

VERSIÓN PRELIMINAR

# Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

**Título:** Python en Ámbitos Científicos  
**Autores:** Facundo Batista & Manuel Carlevaro  
**ISBN-13 (versión electrónica):** ???-?-???-???-?  
© Facundo Batista & Manuel Carlevaro  
**Primera Edición (versión preliminar)**  
Escrito con X<sub>3</sub>TEX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](https://creativecommons.org/licenses/by-nc-sa/4.0/)  
Lugar: Olivos y La Plata, Buenos Aires, Argentina  
Año: 2024  
Web: <http://pyciencia.taniquetil.com.ar/>

10 9 8 7 6 5 4 3 2 1

## Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científicos, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos" que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquivó la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

*Olivos y La Plata, Buenos Aires, Argentina,*

---

Facundo Batista & Manuel Carlevaro

# Índice general

Prefacio	2
Índice general	3
<b>I Python</b>	<b>4</b>
<b>II Herramientas fundamentales</b>	<b>5</b>
<b>III Temas específicos</b>	<b>6</b>
1. Ecuaciones diferenciales ordinarias	7
1.1. Ecuaciones diferenciales ordinarias . . . . .	7
1.2. Solución analítica de EDOs . . . . .	9
1.3. Métodos numéricos para resolver EDOs . . . . .	12
1.4. Solución de EDOs con valores iniciales . . . . .	15
1.5. Lecturas recomendadas . . . . .	19
<b>IV Apéndices</b>	<b>20</b>
A. Zen de Python	21
<b>Bibliografía</b>	<b>22</b>

# Parte I

## Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

## Parte II

### Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

## Parte III

### Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte [II](#).

# 1 | Ecuaciones diferenciales ordinarias

Las ecuaciones diferenciales ordinarias (EDOs) constituyen una de las herramientas más poderosas para la comprensión y predicción del comportamiento de sistemas dinámicos en la naturaleza, en ingeniería y en la sociedad. Un sistema dinámico es un sistema en algún estado, generalmente descrito por un conjunto de variables, que evoluciona en el tiempo. Por ejemplo, un cuerpo a una dada temperatura en un ambiente que está a otra temperatura, la evolución de una población, la propagación de un virus o el precio de un producto en un mercado de oferta y demanda. Las EDOs son ubicuas en todos los ámbitos de la ciencia y la tecnología<sup>1</sup>.

Para representar un sistema dinámico necesitamos expresar las reglas matemáticas que gobiernan su evolución en el tiempo, y para ello podemos usar las leyes de la física, o simplemente la intuición. Estas reglas, cuando las variables que representan el estado del sistema son continuas, toman la forma de ecuaciones diferenciales.

En las ecuaciones diferenciales la incógnita es una función, y para determinarla es necesario resolver dicha ecuación que contiene sus derivadas. Una función diferencial “ordinaria” es un caso especial en cual esta función desconocida tiene una sola variable independiente con respecto de la cual se deriva la función. Si por el contrario, la ecuación contiene derivadas de más de una variable, entonces es una ecuación en “derivadas parciales”, y esto se trata en el capítulo ??.

Por lo general, no existen soluciones analíticas de las EDOs excepto en algunos pocos casos, en los cuales se puede determinar la función desconocida utilizando métodos simbólicos. Si esto no es posible, existen diversos métodos numéricos que permiten obtener soluciones prácticas. En este capítulo exploraremos ambos abordajes para resolver EDOs.

## 1.1. Ecuaciones diferenciales ordinarias

La forma más simple de una EDO es

$$\frac{dy(x)}{dx} = f(x, y(x)) \quad (1.1)$$

<sup>1</sup> Tal vez la ecuación diferencial de mayor impacto en el desarrollo científico y tecnológico sea la Segunda Ley de Newton [2]



Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1
Sympy	1.13.3

[Código disponible](#)

donde  $y(x)$  es la función desconocida y  $f(x, y(x))$  se conoce. Esta ecuación es de primer orden porque solo aparece la derivada primera de  $f$  (omitiremos por comodidad la dependencia de la variable independiente en las expresiones de las derivadas). Más generalmente, una EDO de orden  $n$  se puede representar en forma “explícita” como

$$\frac{d^n y}{dx^n} = f\left(x, y, \frac{dy}{dx}, \dots, \frac{d^{n-1}y}{dx^{n-1}}\right) \quad (1.2)$$

o en forma “implícita” como

$$F\left(x, y, \frac{dy}{dx}, \dots, \frac{d^{n-1}y}{dx^{n-1}}\right) = 0 \quad (1.3)$$

donde  $f$  y  $F$  son funciones conocidas.

Cuando aparecen derivadas de órdenes superiores, siempre se puede reducir al estudio de un conjunto de EDOs de primer orden. Por ejemplo, la ecuación de segundo orden:

$$\frac{d^2 y}{dx^2} + f(x) \frac{dy}{dx} = g(x) \quad (1.4)$$

puede reescribirse como dos ecuaciones de primer orden:

$$\frac{dy}{dx} = z(x) \quad (1.5)$$

$$\frac{dz}{dx} = g(x) - f(x)z(x) \quad (1.6)$$

donde  $z$  es una nueva variable. Esto ejemplifica la forma usual de reducción de una ecuación de orden superior a un sistema lineal de EDOs, simplemente eligiendo las nuevas variables como derivadas de las otras (y de la variable original).

El problema entonces de las EDOs se reduce al estudio de un sistema de  $n$  ecuaciones diferenciales de primer orden acopladas, para las funciones  $y_i$ ,  $i = 1, 2, \dots, n$ , con la forma general

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \dots, y_n), \quad i = 1, \dots, n \quad (1.7)$$

en la que introducimos las nuevas  $n$  funciones  $y_1 = y$ ,  $y_2 = dy/dx$ , ...,  $y_n = d^{n-1}y/dx^{n-1}$ . Este sistema de EDOs de primer orden se puede escribir en una forma matricial:

$$\frac{d}{dx} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} y_2 \\ y_3 \\ \vdots \\ y_n \\ f(x, y_1, \dots, y_n) \end{bmatrix} \quad (1.8)$$

y en una forma vectorial compacta:

$$\frac{d}{dx} \mathbf{y}(x) = \mathbf{f}(x, \mathbf{y}(x)) \quad (1.9)$$

que resulta práctica para obtener soluciones numéricas.

Un problema que involucra EDOs no está completamente determinado por sus ecuaciones, es necesario también especificar las condiciones de borde del problema. Las condiciones de borde son expresiones algebraicas para los valores  $y_i$  en la ecuación 1.7. En general pueden ser satisfechas en un conjunto discreto de puntos, pero no se satisfacen en todos los puntos intermedios. Las condiciones de borde pueden ser tan simple como requerir que ciertas variables tengan valores numéricos dados, o tan complicadas como un conjunto de ecuaciones algebraicas no lineales entre las variables.

Generalmente, la naturaleza de las condiciones de borde determina el método de solución. Se dividen en dos categorías principales:

- **Problemas de valores iniciales:** todos los  $y_i$  tienen valores establecidos en algún valor inicial  $x_i$ , y se requiere determinar los valores de  $y_i$  en algún valor final  $x_f$ , o en alguna lista discreta de puntos (por ejemplo, a intervalos equiespaciados).
- **Problemas de contorno de dos puntos:** en este caso las condiciones de borde se especifican en más de un valor de  $x$ . Típicamente, algunas condiciones se especifican para  $x_i$  y las restantes para  $x_f$ .

En este capítulo abordaremos la solución de problemas de valores iniciales, dejando los problemas de contorno para el capítulo ?? referido a ecuaciones en derivadas parciales.

## 1.2. Solución analítica de EDOs

En 1838, Pierre-François Verhulst propuso un modelo de crecimiento poblacional en el que la tasa de reproducción es proporcional tanto a la población existente como a la cantidad de recursos disponibles. Si llamamos  $P$  a la cantidad de individuos en la población en función del tiempo  $t$ , el modelo queda formalizado por la ecuación:

$$\frac{dP(t)}{dt} = rP(t) \left[ 1 - \frac{P(t)}{K} \right] = rP(t) - \frac{rP(t)^2}{K} \quad (1.10)$$

donde  $r$  define la tasa de crecimiento y  $K$  la capacidad de carga del sistema.

En este modelo, el crecimiento inicial para valores pequeños de  $P$  (omitimos en lo que sigue la dependencia explícita del tiempo) está dominado por el primer término  $rP$ , donde el valor de  $r$  representa el incremento proporcional a la población  $P$  por unidad de tiempo. A medida que la población crece, lo hace también el segundo término  $-rP^2/K$ , disminuyendo de este modo la velocidad de crecimiento, debido a que los miembros de la población compiten entre sí por los recursos disponibles tales como espacio o alimentos, que están representados por la constante  $K$ . Esta competencia entre dos términos de signo opuesto limita el crecimiento, que se detiene cuando  $dP/dt = 0$ .

Resolveremos la EDO representada por la ecuación 1.10 utilizando la función `dsolve` de SymPy, que es capaz de encontrar soluciones analíticas a un conjunto de EDOs elementales. Es necesario enfatizar aquí que la mayoría de las EDO no pueden resolverse en forma analítica. Es posible obtener dichas soluciones para EDOs de primer o segundo orden, o sistemas lineales de primer orden que contienen unas pocas funciones conocidas.

Utilizaremos un *Jupyter-notebook* para codificar la solución el problema y encontrar la solución utilizando SymPy. Para ello es necesario primero definir símbolos para las variables  $P$ ,  $t$ ,

$r$ ,  $K$  y  $P_0$ , siendo esta última la población inicial  $P_0 = P(0)$ ; y también necesitamos representar la función desconocida  $P(t)$ . Para ello en la celda 1 definimos tales cantidades, la función y almacenamos en `edo` la función diferencial dada por la ecuación 1.10.

CELL 01

```
import sympy

t, K, r, P0, C1 = sympy.symbols('t, K, r, P_0, C_1')
P = sympy.Function('P')
edo = P(t).diff(t) - r * P(t) * (1 - P(t)/K)
edo
```

---


$$-r \left(1 - \frac{P(t)}{K}\right) P(t) + \frac{d}{dt}P(t)$$

Junto con las demás variables, definimos también en la primera celda a la constante  $C_1$ , que resultará la constante de integración que se determina con la condición inicial.

A continuación, `sympy.odesolve` recibe la ecuación `ode` y la función  $P(t)$  como argumentos y devuelve la solución en la variable `edo_sol`.

CELL 02

```
edo_sol = sympy.odesolve(edo, P(t))
edo_sol
```

---


$$P(t) = \frac{K e^{C_1 K + r t}}{e^{C_1 K + r t} - 1}$$

En la celda 3 definimos las condiciones iniciales mediante un diccionario, y finalmente en la celda 4 reemplazamos estas condiciones iniciales en la solución guardada en `edo_sol`, lo que genera una expresión (que en este ejemplo guardamos en `C_eq`) que permite resolver la constante de integración  $C_1$  en función de la condición inicial  $P_0$  y de la capacidad de carga  $K$ .

CELL 03

```
ini_cond = {P(0): P0}
ini_cond
```

---

```
{P(0): P_0}
```

CELL 04

```
C_eq = edo_sol.subs(t,0).subs(ini_cond)
C_eq
```

---


$$P_0 = \frac{K e^{C_1 K}}{e^{C_1 K} - 1}$$

Con esta última expresión, podemos despejar  $C_1$  en función de  $P_0$  y  $K$ , con lo cual tenemos completamente resuelto el problema para la condición inicial dada. En este caso, es sencillo

determinar el valor de  $C_1$  utilizando álgebra elemental. Definiendo

$$A = e^{C_1 K}$$

el valor de esta constante resulta

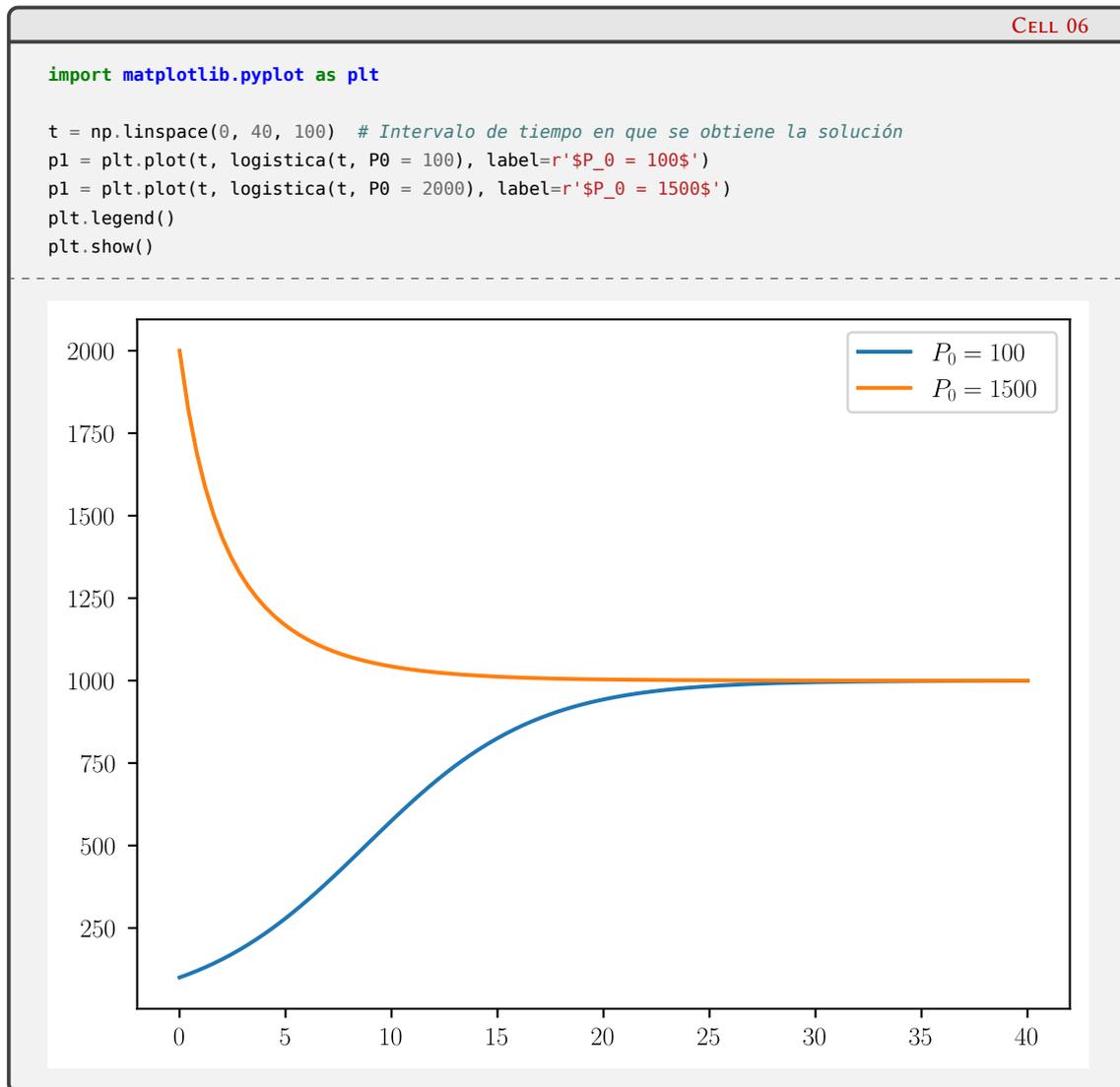
$$A = \frac{P_0}{P_0 - K}$$

En las últimas dos celdas del *notebook* implementamos la solución en la forma usual de las funciones en Python, para poder graficar soluciones según los valores de los parámetros. En particular, visualizamos dos soluciones del problema con valores de la población inicial  $P_0$  por debajo y por encima de la capacidad de carga  $K$  del sistema.

CELL 05

```
import numpy as np

def logistica(t, P0=100, K=1000, r=0.25):
    A = P0 / (P0 - K)
    return K / (1 - np.exp(-r*t) / A)
```



### 1.3. Métodos numéricos para resolver EDOs

Aunque existen problemas representados por EDOs que pueden resolverse analíticamente, por lo general es común que tal solución analítica no exista. Por lo tanto, en la práctica los problemas suelen ser resueltos utilizando métodos numéricos.

Existen muchos abordajes para resolver EDOs numéricamente, y muchos de ellos están diseñados para problemas formulados como un sistema de ecuaciones de primer orden en su forma estándar como se representa en la ecuación 1.9. SciPy ofrece funciones para resolver este tipo de problemas, que veremos en forma práctica más adelante, ya que primero haremos una revisión de los conceptos principales y de la terminología utilizada para la resolución numérica de EDOs.

El método de Euler constituye la base de muchos otros métodos para EDOs, y puede derivarse a partir de la expansión en serie de Taylor de  $y(x)$  alrededor de  $x$ :

$$y(x+h) = y(x) + \frac{dy(x)}{dx}h + \frac{1}{2} \frac{d^2y(x)}{dx^2}h^2 + O(h^3) \quad (1.11)$$

donde por simplicidad de la notación hemos considerado que  $y(x)$  es una función escalar. Si despreciamos términos de segundo orden y mayores, obtenemos la ecuación aproximada

$$y(x+h) \approx y(x) + f(x, y(x))h \quad (1.12)$$

que es precisa hasta el primer orden en el paso  $h$ . Esta ecuación se puede transformar en una fórmula iterativa discretizando la variable  $x$  en el intervalo de interés:  $x_0, x_1, \dots, x_n$ , eligiendo el paso  $h = x_{k+1} - x_k$  y denotando  $y_k = y(x_k)$ , lo que da origen a la fórmula del método de Euler “hacia adelante”:

$$y_{k+1} = y_k + hf(x_k, y_k) \quad (1.13)$$

que avanza una solución desde  $x_k$  hasta  $x_{k+1} \equiv x_k + h$ . Esta fórmula es asimétrica: avanza la solución sobre un intervalo  $h$  utilizando la información de la derivada solo al comienzo del mismo. La fórmula de Euler hacia adelante es una forma explícita debido a que dado el valor de  $y_k$  podemos calcular directamente el de  $y_{k+1}$ .

El objetivo de obtener la solución numérica de un problema con valores iniciales es el de calcular  $y(x)$  en un conjunto de puntos  $x_n$  dada la condición inicial  $Y(x_0) = y_0$ . La fórmula del método de Euler hacia adelante permite calcular los valores sucesivos de  $y_k$  comenzando en  $y_0$ . Sin embargo, este método no es recomendado para su uso práctico por dos razones: (a) el método no es preciso comparado con otros métodos más elegantes con el mismo paso, y (b) no es muy estable.

Una fórmula alternativa que puede derivarse en forma similar es el método de Euler “hacia atrás”, dado por la iteración

$$y_{k+1} = y_k + f(x_{k+1}, y_{k+1}) \quad (1.14)$$

que resulta en un método “implícito” debido a que  $y_{k+1}$  aparece en ambos lados de la ecuación y por lo tanto es necesario resolver una ecuación algebraica, que generalmente es no lineal. Los métodos implícitos resultan más complicados de implementar que los explícitos, y cada iteración requiere un mayor esfuerzo computacional. Sin embargo, tienen la ventaja de tener mejor precisión y una mayor región de estabilidad. La eficacia de cada abordaje (implícito o explícito) depende en definitiva de cada problema, aunque en términos generales los métodos implícitos son útiles en problemas “rígidos”, esto es, en los cuales existen dinámicas con tiempos característicos muy dispares (por ejemplo, dinámicas que incluyen oscilaciones rápidas y lentas).

Existen varios métodos que permiten mejorar los métodos de primer orden de Euler hacia adelante y hacia atrás. Por ejemplo, podemos mantener términos de mayor orden en la expansión en serie de Taylor dada por la ecuación 1.11 que brindan fórmulas iterativas de mayor orden, y que por lo tanto tienen mayor precisión. No obstante, estos métodos requieren evaluar derivadas superiores de  $y(x)$ , lo que puede ser un problema si  $f(x, y(x))$  no se conoce de antemano (y no está dada en forma simbólica).

Una forma de evitar este problema es el de aproximar las derivadas de orden superior utilizando diferencias finitas de las derivadas, o evaluando la función  $f(x, y(x))$  en puntos intermedios del intervalo  $[x_k, x_{k+1}]$ . Por lejos, el esquema más utilizado constituye la fórmula de Runge-Kutta

de cuarto orden:

$$\begin{aligned}
 k_1 &= h(f(x_k, y_k)) \\
 k_2 &= hf\left(x_k + \frac{h}{2}, y_k + \frac{k_1}{2}\right) \\
 k_3 &= hf\left(x_k + \frac{h}{2}, y_k + \frac{k_2}{2}\right) \\
 k_4 &= hf(x_k + h, y_k + k_3) \\
 y_{k+1} &= y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)
 \end{aligned} \tag{1.15}$$

Aquí se realizan cuatro evaluaciones diferentes de la función  $f(x, y(x))$  que son utilizadas en la fórmula explícita 1.15, que es precisa hasta el cuarto orden, con un error de quinto orden. Es posible estimar también el error en la aproximación mediante la combinación de dos métodos de órdenes diferentes. Una combinación muy utilizada es la de esquemas de Runge-Kutta de cuarto y quinto orden, que resulta en un método preciso de cuarto orden con estimación del error, conocido como el método RK45 o Runge-Kutta-Fehlberg. Otro ejemplo es el método Dormand-Prince, que utiliza un control de paso adaptativo y que está disponible en SciPy.

Un método alternativo consiste en usar más de un valor previo  $y_{n-k}$  para calcular  $y_n$ . Estos métodos se denominan multipaso y se pueden expresar en forma general como

$$y_n = \sum_{j=1}^k a_j y_{n-j} + h \sum_{j=0}^k b_j f(x_{n-j}, y_{n-j}) \tag{1.16}$$

De este modo, para calcular  $y_n$  se utilizan  $s$  valores previos de  $y_{n-j}$  y  $f(x_{n-j}, y_{n-j})$ . A partir de esta fórmula general se derivan diferentes esquemas según la elección de los coeficientes  $a_j$  y  $b_j$ . Por ejemplo, los métodos multipaso más importantes para problemas “no rígidos” son los del tipo Adams, que consisten en la elección de los coeficientes para la solución en  $x_n$  tomando  $a_1 = 1$  y  $a_2 = a_3 = \dots = a_k = 0$ . Aquí, las constantes  $b_j$  se elijen de forma tal que se obtiene el máximo orden posible. Debe notarse que si  $b_0 = 0$  el método es explícito y se conoce como métodos de Adams-Bashforth, mientras que si  $b_0 \neq 0$  los métodos resultan implícitos y se denominan Adams-Moulton. Por ejemplo, los métodos de un paso Adams-Bashforth y Adams-Moulton se reducen a los métodos de Euler hacia adelante y hacia atrás, respectivamente.

Los métodos de Adams se implementan, generalmente, en la forma “predictor-corrector”. En este esquema se realiza un cálculo preliminar usando la forma Bashforth del método (explícito). Esta solución aproximada en  $x_n$  se utiliza para evaluar una aproximación al valor de la derivada en este nuevo punto, que luego es utilizada en la fórmula de Moulton como valor inicial para resolver el método implícito.

Una cuestión que es necesaria considerar al utilizar métodos multipaso es la forma de inicializar el proceso. Para un método de  $k$ -pasos, el método ofrece un algoritmo para calcular  $y_k$  en términos de  $y_0, y_1, \dots, y_{k-1}$ , sin embargo se debe pensar en cómo obtener estos valores previos. Una posibilidad es evaluar los primeros  $k - 1$  valores usando una secuencia de métodos de orden bajo, sin embargo, esto introduce errores que anularían las ventajas de un uso posterior de métodos de mayor orden. También es posible utilizar un método de Runge-Kutta para los primeros  $k - 1$  pasos, siempre que el método de Runge-Kutta sea del mismo orden que el método de  $k$ -pasos utilizado posteriormente, de modo que no haya una pérdida de orden.

Una técnica simple que se utiliza para no utilizar métodos alternativos y al mismo tiempo mantener el orden durante el proceso de inicialización es el siguiente. Introducimos, como incógnitas a ser calculadas, aproximaciones a los valores de  $f(x_i, y_i)$  para  $i = -(k-1), -(k-2), \dots, -1$ . Los valores iniciales para estas cantidades se eligen como  $f(x_{i-1}, y_{i-1}) = f(x_0, y_0)$ . A partir de estos valores es posible calcular los valores de  $y_i$  y  $f(x_i, y_i)$  para  $i = 1, 2, \dots, k-1$ . Luego de esto invertimos el orden de la integración cambiando el signo de  $h$  y recalculamos los valores iniciales  $y_i$  y  $f(x_i, y_i)$  para  $i = -1, -2, \dots, -(k-1)$ . Este proceso de alternar integraciones hacia adelante y hacia atrás se repite hasta que se alcanza convergencia, y entonces hemos encontrado valores iniciales aceptables que permitirán avanzar en la integración calculando los pasos  $i = k, i = k+1, \dots$ .

Para finalizar esta sección, queremos mencionar que muchos métodos avanzados de solución de EDOs utilizan un control de paso, o paso adaptativo. La razón de esto es que la precisión y estabilidad de los métodos dependen fuertemente del paso de integración. Entonces, se presentan dos alternativas para ajustar el paso en cada iteración de modo de reducir el costo computacional manteniendo una precisión establecida. Una de estas alternativas surge cuando es posible obtener junto con  $y_n$  una estimación del error. De este modo es posible ajustar el tamaño de  $h_n$  para utilizar valores grandes cuando es posible, y pasos pequeños cuando sea necesario. La otra posibilidad es ajustar el orden del método de modo de utilizar un orden bajo (de menor costo computacional) cuando el error es pequeño, y un método de mayor orden cuando el error estimado no alcanza la precisión requerida con un método de orden menor. Para esta última alternativa, los métodos de Adams son apropiados ya que se pueden cambiar de orden fácilmente.

Existe una gran variedad de métodos para resolver numéricamente EDOs, y no es aconsejable implementar alguno de los métodos descritos en esta sección ya que para fines prácticos es posible utilizar muchas de las bibliotecas que están muy optimizadas y extensamente verificadas, disponibles en paquetes de código abierto y libre como SciPy. Por supuesto, para poder utilizar estas librerías es necesario conocer las ideas básicas y las metodologías para poder elegir correctamente el método apropiado para cada problema.

## 1.4. Solución de EDOs con valores iniciales

El módulo `integrate` de SciPy ofrece la función `integrate.solve_ivp` para resolver numéricamente un sistema de ecuaciones diferenciales con valores iniciales con diferentes métodos de integración: variantes de Runge-Kutta explícito de varios órdenes, un método implícito multipaso basado en una fórmula de diferenciación hacia atrás, y también una implementación de uno de los códigos más robustos y confiables, LSODA, que forma parte del paquete ODEPACK desarrollado en FORTRAN por el Lawrence Livermore National Laboratory<sup>2</sup>. Este integrador alterna automáticamente entre rutinas de integración para métodos rígidos (Adams predictor-corrector) y no rígidos (Gear, método de diferenciación hacia atrás).

Resolveremos la dinámica de un sistema compuesto por una masa suspendida mediante la combinación de un resorte y una banda elástica [3] tal como se esquematiza en la figura 1.1. En este sistema, una masa  $m$  está sujeta a un resorte vertical junta a una banda elástica que provee un sostén adicional. Mientras que el resorte produce una fuerza restauradora tanto en las direcciones ascendentes como descendentes, la banda elástica solo ejerce una fuerza hacia arriba

<sup>2</sup> Se puede ver más información sobre ODEPACK en su [web](#).

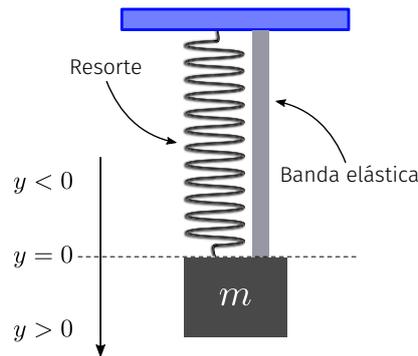


FIGURA 1.1: sistema masa - resorte - banda elástica representada por la ecuación 1.17.

al estirarse cuando la masa desciende. Llamemos  $y(t)$  al desplazamiento de  $m$  de su posición de equilibrio  $y = 0$  al tiempo  $t$ .

Tres fuerzas actúan sobre la masa: la fuerza restauradora de la banda elástica, la fuerza lineal restauradora del resorte, y el peso. Sin la banda elástica la fuerza restauradora del resorte (en el sistema de referencia indicado en la figura) está dada por  $-k_r y$ . La banda elástica aporta una fuerza restauradora  $-k_b y^+$  debido a que solo actúa cuando la masa desciende ( $y > 0$ ), pero no en compresión ( $y < 0$ ). Por lo tanto, la fuerza restauradora combinada resultante es  $-ay^+ + by^-$ , donde  $a > b$  y  $y^+ = y$  si  $y > 0$ , mientras que es 0 si  $y < 0$ , y  $y^- = -y$  si  $y < 0$  y 0 en otro caso.

Queremos estudiar la respuesta de este sistema al someterlo a una pequeña fuerza periódica  $f(t) = \lambda \sin(\mu t)$ , donde  $\lambda$  es la amplitud de la fuerza y  $\mu$  es la frecuencia angular. Combinando estas fuerzas y añadiendo un pequeño término de amortiguación viscosa (proporcional a la velocidad de la masa) el siguiente modelo:

$$\ddot{y} + 0,01\dot{y} + ay^+ - by^- = 10 + \lambda \sin(\mu t) \quad (1.17)$$

donde como es usual, la derivada de  $y$  respecto del tiempo se denota  $\dot{y}$ , y

$$y^+ = \begin{cases} y, & y \geq 0 \\ 0, & y < 0 \end{cases} \quad y^- = \begin{cases} 0, & y \geq 0 \\ -y, & y < 0 \end{cases} \quad (1.18)$$

o simplemente  $y^+ = \max(y, 0)$  y  $y^- = \max(-y, 0)$ . Para algunos valores de los parámetros  $a$ ,  $b$ ,  $\lambda$  y  $\mu$ , el sistema muestra un comportamiento senoidal, pero para otros, la no linealidad que aporta la banda elástica genera un comportamiento caótico. El código siguiente integra numéricamente este sistema.

```

1 #!/usr/bin/env python3
2 """Programa que integra el sistema masa-resorte-banda elástica."""
3
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from scipy.integrate import solve_ivp
7
8 # Configure matplotlib
9 plt.rcParams.update({
10     'text.usetex': True,

```

```

11     'font.size': 14,
12     'axes.labelsize': 'large',
13 })
14
15
16 def pos(x):
17     """Devuelve x si x > 0, 0 en otro caso."""
18     return max(x, 0)
19
20
21 def neg(x):
22     """Devuelve -x si x < 0, 0 en otro caso."""
23     return max(-x, 0)
24
25
26 # Parámetros del sistema
27 a, b = 17, 1 # Combinación de constantes elásticas
28 λ = 15.4 # Amplitud de la excitación periódica
29 μ = 0.75 # Frecuencia angular de la excitación periódica
30
31
32 def sistema(t, z):
33     """Definición del sistema de ecuaciones diferenciales de primer orden.
34
35     Argumentos:
36     t: escalar que representa el tiempo
37     z: lista con [y, dy/dt]
38
39     Devuelve:
40     [dy(t)/dt, d^2y(t)/dt^2]
41     """
42     y, yp = z # yp = dy/dt
43     return [yp, 10 + λ * np.sin(μ * t) - 0.01 * yp - a * pos(y) + b * neg(y)]
44
45
46 # Array con la discretización del intervalo de tiempo en el que se calcula
47 # la solución del sistema de EDOs
48 t = np.linspace(0, 100, 10000)
49
50 # Solución del sistema de EDOs
51 sol = solve_ivp(sistema, [0, 100], [1, 0], t_eval=t)
52
53 # Gráfico de la solución y del diagrama de fase
54 fig, (ax0, ax1) = plt.subplots(1, 2)
55 ax0.plot(sol.t, sol.y[0])
56 ax0.set_ylim(8, -19)
57 ax0.set_xlabel(r'$t$')
58 ax0.set_ylabel(r'$y$')
59 ax0.text(0.05, 0.9, 'a)', transform=ax0.transAxes, fontsize=18)
60 ax1.plot(sol.y[0], sol.y[1])
61 ax1.set_xlabel(r'$y$')
62 ax1.set_ylabel(r"$dy/dt$")
63 ax1.text(0.05, 0.9, 'b)', transform=ax1.transAxes, fontsize=18)
64 plt.show()

```

Importamos el método `solve_ivp` de `scipy.integrate` en la línea 7. Las funciones 1.18 están definidas en las líneas 11 y 15, respectivamente, mientras que los parámetros del sistema para

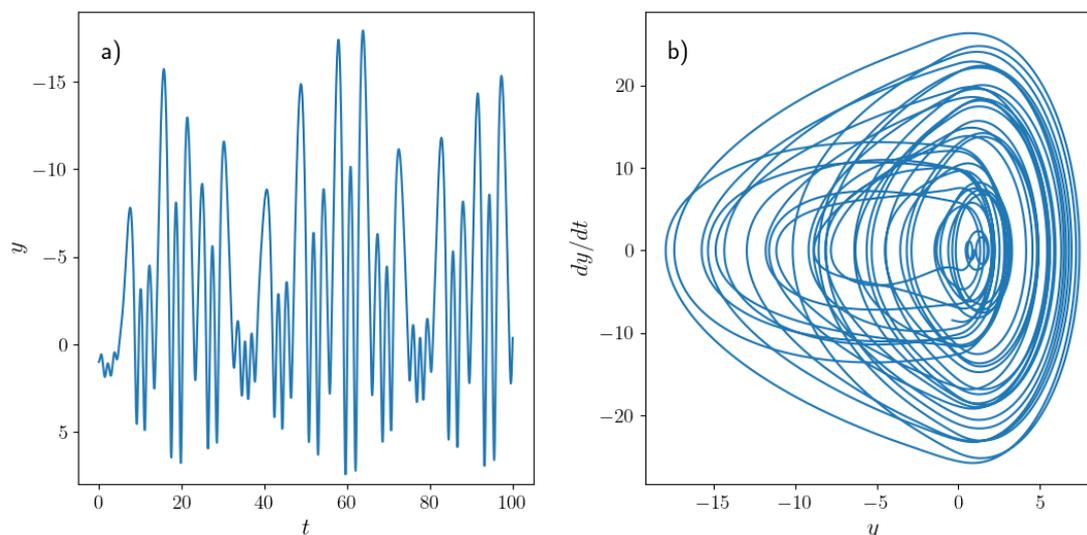


FIGURA 1.2: solución de la ecuación 1.17 En el panel a) se muestra la trayectoria de la masa  $m$ , mientras que en el b) se visualiza el espacio de fase del sistema.

una integración particular están definidos en las líneas 20–22.

Dado que el sistema representado en la ecuación 1.17 es de segundo orden, es necesario primero convertirlo en un sistema de dos ecuaciones de primer orden. Esto se realiza en la función `sistema()`, que recibe como argumentos un escalar  $t$  que representa a la variable tiempo  $t$ , y una lista con los valores de  $y$  y  $\dot{y}$ . La función devuelve entonces una lista con los valores calculados para  $\dot{y}$  y  $\ddot{y}$ , siguiendo el mismo esquema que mostramos en las ecuaciones 1.8 y 1.9.

La solución del sistema se calcula en la línea 39, en la que usamos el método `solve_ivp` al que se le pasa como argumentos la función que representa al sistema de EDOs (`sistema()`), el intervalo de integración en forma de lista con el valor inicial y final  $[t_0, t_f]$ , que en nuestro caso es  $[0, 100]$ , una lista con los valores iniciales  $y(0) = 1$  y  $\dot{y}(0) = 0$ , y finalmente un argumento con nombre (`t_eval`) que contiene un array con los instantes de tiempo en los que guardamos los valores de la solución calculada. En este ejemplo usamos el array definido en la línea 36. Es requisito que los valores del array se encuentren dentro del intervalo definido en el intervalo de integración.

El método `solve_ivp` devuelve un objeto que contiene numerosos campos con la solución del sistema de EDOs e información sobre el proceso de integración, y que nosotros guardamos en `sol`. En este ejemplo, solamente utilizaremos los atributos `t` y `y` de `sol` que contienen el array de valores de tiempo donde se calculó la solución, y una lista con los valores calculados de  $y$  y  $\dot{y}$  correspondientes, respectivamente, con el propósito de visualizar la trayectoria y el espacio de fase, tal como se muestra en la figura 1.2.

El método `solve_ivp` admite la selección de diversos métodos de integración. Por defecto utiliza RK45, como hicimos en el ejemplo, pero se pueden pasar con el argumento `method` los métodos RK23, DOP853, Radau, BDF y LSODA, tal como se indica en la [documentación](#).

## 1.5. Lecturas recomendadas

- John Charles Butcher. *Numerical methods for ordinary differential equations*. Chichester, England Hoboken, NJ: Wiley, 2008. Muy buena revisión de la teoría básica de ecuaciones diferenciales y en diferencias, y un tratamiento extenso de varios métodos numéricos (Euler, Runge-Kutta, métodos multipaso, etc.).
- William H Press et al. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge, UK New York: Cambridge University Press, 2007. Este libro es una de las referencias ineludibles en el arte de la programación de métodos numéricos, y ofrece recursos valiosos para la implementación práctica de algoritmos.

**Parte IV**  
**Apéndices**

## A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [6].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tenemos más de esos!

## Bibliografía

- [1] URL: <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.
- [2] Richard P. Feynman. *The Feynman Lectures on Physics, boxed set: The New Millennium Edition*. Basic Books, ene. de 2011. URL: <https://feynmanlectures.caltech.edu/info/>.
- [3] L D. Humphreys y R. Shammass. «Finding Unpredictable Behavior in a Simple Ordinary Differential Equation». En: *The College Mathematics Journal* 31.5 (2000), págs. 338-346. DOI: [10.1080/07468342.2000.11974171](https://doi.org/10.1080/07468342.2000.11974171). eprint: <https://doi.org/10.1080/07468342.2000.11974171>. URL: <https://doi.org/10.1080/07468342.2000.11974171>.
- [4] John Charles Butcher. *Numerical methods for ordinary differential equations*. Chichester, England Hoboken, NJ: Wiley, 2008.
- [5] William H Press, Saul A Teukolsky, William T Vetterling y Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge, UK New York: Cambridge University Press, 2007.
- [6] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.