Tema 0 - Funciones y Graficación con python

M. en C. Gustavo Contreras Mayén

M. en C. Abraham Lima Buendía

Facultad de Ciencias - UNAM

20 de febrero de 2020

Las funciones en python

Con lo que hemos revisado sobre python, tenemos elementos para iniciar la solución de problemas.

Si agrupamos de manera particular un conjunto de instrucciones, entonces estaremos usando las llamadas *funciones*.

Estructura de una función

La palabra reservada **def** se usa para definir funciones.

Debe seguirle el nombre de la función (identificador), así como una lista de argumentos entre paréntesis, puede ocurrir que haya funciones que no ocupen argumentos.

Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

Estructura de una función

La estructura de una función en `python` es la siguiente:

```
def nombre_funcion(parametro1, parametro2, ...):
    conjunto de instrucciones
    return valores_devueltos
```

donde `parametro1`, `parametro2` son los parámetros. Un parámetro puede ser cualquier objeto de `python`, incluyendo una función.

Estructura de una función

Los parámetros pueden darse por defecto, por lo que en la función son opcionales.

Si no se utiliza la instrucción **return**, la función devuelve un objeto de tipo **None**

Ejemplo de una función

```
1 def cuadrados(entrada):
2     for i in range(len(entrada)):
3         entrada[i] = entrada[i]**2
4     return entrada
5
6 a = [1, 2, 3, 4]
7 respuesta = cuadrados(a)
8 print(respuesta)
```

Cálculo de la serie de Fibonacci

La sucesión fue descrita por Fibonacci como la solución a un problema de la cría de conejos:

"Cierto hombre tenía una pareja de conejos juntos en un lugar cerrado y uno desea saber cuántos son creados a partir de este par en un año, cuando es su naturaleza parir otro par en un simple mes, y en el segundo mes los nacidos parir también"

Cálculo de la serie de Fibonacci

La sucesión fue descrita por Fibonacci como la solución a un problema de la cría de conejos:

"Cierto hombre tenía una pareja de conejos juntos en un lugar cerrado y uno desea saber cuántos son creados a partir de este par en un año, cuando es su naturaleza parir otro par en un simple mes, y en el segundo mes los nacidos parir también"

cómo le hacemos para calcular cuántos conejos hay?

Propuesta de código

```
1 def fib(n):
2     a, b = 0, 1
3     while b < n:
4         print (b)
5         a, b = b, a + b
6
7 fib(2000)
```

Segunda propuesta de código

```
1  def fib_2_(n):
2      resultado = []
3      a, b = 0, 1
4      while a < n:
5          resultado.append(a)
6          a, b = b, a + b
7      return resultado
8
9  fibo100 = fib2(100)
10 print(fibo100)
```

Paso de argumentos

Para que una función sea en verdad útil (y reutilizable), es necesario que podamos pasarle datos a modo de *entradas*.

Los nombres de las entradas (o argumentos) que requiere una función se declaran a continuación del nombre en **def** (siempre entre paréntesis)

Ejemplo de función con argumentos

```
def FuncionSuma (x, y):
    return x + y

print(FuncionSuma(5, 3))
print(FuncionSuma(7, 42.0))
print(FuncionSuma(" hola ", " mundo "))
```

Ejemplo de función con argumentos

Notas:

- Nunca se mencionan los tipos de datos para x e y, ni el tipo de datos que devuelve FuncionSuma.

Ejemplo de función con argumentos

Notas:

- Nunca se mencionan los tipos de datos para x e y, ni el tipo de datos que devuelve FuncionSuma.
- Los argumentos y el valor devuelto son, tal como las variables, simples etiquetas a zonas de memoria.

Paso de argumentos con nombre

Si la función que definimos tiene muchos argumentos, es fácil olvidar el orden en que fueron declarados.

Como un argumento no lleva asociado un tipo, `python` no tiene manera de saber que los argumentos están cambiados.

Paso de argumentos con nombre

Para evitar este tipo de errores, hay una manera de llamar a una función pasando los argumentos en cualquier orden arbitrario: se pasan usando el nombre usado en la declaración.

Ejemplo de paso de argumentos con nombre

```
1 def Prueba (a, b, c):
2     print("a = {}, b = {}, c = {}".format
    (a, b, c)
3
4
5
6 Prueba (1, 2, 3)
7 Prueba (b = 3, a = 2, c = 1)
```

Argumentos con valores por omisión I

Para hacer que algunos argumentos sean opcionales, se les da valores por omisión en el momento de declararlos:

```
1 from math import sqrt
2
3 #el argumento v es necesario
4 def Gamma(v, c=3.0e+8):
5     return sqrt(1.0-(v/c)**2)
6
```

Argumentos con valores por omisión II

```
7
8  #se modifica el valor de c
9  print(Gamma(0.1, 1.0))
10
11 #solo se pasa el valor de v
12 print(Gamma(1.e+7))
```

Devolviendo varios valores en una función

Para hacer que una función devuelva más de un valor, lo que hacemos es definir argumentos de entrada y argumentos de salida.

Veremos la manera en que se deben de ocupar los valores de salida, una vez que se cumple la función.

Devolviendo varios valores en una función

Para devolver múltiples valores en `python`, lo usual es devolver los valores "empaquetados" en una tupla:

Regresando varios valores en una función I

```
1 from math import atan2, sqrt
2
3 def ModuloArgumento(x, y):
4     norma = sqrt(x**2 + y**2)
5     argumento = atan2(y, x)
6     return (norma, argumento)
7
8
9 n, a = ModuloArgumento(3.0, 4.0)
```

Regresando varios valores en una función II

```
10 print("El modulo es: ", n)
11 print("El argumento es: ", a)
```

Número variable de argumentos

¿Cómo le hacemos para que una función acepte un número no prefijado de argumentos?

Es posible pasar una lista o tupla, pero `python` ofrece una mejor solución:

Número variable de argumentos I

```
1 from math import atan, atan2
2
3 #args es una tupla de argumentos
4 def atanVariable(*args):
5     if len(args) == 1:
6         return atan(args[0])
7     else :
8         return atan2(args[0], args[1])
9
```

Número variable de argumentos II

```
10
11 print (atanVariable(0.2)) # 0.19739
12 print (atanVariable(2.0,10.0)) # 0.19739
13 print (atanVariable(-2.0,-10.0)) #
    -2.94419
```

Funciones lambda

`python` admite una interesante sintaxis que permite definir funciones mínimas, de una línea sobre la marcha.

Se trata de las denominadas funciones **lambda**, que pueden utilizarse en cualquier lugar donde se necesite una función.

Funciones lambda I

```
1
2 #primera forma de una funcion lambda
3
4 numero = float(input("Introduce un numero
      real"))
5
6 g = lambda x: x*2
7 print(g(numero))
8
```

Funciones lambda II

```
9  #segunda forma de escribir una funcion
       lambda
10
11 print((lambda x: x*2)(numero))
```

Funciones lambda

Nótese que: la lista de argumentos no está entre paréntesis, y falta la palabra reservada **return** (está implícita, ya que la función entera debe ser una única expresión).

Igualmente, la función no tiene nombre, pero puede ser llamada mediante la variable a que se ha asignado.

Módulos

Es una buena práctica de programación almacenar las funciones en módulos.

Módulos

Es una buena práctica de programación almacenar las funciones en módulos.

Un *módulo* es un archivo en donde se almacenan las funciones, el nombre del módulo es el nombre del archivo.

Cargando una función de un módulo

Un módulo se carga al programa principal con la instrucción

```
from nombre_modulo import funcion1 [, funcion2, ]
```

Cargando una función de un módulo

Un módulo se carga al programa principal con la instrucción

```
from nombre_modulo import funcion1 [, funcion2, ]
```

De esta manera tendremos disponibles las funciones del módulo en el código principal. En caso de que sean pocas las funciones, esta manera de llamar a las funciones es la más conveniente.

Funciones definidas por el usuario

Tenemos en `python` un conjunto de paquetes, librerías y funciones que nos van a ser de utilidad, no sólo para el curso de Física Computacional, sino para resolver algún problema en general.

Funciones definidas por el usuario

En el caso de que elaboremos nuestras funciones, tendremos entonces un conjunto de *funciones definidas por el usuario*.

En un primer momento las podemos tener incluidas dentro de nuestro código principal y en un sólo archivo.

Funciones definidas por el usuario

En un segundo momento, debido a la extensión de nuestro código será necesario elaborar un módulo de funciones para utilizarlo.

Por lo que debemos de revisar la manera en la que se deben de llamar las funciones.

Funciones definidas por el usuario

Usemos como ejemplo en un solo archivo `fibonacci.py` las funciones para calcular la serie de Fibonacci: `fib(n)` y `fib2(n)`

El archivo `fibonacci.py` debe de estar en la misma ruta que el archivo que va a llamar a las funciones.

Llamada a un módulo de usuario

```
1 from fibonacci import fib, fib2
2
3 print('Llamada a fib(n) del modulo de
     usuario\n')
4 fib(2000)
5
6 print('\nLlamada a fib2(n) del modulo de
     usuario\n')
7 fibo100 = fib2(100)
8 print(fibo100)
```

Cargando todo de un paquete

Es posible cargar en memoria todo el contenido de un paquete:

```
import math
```

Cargando todo de un paquete

Es posible cargar en memoria todo el contenido de un paquete:

```
import math
```

El único inconveniente de este llamado es que debemos de hacer la referencia correcta a cada función contenida en el módulo o paquete.

Sintaxis para llamar una función de un paquete

Una vez cargado el paquete, ahora tenemos que llamar debidamente a la función de interés:

```
alfa = math.sqrt(3.14)
```

Sintaxis para llamar una función de un paquete

Una vez cargado el paquete, ahora tenemos que llamar debidamente a la función de interés:

```
alfa = math.sqrt(3.14)
```

Esta forma de escritura se vuelve en ocasiones larga, para ello, es posible simplificar su escritura.

Uso de un alias para la carga de módulos

Al momento de cargar un paquete, podemos establecer un *alias* que nos va a ahorrar la escritura

```
1 import numpy as np
2
3 numero = int(input("Escribe un numero
     positivo: "))
4
5 raiz = np.sqrt(numero)
```

Uso de un alias para la carga de módulos

En caso de que hayamos hecho referencia a una función y no se utilice, `spyder` va a detectar esta situación y nos va a alertar de ello.

Este tipo de advertencias se conocen como warning, el programa se ejecuta y se mantiene ese aviso.

El paquete `math`

Muchas funciones matemáticas no se pueden llamar directo del intérprete, pero para ello existe el paquete `math`.

Podemos ver el contenido de un módulo con la instrucción:

El paquete math

```
import math
dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```

Total de funciones de `math`

El paquete `math` contiene 56 funciones.

Existen otros paquetes con funciones matemáticas que extienden al paquete `math`, por ejemplo, el paquete `numpy`, contiene las funciones de `math`, mas otras adicionales, con un total de 634 funciones.

Otros paquetes de python

El paquete `math` contiene 56 funciones.

Existen otros paquetes con funciones matemáticas que extienden al paquete `math`, por ejemplo, el paquete `numpy`, contiene las funciones de `math`, mas otras adicionales, con un total de 634 funciones.

Paquetes no estándar

Existe una gran variedad de paquetes en `python`, prácticamente para todas las áreas de trabajo: científica, programación web, audio, imagen, etc.

Mencionaremos algunos paquetes importantes para el área de ciencias.

Paquete numpy

NumPy

Su característica más potente es que puede trabajar con matrices de n dimensiones. Ofrece funciones básicas de algebra lineal, transformada de Fourier, capacidades avanzadas con números aleatorios, y herramientas de integración con otros lenguajes de bajo nivel como Fortran, C y C++.

Paquete scipy

scipy está construida sobre la librería numpy. Es una de las más útiles por la gran variedad que tiene de módulos de alto nivel sobre ciencia e ingeniería, como transformada discreta de Fourier, álgebra lineal, funciones especiales y matrices de optimización.

Paquete numpy

Es una librería de gráficos, desde histogramas, hasta gráficos de líneas o mapas de color. También se pueden usar comandos de LaTeXpara agregar expresiones matemáticas a la gráfica.

Paquete pandas

Se utiliza para operaciones y manipulaciones de datos estructurados. Es muy habitual usarlo en la fase de depuración y preparación de los datos.

Paquete seaborn

Está basada en `matplotlib`, se usa para hacer más atractivos los gráficos e información estadística en `python`.

Su objetivo es darle una mayor relevancia a las visualizaciones, dentro de las tareas de exploración e interpretación de los datos.

Graficación con python

Una buena parte del trabajo que tendremos que hacer como físicos es utilizar un conjunto de datos que por si solos, no van a darnos información sobre un modelo o un fenómeno, por ello, será necesario usar gráficas.

De tal manera que contemos con elementos para dar una interpretación de los resultados obtenidos con un código.

Módulo de graficación

`python` incluye un módulo de graficación bastante versátil para generar gráficas y exportarlas a diferentes tipos de archivos.

La librería se llama `matplotlib` que contiene un conjunto de funciones propias para crear gráficas. Haremos algunos ejercicios sencillos para demostrar su potencia.

El módulo pyplot

Consideremos que `matplotlib` es un paquete entero, y `matplotlib.pyplot` es una colección de funciones de estilo.

El módulo pyplot

Consideremos que `matplotlib` es un paquete entero, y `matplotlib.pyplot` es una colección de funciones de estilo.

Cada instrucción `pyplot` aplica un cambio a una figura: por ejemplo, crear una figura, crear un área de trazado en una figura, trazar algunas líneas en un área de trazado, decorar con etiquetas, etc.

Código para el Ejercicio 1

Código 1: Gráfica básica

```
1 import matplotlib.pyplot as plt
2
3 plt.plot([1, 2, 3, 4])
4 plt.ylabel('algunos numeros')
5 plt.show()
```

Gráfica que se obtiene en el Ejercicio 1

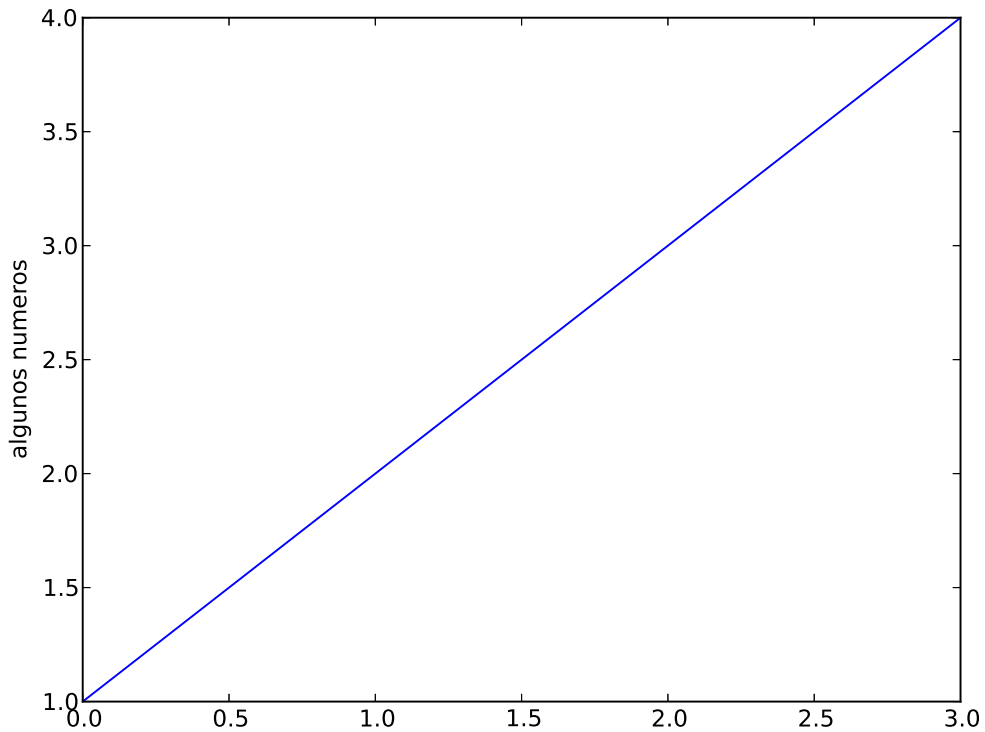


Figura: Gráfica obtenida por el primer código.

Sobre la gráfica obtenida

Viendo la gráfica anterior que obtuvimos con `matplotlib`, te estarás preguntando: ¿por qué tenemos en el eje x el rango $0 - 3$, mientras que en el eje y el rango va de $1 - 4$?

Respuesta

Cuando proporcionamos una única lista o matriz en el comando `plot( )`, entonces `matplotlib` asume que es una secuencia de valores para la variable y, por lo que genera automáticamente los valores para la variable x para nosotros, graficando entonces dos variables (x, y).

Respuesta

Como los índices en `python` comienzan en 0, el conjunto de datos *(vector)* para la variable x por defecto tiene la misma longitud que la variable y, pero inicia en 0.

De ahí que el conjunto de datos para la variable x son $[0, 1, 2, 3]$.

Código para el Ejercicio 2

```
1 import matplotlib.pyplot as plt
2
3 plt.plot([1,2,3,4], [1,4,9,16], 'ro')
4 plt.axis([0,6,0,20])
5 plt.show()
```

Gráfica que se obtiene en el Ejercicio 2

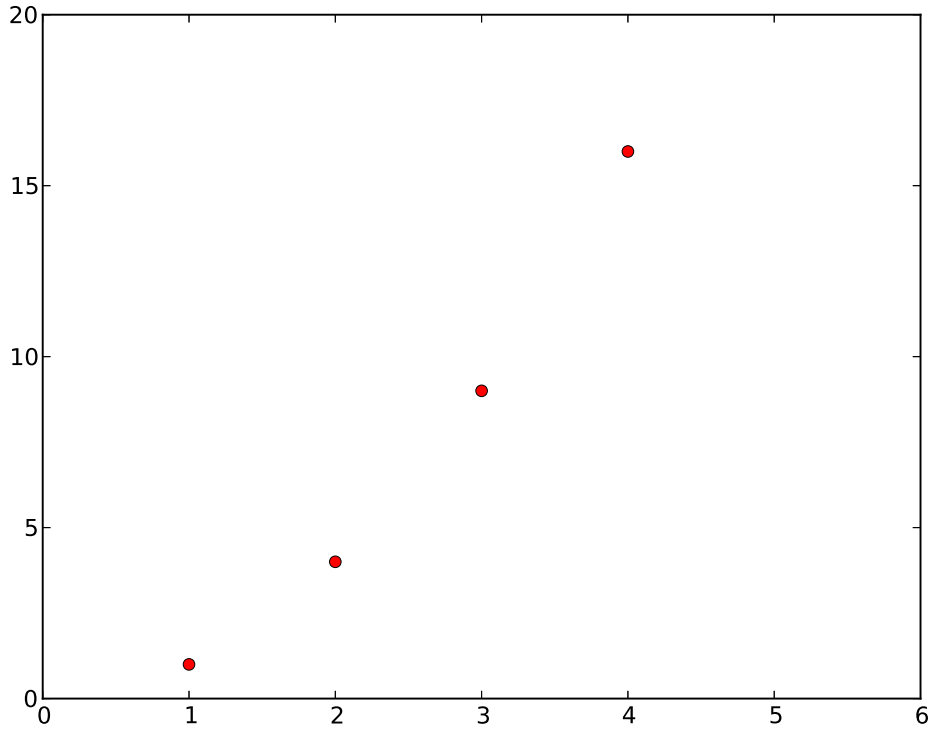


Figura: Gráfica obtenida con el segundo código.

Sobre la segunda gráfica

Se han introducido dos listas en el argumento de `plot ( )`, que corresponden a los valores para los pares (x, y).

Es importante señalar que para generar una gráfica, tanto la variable x como la variable y, *deben de contar con el mismo número de puntos.*

Sobre la segunda gráfica: cadena de formato

La cadena de formato por defecto es `'b-'`, que corresponde al color azul (`b = blue`). Es por ello que en la gráfica 1, los puntos quedaron unidos con la línea azul.

En la gráfica 2, usamos la cadena `'ro'`, que nos generó los círculos de color rojo (`r = red`).

Cadena de formato para el color

caracter	color
'b'	azul
'g'	verde
'r'	rojo
'c'	cyan
'm'	magenta
'y'	amarillo
'k'	negro
'w'	blanco

Esta lista presenta la correspondiente letra para incluir en la cadena de formato de color en la función `plot( )`, es posible personalizar el color a mostrar: consultando la documentación de `matplotlib`.

Cadena de formato para el tipos de línea

carácter	descripción
'-'	línea sólida
'--'	línea cortada
'-.'	línea-punto
':'	línea de puntos
'.'	marca de punto
','	marca de pixel
'o'	marca de círculo
'v'	marca de triángulo hacia abajo
'^'	marca de triángulo hacia arriba

Esta lista muestra el caracter para definir el tipo de línea en el argumento de la función la función `plot( )`.

Alcance de matplotlib

El paquete matplotlib se limita a trabajar con listas, por lo que su uso sería bastante acotado para el procesamiento y análisis numérico.

Por lo general, se utilizan los arreglos del paquete numpy.

Extendiendo matplotlib

De hecho, todas las secuencias se convierten en arreglos de numpy internamente.

El siguiente ejemplo ilustra un trazado de líneas con varios estilos diferentes en una sola instucción utilizando arreglos.

Código para el Ejercicio 3

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 t = np.arange(0., 5., 0.2)
5 plt.plot(t, t, 'r--', t, t**2, 'bs', t, t
     **3, 'g^')
6 plt.show()
```

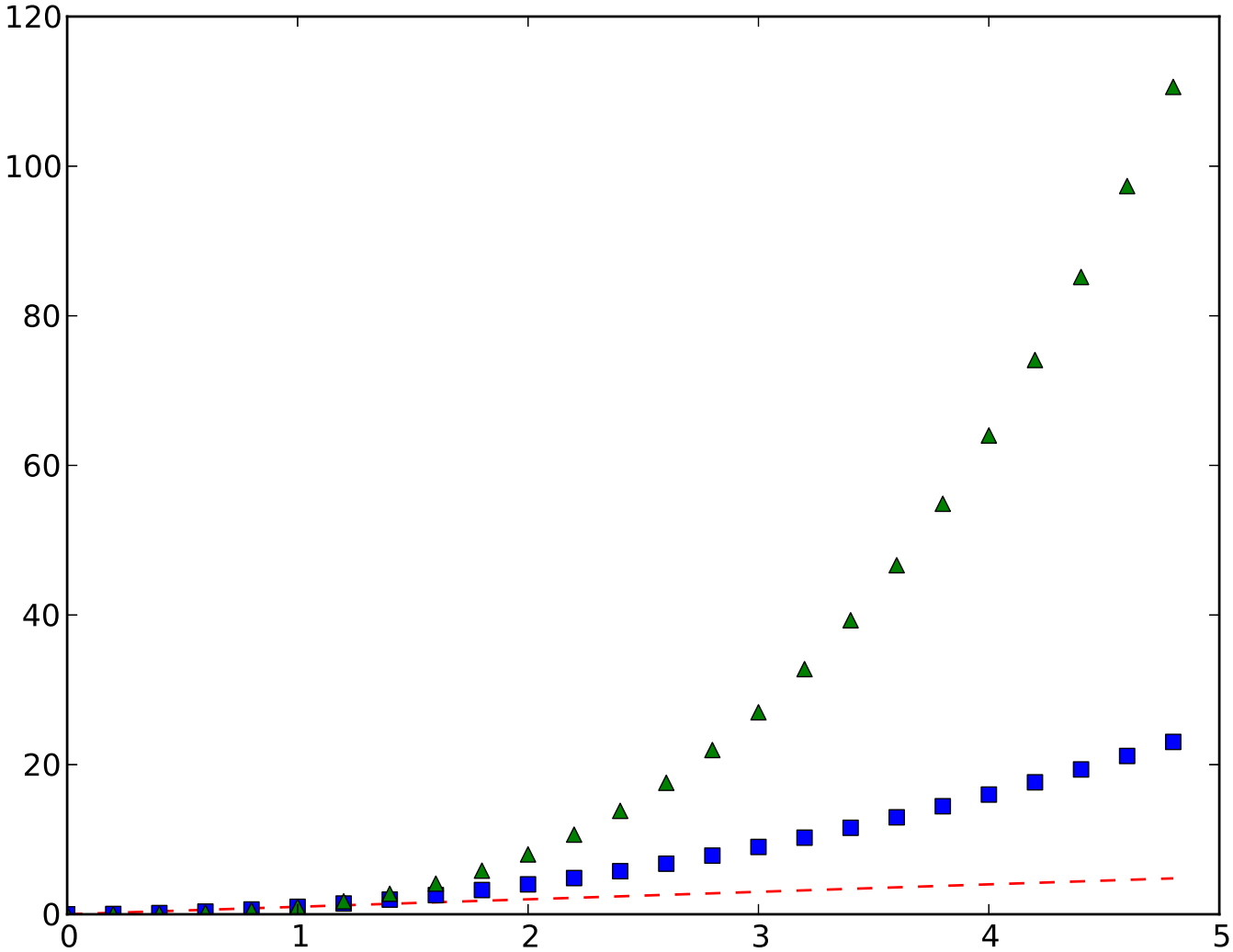


Figura: Gráfica con tres curvas.

Los subplots en python

Ya hemos generado una gráfica que contenga un trazo para una función en particular.

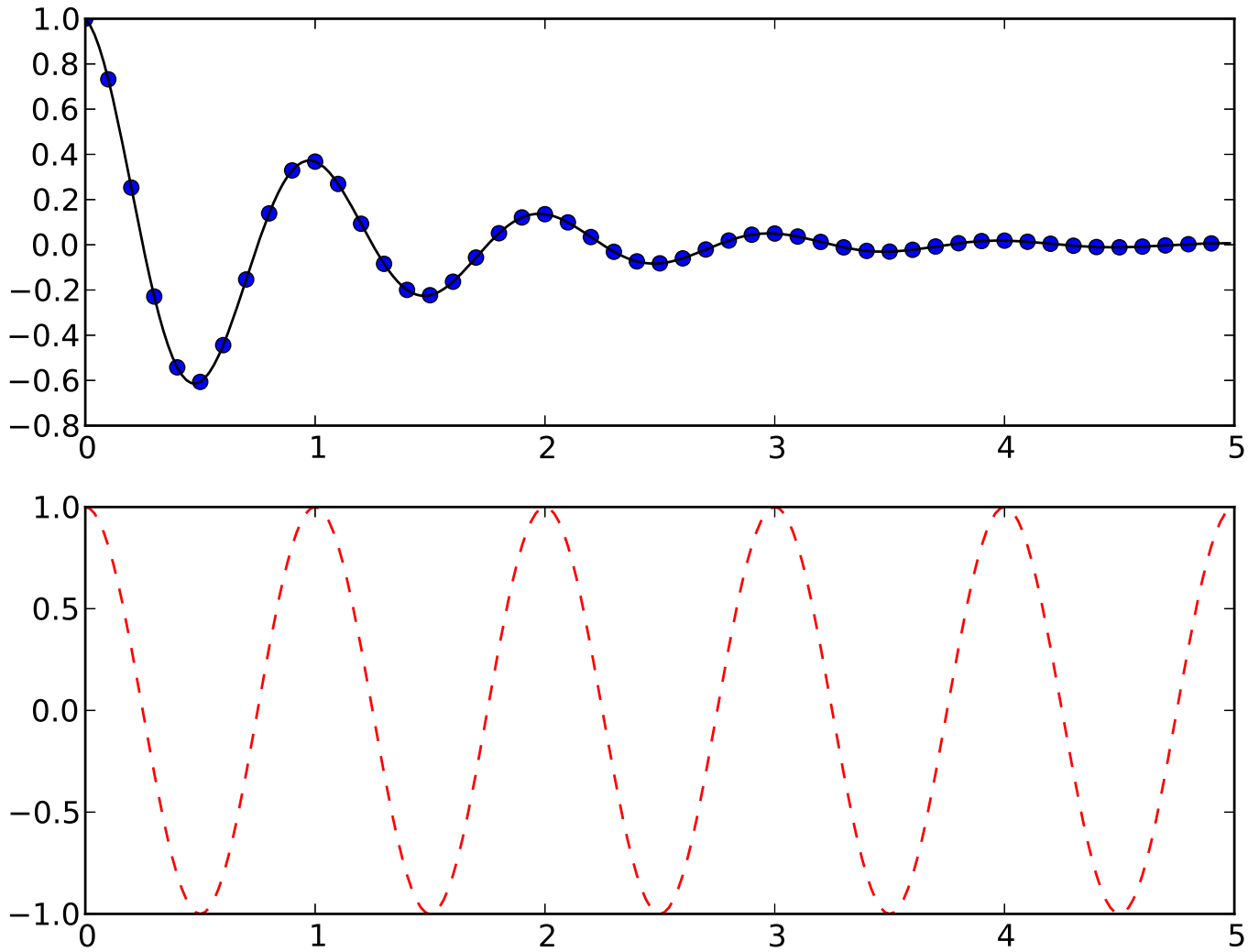
Pero hay que considerar que a veces necesitaremos mostrar dos (o más) gráficas dentro de la misma ventana), en éste caso usaremos los *subplots*.

Código para el Ejercicio 4 I

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(t):
5     return np.exp(-t) * np.cos(2*np.pi*t)
6
7 t1 = np.arange(0.0, 5.0, 0.1)
8 t2 = np.arange(0.0, 5.0, 0.02)
9
```

Código para el Ejercicio 4 II

```
10 plt.figure(1)
11 plt.subplot(211)
12 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
13
14 plt.subplot(212)
15 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
1
2
3
4
5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5

Especificar los `subplots`

El comando `figure( )` aquí es opcional, ya `figure(1)` se crea de forma predeterminada, así mismo `subplot(111)` se crea de forma predeterminada si no se especifica manualmente un eje.

Especificar los `subplots`

La función `figure1()` está avisando el uso de una ventana para mostrar la gráfica.

La función `subplot()` requiere que se señale como parámetros: `(numrows, numcols, fignum)` donde `fignum` varía en rango de `1` a `numrows * numcols`.

Especificar los `subplots`

Las comas en la función `subplot()` son opcionales si `numrows * numcols` < 10.

Por tanto `subplot(211)` es idéntica a la `subplot(2,1,1)`.

Código para el Ejercicio 5 I

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x1 = np.linspace(0, 18, 200)
5 x2 = np.linspace(0, 5, 100)
6
7 plt.figure(1)
8 plt.subplots_adjust(hspace=0.5)
9
```

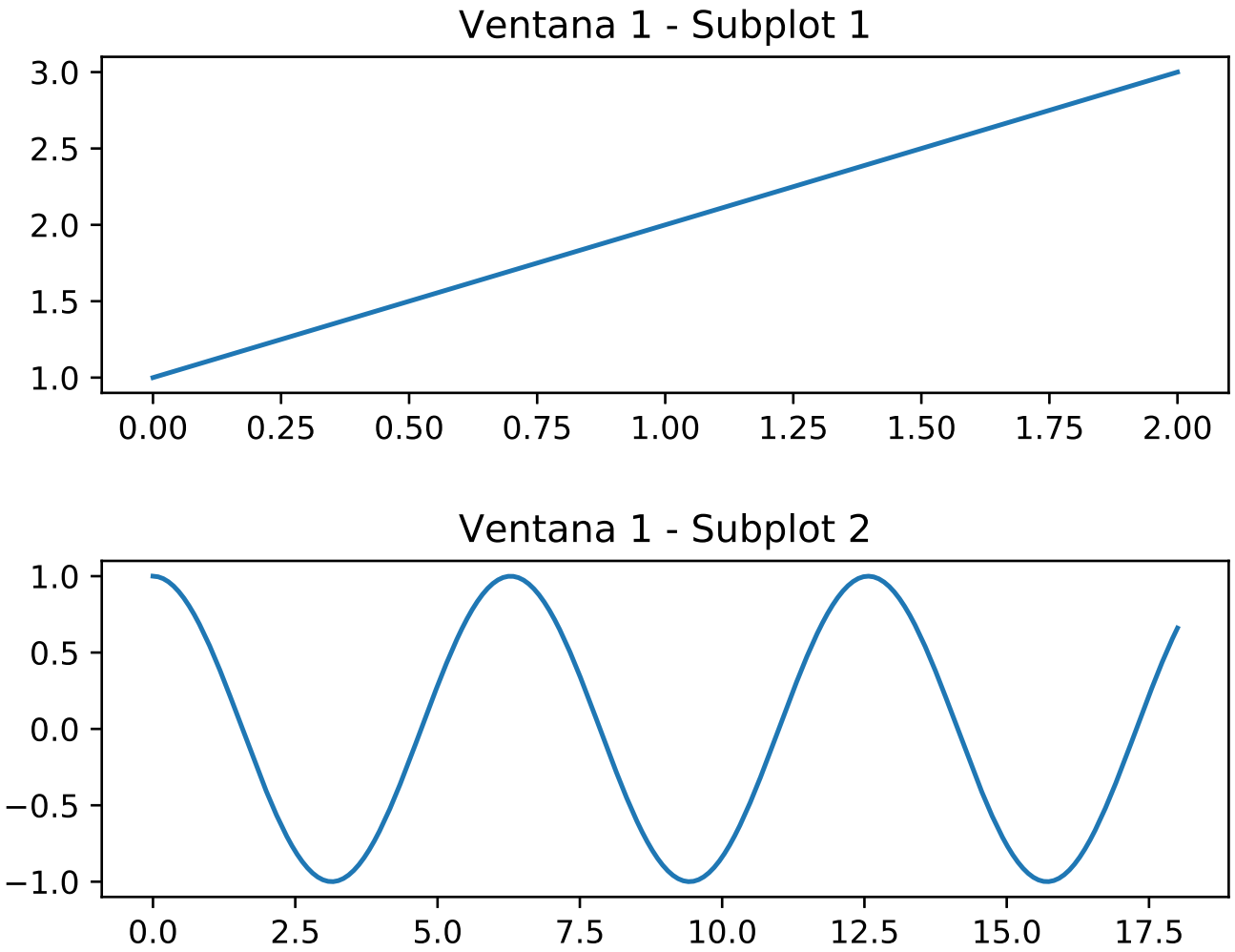
Código para el Ejercicio 5 II

```
10 plt.subplot(211)
11 plt.plot([1, 2, 3])
12 plt.title('Ventana 1 - Subplot 1')
13
14 plt.subplot(212)
15 plt.plot(x1, np.cos(x1))
16 plt.title('Ventana 1 - Subplot 2')
17
18 plt.figure(2)
19 plt.plot(x2, np.exp(x2))
```

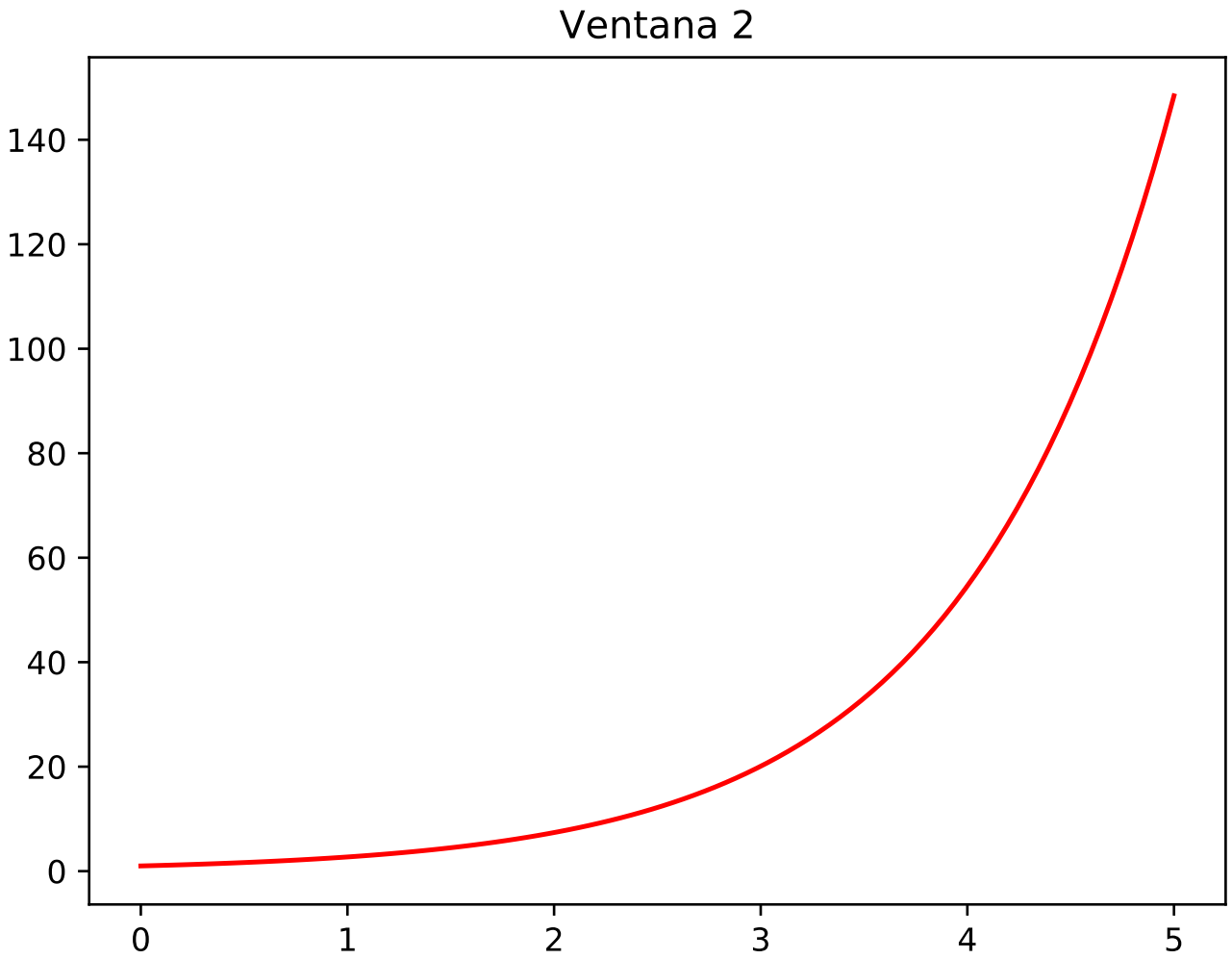
Código para el Ejercicio 5 III

```
20 plt.title('Ventana 2')
21
22 plt.show()
```

Primera ventana con dos subplots



Segunda ventana con una gráfica



Más recursos para graficar con python

Lo que hemos visto es una revisión muy básica y general de cómo generar una gráfica con python.

Más recursos para graficar con python

Hay una enorme cantidad de información sobre matplotlib, que encontrarás en la página oficial de la librería, así como bastante documentación, ejemplos y elementos para extender completamente esta herramienta.

Se les proporcionará una guía breve de graficación, con la intención de que revisen casos prácticos aplicados a la física.

Para cada gráfica que usemos más adelante en el curso, tendrán oportunidad de agregar más elementos que ustedes consideren.

Ejemplos de tipos de gráficas

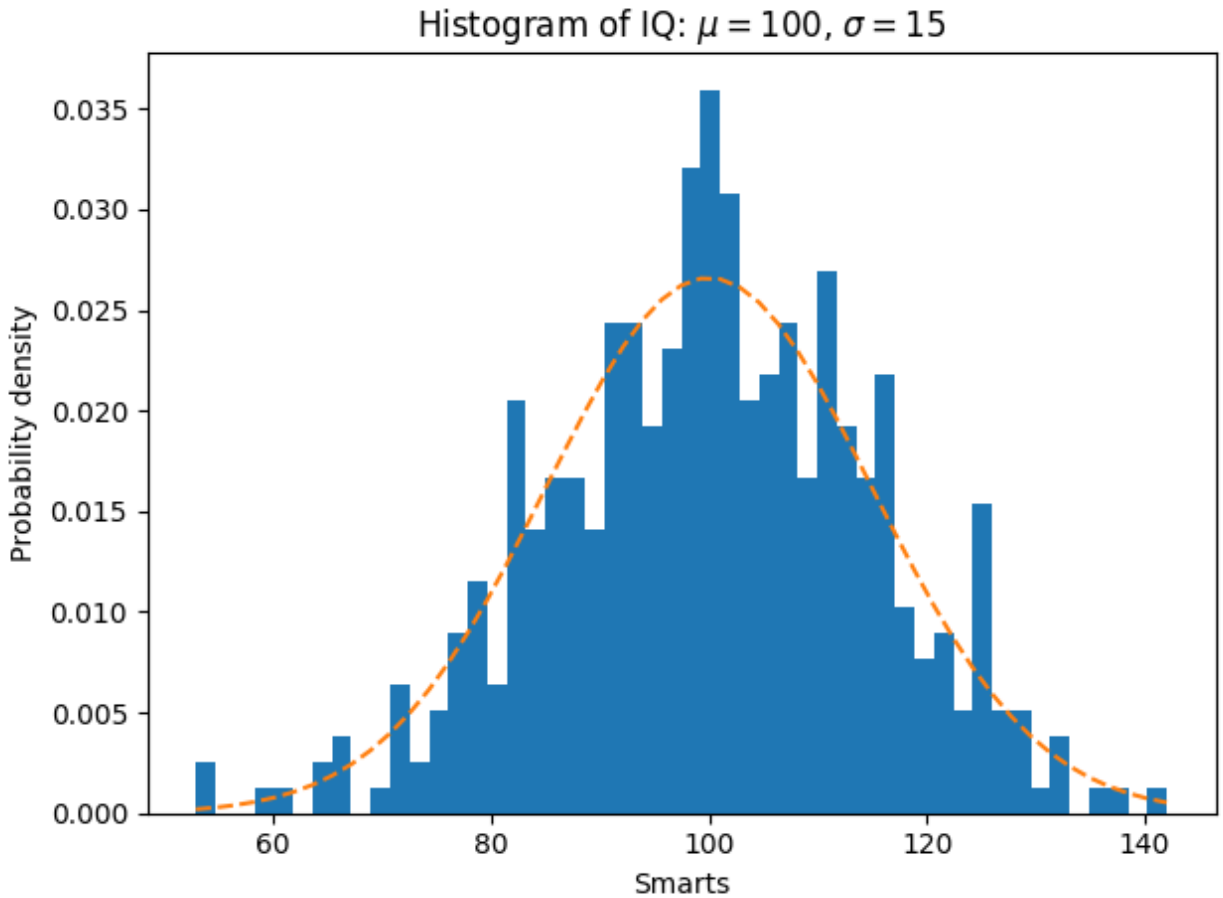


Figura: Histograma

Ejemplos de tipos de gráficas

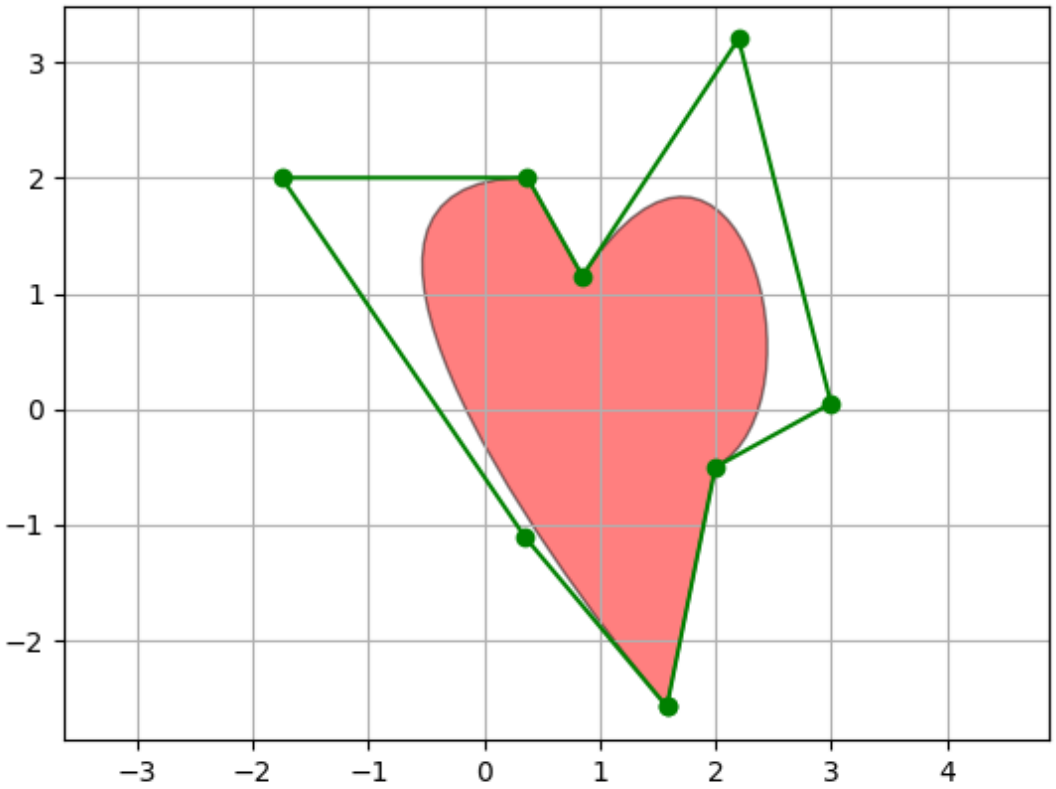


Figura: Contorno

Ejemplos de tipos de gráficas

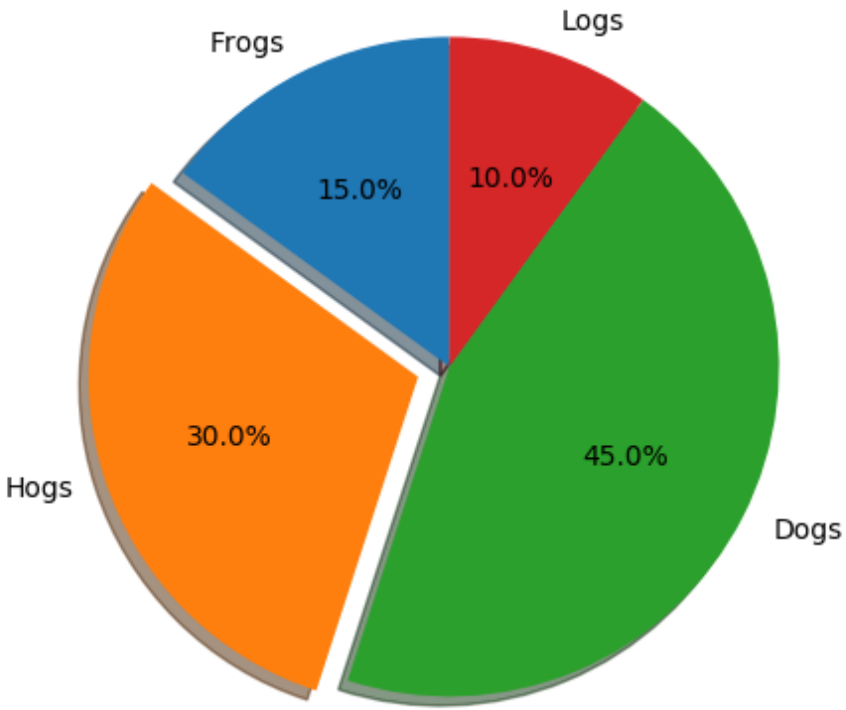


Figura: Pastel

Ejemplos de tipos de gráficas

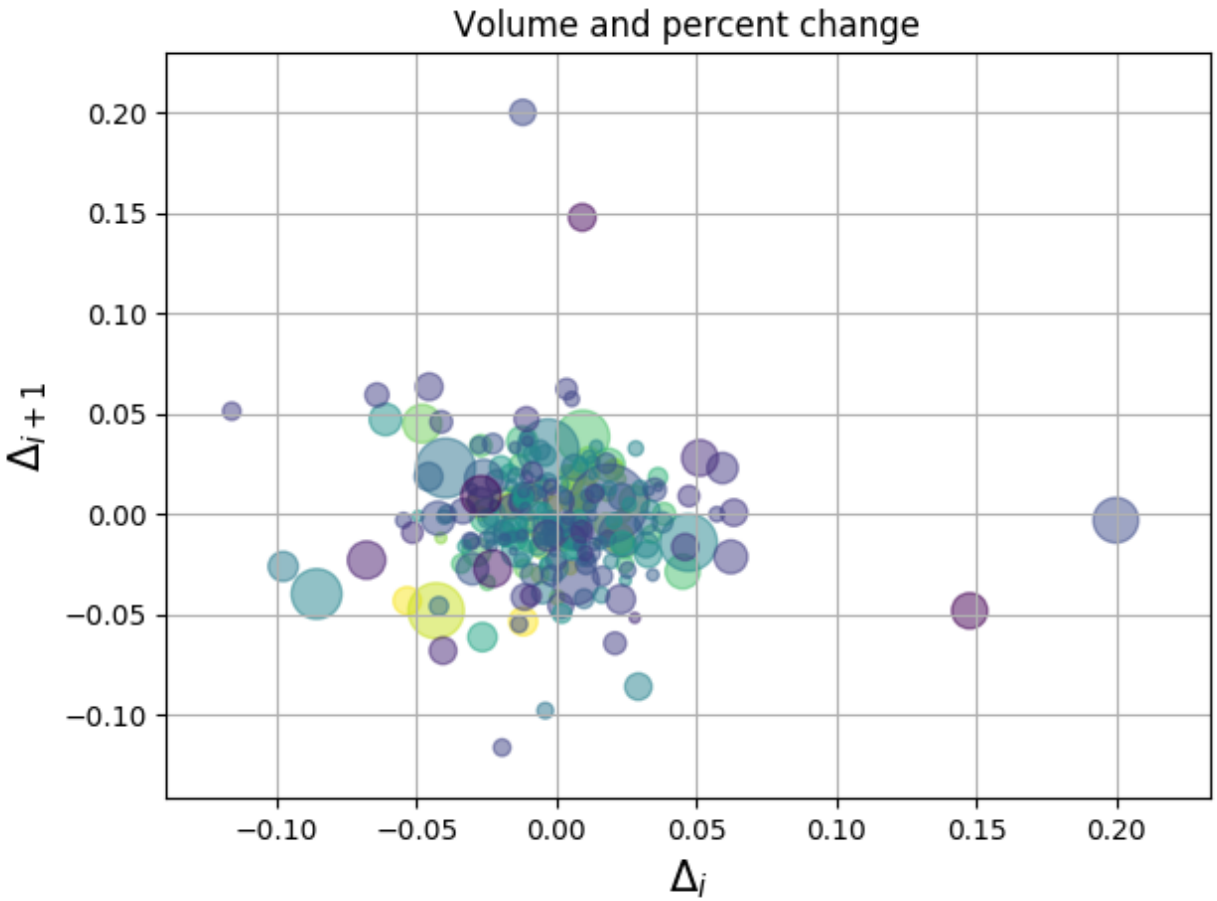


Figura: Dispersión

Ejemplos de tipos de gráficas

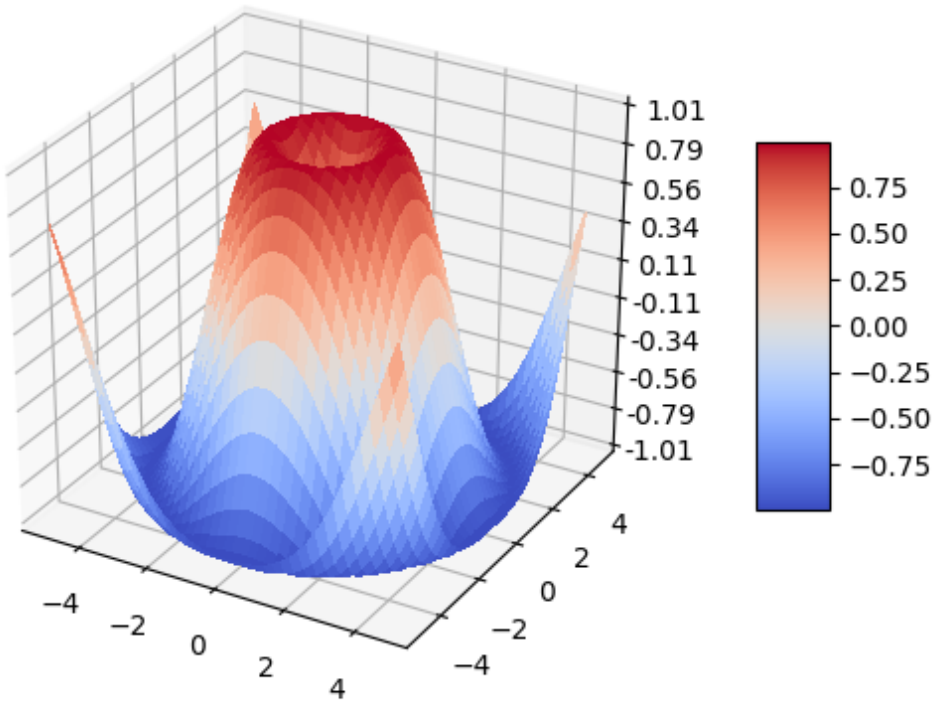


Figura: Superficie 3D

Ejemplos de tipos de gráficas

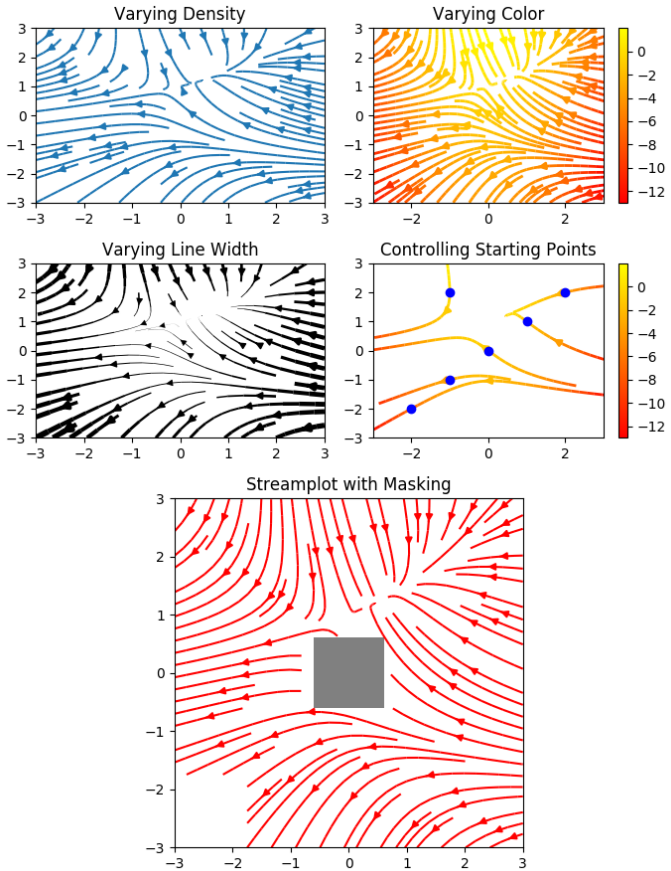


Figura: Campos vectoriales

Ejecicio: movimiento en bicicleta

La bicicleta es un medio muy eficiente de transporte, este es un hecho bien conocido por cualquier persona que monta una.

Nuestro objetivo en este ejercicio es comprender los factores que determinan la velocidad máxima de una bicicleta y estimar la velocidad de un caso real.

Ejecicio: Caso sin fricción

Comenzaremos haciendo caso omiso de la fricción; tendremos que añadirlo al final, por supuesto, pero debemos primero entender cómo lidiar con el caso más simple y sin fricción.

Ecuación de movimiento

La ecuación de movimiento corresponde a la segunda ley de Newton, que escribimos de la forma

$$\frac{dv}{dt} = \frac{F}{m} \tag{1}$$

Ecuación de movimiento

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{F}{m}$$

donde

1. v es la velocidad.

Ecuación de movimiento

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{F}{m}$$

donde

1. v es la velocidad.
2. m es la masa de la combinación de la bicicleta-conductor.

Ecuación de movimiento

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{F}{m}$$

donde

1. v es la velocidad.
2. m es la masa de la combinación de la bicicleta-conductor.
3. t es el tiempo.

Ecuación de movimiento

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{F}{m}$$

donde

1. v es la velocidad.
2. m es la masa de la combinación de la bicicleta-conductor.
3. t es el tiempo.
4. F es la fuerza en la bicicleta que viene del esfuerzo del conductor (supondremos que la bicicleta se mueve sobre un terreno plano)

Definiendo la fuerza

Tratar debidamente a la fuerza F, se complica por la mecánica de la bicicleta: ya que la fuerza ejercida por el ciclista se transmite a las ruedas por medio del plato, engranajes, cadena, etc.

Esto hace que sea muy difícil derivar una expresión exacta para F.

Definición alterna de la fuerza

Sin embargo, hay otra manera de abordar este problema que evita la necesidad de conocer la fuerza.

Este enfoque alternativo implica la formulación del problema en términos de la potencia generada por el ciclista.

Comparando la potencia

Estudios fisiológicos en ciclistas de carreras han demostrado que éstos atletas son capaces de producir una potencia de salida de aproximadamente 400 watts durante largos períodos de tiempo (~ 1 h)

Usando la potencia generada

Usando las ideas de trabajo-energía podemos reescribir (1) como

$$\frac{\mathrm{d}E}{\mathrm{d}t} = P \tag{2}$$

donde E es la energía total, P es la potencia de salida del ciclista.

Expresando la velocidad

Para un trayecto plano la energía es totalmente cinética, es decir,

$$E = \frac{1}{2}mv^2$$

y además

$$\frac{\mathrm{d}E}{\mathrm{d}t} = mv\left(\frac{\mathrm{d}v}{\mathrm{d}t}\right)$$

Expresando la velocidad

Para un trayecto plano la energía es totalmente cinética, es decir,

$$E = \frac{1}{2}mv^2$$

y además

$$\frac{\mathrm{d}E}{\mathrm{d}t} = mv\left(\frac{\mathrm{d}v}{\mathrm{d}t}\right)$$

usando esto en la ec. (2), resulta

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{P}{mv} \tag{3}$$

Expresando la velocidad

Si P es una constante, la ecuación (3), se puede resolver de manera analítica, rearreglando términos:

$$\int_{v_0}^{v} v' \, \mathrm{d}v' = \int_{0}^{t} \frac{P}{m} \, \mathrm{d}t' \tag{4}$$

donde v_0 es la velocidad de la bicicleta en $t = 0$.

Obteniendo la velocidad

Integrando ambos lados de la ecuación (4 y resolviendo para v, tenemos que la velocidad del ciclista es:

$$v = \sqrt{v_0^2 + 2\,P\,\frac{t}{m}} \tag{5}$$

Pasando al código

Como ya tenemos un conjunto de elementos necesarios para resolver el ejercicio, ahora nos enfocamos a traducir en el lenguaje de `python`, lo necesario para la solución.

Recuerda que debemos de almacenar los valores nuevos de velocidad, como se calcula un valor nuevo por cada unidad de tiempo, entonces ya tenemos un par de variables para graficar.

El caso sin fricción I

```
1 import matplotlib.pyplot as plt
2 from math import sqrt
3
4 t = []
5 v = []
6 dt, potencia, masa, tmax = 1, 400, 70, 40
     0
7 nmax = tmax/dt
8
```

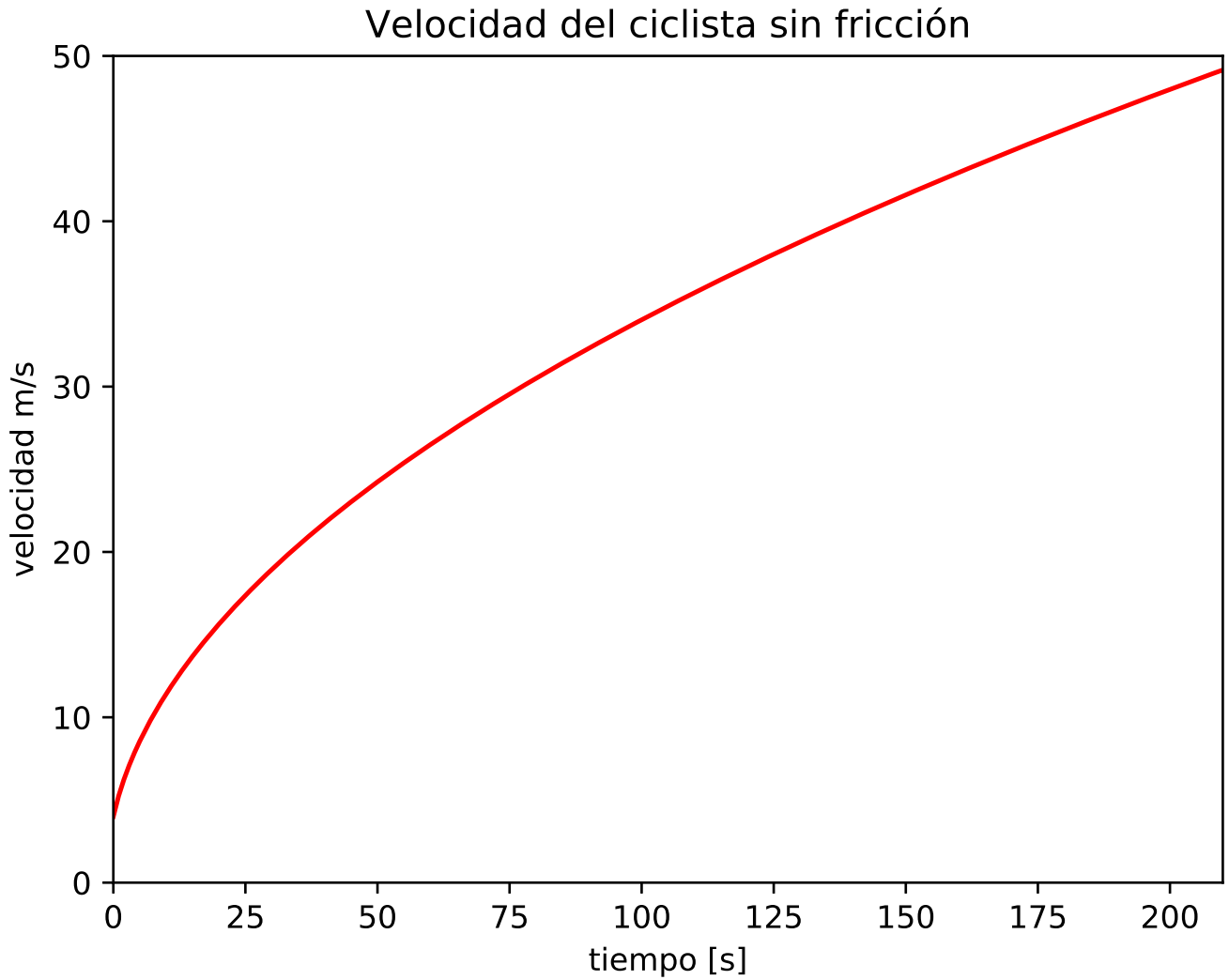
El caso sin fricción II

```
9  t.append(0)
10 v.append(4)
11
12 #calculando nuevos valores
13 for i in range(int(nmax)):
14     ti = t[i] + dt
15     vi = sqrt(v[i]**2 + (2*potencia*dt)/
       masa)
16
17     t.append(ti)
```

El caso sin fricción III

```
18      v.append(vi)
19
20  #Rutina de graficacion
21  plt.plot(v, "r-")
22  plt.xlabel("tiempo [s]")
23  plt.ylabel("velocidad m/s")
24  plt.title("Velocidad del ciclista")
25  plt.show()
```

Resultado de la velocidad sin fricción



Interpretación de los resultados

¿Es congruente la solución numérica con la física?

Interpretación de los resultados

¿Es congruente la solución numérica con la física?

La solución de la ecuación de movimiento (3) es correcta, pero nuestro trabajo no concluye aquí:

Interpretación de los resultados

¿Es congruente la solución numérica con la física?

La solución de la ecuación de movimiento (3) es correcta, pero nuestro trabajo no concluye aquí:

La física no checa

La solución predice que la velocidad del ciclista se incrementará sin límite para tiempos muy largos.

Ajuste en el modelo inicial

Vamos a corregir este resultado: cuando se generaliza el modelo se debe de incluir el efecto de la resistencia del aire.

Ajuste en el modelo inicial

Vamos a corregir este resultado: cuando se generaliza el modelo se debe de incluir el efecto de la resistencia del aire.

El nuevo término que vamos a añadir a la ecuación de movimiento nos obliga a desarrollar una solución numérica, así que con eso en mente se considera un tratamiento numérico de la ec. (3)

Abordaje numérico

Comenzamos con la forma de diferencias finitas para la derivada de la velocidad

$$\frac{\mathrm{d}v}{\mathrm{d}t} \simeq \frac{v_{i+1} - v_i}{\Delta t} \tag{6}$$

donde asumimos que Δt es paso discreto pequeño, y v_i es la velocidad al tiempo $t_i \equiv i\Delta t$.

Abordaje numérico

Por lo que de la ecuación (3), tenemos que la velocidad al paso $i+1$ es:

$$v_{i+1} = v_i + \frac{P}{m\, v_i}\Delta t \tag{7}$$

Calculando la velocidad

Dada la velocidad en un tiempo i (es decir, v_i), podemos usar la ec. (7), para calcular un valor *aproximado* de la velocidad en el siguiente paso v_{i+1}.

Calculando la velocidad

Si conocemos la velocidad inicial v_0, podemos obtener v_1, v_2, y así sucecivamente.

Ahora hay que introducir en la ecuación, los parámetros de fricción debida al aire.

Considerando la fricción del aire

La fuerza debida a la fricción puede aproximarse de manera inicial como

$$F_a \simeq -B_1\, v - B_2\, v^2 \tag{8}$$

Considerando la fricción del aire

$$F_a \simeq -B_1\, v - B_2\, v^2$$

Para velocidades muy bajas, el primer término es el que domina, y su coeficiente B_1 se puede calcular para objetos con formas sencillas.

Considerando la fricción del aire

$$F_a \simeq -B_1\, v - B_2\, v^2$$

Para una velocidad razonable v^2 el término domina sobre los demás, pero el coeficiente B_2 no puede calcularse exactamente en objetos sencillos como una pelota de beisbol, menos para una bicicleta.

Primera aproximación

Podemos aproximar el valor de B_2 como sigue:

Primera aproximación

Podemos aproximar el valor de B_2 como sigue:

Si un objeto se mueve a través de la atmósfera, entonces debe empujar al aire delante fuera del camino.

Primera aproximación

La masa de aire movido en el tiempo $\mathrm{d}t$ es

$$m_{aire} \sim \rho\, A\, v\ \mathrm{d}t$$

donde ρ es la densidad del aire y A el área frontal del objeto.

Primera aproximación

La masa de aire movido en el tiempo $\mathrm{d}t$ es

$$m_{aire} \sim \rho\, A\, v\ \mathrm{d}t$$

donde ρ es la densidad del aire y A el área frontal del objeto.

A este aire se le da una velocidad de orden v, y por lo tanto, su energía cinética es

$$E_{aire} \sim m_{aire}\, v^2/2$$

Primera aproximación

Este es también el trabajo realizado por la fuerza de arrastre (la fuerza sobre el objeto debido a la resistencia del aire) en el tiempo $\mathrm{d}t$, por lo que

$$F_a \, v \, \mathrm{d}t = E_{aire}$$

Primera aproximación

Este es también el trabajo realizado por la fuerza de arrastre (la fuerza sobre el objeto debido a la resistencia del aire) en el tiempo $\mathrm{d}t$, por lo que

$$F_a \, v \, \mathrm{d}t = E_{aire}$$

Poniendo todo esto junto nos encontramos con:

$$F_a \simeq -C \, \rho \, A \, v^2$$

Coeficiente de arrastre

De la expresión

$$F_a \simeq -C\,\rho\,A\,v^2$$

Tenemos que C es el coeficiente de arrastre, podemos argumentar que tiene un valor de 0.5 (¿cómo lo demostrarías?).

Coeficiente de arrastre

Recordemos que estamos haciendo una primera aproximación, si nos interesa contar con una expresión que determine la correcta relación de F_a con v y A, entonces tendremos que hacer mediciones en un túnel de viento, para utilizar el valor correcto de C.

Nueva expresión para la velocidad

Incluyendo este término en la expresión para la velocidad

$$v_{i+1} = v_i + \frac{P}{mv_i}\Delta t - \frac{C\rho A v_i^2}{m}\Delta t \tag{9}$$

Considera los siguientes valores:

1. Coeficiente de arrastre: $C = 0.5$

Nueva expresión para la velocidad

Incluyendo este término en la expresión para la velocidad

$$v_{i+1} = v_i + \frac{P}{mv_i}\Delta t - \frac{C\rho A v_i^2}{m}\Delta t \tag{9}$$

Considera los siguientes valores:

1. Coeficiente de arrastre: $C = 0.5$
2. Sección de área frontal: $0.33\,\mathrm{m}^2$

Nueva expresión para la velocidad

Incluyendo este término en la expresión para la velocidad

$$v_{i+1} = v_i + \frac{P}{mv_i}\Delta t - \frac{C\rho A v_i^2}{m}\Delta t \tag{9}$$

Considera los siguientes valores:

1. Coeficiente de arrastre: $C = 0.5$
2. Sección de área frontal: $0.33\ \text{m}^2$
3. Masa del sistema ciclista-bicicleta: $70\ \text{kg}$

Nueva expresión para la velocidad

Incluyendo este término en la expresión para la velocidad

$$v_{i+1} = v_i + \frac{P}{mv_i}\Delta t - \frac{C\rho A v_i^2}{m}\Delta t \tag{9}$$

Considera los siguientes valores:

1. Coeficiente de arrastre: $C = 0.5$
2. Sección de área frontal: $0.33\,\mathrm{m}^2$
3. Masa del sistema ciclista-bicicleta: $70\,\mathrm{kg}$
4. Velocidad inicial: $4\,\mathrm{m\,s}^{-1}$

Nueva expresión para la velocidad

Incluyendo este término en la expresión para la velocidad

$$v_{i+1} = v_i + \frac{P}{mv_i}\Delta t - \frac{C\rho A v_i^2}{m}\Delta t \tag{9}$$

Considera los siguientes valores:

1. Coeficiente de arrastre: $C = 0.5$
2. Sección de área frontal: $0.33\,\mathrm{m}^2$
3. Masa del sistema ciclista-bicicleta: $70\,\mathrm{kg}$
4. Velocidad inicial: $4\,\mathrm{m\,s}^{-1}$
5. Incremento en el tiempo: $\Delta\, t = 0.1\,\mathrm{s}$

Agregando código I

Agrega lo siguiente en tu código, te recomendamos guardar con otro nombre tu archivo.

```
v2 = []

A, C = 0.33, 0.5

#dentro del ciclo for
vc = v2[i-1] + potencia*dt / (masa* v2[i-1]) - (C*A*v2[i-1] ** 2) * dt/masa
```

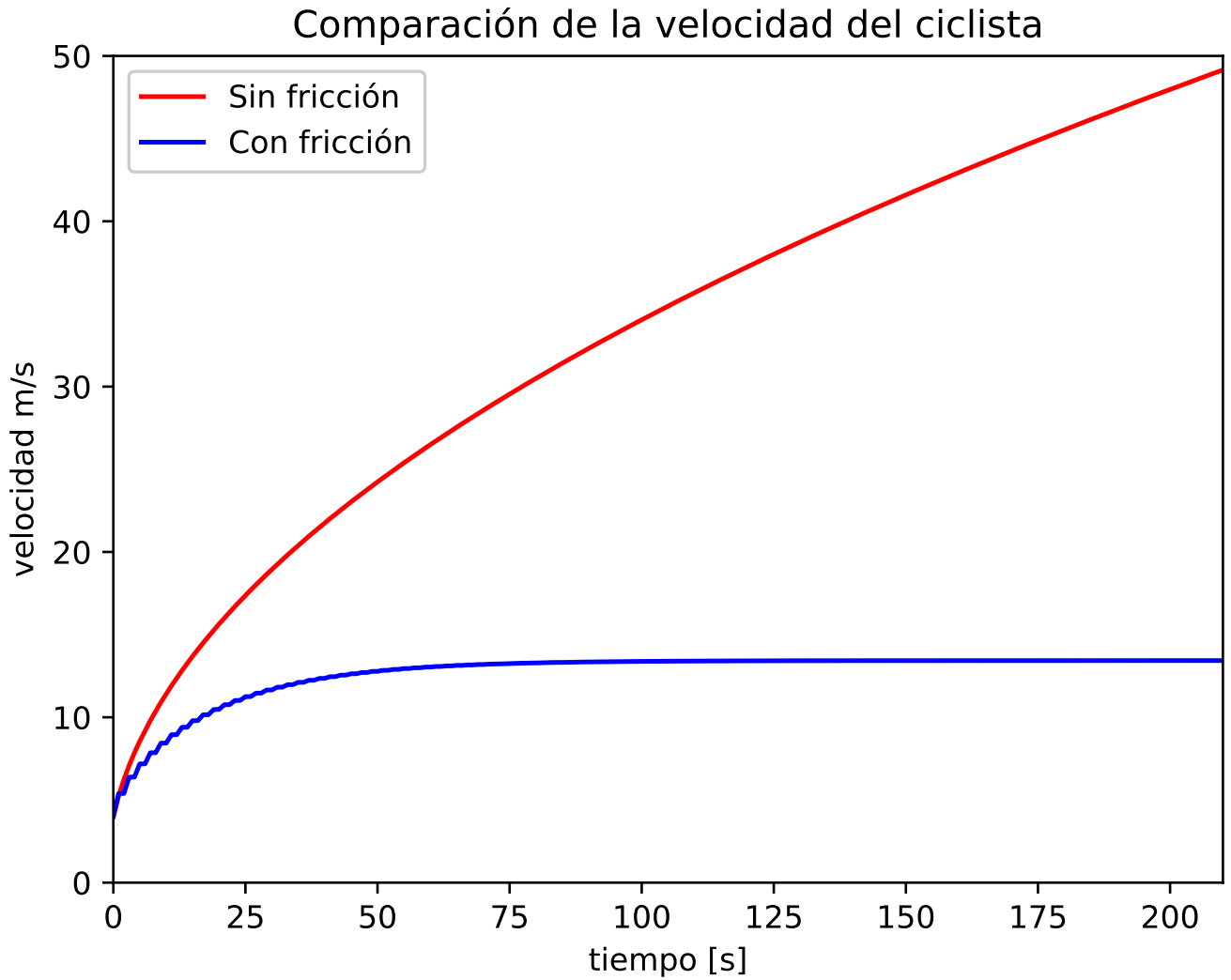
Agregando código II

```
7
8  v2.append(vc)
9
10 #en la graficacion cambia
11 plt.plot(v, "r-", label="Sin friccion")
12
13 #agrega lo siguiente
14 plt.plot(v2, "b-", label="Velocidad con
       friccion")
15 plt.legend(loc="upper left")
```

Agregando código III

```
16
17 plt.title("Comparacion de la velocidad
       del ciclista")
18 plt.axis([0, 210, 0, 50])
19
20 plt.show()
```

Comparando velocidades



Conclusión

Con este ejercicio encontramos que no necesariamente una solución que funcione desde el punto de vista informático, tendrá consistencia con la física.

Será nuestra tarea verificar que esa congruencia se mantenga en nuestros algoritmos y soluciones.