In :  A=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
In :  np.tril(A)
Out:
array([[ 1,  0,  0,  0],
       [ 5,  6,  0,  0],
       [ 9, 10, 11,  0],
       [13, 14, 15, 16]])
In :  np.tril(A,1)
Out:
array([[ 1,  2,  0,  0],
       [ 5,  6,  7,  0],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
In :  np.tril(A,-1)
Out:
array([[ 0,  0,  0,  0],
       [ 5,  0,  0,  0],
       [ 9, 10,  0,  0],
       [13, 14, 15,  0]])
```

- El comando `triu` de `numpy` se comporta como `tril`, pero devuelve una matriz triangular superior.

- Con `poly` de `numpy` obtenemos los coeficientes del polinomio característico de cualquier matriz.

```
In :  np.poly(np.array([[1,1,1],[1,0,1],[0,0,2]]))
Out:  array([ 1., -3., 1., 2.])
```

- La función `eig` de `linalg` devuelve los autovalores y autovectores de una matriz.

```
In :  linalg.eig(np.array([[1,2,1],[-2,1,1],[4,2,0]]))
Out:
(array([ 1.+0.j, -1.+0.j,  2.+0.j]),
 array([[ 4.08248290e-01, -1.62158452e-16,  5.07092553e-01],
        [-4.08248290e-01, -4.47213595e-01, -1.69030851e-01],
        [ 8.16496581e-01,  8.94427191e-01,  8.45154255e-01]]))
```

  Para matrices simétricas (o hermíticas), se recomienda usar el comando `eigh`.

```
In :  linalg.eigh(np.array([[1,2,4],[2,1,1],[4,1,0]]))
Out:
(array([-3.6126785 ,  0.04975857,  5.56291993]),
 array([[ 0.68273282,  0.22774378, -0.69426845],
        [-0.14057965, -0.89148761, -0.43068225],
        [-0.71701692,  0.39164091, -0.57663171]]))
```

- El comando `norm(v,k)` permite calcular la norma de cualquier vector $v = (v_1, \ldots, v_n)$. Por defecto, si no se especifica `k` o si $k = 2$, calcula la norma euclídea o L_2, $\|v\|_2 = \sqrt{v_1^2 + \ldots + v_n^2}$. Otras opciones del parámetro `k` son `1` para la norma L_1, $\|v\|_1 = |v_1| + \ldots + |v_n|$, y `np.inf` para la norma L_∞, $\|v\|_\infty = \text{máx}\{|v_1|, \ldots, |v_n|\}$.

```
In :  v=np.array([1,2,3,4])
In :  linalg.norm(v)
Out:  5.477225575051661
In :  norm(v,1)
Out:  10.0
In :  norm(v,np.inf)
Out:  4.0
```

Para calcular la norma matricial inducida por la norma euclídea ($\|A\| = \sup_{\|x\|=1} \|Ax\|$), debemos especificar `norm(A,2)`, ya que por defecto calcula la norma de Frobenius. Por ejemplo, usando la matriz del punto anterior:

```
In :  A=np.array([[1,2,4],[2,1,1],[4,1,0]])
In :  linalg.norm(A,2)
Out:  5.562919930249272
```

2 Control de flujo de ejecución

Vamos a estudiar las distintas sentencias que nos ofrece Python para controlar el *flujo de ejecución* de un algoritmo. Estas sentencias nos permitirán dirigir el comportamiento de nuestro programa ante distintas situaciones, repetir procesos un número determinado de veces o hasta que se cumpla cierta condición, etc. Para ello disponemos de la sentencia condicional `if`, el bucle finito `for`, el bucle condicionado `while`, y la sentencia `try...except`. Además, con `break` y `continue` podremos saltar iteraciones dentro de un bucle.

2.1 Sentencia condicional `if`

Se trata de una de las herramientas más básicas, que está disponible en todo lenguaje de programación. Nos permite evaluar si se cumple una condición lógica o no. Su forma más sencilla sería:

```
if condición:
    acción
```

Cuando ejecutemos este código, si `condición` es cierta, se ejecutará `acción`. Veamos un ejemplo:

```
v=5
if v==3:
    print("v vale 3")
```

El código anterior no devolvería nada, ya que `v` tiene el valor `5`. Solo ejecuta el comando `print` cuando la condición lógica `v == 3` sea cierta. Podríamos estar interesados en que sucediera algo en caso contrario, esto es, cuando la condición lógica sea falsa. Es posible hacerlo usando `else`. La sintaxis es la siguiente:

```
if condición:
    acción1
else:
    acción2
```

En el ejemplo anterior, podríamos escribir:

```
v=5
if v==3:
    print("v vale 3")
else:
    print("v es distinto de 3")
```

Ahora al ejecutarlo nos devolverá

```
v es distinto de 3
```

Existe una tercera opción, con `elif`, mediante la cual podemos hacer que si la primera condición no es cierta, evalúe otras condiciones:

```
if condición1:
    acción1
elif condición2:
    acción2
elif condición3:
    acción3
...
else:
    acción
```

Podríamos utilizar tantos `elif` como fuesen necesarios. En el ejemplo anterior:

```
v=5
if v==3:
    print("v vale 3")
elif v==5:
    print("v vale 5")
else:
    print("v es distinto de 3 y de 5")
```

2.2 Bucle `for`

Otra de las herramientas clave en programación son los bucles `for`, que nos permiten repetir una acción un número finito de veces. Su sintaxis es:

```
for variable in expresión:
    acción
```

Su funcionamiento es el siguiente: `variable` va tomando de uno en uno los valores en `expresión` y para cada uno de estos valores se ejecuta `acción`. Usualmente, expresión será un rango como `range(100)` o una lista como $[1,3,5,-2,1]$. En el siguiente ejemplo sumamos[4] los números de `1` a `1000`:

```
suma=0
for k in range(1,1001):
    suma=suma+k
print(suma)
```

Podemos *anidar* varios bucles `for`, por ejemplo para recorrer los elementos de una matriz. El siguiente código define una matriz $\texttt{n}\times\texttt{m}$ cuyo elemento $\texttt{A[j,k]}$ vale $(\texttt{j}+1)*(\texttt{k}+1)$:

```
n,m=2,3
A=np.zeros([n,m])
for j in range(n):
    for k in range(m):
        A[j,k]=(j+1)*(k+1)
print(A)
```

[4] Ésta no sería una forma eficiente de sumar los números del `1` al `n`. Una opción preferible sería escribir `sum(range(1,n+1))`, aunque la más eficiente sería usar la fórmula $\texttt{n}*(\texttt{n}+1)/2$.

```
Out:
[[1. 2. 3.]
 [2. 4. 6.]]
```

2.3 Bucle `while`

Con la sentencia `while` podemos crear otro tipo de bucles que se ejecutarán hasta que se verifique cierta condición. Aunque son muy útiles, hay que llevar cuidado a la hora de usarlos, ya que podemos crear un bucle infinito si la condición que imponemos nunca se verifica. La sintaxis es la siguiente:

```
while condición:
    acción
```

El código anterior se ejecuta si `condición` tiene el valor verdadero, en cuyo caso, realizará `acción` y volverá a verificar si se verifica `condición`. Una vez iniciado un bucle `while`, se ejecutará la acción repetidamente hasta que deje de verificarse la `condición`. El siguiente bucle escribirá los números del `0` al `10`:

```
a=0
while a<=10:
    print(a)
    a+=1
```

El siguiente bucle sería infinito, ya que la condición siempre es cierta (para detenerlo, pulsar "Ctrl+c"):

```
a=0
while (a>-2)|(a<=10):
    print(a)
    a+=1
```

2.4 Sentencias `break` y `continue`

Los comandos `break` y `continue` son bastante útiles. Con `break` *escapamos* de cualquier bucle `for` o `while` aunque el bucle no haya terminado. Por ejemplo, el siguiente código nos devuelve el valor $k = 5$, a pesar de que el bucle `for` llegaría normalmente hasta `1000`:

```
for k in range(1,1001):
    if k==5:
        break
print(k)
```

Con la sentencia `continue` saltamos al final del código de un bucle `for` o `while` sin ejecutar el resto del código, continuando con el siguiente valor del bucle. Veamos un ejemplo:

```
for k in range(1,11):
    if k==5:
        continue
    print(k)
```

Al ejecutar el código anterior, mostraría en pantalla los números del `1` al `10` excepto el `5`.

2.5 Sentencia `try...except`

Para controlar posibles errores un comando útil es `try`. Con él le pedimos a Python que intente realizar una acción. Si al hacerlo obtiene cualquier error, en vez de pararse, le decimos que realice otra acción. Veamos su sintaxis:

```
try:
    acción1
except ValorError:
    acción2
```

En el código anterior Python intentará ejecutar `acción1`. Si se produjese el error `ValorError`, continuaría normalmente ejecutando `acción2`, en vez de `acción1`. También es posible no especificar el error, como en el siguiente código, que intenta sumar al valor de `x` una unidad. Si por ejemplo `x` no estuviera definida, en vez de sumarle una unidad, le da el valor cero.

```
try:
    x=x+1
except:
    x=0
```

3 Funciones

En esta sección veremos cómo definir funciones en Python. Existen dos tipos de funciones: las funciones anónimas y las funciones definidas mediante el comando `def`. Las primeras son usadas para funciones matemáticas sencillas, mientras que las segundas se utilizan para crear algoritmos más complicados.

3.1 Funciones anónimas

Se trata de una forma rápida de escribir funciones sencillas en Python. Para ello se utiliza el comando `lambda`, especificando a continuación la/s variable/s que emplea la función y escribiendo tras ":" la función en sí. Por ejemplo, si quisiéramos definir la función $\texttt{f}(\texttt{x}) = \texttt{x}^2$ solo tendríamos que introducir

```
In :  f=lambda x:  x** 2
```

Ahora podríamos calcular el valor de la función en cualquier punto:

```
In :  f(2)
Out:  4
```

Es igual de sencillo definir funciones de más variables:

```
In :  g=lambda x,y,z:  x*y-(1+x)/(1+z**2); g(1,2,3)
Out:  1.8;
```

Podemos aplicar nuestra función a un vector (o matriz), componente a componente, usando `arrays` como variables de entrada:

```
In :  g(np.array([1,1]),np.array([2,3]),np.array([4,-2]))
Out:  array([1.88235294, 2.6 ])
```

3.2 El comando `def`

A la hora de definir funciones más complejas disponemos de la sentencia `def`. En su forma más sencilla tiene la siguiente sintaxis:

```
def nombre():
    comandos
```

donde `nombre` es el nombre con el que llamaremos a la función[5]. Veamos una función sencilla:

```
def hola():
    print("Hola Fran")
```

Ahora podemos llamar a nuestra función como si se tratara de cualquier otro comando del sistema:

```
In :  hola()
Hola Fran
```

Normalmente querremos que nuestra función acepte argumentos de entrada. En este caso escribiremos:

[5] Evitar usar tildes, espacios, la letra "ñ" y caracteres extraños en el nombre de la función.

```
def nombre(ent1,ent2,...,entN):
    comandos
```

donde `ent1`, ..., `entN` son los valores de entrada con los que el usuario llamará a la función. A continuación mostramos una modificación del ejemplo anterior.

```
def hola(nombre):
    print("Hola %s, ¿cómo estás?" %nombre)
```

Ahora nuestra función `hola` acepta un valor de entrada:

```
In :  hola("Pedro")
Out:  Hola  Pedro, ¿cómo estás?
```

Es muy sencillo definir valores por defecto de los argumentos de entrada. Solo tendremos que poner `arg = valor_defecto` cuando definamos la función. En la siguiente modificación de `hola`, si el usuario no introduce ningún argumento, el valor de la variable `nombre` será `"Fran"`.

```
def hola(nombre="Fran"):
    print("Hola %s, ¿cómo estás?" %nombre)
```

```
In :  hola()
Out:  Hola  Fran, ¿cómo estás?
In :  hola("Pedro")
Out:  Hola  Pedro, ¿cómo estás?
```

En la muchas ocasiones estaremos interesados en que nuestra función devuelva uno o varios valores de salida. Para hacerlo escribiremos:

```
def nombre(ent1,ent2,...,entN):
    comandos
    return sal1,sal2,...,salM
```

donde `sal1`, `sal2`, ..., `salM` son las variables de salida. La siguiente función solo tiene una variable de salida:

```
def f(x):
    return x**2+1/x
```

Un ejemplo de función con dos valores de salida es el comando `eig` que hemos visto en la primera sección. Veamos otro ejemplo: imaginemos que queremos escribir una función que dado un vector `v` nos diga cuál es el elemento máximo y en qué posición se encuentra[6]. En este caso, nuestra función, que llamaremos `maximo` tendrá dos valores de salida: `M` que será el valor máximo e `id` que guardará su posición.

```
def maximo(v):
    M,id=v[0],0
    for k in range(len(v)):
        if M<v[k]:
            M,id=v[k],k
    return M,id
```

```
In :  maximo([1,4,2,7,-3,8,5,1])
Out:  (8, 5)
```

[6]La función `max`, que viene ya predefinida en Python, devuelve el valor máximo. Para encontrar la posición donde se encuentra el máximo podemos usar la función `argmax` de `numpy`. La combinación de ambas sería mucho más eficiente que escribir el código con un bucle `for`.

4 Gráficas e imágenes

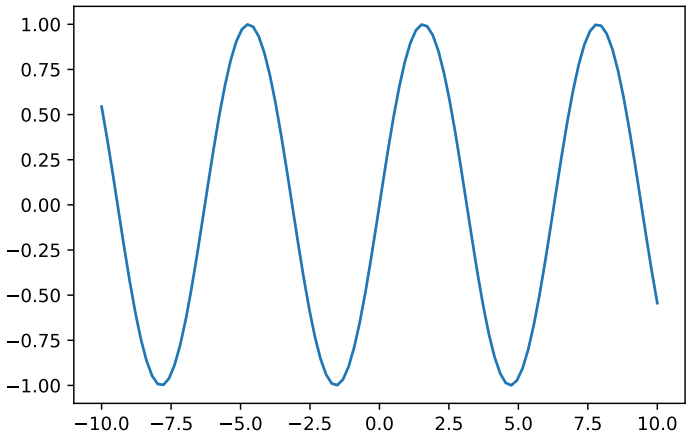
Comenzamos siempre importando la librería `matplotlib.pyplot` mediante el alias `plt`:

```
In :  import matplotlib.pyplot as plt
```

4.1 En el plano: 2D

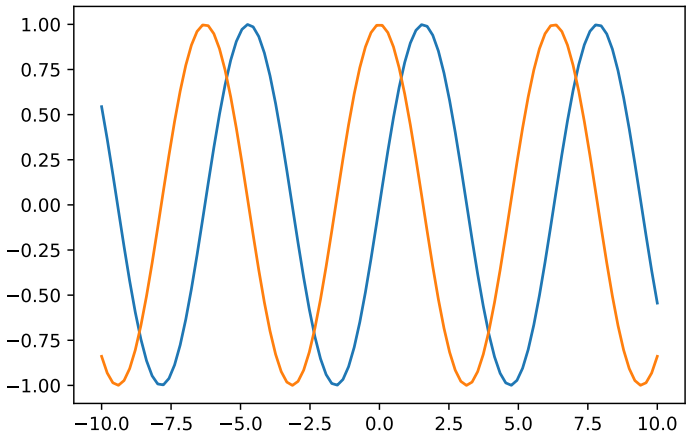
- El comando `plot(x,y)`, donde `x` e `y` son vectores, dibuja una línea en el plano que une consecutivamente los puntos $\big(\texttt{x(1)},\texttt{y(1)}\big),\ldots,\big(\texttt{x(end)},\texttt{y(end)}\big)$. Este comando será muy útil a la hora de dibujar gráficas de funciones: solo tendremos que elegir un número suficiente de puntos para que el gráfico sea una buena aproximación de la gráfica real de la función.

  ```
  In :  x=np.linspace(-10,10,100)
  In :  plt.plot(x,np.sin(x))
  ```



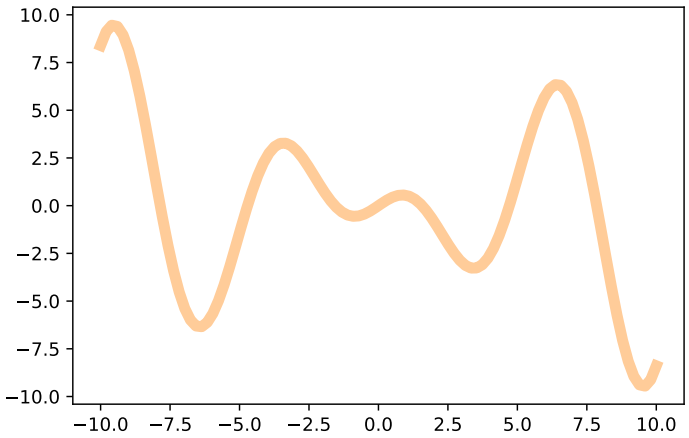
  Podemos dibujar varias funciones a la vez:

  ```
  In :  plt.plot(x,np.sin(x),x,np.cos(x))
  ```



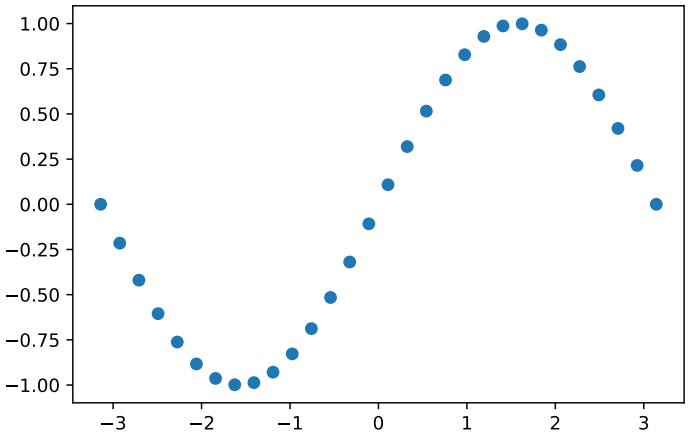
  Existen diversas opciones que pueden ser especificadas mediante el comando plot. Por ejemplo, mediante la opción `linewidth` controlamos el grosor de la línea. Con la opción `color` podemos cambiar el color, introduciendo a continuación una lista con 3 componentes RGB (rojo, verde y azul) con valores entre 0 y 1. También existen algunos colores predefinidos como $'$`red`$'$, $'$`green`$'$, $'$`blue`$'$ o $'$`yellow`$'$.

  ```
  In :  plt.plot(x,x*np.cos(x),color=[1,0.8,0.6],linewidth=6)
  ```



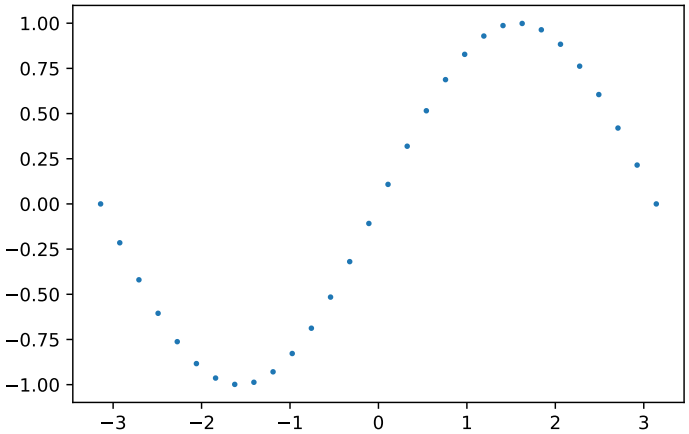
Por defecto el comando `plot` dibuja puntos y los une con una recta (puede especificarse con la opción $'-'$). Es posible modificar este comportamiento con las opciones $'+'$, $'*'$, $'o'$, $'x'$ y $'**'$. Usando una de estas opciones, `plot` dibujará uno de estos símbolos:

```
In :  x=np.linspace(-np.pi,np.pi,30); plt.plot(x,np.sin(x),'o')
```



Se puede cambiar el tamaño de los marcadores anteriores con la opción `markersize`, especificando después un número.

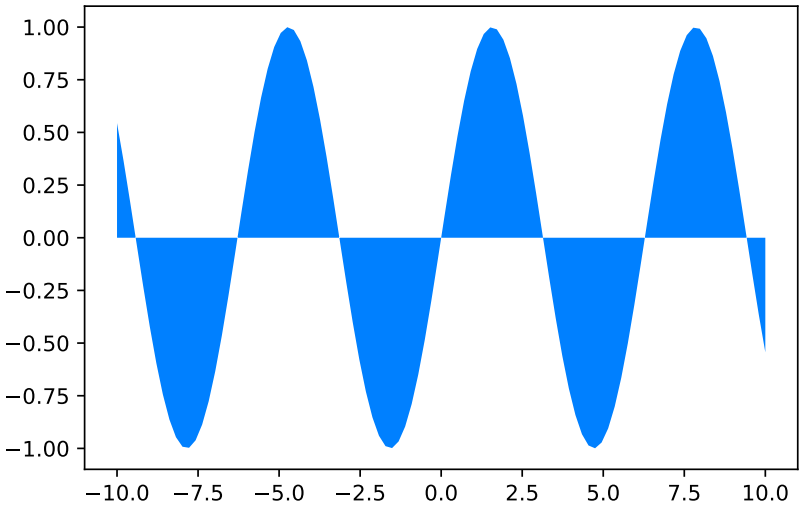
```
In :  plt.plot(x,np.sin(x),'o',markersize=2)
```



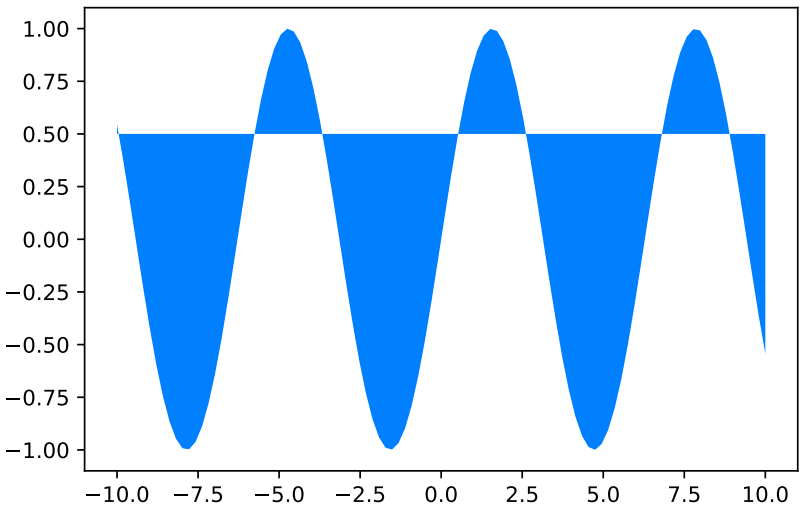
- El comando `fill_between(x,y)`, donde x e y son vectores, se comporta como `plot` con la diferencia de que `fill_between` pinta la gráfica por debajo/arriba hasta el origen. Podemos cambiar

el color con la opción `facecolor`.

```
In :  x=np.linspace(-10,10,100)
In :  plt.fill_between(x,np.sin(x),facecolor=[0,0.5,1])
```



Es posible cambiar la posición hasta la cuál rellena el gráfico mediante un parámetro adicional: `plt.fill_between(x,np.sin(x),0.5,facecolor` $= [0, 0.5, 1]$



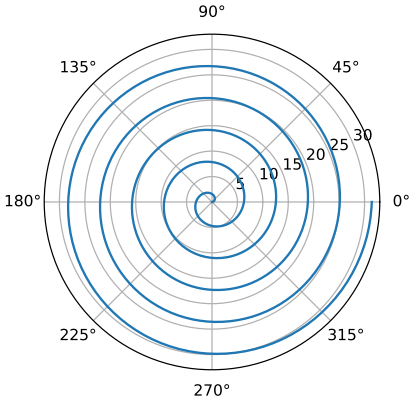
- Usando el comando `plt.axis(False)` conseguimos ocultar los ejes de coordenadas. Mediante `plt.axis('square')` hacemos que ambos ejes midan lo mismo (y por tanto obtenemos un dibujo cuadrado). Usando `plt.axis('equal')` logramos que las longitudes en ambos ejes sean iguales cambiando los límites de los ejes, mientras que con `plt.axis('scaled')` conseguimos lo mismo cambiando las dimensiones de la figura. También es posible especificar los ejes mediante `plt.axis`$([x_{min}, x_{max}, y_{min}, y_{max}])$.

- Las *figuras* son las ventanas donde nos aparecen los gráficos. Es posible crear nuevas figuras mediante el comando `plt.figure`.

- Introduciendo `plt.clf()` borramos la figura activa. También podemos eliminarla escribiendo `plt.close`. Usando `plt.close('all')` cerraremos todas las figuras abiertas.

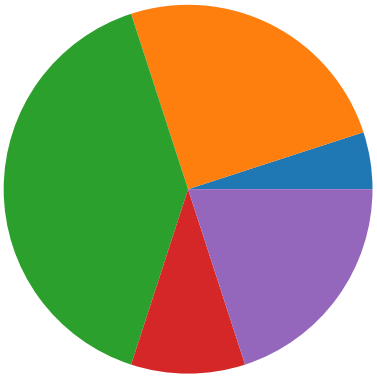
- Para crear gráficos en coordenadas polares, usaremos el comando `polar(theta,rho)`.

```
In :  plt.polar(np.linspace(0,10*np.pi,1000),np.linspace(0,10*np.pi,1000))
```



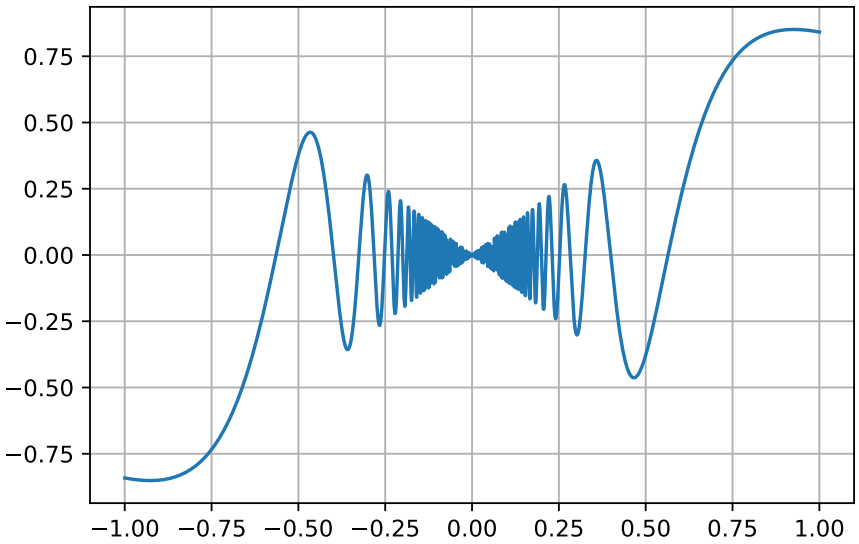
- Es muy fácil crear un gráfico de sectores. Solo hay que introducir un vector con los porcentajes usando el comando `pie`.

```
In :  plt.pie([5,25,40,10,20])
```



- El comando `plt.grid(True)` permite mostrar una rejilla de coordenadas. Para desactivarla escribiremos `grid off`.

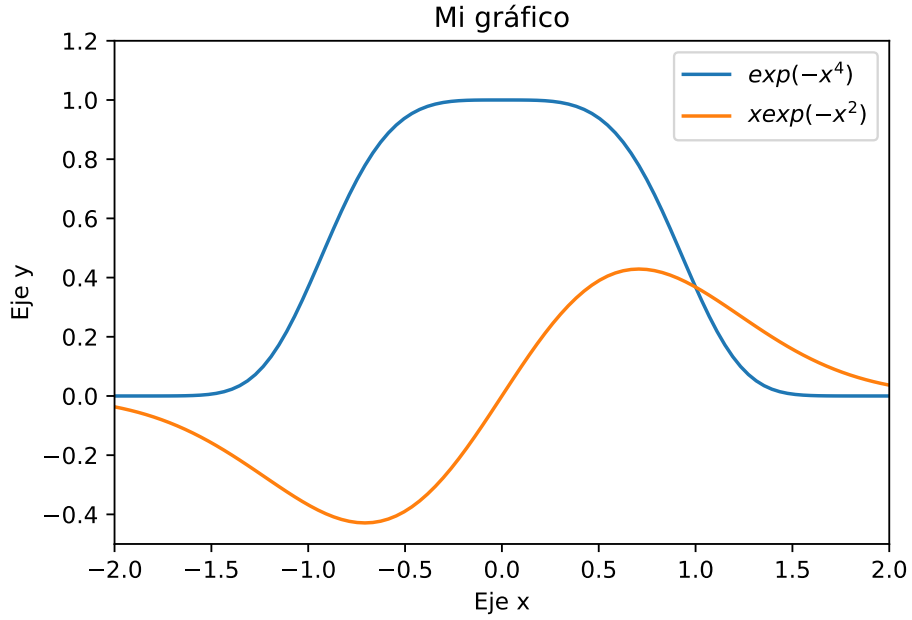
```
In :  x=np.linspace(-1,1,2000); plt.plot(x,x*np.sin(1/x**2)); plt.grid(True)
```



- Al introducir `plt.box(False)` eliminamos la caja que rodea por defecto a las gráficas. Para mostrarla usaremos `plt.box(True)`.

- Para ponerle un título a nuestro gráfico usaremos el comando `plt.title`. Para mostrar un nombre en los ejes disponemos de los comandos `plt.xlabel` e `plt.ylabel`. Si queremos cambiar o introducir una leyenda al gráfico, usaremos `plt.legend`.

```
In :  x=np.linspace(-2,2,100)
In :  plt.plot(x,np.exp(-x**4),x,x*np.exp(-x**2),label="x*exp(-x**2)");
plt.title("Mi gráfico");plt.xlabel("Eje x");plt.ylabel("Eje y");
plt.legend(["$exp(-x^4)$","$x exp(-x^2)$"]);plt.axis([-2,2,-.5,1.2])
```



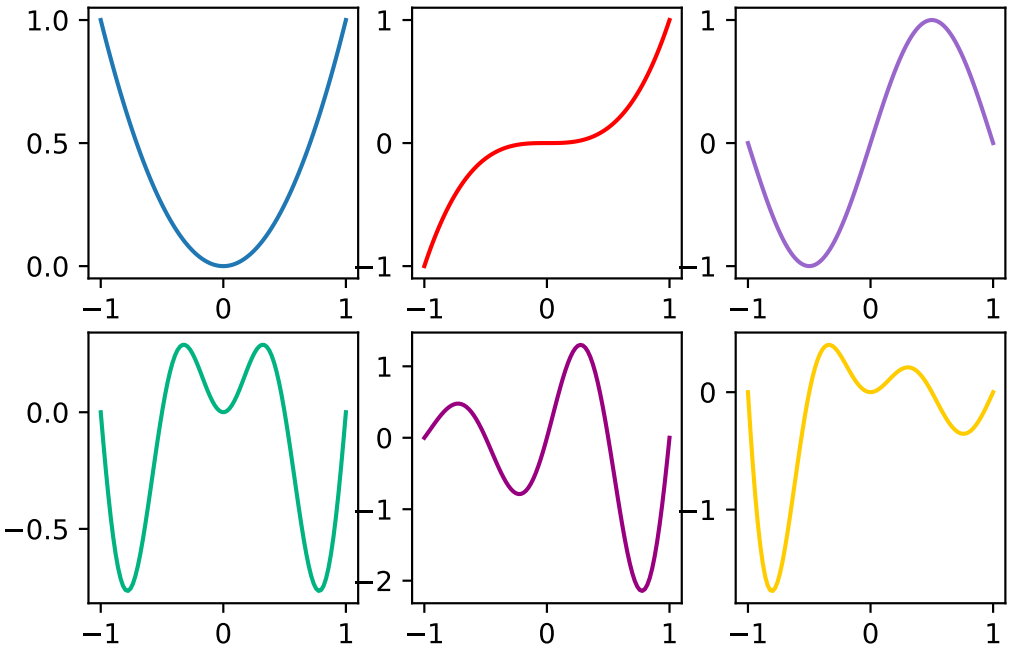
- Para introducir cualquier cadena de texto en un gráfico usaremos `plt.text(pos_x,pos_y,"texto")`, donde `pos_x,pos_y` son las coordenadas del "`texto`" en el gráfico. Se puede cambiar el tamaño con la opción `fontsize`. Existen más parámetros opcionales disponibles.

```
In :  plt.text(1.1,2,"punto de inflexión",fontsize=14)
```

- El comando `plt.arrow` permite dibujar flechas. Si escribimos
```
In :  plt.arrow(x, y, dx, dy)
```
dibujará una flecha desde (x,y) hasta $(x+dx,y+dy)$. El tamaño de la cabeza de la flecha se puede modificar con el parámetro `head_width`. Si queremos que la punta de la flecha acabe en $(x+dx,y+dy)$, debemos establecer el parámetro `length_includes_head` = `True`. Para otras muchas opciones, consúltese la ayuda.

- Con `plt.subplot(fil,col,id)` podemos crear varios gráficos dentro de una misma figura, donde `fil` es el número de filas, `col` el número de columnas, e `id` el índice contando de izquierda a derecha por filas, empezando por arriba.

```
In :  x=np.linspace(-1,1,100)
plt.subplot(2,3,1); plt.plot(x,x**2); plt.subplot(2,3,2);
plt.plot(x,x**3,color='r'); plt.subplot(2,3,3);
plt.plot(x,np.sin(np.pi*x),color=[0.6,0.4,.8]); plt.subplot(2,3,4);
plt.plot(x,x*np.sin(2*np.pi*x),color=[0,.7,.5]); plt.subplot(2,3,5);
plt.plot(x,np.exp(x)*np.sin(2*np.pi*x),color=[0.6,0,.5]);plt.subplot(2,3,6);
plt.plot(x,x*np.exp(-x)*np.sin(2*np.pi*x),color=[1,0.8,0])
```



4.2 En el espacio: 3D

Para dibujar en 3D utilizaremos `mplot3d` del paquete `mpl_toolkits`. Para cargarlo, escribiremos:

```
In :  from mpl_toolkits import mplot3d
```

Para realizar gráficos en dos dimensiones con el comando `plot`, hemos creado un vector de puntos `x` sobre los cuales calculábamos el vector $y = f(x)$. El comando `plot` se encargaba de unir los pares $(x(i), y(i))$ por rectas. Para crear un gráfico en tres dimensiones haremos algo similar, pero el proceso es ligeramente más complicado. Ahora crearemos dos vectores de puntos `x` e `y` y calcularemos los puntos $z = f(x,y)$ para todas las combinaciones posibles de puntos de `x` y de `y`. Necesitaremos pues crear una rejilla con todas las combinaciones posibles. De ello se encarga el comando `meshgrid`:

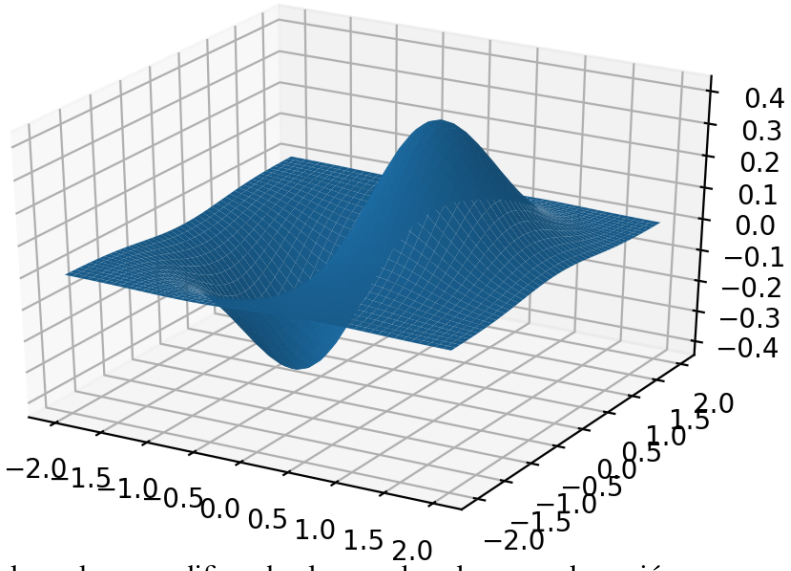
```
In :  [X,Y] = np.meshgrid(range(-3,3), range(3,10,2))
                          vector x      vector y
In :  X
Out:
array([[-3, -2, -1,  0,  1,  2],
       [-3, -2, -1,  0,  1,  2],
       [-3, -2, -1,  0,  1,  2],
       [-3, -2, -1,  0,  1,  2]])
In :  Y
Out:
array([[3, 3, 3, 3, 3, 3],
       [5, 5, 5, 5, 5, 5],
       [7, 7, 7, 7, 7, 7],
       [9, 9, 9, 9, 9, 9]])
```

La función `meshgrid` crea dos matrices, que en este ejemplo hemos llamado `X` e `Y`. En la primera copia el vector x' por filas tantas veces como elementos tiene el vector `y`. En la segunda, el vector `y` es copiado por columnas tantas veces como filas tiene el vector `x`. De este modo tendremos dos matrices `X` e `Y` cuya dimensión será $\texttt{len(y)} \times \texttt{len(x)}$, y los elementos $(X(i,j), Y(i,j))$ formarán todas las combinaciones posibles de pares de elementos del vector `x` y del vector `y`. Solo quedará pues calcular la matriz $z = f(x,y)$ y así tendremos una nube de puntos en tres dimensiones:

```
In :  [X,Y]=np.meshgrid(np.linspace(-2,2),np.linspace(-2,2))
In :  Z=X*np.exp(-X**2-Y**2)
```

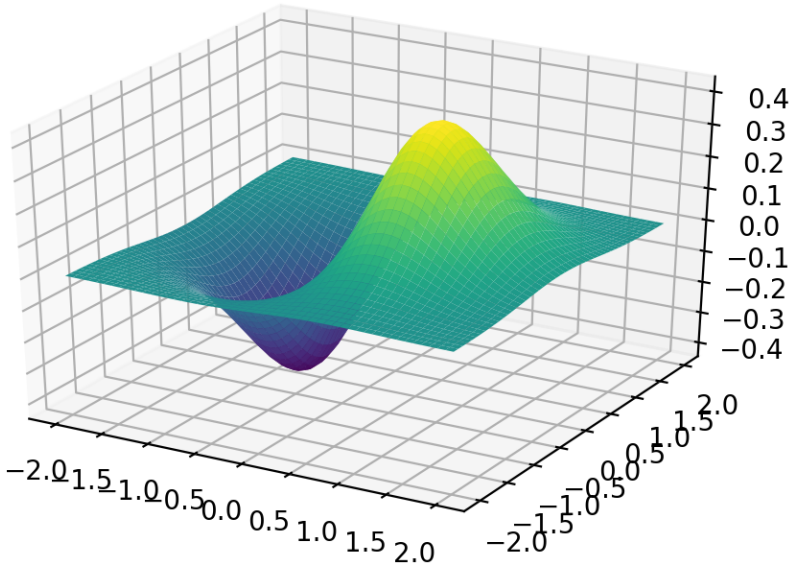
Para crear unos ejes 3D, usamos el comando `plt.axes` con la opción `projection` $=$ `'3d'`. Con el método `plot_surface` de los ejes que hemos creado podremos dibujar nuestra función:

```
In :  fig=plt.figure();ejes = plt.axes(projection='3d');ejes.plot_surface(X,Y,Z)
```



Podemos cambiar los colores modificando el mapa de colores con la opción `cmap`:

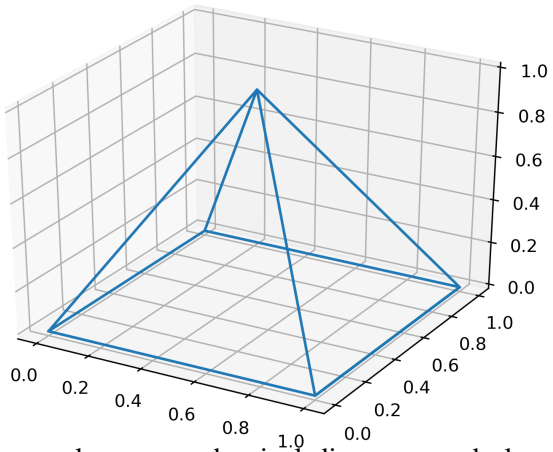
```
In :  ejes.plot_surface(X,Y,Z,cmap='viridis')
```



Podemos ver los nombres de los mapas de colores disponibles con el comando `plt.colormaps()`.

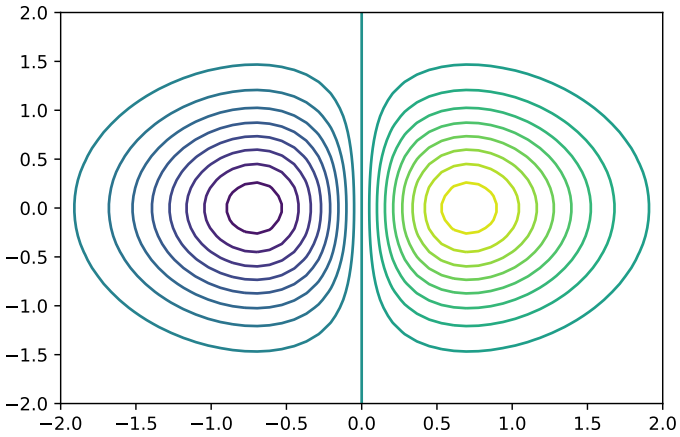
También podemos crear rectas tridimensionales con el método `plot3D` de los ejes que hemos creado. Su funcionamiento es similar al comando `plot`, pero debemos introducir ahora las coordenadas del eje `x`, del `y` y del `z`. Por ejemplo, los siguientes comandos dibujan una pirámide:

```
In :  x=[0,1,1,0,0,0.5,1,1,0.5,0]
In :  y=[0,0,1,1,0,0.5,0,1,0.5,1]
In :  z=[0,0,0,0,0,1,0,0,1,0]
In :  fig=plt.figure();ejes = plt.axes(projection='3d');plt.plot3D(x,y,z)
```



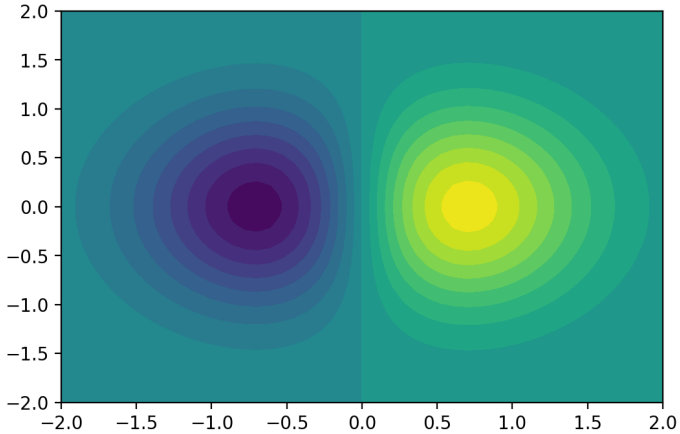
Para crear un gráfico solo con las curvas de nivel disponemos de los comandos `plt.contour`, `contourf` y `contour3`. El primero crea una gráfica en dos dimensiones con las curvas de nivel a partir de las matrices (X,Y,Z). Podemos especificar tanto el número de curvas de nivel como las curvas de nivel que queramos dibujar en el cuarto parámetro de la función:

```
In :  plt.contour(X,Y,Z,20)
```



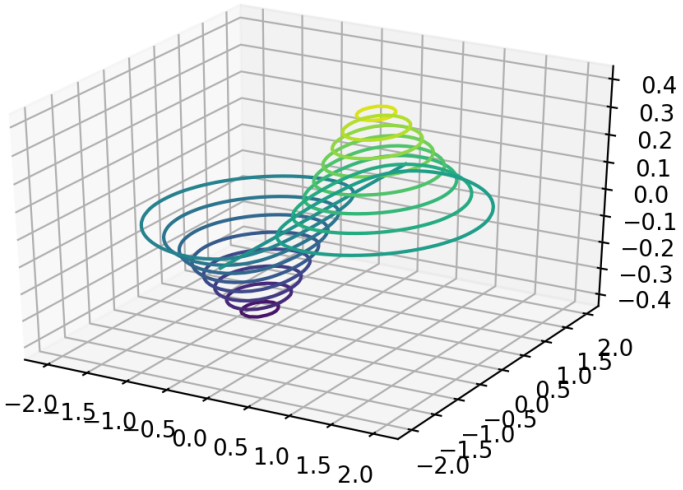
El comando `contourf` es similar, pero pinta las zonas entre las curvas de nivel.

```
In :  plt.contourf(X,Y,Z)
```



Con el método `contour3D` crearemos un gráfico en 3D con las curvas de nivel, donde cada curva estará a la altura `z` que le correspondería en la gráfica. Para usarlo debemos crear unos ejes en 3D, como con el comando `plot_surface`:

```
In :  ejes = plt.axes(projection='3d'); ejes.contour3D(X,Y,Z,20)
```



4.3 Guardar gráficas

Es posible guardar la figura actual (el gráfico) en un archivo mediante el comando `plt.savefig`. Admite multitud de formatos en los que almacenar la imagen; entre ellos los más usuales son pdf, png y jpg. El comando detecta el formato mediante la extensión del nombre del fichero. Podemos cambiar la resolución mediante el parámetro `dpi` (*dots per inch*, puntos por pulgada). Finalmente, para ajustar al máximo la imagen eliminando el espacio en blanco alrededor del gráfico (el marco), especificaremos `bbox_inches='tight'`. Por ejemplo, el siguiente comando guardaría la gráfica actual en el fichero 'Dibujo.png' con una resolución de 300 dpi y con un marco ajustado:

```
In :  plt.savefig('Dibujo.png',bbox_inches='tight',dpi=300)
```

4.4 Imágenes

Una imagen digitalizada es, básicamente, un conjunto de tres matrices de datos correspondientes a la intensidad de cada componente RGB (rojo, verde y azul; del inglés *Red, Green, Blue*) de cada píxel de la imagen[7]. En Python es posible utilizar 256 intensidades distintas de cada uno de estos colores, lo que proporciona un total de casi 16,8 millones de colores distintos. Por tanto cualquier imagen quedará determinada especificando qué cantidad de cada uno de estos colores se debe utilizar en cada uno de los píxeles que la forman. Necesitaríamos 8 bits para almacenar la información de la intensidad de cada color RGB en cada píxel, lo cual requeriría unos archivos de gran tamaño para albergar toda esta información. Por este motivo las imágenes se encuentran normalmente comprimidas, siendo el formato más usual en la actualidad el *jpg*.

Con Python es muy fácil obtener la información de una imagen en una matriz mediante el comando `plt.imread`, el cual es capaz de leer multitud de formatos. Su sintaxis es
`I = plt.imread("nombre_archivo")` donde `I` es el array (*hipermatriz*[8]) en la que queramos guardar los datos de la imagen. Por ejemplo:

```
In :  A=imread("Irlanda.jpg")
```

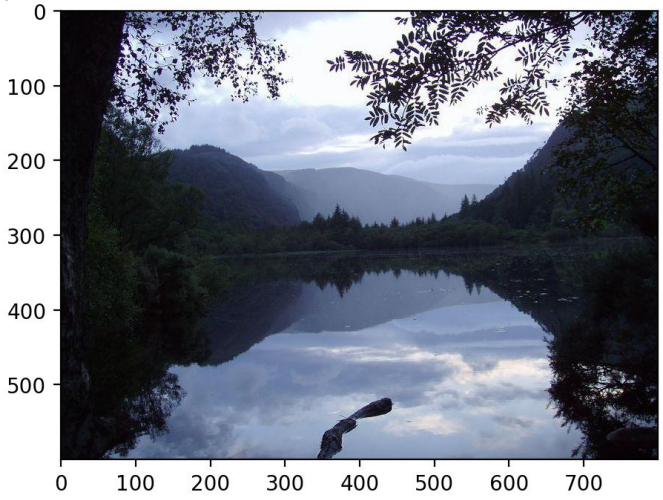
La imagen "Irlanda.jpg" tiene un tamaño 600×800, por lo que el array `A` tendrá un tamaño $600 \times 800 \times 3$ (estará formada por 3 matrices $A[:,:,0]$, $A[:,:,1]$ y $A[:,:,2]$ de tamaño 600×800, una para cada color).

[7] Cualquier color se puede obtener como la composición de los colores primarios rojo, verde y azul.

[8] Las *hipermatrices* son una generalización del concepto de matriz. Una matriz tiene dos dimensiones: filas (dimensión 1) y columnas (dimensión 2). Una hipermatriz de 3 dimensiones de tamaño $m \times n \times k$ podemos imaginarla en el espacio como un cubo formado for k matrices de tamaño $m \times n$. Con el comando `array` podemos trabajar con hipermatrices de cualquier dimensión, no solo 3.

Para mostrar imágenes en pantalla podemos usar el comando: `plt.imshow`.

```
In :  plt.imshow(A)
```



Si quisiéramos guardar la imagen anterior (por ejemplo, tras modificarla o editarla) en un archivo de imagen, usaremos el comando `plt.imsave`:

```
In :  plt.imsave("Irlanda2.jpg",A)
```

La sentencia anterior nos crearía un nuevo archivo llamado “Irlanda2.jpg” que contendría la imagen almacenada en la hipermatriz `A`.

Veamos un ejemplo de manipulación de una imagen. Usaremos una foto de una ardilla, guardada en el archivo “squirrel.jpg”.

```
In :  A=imread("squirrel.jpg")
In :  A.shape
Out:  (300, 400, 3)
```

Vemos que se trata de una imagen de 300 pixels de alto por 400 pixels de ancho. La tercera dimensión corresponde a cada color RGB. Mostramos la imagen con `imshow`:

```
In :  plt.imshow(A)
```

La matriz tridimensional `A` contiene números enteros entre 0 y 255, de tipo ’uint8’. Podemos comprobarlo con el método `.dtype`:

```
In :  A.dtype
Out:  dtype('uint8')
```

A la hora de trabajar con colores, tenemos dos opciones: trabajar con enteros entre 0 y 255, o trabajar con números reales entre 0 y 1. Si queremos realizar operaciones con los

colores de una imagen, la segunda opción será más útil. Para convertir la imagen en números decimales con precisión simple entre 0 y 1 usaremos el método `.astype('float32')` (para usar precisión doble, escribiríamos 'float64'), tras lo cual dividiremos la matriz entre 255:

```
In :  B=A.astype('float32')/255
```

A continuación, mostramos los colores RGB del píxel $[0,0]$ de la imagen en ambos formatos:

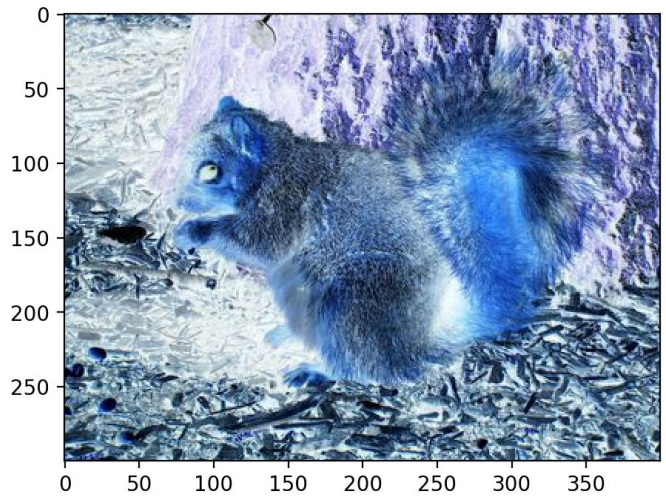
```
In :  A[0,0,:]
Out:  array([65, 36, 22], dtype=uint8)
In :  B[0,0,:]
Out:  array([0.25490198, 0.14117648, 0.08627451], dtype=float32)
```

Si dividimos `B` entre 2, conseguiremos oscurecer la imagen, ya que el valor de cada color será proporcionalmente más bajo:

```
In :  plt.imshow(B/2)
```

Multiplicando por 2 conseguiríamos el efecto contrario. Es fácil conseguir un negativo de la imagen: solo tendremos que restar 255 a cada color de la matriz `A` para obtener el color complementario (o bien 1 a cada color de la mariz `B`).

```
In :  plt.imshow(255-A)
```



El siguiente código voltea la imagen horizontalmente (usando "::-1" tomamos los elementos de cada columna desde el último al primero):

```
In :  plt.imshow(B[:,::-1,:])
```

Para voltearla verticalmente solo tendríamos que modificar ligeramente el código anterior:

```
In :  plt.imshow(B[::-1,:,:])
```

Índice de comandos