Introducción al cómputo científico con PYTHON

Roxana Mitzayé del Castillo Vázquez

TEMAS DE COMPUTACIÓN

las prensas de ciencias

Roxana Mitzayé del Castillo

Introducción al cómputo científico con Python

Facultad de Ciencias
2021

las prensas de ciencias

Esta obra contó con el apoyo del proyecto PAPIME PE100120

Introducción al cómputo científico con Python
1ª edición, 26 de febrero de 2021

Facultad de Ciencias
Ciudad Universitaria, Delegación Coyoacán.
C.P. 04510. Ciudad de México
Coordinación de servicios editoriales: editoriales@ciencias.unam.mx
Plaza Prometeo: tienda.fciencias.unam.mx

ISBN: 978-607-30-4349-6

Diseño de portada: Eliete Martín del Campo Treviño
Apoyo en la formación en LaTeX: Raúl Espejel Morales

Impreso y hecho en México

A mis padres:

"Del mismo modo que tesoros se descubren de la tierra,
la virtud se aparece de las buenas acciones
y la sabiduría aparece de una mente pura y pacífica.
Para caminar con seguridad a través del laberinto de la vida humana,
uno necesita la luz de la sabiduría y la guía de la virtud".

Buda

Titi, mi virtud. Luis Felipe, mi sabiduría.

A Frank:

Mi cable a tierra.
"Hang the DJ".

Agradezco por su apoyo académico y editorial al Dr. Raúl Arturo Espejel ya que este libro se ha logrado gracias a su apoyo incondicional, su experiencia y arduo trabajo. Así mismo, agradezco al Dr. Alipio Calles por su mentoría y su perspectiva sagaz sobre el cómputo científico, de la que he aprendido mucho. También agradezco a la Dra. Karla Paola García Pelagio por su trabajo editorial y a los árbitros, que con sus buenos consejos, ayudaron a mejorar el libro.

En la redacción de este libro se han tomado en cuenta las sugerencias y comentarios de mis queridos alumnos durante los 12 años que he impartido la asignatura de Computación en la Facultad de Ciencias de la UNAM, a quienes agradezco profundamente. De manera especial, agradezco a Jordi Salinas, Diego de la Mora, Anna Flor Cadena y Jessica Söhle Góngora, quienes han proporcionado un invaluable apoyo en la revisión del presente libro.

Este libro fue posible gracias al apoyo de la Dirección General de Asuntos del Personal Académico a través del proyecto **PAPIME PE100120**.

Roxana Mitzayé del Castillo Vázquez

Contenido

PRÓLOGO

El contenido del libro que nos presenta Roxana del Castillo, sobre *Introducción al cómputo científico con Python*, resulta de gran interés. Es una manera fresca y rápida de aprender algo de historia del cómputo científico sin desviarse del objetivo principal del libro, el uso de Python. Sus recuentos históricos se introducen de manera oportuna y contribuyen a la cultura del lector, ya sea un estudiante de las carreras de ingeniería, física o matemáticas. Nos recuerda la aportación de brillantes mexicanos especialistas en computación que pasaron por la facultad de Ciencias de la Universidad Nacional Autónoma de México y que ahora se encuentran en países vanguardias en el tema, aportando y desarrollando conceptos para el mundo informático. El lenguaje del libro hace que el lector se sienta muy cómodo y tenga deseos de continuar el aprendizaje. Como libro de consulta también es muy útil; la autora describe las librerías de Python de una manera muy práctica. El lector, lejos de leer un libro rígido y acartonado, siente que está en contacto y platicando con Roxana. Explica los temas y materiales que forman al Python con ejemplos y programas listos para su ejecución. Otro acierto del libro reside en que cada capítulo lo culmina con ejercicios de matemáticas y física muy amenos que refuerzan la cultura en la que Roxana quiere inmiscuir al lector. En muchos de sus ejemplos, ella es la protagonista.

El capítulo de visualización es absolutamente esencial; para transmitir los resultados de la solución de un problema, la visualización y simulación en computadora son un auxiliar didáctico que, además sirven como instrumentos para prever y evaluar los posibles efectos de un un fenómeno antes de que éste suceda; diseño de medicinas, predicción del tiempo, dispersión de contaminantes, estimación de contagiados en una pandemia, ensayos de aterrizajes en otros planetas para corregir posibles errores, diseño de aviones, etc.

La autora no solo enseña a programar en Python, sino que también nos advierte que hay que tener mucho cuidado con el error numérico para que el CC sea confiable al introducir el apéndice sobre el Manejo de datos numéricos y experimentales.

En caso de que el lector necesite un resumen o repaso de UNIX, la autora nos presenta un capítulo con los elementos esenciales (estructura, kernels, directorios, comandos más importantes, principales editores de texto, entre otros muchos conceptos) de uno de los sistemas operativos especializados para computadoras personales más importante, el LINUX.

En su primer capítulo de introducción, hace ver al lector la importancia de la computación científica (CC) para el desarrollo, no solo de la tecnología, sino también de las ciencias exactas y sociales; está implícito que el desarrollo de la ciencia básica también resulta impactada y necesita de la CC. Baste recordar a uno de los físicos más destacados del siglo XX, Richard Feynman (considerado dentro de los 10 científicos más importantes de todos los tiempos), quien, a sus 22 años, fue el líder del grupo de cómputo de IBM para calcular la energía liberada en la explosión nuclear en el proyecto Manhatan. Feynman dedicó los últimos años de su trayectoria académica a la enseñanza del CC en el Instituto de Tecnología de California; sus alumnos publicaron el libro *The Feynman lectures on computation* en 1996.

Para todos los que admiramos a Issac Newton, constituye un privilegio empezar los cursos de introducción a los métodos numéricos con su legado para resolver problemas de dinámica y métodos numéricos y así obtener las raíces de funciones analíticas. Es uno de los primeros matemáticos de su época, 1697, en resolver la trayectoria óptima en el tiempo para una partícula que se mueve entre dos puntos arbitrarios en el campo gravitacional con aceleración constante; Newton obtiene, con diferencias finitas, la cicloide como solución del problema.

En la búsqueda y comprobación de las teorías fundamentales para el entendimiento del origen de la materia, como el bosón de Higgs, se necesita de un acelerador de partículas como el LHC. El diseño, construcción, operación y análisis de este tipo de equipo nos sería prácticamente imposible sin el CC.

En actividades tan relevantes como la exploración de nuestro sistema solar, la búsqueda de vida en otros planetas, la respuesta a nuestro origen, el reciente aterrizaje en el planeta Marte, ya no se diga en la Luna y en asteroides, el CC es fundamental junto con otras disciplinas de la ingeniería y química. En todos los juegos de guerra necesarios para mantener el equilibrio entre las potencias armamentistas del mundo, el CC juega un papel primordial.

El hecho de que sea una autora la que llame nuestra atención sobre la importancia del cómputo científico con este libro es un magnífico ejemplo del papel que la mujer está jugando en la comunidad científica y en nuestra sociedad. Gracias Rox.

Alipio G. Calles

Capítulo 1

Introducción a Linux

Las computadoras de la actualidad han sido diseñadas de tal forma que puedan realizar diversas tareas simultáneamente y tener múltiples usuarios. Para su correcto funcionamiento deben ser capaces de acceder ordenadamente a los recursos que disponen -por mencionar algunos, escribir o leer datos en un disco duro, mostrar una gráfica en la pantalla, escuchar música, navegar en Internet- ya que si cada programa actuase de forma independiente, podría escribir sus datos sobre los de otro programa conllevando a un bajo desempeño de la computadora. Todas estas tareas no deben sobreponerse entre sí, si a caso, deberán trabajar en paralelo sin afectar su funcionamiento. Los Sistemas Operativos fueron desarrollados para solucionar este tipo de problemas aportando reglas básicas para el funcionamiento eficiente y ordenado de las computadoras.

Un **sistema operativo (SO)** es un programa administrador de recursos que toma la forma de un conjunto de rutinas de software para permitir a los usuarios y aplicaciones acceder a los recursos del sistema de manera ordenada, segura, eficiente y abstracta.

El objetivo principal de los sistemas operativos es hacer eficiente el uso del CPU[1], por lo que una de sus tareas principales es suspender algunos programas que esperan alguna operación para ser completados o no están siendo usados por el usuario. Un sistema operativo también provee una abstracción

[1]La unidad de procesamiento central, conocida por las siglas CPU, del inglés *Central Processing Unit*, es la parte física de la computadora que interpreta las instrucciones. En específico, guarda al sistema operativo y la información del usuario.

conveniente, como la localización de los archivos en el disco que usan los programadores para detallar el uso del hardware, asimismo permite al usuario construir rápida y fácilmente sus propias herramientas para automatizar procesos. Debido a esto, es conveniente que el usuario identifique el tipo de trabajo que desarrollará para escoger el tipo de sistema operativo que necesita.

Los tres sistemas operativos más importantes son; Linux, MacOs y Windows. De estos se derivan una gran cantidad de variaciones[2]. En el que nos enfocaremos será en **Linux**, ya que es uno de los sistemas operativos más poderoso, rápido y con muchas versiones disponibles.

1.1. Historia de Linux

Linux es un sistema operativo derivado de Unix, un sistema multitarea y multiusuario[3], creado en 1969 por Thompson y Ritchie en los laboratorios Bell de AT&T. Inicialmente fue distribuido gratuitamente en las universidades, donde tuvo mucha aceptación.

Unix ha sido desarrollado con la intención de ser colaborativo, hacer accesible, enriquecedora y dinámica la programación. Logró el objetivo para el que fue creado, según las palabras del mismo Ritchie *"Lo que queríamos preservar no era sólo un buen ambiente en el cual programar, sino también un sistema alrededor del cual se formara fraternidad"* [1], [2], con esta mentalidad se ha creado una gran comunidad que ha desarrollado uno de los mejores sistemas operativos.

En 1984, Richard Stallman fundó el proyecto GNU[4] con el objeto de que el conocimiento computacional fuese eficientemente difundido a la sociedad, por lo cual definió el concepto de *Software Libre*. Este proyecto tuvo como tarea inicial desarrollar un sistema operativo libre y abierto. Lo primero que propuso Stallman fue un sistema operativo tal que los usuarios pudiesen usarlo, leer el código fuente, modificarlo y redistribuirlo; luego generalizó la idea a cualquier software. Para llevar a cabo esto, se han promovido estándares de

[2] Con los smartphones, tablets, PC, ebooks-readers, ipads, etc, han emergido nuevos sistemas operativos SO básicos y sencillos como Android, basado en Linux, y IOs, basado en MacOs.

[3] En la actualidad todos los sistemas operativos cumplen con ser multitareas y multiusuarios. El primero en cumplir con estas características fue Unix.

[4] GNU es el acrónimo recursivo de *"GNU's Not Unix"*(GNU no es Unix) nombre elegido debido a que GNU sigue un diseño tipo Unix y se mantiene compatible con éste, pero es software libre y no contiene código de Unix.

programación de códigos abiertos y comunidades computacionales con el fin de mantener estos estándares que definen lo que es hoy el proyecto GNU.

La filosofía GNU apoya el crecimiento de la sociedad como un conjunto, haciendo especial énfasis en la valoración de las libertades personales, impulsando así, la generación de conocimiento dentro del dominio público; que se encuentre al alcance de todos aquellos que estén interesados en su mejora, por encima de intereses particulares y de apropiación del conocimiento por medio de derechos de propiedad intelectual.

En 1991, Linus Torvalds tenía como pasatiempo compilar y modificar un sistema operativo conocido en esa época, MINIX, el cual permitía a sus usuarios leer el código fuente. Al empezar con las modificaciones, entabló mucha comunicación con otros programadores con el mismo interés. A raíz de esta comunicación se dio cuenta que muchas personas veían las mismas desventajas que él veía en MINIX, por lo que decidió hacer otro sistema operativo, más completo y mejor diseñado. En septiembre de ese año, liberó el kernel[5], dando inicio al sistema GNU/LINUX.

Los sistemas tipo Unix, en específico Linux, han sido empleados tradicionalmente por programadores, por esto cuenta con compiladores, editores y herramientas para facilitar la programación, especialmente en lenguaje C. En el caso de Linux tales herramientas han sido desarrolladas por la Free Software Foundation (FSF)[6]. Así mismo la FSF aportó la licencia que cubre al kernel, llamada GPL, y muchos componentes del sistema Linux.

Linux cuenta con un entorno gráfico que, a diferencia de otros sistemas operativos, el usuario puede poner o quitar. La parte visual del sistema de ventanas se llama X-Window y la versión particular que se emplea en Linux se llama XFree86[7].

1.2. Software libre

El software libre se obtiene de forma gratuita en la mayoría de las veces, sin embargo, se debe tener presente que dicha libertad se refiere a la autorización que otorga el desarrollador al usuario para utilizarlo, modificarlo y distribuirlo.

[5]El núcleo del sistema operativo, es el programa principal, en la siguiente sección se describe a profundidad.

[6]Para mayor información se sugiere consultar http://www.fsf.org.

[7]Para mayor información se sugiere consultar http://www.xfree86.org.

Al hablar de software libre se suelen clasificar los distintos grados de libertad a los que pueden tener acceso los usuarios.

- **Libertad grado 0**

  El software se puede **usar**. Es la libertad que otorga casi cualquier software.

- **Libertad grado 1**
  El software se puede **modificar**. Es decir, se puede personalizar, mejorar, adaptar para las necesidades particulares de un determinado usuario.

- **Libertad grado 2**
  El software se puede **distribuir**. Se puede copiar, vender, prestar o compartir a las personas que el usuario desee sin tener que pedir permiso al autor del software.

- **Libertad grado 3**
  El software se puede **distribuir modificado**. Se trata de la suma de 1 y 2. Permite que un usuario haga las modificaciones que quiera al software y, además, pueda compartirlo con otros usuarios.

En general, las ideas del software libre buscan promover la creación de mejor software, por medio de la suma de los pequeños aportes de cada persona y colaborar para que toda la sociedad se vea beneficiada con los avances del mismo, logrando mejorar la sociedad al tener disponibles más y mejores herramientas[8].

1.3. Arquitectura del sistema operativo Linux

Linux cumple con todos los componentes de un sistema operativo típico, el cual es herencia de los sistemas operativos Unix de antaño. A continuación se citan las componentes:

[8]Para más información sobre el software libre puede visitar el sitio de internet del proyecto GNU, http://www.gnu.org, que tiene una gran cantidad de documentos relacionados con la filosofía del software libre.

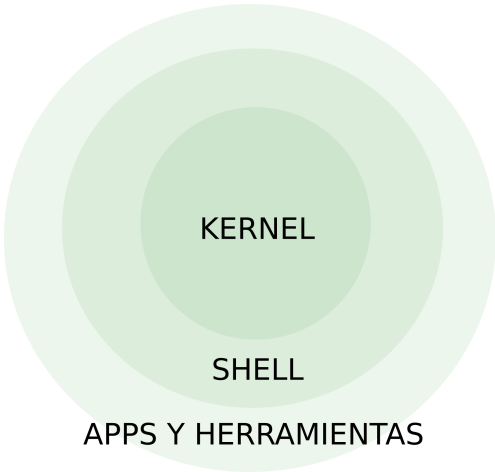


Figura 1.1: Orden jerárquico de los componentes de un sistema operativo tipo Linux. En general, todos los sistemas operativos modernos siguen esta estructura básica.

Kernel: Aplicación encargada de comunicar los procesos con el hardware de la computadoras. Parte fundamental de un sistema operativo. Es el software responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora o, en forma más básica, es el encargado de gestionar recursos a través de servicios de llamada al sistema.

El **kernel** de Linux incluye controladores de los dispositivos de un gran número de computadoras (tarjetas gráficas, tarjetas de red, discos duros, etc). También contiene aplicaciones para el manejo del procesador y de la memoria, y da soporte a diferentes tipos de archivos de sistemas.

El **kernel** se encuentra en código binario puro. Comúnmente, se puede localizar en el archivo `/boot/vmlinuz`, los archivos fuentes se encuentran en `/usr/src/linux`[9].

Shell: El término **shell** se emplea para referirse a programas que proporcionan una interfaz de usuario para dar ordenes al sistema. Están diseñados para facilitar la forma en que se invocan o ejecutan los distintos programas disponibles. Linux soporta dos formas de introducir comandos, la forma

[9]La última versión de las fuentes del kernel de Linux se pueden ver en `http://www.kernel.org`.

textual, línea de comando en la terminal, y la forma gráfica, usando el mouse, por medio del entorno de escritorio GNOME o KDE. Si se está conectado a un escritorio remoto, la manera más eficiente será usar las líneas de comando.

Utilidades del sistema: Un ejemplo de utilidades del sistema son los comandos como `ls`, `cp`, `grep`, `awk`, `sed`, `bc`, `wc`, `more` y muchos otros. Las utilidades del sistema son herramientas muy poderosas que realicen una tarea de manera especifica; algunos de los comandos más usados son `ls`, `cp`, `grep`, `awk`, `more`, entre otros. Por ejemplo, `grep` encuentra texto dentro de un archivo mientras `wc` cuenta el número de palabras, líneas y bytes dentro de un archivo. Los usuarios pueden resolver algunos problemas por la interconexión de estas herramientas.

Una de las utilidades más importantes de los sistemas operativos, son los programas de servicio llamados *Daemons*[10]. Los *Daemons* son programas que hacen un proceso informático ejecutado en segundo plano. Su finalidad principal es ayudar a que el programa ejecutado en primer plano funcione correctamente. Estos programas no pueden ser manipulados por el usuario, se ejecutan de forma continua y, al intentar cerrar un proceso derivado de un *Daemon*, este continuará en ejecución o se reiniciará automáticamente.

Algunos ejemplos de *Daemons* serían aquellos que proporcionan conexiones a redes remotas y administran servicios; por ejemplo: *telnetd* y *sshd* provee facilidades para ingresar de manera remota, *lpd* proveen servicios de impresión, *httpd* sirve para páginas web, etc.

Programas de aplicaciones: Las distribuciones de Linux típicamente presentan varios programas de aplicaciones bastante útiles que vienen incorporados de manera estándar con Linux. Como ejemplo, está el editor de texto Gedit o Vi, entornos de escritorio, juegos o entornos de oficina como LibreOffice. En la actualidad, Linux permite instalar y desinstalar los programas de aplicaciones con bastante facilidad.

[10]La palabra *Daemon* deriva de la palabra en latín que se refiere a un espíritu benefactor que cuida a alguien [3], por lo que los *Daemon* son procesos que cuidan el funcionamiento de la computadora.

1.4. Entornos de escritorio

Se puede decir que un entorno de escritorio es un conjunto de software para ofrecer al usuario una interacción amigable y cómoda con la computadora. Este tipo de software es una solución completa de interfaz gráfica de usuario. Ofrece iconos, ventanas, fondos de pantallas, widgets, barras de herramientas, programas e integración entre aplicaciones.

En general, cada entorno de escritorio se distingue por su aspecto y comportamiento particulares. El primer entorno moderno de escritorio que se comercializó fue desarrollado por Xerox en los años 80.

En Windows, el entorno de escritorio es único y viene incorporado al sistema operativo. En el caso de Unix, hay varios proyectos de entornos de escritorio desarrollados para facilitar su adopción por usuarios sin experiencia; los más conocidos son GNOME[11] y KDE[12], que funcionan sobre Linux y X-Window. Cada uno brinda un entorno de escritorio gráfico, con uso del mouse, y aplicaciones de oficina como a las que están acostumbrados los usuarios de otros sistemas operativos, e.g. procesador de texto, hoja de cálculo, etc.

1.4.1. GNOME

La historia de GNOME (GNU Network Object Model Environment) es sumamente interesante. Éste surgió en agosto de 1997 como un proyecto liderado por los mexicanos Miguel de Icaza y Federico Mena[13] para crear un entorno de escritorio completamente libre para sistemas operativos libres, en especial para GNU/Linux. En esos momentos existía otro proyecto anterior con los mismos objetivos, pero con diferentes medios: KDE. Los primeros desarrolladores de GNOME criticaban a dicho proyecto por basarse en la biblioteca de controles gráficos, cuyas licencias no eran compatibles con GNU.

El proyecto GNOME, según sus creadores, provee un gestor de ventanas "intuitivo y atractivo". Se basa en una plataforma de desarrollo de aplicaciones que se integran con el escritorio. El proyecto pone un gran énfasis en la simplicidad, amabilidad y en hacer que las cosas funcionen. Los objetivos principales del proyecto son:

[11] Para más información véase http://www.gnome.org.

[12] Para más información véase http://www.kde.org.

[13] Miguel de Icaza y Federico Mena se conocieron en la Facultad de Ciencias de la Unam cuando estudiaban la carrera y fue ahí en donde desarrollaron su proyecto GNOME.

1. Libertad para crear un entorno de escritorio con el código fuente disponible, todo esto bajo la licencia de software libre.

2. El aseguramiento de accesibilidad, cualquier persona puede utilizarlo sin importar sus conocimientos técnicos y capacidades físicas.

3. Disponibilidad en muchos idiomas. Hasta el momento ha sido traducido a cien idiomas[14].

El escritorio GNOME es bastante configurable y ha tenido mucha popularidad. Actualmente, existen versiones de GNOME para Windows y variantes más simplificadas como UNITY[15].

1.4.2. KDE

Entorno de escritorio e infraestructura de desarrollo para sistemas Unix/ Linux. De acuerdo con su página web KDE.org es un entorno de escritorio contemporáneo para estaciones de trabajo en Unix. KDE llena las necesidades de un escritorio amigable para estaciones de trabajo Unix.

KDE se basa en el principio de personalizar el entorno. Todos los componentes de KDE pueden ser configurados en mayor o menor medida por el usuario. Las opciones más comunes son accesibles, en su mayoría, desde menús y diálogos de configuración. Los usuarios avanzados pueden optar por editar los archivos de configuración manualmente, obteniendo en algunos casos un mayor control sobre el comportamiento del sistema.

La intención del proyecto KDE es la de crear un entorno de escritorio que no se comporte de un modo predefinido, sino que permita al usuario adecuar el sistema a su gusto y comodidad. Esto no impide que KDE resulte fácil de usar para nuevos usuarios.

El proyecto fue iniciado en octubre de 1996 por el programador alemán Matthias Ettrich, quien buscaba crear una interfaz gráfica unificada para sistemas Unix.

[14] http://www.gnome.org.
[15] UNITY es una versión compacta y muy popular de GNOME.

1.5. Estructura del sistema de archivos Linux

En cualquier sistema operativo moderno la estructura de archivos es jerárquica y depende de los directorios. En general, la estructura del sistema de archivos se asemeja a una estructura de árbol, estando compuesto cada nodo por un directorio o carpeta que contiene otros directorios o archivos. En Windows, cada unidad de disco se identifica como una carpeta básica que sirve de raíz a otras y cuyo nombre es especial: **c:**, **d:**, etc. En los sistemas Linux, existe una única raíz llamada `/` o `root` de la que cuelgan todos los ficheros y directorios y que es independiente de que dispositivo esté conectado a la computadora.

La ruta o *path* es la secuencia de directorios que se ha de recorrer para acceder a un determinado directorio separado por "/". Existen dos formas para acceder a la misma ruta:

- La ruta absoluta que muestra toda la ruta a un fichero: `/home/roxana/Documentos/Carta.txt`
- La ruta relativa a un determinado directorio. Ejemplo. Si estamos en el directorio `/home`, vamos al directorio `roxana`, luego al directorio `Documentos` y por último abrimos el archivo `Carta.txt`:

```
 /
|
+-home/
| |
| +-roxana/
| | |
| | +-Documentos/
| | | |
| | | carta.txt
```

Bajo el directorio `root` se encuentran los ficheros a los que puede acceder el sistema operativo. Estos ficheros se organizan en distintos directorios cuya misión y nombre son estándar para todos los sistemas Linux.

- **/**. Raíz del sistema de archivos.
- **/dev**. Contiene directorios del sistema representando los dispositivos que estén físicamente instalados en el ordenador (Hardware).

- **/etc**. Este directorio está reservado para las carpetas de configuración del sistema. En este directorio no debe aparecer ningún archivo binario (programas). Bajo éste, deben aparecer otros dos subdirectorios:

  /etc/X11. Directorios de configuración de X-Windows.

  /etc/skel. Directorios de configuración básica que son copiados al directorio del usuario cuando se crea uno nuevo, es decir, carpetas compartidas.

- **/lib**. Contiene las librerías necesarias para que se ejecuten los programas que residen en **/bin**. No contiene las librerías de los programas de los usuarios.

- **/proc**. Contiene archivos especiales que pueden recibir o enviar información al **kernel** del sistema. Es recomendable no modificar el contenido de este directorio y sus archivos.

- **/sbin**. Contiene programas que son únicamente accesibles al superusuario **root**.

- **/usr**. Este es uno de los directorios más importantes del sistema, puesto que contiene los programas de uso común para todos los usuarios. Su estructura suele ser:

  - **/usr/bin**. Programas de uso general del sistema.
  - **/usr/doc**. Documentación general del sistema.
  - **/usr/etc**. Archivos de configuración general.
  - **/usr/include**. Archivos de cabecera de C/C++ (.h).
  - **/usr/info**. Archivos de información de GNU.
  - **/usr/lib**. Librerías generales de los programas.
  - **/usr/man**. Manuales accesibles con el comando **man**.
  - **/usr/sbin**. Programas de administración del sistema.
  - **/usr/src**. Código fuente de programas.

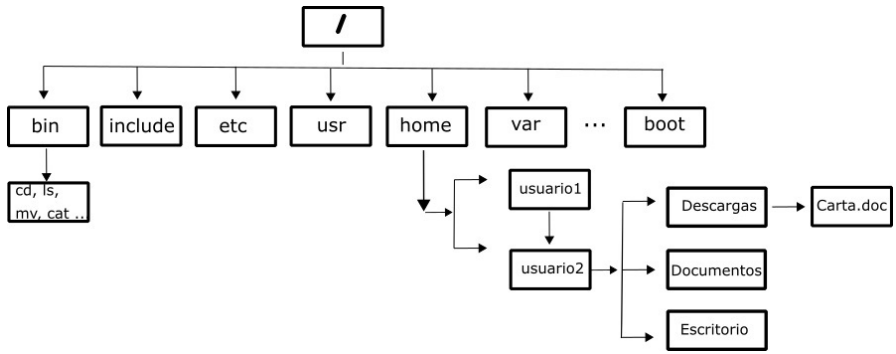


Figura 1.2: Se muestra la jerarquía de los directorios en Linux.

1.5.1. Comandos básicos de Linux

El shell en Linux nos permite administrar el sistema sin usar el entorno gráfico. Todas las distribuciones de Linux incluyen programas binarios, los cuales permiten tener una gran variedad de comandos para manejar el sistema de manera eficiente. Para usar estos comandos, tenemos que abrir una terminal o consola del sistema y empezar a teclear la orden, con su sintaxis apropiada, seguida de un `Enter`. Estos binarios o comandos serán estudiados mediante un orden lógico.

Todos los comandos que se verán a continuación pueden ir acompañados con la opción `-h` que imprime en pantalla de ayuda sobre las opciones básicas de la orden o comando.

Comandos de creación y gestión de archivos y directorios

- `ls`

  Lista el contenido de directorios del sistema. Existen muchas opciones para la orden `ls` (`-a`, `-l`, `-d`, `-r`, ..), que a su vez se pueden combinar de muchas formas. Las más comunes son:

  - `-l`(long): formato de salida largo, con más información que utilizando un listado normal.
  - `-a`(all): se muestran también archivos y directorios ocultos.
  - `-r`(recursive): lista recursivamente los subdirectorios.

- `cd`

  La orden `cd` permite cambiar de directorio de trabajo mediante una sintaxis básica: `cd` *directorio*. Si en Unix tecleamos `cd` sin argumentos iremos a nuestro directorio principal del usuario (e.g. `/HOME/roxana` o `/HOME/est1`). Para subir un nivel en el árbol de Linux, usamos la opción `cd ..` Para ir de un directorio a otro de forma directa y con un paso usamos `cd path`.

- `pwd`

  Imprime en pantalla la ruta completa del directorio de trabajo donde nos encontramos actualmente. No tiene opciones y es una orden útil para saber en todo momento en que punto del sistema de archivos de Linux nos encontramos.

- `mkdir`

  Crea un directorio. La sintaxis es `mkdir directorio`. Para crear un directorio hemos de tener en cuenta los permisos del directorio en que nos encontremos trabajando.

- `touch`

  Actualiza la fecha de última modificación de un archivo o crea un archivo vacío si éste no existe. Con la opción -c no crea este archivo vacío. Su sintaxis es `touch [-c] archivo`.

- `cp`

  Su sintaxis es: `cp origen destino` y, evidentemente, copia el archivo indicado en el parámetro "origen" a otro lugar del disco, indicado en "destino". En Linux es obligatorio especificar el destino de la copia. Si este destino es un directorio, podemos indicar varios orígenes, tanto archivos como directorios, que serán copiados en su interior; estos múltiples orígenes se pueden definir uno a uno por su nombre o bien, mediante comodines, que son los símbolos especiales "*", sustituibles por uno o más caracteres, y "?", sustituidos por uno solo.

Ejemplo

La primera de las dos órdenes siguientes copiará los ficheros `fichero1` y `fichero2` en el directorio `direc1`, y la segunda hará lo mismo con todos los archivos que finalicen en `.h`; en ambos casos el directorio destino ha de existir previamente:

```
roxana:~$  cp  fichero1 fichero2  direc1
roxana:~$  cp  *.h direc1
```

Para copiar de forma recursiva se utilizará la opción `-r`, que copiará tanto ficheros como directorios completos en un determinado destino; por ejemplo, si queremos copiar todo el directorio */etc/* en el directorio actual:

```
roxana:~$  cp -r /etc .
```

La orden anterior no es igual a esta otra:

```
roxana:~$  cp -r /etc/* .
```

La diferencia entre ambas radica en el orden indicado: en el primer caso le decimos al sistema operativo que copie el directorio y todo lo que cuelga de él, por lo que en el destino se creará un nuevo subdirectorio denominado `etc` que contendrá todo lo que había en el origen; en cambio, en el segundo caso, estamos diciendo que se copie todo lo que cuelga del directorio, pero no el propio directorio, por lo que en el destino se copiarán todos los ficheros y subdirectorios del origen, pero sin crear el subdirectorio `etc`. El punto al final indica que el directorio destino será en el que nos encontramos, es decir, significa *aquí*.

- `mv`

  Renombra un archivo o directorio o mueve un archivo de un directorio a otro. Dependiendo del uso, su sintaxis variará: `mv archivo directorio` moverá los archivos especificados a un directorio y `mv archivo1 archivo2` renombrará el primer fichero, asignándole el nombre indicado en `archivo2`.

Ejemplo

```
roxana:~$  mv hola.c prueba.c
```

Aquí renombramos el archivo `hola.c` como `prueba.c`

```
roxana:~$  mv *.c direc1
```

Mueve todos los archivos finalizados en `.c` al directorio `direc1`.

- `ln`

  Asigna un nombre adicional a un archivo, `link`. Para nosotros, la opción más interesante será `-s`, que nos permitirá crear enlaces simbólicos entre archivos del sistema; analizaremos los dos tipos de enlaces de Unix, enlaces duros y enlaces simbólicos, más adelante. La sintaxis básica del mandato es `ln [-s] fuente destino`.

- `rm`

  Elimina archivos o directorios. Sus tres opciones son `-r` (borrado recursivo, de subdirectorios), `-f` (no formula preguntas acerca de los modos de los archivos), e `-i` (interactivo, solicita confirmación antes de borrar cada archivo). Su sintaxis es muy sencilla: `rm -rfi archivo`

- `rmdir`

  Borra directorios si están vacíos. La sintaxis de este mandato será `rmdir directorio`. Si se borran directorios que no estén vacíos, hemos de utilizar `rm -r directorio`.

Tratamiento básico de archivos

- `file`

  La orden `file` nos proporciona información sobre el tipo del archivo, o archivos, especificados como argumento. Es necesario recordar que en Unix no existe el concepto de extensión, por lo que el hecho de que el nombre de fichero acabe en `.txt` o en `.c` no dice nada *a priori* acerca del mismo. Podemos tener un archivo ejecutable que se llame `carta.txt`, un directorio que se llame `prueba.doc`.

- `cat`

  Su función es mostrar en pantalla el contenido de un archivo. Por ejemplo:

```
roxana:~$  cat area_circulo.f
programa area del circulo
real r,a
frite(*,*)'Escribe el radio de la circunferencia'
read(*,*)r
a=(3.14151692)*r*r
write(*,*)'Area=',a
end
```

  El comando `cat` no solo sirve para desplegar el contenido de un archivo, sino que también puede crear un archivo nuevo en el cual se pueda escribir posteriormente de ejecutado el comando. Esto se hace usando la opción >, ejemplo:

```
roxana:~$  cat > saludo.txt
Hola mundo
```

  Esta opción se termina de ejecutar apretando las teclas `Ctrl` y `c` o `Ctrl` y `d`.

  El comando `cat` también nos permite introducir texto a algo que ya esta escrito. Digamos que concluimos un programa, pero al final nos ha faltado introducir el término del programa. Esto lo podemos hacer con la opción >>, ejemplo:

```
roxana:~$  cat >> saludo.txt
de nuevo
```

  Desplegando el programa, con `cat` sencillo, obtenemos:

```
roxana:~$  cat saludo.txt
Hola mundo
de nuevo
```

- `more`

  Visualiza un archivo de pantalla a pantalla, no de forma continua. Es una especie de `cat` con pausas, que permite una cómoda lectura de un archivo. Al final de cada pantalla nos aparecerá un mensaje indicando `--More--`, si en ese momento pulsamos la tecla `Enter`, veremos una o más líneas del archivo. Si pulsamos la tecla `Space`, veremos la siguiente pantalla, si pulsamos `b`, la anterior y si pulsamos `q` saldremos de more. Su sintaxis es `more archivo`.

1.5.2. Herramientas más potentes del sistema

- `head`

  La orden `head` muestra las primeras líneas, 10 por defecto, del archivo que el comando `head` recibe como parámetro. La principal opción de este mandato es la que específica el número de líneas a visualizar (`n`'); y su sintaxis es `head -n archivo`. Ejemplo:

```
roxana:~$  head -3 área_circulo.f
programa área del circulo
real r,a
write(*,*)'Escribe el radio de la circunferencia'
```

- `tail`

  Visualiza el final de un archivo, es la instrucción contraria a `head`. La única opción que nos interesa es (`n`). Si no se especifica este parámetro, `tail` mostrará por defecto las diez últimas líneas del archivo. Su sintaxis es `tail -n archivo`.

- `cmp`

  Esta orden compara el contenido de dos archivos, imprimiendo en pantalla la primera diferencia encontrada, si es que existe; recibe como parámetro los dos nombres de los ficheros a comparar. Ejemplo:

```
roxana:~$  cmp eps.f eps2.f
eps.f eps2.f son distintos: byte 28, linea 2
```

- `diff`

  Esta orden compara dos archivos, indicándonos las líneas que difieren en uno con respecto al otro, si ambos son iguales, la ejecución no producirá ninguna salida en pantalla. Su sintaxis es `diff fichero1 fichero 2`.

  Aunque a primera vista no encontremos ninguna diferencia entre `diff` y `cmp`, ambas instrucciones son muy diferentes; sin ir más lejos, `diff` recorre ambos ficheros por completo, mientras `cmp` se limita a encontrar una diferencia.

- `grep`

  Esta es quizá una de las órdenes más utilizadas e importantes en cualquier sistema Unix; `grep` busca un patrón, que recibe el nombre de parámetro, en un archivo. Su sintaxis es sencilla: `grep [opción] patrón fichero`; al tratarse de una potente herramienta de Unix, muchas de sus opciones son de interés para nosotros, en particular `-v` (imprime las líneas que no coincide con el patrón), `-i` (no distingue mayúsculas, minúsculas) y `-c` (imprime la cuenta de coincidencias).

  La orden `grep` se suele utilizar tanto como instrucción directa, para buscar una cadena en un fichero, como unido a otras órdenes para filtrar su salida mediante |, por ejemplo, las órdenes siguientes son equivalentes:.

```
roxana:~$  grep -i programa epsilon.f
programa epsilon de la maquina
roxana:~$  cat epsilon3.f | grep -i programa
programa epsilon de la maquina
```

- `sort`

  Extrae, compara o mezcla líneas de archivo de texto. Clasifica, según un patrón de clasificación especificado, todas las líneas de archivos que recibe como parámetro; por defecto, este patrón es la línea completa. Las otras opciones de clasificación que nos pueden resultar útiles son:

  - `-d`: Ordenado según el diccionario.
  - `-f`: Interpreta las letras minúsculas como mayúsculas.

  La sintaxis es: `sort -opciones archivos`

- `wc`

  Es un contador de líneas, palabras y caracteres. Es posible utilizarlo como mandato directo y como filtro; también es una potente herramienta en *shell script*[16]. Su sintaxis es `wc [-opción] archivo`, siendo opción una de las siguientes:

  - `-l`: cuenta líneas.
  - `-c`: cuenta caracteres.
  - `-w`: cuenta palabras.

  Se puede usar combinaciones de las tres opciones; por defecto, `wc` asumirá todas: `-wlc`. Si no se indica un nombre de archivo, la orden espera datos de la entrada estándar, es decir el teclado.

1.5.3. Otros comandos

- `echo`

  Está orden imprime sus argumentos en la salida estándar. Por una parte, imprime el mensaje especificado como argumentos (esta opción se usa generalmente en *shells cript*, ya que en otros casos no tiene mucha utilidad visualizar el mensaje en pantalla) y, por otra, consultar el valor de variables de entorno:

```
roxana:~$   echo Hola
Hola
roxana:~$   echo $HOME
/home/roxana
```

- `nohup`

  Mantiene la ejecución de órdenes aunque se desconecte el sistema, ignorando señales de salida y/o pérdida de terminal. Su sintaxis es `nohup orden`. Nos permite dejar trabajos realizándose aunque no estemos físicamente conectados a la computadora, como compilación de grandes programas, de módulos, etc.

[16] El intérprete de comandos o shell es un programa que permite a los usuarios interactuar con el sistema, procesando las órdenes que se le indican. En el apéndice 1 se indican las bases para hacer scripts o programas en shell.

- `man`

  En el sistema operativo Unix, se incorporan unos manuales en los que se describe la sintaxis, opciones y utilidad de las diferentes órdenes del sistema. Este manual es, por supuesto, `man`, que localiza e imprime en pantalla la información solicitada por el usuario.

  En Linux, las páginas del manual se organizan en diferentes categorías, atendiendo a la clasificación de la llamada sobre la que se intenta buscar ayuda. Cada una de estas categorías se almacena en un directorio diferente del disco; si queremos saber la localización de estos directorios, habremos de visualizar la variable de usuario `$MANPATH`. Su sintaxis es `man orden`. Sin embargo, hay una serie de opciones básicas:

  - `-a` (all): Fuerza a `man` a mostrarnos, una a una, todas las diferentes páginas para una misma instrucción o función.
  - `-w` (whereis): No imprime las páginas de ayuda en pantalla, sino que nos indica los archivos, con su ruta absoluta, donde se encuentran tales páginas, que serán ficheros finalizados con la extensión .GHz. En combinación con `-a`, nos dará la ruta de todas las páginas que queremos consultar.

    `Man` también es una orden en Unix, son su propia página de manual. Por lo tanto, la orden `man man` nos dará información sobre la sintaxis completa y todas las opciones de `man`.

1.5.4. Comodines

En Linux, podemos usar métodos abreviados para manejar muchos archivos a la vez o introducir datos. A estos métodos abreviados se les llama comodines o *wildcards* y son simples caracteres que sustituyen objetos. Veamos los más importantes:

Asterisco: *

Este es uno de los comodines más útiles que existen, inclusive se ha heredado de *shell script* a otros lenguajes de programación, como Python. El asterisco * significa "todo". Por ejemplo, si queremos listar todos los archivos de texto, con terminación `.txt` de un directorio:

```
roxana:~$ ls *.txt
```

o copiar todos las imágenes `.png` de la carpeta principal del usuario a la carpeta de `Imágenes`:

```
roxana:~$ cp /home/roxana/*.png /home/roxana/Imagenes
```

La ventaja es que si tenemos 300 imágenes, éstas pasaran casi instantáneamente en un paso.

Punto: .

El punto . en Linux significa "aquí". Regresemos al ejemplo anterior, en el que pasamos todas las imágenes del Escritorio a la carpeta de Imágenes, todavía no queda muy cómodo, ya que habrá que teclear las rutas absolutas del origen y del destino. Sin embargo, si con `cd` nos movemos a la carpeta de Imágenes, ahí podemos sustituir la ruta absoluta del destino por un simple "aquí".

```
roxana:~$ pwd
/home/roxana
roxana:~$ cd Imagenes
roxana:~$ cp /home/roxana/Escritorio/*.png .
```

Mayor que: >

El símbolo de mayor que >, en línea de comandos, sirve para trasladar la información de un comando a un archivo. Esta opción ya la hemos usado con el comando `cat`:

```
roxana:~$ ls -l  > archivo.txt
```

Cada vez que ejecutemos el comando, el contenido del archivo será reemplazado. Si queremos concatenar la información, como lo hacíamos con `cat`, usaremos >>:

```
roxana:~$ ls -l  >> archivo.txt
```

Otro ejemplo: podemos contar las líneas que tiene un archivo, redireccionando la entrada estándar de `wc` hacia un archivo de texto. Así:

```
roxana:~$ wc < archivo.txt
```

Tubería: |

Una tubería | o *pipe* en Linux es una forma práctica de redireccionar la salida estándar de un programa hacia la entrada estándar de otro. Por ejemplo, se pueden ver los procesos que están corriendo en el sistema usando `ps` y redireccionarlos al comando `sort` para que los ordene por el número de identificación `PID`:

```
roxana:~$  ps -a | sort
```

Otro ejemplo sería redireccionar la salida del comando `cat` y pasarla como entrada estándar del comando `wc` para contar las líneas y palabras de un archivo:

```
roxana:~$  cat archivo.txt | wc
```

1.5.5. Permisos en Linux

Ya se ha mencionado que Linux es un sistema multiusuario, por lo que muchas personas pueden utilizar el mismo equipo. Los permisos son un mecanismo que sirve para diferenciar los archivos de un usuario de los demás archivos.

Los permisos de cualquier archivo, inclusive los directorios, se agrupan en 3 grupos de 3 bits cada uno. Es decir:

```
rwx rwx rwx
|   |   |
|   |   | otros
|   grupo
usuario
```

El bit `r` (*read*) indica que es un archivo de lectura, el bit `w` (*write*) es de escritura y el bit `x` (*execute*) indica ejecución.

Con las diferentes combinaciones, se puede configurar un archivo para que pueda ser leído y modificado por su dueño, y sólo leído por el grupo y los demás, por ejemplo el archivo:

```
/etc/pasad:
-rw-r--r-- 1 root root 1509 Par 4 12:44 /etc/pasad
```

Este archivo es del usuario `root` y del grupo del mismo nombre, solamente se puede modificar por su usuario dueño, pero leer por el grupo y los demás.

A diferencia de sistemas operativos como DOS y Windows, el hecho de que un archivo tenga una extensión `.exe` no significa que será un programa ejecutable. Al necesitar restringir los derechos de ejecución de cualquier archivo, la acción de ejecutar cualquier programa estará destinada al permiso correspondiente (bit `x` de ejecución). Por lo que es importante considerar, cuando se escriben programas, que serán interpretados, ya que, al final de cuentas, los archivos serán de texto y para que se ejecuten se deberá activar el permiso de ejecución.

1.5.6. Cambio de permisos: el comando `chmod`

Para cambiar los permisos de los archivos se usa el comando `chmod`. Su sintaxis es la siguiente es `chmod -R modo archivo`. La opción `-R` permite cambiar recursivamente los permisos de todos los archivos dentro de un directorio.

El argumento del comando `chmod` está compuesto por alguna combinación de las letras `u` (usuario dueño), `g` (grupo dueño), y `o` (otros), seguido de un símbolo `+` o `-`, dependiendo si se quiere activar o desactivar un permiso, siguiendo por último una combinación de las letras correspondientes a los distintos permisos: `r`, `w` y `x`. Si se necesita dar permisos de ejecución al usuario y al grupo de un archivo, el comando deberá ejecutarse de la siguiente manera:

```
chmod uf+x archivo
```

También se puede tener los permisos en forma octal con tres dígitos que van del 0 al 7 para el usuario, grupo y otros. Por ejemplo:

```
chmod 777 archivo
```

En la siguiente tabla, está la representación octal y su significado:

Valor	Permiso	Descripción
0	`-`	Ningún permiso
1	`x`	Sólo ejecución de archivos
2	`w`	Sólo escritura
3	`wx`	Escritura y ejecución
4	`r`	Sólo lectura
5	`rx`	Lectura y ejecución
6	`rw`	Lectura y escritura
7	`rwx`	Lectura, escritura y ejecución

Tabla 1.1: Permisos octales

1.5.7. Extensión de archivos

El sistema operativo Unix nos ofrece una gran cantidad de instrucciones orientadas al trabajo con todo tipo de archivos. Esto no es casual, se suele decir, sin exagerar, que en un sistema Unix habitual todo es archivo, desde la memoria física de la computadora hasta la configuración de la impresora, pasando por discos duros, terminales, ratones, etc.

Sin embargo, debe de haber una manera de distinguir los tipos de archivo para que éstos puedan ser utilizados en cualquier sistema operativo, es decir, tener una manera universal de identificar archivos. Para esto es muy útil que se guarden los archivos con extensión. A continuación se mostrarán algunos de estas extensiones:

- **`.txt`**. Archivos de texto.
- **.jpg, .psp, .gif, .png**. Gráficas o imágenes en diversos formatos.
- **.gz, .zip**. Información comprimida, pueden descomprimirse con `gzip -d`, `archivo_destino` y `archivo_original`. respectivamente. Para comprimir en estos formatos se usa **gzip**, **compress** y **zip**.

  Ejemplo: Supongamos que existe un archivo muy largo que deseemos comprimir. Para esto usaremos el comando `gzip archivo.txt`, por lo cual producirá un archivo de salida llamado `archivo.txt.gz`.

- **.html**. Hipertextos HTML que pueden verse con un navegador y editarse con un editor de texto, también hay herramientas especializadas para editar HTML.

- **.tar .tgz .tar.gz**. Directorio con varios archivos empaquetados, pueden desempaquetarse con el comando **tar**.

  Ejemplo: Para crear un empaquetado `programas.tar` a partir del contenido del directorio

  ```
  /home/roxana/compu/programas
  ```

  puede usarse:

  ```
  tar cvf programas.tar /home/roxana/compu/programas
  ```

  Para desempacar el archivo `programas.tar` en el directorio de compu se usa:

  ```
  tar xvf programas.tar
  ```

  Para desempaquetar y descomprimir algo que tiene la extensión **.tgz** o **.tar.gz**, se puede descomprimir primero con `gzip` y el resultado desempaquetarlo con `tar` o puede emplear la opción `z` de `tar`:

  ```
  tar xvfz archivo.tgz
  ```

  De forma análoga, para empaquetar el directorio y comprimir el empaquetado, puede emplear

  ```
  tar cvfz archivo.tgz archivo
  ```

- **.sh**. Script para el interprete de comandos, le dice al sistema operativo Unix como gestionar sus tareas. Puede ejecutarse tecleando el nombre desde la terminal. Para saber más a detalle de la construcción de estos archivos, se recomienda el Apéndice A del presente libro, a modo de introducción.

1.6. Editores de texto

Un editor de texto es un programa que permite crear y modificar archivos digitales compuestos únicamente por texto sin formato, conocidos comúnmente como archivos de texto. El programa lee el archivo e interpreta los bytes leídos según el código de caracteres que usa el editor.

Los editores de texto son incluidos en el sistema operativo o en algún paquete de software instalado y se usan cuando se deben crear o modificar archivos de texto como archivos de configuración o código fuente de algún programa.

El archivo creado por un editor de texto incluye por convención en MS-DOS y Microsoft Windows la extensión **.txt** o **.doc**, aunque pueda ser cambiada a cualquier otra con posterioridad.

Unix y GNU/Linux dan al usuario una total libertad en la denominación de sus archivos. Hay una gran variedad de editores de texto. Algunos son de uso general, mientras que otros están diseñados para escribir o programar en un lenguaje.

1.6.1. Vi

Vi es un simple editor de texto que no formatea el texto en absoluto, pues no centra ni justifica párrafos, pero permite mover, copiar, eliminar o insertar por medio del búfer, permaneciendo la información ahí hasta que los cambios en el archivo se hayan guardado, o bien, hasta que termine la ejecución de la aplicación sin haber guardado las modificaciones.

Vi se encuentra en todo sistema de tipo Unix, de forma que conocerlo es una salvaguarda ante operaciones de emergencia en diversos sistemas operativos. Hay una versión mejorada que se llama **VIM**.

Vi es un editor de dos modos: edición y comandos. En el modo de edición, el texto que ingrese será agregado al texto. En modo de comandos, las teclas que oprima pueden representar algún comando. Cuando comience a editar un texto, estará en modo para dar comandos.

Comandos

- `:q`. Es el comando que debemos utilizar para salir.
- `:q!`. Salir ignorando cambios.

- `:dd`. Borra una línea entera.
- `:i`. Puede insertar texto, pasar a modo de edición, con varias teclas.
- `:a`. Insertar texto antes del carácter sobre el que está el cursor.
- `:I`. Inserta texto después del carácter sobre el que está el cursor.
- `:A`. Inserta texto al comienzo de la línea en la que está el cursor.
- `:o`. Inserta texto al final de la línea en la que está el cursor.
- `:O`. Abre espacio para una nueva línea después de la línea en la que está el cursor y permite insertar texto en la nueva línea.

Para pasar de modo edición a modo comandos se emplea la tecla `ESC`, para desplazarse sobre el archivo se emplean las teclas `j` (abajo), `k` (arriba), `h` (izquierda) e `I` (derecha). También puede emplear las flechas, si su terminal lo permite, `Ctrl+U` (PgUP), `Ctrl+D` (PgDn).

Una de las utilidades más comunes es el uso de `:wq` que corresponde a la unión de las opciones guardar `w` y salir `q`, o bien, el modo forzado es `:q!` que sale de **Vi** sin guardar cambios.

1.6.2. Gedit

Gedit es un completo editor de textos libre que se distribuye junto al gestor de escritorio GNOME para sistemas tipo Unix. Este editor se caracteriza principalmente por su facilidad de uso, conseguida en gran parte gracias a un interfaz gráfico claro y limpio, mostrando únicamente las funcionalidades principales que suelen requerir la mayoría de usuarios.

1.6.3. Kate

Kate es un editor de textos para KDE. **Kate** significa **K**DE **A**dvanced **Text Editor**, es decir, editor de textos avanzados para KDE. Algunas de sus características son:

- Destacado de sintaxis.
- Búsqueda y reemplazo de texto usando expresiones regulares (comodines).

- Seguimiento de código para C++, C, Python, FORTRAN, PHP y otros.
- Mantener múltiples documentos abiertos en una ventana.
- Emulador de terminal basado en Konsole.

1.6.4. EMACS

EMACS es un editor de texto con una gran cantidad de funciones, es muy popular entre programadores y usuarios técnicos. Este editor forma parte del proyecto GNU. En el manual de GNU **EMACS**, lo describe como "un editor extensible, personalizable, auto-documentado y de tiempo real."

EMACS significa, Editor MACroS para el TECO. Fue escrito en 1975 por Richard Stallman junto con Guy Steele. Fue inspirado por las ideas de TECMAC y TMACS, un par de editores TECO-macro escritos por Guy Steele, Dave Moon, Richard Greenblatt, Charles Frankston y otros. Se han lanzado muchas versiones de **EMACS** hasta el momento, pero actualmente hay dos que son usadas comúnmente: **GNU EMACS**, iniciado por Richard Stallman en 1984, y XEmacs, un fork de **GNU EMACS**, que fue iniciado en 1991. Ambos usan una extensión de lenguaje muy poderosa, **EMACS Lisp**, que permite manejar tareas distintas, desde escribir y compilar programas hasta navegar en Internet.

Si tenemos implementado un ambiente X-Window, al teclear **EMACS** en la terminal, aparecerá una ventana nueva. Una vez en **EMACS**, podrá desplazarse sobre el texto con las flechas; para insertar texto, se tiene que escribir justo donde se quiera poner. Los comandos en **EMACS** se indican con secuencias de teclas con `Control`, abreviada con `C` en la documentación de **EMACS**. Algunas secuencias de teclas útiles son:

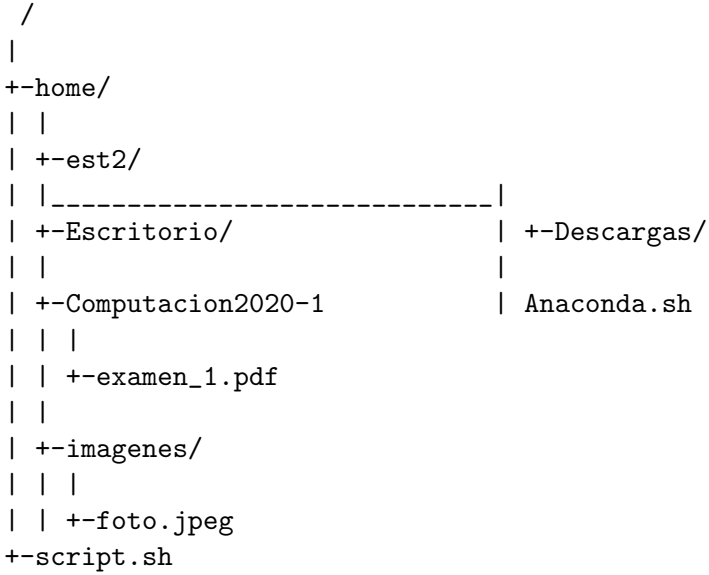
- `C-x C-c`. Para salir de **EMACS**. Note que son dos secuencias de teclas, primero `Control` simultáneamente con `x` y después Control simultáneamente con `c`.
- `C-x C-s`. Para salvar el archivo que se está editando.
- `C-f`. Avanzar a la derecha o a la siguiente línea si se está al final de una. Equivalente a flecha a la derecha.

- `C-b`. Avanzar a la izquierda o al final de la línea anterior si está al comienzo de una. Equivale a flecha a la izquierda.
- `C-n`. Avanzar a la línea siguiente, equivalente a flecha hacia abajo.
- `C-p`. Pasar a la línea anterior, equivale a flecha hacia arriba.
- `M-x`. Para dar un comando especial a **EMACS**; entre los comandos especiales están: `info` para consultar dentro de **EMACS** páginas del manual `info`, `man` para consultar páginas del manual, `shell` para abrir una terminal dentro de **EMACS**.

En la parte inferior de este editor, verá una línea de modo, tiene varios guiones, y debajo de esta el *minibuffer* o área de eco en la que **EMACS** recibe y envía información del usuario. La línea de modo puede aparecer con dos asteriscos al comienzo para indicar que el texto editado se ha modificado y no se ha salvado.

1.7. Ejercicios

1. Da la ruta absoluta de los documentos `Anaconda.sh`, `examen1.pdf` y `foto.png`.



2. Da la ruta relativa de:

   `/home/est5/Documentos/compu/examen1/ejercicio1.py`

3. (i) ¿Cuál es la sintaxis de los comandos de Shell en Linux?
   (ii) Explica como funcionan los comando `ls`, `cd` y `cat` con dos o tres opciones cada uno.
   (iii) Explica como cambiamos permisos y que sintaxis es la correcta.
   (iv) Escribe y explica que son los comodines en Bash y cómo funcionan.

4. El 21 de noviembre de 1969 se transmitió el primer mensaje por la red Arpanet, ahora Internet, entre las universidades UCLA y Standford. Usando la terminal, ¿qué día de la semana paso esto? (usa el comando `cal` de calendario).

5. En línea de comando, haz una carpeta `Ejer_Linux` dentro de tu `Escritorio`. Aquí guardarás todos los archivos generados en este capítulo.

6. En una sola línea, lista todos los archivos, incluyendo los ocultos, de la raíz `/` y guárdalos en un archivo de texto llamado `raiz.txt`.

7. Suponiendo que te encuentras en tu directorio personal, muestra un listado del contenido de `/usr/bin`:
   i) Con una sola línea de comandos.
   ii) Moviéndote paso a paso por los directorios.
   iii) Con dos líneas de comandos.

8. Lista el contenido del directorio `/etc`:
   (i) Ordena el listado por fecha de modificación, muestra primero los archivos más recientes.
   (ii) Ordena el listado por fecha de modificación, muesta primero los archivos más antiguos.
   (iii) Muestra los tamaños de archivo en unidades amigables (KB, MB, GB).
   (iv) Lista primero los directorios y luego los archivos.
   (v) Ordena por tamaño de archivo, de mayor a menor.
   (vi) Ordenar por tamaño de archivo, de menor a mayor.

9. Copia el archivo `/etc/passwd` a la carpeta `Ejer_Linux`, cambia el nombre a contraseña. Usa el comando `grep` para mostrar las líneas que tienen los detalles de tu usuario `est5`.

10. En la línea de comando y usando el comando `cat`, haz un archivo llamado `cita.txt` que diga: " Lo más difícil de aprender en la vida es qué puente hay que cruzar y qué puente hay que quemar.". Luego, en otra línea de comando, concatena la frase "Bertrand Russell (1872-1970)".

11. Crea un directorio en tu directorio de trabajo `Ejer_Linux` con nombre `prueba`. Copia el archivo `gzip` del directorio `/bin` al directorio `prueba`. Crea un duplicado de `gzip` con nombre `gzip2` dentro de `prueba`.

12. Cambia el nombre de `prueba` a `prueba2`. Crea `prueba3` en el mismo nivel que `prueba2` y mueve todos los ficheros de `prueba2` a `prueba3`. Borra `prueba2`.

13. (i) Lista todos los archivos del directorio `etc` que empiecen por `t` en orden inverso.
    (ii) Lista todos los archivos del directorio `dev` que empiecen por `t` y acaben en `C1`.
    (iii) Borra todos los archivos y directorios visibles del directorio `prueba3`
    (iv) Crea el archivo proceso con los procesos que no tienen ningún terminal asignado.
    (v) Muestra cuantos usuarios tiene registrados el sistema (el registro de usuarios está en el archivo `/etc/passwd`).

14. Crea un fichero con nombre `topsecret.txt` en tu directorio de trabajo, que únicamente tenga permiso de ejecución.

15. Ingresa argumentos en línea de comando:
    En el presente problema harás un archivo terminación `.sh`, para esto tendrás que seguir las siguientes instrucciones:
    (i) Abre el editor de texto que prefieras y escribe lo siguiente:

```
#!/bin/bash

echo "Hola Mundo"
```

Guarda este archivo y llámalo `HolaMundo.sh`. Agrega el permiso de ejecución.
(ii) Ejecuta el archivo poniendo en la línea de comando `./HolaMundo.sh`. Explica brevemente que hace el comando `echo`.
(iii) Agrega al final una línea al código que será `echo "Hola $1"`. Luego, ejecuta el archivo con `./HolaMundo.sh nombre`, donde nombre es el argumento. Escribe que es lo que regresa el archivo. ¿Qué hace `$1`?
(iv) Concatena la línea de código `echo -n "$1"| wc -m`. Luego, ejecuta el archivo `./HolaMundo.sh lasquinceletras`. Explica que está pasando a detalle.

Capítulo 2

Introducción a Python

Una de las grandes ventajas que presenta Python sobre otros lenguajes de programación es que combina la programación de alto nivel con ser compacto y fácil de aprender. Hacer un programa en Python es bastante corto e intuitivo. Por esta razón, hay muchas personas que lo consideran como un lenguaje de programación de "muy alto nivel". Otra de las grandes ventajas es su legibilidad y una sintaxis extremadamente elegante, lo que permite que alguien que no sepa programar aprenda Python en muy poco tiempo.

Python fue creado a finales de la década de los 1980 por el holandés Guido van Rossum. Van Rossum nombró a su lenguaje de programación como Python debido a que era muy aficionado a la serie humorística británica Monty Python. Python ofrece un entorno interactivo en el cual se pueden diseñar pequeños programas y ejecutarlos en cuanto se termina el código, sin tener que compilar. En general, lenguajes compilados como C o Fortran necesitan un conjunto de herramientas para formar un entorno de programación; este conjunto de herramientas puede estar formado por editores de texto, compiladores y archivos ejecutables. Entonces, la programación de un lenguaje compilado requiere escribir el código en un editor de texto, luego compilar y por último ejecutar. En forma estricta, compilar es traducir un lenguaje de programación de alto nivel, como es C o Fortran, a un lenguaje de bajo nivel, como Ensamblador. En el caso de un lenguaje interpretado como Python, el entorno de programación se ve reducido a un intérprete en el cual escribimos los comandos y son directamente interpretados. En los lenguajes interpretados, se pueden establecer un "diálogo" con el intérprete: se escribe una orden en la línea de comandos y en seguida nos responde con el resultado. En este libro aprenderemos a programar Python3.

2.1. Python

La mayoría de los sistemas Unix modernos incluyen los compiladores y paquetes básicos de Python. Una forma de comenzar con el interpretador es abrir una terminal y en la línea de comandos escribir `python` para entrar al interpretador.

Para Windows, es necesario entrar al sitio principal de Python, cuya dirección electrónica es www.python.org, y buscar un interpretador compatible con el Windows en cuestión. El interpretador más famoso es **IDLE**, que es compatible para varias plataformas (Windows, Mac, Linux). También se tiene la opción de instalar Anaconda[1] con muchos paquetes incluidos y accesible para Python científico, que al igual que todas las distribuciones de Python, es libre y multiplataforma. Recuerda que usaremos Python3, por lo que fíjate que tengas esta opción antes de empezar a programar.

Si uno ha decidido trabajar en Windows, es necesario descargar un entorno que permita entrar al modo interactivo. Si se ha decidido usar Anaconda, se abrirá un Navegador independiente donde tenemos todas las opciones que presenta el entorno Anaconda. Para la parte introductoria de Python y para entrar directo al entorno interactivo, se recomienda usar Jupyter[2].

Cuando ya nos encontramos en el entorno de Jupyter, podemos escoger *new* o nuevo y, así, ingresaremos al entorno interactivo de Python. Como se mencionó anteriormente, Anaconda, y por ende Jupyter, es multiplataforma, así que puede ser utilizado en cualquier sistema operativo, véase la figura (2.1).

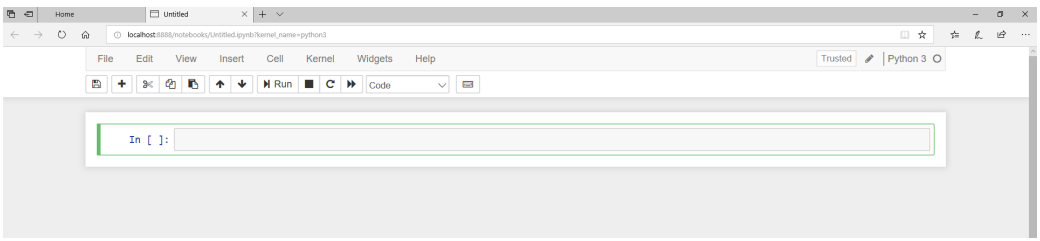


Figura 2.1: Inicio del entorno interactivo de Jupyter en Anaconda.

[1] https://www.anaconda.com/distribution/.

[2] Jupyter es una aplicación desarrollada desde el navegador que proporciona una forma práctica e interactiva para trabajar con Python, se recomienda ver la página oficial https://jupyter.org/

2.2. Operaciones básicas

Una utilidad muy básica de Python es el de usar el interpretador como una calculadora muy avanzada, por lo que podemos utilizar operaciones básicas como las operaciones binarias comunes que conocemos de la primaria; suma; resta; multiplicación; división; potenciación. Algunos ejemplos son:

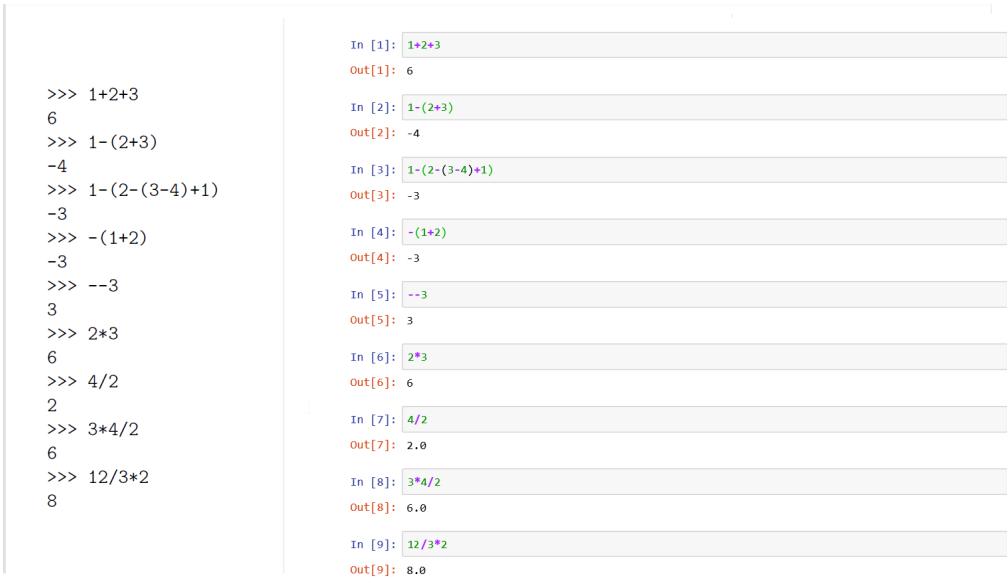


Figura 2.2: Operaciones básicas en el entorno interactivo de Python.

A la izquierda de la figura (2.2), los signos `>>>` representan la línea de comandos en Python, y la falta de estos signos representa la respuesta del interpretador. A la derecha `In[1]` representa la línea de comando en espera de instrucciones del usuario y `Out[2]` representa la respuesta del intérprete Python.

De ahora en adelante, usaremos la notación de la izquierda para indicar que estamos en modo intérprete.

Cabe destacar que Python respeta las reglas de precedencia operacionales, por lo que `2*5+3` es equivalente a $(2 \times 5) + 3$ y no a $2 \times (5 + 3)$; por esta razón se recomienda usar los paréntesis adecuadamente según la operación que se quiera realizar.

En Python, existen más operadores de los que tiene una calculadora normal. Un ejemplo es el operador módulo, que denotaremos con el símbolo de porcentaje: %. Este operador devuelve el residuo de la división entre dos números:

```
>>> 27%5
2
>>> 25%5
0
```

Otro operador importante es el de potencias, que se denota por $**$. La notación convencional matemática será x^n, mientras que en Python utilizaremos `x**n`. El exponente es asociativo a la derecha, por lo que `2**3**2` equivale a $2^{3^2} = 2^9 = 512$ y no $(2^3)^2 = 8^2 = 64$.

Los comentarios en los lenguajes de programación son anotaciones significativas del programador pero que el compilador o intérprete ignorará. Esto sirve para mantener los códigos bien documentados. En Python, los comentarios van antecedidos por el símbolo #.

2.2.1. Tipos de datos

Enteros y Flotantes

Sabemos que existen distintos tipos de números. No es lo mismo referirnos a 0.5 que a 2 o al número π, hacemos una distinción entre reales, enteros, o irracionales. En computación, pasa lo mismo, tenemos que identificar explícitamente que tipo de número estamos utilizando. Una operación entre enteros me devuelve un entero. Por ejemplo, consideremos las siguientes operaciones entre 3 y 2:

```
>>> 3//2      #forzamos la división entera con //
1
>>> 3.0/2.0
1.5
>>> 3./2.
1.5
```

En este ejemplo, es notorio lo que hace Python con la división. En la primera línea, realizamos una división entre enteros y el resultado solo fue un entero; en la tercera línea, hicimos la división entre números reales, por lo que nos dio como resultado un real. Es importante decirle al intérprete con que tipo de

número trabajará. Una de las utilidades para distinguir los tipos de números es que los enteros suelen ocupar menos memoria y las operaciones entre enteros son más rápidas que otro tipo de operaciones.

En computación, se llama a los números reales como números en formato de coma flotante o simplemente flotantes. Un número flotante debe especificarse siguiendo ciertas reglas. Por lo general, consta de dos partes: mantisa y exponente. El exponente se separa de la mantisa usando la letra `e`. Por ejemplo, `2e3` tiene mantisa `2`, exponente `3` y representa al número 2×10^3; otro ejemplo sería `3.2e-3`, es decir 0.0032.

Algo interesante de Python es la declaración de la variable que se hace al mismo tiempo de ser usada, es decir, en `a=1` se está indicando al intérprete de Python que asigne en la memoria una variable llamada `a`, con valor numérico de 1 entero. En la misma línea, asignamos y declaramos el tipo de variable. En detalle, las variables se utilizan para guardar distintos tipos de datos y volverlos a usar posteriormente. En muchas ocasiones hace falta crear algunas variables con cantidades específicas, ya que estas variables se pueden utilizar mucho en una sesión. El acto de dar valor a una variable se denomina asignación. Por ejemplo:

```
>>> pi=3.141516
>>> e=2.71828
```

Se debe considerar que las asignaciones son "mudas", es decir, no darán una respuesta de salida en pantalla. Una asignación no es una ecuación matemática, sino una acción, por lo que es importante el orden de la asignación, es decir, *expresión=variable* no es equivalente a decir *variable=expresión*. La manera adecuada es la segunda. Se puede asignar un nuevo valor a una misma variable cuantas veces se requiera, las variables suelen ser dinámicas y mutables en Python. La variable sólo recordará el último valor asignado hasta que se le asigne otro.

Se pueden asignar valores a muchas variables en una sola línea de comandos de manera que en lugar de escribir,

```
>>> a=2
>>> b=3
>>> c=4
>>> a, b, c
(2, 3, 4)
```

Se puede escribir,

```
>>> a,b,c=2,3,4
>>> a,b,c
(2, 3, 4)
```

Este tipo de asignación múltiple es sumamente popular en la comunidad de Python, debido a que permite economizar espacio de código.

Cadenas

Python también puede analizar los tipos de datos denominados cadena. Una cadena es una secuencia de caracteres (números, marcas de puntuación, letras, etc.) y se distinguen por ir encerradas en comillas. Las cadenas pueden usarse para representar información textual y pueden ser almacenadas en variables, por ejemplo:

```
>>> nombre="Roxana"
>>> nombre
"Roxana"
```

Las cadenas se pueden manipular como si fueran listas, por ejemplo:

```
>>> nombre="Hola mundo"
>>> nombre
"Hola mundo"
>>> nombre[0]
"H"
>>> nombre[1]
"o"
>>> nombre[:2]
"Ho"
>>> nombre[2:4]
"la"
>>> nombre[2:]
"la mundo"
```

También se pueden realizar operaciones con las cadenas. Un ejemplo importante es la operación de concatenación[3]. El símbolo usado para la concatenación

[3]Es la operación por la cual dos caracteres se unen para formar una cadena de caracteres. Concatenar es la acción de poner una idea seguida de otra en orden lógico. En computación, colocamos una variable seguida de la otra variable, e.g.: `a="Hola"` y `b="Mundo"`, tal que `a+b="HolaMundo"`.

es +, el mismo que usamos cuando sumamos enteros y/o flotantes. Sin embargo, aunque el símbolo sea igual, debemos considerar que no es la misma operación:

```
>>> "a"+"b"
"ab"
>>> nombre+"del Castillo"
"Roxanadel Castillo"
>>> nombre+" "+"del Castillo"
"Roxana del Castillo"
```

A veces, se definen operaciones adicionales que uno no esperaría, por ejemplo:

```
>>> "Hola"*10
"HolaHolaHolaHolaHolaHolaHolaHolaHolaHola"
>>> ("Hola"+" ")*10
"Hola Hola Hola Hola Hola Hola Hola Hola Hola Hola "
>>> "-"*50
"--------------------------------------------------"
```

Sin embargo, `"Hola"*"Hola"` no tiene sentido y regresa un error.

Listas

Un elemento usado frecuentemente en Python son las listas. Las listas guardan una secuencia de datos ordenados, la analogía matemática de listas son los vectores. La diferencia principal es que, en Python, se puede almacenar cualquier tipo de datos dentro de las listas y no es necesario fijar su tamaño previamente como en otros lenguajes de programación. Por ejemplo:

```
>>> l=[1,3.0,"hola","fin"]
>>> l[0]
1
>>> l[1]
3.0
>>> l[2]
"hola"
>>> len(l)
4
>>> l[2:]
["hola","fin"]
>>> l[1:3]
[3.0,"hola"]
```

La función `len(l)` nos regresa el tamaño de la lista (de *length* en inglés). Notamos que las listas asignan lugares desde el 0 (Python empieza a contar en 0). Los dos puntos : representan un *hasta*. Por ejemplo, *hasta el lugar dos* ó *hasta el último lugar*, en caso de que no se ponga nada.

Las listas son variables en la memoria que podemos ir redefiniendo conforme evolucionemos el código. Por lo que se le llama como variables mutables, es decir, van cambiando conforme lo necesitamos

Tuples

Un **tuple** es una colección ordenada de elementos guardados en una variable y es inmutable. En Python las tuplas se escriben con paréntesis redondos:

```
>>> t=("calculo","algebra","computacion","fisica","geometria")
>>> print(t)
("calculo", "algebra", "computacion", "fisica", "geometria")
```

Al igual que las listas, podemos obtener elementos específicos de los tuples, es decir, indexar los elementos del tuple:

```
>>> print(t[0])
calculo
```

También podemos indexar negativamente los elementos del tuple, etiquetando de atrás hacia adelante los elementos -1, -2, -3, etc. Este indexado negativo funciona para cualquier variable con un conjunto ordenado de elementos, como es el tuple y la lista:

```
>>> print(t[-1])
geometria
```

Con la indexación negativa, es más fácil tener el último elemento del tuple, sin saber exactamente cuantos elementos tiene.

También podemos especificar un rango de índices cuando indicamos el inicio y el final:

```
>>> print(t[1:4])
("algebra", "computacion", "fisica")
```

Aquí indicamos que regresara del índice 1, recordar que Python empieza a contar de 0, hasta el índice 3. En realidad, indicamos el índice 4, pero Python siempre toma el intervalo semi-abierto [1,4), es decir, el 4 no lo incluye.

Como los tuples son variables inmutables, una vez definidas, no pueden ser cambiadas.

Diccionarios

Los diccionarios son otro tipo de variables que contienen elementos ordenados de datos. Podemos asignarle tanta información a los elementos que es mejor llamarles colecciones. Los diccionarios son entes muy poderosos en Python y son muy recurrentes tanto en el cómputo científico como en la ciencia de datos. Los diccionarios también se denominan "arreglos asociativos", contienen un par de valores, uno es la clave o *Key* y el otro elemento del par correspondiente es su clave: valor. En Python, un diccionario es declarado con llaves y cada elemento es separado por comas, un elemento es asignado con su clave o nombre y su valor correspondiente. Los valores en un diccionario pueden ser de cualquier tipo de datos y pueden duplicarse, mientras que las claves no pueden repetirse y deben ser inmutables:

```
>>> peso={} #Diccionario vacío
>>> peso=dict() #Otra forma de asignar diccionario vacío
>>> peso["H"]=1 #Asignamos elemento (clave y valor)
>>> peso["He"]=2
>>> peso["Cl"]=35.5
>>> peso
{"H": 1, "Cl": 35.5, "He": 2}
>>> peso.keys() #pedimos que solo regrese las claves
["H", "Cl", "He"]
```

Si queremos obtener el valor de una clave, podemos llamarla como si sacáramos el elemento de una lista:

```
>>> peso['H']
1
>>> peso['Cl']
35.5
```

También está la función `get()` que nos ayuda a acceder al elemento de un diccionario:

```
>>> print(peso.get('He'))
2
>>> print(peso.get('Cl'))
35.5
```

Podemos quitar un elemento con la función `pop()`. Esta función quita el elemento de una clave dada:

```
>>> cuadrados={1:1,2:4,3:9,4:16,5:25}
>>> cuadrados
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> print(cuadrados.pop(4))
16
>>> cuadrados
{1: 1, 2: 4, 3: 9, 5: 25}
```

Los elementos de los diccionarios pueden ser cualquier tipo de variable, inclusive podemos usar diccionarios dentro de diccionarios:

```
>>> mezcla_elementos=dict()
>>> mezcla_elementos[1]=1
>>> mezcla_elementos[2]=2.0
>>> mezcla_elementos[3]='tres'
>>> mezcla_elementos[4]=(4,5)
>>> mezcla_elementos[5]=[5,6]
>>> mezcla_elementos[6]={'i':'a','ii':'b'}
>>> print(mezcla_elementos)
{1: 1, 2: 2.0, 3: 'tres', 4: (4, 5), 5: [5, 6],
6: {'i': 'a', 'ii': 'b'}}
```

Sin embargo, las claves no tienen los mismos atributos. No podemos definir claves como listas ni diccionarios porque son mutables:

```
>>> mezcla_claves=dict()
>>> mezcla_claves[1]='entero'
>>> mezcla_claves[2.]='flotante'
>>> mezcla_claves['tres']='cadena'
>>> mezcla_claves[(3,4)]='tuple'
>>> print(mezcla_claves.keys())
dict_keys([1, 2.0, 'tres', (3, 4)])
>>> print(mezcla_claves)
{1: 'entero', 2.0: 'flotante', 'tres': 'cadena',
(3, 4): 'tuple'}
>>> #mezcla_claves[[5,6]]='lista'
TypeError: unhashable type: 'list'
>>> mezcla_claves[{'i':'a','ii':'b'}]='diccionario'}
TypeError: unhashable type: 'dict'
```

De la misma manera que tenemos `get()` y `pop()`, existe un buen número de funciones que nos harán la vida más sencilla en el manejo de diccionarios. A continuación, se muestran algunas de estas funciones:

Función	**Descripción**
copy()	Devuelve una copia del diccionario.
clear()	Elimina todos los elementos.
pop()	Elimina y devuelve un elemento de un diccionario que tenga la clave dada.
popitem()	Elimina el par arbitrario clave-valor del diccionario y lo devuelve como tupla.
get()	Es la función convencional para acceder a un valor para una clave.
cmp()	Compara elementos de dos diccionarios.
fromkeys()	Crea un nuevo diccionario con claves de *seq* y valores establecidos en *value*.

Tabla 2.1: Algunas funciones útiles para trabajar con diccionarios en Python

2.2.2. Asignación con operador

Al realizar cálculos numéricos, es frecuente utilizar la operación $i = i + 1$, no matemáticamente hablando, sino como se ve en computación, es decir, se incrementa el valor de la variable en una cantidad cualquiera. En este caso incrementamos la variable i en 1. En Python se puede escribir como $i+ = 1$. Por ejemplo:

```
>>> a=2
>>> b=3
>>> a+=6*b
>>> a
20
```

Todos los operadores aritméticos tienen una abreviación del mismo tipo asociada. Por ejemplo:

```
>>> z=2
>>> print(z)
2
```

```
>>> z+=2
>>> print(z)
4
>>> z*=2
>>> print(z)
8
>>> z/=2
>>> print(z)
4.0
>>> z-=2
>>> print(z)
2.0
>>> z**=2
>>> print(z)
4.0
>>> z%=2
>>> print(z)
0.0
```

La ventaja de estas operaciones es que deja en explícito el hecho de que estamos modificando una variable basado en su valor anterior.

2.2.3. Operadores lógicos y de comparación

En matemáticas, existen los operadores lógicos o booleanos, los cuales reciben su nombre por el matemático del siglo XIX, George Boole, que desarrolló un sistema algebraico basado en dos valores (cierto, falso) y tres operaciones (conjunción, disyunción y negación). En Python, las operaciones binarias de conjunción, disyunción y negación se denotan por `and`, `or` y `not`, respectivamente. Mientras las variables booleanas son `True` y `False`. Por ejemplo:

```
>>> True and False
False
>>> (True and False) or True
True
>>> not True
False
```

También hay operadores de comparación, los cuales, en Python, tienen una sintaxis particular que se presenta en la siguiente tabla:

operador	comparación
==	es igual que
!=	es distinto de
<	es menor que
<=	es menor o igual que
>	es mayor que
>=	es mayor o igual que

Con todo esto, ya podemos operar los siguientes ejemplos:

```
>>> 5>1
True
>>> 5>=1
True
>>> 5>5
False
>>> 5>=5
True
>>> 1!=0
True
>>> 1!=1
False
>>> -2<=2
True
```

Estos operadores binarios y variables boolenanas son muy importantes para generar una buena estructura de programación, limpia y clara.

Un detalle útil para los futuros códigos que hagamos es que Python también tiene un valor numérico para las variables Booleanas, `True=1` y `False=0`, que inclusive se pueden operar numéricamente:

```
>>> True==1
True
>>> False==0
True
>>> True+3
4
>>> False-1
-1
```

2.2.4. Funciones predefinidas

Como en otros lenguajes de programación, en Python, hay varias funciones predefinidas para ahorrar tiempo al usuario. Un ejemplo es la función *abs*, la cual calcula el valor absoluto de un número:

```
>>> abs(-3)
3
>>> abs(3)
3
```

Esta función es muy conocida, pero existen otras funciones que no lo son. Algunas de estas son:

1. **float**: Está función convierte números enteros a números de tipo flotante. También acepta argumentos de tipo cadena con la condición de que la cadena represente un flotante:

```
>>> c=6
>>> float(c)
6.0
>>> float("3.2")
3.2000000000000002
>>> float("una cadena")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for float(): una cadena
```

2. **int**: Convierte un número de tipo flotante a entero y devuelve el entero que se obtenga eliminando la parte fraccionaria. También acepta como argumento una cadena:

```
>>> int(2.5)
2
>>> int(-2.5)
-2
>>> int("2")
2
```

3. **str**: Convierte a cadenas. Recibe un número y devuelve una representación de éste como cadena:

```
>>> str(4.8)
"4.8"
>>> str(243E47)
"2.43e+49"
```

4. **round**: Esta función redondea un número flotante:

```
>>> round(4.9)
5.0
>>> round(4.1)
4.0
>>> round(-4.1)
-4.0
>>> round(4)
4.0
```

   Nótese que el resultado es de tipo flotante.

Cabe mencionar que el uso de paréntesis después del nombre de la función es obligatorio aún cuando no se tienen argumentos que dar. En caso de que no se sepa exactamente cuál es la sintaxis de una función o qué parámetros se necesitan para usarla, existe un implemento de ayuda en Python. Para usarlo se escribe `help(funcion)` , o `funcion?`, donde `funcion` es la función de la cual se tienen dudas:

```
>>> round?
Docstring:
round(number[, ndigits]) -> number
Round a number to a given precision in decimal digits
(default 0 digits).
This returns an int when called with one argument,
otherwise the same type as the number.
ndigits may be negative.
Type:      builtin_function_or_method
```

2.2.5. Funciones definidas en módulos

En Python también se pueden utilizar las funciones comúnmente conocidas como las funciones trigonométricas, logaritmo, etc. Estas funciones no están disponibles directamente cuando iniciamos la sesión en el modo interactivo. Para usarlas, importamos cada función de una librería.

La librería math

Si queremos utilizar la función seno, coseno, etc. debemos importar la librería `math`.

```
>>> sin(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name "sin" is not defined
>>> from math import sin
>>> sin(0)
0.0
```

Si queremos utilizar la función coseno, tenemos que importarla de la librería; para no tener que repetir este proceso muchas veces, al iniciar la sesión, podemos decirle a Python que importe todo, esto lo hacemos colocando en la línea de comando:

```
>>> from math import *
```

Otra forma de importar funciones, la cual es muy recomendable cuando se empieza a programar en Python, es la siguiente:

```
>>> import math
>>> print math.sin(0)
0.0
>>> print math.cos(0)
1.0
```

De esta manera, importamos toda la librería pero no dejamos de identificar de que librería proviene cada función. Esto también es útil cuando tenemos funciones con el mismo nombre que pueden venir de distintas librerías, así evitamos la confusión al momento de ejecutar el programa.

Librerías	**Descripción**
math	Librería Básica que incluye funciones de una calculadora científica. Algunas funciones definidas aquí son sin, cos y variables *pi*, *exp*, entre otras.
cmath	Librería análoga a math pero las operaciones se hacen con números complejos.
random	Librería para generación de números aleatorios o pseudo-aleatorios.
numpy	Librería designada al cálculo numérico de Python, en especifico, operaciones con arreglos y matrices. Para una mejor descripción, véase el capítulo 4
matplotlib	Librería para visualización y ploteo de datos. Se incluye una kit de herramientas extras para visualización en 3d (mplot3d). Para más detalle, véase el capítulo 5.
sympy	Librería para cálculo simbólico, se presentan más detalles en el capítulo 5.
pandas	Librería encargada de funciones asignadas para Ciencia de datos o *Machine Learning*.

Tabla 2.2: Algunas librerías populares y útiles para el cómputo científico.

Otra variación recurrente es llamar a la librería con un alías, por ejemplo, `import numpy as np` o `import matplotlib as mp`, esto permite que al llamar a las funciones se tenga una sintaxis mas corta. Los alías varían de acuerda a cada usuario. El ejemplo anterior, usando un alías, se reduce a:

```
>>> import math as m
>>> print m.sin(0)
0.0
>>> print m.cos(0)
1.0
```

Más adelante seguiremos viendo algunas otras librerías importantes.

2.3. Algunos programas sencillos

Es bastante latoso teclear todas las instrucciones cada vez que queramos programar, es más fácil guardar todos los comandos de un programa en un archivo de texto. El archivo de texto tiene que tener la terminación **.py**. Un editor de texto recomendado es *spyder* ya que resalta la sintaxis de Python automáticamente una vez que se haya guardado el archivo con la extensión adecuada. Ya que se ha escrito un programa es posible ejecutarlo directamente. Si se invoca al intérprete `python`, seguido del nombre de un archivo desde la línea de comandos, no se iniciará una sesión con el intérprete interactivo, sino que se ejecutará el programa contenido en el archivo en cuestión y luego el control se regresa al shell. Por ejemplo, para correr el programa `volumen.py` ponemos:

```
python volumen.py
```

Si estás en el notebook de Jupyter, es suficiente terminar de escribir el código y apretar la combinación de teclas `Shift Enter` para que el intérprete regrese el resultado del código.

Por ejemplo, el siguiente programa calcula el volumen de una esfera a partir de su radio en cualquier tipo de unidad:

```
from math import pi
radio=1
volumen=4.0/3.0*pi*radio**3
print(volumen)
```

Si se desea calcular el volumen de una esfera de 3 unidades de radio, se debe editar el archivo que contiene el programa, corrigiendo la línea donde se declara el radio:

```
from math import pi
radio=3
volumen=4.0/3.0*pi*radio**3
print(volumen)
```

Para no tener que modificar el programa cada vez que se quiera calcular el volumen de una esfera diferente, se requiere poder pedirle al usuario esta información por medio de una función. Esta función es `input` (en inglés significa *entrada*), que hace lo siguiente: detiene la ejecución del programa y espera a que el usuario escriba un valor del radio y pulse la tecla de retorno o *enter*.

En ese momento, prosigue la ejecución y la función devuelve una cadena con el texto que tecleó el usuario.

Si deseas que el radio sea un valor flotante, debes transformar la cadena devuelta por `input` en un dato de tipo flotante, usando la función `float`. La función `float` recibe como argumento la cadena que devuelve `input` y proporciona un número flotante. Recuerda que existe otra función de conversión, `int`, que devuelve un entero en lugar de un flotante. Por otra parte, `input` es una función y, por lo tanto, el uso de los paréntesis que siguen a su nombre es obligatorio, incluso cuando no tenga argumentos. El programa anterior queda:

```
from math import pi
radio=float(input())
volumen=4.0/3.0*pi*radio**3
print(volumen)
```

Al ejecutar el programa desde la línea de comandos, la computadora parece quedar bloqueada. No lo está, está en espera de que se introduzca el radio. El programa no es muy específico, ya que deja la computadora bloqueada hasta que el usuario teclee un número y no informa qué dato requiere. Vamos a hacer que el programa indique, mediante un mensaje, qué dato desea que se teclee. La función `input` acepta también un argumento, una cadena con el mensaje que se debe mostrar. Así que el programa será:

```
from math import pi
radio=float(input("Da el radio: "))
volumen=4.0/3.0*pi*radio**3
print(volumen)
```

Ahora, cada vez que se ejecute el programa, se mostrará en pantalla el mensaje. El usuario debe teclear el valor del radio, que va apareciendo en la misma consola.

Las cadenas pueden usarse también para mostrar textos en pantalla en cualquier momento a través de sentencias `print`. Modificando el programa anterior, da:

```
from math import pi
print "Programa para calcular el volumen de esfera"
radio=float(input("Dame el radio: "))
volumen=4.0\3.0*pi*radio**3
print(volumen)
```

Cuando se ejecute este programa, fíjate en que las cadenas que se muestran con `print` no aparecen entre comillas. El usuario del programa no está interesado en saber que le estamos mostrando datos del tipo cadena: sólo le interesa el texto de ellas.

Una sentencia `print` puede mostrar más de un resultado en una misma línea, basta con separar con comas todos los valores que se deseen mostrar. Cada una de las comas se traduce en un espacio de separación:

```
from math import pi
print("Programa para calcular el volumen de esfera")
radio=float(input("Dame el radio en metros: "))
volumen=4.0/3.0*pi*radio**3
print("Volumen de la esfera:", volumen, "m**3")
```

Con la sentencia `print`, se puede controlar, hasta cierto punto, la apariencia de la salida, aunque no se tiene un control de todo. Sin embargo, Python permite controlar con absoluta precisión la salida a la pantalla. Para ello se hace uso del operador % como sigue:

```
numero=int(input("Dame un numero"))
print("%d elevado a %d es %d" % (numero, 2, numero**2))
print("%d elevado a %d es %d" % (numero, 3, numero**3))
print("%d elevado a %d es %d" % (numero, 4, numero**4))
print("%d elevado a %d es %d" % (numero, 5, numero**5))
```

Cada una de las cuatro últimas líneas presenta este aspecto:

```
print(cadena % (valor, valor, valor))
```

La cadena aquí es especial, pues tiene unas marcas de la forma `%d`. Después de la cadena, aparece el operador %, que aquí combina una cadena, a su izquierda, con una serie de valores, a su derecha. Para entender claramente qué es lo que hacen las cuatro últimas líneas, ejecutemos el programa:

```
Dame un numero:3
3 elevado a 2 es 9
3 elevado a 3 es 27
3 elevado a 4 es 81
3 elevado a 5 es 243
```

Cada marca de formato `%d` en la cadena `"%d elevado a %d es %d"` ha sido sustituida por un número entero. El fragmento `%d` significa *aquí va un número entero*. ¿Qué número? El número que resulta de evaluar cada una de las tres expresiones que aparecen separadas por comas y entre paréntesis a la derecha del operador %.

El operador % especifica el formato de la cadena que se va a presentar en pantalla en ese punto. Se puede usar % para manipular más detalladamente la forma en que se muestra el número flotante de la cadena de caracteres de salida. Por ejemplo,

```
from math import pi
r = float(input("Dame el radio: "))
a = pi*radio**2
print("Area del circulo de radio %f= %f"%(r, a))
print("Area del circulo de radio %6.3f=%6.3f"%(r, a))
```

Ejecutando el programa se tiene:

```
Dame el radio: 2
El area del circulo de radio 2.00000 es 12.566371
El area del circulo de radio 2.000 es 12.566
```

La marca `%f` indica que ahí aparecerá un flotante. Se puede meter un número con decimales entre el % y la `f`. ¿Qué significa? Indica cuantas casillas se desea que ocupe el flotante y, de ellas, cuantas se quiere que ocupe los números decimales. En el segundo caso, se indica que el número ocupará seis lugares en la presentación del `print`, de los cuales tres serán dígitos de la parte decimal.

Se ha visto que hay marca de formato para enteros y flotantes. También hay una marca para cadenas: `%s`. El siguiente programa lee el nombre de una persona y la saluda:

```
nombre = input("Da tu nombre: ")
print "Hola, %s "% (nombre)
```

Da como resultado:

```
Da tu nombre: Roxana
Hola, Roxana
```

Además de preceder las líneas con # para comentar el código, podemos comentar bloques completos de líneas usando 3 comillas simples juntas; ''' para abrir el comentario y otras ''' para cerrar.

Más adelante, se harán programas más complicados, así que es muy importante aumentar la legibilidad de un programa intercalando comentarios que expliquen su finalidad o que aclaren sus pasajes más oscuros. Como esos comentarios sólo tienen por objeto facilitar la legibilidad de los programas para los programadores, pueden escribirse en el idioma que se quiera. Sin embargo, se recomienda que no se utilicen los acentos al escribir al español, ya que pueden causar problemas al momento de transferir el programa entre distintas máquinas.

2.4. Ejercicios

1. Evalúa el polinomio $x^4 + x^3 + 2x^2 - x$ en $x = 1.1$. Utiliza variables para evitar teclear varias veces el valor de x.

2. Da con el mayor detalle posible el resultado de evaluar las siguientes expresiones:

   - `(2+3)*1+2`
   - `(2+3)*(1+2)`
   - `1/2/4.0`
   - `4**0.5`
   - `4.0**(1.0/2)+1/2.0`
   - `3e3/10`
   - `10/5e-3 +1`
   - `True ==True != False`
   - `1<2<3<4<5`
   - `palindromo ="abcba"`
     `(4*"<"+ palindromo +">"*4)*2`
   - `"abalorio" <"abecedario"`
   - `124 < 13`
   - `"124"< "13"`
   - `str(2.1)+ str(1.2)`
   - `str(2+3)`
   - `str(int (2.1)+float(3))`

3. Evalúa las siguientes expresiones, primero en papel y luego en el intérprete de Python.

   a) `int(exp(2*log(3)))`

   b) `round(4*sin(3*pi/2))`

   c) `abs(log10(0.01)*sqrt(25))`

4. Escribe un programa que almacene en una lista los números del 1 al 10 y los muestre, en orden inverso, separados por comas.

5. Escribe un programa que almacene el abecedario en una lista, elimine de ésta las letras que ocupen posiciones que sean múltiplos de 3 y muestre la lista resultante.

6. Escribe un programa que pida al usuario una palabra y muestre si es un palíndromo.

7. Pide números y mételos en una lista, cuando el usuario meta un 0 ya dejaremos de insertar. Por último, muestra los números ordenados de menor a mayor.

8. Escribe un programa que pida al usuario una palabra y muestre un conteo de cada vocal.

9. Pide una frase por teclado, mete las palabras en una lista sin espacios.

10. Pide una cadena por teclado, mete los caracteres en una lista sin espacios.

11. Haz un programa que pida el nombre de una persona y lo muestre 1000 veces, pero dejando un espacio de separación entre cada copia del nombre.

12. Escribe un programa que guarde en una variable el diccionario 'Euro': 25.82, 'Dolar': 24.113, 'Yen': 0.2189,'Dolar Canadiense': 16.7842 y le pregunte al usuario por una divisa y muestre su valor de cambio o un mensaje de aviso si la divisa no está en el diccionario. Completa con todas las divisas que puedas.

13. Escribe un programa que pregunte al usuario su nombre, edad, dirección y teléfono y lo guarde en un diccionario.

14. Escribe un programa que almacene el diccionario con las calificaciones de las asignaturas de un semestre 'Algebra': 10, 'Física': 9, 'Calculo': 8,'Computacion': 10, 'Geometria': 9, y, después, muestre por pantalla las calificaciones de cada asignatura en el formato `asignatura` tiene `calificacion` , donde `asignatura` es cada una de las asignaturas del curso, y `calificacion` son sus calificaciones. Al final, debe mostrar también el número promedio del curso.

15. Diseña un programa que pida el valor de los tres lados de un triángulo y calcule el valor de su área y perímetro. Recuerda que el área A de un triángulo puede calcularse a partir de sus tres lados a,b y c: así $A = \sqrt{s(s-a)(s-b)(s-c)}$ donde $s = (a+b+c)/2$. Esta expresión es la fórmula de Herón de Alejandría.

16. Un satélite se encuentra en una órbita circular alrededor de la Tierra, tal que completa un círculo en T segundos; T es el periodo de la órbita:
i) Muestra que la altitud h sobre la superficie de la Tierra será:

$$h = \left(\frac{GMT^2}{4\pi^2}\right)^{\frac{1}{3}} - R$$

donde G es la constante de gravitación universal y tiene un valor $G = 6.67 \times 10^{-11}$ $\mathrm{m^3 kg^{-1} s^{-2}}$, M es la masa de la Tierra con valor $M = 5.97 \times 10^{24}$kg y R es el radio de la Tierra con valor $R = 6371$ kg.
(ii) Escribe un programa que le pregunte al usuario el periodo que desea que tenga su satélite girando alrededor de la Tierra y que le regrese la altitud, en metros, que deberá tener.
(iii) ¿Qué altitud deberá tener el satélite si queremos que orbite la Tierra con un periodo de un día (órbita geosíncrona), 90 minutos, 45 minutos? ¿Qué observas?
(iv) En práctica, una órbita geosíncrona es aquella cuya órbita tiene un periodo de un día sideral, que es de 23.93 horas, no de 24 horas. ¿Por qué es esto? ¿Cuál será la diferencia en altitud entre T= 23.93 horas y T= 24 horas?

Capítulo 3

Estructura básica de programación

Se dice que un programa de computadora es un conjunto de instrucciones que producirán la ejecución de una determinada tarea.

En general, el proceso de programación consiste en:

- Definir y analizar el problema a desarrollar.
- Diseñar el algoritmo.
- Codificar el programa.
- Depurar y comprobar el programa.

Un algoritmo es un procedimiento que describe, de manera inequívoca, una serie finita de pasos a seguir en un orden determinado. Su finalidad es implantar un procedimiento para resolver un problema o aproximar una posible solución.

En el proceso de diseño del algoritmo o en la codificación del programa se deben definir las acciones o instrucciones que resolverán el problema. Las acciones o instrucciones se deben escribir y posteriormente almacenar en la memoria en el orden a ser ejecutadas, es decir, escribirse de manera secuencial.

Se dice que un programa es lineal si las instrucciones se ejecutan secuencialmente, sin bifurcaciones, decisión ni comparaciones.

Un programa es no lineal cuando se interrumpe la secuencia mediante instrucciones de bifurcación.

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se

combinan. Sólo las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora, los programas que contienen errores de sintaxis son rechazados por la máquina. Más adelante se analizarán estos elementos.

3.1. Estructuras de control

Los programas que se han hecho hasta el momento presentan siempre una misma secuencia de acciones:

- Se piden datos al usuario, asignando a variables valores obtenidos con `input`.
- Se efectúan cálculos con los datos introducidos por el usuario, guardando el resultado en variables, mediante asignaciones.
- Se muestran en pantalla los resultados almacenados en variables, mediante la sentencia `print`.

Es posible alterar esta secuencia de ejecución de los programas para hacer que tomen decisiones a partir de los datos y resultados intermedios y/o ejecuten ciertas sentencias y otras no.

El primer tipo de alteración de la secuencia o flujo de control se efectúa con sentencias condicionales o de selección; el segundo tipo, con sentencias iterativas o de repetición. Las sentencias que permiten alterar el flujo de ejecución se engloban en las denominadas estructuras de *control de flujo*.

3.2. Sentencias condicionales: `if`

¿Cómo podemos hacer que cierta parte del programa se ejecute o deje de hacerlo de acuerdo a una condición determinada? Los lenguajes de programación convencionales presentan una sentencia especial cuyo significado es: *al llegar a este punto, ejecuta esta(s) acción(es) sólo si esta condición es cierta*. Este tipo de sentencia se denomina *condicional* o *selección* y en Python se escribe de la siguiente forma:

```
if condición:
        acción
```

Veamos un ejemplo. Diseñemos un programa para resolver cualquier ecuación de primer grado de la forma

$$ax + b = 0 \tag{3.1}$$

donde x es la incógnita. Los pasos a seguir serán:

Algoritmo:

1. Pedir el valor de a y b (dígase que son de tipo flotante).
2. Calcular el valor de x como $\frac{-b}{a}$.
3. Mostrar en pantalla el valor de x.

Código en Python:

```
a=float(input("Valor de a: "))
b=float(input("Valor de b: "))
x=-b/a
print("Solucion: ",x)
```

El programa no funciona bien cuando a vale 0, se produce un error de división entre cero. Para evitar la división entre cero, pondremos un condicional, el cual excluirá el caso en que a=0. Por lo que el programa será:

```
a=int(input("Valor de a: "))
b=int(input("Valor de b: "))
if a!=0:
    x=-float(b)/a
    print("Solucion: ",x)
if a==0:
    print("La ecuacion no tiene solución")
```

Analicemos las líneas del programa. En la línea 3 aparece la sentencia condicional **if** seguida de lo que debe ser una condición. La condición se lee fácilmente si se sabe que `!=` significa *distinto que*, por lo que la línea quiere decir *si a es distinto que 0*. La línea que empieza con `if` debe terminar obligatoriamente con dos puntos y se debe dejar una indentación[1] en la línea después de los

[1] Indentación: es un anglicismo (de la palabra inglesa *indentation*) de uso común en informática que significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores para separarlo del texto adyacente, lo que en el ámbito de la imprenta, se ha denominado siempre como sangrado o sangría.

dos puntos. Esta mayor indentación indica que la ejecución de estas dos líneas depende de que se satisfaga la condición `a!= 0`. Sólo cuando ésta sea cierta se ejecutan las líneas de mayor sangrado. Así, cuando a valga cero, estas líneas no se ejecutarán, evitando de este modo el error de división entre cero. Las dos últimas líneas empiezan con una sentencia condicional donde en lugar de `!=`, el operador de comparación utilizado es ==. La sentencia se lee *si a es igual a 0*.

Se debe notar la diferencia entre == y =. La segunda asigna a una variable un valor (puede ser una lista, cadena de caracteres, número flotante o entero, o un arreglo), mientras que la primera hace la comparación, es explícitamente *igual que o idéntico que*.

3.2.1. Sentencias condicionales anidadas

Las estructuras de control pueden anidarse, es decir, aparecer unas dentro de otras. El programa que se mostró como ejemplo quedaría:

```
a=float(input("Valor de a: "))
b=float(input("Valor de b: "))

if a!=0:
   x=-float(b)/a
   print ("Solución: ",x)
if a==0:
   if b!=0:
      print ("La ecuación no tiene solución")
   if b==0:
      print ("La ecuación tiene infinitas soluciones")
```

Recuerda que la indentación en Python determina de que sentencia depende cada bloque de sentencias.

Para afianzar los conceptos presentados, se presenta otro ejemplo para resolver ecuaciones de segundo grado de la forma

$$ax^2 + bx + c = 0 \tag{3.2}$$

¿Cuáles son los datos del problema? Los coeficientes a, b y c. ¿Qué se desea calcular? Los valores de x que hacen cierta la ecuación. Dichos valores son:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{3.3}$$

Un problema que debe considerar es la división entre cero que tiene lugar cuando a vale 0. Tratemos de evitar el problema de la división por cero del mismo modo que antes, pero mostrando un mensaje distinto, pues cuando a vale 0, la ecuación no es de segundo grado, sino de primer grado. Usaremos la función `sqrt` que existe en el modulo `math` para denotar la operación raíz cuadrada:

```
from math import sqrt
a=float(input("Valor de a: "))
b=float(input("Valor de b: "))
c=float(input("Valor de c: "))
if a!=0:
   x1=(-b+sqrt(b**2-4*a*c))/(2*a)
   x2=(-b-sqrt(b**2-4*a*c))/(2*a)
   print (" x1=%4.3f y x2=%4.3f" %(x1,x2))
if a==0:
   print ("No es una ecuación de segundo grado")
```

Hasta este punto, el código sigue incompleto, ya que falta considerar el caso complejo (cuando el argumento de la raíz cuadrada es cero) y otros detalles. Por lo que se necesitan otras estructuras para considerar los otros casos de la ecuación.

3.2.2. En caso contrario: `else`

Hasta el momento, se han desarrollado sentencias condicionales que conducen a ejecutar una acción `if` si se cumple una condición y a ejecutar otra si esa misma condición no se cumple:

```
if condición:
    acciones
if condición contraria:
    otras acciones
```

Este tipo de combinación es muy frecuente, hasta el punto de que se ha incorporado al lenguaje de programación una forma abreviada que significa lo mismo:

```
if condición:
    acciones
```

```
else:
    otras acciones
```

La palabra `else` significa *sino* o *en caso contrario*. Es muy importante que se respete la indentación: las acciones siempre un poco a la derecha y el `if` y el `else` alineados en la misma columna. De esta manera, el programa queda:

```
from math import *
a=float(input("Valor de  a: "))
b=float(input("Valor de  b: "))
c=float(input("Valor de  c: "))
if a!=0:
   x1=(-b+sqrt(b**2-4*a*c))/(2*a)
   x2=(-b-sqrt(b**2-4*a*c))/(2*a)
   print ("x1=%4.3f y x2=%4.3f" % (x1,x2))
else:
   if b!=0:
      x=-c/b
      print ("Solución de la ecuación: x=%4.3f" %x)
   else:
      if c!=0:
         print ("No tiene solución")
      else:
         print ("Tiene infinidad soluciones")
```

En este ejemplo aparecen sentencias condicionales anidadas en otras sentencias condicionales que, a su vez, están anidadas.

3.2.3. Forma compacta para estructuras condicionales: `elif`

Dentro de la versatilidad de Python, se encuentra la posibilidad de unir la estructura `else` e `if`, para usar un `elif`, lo cual permite ahorrar indentaciones y hacer el código más corto y simple.

A continuación se introducirá esta nueva estructura sintáctica planteando un nuevo problema. Imagínese que se tiene un programa que, a partir del radio de una circunferencia, calcula su diámetro, perímetro o área. Sólo se requiere mostrar al usuario una de las tres cosas. Véase este ejemplo:

```
from math import *
radio =float(input("Da el radio del circulo: "))
#Menu
print("Escoge una opción: ")
print("a) Calcular el diámetro")
print("b) Calcular el perímetro")
print("c) Calcular el área")
opcion = input("Teclea a, b o c)
if opcion =="a": #calculo del diametro
    diametro=2*radio
    print("El diametro es",diametro)
else:
    if opcion =="b": #calculo el perimetro
        perimetro=2*pi*radio
        print("El perimetro es", perimetro)
    else:
        if opcion =="c" #calculo de area
            area=pi*radio**2
            print("El area es", area)
```

En el programa anterior se presenta un problema estético, ya que este programa es difícil de entender por la indentación. Cada opción aparece indentada más a la derecha que la anterior, así que el cálculo del área acaba con tres niveles de sangrado. ¿Qué ocurrirá si se tiene que elaborar un menú muy complicado con muchas opciones? El sangrado terminaría tan a la derecha, que leer el programa se complicaría. Python permite una forma muy compacta de expresar fragmentos de código de la siguiente forma:

```
if condición:
    ...
    ...
else:
    if condición:
    ...
```

Si se tiene un `else` inmediatamente seguido por un `if`, todo se puede escribir de la siguiente manera:

```
if condicion:
    ....
elif otra condición:
    ....
```

Por lo que se ahorra un espacio de sangrado. El último programa se convierte en:

```
from math import *
radio =float(input("Da el radio del circulo: "))
#Menu
print("Escoge una opción: ")
print("a) Calcular el diámetro")
print("b) Calcular el perímetro")
print("c) Calcular el área")
opcion = input("Teclea a, b o c)
if opcion =="a": # calculo del diametro
    diametro=2*radio
    print("El diametro es",diametro)
elif opcion =="b": # calculo el perimetro
    perimetro=2*pi*radio
    print("El perimetro es", perimetro)
elif opcion =="c" # calculo de area
    area=pi*radio**2
    print("El area es", area)
else:
    print("Solo hay tres opciones a,b o c")
```

Otro ejemplo más sencillo del uso de condicionales es el siguiente:

```
x=-1
if x<0:
    print "Negativo"
elif x<5:
    print "Pequeno"
else:
    print "Grande"
```

Veamos como se aplica esto al ejemplo de la ecuación cuadrática:

```
from math import *
a=float(input("Valor de  a: "))
b=float(input("Valor de  b: "))
c=float(input("Valor de  c: "))
if a!=0:
```

```
    x1=(-b+sqrt(b**2-4*a*c))/(2*a)
    x2=(-b-sqrt(b**2-4*a*c))/(2*a)
    print ("x1=%4.3f y x2=%4.3f" % (x1,x2))
elif b!=0:
    x=-c/b
    print ("Solucion de la ecuacion: x=%4.3f" %x)
    elif c!=0:
        print ("La ecuacion no tiene solución")
else:
    print ("La ecuación tiene infinidad soluciones")
```

Hasta el momento, sólo se ha modificado la primera estructura de control, la cual nos da la selección de casos. Pero, para hacer programas más útiles, se requieren más estructuras de control.

3.3. Sentencias iterativas

3.3.1. La sentencia `while`

La primera estructura de iteración que se verá aquí es la sentencia `while`. En inglés, *while* significa *mientras*. Con la estructura mientras, le decimos que repita una acción, una y otra vez, hasta que se cumpla la condición. En carreras científicas, los alumnos están muy acostumbrados a estas estructuras cuando realizan el estudio de series y sumatorias. En Python, la sintaxis de la sentencia `while` es:

```
while condicion:
    acción
    acción
    ...
    acción
```

En Python, *while* tiene el significado *mientras se cumpla esta condición, repite estas acciones*. Las sentencias que denotan estas repeticiones se denominan *bucles* o *ciclos*. Veamos un ejemplo:

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

```
[Out]  1
   2
   3
   4
   5
   6
   7
   8
   9
   10
```

En este ejemplo, se crea una variable llamada `contador` que va avanzando de uno en uno hasta 10 y se imprime en pantalla. Es una manera interesante para poner a contar a Python del 1 al 10. Los bucles son muy útiles a la hora de confeccionar programas, pero también son engañosos, ya que se corre el riesgo que no finalicen nunca. Por ejemplo:

```
contador=0
while contador<10:
     print(contador)
```

```
[Out]  0
   0
   0
   0
   0
   ...
```

La condición del bucle siempre se satisface, ya que dentro del bucle nunca se modifica el valor de `contador` y si `contador` no se modifica, entonces jamás llegará a 10. La computadora empieza a mostrar el número 0 una y otra vez, sin finalizar nunca. Se tiene entonces un ciclo infinito. Cuando se ejecuta un bucle sin fin, el proceso queda suspendido y no devuelve el control. Normalmente se puede recuperar pulsando `ctrl+c` o matando el kernel de Jupyter, según sea el caso.

Ahora que se ha presentado lo fundamental de los bucles, resolvamos algunos ejemplos concretos. Hagamos un programa que calcule la suma de los 100 primeros números, es decir, un programa que calcule la sumatoria de Gauss:

$$\sum_{i=1}^{100} i \tag{3.4}$$

El programa queda de la siguiente manera:

```
sumatoria=0
i=0
while i<=100:
    sumatoria += i
    i+=1
print("La suma es",sumatoria)
```

```
[Out] La suma es 5050
```

En este ejemplo `sumatoria`, depende de como cambie `i` para ir sumando sobre este valor. Si deseamos sumar los primeros 100 números pares, modificaremos el contador, nombrado, en este código, como `i`:

```
sumatoria=0
i=0
while i<=100:
     sumatoria += i
     i+=2 #avanzamos i de dos en dos
print("La suma es",sumatoria)
```

```
[Out] La suma es 171700
```

Construyamos un algoritmo que encuentre la raíz cuadrada de un número[2]. Para empezar el algoritmo, se toma un segmento que va de 0 a y. Luego cortamos a la mitad este segmento y y este nuevo punto lo llamamos x_0. De nuevo cortamos el intervalo que va de 0 a x_0 e iteramos este proceso. Este método se puede generalizar y llegar al método de bisección. Es decir,

[2]Este algoritmo era conocido por los babilonios, si el lector quiere profundizar en las matemáticas babilónicas, se le recomiendan dos lecturas [4] y [5].

$x_0 = y/2$, está mitad será menor que la raíz cuadrada de y, es decir, $x_0 < \sqrt{y}$. Jugando un poco con los intervalos:

$$\begin{array}{rcl} x_0 & < & \sqrt{y} \\ \dfrac{x_0}{y} & < & \dfrac{\sqrt{y}}{y} \\ \dfrac{y}{x_0} & > & \dfrac{y}{\sqrt{y}} = \sqrt{y} \end{array}$$

Con esta desigualdad, proponemos un intervalo que se aproxime mejor a la raíz cuadrada que la mitad del intervalo $x_0 = y/2$, la propuesta será $x_1 = \frac{1}{2}(x_0 + \frac{y}{x_0})$. Este procedimiento será repetido hasta encontrar la raíz, por lo que el paso $(n+1)$ estará determinado por el resultado del paso n de la forma:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right) \tag{3.5}$$

Este algoritmo fue inventado por los babilonios, hagamos paso por paso:

$$\begin{array}{rcl} y & = & \text{número del que queremos obtener la raíz} \\ x_0 & = & y/2. \\ x_1 & = & \dfrac{1}{2}(x_0 + \dfrac{y}{x_0}) \\ x_2 & = & \dfrac{1}{2}(x_1 + \dfrac{y}{x_1}) \\ \vdots & = & \vdots \\ x_n & = & \dfrac{1}{2}\left(x_{n-1} + \dfrac{y}{x_{n-1}}\right) \\ x_{n+1} & = & \dfrac{1}{2}\left(x_n + \dfrac{y}{x_n}\right) \end{array}$$

Lo más interesante de este algoritmo, es que los babilonios conocían $\sqrt{2}$. Veamos los primeros cuatro pasos del algoritmo para encontrar $\sqrt{2}$. Empezamos con un segmento de tamaño 2:

$$\begin{aligned} y &= 2 \\ x_0 &= 1 \\ x_1 &= \frac{1}{2}(x_0 + \frac{y}{x_0}) = \frac{1}{2}(1 + \frac{2}{1}) = \frac{3}{2} = 1.5 \\ x_2 &= \frac{1}{2}(x_1 + \frac{y}{x_1}) = \frac{1}{2}(\frac{3}{2} + \frac{2}{3/2}) = \frac{17}{12} \approx 1.41666 \\ x_3 &= \frac{1}{2}(x_2 + \frac{y}{x_2}) = \frac{1}{2}(\frac{17}{12} + \frac{2}{17/12}) = \frac{577}{408} \approx 1.41421569 \end{aligned}$$

Como se puede ver, este algoritmo encuentra la raíz del número y con una precisión de cinco cifras significativas en cinco pasos, es extremadamente eficiente. Calculemos la raíz cuadrada de un número flotante dado por el usuario en 10 pasos. El código quedará de la siguiente forma:

```
#raices cuadradas con método babilónico
y=float(input("Da el numero y: "))
x=1.0
i=0
while i<10:
    x_nuevo=1./2.*(x+(y/x))
    print(i, x_nuevo)
    x=x_nuevo
    i+=1
```

Algo que es notorio es ver que el programa converge sumamente rápido. El algoritmo tiene una convergencia cuadrática, es decir, la convergencia se duplica a cada paso. Es un algoritmo muy sencillo, por lo que es muy eficiente.

Otro ejemplo bastante sencillo que muestra el poder de la sentencia `while` es la serie de Fibonacci, que regresa la secuencia de números 0, 1, 1, 2, 3, 5, 8, 13, 21:

```
a, b = 0, 1
while b < 30:
    print(b)
    a,b = b, a+b
```

```
[Out] 1
      1
      2
      3
      5
      8
      13
      21
```

Notase que en la primer y última línea se utiliza la asignación simultánea de dos variables.

Otra variación muy interesante de la estructura `while` es incluir la sentencia `break`, la cual nos permite romper abruptamente el ciclo. Un ejemplo muy requerido en cómputo científico es crear un menú, por lo cual usaremos `while-break`:

```
print('Menu')
while True:
    opcion=int(input('Escribe la opción que gustes, para
    finalizar usa 0\t'))
    if opcion==1:
        print('Primera opción')
    elif opcion==2:
        print('Segunda opción')
    elif opcion==0:
        print('Adios')
        break
    else:
        print('da una opción valida')
```

Esta estructura es un poco truculenta. Usamos un bucle infinito, que el usuario romperá hasta que entre a la opción que incluye `break`.

3.3.2. Bucle `for`

Hay otro tipo de bucle en Python, el bucle `for`, que se puede interpretar como *para todo elemento de una serie, hacer una acción*. Un bucle `for` presenta el siguiente aspecto:

```
for variable in serie de valores:
   acción
   acción
   ....
```

Véase el siguiente ejemplo:

```
numero=int(input("Da un número"))
print("%d elevado a %d es %d" % (numero, 2, numero**2))
print("%d elevado a %d es %d" % (numero, 3, numero**3))
print("%d elevado a %d es %d" % (numero, 4, numero**4))
print("%d elevado a %d es %d" % (numero, 5, numero**5))
```

Con el bucle `for` se puede resumir este programa:

```
numero=int(input("Da un entero"))
for potencia in [2,3,4,5]:
      print("%d elevado a %d es %d" \
                  %(numero, potencia, numero**potencia))
```

El bucle se lee de forma natural de la siguiente manera: *para toda potencia de la serie de valores 2,3,4 y 5, haz la siguiente acción.* La diagonal inversa al final de la línea 3, indica que la orden o sentencia en Python continua en la línea de abajo. Normalmente se pone una lista para indicar los valores necesarios, sin embargo, si queremos aplicar este código a una lista mayor, digamos `l=[0,1,2,3,4,5,6,7,8,9,10]` o una lista que tenga elementos del 0 al 100, tendremos un proceso muy tedioso. Por esto, es muy útil usar la función `range`, la cual tiene la sintaxis `range(valor_inicial,valor_final, tamaño_paso)`. La función `range` toma el intervalo semi-abierto $[valor\_inicial, valor\_final)$ para generar la lista, es decir, no incluye el último valor.

Por ejemplo:

```
lista=range(0,10,1)
print(list(lista))
```

```
[Out] [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La función `range` devuelve una variable que tiene un rango con la asignación de dos valores default para `valor_inicial=0` y `tamaño_paso=1`. La función `list()` hace la variable en lista, siempre y cuando esté en condiciones adecuadas de ser una lista. Esta función es análoga a `string()` o `float()`.

Por ejemplo:

```
list(range(10))
```

```
[Out] [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(3,7))
[3, 4, 5, 6]
```

```
[Out] [3, 4, 5, 6]
```

De esta manera podemos escribir en el bucle `for`:

```
  for i in range(1,6):
        print i
```

```
[Out] 1
    2
    3
    4
    5
```

Al ejecutar el programa se verá en la pantalla los números del 1 al 5. La lista que devuelve `range` es usada por el bucle `for` como una serie de valores que recorrer.

Con esta nueva estructura, nuestros códigos anteriores quedan más compactos, debido a que el contador queda en la misma línea que la declaración de iteración o bucle. Veamos como queda la sumatoria de Gauss con la estructura de control `for`:

```
sumatoria=0
for i in range(0,11):
    sumatoria+=i
    print(sumatoria)
```

```
[Out] 0
    1
    3
    6
    10
    15
    21
    28
    36
    45
    55
```

3.3.3. Diferencias entre el bucle `for` y el bucle `while`

Cabe destacar la diferencia entre los bucles `for` y `while`. El bucle `while` es un bucle que no se sabe cuando va a terminar, es decir, no se conoce el número de pasos que tomará. En cambio, en el bucle `for` se especifica, desde el comienzo, cuál será el número de pasos que tomará el bucle.

3.4. Números pseudoaleatorios

Los números completamente aleatorios (no deterministas) son fáciles de imaginar conceptualmente, por ejemplo, podemos imaginar lanzar una moneda, lanzar un dado o la lotería. Quizás se hayan preguntado cómo las máquinas predecibles, como las computadoras, pueden generar aleatoriedad. En realidad, la mayoría de los números aleatorios utilizados en los programas de computadora son pseudoaleatorios, lo que significa que se generan de manera predecible utilizando una fórmula matemática.

Un número pseudoaleatorio es un número generado en un proceso que parece producir números al azar, pero no lo hace realmente. Las secuencias de números pseudoaleatorios no muestran ningún patrón o regularidad aparente desde un punto de vista estadístico, a pesar de haber sido generadas por un algoritmo completamente determinista en el que las mismas condiciones iniciales producen siempre el mismo resultado. La condición inicial se llama semilla (*seed*).

El Mersenne twister es un generador de números pseudoaleatorios desarrollado en 1997 por Makoto Matsumoto y Takuji Nishimura muy respetado por su calidad.

Su nombre viene del hecho de que la longitud del periodo corresponde a un número primo de Mersenne. Existen al menos dos variantes de este algoritmo, distinguiéndose únicamente en el tamaño de primos Mersenne utilizados. El más reciente y más utilizado es el Mersenne Twister MT19937, con un tamaño de palabra de 32-bit. Existe otra variante con palabras de 64 bits, el MT19937-64, la cual genera otra secuencia.

Un número de Mersenne es un número entero positivo M_n que es una unidad menor que una potencia entera positiva de 2:

$$M_n = 2^n - 1$$

A mayo de 2021, solo se conocen 51 números primos de Mersenne, siendo el mayor de ellos $M_{82589933} = 2^{82589933} - 1$, un número de más de 24 millones de cifras. El número primo más grande que se conocía en una fecha dada casi siempre ha sido un número primo de Mersenne.

Empecemos con la generación de números pseudoaleatorios en Python. Importaremos la librería `random` la cual tiene cargadas muchas funciones para la generación de números pseudoaleatorios obtenidos por el algoritmo de Mersenne Twister.

```
from random import *
for i in range(10):
    print(i,random())
#Generamos un números pseudoaleatorio entre 0 y 1, flotante
```

```
[Out]   0 0.47214271545271336
   1 0.1007012080683658
   2 0.4341718354537837
   3 0.6108869734438016
   4 0.9130110532378982
   5 0.9666063677707588
   6 0.47700977655271704
   7 0.8653099277716401
   8 0.2604923103919594
   9 0.8050278270130223
```

Como se puede apreciar, la función `random()` genera un número pseudoaleatorio entre 0 y 1 de tipo flotante. En el ejemplo anterior, se repitió la función `random()` diez veces para ver el resultado. Se sugiere al lector que corra en varias ocasiones el código y vea que cada vez se obtienen resultados diferentes. Si en lugar de obtener valores flotantes se requiere tener números pseudoaleatorios enteros dentro de un intervalo, se puede usar la función `randint(a,b)`, donde $[a, b]$ es el intervalo en cuestión:

```
from random import *
for i in range(10):
    print(i,randint(1,10))
```

```
[Out]   0 4
   1 2
   2 7
   3 7
   4 3
   5 8
   6 5
   7 9
   8 4
   9 4
```

Esta función `randint()` es bastante útil, por ejemplo, puede servir para hacer un sorteo. En el siguiente ejemplo, se ve un sorteo de regalos repetido 5 veces, cada vez se da un regalo obtenido de forma aleatoria.

```
from random import *
regalos = ['sartén', 'jamón', 'mp4', 'muñeca', 'tv',
           'patín', 'balón', 'reloj', 'bicicleta', 'anillo']
for sorteo in range(5): #el sorteo se hace 5 veces
    regalo = regalos[randint(0, 9)] # encuentro el i-esimo
    elemento i=aleatorio
    print('Sorteo', sorteo + 1, ':', regalo)
```

```
[Out]   Sorteo 1 : sartén
   Sorteo 2 : reloj
   Sorteo 3 : anillo
   Sorteo 4 : reloj
   Sorteo 5 : sartén
```

En este ejemplo se encuentra un elemento pseudoaleatorio de la lista `regalos`.

También la función `randint()` permite generar números pseudoaleatorios negativos:

```
for numero in range(10):
    print(randint(-3, 3))
```

```
[Out]   0
   -2
   3
   -3
   -3
   -3
   0
   0
   1
   1
```

Por otro lado, la función `randrange()` devuelve números enteros, que van desde un valor inicial a otro final, separados entre sí un número de valores determinados. Esta separación, o paso, se utiliza, en primer lugar, con el valor inicial para calcular el siguiente valor y los sucesivos hasta llegar al valor final o al más cercano posible.

Otro ejemplo sería obtener 25 números pseudoaleatorios que van desde el 3 al 16, sin incluir el número 16. Es decir, usar la lista `[3, 6, 9, 12, 15]` y obtener 25 números pseudoaleatorios dentro de ésta:

```
print('\n Valores posibles: 3, 6, 9, 12, 15')
for i in range(25):
    print(randrange(3, 16, 3), end='\t')
```

```
[Out]
   Valores posibles: 3, 6, 9, 12, 15
   15 6 9 9 12 9 12 12 3 3 9 12 9 12 6 9 3 9 15 6 15 12 3 6 3
```

Otra función muy útil es `uniform()`, la cual recibe dos valores que definen un intervalo y devuelve un número tipo `float` incluido entre los valores indicados:

```
for numero in range(3):
    print(uniform(100, 105), end=' ')
```

```
[Out] 103.38460483408 100.59455143276 101.98976800801
```

Si se quisiera truquear o hackear el sorteo y obtener la misma secuencia de números pseudoaleatoria, se puede utilizar la función `seed()`, que fija mediante una semilla, i.e., fija el mismo comienzo en cada secuencia, permitiendo con ello obtener series con los mismos valores.

A continuación, se muestra un ejemplo donde se realizan dos series de cinco sorteos y en cada serie se obtienen los mismos regalos en el mismo orden.

La semilla, en este caso, se fija con un valor numérico, 1 en esta situación, pero también se puede utilizar una cadena o una unidad de tiempo obtenida con la función `time()` del módulo `time`; o incluso, puede expresarse con cualquier objeto `hashable` de Python.

```
from random import *
regalos = ['sartén', 'jamón', 'mp4', 'muñeca', 'tv',
           'patín', 'balón', 'reloj', 'bicicleta', 'anillo']
for serie in range(2): #repetición de la serie de sorteo
    print('\nserie:', serie + 1)
    seed(1) #iniciando la sucesión con M0=1
    for sorteo in range(5):
        regalo = regalos[randint(0, 9)]
        print('Sorteo', sorteo + 1, ':', regalo)
```

```
[Out]
   serie: 1
   Sorteo 1 : mp4
   Sorteo 2 : anillo
   Sorteo 3 : jamón
   Sorteo 4 : tv
   Sorteo 5 : jamón

   serie: 2
   Sorteo 1 : mp4
   Sorteo 2 : anillo
   Sorteo 3 : jamón
```

```
Sorteo 4 : tv
Sorteo 5 : jamón
```

Se recomienda al lector que modifique el valor de `seed()` y vea los distintos resultados:

```
s=0
while 1:
    x=randrange(0, 20, 1)
    s+=x
    if s>=150:
        break
    print(x,s)
```

```
[Out]   11 11
   7 18
   7 25
   14 39
   9 48
   0 48
   13 61
   17 78
   3 81
   5 86
   9 95
   3 98
   10 108
   16 124
   13 137
```

La función `choice()` se utiliza para seleccionar elementos al azar de una lista. En el siguiente ejemplo se obtiene un elemento de los cinco opciones existentes en una lista:

```
transporte = ['bici', 'moto', 'coche', 'tren', 'avión']
print('Hoy viajaré en:', choice(transporte))
```

```
[Out]   Hoy viajaré en: tren
```

La función `shuffle()` cambia o mezcla aleatoriamente el orden de los elementos de una lista antes de realizar la selección de alguno de ellos. Esta mezcla recuerda, en el caso de los juegos de cartas, la acción de barajar un número de veces antes de repartir o seleccionar. En el siguiente ejemplo, se realizan dos mezclas antes de que se obtenga el elemento:

```
lista = ['rojo', 'verde', 'amarillo']

shuffle(lista)
print('mezcla1', lista)

shuffle(lista)
print('mezcla2', lista)
print(lista[randint(0,2)])
```

```
[Out]   mezcla1 ['amarillo', 'rojo', 'verde']
   mezcla2 ['rojo', 'amarillo', 'verde']
   amarillo
```

Con todo lo que se ha practicado, se concluye con un ejemplo más interesante. A continuación, se muestra un programa en el que el usuario puede jugar *piedra, papel o tijera* contra la computadora. Para hacer esto, se usa una lista `l` que contiene los elementos que se escogen para jugar. Con la variable `opción`, se hace una selección aleatoria de un elemento de la lista usando `choice`, luego se comparan los resultados para definir el ganador. El código queda de la siguiente forma:

```
from random import choice
x=input('Escoge: entre piedra, papel o tijera:\t')
l=['piedra','papel','tijera']
opcion=choice(l) #selecciona computadora
print(x,opcion)
if x==opcion:
    print('Tu: %s, yo: %s, hay empate' %(x, opcion))
if x!=opcion:
    if x=='papel':
        if opcion=='piedra':
            print('Tu ganas')
        else:
            print('yo gano')
    if x=='tijera':
        if opcion=='piedra':
```

```
            print('yo gano')
        if opcion=='papel':
            print('tu ganas')
    if x=='piedra':
        if opcion=='tijera':
            print('tu ganas')
        if opcion=='papel':
            print('yo gano')
```

```
[Out]   Escoge: entre piedra, papel o tijera: tijera
   tijera papel
   tu ganas
```

3.5. Funciones

Al resolver un problema muy complicado, lo más conveniente siempre es dividirlo y convertirlo en muchos subproblemas más simples. Lo mismo sucede con la programación. Para encontrar los errores fácilmente, es más sencillo cortar el código, tal que se tengan muchas funciones que hagan pequeñas porciones del problema. Si se necesita hacer una modificación al programa, se puede modificar sólo la sección que se requiere sin tener que realizar mayores cambios al programa entero. Suele resultar mejor dividir un programa en pequeños subprogramas o módulos, que escribir un programa completo de forma corrida. A esto se le llama programación modular.

Para poder dividir el programa en módulos, es necesario introducir el concepto de **función**. Al igual que en el lenguaje matemático, a una función se le da información o variables para que devuelva algo o simplemente se le asigna alguna parte del trabajo por hacer. Para poder usar una función en Python, es necesario definirla primero. Para ello usamos `def-return`:

Sintaxis:

```
def nombre_funcion(varible_entrada):
    acción a realizar
    return variable_salida
```

Veamos un ejemplo, en el cual definimos la función cuadrado, que recibe como variable de entrada una x y regresa una variable de salida x^2:

```
>>> def cuadrado(x):
...     return x*x
...
>>> cuadrado(4)
16
>>> cuadrado(2)
4
>>> cuadrado(27)
729
>>> a=1+cuadrado(3)
>>> a
10
```

La sintaxis consiste en declarar que definiremos una función usando el comando `def`, luego decimos su nombre, que en este caso será `cuadrado`. Esta función tendrá variables de entrada, las cuales serán definidas por el usuario al llamar a la función. La primera línea de la declaración de la función será terminada con dos puntos en donde indicaremos a Python que continuará esa estructura. En el cuerpo de la función, pondremos lo que se hará. Para finalizar la función, tenemos que decirle explícitamente a Python que es lo que regresará esta función, es decir, especificar la variable de salida.

La función puede no requerir de ningún valor o requerir de muchos, dependiendo de cual sea el uso que se le quiera dar. Además, dentro de ella se pueden agregar cualquier número de líneas, siempre y cuando estén indicadas por el sangrado adecuado.

Ahora se hará una función que reciba dos listas y que regrese el producto punto de ambas:

```
def punto(a,b):
    c=0
    if len(a)!=len(b):
        return("Error: Listas con distinta dimensión")
    else:
        for i in range(len(a)):
            c+=a[i]*b[i]
        return c

a=[3,7]
b=[1,6]
print(punto(a,b))
```

Lo primero que hacemos es medir el tamaño de las listas. La medición de las listas se hace con la función `len(lista)`. Si las listas no tienen la misma dimensión, entonces se regresará un error. En caso contrario, las listas tienen el mismo tamaño, entonces hacemos el producto punto.

Recordemos que el producto punto de dos vectores en $\mathbf{R^2}$ es $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y = a_1 b_1 + a_2 b_2$, en $\mathbf{R^3}$ será $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z = a_1 b_1 + a_2 b_2 + a_3 b_3$ y en $\mathbf{R^n}$ será $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{i=1}^{n} a_i b_i$

También podemos usar una función de función, esto se debe a que Python recibe una variable en la función y no sabe que tipo de variable o función es hasta que se usa. Veamos un ejemplo:

Una función compuesta indica la función f compuesta de una función g y sigue la regla de composición $(f \circ g)(x) = f(g(x))$. Si $f(x) = 3x - 1$ y $g(x) = x^3 + 2$, evaluar $(f \circ g)(10) = f(g(10))$:

```
def f(x):
    return (3*x-1)

def g(y):
    return y**3+2

def fog(f,g,z):
    variable=g(z)
    return f(variable)

print(fog(f,g,10))
```

La función `fog(f,g,z)` recibe dos funciones, las que sean, así como la variable a operar y regresa la composición de funciones. Es importante notar que z solo existe en la función `fog(f,g,z)`. Si se trata de llamar z fuera de la función, Python regresará un error, indicando que no conoce dicha variable. A este tipo de variables se les llama **variable local**, ya que existen en partes locales de código. Una **variable global** vivirá a lo largo de todo el programa.

3.5.1. `Yield`

El comando `yield` es una orden muy similar al comando `return`, ya que se usa para asignar la salida de una función. La gran diferencia que existe entre `yield` y `return` es que `yield` pausa la ejecución de la función y guarda el resultado hasta que se use de nuevo.

Ejemplo, una función tradicional con `return` es:

```
def cuadrados(lista_numero):
    return [numero*numero for numero in lista_numero]
print(cuadrados([1,2,3,4,5]))
```

```
[Out]   [1, 4, 9, 16, 25]
```

De esta forma, todos los cuadrados de los números se generan de una vez, pero, si la lista que se evalúa en la función es muy grande, el costo computacional será mucho, por lo que conviene pensar en otra forma. Lo mejor es calcular los cuadrados de los números uno por uno, por lo que se usará el comando `yield`:

```
def cuadrados(lista_numero):
    for numero in lista_numero:
        yield numero*numero

print(cuadrados([1,2,3,4,5]))

# podemos iterar sobre el generador
# si ejecutamos
for cuadrado in cuadrados([1,2,3]):
    print(cuadrado)
```

```
[Out]<generator object cuadrados at 0x000002560CB93200>
1
4
9
```

Como se puede ver, cuando se usa, imprime la función evaluada; no se genera la lista resultante, sino aparece el mensaje de salida `<generator object cuadrados at 0x000002560CB93200>`, el cual indica que tienen valores iterables guardados. Para obtener los valores de la función, se necesita usar un `for` extra y especificar los valores a evaluar. De esta forma, solamente calcularemos los cuadrados que se necesitan.

El comando `yield` es muy útil, ya que puede generar secuencias infinitas que, obviamente, no se pueden calcular con antelación. Por ejemplo, calcular todos los números naturales hasta 1000:

```
def numeros_naturales():
    n = 1
    while True:
        yield n
        n += 1

for natural in numeros_naturales():
    print(natural)
    if natural>999:
        break
```

3.6. Recursividad

Hasta el momento, sólo hemos usado iterar procesos para encontrar la solución de series y sucesiones. De esta manera, podemos definir la iteración como un proceso en el que se repite una operación durante un número determinado de veces. Normalmente, la salida de una iteración es la entrada del siguiente paso de la iteración. Por otro lado, existe una manera más ingeniosa de proceder, esto es la recursividad o recursión. La recursividad es la forma de expresar la solución de un problema en términos de sí mismo. Un ejemplo interesante y no computacional sobre recursión es la película *Inception* o *El origen*, en donde se muestra a los personajes entrando en un sueño dentro de otro sueño de manera recursiva. Para aclarar recursividad, veamos el ejemplo de la suma de Gauss.

Según cuentan las biografías de Carl Friedrich Gauss, cuando era un niño, su maestro J.B. Büttner castigó a todos los niños del salón, haciéndolos que sumaran del 1 al 100. Según esto, Gauss resolvió el problema muy rápidamente dejando atónitos a los presentes. La idea genial del niño Gauss fue encontrar una expresión cerrada para dicha suma.

$$\text{gauss}(n) = \sum_{i=0}^{n} i = 0 + 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Veamos con detalle esta expresión para deducir su forma recursiva:

$$
\begin{aligned}
\text{gauss}(5) &= & 0+1+2+3+4+5 &= & \text{gauss}(4)+5\\
\text{gauss}(4) &= & 0+1+2+3+4 &= & \text{gauss}(3)+4\\
\text{gauss}(3) &= & 0+1+2+3 &= & \text{gauss}(2)+3\\
\text{gauss}(2) &= & 0+1+2 &= & \text{gauss}(1)+2\\
\text{gauss}(1) &= & 0+1 &= & \text{gauss}(0)+1\\
\text{gauss}(0) &= & 0 & &
\end{aligned}
$$

Entonces, podemos deducir la forma recursiva de la siguiente forma:

$$\text{gauss(n)} = \text{gauss(n} - 1) + \text{n}$$

Por lo que tenemos tres formas distintas para programar la suma de Gauss:

```
def gauss_directa(n):
    return (n*(n+1))//2

def gauss_iterativa(n):
    s=0
    for i in range(n+1):
        s=s+i
    return (s)

def gauss_recursivo(n):
    if n==0: #empezamos con el caso trivial
        return 0
    else:
        return gauss_recursivo(n-1)+n

print(gauss_directa(100))
print(gauss_iterativa(100))
print(gauss_recursivo(100))
```

Incluimos el caso trivial como la base para hacer recursión.

3.6.1. Memoisación

Por si solo, la recursividad no es tan eficiente, sin embargo, existen procedimientos que hacen que la recursión sea una de las herramientas más poderosas en la programación. Una de estas herramientas se llama, en inglés, *Memoising*, en español no existe una palabra relacionada o equivalente, por lo que usaremos el anglicismo memoisación para referirnos a esta técnica. La memoisación es una técnica para implementar programación dinámica y hacer que los algoritmos recursivos sean eficientes. A menudo, tiene los mismos beneficios que la programación dinámica regular sin requerir cambios importantes en el algoritmo recursivo más natural original.

La idea principal para la eficiencia de la recursión consiste en que si las llamadas recursivas con los mismos argumentos se hacen repetidamente, entonces el algoritmo recursivo ineficiente se puede memoisar guardando estas soluciones de subproblemas en una tabla para que no tengan que ser recalculadas.

Para implementar la memoisación de algoritmos recursivos, se mantiene una tabla con soluciones de subproblemas, la estructura de control para completar la tabla ocurre durante la ejecución normal del algoritmo recursivo. Esto se puede resumir en pasos:

1. En memoisación, tendremos una entrada en una tabla o arreglo para la solución de cada subproblema.

2. Cada entrada de la tabla, inicialmente, contendrá un valor especial para indicar que la entrada aún no se ha completado.

3. Cuando se encuentra por primera vez el subproblema, su solución se calcula y se almacena en la tabla.

4. Posteriormente, el valor se busca en lugar de calcularse.

Veamos el ejemplo de la sucesión de Fibonacci:

```
def fib_rec(n):
    if n==0 or n==1:
        return n
    else:
        return fib_rec(n - 1) + fib_rec(n - 2)
for i in range(11):
    print(fib_rec(i))
```

Si hacemos este algoritmo hasta 40, podemos ver lo lento que es, ya en el valor 30 el programa empieza a tardar bastante. Sin embargo, si usamos memoisación, haremos más eficiente el algoritmo:

```
def memoisacion(f):
    memo = {} #diccionario vacio
    def auxiliar(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return auxiliar
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib = memoisacion(fib)
for i in range(41):
    print(i,fib(i))
```

No usaremos una tabla como tal, sino usaremos un diccionario para guardar la información. Fíjate como definimos una función dentro de otra función. La función auxiliar guarda los nuevos elementos de la función en el diccionario, el tiempo de cómputo se reduce drásticamente. La ventaja es que nuestra función `memoisación` la puedes usar para cualquier estructura recursiva. Probemos con la suma de Gauss:

```
def memoisacion(f):
    memo = {} #diccionario vació
    def auxiliar(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return auxiliar
def gauss_recursivo(n):
    if n==0: #empezamos con el caso trivial
        return 0
    else:
        return float(gauss(n-1)+n)

gauss = memoisacion(gauss_recursivo)
for i in range(5000):
    print(i,gauss_recursivo(i))
```

Como podemos ver, la memorización simplemente guarda los datos en tablas o diccionarios, de esta manera, cambiamos menos pasos de CPU por más memoria.

3.7. Ejercicios

1. Construye un programa que diga si un número entero es o no primo. *Hint: Determina si un número i divide a otro número z. Cambia el valor de una variable dentro de un ciclo cuando encuentres un divisor. De esta forma, al terminar el ciclo sabrás si el número z es primo o no.*

2. Elabora un programa que muestre los primeros N números primos. N es solicitado al usuario.
   Hint: Usa el programa que elaboraste en el ejercicio anterior para analizar cada uno de los números naturales, cada vez que encuentres un número primo, muéstralo e incrementa un contador que te servirá como condición de paro en un ciclo `while`.

3. Haz un programa que pida el valor de dos enteros, n y m, y que muestre en pantalla el valor de:

$$\sum_{i=n}^{m} i^2$$

   Compara el resultado con el resultado analítico.

4. Elabora un programa que pida el valor de dos enteros, n y m, y calcule la sumatoria de todos los números pares comprendidos entre n y m.

5. Elabora un programa que te permita estimar el límite de la serie:

$$\sum_{n=1}^{\infty} \frac{1}{2^n}$$

6. Elabora un programa que genere una sucesión de números aleatorios en el intervalo [0.01,0.20], cuya suma sea menor que 1.50.

7. (i) Diseña un programa en el que se le pida al usuario un ángulo en grados. Si el ángulo está fuera del rango $[0, 360]$ reduzca el ángulo en este intervalo.
   Hint: Si el ángulo es negativo, considérese que los ángulos negativos giran en el sentido de las manecillas del reloj. A un ángulo negativo θ', lo podemos transformar a un ángulo positivo θ sumándole 360, tal que $\theta' = 360 - \theta$. También considera el caso en el que el ángulo sea mayor a 360).
   (ii) Haz la función `angulo(x)` que haga el inciso anterior.

8. Haz un programa que pida el valor de dos enteros n y m y que muestre en pantalla el valor de:
$$\sum_{i=n}^{m}(i+1)^2$$

9. **Algoritmo de fracciones continuas**
   Para la solución de ecuaciones cuadráticas, en ocasiones se tienen fuertes singularidades y la fórmula ya no es funcional. Una manera de darle la vuelta a estas singularidades es reescribir la ecuación a evaluar en términos de una expansión Taylor. Otra posibilidad es utilizar las llamadas fracciones continuas, que pueden verse como generalizaciones de una expansión de Taylor. Un enfoque posible de las fracciones continuas es la sustitución sucesiva. Vamos a ilustrar esto con un ejemplo simple:
$$x^2 + 4x - 1 = 0, \tag{3.6}$$
   que puede reescribirse como:
$$x = \frac{1}{4+x},$$
   que, a su vez, podría representarse a través de un proceso iterativo de sustitución:
$$x_{n+1} = \frac{1}{4+x_n},$$
   con $x_0 = 0$. Esto significa que si tenemos,

$$\begin{aligned} x_1 &= \tfrac{1}{4} \\ x_2 &= \tfrac{1}{4+\frac{1}{4}} \\ x_3 &= \tfrac{1}{4+\frac{1}{\frac{1}{4}}} \\ &\vdots \end{aligned}$$

Esto puede ser reescrito de forma compacta como:

$$x_n = x_0 + \cfrac{a_1}{x_1 + \cfrac{a_2}{x_2 + \cfrac{a_3}{x3 + \cfrac{a_4}{x_4 + \cdots}}}}$$

(i) Escribe un programa que resuelva la ecuación (3.6) con la implementación del algoritmo de fracción continua y lo haga iterativamente. La solución exacta es $x = 0.23607$, ¿Cuántas iteraciones necesitamos para tener 3 cifras decimales correctas?
(ii) Generaliza el código para que el usuario introduzca los coeficientes de la función y el número de precisión que desea obtener.

10. Define una función que convierta grados Fahrenheit en centígrados y otra que convierta grados centígrados a Kelvin:

11. Haz un algoritmo recursivo de la Torre de Hanói. El problema sigue así: Dadas 3 pilas, una con un conjunto de N discos de tamaño creciente, determina el mínimo número u óptimo de pasos que lleva mover todos los discos desde su posición inicial a otra pila sin colocar un disco de mayor tamaño sobre uno de menor tamaño. La definición de la función es:

$$\text{hanoi}(n) = \begin{cases} \text{si } n = 1 \rightarrow 1, \\ \text{si } n > 1 \rightarrow 2 * \text{hanoi}(n-1) + 1 \end{cases}$$

Con la relación de recurrencia:

$$h_n = 2h_{n-1} + 1h_1 = 1$$

Compara recursión y recursión+memoisación.

Capítulo 4

Arreglos

En muchas situaciones necesitamos procesar una colección de valores ordenados y relacionados entre sí. Por ejemplo, una tabla de tiempo contra temperatura. Si usamos lo visto en los capítulos anteriores, podríamos manipular esta tabla a partir de dos listas; lista de tiempos y lista de temperaturas. Sin embargo, esto puede llegar a ser muy tediosos, extremadamente complicado y llevar a un manejo de datos erróneo. Para corregir este problema, los lenguajes de programación incluyen características de estructura de datos, es decir, el uso de arreglos.

Un arreglo es un conjunto finito y ordenado de elementos homogéneos. La propiedad "ordenada"significa que existe un primero, segundo, tercero, $\cdots$, n-enésimo elementos del arreglo para ser identificado. En términos computacionales, un arreglo es una variable secuencial alojada en la memoria, para la cual pueden ser seleccionados individualmente elementos mediante el uso de subíndices.

Los arreglos, al igual que las matrices o tensores, existen con distintos orden, es decir, para un arreglo de orden cero se tiene un número o escalar. Un arreglo de orden uno es un vector o una lista. Un arreglo de orden dos es equivalente a una matriz. Mientras que un arreglo de orden tres tendrá la forma de un paralelepípedo. En las siguientes secciones veremos con detalle estos arreglos, descartando el arreglo de orden cero, que es el caso trivial y ya fue abordado cuando vimos la asignación de variables (capítulo dos).

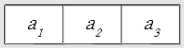
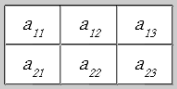
Orden	Equivalente	Esquema
0	Escalar	
1	Vector o Lista	
2	Matriz	
3	Arreglo o Matriz Paralelepípedo	

Figura 4.1: El análogo de los arreglos en dimensión 0, es un escalar; en dimensión 1, es una lista o vector; en dimensión 2, es una matriz rectangular, y en dimensión 3, es un arreglo ordenado que asemeja a un paralelepípedo.

4.1. Arreglos de orden uno

Si has trabajado con vectores, el arreglo de orden uno te parecerá muy natural. Los arreglos de orden uno, de hecho, son listas o vectores. En el capítulo 2 vimos como asignar y manipular básicamente a los vectores. Pero lo que no hemos visto con detalle es como operarlos como vectores.

Otra forma de asignar vectores, además de la vista anteriormente, es usar la librería `numpy` que significa "*numeric python*". En esta librería podemos asignar arreglos unidimensionales de forma sencilla y operarlos con los operadores binarios que conocemos. Ejemplo:

```
>>> import numpy as np
>>> x = np.array([1,5,2])
>>> y = np.array([7,4,1])
>>> print(x + y)
[8 9 3]
>>> print(x * y)
[ 7 20  2]
>>> print(x - y)
```

```
[-6  1  1]
>>> print(x / y)
[0.14285714     1.25     2.]
>>> print(x % y)
[1 1 0]
```

Como se puede observar, las operaciones binarias actúan elemento por elemento de los vectores, respetando la dimensión del vector. La suma y resta de vectores es correcta, se hace entrada por entrada. Podría pensarse que las otras operaciones son correctas, esto será cierto si sólo pensamos que los operadores actúen entrada por entrada, pero si queremos obtener un producto punto o producto cruz, tendremos que hacer algo completamente diferente.

Recordemos que el producto punto de dos vectores con dimensión n-enésima, se define como:

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \cdots + a_3 b_3 = \sum_{i=1}^{n} a_i b_i$$

Veamos como queda programado en Python

```
def punto(a,b):
        n,m=len(a),len(b)
        if n!=m:
            return 'Vectores con dimensiones distintas'
        else:
            sum=0
            for i in range(n):
                sum+=a[i]*b[i]
            return sum
```

Para usar la función, tecleamos:

```
>>> x = np.array([1,5,2])
>>> y = np.array([7,4,1])
>>> print(punto(x,y))
29
```

Otro ejemplo esencial es el producto vectorial, con la definición

$$\vec{c} = \vec{a} \times \vec{b} = \begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \end{bmatrix} = (a_1 b_2 - a_2 b_1)\hat{i} + (a_2 b_0 - a_0 b_2)\hat{j} + (a_0 b_1 - a_1 b_0)\hat{k}.$$

Debemos recordar un poco la notación de vectores, el vector c puede ser expresado como: $\vec{c} = c_0\hat{i} + c_1\hat{j} + c_2\hat{k}$ o $\vec{c} = (c_0, c_1, c_2)$. Con esto, el código queda:

```
def cruz(a,b):
    n,m=len(a),len(b)
    if n!=m or n!=3:
        return 'Con estos vectores no funciona'
    else:
        c0=a[1]*b[2]-a[2]*b[1]
        c1=a[2]*b[0]-a[0]*b[2]
        c2=a[0]*b[1]-a[1]*b[0]
        c=[c0,c1,c2]
        return c
>>> x = np.array([2,0,1])
>>> y = np.array([1,-1,3])
>>> print(cruz(x,y))
[1, -5, -2]
```

4.1.1. Comprensión de listas

Algo muy útil en Python es lo usualmente llamado "comprensión de listas". En un sólo paso, haremos la creación de una lista usando una estructura `for` en la misma línea. La sintaxis es:

```
lista=[x for x in seq]
```

Veamos un ejemplo de la creación de una lista con el método de comprensión y el método tradicional:

```
>>> lista_a=[]
>>> for i in range(1,10):
    lista_a.append(i)
>>> print(lista_a)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> #Metodo de comprension de Listas
>>> lista_b=[x for x in range(1,10)]
>>> #Ahorramos espacio
>>> print(lista_b)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La comprensión de listas no sólo funciona con la estructura de control `for`, sino que podemos usar todas las estructuras de control que queramos:

```
>>> lista_b  = [i * i for i in range(10)  if i % 2 == 0]
>>> print (lista_b)
[0, 4, 16, 36, 64]
```

Veamos otro ejemplo, en el cual proporcionamos una frase, guardamos las vocales de la frase en una lista hecha en una línea y luego imprimimos el resultado.

```
>>> frase = 'El pasado está escrito en la memoria \
    y el futuro presente en el deseo: Carlos Fuentes'
>>> vocales=[i for i in frase if i in 'aeiouAEIOUáéíóú']
>>> print(vocales)
```

Veamos que la estructura condicional *if* selecciona elementos de la cadena de caracteres *aeiouAEIOUáéíóú*. Pero esta cadena queda muy incómoda de programar, por lo que podemos usar la función `lower()`, que devuelve en minúscula la cadena dada. Si no existen caracteres en mayúscula, devuelve la cadena original. La función inversa será `upper()`. También usaremos la función `isalpha()` para verificar que la cadena contiene letras y no números, sin importar si están en mayúsculas o minúsculas. Con estas funciones, haremos nuestra propia función que haga la selección de consonantes en la frase dada:

```
>>> def consonante(letras):
        vocales  = 'aeiouáéíóú'
        a=letras.isalpha()
        b=letras.lower()
        return a and b not in vocales
>>> consonantes = [i for i in frase if consonante(i)]
>>> print(consonantes)
```

En paralelo, la comprensión de listas también puede usarse en diccionarios, veamos un pequeño ejemplo:

```
>>> cubos = {i: i**3 for i in range(5)}
>>> print(cubos)
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
```

Las comprensiones de listas son útiles para escribir códigos elegantes que sea fácil de leer y depurar, pero en ocasiones hace que los códigos se ejecuten más lentamente o usen más memoria. Si nuestro código tiene menos rendimiento, entonces necesitamos usar otra alternativa.

4.2. Arreglos de orden dos

Se definen matrices como arreglos de orden 2, cuyos elementos se asignan con dos índices. El arreglo de orden 2 se puede considerar como un vector de vectores, de manera más intuitiva, como se ve a continuación, una matriz:

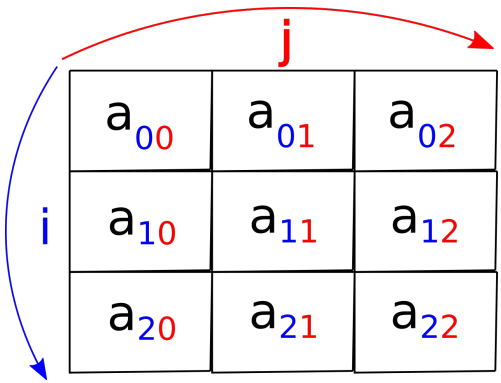


Figura 4.2: Notación de índices de un arreglo de orden 2.

Es, por consiguiente, un conjunto de elementos del mismo tipo, donde el orden de los componentes es significativo y en el que se necesita especificar dos índices para poder identificar cada elemento del arreglo.

En Python, como primera aproximación, usaremos el arreglo de orden dos como una lista de listas, donde las listas interiores son los renglones. Hagamos un pequeño código que le pida las dimensiones de la lista de listas al usuario, es decir, el número de renglones y columnas; y luego, que el usuario introduzca los elementos:

```
m=int(input('Cuantos renglones tiene tu arreglo'))
n=int(input('Cuantas columnas tiene tu arreglo'))
A=[] #A es una lista vacía
#creamos una matriz de m,n dimensiones llena de ceros
for i in range(m):
    A.append([0]*n)
print(A)
#Pedimos elemento por elemento
for i in range(m): #vamos con los renglones
    for j in range(n): #ahora las columnas
        A[i][j]=int(input('Da A(%d,%d)'%(i,j)))
print(A)
```

De este pequeño código podemos ver que pedimos las dimensiones de la matriz, asignamos una variable vacía tipo lista, luego llenamos la variable `A` de ceros y, por último, con dos ciclos anidados, le pedimos al usuario que nos de elemento por elemento:

```
>>> Ll=[[1,2,3],[4,5,6]]
>>> print(Ll)
[[1, 2, 3], [4, 5, 6]]
>>> print('primer elemento es', Ll[0][0])
primer elemento es 1
>>> print('ultimo renglon ultima columna', Ll[-1][-1])
ultimo renglon y ultima columna es 6
>>> print('primer renglon es', Ll[0])
primer renglon es [1, 2, 3]
```

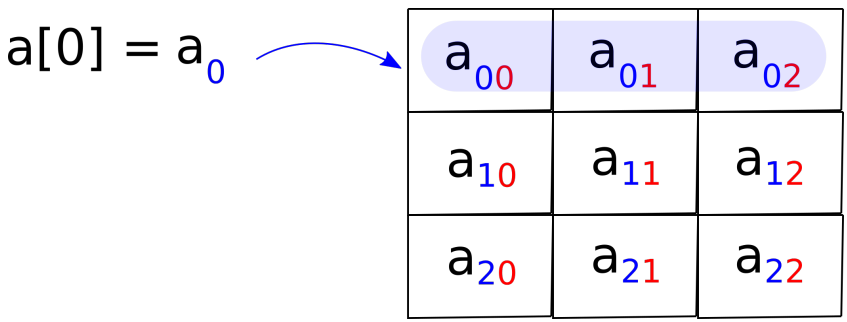


Figura 4.3: Obtención de una fila en un arreglo de orden 2.

Como se ve en el código, si usamos un único índice, lo que nos regresará será un renglón. Para encontrar columnas, usaremos la comprensión de listas, donde recorremos los renglones y sólo guardamos el primer elemento de cada renglón en la lista `col`:

```
>>> col = [renglon[0] for renglon in Ll]
>>> print(col)
[1, 4]
```

El inconveniente principal que tiene la comprensión de listas para obtener las columnas de una matriz es que hace que el código se vuelva engorroso y probablemente lento. Es aquí donde nos conviene mucho usar la función `array`, que convierte las listas de listas en arreglos, ya los identifica como tablas, y así es más práctico manipularlos:

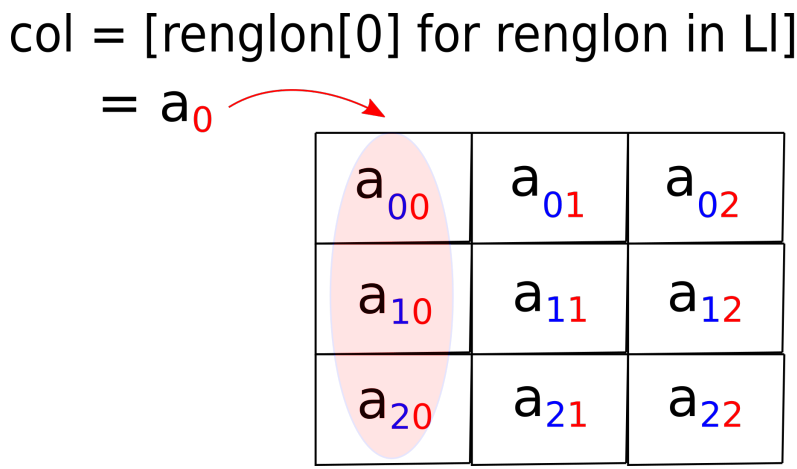


Figura 4.4: Obtención de una columna en un arreglo de orden 2.

```
>>> from numpy import array
>>> Ll=[[1,2,3],[4,5,6]]
>>> print(Ll)
[[1, 2, 3], [4, 5, 6]]
>>> A=array(Ll)
>>> print(A)
[[1 2 3]
 [4 5 6]]
>>> print('primer elemento es', A[0,0])
primer elemento es 1
>>> print('segundo renglon es',A[1])
segundo renglon es [4 5 6]
>>> print('primera columna es',A[:,0])
primera columna es [1 4]
```

Del código anterior, es importante notar que la función `array` recibe una lista de listas y regresa esta lista de listas en forma de una tabla. La segunda diferencia es que ya no necesitamos abrir y cerrar paréntesis cuadrados para cada elemento, será suficiente poner los dos índices que necesitamos para referenciar al elemento dentro de un sólo paréntesis cuadrado. La última diferencia será en el uso del comodín *hasta* `:`, podemos obtener las columnas y esto nos simplifica mucho la programación.

Ya que hemos hecho la distinción entre lista de listas y un `array`, veamos como hacer variables asignadas en memoria de tipo arreglo, llenas de ceros o unos. Esto lo haremos usando el módulo `numpy`:

```
>>> from numpy import zeros,ones,eye
>>> A0=zeros(4)
>>> print(A0)
[0. 0. 0. 0.]
>>> A0=zeros([4,4],float)
>>> print(A0)
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
>>> A1=ones([4,3],int)
>>> print(A1)
[[1 1 1]
 [1 1 1]
 [1 1 1]
 [1 1 1]]
>>> Id=eye(3)
print(Id)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Estas funciones nos permiten ahorrar tiempo y espacio en el código, haciéndolos más claros. Regresando a la obtención de elementos seleccionados del arreglo, pensemos en el caso en que tenemos un arreglo y queremos encontrar un sub-arreglo. Esto lo haremos con el procedimiento de recorrido ya conocido para listas en donde especificamos el inicio y el final. Podemos omitir algún valor usando comodines:

```
>>> a2 = array([[10, 11, 12, 13, 14],
           [15, 16, 17, 18, 19],
           [20, 21, 22, 23, 24],
           [25, 26, 27, 28, 29]])
>>> print(a2[1:,2:4])
[[17 18]
 [22 23]
 [27 28]]
```

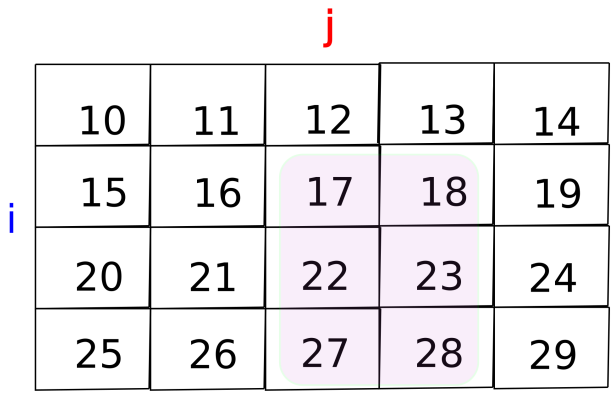


Figura 4.5: Selección de una submatriz usando el método de desplazamiento o *slicing*.

4.2.1. Cuadrados mágicos

Un ejemplo básico de los arreglos de orden 2 son los cuadrados mágicos. Un cuadrado mágico consiste en una distribución de números en filas y columnas, formando un cuadrado, de forma que los números de cada columna y diagonal suman lo mismo N. A esta N la nombraremos como la propiedad del cuadrado mágico. Se pueden formar cuadrados mágicos con cualquier tipo de número, ya sea natural, entero, fraccionario, números complejos, potencia de un número, etcétera.

Los cuadrados mágicos tienen su origen en China, donde eran conocidos varios siglos antes de Cristo. Según cuenta la leyenda, el primer cuadrado mágico fue revelado al emperador Yu a través de una tortuga divina que apareció en el río Luo, que lo llevaba grabado en su caparazón. Este emperador reinó en el siglo XII A.C. De China, pasaron a Japón, luego a la India y, de allí, a Arabia. Al mundo occidental llegó su conocimiento mucho más tarde, a través de los árabes [6]. El primer cuadrado mágico del que se tiene documentación en Europa aparece en el grabado Melancolía de Alberto Durero, que puede observarse a continuación [7].

Los cuadrados mágicos se clasifican de acuerdo con el número de celdas que tiene cada fila o columna: un cuadrado con tres celdas se dice que es de tercer orden, de cinco celdas será de quinto orden. Los cuadrados mágicos presentan propiedades únicas, veamos algunas de las propiedades:

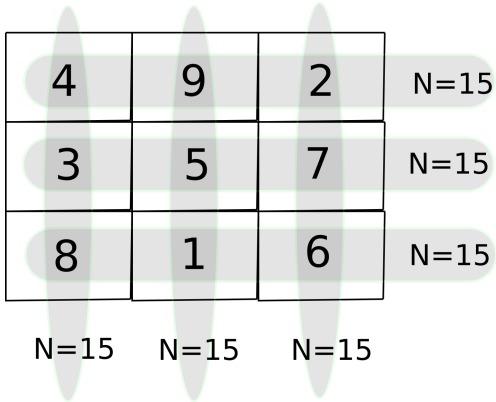


Figura 4.6: Cuadrado mágico de 3×3 cuya suma de renglones, columnas y diagonales da 15

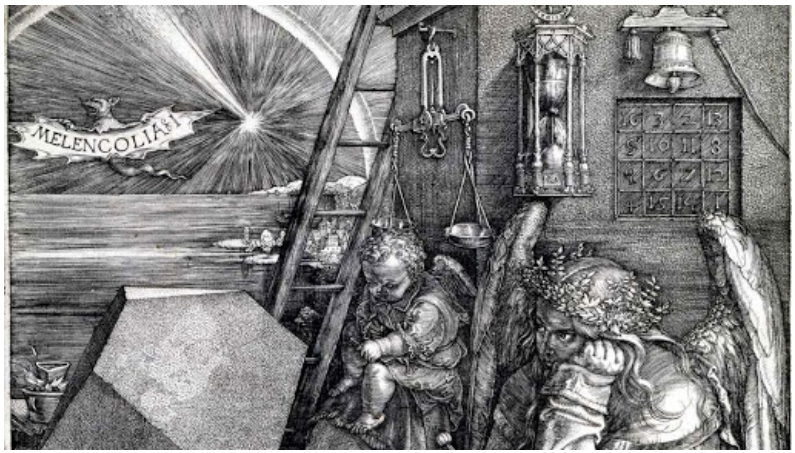


Figura 4.7: Grabado Melancolía de Alberto Durero. El grabado se encuentra expuesto en la Galería Nacional de Arte de Karlsruhe, Alemania.

- Se puede sumar o restar un escalar a un cuadrado mágico, obteniendo otro cuadrado mágico como resultado. Teniendo que la nueva N', será $N' = N \pm c$, donde c es el escalar que se suma o se resta. en el ejemplo

siguiente $c = 2$:

$$\begin{bmatrix} 7 & 2 & 3 \\ 0 & 4 & 8 \\ 5 & 6 & 1 \end{bmatrix} + 2 = \begin{bmatrix} 9 & 4 & 5 \\ 2 & 6 & 10 \\ 7 & 8 & 3 \end{bmatrix}$$

- Se puede multiplicar ó dividir un cuadrado mágico por un escalar. Para el caso de la multiplicación, la nueva N 'es N '$= cN$. En la división se tendrá N '$= \frac{N}{c}$:

$$\begin{bmatrix} 7 & 2 & 3 \\ 0 & 4 & 8 \\ 5 & 6 & 1 \end{bmatrix} * 2 = \begin{bmatrix} 14 & 4 & 6 \\ 0 & 8 & 16 \\ 10 & 12 & 2 \end{bmatrix}$$

- Es posible realizar sumas y restas entre cuadrados mágicos obteniendo otro cuadrado mágico:

$$\begin{bmatrix} 7 & 2 & 3 \\ 0 & 4 & 8 \\ 5 & 6 & 1 \end{bmatrix} + \begin{bmatrix} 15 & 0 & 21 \\ 18 & 12 & 6 \\ 3 & 24 & 9 \end{bmatrix} = \begin{bmatrix} 22 & 2 & 24 \\ 18 & 16 & 14 \\ 8 & 30 & 10 \end{bmatrix}$$

Además de estas propiedades, los cuadrados mágicos de 4×4 tienen propiedades adicionales:

- Se puede intercambiar entre sí dos filas junto con dos columnas simétricas en bloque. Es decir, todos los números de una fila con todos los números de otra fila, haciendo lo mismo con los números de las filas y columnas que sean simétricas a ellas respecto de los ejes vertical, horizontal y de las diagonales. Intercambiamos la primera fila con la cuarta fila, al igual que la primera y cuarta columna:

$$\begin{bmatrix} 8 & 4 & 3 & 15 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 6 & 10 & 13 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 6 & 10 & 13 & 1 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 8 & 4 & 3 & 15 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 10 & 13 & 6 \\ 14 & 9 & 2 & 5 \\ 0 & 7 & 12 & 11 \\ 15 & 4 & 3 & 8 \end{bmatrix}$$

- Una propiedad muy interesante que hay en los cuadrados mágicos de 4x4 es aquella en donde tenemos los números de casillas situadas en vértices

de rectángulos concéntricos paralelos al cuadrado y que conservan la propiedad de sumar N:

$$\begin{bmatrix} 8 & 4 & 3 & 15 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 6 & 10 & 13 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 8 & - & - & 15 \\ - & - & - & - \\ - & - & - & - \\ 6 & - & - & 1 \end{bmatrix} \rightarrow \begin{bmatrix} - & - & - & - \\ - & 9 & 2 & - \\ - & 7 & 12 & - \\ - & - & - & - \end{bmatrix}$$

Existen métodos generales para la construcción de cuadrados mágicos de orden pares e impares [8]. A continuación, abordaremos un algoritmo general para cuadrados mágicos de 3x3, que puede ser extrapolado a cualquier cuadrado mágico de orden impar.

Sean a,b,c,d,e y f números enteros cualesquiera, tal que:

a	b	c
	K	
d	e	f

$$\begin{aligned} a+b+c &= N \\ a+K+f &= N \\ b+K+e &= N \\ c+K+d &= N \\ d+e+f &= N \end{aligned}$$

N es la propiedad del cuadrado mágico. Sumando miembro a miembro de las tres igualdades centrales:

$$\begin{aligned} (a+b+c)+3K+(d+e+f) &= 3N \\ N+3K+N &= 3N \\ 3K &= N \\ K &= \frac{N}{3} \end{aligned}$$

De manera análoga, se puede demostrar que para un cuadrado de 5×5 se tiene $K=\frac{N}{5}$, para un cuadrado de 7×7 se cumple con $K=\frac{N}{7}$ y así consecutivamente para los cuadrados mágicos impares. Usando esta propiedad, propondremos un algoritmo propio para construir cuadrados mágicos de orden tres:

a+b	a-(b+c)	a+c
a-(b-c)	a	a+(b-c)
a-c	a+(b+c)	a-b

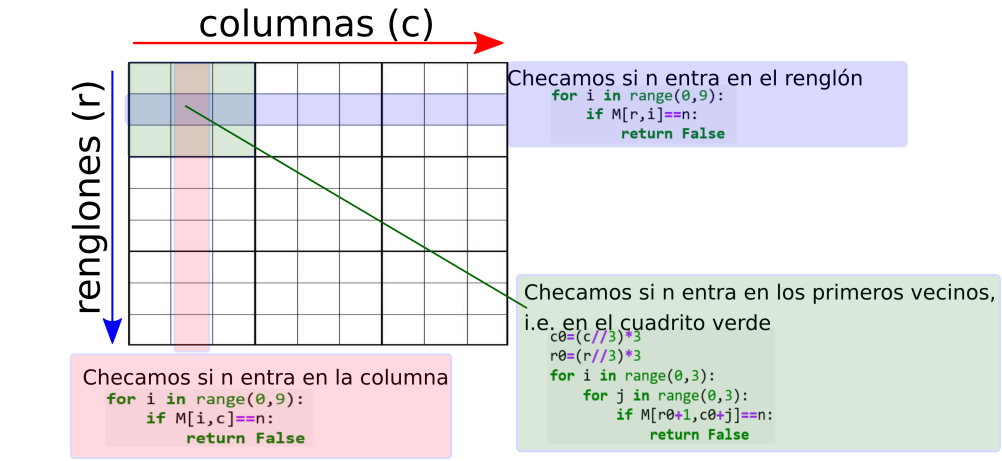
Con el cual el usuario pondría dos casillas y la propiedad, y el programa regresará el cuadrado mágico correspondiente. El código quedará así:

```
from numpy import *
N=float(input('Da la propiedad \t'))
A=zeros([3,3])
A[0,0]=float(input('Da el elemento a[0,0] \t'))
A[2,0]=float(input('Da el elemento a[2,0] \t'))
a=N/3.0
b=A[0,0]-a
c=a-A[2,0]
#Rellenando el cuadrado mágico
A[0,1]=a-(b+c)
A[0,2]=a+c
A[1,0]=a-(b-c)
A[1,1]=a
A[1,2]=a+(b-c)
A[2,1]=a+(b+c)
A[2,2]=a-b
print(A)
```

Un subgrupo de los cuadrados mágicos son los cuadros latinos. Los cuadrados latinos ya no tienen una propiedad N a la cual habrá que aplicarle una operación binaria para obtenerla, sino que sólo tendremos que acomodar números consecutivos en el cuadrado. El mejor ejemplo de un cuadrado latino, y el más famoso, son los sudokus. Trataremos de hacer una función que resuelva sudokus[1].

Usaremos r para indicar el lugar del renglón, c para indicar el lugar de las columnas. Lo primero que haremos es una función que se llama posibilidades, la cual verá si el valor n puede estar en la casilla dada. Entonces, la función en Python queda como:

[1]Para más información sobre este algoritmo, recomendamos ver el canal de Youtube Computerphile.




```
from numpy import *
def posibilidades(r,c,n):
    for i in range(0,9):
        if M[r,i]==n:
            return False
    for i in range(0,9):
        if M[i,c]==n:
            return False
    c0=(c//3)*3
    r0=(r//3)*3
    for i in range(0,3):
        for j in range(0,3):
            if M[r0+1,c0+j]==n:
                return False
    return True
```

Con esta función probabilidad, vemos si n no se encuentra en el renglón, columna y cuadradito interior. Si, en efecto, no se encuentra, entonces regresamos un `True` (verdadero), indicando que esa posibilidad si existe. Usaremos la notación de arreglos, propia del modulo `numpy`. Para los casos en que no exista un valor en la celda, usaremos un 0 (en lugar de vacío).

Veamos un ejemplo:

```
M=array([[5,3,0,0,7,0,0,0,0],[6,0,0,1,9,5,0,0,0],
    [0,9,8,0,0,0,0,6,0],[8,0,0,0,6,0,0,0,3],
    [4,0,0,8,0,3,0,0,1],[7,0,0,0,2,0,0,0,6],
    [0,6,0,0,0,0,2,8,0],[0,0,0,4,1,9,0,0,5],
    [3,0,0,2,8,0,0,7,9]])
#Veamos que valores pueden ir en el renglón 0, columna2
for i in range(1,10):
    print('Puede ir %d? = %s'%(i,posibilidades(0,2,i)))
```

Que regresa las siguientes evaluaciones:

```
Puede ir 1? = True
Puede ir 2? = True
Puede ir 3? = False
Puede ir 4? = True
Puede ir 5? = False
Puede ir 6? = False
Puede ir 7? = False
Puede ir 8? = False
Puede ir 9? = False
```

Entonces, en el renglón 0, columna 2 pueden ir un 1,2 y 4. Esto lo haremos para todas las casillas, i.e., recursivamente en una función llamada `sudoku()`. Antes de empezar con la función `sudoku()`, veamos como se incorporará la función posibilidades que hará la recursividad:

```
if M[r,c] == 0:
    for n in range(1,10):
        if posibilidades(r,c,n)==True:
            M[r,c]=n
            sudoku()
            M[r,c]=0
    return
```

Checamos todas las posibilidades que tenemos.
La recursión nos permite que esto se haga una vez tras otra.
Si nuestras opciones no son buenas, hay conflicto en algún paso de la recursividad, mejor ponemos cero. Así esperamos una mejor selección.

Lo que hacemos aquí es checar todas las posibilidades. Si ninguna funciona, continuamos, si no ponemos un `return` en esta estructura, seguirá en un ciclo infinito. Al final, imprimimos el arreglo. Notase que imprime todas las posibilidades de Sudokus:

```
from numpy import *
def posibilidades(r,c,n):
    for i in range(0,9):
        if M[r,i]==n:
            return False
    for i in range(0,9):
        if M[i,c]==n:
            return False
    c0=(c//3)*3
    r0=(r//3)*3
    for i in range(0,3):
        for j in range(0,3):
            if M[r0+1,c0+j]==n:
                return False
    return True

def sudoku():
    for r in range(9):
        for c in range(9):
            if M[r,c] == 0:
                for n in range(1,10):
                    if posibilidades(r,c,n)==True:
                        M[r,c]=n
                        sudoku()
                        M[r,c]=0
                return
    print(matrix(M))
```

Prueba con el sudoku definido anteriormente M, podrás ver que, para ese caso en particular, la función regresa nueves posibles soluciones. Luego, puedes probar la función con todos los sudokus que encuentres, verás que entre más valores determinados, menos lugares vacíos, tendrás menos posibles soluciones.

4.2.2. Matrices

Hasta el momento, hemos visto el manejo y las propiedades generales de arreglos. Hemos mencionado que estos arreglos de orden 2 podrían ser vistos como matrices. Sin embargo, no hemos profundizado en este tema. Ahora tocará dedicar nuestro tiempo para realizar operaciones matriciales, ya que es uno de los temas pilares en el cómputo científico, sin mencionar en la ciencia en general.

Ya hemos visto las propiedades importantes de los cuadrados mágicos. Como hemos observado, los cuadrados mágicos son matrices cuadradas con propiedades muy bien determinadas. Ahora, generalicemos para matrices rectangulares.

Sean dos matrices de $m \times n$, para $0 \leq i < m$ y $0 \leq j < n$. Al conjunto de todas las matrices de $m \times n$ le podemos aplicar las siguientes operaciones:

- La suma de dos matrices da como resultado una tercera matriz:

$$C_{ij} = A_{ij} + B_{ij}$$

- La multiplicación de un escalar por una matriz da como resultado otra matriz:

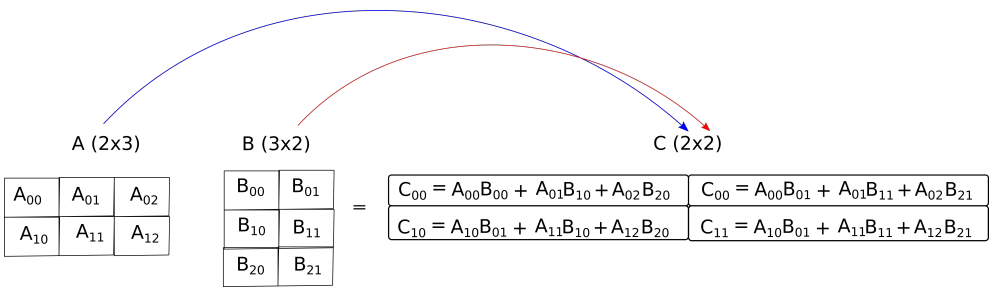
$$C_{ij} = cA_{ij}$$

- Sean A una matriz de $m \times n$, y B una matriz de $n \times p$. Definimos el producto de A por B que regresa una matriz C de $m \times p$ tal que:

$$C_{ij} = \sum_{k=0}^{n} A_{ik} B_{kj} \tag{4.1}$$

Las primeras dos operaciones se dejarán como ejercicios al lector. La multiplicación de matrices puede ser un ejercicio un poco complicado, por lo que será abordado aquí.

Veamos con un ejemplo sencillo. Multipliquemos una matriz A de 2×3 y una matriz B de 3×2.



Comúnmente se le conoce como "`la regla del karatazo`" y es por su forma nemotécnica en la cual tomamos una columna de la matriz B y la

multiplicamos completa por un renglón de la matriz A. Si nos fijamos en la ecuación 4.1, se ve que el primer renglón es $C_{00} = \sum_{k=0}^{n} A_{0k}B_{k0} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$ y así consecutivamente. La ecuación 4.1 será la expresión a programar:

```
from numpy import *
def multi(A,B):
    ren_A=len(A)
    col_A=len(A[0,:])
    ren_B=len(B)
    col_B=len(B[0,:])
    if col_A != ren_B:
        return 'No es posible operación'
    C=zeros([ren_A,col_B],float)
    for i in range(ren_A):
        for j in range(col_B):
            for k in range(ren_B):
                C[i,j]+=A[i,k]*B[k,j]
    return C

A=array([[1,2,3],[4,5,6]],int)
B=array([[9,8],[6,5],[3,2]],int)
print(multi(A,B))
```

Para llevar a cabo la multiplicación, necesitamos 3 ciclos anidados, en el ciclo interior queda el índice k que es en donde operamos la suma. Como k se encuentra en la suma, desaparece este índice del resultado final, comúnmente se dice que el índice se contrae.

Otro problema importante para el cómputo científico, es la solución de un sistema de ecuaciones lineales de la siguiente forma:

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \cdots + a_{0n}x_n &= c_0 \\ a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= c_1 \\ a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= c_2 \\ \vdots &= \vdots \\ a_{n0}x_0 + a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= c_n \end{aligned}$$

En notación matricial, este sistema de ecuaciones queda como:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n0} & a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

Podemos reducir este sistema de ecuaciones lineales a una sola ecuación matricial, que tiene contenida toda la información:

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{c} \tag{4.2}$$

donde $\boldsymbol{A}$ es la notación que usaremos para definir a la matriz A.

Existen muchos métodos para resolver esta ecuación matricial 4.2 [9, 10], el más popular es el llamado "*Eliminación Gaussiana*"

El método de Eliminación Gaussiana consiste en aplicar operaciones básicas sobre una matriz ampliada, tal que encontremos su matriz equivalente de la forma triangular superior o, como comúnmente se le llama, su matriz escalonada:

$$\left[\begin{array}{ccccc|c} a_{00} & a_{01} & a_{02} & \cdots & a_{0n} & c_0 \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n} & c_1 \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n} & c_2 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ a_{n0} & a_{n1} & a_{n2} & \cdots & a_{nn} & c_n \end{array}\right] \Rightarrow \left[\begin{array}{ccccc|c} u_{00} & u_{01} & u_{02} & \cdots & u_{0n} & d_0 \\ 0 & u_{11} & u_{12} & \cdots & u_{1n} & d_1 \\ 0 & 0 & u_{22} & \cdots & u_{2n} & d_2 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} & d_n \end{array}\right]$$

Para hacer esto, usaremos las operaciones básicas que llevan a $\boldsymbol{A}$ su matriz equivalente $\boldsymbol{U}$:

- Multiplicar un renglón o una columna de la matriz aumentada por una constante. Si multiplicamos cualquier renglón de la matriz $\boldsymbol{A}$ por cualquier constante y hacemos lo mismo para el vector $\boldsymbol{c}$, no estamos modificando la solución.
- Sumar o restar cualquier renglón o columna de la matriz aumentada. En realidad, lo que estamos haciendo en este paso es tomar la combinación lineal de dos ecuaciones del sistema de ecuaciones, por lo que la solución no cambiará.
- Intercambiar renglones de la matriz aumentada. Esto se hace, principalmente, para evitar la división entre cero. A esta operación básica, por sí sola, se le llama " *Método de Pivoteo*".

Con la combinación de estas operaciones, podemos plantear una forma general para la obtención de ceros en la matriz aumentada:

$$R_j = R_j - \left[\frac{a_{ji}}{a_{ii}}\right] R_i$$

donde R_j se refiere al renglón j-ésimo de la matriz aumentada. Esta expresión tiene contenida las dos primeras operaciones para convertir $\boldsymbol{A} \Rightarrow \boldsymbol{U}$.
En notación de código de Python, la expresión general es:

$$A_j = A_j - \left[\frac{mult}{div}\right] A_i,$$

donde $mult$ es $A[j,i]$ y div es $A[i,i]$. Veamos como queda el código:

```
from numpy import array, empty
A=array([[2,1,4,1],[3,4,-1,-1],[1,-4,1,5],[2,-2,1,3]],float)
v=array([-4,3,9,7],float)
n=len(v)
for i in range(n):
    div=A[i,i]
    for j in range(i+1,n):
        mult=A[j,i]
        A[j,:]-=(mult*A[i,:])/div
        v[j]-=(mult*v[i])/div
        print('--paso renglon=%d, columna=%d--'%(i,j))
        print(A)
        print(v)
x=empty(n,float)
for i in range(n-1,-1,-1):
    x[i]=v[i]
```

```
    for j in range(i+1,n):
        x[i]-=A[i,j]*x[j]
print(x)
```

Al ejecutar el código, podrás ver todos los pasos que hace el programa para transformar la matriz A a su matriz equivalente triangular superior U. Ahora, faltaría hacer la diagonal igual a uno, por lo que sólo queda hacer la división de los renglones entre los elementos de la diagonal:

$$R_i = \frac{R_i}{a_{ii}}$$

En notación del código, la expresión general es:

$$A_j = \left[\frac{1}{div}\right] A_j - \left[\frac{mult}{div}\right] A_i$$

De aquí ya tenemos una matriz triangular superior de la forma:

$$\begin{bmatrix} 1 & a_{01} & a_{02} & \cdots & a_{0n} \\ 0 & 1 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & 1 & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

Para terminar la solución del sistema de ecuaciones y que Python regrese el resultado completo, es necesario hacer una sustitución hacia atrás tal que:

$$\begin{aligned} x_n &= c_n \\ x_{n-1} &= c_{n-1} - a_{n-1,n}x_n \\ x_{n-2} &= c_{n-2} - a_{n-2,n-1}x_{n-1} - a_{n-1,n}x_n \\ &\cdots \\ x_j &= c_j - a_{j,j+1}x_{j+1} - a_{j,j+1}x_{j+2} \end{aligned}$$

Generalizando la fórmula, queda:

$$x_j = c_j - \sum_{k=j+1}^{n} a_{j,k}x_k$$

Veamos un ejemplo en concreto con el código completo:

$$\begin{aligned} 2w + x + 4y + z &= -4 \\ 3w + 4x - y - z &= 3 \\ w - 4x + y + 5z &= 9 \\ 2w - 2x + y + 3z &= 7 \end{aligned} \quad \Rightarrow \quad \left[\begin{array}{cccc|c} 2 & 1 & 4 & 1 & -4 \\ 3 & 4 & -1 & -1 & 3 \\ 1 & -4 & 1 & 5 & 9 \\ 2 & -2 & 1 & 3 & 7 \end{array}\right]$$

En Python, quedaría:

```
from numpy import array, zeros
A=array([[2,1,4,1],[3,4,-1,-1],[1,-4,1,5],[2,-2,1,3]],float)
c=array([-4,3,9,7],float)
n=len(c)
for i in range(n):
    for j in range(i+1,n):
        multi=(A[j,i]/A[i,i])
        A[j]=A[j]-multi*A[i]
        c[j]=c[j]-multi*c[i]
for i in range(n):
    div=A[i,i]
    A[i]=A[i]/div
    c[i]=c[i]/div
print('La Matriz aumentada queda como')
print(A,c)

x=zeros(n,float)
for i in range(n-1,-1,-1): #ciclo hacia atras
    x[i]=c[i]
    for k in range(i+1,n):
        x[i]-=A[i,k]*x[k]
print('El vector solución es')
print(x)
```

Este procedimiento es el método de Gauss. Para sacar matrices inversas, se hace el mismo procedimiento, con la diferencia de que no se opera el vector c, sino que se aumenta la matriz con la identidad. Veamos el procedimiento, paso por paso:

$$A = \begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}$$

$$\left[\begin{array}{ccc|ccc} 2 & 1 & -1 & 1 & 0 & 0 \\ -3 & -1 & 2 & 0 & 1 & 0 \\ -2 & 1 & 2 & 0 & 0 & 1 \end{array}\right].$$

Ahora se realizan las operaciones elementales sobre las filas de la matriz aumentada que sean necesarias para obtener la forma escalonada reducida de la matriz A;

sumando, tanto a la segunda como a la tercera fila la primera obtenemos:

$$\left[\begin{array}{ccc|ccc} 2 & 1 & -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 2 & 1 & 1 & 0 & 1 \end{array}\right].$$

Multiplicamos la segunda fila por -1 y la intercambiamos con la primera:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -1 & -1 & -1 & 0 \\ 2 & 1 & -1 & 1 & 0 & 0 \\ 0 & 2 & 1 & 1 & 0 & 1 \end{array}\right].$$

Ya tenemos el pivote de la primera fila que usamos para hacer ceros debajo:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -1 & -1 & -1 & 0 \\ 0 & 1 & 1 & 3 & 2 & 0 \\ 0 & 2 & 1 & 1 & 0 & 1 \end{array}\right].$$

Ahora usamos el pivote de la segunda fila:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -1 & -1 & -1 & 0 \\ 0 & 1 & 1 & 3 & 2 & 0 \\ 0 & 0 & -1 & -5 & -4 & 1 \end{array}\right]$$

Por último cambiamos de signo la tercera fila y usamos el pivote correspondiente:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 4 & 3 & -1 \\ 0 & 1 & 0 & -2 & -2 & 1 \\ 0 & 0 & 1 & 5 & 4 & -1 \end{array}\right]$$

El proceso ha finalizado porque en la parte izquierda tenemos la forma escalonada reducida de A y, puesto que ésta es la matriz identidad, entonces A tiene inversa y su inversa es la matriz que aparece a la derecha, en el lugar que al principio ocupaba la identidad. Cuando la forma escalonada reducida que aparece no es la identidad, es que la matriz de partida no tiene inversa. A este procedimiento se le llama "*Gauss-Jordan*", el código se deja de ejercicio al lector.

4.3. Ejercicios

1. Elabora una función que reciba una lista y regrese el promedio de los elementos.

2. Elabora un programa en el cual introduzcas los elementos de un vector por medio del teclado y te imprima en pantalla el elemento máximo.

3. Escribe una función que reciba tres vectores y regrese la combinación de producto punto y producto vectorial dada por la identidad vectorial:

$$\vec{A} \cdot \vec{B} \times \vec{C} = \vec{B} \cdot \vec{C} \times \vec{A} = \vec{C} \cdot \vec{A} \times \vec{B}$$

4. Escribe una función que reciba tres vectores y regrese el producto vectorial, usando la identidad vectorial:

$$\vec{A} \times (\vec{B} \times \vec{C}) = \vec{C} \times (\vec{B} \times \vec{A}) = \vec{B}(\vec{A} \cdot \vec{C}) - \vec{C}(\vec{A} \cdot \vec{B})$$

Cuadrados mágicos en Python

5. Construye un cuadrado mágico con los nueve primeros números pares de modo que las filas, columnas y diagonales sumen 30.
6. Construye un cuadrado mágico de 4x4 (suma=34). Los elementos de cada una de las nueve matrices 2x2 que componen el cuadrado también deben sumar 34.
7. Construye tu propio algoritmo de solución de cuadrados mágicos de 5x5.

Matrices

8. Haz una función que permita calcular la desviación estándar σ de una lista de números ($N \leq 15$). Sabiendo que:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - m)^2}{n-1}}$$

9. Escribe una función que reciba una matriz y regrese la matriz transpuesta.
10. Escribe una función que reciba una matriz compleja y regrese la matriz compleja conjugada.
11. Escribe una función que reciba dos matrices y regrese la suma matricial.
12. Escribe una función que reciba una matriz y un escalar y regrese la multiplicación del escalar por la matriz.
13. Escribe una función que reciba una matriz y regrese la matriz inversa usando el algoritmo de Gauss-Jordan.

Capítulo 5

Visualización

El científico computacional tiene la necesidad de mostrar visualmente los datos obtenidos por sus códigos, ya sea colocando sus datos en gráficas o animaciones. Para introducir al lector en esta tarea, empezaremos con visualización en 2-d de listas, los que presuponemos fueron obtenidos con algún otro procedimiento, ya sea en el laboratorio o con alguna simulación. Para esto usaremos el módulo `pylab`. En realidad, `pylab` no es una librería *per se*, sino que es un conglomerado de librerías, entre las que se encuentra `numpy`, `scipy`, `sympy`, `pandas`, `matplotlib` e `ipython`. En este capítulo usaremos algunas librerías de `pylab` para llevar a cabo la visualización. También veremos visualizaciones en 3-d. Para finalizar este capítulo haremos una breve introducción a la computación simbólica con la librería `sympy`.

5.1. Matplotlib

En esta sección usaremos la librería `matplotlib` para hacer visualización de datos en 2-d, ya que es una de las mejores herramientas que existen para gráficas bidimensionales. Para empezar, emplearemos la función `plot()`, que recibe dos listas y regresa la graficación de la primera como los datos en el eje de las abscisas y de la segunda como los datos del eje ordenado. Veamos un ejemplo:

```
from pylab import *

t=arange(0.0,2.0*pi,0.1)
plot(t,t)

show()
```

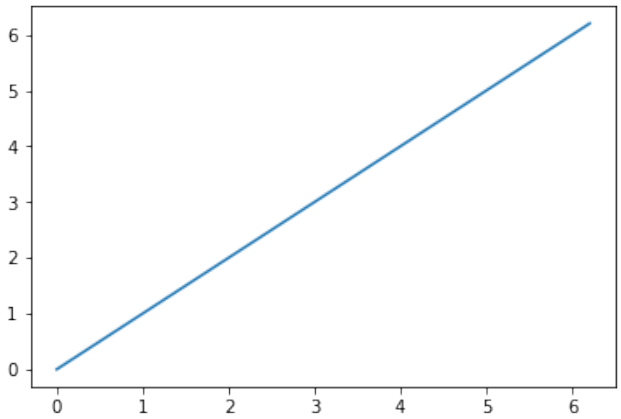


Figura 5.1: Representación de la recta identidad usando la función `plot` de `matplotlib`.

En este ejemplo hemos representado una línea que corresponde a la identidad. Recordemos que la función `arange()` es del módulo `numpy` y nos regresa una lista de valores que, en este caso, empiezan en cero y terminan en 2π con pasos de 0.1. Luego, usamos la función `plot()` para graficar la función identidad repitiendo la lista t en el eje x y en el eje y. Por último, usamos la función `show()` para mostrar en pantalla. Esta opción puede sonar trivial, pero consideremos que también existe la función `savefig()`[1] que permite guardar la imagen de salida.

Para controlar más el formato de la gráfica, podemos especificar el tipo de punto, línea y color del punto. No olvidemos que cualquier gráfico necesita expresar claramente que tipo de información maneja, por lo que es indispensable que etiquetemos apropiadamente los ejes y coloquemos el título de la gráfica. Usamos la función `xlabel()` para etiquetar el eje de las abscisas, `ylabel()` para el eje de las ordenadas, la función `title()` para el título de la gráfica. Por último, incluimos la función `grid()` para que nos muestre la gráfica cuadriculada.

```
from pylab import *
t=arange(0.0,2.0*pi+0.1,0.1)
plot(t,3*t,'g-+')
xlabel("abscisas")
ylabel("ordenadas")
title('Gráfica')
grid()
```

[1] La función `savefig()` recibe como argumento el nombre de la imagen a generar. Nos permite usar todos los formatos de imagen que queramos, *e.g.*, png, jpg, eps, pdf, tiff, etc.

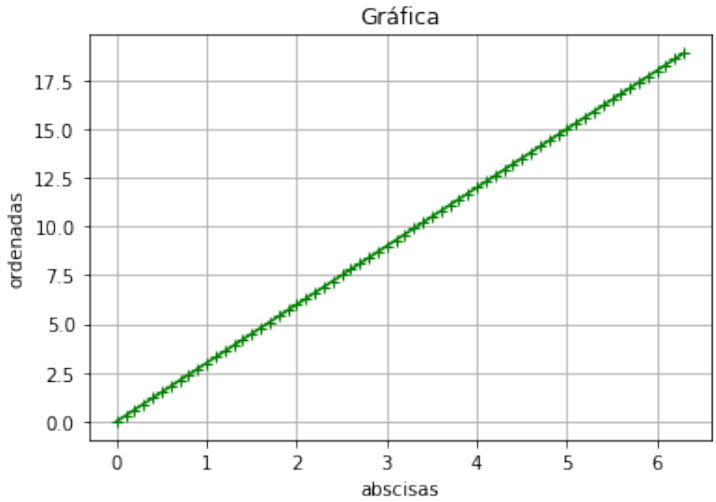


Figura 5.2: Representación de la recta $y = 3t$ con formatos básicos para la visualización.

En la siguiente tabla están los colores más populares con la letra característica para ser incorporado en la `plot()`, así mismo, incluimos el código para el estilo de línea y la marca de la gráfica.

Tabla 5.1: Marcadores, colores y tipos de líneas básicos que pueden ser utilizados en la visualización hecha con la librería `matplotlib`.

Colores	
b	blue
g	green
r	red
m	magenta
y	yellow
k	black
w	white

Línea	
-	Línea Sólida
–	Línea Discontinua
:	Línea punteada
-.	Línea punteada discontinua
None	Ninguna Línea

Marca	
+	Cruz
.	Punto
o	Círculo
*	Estrellas
p	Pentágonos
s	Cuadrados
x	Taches
D	Diamantes
h	Hexágonos
^	Triángulos

En el siguiente ejemplo, haremos una gráfica del área de una variedad de círculos y cuadrados, aquí combinamos dos gráficas con formato propio. En este ejemplo etiquetaremos cada curva, introduciendo una cuadro informativo dentro de la gráfica llamado leyenda, `legend()`, y agregando una etiqueta en la función `plot()` con la opción `label`.

```
from pylab import *
radio = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
area = [3.1416, 12.5663, 28.2743, 50.2654, 78.5397, 113.0972]
cuadrados = [1.0, 4.0, 9.0, 16.0, 25.0, 36.0]
plot(radio, area, marker="o",color="purple", label="Circulos")
plot(radio, cuadrados, linestyle="--",marker="s", color="r",
    label="Cuadrados")
xlabel("Radios/lados")
ylabel("Area")
title("Areas")
legend() # El cuadro con las etiquetas de las gráficas
show() # Para que aparezca en pantalla
```

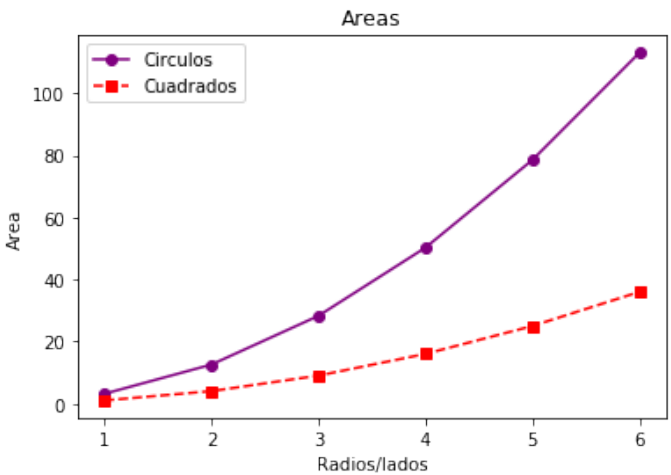


Figura 5.3: Visualización de dos curvas en una misma figura. Cada curva tiene su propio formato.

La función `legend()` puede incluir las etiquetas, así mismo, podemos modifica la locación en donde aparecerá la tabla de leyendas. Podemos localizar el cuadro de leyenda arriba, usando la opción `upper`, o abajo, con la opción `lower`. Igualmente, podemos localizarlo a la izquierda (`left`), derecha (`right`) o centrado (`center`).

```
t=arange(0.0,5.0,0.05)
s1=sin(2*pi*t) #la solución del oscilador ideal
s2=s1*exp(-t) #la solución del oscilador atenuado
plot(t,s1,t,s2)
legend(("ideal","atenuado"),loc="center")
```

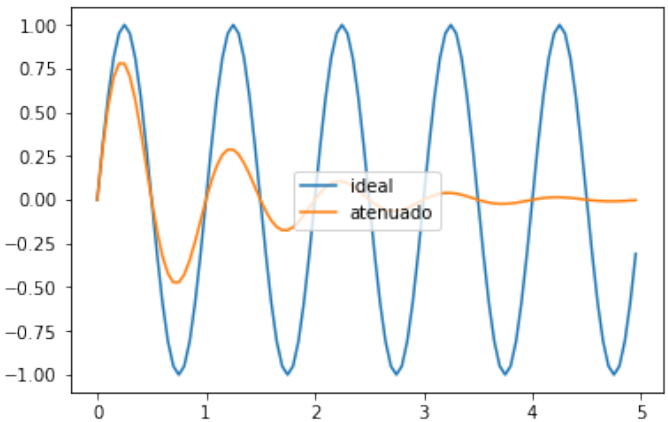


Figura 5.4: Ejemplo de dos curvas en una misma figura que representan la solución del oscilador armónico ideal y con amortiguamiento.

Si se requiere tener las dos gráficas separadas, por ejemplo, una abajo de la otra, lo más conveniente es usar la función `subplot()`. Esta función permite subdividir una ventana de figuras en varias celdas, tal que sea posible realizar una representación gráfica distinta en cada una de ellas. La sintaxis consiste en usar `subplot(m,n,k)`, donde m, n y k son enteros. La ventana de figura será subdividida en $m \times n$ celdas, k será el lugar donde pondremos la gráfica. Siempre $k > m \times n$, suponiendo que nos desplazamos por las columnas:

```
from pylab import *
def f(t):
    return cos(2*pi*t)*exp(-t)
t1=arange(0.0,5.0,0.1)
t2=arange(0.0,5.0,0.02)
#Grafica con 2 renglones y 1 columna
subplot(211) #La primera gráfica del subplot
plot(t1,f(t1),'bo',t2,f(t2),'k')
```

```
grid(True)
title('Dos subgraficas')
ylabel('Amortiguamiento')
subplot(212)
plot(t2,cos(2*pi*t2),'r>')
grid(True)
xlabel('tiempo (s)')
ylabel('sin amortiguamiento')
title('Sin amortiguamiento')
tight_layout() #función que optimiza el espacio entre gráficas
show()
```

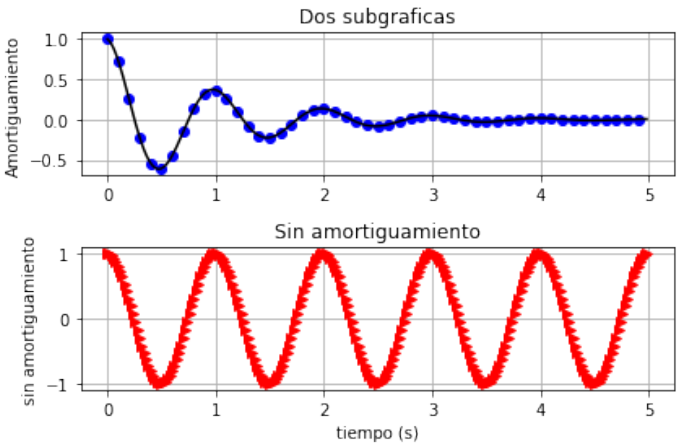


Figura 5.5: Ejemplo de dos curvas en dos figuras incorporadas con la función `subplot()`, que representan la solución del oscilador armónico ideal y con amortiguamiento.

La función `tight_layout()` es sumamente útil ya que optimiza de forma automática todas las gráficas que se tengan en un `suplot()`.

5.1.1. Parametrización de curvas

De manera intuitiva, la parametrización de una curva es cuando damos la ecuación de la curva en términos de un parámetro. Ejemplo, la recta que pasa por el punto P con dirección del vector $\vec{V}$ tiene la forma paramétrica de $R(t) = P + tV$ donde t es el parámetro a recorrer.

La ecuación de la recta que pasa por el punto $(1, 2)$ y tiene dirección del vector $(3, 4)$ es $R(t) + (1, 2) + t(3, 4) = (3t + 1, 4t + 2)$.

Una parametrización de una curva C es una función vectorial, tal que:

$$c : I \in \mathbb{R} \to \mathbb{R}^n,$$

al variar el parámetro t, su imagen $c(t)$ describe los puntos de la curva C.

La parametrización del circulo unitario es:

$$\begin{array}{rcl} \gamma : [0, 2\pi] & \longrightarrow & \mathbb{R}^2 \\ \theta & \longrightarrow & (\cos\theta, \sin\theta) \end{array}$$

```
from pylab import *
t=arange(0.0,2.0*pi+0.1,0.1)
plot(1*cos(t),1*sin(t),'.-k')#ecuaciones paramétricas
xlabel('abscisas')
ylabel('ordenadas')
title('Gráfica')
xlim(-5,5)
ylim(-3,3)
grid(True)
```

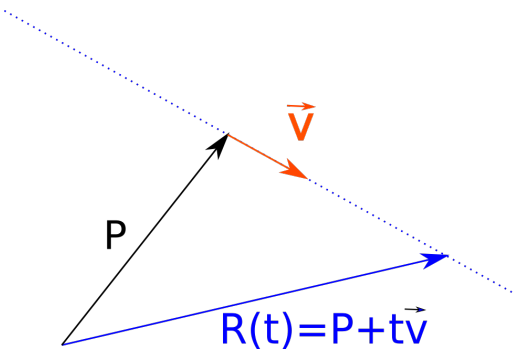


Figura 5.6: Esquema básico de la parametrización de una recta.

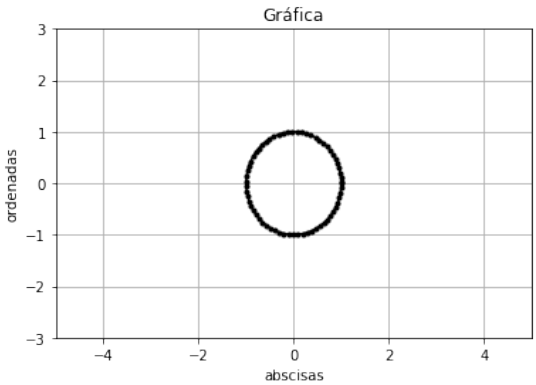


Figura 5.7: Parametrización del círculo unitario.

El siguiente ejemplo es la parametrización de una espiral desplazada a -3 en el eje y:

$$\begin{aligned} \gamma : [0, 2\pi] &\longrightarrow \mathbb{R}^2 \\ \theta &\longrightarrow (\theta \cos\theta, \theta \sin\theta - 3) \end{aligned}$$

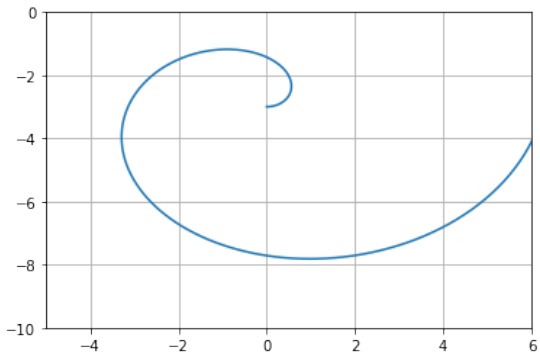


Figura 5.8: Espiral desplazada a -3 en el eje y.

```
from pylab import *  #Espiral
theta=arange(0,2*pi,0.01)
plot(theta*cos(theta),theta*sin(theta)-3)
grid()
xlim(-5, 6)
ylim(-10, 0)
show()
```

También existe la función `polar`, que recibe dos listas y las gráfica en coordenadas polares. Por ejemplo, la función $r = \sin(5z)$:

```
from pylab import *

z=arange(0.0,2*pi,0.01)
r=sin(5*z) #paramétrica en polares

polar(z,r) #plot ya no, ahora polar
```

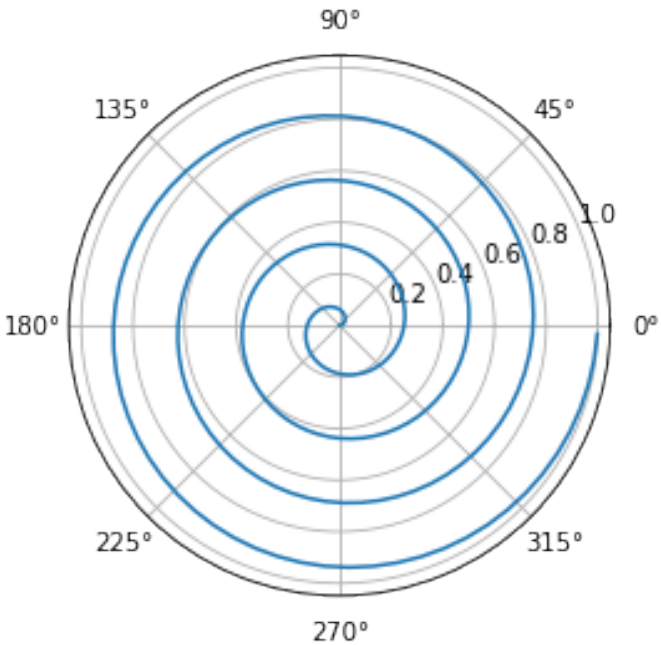


Figura 5.9: Visualización de la función $r = \sin(5z)$ en diagrama polar.

```
from pylab import *

theta=arange(0,8*pi,0.1)
radio=theta/(8*pi)

polar(theta,radio)
```

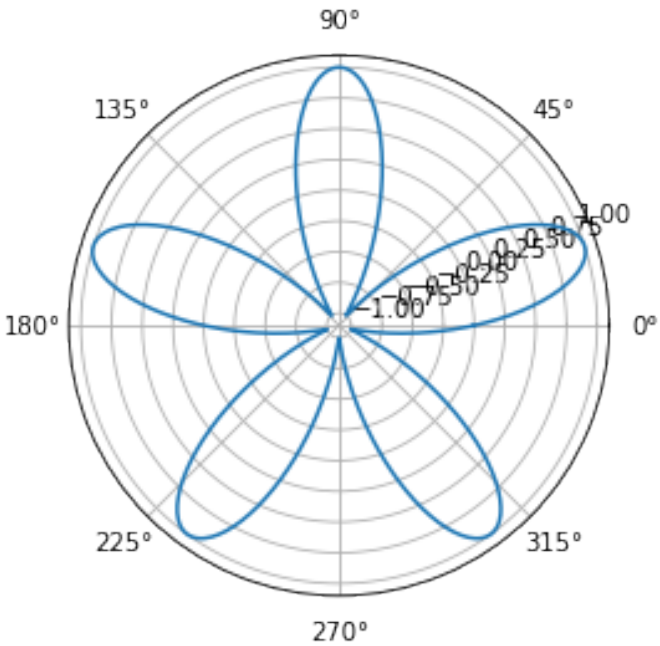


Figura 5.10: Visualización de la función $r = \frac{\theta}{8\pi}$ en diagrama polar.

Otro ejemplo ilustrativo será una hipocicloide con $R = 1$ y $k = \frac{R_1}{R}$ tal que:

$$\begin{aligned} \gamma : [0, 2\pi] &\longrightarrow \mathbb{R}^2 \\ \theta &\longrightarrow \left[(R_1 - R)\cos(t) + R\cos\left(\left(1 - \frac{R_1}{R}\right)t\right) \right. \\ &\qquad \left. (R_1 - R)\sin(t) + R\sin\left(\left(1 - \frac{R_1}{R}\right)t\right) \right] \end{aligned}$$

Cuando $\frac{R_1}{R} = k$ es un número racional, la curva se cierra. Si $\frac{R_1}{R} = k$ es irracional la curva da infinitas vueltas formando una curva trascendente. Véase el siguiente código, en donde se muestran varias combinaciones de curvas de la hipocicloide:

```
from pylab import *

def f(t):
    c=1-R1/R
    return ((R1-R)*cos(t))+(R*cos(c*t))
def g(t):
```

```
    c=1-R1/R
    return ((R1-R)*sin(t))+(R*sin(c*t))

t=arange(0,12*pi,0.1)
R=1

figure(figsize=(20,10))
subplot(241)
k=3
R1=k*R
plot(f(t),g(t),'b',label='k=3')
legend()
grid(True)

subplot(242)
k=4
R1=k*R
plot(f(t),g(t),'r',label='k=4')
legend()
grid(True)

subplot(243)
k=5
R1=k*R
plot(f(t),g(t),'g',label='k=5')
legend()
grid(True)

subplot(244)
k=6
R1=k*R
plot(f(t),g(t),'c',label='k=6')
legend()
grid(True)

subplot(245)
k=2.1
R1=k*R
plot(f(t),g(t),'b',label='k=2.1')
legend()
grid(True)

subplot(246)
k=3.8
R1=k*R
```

```
plot(f(t),g(t),'r',label='k=3.8')
legend()
grid(True)

subplot(247)
k=5.6
R1=k*R
plot(f(t),g(t),'g',label='k=5.6')
legend()
grid(True)

subplot(248)
k=7.2
R1=k*R
plot(f(t),g(t),'c',label='k=7.2')
legend()
grid(True)

show()
```

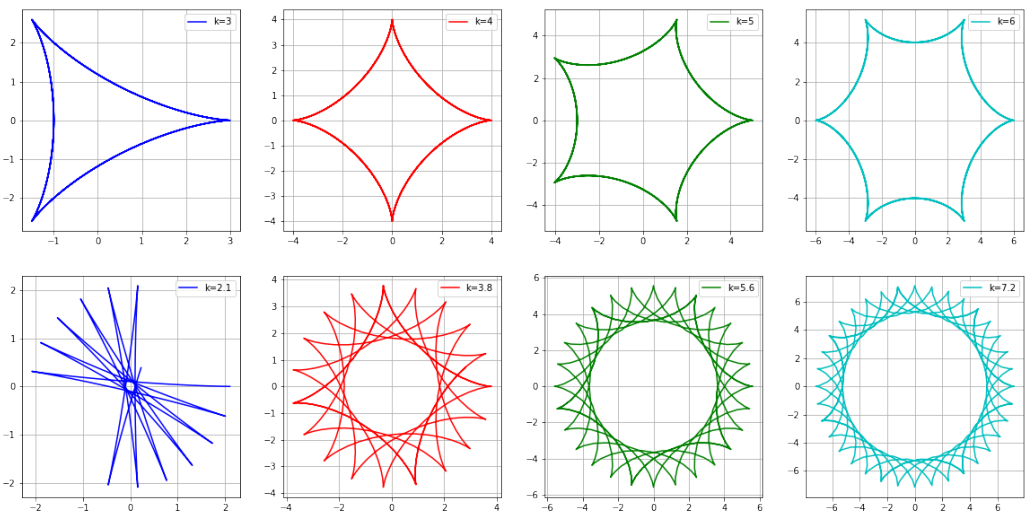


Figura 5.11: Hipocicloides con valores enteros y racionales de k

5.1.2. Visualización de vectores

Un ejemplo muy ilustrativo es la visualización de vectores. Veamos visualmente como es la suma de vectores usando `pylab`. Sean dos vectores en el plano $\vec{a} = (-1.7, 1.5)$ y $\vec{b} = (2.1, 1.9)$, la suma es:

$$\vec{c} = \vec{a} + \vec{b} = (a_0 + b_0, a_1 + b_1) = (-1.7 + 2.1, 1.5 + 1.9)$$

Veamos el código:

```
#suma de vectores
from pylab import *

a=array([-1.7,1.5])
b=array([2.1,1.9])

c=[]
for i in range(len(a)):
    x=a[i]+b[i]
    c.append(x)
c=array(c)
print(c)

quiver([-1.7], [1.5],  color=['r'], scale_units='x', scale=1)
quiver([2.1], [1.9],  color=['b'], angles='xy', scale_units='xy',
     scale=1)
quiver(c[0],c[1],color=['g'], angles='xy', scale_units='xy',
    scale=1)

xlim(-4, 4)
ylim(-4, 4)
grid()

show()
```

En el código se ha definido $\vec{a}$ y $\vec{b}$ como arreglos de orden uno para luego ser sumados y obtener $\vec{c}$. En la visualización se usa la función `quiver()` [11], que recibe las coordenadas iniciales y finales del vector como `quiver(x0, y0,` x_{final}, y_{final}`)`, en este ejemplo, usamos las coordenadas iniciales como el origen, por lo que no las indicamos, ya que son el default de la función. La opción `angles='xy'` indica que la flecha apunta de $(x_{inicial}, y_{inicial})$ a $(x_{inicial} + x_{final}, y_{inicial} + y_{final})$, y aplicamos esta opción a $\vec{b}$ y $\vec{c}$. Si esto no se especifica, la función `quiver()` toma el default como 45 grados en sentido anti-horario desde el eje x, que es lo que necesitamos para el vector $\vec{a}$. La opción `scale_units` indica en que unidades estará el vector en función de los ejes, por ejemplo, $\vec{b}$ y $\vec{c}$ tienen las unidades de los ejes, sin embargo, $\vec{a}$ lo reducimos

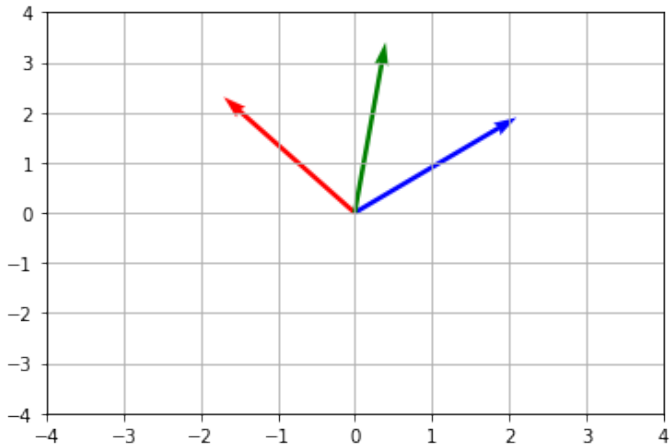


Figura 5.12: Visualización de la suma de vectores por la regla del paralelogramo. Se usa la función `quiver()` para dibujar los vectores.

a 0.5 de la escala del eje x indicando sólo la opción `scale_units='x'`. Por último, la opción `scale` da el tamaño de las flechas en términos de las unidades indicadas con `scale_units`, la opción `scale=1` proporciona la escala 1 a 1 entre vectores y ejes (la misma escala).

Si se quiere visualizar un conjunto grande de vectores, como sería un campo vectorial, en donde los vectores siguen a una función determinada en los ejes, se necesita usar la función `meshgrid()` de `numpy`. Un ejemplo sería un conjunto de vectores que siguen en x la función $\frac{x}{5}$ y en y la función $\frac{y}{5}$:

$$\vec{F} = \frac{x}{5}\hat{i} + \frac{y}{5}\hat{j} \tag{5.1}$$

```
from pylab import *
x = arange(-5,6,1)
y = arange(-5,6,1)

X, Y = meshgrid(x, y)
u, v = X/5, Y/5

quiver(X,Y,u,v)
show()
```

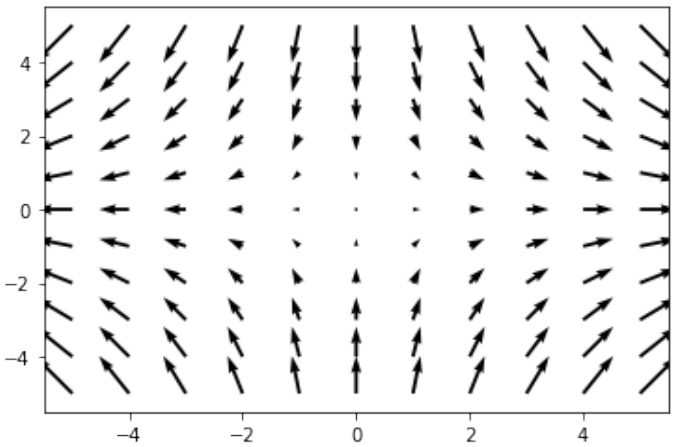


Figura 5.13: Visualización de un campo vectorial definido por la ecuación $\vec{F} = \frac{x}{5}\hat{i} + \frac{y}{5}\hat{j}$.

La función `meshgrid()` crea una cuadrícula para el plano con un mallado en (x, y); en este caso, el mallado es homogéneo en x y en y, mapeando el plano desde 0 a 2.2 con pasos de 0.2. En la cuadrícula se coloca un vector en cada cuadro creado por esta función. Definimos las variables u y v que guardan el comportamiento del campo vectorial dado por las ecuaciones 5.1. Posteriormente, se usa la función `quiver()` con los valores iniciales y los valores finales. La diferencia primordial entre el ejemplo anterior y éste, es que en el anterior se desplegó un vector y este ejemplo se despliega un conjunto de vectores usando las variables X, Y, u y v como arreglos y no flotantes.

Incluyamos más información de lo vectores del campo vectorial, si incluimos la pendiente de cada vector del campo tal que la pendiente se comporte con una función definida, se dice que tenemos un campo gradiente. La función que define la pendiente de los vectores, comúnmente, se llama función gradiente.

Sea una función gradiente de la forma:

$$z = xe^{-x^2-y^2}$$

```
from pylab import *
x = arange(-2,2.2,0.2)
y = arange(-2,2.2,0.2)
X, Y = meshgrid(x, y)
z = X*exp(-X**2 -Y**2)
dx, dy = gradient(z)
quiver(X,Y,dx,dy)
show()
```

Los valores dx y dy son las derivadas obtenidas con la función de `gradient()`, que es propia de `numpy`, y son calculadas con un método de diferenciación numérico. La función `quiver()` recibe la información de la visualización de los vectores obtenido con `meshgrid()` y la información de las pendientes de estos en los dos ejes.

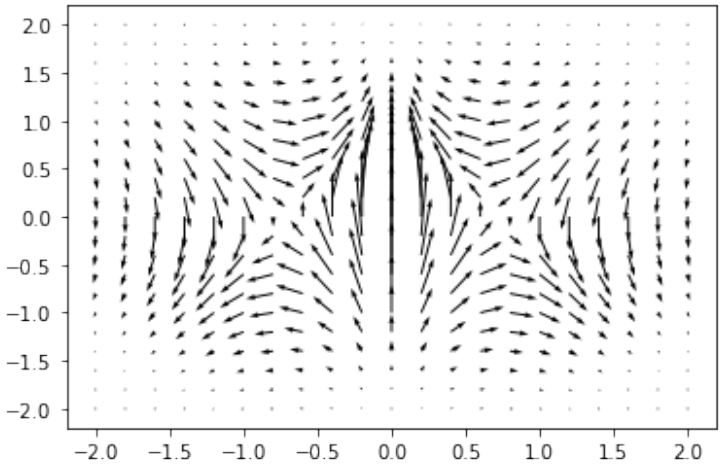


Figura 5.14: Visualización de un campo vectorial gradiente descrito por la ecuación $z = xe^{-x^2-y^2}$

Por último, se hará un ejemplo en el que se tendrá cuatro puntos que representan vórtices del campo vectorial, es decir, cuatro puntos centrales en los que los vectores parecen girar alrededor de éstos. La función que describe los cuatro vórtices de este ejemplo es:

$$\vec{F} = \sin x \cos y\hat{i} - \cos x \sin y\hat{j}$$

Además de poner estos cuatro vórtices, agregaremos colores a la visualización, tal que los colores adquieran un valor asignado con la función `gradient()` del ejemplo anterior:

```
from pylab import *
x = np.arange(-2,2.2,0.2) # Definimos los colores
y = np.arange(-2,2.2,0.2) # en funcion del gradiente
X, Y = np.meshgrid(x, y)
z = X*exp(-X**2 -Y**2)
dx, dy = gradient(z)
n = -1
color_arreglo = sqrt(((dx-n)/2)**2 + ((dy-n)/2)**2)
```

```
x = arange(0,2*pi+2*pi/20,2*pi/20) # Hacemos la visualización
y = arange(0,2*pi+2*pi/20,2*pi/20) # de los vectores
X,Y = meshgrid(x,y)
u = sin(X)*cos(Y); v = -cos(X)*sin(Y)
quiver(X,Y,u,v,color_arreglo)
show()
```

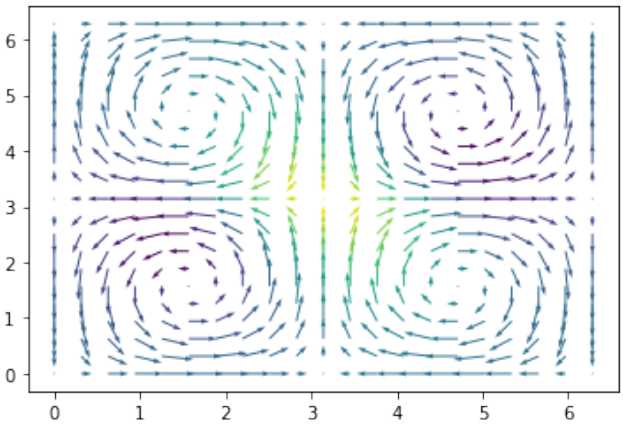


Figura 5.15: Visualización de un campo vectorial rotacional descrito por la ecuación $\vec{F} = \sin x \cos y\hat{i} - \cos x \sin y\hat{j}$.

5.1.3. Curvas de nivel

En ocasiones se tiene funciones tridimensionales que son difíciles de visualizar, por lo que conviene hacer proyecciones de estas superficies sobre planos. Estas proyecciones se llaman curvas de nivel de la superficie. Para llevar a cabo esta tarea, usamos la opción `contour` que usa un mallado del plano xy generado por la función `meshgrid()` y el valor del eje z adquiere un valor definido por una función. Veamos un ejemplo [12]:

```
from pylab import *
def f(x, y):
    return np.sin(x) ** 2 + np.cos(5 + x * y) + x
x = np.linspace(0, 5, 100)
y = np.linspace(0, 5, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
contour(X,Y,Z)
show()
```

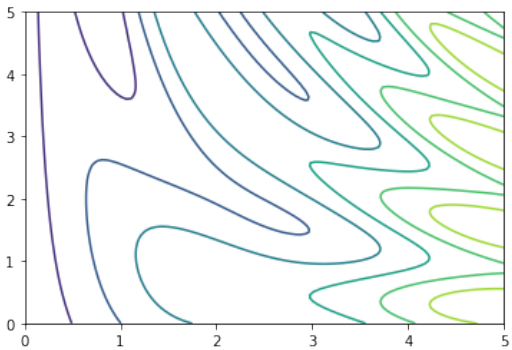


Figura 5.16: Visualización de las curvas de nivel generadas por la proyección en varios planos en z de la superficie $\sin(x)^2 + \cos(5 + xy) + x$.

Si se requiere que las curvas de nivel sean rellenadas, cambiamos la función `contour` por la función `contourf`.

```
from pylab import *
def f(x, y):
    return np.sin(x) ** 2 + np.cos(5 + x * y) + x
x = np.linspace(0, 5, 100)
y = np.linspace(0, 5, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
contourf(X,Y,Z)
show()
```

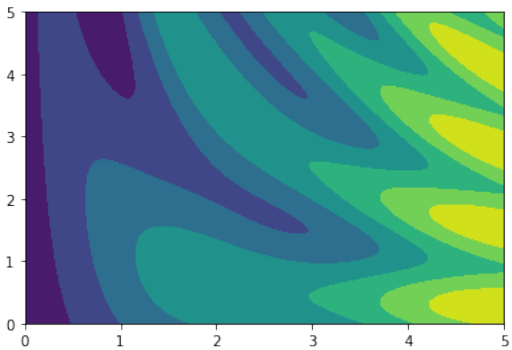


Figura 5.17: Visualización de las curvas de nivel rellenas generadas por la proyección en varios planos en z de la superficie $\sin(x)^2 + \cos(5 + xy) + x$.

Si queremos saber que valor de z representa el color visualizado, podemos incorporar una barra de color donde se muestre la escala:

```
from pylab import *
def f(x, y):
    return np.sin(x) ** 2 + np.cos(5 + x * y) + x
x = np.linspace(0, 5, 100)
y = np.linspace(0, 5, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
g= contourf(X,Y,Z)
colorbar(g)
show()
```

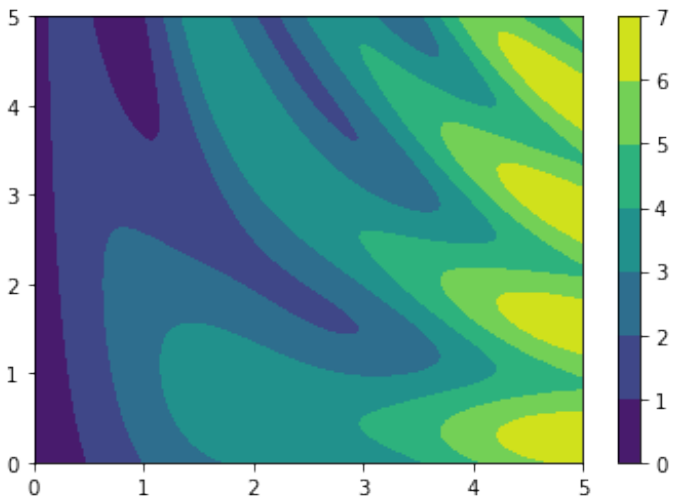


Figura 5.18: Visualización de las curvas de nivel rellenas, generadas por la proyección en varios planos en z de la superficie $\sin(x)^2 + \cos(5 + xy) + x$. Se ha agregado una barra de color para saber la interpretación de los colores en función de la variable z.

Para poner la barra de color, es necesario llamar a la gráfica como un objeto g que será visualizado y al que se le aplicará la función `colorbar` para que se muestre la barra de colores.

5.1.4. Histogramas

Para graficar valores discretos o distribuciones estadísticas, es útil usar un histograma. También son útiles para plotear una distribución de números que cruzan un rango de posibles valores. Como ejemplo se tomará una lista de números aleatorios y se verá cuantas veces de ocurrencia presentan. Los números aleatorios serán generados con distribuciones normales o también conocidas como gaussianas:

```
from pyplot import *
from random import *

numeros = normal(size=1000)
hist(numeros)

title("Histograma Gaussiano")
xlabel("Valor")
ylabel("Frecuencia")
show()
```

La función `normal` pertenece a la librería `random` y genera los números aleatorios con la distribución que vemos en la figura `histo`. La función `hist()` está diseñada para graficar histogramas.

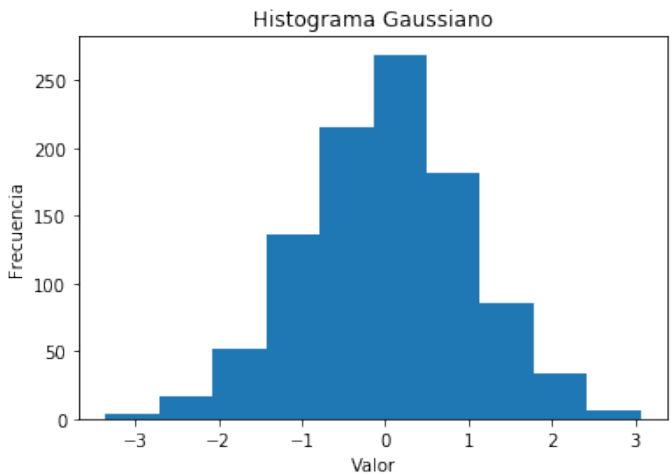


Figura 5.19: Visualización de una lista de números aleatorios versus el número de ocurrencia que presentan. Los números aleatorios son generados con una distribución normal o gaussiana.

La función `histo` tiene la opción `bins` como el número de distintas categorías mostradas en el eje x, el default en Python es diez. Si se quiere cambiar la resolución de nuestra gráfica, es decir, cambiar el número de `bins` a veinte, usamos `hist(numeros, bins=20)`. Si se quiere graficar una distribución de probabilidad en lugar de un conteo de frecuencias, se tendrá que dividir entre una constante de normalización. Por lo general, se quiere que el 100 % de probabilidad se encuentre en 1, `hist(numeros, bins=20,normed=True)`. Si se requiere cambiar la apariencia del histograma, que los cuadros no estén llenados, sino que sólo se vean los escalones, usamos la opción `histtype`, e.g., `hist(numeros, bins=20,normed=True,histtype='step')`.

A continuación, se hará un histograma con la distribución normal o gaussiana en azul y una distribución uniforme de números aleatorios desde -3 a 3 en rojo con 50 % de transparencia.

```
from pylab import *
from numpy.random import normal, uniform

numeros = normal(size=1000)
uniformes = uniform(low=-3, high=3, size=1000)
hist(numeros, bins=20, histtype='stepfilled', color='b',label='
    Gaussiana')
```

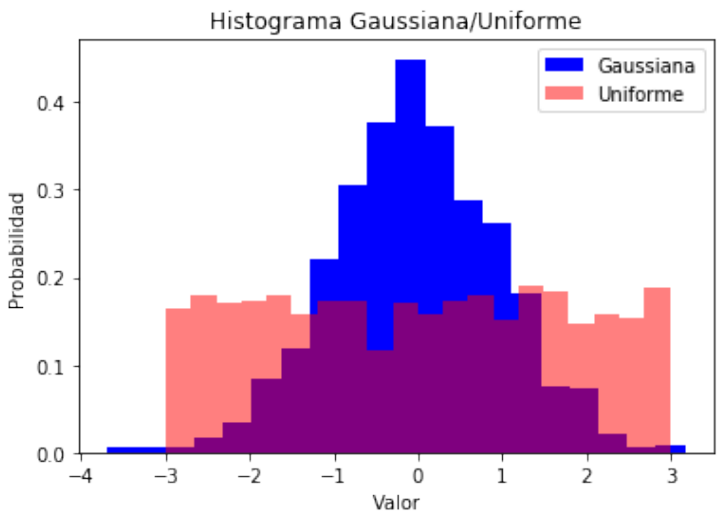


Figura 5.20: Histograma con la distribución normal y una distribución uniforme de números aleatorios desde -3 a 3 en rojo con 50 % de transparencia sobre el color azul de la distribución normal.

```
hist(uniformes, bins=20, histtype='stepfilled', color='r', alpha
    =0.5, label='Uniforme')
title("Histograma Gaussiana/Uniforme")
xlabel("Valor")
ylabel("Probabilidad")
legend()
savefig('uniforme.jpeg') #png, tiff,  eps,
```

5.1.5. Gráficas en 3D

Para habilitar la graficación en 3-d en Matplotlib, se tiene que habilitar un kit de herramientas extra *(toolkit)* llamado `mplot3d`. El kit de herramientas `mplot3d` agrega capacidades simples de trazado en 3D a `matplotlib` al proporcionar un objeto de ejes que puede crear una proyección 2D de una escena 3D. El gráfico resultante tendrá el mismo aspecto que los gráficos 2D normales. El kit de herramientas `mplot3d` incluye formatos interactivos que brindan la capacidad de rotar y hacer zoom en la escena 3D. Se puede rotar la escena 3D simplemente haciendo clic y arrastrando la escena. El zoom se realiza haciendo clic con el botón derecho en la escena y arrastrando el mouse hacia arriba y hacia abajo. Tenga en cuenta que no se usa el botón de zoom como se usaría para gráficos 2D normales.

A continuación se hará un ejemplo de una helicoidal con variación del radio. Para hacer esto, se cambia la forma en la que importamos las librerías, de `from matplotlib import *` a `import matplotlib as mpl`. La diferencia consiste en que, en la primera opción, cargamos toda la librería al inicio, mientras que en la segunda, vamos cargando las funciones conforme se van utilizando. Si se opta por la segunda opción, como es nuestro caso, se tendrá que indicar el modulo o librería y luego la función, esto es para que el intérprete de Python sepa de donde proviene la función en cuestión:

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='helicoidal')
ax.legend()
plt.show()
```

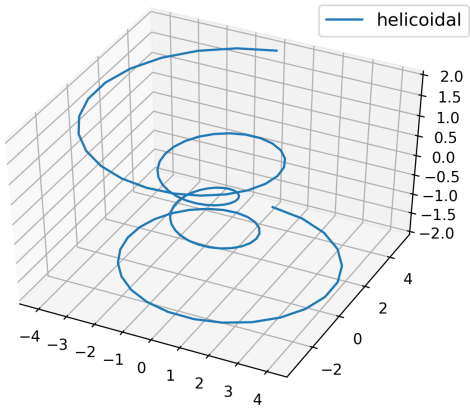


Figura 5.21: Representación tridimensional de una helicoidal con radio variante.

En el código se importa la librería `matplotlib` y se le asigna un alias de `mpl`. En seguida, del kit de herramientas `mplot3d` importamos la función `Axes3D`, la cual asigna un eje coordenado en 3-d. Importamos la librería `pyplot` como `plt` debido a que contiene funciones extras para plotear, como es la función `figure()`, que asigna la figura o gráfica a generar a una variable `fig = plt.figure()` y a esta variables se le irán agregando elementos, como sería el tamaño de la figura, el título, etc. La línea `ax = fig.gca(projection='3d')` asigna a una variable `ax` la gráfica en 3-d que está insertada en la figura `fig`. Por último, hacemos la gráficación en 3-d sobre `ax` y no sobre `fig` dado que `ax` contiene todos los elementos 3-d, esto se hace con la función `ax.plot(x, y, z, label='helicoidal')` recibiendo 3 listas:

Los gráficos de superficie son un excelente ejemplo para visualizar las relaciones entre tres variables en todo 3-d, ya que proporcionan una estructura completa y una visión de cómo cambia el valor de cada variable en los ejes de las otras dos. La construcción de una gráfica de superficie con `mplot3d` es un proceso de tres pasos.

Primero, es necesario generar los puntos que conformarán la gráfica de superficie. Si somos estrictos, por continuidad, se necesitaría un conjunto infinito de números para generar la superficie. Sin embargo, como esto es una tarea imposible, se genera un conjunto que sea suficiente para graficar estos puntos, por ejemplo, se usa cien o mil puntos en cada eje. Quedan arbitrarios los puntos en x y y y luego se calculan los puntos z usando una función:

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = plt.axes(projection="3d")
def z_function(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = z_function(X, Y)
```

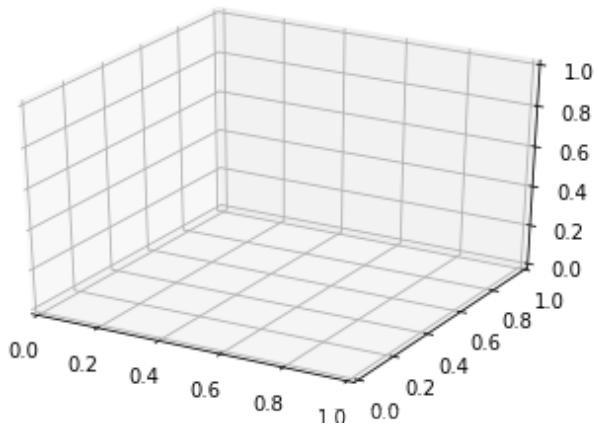


Figura 5.22: Representación del objeto desplegado con `figure()` y `axes()`, el cual se irá llenando con los ploteos deseados.

Segundo, trazar una estructura de alambre dentro de la función a programar:

```
fig = plt.figure()
ax = plt.axes(projection="3d")
ax.plot_wireframe(X, Y, Z, color='green')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```

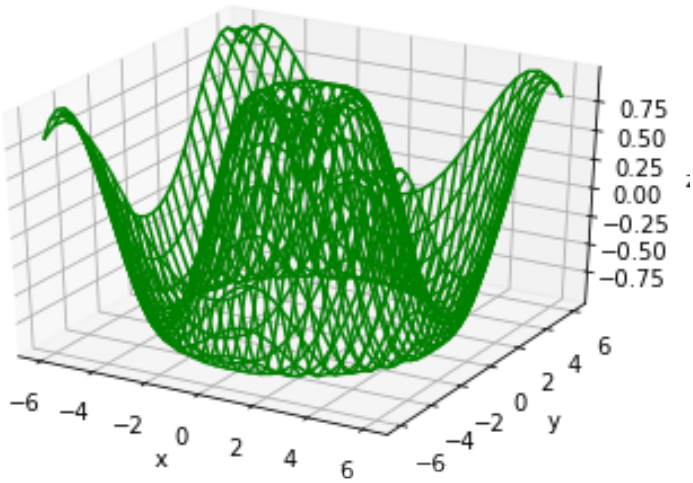


Figura 5.23: Representación como alambre de la función.

Finalmente, se proyecta la superficie dentro de la estimación de estructura de alambre o mallado y se extrapolan todos los puntos:

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, cmap='twilight') # opciones para cmap
    winter, hot ,twilight, summer, spring, magma, Spectral,
    rainbow, prism, plasma
```

Ahora, se hará un ejemplo un poco más interesante, un toro de revolución que sigue las ecuaciones:

$$\begin{aligned} X &= (R + r\cos(\phi))\cos(\theta) \\ Y &= (R + r\cos(\phi))\sin(\theta) \\ Z &= r\sin(\phi) \end{aligned}$$

R es el radio mayor del toro de revolución y r es el radio menor, el radio del círculo generador. Se genera un mallado que se presentará en términos del ángulo azimutal ϕ y el ángulo axial θ:

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Generar el mallado del toro
```

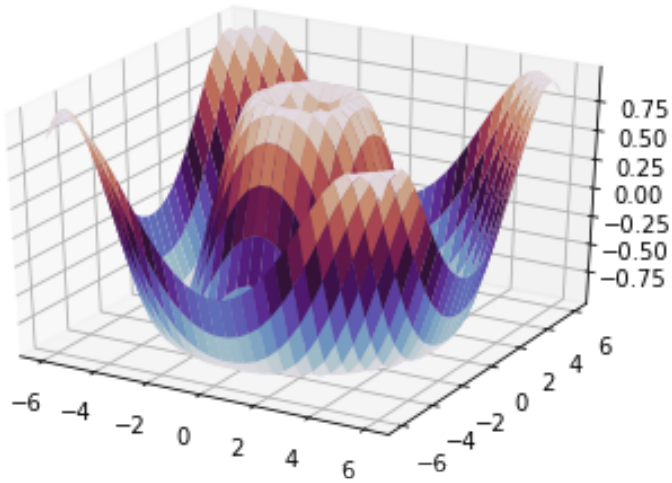


Figura 5.24: Representación de la función coloreada con un mapa de colores relacionados con el atardecer (`cmap='twilight'`).

```
angulo = np.linspace(0, 2 * np.pi, 32)
theta, phi = np.meshgrid(angulo, angulo)

#Generamos el toro de revolución
r, R = .25, 1.
X = (R + r * np.cos(phi)) * np.cos(theta)
Y = (R + r * np.cos(phi)) * np.sin(theta)
Z = r * np.sin(phi)

fig = plt.figure()
ax = fig.gca(projection = '3d')
ax.set_zlim3d(-1, 1)
ax.plot_surface(X, Y, Z, color = 'w')
plt.show()
```

En esta visualización se ha agregado un mallado uniforme en el toro de revolución usando la función `meshgrid()`. Se ha limitado los parámetros del eje z, tal que la figura tenga los mismos valores en los tres ejes, esto con el objetivo de que el toro de revolución sea visible. Se recomienda al lector que remueva la línea `ax.set_zlim3d(-1, 1)` y juegue con los parámetros para encontrar la mejor visualización del toro de revolución.

Una de las cosas más divertidas que se pueden hacer con `mplot3d` es plotear con colores, tal que los colores cambien en función de alguna variable. En el siguiente ejemplo, los colores cambiarán en términos de la variable z. Esto se hace con la línea `colors.Normalize(min(z), max(z)`, la función `min(z)` da el valor mínimo y `max(z)` da el máximo. Graficamos punto por punto hasta llegar a cien usando un `for`:

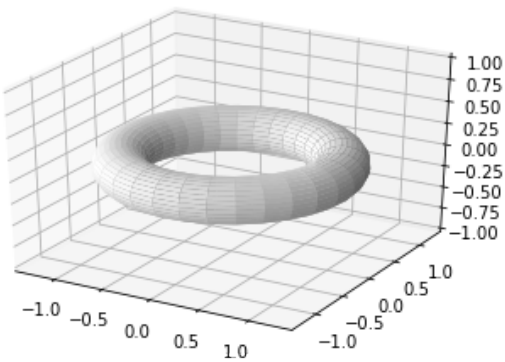


Figura 5.25: Representación de un toro de revolución con un mallado uniforme.

```
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import colors
fig = plt.figure()
ax = fig.gca(projection='3d')
N = 100
y = np.ones((N,1))
x = np.arange(1,N + 1)
z = 5*np.sin(x/5.)
cn = colors.Normalize(min(z), max(z))
for i in range(N-1):
    ax.plot(x[i:i+2], y[i:i+2], z[i:i+2],
      color=plt.cm.jet(cn(z[i])))
plt.show()
```

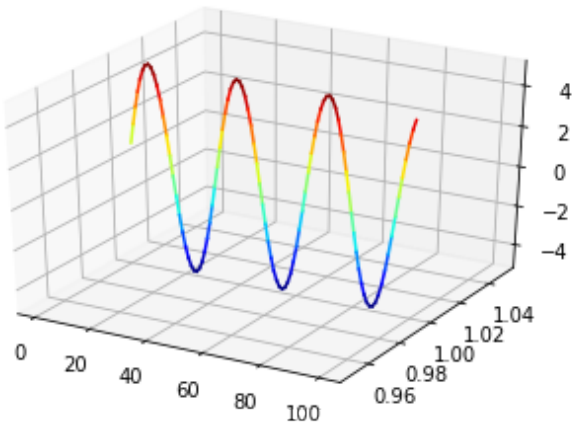


Figura 5.26: Visualización con colores que cambian en función de la variable z.

5.2. SymPy

SymPy es un paquete de Python de computación simbólica. La computación simbólica se ocupa de la computación de objetos matemáticos simbólicamente. Esto significa que los objetos matemáticos se representan exactamente, no aproximadamente, y las expresiones matemáticas con variables no evaluadas se dejan en forma simbólica.

Sympy permite al usuario realizar operaciones numéricas y operaciones analíticas o con símbolos. Por ejemplo:

```
import math
import sympy
l1=math.sqrt(8)
l2=sympy.sqrt(8)
print(l1,l2)
```

```
Out: (2.8284271247461903, 2*sqrt(2))
```

La variable l_1 es la cantidad evaluada numéricamente, mientras que, la variable l_2 es la cantidad indicada simbólicamente guardada por SymPy.

Sympy reconoce tres tipos de variables: Reales, Enteros y Racionales. Las dos primeras ya se han visto anteriormente, la clase `Rational` representa un número racional, dejando indicada la fracción:

```
a=sympy.Rational(1,4)
print(a)
```

```
Out: 1/4
```

SymPy usa la librería `mpmath` como auxiliar, lo que hace posible ejecutar cálculos numéricos usando aritmética con precisión arbitraria. Debido a esto algunas constantes especiales como e, π o ∞ son tratadas como símbolos y pueden ser evaluadas numéricamente:

```
sympy.exp,sympy.pi,sympy.oo
```

```
Out: (exp, pi, oo)
```

También existe la función `evalf()` que evalúa la función:

```
from sympy import *
pi.evalf(),(pi**2+exp(1)).evalf(),oo.evalf()
```

```
Out: (3.14159265358979, 12.5878862295484, oo)
```

La función `evalf()` es una buena herramienta si se quiere hacer evaluaciones simples, pero, si se tiene la intención de evaluar una expresión más compleja o con más puntos, existen formas más eficientes. Por ejemplo, si se quisiera evaluar una expresión en mil puntos, usar solo SymPy sería mucho más lento de lo necesario. En su lugar, se podrá auxiliar con la librería NumPy. La forma más sencilla de convertir una expresión simbólica en una expresión que se pueda evaluar numéricamente es utilizar la función `lambdify()`, que actúa como una función `lambda`, excepto que convierte los nombres de SymPy a los nombres de la biblioteca numérica dada, generalmente NumPy. Por ejemplo:

```
from sympy import *
import numpy
a = numpy.arange(10)
expr = cos(x)
f = lambdify(x, expr, "numpy")
print(f(a))
```

```
Out: [ 1.          0.54030231 -0.41614684 -0.9899925
  -0.65364362  0.28366219 0.96017029  0.75390225
  -0.14550003 -0.91113026]
```

Otro ejemplo del uso de `lambdify()`:

```
f = lambdify(x, x + 1)
print(f(1),f(2))
```

```
Out: 2 3
```

Hasta el momento, se ha usado SymPy como una calculadora bastante poderosa, pero no se ha hecho nada simbólico. Para empezar con el cálculo simbólico, lo primero que se tiene que hacer es declarar las variables simbólicas de forma explícita:

```
import sympy
x=sympy.Symbol('x')
y=sympy.Symbol('y')
z=sympy.Symbol('z')
```

De esta manera, las variables simbólicas pueden ser usadas con total libertad:

```
print(x+y+z)
print(x**y/z)
print((x+y)**3)
print(expand((x+y)**3))
print(expand(cos(x+y)))
print(expand(cos(x+y), trig=True))
```

```
x + y + z
x**y/z
(x + y)**3
x**3 + 3*x**2*y + 3*x*y**2 + y**3
cos(x + y)
-sin(x)*sin(y) + cos(x)*cos(y)
```

La función `expand()` regresa la expansión algebraica del polinomio introducido como argumento. La opción `trig=True` indica que se usen identidades trigonométricas para la expansión:

```
expand(cos(x+y), trig=True)
```

```
-sin(x)sin(y)+cos(x)cos(y)
```

Si por el contrario de expandir, se desea simplificar, se utilizará la función `simplify()`:

```
simplify((x+x*y)/x)
```

```
y+1
```

La simplificación es muy general, pero si queremos ser más específicos y que la simplificación se haga de una forma en particular, se pueden usar alternativas más precisas que `simplify()`: `powsimp()` (simplificación de exponentes), `trigsimp()` (para expresiones trigonométricas), `logcombine()`, `radsimp()` y `together()`:

```
trigsimp(cos(x)/sin(x))
```

```
tan(x)
```

También SymPy tiene la opción para resolver ecuaciones algebraicas, como sería $x^2 - 2 = 0$.

```
solve(x**2 - 2, x)
```

```
[-sqrt(2), sqrt(2)]
```

El poder real de un sistema de cálculo simbólico como SymPy es la capacidad de hacer todo tipo de cálculos simbólicamente. SymPy puede simplificar expresiones, calcular derivadas, integrales y límites, resolver ecuaciones, trabajar con matrices y mucho, mucho más y hacerlo todo simbólicamente. Incluye módulos para graficar, imprimir en LaTeX, módulos especializados en física, estadística, combinatoria, teoría de números, geometría, lógica y más.

Posiblemente, el lector no esté totalmente familiarizado con los ejemplos que se verán a continuación, ya que son tópicos avanzados. Sin embargo, creo que es importante que el lector haga estos ejercicios antes de adentrarse a los temas, ya que le ayudarán a tener la comprobación computacional de cualquier ejercicio que pudiera llegar a hacer.

Los límites son fáciles de usar en SymPy con la sintaxis `limit(f, variable, punto)`. Para calcular el límite de $f(x)$ como $x \to 0$, deberás usar la expresión `limit(f, x, 0)`:

```
print(limit(sin(x)/x,x,0))
print(limit(x**x,x,0))
print(limit(x, x, oo))
print(limit(1/x**2, x, oo))
```

```
1
1
oo
0
```

La derivación de cualquier expresión usa la función `diff(func,var)`, por ejemplo:

```
diff(sin(x), x)
```

```
cos(x)
```

Otro ejemplo sería:

```
diff(sin(2*x), x)
```

```
2cos(2x)
```

Para comprobar que tan correcto es la derivación que presenta SymPy, haremos una derivada con la función `diff()` y otra con la definición de derivada:

$$f'(x) = \lim_{y \to 0} \frac{f(x+y) - f(x)}{y}$$

```
print('usando la funcion diff',diff(tan(x), x))
print('usando la definición',limit((tan(x+y)-tan(x))/y, y, 0))
```

```
usando la función diff tan(x)**2 + 1
usando la definición tan(x)**2 + 1
```

La función `diff()` también permite tener derivadas de orden superior, n, usando la sintaxis `diff(func, var, n)`. Por ejemplo:

```
print(diff(sin(2*x), x, 1)) #orden 1
print(diff(sin(2*x), x, 2)) #orden 2
print(diff(sin(2*x), x, 3)) #orden 3
```

```
2*cos(2*x)
-4*sin(2*x)
-8*cos(2*x)
```

Si se utiliza un Jupyter Notebook, se puede modificar la visualización de salida de la respuesta del intérprete. Para esto, está la opción `init_printing(use_unicode=True)`.

```
from sympy import *
x, t, z, nu = symbols('x t z nu')
init_printing(use_unicode=True)
diff(sin(x)*exp(x), x)
```

$$e^x \sin(x) + e^x \cos(x)$$

Así mismo, se pueden obtener expansiones de series de Taylor. Las series de Taylor tienen la siguiente forma:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^n(x)}{n!}(x-a)^n$$

Por ejemplo, la serie de Taylor de la función $\cos(x)$ es:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

o la serie de Taylor de la función $\exp(x)$ es:

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} - \cdots$$

La librería SymPy las calcula de la siguiente forma:

```
print(series(cos(x), x))
print(series(exp(x), x))
```

```
1 - x**2/2 + x**4/24 + O(x**6)
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + O(x**6)
```

se redondean los términos hasta sexto orden con $\mathcal{O}(x**6)$.
SymPy permite obtener integrales indefinidas o definidas de funciones especiales o de funciones trascendentes elementales. La función que utiliza para esta tarea es `integrate()`, que usa una extensión del algoritmo Risch-Norman y reconocimiento de patrones. Un ejemplo básico será la integral indefinida de:

$$\int \frac{1}{x} dx$$

```
integrate(1/x,x)
```

```
log(x)
```

otro ejemplo sería:

```
integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
```

$$e^x \sin(x)$$

Por último, se hará una integral definida de $-\infty$ a ∞:

```
integrate(sin(x**2), (x, -oo, oo))
```

$$\frac{\sqrt{2}\sqrt{\pi}}{2}$$

Sympy tiene su propio tipo de variable asignado a matriz, con él se pueden definir matrices numéricas o simbólicas y operar con ellas:

```
Matrix([[1,0], [0,1]])
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

A diferencia de un `array` o `matrix` de NumPy, en una matriz de Sympy se pueden incluir símbolos:

```
x = Symbol('x')
y = Symbol('y')
A = Matrix([[1,x], [y,1]])
print(A)
print(A**2)
```

```
Matrix([[1, x], [y, 1]])
Matrix([[x*y + 1, 2*x], [2*y, x*y + 1]])
```

Muchas de las funciones anteriores ya existen en Numpy con el mismo nombre (`ones()`, `eye()`, etc.), pero son redefinidas en SymPy con el mismo nombre, por lo que es recomendable importar los módulos como `import numpy as np` para indicar de que módulo es la función en cuestión. Por ejemplo:

```
import numpy as np
import sympy as sp
print('Identidad en SymPy',sp.eye(3))
print('Identidad en NumPy',np.eye(3))
```

```
Identidad en SymPy Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
Identidad en NumPy [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Nótese la diferencia, en SymPy, se guardó como un objeto tipo `Matrix`, mientras que en NumPy, se despliega como un arreglo. Sin embargo, a pesar de las diferencias, es posible operar entre ellas, salvo que las matrices de Numpy no pueden tener una conversión simbólica. La selección de elementos de matrices de Sympy se hace de forma idéntica a los `array` o `matrix` de Numpy:

```
# Multiplico la matriz identidad por x
x = sp.Symbol('x')
M = sp.eye(3) * x
M
```

$$\begin{bmatrix} x & 0 & 0 \\ 0 & x & 0 \\ 0 & 0 & x \end{bmatrix}$$

Podemos hacer sustituciones de variables por otras, e.g., sustituir la variable x por y:

```
y = sp.Symbol('y')
M.subs(x, y)
```

$$\begin{bmatrix} y & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & y \end{bmatrix}$$

Así mismo, se puede obtener un determinante de una matriz:

```
M = sp.Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
M.det()
```

```
-28
```

O la inversa de una matriz:

```
M.inv()
```

$$\begin{bmatrix} -\frac{3}{14} & \frac{1}{14} & \frac{1}{2} \\ -\frac{1}{28} & \frac{5}{28} & -\frac{1}{14} \\ \frac{3}{7} & -\frac{1}{7} & y \end{bmatrix}$$

Para finalizar la parte de matrices, SymPy tiene habilitadas las funciones `eigenvals()` y `eigenvects()` que regresan los eigenvalores o valores propios de una matriz y los eigenvectores o vectores propios:

```
M = Matrix([[3, -2,  4, -2], [5,  3, -3, -2], [5, -2,  2, -2],
    [5, -2, -3,  3]])
print(M.eigenvals())
print(M.eigenvects())
```

```
{3: 1, -2: 1, 5: 2}
[(-2, 1, [Matrix([
[0],
[1],
[1],
[1]])]), (3, 1, [Matrix([
[1],
[1],
[1],
[1]])]), (5, 2, [Matrix([
[1],
[1],
[1],
[0]]), Matrix([
[ 0],
[-1],
[ 0],
[ 1]])])]
```

5.3. Ejercicios

1. Haz la siguiente curva paramétrica en Matplotlib:

$$x = r(t - \sin(t))$$
$$y = r(1 - \cos(t))$$

2. Repite la gráfica 5.11, pero en lugar de visualizar la hipocicloide, plotea la epicicloide con la ecuación:

$$x = (R_1 - R)\cos(t) + R\cos\left(\frac{R_1 - R}{R}t\right)$$
$$y = (R_1 - R)\sin(t) - R\sin\left(\frac{R_1 - R}{R}t\right)$$

3. Realiza la visualización para la operación vectorial $\vec{b} = c\vec{a}$ con $\vec{a} = (-1.7, 1.5)$, juega con c y explica que pasa.

4. Escribe una expresión simbólica para

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Recuerda que la función exponencial en sympy es $\exp(x)$ y crear las variables simbólicas σ y μ.

5. Calcula $\lim_{x\to 0} \frac{\sin(x)}{x}$.
6. Calcula la derivada de $\log(x)$ para x.
7. Con `solve()`, también se pueden resolver sistemas de ecuaciones de manera simbólica. Ejemplo:

$$\begin{aligned} x + 5y - 2 &= 0 \\ -3x + 6y - 15 &= 0 \end{aligned}$$

Investiga como resolver este sistema de ecuaciones. Recuerda que en el intérprete de Python puedes usar la opción `solve?` para entrar al manual.

8. Crea la siguiente matriz con `sympy`

$$\begin{bmatrix} 1 & 0 & 1 \\ -1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

Ahora crea una interpretación matricial del vector columna:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Multiplica las dos matrices para obtener

$$\begin{bmatrix} x + z \\ -x + 2y + 3z \\ x + 2y + 3z \end{bmatrix}$$

Capítulo 6

Introducción al manejo de datos numéricos y experimentales

6.1. Errores numéricos

Existen dos causas principales de errores en cálculos numéricos:

- Errores de truncamiento: Se debe a las aproximaciones utilizadas en la fórmula matemática del modelo. Podría decirse, que es donde se corta la serie de potencias de la aproximación.

  Ejemplos:
  i) Serie de Taylor:

  $$e^x \approx 1 + x + \frac{1}{2}x^2$$

  ii) Diferenciación:

  $$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h}$$

  Para una h finita dada por el programador.

- Errores de redondeo: Se asocia con el número limitado de dígitos con que se representan los números de una computadora. Los errores de redondeo están muy asociados con la capacidad de la computadora.

En cálculos numéricos, los números decimales que se utilizan quizá no sean exactos. Estos números casi siempre se redondean de manera intencional y está limitación de dígitos de la computadora causa los errores de redondeo.

En el siguiente código se analizará una simple suma de flotantes, $x = 0.1 + 0.2$. Se hará una sentencia condicional, tal que si no hay error de precisión Python regresará la suma exacta, caso contrario, regresará un error.

```
x=0.1+0.2
if x==0.3:
    print(x)
else:
    print('Error: x distinto a 0.3')
```

```
Error: x distinto a 0.3
```

¿Por qué no se imprimió x terminando el `if`? Ocurre que, por error de redondeo 0.1+0.2 no es exactamente 0.3, sino que es 0.300000000000003. Por lo cual no se cumple la condición del `if`. Esto es bastante común en el cómputo científico y en la física computacional, por lo que la mejor manera de tratar esto, es introducir un valor épsilon en el código. El valor épsilon es una tolerancia, tal que si el número obtenido se aproxima al valor real $\pm$ la tolerancia, se cumple la condición. A continuación se muestra el código con la tolerancia:

```
epsilon=1e-12
if abs(x-0.3)<epsilon:
    print(x)
```

```
0.30000000000000004
```

En este caso, si el número coincide en 12 cifras decimales a la solución esperada, se imprimen en pantalla. Aquí podemos decir que el error de redondeo es de 1×10^{-12}.

En algunos casos, los errores de redondeo causan efectos muy serios y hacen que los resultados de los cálculos carezcan de sentido. Por lo tanto, es importante aprender algunos aspectos básicos de las operaciones aritméticas en las computadoras y comprender bajo que circunstancias pueden ocurrir severos errores de redondeo.

El error total será la suma del error de truncamiento y el error de redondeo.

$$E_{\text{total}} = E_{\text{truncamiento}} + E_{\text{redondeo}}$$

Generalmente el error por truncamiento es mucho mayor, en ordenes de magnitud, al error de redondeo, por lo que el error total sólo será el error de truncamiento, por ejemplo, si el error de truncamiento es del orden de 1×10^{-3} y el error de precisión es 1×10^{-16}, que es el valor común en Python para las variables de simple precisión, el error total será proporcional al error de truncamiento 1×10^{-3}.

Los errores numéricos son tratados como si fuesen incertidumbres en el laboratorio. Por lo que analizaremos el caso de los errores experimentales y los errores numéricos en paralelo. Para una lectura más especializada del tema, se recomienda el libro de Berta Oda [13].

6.1.1. Cálculos numéricos directos

Cálculos numéricos reproducibles

Si no se detectan variación de una medida a otra, quiere decir que el error no rebasa la mitad de la mínima escala del instrumento de medición. Por ejemplo, si al medir la longitud de un objeto con una regla cuya mínima escala está en mm se obtiene 28.4 cm, la incertidumbre será de 0.005 cm y el resultado se reporta como:

$$(28.4 \pm 0.05cm)$$

En computación, el análogo a las mediciones directas se hará con la selección de los valores de la variable independiente, el valor de las abscisas del cálculo numérico. La incertidumbre será la mitad de la mínima escala escogida para recorrer el eje de las abscisas, por ejemplo, si caminamos en x con pasos de h, la incertidumbre será $h/2$.

Cálculos numéricos no reproducibles

Supóngase que se realiza un experimento en el que se hace una medición en repetidas ocasiones fijando condiciones, pero se presentan variaciones en ellas. En computación, pueden ser valores pseudoaleatorios. Para este tipo de casos, se dice que el valor más representativo es el promedio:

$$\overline{x} = \frac{x_1 + x_2 + x_3 + \cdots + x_n}{n} = \sum_{i=1}^{n} \frac{x_i}{n},$$

donde x_i son las mediciones experimentales o valores computacionales obtenidos en las n repeticiones. ¿Cómo encontramos los errores de redondeo y/o truncamiento para estos datos no reproducibles? En general, el tratamiento riguroso de incertidumbres puede hacerse con estadística. El error debe reflejar la dispersión de los valores obtenidos y, de preferencia, debe ser muy pequeña. Cuando el promedio se toma como el valor verdadero, entonces, el error será asignado como la desviación absoluta máxima, que es simplemente la mayor de las diferencias absolutas entre el valor promedio y las lecturas obtenidas.

Por ejemplo, se hace una medición del tiempo de enfriamiento de un recipiente con agua. El recipiente es calentado hasta 90 grados Celsius y se mide el tiempo que tarda en llegar a temperatura ambiente. Se repite el experimento 5 veces:

t1	33.45
t2	35.25
t3	30.05
t4	29.95
t5	32.05

donde el promedio de los tiempos representa el tiempo más probable ocurrido:

$$\bar{t} = \frac{t_1 + t_2 + t_3 + t_4 + t_5}{5} = 32.15$$

El error asociado se obtendrá haciendo la diferencia entre el valor máximo y el valor mínimo. Es decir:

$$35.45 - 29.65 = 5.30$$

El resultado se reporta de la siguiente manera:

$$t = 32.15 \pm 5.30$$

```
tiempo=[33.45,35.25,30.05,29.95,32.05]
n=len(tiempo)
suma=0
for i in range(n):
    suma+=tiempo[i]
promedio=suma/n
error=max(tiempo)-min(tiempo)
print('El tiempo resultante es: ', promedio,'+-',error)
```

```
El tiempo resultante es:  32.15 +- 5.300000000000001
```

Otro ejemplo computacional será usar un generador de números pseudoulatorios:

```
from random import *
l=[]
for i in range(10):
    x=uniform(0.8,2.8)
    l.append(x)
n=len(l)
suma=0
for i in range(n):
    suma+=l[i]
promedio=suma/n
error=max(l)-min(l)
print('La variable pseudoaleatoria resultante es: \n', promedio,'
    +-',error)
```

```
La variable pseudoaleatoria resultante es:
 1.9741974804045994 +- 1.8850339195375183
```

Si el número de repeticiones aumenta, la dispersión del error se hará mayor, dado que la desviación absoluta máxima crecerá, por lo que habrá que pensar mejor en el método para obtener las incertidumbres.

Error absoluto

Se define al error absoluto como la diferencia absoluta entre el valor verdadero de una magnitud y el valor obtenido por la computadora. Por ejemplo, ¿Cuál es el valor real del número irracional e y su aproximación $\frac{19}{7}$?

$$E_{\text{abs}} = |\text{valor}_{\text{real}} - \text{valor}_{\text{aprox}}|$$

```
from math import *
def error_absoluto(real,approx):
    return abs(real-approx)
x_teorica=e
x_approx=19/7
print( error_absoluto(x_teorica,x_approx))
```

```
[Out]  0.003996114173330678
```

Error relativo

Es el cociente que resulta entre el error absoluto y el valor real:

$$E_{\text{relativo}} = \frac{\text{Error}_{\text{abs}}}{\text{valor}_{\text{real}}}$$

```
from math import *
def error_relativo(real,approx):
    error_absoluto= abs(real-approx)
    return error_absoluto/real
x_teorica=e
x_approx=19/7
print(error_relativo(x_teorica,x_approx))
```

```
[Out]  0.00147008824894217
```

Hay varias consideraciones que tomar si no se conoce el valor teórico, lo que es muy frecuente. La primer consideración es que el valor teórico puede ser tomado como el valor promedio de las mediciones, o aproximado al valor calculado y el error absoluto se tomará como el error del cálculo numérico.
A modo de ejemplo, supongamos que no conocemos los valores de pi y e, pero sabemos que se aproximan $pi \approx 22/7$ y $e \approx 19/7$ y que tienen un error absoluto aproximado a 0.003996. Entonces, el error relativo de las dos mediciones es:
Para el valor de pi:

$$E_{\text{relativo}} = \frac{0.003996}{22/7} = 0.0012715$$

Y error relativo del valor de e:

$$E_{\text{relativo}} = \frac{0.003996}{19/7} = 0.0027972$$

Pensando en datos experimentales, con una cinta métrica con mínima escala de 0.01cm se toman las siguientes mediciones:

Ancho de una puerta	$a = 150.0\text{cm} \pm 0.05\text{cm}$
Largo de un lápiz	$a = 10.0\text{cm} \pm 0.05\text{cm}$

El error absoluto será tomado como el error directo obtenido de la cinta métrica.
El error relativo para el ancho de la puerta es:

$$E_{\text{relativo}} = \frac{0.05\text{cm}}{150.0\text{cm}} = 0.00033.$$

Error relativo del largo del lápiz:

$$E_{\text{relativo}} = \frac{0.05\text{cm}}{10.0\text{cm}} = 0.0055$$

El error absoluto es el mismo para la medición del ancho de la puerta y del largo del lápiz, pero no es lo mismo repartir un error de 0.05cm en 150.0cm que en 10.0cm. Como puede verse, el error relativo da el error en proporción de la medición.

Error porcentual

Es el índice más comúnmente usado para especificar la precisión de una medida y se evalúa multiplicando la incertidumbre relativa por 100 %:

$$E_{\%} = (E_{\text{relativo}})(100\,\%)$$

Para el caso en la aproximación de e con el conocimiento del valor teórico de e, se tiene:

```
from math import *
def error_por(real,approx):
    error_relativo= (abs(real-approx))/real
    return error_relativo*100
x_teorica=e
x_approx=19/7
print(error_por(x_teorica,x_approx),'%')
```

```
[Out]   0.14700882489421702%
```

Para los ejemplos con los valores aproximados de pi y e, los errores porcentuales son:

$$E_{\%\text{pi}} = 0.0012715 \times 100\,\% = 0.127\,\%$$

$$E_{\%\text{e}} = 0.0027972 \times 100\,\% = 0.279\,\%$$

Con respecto al ejemplo de la medición de la cinta métrica, se dice que el error porcentual de las mediciones son:

$$E_{\%\text{puerta}} = 0.00033 \times 100\,\% = 0.033\,\%$$

$$E_{\%\text{lápiz}} = 0.0055 \times 100\,\% = 0.55\,\%$$

6.1.2. Cálculos numéricos indirectos

Los cálculos numéricos indirectos son aquellos que se obtienen como resultado de operaciones realizadas con dos o más cálculos numéricos directos. Si continuamos con la analogía de entre el cálculo numérico y los datos experimentales, decimos que son como las mediciones indirectas. Ejemplo de algunas mediciones indirectas:

- La constante de Planck.
- La carga del electrón.
- El radio de la tierra.

Las mediciones indirectas en computación serán aquellas que obtenemos combinando en una operación matemática, dos variables determinadas previamente. En las mediciones indirectas, el error numérico de cada una de las variables se propaga hasta el resultado final, por lo que es indispensable que se considere para determinar correctamente el error. Para entender mejor como evaluar el error numérico o experimental

de los cálculos numéricos indirectos o mediciones indirectas, es importante analizar como es la propagación del error. La propagación del error se analizará a detalle en la siguiente sección.

6.2. Propagación de errores numéricos

Cuando se realizan operaciones aritméticas finita con valores numéricos, el resultado siempre tiene un error numérico asociado a los dos operandos.

- Suma
  Si una magnitud es el resultado de la adición de otras dos:

  $$Z = X + Y$$

  Tanto X como Y tienen sus respectivos errores numéricos asociados; la propagación del error numérico dará como resultado el error de Z:

  $$X = X_0 \pm \delta X$$
  $$Y = Y_0 \pm \delta Y$$

  Se denota a X como la variable real, X_0 la variable numérica obtenida y δX el error numérico asociado a la obtención de X_0. Lo mismo ocurre para Y:

  $$Z = (X_0 \pm \delta X) + (Y_0 \pm \delta Y)$$
  $$Z = (X_0 + Y_0) \pm (\delta X + \delta Y)$$
  $$Z = Z_0 \pm \delta Z$$

  Como se puede ver, la propagación del error final resulta de la suma de los errores individuales.

- Resta
  Al igual que con la suma, se define la propagación del error en términos de los errores de las variables restadas:

  $$Z = X - Y$$
  $$Z = (X_0 \pm \delta x) - (Y_0 \pm \delta Y)$$

  $$Z = (X_0 \pm \delta X) - (Y_0 \pm \delta Y) = (X_0 - Y_0) \pm (\delta X + \delta Y)$$

- Multiplicación
  La propagación de incertidumbres para la multiplicación es:

$$Z = XY$$

$$X = X_0 \pm \delta X$$
$$Y = Y_0 \pm \delta Y$$

  Por lo que Z queda:

$$Z = (X_0 \pm \delta X)(Y_0 \pm \delta Y)$$
$$Z = X_0 Y_0 \pm \delta X Y_0 \pm X_0 \delta Y \pm \delta X \delta Y$$

  El último termino es el producto de los errores numéricos, que tiene un valor muy pequeño a comparación con los otros términos, por lo que se considera despreciable:

$$Z = X_0 Y_0 \pm (X_0 \delta Y + Y_0 \delta X)$$
$$Z = Z_0 \pm \delta Z$$

  El error relativo de Z con δZ y Z_0 es:

$$E_{\text{relativo}} Z = \frac{\delta Z}{Z_0} = \frac{X_0 \delta Y + Y_0 \delta X}{X_0 Y_0}$$
$$E_{\text{relativo}} Z = \frac{\delta Y}{Y_0} + \frac{\delta X}{X_0}$$

  Donde el error relativo es la suma de los errores relativos de cada uno de los factores.

- División

  El tratamiento de propagación de errores por medio de la división de los errores es muy parecido al tratamiento de la multiplicación. Sea:

$$Z = \frac{X}{Y}$$

Donde definimos:

$$X = X_0 \pm \delta X$$
$$Y = Y_0 \pm \delta Y$$

Por lo cual, se tiene que:

$$Z = \frac{X_0 \pm \delta X}{Y_0 \pm \delta Y} = Z_0 \pm \delta z$$

A diferencia de los casos anteriores, en la división determinaremos los valores máximos y mínimos de Z siendo estos:
Valor máximo:

$$Z_0 + \delta Z = \frac{X_0 + \delta X}{Y_0 - \delta Y}$$

Valor mínimo:

$$Z_0 - \delta Z = \frac{X_0 - \delta X}{Y_0 + \delta Y}$$

Restando miembro a miembro el valor mínimo con respecto al máximo se obtiene:

$$(Z_0 + \delta Z) - (Z_0 - \delta Z) = 2\delta Z \tag{6.1}$$

Por otro lado:

$$\begin{aligned}(Z_0 + \delta Z) - (Z_0 - \delta Z) &= \frac{X_0 + \delta X}{Y_0 - \delta Y} - \frac{X_0 - \delta X}{Y_0 + \delta Y} \\ &= \frac{(Y_0 + \delta Y)(X_0 + \delta X) - (Y_0 - \delta Y)(X_0 + \delta X)}{(Y_0 + \delta Y)(Y_0 - \delta Y)}\end{aligned} \tag{6.2}$$

Igualando la ecuación 6.1 y 6.2:

$$\begin{aligned}2\delta Z = &\frac{Y_0X_0 + Y_0\delta X + X_0\delta Y + \delta Y\delta X}{Y_0^2 - (\delta Y)^2} \\ &- \frac{[Y_0X_0 - Y_0\delta X - X_0\delta Y + \delta Y\delta X]}{Y_0^2 - (\delta Y)^2}\end{aligned}$$

Manipulando algebraicamente los términos y despreciando la multiplicación de errores numéricos:

$$2\delta Z = \frac{2(X_0\delta Y + Y_0\delta X)}{Y_0^2}$$

Z queda de la siguiente forma:

$$Z_0 = \frac{X_0}{Y_0}$$

Por lo que el error numérico de Z es:

$$\delta Z = \frac{X_0 \delta Y + Y_0 \delta X}{Y_0^2}$$

Analizando el error relativo de Z:

$$E_{\text{relativo}} Z = \frac{\delta Z}{Z_0} = \frac{\frac{X_0 \delta Y + Y_0 \delta X}{Y_0^2}}{\frac{X_0}{Y_0}}$$

$$E_{\text{relativo}} Z = \frac{Y_0(X_0 \delta Y + Y_0 \delta X)}{X_0 Y_0^2} = \frac{\delta Y}{Y_0} + \frac{\delta X}{X_0}$$

Es interesante ver que, al igual que en la multiplicación, la división cumple con que el error relativo de la división es la suma de los errores relativos de las componentes.

A continuación se hará una función que reciba la lista de los datos x, los errores o incertidumbres relacionados a cada valor de x, los datos de y y la lista que contiene los errores o incertidumbres de y. Véase que ponemos el error en su propia lista, debido a que se puede tener el caso en que cada valor tenga su propio error. En el caso en que el error sea el mismo para todos los valores, se puede hacer una lista con el error repetido en todos los elementos con `deltax=[0.05]*n`, donde `n` es el número de datos obtenidos. La función queda como:

```
def incertidumbres_indirectas(x,deltax,y,deltay,operacion):
    w1,w2=[],[] #w1 y w2 guardaran los datos encontrados de las
    mediciones indirectas, z y deltaz
    if operacion=='suma':
        for i in range(len(x)):
            z=x[i]+y[i]
            deltaz=deltax[i]+deltay[i]
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    elif operacion=='resta':
        for i in range(len(x)):
            z=x[i]-y[i]
            deltaz=deltax[i]+deltay[i]
            w1.append(z)
```

```
            w2.append(deltaz)
        return w1,w2
    elif operacion=='multiplicacion':
        for i in range(len(x)):
            z=x[i]*y[i]
            deltaz=x[i]*deltay[i]+y[i]*deltax[i]
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    elif operacion=='division':
        for i in range(len(x)):
            z=x[i]/y[i]
            deltaz=(x[i]*deltay[i]+y[i]*deltax[i])/y[i]**2.0
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    else:
        print('operacion no definida, por lo que z y delta z
   seran ceros')
        w1,w2=zeros(len(x)),zeros(len(x))
        return w1,w2
```

El código mostrado arriba no solo sirve para evaluar los errores numéricos, sino que también es aplicable a la medición de incertidumbres en el laboratorio.
Algo que le falta al código, es que se puedan obtener las listas de otro archivo. Para esto, haremos una función que reciba un archivo con datos y regrese cuatro listas. La función queda como:

```
def lectura_datos(f):
    z=f.read()
    lista = z.split('\n')
    n=len(lista)
    l1,l2,l3,l4=[],[],[],[]
    for i in range(n):
        x=lista[i].split('\t')
        a,b,c,d=float(x[0]),float(x[1]),float(x[2]),float(x[3])
        l1.append(a)
        l2.append(b)
        l3.append(c)
        l4.append(d)
    return l1,l2,l3,l4

archivo_lectura=open('archivo.dat','r',errors="ignore") x,deltax,
   y,deltay=lectura_datos(archivo_lectura)
archivo_lectura.close()
```

Veamos a detalle la función de lectura de datos. Esta función recibe un archivo, hace la acción de leer el archivo que previamente se abrió `f.read()` y esta lectura le asigna una variable llamada `z`. La variable `z` guarda toda la información del archivo como una cadena de caracteres, e.g., `z=1\t2\t3\n4\t5\t6\n`.

El siguiente paso consiste en separar las líneas de `z` usando la secuencia de retorno `\n`, que significa salto de línea, y guardamos en una lista, cada línea de `z`, por ejemplo, `lista=[1\t2\t3,4\t5\t6]`. Ya que sabemos cuantos renglones tiene `z`, generamos listas vacías, en donde guardaremos los datos de cada columna, separados con `\t`. La separación de las columnas se hará con `x=lista[i].split('\t')` dentro de un `for` que va recorriendo los renglones. Por último, guardamos estos elementos en las listas previamente convertidas en flotantes y esto será lo que regrese la función.

Para usar la función, se tiene que abrir el archivo que se analizará. A este archivo lo llamaremos `archivo_lectura`, recibirá un estatus sólo de lectura y le diremos que, en caso de presentar error de lectura, los ignore. Ignoramos los errores de lectura debido a que puede existir un error en las codificaciones entre las versiones de los editores de texto, por ejemplo, nuestro editor de Python puede estar en `utf-8` y el editor de datos en `utf-16`. No olvidemos que todo lo que se abre, se tiene que cerrar, así que cerramos el archivo con `archivo_lectura.close()`.

La función que hemos hecho sirve muy bien para archivos con terminación **dat, txt, xls y xlsx**. Sin embargo, si los archivos tienen terminación **txt**, hay otra función más corta y cómoda, que ya está programada en el modulo `numpy`.

```
from numpy import loadtxt
datos=loadtxt('archivo.txt',float)
```

La ventaja de usar la función `loadtxt` es que nos ahorra hacer nuestra propia función, así como abrir y cerrar los archivos mediante Python. La desventaja es que sólo funciona para archivos de texto **txt**.

Otra forma de importar datos de un archivo **csv** será usar la librería `csv`[1], que permite abrir y cerrar archivos de este tipo. La parte del código a utilizar será:

```
import csv
with open('a.csv', 'r') as archivo:
    z = csv.reader(archivo)
    for renglon in z:
        print(renglon)
```

Con la misma lógica se puede hacer una función que guarde en un archivo, con la terminación que queramos, todos los datos obtenidos, incluyendo `z` y `deltaz`. En el apéndice B, se muestra el código completo de los errores o incertidumbres indirectas.

[1] Los archivos con extensión **csv** son archivos de datos en donde las columnas están separadas por comas.

6.3. Aproximación discreta por mínimos cuadrados

El objetivo del científico es describir los fenómenos naturales lo mejor posible. Podemos decir que tenemos pleno conocimiento de algún fenómeno cuando se tiene la capacidad de predecir lo que ocurrirá. Estas predicciones se pueden lograr expresando el fenómeno físico con una expresión matemática. Es muy importante abstraer y expresar en matemáticas los fenómenos, sólo así lograremos conocerlos a su plenitud.

Una manera en que el ser humano ha logrado describir los fenómenos, en específico, los fenómenos físicos, ha sido mediante la elaboración de experimentos, por lo que en esta sección nos enfocaremos en métodos computacionales para tratamiento de datos experimentales.

Empecemos con un ejemplo. Sea una fuerza aplicado a un resorte de material uniforme, por la ley de Hooke se establece que la fuerza será una función proporcional a la longitud de estiramiento $F(l) = k(l-E)$, donde $F(l)$ representa la fuerza requerida para extender el resorte l unidades, la constante E representa la longitud del resorte sin que se aplique fuerza alguna y la constante k es la constante del resorte.

Supongamos que queremos determinar la constante del resorte cuya longitud inicial es de 0.053 m. Aplicamos al resorte las fuerzas de 2.4 N y 6 N consecutivamente, por lo que observamos que su longitud aumenta a 0.070, 0.094 y 0.0123 m, respectivamente. En un análisis rápido de los datos experimentales, se puede ver que los datos no forman una línea recta. Aunque se podría utilizar un par aleatorio de estos puntos para aproximar la constante del resorte, parecería más razonable extender el estudio a más datos y así obtener una línea recta que aproxime mejor todos los puntos de datos. Los datos completos son:

Fuerza [N]	Longitud [m]
5.3	0
7	0.02
9.4	0.04
12.3	0.06
15	0.08
18.1	0.010
21	0.012
23.5	0.014
26.2	0.016

Se quiere encontrar una línea óptima con la cual se pueda aproximar la función aunque la aproximación no toque todos los puntos. Tomamos la ecuación de la línea:

$$y_i = ax_i + b.$$

La i representa al i-ésimo valor de la línea de aproximación. La variable a representa la pendiente de la recta y b es la ordenada al origen. Se requiere determinar la mejor

línea de aproximación, cuando el error absoluto, es la suma de los cuadrados de las diferencias entre los valores de y en la aproximación y los valores de y dados, tal que el error absoluto es:

$$\sum_{i}^{n}(y_i^{\text{experimental}} - y_i^{\text{teorico}})^2$$

pero $y_i^{\text{teorico}} = ax_i + b$
Obteniendo:

$$\sum_{i}^{n}[y_i - (ax_i + b)]^2 \tag{6.3}$$

El problema general de ajustar la línea óptima de mínimos cuadrados a un conjunto de datos es que requiere reducir al mínimo la ecuación 6.3 respecto a los parámetros a y b. Tomando la condición necesaria para encontrar un mínimo, encontraremos el mínimo respecto a a:

$$\begin{aligned}
0 &= \frac{d}{da}\sum_{i=1}^{n}[y_i - (ax_i + b)]^2 \\
0 &= 2\sum_{i=1}^{n}[y_i - ax_i - b](-x_i) \\
0 &= 2\sum_{i=1}^{n}(-x_i y_i) + 2\sum_{i=1}^{n}(ax_i^2) + b + 2\sum_{i=1}^{n}(bx_i) \\
\sum_{i=1}^{n} x_i y_i &= a\sum_{i=1}^{n} x_i^2 + b\sum_{i=1}^{n} x_i
\end{aligned} \tag{6.4}$$

Respecto a b:

$$\begin{aligned}
0 &= \frac{d}{db}\sum_{i=1}^{n}[y_i - (ax_i + b)]^2 \\
0 &= 2\sum_{i=1}^{n}[y_i - ax_i - b](-1) \\
0 &= 2\sum_{i=1}^{n}(-y_i) + 2\sum_{i=1}^{n}(ax_i) + 2\sum_{i=1}^{n} b \\
a\sum_{i=1}^{n} x_i + bn &= \sum_{i=1}^{n} y_i
\end{aligned} \tag{6.5}$$

Resolviendo el sistema de ecuaciones que se ha obtenido en 6.4 y 6.5, se tiene:

$$a = \frac{n(\sum_{i=1}^{n} x_i y_i) - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{m(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)^2},$$

$$b = \frac{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i (\sum_{i=1}^{n} x_i y_i)}{n(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)^2}.$$

La mejor aproximación se obtiene con la ecuación $P(x_i) = ax_i - b$, donde a y b son las constantes encontradas aquí. A continuación haremos una función llamada `minimos_cuadrados`, que recibe la lista de los datos x y la lista de y, regresando la pendiente a y la ordenada al origen b que construyen a la mejor recta aproximada de estos datos. El código queda como:

```
from pylab import *
def minimos_cuadrados(x,y):
    n=len(x)
    sumax=0.0
    sumay=0.0
    sumaxy=0.0
    sumaxx=0.0
    for i in range(n):
        sumax+=x[i]
        sumay+=y[i]
        sumaxy+=x[i]*y[i]
        sumaxx+=x[i]*x[i]
    pendiente=(n*sumaxy-sumax*sumay)/(n*sumaxx-sumax*sumax)
    ordenada=(sumaxx*sumay-sumax*sumaxy)/(n*sumaxx-sumax*sumax)
    return pendiente, ordenada
longitud=[0,2,4,6,8,10,12,14,16]
fuerza=[5.3,7,9.4,12.3,15,18.1,21,23.5,26.2]
delta_longitud=0.8 #error en x de la medición
delta_fuerza=0.8 #error en y de la medición
a,b= minimos_cuadrados(longitud,fuerza)
recta = lambda x: a*x[i]+b
lista=[]
for i in range(len(longitud)):
    linea=recta(longitud)
    lista.append(linea)
plot(longitud,fuerza,'o') #datos experimentales
plot(longitud,lista,'orange') #recta encontrada
errorbar(longitud,lista, yerr=delta_fuerza, xerr=delta_longitud,
    label='y=%5.4fx+%5.4f'%(a,b))
grid(True)
title('Aproximación lineal de mínimos cuadrados')
```

```
xlabel('x')
ylabel('y')
legend()
show()
```

Como puede verse, el código queda muy sencillo. Lo interesante de este código consiste en que se está haciendo la visualización de las barras de error. Las barras de error reciben su tamaño en x y en y.

En la gráfica obtenida se puede ver los datos experimentales representados como puntos azules y la recta aproximada con mínimos cuadrados en naranja.

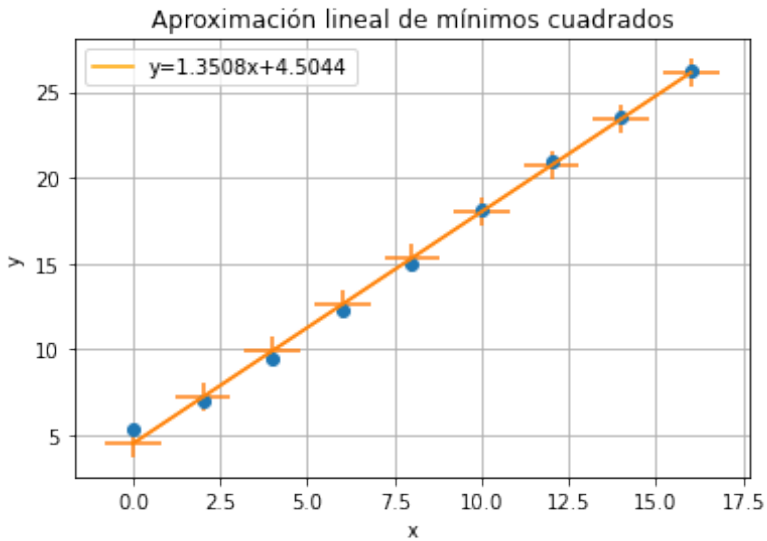


Figura 6.1: Aplicación de la recta de mínimos cuadrados para los datos obtenidos de la ley de Hooke.

A modo de ejemplo, encontremos a mano la mejor recta que aproxima los datos que se encuentran en la siguiente tabla usando la aproximación de mínimos cuadrados. Posteriormente, haremos la comparación con nuestro código. La tabla es:

x_i	y_i	x_i^2	$x_i y_i$
1	1.3	1	1.3
2	3.5	4	7.0
3	4.2	9	12.6
4	5.0	16	20.0
5	7.0	25	35.0
6	8.8	36	52.8
7	10.1	49	70.7
8	12.5	64	100.0
9	13.0	81	117.0
10	15.6	100	156.0
$\sum x_i = 55$	$\sum y_i = 81.0$	$\sum x_i^2 = 385$	$\sum x_i y_i = 572.4$

La pendiente y la ordenada al origen son:

$$a = \frac{10(572.4) - 55(81)}{10(385) - (55)^2} = 1.538$$

$$b = \frac{385(81) - 55(572.4)}{10(385) - (55)^2} = -0.360$$

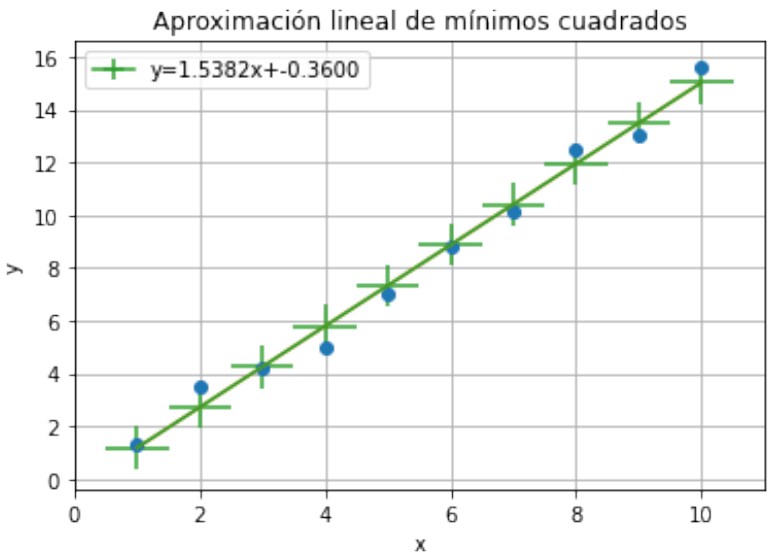


Figura 6.2: Aplicación de la recta de mínimos cuadrados.

Para determinar el error, podemos usar las relaciones de error absoluto, error relativo y error porcentual vistas previamente. En donde los valores reales serán los datos de y obtenidos en el laboratorio y los datos aproximados serán los evaluados por el polinomio encontrado. En este caso, el error absoluto del ejemplo es:

$$E = \sum_{i=1}^{10} (y_i - P(x_i))^2 \approx 2.34.$$

Para generalizar a polinomios de mayor grado consideramos al conjunto finitos de datos de la forma $\{(x_i y_i | i = 1, 2, \cdots, m)\}$. Estos datos serán aproximados por un polinomio algebraico de la forma $P_l(x) = \sum_{k=0}^{l} a_k x^k$ de grado $n < m - 1$ mediante el procedimiento de mínimos cuadrados. Para disminuir al mínimo el error de mínimos cuadrados, es necesario determinar los coeficientes correspondientes $a_0, a_1, \cdots, a_n$, tal que la aproximación del polinomio es:

$$E = \sum_{i=1}^{n} [y_i - P_l(x_i)]^2 = \sum_{i=1}^{n} y_i^2 - 2\sum_{i=1}^{n} P_l(x_i) y_i + \sum_{i=1}^{n} (P_l(x_i))^2$$

Sustituimos $P_l(x) = \sum_{k=0}^{l} a_k x^k$:

$$E = \sum_{i=1}^{n} y_i^2 - 2\sum_{i=1}^{n} \left(\sum_{j=0}^{l} a_j x_i^j \right) y_i + \sum_{j=0} (a_j x_i^j)^2$$

$$E = \sum_{i=1}^{n} y_i^2 - 2\sum_{j=0}^{l} a_j \left(\sum_{i=1}^{n} y_i x_i^j \right) + \sum_{j=0}^{l} \left(\sum_{i=1}^{n} a_j a_k (x_i^j)^{j+k} \right).$$

Para minimizar el polinomio propuesto, se tiene que encontrar las respectivas derivadas:

$$\frac{\partial E}{\partial a_j} = 0.$$

Para todos los valores de j, que van como $j = 0, 1, \cdots, n$. Por lo que, para cada j, se tiene:

$$0 = \frac{\partial E}{\partial a_j} = -2\sum_{y=1}^{n} y_i x_i^j + 2\sum_{k=0}^{l} a_k \sum_{i=1}^{l} x_i^{j+k}$$

Esto nos da $n + 1$ ecuaciones para $n + 1$ incógnitas, donde j va desde 0 hasta n. Conviene escribir las ecuaciones como sigue:

$$a_0\sum_{i=1}^{n}x_i^0+a_1\sum_{i=1}^{n}x_i^1+\cdots+a_n\sum_{i=1}^{n}x_i^n=\sum_{i=1}^{n}y_ix_i^0$$
$$a_0\sum_{i=1}^{n}x_i^1+a_1\sum_{i=1}^{n}x_i^2+\cdots+a_n\sum_{i=1}^{n}x_i^{n+1}=\sum_{i=1}^{n}y_ix_i^1$$
$$a_0\sum_{i=1}^{n}x_i^n+a_1\sum_{i=1}^{n}x_i^{n+1}+\cdots+a_n\sum_{i=1}^{n}x_i^{2n}=\sum_{i=1}^{n}y_ix_i^n$$

De aquí solo queda por resolver algebraicamente el sistema de ecuaciones lineales. Para entender la metodología, veamos un ejemplo para un polinomio cuadrática. La tabla de datos a la que se le aproximará el polinomio de grado dos es:

i	x_i	y_i
1	0.0	1.0
2	0.25	1.2840
3	0.50	1.6487
4	0.75	2.1170
5	1.00	2.7183

Las ecuaciones son:

$$5a_0+2.5a_1+1.875a_2=8.7680$$
$$2.5a_0+1.875a_1+1.5625a_2=5.4514$$
$$1.875a_0+1.562a_1+1.3828a_2=4.4015$$

Para resolver el sistema de ecuaciones lineales usamos el código del capítulo 4. A continuación se presenta el respectivo código con una simple modificación:

```
from numpy import array, zeros
X=array([[5,2.5,1.87],[2.5,1.87,1.56],[1.87,1.56,1.38]],float)
c=array([8.7680,5.4514,4.4015],float)
n=len(c)

for i in range(n):
    for j in range(i+1,n):
        multi=(X[j,i]/X[i,i])
        X[j]=X[j]-multi*X[i]
        c[j]=c[j]-multi*c[i]
for i in range(n):
    div=X[i,i]
```

```
    X[i]=X[i]/div
    c[i]=c[i]/div
a=zeros(n,float)
for i in range(n-1,-1,-1): #ciclo hacia atras
    a[i]=c[i]
    for k in range(i+1,n):
        a[i]-=X[i,k]*a[k]
print('Los coeficientes de la aproximación cuadratica son')
print('a0 = ',a[0])
print('a1 = ',a[1])
print('a2 = ',a[2])

x=[0,0.25,0.50,0.75,1]
delta_x=0.125
y=[1.0,1.2,1.65,2.12,2.71]
delta_y=0.125
#Visualizacion
def cuadratica(a,x):
    valor=a[2]*x[i]**2+a[1]*x[i]+a[0]
    return valor

lista1=[]
for i in range(len(x)):
    approx=cuadratica(a,x)
    lista1.append(approx)

plot(x,y,'o') #datos experimentales
plot(x,lista1) #cuadratica encontrada con la aproximación
errorbar(x,lista1, yerr=delta_y, xerr=delta_x,label='%3.2fx
    **2+%3.2fx+%3.2f=0'%(a[2],a[1],a[0]))
grid(True)
title('Aproximación cuadrática de mínimos cuadrados')
xlabel('x')
ylabel('y')
legend()
show()
```

```
Los coeficientes de la aproximación cuadratica son
a0 =  1.0050755189757687
a1 =  0.8646758481938552
a2 =  0.8431641518061435
```

Con la visualización dada por la siguiente gráfica:

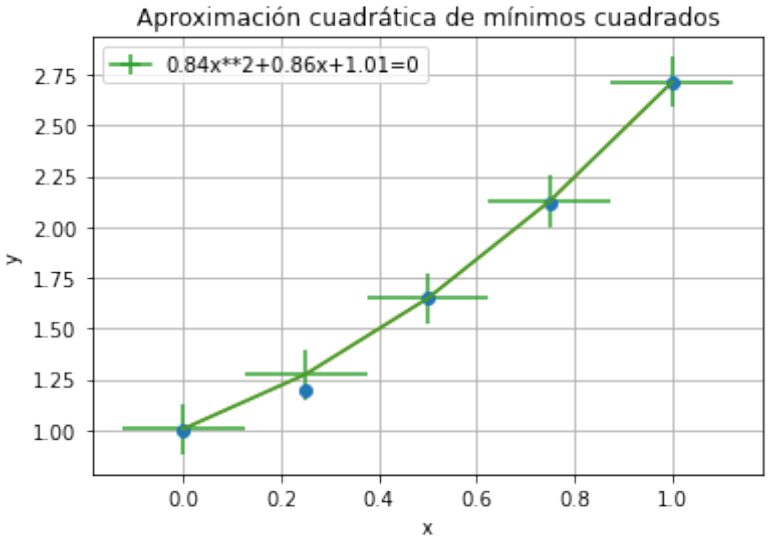


Figura 6.3: Aplicación de la recta de mínimos cuadrados.

Del código, se puede resolver cualquier aproximación a mínimos cuadrados que se quiera. La dificultad se encontrará en plantear el polinomio y de ahí se introduce los valores de x y y tal que se obtenga la aproximación correspondiente.

6.4. Ejercicios

1. Considera los números:

$$x = 1$$
$$y = 1 + 10^{-14}\sqrt{2}$$

Se puede ver que:

$$10^{14}(y - x) = \sqrt{2} \tag{6.6}$$

Escribe un programa en Python que determine la ecuación (6.6). También incluye una parte que obtenga el valor absoluto, relativo y porcentual de esta aproximación a $\sqrt{2}$.

2. **Errores de redondeo en la solución de la ecuación cuadrática**

   Sea la ecuación cuadrática $ax^2 + bx + c = 0$ con soluciones real.
   i) Resuelve la ecuación cuadrática con el programa que hicimos en el capítulo 3 usando la ecuación $0.001x^2 + 1000x + 0.001$.
   ii) Reescribe el programa pero ahora con la solución:

$$x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}.$$

   Esta expresión se obtiene si a la solución anterior se le multiplica arriba y abajo por $-b \pm \sqrt{b^2 - 4ac}$. Vuelve a probar con la ecuación $0.001x^2 + 1000x + 0.001$. Explica que esta pasando con los errores de precisión.

3. Obtén el polinomio de primer y segundo grado para los datos que se presentan en la siguiente tabla:

x_i	y_i
1.0	1.84
1.1	1.96
1.3	2.21
1.5	2.45
1.9	2.94
2.1	3.18

   En cada caso calcula el error y grafica los datos con la información de los polinomios encontrados.

4. Determina la aproximación por mínimos cuadrados de la forma $y = be^{ax}$, considera que los datos obtenidos en laboratorio son $(x_i, \ln y_i)$. Prueba tu aproximación con los datos de la siguiente tabla:

i	x_i	y_i	$\ln y_i$	x_i^2	$x_i \ln y_i$
1	1.0	5.10	1.629	1.0	1.629
2	1.25	5.79	1.759	1.5625	2.195
3	1.5	6.53	1.876	2.2500	2.814
4	1.75	7.45	2.008	3.0625	3.514
5	2.0	8.46	2.135	4.00	4.270

Apéndice A

Bash

En la arquitectura del sistema operativo Linux, el shell es el programa que comunica a las aplicaciones con el kernel, como un traductor. Los comandos básicos de Linux que hemos visto son comandos shell. Existen distintos programas shell y uno puede instalar el que más le parezca pertinente, `Bash, Ash, Csh, Zsh, Ksh, Tcsh`. El más popular es Bash y está instalado de default en todos los sistemas Linux.

El lenguaje de comandos Bash significa **B**ourne-**A**gain **Sh**ell, fue escrito por Briana Fox bajo la filosofía free-software y fue un reemplazo a otro shell que había en esa época (1989) llamado *shell bourne*. Actualmente, se encuentra en la mayoría de las distribuciones Linux, Solaris y MacOs. También está disponible para Windows y Android. A pesar de que Bash data de finales de los años 80 es muy popular en la actualidad, tan es así, que el 24 de septiembre de 2014, se divulgo un bug o error importante de seguridad llamado Shellshock o Bashdoor. Varios servicios de internet y servidores usan Bash para ciertos procesos, por lo que el peligro consistía en que el atacante podía ejecutar comandos arbitrarios en versiones vulnerables y ganaba acceso al sistema atacado; así podían entrar, como por una puerta trasera.

Las ventajas de usar script Bash sobre hacer los procesos a mano o con el mousse son ahorrar mucho tiempo y eliminar tareas repetitivas.

Para empezar con Bash, es necesario que abras el editor de texto que prefieras, escribas los comandos o programa y lo guardes con la terminación .sh. Hagamos nuestro primer programa en Bash, llamado HolaMundo.sh; que se verá así:

```
#!/bin/bash
# Script de hola mundo
echo "hola mundo"
```

Para ejecutar el programa, en la línea de comandos tecleamos `bash HolaMundo.sh`. La primera línea es muy importante, ya que indica que es un script Bash, también indica

la ruta absoluta donde se encuentran los archivos binarios para la ejecución. Cuando escribimos código o programamos, en ocasiones, es importante colocar comentarios en el programa para que futuros programadores entiendan que estamos haciendo, i.e., ayuda a organizar el código. Los comentarios en Bash van precedidos por #. El siguiente comando es `echo`, que funciona como un comando de salida o un `print` en Python.

En Bash, las asignaciones de variables se llevan a cabo con un símbolo de igual. Las variables son "Case Sensitive", es decir, las variables son susceptibles al uso de minúsculas y mayúsculas. También es importante resaltar que las variables en Bash van siendo declaradas, por ejemplo, si son enteras o flotantes, conforme se van asignando. En Bash, para invocar a una variable, usamos el símbolo $:

```
#!/bin/bash
#Asignando variables
a="hola"
#Llamando a la variable $1a
$a
#Mostrando el contenido de la variable
echo $a
```

Hasta el momento, todo parece un poco intrascendente. Sin embargo, combinando las estructuras de control con los comandos básicos de Linux, o mejor dicho, de Bash, podemos tener un programa de manejo de datos y archivos muy poderoso.

La estructura condicional `if` tendrá a la condición dentro de [ ] y siempre irá acompañada con `then` para la sentencia. Todo lo que abrimos en Bash irá cerrado, por lo que tendremos que poner un `fi` para cerrar.

```
#!/bin/bash
numero=99
if [ $numero -eq 100 ]
then
  echo "numero es 100"
else
  if [ $numero -gt 100 ]
  then
    echo "numero es mayor a 100"
  else
  echo "numero es menor a 100"
  fi
fi
```

Los operadores de comparación serán asignados de la siguiente forma;

- Igual que `-eq`.
- Menor que `-lt`.
- Menor o igual que `-le`.
- Mayor que: `-gt`.
- Mayor o igual que `-ge`.

Otra estructura importante para Bash, que no existe en Python, es la estructura `Case-Esac`, que sirve para hacer menús de forma fácil. En esta estructura, con un comando `read`, pediremos que nos de la opción que deseé el usuario:

```
read opcion
case $opcion in
    s|S)
       echo "ha escogido la opción SI"
    ;;
    n|N)
       echo "ha escogido la opción NO"
    ;;
    *)
       echo "opción no válida"
    ;;
esac
```

La siguiente estructura de control en Bash consiste en el bucle `for-do-done`, el cual va acompañado con un `do` y terminado con un `done`. Veamos un ejemplo que crea cien archivos llamados `Arch0.txt`, `Arch1.txt`, `Arch2.txt`, ..., `Arch100.txt`, y adentro de estos archivos va un hola.

```
for i in {0..100}
do
    echo hola > "Arch${i}.txt"
done
```

Otra de las versatilidades de Bash es que no es forzoso poner todo en archivos y luego ejecutar, sino que podemos poner en la terminal la estructura de control y ejecutar con un simple `Enter`:

```
roxana:~$ for i in {1..100}; do echo hola > "Arch${i}.txt"; done
```

Nota que las líneas son separadas con punto y coma.

La última estructura que veremos es el bucle `while-do-done`, el cual se repite secuencialmente mientras se cumpla una condición lógica. Esta estructura puede ir acompañada con un `break`, de tal forma que, si se llega a una determinada condición, se rompa abruptamente el bucle:

```
i=0
while [ $i -lt 15 ]
do
  echo "Numero: $i"
  ((i++))
  if [[ "$i" == '12' ]]; then
    break
  fi
done
```

Aquí decimos mientras $i < 15$, imprime el número. Pero no recorremos los valores del 0 al 15, sino que tronamos el bucle antes, en 12.

Como se puede ver, en el fondo, las estructuras de control de Bash son las mismas que las de Python, por lo que, si uno aprende la sintaxis, es suficiente para empezar en Bash. Se recomienda al lector profundizar más en este lenguaje, ya que es sumamente popular y útil en el cómputo de alto rendimiento y en la computación científica.

Apéndice B

Introducción al manejo de datos numéricos y experimentales: Códigos completos

B.1. Código de errores numéricos indirectos

```
from numpy import *
def lectura_datos(f):
    z=f.read() # hacer la acción de leer el archivo que se abrio
    en f
    lista = z.split('\n') #hago una lista, proveniente de una
    cadena de caracteres z. A esta lista le quito el cambio de lí
    nea (\n).
    n=len(lista)
    l1,l2,l3,l4=[],[],[],[]
    for i in range(n):
        x=lista[i].split('\t') #separo los elementos de una
    cadena de caracteres en una lista
        a,b,c,d=float(x[0]),float(x[1]),float(x[2]),float(x[3]) #
    cada elemento de la lista x lo convierto en real
        l1.append(a)
        l2.append(b)
        l3.append(c)
        l4.append(d)
    return l1,l2,l3,l4

def incertidumbres_indirectas(x,deltax,y,deltay,operacion):
```

```
    w1,w2=[],[] #w1 y w2 guardaran los datos encontrados de las
    mediciones indirectas, z y deltaz
    if operacion=='suma':
        for i in range(len(x)):
            z=x[i]+y[i]
            deltaz=deltax[i]+deltay[i]
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    elif operacion=='resta':
        for i in range(len(x)):
            z=x[i]-y[i]
            deltaz=deltax[i]+deltay[i]
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    elif operacion=='multiplicacion':
        for i in range(len(x)):
            z=x[i]*y[i]
            deltaz=x[i]*deltay[i]+y[i]*deltax[i]
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    elif operacion=='division':
        for i in range(len(x)):
            z=x[i]/y[i]
            deltaz=(x[i]*deltay[i]+y[i]*deltax[i])/y[i]**2.0
            w1.append(z)
            w2.append(deltaz)
        return w1,w2
    else:
        print('esa no es una operacion definida en el programa,
    por lo que z y delta z, seran ceros')
        w1,w2=zeros(len(x)),zeros(len(x))
        return w1,w2

def escritura_datos(x,deltax,y,deltay,z,deltaz,
    variable_apertura_archivo_escritura):
    lista1=[]
    for i in range(len(x)): #convierto cada entrada de las listas
     en cadenas
        k1,k2,k3,k4,k5,k6=str(x[i]),str(deltax[i]),str(y[i]),str(
    deltay[i]),str(z[i]),str(deltaz[i])
        #pego todas las listas, cuyos elementos son cadenas de
    caracteres
```

```
        lista1.append(k1+'\t'+k2+'\t'+k3+'\t'+k4+'\t'+k5+'\t'+k6)
    o='\n'.join(lista1)+'\t'
    return variable_apertura_archivo_escritura.write('x\tdeltax\
    ty\tdeltay\tz\tdeltaz\n'+o)


operacion=input('que operacion haras, suma, resta, multiplicacion
     o division:\t')
variable_apertura_archivo_lectura=open('a.dat','r', errors="
    ignore") #le digo que abrirá un archivo y que su status sólo
    es de lectura.
x,deltax,y,deltay=lectura_datos(variable_apertura_archivo_lectura
    ) #use la funcion lectura_datos para el archivo que abri
z,deltaz=incertidumbres_indirectas(x,deltax,y,deltay,operacion) #
    calcule z y deltaz con la funcion llamada
    incertidumbres_indirectas
variable_apertura_archivo_escritura=open('datos_1.dat','w') #
    crear nuevo archivo llamado datos_1.dat
escritura_datos(x,deltax,y,deltay,z,deltaz,
    variable_apertura_archivo_escritura) #hacer la acción de
    escribir en el archivo indicado.

#cierro todos los archivos
variable_apertura_archivo_escritura.close()
variable_apertura_archivo_lectura.close()
```

Bibliografía

[1] D.M. Ritchie, K. Thompson. The UNIX Time-Sharing System. Communications of the ACM. Vol. 17, número 7, 1974.

[2] D.M. Ritchie, K. Thompson. The UNIX Time-Sharing System- A Retrospective. Tenth Hawaii International Conference on the System Sciences, Honolulu, January, 1977

[3] H. Gust, K.-U. Kühnberger y U. Schmid. Capítulo 10: Ontologies as a cue for the metaphorical meaning of technical concepts. (pp. 204-225) Libro: Mental States. Volumen I: Evolution, function, nature. Editado por A.C Schalley y D. Khlentzos. Editorial John Benjamins Publishing Company. Amsterdam/Philadelphia, 2007

[4] George Gheverghese Joseph, The Crest of the Peacock: Non-European Roots of Mathematics – 2nd. ed. London: Penguin Books, 2000

[5] Swapan Kumar Adhikari. Babylonian Mathematics. Indian Journal of History of Science, 33(1), 1998

[6] Jacques Sesiano. Magic Squares: Their History and Construction from Ancient Times to AD 1600. – 1st. ed Springer International Publishing, 2019

[7] Marcel Henrique Rodrigues. Albrech Dürer and the 16th century melancholy. The International Visual Culture Review, 2, 2020.

[8] Jack Chernick. Solution of the General Magic Square. The American Mathematical Monthly , Mar., 1938, Vol. 45, No. 3 (Mar., 1938), pp. 172-175

[9] G. Dahlquist, A. Björck. Numerical Methods in Scientific Computing. Society for Industrial and Applied Mathematics SIAM, primera edición, Philadelphia (2008).

[10] S. Nakamura. Métodos Numéricos Aplicados con Software. Pearson Educación, primera edición. México (1992).

[11] P.D Kazarinoff. (2018-2020) Quiver plots using Python, matplotlib and Jupyter notebook. Recuperado de url=https://pythonforundergradengineers.com/quiver-plot-with-matplotlib-and-jupyter-notebooks.html, journal=Python for Undergraduate Engineers

[12] https://www.interactivechaos.com/manual/tutorial-de-matplotlib/mapas-de-color

[13] Berta Oda Noda. Introducción al análisis gráfico de datos experimentales. Editorial: Prensas de Ciencias Año 2005, 212 páginas

[14] Richard L. Burden, J. Douglas Faires. Análisis Numérico, Sexta Edición. Editorial: Matemáticas International Thomson. Año 1998.

Temas selectos de computación
se terminó de editar el 23 de marzo de 2021
en la Coordinación de Servicios Editoriales
de la Facultad de Ciencias de la UNAM.

El cuidado de la edición estuvo a cargo de
Mercedes Perelló Valls

En este libro el lector encontrará una manera fresca y rápida de aprender Python, así como algo de historia del cómputo científico, sin desviarse del objetivo principal. Sus referencias históricas son oportunas y contribuyen a la cultura del lector, ya sea un estudiante de ingeniería, física o matemáticas. El lenguaje del libro hace que el lector se sienta muy cómodo y tenga deseos de continuar el aprendizaje.

Como libro de consulta también es muy útil pues se describen las librerías de Python de una manera muy práctica, lo que lo coloca muy lejos de ser un libro rígido y acartonado. La explicación de los temas y material que forma el Python siempre se presenta con ejemplos y programas listos para su ejecución. Cada capítulo culmina con ejercicios de problemas de matemáticas y física muy amenos, que refuerzan la cultura que la autora quiere transmitir al lector.

La Dra. Roxana del Castillo es Profesora de Tiempo Completo de la Facultad de Ciencias de la UNAM. Realizó estudios postdoctorales en el área de conversión de gases de efecto invernadero a productos energéticos de alto valor industrial. Sus campos de especialidad son: nanociencia, materiales de baja dimensionalidad, física computacional y fisicoquímica de superficies. Su obra científica consta de 17 artículos en revistas internacionales indizadas y un capítulo de libro. Tiene experiencia docente de más de 13 años impartiendo asignaturas de Física computacional y Computación en la carrera de Física.

Imagen de la portada:
Detalle de ***Abstracción*** (Óleo y oleorresina sobre cartón)
Willem de Kooning,1949 - 1950
Museo Nacional Thyssen-Bornemisza, Madrid

ISBN: 978-607-30-4462-2