Departamento de Física, Facultad de Ciencias, Universidad de Chile.
Las Palmeras 3425, Ñuñoa. Casilla 653, Correo 1, Santiago
FONO: 562 978 7276 FAX: 562 271 2973
E-MAIL: secretaria@fisica.ciencias.uchile.cl

Apuntes de un curso de

PROGRAMACIÓN Y MÉTODOS NUMÉRICOS

Sexta edición, revisión 061102-10

José Rogan C.
Víctor Muñoz G.

Agradecientos:

Denisse Pastén Daniel Asenjo y Max Ramírez. De la promoción del 2006 a: Paulina Chacón, Sergio Valdivia y Elizabeth Villanueva

Índice

Índice de figuras

Parte I

Computación

Capítulo 1

Elementos del sistema operativo UNIX.

versión final revisada 6.4, 10 de Octubre del 2006

1.1. Introducción.

En este capítulo se intentará dar los elementos básicos para poder trabajar en un ambiente UNIX. Sin pretender cubrir todos los aspectos del mismo, nuestro interés se centra en dar las herramientas al lector para que pueda realizar los trabajos del curso bajo este sistema operativo. Como comentario adicional, conscientemente se ha evitado la traducción de gran parte de la terminología técnica teniendo en mente que documentación disponible se encuentre, por lo general, en inglés y nos interesa que el lector sea capaz de reconocer los términos.

El sistema operativo UNIX es el más usado en investigación científica, tiene una larga historia y muchas de sus ideas y métodos se encuentran presentes en otros sistemas operativos. Algunas de las características relevantes del UNIX moderno son:

- Memoria grande, lineal y virtual: Un programa en una máquina de 32 Bits puede acceder y usar direcciones hasta los 4 GB en una máquina de sólo 4 MB de RAM. El sistema sólo asigna memoria auténtica cuando le hace falta, en caso de falta de memoria de RAM, se utiliza el disco duro (*swap*).
- Multitarea (*Multitasking*): Cada programa tiene asignado su propio "espacio" de memoria. Es **imposible** que un programa afecte a otro sin usar los servicios del sistema operativo. Si dos programas escriben en la misma dirección de memoria cada uno mantiene su propia "idea" de su contenido.
- Multiusuario: Más de una persona puede usar la máquina al mismo tiempo. Programas de otros usuarios continúan ejecutándose a pesar de que un nuevo usuario entre a la máquina.
- Casi todo tipo de dispositivo puede ser accedido como un archivo.
- Existen muchas aplicaciones diseñadas para trabajar desde la línea de comandos. Además, la mayoría de las aplicaciones permiten que la salida de una pueda ser la entrada de la otra.
- Permite compartir dispositivos (como disco duro) entre una red de máquinas.

Por su naturaleza de multiusuario, **nunca** se debería apagar impulsivamente una máquina UNIX[1], ya que una máquina apagada sin razón puede matar trabajos de días, perder los últimos cambios de tus archivos e ir degradando el sistema de archivos en dispositivos como el disco duro.

Entre los sistemas operativos UNIX actuales cabe destacar:

- Linux fue originalmente desarrollado primero para computadores personales PCs basados en x86 de 32 bits (386 o superiores). Hoy Linux también corre sobre (al menos en) el Compaq Alpha AXP, Sun SPARC y UltraSPARC, Motorola 68000; en particular, para las estaciones Sun3, computadores personales Apple Macintosh, Atari y Amiga; PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64, AXIS CRIS y arquitecturas Renesas M32R.
- SunOS[2]: disponible para la familia 68K así como para la familia SPARC de estaciones de trabajo SUN
- Solaris[3] : disponible para la familia SPARC de SUN así como para la familia x86.
- OSF1[4]: disponible para Alpha.
- Ultrix: disponible para VAX de Digital
- SYSVR4[5]: disponible para la familia x86, vax.
- IRIX: disponible para MIPS.
- AIX[6]: disponible para RS6000 de IBM y PowerPC.

1.2. Ingresando al sistema.

En esta sección comentaremos las operaciones de comienzo y fin de una sesión en UNIX así como la modificación de la contraseña (que a menudo no es la deseada por el usuario, y que por lo tanto puede olvidar con facilidad).

1.2.1. Terminales.

Para iniciar una sesión es necesario poder acceder a un terminal. Pueden destacarse dos tipos de terminales:

- Terminal de texto: consta de una pantalla y de un teclado. Como indica su nombre, en la pantalla sólo es posible imprimir caracteres de texto.

[1] Incluyendo el caso en que la máquina es un PC normal corriendo Linux u otra versión de UNIX.
[2] SunOS 4.1.x también se conoce como Solaris 1.
[3] También conocido como SunOS 5.x, solaris 2 o Slowaris :-).
[4] También conocido como Dec Unix.
[5] También conocido como Unixware y Novell-Unix.
[6] También conocido como Aches.

- Terminal gráfico: Consta de pantalla gráfica, teclado y *mouse*. Dicha pantalla suele ser de alta resolución. En este modo se pueden emplear ventanas que emulan el comportamiento de un terminal de texto (`xterm` o `gnome-terminal`).

1.2.2. Login.

El primer paso es encontrar un terminal libre donde aparezca el *login prompt* del sistema:

```
Debian GNU/Linux 3.1 hostname tty2

hostname login:
```

También pueden ocurrir un par de cosas:

- La pantalla no muestra nada.
  - Comprobar que la pantalla esté encendida.
  - Pulsar alguna tecla o mover el *mouse* para desactivar el protector de pantalla.
- Otra persona ha dejado una sesión abierta. En este caso existe la posibilidad de intentar en otra máquina o bien finalizar la sesión de dicha persona (si ésta no se halla en las proximidades).

  Una vez que se haya superado el paso anterior de encontrar el *login prompt* se procede con la introducción del *Username* al *prompt* de *login* y después la contraseña (*password*) adecuada.

1.2.3. Passwords.

El *password* puede ser cualquier secuencia de caracteres a elección. Deben seguirse las siguientes pautas:

- Debe ser fácil de recordar por uno mismo. Si se olvida, deberá pasarse un mal rato diciéndole al administrador de sistema que uno lo ha olvidado.
- Para evitar que alguna persona no deseada obtenga el *password* y tenga libre acceso a los archivos de tu cuenta:
  - Las mayúsculas y minúsculas no son equivalentes, sin embargo se recomienda que se cambie de una a otra.
  - Los caracteres numéricos y no alfabéticos también ayudan. Debe tenerse sin embargo la precaución de usar caracteres alfanuméricos que se puedan encontrar en todos los terminales desde los que se pretenda acceder.
  - Las palabras de diccionario deben ser evitadas.

- Debe cambiarlo si cree que su *password* es conocido por otras personas, o descubre que algún intruso[7] está usando su cuenta.
- El *password* debe ser cambiado con regularidad.

La orden para cambiar el password en UNIX es `passwd`. A menudo cuando existen varias máquinas que comparten recursos (disco duro, impresora, correo electrónico, ...), para facilitar la administración de dicho sistema se unifican los recursos de red (entre los que se hayan los usuarios de dicho sistema) en una base de datos común. Dicho sistema se conoce como NIS (*Network Information Service*)[8]. Si el sistema empleado dispone de este servicio, la modificación de la contraseña en una máquina supone la modificación en todas las máquinas que constituyan el dominio NIS.

1.2.4. Cerrando la sesión.

Es importante que nunca se deje abierta una sesión, pues algún "intruso" podría tener libre acceso a archivos de tu propiedad y manipularlos de forma indeseable para ti. Para evitar todo esto basta teclear `logout` o `exit` y habrá acabado tu sesión de UNIX en dicha máquina[9].

1.3. El Proyecto Debian.

El proyecto Debian es una asociación de personas que han creado un sistema operativo gratis y de código abierto (*free*). Este sistema operativo se denomina Debian GNU/Linux o simplemente Debian.

Actualmente Debian ocupa el kernel Linux desarrollado por Linus Torvalds apoyado por miles de programadores de todo el mundo. Hay planes para implementar otros kernels como Hurd desarrollado por la GNU.

La mayoría de las herramientas del sistema operativo Debian provienen de la GNU y por ende son *free*. Cabe destacar que actualmente Debian tiene un total de más de 15490 paquetes (por paquetes entendemos software precompilado en un formato que permite su fácil instalación en nuestra máquina). Entre estos paquetes encontramos desde las herramientas básicas para procesar texto, hojas de cálculo, edición de imágenes, audio, video, hasta aplicaciones de gran utilidad científica. Es importante recordar que todo este *software* es *free* y por lo tanto está al alcance de todos sin la necesidad de comprar licencias y pagar por actualizaciones. También existe la posibilidad de modificar el *software* ya que tenemos acceso al código fuente de los programas.

Debian siempre mantiene activas al menos tres versiones que tienen las siguientes clasificaciones

- *stable* (estable): Es la ultima versión oficial de Debian que ha sido probada para asegurar su estabilidad. Actualmente corresponde a la versión 3.1r0 llamada *sarge*.

[7] Intruso es cualquier persona que no sea el usuario.

[8] Antiguamente se conocía como YP (*Yellow Pages*), pero debido a un problema de marca registrada de *United Kingdom of British Telecomunications* se adoptaron las siglas NIS.

[9] En caso que se estuviera trabajando bajo X-Windows debes cerrar la sesión con `Log out of Gnome`.

- *testing* (en prueba): Esta es la versión que se está probando para asegurar su estabilidad y luego pasarla a la versión estable. Se llama *etch*.
- *unstable* (inestable): Aquí es donde los programadores verdaderamente desarrollan Debian y por esto no es muy estable y no se recomienda para el uso diario. Esta versión se denomina *sid*.

Para información sobre Debian y cómo bajarlo visite la página oficial `http://www.debian.org`.

1.4. Archivos y directorios.

Aunque haya diferentes distribuciones y cada una traiga sus programas, la estructura básica de directorios y archivos es más o menos la misma en todas:

```
/-|--> bin
  |--> boot
  |--> cdrom
  |--> dev
  |--> etc
  |--> floppy
  |--> home
  |--> initrd
  |--> lib
  |--> media
  |--> mnt
  |--> opt
  |--> proc
  |--> root
  |--> sbin
  |--> sys
  |--> tmp
  |--> usr--|--> bin
            |--> doc
            |--> games
            |--> include
            |--> lib
            |--> local -|--> bin
                        |--> lib
            |--> sbin
            |--> share
            |--> src --> linux
            |--> X11R6
  |--> var--|--> lock
            |--> log
            |--> mail
            |--> www
```

El árbol que observamos muestra un típico árbol de directorios en Linux. Pueden variar, sin embargo, algunos de los nombres de estos directorios dependiendo de la distribución o versión de Linux que se esté usando. Algunos directorios destacados son:

- `/home` - Espacio reservado para las cuentas de los usuarios.
- `/bin`, `/usr/bin` - Binarios (ejecutables) básicos de UNIX.
- `/etc`, aquí se encuentran los archivos de configuración de todo el software de la máquina.
- `/proc`, es un sistema de archivo virtual. Contiene archivos que residen en memoria pero no en el disco duro. Hace referencia a los programas que se están corriendo en el momento en el sistema.
- `/dev` (*device*) (dispositivo). Aquí se guardan los controladores de dispositivos. Se usan para acceder a los dispositivos físicos del sistema y recursos como discos duros, *modems*, memoria, *mouse*, etc. Algunos dispositivos:
  - `hd`: `hda1` será el disco duro IDE, primario (`a`), y la primera partición (`1`).
  - `fd`: los archivos que empiecen con las letras `fd` se referirán a los controladores de las disketteras: `fd0` sería la primera diskettera, `fd1` sería la segunda y así sucesivamente.
  - `ttyS`: se usan para acceder a los puertos seriales como por ejemplo `ttyS0`, que es el puerto conocido como `com1`.
  - `sd`: son los dispositivos SCSI. Su uso es muy similar al del `hd`.
  - `lp`: son los puertos paralelos. `lp0` es el puerto conocido como `LPT1`.
  - `null`: éste es usado como un agujero negro, ya que todo lo que se dirige allí desaparece.
  - `tty`: hacen referencia a cada una de las consolas virtuales. Como es de suponer, `tty1` será la primera consola virtual, `tty2` la segunda, etc.
- `/usr/local` - Zona con las aplicaciones no comunes a todos los sistemas UNIX, pero no por ello menos utilizadas. En `/usr/share/doc` se puede encontrar información relacionada con dicha aplicación (en forma de páginas de manual, texto, html o bien archivos dvi, Postscript o pdf). También encontramos archivos de ejemplo, tutoriales, *HOWTO*, etc.

1.5. Órdenes básicas.

Para ejecutar un comando, basta con teclear su nombre (también debes tener permiso para hacerlo). Las opciones o modificadores empiezan normalmente con el caracter `-` (p. ej. `ls -l`). Para especificar más de una opción, se pueden agrupar en una sola cadena de caracteres (`ls -l -h` es equivalente a `ls -lh`). Algunos comandos aceptan también opciones dadas por palabras completas, en cuyo caso usualmente comienzan con `--` (`ls --color=auto`).

1.5.1. Archivos y directorios.

En un sistema computacional la información se encuentra en archivos que la contienen (tabla de datos, texto ASCII, fuente en lenguaje C, Fortran o C++, ejecutable, imagen, mp3, figura, resultados de simulación, ...). Para organizar toda la información se dispone de una entidad denominada directorio, que permite el almacenamiento en su interior tanto de archivos como de otros directorios[10].

Se dice que la estructura de directorios en UNIX es jerárquica o arborescente, debido a que todos los directorios nacen en un mismo punto (denominado directorio raíz). De hecho, la zona donde uno trabaja es un nodo de esa estructura de directorios, pudiendo uno a su vez generar una estructura por debajo de ese punto. Un archivo se encuentra situado siempre en un directorio y su acceso se realiza empleando el camino que conduce a él en el **Árbol de Directorios del Sistema**. Este camino es conocido como el *path*. El acceso a un archivo se puede realizar empleando:

- Path Absoluto, aquél que empieza con `/`

  Por ejemplo : `/etc/printcap`

- Path Relativo, aquél que NO empieza con `/`

  Por ejemplo : `../examples/rc.dir.01`

- Nombres de archivos y directorios pueden usar un máximo de 255 caracteres, cualquier combinación de letras y símbolos (el caracter `/` no se permite).

Los caracteres comodín pueden ser empleados para acceder a un conjunto de archivos con características comunes. El signo `*` puede sustituir cualquier conjunto de caracteres[11] y el signo `?` a cualquier caracter individual. Por ejemplo:[12]

```
bash$ ls
f2c.1          flexdoc.1    rcmd.1    rptp.1   zforce.1
face.update.1  ftptool.1    rlab.1    rxvt.1   zip.1
faces.1        funzip.1     robot.1   zcat.1   zipinfo.1
flea.1         fvwm.1       rplay.1   zcmp.1   zmore.1
flex.1         rasttoppm.1  rplayd.1  zdiff.1  znew.1
bash$ ls rp*
rplay.1        rplayd.1     rptp.1
bash$ ls *e??
face.update.1  zforce.1     zmore.1
```

Los archivos cuyo nombre comiencen por `.` se denominan **ocultos**, así por ejemplo en el directorio de partida de un usuario.

```
bash$ ls -a  user
.                .alias       .fvwmrc  .login    .xinitrc
..               .cshrc       .joverc  .profile
.Xdefaults       .enviroment  .kshrc   .tcshrc
```

[10] Normalmente se acude a la imagen de una carpeta que puede contener informes, documentos o bien otras carpetas, y así sucesivamente.

[11] Incluido el punto '.', UNIX no es DOS.

[12] `bash$` es el *prompt* en todos los ejemplos.

Algunos caracteres especiales para el acceso a archivos son:

`.`	Directorio actual
`..`	Directorio superior en el árbol
`~`	Directorio `$HOME`
`~user`	Directorio `$HOME` del usuario user

1.5.2. Órdenes relacionadas con directorios.

`ls` (LiSt)
Este comando permite listar los archivos de un determinado directorio. Si no se le suministra argumento, lista los archivos y directorios en el directorio actual. Si se añade el nombre de un directorio el listado es del directorio suministrado. Existen varias opciones que modifican su funcionamiento entre las que destacan:

- `-l` (Long listing) proporciona un listado extenso, que consta de los permisos[13] de cada archivo, el usuario, el tamaño del archivo, . . . , etc. Adicionalmente la opción `-h` imprime los tamaños en un formato fácil de leer (Human readable).
- `-a` (list All) lista también los archivos ocultos.
- `-R` (Recursive) lista recursivamente el contenido de todos los directorios que se encuentre.
- `-t` ordena los archivos por tiempo de modificación.
- `-S` ordena los archivos por tamaño.
- `-r` invierte el sentido de un orden.
- `-p` agrega un caracter al final de cada nombre de archivo, indicando el tipo de archivo (por ejemplo, los directorios son identificados con un `/` al final).

`pwd` (Print Working Directory)
Este comando proporciona el nombre del directorio actual.
`cd` (Change Directory)
Permite moverse a través de la estructura de directorios. Si no se le proporciona argumento se provoca un salto al directorio `$HOME`. El argumento puede ser un nombre absoluto o relativo de un directorio. `cd -` vuelve al último directorio visitado.
`mkdir` (MaKe DIRectory)
Crea un directorio con el nombre (absoluto o relativo) proporcionado.
`rmdir` (ReMove DIRectory)
Elimina un directorio con el nombre (absoluto o relativo) suministrado. Dicho directorio debe de estar vacío.

[13] Se comentará posteriormente este concepto.

1.5.3. Visitando archivos.

Este conjunto de órdenes permite visualizar el contenido de un archivo sin modificar su contenido.

`cat`
Muestra por pantalla el contenido de un archivo que se suministra como argumento.

`more`
Este comando es análogo al anterior, pero permite la paginación.

`less`
Es una versión mejorada del anterior. Permite moverse en ambas direcciones. Otra ventaja es que no lee el archivo entero antes de arrancar.

1.5.4. Copiando, moviendo y borrando archivos.

`cp` (CoPy)
Copia un archivo(s) con otro nombre y/o a otro directorio, por ejemplo, el comando para copiar el `archivo1.txt` con el nombre `archivo2.txt` es:

```
cp archivo1.txt archivo2.txt
```

Veamos algunas opciones:

- `-a` copia en forma recursiva, no sigue los *link* simbólicos y preserva los atributos de lo copiado.
- `-i` (interactive), impide que la copia provoque una pérdida del archivo destino si éste existe[14].
- `-R` (recursive), copia un directorio y toda la estructura que cuelga de él.

`mv` (MoVe)
Mueve un archivo(s) a otro nombre y/o a otro directorio, por ejemplo, el comando para mover el `archivo1.txt` al nombre `archivo2.txt` es:

```
mv archivo1.txt archivo2.txt
```

Este comando dispone de opciones análogas al anterior.

`rm` (ReMove)
Borra un archivo(s). En caso de que el argumento sea un directorio y se haya suministrado la opción `-r`, es posible borrar el directorio y todo su contenido. La opción `-i` pregunta antes de borrar.

[14] Muchos sistemas tienen esta opción habilitada a través de un alias, para evitar equivocaciones.

1.5.5. Espacio de disco.

El recurso de almacenamiento en el disco es siempre limitado. A continuación se comentan un par de comandos relacionados con la ocupación de este recurso:
`du` (Disk Usage)
Permite ver el espacio de disco ocupado (en bloques de disco[15]) por el archivo o directorio suministrado como argumento. La opción `-s` impide que cuando se aplique recursividad en un directorio se muestren los subtotales. La opción `-h` imprime los tamaños en un formato fácil de leer (Human readable).
`df` (Disk Free)
Muestra los sistemas de archivos que están montados en el sistema, con las cantidades totales, usadas y disponibles para cada uno. `df -h` muestra los tamaños en formato fácil de leer.

1.5.6. Links.

`ln` (LiNk)
Permite realizar un enlace (link) entre dos archivos o directorios. Un enlace puede ser:

- *hard link*: se puede realizar sólo entre archivos del mismo sistema de archivos. El archivo enlazado apunta a la zona de disco donde se ubica el archivo original. Por tanto, si se elimina el archivo original, el enlace sigue teniendo acceso a dicha información. Es el enlace por omisión.
- *symbolic link*: permite enlazar archivos/directorios[16] de diferentes sistemas de archivos. El archivo enlazado apunta al nombre del original. Así si se elimina el archivo original el enlace apunta hacia un nombre sin información asociada. Para realizar este tipo de enlace debe emplearse la opción `-s`.

Un enlace permite el uso de un archivo en otro directorio distinto del original sin necesidad de copiarlo, con el consiguiente ahorro de espacio. Veamos un ejemplo. Creemos un enlace clásico en Linux, al directorio existente `linux-2.6.12.5` nombrémoslo sencillamente linux.

```
mitarro:/usr/src# ln -s linux-2.6.12.5 linux
```

1.5.7. Protección de archivos.

Dado que el sistema de archivos UNIX es compartido por un conjunto de usuarios, surge el problema de la necesidad de privacidad. Sin embargo, dado que existen conjuntos de personas que trabajan en común, es necesaria la posibilidad de que un conjunto de usuarios puedan tener acceso a una serie de archivos (que puede estar limitado para el resto de los usuarios). Cada archivo y directorio del sistema dispone de un propietario, un grupo al que pertenece y unos **permisos**. Existen tres tipos fundamentales de permisos:

- **lectura** (**r**-*Read*): en el caso de un archivo, significa poder examinar el contenido del mismo; en el caso de un directorio significa poder entrar en dicho directorio.

[15] 1 bloque normalmente es 1 Kbyte.
[16] Debe hacerse notar que los directorios sólo pueden ser enlazados simbólicamente.

- **escritura** (**w**-*Write*): en el caso de un archivo significa poder modificar su contenido; en el caso de un directorio es crear un archivo o directorio en su interior.
- **ejecución** (**x**-*eXecute*): en el caso de un archivo significa que ese archivo se pueda ejecutar (binario o archivo de procedimientos); en el caso de un directorio es poder ejecutar alguna orden dentro de él.

Se distinguen tres grupos de personas sobre las que se deben especificar permisos:

- **user**: el usuario propietario del archivo.
- **group**: el grupo propietario del archivo (excepto el usuario). Como ya se ha comentado, cada usuario puede pertenecer a uno o varios grupos y el archivo generado pertenece a uno de los mismos.
- **other**: el resto de los usuarios (excepto el usuario y los usuarios que pertenezcan al grupo)

También se puede emplear *all* que es la unión de todos los anteriores. Para visualizar las protecciones de un archivo o directorio se emplea la orden `ls -l`, cuya salida es de la forma:
`-rw-r--r-- ...otra información... nombre`
Los 10 primeros caracteres muestran las protecciones de dicho archivo:

- El primer caracter indica el tipo de archivo de que se trata:
  - archivo
  - `d` directorio
  - `l` enlace (*link*)
  - `c` dispositivo de caracteres (p.e. puerta serial)
  - `b` dispositivo de bloques (p.e. disco duro)
  - `s` socket (conexión de red)
- Los caracteres 2, 3, 4 son los permisos de usuario
- Los caracteres 5, 6, 7 son los permisos del grupo
- Los caracteres 8, 9, 10 son los permisos del resto de usuarios

  Así en el ejemplo anterior `-rw-r--r--` se trata de un archivo donde el usuario puede leer y escribir, mientras que el grupo y el resto de usuarios sólo pueden leer. Estos suelen ser los permisos por omisión para un archivo creado por un usuario. Para un directorio los permisos por omisión suelen ser: `drwxr-xr-x`, donde se permite al usuario "entrar" en el directorio y ejecutar órdenes desde él.

  `chmod` (CHange MODe)
  Esta orden permite modificar los permisos de un archivo. Con opción `-R` es recursiva.
  `chmod` *permisos files*
  Existen dos modos de especificar los permisos:

- Modo absoluto o modo numérico. Se realiza empleando un número que resulta de la `OR` binario de los siguientes modos:

400	lectura por el propietario.
200	escritura por el propietario.
100	ejecución (búsqueda) por el propietario.
040	lectura por el grupo.
020	escritura por el grupo.
010	ejecución (búsqueda) por el grupo.
004	lectura por el resto.
002	escritura por el resto.
001	ejecución (búsqueda) por el resto.
4000	Set User ID, cuando se ejecuta el proceso corre con los permisos del dueño del archivo.

  Por ejemplo:
  `chmod 640 *.txt`
  Permite la lectura y escritura por el usuario, lectura para el grupo y ningún permiso para el resto, de un conjunto de archivos que acaban en `.txt`

- Modo simbólico o literal. Se realiza empleando una cadena (o cadenas separadas por comas) para especificar los permisos. Esta cadena se compone de los siguientes tres elementos: `who operation permission`

  - `who` : es una combinación de:
    - `u` : user
    - `g` : group
    - `o` : others
    - `a` : all (equivalente a `ugo`)

    Si se omite este campo se supone `a`, con la restricción de no ir en contra de la máscara de creación (`umask`).

  - `operation`: es una de las siguientes operaciones:
    - `+` : añadir permiso.
    - `-` : eliminar permiso.
    - `=` : asignar permiso, el resto de permisos de la misma categoría se anulan.

  - `permission`: es una combinación de los caracteres:
    - `r` : *read*.
    - `w` : *write*.
    - `x` : *execute*.
    - `s` : en ejecución fija el usuario o el grupo.

Por ejemplo:
`chmod u+x tarea`
Permite la ejecución por parte del usuario[17] del archivo `tarea`.

`chmod u=rx, go=r *.txt`
Permite la lectura y ejecución del usuario, y sólo la lectura por parte del grupo y el resto de usuarios.

`umask`
Esta es una orden intrínseca del Shell que permite asignar los permisos que se desea tengan los archivos y directorios por omisión. El argumento que acompaña a la orden es un número octal que aplicará una XOR sobre los permisos por omisión (`rw-rw-rw-`) para archivos y (`rwxrwxrwx`) para directorios. El valor por omisión de la máscara es `022` que habilita al usuario para lectura-escritura, al grupo y al resto para lectura. Sin argumentos muestra el valor de la máscara.
`chgrp` (CHange GRouP)
Cambia el grupo propietario de una serie de archivos/directorios
`chgrp` *grupo files*
El usuario que efectúa esta orden debe pertenecer al grupo mencionado.
`chown` (CHange OWNer)
Cambia el propietario y el grupo de una serie de archivos/directorios
`chown` *user:group files*
La opción `-r` hace que la orden se efectúe recursivamente.
`id`
Muestra la identificación del usuario[18], así como el conjunto de grupos a los que el usuario pertenece.

```
user@hostname:~$ id
uid=1000(user) gid=1000(group) groups=1000(group),25(floppy),29(audio)
user@hostname:~$
```

1.5.8. Filtros.

Existe un conjunto de órdenes en UNIX que permiten el procesamiento de archivos de texto. Se denominan **filtros** (*Unix Filters*), porque normalmente se trabaja empleando redirección recibiendo datos por su `stdin`[19] y retornándolos modificados por su `stdout`[20].

Para facilitar la comprensión de los ejemplos siguientes supondremos que existen dos archivos llamados `mylist.txt` y `yourlist.txt` que tienen en su interior:

[17] Un error muy frecuente es la creación de un archivo de órdenes (*script file*) y olvidar permitir la ejecución del mismo.
[18] A pesar de que el usuario se identifica por una cadena denominada *username*, también existe un número denominado UID que es un identificativo numérico de dicho usuario.
[19] Entrada estándar.
[20] Salida estándar.

```
mylist.txt       yourlist.txt  tercero.txt

1 190            1 190           11  b
2 280            2 281           33  c
3 370            3 370           222 a
```

`echo`
Éste no es propiamente un filtro, pero nos será muy útil más adelante. Despliega sobre la pantalla un mensaje, sin argumento despliega una línea en blanco. La opción `-n` elimina el cambio de línea al final del mensaje.

```
user@hostname:~$ echo Hola Mundo
Hola Mundo
user@hostname:~$ echo ; echo chao; echo

chao

user@hostname:~$
```

Varias instrucciones pueden ser separadas por `;`

`cat`
Es el filtro más básico, copia la entrada a la salida.

```
user@hostname:~$ cat  mylist.txt
1 190
2 280
3 370
user@hostname:~$
```

También lo podemos usar para crear un archivo

```
user@hostname:~$ cat  > myfile.txt
Este es mi archivo
con muchas lineas
^D
user@hostname:~$
```

`seq`
Genera una secuencia de números naturales consecutivos.

```
user@hostname:~$ seq 4 8
4
5
6
7
8
```

`cut`

Para un archivo compuesto por columnas de datos, permite escribir sobre la salida cierto intervalo de columnas. La opción `-b N-M` permite indicar el intervalo en bytes que se escribirán en la salida.

```
user@hostname:~$ cut -b 3-4 mylist.txt
19
28
37
user@hostname:~$
```

`paste`

Mezcla líneas de distintos archivos. Escribe líneas en el `stdout` pegando secuencialmente las líneas correspondientes de cada uno de los archivo separadas por tab. Ejemplo, supongamos que tenemos nuestros archivos `mylist.txt` y `yourlist.txt` y damos el comando

```
user@hostname:~$ paste mylist.txt yourlist.txt
1 190   1 190
2 280   2 281
3 370   3 370
user@hostname:~$
```

`sed`

Es un editor de flujo. Veamos algunos ejemplos

```
user@hostname:~$ sed = mylist.txt
1
1 190
2
2 280
3
3 370
user@hostname:~$
```

Numera las líneas.

```
user@hostname:~$ sed -n -e '3p' mylist.txt
3 370
user@hostname:~$
```

Sólo muestra la línea 3. El modificador `-n` suprime la impresión de todas las líneas excepto aquellas especificadas por `p`. El modificador `-e` corre un *script*, secuencia de comandos. Separando por coma damos un rango en el número de líneas.

```
user@hostname:~$ sed -e '2q' mylist.txt
1 190
2 280
user@hostname:~$
```

Muestra hasta la línea 2 y luego se sale de `sed`.

```
user@hostname:~$ sed -e 's/0/a/g' mylist.txt
1 19a
2 28a
3 37a
user@hostname:~$
```

Reemplaza todos los 0 del archivo por la letra a. Éste es uno de los usos más comunes.

```
user@hostname:~$ sed -e '/2 2/s/0/a/g' mylist.txt
1 190
2 28a
3 370
user@hostname:~$
```

Busca las líneas con la secuencia `2 2` y en ellas reemplaza todos los 0 por la letra a.

```
user@hostname:~$ sed -e 's/1/XX/2' mylist.txt
1 XX90
2 280
3 370
user@hostname:~$
```

Reemplaza la segunda aparición de un 1 en una línea por los caracteres XX.

A continuación mostramos otras posibilidades del comando `sed`

- Para remover una línea especifica (`X`) de un archivo (`file.txt`)

  ```
  user@hostname:~$ sed -e 'Xd' file.txt
  ```

- Para remover un intervalo de líneas de un archivo

  ```
  user@hostname:~$ sed -e 'X,Yd' file.txt
  ```

- Para mostrar sólo las líneas `X` e `Y` de un archivo

  ```
  user@hostname:~$ sed -n -e 'Xp;Yp' file.txt
  ```

- Para mostrar un archivo salvo las líneas que contengan `key`

  ```
  user@hostname:~$ sed -e '/key/d' file.txt
  ```

- Para mostrar de un archivo sólo las líneas que contengan `key`

  ```
  user@hostname:~$ sed -n -e '/key/p' file.txt
  ```

- Para mostrar un archivo salvo las líneas que comienzan con #

```
user@hostname:~$ sed -e '/^#/d' file.txt
```

Expresiones regulares:

^	Matches al comienzo de la línea
$	Matches al final de la línea
.	Matches cualquier caracter.
[]	Matches con todos los caracteres dentro de los paréntesis

`diff`

Permite comparar el contenido de dos archivos o directorios

```
user@hostname:~$ diff mylist.txt yourlist.txt
2c2
< 2 280
---
> 2 281
user@hostname:~$
```

Hay una diferencia entre los archivos en la segunda fila.

`sort`

Permite ordenar alfabéticamente

```
user@hostname:~$ sort tercero.txt
11  b
222 a
33  c
user@hostname:~$

user@hostname:~$ sort -r tercero.txt
33  c
222 a
11  b
user@hostname:~$

user@hostname:~$ sort -n tercero.txt
11  b
33  c
222 a
user@hostname:~$

user@hostname:~$ sort -k 2 tercero.txt
222 a
11  b
33  c
user@hostname:~$
```

La opción `-n` considera los valores numéricos y la opción `-r` invierte el orden. La opción `-k` permite especificar la columna a usar para hacer el `sort`.

`find`

Permite la búsqueda de un archivo en la estructura de directorios

```
find . -name file.dat -print
```

Comenzando en el directorio actual recorre la estructura de directorios buscando el archivo `file.dat`, cuando lo encuentre imprime el path al mismo.

```
find . -name '*~' -exec rm '{}' \;
```

Busca en la estructura de directorios un archivo que acabe en `~` y lo borra. `xargs` ordena repetir orden para cada argumento que se lea desde *stdin*. Permite el uso muy eficiente de `find`.

```
find . -name '*.dat' -print | xargs mv ../data \;
```

Busca en la estructura de directorios todos los archivos que acaben en `.dat`, y los mueve al directorio `../data`.

`grep`

Permite la búsqueda de una cadena de caracteres en uno o varios archivos, imprimiendo el nombre del archivo y la línea en que se encuentra la cadena.

```
user@hostname:~$ grep 1 *list.txt
mylist.txt:1 190
yourlist.txt:1 190
yourlist.txt:2 281
user@hostname:~$
```

Algunas opciones útiles

- `-c` Elimina la salida normal y sólo cuenta el número de apariciones de la cadena en cada archivo.
- `-i` Ignora para la comparación entre la cadena dada y el archivo, si la cadena está en mayúsculas o minúsculas.
- `-n` Incluye el número de líneas en que aparece la cadena en la salida normal.
- `-r` Hace la búsqueda recursiva.
- `-v` Invierte la búsqueda mostrando todas las líneas donde no aparece la cadena pedida.

`head`

Muestra las primeras diez líneas de un archivo.
`head -30 file` Muestra las 30 primeras líneas de *file*.

```
user@hostname:~$ head -1  mylist.txt
1 190
user@hostname:~$
```

`tail`

Muestra las diez últimas líneas de un archivo.
`tail -30 file` Muestra las 30 últimas líneas de *file*.
`tail +30 file` Muestra desde la línea 30 en adelante de *file*.

```
user@hostname:~$ tail -1  mylist.txt
3 370
user@hostname:~$
```

La opción `-f` permite que se actualice la salida cuando el archivo crece.

`awk`

Es un procesador de archivos de texto que permite la manipulación de las líneas de forma tal que tome decisiones en función del contenido de la misma. Ejemplo, supongamos que tenemos nuestro archivo `mylist.txt` con sus dos columnas

```
user@hostname:~$ awk '{print }'  mylist.txt
1 190
2 280
3 370
user@hostname:~$
```

Funciona como el comando `cat`

```
user@hostname:~$ awk '{print $2, $1 }'  mylist.txt
190 1
280 2
370 3
user@hostname:~$
```

Imprime esas dos columnas en orden inverso.

```
user@hostname:~$ awk '{print ``a'', 8*$1, $2-1 }'  mylist.txt
a 8 189
a 16 279
a 24 369
user@hostname:~$
```

Permite operar sobre las columnas.

```
user@hostname:~$ awk '{ if (NR>1 && NR < 3) print}' mylist.txt
2 280
user@hostname:~$
```

Sólo imprime la línea 2.

`tar`

Este comando permite la creación/extracción de archivos contenidos en un único archivo denominado `tarfile` (o `tarball`). Este `tarfile` suele ser luego comprimido con `gzip`, la versión de compresión **gnu**[21] o bien con `bzip2`.

La acción a realizar viene controlada por el primer argumento:

- `c` (Create) creación

[21] **gnu** es un acrónimo recursivo, significa: **gnu**'s Not UNIX! **gnu** es el nombre del producto de la *Free Software Foundation*, una organización dedicada a la creación de programas compatibles con UNIX (y mejorado respecto a los estándars) y de libre distribución. La distribución de Linux **gnu** es **debian**.

- `x` (eXtract) extracción
- `t` (lisT) mostrar contenido
- `r` añadir al final
- `u` (Update) añadir aquellos archivos que no se hallen en el tarfile o que hayan sido modificados con posterioridad a la versión que aparece.

A continuación se colocan algunas de las opciones:

- `v` Verbose (indica qué archivos son agregados a medida que son procesados)
- `z` Comprimir o descomprimir el contenido con `gzip`.
- `j` Comprimir o descomprimir el contenido con `bzip2`.
- `f` File: permite especificar el archivo para el tarfile.

Veamos algunos ejemplos:

```
tar cvf simul.tar *.dat
```

Genera un archivo `simul.tar` que contiene todos los archivos que terminen en `.dat` del directorio actual. A medida que se va realizando indica el tamaño en bloques de cada archivo añadido modo *verbose*.

```
tar czvf simul.tgz *.dat
```

Igual que en el caso anterior, pero el archivo generado simul.tgz ha sido comprimido empleando gzip.

```
tar tvf simul.tar
```

Muestra los archivos contenidos en el tarfile `simul.tar`.

```
tar xvf simul.tar
```

Extrae todos los archivos contenidos en el tarfile `simul.tar`.

`wc` (*Word Count*) Contabiliza el número de líneas, palabras y caracteres de un archivo.

```
user@hostname:~$ wc  mylist.txt
      3       6      18 mylist.txt
user@hostname:~$
```

El archivo tiene 3 líneas, 6 palabras, considerando cada número como una palabra *i.e.* 1 es la primera palabra y 190 la segunda, y finalmente 18 caracteres. ¿Cuáles son los 18 caracteres?

1.5.9. Otros usuarios y máquinas

`users` `who` `w`
Para ver quién está conectado en la máquina.

`ping`
Verifica si una máquina está conectada a la red y si el camino de Internet hasta la misma funciona correctamente.

`finger`

`finger user`, muestra información[22] sobre el usuario `user` en la máquina local.
`finger user@hostname`, muestra información sobre un usuario llamado `user` en una máquina `hostname`.
`finger @hostname`, muestra los usuarios conectados de la máquina `hostname`.

1.5.10. Fecha

`cal`

Muestra el calendario del mes actual. Con la opción `-y` y el año presenta el calendario del año completo.

`date`

Muestra el día y la hora actual.

1.5.11. Diferencias entre sistemas.

Cuando se transfieren archivos de texto entre WINDOWS y UNIX sin las precauciones adecuadas pueden aparecer los siguientes problemas:

- En UNIX no existe restricción respecto a la longitud del nombre, y aunque pueden llevar extensión, no es obligatorio. También pueden tener más de una extensión `algo.v01.tar.gz`, esto complica a otros sistemas que usan sólo una extensión de tres caracteres.
- El cambio de línea en un archivo de texto WINDOWS se compone de *Carriage Return* y *Line Feed*. Sin embargo, en UNIX sólo existe el *Carriage Return*. Así un archivo de UNIX visto desde WINDOWS parece una única línea. El caso inverso es la aparición del caracter `^M` al final de cada línea. Además, el fin de archivo en WINDOWS es `^Z` y en UNIX es `^D`.

Usando el comando `tr` se puede transformar un archivo con cambios de líneas para dos en uno para UNIX. Sabiendo que `^M` es ASCII 13 decimal, pero 15 en octal:

```
tr -d '\015' < datafile > TEMPFILE
mv -f TEMPFILE datafile
```

En Debian, instalando el paquete `sysutils`, queda instalado el comando `dos2unix` que también lo hace.

1.6. Shells.

El sistema operativo UNIX soporta varios intérpretes de comandos o *shells*, que ayudan a que la interacción con el sistema sea lo más cómoda y amigable posible. La elección de cuál es la *shell* más cómoda es algo personal; en este punto sólo indicaremos las cuatro más significativas y populares:

[22] La información proporcionada es el nombre completo del usuario, las últimas sesiones en dicha máquina, si ha leído o no su correo y el contenido de los archivos `.project` y `.plan` del usuario.

- **sh** : Bourne SHell, el *shell* básico, no pensado para uso interactivo.
- **csh** : C-SHell, *shell* con sintaxis como el lenguaje "C". El archivo de configuración es `.cshrc` (en el directorio `$HOME`).
- **tcsh** : alTernative C-Shell (Tenex-CSHell), con editor de línea de comando. El archivo de configuración es `.tcshrc`, o en caso de no existir, `.cshrc` (en el directorio `$HOME`).
- **bash** : Bourne-Again Shell, con lo mejor de sh, ksh y tcsh. El archivo de configuración es `.bash_profile` cuando se entra a la cuenta por primera vez, y después el archivo de configuración es `.bashrc` siempre en el directorio `$HOME`. La línea de comando puede ser editada usando comandos (secuencias de teclas) del editor `emacs`. Es el *shell* por defecto de Linux.

Si queremos cambiar de *shell* en un momento dado, sólo será necesario que tecleemos el nombre del mismo y estaremos usando dicho *shell*. Si queremos usar de forma permanente otro *shell* del que tenemos asignado por omisión[23] podemos emplear la orden `chsh` que permite realizar esta acción.

En los archivos de configuración se encuentran las definiciones de las variables de entorno (*enviroment variables*) como camino de búsqueda `PATH`, los aliases y otras configuraciones personales. Veamos unos caracteres con especial significado para el Shell:

- ` [24] permite que el output de un comando reemplace al nombre del comando. Por ejemplo: `echo ‘pwd‘` imprime por pantalla el nombre del directorio actual.

```
user@hostname:~$ echo ‘pwd‘
/home/user
user@hostname:~$
```

- ' [25] preserva el significado literal de cada uno de los caracteres de la cadena que delimita.

```
user@hostname:~$ echo ’Estoy en ‘pwd‘’
Estoy en ‘pwd‘
user@hostname:~$
```

- " [26] preserva el significado literal de todos los caracteres de la cadena que delimita, salvo $, ‘, \.

```
user@hostname:~$ echo "Estoy en ‘pwd‘"
Estoy en /home/user
user@hostname:~$
```

- ; permite la ejecución de más de una orden en una sola línea de comando.

```
user@hostname:~$ mkdir textos; cd textos; cp ../*.txt . ; cd ..
user@hostname:~$
```

[23] Por omisión se asigna bash.
[24] Acento agudo o inclinado hacia atrás, *backquote*.
[25] Acento usual o inclinado hacia adelante, *single quote*.
[26] *double quote*.

1.6.1. Variables de entorno.

Las variables de entorno permiten la configuración, por defecto, de muchos programas cuando ellos buscan datos o preferencias. Se encuentran definidas en los archivos de configuración anteriormente mencionados. Para referenciar a las variables se debe poner el símbolo `$` delante, por ejemplo, para mostrar el camino al directorio por defecto del usuario `user`:

```
user@hostname:~$ echo $HOME
/home/user
user@hostname:~$
```

Las variables de entorno más importantes son:

- `HOME` - El directorio por defecto del usuario.
- `PATH` - El camino de búsqueda, una lista de directorios separados con ':' para buscar programas.
- `EDITOR` - El editor por defecto del usuario.
- `DISPLAY` - Bajo el sistema de X windows, el nombre de máquina y pantalla que está usando. Si esta variable toma el valor `:0` el despliegue es local.
- `TERM` - El tipo de terminal. En la mayoría de los casos bajo el sistema X windows se trata de `xterm` y en la consola en Linux es `linux`. En otros sistemas puede ser `vt100`.
- `SHELL` - La *shell* por defecto.
- `MANPATH` - Camino para buscar páginas de manuales.
- `PAGER` - Programa de paginación de texto (`less` o `more`).
- `TMPDIR` - Directorio para archivos temporales.

1.6.2. Redirección.

Cuando un programa espera que se teclee algo, aquello que el usuario teclea se conoce como el *Standard Input*: `stdin`. Los caracteres que el programa retorna por pantalla es lo que se conoce como *Standard Output*: `stdout` (o *Standard Error*: `stderr`[27]). El signo `<` permite que un programa reciba el `stdin` desde un archivo en vez de la interacción con el usuario. Por ejemplo: `mail root < file`, invoca el comando `mail` con argumento (destinatario del mail) `root`, siendo el contenido del mensaje el contenido del archivo `file` en vez del texto que usualmente teclea el usuario. Más a menudo aparece la necesidad de almacenar en un archivo la salida de un comando. Para ello se emplea el signo `>`. Por ejemplo, `man bash > file`, invoca el comando `man` con argumento (información deseada) `bash` pero indicando que la información debe ser almacenada en el archivo `file` en vez de ser mostrada por pantalla.

En otras ocasiones uno desea que la salida de un programa sea la entrada de otro. Esto se logra empleando los denominados *pipes*, para ello se usa el signo `|`. Este signo permite que

[27] Si estos mensajes son de error.

el `stdout` de un programa sea el `stdin` del siguiente. Por ejemplo:
`zcat manual.gz | more`
Invoca la orden de descompresión de `zcat` y conduce el **flujo** de caracteres hacia el paginador `more`, de forma que podamos ver página a página el archivo descomprimido. A parte de los símbolos mencionados existen otros que permiten acciones tales como:

- `>>` Añadir el `stdout` al final del archivo indicado (*append*).[28]
- `>&` o `&>` (sólo `csh`, `tcsh` y `bash`) Redireccionar el `stdout` y `stderr`. Con `2>` redireccionó sólo el `stderr`.
- `>>&` Igual que `>&` pero en modo *append*.

1.6.3. Ejecución de comandos.

- Si el comando introducido es propio del *shell* (*built-in*), se ejecuta directamente.
- En caso contrario:
  - Si el comando contiene `/`, el *shell* lo considera un `PATH` e intenta resolverlo (entrar en cada directorio especificado para encontrar el comando).
  - En caso contrario el *shell* busca en una tabla *hash table* que contiene los nombres de los comandos que se han encontrado en los directorios especificados en la variable `PATH`, cuando ha arrancado el *shell*.

1.6.4. Aliases.

Para facilitar la entrada de algunas órdenes o realizar operaciones complejas, los *shells* interactivos permiten el uso de aliases. La orden `alias` permite ver qué aliases hay definidos y también definir nuevos. Es corriente definir el alias `rm =‘rm -i’`, de esta forma la orden siempre pide confirmación para borrar un archivo. Si alguna vez quieres usar `rm` sin alias, sólo hace falta poner delante el símbolo \, denominado *backslash* . Por ejemplo `\rm` elimina los alias aplicados a `rm`. Otro ejemplo, bastante frecuente (en `tcsh/csh`) podría ser (debido a la complejidad de la orden): `alias ffind ’find . -name \!* -print’`. Para emplearlo: `ffind tema.txt`, el resultado es la búsqueda recursiva a partir del directorio actual de un archivo que se llame `tema.txt`, mostrando el camino hasta el mismo.

1.6.5. Las shells csh y tcsh.

Son dos de los Shells interactivos más empleados. Una de las principales ventajas de `tcsh` es que permite la edición de la línea de comandos, y el acceso a la historia de órdenes usando las teclas de cursores.[29]

[28] En `bash`, si el archivo no existe, es creado.
[29] `bash` también lo permite.

Comandos propios.

Los comandos propios o intrínsecos, *Built-In Commands*, son aquéllos que proporciona el propio *shell* [30].

`alias name def`
Asigna el nombre `name` al comando `def`.

`history`
Muestra las últimas órdenes introducidas en el *shell*. Algunos comandos relacionados con el *Command history* son:

- `!!`
  Repite la última orden.
- `!n`
  Repite la orden n-ésima.
- `!string`
  Repite la orden más reciente que empiece por la cadena `string`.
- `!?string`
  Repite la orden más reciente que contenga la cadena `string`.
- `^str1^str2` o `!!:s/str1/str2/`
  (*substitute*) Repite la última orden reemplanzando la primera ocurrencia de la cadena `str1` por la cadena `str2`.
- `!!:gs/str1/str2/`
  (*global substitute*) Repite la última orden reemplazando todas las ocurrencias de la cadena `str1` por la cadena `str2`.
- `!$`
  Es el último argumento de la orden anterior que se haya tecleado.

`repeat count command`
Repite `count` veces el comando `command`.

`rehash`
Rehace la tabla de comandos (*hash table*).

`set variable = VALUE`
Asigna el valor de una variable del *shell*.

`set`
Muestra el valor de todas las variables.

[30] A diferencia de los comandos que provienen de un ejecutable situado en alguno de los directorios de la variable `PATH`.

`setenv VARIABLE VALUE`
Permite asignar el valor de una variable de entorno.

`source file`
Ejecuta las órdenes del fichero `file` en el *shell* actual.

`unset variable`
Borra la asignación del valor de una variable del *shell*.

`unsetenv VARIABLE VALUE`
Borra la asignación del valor de una variable de entorno.

`umask value`
Asigna la máscara para los permisos por omisión.

`unalias name`
Elimina un alias asignado.

Variables propias del shell.

Existe un conjunto de variables denominadas *shell variables*, que permiten modificar el funcionamiento del *shell*.
`filec` (*FILE Completion*)
Es una variable *toggle* que permite que el *shell* complete automáticamente el nombre de un archivo o un directorio[31]. Para ello, si el usuario introduce sólo unos cuantos caracteres de un archivo y pulsa el `TAB`, el *shell* completa dicho nombre. Si sólo existe una posibilidad, el completado es total y el *shell* deja un espacio tras el nombre. En caso contrario hace sonar un pitido. Pulsando `Ctrl-D` el *shell* muestra las formas existentes para completar.
`prompt`
Es una variable de cadena que contiene el texto que aparece al principio de la línea de comandos.
`savehist`
Permite definir el número de órdenes que se desea almacenar al abandonar el *shell*. Esto permite recordar las órdenes que se ejecutaron en la sesión anterior.

1.6.6. Las shell sh y bash.

Sólo `bash` puede considerarse un *shell* interactivo, permitiendo la edición de la línea de comandos, y el acceso a la historia de órdenes (*readline*). En uso normal (historia y editor de línea de comandos) `BASH` es compatible con `TCSH` y `KSH`. El modo de completado (*file completion*) es automático (usando `TAB` sólo) si el *shell* es interactivo.

[31] `bash` permite no sólo completar ficheros/directorios sino también comandos.

Comandos propios del shell.

Los comandos `umask`, `source`, `history`, `unalias`, `hash` [32], funcionan igual que en la *shell* `TCSH`.

`help`
Ayuda interna sobre los comandos del *shell*.

`VARIABLE=VALUE`
Permite asignar el valor de una variable de entorno. Para que dicha variable sea "heredada" es necesario emplear: `export VARIABLE` o bien combinarlas: `export VARIABLE=VALUE`.

`for var in wordlist do comandos done`
A la variable `var`, que puede llamarse de cualquier modo, se le asignan sucesivamente los valores de la cadena `wordlist`, y se ejecuta el conjunto de comandos. El contenido de dicha variable puede ser empleado en los comandos: `$var`. Ejemplo:

```
$ for i in 1 2 tres 4; do echo $i; done
1
2
tres
4
```

`alias`
En `bash`, `alias` sólo sirve para substitución simple de una cadena por otra. Por ejemplo: `alias ls='ls -F'`. Para crear alias con argumentos se usan funciones, ver la documentación.

1.6.7. Archivos de *script*.

Un archivo de *script* es una sucesión de comandos de la *shell* que se ejecutan secuencialmente. Veamos un ejemplo simple:

```
#!/bin/bash
variable=''/home/yo''
cp $1 /tmp/$2
rm $1
cd $variable
# Hecho por mi
```

La primera línea declara la *shell* específica que se quiere usar. En la segunda línea hay una declaración de una variable interna. La tercera contiene los dos primeros argumentos con que fue llamado el *script*. Por ejemplo, si el anterior *script* está en un archivo llamado `ejemplo`, el comando `ejemplo file1 file2` asocia `$1` a `file1` y `$2` a `file2`. La línea 5 hace uso de la variable interna dentro de un comando. La última línea, que comienza con un `#` corresponde

[32] En `bash/sh` la *hash table* se va generando dinámicamente a medida que el usuario va empleando las órdenes. Así el arranque del *shell* es más rápido, y el uso de orden equivalente `hash -r` casi nunca hace falta.

a un comentario. Notemos que la primera también es un comentario, pero la combinación `#!` en la primera línea fuerza a que se ejecute esa *shell*.

Esto sólo es una mínima pincelada de una herramienta muy poderosa y útil. Los comandos disponibles en la *shell* conforman un verdadero lenguaje de programación en sí, y los *scripts* pueden diseñarse para realizar tareas monótonas y complejas. Éste es un tema que le será útil profundizar.

1.7. Ayuda y documentación.

Para obtener ayuda sobre comandos de UNIX, se puede emplear la ayuda *on-line*, en la forma de páginas de manual. Así `man comando` proporciona la ayuda sobre el `comando` deseado. Por ejemplo, para leer el manual de los shells, puedes entrar: `man sh csh tcsh bash`, la orden formatea las páginas y te permite leer los manuales en el orden pedido. En el caso de `bash` se puede usar el comando `help`, por ejemplo, `help alias`. Además, para muchos comandos y programas se puede obtener información tipeando `info comando`. Finalmente, algunos comandos tienen una opción de ayuda (`--help`), para recordar rápidamente las opciones más comunes disponibles (`ls --help`).

1.8. Procesos.

En una máquina existen una multitud de procesos que pueden estar ejecutándose simultáneamente. La mayoría de ellos no corresponden a ninguna acción realizada por el usuario y no merecen que se les preste mayor atención. Estos procesos corresponden a programas ejecutados en el arranque del sistema y tienen que ver con el funcionamiento global del servidor. En general, los programas suelen tener uno de estos dos modos de ejecución:

- **foreground**: Son aquellos procesos que requieren de la interacción y/o atención del usuario mientras se están ejecutando, o bien en una de sus fases de ejecución (*i.e.* introducción de datos). Así por ejemplo, la consulta de una página de manual es un proceso que debe ejecutarse claramente en *foreground*.
- **background**: Son aquellos procesos que no requieren de la interacción con el usuario para su ejecución. Si bien el usuario desearía estar informado cuando este proceso termine. Un ejemplo de este caso sería la impresión de un archivo.

Sin embargo, esta división que a primera vista pueda parecer tan clara y concisa, a menudo en la práctica aparece la necesidad de conmutar de un modo al otro, detención de tareas indeseadas, etc. Así por ejemplo, puede darse el caso de que estemos leyendo una página de manual y de repente necesitemos ejecutar otra tarea. Un proceso viene caracterizado por:

- *process number*
- *job number*

Veamos algunas de las órdenes más frecuentes para la manipulación de procesos:

- `comando &` Ejecución de un comando en el *background*. [33]

[33] Por omisión un comando se ejecuta siempre en el *foreground*.

- `Ctrl-Z` Detiene el proceso que estuviera ejecutándose en el *foreground* y lo coloca detenido en el *background*.
- `Ctrl-C` Termina un proceso que estaba ejecutándose en *foreground*.
- `Ctrl-\` Termina de forma definitiva un proceso que estaba ejecutándose en *foreground*.
- `ps x` Lista todos los procesos que pertenezcan al usuario, incluyendo los que no están asociados a un terminal.
- `jobs` Lista los procesos que se hayan ejecutado desde el *shell* actual, mostrando el *job number*.
- `fg (job number)` Pasa a ejecución en *foreground* un proceso que se hallase en *background*.
- `bg (job number)` Pasa a ejecución en *background* un proceso que se hallase detenido con `Ctrl-Z`.
- `kill (process number)` Envía una señal[34] a un proceso UNIX. En particular, para enviar la señal de término a un programa, damos el comando `kill -KILL`, pero no hace falta al ser la señal por defecto.

Cuando se intenta abandonar una sesión con algún proceso aún detenido en el *background* del *shell*, se informa de ello con un mensaje del tipo: `There are stopped jobs` si no importa, el usuario puede intentar abandonar de nuevo el *shell* y éste matará los *jobs*, o puedes utilizar `fg` para traerlos al *foreground* y ahí terminar el mismo.

1.9. Editores.

Un editor es un programa que permite crear y/o modificar un archivo. Existen una multitud de editores diferentes, y al igual que ocurre con los *shells*, cada usuario tiene alguno de su predilección. Mencionaremos algunos de los más conocidos:

- **vi** - El editor standard de UNIX.
- **emacs (xemacs)** - Editor muy configurable escrito en lenguaje Lisp. Existen muchos modos para este editor (lector de mail, news, www,...) que lo convierten en un verdadero *shell* para multitud de usuarios. Las últimas versiones del mismo permiten la ejecución desde X-windows o terminal indistintamente con el mismo binario. Posee un tutorial en línea, comando `C-H t` dentro del editor. El archivo de configuración personalizada es: `$HOME/.emacs`.
- **jove** - Basado en Emacs, (Jonathan's Own Version of Emacs). Posee tutorial en una utilidad asociada: `teachjove`. El archivo de configuración personalizada es: `$HOME/.joverc`.

[34] Para ver las señales disponibles entra la orden `kill -l` (l por *list*).

- **jed** - Editor configurable escrito en S-Lang. Permite la emulación de editores como emacs y Wordstar. Posee una ayuda en línea `C-H C-H`. El archivo de configuración personalizada es: `$HOME/.jedrc`.
- **gedit** - Un pequeño y liviano editor de texto para Gnome
- **xjed** - Versión de jed para el X-windows system. Presenta como ventaja que es capaz de funcionar en muchos modos: lenguaje C, Fortran, TeX, etc., reconociendo palabras clave y signos de puntuación, empleando un colorido distinto para ellos. El archivo de configuración personalizada es el mismo que el de jed.

Dado que los editores del tipo de **gedit** disponen de menús auto explicativos, daremos a continuación unas ligeras nociones sólo de `vi` y `emacs`.

1.9.1. El editor vi.

El `vi` es un editor de texto muy poderoso pero un poco difícil de usar. Lo importante de este editor es que se puede encontrar en cualquier sistema UNIX y sólo hay unas pocas diferencias entre un sistema y otro. Explicaremos lo básico solamente. Comencemos con el comando para invocarlo:

```
localhost:/# vi
```

```
~
~
~
/tmp/vi.9Xdrxi: new file: line 1
```

La sintaxis para editar un archivo es:

```
localhost:/# vi nombre.de.archivo
```

```
~
~
~
nombre.de.archivo: new file: line 1
```

Insertar y borrar texto en vi.

Cuando se inicia el `vi`, editando un archivo, o no, se entra en un modo de órdenes, es decir, que no se puede empezar a escribir directamente. Si se quiere entrar en modo de inserción de texto se debe presionar la tecla `i`. Entrando en el modo de inserción, se puede empezar a escribir. Para salir del modo de inserción de texto y volver al modo de órdenes se apreta `ESC`.

```
Aqui ya estamos escribiendo porque apretamos
la tecla 'i' al estar en modo ordenes.
~
~
```

La tecla `a` en el modo de órdenes también entra en modo de inserción de texto, pero en vez de comenzar a escribir en la posición del cursor, empieza un espacio después.

La tecla `o` en el modo de órdenes inserta texto pero desde la línea que sigue a la línea donde se está ubicado.

Para borrar texto, hay que salir al modo órdenes, y presionar la tecla `x` que borrará el texto que se encuentre sobre el cursor. Si se quiere borrar las líneas enteras, entonces se debe presionar dos veces la tecla `d` sobre la línea que deseo eliminar. Si se presionan las teclas `dw` se borra la palabra sobre la que se está ubicado.

La letra `R` sobre una palabra se puede escribir encima de ella. Esto es una especie de modo de inserción de texto pero sólo se podrá modificar la palabra sobre la que se está situado. La tecla `~` cambia de mayúscula a minúscula la letra sobre la que se está situado.

Moverse dentro de vi.

Estando en modo órdenes podemos movernos por el archivo que se está editando usando las flechas hacia la izquierda, derecha, abajo o arriba. Con la tecla `0` nos movemos al comienzo de la línea y con la tecla `$` nos movemos al final de la misma.

Con las teclas `w` y `b` nos movemos al comienzo de la siguiente palabra o al de la palabra anterior respectivamente. Para moverme hacia la pantalla siguiente la combinación de teclas `CTRL F` y para volver a la pantalla anterior `CTRL B`. Para ir hasta el principio del archivo se presiona la tecla `G`.

Opciones de comandos.

Para entrar al menú de comandos se debe presionar la tecla `:` en el modo de órdenes. Aparecerán los dos puntos (`:`). Aquí se pueden ingresar ordenes para guardar, salir, cambiar de archivo entre otras cosas. Veamos algunos ejemplos:

- `:w` Guardar los cambios.
- `:w otherfile.txt` Guardar con el nuevo nombre `otherfile.txt`
- `:wq` Guardar los cambios y salir.
- `:q!` Salir del archivo sin guardar los cambios.
- `:e file1.txt` Si deseo editar otro archivo al que se le pondrá por nombre `file1.txt`.
- `:r file.txt` Si se quiere insertar un archivo ya existente, por ejemplo `file.txt`.
- `:r! comando` Si se quiere ejecutar algún comando del *shell* y que su salida aparezca en el archivo que se está editando.

1.9.2. Editores modo emacs.

El editor `GNU Emacs`, escrito por Richard Stallman de la *Free Software Foundation*, es uno de los que tienen mayor aceptación entre los usuarios de UNIX, estando disponible bajo licencia **GNU GPL**[35] para una gran cantidad de arquitecturas. También existe otra versión de emacs llamada `XEmacs` totalmente compatible con la anterior pero presentando mejoras significativas respecto al `GNU Emacs`. Dentro de los "inconvenientes" que presenta es que no viene por defecto incluido en la mayoría de los sistemas UNIX. Las actuales distribuciones de Linux y en particular Debian GNU/Linux contienen ambas versiones de emacs, tanto `GNU Emacs` como `XEmacs`, como también versiones de `jove`, `jed`, `xjed` y muchos otros editores. Para mayor información ver Apéndice.

1.10. El sistema X Windows.

El *X Windows system* es el sistema estándar de ventanas en las estaciones de trabajo. Lo usual actualmente es que el sistema de ventanas sea arrancado automáticamente cuando la máquina parte. En el sistema *X Windows* deben distinguirse dos conceptos:

- **server** : Es un programa que se encarga de escribir en el dispositivo de video y de capturar las entradas (por teclado, ratón, etc.). Asimismo se encarga de mantener los recursos y preferencias de las aplicaciones. Sólo puede existir un server para cada pantalla.
- **client** : Es cualquier aplicación que se ejecute en el sistema *X Windows*. No hay límite (en principio) en el número de clientes que pueden estarse ejecutando simultáneamente. Los clientes pueden ser locales o remotos.

Window Manager (WM) Es un cliente con "privilegios especiales": controla el comportamiento (forma, tamaño,...) del resto de clientes. Existen varios, destacando:

- **icewm** : *Ice Window Manager*, uno de los *window managers* gnome compatible.
- **sawfish** : *Window managers* gnome compatible, altamente configurable y muy integrado al gnome *desktop*.
- **Metacity** : *Window managers* gnome 2 compatible.

El *look and feel* (o GUI) de *X Windows* es extremadamente configurable, y puede parecer que dos máquinas son muy distintas, pero esto se debe al `WM` que se esté usando y no a que las aplicaciones sean distintas.

Para configurar tu sesión es necesario saber qué programas estás usando y ver las páginas de manual. Los archivos principales son:

- `.xinitrc` o `.xsession` archivo leído al arrancar *X Windows*. Aquí se pueden definir los programas que aparecen al inicio de tu sesión.

[35] La licencia de GNU, da el permiso de libre uso de los programas con sus fuentes, pero los autores mantienen el *Copyright* y no es permitido distribuir los binarios sin acceso a sus fuentes. Los programas derivados de dichos fuentes heredan la licencia GNU.

- `.fvwmrc` archivo de configuración del `fvwm`. Ver las páginas del manual de `fvwm`.
- `.olwmrc` archivo de configuración del `olwm`. Ver las páginas del manual de `olwm`.
- `.Xdefaults` Configuración general de las aplicaciones de X Windows. Aquí puedes definir los *resources* que encontrarás en los manuales de las aplicaciones de X.

En caso de que tengas que correr una aplicación de X que no esté disponible en la máquina que estás usando, eso no representa ningún problema. Las órdenes necesarias son (por ejemplo, para arrancar un `gnome-terminal` remoto):

```
user@hostname1:~$ ssh -XC userB@hostname2
userB@hostname2's password:
userB@hostname2:~$ gnome-terminal &
```

Las opciones `XC` en el comando `ssh` corresponden a que exporte el `DISPLAY` y que comprima, respectivamente. La forma antigua

```
userA@hostname1:~$ xhost +hostname2
hostname2 being added to access control list
user@hostname1:~$ ssh userB@hostname2
userB@hostname2's password:
userB@hostname2:~$ export DISPLAY=hostname1:0
userB@hostname2:~$ gnome-terminal &
```

Si todo está previamente configurado, es posible que no haga falta dar el *password*.

Cuando quieres salir, normalmente puedes encontrar un ícono con la opción `Log out`, en un menú o panel de la pantalla.

1.11. Uso del ratón.

El ratón es un dispositivo esencial en el uso de programas X, sin embargo, la función que realiza en cada uno de ellos no está normalizada.

Comentaremos la pauta seguida por la mayoría de las aplicaciones, pero debe tenerse presente que es muy frecuente encontrar aplicaciones que no las respetan.[36]

- **Botón izquierdo** (LB): Seleccionar. Comienza el bloque de selección.
- **Botón central** (MB): Pegar. Copia la selección en la posición del cursor.
- **Botón derecho** (RB): Habitualmente ofrece un menú para partir aplicaciones.

Existen dos modos para determinar cuál es la **ventana activa**, aquélla que recibe las entradas de teclado:

[36] Las aplicaciones que son conscientes de un uso anormal y están realizadas por programadores inteligentes, muestran en pantalla la función de cada botón cuando son posibles varias alternativas.

- *Focus Follows Mouse*: La ventana que contenga al ratón es la que es activa. No usado por defecto actualmente.

- *Click To Focus*: La ventana seleccionada es la activa. El modo que esté activo depende de la configuración del *Window Manager*.

1.12. Internet.

En esta sección denominaremos `unix1` a la máquina local (desde donde ejecutamos la orden) y `unix2` a la máquina remota (con la que interaccionamos). Ambos son los `hostnames` de las respectivas máquinas. Existen algunos conceptos que previamente debemos comentar:

- `IP-number`: es un conjunto de 4 números separados por puntos (p.e. 200.89.74.6) que se asocia a cada máquina. No puede haber dos máquinas conectadas en la misma red con el mismo número.

- `hostname`: es el nombre que tiene asociada la máquina (p.e. `macul`). A este nombre se le suelen añadir una serie de sufijos separados por puntos que constituye el denominado dominio (p.e. `macul.ciencias.uchile.cl`). Una máquina por tanto puede tener más de un nombre reconocido (se habla en este caso de alias). Se denomina resolución a la identificación entre un `hostname` y el `IP-number` correspondiente. La consulta se realiza inicialmente en el archivo `/etc/hosts`, donde normalmente se guardan las identificaciones de las máquinas más comúnmente empleadas. En caso de que no se logre se accede al servicio `DNS` (*Domain Name Service*), que permite la identificación (resolución) entre un `hostname` y un `IP-number`.

- `mail-address`: es el nombre que se emplea para enviar correo electrónico. Este nombre puede coincidir con el nombre de una máquina, pero se suele definir como un alias, con objeto de que la dirección no deba de cambiarse si la máquina se estropea o se cambia por otra.

1.12.1. Acceso a la red.

Existen muchos programas para la conexión de la red, los más usados son:

- `telnet unix2`, hace un login en la máquina `unix2`, debe ingresarse el usuario y su respectiva `passwd`. Además, permite especificar el puerto en conexión en la máquina remota.

- `ssh nombre@unix2`, muy similar a `telnet` pero se puede especificar el usuario, si no se especifica se usa el nombre de la cuenta local. Además, el `passwd` pasa encriptado a través de la red. `ssh nombre@unix2 comando`, muy similar a `rsh`, el `passwd` pasa encriptado y ejecuta el comando en la máquina remota, mostrando el resultado en la máquina local.

- `scp file1 usuario2@unix2:path/file`, copia el archivo `file1`, del usuario1, que se encuentra en el directorio local en la máquina `unix1`, en la cuenta del `usuario2` en la máquina `unix2` en `$HOME/path/file`. Si no se especifica el nombre del usuario se usa el nombre de la cuenta local. Si se quiere copiar el archivo `file2` del `usuario3` en `unix2` en la cuenta actual de `unix1` el comando sería: `scp usuario3@unix2:file2 .`. Antes de realizar cualquiera de las copias el sistema preguntará por el `passwd` del usuario en cuestión en la máquina `unix2`. Nuevamente, el `passwd` pasa encriptado a través de la red.

- `talk usuario1@unix2`, intenta hacer una conexión para hablar con el `usuario1` en la máquina `unix2`. Existen varias versiones de `talk` en los diferentes sistemas operativos, de forma que no siempre es posible establecer una comunicación entre máquinas con sistemas operativos diferentes.

- `ftp unix2`, (file transfer protocol) aplicación para copiar archivos entre máquinas de una red. `ftp` exige un nombre de cuenta y password para la máquina remota. Algunas de las opciones más empleadas (una vez establecida la conexión) son:

  - `bin`: Establece el modo de comunicación binario. Es decir, transfiere una imagen exacta del archivo.
  - `asc`: Establece el modo de comunicación `ascii`. Realiza las conversiones necesarias entre las dos máquinas en comunicación. Es el modo por defecto.
  - `cd`: Cambia directorio en la máquina remota.
  - `lcd`: Cambia directorio en la máquina local.
  - `ls`: Lista el directorio remoto.
  - `!ls`: Lista el directorio local.
  - `prompt` : No pide confirmación para transferencia múltiple de archivos.
  - `get rfile [lfile]`: transfiere el archivo `rfile` de la máquina remota a la máquina local denominándolo `lfile`. En caso de no suministrarse el segundo argumento supone igual nombre en ambas máquinas.
  - `put lfile [rfile]` : transfiere el archivo `lfile` de la máquina local a la máquina remota denominándolo `rfile`. En caso de no suministrarse el segundo argumento supone igual nombre en ambas máquinas. También puede usarse `send`.
  - `mget rfile` : igual que `get`, pero con más de un archivo (`rfile` puede contener caracteres comodines).
  - `mput lfile` : igual que `put`, pero con más de un archivo (`lfile` puede contener caracteres comodines).

  Existen versiones mejoradas de `ftp` con muchas más posibilidades, por ejemplo, `ncftp`. También existen versiones gráficas de clientes `ftp` donde la elección de archivo, el sentido de la transferencia y el modo de ésta, se elige con el *mouse* (p.e. `wxftp`).

- `rlogin -l nombre unix2`, (*remote login*), hace un `login` a la máquina `unix2` como el usuario `nombre` por defecto, sin los argumentos `-l nombre` `rlogin` usa el nombre de la cuenta local. Normalmente `rlogin` pide el *password* de la cuenta remota, pero con el uso del archivo `.rhosts` o `/etc/hosts.equiv` esto no es siempre necesario.

- `rsh -l nombre unix2 orden`, (*remote shell*), ejecuta la orden en la máquina `unix2` como usuario `nombre`. Es necesario que pueda entrar en la máquina remota sin *password* para ejecutar una orden remota. Sin especificar orden actúa como `rlogin`.

1.12.2. El correo electrónico.

El correo electrónico (`e-mail`) es un servicio para el envío de mensajes entre usuarios, tanto de la misma máquina como de diferentes máquinas.

Direcciones de correo electrónico.

Para mandar un `e-mail` es necesario conocer la dirección del destinatario. Esta dirección consta de dos campos que se combinan intercalando entre ellos el @ (*at*): `user@domain`

- `user` : es la identificación del usuario (*i.e.* `login`) en la máquina remota.

- `domain` : es la máquina donde recibe correo el destinatario. A menudo, es frecuente que si una persona tiene acceso a un conjunto de máquinas, su dirección de correo no corresponda con una máquina sino que corresponda a un alias que se resolverá en un nombre específico de máquina en forma oculta para el que envía.

Si el usuario es local, no es necesario colocar el campo `domain` (ni tampoco el @).

Nomenclatura.

Veamos algunos conceptos relacionados con el correo electrónico:

- **Subject** : Es una parte de un mensaje que piden los programas al comienzo y sirve como título para el mensaje.

- **Cc** (Carbon Copy) : Permite el envío de copias del mensaje que está siendo editado a terceras personas.

- **Reply** : Cuando se envía un mensaje en respuesta a otro se suele añadir el comienzo del *subject:* `Re:`, con objeto de orientar al destinatario sobre el tema que se responde. Es frecuente que se incluya el mensaje al que se responde para facilitar al destinatario la comprensión de la respuesta.

- **Forward** : Permite reenviar un mensaje completo (con modificaciones o sin ellas) a una tercera persona. Notando que Forward envía también los archivos adjuntos, mientras que la opción Reply no lo hace.

- **Forwarding Mail** : Permite a un usuario que disponga de cuentas en varias máquinas no relacionadas, de concentrar su correo en una cuenta única[37]. Para ello basta con tener un archivo `$HOME/.forward` que contenga la dirección donde desea centralizar su correo.

- **Mail group** : Un grupo de correo es un conjunto de usuarios que reciben el correo dirigido a su grupo. Existen órdenes para responder a un determinado correo recibido por esa vía de forma que el resto del grupo sepa lo que ha respondido un miembro del mismo.

- **In-Box** : Es el archivo donde se almacena el correo que todavía no ha sido leído por el usuario. Suele estar localizado en `/var/spool/mail/user`.

- **Mailer-Daemon** : Cuando existe un problema en la transmisión de un mensaje se recibe un mensaje proveniente del *Mailer-Daemon* que indica el problema que se ha presentado.

Aplicación mail.

Es posiblemente la aplicación más simple. Para la lectura de mail teclear simplemente: `mail` y a continuación aparece un índice con los diferentes mensajes recibidos. Cada mensaje tiene una línea de identificación con número. Para leer un mensaje basta teclear su número y a continuación `return`. Para enviar un mensaje: `mail (address)` se pregunta por el `Subject:` y a continuación se introduce el mensaje. Para acabar se teclea sólo un punto en una línea o bien `Ctr-D`. Por último, se pregunta por `Cc:`. Es posible personalizar el funcionamiento mediante el archivo `$HOME/.mailrc`. Para enviar un archivo de texto a través del correo se suele emplear la redirección de entrada: `mail (address) < file`. Si queremos enviar un archivo binario en forma de `attach` en el mail, el comando es `mpack archivo-binario address`.

1.12.3. Ftp anonymous.

Existen servidores que permiten el acceso por `ftp` a usuarios que no disponen de cuenta en dichas máquinas. Para ello se emplea como `login` de entrada el usuario `anonymous` y como `passwd` la dirección de *e-mail* personal. Existen servidores que no aceptan conexiones desde máquinas que no están declaradas correctamente en el servicio de nombre (*dns*), así como algunas que no permiten la entrada a usuarios que no se identifican correctamente. Dada la sobrecarga que existe, muchos de los servidores tienen limitado el número de usuarios que pueden acceder simultáneamente.

1.12.4. WWW.

WWW son las siglas de *World-Wide Web*. Este servicio permite el acceso a información entrelazada (dispone de un texto donde un término puede conducir a otro texto): *hyperlinks*. Los archivos están realizados en un lenguaje denominado *html*. Para acceder a este servicio

[37]Este comando debe usarse con conocimiento pues en caso contrario podría provocar un *loop* indefinido y no recibir nunca correo.

es necesario disponer de un lector de dicho lenguaje conocido como *browser* o navegador. Destacan actualmente: Firefox, Mozilla, Opera, Camino (para MAC) y el simple pero muy rápido Lynx.

1.13. Impresión.

Cuando se quiere obtener una copia impresa de un archivo se emplea el comando `lpr`.

`lpr file` - Envía el archivo `file` a la cola de impresión por defecto. Si la cola está activada, la impresora lista y ningún trabajo por encima del enviado, nuestro trabajo será procesado de forma automática.

A menudo existen varias posibles impresoras a las que poder enviar los trabajos. Para seleccionar una impresora en concreto (en vez de la por defecto) se emplea el modificador: `lpr -Pimpresora`, siendo `impresora` el nombre lógico asignado a esta otra impresora. Para recibir una lista de las posibles impresoras de un sistema, así como su estado, se puede emplear el comando `/usr/sbin/lpc status`. La lista de impresoras y su configuración también está disponible en el archivo `/etc/printcap`.

Otras órdenes para la manipulación de la cola de impresión son:

- `lpq [-Pimpresora]`, permite examinar el estado de una determinada cola (para ver la cantidad de trabajos sin procesar de ésta, por ejemplo).
- `lprm [-Pimpresora] jobnumber`, permite eliminar un trabajo de la cola de impresión.

Uno de los lenguajes de impresión gráfica más extendidos en la actualidad es *PostScript*. La extensión de los archivos *PostScript* empleada es `.ps`. Un archivo *PostScript* puede ser visualizado e impreso mediante los programas: `gv`, `gnome-gv` o `ghostview`. Por ello muchas de las impresoras actuales sólo admiten la impresión en dicho formato.

En caso de desear imprimir un archivo `ascii` deberá previamente realizarse la conversión a *PostScript* empleando la orden `a2ps`: `a2ps file.txt` Esta orden envía a la impresora el archivo `ascii file.txt` formateado a 2 páginas por hoja. Otro programa que permite convertir un archivo `ascii` en postscript es `enscript`.

Otro tipo de archivos ampliamente difundido y que habitualmente se necesita imprimir es el conocido como *Portable Document Format*. Este tipo de archivo poseen una extensión `.pdf` y pueden ser visualizados e impresos usando aplicaciones tales como: `xpdf`, `acroread` o `gv`.

1.14. Compresión.

A menudo necesitamos comprimir un archivo para disminuir su tamaño, o bien crear un respaldo (*backup*) de una determinada estructura de directorios. Se comentan a continuación una serie de comandos que permiten ejecutar dichas acciones.

El compresor `compress` está relativamente fuera de uso[38] pero aún podemos encontrarnos con archivos comprimidos por él.

[38]Este comando no se incluye en la instalación básica. Debemos cargar el paquete `ncompress` para tenerlo

- `uncompress file.Z` : descomprime el archivo, creando el archivo `file`. Destruye el archivo original.
- `zcat file.Z` : muestra por el `stdout` el contenido descomprimido del archivo (sin destruir el original).
- `compress file` : comprime el archivo, creando el archivo `file.Z`. Destruye el archivo original.

Otra alternativa de compresor mucho más usada es `gzip`, el compresor de `GNU` que posee una mayor razón de compresión que `compress`. Veamos los comandos:

- `gzip file` : comprime el archivo, creando el archivo `file.gz`. Destruye el archivo original.
- `gunzip file.gz` : descomprime el archivo, creando el archivo `file`. Destruye el archivo original.
- `zless file.gz` : muestra por el `stdout` el contenido descomprimido del archivo paginado por `less`.

La extensión empleada en los archivos comprimidos con `gzip` suele ser `.gz`, pero a veces se usa `.gzip`. Adicionalmente el programa `gunzip` también puede descomprimir archivos creados con `compress`.

La opción con mayor tasa de compresión que `gzip` es `bzip2` y su descompresor `bunzip2`.

- `bzip2 file` : comprime el archivo, creando el archivo `file.bz2`. Destruye el archivo original.
- `bunzip2 file.bz2` : descomprime el archivo, creando el archivo `file`. Destruye el archivo original.
- `bzcat file.bz2` : muestra por el `stdout` el contenido descomprimido del archivo. Debemos usar un paginador, adicionalmente, para verlo por páginas.

La extensión usada en este caso es `.bz2`. El *kernel* de Linux se distribuye en formato `bzip2`. También existe una versión paralelizada llamada `pbzip2`. Uno de los mejores algoritmos de compresión está disponible para Linux en el programa `p7zip`. Veamos un ejemplo: un archivo `linux-2.6.18.tar` que contiene el kernel 2.6.18 de Linux que tiene un tamaño de 230 Mb. Los resultados al comprimirlo con `compress`, `gzip`, `bzip2` y `7za` son: `linux-2.6.18.tar.Z` 91 Mb, `linux-2.6.18.tar.gz` 51 Mb, `linux-2.6.18.tar.bz2` 40 Mb y `linux-2.6.18.tar.7z` 33 Mb.[39]

Existen también versiones de los compresores compatibles con otros sistemas operativos: `zip`, `unzip`, `unarj`, `lha`, `rar` y `zoo`.

[39] Los comandos `gzip` y `bzip2` fueron dados con la opción `--best` para lograr la mayor compresión. El comando usado para la compresión con `7z` fue: `7za a -t7z -m0=lzma -mx=9 -mfb=64 -md=32m -ms=on file.tar.7z file.tar`, note la nueva extensión `7z`. Para descomprimir con `7z` basta `7z e file.tar.7z`

En caso que se desee crear un archivo comprimido con una estructura de directorios debe ejecutarse la orden:

```
tar cvzf  nombre.tgz directorio
```

o bien

```
tar cvjf  nombre.tbz directorio
```

En el primer caso comprime con `gzip` y en el segundo con `bzip2`. Para descomprimir y restablecer la estructura de directorio almacenada se usan los comandos:

```
tar xvzf  nombre.tgz directorio
```

si se realizó la compresión con `gzip` o bien

```
tar xvjf  nombre.tbz directorio
```

si se realizó la compresión con `bzip2`.

Capítulo 2

Una breve introducción a Python.

versión 1.7, 12 de Octubre del 2006

En este capítulo se intentará dar los elementos más básicos del lenguaje de programación Python. No se pretende más que satisfacer las mínimas necesidades del curso, sirviendo como un ayuda de memoria de los tópicos abordados, para futura referencia. Se debe consignar que no se consideran todas las posibilidades del lenguaje y las explicaciones están reducidas al mínimo.

2.1. Introducción a programación.

2.1.1. ¿Qué es programar?

A continuación, presentamos algunas alternativas de respuesta a esta pregunta:

- Hacer un programa.
- Hacer que un computador haga una secuencia de instrucciones que uno le pide.
- Darle, de alguna forma, una secuencia de pasos lógicos para que un computador los ejecute con la intención de alcanzar algún objetivo.
- Escribir una precisa secuencia de comandos o instrucciones, en algún lenguaje que el computador entienda (a este tipo de lenguaje lo llamaremos *lenguaje de programación*) para que un computador las realice exactamente, paso a paso.

Un programa puede ser tan corto como una sola línea de código, o tan largo como varios millones de líneas de código

2.1.2. Lenguajes de programación.

Existen diferentes tipos de lenguajes de programación, algunos más cercanos a la máquina y menos al programador; otros más cercanos al programador y distantes de la máquina. Realmente existe toda una jerarquía entre los lenguajes de programación. Veamos algunos ejemplos:

Código de Máquina binario.

Es el lenguaje de la CPU, y el lenguaje de más bajo nivel. Compuesto de 0 y 1 `binario`, lo que está muy cerca de la máquina pero muy lejos del programador. No es fácil de escribir para el programador.

Un programa simple, como *Hola mundo*, se vería en código binario algo así como:

```
10001010101010011110010
10001011010101000111010
11000101000101010000111
00101010101010010110000
11110010101010101000011
10001010010101010101001
00101010101101010101001
```

Lenguaje de Ensamblador (Assembler).

Una secuencia de abreviaturas para el lenguaje de máquina. Está cerca de la máquina pero no tanto como el anterior. Veamos el programa *Hola mundo* en lenguaje de Ensamblador de la familia de procesadores X86.

```
title Programa Hola Mundo (hello.asm)
; Este programa muestra "Hola, Mundo!" dosseg
.model small
.stack 100h .data
hello_message db 'Hola, Mundo!', 0dh, 0ah, '$' .code
main proc
mov ax, @ .data
mov ds,ax mov ah,9
mov dx, offset hello_message
int 21h mov ax,4C00h
int 21h
main endp
end main
```

Lenguaje de alto nivel.

Utilizan declaraciones en los programas, expresiones como palabras y expresiones algebraicas. Fueron desarrollados en las décadas del 50 y 60. Son lenguajes que están más cerca de los programadores que de la máquina, por lo tanto, necesitan ser traducidos para que los entienda la máquina. Este proceso se puede hacer de dos maneras: compilando o interpretando el programa.

Lenguajes Compilados.

En este caso, otro programa (el compilador) reescribe el programa inicial en lenguaje de máquina para que la CPU pueda entenderlo. Esto se hace de una sola vez y el programa

final se guarda en esta nueva forma (ejecutable). Un programa compilado se estima que será considerablemente más largo que el original. Algunos de los lenguajes compilados más notables son Fortran, C y C++. Un ejemplo del programa *Hola mundo* escrito en C++ es dado a continuación:

```
//
// Programa Hola Mundo
//
#include <iostream>

using namespace std;

int main()
{
    cout << "Hola mundo" << endl;
    return 0;
}
```

Lenguajes interpretados.

En este caso un programa (Intérprete) traduce las declaraciones del programa original a lenguaje de máquina, línea por línea, a medida que va corriendo dicho programa original. Un programa interpretado será más pequeño que uno compilado pero tardará más tiempo en ser ejecutado. Existe gran cantidad de este tipo de lenguajes, Python, Perl, Bash, por nombrar algunos. Un ejemplo del programa *Hola mundo* escrito en Python es dado a continuación:

```
# Programa Hola mundo
print "Hola Mundo"
```

Lenguajes especializados.

Desde el punto de vista de la funcionalidad de los lenguajes podemos separarlos en lenguajes de carácter general y lenguajes especializados. Los lenguajes de carácter general son aquellos que sirven para programar una gran número de problemas, por ejemplo C o C++. Los lenguajes especializados han sido diseñados para realizar tareas específicas. Ejemplos de ello son PHP y JavaScript, especializados en crear páginas web, o SQL, creado para manipular información en bases de datos.

Un lista de lenguajes.

A continuación, damos una lista, incompleta, de algunos de los lenguajes de programación más comunes en la actualidad:

ABC, Ada, ASP, Awk, BASIC, C, **C++**, C#, Caml, Cobol, código de máquina, Corba, Delphi, Eiffel, Erlang, Fortran, Haskell, Java, JavaScript, Lisp, Logo, Modula, Modula 2, Mozart, Mumps, Oberon, Objetive C, Oz, Pascal, Perl, PHP, **Python**, Realbasic, Rebol, Rexx, RPG, Ruby, Scheme, Smaltalk, SQL, Squeak, TCL, Visual Basic.

2.1.3. Lenguajes naturales y formales.

Lenguajes naturales.

Son lenguajes hablados por la gente (por ejemplo: Español, Inglés, Alemán o Japonés). Una de sus caracteristicas es que son ambiguos, por ejemplo: "Dame esa cosa.º "¡Oh, seguro, Grande!". En ambos ejemplos no es claro a que se están refiriendo y se necesita un contexto para entenderlos. Muchas veces estos lenguajes son redundantes y están llenos de expresiones idiomáticas las cuales no deben ser tomadas literalmente, por ejemplo: "Me podría comer una vaca", "Me mataste", o "Ándate a la punta del cerro".

Lenguajes formales.

Hecho por el hombre, como las Matemáticas, Química o los lenguajes de Programación de computadores. Se caracterizan por ser inambiguos. Por ejemplo, una expresión matemática: $1 + 4 = 5$; o una expresión en química: $CH_4+2O_2 \rightarrow 2H_2O+CO_2$; o, finalmente, una expresión en lenguaje de programación `print "Hola mundo"`. Los lenguajes formales son además concisos y estrictamente literales.

Sintaxis.

Los lenguajes, tanto naturales como formales, tienen reglas de sintaxis. Por una parte, están los *tokens*, que corresponden a los elementos básicos (*i.e.* letras, palabras, símbolos) del lenguaje:

- Tokens correctos: 1+3=4; gato, H_2O.
- Tokens incorrectos: 2@+#=!;C;H_hO.

Por otro lado, tenemos las estructuras, esto es la manera en que los *tokens* son organizados:

- Estructuras correctas: $1 + 3 = 4$, gato, H_2O.
- Estructuras incorrectas: $13+ = 4$, gtoa, $_2HO$.

2.1.4. Sacar los errores de un programa.

Los errores en un programa son llamados *bugs*. Al proceso de rastrear los errores y corregirlos se le conoce como *debugging*. Un programa especializado en hacer *debugging* es llamado *debugger*. **El debugging es una de las más importantes habilidades en programación.** Los tres principales tipos de errores o *bugs* y sus consecuencias para la ejecución del programa son:

1. Errores de sintaxis
   - Usar un *token* o estructuralos en forma incorrecta
   - El programa termina con un mensaje de error.
2. Errores de ejecución (*run-time error*)

- Errores que ocurren durante la ejecución.
- El programa deja de correr abruptamente.

3. Errores lógicos

- Errores en cómo el programa está lógicamente construido.
- El programa corre, pero hace cosas mal.

2.2. Python.

El Lenguaje Python fue inventado alrededor de 1990 por el científico en computación holandes Guido van Rossem y su nombre es un tributo a la grupo cómico *Monty Python* del cual Guido es admirador. El sitio oficial del lenguage en la *web* es `http://www.python.org`.

2.2.1. Interactivo versus *scripting*.

El programa Python (como programa, no como lenguaje) posee un ambiente interactivo que nos permite ejecutar instrucciones del lenguaje Python directamente. Para ello, basta dar el comando:

```
username@host:~$ python
Python 2.4.3 (#2, Apr 27 2006, 14:43:32)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El programa ofrece un *prompt* (`>>>`), esperando instrucciones del usuario. Las instrucciones son interpretadas y ejecutadas de inmediato. Esta forma de usar Python tiene la ventaja de la retroalimentación inmediata; de inmediato el programador sabe si la instrucción está correcta o incorrecta. Sin embargo, tiene la desventaja de que el código no es guardado, y no puede por tanto ser reutilizado.

Por otra parte, cuando escribimos un archivo de instrucciones en Python (*script*), tenemos la ventaja de que el código sí es almacenado, pudiendo ser reutilizado. En este caso las desventajas son que la retroalimentación no es inmediata y que habitualmente requiere más *debugging* para que el código funcione correctamente.

2.2.2. Creando un *script*.

Para crear un *script* de Python requerimos de un editor (vi, jed, xemacs, gedit... elija su favorito). Para ser ordenado, grábelo con extensión (supongamos que lo grabamos como `archivo.py`), para poder identificarlo rápidamente más adelante. Recuerde darle los permisos de ejecución adecuados (`chmod u+x archivo.py`). Para ejecutarlo basta ubicarse en el directorio donde está el archivo y dar el comando

```
jrogan@manque:~/InProgress/python$ ./archivo.py
```

2.3. Lenguaje Pyhton.

2.3.1. Algunos tipos básicos.

- **Cadenas de caracteres** (*strings*): Usualmente un conjunto de caracteres, *i.e.* un texto. Están delimitados por `"comillas"` simples o dobles.
- **Números enteros**: Los números, que pertenecen al conjunto $\mathbb{Z}$, es decir, sin decimales. No pueden ser mayores que un tamaño fijo, en torno a dos billones (2×10^{12}) en un sistema de 32 bits usando signo $\pm$. Cuando se dividen entre ellos sólo dan valores enteros. Por ejemplo: `4/3=1`.
- **Números con punto flotante**: Los números, que pertenecen al conjunto $\mathbb{R}$, es decir, con decimales (un número finito de ellos).
- **Enteros largos**: Números enteros mayores que 2×10^{12}. Los distinguimos por una `L` al final del número, por ejemplo: `23434235234L`.

Varios de los otros tipos serán explicados más adelante en el capítulo

Tipo	Descripción	Ejemplo
bool	booleano	True o False
int	entero	117
long int	entero largos	23434235234L
float	número con punto flotante	1.78
complex	número complejo	0.5 +2.0j
str	*string*	'abc'
tuple	tuplas	(1, 'hum', 2.0)
list	listas	[1, 'hum', 2.0]
dict	diccionario	'a':7.0, 23: True
file	archivo	file('stuff.dat', 'w')

Cuadro 2.1: Los tipos del lenguaje Python.

Trabajando con números.

Use el tipo de número en el cual quiere obtener su resultado. Es decir, si usted desea un valor con decimales, use al menos un número con decimales en el cálculo. Por ejemplo: 15/2.0 producirá 7.5 y 15/2 producirá 7, porque son ambos enteros. Si desea enteros largos, use al menos un entero largo en su expresión, por ejemplo: 23434235234L/2.

Número complejos.

Los números complejos son también soportados en Python. Los números imaginarios puros son escritos con un sufijo `j` o `J`. Los números complejos con parte real no nula son escritos como *real*+*imag*`j`, o pueden ser creados con la función `complex(`*real*,*imag*`)`. Ejemplos

```
>>> 1j*1J
(-1+0j)
>>> 1j*complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Cadenas de caracteres (*strings*).

Una cadena de caracteres debe estar entre apóstrofes o comillas simples o dobles. Por ejemplo:

- `nombre = "Este es tu nombre"`
- `nombre2= ‘Este es tambien su nombre’`

Si una cadena de caracteres necesita un apóstrofe dentro de ella, anteponga un \ al apóstrofe extra. Ejemplos:

- `titulo = "Ella dijo: \"Te amo\""`
- `titulo2 = ‘I\’m a boy’`

Algunas cadenas de caracteres con significado especial empiezan con el caracter \ (*String Backslash Characters*).

- `\\` = Incluya \.
- `\’` = Apóstrofe simple.
- `\"` = Apóstrofe doble.
- `\n` = Línea nueva.

2.3.2. Imprimiendo en la misma línea.

Agregando una coma (`,`) al final de una instrucción `print` hará que el próximo comando `print` aparezca en la misma línea. Ejemplo

```
print num1,"+", num2, "=",
print respuesta
```

2.3.3. Imprimiendo un texto de varias líneas.

Si queremos imprimir un texto que tenga varias líneas podemos usar dos formas distintas de la función `print` usando el caracter `\n` o bien usando un texto entre triple comilla

```
>>> print "primera linea\nsegunda linea"
primera linea
segunda linea
>>> print """primera linea
... segunda linea"""
primera linea
segunda linea
```

2.3.4. Variables.

Las variable son un nombre, usado dentro del programa, para referirse a un objeto o valor. Las limitaciones y consideraciones que hay que tener en cuenta para darle nombre a una variable son:

- No puede ser una palabra reservada del lenguaje (*i.e.* `print, and, or, not`).
- No puede comenzar por un número.
- Las mayúsculas y las minúsculas son diferentes.
- No puede incluir caracteres ilegales (*i.e.* `$,%,+,=`).

Cree variables cuyos nombres signifiquen algo:

- MAL : `diy=365`
- BIEN: `days_in_year=365`

2.3.5. Asignación de variables.

Para asignarle un valor a una variable, digamos `num`, basta poner el nombre de la variable a la izquierda un signo igual y al lado derecho el valor o expresión que queremos asignarle

```
num=8.0
num=pi*3.0**2
```

Un mismo valor puede ser asignado a varias variables simultaneamente

```
>>> x=y=z=0  # Todas las variables valen cero
>>> print x,y,z
0 0 0
```

O bien podemos hacer asignasiones diferentes valores a diferentes variables en una misma asignación

```
>>> a,b=0,1
>>> print a,b
0 1
>>> a,b=b,a+b
>>> print a,b
1 1
>>> a,b,c=0,1,2
>>> print a,b,c
0 1 2
```

2.3.6. Operaciones matemáticas.

Con Python podemos realizar las operaciones básicas: suma ($+$), resta ($-$), multiplicación ($*$) y división ($/$). Operaciones menos básicas también están disponibles: el exponente ($**$), el módulo ($\%$).

Entre las operaciones hay un orden de precedencia, unas se realizarán primero que otras. A continuación damos el orden de precedencia, partiendo por lo que se hace primero:

- Paréntesis, exponentes, multiplicación y división,
- Suma y resta.
- De izquierda a derecha.

Como ejemplo de la importancia de saber el orden de precedencia veamos los siguiente ejemplos:

$$2 * (3 - 1) = 4 \qquad \text{y} \qquad 2 * 3 - 1 = 5$$

2.3.7. Operaciones con *strings*.

Dos de las operaciones más comunes con *strings*:

- Concatenación: se pueden concatenar dos *strings* al sumarlos, veamos un ejemplo:

```
>>> x = "Hola"
>>> y = "Mundo"
>>> print x+y
>>> HolaMundo
```

- Repetición:

```
>>> z = "Ja"
>>> print z*3
>>> JaJaJa
```

2.3.8. Composición.

Se pueden combinar sentencias simples en una compuesta, a través del operador ",":

```
>>> x = "Elizabeth"
>>> print "Tu nombre es : ",x
>>> Tu nombre es : Elizabeth
```

En el ejemplo, `x` fue asignado explícitamente a una variable, pero naturalmente cualquier tipo de asignación es posible, por ejemplo:

```
>>> promedio=(nota+extra_creditos)/posibles
>>> print "Tu promedio es : ",promedio
```

2.3.9. Comentarios.

Los comentarios son anotaciones que usted escribe para ayudar a explicar lo que está haciendo en el programa. Los comentarios comienzan con `#`. Lo escrito después de `#`, hasta el final de la línea, es ignorado por el intérprete. Por ejemplo:

```
dias = 60 #disponibles para el proyecto
```

Naturalmente, los comentarios no son muy útiles cuando se trabaja interactivamente con Python, pero sí lo son cuando se escribe un script. De este modo se pueden insertar explicaciones en el código que ayuden a recordar qué hace el programa en cada una de sus secciones, o explicarlo a terceros. Es buena costumbre de programación que las primeras líneas de un código sean comentarios que incluyan el nombre del programador y una breve descripción del programa.

2.3.10. Entrada (input).

Para leer *strings* del *stdin* use la instrucción `raw_input()`, por ejemplo

```
nombre = raw_input("Cual es tu nombre?")
```

Si necesita leer números del *stdin* use la instrucción `input()`:

```
numero=input("Cuantos?")
```

En ambos casos, el mensaje entre comillas dentro de los paréntesis es opcional, sin embargo, aclara al usuario lo que el programa le está solicitando. En el siguiente par de ejemplos, el programa solicita información al usuario, salvo que en el primero, el programa queda esperando una respuesta del usuario, quien, a menos que sepa de antemano qué quiere el programa, no tiene modo de saber por qué le programa no se continuúa ejecutando.

Ejemplo sin mensaje (queda esperando para siempre una respuesta):

```
>>> nombre = raw_input()
```

Ejemplo con mensaje:

```
>>> nombre = raw_input("Cual es tu nombre?")
Cual es tu nombre? Pedro
>>>
```

2.3.11. Interfaz con el usuario.

Siempre que escriba un programa debe tener presente que alguien, que puede no ser usted mismo, lo puede usar alguna vez. Lo anterior significa, en particular, que un programa sin documentación es muy difícil de usar. Pero además es importante cuidar la parte del programa que interactúa con el usuario, es decir la *interfaz con el usuario*. Esta interfaz podrían ser tanto mensajes simples de texto como sofisticadas ventanas gráficas. Lo importante es que ayuden al usuario a ejecutar correctamente el programa.

Revisemos una mala interfaz con el usuario. Tenemos un programa que no sabemos lo que hace, pero al ejecutarse resulta lo siguiente:

```
>>> Entre un numero
>>> Entre otro numero
>>> La respuesta es 12
```

Hay una evidente falta de instrucciones de parte del programador para el usuario, que primero no sabe para qué se le pide cada número, y luego no sabe qué hizo con ellos, sólo la respuesta, 12, sin mayor explicación de lo que significa.

Como contraparte, una buena interfaz con el usuario tiene documentación anexa o bien ayuda en el mismo programa. Esta documentación debiera explicar que hace el programa, los datos que necesitará y el o los resultados que entregará cuando finalice.

Cada vez que se le pide algo al usuario deberían estar claras las siguientes preguntas: ¿qué es exactamente lo que se supone que yo tipee?; ¿los números que ingreso deben tener decimales?; ¿o deben ser sin decimales?; ¿en qué unidades de medidas debo ingresarlos?; ¿los números que se piden son grandes o son números pequeños? Si se trata de palabras, ¿debo ingresarlas en minúsculas o mayúsculas?

Algunos lineamientos básicos que debería observar para construir interfaces con el usuario que sea claras son los siguientes:

- Parta con un título e indicaciones dentro del programa.
- Cuando pregunte por un dato que quiere que el usuario ingrese, dele la ayuda necesaria, por ejemplo

  ```
  Entre el largo en metros (0-100):
  ```

- Que las preguntas tengan sentido.
- Use espacios y caracteres especiales para mantener la pantalla despejada.
- Indíquele al usuario que el programa terminó.

Una versión mejorada del programa anterior podría ser la siguiente:

```
>>>
Calculo de la suma de dos numeros
Ingrese un numero entero: 5
Ingrese otro numero entero: 7
La suma es 12
```

2.4. Funciones Pre-hechas.

Una función define un conjunto de instrucciones. Es un conjunto de código que puede ser usado una y otra vez. Puede ser creado por usted o importado desde algún módulo. Ejemplos de funciones:

- De cálculo matemático

  `log, sen, cos, tan, exp, hypot`.

- Funciones que generan números al azar, funciones de ingreso, funciones que hacen cambios sobre un *string*.

- Código hecho por el usuario que puede ser reciclado.

Hay un grupo de funciones que vienen hechas, es decir, listas para usar. Para encontrar qué funciones están disponibles tenemos la documentación del Python y un sitio web `http://www.python.org/doc/current/modindex.html`

Estas funciones pre-hechas vienen en grupos llamados módulos. Para importar en nuestro código el módulo apropiado, que contiene la función que nos interesa, usamos el comando

```
import modulo_name
```

Una vez importado el módulo, cuando queremos llamar a la función para usarla, debemos dar el comando

```
modulo_name.function(arguments)
```

Veamos un ejemplo com la función `hypot` del módulo matemático

```
import math
math.hypot(8,9)
```

Si analizamos las líneas anteriores de código debemos decir que el módulo que contiene las funciones matemáticas se llama `math` y éste incluye la función `hypot` que devuelve el largo de la hipotenusa. El símbolo `.` separa el nombre del módulo del de la función. Por supuesto `hypot` es el nombre de la función y `()` es el lugar para los argumentos. Una función podría no tener argumentos, pero aún así deben ir los paréntesis, son obligatorios. Los números `8,9` son enviados a la función para que los procese. En el ejemplo, estos números corresponden a los dos catetos de un triángulo rectángulo.

En las secciones anteriores vimos funciones especializadas en el ingreso de *strings* y de números. Nos referimos a `input()` para números y a `raw_input()` para *strings*. En este caso, `input` e `raw_input` corresponden al nombre de las funciones, y entre los paréntesis se acepta un *string* como argumento, el cual es desplegado como *prompt* cuando se da el comando. Como vimos, este argumento es opcional en ambas funciones, sin embargo, lo incluyan o no, siempre se deben poner los paréntesis.

Funciones como `input()` y `raw_input()` están incorporadas al lenguaje y no necesitamos importar ningún módulo para usarlas.

2.4.1. Algunas funciones incorporadas.

- `float(obj)` Convierte un *string* u otro número a un número de punto flotante. Con decimales.
- `int(obj)` Convierte un *string* u otro número a un número entero. Sin decimales.
- `long(obj)` Convierte un *string* u otro número a un número entero largo. Sin decimales.
- `str(num)` Convierte un número a un *string*.
- `divmod(x,y)` Devuelve los resultados de `x/y` y `x%y`.
- `len(s)` Retorna el largo de un *string* u otro tipo de dato (una lista o diccionario).
- `pow(x,y)` Retorna `x` a la potencia `y`.
- `range(start,stop,step)` Retorna un conjunto de números desde `start` hasta `stop`, con un paso igual a `step`.
- `round(x,n)` Retorna el valor del punto flotante `x` redondeado a `n` digitos después del punto decimal. Si `n` es omitido el valor por defecto es cero.

2.4.2. Algunas funciones del módulo `math`.

- `acos(x), asin(x), atan(x)` El arcocoseno, el arcoseno y la arcotangente de un número.
- `cos(x), sin(x), tan(x)` El coseno, el seno y la tangente de un número.
- `log(x), log10(x)` El logaritmo natural y el logaritmo en base 10 de un número.
- `pow(x,y)` Retorna `x` a la potencia `y`.
- `hypot(x,y)` Retorna el largo de la hipotenusa de un triángulo rectángulo de catetos `x` e `y`.

2.4.3. Algunas funciones del módulo `string`.

- `capitalize(string)` Pone en mayúscula la primera letra de la primera palabra.
- `capwords(string)` Pone en mayúscula la primera letra de todas las palabras.
- `lower(string)` Todas las letras en minúsculas.
- `upper(string)` Todas las letras en mayúsculas.
- `replace(string,old,new)` reemplaza todas las palabras `old` en `string` por las palabras `new`.
- `center(string, width)` Centra el `string` en un campo de un ancho dado por `width`.

- `rjust(string, width)` Justifica a la derecha el `string` en un campo de un ancho dado por `width`.
- `ljust(string, width)` Justifica a la izquierda el `string` en un campo de un ancho dado por `width`.

2.4.4. Algunas funciones del módulo `random`.

- `randrange(start, stop, step)` Da un número al azar entre el número `start` y el número `stop`. El número `step` es opcional.
- `choice(sequence)` Elige al azar un objeto que pertenece a la sequencia `sequence` (una lista). Por ejemplo `sequence=["a", "b", "c", "d", "e"]`.

2.4.5. Algunos otros módulos y funciones.

Una función del módulo `time`:

- `sleep(x)` El computador queda en pausa por `x` segundos.

Un par de funciones del módulo `calendar`:

- `prcal(year)` Imprime un calendario para el año `year`.
- `prmoth(year, month)` Imprime un calendario para el mes `month` del año `year`.

2.5. Funciones hechas en casa.

Una función define un conjunto de instrucciones. A menudo son almacenadas en conjuntos llamados módulos. Pueden o no necesitar argumentos. Pueden o no retornar un valor al programa.

2.5.1. Receta para una función.

Para crear una función primero hay que definir la función, darle un nombre, y escribir el conjunto de instrucciones que la constituyen. La función realizará las instrucciones cuando es llamada. Después, en el programa, llame la función que ya definió. A continuación veamos la definición formal de una función hecha por nosotros

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

Comenzamos con la palabra `def`, la cual es un palabra requerida. Debe ir en minúsculas. Luego `nombre` es el nombre que uno le da a la función. Después vienes los argumentos (`argumentos`) que corresponden a las variables que se le pasan a la función para que las utilice. Finalmente, `:`, requeridos al final de la línea que define una función. El bloque de `commandos` asociados a la función deben tener sangría para identificarlos como parte de la misma. A continuación un ejemplo concreto:

```
# Definiendo la funcion
def mi_function():
    print "Nos gusta mucho la Fisica"

# Usando la funcion
mi_function()
```

La definición de una función puede ser en cualquier parte del programa con la salvedad que debe ser antes de que la función misma sea llamada. Una vez definida la función ellas se ejecutarán cuando sean llamadas. Cuando enviamos valores a nuestras funciones se crean las variables nombradas en la definición. Por ejemplo:

```
def mi_function(nombre1, nombre2):
        print nombre1+nombre2
```

Los nombres de las variables sólo serán válidos dentro de la función (es decir, las variables son *locales*). Las funciones pueden usar sólo variables locales.

2.5.2. Pasando los valores.

Para enviar los valores a nuestra función ponga los valores en la llamada de la función. El tipo de los valores debe estar de acuerdo con lo que la función espera. Las funciones pueden tomar variables u otras funciones como argumentos. Veamos un ejemplo:

```
def mi_function(nombre1, nombre2):
    print nombre1,nombre2

mi_function("azul","rojo")
```

2.6. Condicionales.

Los condicionales prueban si una cierta çondición"es o no cierta. Por ejemplo, ¿el usuario tipeo la palabra correcta? o ¿El número es mayor que 10? El resultado de la condición decide que sucederá, por ejemplo, a todos los números mayores que 100 réstele 20, cuando la palabra ingresada sea la correcta, imprima "¡Bien!"

2.6.1. Posibles condicionales.

- `x == y`
  x es igual a y.
- `x != y`
  x no es igual a y.
- `x >y`
  x es mayor que y.

- `x <y`
  x es menor que y.

- `x >= y`
  x es mayor igual a y.

- `x <= y`
  x es menor igual a y.

A continuación, algunos ejemplos de los anteriores condicionales:

- `x == 125:`

- `passwd == "nix":`

- `num >= 0:`

- `letter >"L":`

- `num/2 == (num1-num):`

- `num%5 != 0:`

2.6.2. El `if`.

A continuación, estudiemos la instrucción `if`, partamos de la forma general de la instrucción:

```
if condition:
   statements
```

Primero la palabra clave `if`, luego la condición `condition`, que puede ser algo como `x<y` o `x==y`, etc. La línea termina con `:` requerido al final de una línea por la sintaxis del `if`. En las líneas siguientes `statements`, viene las instrucciones a seguir si la condición es cierta. Estas instrucciones deben ir con sangría (*indent*).

Un ejemplo de una construcción `if` simple.

```
num = input("Entre su edad")
if num >= 30:
       old-person(num)
       print
       print "Gracias"
```

2.6.3. El `if...else`.

La forma general de la construcción `if...else` a continuación:

```
if condition:
   statements_1
else:
   statements_2
```

El `else` debe de estar después de una prueba condicional. Sólo se ejecutará cuando condición evaluada en el `if` sea falsa. Use esta construcción cuando tenga dos conjuntos diferentes de instrucciones a realizar dependiendo de la condición. Un ejemplo

```
if x%2 == 0:
   print "el numero es par"
else:
   print "el numero es impar"
```

2.6.4. El `if...elif...else`.

La forma general de la construcción `if...elif...else`, a continuación:

```
if condition_1:
   statements_1
elif condition_2:
   statements_2
else:
   statements_3
```

Para más de dos opciones use la construcción con `elif`. `elif` es la forma acortada de las palabras *else if*. Las instrucciones asociadas a la opción `else` se ejecutarán si todas las otras fallan. Un ejemplo concreto

```
if x<0 :
    print x," es negativo"
elif x==0 :
    print x," es cero"
elif x>0 :
    print x," es positivo"
else:
    print x," Error, no es un numero"
```

2.6.5. La palabra clave `pass`.

El comando `pass` no hace realiza acción alguna. Un ejemplo

```
if x<0:
   HagaAlgo()
else:
   pass
```

2.6.6. La palabra clave `return`.

El comando `return` que sale de una función. Un ejemplo

```
import math

def raiz(num):
    if num<0:
         print "Ingrese un numero positivo"
         return
    print math.sqrt(n)
```

Los condicionales como el `if` son especialmente útil para atrapar y manejar errores. Use el `else` para atrapar el error cuando la condición no es satisfecha.

Los `if` pueden ser anidados. Sea cuidadoso, ya que la anidación puede producir confusión y debería ser usada con moderación y mesura.

2.6.7. Reciclando variables.

Una vez que una variable es creada su valor puede ser reasignado, veamos un ejemplo donde la variable `card_value` es reutilizada

```
card_value=card1+card2
if hit==yes:
   card_value=card1+card2+card3
else:
   pass
print card_value
```

2.7. Recursión.

Se llama recursión cuando una función se llama a si misma. La recursión permite repetir el uso de una función incluso dentro de la misma función. Un ejemplo es

```
def count(x):
   x=x+1
   print x
   count(x)
```

Si la función nunca para, esta recursión es llamada recursión infinita. Para prevenir este situación creamos un caso base. El caso base es la condición que causará que la función pare de llamarse a si misma. Un ejemplo

```
def count(x):
   if x<100:
      x=x+1
```

```
      print x
      count(x)
   else:
      return
```

2.8. Funciones que tiene un valor de retorno.

Podemos crear funciones que retornen un valor al programa que las llamo. Por ejemplo

```
def sumalos(x,y):
   new = x+y
   return new

# Llamada a la funcion
sum = sumalos(5,6)
```

2.9. Operadores lógicos.

Los condicionales pueden ser unidos usando las palabras reservadas `and`, `or` o `not`. Si ocupamos un `and` para unir dos condiciones lógicas tenemos que ambas condiciones deben satisfacerse para que el condicional sea cierto. En el caso de ocupar `or` para unir dos condiciones lógicas una de ellas debe ser satisfecha para que el condicional sea cierto. Finalmente el `not` se antepone a una condición y la niega, es decir, será cierto si la condición no es satisfecha. En todos los caso se aplica que cierto==1 y falso==0 (o != 1).

A continuación, algunos ejemplos de operadores lógicos:

- `if x>0 and x<10:`
- `if y>0 and x>0:`
- `if pwd=="code"  or pwd=="monster":`
- `if y>0 or x<0:`
- `if not(x<y):`
- `if x>y or not(x<0):`

2.9.1. Forma alternativa.

Cuando pruebe valores para < o >, estas pruebas pueden ser escritas como un sólo condicional sin usar el `and`. Veamos ejemplos

```
if 0<x<100:

if 1000>=x >=0:
```

2.9.2. Desarrollando programas.

Para desarrollar sus primeros programas parta escribiendo en sus propias palabras lo que el programa debería hacer. Convierta esta descripción en una serie de pasos en sus propias palabras. Para cada uno de los pasos propuestos traduzca sus palabras en código en Python. Dentro del código incluya instrucciones que impriman el valor de las variables para probar que ellas están haciendo lo que usted esperaba. Ejemplo

```
print "Numero = ", num
```

2.10. Iteraciones con `while`.

La palabra reservada `while` puede ser usada para crear una iteración. La instrucción `while` necesita un contador que se incremente. Ejemplo

```
while x < 10:
     print x
     x = x+1
```

Para hacer una sección de código reusable, en vez de usar valores contantes use variables. Primero un ejemplo no generalizado

```
while x < 12:
     print 2*x
     x = x+1
```

Ahora el mismo ejemplo generalizado

```
while x < max_num:
     print num*x
     x = x+1
```

Podemos hacer tablas usando el código de escape del tabulador (`\t`) en un *string*. Los tabuladores mantienen los items alineados dando una salida ordenada. Ejemplo

```
while x < 10:
     print item1, "\t", item2
     x = x+1
```

2.11. Los *strings*.

Los *strings* son hechos de pequeñas unidades, cada caracter individual. Cada uno de los caracteres tiene una dirección numérica dentro del *string*, donde el primer caracter tiene la dirección cero (0). Cada caracter individual, o conjunto de caracteres, en un *string* puede ser acccesado usando sus direcciones numéricas. Use `[ ]` para accesar caracteres dentro de un *string*. Veamos un ejemplo

```
palabra = "computador"
letra = palabra[0]
```

Para acceder un conjunto de caracteres dentro de un *string* lo podemos hacer como sigue:

- Use `[#:#]` para obtener un conjunto de letras.
  `parte = palabra[1:3]`
- Para tomar desde el comienzo a un punto dado en el *string*.
  `parte = palabra[:4]`
- Para tomar desde un punto dado al final del *string*.
  `parte = palabra[3:]`

2.11.1. Índice negativos.

Veamos que pasa cuando usamos índices negativos

```
>>> a="hola"
>>> a[0]
'h'
>>> a[-1]
'a'
>>> a[-2]
'l'
>>> a[-3]
'o'
>>> a[-4]
'h'
>>> a[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

2.11.2. ¿Cuán largo es un *string*?

Para encontrar cuántos caracteres tiene un *string* usamos la función `len(string)`. La función `len` requiere un *string* como argumento. Un ejemplo:

```
palabra = "computador"
largo = len(palabra)
```

2.11.3. Recorriendo un *string*.

Uno puede desear hacer una prueba sobre cada una de las letras que componen el *string* todas de una vez. Hay dos maneras de hacer esto usando una instrucción `while` o una instrucción `for` para realizar el ciclo o *loop*. Primero veamos el ciclo con `while`:

```
palabra = "computador"
indice = 0
while indice < len(palabra):
      letra = palabra[indice]
      print letra
      indice=indice +1
```

2.11.4. El ciclo `for`.

Una manera más compacta de escribir el ciclo `while` anterior es usando un ciclo `for`, veamos cómo queda el código

```
palabra = "computador"
for letra in palabra:
      print letra
```

Notemos que hemos creado la variable `letra` cuando creamos el ciclo `for`. A continuación un ejemplo más completo del ciclo `for`:

```
#!/usr/bin/env python
# -*- coding: iso-8859-1 -*-
# Programa que cuenta vocales

import string

palabra = raw_input("Entre una palabra : ")
palabra_min = string.lower(palabra)
vocales="aeiouáéíóú"
contador = 0
for letra in palabra_min:
    if letra in vocales:
       contador=contador +1
    else:
       pass
print "El número de vocales en la palabra que ingresó fueron : ", contador
```

Notemos la segunda línea de este programa que nos permite ingresar e imprimir *strings* con caracteres acentuados y caracteres especiales.

2.11.5. Comparando *strings*.

Los *strings* pueden ser usados en comparaciones. Ejemplo

```
if palabra < "cebu":
   print palabra
```

De acuerdo a Python, todas las letras mayúsculas son mayores que las letras minúsculas. Así Z>a.

Una buena idea es convertir todos los *strings* a mayúscula o minúscula, según sea el caso, antes de hacer comparaciones. Recordemos que el módulo `string` contiene varias funciones útiles incluyendo: `lower(string)`, `upper(string)` y `replace(string,string,string)`. Revise la documentación.

2.12. Listas.

Una lista es un conjunto ordenado de elementos. Las listas están encerradas entre paréntesis `[ ]`. Cada item en una lista está separado por una coma. Veamos ejemplos de listas

```
mascotas = ["perros", "gatos", "canarios", "elefantes"]
numeros = [1,2,3,4,5,6]
cosas = [ 1, 15, "gorila", 23.9, "alfabeto"]
```

Un elemento de una lista puede ser otra lista. Una lista dentro de otra lista es llamada *lista anidada*. A continuación un ejemplo de listas anidadas

```
para_hacer = ["limpiar", ["comida perro", "comida gato","comida pez"], "cena"]
```

2.12.1. Rebanando listas.

Una lista puede ser accesada al igual que un *string* usando el operador `[ ]`. Ejemplo

```
lista=["Pedro", "Andres", "Jaime", "Juan"]
print lista[0]
>>> Pedro
print lista[1:]
>>> Andres Jaime Juan
```

Para accesar un item en una lista anidada hay que proveer dos índices. Ejemplo

```
lista_palabras = ["perro", ["fluffy", "mancha", "toto"], "gato"]
print lista_palabras[1][2]
>>> toto
```

2.12.2. Mutabilidad.

A diferencia de los *strings* las listas son mutables, lo que significa que se pueden cambiar. Ejemplo

```
string = "perro"
string [2] = "d" # Esta NO es un instruccion VALIDA
```

En cambio en una lista

```
lista = ["p", "e", "r", "r", "o"]
lista [2] = "d"
>>> ["p", "e", "d", "r", "o"]
```

Como se muestra en la comparación anterior una lista puede ser cambiada usando el operador `[ ]`. Ejemplo

```
lista=["Pedro", "Andres", "Jaime", "Juan"]
lista[0]="Matias"
print lista
>>> Matias Andres Jaime Juan
```

2.12.3. Agregando a una lista.

Para agregar items al final de una lista use `list.append(item)`. Ejemplo

```
lista=["Pedro", "Andres", "Jaime", "Juan"]
lista.append("Matias")
print lista
>>> Pedro Andres Jaime Juan Matias
```

2.12.4. Operaciones con listas.

las listas se pueden sumar resultando una solo lista que incluye ambas lista iniciales. Además, podemos multiplicar una lista por un entero n obteniendo una lista con n replicas de la lista inicial. Veamos ejemplos de ambas operaciones

```
lista1=["Pedro", "Andres", "Jaime", "Juan"]
lista2=["gato", 2]
lista1+lista2
['Pedro', 'Andres', 'Jaime', 'Juan', 'gato', 2]
lista2*2
['gato', 2, 'gato', 2]
2*lista2
['gato', 2, 'gato', 2]
```

2.12.5. Borrando items de una lista.

Use el comando `del` para remover items basado en el índice de posición. Ejemplo

```
lista=["Pedro", "Andres", "Jaime", "Juan"]
del lista[1]
print lista
>>> Pedro Jaime Juan
```

Para remover items desde una lista sin usar el índicde de posición, use el siguiente comando `nombre_lista.remove("item")`. Un ejemplo

```
jovenes = ["Pancho", "Sole", "Jimmy"]
jovenes.remove("Jimmy")
print jovenes
>>> Pancho Sole
```

2.12.6. ¿Qué contiene una lista?

Con la palabra reservada `in` podemos preguntar si un item está en la lista, veamos un ejemplo

```
lista = ["rojo", "naranjo", "verde", "azul"]
if "rojo" in lista:
    haga_algo()
```

La palabra clave `not` puede ser combinada con `in` para hacer la pregunta contraria, es decir, si un item no está en un lista. Veamos un ejemplo

```
lista = ["rojo", "naranjo", "verde", "azul"]
if "purpura" not in lista:
    haga_algo()
```

2.12.7. Un ciclo `for` y las listas.

Los ciclos `for` pueden ser usados con listas de la misma manera que lo eran con *strings*, un ejemplo para mostrarlo

```
email = ["oto@mail.com", "ana@mail.com"]
for item in email:
    envie_mail(item)
```

2.12.8. Otros trucos con listas.

- `len(list_name)` Da el largo de la lista, su número de elementos.
- `list_name.sort()` Pone la lista en orden alfabético y numérico.
- `random.choice(list_name)` Escoge un elemento al azar de la lista.
- `string.split(list_name)` Convierte un *string*, como una frase, en una lista de palabras.
- `string.join(list_name)` Convierte una lista de palabras en una frase dentro de un *string*.

2.12.9. Generando listas de números.

La función `range(num_init, num_fin, num_paso)` toma tres argumentos enteros, el número de partida, el numero final y el paso, para generar una lista de enteros que comienza en el número de partida, termina con un número menor que el final saltandose el paso señalado, si se omite el paso el salto será de uno en uno. Veamos ejemplos

```
range(10) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(2,10) = [2, 3, 4, 5, 6, 7, 8, 9]
range(0,11,2) = [0, 2, 4, 6, 8, 10]
```

2.13. Tuplas.

Una tupla es una lista inmutable. Una tupla no puede modificarse de ningún modo después de su creación.

```
>>> t = ("a", "b", 8)
>>> t[0]
'a'
```

Una tupla se define del mismo modo que una lista, salvo que el conmjunto se encierra entre paréntesis `( )`, en lugar de entre corchetes `[ ]`. Los elementos de la tupla tienen un orden definido, como los de la lista. Las tuplas tienen primer índice 0, como las listas, de modo que el primer elemento de una tupla no vacía es siempre t[0]. Los índices negativos cuentan desde el final de la tupla, como en las listas. Las porciones funcionan como en las listas. Advierta que al extraer una porción de una lista, se obtiene una nueva lista; al extraerla de una tupla, se obtiene una nueva tupla. No hay métodos asociados a tuplas (tal como `append()` en una lista).

No pueden añadirse elementos a una tupla, no pueden eliminarse elementos de una tupla, no pueden buscarse elementos en una tupla, se puede usar `in` para ver si un elelmento existe en la tupla.

Las tuplas son más rápidas que las listas. Si está definiendo un conjunto constante de valores y todo lo que va ha hacer con él es recorrerla, utilice una tupla en lugar de una lista. Una tupla puede utilizarse como clave en un diccionario, pero las listas no. Las tuplas pueden convertirse en listas y viceversa. La función incorporada `tuple(lista)` toma una lista y devuelve una tupla con los mismos elementos. La función list toma una tupla y devuelve una lista.

2.13.1. El comando break.

El comando `break` es capaz de salirse de un ciclo `for o \verb`while—. Un ciclo puese tener una sección `else` esta es ejecutada cuando el ciclo termina por haber agotado la lista en un ciclo `for` o cuando la comparación llega a ser falsa en un ciclo `while`, pero no cuando el ciclo es ternimado con `break`. A continuación, un programa que muestra este hecho y sirve para encontrar números primos

```
for n in range(2,10):
    for x in range(2,n):
        if n % x ==0:
            print n, "igual a", x,"*", n/x
            break
    else:
        print n,"es un numero primo"
```

2.14. Trabajando con archivos.

El lenguaje Python puede ser usado para crear programas que manipulan archivos sobre un sistema de archivos en un computador. El módulo `os` contiene las funciones necesarias para buscar, listar, renombrar y borrar archivos. El módulo `os.path` contiene unas pocas funciones especializadas para manipular archivos. Las funciones necesarias para abrir, leer y escribir archivos son funciones intrínsecas de Python.

2.14.1. Funciones del módulo `os`.

Funciones que sólo dan una mirada.

- `getcwd()` Retorna el nombre el directorio actual.
- `listdir(path)` Retorna una lista de todos los archivos en un directorio.
- `chdir(path)` Cambia de directorio. Mueve el foco a un directorio diferente.

Función que ejecuta un comando del sistema operativo.

- `system('comando')` Ejecuta el `comando`

Funciones que agregan.

- `mkdir(path)` Hace un nuevo directorio con el nombre dado.
- `mkdirs(path)` Hace un subdirectorio y todos los directorios del `path` requeridos.

Funciones que borran o remueven.

- `remove(path)` Borra un archivo.
- `rmdir(path)` Borra un directorio vacío.
- `removedirs(path)` Borra un directorio y todo dentro de él.

Funciones que cambian.

- `rename(viejo,nuevo)` Cambia el nombre de un archivo de `viejo` a `nuevo`
- `renames(viejo,nuevo)` Cambia el nombre de un archivo de `viejo` a `nuevo` cambiando los nombres de los directorios cuando es necesario.

2.14.2. Funciones del módulo `os.path`.

Funciones que verifican.

- `exists(file)` Retorna un booleano si el archivo `file` existe.
- `isdir(path)` Retorna un booleano si el `path` es un directorio.
- `isfile(file)` Retorna un booleano si el `file` es un archivo.

2.14.3. Ejemplo de un código.

Un programa que borra todo un directorio

```
import os, os.path
path = raw_input("Directorio a limpiar : ")
os.chdir(path)
files= os.listdir(path)
print files
for file in files:
    if os.path.isfile(file):
        os.remove(file)
        print "borrando", file
    elif os.path.isdir(file):
        os.removedirs(file)
        print "removiendo", file
    else:
        pass
```

2.14.4. Abriendo un archivo.

Para abrir un archivos debemos dar la instrucción `open(filename,mode)` donde `filename` es el nombre del archivo y el `mode` corresponde a una de tres letras `"r"` para lectura solamente del archivo, `"w"` para escritura y `"a"` para agregar al final del archivo. para poder manejar un archivo abierto hay que crear una variable con él. Ejemplo

```
openfile= open("archivo.txt","a")
```

2.14.5. Leyendo un archivo.

- `file.read()` Lee el archivo completo como un *string*.
- `file.readline()` Lee una línea en un *string*.
- `file.readlines()` Lee el archivo completo, cada línea llega a ser un item tipo *string* en una lista.

2.14.6. Escribiendo a un archivo.

- `file.write(string)` Escribe el `string` al archivo. Cómo se escribira este depende de en qué modo el archivo fue abierto.
- `file.writelines(list)` Escribe todos los items tipo *string* en la lista `list`. Cada elemento en la lista estará en la misma línea a menos que un elemento contenga una caracter de `newline`

Si queremos usar la instruccion `print` para escribir sobre un archivo abierto, digamos `salida`, podemos usar la instrucción

```
salida = open("datos.txt", "w")
print >> salida, datos  # Imprime datos en el archivo datos.txt
print >> salida         # Imprime una linea en blanco en el archivo datos.txt
salida.close()
```

2.14.7. Cerrando un archivo.

- `file.close()` Cierra un archivo previamente abierto.

2.14.8. Archivos temporales.

Las funciones en el módulo `tempfile` puede ser usadas para crear y manejar archivos temporales. La instrucción `tempfile.mkstemp()` devuelve una lista en el que el segundo item es un nombre al azar que no ha sido usado. Los archivos temporales estarán localizados en el directorio temporal por defecto.

2.14.9. Ejemplo de lectura escritura.

```
# Programa que reemplaza una palabra vieja por otra nueva

import string, tempfile, os

# Preguntarle al usuario por Informacion
filename = raw_input("Nombre del archivo: ")
find = raw_input("Busque por: ")
replace = raw_input("Reemplacelo por: ")
# Abra el archivo del usuario, lealo y cierrelo
file = open(filename, "r")
text = file.readlines()
file.close()
# Edite la informacion del archivo del usuario

nueva = []

for item in text:
```

```
    line = string.replace(item, find, replace)
    nueva.append(line)

# Cree un nuevo archivo temporal
newname=tempfile.mkstemp()
temp_filename=newname[1]
newfile = open(temp_filename, "w")
newfile.writelines(nueva)
newfile.close()

# Cambie los nombres de los archivos y borra los temporales
oldfile=filename+"~"
os.rename(filename, oldfile)
os.system(" cp "+temp_filename+" "+filename)
os.remove(temp_filename)
```

2.15. Excepciones.

Las palabras reservadas `try` y `except` pueden ser usadas para atrapar errores de ejecución en el código. Primero en el bloque `try` se ejecuta un grupo de instrucciones o alguna función. Si estas fallan o devuelven un error, el bloque de comandos encabezados por `except` se ejecutará. Puede ser usado para manejar programas que pueden fallar bajo alguna circunstancia de una forma muy elegante.

```
# Sample Try / Except
def open_file():
    filename = raw_input("Entre el nombre del archivo: ")
    try:
       file = open(filename)
    except:
       print "Archivo no encontrado."
       open_file()
    return file
```

2.16. Diccionarios.

Las listas son colecciones de items (*strings*, números o incluso otras listas). Cada item en la lista tiene un índice asignado.

```
lista = ["primero", "segundo", "tercero"]
print lista[0]
>>> primero
print lista[1]
>>> segundo
```

```
print lista[2]
>>> tercero
```

Para acceder a un valor de la lista uno debe saber su índice de posición. Si uno remueve un item desde la lista, el índice puede cambiar por el de otro item en la lista.

Un diccionario es una colección de items que tiene una llave y un valor. Ellos son parecidos a las listas, excepto que en vez de tener asignado un índice uno crea los índices.

```
lista = ["primero", "segundo", "tercero"]
diccionario = {0:"primero", 1:"segundo", 2:"tercero"}
```

Para crear un diccionario debemos encerrar los item entre paréntesis de llave `{}`. Debemos proveer una llave y un valor, un signo `:` se ubica entre la llave y el valor `(llave:valor)`. cada llave debe ser única. Cada par llave:valor está separado por una coma. Veamos un par de ejemplos con diccionarios

```
ingles = {'one':'uno', 'two':'dos'}
```

Uno en japonés

```
nihon_go = {}
nihon_go["ichi"] = "uno"
nihon_go["ni"] = "dos"
nihon_go["san"] = "tres"
print nihon_go
{ 'ichi':'uno', 'ni':'dos', 'san':'tres'}
```

Para acceder el valor de un item de un diccionario uno debe entrar la llave. Los diccionarios sólo trabajan en una dirección. Uno debe dar la llave y le devolverán el valor. Uno no puede dar el valor y que le devuelvan la llave. Ejemplo

```
nihon = { 'ichi':'uno', 'ni':'dos', 'san':'tres'}
print nihon['ichi']
uno
```

Notemos que este diccionario traduce del japonés al español pero no del español al japonés.

2.16.1. Editando un diccionario.

- Para cambiar un valor de un par, simplemente reasignelo
  `nihon["ichi"]=1`
- Para agregar un par valor:llave, entrelo
  `nihon["shi"]=cuatro`
- Para remover un par use del
  `del nihon["ichi"]`
- Para ver si una llave ya existe, use la función `has_key()`
  `nihon.has_key("ichi")`

- Para copiar el diccionario entero use la función o método copy().
  `japones= nihon.copy()`

Los diccionarios son mutables. Uno no tiene que reasignar el diccionario para hacer cambios en él.

Los diccionarios son útiles cada vez que usted tiene items que desea ligar juntos. También son útiles haciendo subtituciones (reemplace todos los x por y). Almacenando resultados para una inspección rápida. Haciendo menús para programas. Creando mini bases de datos de información.

2.16.2. Un ejemplo de código, un menú.

```
import string
def add(num1,num2):
    print num1+num2
def mult(num1,num2):
    print num1*num2

# Programa
num1 = input("Entre el primer numero: ")
num2 = input("Entre el segundo numero: ")
menu = {'S':add, 'M':mult}
print "[S] para sumar, [M] para multiplicar: "
choice = string.upper(raw_input())
menu[choice] (num1,num2)
```

2.16.3. Tuplas y diccionarios como argumentos.

Si un parámetro formal de la forma `**name` está presente, la función recibe un diccionario. Si un parámetro formal de la forma `*name` está presente, la función recibe una tupla. Los parámetros formales `*name` deben ir antes que los `**name`.

2.17. Modules y Shelve.

Algunos problemas que se presentan a medida que los códigos crecen son: con cientos de líneas es muy fácil perderse. Cuando se trabaja en equipo es difícil hacerlo con códigos muy grandes. Sería bueno poder separar el código en pequeños archivos independientes. Por otra parte, no podemos salvar estructuras de datos, es decir, si creamos un diccionario, sabemos como salvarlo como un archivo de texto, pero no podemos leerlo como un diccionario. Sería bueno poder salvar las listas como listas, los diccionarios como diccionarios y así luego poder leerlos e incorporarlos a nuestro código en forma fácil.

2.17.1. Partiendo el código.

El código puede ser dividido en archivos separados llamados `modules`. Ya hemos usado varios de estos módulos trabajando en Python (`string`, `math`, `random`, etc.). Para crear un módulo sólo tipee su código Python y sálvelo de la manera usual. Su módulo salvado puede ser importado y usado tal como cualquier otro módulo de Python.

2.17.2. Creando un módulo.

Lo principal para almacenar nuestro código en un módulo es escribir código reusable. El código debe ser principalmente funciones y clases. Evite tener variables o comandos fuera de la definición de funciones. Las funciones pueden requerir valores desde el programa quien las llama. Salve el código como un archivo regular `.py`. Luego en el programa principal, importe el módulo y úselo.

2.17.3. Agregando un nuevo directorio al *path*.

Cuando Python busca módulos sólo lo hace en ciertos directorios. La mayoría de los módulos que vienen con Pyhton son salvados en `/usr/lib/python`. Cuando salve sus propios módulos seguramente lo hará en un lugar diferente, luego es necesario agregar el nuevo directorio a `sys.path`. Hay que consignar que el directorio desde el cual se invoca Python, si está en el `path`. Para editar el `sys.path`, en el modo interactivo tipee

```
>>> import sys
>>> sys.path #imprime los actuales directorios en el path
>>> sys.path.append('/home/usuario/mis_modulos')
```

Dentro de un *script* usamos para importar mi módulos `mis_funciones` que está salvado en mi directorio de módulos

```
import sys
sys.path.append('/home/usuario/mis_modulos')

import mis_funciones
```

2.17.4. Haciendo los módulos fáciles de usar.

Los comentarios con triple comilla son usados para agregar documentación al código, ejemplo

```
def mi_funcion(x,y)
    """mi_funcion( primer nombre, ultimo nombre) """
```

Use al principio y al final triple comilla. El texto en triple comilla debería explicar lo que la función, clase o módulo hace. Para obtener estas líneas de documentación las podré ver en el modo interactivo si da el comando `help(module.mi_funcion)`.

2.17.5. Usando un módulo.

En el programa principal incluya el comando `import` y el nombre del módulo. Cuando llame a una función del módulo debe incluir el nombre del módulo `.` el nombre de la función, esta no es la única manera de importarlos, veamos unos ejemplo, primero la forma habitual:

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff
import stuff

print stuff.f(1,2)
print stuff.C(1).m(2)
```

una segunda forma

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff
from stuff import f, C

print f(1,2)
print C(1).m(2)
```

una última manera

```
# Sean f(x,y) una funcion y C una clase con un metodo m(x) del modulo stuff
import stuff as st

print st.f(1,2)
print st.C(1).m(2)
```

2.17.6. Trucos con módulos.

Un módulo puede ser corrido como programa independiente si incluímos las siguientes líneas en el final

```
if __name__ == '__main__':
   run_function()
```

Sustituya `run_function()` por el nombre de la función principal en el módulo.

2.17.7. Preservando la estructura de la información.

Existen dos métodos de preservar la data:

- El módulo `pickle` almacena una estructura de datos de Python en un archivo binario. Está limitado a sólo una estructura de datos por archivo.
- El módulo `shelve` almacena estructuras de datos de Python pero permite más de una estructura de datos y puede ser indexado por una llave.

2.17.8. ¿Cómo almacenar?

Importe el módulo `shelve`, abra un archivo *shelve*, asigne un item, por llave, al archivo *shelve*. Para traer la data de vuelta al programa, abra el archivo *shelve* y accese el item por llave. Los *shelve* trabajan como un diccionario, podemos agregar, accesar y borrar items usando sus llaves.

2.17.9. Ejemplo de *shelve*.

```
import shelve
colores = ["verde", "rojo", "azul"]
equipos = ["audax", "union", "lachile"]
shelf = shelve.open('mi_archivo')
# Almacenendo items
shelf['colores'] = colores
shelf['equipos'] = equipos
# trayendolos de vuelta
newlist = shelf['colores']
# Cerrando
shelf.close()
```

2.17.10. Otras funciones de *shelve*.

- Para tomar una lista de todas las llaves disponibles en un archivo *shelve*, use la función `keys()`:
  `lista = shelf.keys()`
- Para borrar un item, use la función `del`:
  `del shelf('ST')`
- Para ver si una llave existe, use la función `has_key()`:
  `if shelf.has_key('ST'): print "si"`

2.18. Clases y métodos.

Las clases son colecciones de data y funciones que pueden ser usadas una y otra vez. Para crear una clase parta con la palabra reservada `class`, luego necesita un nombre para la clase. Los nombres de las clases, por convención, tiene la primera letra en mayúscula. Después del nombre termine la línea con un `:` y entonces cree el cuerpo de la clase, las instrucciones que forman el cuerpo deben ir con sangría. En el cuerpo cree las definiciones de las funciones, cada función debe tomar `self` como parámetro.

```
class MiClase:
   def hola(self):
       print "Bienvenido. "
   def math(self,v1,v2):
```

```
       print v1+v2
```

Declarando y luego usando la clase

```
class MiClase:
   def hola(self):
       print "Bienvenido. "
   def math(self,v1,v2):
       print v1+v2

fred = MiClase()
fred.hola()
fred.math(2,4)
```

Creamos una instancia de una clase al asignarla a una variable, `fred = MiClase()`. Para aplicar una función o método a una nueva instancia debemos especificar en forma completa la instancia y el método `fred.hola()`. Si el método toma argumentos debemos estar seguro de incluir todos los valores necesitados, por ejemplo `fred.math(2,4)`.

2.18.1. Clase de muestra LibretaNotas.

```
class LibretaNotas:
   def __init__(self, name, value):
        self.nombre = name
        self.puntaje = value
        self.evaluaciones = 1
   def sumanota(self, puntos):
        self.evaluaciones += 1
        self.puntaje += puntos
        self.promedio = self.puntaje/float(self.evaluaciones)
   def promedio(self)
        print self.nombre, ": promedio =", self.promedio
```

Las variables dentro de una clase deben ser accesadas poniendo el prefijo a su nombre la palabra `self`.

La función `__init__` es la que correrá si una nueva instancia de la clase es creada. Esta función es especial y se conoce como el constructor de la clase.

Usando la clase `LibretaNotas`

```
eli   = LibretaNota('Elizabeth', 6.5)
mario =  LibretaNota('Mario', 6.0)
carmen =  LibretaNota('Carmen', 6.1)

eli.sumanota(6.2)
mario.sumanota(6.1)
carmen.sumanota(6.3)
eli.sumanota(6.8)
```

```
mario.sumanota(6.7)
carmen.sumanota(6.6)

eli.promedio()
mario.promedio()
carmen.promedio()
```

Cada nueva instancia de la clase `LibretaNotas` debe tener un nombre y una primera nota porque así lo requiere la función constructor `__init__`. Notemos que cada instancia tiene su propio promedio.

2.18.2. Valores por defecto.

Una función puede tener valores por defecto, estos valores son usados sólo cuando la función es llamada sin especificar un valor. Ejemplo

```
def potencia(voltaje, corriente=0):
        return voltaje*corriente
# Podemos llamarla
potencia(220)
potencia(voltaje=110, corriente=5)
```

Veamos un ejemplo de valores por defecto en una clase

```
class Dinero:
    def __init__(self, amount = 0) :
           self.amount=amount
    def print(self):
           print "Tienes", self.amount, "de dinero"
# Llamadas posibles

mi_dinero = Dinero(100)
tu_dinero = Dinero()
mi_dinero.print()
tu_dinero.print()
```

2.19. Sobrecarga de Operadores.

Las clases pueden redefinir comandos regulares de Python. Cambiar como los comandos regulares trabajan en una clase se llama sobrecarga de operadores. Para definir la sobrecarga, debemos usar nombres especiales cuando definimos los métodos. Las definiciones sobrecargadas deben retornar un valor. Veamos un ejemplo de sobrecarga

```
class Loca:
    def __init__(self, num1=2):
        self.num1=num1
```

```
    def __del__(self):
        print "%d Destruido!" % self.num1
    def __repr__(self):
        return "Su numero es %d" % self.num1
    def __add__(self, newnum):
        return newnum*self.num1
```

Notemos que tenemos un método nuevo el destructor que el la función que se invoca cuando una variable se destruye. A continuación, una lista de nombre especiales para sobrecargar los operadores

- `__add__` La suma $+$.
- `__sub__` La resta $-$.
- `__mul__` La multiplicación $*$.
- `__div__` La división $/$.
- `__pow__` La exponenciación $**$.
- `__repr__` El comando print.
- `__len__` El comando `len(x)`.
- `__call__` El comando de llamada `x()`.
- `__init__` El método constructor.
- `__del__` El método destructor.

2.19.1. Función *driver*.

Una función *driver* es puesta al final de la clase para probarla. La función toma la forma

```
if __name__ == '__main__':
   haga algo
```

Cuando una clase o módulo es llamado como programa principal, correrá este código.

2.19.2. Atributos de las clases.

La información que necesita acceder una clase en cada una de sus instancias u objetos puede ser almacenada en atributos de la clase. Los valores serán creados fuera de las definiciones de las funciones de la clase. La data será accesada dentro de las definición de las funciones o métodos de la clase usando la notación `NombredelaClase.NombredelaVariable`. Un ejemplo

```
class Cuentas:
    alumnos=[]
    def __init__(self,nombre):
        self.nombre=nombre
        Cuentas.alumnos.append(self.nombre)
    def __del__(self)
        Cuentas.alumnos.remove(self.nombre)
```

Notese que la lista de alumnos es siempre llamada por su nombre completo `Cuentas.alumnos`. Para acceder a la lista fuera de la clase, use su nombre completo `Cuentas.alumnos`.

2.19.3. Ejemplo de clase vectores.

Escribamso un módulo `vec2d.py` con una clase de vectores bidimensionales sobrecargando la suma, la resta, el producto, la impresión, entre otros métodos

```
from math import sqrt

class Vec2d:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def module(self):
        return sqrt(self.x**2+self.y**2)
    def __repr__(self):
        return "(%11.5f,%11.5f)" % (self.x,self.y)
    def __add__(self,newvec):
        return Vec2d(self.x+newvec.x,self.y+newvec.y)
    def __sub__(self,newvec):
        return Vec2d(self.x-newvec.x,self.y-newvec.y)
    def __mul__(self,newvec):
        return self.x*newvec.x+self.y*newvec.y
```

Ahora un programa que ocupa el módulo

```
#!/usr/bin/env python

from vec2d import *

a=Vec2d(1.3278,2.67)
b=Vec2d(3.1,4.2)

print a, b
print a+b
print a-b
print a*b
print a.module(),b.module()
```

2.20. Algunos módulos interesantes.

Hay muchos módulos que le pueden ser útiles, aquí le sugerimos unos pocos particularmente importantes.

2.20.1. El módulo `Numeric`.

Extensión numérica de Python que agrega poderosos arreglos multidimensionales.

```
>>> import Numeric as num
>>> a = num.zeros((3,2), num.Float)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> a[1]=1
>>> a
array([[ 0.,  0.],
       [ 1.,  1.],
       [ 0.,  0.]])
>>> a[0][1]=3
>>> a
array([[ 0.,  3.],
       [ 1.,  1.],
       [ 0.,  0.]])
>>> a.shape
(3, 2)
```

2.20.2. El módulo `Tkinter`.

Un módulo para escribir aplicaciones gráficas portables con Phyton y Tk.

```
#!/usr/bin/env python
from Tkinter import *

root = Tk()
root.title("Mi ventana")
btn = Button(root, text="Salir")
btn.grid()
def stop(event):
    root.destroy()
btn.bind('<Button-1>', stop)
root.mainloop()
```

2.20.3. El módulo `Visual`.

Un módulo que permite crear y manipular objetos 3D en un espacio 3D.

Capítulo 3

Una breve introducción a C++.

versión 7.30, 02 de Noviembre del 2006

En este capítulo se intentará dar los elementos básicos del lenguaje de programación C++. No se pretende más que satisfacer las mínimas necesidades del curso, sirviendo como un ayuda de memoria de los tópicos abordados, para futura referencia. Se debe consignar que no se consideran todas las posibilidades del lenguaje y las explicaciones están reducidas al mínimo.

3.1. Estructura básica de un programa en C++.

3.1.1. El programa más simple.

El primer ejemplo de todo manual es el que permite escribir "Hola"en la pantalla.

```
//
// Los comentarios comienzan con //
//
#include <iostream>
using namespace std;

int main()
{
    cout << "Hola." << endl;
    return 0 ;
}
```

Las tres primeras líneas corresponden a comentarios, todo lo que está a la derecha de los caracteres `//` son comentarios y no serán considerados en la compilación. En la línea siguiente se incluye un archivo de cabecera, o *header*, con la instrucción de preprocesador `#include`. El nombre del archivo se puede escribir como `<nombre>` o bien `"nombre.h"`. En el primer caso el archivo `nombre` será buscado en el *path* por defecto para los `include`, típicamente `/usr/include` o `/usr/include/c++/3.x/` en el caso de *headers* propios de C++; en el segundo caso la búsqueda se hace en el directorio local. También podríamos incluir un *path* completo cuando se ocupan las comillas. En nuestro ejemplo se incluye el archivo `iostream`, en el cual se hacen las definiciones adecuadas para el manejo de la entrada y salida en C++. Este archivo es necesario para enviar luego un mensaje a pantalla.

La función `int main` es donde comienza a ejecutarse el programa; siempre debe haber una función `main` en nuestro programa. Debido a imposiciones del sistema operativo la función `main` devuelve un entero y por tanto debe ser declarada `int`. Los paréntesis vacíos `()` indican que el `main` no tiene argumentos de entrada (más adelante se verá que puede tenerlos). Lo que está encerrado entre llaves `{}` corresponde al cuerpo de la función `main`. Cada una de las líneas termina con el carácter `;`. El identificador predefinido `cout` representa la salida a pantalla. El operador `<<` permite que lo que está a su derecha se le dé salida por el dispositivo que está a su izquierda, en este caso `cout`. Si se quiere enviar más de un objeto al dispositivo que está al inicio de la línea agregamos otro operador `<<`, y en este caso lo que está a la derecha del operador se agregará a lo que está a la izquierda y todo junto será enviado al dispositivo. En nuestro caso se ha enviado `endl`, un objeto predefinido en el archivo `iostream` que corresponde a un cambio de línea, el cual será agregado al final del mensaje. La línea final contiene la instrucción de retorno del entero cero, `return 0`.

Si escribimos nuestro primer programa en el editor `xemacs` con el nombre de `primero.cc` las instrucciones para editarlo, compilarlo y correrlo serán:

```
jrogan@pucon:~/tmp$ xemacs primero.cc
jrogan@pucon:~/tmp$ g++ -Wall -o primero primero.cc
jrogan@pucon:~/tmp$ ./primero
Hola.
jrogan@pucon:~/tmp$
```

Luego de la compilación, un archivo ejecutable llamado `primero` es creado en el directorio actual. Si el directorio actual no está en el `PATH`, nuestro programa debe ser ejecutado anteponiendo `./`. Si está en el `PATH`, para ejecutarlo basta escribir `primero`. (Para agregar el directorio local al `PATH` basta editar el archivo `~/.bashrc` agregarle una línea como `PATH="${PATH}:."` y ejecutar en la línea de comando `source ~/.bashrc` para que los cambios tengan efecto.)

3.1.2. Definición de funciones.

Las funciones en C++ son muy importantes, pues permiten aislar parte del código en una entidad separada. Esto es un primer paso a la *modularización* de nuestro programa, es decir, a la posibilidad de escribirlo en partes que puedan ser editadas de modo lo más independiente posible. Ello facilita enormemente la creación de código complicado, pues simplifica su modificación y la localización de errores. Nos encontraremos frecuentemente con este concepto.

Aprovecharemos de introducir las funciones modificando el primer programa de manera que se delegue la impresión del mensaje anterior a una función independiente:

```
//
// Segunda version incluye funcion adicional
//
#include <iostream>
using namespace std;
```

```
void PrintHola()
{
    cout << "Hola." << endl;
}

int main()
{
    PrintHola();
    return 0;
}
```

La función debe estar definida antes de que sea ocupada, por eso va primero en el código fuente. Como ya se dijo antes, la ejecución del programa comienza en la función `main` a pesar de que no está primera en el código fuente. Los paréntesis vacíos indican que la función `PrintHola` no tiene argumentos y la palabra delante del nombre de la función indica el tipo de dato que devuelve. En nuestro caso la palabra `void` indica que no devuelve nada a la función `main`.

Una alternativa al código anterior es la siguiente:

```
#include <iostream>
using namespace std;

void PrintHola();

int main()
{
    PrintHola();
    return 0 ;
}

void PrintHola()
{
    cout << "Hola." << endl;
}
```

En esta versión se ha separado la *declaración* de la función de su *implementación*. En la declaración se establece el nombre de la función, los argumentos que recibe, y el tipo de variable que entrega como resultado. En la implementación se da explícitamente el código que corresponde a la función. Habíamos dicho que una función debe estar definida antes que sea ocupada. En verdad, basta con que la función esté declarada. La implementación puede ir después (como en el ejemplo anterior), o incluso en un archivo distinto, como veremos más adelante. La separación de declaración e implementación es otro paso hacia la modularización de nuestro programa.

3.1.3. Nombres de variables.

Nuestros datos en los programas serán almacenados en objetos llamados variables. Para referirnos a ellas usamos un nombre que debe estar de acuerdo a las siguientes reglas:

- Deben comenzar con una letra (mayúsculas y minúsculas son distintas).
- Pueden contener números, pero no comenzar por uno.
- Pueden contener el símbolo `_` (*underscore*).
- Longitud arbitraria.
- No pueden corresponder a una de las palabras reservadas de C++[1]:

```
asm       delete  if        return    try
auto      do      inline    short     typedef
break     double  int       signed    union
case      else    long      sizeof    unsigned
catch     enum    new       static    virtual
char      extern  operator  struct    void
class     float   private   switch    volatile
const     for     protected template  while
continue  friend  public    this
default   goto    register  throw
```

3.1.4. Tipos de variables.

Todas las variables a usar deben ser *declaradas* de acuerdo a su tipo. Por ejemplo, si usamos una variable `i` que sea un número entero, debemos, antes de usarla, declararla, y sólo entonces podemos asignarle un valor:

```
int i;
i=10;
```

Esta necesidad de declarar cada variable a usar se relaciona con la característica de C++ de ser fuertemente "tipeado"[2]. Algunos de los errores más habituales en programación se deben al intento de asignar a variables valores que no corresponden a sus tipos originales. Si bien esto puede no ser muy grave en ciertos contextos, a medida que los programas se vuelven más complejos puede convertirse en un verdadero problema. El compilador de C++ es capaz de detectar los usos indebidos de las variables pues conoce sus tipos, y de este modo nuestro código se vuelve más seguro.

Es posible reunir las acciones de declaración e inicialización en una misma línea:

```
int i=10;
```

[1] A esta tabla hay que agregar algunas palabras adicionales, presentes en versiones más recientes de C++, como `namespace` y `using`

[2] Una traducción libre del término inglés *strongly typed*.

o declarar más de una variable del mismo tipo simultáneamente, e inicializar algunas en la misma línea:

```
int r1, r2, r3 = 10;
```

A veces se requiere que una variable no varíe una vez que se le asigna un valor. Por ejemplo, podríamos necesitar definir el valor de $\pi = 3.14159...$, y naturalmente no nos gustaría que, por un descuido, a esa variable se le asignara otro valor en alguna parte del programa. Para asegurarnos de que ello no ocurra, basta agregar el modificador `const` a la variable:

```
const float pi = 3.14159;
```

Para números reales se puede usar la notación exponencial. Por ejemplo, `1.5e-3` representa el número 1.5×10^{-3}.

Una variable puede ser declarada sólo una vez, pero naturalmente se le pueden asignar valores en un número arbitrario de ocasiones.

Los 15 tipos de datos aritméticos fundamentales disponibles son[3]:

Booleanos y caracteres

`bool`	Booleanas *true* o *false*.
`char`	Caracteres de 0 a 255 o -128 a 127, usa 8 bits,
`signed char`	Caracteres -128 a 127,
`unsigned char`	Caracteres de 0 a 255.

Enteros

`short`	Enteros entre $-2^{15} = -32768$ y $2^{15} - 1 = 32767$
`unsigned short`	Enteros entre 0 y 65535
`int`	Enteros entre $-2^{31} = -2147483648$ y $2^{31} - 1 = 2147483647$
`unsigned int`	Enteros entre 0 y $2^{32} = 4294967295$
`long` (32 bits)	Entero entre -2147483648 y 2147483647,
`long` (64 bits)	Entero entre -9223372036854775808 y 9223372036854775807,
`unsigned long` (32 bits)	Enteros entre 0 y 4294967295,
`unsigned long` (64 bits)	Enteros entre 0 y 18446744073709551615,
`long long`	Enteros entre -9223372036854775808 y 9223372036854775807
`unsigned long long`	Enteros entre 0 y 18446744073709551615

Float

`float`	Reales x tal que $1.17549435 \times 10^{-38} \leq \|x\| \leq 3.40282347 \times 10^{38}$, (Precisión de 7 dígitos decimales.)
`double`	Reales x tal que $2.2250738585072014 \times 10^{-308} \leq \|x\| \leq 1.7976931348623157 \times 10^{308}$, (Precisión de 15 dígitos decimales.)
`long double`	Reales x tal que $3.36210314311209350626 \times 10^{-4932} \leq \|x\| \leq 1.18973149535723176502 \times 10^{4932}$, (Precisión de 18 dígitos decimales.)

[3]Los valores de los rangos indicados son simplemente representativos y dependen de la máquina utilizada (32 bits o 64 bits). Además, estos valores no corresponden exactamente a las versiones más recientes de C++.

Las variables tipo `char` alojan caracteres, debiendo inicializarse en la forma:

```
char c = 'a';
```

Además de las letras mayúsculas y minúsculas, y símbolos como `&`, `(`, `:`, etc., hay una serie de caracteres especiales (*escape codes*) que es posible asignar a una variable `char`. Ellos son:

newline	`\n`
horizontal tab	`\t`
vertical tab	`\v`
backspace	`\b`
carriage return	`\r`
form feed	`\f`
alert (bell)	`\a`
backslash	`\\`
single quote	`\'`
double quote	`\"`

Por ejemplo, la línea:

```
cout << "Primera columna\t Segunda columna\n
          Segunda linea" << endl;
```

corresponde al *output*

```
Primera columna         Segunda columna
Segunda linea
```

3.1.5. Ingreso de datos desde el teclado.

El *header* `iostream` define un objeto especial llamado `cin` que está asociado al teclado o `stdin`. Con el operador `>>` asignamos la entrada en el dispositivo de la izquierda a la variable de la derecha; una segunda entrada requiere de otro operador `>>` y de otra variable. En el siguiente ejemplo veremos una declaración simultánea de dos variables del mismo tipo `i` y `j`, un mensaje a pantalla con las instrucciones a seguir, el ingreso de dos variables desde el teclado y luego su escritura en la pantalla.

```
#include <iostream>
using namespace std;

int main()
{
    int i, j ;
    cout << "Ingrese dos numeros enteros: " ;
    cin >> i >> j ;
    cout << "Los dos numeros ingresados fueron: " << i <<" "<< j << endl ;
    return 0;
}
```

3.1.6. Operadores aritméticos.

Existen operadores binarios (*i.e.*, que actúan sobre dos variables, una a cada lado del operador) para la suma, la resta, la multiplicación y la división:

```
+   -   *   /
```

3.1.7. Operadores relacionales.

Los símbolos para los operadores relacionales de igualdad, desigualdad, menor, menor o igual, mayor y mayor o igual son:

```
==     !=      <      <=      >      >=
```

Para las relaciones lógicas `AND`, `OR` y `NOT`:

```
&&     ||     !
```

3.1.8. Asignaciones.

a) Asignación simple. Podemos asignar a una variable un valor explícito, o el valor de otra variable:

```
i = 1;
j = k;
```

Una práctica habitual en programación es iterar porciones del código. La iteración puede estar determinada por una variable cuyo valor aumenta (disminuye) cada vez, hasta alcanzar cierto valor máximo (mínimo), momento en el cual la iteración se detiene. Para que una variable `x` aumente su valor en `2`, por ejemplo, basta escribir:

```
x = x + 2;
```

Si `x` fuera una variable matemática normal, esta expresión no tendría sentido. Esta expresión es posible porque el compilador interpreta a `x` de modo distinto a cada lado del signo igual: a la derecha del signo igual se usa el valor contenido en la variable `x` (por ejemplo, 10); a la izquierda del signo igual se usa la dirección de memoria en la cual está alojada la variable `x`. De este modo, la asignación anterior tiene el efecto de colocar en la dirección de memoria que contiene a `x`, el valor que tiene `x` más `2`. En general, todas las variables tienen un *rvalue* y un *lvalue*: el primero es el valor usado a la derecha (*right*) del signo igual en una asignación, y el segundo es el valor usado a la izquierda (*left*), es decir, su dirección de memoria.

b) Asignación compuesta.

La expresión `x = x + 2` se puede reemplazar por `x += 2`.

Existen los operadores `+=` `-=` `*=` `/=`

c) Operadores de incremento y decremento.

La expresión `x = x + 1` se puede reescribir `x += 1` o bien `x++`.

Análogamente, existe el operador `--`. Ambos operadores unarios, `++` y `--` pueden ocuparse como prefijos o sufijos sobre una variable y su acción difiere en ambos casos. Como prefijo la operación de incremento o decremento se aplica antes de que el valor de la variable sea usado en la evaluación de la expresión. Como sufijo el valor de la variable es usado en la evaluación de la expresión antes que la operación de incremento o decremento. Por ejemplo, supongamos que inicialmente $x = 3$. Entonces la instrucción `y=x++` hace que $y = 3$, $x = 4$; por su parte, `y=++x` hace que $y = 4$, $x = 4$.

Con estas consideraciones, deberíamos poder convencernos de que la salida del siguiente programa es `3 2 2-1 1 1` :

```
// Ejemplo de operadores unarios ++ y --.
#include <iostream>
using namespace std;

int main()
{
  int y ; int x = (y = 1) ;
  int w = ++x + y++;
  cout << w <<" " << x << " " << y << "-" ;
  w = x-- - --y;
  cout <<  w << " " << x << " " << y << endl ;
  return 0;
}
```

Los operadores para asignación compuesta, y los de incremento y decremento, no son sólo abreviaciones. En realidad hay que preferirlas porque implican optimizaciones en el ejecutable resultante.

3.1.9. Conversión de tipos.

Una consecuencia de que C++ sea fuertemente "tipeado" es que no se pueden hacer operaciones binarias con objetos de tipos distintos. En la siguiente expresión,

```
int i = 3;
float x = 43.8;
cout << "Suma = " << x + i << endl;
```

el computador debe sumar dos variables de tipos distintos, y en principio la operación es imposible. La estrategia para resolver este problema es convertir ambas variables a un tipo común antes de efectuar la suma (en inglés, decimos que hacemos un *cast* de un tipo a otro. Existen dos modos de proceder:

a) Conversión explícita.

Si `i` es un `int`, por ejemplo, entonces `float(i)` la convierte en `float`. Así, el programa anterior se puede reescribir:

```
int i = 3;
float x = 43.8;
cout << "Suma = " << x + float(i) << endl;
```

Ahora la suma es claramente entre dos variables `float`, y se puede realizar. Sin embargo, esto es bastante tedioso, por cuanto el programador debe realizar el trabajo de conversión personalmente cada vez que en su código se desee sumar un real con un número entero.

b) Conversión implícita.

En este caso, el compilador realiza las conversiones de modo automático, prefiriendo siempre la conversión desde un tipo de variable de menor precisión a uno de mayor precisión (de `int` a `double`, de `short` a `int`, etc.). Así, a pesar de lo que dijimos, el código anterior habría funcionado en su forma original. Evidentemente esto es muy cómodo, porque no necesitamos hacer una conversión explícita cada vez que sumamos un entero con un real. Sin embargo, debemos estar conscientes de que esta comodidad sólo es posible porque ocurren varias cosas: primero, el compilador detecta el intento de operar sobre dos variables que no son del mismo tipo; segundo, el compilador detecta, en sus reglas internas, la posibilidad de cambiar uno de los tipos (`int` en este caso) al otro (`float`); tercero, el compilador realiza la conversión, y finalmente la operación se puede llevar a cabo. Entender este proceso nos permitirá aprovechar las posibilidades de la conversión implícita de tipos cuando nuestro código involucre tipos de variables más complicados, y entender varios mensajes de error del compilador.

Es interesante notar cómo las conversiones implícitas de tipos pueden tener consecuencias insospechadas. Consideremos las tres expresiones:

i) `x = (1/2) * (x + a/x) ;`

ii) `x = (0.5) * (x + a/x) ;`

iii) `x = (x + a/x)/2 ;`

Si inicialmente `x=0.5` y `a=0.5`, por ejemplo, i) entrega el valor `x=0`, mientras ii) y iii) entregan el valor `x=1.5`. Lo que ocurre es que `1` y `2` son enteros, de modo que `1/2 = 0`. De acuerdo a lo que dijimos, uno esperaría que en i), como conviven números reales con enteros, los números enteros fueran convertidos a reales y, por tanto, la expresión tuviera el resultado esperado, 1.5. El problema es la *prioridad* de las operaciones. No todas las operaciones tienen igual prioridad (las multiplicaciones y divisiones se realizan antes que las sumas y restas, por ejemplo), y esto permite al compilador decidir cuál operación efectuar primero. Cuando se encuentra con operaciones de igual prioridad (dos multiplicaciones, por ejemplo), se procede a efectuarlas de izquierda a derecha.

Pues bien, en i), la primera operación es `1/2`, una división entre enteros, *i.e.* cero. En ii) no hay problema, porque todas son operaciones entre reales. Y en iii) la primera

operación es el paréntesis, que es una operación entre reales. Al dividir por 2 éste es convertido a real antes de calcular el resultado.

i) aún podría utilizarse, cambiando el prefactor del paréntesis a `1.0/2.0`, una práctica que sería conveniente adoptar como *standard* cuando queremos utilizar enteros dentro de expresiones reales, para evitar errores que pueden llegar a ser muy difíciles de detectar.

3.2. Control de flujo.

3.2.1. `if`, `if... else`, `if... else if`.

Las construcciones siguientes permiten controlar el flujo del programa en base a si una expresión lógica es verdadera o falsa.

a) En el caso de la sentencia `if` se evaluará la expresión `(a==b)`, si ella es cierta ejecutará la o las líneas entre los paréntesis de llave y si la expresión es falsa el programa se salta esa parte del código.

```
if (a==b) {
    cout << "a es igual a b" << endl;
}
```

En este y en muchos de los ejemplos que siguen, los paréntesis cursivos son opcionales. Ellos indican simplemente un grupo de instrucciones que debe ser tratado como una sola instrucción. En el ejemplo anterior, los paréntesis cursivos después del `if` (o después de un `while`, `for`, etc. más adelante) indican el conjunto de instrucciones que deben o no ejecutarse dependiendo de si cierta proposición es verdadera o falsa. Si ese conjunto de instrucciones es una sola, se pueden omitir los paréntesis:

```
if (a==b) cout << "a es igual a b" << endl;
```

b) En el caso `if... else` hay dos acciones mutuamente excluyentes. La sentencia `if (c!=b)` evaluará la expresión `(c!=b)`. Si ella es cierta ejecutará la o las líneas entre los paréntesis de llave que le siguen, saltándose la o las líneas entre los paréntesis de llave que siguen a la palabra clave `else`. Si la expresión es falsa el programa se salta la primera parte del código y sólo ejecuta la o las líneas entre los paréntesis de llave que siguen a `else`.

```
if (c!=d) {
    cout << "c es distinto de d" << endl;
}
else {
    cout << "c es igual a d" << endl;
}
```

c) En el último caso se evaluará la expresión que acompaña al `if` y si ella es cierta se ejecutará la o las líneas entre los paréntesis de llave que le siguen, saltándose todo el resto de las líneas entre los paréntesis de llave que siguen a las palabras claves `else if` y `else`. Si la primera expresión es falsa el programa se salta la primera parte del código y evalúa la expresión que acompaña al primer `else if` y si ella es cierta ejecutará la o las líneas entre los paréntesis de llave que le siguen, saltándose todo el resto de las líneas entre los paréntesis que siguen a otros eventuales `else if` o al `else`. Si ninguna de las expresiones lógicas resulta cierta se ejecutará la o las líneas entre los paréntesis que siguen al `else`.

```
if (e > f) {
    cout << "e es mayor que f" << endl;
}
else if (e == f) {
    cout << "e es igual a f" << endl;
}
else {
    cout << "e es menor que f" << endl;
}
```

Para C++, una expresión verdadera es igual a `1`, y una falsa es igual a `0`. Esto es, cuando escribimos `if(e>f)`, y `e>f` es falsa, en realidad estamos diciendo `if(0)`. A la inversa, `0` es considerada una expresión falsa, y cualquier valor no nulo es considerado una expresión verdadera. Así, podríamos hacer que una porción del código siempre se ejecute (o nunca) poniendo directamente `if(1)` o `if(0)`, respectivamente.

Naturalmente, lo anterior no tiene mucho sentido, pero un error habitual (y particularmente difícil de detectar) es escribir `a = b` en vez de `a == b` en una expresión lógica. Esto normalmente trae consecuencias indeseadas, pues la asignación `a = b` es una función que se evalúa siempre al nuevo valor de `a`. En efecto, una expresión como `a=3` siempre equivale a verdadero, y `a=0` siempre equivale a falso. Por ejemplo, en el siguiente programa:

```
#include <iostream>
using namespace std;

int main(){
  int k=3;
  if (k==3){
    cout << "k es igual a 3" << endl;
  }
  k=4;
  if (k=3){
    cout << "k es igual a 3" << endl;
  }
  return 0;
}
```

la salida siempre es:

```
k es igual a 3
k es igual a 3
```

aunque entre los dos `if` el valor de `k` cambia.

3.2.2. Expresión condicional.

Una construcción `if else` simple, que sólo asigna un valor distinto a una misma variable según si una proposición es verdadera o falsa, es muy común en programación. Por ejemplo:

```
if (a==b) {
    c = 1;
} else {
    c = 0;
}
```

Existen dos maneras de compactar este código. Éste se puede reemplazar por

```
if (a==b) c = 1;
else c = 0;
```

Sin embargo, esto no es recomendable por razones de claridad al leer el código. Una expresión más compacta y clara, se consigue usando el operador ternario `? :`

```
c = (a==b) ? 1 : 0;
```

Como en el caso de los operadores de incremento y decremento, el uso del operador `?` es preferible para optimizar el ejecutable resultante.

3.2.3. `switch`.

La instrucción `switch` permite elegir múltiples opciones a partir del valor de una variable entera. En el ejemplo siguiente tenemos que si `i==1` la ejecución continuará a partir del caso `case 1:`, si `i==2` la ejecución continuará a partir del caso `case 2:` y así sucesivamente. Si `i` toma un valor que no está enumerado en ningún `case` y existe la etiqueta `default`, la ejecución continuará a partir de ahí. Si no existe `default`, la ejecución continúa luego del último paréntesis cursivo.

```
switch (i)
{
case 1:
     {
         cout << "Caso 1." << endl;
     }
     break;
case 2:
```

```
    {
        cout << "Caso 2." << endl;
    }
    break;
default:
    {
        cout << "Otro caso." << endl;
    }
    break;
}
```

La instrucción `break` permite que la ejecución del programa salte a la línea siguiente después de la serie de instrucciones asociadas a `switch`. De esta manera sólo se ejecutarán las líneas correspondientes al `case` elegido y no el resto. Por ejemplo, si `i==1` veríamos en pantalla sólo la línea `Caso 1.` En el otro caso, si no existieran los `break`, y también `i==1`, entonces veríamos en pantalla las líneas `Caso 1.`, `Caso 2.` y `Otro caso.` La instrucción `default` es opcional.

3.2.4. `for`.

Una instrucción que permite repetir un bloque de instrucciones un número definido de veces es el `for`. Su sintaxis comienza con una o varias inicializaciones, luego una condición lógica de continuación mientras sea verdadera, y finalmente una o más expresiones que se evalúan vuelta por vuelta no incluyendo la primera vez. Siguiendo al `for(...)` viene una instrucción o un bloque de ellas encerradas entre paréntesis de llave. En el ejemplo siguiente la variable entera `i` es inicializada al valor 1, luego se verifica que la condición lógica sea cierta y se ejecuta el bloque de instrucciones. A la vuelta siguiente se evalúa la expresión a la extrema derecha (suele ser uno o más incrementadores), se verifica que la condición lógica se mantenga cierta y se ejecuta nuevamente el bloque de instrucciones. Cuando la condición lógica es falsa se termina el *loop*, saltando la ejecución a la línea siguiente al paréntesis que indica el fin del bloque de instrucciones del `for`. En este ejemplo, cuando `i=4` la condición de continuación será falsa y terminará la ejecución del `for`.

```
for (int i = 1; i < 4; i++) {
    cout << "Valor del indice: " << i << endl;
}
```

El *output* correspondiente es:

```
Valor del indice: 1
Valor del indice: 2
Valor del indice: 3
```

Cualquier variable declarada en el primer argumento del `for` es local al *loop*. En este caso, la variable `i` es local, y no interfiere con otras posibles variables `i` que existan en nuestro código.

`for` es una instrucción particularmente flexible. En el primer y tercer argumento del `for` se puede colocar más de una instrucción, separadas por comas. Esto permite, por ejemplo, involucrar más de una variable en el ciclo. El código:

```
for (int i=0, k=20; (i<10) && (k<50); i++, k+=6) {
    cout << "i + k = " << i + k << endl;
}
```

resulta en el *output*:

```
i + k = 20
i + k = 27
i + k = 34
i + k = 41
i + k = 48
```

Además, la condición de continuación (segundo argumento del `for`), no tiene por qué depender de las variables inicializadas en el primer argumento. Y el tercer argumento no tiene por qué ser un incremento o decremento de las variables del *loop*; puede ser cualquier expresión que queramos ejecutar cada vez que un ciclo termina. En el siguiente ejemplo, además de incrementar los contadores en cada ciclo, se envía un mensaje a pantalla:

```
for (int i=1, k=2;k<5 && i<20;k++, i+=2, cout << "Fin iteracion" << endl){
   cout << " i = " << i <<'','';
   cout << " k = " << k << endl;
}
```

El resultado de las iteraciones:

```
 i = 1, k = 2
Fin iteracion
 i = 3, k = 3
Fin iteracion
 i = 5, k = 4
Fin iteracion
```

Todos los argumentos del `for` son opcionales (no los `;`), por lo cual se puede tener un `for` que carezca de inicialización y/o de condición de continuación y/o de una expresión que se evalúe en cada iteración.

Un caso típico en que se aprovecha la opcionalidad de los argumentos del `for` es para tener un *loop* infinito, que puede servir para dejar el programa en pausa indefinida. Para salir del *loop* (y en general, para detener cualquier programa en C++), hay que presionar `^C`:

```
for (; ; ) cout << "Este es un loop infinito, ^C para detenerlo"<< endl;
```

Se puede además, salir abruptamente del *loop* con `break`. El código:

```
  for(int indice=0; indice<10; indice++) {
    int cuadrado = indice*indice ;
    cout << indice << " " ;
    if(cuadrado > 10 ) break ;
  }
  cout << endl;
```

da la salida a pantalla:

```
0 1 2 3 4
```

aun cuando la condición de continuación permite que `indice` llegue hasta `9`.

Finalmente, las variables involucradas en el `for` pueden ser modificadas dentro del ciclo. Por ejemplo, modifiquemos uno de los ejemplos anteriores, cambiando la variable `k` en medio del ciclo:

```
for (int i=1, k=2;k<5 && i<8;k++, i+=2, cout << "Fin iteracion" << endl){
  cout << " i = " << i  << ", k = " << k << endl;
  k = k+5;
}
```

El resultado es:

```
 i = 1, k = 2
Fin iteracion
```

En vez de pasar por el ciclo tres veces, como ocurría originalmente, el programa sale del *loop*, al cabo del primer ciclo, $k = 2 + 5 = 7 > 5$.

En general no es una buena práctica modificar las variables internas del ciclo en medio de él, porque no es muy ordenado, y el desorden normalmente conduce a los errores en programación, pero ocasionalmente puede ser útil hacer uso de esta libertad que proporciona el lenguaje. Los ciclos `for` pueden anidarse, tal que uno contenga a otro completamente.

3.2.5. `while`.

La instrucción `while` permite repetir un bloque de instrucciones encerradas entre paréntesis de llave mientras la condición lógica que acompaña al `while` se mantenga cierta. La condición es evaluada antes de que comience la primera iteración; si es falsa en ésta o en una posterior evaluación no se ejecuta el bloque de instrucciones que le siguen y se continúa la ejecución en la línea siguiente al paréntesis que indica el fin del bloque asociado al `while`. Hay que notar que la instrucción `while` podría no ejecutarse ni una sola vez si la condición no se cumple inicialmente. Un ejemplo simple:

```
int i=1;
while (i < 3) {
    cout << i++ << " ";
}
```

que da por resultado: `1 2`.

En el siguiente *loop*, la salida será: `5 4 3 2 1` (¿Por qué?)

```
int k=5 ;
while(k) {
    cout << k-- <<" ";
}
```

3.2.6. `do... while`.

La instrucción `do... while` es análoga a `while`, salvo que la condición lógica es evaluada después de la primera iteración. Por tanto, el bloque se ejecuta al menos una vez, siempre. Un ejemplo simple:

```
do {
    cout << i++ << endl;
} while (i<=20);
```

Podemos construir de otra manera un *loop* infinito usando `do while`

```
do {
    cout << "Este es un segundo loop infinito, ^C para detenerlo"<< endl;
} while (1);
```

3.2.7. `goto`.

Existe también en C++ una instrucción `goto` que permite saltar de un punto a otro del programa (`goto salto;` permite saltar a la línea que contiene la instrucción `salto:`). Sin embargo, se considera una mala técnica de programación usar `goto`, y siempre se puede diseñar un programa evitándolo. Es altamente no recomendable, pero si su utilización simplifica el código se puede usar.

3.3. Funciones.

Las funciones nos permiten programar partes del procedimiento por separado. Un ejemplo simple de ellas lo vimos en la subsección 3.1.2.

3.3.1. Funciones tipo `void`.

Un caso especial de funciones es aquel en que el programa que llama la función no espera que ésta le entregue ningún valor al terminar. Por ejemplo, en la subsección 3.1.2, la función `PrintHola` simplemente imprime un mensaje en pantalla. El resto del programa no necesita de ningún resultado parcial proveniente de la ejecución de dicha función. La definición de estas funciones debe ir precedida de la palabra `void`, como en el ejemplo citado.

3.3.2. `return`.

Si deseamos definir una función que calcule una raíz cuadrada, evidentemente esperamos que la función nos entregue un resultado: el valor de la raíz cuadrada. En este caso hay que traspasar el valor de una variable desde la función al programa que la llamó. Esto se consigue con `return`. Veamos un ejemplo muy simple:

```
int numero(){
    int i = 3;
    return i;
}

int main(){
    cout << "Llamamos a la funcion" << endl;
    cout << "El numero es: " << numero() << endl;
    int i = 5;
    i = i + numero();
    cout << "El numero mas 5 es: " << i << endl;
    return 0;
}
```

En este caso, la función simplemente entrega el valor de la variable interna `i`, es decir 3, el cual puede ser usado para salida en pantalla o dentro de operaciones matemáticas corrientes.

Separando declaración e implementación de la función, el ejemplo anterior se escribe:

```
int numero();

int main(){ ... }

int numero(){
   int i = 3;
   return i;
}
```

Dos observaciones útiles:

a) La declaración de la función lleva antepuesto el tipo de variable que la función entrega. En el ejemplo, la variable entregada es un entero, `i`, y la declaración debe ser, por tanto, `int numero()`. Podemos tener funciones tipo `double`, `char`, `long`, etc., de acuerdo al tipo de variable que corresponde a `return`.

b) La variable `i` que se usa dentro de `main()` y la que se usa dentro de `numero()` son distintas. A pesar de que tienen el mismo nombre, se pueden usar independientemente como si se llamaran distinto. Se dice que `i` es una variable *local*.

Después de `return` debe haber una expresión que se evalúe a una variable del tipo correspondiente, ya sea explícitamente o a través de un *cast* implícito. Las siguientes funciones devuelven un `double` al programa:

```
double f1(){
  double l = 3.0;
  return l;
}

double f2(){
  double l = 3.0, m = 8e10;
  return l*m;
}

double f3(){
  int l = 3;
  return l;
}
```

Sin embargo, la siguiente función hará que el compilador emita una advertencia, pues se está tratando de devolver un `double` donde debería ser un `int`, y la conversión implica una pérdida de precisión:

```
int f4(){
  double l=3.0;
  return l;
}
```

Naturalmente, podemos modificar la función anterior haciendo una conversión explícita antes de devolver el valor: `return int(l)`.

3.3.3. Funciones con parámetros.

Volviendo al ejemplo de la raíz cuadrada, nos gustaría llamar a esta función con un parámetro (el número al cual se le va a calcular la raíz cuadrada). Consideremos por ejemplo una función que necesita un solo parámetro, de tipo `int`, y cuyo resultado es otro `int`:

```
int funcion(int i){
    i+=4;
    return i;
}

int main(){
    int i = 3;
    cout << "El valor de la funcion es " << funcion(i)
         << endl;
    cout << "El valor del parametro es " << i << endl;
    return 0 ;
}
```

El resultado en pantalla es:

```
El valor de la funcion es 7
El valor del parametro es 3
```

La función `funcion` entrega el valor del parámetro más 4. Usamos el mismo nombre (`i`) para las variables en `main` y `funcion`, pero son variables locales, así que no interfieren. Lo importante es notar que cuando se llama a la función, la reasignación del valor de `i` (`i+=4`) ocurre sólo para la variable local en `funcion`; el parámetro externo mantiene su valor.

Separando declaración e implementación el ejemplo anterior se escribe:

```
int funcion(int);

int main(){...}

int funcion(int i){
   i+=4;
   return i;
}
```

Si nuestra función necesita más parámetros, basta separarlos con comas, indicando para cada uno su tipo:

```
int funcion2(int,double);
void funcion3(double,int,float);
double funcion4(float);
```

El compilador verifica cuidadosamente que cada función sea llamada con el número de parámetros adecuados, y que cada parámetro corresponda al tipo especificado. En los ejemplos anteriores, `funcion2` debe ser llamada siempre con dos argumentos, el primero de los cuales es `int` y el segundo `double`. Como siempre, puede ser necesario un *cast* implícito (si se llama `funcion2` con el segundo argumento `int`, por ejemplo), pero si no existe una regla de conversión automática (llamando a `funcion2` con el primer argumento `double`, por ejemplo), el compilador enviará una advertencia. Además, el compilador verifica que el valor de retorno de la función sea usado como corresponde. Por ejemplo, en las dos líneas:

```
double m = funcion2(2,1e-3);
int k = funcion4(0.4);
```

la primera compilará exitosamente (pero hay un *cast* implícito), y la segunda dará una advertencia.

Existen dos modos de transferir parámetros a una función:

a) Por valor. Se le pasan los parámetros para que la función que es llamada copie sus valores en sus propias variables locales, las cuales desaparecerán cuando la función termine y no tienen nada que ver con las variables originales.

   Hasta ahora, en todos los ejemplos de esta subsección el traspaso de parámetros ha sido por valor. En la función `int funcion(int)`, en el código de la página 100, lo que ha ocurrido es que la función copia el *valor* de la variable externa `i` en una nueva variable (que también se llama `i`, pero está en otra dirección de memoria). El valor con el que trabaja la función es la copia, manteniendo inalterada la variable original.

b) Por referencia. Se le pasa la dirección de memoria de los parámetros. La función llamada puede modificar el valor de tales variables.

La misma función de la página 100 puede ser modificada para que el paso de parámetros sea por referencia, modificando la declaración:

```
int funcion(int &);

int main()
{
    int i = 3;
    cout << "El valor de la funcion es " << funcion(i)
         << endl;
    cout << "El valor del parametro es " << i << endl;
    return 0;
}
int funcion(int & i)
{
    i+=4;
    return i;
}
```

En vez de traspasarle a `funcion` el valor del parámetro, se le entrega la *dirección* de memoria de dicha variable. Debido a ello, `funcion` puede modificar el valor de la variable. El resultado en pantalla del último programa será:

```
El valor de la funcion es 7
El valor del parametro es 7
```

Debido a que las variables dejan de ser locales, el paso de parámetros por referencia debe ser usado con sabiduría. De hecho el ejemplo presentado es poco recomendable. Peor aún, el problema es no sólo que las variables dejan de ser locales, sino que *es imposible saber que no lo son* desde el `main`. En efecto, el `main` en ambas versiones de `funcion` es el mismo. Lo único que cambió es la declaración de la función. Puesto que un usuario normal usualmente no conoce la declaración e implementación de cada función que desea usar (pues pueden haber sido hechas por otros programadores), dejamos al usuario en la indefensión.

Por otro lado, hay al menos dos situaciones en que el paso de referencia es la única opción viable para entregar los parámetros. Un caso es cuando hay que cuidar el uso de la memoria. Supongamos que una función necesita un parámetros que es una matriz de 10 millones de filas por 10 millones de columnas. Seguramente estaremos llevando al límite los recursos de nuestra máquina, y sería una torpeza pasarle la matriz por valor: ello involucraría, primero, duplicar la memoria utilizada, con el consiguiente riesgo de que nuestro programa se interrumpa; y segundo, haría el programa más lento, porque la función necesitaría llenar su versión local de la matriz elemento por elemento. Es decir, nada de eficiente. En esta situación, el paso por referencia es lo adecuado.

Un segundo caso en que el paso por referencia es recomendable es cuando efectivamente nuestra intención es cambiar el valor de las variables. El ejemplo típico es el intercambio de dos variables entre sí, digamos `a1=1` y `a2=3`. Luego de ejecutar la función queremos que `a1=3` y `a1=1`. El siguiente código muestra la definición y el uso de una función para esta tarea, y por cierto requiere el paso de parámetros por referencia:

```
#include <iostream>
void  swap(int &,int &);
using namespace std;

int main(){

  int i = 3, k=10;
  swap(i,k);
  cout <<  "Primer argumento: " << i << endl;
  cout <<  "Segundo argumento: " << k << endl;
  return 0 ;
}

void swap(int & j,int & p){
  int temp = j;
  j = p;
  p = temp;
}
```

El *output* es:

```
Primer argumento: 10
Segundo argumento: 3
```

En el ejemplo de la matriz anterior, sería interesante poder pasar el parámetro por referencia, para ahorrar memoria y tiempo de ejecución, pero sin correr el riesgo de que nuestra matriz gigantesca sea modificada por accidente. Afortunadamente existe el modo de hacerlo, usando una palabra que ya hemos visto antes: `const`. En el siguiente código:

```
int f5(const int &);
int main(){...}
int f5(const int & i){...};
```

`f5` recibirá su único argumento por referencia, pero, debido a la presencia del modificador `const`, el compilador avisará si se intenta modificar el argumento en medio del código de la función.

3.3.4. Parámetros por defecto.

C++ permite que omitamos algunos parámetros de la función llamada, la cual reemplaza los valores omitidos por otros predeterminados. Tomemos por ejemplo la función

`int funcion(int);` de la subsección 3.3.3, y modifiquémosla de modo que si no le entregamos parámetros, asuma que el número entregado fue 5:

```
int funcion(int i = 5){
    i+=4;
    return i;
}

int main(){
    cout << "El resultado default es " << funcion() << endl;
    int i = 3;
    cout << "Cuando el parametro vale " << i <<
            " el resultado es " << funcion(i) << endl;
    return 0;
}
```

El *output* correspondiente es:

```
El resultado default es 9
Cuando el parametro vale 3 el resultado es 7
```

Separando declaración e implementación:

```
int funcion(int = 5);
main(){...}
int funcion(int i){
    i+=4;
    return i;
}
```

Si una función tiene n argumentos, puede tener $m \leq n$ argumentos opcionales. La única restricción es que, en la declaración e implementación de la función, los parámetros opcionales ocupen los últimos m lugares:

```
void f1(int,int = 4);
int f2(double,int = 4, double = 8.2);
double f3(int = 3,double = 0.0, int = 0);
```

En este caso, `f1(2)`, `f1(2,8)`, `f2(2.3,5)`, `f3(3)`, `f3()`, y muchas otras, son todas llamadas válidas de estas funciones. Cada vez, los parámetros no especificados son reemplazados por sus valores predeterminados.

3.3.5. Ejemplos de funciones: raíz cuadrada y factorial.

Raíz cuadrada.

Con lo visto hasta ahora, ya podemos escribir un programa que calcule la raíz cuadrada de una función. Para escribir una función, debemos tener claro qué se espera de ella: cuántos

y de qué tipo son los argumentos que recibirá, qué tipo de valor de retorno deberá tener, y, por cierto, un nombre adecuado. Para la raíz cuadrada, es claro que el argumento es un número. Pero ese número podría ser un entero o un real, y eso al compilador no le da lo mismo. En este punto nos aprovechamos del *cast* implícito: en realidad, basta definir la raíz cuadrada con argumento `double`; de este modo, si se llama la función con un argumento `int`, el compilador convertirá automáticamente el `int` en `double` y nada fallará. En cambio, si la definiéramos para `int` y la llamamos con argumento `double`, el compilador se quejaría de que no sabe efectuar la conversión. Si el argumento es `double`, evidentemente esperamos que el valor de retorno de la función sea también un `double`. Llamando a la función `raiz`, tenemos la declaración:

```
double raiz(double);
```

Debido a la naturaleza de la función raíz cuadrada, `raiz()` no tendría sentido, y por tanto no corresponde declararla con un valor *default*.

Ahora debemos pensar en cómo calcular la raíz cuadrada. Usando una variante del método de Newton-Raphson, se obtiene que la secuencia

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

converge a $\sqrt{a}$ cuando $n \to \infty$. Por tanto, podemos calcular la raíz cuadrada con aproximaciones sucesivas. El cálculo terminará en el paso N, cuando la diferencia entre el cuadrado de la aproximación actual, x_N, y el valor de a, sea menor que un cierto número pequeño: $|\, x_N^2 - a \,| < \epsilon \ll 1$. El valor de ϵ determinará la precisión de nuestro cálculo. Un ejemplo de código lo encontramos a continuación:

```
#include <iostream>
#include <cmath>
using namespace std;

double raiz(double);

int main(){

  double r;

  cout.precision(20);
  cout << "Ingrese un numero: " << endl;
  cin >> r;
  cout << raiz(r) << endl;
  return 0 ;
}

double raiz(double a){
  double x =a/2.0 ; // para comenzar
  double dx = 1e3, epsilon = 1e-8;
```

```
  while (fabs(dx)>epsilon){
    x = (x + a/x)/2;
    dx = x*x - a;
    cout << "x = " << x << ", precision = " << dx << endl;
   }
  return x;
}
```

Luego de la declaración de la función `raiz`, está `main`, y al final la implementación de `raiz`. En `main` se pide al usuario que ingrese un número, el cual se aloja en la variable `r`, y se muestra en pantalla el valor de su raíz cuadrada. La instrucción `cout.precision(20)` permite que la salida a pantalla muestre el resultado con 20 cifras significativas.

En la implementación de la función hay varios aspectos que observar. Se ha llamado `x` a la variable que contendrá las sucesivas aproximaciones a la raíz. Al final del ciclo, `x` contendrá el valor (aproximado) de la raíz cuadrada. `dx` contiene la diferencia entre el cuadrado de `x` y el valor de `a`, `epsilon` es el número (pequeño) que determina si la aproximación es satisfactoria o no.

El ciclo está dado por una instrucción `while`, y se ejecuta mientras `dx>epsilon`, es decir, termina cuando `dx` es suficientemente pequeño. El valor absoluto del real `dx` se obtiene con la función matemática `fabs`, disponible en el *header* `cmath` incluido al comienzo del programa. Observar que inicialmente `dx=1e3`, esto es un valor muy grande; esto permite que la condición del `while` sea siempre verdadera, y el ciclo se ejecuta al menos una vez.

Dentro del ciclo, se calcula la nueva aproximación, y se envía a pantalla un mensaje con la aproximación actual y la precisión alcanzada (dada por `dx`). Eventualmente, cuando la aproximación es suficientemente buena, se sale del ciclo y la función entrega a `main` el valor de `x` actual, que es la última aproximación calculada.

Factorial.

Otro ejemplo útil es el cálculo del factorial, definido para números naturales:

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \ , \quad 0! \equiv 1 \ .$$

La estrategia natural es utilizar un ciclo `for`, determinado por una variable entera i, que va desde 1 a n, guardando los resultados en una variable auxiliar que contiene el producto de todos los números naturales desde 1 hasta i:

```
#include <iostream>
using namespace std;

int factorial(int);

int main(){
  int n=5 ;
  cout << "El factorial de " << n << " es: " << factorial(n) << endl;
```

```
  return 0 ;
}

int factorial(int i)
{
  int f =1;
  for (int j=1;j<=i;j++){
    f = f*j;
  }
  return f;
}
```

Observar que la variable auxiliar `f`, que contiene el producto de los primeros i números naturales, debe ser inicializada a 1. Si se inicializara a 0, `factorial(n)` sería 0 para todo `n`.

Esta función no considera el caso `n=0`, pero al menos para el resto de los naturales funcionará bien.

3.3.6. Alcance, visibilidad, tiempo de vida.

Con el concepto de función hemos apreciado que es posible que coexistan variables con el mismo nombre en puntos distintos del programa, y que signifiquen cosas distintas. Conviene entonces tener en claro tres conceptos que están ligados a esta propiedad:

Alcance (*scope*) La sección del código durante la cual el nombre de una variable puede ser usado. Comprende desde la declaración de la variable hasta el final del cuerpo de la función donde es declarada.

Si la variable es declarada dentro de una función es *local*. Si es definida fuera de todas las funciones (incluso fuera de `main`), la variable es *global*.

Visibilidad Indica cuáles de las variables, actualmente al alcance, pueden ser accesadas. En nuestros ejemplos (subsección 3.3.3), la variable `i` en `main` aún está al alcance dentro de la función `funcion`, pero no es visible, y por eso es posible reutilizar el nombre.

Tiempo de vida Indica cuándo las variables son creadas y cuándo destruidas. En general este concepto coincide con el alcance (las variables son creadas cuando son declaradas y destruidas cuando la función dentro de la cual fueron declaradas termina), salvo porque es posible definir: (a) variables *dinámicas*, que no tienen alcance, sino sólo tiempo de vida; (b) variables *estáticas*, que conservan su valor entre llamadas sucesivas de una función (estas variables tienen tiempo de vida mayor que su alcance). Para declarar estas últimas se usa un modificador `static`.

El efecto del modificador `static` se aprecia en el siguiente ejemplo:

```
#include <iostream>

int f();
```

```
using namespace std;

int main(){

  cout << f() << endl;
  cout << f() << endl;
  return 0;
}

int f(){
  int x=0;
  x++;
  return x;
}
```

La función `f` simplemente toma el valor inicial de `x` y le suma 1. Como cada vez que la función es llamada la variable local `x` es creada e inicializada, el resultado de este programa es siempre un 1 en pantalla:

```
1
1
```

Ahora modifiquemos la función, haciendo que `x` sea una variable estática:

```
#include <iostream>

int f();
using namespace std;

int main(){

  cout << f() << endl;
  cout << f() << endl;
  return 0 ;
}

int f(){
  static int x=0;
  x++;
  return x;
}
```

Ahora, al llamar a `f` por primera vez, la variable `x` es creada e inicializada, pero no destruida cuando la función termina, de modo que conserva su valor cuando es llamada por segunda vez:

```
1
2
```

Veamos un ejemplo de una variable estática en el cálculo del factorial:

```
int factorial2(int i=1){
    static int fac = 1;
    fac*=i;
    return fac ;
}

int main (){
  int n=5;
  int m=n;
  while(n>0)  factorial2(n--);
  cout << "El factorial de "<< m << " es = " << factorial2() << endl;
  return 0 ;
}
```

La idea, si se desea calcular el factorial de 5, por ejemplo, es llamar a la función `factorial2` una vez, con argumento $n = 5$, y después disminuir n en 1. Dentro de la función, una variable estática (`fac`) aloja el valor $1 * 5 = 5$. Luego se llama nuevamente con $n = 4$, con lo cual `fac=1*5*4`, y así sucesivamente, hasta llegar a $n = 1$, momento en el cual `fac=1*5*4*3*2*1`. Al disminuir n en 1 una vez más, la condición del `while` es falsa y se sale del ciclo. Al llamar una vez más a `factorial2`, esta vez sin argumentos, el programa asume que el argumento tiene el valor predeterminado 1, y así el resultado es `1*5*4*3*2*1*1`, es decir 5!.

Observemos el uso del operador de decremento en este programa: `factorial2(n--)` llama a la función con argumento `n` y *después* disminuye `n` en 1. Ésto es porque el operador de decremento está actuando como sufijo, y es equivalente a las dos instrucciones:

```
factorial2(n);
n--;
```

Si fuera un prefijo [`factorial2(n--)`], primero disminuiría `n` en 1, y llamaría luego a `factorial2` con el nuevo valor de `n`

Este ejemplo de cálculo del factorial ilustra el uso de una variable estática, que aloja los productos parciales de los números enteros, pero no es un buen ejemplo de una función que calcule el factorial, porque de hecho esta función no lo calcula: es `main` quien, a través de sucesivas llamadas a `factorial2`, calcula el factorial, pero la función en sí no.

3.3.7. Recursión.

C++ soporta un tipo especial de técnica de programación, la recursión, que permite que una función se llame a sí misma (esto es no trivial, por cuanto si definimos, digamos, una función `f`, dentro del cuerpo de la implementación no hay ninguna declaración a una función `f`, y por tanto en principio no se podría usar `f` porque dicho nombre no estaría en *scope*; C++ permite soslayar este hecho). La recursión permite definir de modo muy compacto una función que calcule el factorial de un número entero `n`.

```
int factorial3(int n){
    return (n<2) ? 1: n * factorial3(n-1);
}

int main(){
  int n=5;
  cout << "El factorial de "<< n << " es = " << factorial3(n) << endl;
  return 0;
}
```

En este tercer ejemplo, el factorial de n es definido en función del factorial de $n-1$. Se ha usado la expresión condicional (operador `?`) para compactar aún más el código. Por ejemplo, al pedir el factorial de 5 la función se pregunta si $5 < 2$. Esto es falso, luego, la función devuelve a `main` el valor `5*factorial3(4)`. A su vez, `factorial3(4)` se pregunta si $4 < 2$; siendo falso, devuelve *a la función que la llamó* (es decir, a `factorial3(5)`), el valor `4*factorial3(3)`. El proceso sigue hasta que `factorial(2)` llama a `factorial3(1)`. En ese momento, $1 < 2$, y la función `factorial3(1)`, en vez de llamar nuevamente al factorial, devuelve a la función que la llamó el valor 1. No hay más llamadas a `factorial3`, y el proceso de recursión se detiene. El resultado final es que `main` recibe el valor `factorial3(5)` = `5*factorial3(4)` = $\cdots$ = `5*4*3*2*factorial3(1)` = `5*4*3*2*1`= 120.

Este tercer código para el cálculo del factorial sí considera el caso $n = 0$, y además es más eficiente, al ser más compacto.

La recursión debe ser empleada con cuidado. Es importante asegurarse de que existe una condición para la cual la recursión se detenga, de otro modo, caeríamos en una recursión infinita que haría inútil nuestro programa. En el caso del factorial, pudimos verificar que dicha condición existe, por tanto el programa es finito. En situaciones más complicadas puede no ser tan evidente, y es responsabilidad del programador —como siempre— revisar que todo esté bajo control.

3.3.8. Funciones internas.

Existen muchas funciones previamente implementadas en C++ almacenadas en distintas bibliotecas. Una de las bibliotecas importante es la matemática. Para usarla uno debe incluir el archivo de *header* `<cmath>` y luego al compilar agregar al final del comando de compilación `-lm`:

```
g++ -Wall -o <salida> <fuente>.cc -lm
```

si se desea crear un ejecutable `<salida>` a partir del código en `<fuente>.cc`.

Veamos algunas de estas funciones:

`pow(x,y)`	Eleva a potencia, x^y
`fabs(x)`	Valor absoluto
`sqrt(x)`	Raíz cuadrada
`sin(x) cos(x)`	Seno y coseno
`tan(x)`	Tangente
`atan(x)`	Arcotangente de x en $[-\pi, \pi]$
`atan2(y, x)`	Arcotangente de y/x en $[-\pi, \pi]$
`exp(x)`	Exponencial
`log(x) log10(x)`	Logaritmo natural y logaritmo en base 10
`floor(x)`	Entero más cercano hacia abajo (e.g. `floor(3.2)=3`)
`ceil(x)`	Entero más cercano hacia arriba (e.g. `ceil(3.2)=4`)
`fmod(x,y)`	El resto de x/y (e.g. `fmod(7.3, 2)=1.3`)

Para elevar a potencias enteras, es más conveniente usar la forma explícita en vez de la función `pow`, *i.e.* calcular `x^3` como `x*x*x` es más eficiente computacionalmente que `pow(x,3)`, debido a los algoritmos que usa `pow` para calcular potencias. Éstos son más convenientes cuando las potencias no son enteras, en cuyo caso no existe una forma explícita en términos de productos.

3.4. Punteros.

Una de las ventajas de C++ es permitir el acceso directo del programador a zonas de memoria, ya sea para crearlas, asignarles un valor o destruirlas. Para ello, además de los tipos de variables ya conocidos (`int`, `double`, etc.), C++ proporciona un nuevo tipo: el *puntero*. El puntero no contiene el valor de una variable, sino la dirección de memoria en la cual dicha variable se encuentra.

Un pequeño ejemplo nos permite ver la diferencia entre un puntero y la variable a la cual ese puntero "apunta":

```
int main(){
    int i = 42;
    int * p = &i;
    cout << "El valor del puntero es: " << p << endl;
    cout << "Y apunta a la variable: " << *p << endl;
    return 0;
}
```

En este programa definimos una variable `i` entera. Al crear esta variable, el programa reservó un espacio adecuado en algún sector de la memoria. Luego pusimos, en esa dirección de memoria, el valor `42`. En la siguiente línea creamos un puntero a `i`, que en este caso denominamos `p`. Los punteros no son punteros a cualquier cosa, sino punteros a un tipo particular de variable. Ello es manifiesto en la forma de la declaración: `int * p`. En la misma línea asignamos a este puntero un valor. Ese valor debe ser también una dirección de memoria, y para eso usamos `&i`, que es la dirección de memoria donde está `i`. Ya hemos visto antes el uso de `&` para entregar una dirección de memoria, al estudiar paso de parámetros a funciones por referencia (3.3.3).

Al ejecutar este programa vemos en pantalla los mensajes:

```
El valor del puntero es: 0xbffff9d8
Y apunta a la variable: 42
```

Primero obtenemos un número hexadecimal imposible de determinar *a priori*, y que corresponde a la dirección de memoria donde quedó ubicada la variable `i`. La segunda línea nos da el valor de la variable que está en esa dirección de memoria: 42. Puesto que `*` aplicado a un puntero entrega el contenido de esa dirección de memoria, se le denomina *operador de desreferenciación*.

En este ejemplo, hemos creado un puntero que contiene la dirección de memoria de una variable preexistente: declaramos una variable, esa variable queda en alguna dirección de memoria, y después asignamos esa dirección de memoria a un puntero. En este caso, podemos referirnos a la variable tanto por su nombre (`i`) como por su puntero asociado (`p_i`).

También es posible crear directamente una dirección de memoria, sin necesidad de crear una variable antes. En este caso, la única forma de manipular este objeto es a través de su puntero, porque no existe ninguna variable y por tanto ningún nombre asociado a él. Esto se hace con el operador `new`. El mismo ejemplo anterior puede ser reescrito usando sólo punteros:

```
int  main(){
    int * p = new int;
    *p = 42;
    cout << "El valor del puntero es: " << p << endl;
    cout << "Y apunta a la variable: " << *p << endl;
    delete p;
    return 0;
}
```

La primera línea crea un nuevo puntero a `int` llamado `p`. `new` verifica que haya suficiente memoria para alojar un nuevo `int`, y si es así reserva ese espacio de memoria. En `p` queda la dirección de la memoria reservada. Esto es equivalente a la declaración `int i;` del programa anterior, salvo que ahora la única manera de accesar esa dirección de memoria es a través del puntero `p`. A continuación se coloca *dentro* de esa dirección (observar la presencia del operador de desreferenciación `*`) el número `42`. El programa manda a pantalla la misma información que la versión anterior, salvo que seguramente el valor de `p` será distinto.

Finalmente, ya que el puntero no volverá a ser usado, la dirección de memoria debe ser liberada para que nuestro u otros programas puedan utilizarla. Ello se realiza con el operador `delete`. Todo puntero creado con `new` debe ser, cuando ya no se utilice, borrado con `delete`. Ello evitará desagradables problemas en nuestro programa debido a fuga de memoria (*memory leak*).

Los punteros tienen gran importancia cuando de manejar datos dinámicos se trata, es decir, objetos que son creados durante la ejecución del programa, en número imposible de predecir al momento de compilar. Por ejemplo, una aplicación `X-windows` normal que crea una, dos, tres, etc. ventanas a medida que uno abre archivos. En este caso, cada ventana es un objeto dinámico, creado durante la ejecución, y la única forma de manejarlo es a través de un puntero a ese objeto, creado con `new` cuando la ventana es creada, y destruido con `delete` cuando la ventana es cerrada.

3.5. Matrices o arreglos.

3.5.1. Declaración e inicialización.

Podemos declarar (e inicializar inmediatamente) matrices de enteros, reales de doble precisión, caracteres, etc., según nuestras necesidades.

```
int a[5];
double r[3] = {3.5, 4.1, -10.8};
char palabra[5];
```

Una vez declarada la matriz (digamos `a[5]`), los valores individuales se accesan con `a[i]`, con `i` desde 0 a 4. Por ejemplo, podemos inicializar los elementos de la matriz así:

```
a[0] = 3;
a[3] = 5; ...
```

o si queremos ingresarlos desde el teclado:

```
for (i = 0; i < 5; i++){
    cin >> a[i];
}
```

Y si deseamos escribirlos en pantalla:

```
for (i = 0; i < 5; i++){
    cout  <<  a[i] ;
}
```

3.5.2. Matrices como parámetros de funciones.

Si deseamos, por ejemplo, diseñar una función que mande los elementos de una matriz a pantalla, necesitamos entregarle como parámetro la matriz que va a utilizar. Para ello se agrega `[]` luego del nombre de la variable, para indicar que se trata de una matriz:

```
void PrintMatriz(int, double []);

int main(){
     double matriz[5] = {3.5, 5.2, 2.4, -0.9, -10.8};
     PrintMatriz(5, matriz);
     return 0;
}

void PrintMatriz(int i, double a[]){
     for (int j = 0; j < i; j++){
          cout << "Elemento " << j << " = " << a[j] << endl;
     }
}
```

Observemos que la función debe recibir dos parámetros, uno de los cuales es la dimensión de la matriz. Esto se debe a que cuando las matrices son usadas como parámetros la información de su dimensión no es traspasada, y debe ser comunicada independientemente. Una ligera optimización al programa anterior es modificar `main` a:

```
int main()
{
     const int dim = 5;
     double matriz[dim] = {3.5, 5.2, 2.4, -0.9, -10.8};
     PrintMatriz(dim, matriz);
     return 0;
}
```

De este modo, si eventualmente cambiamos de opinión y deseamos trabajar con matrices de longitud distinta, sólo hay que modificar una línea de código (la primera) en todo el programa, el cual puede llegar a ser bastante largo por cierto. (En el ejemplo, también habría que cambiar la línea de inicialización de la matriz, porque asume que la matriz requiere sólo 5 elementos, pero de todos modos debería ser clara la enorme conveniencia.) Podemos reescribir este programa con un comando de preprocesador para hacer la definición de la dimensión:

```
#include <iostream>
#define DIM 5
using namespace std;

int main(){
     double matriz[DIM] = {3.5, 5.2, 2.4, -0.9, -10.8};
     PrintMatriz(DIM, matriz);
     return 0;
}
```

Sin embargo, ninguna de estas alternativas resuelve el problema de que el compilador espera que la dimensión de una matriz sea un entero constante, determinado en el momento de la compilación (no de la ejecución).

3.5.3. Asignación dinámica.

La reserva de memoria para la matriz podemos hacerla en forma dinámica ocupando el operador `new` que pedirá al sistema la memoria necesaria, si está disponible el sistema se la asignará. Como con cualquier puntero, una vez desocupado el arreglo debemos liberar la memoria con el comando `delete`.

```
#include <iostream>
using namespace std;

int main()
{
     cout<<"Ingrese la dimension deseada :" ;
```

```
     int dim ;
     cin >> dim ;
     double * matriz = new double[dim] ; // Reserva la memoria
     for(int i=0; i < dim; i++) {
        cout << "Ingrese elemento "<< i <<" : ";
        cin >> matriz[i] ;
     }

     for (int i=0;i<dim;i++){
        cout << matriz[i] << ", ";
     }
     cout << endl;

     delete [] matriz;       // Libera la memoria reservada
     return 0;
}
```

Este ejemplo permite apreciar una gran ventaja del uso de punteros, al permitirnos liberarnos de definir la dimensión de una matriz como una constante. Aquí, `dim` es simplemente un `int`. La asignación dinámica permite definir matrices cuya dimensión se determina recién durante la ejecución.

Observemos finalmente que la liberación de memoria, en el caso de arreglos, se hace con el operador `delete []`, no `delete` como en los punteros usuales.

3.5.4. Matrices multidimensionales.

Es fácil declarar e inicializar matrices de más de una dimensión:

```
double array[10][8];
int array[2][3] = {{1, 2, 3},
                   {4, 5, 6}};
```

Una operación usual es definir primero las dimensiones de la matriz, y luego llenar sus elementos uno por uno (o desplegarlos en pantalla), recorriendo la matriz ya sea por filas o por columnas. Hay que tener cuidado del orden en el cual uno realiza las operaciones. En el siguiente código, definimos una matriz de 10 filas y 3 columnas, la llenamos con ceros elemento por elemento, y luego inicializamos tres de sus elementos a números distintos de cero. Finalmente desplegamos la matriz resultante en pantalla:

```
#include <iostream>
using namespace std;

int main(){
  const int dimx=3, dimy=10;

  double a[dimy][dimx];
```

```
   for (int i=0;i<dimy;i++){
     for (int j=0;j<dimx;j++){
        a[i][j]=0;
     }
   }

  a[0][0]=1;
  a[3][2]=2;
  a[9][2]=3;

   for (int i=0;i<dimy;i++){
     for (int j=0;j<dimx;j++){
       cout << a[i][j] << ", ";
     }
     cout << endl;
   }
   return 0;
}
```

Inicializar los elementos a cero inicialmente es particularmente relevante. Si no, la matriz se llenaría con elementos aleatorios.

También es posible definir arreglos bidimensionales dinámicamente. En el siguiente ejemplo, se define una matriz de 200 filas y 400 columnas, inicializándose sus elementos a cero, y finalmente se borra:

```
int main()
{

  int width;
  int height;

  width = 200;
  height = 400;


  double ** matriz = new double * [width];

  for (int i=0;i<width;i++){
    matriz[i] = new double[height];
  }

  for (int i=0;i<width;i++){
    for (int j=0;j<height;j++){
      matriz[i][j] = 0;
    }
```

```
  }

  for (int i=0;i<width;i++){
    delete [] matriz[i];
  }
  delete [] matriz;
  return 0;
}
```

Primero se crea, con `new`, un puntero (`matriz`) de dimensión 200, que representará las filas. Cada uno de sus elementos (`matriz[i]`), a su vez, será un nuevo puntero, de dimensión 400, representando cada columna. Por tanto, `matriz` debe ser un puntero a puntero (de dobles, en este caso), y así es definido inicialmente (`double ** ...`). Esto puede parecer extraño a primera vista, pero recordemos que los punteros pueden ser punteros a cualquier objeto, en particular a otro puntero. Luego se crean los punteros de cada columna (primer ciclo `for`). A continuación se llenan los elementos con ceros (segundo ciclo `for`). Finalmente, se libera la memoria, en orden inverso a como fue asignada: primero se libera la memoria de cada columna de la matriz (`delete [] matriz[i]`, tercer ciclo `for`), y por último se libera la memoria del puntero a estos punteros (`delete [] matriz`).

3.5.5. Matrices de caracteres: cadenas (strings).

Una palabra, frase o texto más largo es representado internamente por C++ como una matriz de `char`s. A esto se le llama "cadena" (string). Sin embargo, esto ocasiona un problema, pues las matrices deben ser definidas con dimensión constante (a menos que sean definidas dinámicamente), y las palabras pueden tener longitud arbitraria. La convención de C++ para resolver el problema es aceptar que una cadena tiene longitud arbitraria, pero debe indicar dónde termina. Esto se hace con el `char` nulo: `'\0'`. Así, para asignar a la variable `palabra` el valor "Hola", debe definirse como una matriz de dimensión 5 (una más que el número de letras):

```
char palabra[5] = {'H', 'o', 'l', 'a', '\0'};
```

Para escribir "Hola" en pantalla basta recorrer los elementos de `palabra` uno a uno:

```
for (i = 0; i < 5; i++)
{
    cout << palabra[i];
}
```

Si tuviéramos que hacer esto cada vez que queremos escribir algo a pantalla no sería muy cómodo. Por ello, también podemos escribir "Hola" en pantalla simplemente con `cout << "Hola"`, y de hecho ése fue el primer ejemplo de este capítulo. De hecho, la declaración de `palabra` podría haberse escrito:

```
char palabra[5] = "Hola";
```

Esto ya es bastante más cómodo, aunque persiste la inconsistencia de definir `palabra` con dimensión 5, cuando en realidad al lado derecho de la asignación hay un objeto con sólo 4 elementos (visibles).

Éste y otros problemas asociados con el manejo convencional de cadenas en C++ se resuelven incluyendo el *header* `string`.

Usando string.

El código anterior se puede reescribir:

```
#include <iostream>
#include <string>
using namespace std;

int main(){
  string palabra = "Hola";
  cout << palabra << endl;
  return 0;
}
```

Observar que la línea a incluir es `#include <string>`, *sin la extensión ".h"*. Al incluir `string`, las cadenas pueden ser declaradas como objetos tipo `string` en vez de arreglos de `char`. El hecho de que ya no tengamos que definir a priori la dimensión de la cadena es una gran ventaja. De hecho, permite ingresar palabras desde el teclado trivialmente, sin preocuparse de que el *input* del usuario sea demasiado grande (tal que supere la dimensión del arreglo que podamos haber declarado inicialmente) o demasiado corto (tal que se traduzca en un despilfarro de memoria por reservar más memoria para el arreglo de la que realmente se necesita):

```
#include <iostream>
#include <string>
using namespace std;

int main(){
  string palabra;
  cin >> palabra;
  return 0;
}
```

Además, este nuevo tipo `string` permite acceder a un sin número de funciones adicionales que facilitan enormemente el manejo de cadenas. Por ejemplo, las cadenas se pueden sumar, donde la suma de cadenas `a` y `b` está definida (siguiendo la intuición) como la cadena que resulta de poner `b` a continuación de `a`:

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main(){
  string texto1 = "Primera palabra";
  string texto2 = "Segunda palabra";
  cout << texto1 << endl << texto2 << endl;
  cout << texto1 + ", " + texto2  << endl;
  // La ultima linea es equivalente a:
  // string texto3 = texto1 + ", " + texto2;
  // cout << texto3 << endl;
  return 0 ;
}
```

El *output* de este programa será:

```
Primera palabra
Segunda palabra
Primera palabra, Segunda palabra
```

Ingreso con espacios.

Dijimos que es muy fácil ingresar una cadena desde el teclado, pues no es necesario definir la dimensión desde el comienzo. Sin embargo, el código anterior, usando `cin`, no es muy general, porque el *input* termina cuando el usuario ingresa el primer cambio de línea o el primer espacio. Esto es muy cómodo cuando queremos ingresar una serie de valores (por ejemplo, para llenar un arreglo), pues podemos ingresarlos ya sea en la forma: `1<Enter> 2<Enter> 3<Enter>`, etc., o `1 2 3`, etc, pero no es óptimo cuando deseamos ingresar texto, que podría constar de más de una palabra y, por tanto, necesariamente incluiría espacios (por ejemplo, al ingresar el nombre y apellido de una persona). Sin explicar demasiado por qué, digamos que la solución a este problema es utilizar una función asociada a `cin` llamada `get`, y leer desde el teclado hasta que el usuario dé el primer cambio de línea. Un ejemplo simple lo encontramos en el siguiente código:

```
#include <iostream>
#include <string>
using namespace std;

int main(){
  string texto1 = "El resultado es: " ;
  string texto2 ="";
  char ch;

  cout << "Entre un string:" << endl;
  while( ( ch = cin.get() ) != '\n' )  texto2 = texto2 + ch;

  cout << texto1 + texto2 << endl;
  return 0;
}
```

Observamos que `cin.get()` no necesita argumento y devuelve un `char` el cual es primero comparado con el caracter de fin de línea y luego acumulado en `texto2`.

3.6. Manejo de archivos.

Una operación usual en todo tipo de programas es la interacción con archivos. Ya sea que el programa necesite conocer ciertos parámetros de configuración, hacer análisis estadístico sobre un gran número de datos generados por otro programa, entregar las coordenadas de los puntos de una trayectoria para graficarlos posteriormente, etc., lo que se requiere es un modo de ingresar datos desde, o poner datos en, archivos. En C++ ello se efectúa incluyendo el *header* `fstream`.

3.6.1. Archivos de salida.

Observemos el siguiente programa:

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){
  ofstream nombre_logico("nombre_fisico.dat");
  int i = 3, j;
  cout << i << endl;
  nombre_logico << i << endl;
  cout << "Ingrese un numero entero: ";
  cin >> j;
  cout << j << endl;
  nombre_logico << j << endl;
  nombre_logico.close();
  return 0;
}
```

La primera línea de `main` define un objeto de tipo `ofstream` (*output file stream*). Esto corresponde a un archivo de salida. Dentro de `main` este archivo será identificado por una variable llamada `nombre_logico`, y corresponderá a un archivo en el disco duro llamado `nombre_fisico.dat`. Naturalmente, el identificador `nombre_logico` puede ser cualquier nombre de variable válido para C++, y `nombre_fisico.dat` puede ser cualquier nombre de archivo válido para el sistema operativo. En particular, se pueden también dar nombres que incluyan *paths* absolutos o relativos:

```
ofstream nombre_logico_1("/home/vmunoz/temp/nombre_fisico.dat");
ofstream nombre_logico_2("../nombre_fisico.dat");
```

Cuando creamos un objeto del tipo archivo, sin importar si es de salida o de entrada, podemos inicializarlo con un nombre de archivo físico. Este nombre lo podemos almacenar

previamente en una variable de `string`, llamemosla `mi_nombre_archivo`. En este caso, cuando creamos el objeto `ofstream` debemos usar un método del objeto `string` que devuelve un puntero a `char`, para poder inicializar el objeto `ofstream`. Veamos la sintáxis explícitamente

```
string mi_nombre_archivo=''archivo.txt'';
ofstream nombre_logico_1( mi_nombre_archivo.c_str());
```

Las líneas tercera y sexta de `main` envían a `nombre_logico` (es decir, escribe en `nombre_fisico.dat`), las variables `i` y `j`. Observar la analogía que existe entre estas operaciones y las que envían la misma información a pantalla.[4] Si ejecutamos el programa y en el teclado ingresamos el número `8`, al finalizar la ejecución el archivo `nombre_fisico.dat` tendrá los dos números escritos:

```
3
8
```

Finalmente, el archivo creado debe ser cerrado (`nombre_logico.close()`). Si esta última operación se omite en el código, no habrá errores de compilación, y el programa se encargará de cerrar por sí solo los archivos abiertos durante su ejecución, pero un buen programador debiera tener cuidado de cerrarlos explícitamente. Por ejemplo, un mismo programa podría desear utilizar un mismo archivo más de una vez, o varios programas podrían querer acceder al mismo archivo, y si no se ha insertado un `close` en el punto adecuado esto podría provocar problemas.

El archivo indicado al declarar la variable de tipo `ofstream` tiene modo de escritura, para permitir la salida de datos hacia él. Si no existe un archivo llamado `nombre_fisico.dat` es creado; si existe, los contenidos antiguos se pierden y son reemplazados por los nuevos. No siempre deseamos este comportamiento. A veces deseamos agregar la salida de un programa a un archivo de texto ya existente. En ese caso la declaración del archivo es diferente, para crear el archivo en modo *"append"*:

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){

  ofstream nombre_logico("nombre_fisico.dat",ios::app);
  int i = 3;

  nombre_logico << i << endl;
  nombre_logico.close();
  return 0;
}
```

[4] Esta analogía no es casual y se entiende con el concepto de *clases* (Sec. 3.8). `fstream` e `iostream` definen clases que heredan sus propiedades de un objeto abstracto base, común a ambas, y que en el caso de `iostream` se concreta en la salida estándar —pantalla—, y en el de `fstream` en un archivo.

Si ejecutamos este programa y el archivo `nombre_fisico.dat` no existe, será creado. El resultado será un archivo con el número `3` en él. Al ejecutarlo por segunda vez, los datos se ponen a continuación de los ya existentes, resultando el archivo con el contenido:

```
3
3
```

La línea del tipo `ofstream a("b")` es equivalente a una del tipo `int i=3`, declarando una variable (`a`/`i`) de un cierto tipo (`ofstream`/`int`) y asignándole un valor simultáneamente `"b"`/`3`. Como para los tipos de variables predefinidos de C++, es posible separar declaración y asignación para una variable de tipo `ofstream`:

```
ofstream a;
a.open("b");
```

es equivalente a `ofstream a("b")`. Esto tiene la ventaja de que podríamos usar el mismo nombre lógico para identificar dos archivos físicos distintos, usados en distintos momentos del programa:

```
ofstream a;
a.open("archivo1.txt");
// Codigo en que "archivo1.txt" es utilizado
a.close();
a.open("archivo2.txt");
// Ahora "archivo2.txt" es utilizado
a.close();
```

Observar la necesidad del primer `close`, que permitirá liberar la asociación de `a` a un nombre físico dado, y reutilizar la variable lógica en otro momento.

En los ejemplos hemos escrito solamente variables de tipo `int` en los archivos. Esto por cierto no es restrictivo. Cualquiera de los tipos de variables de C++ —`float`, `double`, `char`, etc.— se puede enviar a un archivo del mismo modo. Dicho esto, en el resto de esta sección seguiremos usando como ejemplo el uso de `int`.

3.6.2. Archivos de entrada.

Ya sabemos que enviar datos a un archivo es tan fácil como enviarlos a pantalla. ¿Cómo hacemos ahora la operación inversa, de leer datos desde un archivo? Como es de esperar, es tan fácil como leerlos desde el teclado. Para crear un archivo en modo de lectura, basta declararlo de tipo `ifstream` (*input file stream*). Por ejemplo, si en `nombre_logico.dat` tenemos los siguientes datos:

```
3
6
9
12
```

el siguiente programa,

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){

  ifstream nombre_logico("nombre_fisico.dat");
  int i, j,k,l;

  nombre_logico >> i >> j >> k >> l;

  cout << i << "," << j << "," << k << "," << l << endl;

  nombre_logico.close();
  return 0;
}
```

será equivalente a asignar `i=3`, `j=6`, `k=9`, `l=12`, y luego enviar los datos a pantalla. Observar que la sintaxis para ingresar datos desde un archivo, `nombre_logico >> i`, es idéntica a `cin >> i`, para hacerlo desde el teclado. Al igual que `cin`, espacios en blanco son equivalentes a cambios de línea, de modo que el archivo podría haber sido también:

```
3 6 9 12
```

Por cierto, el ingreso de datos desde un archivo se puede hacer con cualquier técnica, por ejemplo, usando un `for`:

```
  ifstream nombre_logico("nombre_fisico.dat");
  int i;
  for (int j=0;j<10;j++){
    nombre_logico >> i;
    cout << i << ",";
  }
  nombre_logico.close();
}
```

Como con `ofstream`, es posible separar declaración e implementación:

```
ifstream a;
a.open("b");
a.close();
```

3.6.3. Archivos de entrada y salida.

Ocasionalmente nos encontraremos con la necesidad de usar un mismo archivo, en el mismo programa, a veces para escribir datos, y otras veces para leer datos. Por ejemplo, podríamos tener una secuencia de datos en un archivo, leerlos, y de acuerdo al análisis de

esos datos agregar más datos a continuación del mismo archivo, o reemplazar los datos ya existentes con otros. Necesitamos entonces un tipo de variable flexible, que pueda ser usado como entrada y salida. Ese tipo es `fstream`. Todo lo que hemos dicho para `ofstream` y `ifstream` por separado es cierto simultáneamente para `fstream`.[5] Para especificar si el archivo debe ser abierto en modo de escritura o lectura, `open` contiene el argumento `ios::out` o `ios::in`, respectivamente. Por ejemplo, el siguiente código escribe el número `4` en un archivo, y luego lo lee desde el mismo archivo:

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){
  fstream nombre_logico;
  nombre_logico.open("nombre_fisico.dat",ios::out);
  int i = 4,j;

  nombre_logico << i << endl;
  nombre_logico.close();

  nombre_logico.open("nombre_fisico.dat",ios::in);

  nombre_logico >> j;
  cout << j << endl;
  nombre_logico.close();
  return 0;
}
```

Las dos primeras líneas de `main` separan declaración y asignación, y son equivalentes a `fstream nombre_logico("nombre_fisico.dat",ios::out);`, pero lo hemos escrito así para hacer evidente la simetría entre el uso del archivo como salida primero y como entrada después.

De lo anterior, se deduce que:

```
fstream archivo_salida("salida.dat",ios::out);
fstream archivo_entrada("entrada.dat",ios::in);
```

es equivalente a

```
ofstream archivo_salida("salida.dat");
ifstream archivo_entrada("entrada.dat");
```

[5] Nuevamente, este hecho se debe al concepto de clases que subyace a las definiciones de estos tres tipos de variables; `fstream` es una clase derivada a la vez de `ofstream` y de `ifstream`, heredando las propiedades de ambas.

3.7. main como función.

Para ejecutar un programa compilado en C++, escribimos su nombre en el *prompt*:

```
user@host:~/$ programa
```

Si el mismo usuario desea ejecutar alguno de los comandos del sistema operativo, debe hacer lo mismo:

```
user@host:~/$ ls
```

Sin embargo, `ls` es en realidad el nombre de un archivo ejecutable en el directorio `/bin`, de modo que en realidad no hay diferencias entre nuestro programa y un comando del sistema operativo en ese sentido. Sin embargo, éstos pueden recibir argumentos y opciones. Por ejemplo, para ver todos los archivos que comienzan con `l` en el directorio local basta con darle a `ls` el argumento `l*`: `ls l*`. Si queremos ordenar los archivos en orden inverso de modificación, basta dar otro argumento, en forma de opción: `ls -tr l*`. Se ve entonces que los argumentos de un archivo ejecutable permiten modificar el comportamiento del programa de modos específicos.

¿Es posible hacer lo mismo con archivos ejecutables hechos por el usuario? La respuesta es sí, y para eso se usan los argumentos del `main`. Recordemos que `main` es una función, pero hasta el momento no hemos aprovechado esa característica. Simplemente sabemos que el programa empieza a ejecutarse en la línea donde está la función `main`. Además, siempre hemos escrito esa línea como `main()`. Sin embargo, `main`, como cualquier función, es capaz de aceptar argumentos. Específicamente, acepta dos argumentos, el primero es un entero (que cuenta el número de argumentos que `main` recibió), y el segundo es un puntero a un arreglo de caracteres (que contiene los distintos argumentos, en forma de cadenas de caracteres, que se le entregaron).

Por ejemplo:

```
#include <iostream>
using namespace std;

int main( int argc, char * argv[])
{
     for(int i = 0; i < argc; i++) {
         cout << argv[i] << endl ;
     }
     return 0;
}
```

Si llamamos a este programa `argumentos`, obtenemos distintas salidas al llamarlo con distintos argumentos:

```
user@host:~/$ argumentos
argumentos
user@host:~/$ argumentos ap k 5
```

```
argumentos
ap
k
5
user@host:~/$ argumentos -t -s 4 arg1
argumentos
-t
-s
4
arg1
```

Observar que el primer argumento del programa es siempre el nombre del propio programa. Naturalmente, éste es un ejemplo muy simple. Es tarea del programador decidir cómo manejar cada una de las opciones o argumentos que se le entregan al programa desde la línea de comandos, escribiendo el código correspondiente.

3.7.1. Tipo de retorno de la función `main`.

Un segundo aspecto con el cual no hemos sido sistemáticos es que `main`, como toda función, tiene un tipo de retorno. En el caso de `main`, ese tipo debe ser `int`. Este `int` es entregado al sistema operativo, y puede servir para determinar si el programa se ejecutó con normalidad o si ocurrió algo anormal. Podríamos hacer ese valor de retorno igual a 0 o 1, respectivamente. Así, la siguiente estructura es correcta:

```
int main(){
   // Codigo



   return 0;
}
```

En este caso, el programa entrega siempre el valor 0 al sistema operativo.

Los códigos del tipo:

```
main(){
   // Codigo



}
```

o

```
void main(){
   // Codigo
```

```
}
```

también compilan, pero el compilador emite una advertencia si es llamado con la opción `-Wall` (*Warning all*). En el primer caso, la advertencia es:

```
warning: ANSI C++ forbids declaration ‘main’ with no type
```

En el segundo:

```
return type for ‘main’ changed to ‘int’
```

En general, siempre es conveniente compilar con la opción `-Wall`, para lograr que nuestro código esté realmente correcto (`g++ -Wall <archivo>.cc -o <archivo>`).

3.8. Clases.

C++ dispone de una serie de tipos de variables con las cuales nos está permitido operar: `int`, `double`, `char`, etc. Creamos variables de estos tipos y luego podemos operar con ellas:

```
int x, y;
x = 3;
y = 6;
int z = x + y;
```

No hay, sin embargo, en C++, una estructura predefinida que corresponda a números complejos, vectores de dimensión n o matrices, por ejemplo. Y sin embargo, nos agradaría disponer de números complejos que pudiéramos definir como

```
z = (3,5);
w = (6,8);
```

y que tuvieran sentido las expresiones

```
a = z + w;
b = z * w;
c = z / w;
d = z + 3;
e = modulo(z);
f = sqrt(z);
```

Todas estas expresiones son completamente naturales desde el punto de vista matemático, y sería bueno que el lenguaje las entendiera. Esto es imposible en el estado actual, pues, por ejemplo, el signo `+` es un operador que espera a ambos lados suyos un número. Sumar cualquier cosa con cualquier cosa no significa nada necesariamente, así que sólo está permitido operar con números. Pero los humanos sabemos que los complejos son números. ¿Cómo decírselo al computador? ¿Cómo convencerlo de que sumar vectores o matrices es también posible matemáticamente, y que el mismo signo `+` debería servir para todas estas operaciones?

La respuesta es: a través del concepto de *clases*. Lo que debemos hacer es definir una clase de números complejos. Llamémosla `Complejo`. Una vez definida correctamente, `Complejo` será un tipo más de variable que el compilador reconocerá, igual que `int`, `double`, `char`, etc. Y será tan fácil operar con los `Complejos` como con todos los tipos de variables preexistentes. Esta facilidad es la base de la extensibilidad de que es capaz C++, y por tanto de todas las propiedades que lo convierten en un lenguaje muy poderoso.

Las clases responden a la necesidad del programador de construir objetos o tipos de datos que respondan a sus necesidades. Si necesitamos trabajar con vectores de 5 coordenadas, será natural definir una clase que corresponda a vectores con 5 coordenadas; si se trata de un programa de administración de personal, la clase puede corresponder a un empleado, con sus datos personales como elementos.

Si bien es cierto uno puede trabajar con clases en el contexto de orientación al procedimiento, las clases muestran con mayor propiedad su potencial con la orientación al objeto, donde cada objeto corresponde a una clase. Por ejemplo, para efectuar una aplicación para `X-windows`, la ventana principal, las ventanas de los archivos abiertos, la barra de menú, las cajas de diálogo, los botones, etc., cada uno de estos objetos estará asociado a una clase.

3.8.1. Definición.

Digamos que queremos una clase para representar a los empleados de una empresa. Llamémosla `Persona`. La convención aceptada es que los nombres de las clases comiencen con mayúscula. Esto es porque las clases, recordemos, corresponderán a tipos de variables tan válidos como los internos de C++ (`int`, `char`, etc.). Al usar nombres con mayúscula distinguimos visualmente los nombres de un tipo de variable interno y uno definido por el usuario.

La estructura mínima de la definición de la clase `Persona` es:

```
class Persona
{

};
```

Todas las características de la clase se definen entre los paréntesis cursivos.

3.8.2. Miembros.

Se denomina *miembros* de una clase a todas las variables y funciones declaradas dentro de una clase. Por ejemplo, para personas, es natural caracterizarlas por su nombre y su edad. Y si se trata de empleados de una empresa, es natural también tener una función que entregue su sueldo:

```
class Persona
{
     string nombre;
     fecha nacimiento;
     int rut;
```

```
     double edad();
};
```

Los miembros de una clase pueden tener cualquier nombre, excepto el nombre de la propia clase dentro de la cual se definen, ese nombre está reservado.

3.8.3. Miembros públicos y privados.

Una clase distingue información (datos o funciones) privada (accesible sólo a otros miembros de la misma clase) y pública (accesible a funciones externas a la clase). La parte privada corresponde a la estructura interna de la clase, y la parte pública a la implementación (típicamente funciones), que permite la interacción de la clase con el exterior.

Consideremos ahora nuestro deseo de tener una clase que represente números complejos. Un número complejo tiene dos números reales (parte real e imaginaria), y ésos son elementos privados, es decir, parte de su estructura interna. Sin embargo, nos gustaría poder modificar y conocer esas cantidades. Eso sólo puede hacerse a través de funciones públicas.

```
class Complejo
{
private:
    double real, imaginaria;
public:
    void setreal(double);
    void setimag(double);
    double getreal();
    double getimag();
};
```

En este ejemplo, los miembros privados son sólo variables, y los miembros públicos son sólo funciones. Éste es el caso típico, pero puede haber variables y funciones de ambos tipos.

3.8.4. Operador de selección (.).

Hemos definido una clase de números complejos y funciones que nos permiten conocer y modificar las partes real e imaginaria. ¿Cómo se usan estos elementos? Consideremos el siguiente programa de ejemplo:

```
using namespace std;

class Complejo
{
private:
    double real, imaginaria;
public:
    void setreal(double);
    void setimag(double);
```

```
    double getreal();
    double getimag();
};

int main()
{
    Complejo z, w;

    z.setreal(3);
    z.setimag(2.8);
    w.setreal(1.5);
    w.setimag(5);
    cout << "El primer numero complejo es: " << z.getreal()
         << " + i*" << z.getimag() << endl;
    cout << "El segundo es: " << w.getreal() << " + i*"
         << z.getimag() << endl;
    return 0;
}
```

Vemos en la primera línea de `main` cómo la clase `Complejo` se usa del mismo modo que usaríamos `int` o `double`. Ahora `Complejo` es un tipo de variable tan válido como los tipos predefinidos por C++. Una vez definida la variable, el operador de selección (`.`) permite acceder a las funciones públicas correspondientes a la clase `Complejo`, aplicadas a la variable particular que nos interesa: `z.setreal(3)` pone en la parte real del `Complejo z` el número 3, y `w.setreal(1.5)` hace lo propio con `w`.

3.8.5. Implementación de funciones miembros.

Ya sabemos cómo declarar funciones miembros en el interior de la clase y cómo usarlas. Ahora veamos cómo se implementan.

```
void Complejo::setreal(double x)
{
    real = x;
}

void Complejo::setimag(double x)
{
    imaginaria = x;
}

double Complejo::getreal()
{
    return real;
}
```

```
double Complejo::getimag()
{
    return imaginaria;
}
```

Como toda función, primero va el tipo de la función (`void` o `double` en los ejemplos), luego el nombre de la función y los argumentos. Finalmente la implementación. Lo diferente es que el nombre va precedido del nombre de la clase y el operador "`::`" .

3.8.6. Constructor.

Al declarar una variable, el programa crea el espacio de memoria suficiente para alojarla. Cuando se trata de variables de tipos predefinidos en C++ esto no es problema, pero cuando son tipos definidos por el usuario, C++ debe saber cómo construir ese espacio. La función que realiza esa tarea se denomina *constructor*.

El constructor es una función pública de la clase, que tiene el mismo nombre que ella. Agreguemos un constructor a la clase `Complejo`:

```
class Complejo
{
private:
    double real,imaginaria;
public:
    Complejo(double,double);
    void setreal(double);
    void setimag(double);
    double getreal();
    double getimag();
};

Complejo::Complejo (double x, double y)
: real(x), imaginaria(y)
{}
```

Definir el constructor de esta manera nos permite crear en nuestro programa variables de tipo `Complejo` y asignarles valores sin usar `setreal()` o `setimag()`:

```
Complejo z (2, 3.8);
Complejo w = Complejo(6.8, -3);
```

En el constructor se inicializan las variables internas que nos interesa inicializar al momento de crear un objeto de esta clase.

Si una de las variables internas a inicializar es una cadena de caracteres, hay que inicializarla de modo un poco distinto. Por ejemplo, si estamos haciendo una clase `OtraPersona` que sólo tenga el nombre de una persona, entonces podemos definir la clase y su constructor en la forma:

```
class OtraPersona
{
private:
    char nombre[20];
public:
    Persona(char []);
};
Persona::Persona(char a[])
{
   strcpy(nombre,a);
}
```

Si uno no especifica el constructor de una clase C++ crea uno *default*, pero en general será insuficiente para cualquier aplicación realmente práctica. Es una mala costumbre ser descuidado y dejar estas decisiones al computador.

3.8.7. Destructor.

Así como es necesario crear espacio de memoria al definir una variable, hay que deshacerse de ese espacio cuando la variable deja de ser necesaria. En otras palabras, la clase necesita también un *destructor*. Si la clase es `Complejo`, el destructor es una función pública de ella, llamada `~Complejo`.

```
class Complejo
{
private:
    double real, imaginaria;
public:
    Complejo(double,double);
    ~Complejo();
    void setreal(double);
    void setimag(double);
    double getreal();
    double getimag();
};

Complejo::Complejo (double x, double y): real(x), imaginaria(y)
{
}

Complejo::~Complejo()
{
}
```

Como con los constructores, al omitir un destructor C++ genera un *default*, pero es una mala costumbre. . . , etc.

3.8.8. Arreglos de clases.

Una clase es un tipo de variable como cualquier otra de las predefinidas en C++. Es posible construir matrices con ellas, del mismo modo que uno tiene matrices de enteros o caracteres. La única diferencia con las matrices usuales es que no se pueden sólo declarar, sino que hay que inicializarlas simultáneamente. Por ejemplo, si queremos crear una matriz que contenga 2 números complejos, la línea

```
Complejo z[2];
```

es incorrecta, pero sí es aceptable la línea

```
Complejo z[2] = {Complejo(3.5,-0.8), Complejo(-2,4)};
```

3.9. Sobrecarga.

Para que la definición de nuevos objetos sea realmente útil, hay que ser capaz de hacer con ellos muchas acciones que nos serían naturales. Como ya comentamos al introducir el concepto de clase, nos gustaría sumar números complejos, y que esa suma utilizara el mismo signo `+` de la suma usual. O extraerles la raíz cuadrada, y que la operación sea tan fácil como escribir `sqrt(z)`. Lo que estamos pidiendo es que el operador `+` o la función `sqrt()` sean *polimórficos*, es decir, que actúen de distinto modo según el tipo de argumento que se entregue. Si `z` es un real, `sqrt(z)` calculará la raíz de un número real; si es complejo, calculará la raíz de un número complejo.

La técnica de programación mediante la cual podemos definir funciones polimórficas se llama *sobrecarga*.

3.9.1. Sobrecarga de funciones.

Digamos que la raíz cuadrada de un número complejo $a + ib$ es $(a/2) + i(b/2)$. (Es más complicado en realidad, pero no queremos escribir las fórmulas ahora.)

Para sobrecargar la función `sqrt()` de modo que acepte números complejos basta definirla así:

```
Complejo sqrt(Complejo z)
{
     return Complejo (z.getreal()/2, z.getimag()/2);
}
```

Observemos que definimos una función `sqrt` que acepta argumentos de tipo `Complejo`, y que entrega un número del mismo tipo. Cuando pidamos la raíz de un número, el computador se preguntará si el número en cuestión es un `int`, `double`, `float` o `Complejo`, y según eso escogerá la versión de `sqrt` que corresponda.

Con la definición anterior podemos obtener la raíz cuadrada de un número complejo simplemente con las instrucciones:

```
Complejo z(1,3);
Complejo raiz = sqrt(z);
```

3.9.2. Sobrecarga de operadores.

¿Cómo le decimos al computador que el signo + también puede aceptar números complejos? La respuesta es fácil, porque para C++ un operador no es sino una función, y la acción de sobrecargar que ya vimos sirve en este caso también. La sintaxis es:

```
Complejo operator + (Complejo z, Complejo w)
{
    return Complejo (z.getreal() + w.getreal(),
                     z.getimag() + w.getimag());
}
```

3.9.3. Coerción.

Sabemos definir $a+b$, con a y b complejos. Pero ¿qué pasa si a o b son enteros? ¿O reales? Pareciera que tendríamos que definir no sólo

```
Complejo operator + (Complejo a, Complejo b);
```

sino también todas las combinaciones restantes:

```
Complejo operator + (Complejo a, int b);
Complejo operator + (Complejo a, float b);
Complejo operator + (int a, Complejo b);
```

etcétera.

En realidad esto no es necesario. Por cierto, un número real es un número complejo con parte imaginaria nula, y es posible hacerle saber esto a C++, usando la posibilidad de definir funciones con parámetros *default*. Basta declarar (en el interior de la clase) el constructor de los números complejos como

```
Complejo (double, double = 0);
```

Esto permite definir un número complejo con la instrucción:

```
Complejo c = Complejo(3.5);
```

resultando el número complejo $3.5 + i \cdot 0$. Y si tenemos una línea del tipo:

```
Complejo c = Complejo(3,2.8) + 5;
```

el computador convertirá implícitamente el entero 5 a `Complejo` (sabe cómo hacerlo porque el constructor de números complejos acepta también un solo argumento en vez de dos), y luego realizará la suma entre dos complejos, que es entonces la única que es necesario definir.

3.10. Herencia.

Herencia es el mecanismo mediante el cual es posible definir clases a partir de otras, preservando parte de las propiedades de la primera y agregando o modificando otras.

Por ejemplo, si definimos la clase `Persona`, toda `Persona` tendrá una variable miembro que sea su `nombre`. Si definimos una clase `Hombre`, también será `Persona`, y por tanto debería tener `nombre`. Pero además puede tener `esposa`. Y ciertamente no toda `Persona` tiene `esposa`. Sólo un `Hombre`.

C++ provee mecanismos para implementar estas relaciones lógicas y poder definir una clase `Hombre` a partir de `Persona`. Lo vemos en el siguiente ejemplo:

```
class Persona
{
private:
    string nombre;
public:
    Persona(string = "");
    ~Persona();
    string getname();
}

class Hombre : public Persona
{
private:
    string esposa;
public:
    Hombre(string a) : Persona(a)
    { };
    string getwife();
    void setwife(string);
};
```

Primero definimos una clase `Persona` que tiene `nombre`. Luego definimos una clase `Hombre` *a partir de* `Persona` (con la línea `class Hombre : public Persona`). Esto permite de modo automático que `Hombre` tenga también una variable `nombre`. Y finalmente, dentro de la clase `Hombre`, se definen todas aquellas características adicionales que una `Persona` no tiene pero un `Hombre` sí: `esposa`, y funciones miembros para modificar y obtener el nombre de ella.

Un ejemplo de uso de estas dos clases:

```
Persona cocinera("Maria");
Hombre panadero("Claudio");
panadero.setwife("Estela");

cout << cocinera.getname() << endl;
cout << panadero.getname() << endl;
cout << panadero.getwife() << endl;
```

Observemos que `panadero` también tiene una función `getname()`, a pesar de que la clase `Hombre` no la define explícitamente. Esta función se ha *heredado* de la clase de la cual `Hombre` se ha derivado, `Persona`.

3.11. Ejemplo: la clase de los complejos.

A continuación, una clase de complejos más completa. Primero el archivo de *headers* `complejos.h`, con las definiciones:

```
#ifndef _complejos_
#define _complejos_

#include <iostream>
#include <cmath>

class Complejo {
 private:
   double real, imaginaria;
 public:
   Complejo();
   Complejo(double,double=0);
   ~Complejo();
   void setreal(double);
   void setimag(double);
   double getreal();
   double getimag();
   double getmodule();
   double getmodule2();
   double getphase();
};

Complejo operator + (Complejo,Complejo);
Complejo operator - (Complejo,Complejo);
Complejo operator * (Complejo,Complejo);
Complejo operator / (Complejo,Complejo);
Complejo conjugate(Complejo);
Complejo inverse(Complejo);
Complejo sqrt(Complejo);
Complejo log(Complejo);
bool operator == (Complejo,Complejo);
bool operator != (Complejo,Complejo);

std::ostream & operator << (std::ostream &, Complejo);

#endif
```

y la implementación de lo anterior

```
#include "complejos.h"

Complejo::Complejo(double x, double y)
  :real(x), imaginaria(y)
{}

Complejo::Complejo()
  :real(0), imaginaria(0)
{}

Complejo::~Complejo()
{}

void Complejo::setreal(double x)
{
   real = x;
}

void Complejo::setimag(double x)
{
   imaginaria = x;
}

double Complejo::getreal()
{
   return real;
}

double Complejo::getimag()
{
   return imaginaria;
}

double Complejo::getmodule()
{
   return sqrt(real*real+imaginaria*imaginaria);
}

double Complejo::getmodule2()
{
   return real*real+imaginaria*imaginaria;
}

double Complejo::getphase()
```

```
{
   return atan2(real,imaginaria);
}


Complejo operator + (Complejo z,Complejo w)
{
   return Complejo(z.getreal()+w.getreal(), z.getimag()+w.getimag());
}

Complejo operator - (Complejo z,Complejo w)
{
   return Complejo(z.getreal()-w.getreal(), z.getimag()-w.getimag());
}

Complejo operator * (Complejo z,Complejo w)
{
   return Complejo(z.getreal()*w.getreal()- z.getimag()+w.getimag(),
                   z.getreal()*w.getimag()+ z.getimag()+w.getreal());
}

Complejo operator / (Complejo z,Complejo w)
{
   return z*inverse(w);
}

Complejo conjugate(Complejo z)
{
    return Complejo(z.getreal(), -z.getimag());
}

Complejo inverse(Complejo z)
{
    return Complejo(z.getreal()/z.getmodule2(), -z.getimag()/z.getmodule2());
}

Complejo sqrt(Complejo z)
{
    return Complejo(sqrt(z.getmodule())*cos(z.getphase()/2.0),
                    sqrt(z.getmodule())*sin(z.getphase()/2.0)) ;
}
Complejo log(Complejo z)
{
    return Complejo(log(z.getmodule()), z.getphase());
}
```

```
bool operator == (Complejo z,Complejo w)
{
   return bool(z.getreal()==w.getreal() && z.getimag()==w.getimag());
}


bool operator != (Complejo z,Complejo w)
{
   return bool(z.getreal()!=w.getreal() || z.getimag()!=w.getimag());
}


std::ostream & operator << (std::ostream & os , Complejo z)
{
   os << z.getreal();
   if(z.getimag() !=0) os << " + i*"<< z.getimag();
   return os;
}
```

3.12. Compilación y debugging.

3.12.1. Compiladores.

El comando para usar el compilador de lenguaje C es `gcc`, para usar el compilador de C++ es `g++` y para usar el compilador de fortran 77 es `g77`. Centrémosnos en el compilador de C++, los demás funcionan en forma muy similar. Su uso más elemental es:
`g++ filename.cc`
Esto compila el archivo `filename.cc` y crea un archivo ejecutable que se denomina `a.out` por omisión. Existen diversas opciones para el compilador, sólo comentaremos una pocas.

- `-c` realiza sólo la compilación pero no el *link*:
  `g++ -c filename.cc`
  genera el archivo `filename.o` que es código objeto.

- `-o exename` define el nombre del ejecutable creado, en lugar del por defecto `a.out`.
  `g++ -o outputfile filename.cc`

- `-lxxx` incluye la biblioteca `/usr/lib/libxxx.a` en la compilación.
  `g++ filename.cc -lm`
  En este caso se compila con la biblioteca matemática `libm.a`.

- `-g` permite el uso de un debugger posteriormente.

- `-On` optimización de grado `n` que puede tomar valores de 1 (por defecto) a 3. El objetivo inicial del compilador es reducir el tiempo de la compilación. Con `-On`, el compilador

trata de reducir el tamaño del ejecutable y el tiempo de ejecución, con `n` se aumenta el grado de optimización.

- `-Wall` notifica todos los posibles *warnings* en el código que está siendo compilado.
- `-L/path1 -I/path2/include` incluye en el camino de búsqueda `/path1/` para las bibliotecas y `/path2/include` para los archivos de cabecera (*headers*).

El compilador `gcc` (*the GNU C compiler*) es compatible ANSI.

3.13. make & Makefile.

Frecuentemente los programas están compuestos por diferentes subprogramas que se hayan contenidos en diferentes archivos. La orden de compilación necesaria puede ser muy engorrosa, y a menudo no es necesario volver a compilar todos los archivos, sino sólo aquellos que hayan sido modificados. UNIX dispone de una orden denominada `make` que evita los problemas antes mencionados y permite el mantenimiento de una biblioteca personal de programas. Este comando analiza qué archivos fuentes han sido modificados después de la última compilación y evita recompilaciones innecesarias.

En su uso más simple sólo es necesario suministrar una lista de dependencias y/o instrucciones a la orden `make` en un archivo denominado `Makefile`. Una dependencia es la relación entre dos archivos de forma que un archivo se considera actualizado siempre que el otro tenga una fecha de modificación inferior a éste. Por ejemplo, si el archivo `file.cc` incluye el archivo `file.h`, no se puede considerar actualizado el archivo `file.o` si el archivo `file.cc` o el archivo `file.h` ha sido modificado después de la última compilación. Se dice que el archivo `file.o` depende de `file.cc` y el archivo `file.cc` depende del archivo `file.h`. El Makefile se puede crear con un editor de texto y tiene el siguiente aspecto para establecer una dependencia:

```
# Esto es un ejemplo de Makefile.
# Se pueden poner comentarios tras un caracter hash (#).

FILE1: DEP1 DEP2
        comandos para generar FILE1
ETIQUETA1: FILE2
FILE2: DEP3 DEP4
        comandos para generar FILE2
ETIQUETA2:
        comandos
```

Se comienza con un destino, seguido de dos puntos (`:`) y los prerrequisitos o dependencias necesarios. También puede ponerse una etiqueta y como dependencia un destino, o bien una etiqueta y uno o más comandos. Si existen muchos prerrequisitos, se puede finalizar la línea con un *backslash* (\) y continuar en la siguiente línea.

En la(s) línea(s) siguiente(s) se escriben uno o más comandos. Cada línea se considera como un comando independiente. Si se desea utilizar múltiples líneas para un comando,

se debería poner un *backslash* (\) al final de cada línea del comando. El comando `make` conectará las líneas como si hubieran sido escritas en una única línea. En esta situación, se deben separar los comandos con un punto y coma (`;`) para prevenir errores en la ejecución de el *shell*. **Los comandos deben ser indentados con un tabulador, no con 8 espacios**

Make lee el `Makefile` y determina para cada archivo destino (empezando por el primero) si los comandos deben ser ejecutados. Cada destino, junto con los prerrequisitos o dependencias, es denominado una regla.

Si `make` se ejecuta sin argumentos, sólo se ejecutará el primer destino. Veamos un ejemplo:

```
file.o: file.cc file.h
        g++ -c file.cc
```

En este caso se comprueban las fechas de las última modificaciones de los archivos `file.cc` y `file.h`; si esta fecha es más reciente que las del archivo `file.o` se procede a la compilación.

El comando `make` se puede suministrar con un argumento, que indica la etiqueta situada a la izquierda de los dos puntos. Así en el ejemplo anterior podría invocarse `make file.o`..

Gracias a las variables, un `Makefile` se puede simplificar significativamente. Las variables se definen de la siguiente manera:

```
VARIABLE1=valor1
VARIABLE2=valor2
```

Una variable puede ser utilizada en el resto del `Makefile` refiriéndonos a ella con la expresión `$(VARIABLE)`. Por defecto, `make` sabe las órdenes y dependencias (reglas implícitas) para compilar un archivo `*.cc` y producir un archivo `*.o`, entonces basta especificar solamente las dependencias que make no puede deducir a partir de los nombres de los archivos, por ejemplo:

```
OUTPUTFILE = prog
OBJS = prog.o misc.o aux.o
INCLUDESMISC = misc.h aux.h
INCLUDESFILE = foo.h $(INCLUDESMISC)
LIBS = -lmylib -lg++ -lm

prog.o: $(INCLUDESFILE)
misc.o: $(INCLUDESMISC)
aux.o: aux.h

$(OUTPUTFILE): $(OBJS)
        gcc $(OBJS) -o $(OUTPUTFILE) $(LIBS)
```

Las reglas patrones son reglas en las cuales se especifican múltiples destinos y construye el nombre de las dependencias para cada blanco basada en el nombre del blanco. La sintaxis de una regla patrón:

```
destinos ... : destino patron:dependencias patrones
        comandos
```

La lista de destinos especifica aquellos sobre los que se aplicará la regla. El destino patrón y las dependencias patrones dicen cómo calcular las dependencias para cada destino. Veamos un ejemplo:

```
objects = foo.o \
          bar.o

all: $(objects)

$(objects): %.o: %.cc
        $(CXX) -c $(CFLAGS) $< -o $@
```

Cada uno de los destinos (`foo.o bar.o`) es comparado con el destino patrón `%.o` para extraer parte de su nombre. La parte que se extrae es conocida como el tronco o *stem*, `foo` y `bar` en este caso. A partir del tronco y la regla patrón de las dependencias `%.cc` `make` construye los nombres completos de ellas (`foo.cc bar.cc`). Además, en el comando del ejemplo anterior aparecen un tipo de variables especiales conocidas como automáticas. La variable `$<` mantiene el nombre de la dependencia actual y la variable `$@` mantiene el nombre del destino actual. Finalmente un ejemplo completo:

```
#
# Makefile para el programa eapp
#_____________________________________________________
CXX = g++
CXXFLAGS = -Wall -O3 -mcpu=i686 -march=i686
#CXXFLAGS = -Wall -g
LIBS = -lm
BIN = eapp

OBJECTS = eapp.o md.o atoms.o vectores.o metalesEA.o Monte_Carlo.o\
        string_fns.o nlista.o rij2.o velver2.o cbc2.o nforce.o \
        temperature.o controles.o inparamfile.o \
        azar.o velver3.o funciones.o observables.o

$(BIN): $(OBJECTS)

        $(CXX) $(OBJECTS) -o $(BIN) $(LIBS)

$(OBJECTS): %.o:%.cc
        $(CXX)  -c $(CXXFLAGS) $< -o $@
clean:
        rm -fr  $(OBJECTS) $(BIN)
#End
```

Capítulo 4

Gráfica.

versión 4.12, 24 de Octubre del 2003

En este capítulo quisiéramos mostrar algunas de las posibilidades gráficas presentes en Linux. Nuestra intensión es cubrir temas como la visualización, conversión, captura y creación de archivos gráficos. Sólo mencionaremos las aplicaciones principales en cada caso centrándonos en sus posibilidades más que en su utilización específica, ya que la mayoría posee una interfase sencilla de manejar y una amplia documentación.

4.1. Visualización de archivos gráficos.

Si disponemos de un archivo gráfico conteniendo algún tipo de imagen lo primero que es importante determinar es en qué tipo de formato gráfico está codificada. Existe un número realmente grande de diferentes tipos de codificaciones de imágenes, cada una de ellas se considera un formato gráfico. Por razones de reconocimiento inmediato del tipo de formato gráfico se suelen incluir en el nombre del archivo, que contiene la imagen, un trío de letras finales, conocidas como la extensión, que representan el formato. Por ejemplo: bmp, tiff, jpg, ps, eps, fig, gif entre muchas otras, si uno quiere asegurarse puede dar el comando:

```
jrogan@huelen:~$file mono.jpg
mono.jpg: JPEG image data, JFIF standard 1.01, resolution (DPCM), 72 x 72
```

¿De qué herramientas disponemos en Linux para visualizar estas imágenes? La respuesta es que en Linux disponemos de variadas herramientas para este efecto.

Si se trata de archivos de tipo *PostScript* o *Encapsulated PostScript*, identificados por la extensión `ps` o `eps`, existen las aplicaciones `gv`, `gnome-gv` o `kghostview`, todos programas que nos permitirán visualizar e imprimir este tipo de archivos. Si los archivos son tipo *Portable Document Format*, con extensión `pdf`, tenemos las aplicaciones `gv`, `acroread` o `xpdf`, Con todas ellas podemos ver e imprimir dicho formato. Una mención especial requieren los archivos *DeVice Independent* con extensión `dvi` ya que son el resultado de la compilación de un documento TEX o LATEX, para este tipo de archivo existen las aplicaciones `xdvi`, `advi`, `gxdvi` y `kdvi` por nombrar las principales. La aplicación `xdvi` sólo permite visualizar estos archivos y no imprimirlos, la mayoría de las otras permiten imprimirlo directamente. Si usa `xdvi` y desea imprimir el documento debe transformar a `ps` vía `dvips` y luego se imprime como cualquier otro *Postscript*.

Para la gran mayoría de formatos gráficos más conocidos y usualmente usados para almacenar fotos existen otra serie se programas especializados en visualización que son capaces

de entender la mayoría de los formatos más usados. Entre estos programas podemos mencionar: *Eye of Gnome* (`eog`), *Electric Eyes* (`eeyes`), `kview` o `display`. Podemos mencionar que aplicaciones como `display` entienden sobre ochenta formatos gráficos distintos entre los que se encuentran la mayoría de los formatos conocidos más otros como `ps`, `eps`, `pdf`, `fig`, `html`, entre muchos otros. Una mención especial merece el utilitario `gthumb` que nos permite hacer un *preview* de un directorio con muchas imagenes de manera muy fácil.

4.2. Modificando imágenes

Si queremos modificaciones como rotaciones, ampliaciones, cortes, cambios de paleta de colores, filtros o efectos sencillos, `display` es la herramienta precisa. Pero si se desea intervenir la imagen en forma profesional, el programa `gimp` es el indicado. El nombre `gimp` viene de *GNU Image Manipulation Program*. Se puede usar esta aplicación para editar y manipular imágenes. Pudiendo cargar y salvar en una variedad de formatos, lo que permite usarlo para convertir entre ellos. La aplicación `gimp` puede también ser usado como programa de pintar, de hecho posee una gran variedad de herramientas en este sentido, tales como brocha de aire, lápiz clonador, tijeras inteligentes, curvas bezier, etc. Además, permite incluir *plugins* que realizan gran variedad de manipulaciones de imagen. Como hecho anecdótico podemos mencionar que la imagen oficial de Tux, el pingüino mascota de Linux, fue creada en `gimp`. Sin embargo, si `gimp` le parece muy profesional o usted sólo necesita un programa para dibujar en el cual se entretenga su hermano menor `tuxpaint` es la alternativa.

4.3. Conversión entre formatos gráficos.

El problema de transformar de un formato a otro es una situación usual en el trabajo con archivos gráficos. Muchos *softwares* tienen salidas muy restringidas en formato o bien usan formatos arcaicos (`gif`) por ejemplo. De ahí que se presenta la necesidad de convertir estos archivos de salida en otros formatos que nos sean más manejables o prácticos. Como ya se mencionó, `gimp` puede ser usado para convertir entre formatos gráficos. También `display` permite este hecho. Sin embargo, en ambos casos la conversión es vía menús, lo cual lo hace engorroso para un gran número de conversiones e imposible para conversiones de tipo automático. Existe un programa llamado `convert` que realiza conversiones desde la línea de comando. Este programa junto con `display`, `import` y varios otros forman la *suite* gráfica *ImageMagick*, una de las más importantes en UNIX, en general, y en especial en Linux y que ya ha sido migrada a otras plataformas. Además, de la clara ventaja de automatización que proporciona `convert`, posee otro aspecto interesante, puede convertir un grupo de imágenes asociadas en una secuencia de animación o película. Veamos la sintaxis para este programa:

```
user@host:~/imagenes$convert cockatoo.tiff cockatoo.jpg
```

```
user@host:~/secuencias$convert -delay 20 dna*.png dna.mng
```

En el primer caso convierte el archivo `cockatoo` de formato `tiff` a formato `jpg`. En el segundo, a partir de un conjunto de archivos con formato `png` llamados `dna` más un número

correlativo, crea una secuencia animada con imágenes que persisten por 20 centésimas de segundos en un formato conocido como `mng`.

4.4. Captura de pantalla.

A menudo se necesita guardar imágenes que sólo se pueden generar a tiempo de ejecución, es decir, mientras corre nuestro programa genera la imagen pero no tiene un mecanismo propio para exportarla o salvarla como imagen. En este caso necesitamos capturar la pantalla y poderla almacenar en un archivo para el cual podamos elegir el formato. Para estos efectos existe un programa, miembro también de la *suite ImageMagick*, llamado `import` que nos permite hacer el trabajo. La sintaxis del comando es

```
import figure.eps
```

```
import -window root root.jpg
```

En el primer caso uno da el comando en un terminal y queda esperando hasta que uno toque alguna de las ventanas, la cual es guardada en este caso en un archivo `figure.eps` en formato *PostScript*. La extensión le indica al programa qué formato usar para almacenar la imagen. En el segundo caso uno captura la pantalla completa en un archivo `root.jpeg`. Este comando puede ser dado desde la consola de texto para capturar la imagen completa en la pantalla gráfica.

4.5. Creando imágenes.

Para imágenes artísticas sin duda la alternativa es `gimp`, todo le que se dijo respecto a sus posibilidades para modificar imágenes se aplica también en el caso de crearlas. En el caso de necesitar imágenes más bien técnicas como esquemas o diagramas o una ilustración para aclarar un problema las alternativas pueden ser `xfig`, `sodipodi` o `sketch` todas herramientas vectoriales muy poderosa. Este tipo de programas son manejados por medio de menús y permiten dibujar y manipular objetos interactivamente. Las imágenes pueden ser salvadas, en formato propios y posteriormente editadas. La gran ventaja de estos programas es que trabaja con objetos y no con bitmaps. Además, puede exportar las imágenes a otros formatos: *PostScript* o *Encapsulated PostScript* o bien `gif` o `jpeg`.

Habitualmente los dibujos necesarios para ilustrar problemas en Física en tareas, pruebas y apuntes son realizados con *software* de este tipo, principalmente `xfig`, luego exportados a *PostScript* e incluídos en los respectivos archivos LaTeX. También existe una herramienta extremadamente útil que permite convertir un archivo *PostScript*, generado de cualquier manera, a un archivo `fig` que puede ser editado y modificado. Esta aplicación que transforma se llama `pstoedit` y puede llegar a ser realmente práctica. Otra herramienta interesante es `autotrace` que permite pasar una figura en *bitmap* a forma vectorial.

Una aparente limitación de este tipo de *software* es que se podría pensar que no podemos incluir curvas analíticas, es decir, si necesitamos ilustrar una función gaussiana no podemos pretender "dibujarla" con las herramientas de que disponen. Sin embargo, este problema puede ser resuelto ya que *software* que grafica funciones analíticas, tales como `gnuplot`,

permite exportar en formato que entienden los programas vectoriales (`fig`, por ejemplo) luego podemos leer el archivo y editarlo. Además, `xfig` permite importar e incluir imágenes del tipo *bitmap*, agregando riqueza a los diagramas que puede generar.

Una característica importante de este tipo de programas es que trabajen por capas, las cuales son tratadas independientemente, uno puede poner un objeto sobre otro o por debajo de otro logrando diferentes efectos. Algunos programas de presentación gráficos basados en LaTeX y `pdf` están utilizando esta capacidad en `xfig` para lograr animaciones de imágenes.

Finalmente el programa `xfig` permite construir una biblioteca de objetos reutilizables ahorrando mucho trabajo. Por ejemplo, si uno dibuja los elementos de un circuito eléctrico y los almacena en el lugar de las bibliotecas de imágenes podrá incluir estos objetos en futuros trabajos. El programa viene con varias bibliotecas de objetos listas para usar.

4.6. Graficando funciones y datos.

Existen varias aplicaciones que permiten graficar datos de un archivo, entre las más populares están: `gnuplot`, `xmgrace` y `SciGraphica`. La primera está basada en la línea de comando y permite gráficos en 2 y 3 dimensiones, pudiendo además, graficar funciones directamente sin pasar por un archivo de datos. Las otras dos son aplicaciones basadas en menús que permiten un resultado final de mucha calidad y con múltiples variantes. La debilidad en el caso de `xmgrace` es que sólo hace gráficos bidimensionales.

El programa `gnuplot` se invoca de la línea de comando y da un *prompt* en el mismo terminal desde el cual se puede trabajar, veamos una sesión de `gnuplot`:

```
jrogan@huelen:~$ gnuplot

        G N U P L O T
        Version 3.7 patchlevel 2
        last modified Sat Jan 19 15:23:37 GMT 2002
        System: Linux 2.4.19

        Copyright(C) 1986 - 1993, 1998 - 2002
        Thomas Williams, Colin Kelley and many others

        Type ‘help‘ to access the on-line reference manual
        The gnuplot FAQ is available from
        http://www.gnuplot.info/gnuplot-faq.html

        Send comments and requests for help to <info-gnuplot@dartmouth.edu>
        Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>


Terminal type set to 'x11'
gnuplot> plot sqrt(x)
gnuplot> set xrange[0:5]
gnuplot> set xlabel" eje de las x"
```

```
gnuplot> replot
gnuplot> set terminal postscript
Terminal type set to 'postscript'
Options are 'landscape noenhanced monochrome dashed defaultplex "Helvetica" 14'
gnuplot> set output "mygraph.ps"
gnuplot> replot
gnuplot> set terminal X
Terminal type set to 'X11'
Options are '0'
gnuplot> set xrange[-2:2]
gnuplot> set yrange[-2:2]
gnuplot> splot exp(-x*x-y*y)
gnuplot> plot "myfile.dat" w l
gnuplot> exit
jrogan@huelen:~$
```

En el caso de `xmgrace` y `SciGraphica` mucho más directo manejarlo ya que está basado en menús. Además, existe abundante documentación de ambos *softwares*. El *software* `SciGraphica` es una aplicación de visualización y análisis de data científica que permite el despliegue de gráficos en 2 y 3 dimensiones, además, exporta los resultados a formato *PostScript*. Realmente esta aplicación nació como un intento de clonar el programa comercial `origen` no disponible para Linux.

4.7. Graficando desde nuestros programas.

Finalmente para poder graficar desde nuestro propio programa necesitamos alguna biblioteca gráfica, en nuestro caso usaremos la biblioteca `iglu`, hecha completamente en casa. La página de `iglu` está en : `http://aristoteles.ciencias.uchile.cl/homepage/iglu/iglu.html` El comando de compilación incluido en un *script*, que llamaremos `iglu_compila`, y que contiene las siguientes líneas:

```
#!/bin/bash
g++ -Wall -O3 -o $1 $1.cc -I. -I$HOME/iglu/ \
-L/usr/X11R6/lib/ -L$HOME/iglu/ -liglu -lX11 -lm
```

Veamos algunos ejemplos:

```
/* Ejemplo: sen(x) */
#include ``iglu.h''
#include <cmath>

int main()
{
   IgluDibujo v(``Funcion Seno'');
   const int N=100;
   double x[N], y[N];
```

```
   v.map_coordinates(0,2*M_PI,-1.2,1.2);
   double dx = 2*M_PI/(N-1);
   for (int i=0;i<N;i++){
     x[i] = i*dx;
     y[i] = sin(x[i]);
   }

   v.plot_line(x,y,N);
   v.wait();
   return 0;
}
```

Este programa grafica la función seno con un número de puntos dado.

Otro caso, una primitiva animación

```
// Ejemplo  sen(x-vt)
#include "iglu.h"
#include <cmath>

int main(){
  IgluDibujo v(‘‘Pelicula’’);
  const int N=100, Nt=100;
  double x[N], y[N];
  v.map_coordinates(0,2*M_PI,-1.2,1.2);
  double dx = 2*M_PI/(N-1), dt = 1, t=0;
  for (int j=0;j<Nt;j++){
    v.clean();
    t += dt;
    for (int i=0;i<N;i++){
      x[i] = i*dx;
      y[i] = sin(x[i]-.1*t);
    }
    v.plot_line(x,y,N);
    v.wait(.03);
    v.flush();
  }
  v.wait();
  return 0;
}
```

Capítulo 5

El sistema de preparación de documentos TEX .

versión 5.0, 30 de Julio del 2003

5.1. Introducción.

TEX es un procesador de texto o, mejor dicho, un avanzado sistema de preparación de documentos, creado por Donald Knuth, que permite el diseño de documentos de gran calidad, conteniendo textos y fórmulas matemáticas. Años después, LATEX fue desarrollado por Leslie Lamport, facilitando la preparación de documentos en TEX, gracias a la definición de "macros" o conjuntos de comandos de fácil uso.

LATEX tuvo diversas versiones hasta la 2.09. Actualmente, LATEX ha recibido importantes modificaciones, siendo la distribución actualmente en uso y desarrollo LATEX 2_ε, una versión transitoria en espera de que algún día se llegue a la nueva versión definitiva de LATEX, LATEX3. En estas páginas cuando digamos LATEX nos referiremos a la versión actual, LATEX 2_ε. Cuando queramos hacer referencia a la versión anterior, que debería quedar progresivamente en desuso, diremos explícitamente LATEX 2.09.

5.2. Archivos.

El proceso de preparación de un documento LATEX consta de tres pasos:

1. Creación de un archivo con extensión `tex` con algún editor.

2. Compilación del archivo `tex`, con un comando del tipo `latex <archivo>.tex` o `latex <archivo>`. Esto da por resultado tres archivos adicionales, con el mismo nombre del archivo original, pero con extensiones distintas:

   a) `dvi`. Es el archivo procesado que podemos ver en pantalla o imprimir. Una vez compilado, este archivo puede ser enviado a otro computador, para imprimir en otra impresora, o verlo en otro monitor, independiente de la máquina (de donde su extensión `dvi`, *device independent*).

b) `log`. Aquí se encuentran todos los mensajes producto de la compilación, para consulta si es necesario (errores encontrados, memoria utilizada, mensajes de advertencia, etc.).

c) `aux`. Contiene información adicional que, por el momento, no nos interesa.

3. Visión en pantalla e impresión del archivo procesado a través de un programa anexo (`xdvi` o `dvips`, por ejemplo), capaz de leer el `dvi`.

5.3. Input básico.

5.3.1. Estructura de un archivo.

En un archivo no pueden faltar las siguientes líneas:

```
\documentclass[12pt]{article}

\begin{document}

\end{document}
```

Haremos algunas precisiones respecto a la primera línea más tarde. Lo importante es que una línea de esta forma debe ser la primera de nuestro archivo. Todo lo que se encuentra antes de `\begin{document}` se denomina *preámbulo*. El texto que queramos escribir va entre `\begin{document}` y `\end{document}`. Todo lo que se encuentre después de `\end{document}` es ignorado.

5.3.2. Caracteres.

Pueden aparecer en nuestro texto todos los caracteres del código ASCII no extendido (teclado inglés usual): letras, números y los signos de puntuación:

```
.  :  ;  ,  ?  !  ‘  ’  (  )  [  ]  -  /  * @
```

Los caracteres especiales:

```
#  $  %  &  ~  _  ^  \  {  }
```

tienen un significado específico para LaTeX. Algunos de ellos se pueden obtener anteponiéndoles un *backslash*:

```
#  \#     $  \$     %  \%     &  \&     {  \{     }  \}
```

Los caracteres

```
+  =  |  <  >
```

generalmente aparecen en fórmulas matemáticas, aunque pueden aparecer en texto normal. Finalmente, las comillas dobles (") casi nunca se usan.

Los espacios en blanco y el fin de línea son también caracteres (invisibles), que LaTeX considera como un mismo carácter, que llamaremos espacio, y que simbolizaremos ocasionalmente como ␣ .

Para escribir en castellano requeriremos además algunos signos y caracteres especiales:

ñ `\~n` á `\'a` í `\'{\i}` ü `\"u` ¡ `!‘` ¿ `?‘`

5.3.3. Comandos.

Todos los comandos comienzan con un backslash, y se extienden hasta encontrar el primer carácter que no sea una letra (es decir, un espacio, un número, un signo de puntuación o matemático, etc.).

5.3.4. Algunos conceptos de estilo.

LaTeX es consciente de muchas convenciones estilísticas que quizás no apreciamos cuando leemos textos bien diseñados, pero las cuales es bueno conocer para aprovecharlas.

a) Observemos la siguiente palabra: fino. Esta palabra fue generada escribiendo simplemente `fino`, pero observemos que las letras ‘f’ e ‘i’ no están separadas, sino que unidas artísticamente. Esto es una *ligadura*, y es considerada una práctica estéticamente preferible. LaTeX sabe esto e inserta este pequeño efecto tipográfico sin que nos demos cuenta.

b) Las comillas de apertura y de cierre son distintas. Por ejemplo: ‘insigne’ (comillas simples) o “insigne” (comillas dobles). Las comillas de apertura se hacen con uno o con dos acentos graves (‘), para comillas simples o dobles, respectivamente, y las de cierre con acentos agudos (’): `‘insigne’`, `‘‘insigne’’`. No es correcto entonces utilizar las comillas dobles del teclado e intentar escribir `"insigne"` (el resultado de esto es el poco estético ”insigne”).

c) Existen tres tipos de guiones:

Corto	Saint-Exupéry	`-`	(entre palabras, corte en sílabas al final de la línea)
Medio	páginas 1–2	`--`	(rango de números)
Largo	un ejemplo —como éste	`---`	(puntuación, paréntesis)

d) LaTeX inserta después de un punto seguido un pequeño espacio adicional respecto al espacio normal entre palabras, para separar sutilmente frases. Pero, ¿cómo saber que un punto termina una frase? El criterio que utiliza es que todo punto termina una frase cuando va precedido de una minúscula. Esto es cierto en la mayoría de los casos, así como es cierto que generalmente cuando un punto viene después de una mayúscula no hay fin de frase:

> China y U.R.S.S. estuvieron de acuerdo. Sin embargo...

Pero hay excepciones:

> En la pág. 11 encontraremos noticias desde la U.R.S.S. Éstas fueron entregadas...

Cuando estas excepciones se producen, nosotros, humanos, tenemos que ayudarle al computador, diciéndole que, aunque hay un punto después de la "g", no hay un fin de frase, y que el punto después de la última "S" sí termina frase. Esto se consigue así:

```
En la p\'ag.\ 11 encontraremos noticias desde la
U.R.S.S\@. \'Estas fueron entregadas...
```

d) Énfasis de texto:

Éste es un texto *enfatizado*.	`\'Este es un texto {\em enfatizado}.`
Otro texto *enfatizado*.	`Otro texto \emph{enfatizado}.`

Al enfatizar, pasamos temporalmente a un tipo de letra distinto, la *itálica*. Esta letra es ligeramente inclinada hacia adelante, lo cual puede afectar el correcto espaciado entre palabras. Comparemos, por ejemplo:

Quiero *hoy* mi recompensa.	`Quiero {\em hoy} mi recompensa.`
Quiero *hoy* mi recompensa.	`Quiero {\em hoy\/} mi recompensa.`
Quiero *hoy* mi recompensa.	`Quiero \emph{hoy} mi recompensa.`

La segunda y tercera frase tienen un pequeño espacio adicional después de "hoy", para compensar el espacio entre palabras perdido por la inclinación de la itálica. Este pequeño espacio se denomina *corrección itálica*, y se consigue usando `\emph`, o, si se usa `\em`, agregando `\/` antes de cerrar el paréntesis cursivo. La corrección itálica es innecesaria cuando después del texto enfatizado viene un punto o una coma. LaTeX advierte esto y omite el espacio adicional aunque uno lo haya sugerido.

5.3.5. Notas a pie de página.

```
Insertemos una nota a pie de p\'agina.\footnote{Como \'esta.}
```

LaTeX colocará una nota a pie de página[1] en el lugar apropiado.

[1] Como ésta.

5.3.6. Fórmulas matemáticas.

LaTeX distingue dos modos de escritura: un modo de texto, en el cual se escriben los textos usuales como los ya mencionados, y un modo matemático, dentro del cual se escriben las fórmulas. Cualquier fórmula *debe* ser escrita dentro de un modo matemático, y si algún símbolo matemático aparece fuera del modo matemático el compilador acusará un error.

Hay tres formas principales para acceder al modo matemático:

a) `$x+y=3$`

b) `$$xy=8$$`

c)
```
\begin{equation}
x/y=5
\end{equation}
```

Estas tres opciones generan, respectivamente, una ecuación en el texto: $x + y = 3$, una ecuación separada del texto, centrada en la página:

$$xy = 8$$

y una ecuación separada del texto, numerada:

$$x/y = 5 \tag{5.1}$$

Es importante notar que al referirnos a una variable matemática en el texto debemos escribirla en modo matemático:

Decir que la incógnita es x es incorrecto. No: la incógnita es x.	`Decir que la inc{\'o}gnita es x es incorrecto. No: la inc{\'o}gnita es $x$.`

5.3.7. Comentarios.

Uno puede hacer que el compilador ignore parte del archivo usando `%`. Todo el texto desde este carácter hasta el fin de la línea correspondiente será ignorado (incluyendo el fin de línea).

Un pequeño comentario.	`Un peque{\~n}o co% Texto ignorado mentario.`

5.3.8. Estilo del documento.

Las características generales del documento están definidas en el preámbulo. Lo más importante es la elección del *estilo*, que determina una serie de parámetros que al usuario normal pueden no importarle, pero que son básicas para una correcta presentación del texto: ¿Qué márgenes dejar en la página? ¿Cuánto dejar de sangría? ¿Tipo de letra? ¿Distancia entre líneas? ¿Dónde poner los números de página? Y un largo etcétera.

Todas estas decisiones se encuentran en un *archivo de estilo* (extensión `cls`). Los archivos standard son: `article`, `report`, `book` y `letter`, cada uno adecuado para escribir artículos cortos (sin capítulos) o más largos (con capítulos), libros y cartas, respectivamente.

La elección del estilo global se hace en la primera línea del archivo:[2]

```
\documentclass{article}
```

Esta línea será aceptada por el compilador, pero nos entregará un documento con un tamaño de letra pequeño, técnicamente llamado de 10 puntos ó 10pt (1pt = 1/72 pulgadas). Existen tres tamaños de letra disponibles: 10, 11 y 12 pt. Si queremos un tamaño de letra más grande, como el que tenemos en este documento, se lo debemos indicar en la primera línea del archivo:

```
\documentclass[12pt]{article}
```

Todas las decisiones de estilo contenidas dentro del archivo `cls` son modificables, existiendo tres modos de hacerlo:

a) Modificando el archivo `cls` directamente. Esto es poco recomendable, porque dicha modificación (por ejemplo, un cambio de los márgenes) se haría extensible a todos los archivos compilados en nuestro computador, y esto puede no ser agradable, ya sea que nosotros seamos los únicos usuarios o debamos compartirlo. Por supuesto, podemos deshacer los cambios cuando terminemos de trabajar, pero esto es tedioso.

b) Introduciendo comandos adecuados en el preámbulo. Ésta es la opción más recomendable y la más usada. Nos permite dominar decisiones específicas de estilo válidas sólo para el archivo que nos interesa.

c) Creando un nuevo archivo `cls`. Esto es muy recomendable cuando las modificaciones de estilo son abundantes, profundas y deseen ser reaprovechadas. Se requiere un poco de experiencia en LaTeX para hacerlo, pero a veces puede ser la única solución razonable.

En todo caso, la opción a usar en la gran mayoría de los casos es la b) (Sec. 5.9).

5.3.9. Argumentos de comandos.

Hemos visto ya algunos comandos que requieren argumentos. Por ejemplo: `\begin{equation}`, `\documentclass[12pt]{article}`, `\footnote{Nota}`. Existen dos tipos de argumentos:

1. **Argumentos obligatorios.** Van encerrados en paréntesis cursivos: `\footnote{Nota}`, por ejemplo. Es obligatorio que después de estos comandos aparezcan los paréntesis. A veces es posible dejar el interior de los paréntesis vacío, pero en otros casos el compilador reclamará incluso eso (`\footnote{}` no genera problemas, pero `\documentclass{}` sí es un gran problema).

   Una propiedad muy general de los comandos de LaTeX es que las llaves de los argumentos obligatorios se pueden omitir cuando dichos argumentos tienen sólo un carácter. Por ejemplo, `\~n` es equivalente a `\~{n}`. Esto permite escribir más fácilmente muchas expresiones, particularmente matemáticas, como veremos más adelante.

[2] En LaTeX 2.09 esta primera línea debe ser `\documentstyle[12pt]article`, y el archivo de estilo tiene extensión `sty`. Intentar compilar con LaTeX 2.09 un archivo que comienza con `\documentclass` da un error. Por el contrario, la compilación con LaTeX 2ε de un archivo que comienza con `\documentstyle` no genera un error, y LaTeX entra en un *modo de compatibilidad*. Sin embargo, interesantes novedades de LaTeX 2ε respecto a LaTeX 2.09 se pierden.

2. **Argumentos opcionales.** Van encerrados en paréntesis cuadrados. Estos argumentos son omitibles, `\documentclass[12pt]...` . Ya dijimos que `\documentclass{article}` es aceptable, y que genera un tamaño de letra de 10pt. Un argumento en paréntesis cuadrados es una opción que modifica la decisión default del compilador (en este caso, lo obliga a usar 12pt en vez de sus instintivos 10pt).

5.3.10. Título.

Un título se genera con:

```
\title{Una breve introducci\'on}
\author{V\'{\i}ctor Mu\~noz}
\date{30 de Junio de 1998}
\maketitle
```

`\title`, `\author` y `\date` pueden ir en cualquier parte (incluyendo el preámbulo) antes de `\maketitle`. `\maketitle` debe estar después de `\begin{document}`. Dependiendo de nuestras necesidades, tenemos las siguientes alternativas:

a) Sin título:

```
\title{}
```

b) Sin autor:

```
\author{}
```

c) Sin fecha:

```
\date{}
```

d) Fecha actual (en inglés): omitir `\date`.

e) Más de un autor:

```
\author{Autor_1 \and Autor_2 \and Autor_3}
```

Para artículos cortos, LaTeX coloca el título en la parte superior de la primera página del texto. Para artículos largos, en una página separada.

5.3.11. Secciones.

Los títulos de las distintas secciones y subsecciones de un documento (numerados adecuadamente, en negrita, como en este texto) se generan con comandos de la forma:

```
\section{Una secci\'on}
\subsection{Una subsecci\'on}
```

Los comandos disponibles son (en orden decreciente de importancia):

`\part`	`\subsection`	`\paragraph`
`\chapter`	`\subsubsection`	`\subparagraph`
`\section`		

Los más usados son `\chapter`, `\section`, `\subsection` y `\subsubsection`. `\chapter` sólo está disponible en los estilos `report` y `book`.

5.3.12. Listas.

Los dos modos usuales de generar listas:

a) Listas numeradas (ambiente `enumerate`):

1. Nivel 1, ítem 1.
2. Nivel 1, ítem 2.
   a) Nivel 2, ítem 1.
      1) Nivel 3, ítem 1.
3. Nivel 1, ítem 3.

```
\begin{enumerate}
\item Nivel 1, \'{\i}tem 1.
\item Nivel 1, \'{\i}tem 2.
\begin{enumerate}
\item Nivel 2, \'{\i}tem 1.
\begin{enumerate}
\item Nivel 3, \'{\i}tem 1.
\end{enumerate}
\end{enumerate}
\item Nivel 1, \'{\i}tem 3.
\end{enumerate}
```

b) Listas no numeradas (ambiente `itemize`):

- Nivel 1, ítem 1.
- Nivel 1, ítem 2.
  - Nivel 2, ítem 1.
    - Nivel 3, ítem 1.
- Nivel 1, ítem 3.

```
\begin{itemize}
\item Nivel 1, {\'\i}tem 1.
\item Nivel 1, {\'\i}tem 2.
\begin{itemize}
\item Nivel 2, {\'\i}tem 1.
\begin{itemize}
\item Nivel 3, {\'\i}tem 1.
\end{itemize}
\end{itemize}
\item Nivel 1, {\'\i}tem 3.
\end{itemize}
```

Es posible anidar hasta tres niveles de listas. Cada uno usa tipos distintos de rótulos, según el ambiente usado: números arábes, letras y números romanos para `enumerate`, y puntos, guiones y asteriscos para `itemize`. Los rótulos son generados automáticamente por cada `\item`, pero es posible modificarlos agregando un parámetro opcional:

a) Nivel 1, ítem 1.
b) Nivel 1, ítem 2.

```
\begin{enumerate}
\item[a)] Nivel 1, \'{\i}tem 1.
\item[b)] Nivel 1, \'{\i}tem 2.
\end{enumerate}
```

`\item` es lo primero que debe aparecer después de un `\begin{enumerate}` o `\begin{itemize}`.

5.3.13. Tipos de letras.

Fonts.

Los fonts disponibles por default en LaTeX son:

roman	*italic*	Small Caps
boldface	*slanted*	`typewriter`
sans serif		

Los siguientes modos de cambiar fonts son equivalentes:

texto	`{\rm texto}`	`\textrm{texto}`
texto	`{\bf texto}`	`\textbf{texto}`
texto	`{\sf texto}`	`\textsf{texto}`
texto	`{\it texto}`	`\textit{texto}`
texto	`{\sl texto}`	`\textsl{texto}`
Texto	`{\sc Texto}`	`\textsc{texto}`
`texto`	`{\tt texto}`	`\texttt{texto}`

`\rm` es el default para texto normal; `\it` es el default para texto enfatizado; `\bf` es el default para títulos de capítulos, secciones, subsecciones, etc.

`\textrm`, `\textbf`, etc., sólo permiten cambiar porciones definidas del texto, contenido entre los paréntesis cursivos. Con `\rm`, `\bf`, etc. podemos, omitiendo los paréntesis, cambiar el font en todo el texto posterior:

Un cambio local de fonts *y uno global, interminable e infinito...*

```
Un cambio {\sf local} de fonts
\sl y uno global, interminable
e infinito...
```

También es posible tener combinaciones de estos fonts, por ejemplo, ***bold italic***, pero no sirven los comandos anteriores, sino versiones modificadas de `\rm`, `\bf`, etc.:

```
\rmfamily
\sffamily
\ttfamily
\mdseries
```

```
\bfseries
\upshape
\itshape
\slshape
\scshape
```

Por ejemplo:

texto	`{\bfseries\itshape texto}`
texto	`{\bfseries\upshape texto}` (= `{\bf texto}`)
TEXTO	`{\ttfamily\scshape texto}`
texto	`{\sffamily\bfseries texto}`
texto	`{\sffamily\mdseries texto}` (= `{\sf texto}`)

Para entender el uso de estos comandos hay que considerar que un font tiene tres *atributos*: `family` (que distingue entre `rm`, `sf` y `tt`), `series` (que distingue entre `md` y `bf`), y `shape` (que distingue entre `up`, `it`, `sl` y `sc`). Cada uno de los comandos `\rmfamily`, `\bfseries`, etc., cambia sólo uno de estos atributos. Ello permite tener versiones mixtas de los fonts, como un *slanted sans serif*, imposible de obtener usando los comandos `\sl` y `\sf`. Los defaults para el texto usual son: `\rmfamily`, `\mdseries` y `\upshape`.

Tamaño.

Los tamaños de letras disponibles son:

texto	`\tiny`	texto	`\normalsize`	texto	`\LARGE`
texto	`\scriptsize`	texto	`\large`	texto	`\huge`
texto	`\footnotesize`	texto	`\Large`	texto	`\Huge`
texto	`\small`				

Se usan igual que los comandos de cambio de font `\rm`, `\sf`, etc., de la sección 5.3.13.

`\normalsize` es el default para texto normal; `\scriptsize` para sub o supraíndices; `\footnotesize` para notas a pie de página.

5.3.14. Acentos y símbolos.

LaTeX provee diversos tipos de acentos, que se muestran en la Tabla 5.1 (como ejemplo consideramos la letra "o", pero cualquiera es posible, por supuesto). (Hemos usado acá el hecho de que cuando el argumento de un comando consta de un carácter, las llaves son omitibles.)

Otros símbolos especiales y caracteres no ingleses disponibles se encuentran en la Tabla 5.2.

ó	\'o	õ	\~o	ǒ	\v o	o̧	\c o
ò	\`o	ō	\=o	ő	\H o	ọ	\d o
ô	\^o	ȯ	\. o	o͡o	\t{oo}	o̲	\b o
ö	\"o	ŏ	\u o	o̊	\r o		

Cuadro 5.1: Acentos.

†	\dag	œ	\oe	ł	\l
‡	\ddag	Œ	\OE	Ł	\L
§	\S	æ	\ae	ß	\ss
¶	\P	Æ	\AE	SS	\SS
©	\copyright	å	\aa	¿	?`
ⓐ	\textcircled a	Å	\AA	¡	!`
␣	\textvisiblespace	ø	\o		
£	\pounds	Ø	\O		

Cuadro 5.2: Símbolos especiales y caracteres no ingleses.

5.3.15. Escritura de textos en castellano.

LaTeX emplea sólo los caracteres ASCII básicos, que no contienen símbolos castellanos como ¿, ¡, ñ, etc. Ya hemos visto que existen comandos que permiten imprimir estos caracteres, y por tanto es posible escribir cualquier texto en castellano (y otros idiomas, de hecho).

Sin embargo, esto no resuelve todo el problema, porque en inglés y castellano las palabras se cortan en "sílabas" de acuerdo a reglas distintas, y esto es relevante cuando se debe cortar el texto en líneas. LaTeX tiene incorporados algoritmos para cortar palabras en inglés y, si se ha hecho una instalación especial de LaTeX en nuestro computador, también en castellano u otros idiomas (a través del programa **babel**, que es parte de la distribución standard de LaTeX 2_ε). En un computador con babel instalado y configurado para cortar en castellano basta incluir el comando `\usepackage[spanish]{babel}` en el preámbulo para poder escribir en castellano cortando las palabras en sílabas correctamente.[3]

Sin embargo, ocasionalmente LaTeX se encuentra con una palabra que no sabe cortar, en cuyo caso no lo intenta y permite que ella se salga del margen derecho del texto, o bien toma decisiones no óptimas. La solución es sugerirle a LaTeX la silabación de la palabra. Por ejemplo, si la palabra conflictiva es `matem\'aticas` (generalmente hay problemas con las palabras acentuadas), entonces basta con reescribirla en la forma: `ma\-te\-m\'a\-ti\-cas`. Con esto, le indicamos a LaTeX en qué puntos es posible cortar la palabra. El comando `\-` no tiene ningún otro efecto, de modo que si la palabra en cuestión no queda al final de la línea, LaTeX por supuesto ignora nuestra sugerencia y no la corta.

Consideremos el siguiente ejemplo:

[3]Esto resuelve también otro problema: los encabezados de capítulos o índices, por ejemplo, son escritos "Capítulo" e "Índice", en vez de "Chapter" e "Index", y cuando se usa el comando `\date`, la fecha aparece en castellano.

Podemos escribir matemáticas. O matemáticas.

```
Podemos escribir matem\'aticas.
O matem\'aticas.
```

Podemos escribir matemáticas. O matemáticas.

```
Podemos escribir
ma\-te\-m\'a\-ti\-cas.
O ma\-te\-m\'a\-ti\-cas.
```

En el primer caso, LaTeX decidió por sí mismo dónde cortar "matemáticas". Como es una palabra acentuada tuvo problemas y no lo hizo muy bien, pues quedó demasiado espacio entre palabras en esa línea. En el segundo párrafo le sugerimos la silabación y LaTeX pudo tomar una decisión más satisfactoria. En el mismo párrafo, la segunda palabra "matemáticas" también tiene sugerencias de corte, pero como no quedó al final de línea no fueron tomadas en cuenta.

5.4. Fórmulas matemáticas.

Hemos mencionado tres formas de ingresar al modo matemático: `$...$` (fórmulas dentro del texto), `$$...$$` (fórmulas separadas del texto, no numeradas) y `\begin{equation} ... \end{equation}` (fórmulas separadas del texto, numeradas). Los comandos que revisaremos en esta sección sólo pueden aparecer dentro del modo matemático.

5.4.1. Sub y supraíndices.

x^{2y} `x^{2y}` x^{y^2} `x^{y^{2}}` (ó `x^{y^2}`) x_1^y `x^y_1` (ó `x_1^y`)
x_{2y} `x_{2y}` x^{y_1} `x^{y_{1}}` (ó `x^{y_1}`)

`\textsuperscript` permite obtener supraíndices fuera del modo matemático:

La 3ª es la vencida.

```
La 3\textsuperscript{a}
es la vencida.
```

5.4.2. Fracciones.

a) Horizontales

$n/2$ `n/2`

b) Verticales

$\frac{1}{2}$ `\frac{1}{2}`, `\frac 1{2}`, `\frac{1}2` ó `\frac 12`

$x = \frac{y + z/2}{y^2 + 1}$

```
x = \frac{y + z/2}{y^2+1}
```

$\frac{x+y}{1 + \frac{y}{z+1}}$

```
\frac{x+y}{1 + \frac y{z+1}}
```

La forma a) es más adecuada y la preferida para fracciones dentro del texto, y la segunda para fórmulas separadas. `\frac` puede aparecer en fórmulas dentro del texto ($\frac{1}{2}$ con `$\frac 12$`), pero esto es inusual y poco recomendable estéticamente, salvo estricta necesidad.

5.4.3. Raíces.

$\sqrt{n}$	`\sqrt{n}` ó `\sqrt n`
$\sqrt{a^2 + b^2}$	`\sqrt{a^2 + b^2}`
$\sqrt[n]{2}$	`\sqrt[n]{2}`

5.4.4. Puntos suspensivos.

a) $\ldots$ `\ldots`

Para fórmulas como

$$a_1 a_2 \ldots a_n \qquad \texttt{a\_1 a\_2 \textbackslash ldots a\_n}$$

b) $\cdots$ `\cdots`

Entre símbolos como $+$, $-$, $=$:

$$x_1 + \cdots + x_n \qquad \texttt{x\_1 + \textbackslash cdots + x\_n}$$

c) $\vdots$ `\vdots`

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

d) $\ddots$ `\ddots`

$$I_{n\times n} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix}$$

`\ldots` puede ser usado también en el texto usual:

Arturo quiso salir...pero se detuvo.	`Arturo quiso salir\ldots pero se detuvo.`

No corresponde usar tres puntos seguidos (...), pues el espaciado entre puntos es incorrecto.

Minúsculas

α	\alpha	θ	\theta	o	o	τ	\tau
β	\beta	ϑ	\vartheta	π	\pi	υ	\upsilon
γ	\gamma	ι	\iota	ϖ	\varpi	ϕ	\phi
δ	\delta	κ	\kappa	ρ	\rho	φ	\varphi
ϵ	\epsilon	λ	\lambda	ϱ	\varrho	χ	\chi
ε	\varepsilon	μ	\mu	σ	\sigma	ψ	\psi
ζ	\zeta	ν	\nu	ς	\varsigma	ω	\omega
η	\eta	ξ	\xi				

Mayúsculas

Γ	\Gamma	Λ	\Lambda	Σ	\Sigma	Ψ	\Psi
Δ	\Delta	Ξ	\Xi	Υ	\Upsilon	Ω	\Omega
Θ	\Theta	Π	\Pi	Φ	\Phi		

Cuadro 5.3: Letras griegas.

5.4.5. Letras griegas.

Las letras griegas se obtienen simplemente escribiendo el nombre de dicha letra (en inglés): `\gamma`. Para la mayúscula correspondiente se escribe la primera letra con mayúscula: `\Gamma`. La lista completa se encuentra en la Tabla 5.3.

No existen símbolos para α, β, η, etc. mayúsculas, pues corresponden a letras romanas (A, B, E, etc.).

5.4.6. Letras caligráficas.

Letras caligráficas mayúsculas $\mathcal{A}$, $\mathcal{B}$, ..., $\mathcal{Z}$ se obtienen con `\cal`. `\cal` se usa igual que los otros comandos de cambio de font (`\rm`, `\it`, etc.).

Sea $\mathcal{F}$ una función con $\mathcal{F}(x) > 0$.

```
Sea $\cal F$ una funci\'on
con ${\cal F}(x) > 0$.
```

No son necesarios los paréntesis cursivos la primera vez que se usan en este ejemplo, porque el efecto de `\cal` está delimitado por los `$`.

5.4.7. Símbolos matemáticos.

LaTeX proporciona una gran variedad de símbolos matemáticos (Tablas 5.4, 5.5, 5.6, 5.7). La negación de cualquier símbolo matemático se obtiene con `\not`:

$x \not< y$

$a \notin \mathcal{M}$

```
x \not < y
a \not \in {\cal M}
```

[Notemos, sí, en la Tabla 5.5, que existe el símbolo $\neq$ (`\neq`).]

$\pm$	\pm	$\cap$	\cap	$\diamond$	\diamond	$\oplus$	\oplus
$\mp$	\mp	$\cup$	\cup	$\triangle$	\bigtriangleup	$\ominus$	\ominus
$\times$	\times	$\uplus$	\uplus	$\triangledown$	\bigtriangledown	$\otimes$	\otimes
$\div$	\div	$\sqcap$	\sqcap	$\triangleleft$	\triangleleft	$\oslash$	\oslash
$*$	\ast	$\sqcup$	\sqcup	$\triangleright$	\triangleright	$\odot$	\odot
$\star$	\star	$\vee$	\lor	$\bigcirc$	\bigcirc		
$\circ$	\circ	$\wedge$	\land	$\dagger$	\dagger		
$\bullet$	\bullet	$\setminus$	\setminus	$\ddagger$	\ddagger		
$\cdot$	\cdot	$\wr$	\wr	$\amalg$	\amalg		

Cuadro 5.4: Símbolos de operaciones binarias.

$\leq$	\leq	$\geq$	\geq	$\equiv$	\equiv	$\models$	\models
$\prec$	\prec	$\succ$	\succ	$\sim$	\sim	$\perp$	\perp
$\preceq$	\preceq	$\succeq$	\succeq	$\simeq$	\simeq	$\mid$	\mid
$\ll$	\ll	$\gg$	\gg	$\asymp$	\asymp	$\parallel$	\parallel
$\subset$	\subset	$\supset$	\supset	$\approx$	\approx	$\bowtie$	\bowtie
$\subseteq$	\subseteq	$\supseteq$	\supseteq	$\cong$	\cong	$\neq$	\neq
$\smile$	\smile	$\sqsubseteq$	\sqsubseteq	$\sqsupseteq$	\sqsupseteq	$\doteq$	\doteq
$\frown$	\frown	$\in$	\in	$\ni$	\ni	$\propto$	\propto
$\vdash$	\vdash	$\dashv$	\dashv				

Cuadro 5.5: Símbolos relacionales.

$\leftarrow$	\gets	$\longleftarrow$	\longleftarrow	$\uparrow$	\uparrow
$\Leftarrow$	\Leftarrow	$\Longleftarrow$	\Longleftarrow	$\Uparrow$	\Uparrow
$\rightarrow$	\to	$\longrightarrow$	\longrightarrow	$\downarrow$	\downarrow
$\Rightarrow$	\Rightarrow	$\Longrightarrow$	\Longrightarrow	$\Downarrow$	\Downarrow
$\leftrightarrow$	\Leftrightarrow	$\longleftrightarrow$	\Longleftrightarrow	$\updownarrow$	\Updownarrow
$\mapsto$	\mapsto	$\longmapsto$	\longmapsto	$\nearrow$	\nearrow
$\hookleftarrow$	\hookleftarrow	$\hookrightarrow$	\hookrightarrow	$\searrow$	\searrow
$\leftharpoonup$	\leftharpoonup	$\rightharpoonup$	\rightharpoonup	$\swarrow$	\swarrow
$\leftharpoondown$	\leftharpoondown	$\rightharpoondown$	\rightharpoondown	$\nwarrow$	\nwarrow
$\rightleftharpoons$	\rightleftharpoons				

Cuadro 5.6: Flechas

$\aleph$	\aleph	$\prime$	\prime	$\forall$	\forall	∞	\infty
$\hbar$	\hbar	$\emptyset$	\emptyset	$\exists$	\exists	$\triangle$	\triangle
$\imath$	\imath	∇	\nabla	$\neg$	\lnot	$\clubsuit$	\clubsuit
$\jmath$	\jmath	$\surd$	\surd	$\flat$	\flat	$\diamondsuit$	\diamondsuit
ℓ	\ell	$\top$	\top	$\natural$	\natural	$\heartsuit$	\heartsuit
$\wp$	\wp	$\bot$	\bot	$\sharp$	\sharp	$\spadesuit$	\spadesuit
$\Re$	\Re	$\|$	|	\	\backslash		
$\Im$	\Im	$\angle$	\angle	∂	\partial		

Cuadro 5.7: Símbolos varios.

$\sum$	$\sum$	\sum	$\bigcap$	$\bigcap$	\bigcap	$\bigodot$	$\bigodot$	\bigodot
$\prod$	$\prod$	\prod	$\bigcup$	$\bigcup$	\bigcup	$\bigotimes$	$\bigotimes$	\bigotimes
$\coprod$	$\coprod$	\coprod	$\bigsqcup$	$\bigsqcup$	\bigsqcup	$\bigoplus$	$\bigoplus$	\bigoplus
$\int$	$\int$	\int	$\bigvee$	$\bigvee$	\bigvee	$\bigoplus$	$\bigoplus$	\bigoplus
$\oint$	$\oint$	\oint	$\bigwedge$	$\bigwedge$	\bigwedge			

Cuadro 5.8: Símbolos de tamaño variable.

Algunos símbolos tienen tamaño variable, según aparezcan en el texto o en fórmulas separadas del texto. Se muestran en la Tabla 5.8.

Estos símbolos pueden tener índices que se escriben como sub o supraíndices. Nuevamente, la ubicación de estos índices depende de si la fórmula está dentro del texto o separada de él:

$$\sum_{i=1}^{n} x_i = \int_0^1 f$$ `$$\sum_{i=1}^n x_i = \int_0^1 f $$`

$\sum_{i=1}^{n} x_i = \int_0^1 f$ `$\sum_{i=1}^n x_i = \int_0^1 f $`

5.4.8. Funciones tipo logaritmo.

Observemos la diferencia entre estas dos expresiones:

$x = logy$ `$x  = log y$`

$x = \log y$ `$x  = \log y$`

En el primer caso LaTeX escribe el producto de cuatro cantidades, l, o, g e y. En el segundo, representa correctamente nuestro deseo: el logaritmo de y. Todos los comandos de la Tabla 5.9 generan el nombre de la función correspondiente, en letras romanas.

Algunas de estas funciones pueden tener índices:

$$\lim_{n\to\infty} x_n = 0$$ `$$\lim_{n\to\infty} x_n = 0 $$`

$\lim_{n\to\infty} x_n = 0$ `$\lim_{n\to\infty} x_n = 0 $`

\arccos	\cos	\csc	\exp	\ker	\limsup	\min	\sinh
\arcsin	\cosh	\deg	\gcd	\lg	\ln	\Pr	\sup
\arctan	\cot	\det	\hom	\lim	\log	\sec	\tan
\arg	\coth	\dim	\inf	\liminf	\max	\sin	\tanh

Cuadro 5.9: Funciones tipo logaritmo

(	(	)	)	↑	\uparrow
[	[	]	]	↓	\downarrow
{	\{	}	\}	↕	\updownarrow
⌊	\lfloor	⌋	\rfloor	⇑	\Uparrow
⌈	\lceil	⌉	\rceil	⇓	\Downarrow
⟨	\langle	⟩	\rangle	⇕	\Updownarrow
/	/	\	\backslash		
|	|	‖	\\|		

Cuadro 5.10: Delimitadores

5.4.9. Matrices.

Ambiente `array`.

Se construyen con el ambiente `array`. Consideremos, por ejemplo:

$$\begin{array}{clr} a+b+c & uv & 27 \\ a+b & u+v & 134 \\ a & 3u+vw & 2.978 \end{array}$$

La primera columna está alineada al centro (`c`, center); la segunda, a la izquierda (`l`, left); la tercera, a la derecha (`r`, right). `array` tiene un argumento obligatorio, que consta de tantas letras como columnas tenga la matriz, letras que pueden ser `c`, `l` o `r` según la alineación que queramos obtener. Elementos consecutivos de la misma línea se separan con `&` y líneas consecutivas se separan con `\\`. Así, el ejemplo anterior se obtiene con:

```
\begin{array}{clr}
a+b+c & uv & 27 \\
a+b & u + v & 134 \\
a & 3u+vw & 2.978
\end{array}
```

Delimitadores.

Un delimitador es cualquier símbolo que actúe como un paréntesis, encerrando una expresión, apareciendo a la izquierda y a la derecha de ella. La Tabla 5.10 muestra todos los delimitadores posibles.

Para que los delimitadores tengan el tamaño correcto para encerrar la expresión correspondiente hay que anteponerles `\left` y `\right`. Podemos obtener así expresiones matriciales:

$$\left(\begin{array}{cc} a & b \\ c & d \end{array} \right)$$

```
\left(\begin{array}{cc}
a&b\\
c&d
\end{array}\right)
```

$$v = \left(\begin{array}{c} 1 \\ 2 \\ 3 \end{array} \right)$$

```
v = \left(\begin{array}{c}
1\\
2\\
3
\end{array}\right)
```

$$\Delta = \left| \begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right|$$

```
\Delta = \left|\begin{array}{cc}
a_{11} & a_{12}\\
a_{21} & a_{22}
\end{array}\right|
```

`\left` y `\right` deben ir de a pares, pero los delimitadores no tienen por qué ser los mismos:

$$\left(\begin{array}{c} a \\ b \end{array} \right[$$

```
\left(\begin{array}{c}
a\\
b
\end{array}\right[
```

Tampoco es necesario que los delimitadores encierren matrices. Comparemos, por ejemplo:

$$(\vec{A} + \vec{B}) = (\frac{d\vec{F}}{dx})_{x=a}$$

```
(\vec A + \vec B) =
( \frac{d \vec F}{dx} )_{x=a}
```

$$\left(\vec{A} + \vec{B}\right) = \left(\frac{d\vec{F}}{dx}\right)_{x=a}$$

```
\left(\vec A + \vec B\right) =
\left( \frac{d \vec F}{dx} \right)_{x=a}
```

El segundo ejemplo es mucho más adecuado estéticamente.

Algunas expresiones requieren sólo un delimitador, a la izquierda o a la derecha. Un punto (`.`) representa un delimitador invisible. Los siguientes ejemplos son típicos:

$$\int_a^b dx \frac{df}{dx} = f(x) \Big|_a^b$$

```
\left. \int_a^b  dx \frac{df}{dx} =
f(x) \right |_a^b
```

$$f(x) = \left\{ \begin{array}{cl} 0 & x < 0 \\ 1 & x > 0 \end{array} \right.$$

```
f(x) = \left\{ \begin{array}{cl}
0 & x<0 \\
1 & x>0
\end{array} \right.
```

Fórmulas de más de una línea.

`eqnarray` ofrece una manera de ingresar a modo matemático (en reemplazo de `$`, `$$` o `equation`) equivalente a un `array` con argumentos `{rcl}`:

$\hat{a}$	`\hat a`	$\acute{a}$	`\acute a`	$\bar{a}$	`\bar a`	$\dot{a}$	`\dot a`
$\check{a}$	`\check a`	$\grave{a}$	`\grave a`	$\vec{a}$	`\vec a`	$\ddot{a}$	`\ddot a`
$\breve{a}$	`\breve a`	$\tilde{a}$	`\tilde a`				

Cuadro 5.11: Acentos matemáticos

$$x = a+b+c+ d+e$$

```
\begin{eqnarray*}
x& = & a + b + c +\\
&& d + e
\end{eqnarray*}
```

El asterisco impide que aparezcan números en las ecuaciones. Si deseamos que numere cada línea como una ecuación independiente, basta omitir el asterisco:

$$x = 5 \quad (5.2)$$
$$a+b = 60 \quad (5.3)$$

```
\begin{eqnarray}
x& = & 5 \\
a + b& = & 60
\end{eqnarray}
```

Si queremos que solamente algunas líneas aparezcan numeradas, usamos `\nonumber`:

$$x = a+b+c+ d+e \quad (5.4)$$

```
\begin{eqnarray}
x& = & a + b + c + \nonumber\\
&& d + e
\end{eqnarray}
```

El comando `\eqnarray` es suficiente para necesidades sencillas, pero cuando se requiere escribir matemática de modo intensivo sus limitaciones comienzan a ser evidentes. Al agregar al preámbulo de nuestro documento la línea `\usepackage{amsmath}` quedan disponibles muchos comandos mucho más útiles para textos matemáticos más serios, como el ambiente `equation*`, `\split`, `\multline` o `\intertext`. En la sección 5.8.2 se encuentra una descripción de estos y otros comandos.

5.4.10. Acentos.

Dentro de una fórmula pueden aparecer una serie de "acentos", análogos a los de texto usual (Tabla 5.11).

Las letras i y j deben perder el punto cuando son acentuadas: $\vec{i}$ es incorrecto. Debe ser $\vec{\imath}$. `\imath` y `\jmath` generan las versiones sin punto de estas letras:

$$\vec{\imath} + \hat{\jmath}$$

```
\vec \imath + \hat \jmath
```

5.4.11. Texto en modo matemático.

Para insertar texto dentro de modo matemático empleamos `\mbox`:

$V_{\text{crítico}}$ `V_{\mbox{\scriptsize cr\'{\i}tico}}`

Bastante más óptimo es utilizar el comando `\text`, disponible a través de `amsmath` (sección 5.8.2).

5.4.12. Espaciado en modo matemático.

TEX ignora los espacios que uno escribe en las fórmulas y los determina de acuerdo a sus propios criterios. A veces es necesario ayudarlo para hacer ajustes finos. Hay cuatro comandos que agregan pequeños espacios dentro de modo matemático:

`\,`	espacio pequeño	`\:`	espacio medio
`\!`	espacio pequeño (negativo)	`\;`	espacio grueso

Algunos ejemplos de su uso:

$\sqrt{2}\,x$	`\sqrt 2 \, x`	en vez de	$\sqrt{2}x$
$n/\log n$	`n / \!\log n`	en vez de	$n/\log n$
$\int f\,dx$	`\int f \, dx`	en vez de	$\int f dx$

El último caso es quizás el más frecuente, por cuanto la no inserción del pequeño espacio adicional entre f y dx hace aparecer el integrando como el producto de tres variables, f, d y x, que no es la idea.

5.4.13. Fonts.

Análogamente a los comandos para texto usual (Sec. 5.3.13), es posible cambiar los fonts dentro del modo matemático:

(A, x)	`\mathrm{(A,x)}`
(A, x)	`\mathnormal{(A,x)}`
$(\mathcal{A}, \mathcal{B})$	`\mathcal{(A,B)}`
$(\mathbf{A}, \mathbf{x})$	`\mathbf{(A,x)}`
(A, x)	`\mathsf{(A,x)}`
$(\mathtt{A}, \mathtt{x})$	`\mathtt{(A,x)}`
(A, x)	`\mathit{(A,x)}`

(Recordemos que la letras tipo `\cal` sólo existen en mayúsculas.)

Las declaraciones anteriores permiten cambiar los fonts de letras, dígitos y acentos, pero no de los otros símbolos matemáticos:

$\mathbf{\tilde{A} \times 1}$ `\mathbf{\tilde A \times 1}`

Como en todo ambiente matemático, los espacios entre caracteres son ignorados:

Hola `\mathrm{H o l a}`

Finalmente, observemos que `\mathit` corresponde al font itálico, en tanto que `\mathnormal` al font matemático usual, que es también itálico... o casi:

$different$ `$different$`
$different$ `$\mathnormal{different}$`

different `$\mathit{different}$`
different `\textit{different}`

5.5. Tablas.

`array` nos permitió construir matrices en modo matemático. Para tablas de texto existe `tabular`, que funciona de la misma manera. Puede ser usado tanto en modo matemático como fuera de él.

Nombre	:	Juan Pérez
Edad	:	26
Profesión	:	Estudiante

```
\begin{tabular}{lcl}
Nombre&:&Juan P\'erez\\
Edad&:&26\\
Profesi\'on&:&Estudiante
\end{tabular}
```

Si deseamos agregar líneas verticales y horizontales para ayudar a la lectura, lo hacemos insertando `|` en los puntos apropiados del argumento de `tabular`, y `\hline` al final de cada línea de la tabla:

Item	Gastos
Vasos	$ 500
Botellas	$ 1300
Platos	$ 500
Total	$ 2300

```
\begin{tabular}{|l|r|}\hline
Item&Gastos\\ \hline
Vasos& \$ 500 \\
Botellas & \$ 1300 \\
Platos & \$ 500 \\ \hline
Total& \$ 2300 \\ \hline
\end{tabular}
```

5.6. Referencias cruzadas.

Ecuaciones, secciones, capítulos y páginas son entidades que van numeradas y a las cuales podemos querer referirnos en el texto. Evidentemente no es óptimo escribir explícitamente el número correspondiente, pues la inserción de una nueva ecuación, capítulo, etc., su eliminación o cambio de orden del texto podría alterar la numeración, obligándonos a modificar estos números dispersos en el texto. Mucho mejor es referirse a ellos de modo simbólico y dejar que TEX inserte por nosotros los números. Lo hacemos con `\label` y `\ref`.

La ecuación de Euler

$$e^{i\pi} + 1 = 0 \qquad (5.5)$$

reúne los números más importantes. La ecuación (5.5) es famosa.

```
La ecuaci\'on de Euler
\begin{equation}
\label{euler}
 e^{i\pi} + 1 = 0
\end{equation}
re\'une los n\'umeros
m\'as importantes.
La ecuaci\'on (\ref{euler})
es famosa.
```

El argumento de `\label` (reiterado luego en `\ref`) es una etiqueta simbólica. Ella puede ser cualquier secuencia de letras, dígitos o signos de puntuación. Letras mayúsculas y minúsculas son diferentes. Así, `euler`, `eq:euler`, `euler_1`, `euler1`, `Euler`, etc., son etiquetas válidas y distintas. Podemos usar `\label` dentro de `equation`, `eqnarray` y `enumerate`.

También podemos referenciar páginas con `\pageref`:

Ver página 170 para más detalles.

[Texto en pág. 170]

El significado de la vida...

```
Ver p\'agina
\pageref{significado}
para m\'as detalles.
...
El significado
\label{significado}
de la vida...
```

LaTeX puede dar cuenta de las referencias cruzadas gracias al archivo `aux` (auxiliar) generado durante la compilación.

Al compilar por primera vez el archivo, en el archivo `aux` es escrita la información de los `\label` encontrados. Al compilar por segunda vez, LaTeX lee el archivo `aux` e incorpora esa información al `dvi`. (En realidad, también lo hizo la primera vez que se compiló el archivo, pero el `aux` no existía entonces o no tenía información útil.)

Por tanto, para obtener las referencias correctas hay que compilar dos veces, una para generar el `aux` correcto, otra para poner la información en el `dvi`. Toda modificación en la numeración tendrá efecto sólo después de compilar dos veces más. Por cierto, no es necesario preocuparse de estos detalles a cada momento. Seguramente compilaremos muchas veces el archivo antes de tener la versión final. En todo caso, LaTeX avisa, tras cada compilación, si hay referencias inexistentes u otras que pudieron haber cambiado, y sugiere compilar de nuevo para obtener las referencias correctas. (Ver Sec. 5.14.2.)

5.7. Texto centrado o alineado a un costado.

Los ambientes `center`, `flushleft` y `flushright` permiten forzar la ubicación del texto respecto a los márgenes. Líneas consecutivas se separan con `\\`:

Una línea centrada,
otra
y otra más.

Ahora el texto continúa

alineado a la izquierda

y finalmente

dos líneas
alineadas a la derecha.

```
\begin{center}
Una l\'{\i}nea centrada,\\
otra\\
y otra m\'as.
\end{center}
Ahora el texto contin\'ua
\begin{flushleft}
alineado a la izquierda
\end{flushleft}
y finalmente
\begin{flushright}
dos l\'{\i}neas\\
alineadas a la derecha.
\end{flushright}
```

5.8. Algunas herramientas importantes

Hasta ahora hemos mencionado escencialmente comandos disponibles en LaTeX standard. Sin embargo, éstos, junto con el resto de los comandos básicos de LaTeX, se vuelven insuficientes cuando

se trata de ciertas aplicaciones demasiado específicas, pero no inimaginables: si queremos escribir un texto de alta matemática, o usar LaTeX para escribir partituras, o para escribir un archivo `.tex` en un teclado croata.... Es posible que con los comandos usuales LaTeX responda a las necesidades, pero seguramente ello será a un costo grande de esfuerzo por parte del autor del texto. Por esta razón, las distribuciones modernas de LaTeX incorporan una serie de extensiones que hacen la vida un poco más fácil a los eventuales autores. En esta sección mencionaremos algunas extensiones muy útiles. Muchas otras no están cubiertas, y se sugiere al lector consultar la documentación de su distribución para saber qué otros paquetes se encuentran disponibles.

En general, las extensiones a LaTeX vienen contenidas en *paquetes* (*"packages"*, en inglés), en archivos `.sty`. Así, cuando mencionemos el paquete `amsmath`, nos referimos a características disponibles en el archivo `amsmath.sty`. Para que los comandos de un paquete `<package>.sty` estén disponibles, deben ser cargados durante la compilación, incluyendo en el preámbulo del documento la línea:

```
\usepackage{<package>}
```

Si se requiere cargar más de un paquete adicional, se puede hacer de dos formas:

```
\usepackage{<package1>,<package2>}
```

o

```
\usepackage{<package1>}
\usepackage{<package2>}
```

Algunos paquetes aceptan opciones adicionales (del mismo modo que la clase `article` acepta la opción `12pt`):

```
\usepackage[option1,option2]{<package1>}
```

Revisemos ahora algunos paquetes útiles.

5.8.1. `babel`

Permite el procesamiento de textos en idiomas distintos del inglés. Esto significa, entre otras cosas, que se incorporan los patrones de silabación correctos para dicho idioma, para cortar adecuadamente las palabras al final de cada línea. Además, palabras claves como "Chapter", "Index", "List of Figures", etc., y la fecha dada por `\date`, son cambiadas a sus equivalentes en el idioma escogido. La variedad de idiomas disponibles es enorme, pero cada instalación de LaTeX tiene sólo algunos de ellos incorporados. (Ésta es una decisión que toma el administrador del sistema, de acuerdo a las necesidades de los usuarios. Una configuración usual puede ser habilitar la compilación en inglés, castellano, alemán y francés.)

Ya sabemos como usar `babel` para escribir en castellano: basta incluir en el preámbulo la línea

```
\usepackage[spanish]{babel}
```

5.8.2. $\mathcal{AMS}$-LaTeX

El paquete `amsmath` permite agregar comandos para escritura de textos matemáticos profesionales, desarrollados originalmente por la American Mathematical Society. Si un texto contiene abundante matemática, entonces seguramente incluir la línea correspondiente en el preámbulo:

```
\usepackage{amsmath}
```

aliviará mucho la tarea. He aquí algunas de las características adicionales disponibles con $\mathcal{AMS}$-LaTeX.

Ambientes para ecuaciones

Con `equation*` generamos una ecuación separada del texto, no numerada:

$$x = 2y - 3$$

```
\begin{equation*}
x = 2y - 3
\end{equation*}
```

`multline` permite dividir una ecuación muy larga en varias líneas, de modo que la primera línea quede alineada con el margen izquierdo, y la última con el margen derecho:

$$\sum_{i=1}^{15} = 1 + 2 + 3 + 4 + 5+ \\ 6 + 7 + 8 + 9 + 10+ \\ 11 + 12 + 13 + 14 + 15 \quad (5.6)$$

```
\begin{multline}
\sum_{i=1}^{15} = 1 +2+3+4+5+\\
6+7+8+9+10+\\
11+12+13+14+15
\end{multline}
```

`align` permite reunir un grupo de ecuaciones consecutivas alineándolas (usando `&`, igual que la alineación vertical de `tabular` y `array`). `gather` hace lo mismo, pero centrando cada ecuación en la página independientemente.

$$a_1 = b_1 + c_1 \quad (5.7)$$
$$a_2 = b_2 + c_2 - d_2 + e_2 \quad (5.8)$$

```
\begin{align}
a_1 &= b_1 + c_1 \\
a_2 &= b_2 + c_2 - d_2 + e_2
\end{align}
```

$$a_1 = b_1 + c_1 \quad (5.9)$$
$$a_2 = b_2 + c_2 - d_2 + e_2 \quad (5.10)$$

```
\begin{gather}
a_1 = b_1 + c_1 \\
a_2 = b_2 + c_2 - d_2 + e_2
\end{gather}
```

Con `multline*`, `align*` y `gather*` se obtienen los mismos resultados, pero con ecuaciones no numeradas.

`split` permite escribir una sola ecuación separada en líneas (como `multline`), pero permite alinear las líneas con `&` (como `align`). `split` debe ser usado dentro de un ambiente como `equation`, `align` o `gather` (o sus equivalentes con asterisco):

$$\begin{aligned} a_1 &= b_1 + c_1 \\ &= b_2 + c_2 - d_2 + e_2 \end{aligned} \tag{5.11}$$

```
\begin{equation}
\begin{split}
a_1& = b_1 + c_1 \\
   & =  b_2 + c_2 - d_2 + e_2
\end{split}
\end{equation}
```

Espacio horizontal

`\quad` y `\qquad` insertan espacio horizontal en ecuaciones:

$$x > y \ , \quad \forall x \in A$$
$$x \leq z \ , \qquad \forall z \in B$$

```
\begin{gather*}
x > y \ , \quad \forall\, x \in A \\
x \leq z \ , \qquad \forall\, z \in B
\end{gather*}
```

Texto en ecuaciones

Para agregar texto a una ecuación, usamos `\text`:

$$x = 2^n - 1 \ , \quad \text{con } n \text{ entero}$$

```
\begin{equation*}
x = 2^n - 1 \ , \quad \text{con $n$ entero}
\end{equation*}
```

`\text` se comporta como un buen objeto matemático, y por tanto se pueden agregar subíndices textuales más fácilmente que con `\mbox` (ver sección 5.4.11):

$V_{\text{crítico}}$

```
$V_{\text{cr\'{\i}tico}}$
```

Referencia a ecuaciones

`\eqref` es equivalente a `\ref`, salvo que agrega los paréntesis automáticamente:

La ecuación (5.5) era la de Euler.

```
La ecuaci\'on \eqref{euler} era la de Euler.
```

Ecuaciones con casos

Ésta es una construcción usual en matemáticas:

$$f(x) = \begin{cases} 1 & \text{si } x < 0 \\ 0 & \text{si } x > 0 \end{cases}$$

```
f(x)=
\begin{cases}
1&\text{si $x<0$} \\
0&\text{si $x>0$}
\end{cases}
```

Notar cómo es más simple que el ejemplo con los comandos convencionales en la sección 5.4.9.

Texto insertado entre ecuaciones alineadas

Otra situación usual es insertar texto entre ecuaciones alineadas, preservando la alineación:

$$x_1 = a + b + c \ ,$$
$$x_2 = d + e \ ,$$

y por otra parte

$$x_3 = f + g + h \ .$$

```
\begin{align*}
x_1 &=  a + b + c  \ , \\
x_2 &=  d + e \ , \\
\intertext{y por otra parte}
x_3 &=  f + g + h \ .
\end{align*}
```

Matrices y coeficientes binomiales

La complicada construcción de matrices usando `array` (sección 5.4.9), se puede reemplazar con ambientes como `pmatrix` y `vmatrix`, y comandos como `\binom`.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

```
\begin{pmatrix}
a&b\\
c&d
\end{pmatrix}
```

$$\Delta = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

```
\Delta = \begin{vmatrix}
a_{11} & a_{12}\\
a_{21} & a_{22}
\end{vmatrix}
```

$$v = \binom{k}{2}$$

```
v = \binom{k}{2}
```

Podemos observar que el espaciado entre los paréntesis y el resto de la fórmula es más adecuado que el de los ejemplos en la sección 5.4.9.

Flechas extensibles

Las flechas en la tabla 5.6 vienen en ciertos tamaños predefinidos. `amsmath` proporciona flechas extensibles `\xleftarrow` y `\xrightarrow`, para ajustar sub o superíndices demasiado anchos. Además, tienen un argumento opcional y uno obligatorio, para colocar material sobre o bajo ellas:

$$A \xleftarrow{n+\mu-1} B \xrightarrow[T]{n\pm i-1} C \underset{U}{\longrightarrow} D$$

```
A \xleftarrow{n+\mu-1} B \xrightarrow[T]{n\pm i-1} C \xrightarrow[U]{} D
```

5.8.3. `fontenc`

Ocasionalmente, LaTeX tiene problemas al separar una palabra en sílabas. Típicamente, eso ocurre con palabras acentuadas, pues, debido a la estructura interna del programa, un carácter como la "á" en "matemáticas" no es tratado igual que los otros. Para solucionar el problema, y poder cortar en sílabas palabras que contengan letras acentuadas (además de acceder a algunos caracteres adicionales), basta incluir el paquete `fontenc`:

```
\usepackage[T1]{fontenc}
```

Técnicamente, lo que ocurre es que la codificación antigua para fonts es la `OT1`, que no contiene fonts acentuados, y que por lo tanto es útil sólo para textos en inglés. La codificación `T1` aumenta los fonts disponibles, permitiendo que los caracteres acentuados sean tratados en igual pie que cualquier otro.

5.8.4. enumerate

`enumerate.sty` define una muy conveniente extensión al ambiente `enumerate` de LaTeX. El comando se usa igual que siempre (ver sección 5.3.12), con un argumento opcional que determina el tipo de etiqueta que se usará para la lista. Por ejemplo, si queremos que en vez de números se usen letras mayúsculas, basta usar `\begin{enumerate}[A]`:

A Primer ítem.

B Segundo ítem.

Si queremos etiquetas de la forma "1.-", `\begin{enumerate}[1.-]`:

1.- Primer ítem.

2.- Segundo ítem.

Si deseamos insertar un texto que no cambie de una etiqueta a otra, hay que encerrarlo entre paréntesis cursivos (`\begin{enumerate}[{Caso} A:]`):

Caso A: Primer ítem.

Caso B: Segundo ítem.

5.8.5. Color.

A través de PostScript es posible introducir color en documentos LaTeX. Para ello, incluimos en el preámbulo el paquete `color.sty`:

```
\usepackage{color}
```

De este modo, está disponible el comando `\color`, que permite especificar un color, ya sea por nombre (en el caso de algunos colores predefinidos), por su código `rgb` (red-green-blue) o código `cmyk` (cian-magenta-yellow-black). Por ejemplo:

Un texto en azul
Un texto en un segundo color
Un texto en un tercer color

```
Un texto en {\color{blue} azul}

Un texto en un
{\color[rgb]{1,0,1} segundo color}

Un texto en un
{\color[cmyk]{.3,.5,.75,0} tercer color}
```

Los colores más frecuentes (azul, amarillo, rojo, etc.) se pueden dar por nombre, como en este ejemplo. Si se da el código `rgb`, se deben especificar tres números entre 0 y 1, que indican la cantidad

de rojo, verde y azul que constituyen el color deseado. En el ejemplo, le dimos máxima cantidad de rojo y azul, y nada de verde, con lo cual conseguimos un color violeta. Si se trata del código `cmyk` los números a especificar son cuatro, indicando la cantidad de cian, magenta, amarillo y negro. En el ejemplo anterior pusimos una cantidad arbitraria de cada color, y resultó un color café. Es evidente que el uso de los códigos `rgb` y `cmyk` permite explorar infinidad de colores.

Observar que `\color` funciona de modo análogo a los comandos de cambio de font de la sección 5.3.13, de modo que si se desea restringir el efecto a una porción del texto, hay que encerrar dicho texto entre paréntesis cursivos. Análogamente al caso de los fonts, existe el comando `\textcolor`, que permite dar el texto a colorear como argumento:

Un texto en azul
Un texto en un segundo color
Un texto en un tercer color

```
Un texto en \textcolor{blue}{azul}

Un texto en un
\textcolor[rgb]{1,0,1}{segundo color}

Un texto en un
\textcolor[cmyk]{.3,.5,.75,0}{tercer color}
```

5.9. Modificando el estilo de la página.

TEX toma una serie de decisiones por nosotros. Ocasionalmente nos puede interesar alterar el comportamiento normal. Disponemos de una serie de comandos para ello, los cuales revisaremos a continuación. Todos deben aparecer en el preámbulo, salvo en los casos que se indique.

5.9.1. Estilos de página.

a) Números de página.

Si se desea que los números de página sean arábicos (1, 2, 3...):

```
\pagenumbering{arabic}
```

Para números romanos (i, ii, iii,...):

```
\pagenumbering{roman}
```

`arabic` es el default.

b) Estilo de página.

El comando `\pagestyle` determina dónde queremos que vayan los números de página:

`\pagestyle{plain}`	Números de página en el extremo inferior, al centro de la página. (Default para estilos `article`, `report`.)
`\pagestyle{headings}`	Números de página y otra información (título de sección, etc.) en la parte superior de la página. (Default para estilo `book`.)
`\pagestyle{empty}`	Sin números de página.

5.9.2. Corte de páginas y líneas.

TEX tiene modos internos de decidir cuándo cortar una página o una línea. Al preparar la versión final de nuestro documento, podemos desear coartar sus decisiones. En todo caso, no hay que hacer esto antes de preparar la versión verdaderamente final, porque agregar, modificar o quitar texto puede alterar los puntos de corte de líneas y páginas, y los cortes inconvenientes pueden resolverse solos.

Los comandos de esta sección no van en el preámbulo, sino en el interior del texto.

Corte de líneas.

En la página 159 ya vimos un ejemplo de inducción de un corte de línea en un punto deseado del texto, al dividir una palabra en sílabas.

Cuando el problema no tiene relación con sílabas disponemos de dos comandos:

`\newline`	Corta la línea y pasa a la siguiente en el punto indicado.
`\linebreak`	Lo mismo, pero justificando la línea para adecuarla a los márgenes.

Un corte de línea
no justificado a los márgenes
en curso.

```
Un corte de l\'{\i}nea\newline
no justificado a los m\'argenes
en curso.
```

Un corte de línea
justificado a los márgenes
en curso.

```
Un corte de l\'{\i}nea\linebreak
justificado a los m\'argenes
en curso.
```

Observemos cómo en el segundo caso, en que se usa `\linebreak`, la separación entre palabras es alterada para permitir que el texto respete los márgenes establecidos.

Corte de páginas.

Como para cortar líneas, existe un modo violento y uno sutil:

`\newpage`	Cambia de página en el punto indicado. Análogo a `\newline`.
`\clearpage`	Lo mismo, pero ajustando los espacios verticales en el texto para llenar del mejor modo posible la página.

`\clearpage`, sin embargo, no siempre tiene efectos visibles. Dependiendo de la cantidad y tipo de texto que quede en la página, los espacios verticales pueden o no ser ajustados, y si no lo son, el resultado termina siendo equivalente a un `\newpage`. TEX decide en última instancia qué es lo óptimo.

Adicionalmente, tenemos el comando:

`\enlargethispage{<longitud>}`	Cambia el tamaño de la página actual en la cantidad `<longitud>`.

(Las unidades de longitud que maneja TEX se revisan a continuación.)

Unidades de longitud y espacios.

a) Unidades.

TEX reconoce las siguientes unidades de longitud:

`cm`	centímetro
`mm`	milímetro
`in`	pulgada
`pt`	punto (1/72 pulgadas)
`em`	ancho de una "M" en el font actual
`ex`	altura de una "x" en el font actual

Las cuatro primeras unidades son absolutas; las últimas dos, relativas, dependiendo del tamaño del font actualmente en uso.

Las longitudes pueden ser números enteros o decimales, positivos o negativos:

```
1cm     1.6in     .58pt     -3ex
```

b) Cambio de longitudes.

TEX almacena los valores de las longitudes relevantes al texto en comandos especiales:

`\parindent`	Sangría.
`\textwidth`	Ancho del texto.
`\textheight`	Altura del texto.
`\oddsidemargin`	Margen izquierdo menos 1 pulgada.
`\topmargin`	Margen superior menos 1 pulgada.
`\baselineskip`	Distancia entre la base de dos líneas de texto consecutivas.
`\parskip`	Distancia entre párrafos.

Todas estas variables son modificables con los comandos `\setlength`, que le da a una variable un valor dado, y `\addtolength`, que le suma a una variable la longitud especificada. Por ejemplo:

`\setlength{\parindent}{0.3em}` (`\parindent` = 0.3 cm.)

`\addtolength{\parskip}{1.5cm}` (`\parskip` = `\parskip` + 1.5 cm.)

Por default, el ancho y altura del texto, y los márgenes izquierdo y superior, están definidos de modo que quede un espacio de una pulgada ($\simeq$ 2.56 cm) entre el borde del texto y el borde de la página.

Un problema típico es querer que el texto llene un mayor porcentaje de la página. Por ejemplo, para que el margen del texto en los cuatro costados sea la mitad del default, debemos introducir los comandos:

```
\addtolength{\textwidth}{1in}
\addtolength{\textheight}{1in}
\addtolength{\oddsidemargin}{-.5in}
\addtolength{\topmargin}{-.5in}
```

Las dos primeras líneas aumentan el tamaño horizontal y vertical del texto en 1 pulgada. Si luego restamos media pulgada del margen izquierdo y el margen superior, es claro que la distancia entre el texto y los bordes de la página sera de media pulgada, como deseábamos.

c) Espacios verticales y horizontales.

Se insertan con `\vspace` y `\hspace`:

`\vspace{3cm}`	Espacio vertical de 3 cm.
`\hspace{3cm}`	Espacio horizontal de 3 cm.

Algunos ejemplos:

Un primer párrafo de un pequeño texto.

Y un segundo párrafo separado del otro.

Tres palabras separadas del resto.

```
Un primer p\'arrafo de un
peque\~no texto.

\vspace{1cm}
Y un segundo p\'arrafo
separado del otro.

Tres\hspace{.5cm}palabras
\hspace{.5cm}separadas
del resto.
```

Si por casualidad el espacio vertical impuesto por `\vspace` debiese ser colocado al comienzo de una página, TEX lo ignora. Sería molesto visualmente que en algunas páginas el texto comenzara algunos centímetros más abajo que en el resto. Lo mismo puede ocurrir si el espacio horizontal de un `\hspace` queda al comienzo de una línea.

Los comandos `\vspace*{<longitud>}` y `\hspace*{<longitud>}` permiten que el espacio en blanco de la `<longitud>` especificada no sea ignorado. Ello es útil cuando invariablemente queremos ese espacio vertical u horizontal, aunque sea al comienzo de una página o una línea —por ejemplo, para insertar una figura.

5.10. Figuras.

Lo primero que hay que decir en esta sección es que LATEX es un excelente procesador de texto, tanto convencional como matemático. Las figuras, sin embargo, son un problema aparte.

LATEX provee un ambiente `picture` que permite realizar dibujos simples. Dentro de la estructura `\begin{picture}` y `\end{picture}` se pueden colocar una serie de comandos para dibujar líneas, círculos, óvalos y flechas, así como para posicionar texto. Infortunadamente, el proceso de

ejecutar dibujos sobre un cierto umbral de complejidad puede ser muy tedioso para generarlo directamente. Existe software (por ejemplo, `xfig`) que permite superar este problema, pudiéndose dibujar con el mouse, exportando el resultado al formato `picture` de LaTeX. Sin embargo, `picture` tiene limitaciones (no se pueden hacer líneas de pendiente arbitraria), y por tanto no es una solución óptima.

Para obtener figuras de buena calidad es imprescindible recurrir a lenguajes gráficos externos, y LaTeX da la posibilidad de incluir esos formatos gráficos en un documento. De este modo, tanto el texto como las figuras serán de la más alta calidad. Las dos mejores soluciones son utilizar Metafont o PostScript. Metafont es un programa con un lenguaje de programación gráfico propio. De hecho, los propios fonts de LaTeX fueron creados usando Metafont, y sus capacidades permiten hacer dibujos de complejidad arbitraria. Sin embargo, los dibujos resultantes no son trivialmente reescalables, y exige aprender un lenguaje de programación específico.

Una solución mucho más versátil, y adoptada como el estándar en la comunidad de usuarios de LaTeX, es el uso de PostScript. Como se mencionó brevemente en la sección 1.13, al imprimir, una máquina UNIX convierte el archivo a formato PostScript, y luego lo envía a la impresora. Pero PostScript sirve más que para imprimir, siendo un lenguaje de programación gráfico completo, con el cual podemos generar imágenes de gran calidad, y reescalables sin pérdida de resolución. Además, muchos programas gráficos permiten exportar sus resultados en formato PostScript. Por lo tanto, podemos generar nuestras figuras en alguno de estos programas (`xfig` es un excelente software, que satisface la mayor parte de nuestras necesidades de dibujos simples; `octave` o `gnuplot` pueden ser usados para generar figuras provenientes de cálculos científicos, etc.), lo cual creará un archivo con extensión `.ps` (PostScript) o `.eps` (PostScript encapsulado).[4] Luego introducimos la figura en el documento LaTeX, a través del paquete `graphicx`.

5.10.1. `graphicx.sty`

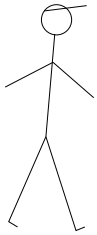
Si nuestra figura está en un archivo `figura.eps`, la instrucción a utilizar es:

```
\documentclass[12pt]{article}
\usepackage{graphicx}
\begin{document}
... Texto ...
\includegraphics[width=w, height=h]{figura.eps}
...
\end{document}
```

Los parámetros `width` y `height` son opcionales y puede omitirse uno para que el sistema escale de acuerdo al parámetro dado. Es posible variar la escala completa de la figura o rotarla usando comandos disponibles en `graphicx`.

[4] `eps` es el formato preferido, pues contiene información sobre las dimensiones de la figura, información que es utilizada por LaTeX para insertar ésta adecuadamente en el texto.

Una figura aquí:



puede hacer más agradable el
texto.

```
Una figura aqu\'{\i}:

\begin{center}
\includegraphics[height=3cm]{figura.eps}
\end{center}

puede hacer m\'as agradable
el texto.
```

En este ejemplo, indicamos sólo la altura de la figura (3cm). El ancho fue determinado de modo que las proporciones de la figura no fueran alteradas. Si no se especifica ni la altura ni el ancho, la figura es insertada con su tamaño natural.

Observemos también que pusimos la figura en un ambiente `center`. Esto no es necesario, pero normalmente uno desea que las figuras estén centradas en el texto.

5.10.2. Ambiente `figure`.

Insertar una figura es una cosa. Integrarla dentro del texto es otra. Para ello está el ambiente `figure`, que permite: (a) posicionar la figura automáticamente en un lugar predeterminado o especificado por el usuario; (b) numerar las figuras; y (c) agregar un breve texto explicativo junto a la figura.

Coloquemos la misma figura de la sección anterior dentro de un ambiente `figure`. El input:

```
\begin{figure}[h]
\begin{center}
\includegraphics[height=3cm]{figura.eps}
\end{center}
\caption{Un sujeto caminando.}
\label{caminando}
\end{figure}
```

da como resultado:

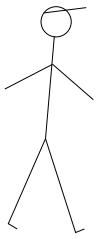


Figura 5.1: Un sujeto caminando.

`figure` delimita lo que en TEX se denomina un *objeto flotante*, es decir, un objeto cuya posición no está determinada *a priori*, y se ajusta para obtener los mejores resultados posibles. TEX considera (de acuerdo con la tradición), que la mejor posición para colocar una figura es al principio o al final

de la página. Además, lo ideal es que cada página tenga un cierto número máximo de figuras, que ninguna figura aparezca en el texto antes de que sea mencionada por primera vez, y que, por supuesto, las figuras aparezcan en el orden en que son mencionadas. Éstas y otras condiciones determinan la posición que un objeto flotante tenga al final de la compilación. Uno puede forzar la decisión de LaTeX con el argumento opcional de `figure`:

`t`	(*top*)	extremo superior de la página
`b`	(*bottom*)	extremo inferior de la página
`h`	(*here*)	aquí, en el punto donde está el comando
`p`	(*page of floats*)	en una página separada al final del texto

El argumento adicional `!` suprime, para ese objeto flotante específico, cualquier restricción que exista sobre el número máximo de objetos flotantes en una página y el porcentaje de texto mínimo que debe haber en una página.

Varios de estos argumentos se pueden colocar simultánemente, su orden dictando la prioridad. Por ejemplo,

```
\begin{figure}[htbp]
...
\end{figure}
```

indica que la figura se debe colocar como primera prioridad aquí mismo; si ello no es posible, al comienzo de página (ésta o la siguiente, dependiendo de los detalles de la compilación), y así sucesivamente.

Además, `figure` numera automáticamente la figura, colocando el texto "Figura N:", y `\caption` permite colocar una leyenda, centrada en el texto, a la figura. Puesto que la numeración es automática, las figuras pueden ser referidas simbólicamente con `\label` y `\ref` (sección 5.6). Para que la referencia sea correcta, `\label` debe estar dentro del argumento de `\caption`, o después, como aparece en el ejemplo de la Figura 5.1 (`\ref{caminando}`!).

Finalmente, notemos que la figura debió ser centrada explícitamente con `center`. `figure` no hace nada más que tratar la figura como un objeto flotante, proporcionar numeración y leyenda. El resto es responsabilidad del autor.

5.11. Cartas.

Para escribir cartas debemos emplear el estilo `letter` en vez del que hemos utilizado hasta ahora, `article`. Comandos especiales permiten escribir una carta, poniendo en lugares adecuados la dirección del remitente, la fecha, la firma, etc.

A modo de ejemplo, consideremos el siguiente input:

```
\documentclass[12pt]{letter}

\usepackage[spanish]{babel}

\begin{document}

\address{Las Palmeras 3425\\
\~Nu\~noa, Santiago}
\date{9 de Julio de 1998}
```

```
\signature{Pedro P\'erez \\ Secretario}

\begin{letter}{Dr.\ Juan P\'erez \\ Las Palmeras 3425 \\
\~Nu\~noa, Santiago}
\opening{Estimado Juan}

A\'un no tenemos novedades.

Parece incre\'{\i}ble, pero los recientes acontecimientos nos han superado,
a pesar de nuestros esfuerzos.  Esperamos que mejores tiempos nos
aguarden.

\closing{Saludos,}
\cc{Arturo Prat \\ Luis Barrios}

\end{letter}
\end{document}
```

El resultado se encuentra en la próxima página.

Las Palmeras 3425
Ñuñoa, Santiago

9 de Julio de 1998

Dr. Juan Pérez
Las Palmeras 3425
Ñuñoa, Santiago

Estimado Juan

Aún no tenemos novedades.

Parece increíble, pero los recientes acontecimientos nos han superado, a pesar de nuestros esfuerzos. Esperamos que mejores tiempos nos aguarden.

Saludos,

Pedro Pérez
Secretario

Copia a: Arturo Prat
Luis Barrios

Observemos que el texto de la carta está dentro de un ambiente `letter`, el cual tiene un argumento obligatorio, donde aparece el destinatario de la carta (con su dirección opcionalmente).

Los comandos disponibles son:

`\address{<direccion>}`	`<direccion>` del remitente.
`\signature{<firma>}`	`<firma>` del remitente.
`\opening{<apertura>}`	Fórmula de `<apertura>`.
`\closing{<despedida>}`	Fórmula de `<despedida>`.
`\cc{<copias>}`	Receptores de `<copias>` (si los hubiera).

Uno puede hacer más de una carta con distintos ambientes `letter` en un mismo archivo. Cada una tomará el mismo remitente y firma dados por `\address` y `\signature`. Si deseamos que `\address` o `\signature` valgan sólo para una carta particular, basta poner dichos comandos entre el `\begin{letter}` y el `\opening` correspondiente.

Por ejemplo, la siguiente estructura:

```
\documentclass[12pt]{letter}
\begin{document}
\address{<direccion remitente>}
\date{<fecha>}
\signature{<firma>}

\begin{letter}{<destinatario 1>}
\opening<apertura 1>
...
\end{letter}

\begin{letter}{<destinatario 2>}
\address{<direccion remitente 2>}
\signature{<firma 2>}
\opening<apertura 2>
...
\end{letter}

\begin{letter}{<destinatario 3>}
\opening<apertura 3>
...
\end{letter}
\end{document}
```

dará origen a tres cartas con la misma dirección de remitente y firma, salvo la segunda.

En todos estos comandos, líneas sucesivas son indicadas con `\\`.

5.12. LaTeX y el formato pdf.

Junto con PostScript, otro formato ampliamente difundido para la transmisión de archivos, especialmente a través de Internet, es el formato `pdf` (Portable Document Format). Para generar un

archivo `pdf` con LaTeX es necesario compilarlo con `pdflatex`. Así, `pdflatex <archivo>` generará un archivo `<archivo>.pdf` en vez del `<archivo>.dvi` generado por el compilador usual.

Si nuestro documento tiene figuras, sólo es posible incluirlas en el documento si están también en formato `pdf`. Por tanto, si tenemos un documento con figuras en PostScript, debemos introducir dos modificaciones antes de compilar con `pdflatex`:

a) Cambiar el argumento de `\includegraphics` (sección 5.10) de `<archivo_figura>.eps` a `<archivo_figura>.pdf`.

b) Convertir las figuras PostScript a `pdf` (con `epstopdf`, por ejemplo). Si tenemos una figura en el archivo `<archivo_figura>.eps`, entonces `epstopdf <archivo_figura>.eps` genera el archivo correspondiente `<archivo_figura>.pdf`.

Observar que el mismo paquete `graphicx` descrito en la sección 5.10 para incluir figuras PostScript permite, sin modificaciones, incluir figuras en `pdf`.

5.13. Modificando LaTeX.

Esta sección se puede considerar "avanzada". Normalmente uno se puede sentir satisfecho con el desempeño de LaTeX, y no es necesaria mayor intervención. A veces, dependiendo de la aplicación y del autor, nos gustaría modificar el comportamiento default. Una alternativa es definir nuevos comandos que sean útiles para nosotros. Si esos nuevos comandos son abundantes, o queremos reutilizarlos frecuentemente en otros documentos, lo conveniente es considerar crear un nuevo paquete o incluso una nueva clase. Examinaremos a continuación los elementos básicos de estas modificaciones.

5.13.1. Definición de nuevos comandos.

El comando `\newcommand`

Un nuevo comando se crea con:

```
\newcommand{<comando>}{<accion>}
```

El caso más sencillo es cuando una estructura se repite frecuentemente en nuestro documento. Por ejemplo, digamos que un sujeto llamado Cristóbal no quiere escribir su nombre cada vez que aparece en su documento:

Mi nombre es Cristóbal.
Sí, como oyes, Cristóbal.
Cristóbal Loyola.

```
\newcommand{\nombre}{Crist\'obal}
...
\begin{document}
...
Mi nombre es \nombre. S\'{\i}, como oyes,
\nombre. \nombre\ Loyola.
```

Un `\newcommand` puede aparecer en cualquier parte del documento, pero lo mejor es que esté en el preámbulo, de modo que sea evidente qué nuevos comandos están disponibles en el presente documento. Observemos además que la definición de un comando puede contener otros comandos (en este caso, `\'`). Finalmente, notamos que ha sido necesario agregar un espacio explícito con `\ `, al escribir "Cristóbal Loyola": recordemos que un comando comienza con un backslash y termina

con el primer carácter que no es letra. Por tanto, `\nombre Loyola` ignora el espacio al final de `\nombre`, y el output sería "CristóbalLoyola".

También es posible definir comandos que funcionen en modo matemático:

Sea $\dot{x}$ la velocidad, de modo que $\dot{x}(t) > 0$ si $t < 0$.

```
\newcommand{\vel}{\dot x}

Sea $\vel$ la velocidad, de modo que
$ \vel(t)> 0$ si $t<0$.
```

Como `\vel` contiene un comando matemático (`\dot`), `\vel` sólo puede aparecer en modo matemático.

Podemos también incluir la apertura de modo matemático en la definición de `\vel`: `\newcommand{\vel}{$\dot x$}`. De este modo, `\vel` (no `$\vel$`) da como output directamente $\dot{x}$. Sin embargo, esta solución no es óptima, porque la siguiente ocurrencia de `\vel` da un error. En efecto, si `\vel` = `$\dot x$`, entonces `$ \vel(t)>0$` = `$ $\dot x$> 0$`. En tal caso, LaTeX ve que un modo matemático se ha abierto y cerrado inmediatamente, conteniendo sólo un espacio entremedio, y luego, *en modo texto*, viene el comando `\dot`, que es matemático: LaTeX acusa un error y la compilación se detiene.

La solución a este problema es utilizar el comando `\ensuremath`, que asegura que haya modo matemático, pero si ya hay uno abierto, no intenta volverlo a abrir:

Sea $\dot{x}$ la velocidad, de modo que $\dot{x}(t) > 0$ si $t < 0$.

```
\newcommand{\vel}{\ensuremath{\dot x}}

Sea \vel\ la velocidad, de modo que
$ \vel(t)> 0$ si $t<0$.
```

Un caso especial de comando matemático es el de operadores tipo logaritmo (ver Tabla 5.9). Si queremos definir una traducción al castellano de `\sin`, debemos usar el comando `\DeclareMathOperator` disponible via `amsmath`:

Ahora podemos escribir en castellano, $\operatorname{sen} x$.

```
\usepackage{amsmath}
\DeclareMathOperator{\sen}{sen}
...
Ahora podemos escribir en castellano, $\sen x$.
```

A diferencia de `\newcommand`, `\DeclareMathOperator` sólo puede aparecer en el preámbulo del documento.

Un nuevo comando puede también ser usado para ahorrar tiempo de escritura, reemplazando comandos largos de LaTeX:

1. El primer caso.
2. Ahora el segundo.
3. Y el tercero.

```
\newcommand{\be}{\begin{enumerate}}
\newcommand{\ee}{\end{enumerate}}

\be
\item El primer caso.
\item Ahora el segundo.
\item Y el tercero.
\ee
```

Nuevos comandos con argumentos

Podemos también definir comandos que acepten argumentos. Si el sujeto anterior, Cristóbal, desea escribir cualquier nombre precedido de "Nombre:" en itálica, entonces puede crear el siguiente comando:

Nombre: Cristóbal
Nombre: Violeta

```
\newcommand{\nombre}[1]{\textit{Nombre:} #1}

\nombre{Crist\'obal}

\nombre{Violeta}
```

Observemos que `\newcommand` tiene un argumento opcional, que indica el número de argumentos que el nuevo comando va a aceptar. Esos argumentos se indican, dentro de la definición del comando, con `#1`, `#2`, etc. Por ejemplo, consideremos un comando que acepta dos argumentos:

$f(x, y) + f(x_3, y*) = 0$.

```
\newcommand{\fn}[2]{f(#1,#2)}

$$ \fn{x}{y} + \fn{x_3}{y*} = 0 \ . $$
```

En los casos anteriores, todos los argumentos son obligatorios. LaTeX permite definir comandos con un (sólo un) argumento opcional. Si el comando acepta n argumentos, el argumento opcional es el `#1`, y se debe indicar, en un segundo paréntesis cuadrado, su valor default. Así, podemos modificar el comando `\fn` del ejemplo anterior para que el primer argumento sea opcional, con valor default x:

$f(x, y) + f(x_3, y*) = 0$.

```
\newcommand{\fn}[2][x]{f(#1,#2)}

$$ \fn{y} + \fn[x_3]{y*} = 0 \ . $$
```

Redefinición de comandos

Ocasionalmente no nos interesa definir un nuevo comando, sino redefinir la acción de un comando preexistente. Esto se hace con `\renewcommand`:

La antigua versión de `ldots`: La antigua versi\'on de {\tt ldots}: \ldots
...
La nueva versión de `ldots`: •••

```
\renewcommand{\ldots}{\textbullet \textbullet
\textbullet}

La nueva versi\'on de {\tt ldots}: \ldots
```

Párrafos y cambios de línea dentro de comandos

En el segundo argumento de `\newcommand` o `\renewcommand` puede aparecer cualquier comando de LaTeX, pero ocasionalmente la aparición de líneas en blanco (para forzar un cambio de párrafo) puede provocar problemas. Si ello ocurre, podemos usar `\par`, que hace exactamente lo mismo. Además, la definición del comando queda más compacta:

```
\newcommand{\comandolargo}{\par Un nuevo comando que incluye un cambio de
  p\'arrafo, porque deseamos incluir bastante texto.\par \'Este es el
  nuevo p\'arrafo.\par}

Observemos en acci\'on el comando: \comandolargo Listo.
```

da como resultado:

Observemos en acción el comando:
Un nuevo comando que incluye un cambio de párrafo, porque deseamos incluir bastante texto.
Éste es el nuevo párrafo.
Listo.

Un ejemplo más útil ocurre cuando queremos asegurar un cambio de párrafo, por ejemplo, para colocar un título de sección:

Observemos en acción el comando:

Sección: Ejemplo

Listo.

```
\newcommand{\seccion}[1]{\par\vspace{.5cm}
{\bf Secci\'on: #1}\par\vspace{.5cm}}

Observemos en acci\'on el comando:
\seccion{Ejemplo} Listo.
```

Además de las líneas en blanco, los cambios de línea pueden causar problemas dentro de la definición de un nuevo comando. El ejemplo anterior, con el comando `\seccion`, es un buen ejemplo: notemos que cuando se definió, pusimos un cambio de línea después de `\vspace{.5cm}`. Ese cambio de línea es interpretado (como todos los cambios de línea) como un espacio en blanco, y es posible que, bajo ciertas circunstancias, ese espacio en blanco produzca un output no deseado. Para ello basta utilizar sabiamente el carácter `%`, que permite ignorar todo el resto de la línea, *incluyendo el cambio de línea*. Ilustremos lo anterior con los siguientes tres comandos, que subrayan (comando `\underline`) una palabra, y difieren sólo en el uso de `%` para borrar cambios de línea:

Notar la diferencia entre:
Un texto de prueba , Un texto de prueba , y Un texto de prueba.

```
\newcommand{\texto}{
Un texto de prueba
}
\newcommand{\textodos}{%
Un texto de prueba
}
\newcommand{\textotres}{%
Un texto de prueba%
}

Notar la diferencia entre:

\underline{\texto},
\underline{\textodos},
y
\underline{\textotres}.
```

`\texto` conserva espacios en blanco antes y después del texto, `\textodos` sólo el espacio en

blanco después del texto, y `\textotres` no tiene espacios en blanco alrededor del texto.

Nuevos ambientes

Nuevos ambientes en LaTeX se definen con `\newenvironment`:

```
\newenvironment{<ambiente>}{<comienzo ambiente>}{<final ambiente>}
```

define un ambiente `<ambiente>`, tal que `\begin{ambiente}` ejecuta los comandos `<comienzo ambiente>`, y `\end{ambiente}` ejecuta los comandos `<final ambiente>`.

Definamos un ambiente que, al comenzar, cambia el font a itálica, pone una línea horizontal (`\hrule`) y deja un espacio vertical de .3cm, y que al terminar cambia de párrafo, coloca `XXX` en sans serif, deja un nuevo espacio vertical de .3cm, y vuelve al font roman:

```
\newenvironment{na}{\it \hrule \vspace{.3cm}}{\par\sf XXX \vspace{.3cm}\rm}
```

Entonces, con

```
\begin{na}
  Hola a todos. Es un placer saludarlos en este d\'{\i}a tan especial.

Nunca esper\'e una recepci\'on tan calurosa.
\end{na}
```

obtenemos:

Hola a todos. Es un placer saludarlos en este día tan especial.
Nunca esperé una recepción tan calurosa.
XXX

Los nuevos ambientes también pueden ser definidos de modo que acepten argumentos. Como con `\newcommand`, basta agregar como argumento opcional a `\newenvironment` un número que indique cuántos argumentos se van a aceptar:

```
\newenvironment{<ambiente>}[n]{<comienzo ambiente>}{<final ambiente>}
```

Dentro de `<comienzo ambiente>`, se alude a cada argumento como `#1`, `#2`, etc. Los argumentos no pueden ser usados en los comandos de cierre del ambiente (`<final ambiente>`). Por ejemplo, modifiquemos el ambiente `na` anterior, de modo que en vez de colocar una línea horizontal al comienzo, coloque lo que le indiquemos en el argumento:

```
\newenvironment{na}[1]{\it #1 \vspace{.3cm}}{\par\sf XXX\hrule\vspace{.3cm}\rm}
```

Ahora usémoslo dos veces, cada una con un argumento distinto:

El mismo ejemplo anterior, ahora es

Hola a todos...
XXX

Pero podemos ahora cambiar el comienzo:
XXX Hola a todos...

XXX

```
El mismo ejemplo anterior, ahora es

\begin{na}{\hrule}
  Hola a todos...
\end{na}

Pero podemos ahora cambiar el comienzo:

\begin{na}{\it XXX}
  Hola a todos...
\end{na}
```

5.13.2. Creación de nuevos paquetes y clases

Si la cantidad de nuevos comandos y/o ambientes que necesitamos en nuestro documento es suficientemente grande, debemos considerar crear un nuevo paquete o una nueva clase. Para ello hay que tener clara la diferencia entre uno y otro. En general, se puede decir que si nuestros comandos involucran alterar la apariencia general del documento, entonces corresponde crear una nueva clase (`.cls`). Si, por el contrario, deseamos que nuestros comandos funcionen en un amplio rango de circunstancias, para diversas apariencias del documento, entonces lo adecuado es un paquete (`.sty`).

Consideremos por ejemplo la experiencia de los autores de estos apuntes. Para crear estos apuntes necesitamos básicamente la clase `book`, con ciertas modificaciones: márgenes más pequeños, inclusión automática de los paquetes `amsmath`, `babel` y `graphicx`, entre otros, y definición de ciertos ambientes específicos. Todo ello afecta la apariencia de este documento, cambiándola de manera apreciable, pero a la vez de un modo que en general no deseamos en otro tipo de documento. Por ello lo hemos compilado usando una clase adecuada, llamada `mfm2.cls`.

Por otro lado, uno de los autores ha necesitado escribir muchas tareas, pruebas y controles de ayudantía en su vida, y se ha convencido de que su trabajo es más fácil creando una clase `tarea.cls`, que sirve para esos tres propósitos, definiendo comandos que le permiten especificar fácilmente la fecha de entrega de la tarea, o el tiempo disponible para una prueba, los nombres del profesor y el ayudante, etc., una serie de comandos específicos para sus necesidades.

Sin embargo, tanto en este documento que usa `mfm2.cls`, como en las tareas y pruebas que usan `tarea.cls`, se utilizan algunos comandos matemáticos que no vienen con LaTeX, pero que son recurrentes, como `\sen` (la función seno en castellano), `\modulo` (el módulo de un vector), o `\TLaplace` (la transformada de Laplace). Para que estos comandos estén disponibles en cualquier tipo de documento, necesitamos reunirlos en un paquete, en este caso `addmath.sty`. De este modo, `mfm2.cls`, `tarea.cls` o cualquier otra clase pueden llamar a este paquete y utilizar sus comandos.

Estructura básica.

La estructura básica de un paquete o una clase es:

a) Identificación: Información general (nombre del paquete, fecha de creación, etc.). (Obligatoria.)

b) Declaraciones preliminares: Opcionales, dependiendo del paquete o clase en cuestión.

c) Opciones: Comandos relacionados con el manejo de las opciones con las cuales el paquete o clase pueden ser invocados. (Opcional.)

d) Más declaraciones: Aquí van los comandos que constituyen el cuerpo de la clase o paquete. (Obligatoria: si no hay ninguna declaración, el paquete o clase no hace nada, naturalmente.)

La identificación está consituida por las siguientes dos líneas, que deben ir al comienzo del archivo:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{<paquete>}[<fecha> <otra informacion>]
```

La primera línea indica a LaTeX que éste es un archivo para LaTeX 2_ε. La segunda línea especifica que se trata de un paquete, indicando el nombre del mismo (es decir, el nombre del archivo sin extensión) y, opcionalmente, la fecha (en formato `YYYY/MM/DD`) y otra información relevante. Por ejemplo, nuestro paquete `addmath.sty` comienza con las líneas:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{addmath}[1998/09/30 Macros matematicos adicionales (VM)]
```

Si lo que estamos definiendo es una clase, usamos el comando `\ProvidesClass`. Para nuestra clase `mfm2.cls`:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]
```

A continuación de la identificación vienen los comandos que se desean incorporar a través de este paquete o clase.

Como hemos dicho, `addmath.sty` contiene muchos nuevos comandos matemáticos que consideramos necesario definir mientras escribíamos estos apuntes. Veamos los contenidos de una versión simplificada de dicho paquete:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{addmath}[1998/09/30 Macros matematicos adicionales (VM)]

\newcommand{\prodInt}[2]{\ensuremath \left(\, #1\, |\, #2\, \right ) }
\newcommand{\promedio}[1]{\langle #1 \rangle}
\newcommand{\intii}{\int_{-\infty}^{\infty}}
\newcommand{\grados}{\ensuremath{^\circ}}
\newcommand{\Hipergeometrica}[4]{{}_2F_1\left (#1, #2, #3\, ; #4\right )}
...
```

De este modo, incluyendo en nuestro documento el paquete con `\usepackage{addmath}`, varios nuevos comandos están disponibles:

$(x \mid y)$	`\prodInt{x}{y}`
$\langle x \rangle$	`\promedio{x}`
$\displaystyle\int_{-\infty}^{\infty} dz\, f(z)$	`\intii dz\, f(z)`
$\angle ABC = 90^\circ$	`\angle\, ABC = 90\grados`
${}_2F_1\,(a, b, c\,; d)$	`\Hipergeometrica{a}{b}{c}{d}`

Incluyendo otros paquetes y clases

Los comandos `\RequirePackage` y `\LoadClass` permiten cargar un paquete o una clase, respectivamente.[5]. Esto es de gran utilidad, pues permite construir un nuevo paquete o clase aprovechando la funcionalidad de otros ya existentes.

Así, nuestro paquete `addmath.sty` define bastantes comandos, pero nos gustaría definir varios más que sólo pueden ser creados con las herramientas de $\mathcal{AMS}$-LaTeX. Cargamos entonces en `addmath.sty` el paquete `amsmath` y otros relacionados, y estamos en condiciones de crear más comandos:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{addmath}[1998/09/30 Macros matematicos adicionales (VM)]

\RequirePackage{amsmath}
\RequirePackage{amssymb}
\RequirePackage{euscript}
...
\newcommand{\norma}[1]{\ensuremath \left\lVert\, #1 \,\right\rVert}
\newcommand{\intC}{{\sideset{^*}{}\int}}
\DeclareMathOperator{\senh}{senh}
...
```

Por ejemplo:

$\| x \|$	`\norma{x}`
${}^*\!\int dz\, f(z)$	`\intC dz \, f(z)`
$\operatorname{senh}(2y)$	`\senh (2y)`

La posibilidad de basar un archivo `.sty` o `.cls` en otro es particularmente importante para una clase, ya que contiene una gran cantidad de comandos y definiciones necesarias para compilar el documento exitosamente. Sin embargo, un usuario normal, aun cuando desee definir una nueva clase, estará interesado en modificar sólo parte del comportamiento. Con `\LoadClass`, dicho usuario puede cargar la clase sobre la cual se desea basar, y luego introducir las modificaciones necesarias, facilitando enormemente la tarea.

Por ejemplo, al preparar este documento fue claro desde el comienzo que se necesitaba esencialmente la clase `book`, ya que sería un texto muy extenso, pero también era claro que se requerían ciertas modificaciones. Entonces, en nuestra clase `mfm2.cls` lo primero que hacemos es cargar la clase `book`, más algunos paquetes necesarios (incluyendo nuestro `addmath`), y luego procedemos a modificar o añadir comandos:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]

\LoadClass[12pt]{book}

\RequirePackage[spanish]{babel}
```

[5]Estos comandos sólo se pueden usar en un archivo `.sty` o `.cls` Para documentos normales, la manera de cargar un paquete es `\usepackage`, y para cargar una clase es `\documentclass`.

```
\RequirePackage{enumerate}
\RequirePackage{addmath}
```

En un archivo `.sty` o un `.cls` se pueden cargar varios paquetes con `\RequirePackage`. `\LoadClass`, en cambio, sólo puede aparecer en un `.cls`, y sólo es posible usarlo una vez (ya que normalmente clases distintas son incompatibles entre sí).

Manejo de opciones

En el último ejemplo anterior, la clase `mfm2` carga la clase `book` con la opción `12pt`. Esto significa que si nuestro documento comienza con `\documentclass{mfm2}`, será compilado de acuerdo a la clase `book`, en 12 puntos. No es posible cambiar esto desde nuestro documento. Sería mejor que pudiéramos especificar el tamaño de letra fuera de la clase, de modo que `\documentclass{mfm2}` dé un documento en 10 puntos, y `\documentclass[12pt]{mfm2}` uno en 12 puntos. Para lograr esto hay que poder pasar opciones desde la clase `mfm2` a `book`.

El modo más simple de hacerlo es con `\LoadClassWithOptions`. Si `mfm2.cls` ha sido llamada con opciones `<opcion1>,<opcion2>, etc.`, entonces `book` será llamada con las mismas opciones. Por tanto, basta modificar en `mfm2.cls` la línea `\LoadClass[12pt]{book}` por:

```
\LoadClassWithOptions{book}
```

`\RequirePackageWithOptions` es el comando análogo para paquetes. Si una clase o un paquete llaman a un paquete `<paquete_base>` y desean pasarle todas las opciones con las cuales han sido invocados, basta indicarlo con:

```
\RequirePackageWithOptions{<paquete_base>}
```

El ejemplo anterior puede ser suficiente en muchas ocasiones, pero en general uno podría llamar a nuestra nueva clase, `mfm2`, con opciones que no tienen nada que ver con `book`. Por ejemplo, podríamos llamarla con opciones `spanish,12pt`. En tal caso, debería pasarle `spanish` a babel, y `12pt` a book. Más aún, podríamos necesitar definir una *nueva* opción, que no existe en ninguna de las clases o paquetes cargados por `book`, para modificar el comportamiento de `mfm2.cls` de cierta manera específica no prevista. Estas dos tareas, discriminar entre opciones antes de pasarla a algún paquete determinado, y crear nuevas opciones, constituyen un manejo más avanzado de opciones. A continuación revisaremos un ejemplo combinado de ambas tareas, extraido de la clase con la cual compilamos este texto, `mfm2.cls`.

La idea es poder llamar a `mfm2` con una opción adicional `keys`, que permita agregar al `dvi` información sobre las etiquetas (dadas con `\label`) de ecuaciones, figuras, etc., que aparezcan en el documento (veremos la utilidad y un ejemplo de esto más adelante). Lo primero es *declarar* una nueva opción, con:

```
\DeclareOption{<opcion>}{<comando>}
```

`<opcion>` es el nombre de la nueva opción a declarar, y `<comando>` es la serie de comandos que se ejecutan cuando dicha opción es especificada.

Así, nuestro archivo `mfm2.cls` debe ser modificado:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]
...
\DeclareOption{keys}{...}
```

```
...
\ProcessOptions\relax
...
```

Observamos que después de declarar la o las opciones (en este caso `keys`), hay que *procesarlas*, con `\ProcessOptions`.[6]

Las líneas anteriores permiten que `\documentclass{mfm2}` y `\documentclass[keys]{mfm2}` sean ambas válidas, ejecutándose o no ciertos comandos dependiendo de la forma utilizada.

Si ahora queremos que `\documentclass[keys,12pt]{mfm2}` sea una línea válida, debemos procesar `keys` dentro de `mfm2.cls`, y pasarle a `book.cls` las opciones restantes. El siguiente es el código definitivo:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{mfm2}[2002/03/25 Estilo para apuntes MFM II (VM)]
\newif\ifkeys\keysfalse
\DeclareOption{keys}{\keystrue}
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{book}}
\ProcessOptions\relax
\LoadClass{book}

\RequirePackage[spanish]{babel}
\RequirePackage{amsmath}
\RequirePackage{theorem}
\RequirePackage{epsfig}
\RequirePackage{ifthen}
\RequirePackage{enumerate}
\RequirePackage{addmath}
\ifkeys\RequirePackage[notref,notcite]{showkeys}\fi

<nuevos comandos de la clase mfm2.cls>
```

Sin entrar en demasiados detalles, digamos que la opción `keys` tiene el efecto de hacer que una cierta variable lógica `\ifkeys`, sea verdadera (cuarta línea del código). La siguiente línea (`\DeclareOption*...`) hace que todas las opciones que no han sido procesadas (`12pt`, por ejemplo) se pasen a la clase `book`. A continuación se procesan las opciones con `\ProcessOptions`, y finalmente se carga la clase `book`.

Las líneas siguientes cargan todos los paquetes necesarios, y finalmente se encuentran todos los nuevos comandos y definiciones que queremos incluir en `mfm2.cls`.

Observemos que la forma particular en que se carga el paquete `showkeys`. Ésa es precisamente la función de la opción `keys` que definimos: `showkeys.sty` se carga con ciertas opciones sólo si se da la opción `keys`.

¿Cuál es su efecto? Consideremos el siguiente texto de ejemplo, en que `mfm2` ha sido llamada sin la opción `keys`:

[6] `\relax` es un comando de TeX que, esencialmente, no hace nada, ni siquiera introduce un espacio en blanco, y es útil incluirlo en puntos críticos de un documento, como en este ejemplo.

```
\documentclass[12pt]{mfm2}
\begin{document}
La opci\'on \verb+keys+ resulta muy \'util cuando tengo objetos numerados
autom\'aticamente, como una ecuaci\'on:
\begin{equation}
  \label{newton}
  \vec F = m \vec a \ .
\end{equation}
y luego quiero referirme a ella: Ec.\  \eqref{newton}.
```

En el primer caso, se ha compilado sin la opción `keys`, y en el segundo con ella. El efecto es que, si se usa un `\label` en cualquier parte del documento, aparece en el margen derecho una caja con el nombre de dicha etiqueta (en este caso, `newton`). Esto es útil para cualquier tipo de documentos, pero lo es especialmente en textos como estos apuntes, muy extensos y con abundantes referencias. En tal caso, tener un modo visual, rápido, de saber los nombres de las ecuaciones sin tener que revisar trabajosamente el archivo fuente es una gran ayuda. Así, versiones preliminares pueden ser compiladas con la opción `keys`, y la versión final sin ella, para no confesar al lector nuestra mala memoria o nuestra comodidad.

Caso 1: `\documentclass[12pt]{mfm2}`

La opción `keys` resulta muy útil cuando tengo objetos numerados automáticamente, como una ecuación:

$$\vec{F} = m\vec{a} \ . \tag{1}$$

y luego quiero referirme a ella: Ec. (1).

Caso 2: `\documentclass[keys,12pt]{mfm2}`

La opción `keys` resulta muy útil cuando tengo objetos numerados automáticamente, como una ecuación:

$$\vec{F} = m\vec{a} \ . \tag{1}$$ `newton`

y luego quiero referirme a ella: Ec. (1).

5.14. Errores y advertencias.

5.14.1. Errores.

Un mensaje de error típico tiene la forma:

```
LaTeX error. See LaTeX manual for explanation.
             Type H <return> for immediate help.
! Environment itemie undefined.
\@latexerr ...or immediate help.}\errmessage {#1}
                                                  \endgroup
l.140 \begin{itemie}

?
```

La primera línea nos comunica que LaTeX ha encontrado un error. A veces los errores tienen que ver con procesos más internos, y son encontrados por TeX. Esta línea nos informa quién encontró el error.

La tercera línea comienza con un signo de exclamación. Éste es el indicador del error. Nos dice de qué error se trata.

Las dos líneas siguientes describen el error en términos de comandos de bajo nivel.

La línea 6 nos dice dónde ocurrió el error: la línea 140 en este caso. Además nos informa del texto conflictivo: `\begin{itemie}`.

En realidad, el mensaje nos indica dónde LaTeX advirtió el error por primera vez, que no es necesariamente el punto donde el error se cometió. Pero la gran mayoría de las veces la indicación es precisa. De hecho, es fácil darse cuenta, con la tercera línea
(`Environment itemie undefined`)
y la sexta (`\begin{itemie}`) que el error consistió en escribir `itemie` en vez de `itemize`. La información de LaTeX es clara en este caso y nos dice correctamente qué ocurrió y dónde.

Luego viene un `?`. LaTeX está esperando una respuesta de nosotros. Tenemos varias alternativas. Comentaremos sólo cuatro, típicamente usadas:

(a) `h <Enter>`

Solicitamos ayuda. TeX nos explica brevemente en qué cree él que consiste el error y/o nos da alguna recomendación.

(b) `x <Enter>`

Abortamos la compilación. Deberemos volver al editor y corregir el texto. Es la opción más típica cuando uno tiene ya cierta experiencia, pues el mensaje basta para reconocer el error.

(c) `<Enter>`

Ignoramos el error y continuamos la compilación. TeX hace lo que puede. En algunos casos esto no tiene consecuencias graves y podremos llegar hasta el final del archivo sin mayores problemas. En otros casos, ignorar el error puede provocar que ulteriores comandos —perfectamente válidos en principio— no sean reconocidos y, así, acumular

muchos errores más. Podemos continuar con `<Enter>` sucesivos hasta llegar al final de la compilación.

(d) `q <Enter>`

La acción descrita en el punto anterior puede llegar a ser tediosa o infinita. `q` hace ingresar a TEX en `batchmode`, modo en el cual la compilación prosigue ignorando todos los errores hasta el final del archivo, sin enviar mensajes a pantalla y por ende sin que debamos darle infinitos `<Enter>`.

Las opciones (c) y (d) son útiles cuando no entendemos los mensajes de error. Como TEX seguirá compilando haciendo lo mejor posible, al mirar el `dvi` puede que veamos más claramente dónde comenzaron a ir mal las cosas y, por tanto, por qué.

Como dijimos, LATEX indica exactamente dónde encontró el error, de modo que hemos de ponerle atención. Por ejemplo, si tenemos en nuestro documento la línea:

```
... un error inesperado\fotnote{En cualquier punto.}
puede decidir...
```

generaría el mensaje de error:

```
! Undefined control sequence.

l.249 ...un error inesperado\fotnote
                                    {En cualquier punto.}
?
```

En la línea de localización, LATEX ha cortado el texto justo después del comando inexistente. LATEX no sólo indica la línea en la cual detectó el error, sino el punto de ella donde ello ocurrió. (En realidad, hizo lo mismo —cortar la línea para hacer resaltar el problema— en el caso expuesto en la pág. 198, pero ello ocurrió en medio de comandos de bajo nivel, así que no era muy informativo de todos modos.)

Errores más comunes.

Los errores más comunes son:

a) Comando mal escrito.

b) Paréntesis cursivos no apareados.

c) Uso de uno de los caracteres especiales `#`, `$`, `%`, `&`, `_`, `{`, `}`, `~`, `^`, `\` como texto ordinario.

d) Modo matemático abierto de una manera y cerrado de otra, o no cerrado.

e) Ambiente abierto con `\begin...` y cerrado con un `\end...` distinto.

f) Uso de un comando matemático fuera de modo matemático.

g) Ausencia de argumento en un comando que lo espera.

h) Línea en blanco en ambiente matemático.

Algunos mensajes de error.

A continuación, una pequeña lista de errores (de LaTeX y TeX) en orden alfabético, y sus posibles causas.

`*`

Falta `\end{document}`. (Dar `Ctrl-C` o escribir `\end{document}` para salir de la compilación.)

`! \begin{...} ended by \end{...}`

Error e) de la Sec. 5.14.1. El nombre del ambiente en `\end{...}` puede estar mal escrito, sobra un `\begin` o falta un `\end`.

`! Double superscript` (o `subscript`).

Una expresión como `x^2^3` o `x_2_3`. Si se desea obtener x^{2^3} (x_{23}), escribir `{x^2}^3` (`{x_2}_3`).

`! Environment ... undefined.`

`\begin{...}` con un argumento que corresponde a un ambiente no definido.

`! Extra alignment tab has been changed.`

En un `tabular` o `array` sobra un `&`, falta un `\\`, o falta una `c`, `l` ó `r` en el argumento obligatorio.

`! Misplaced alignment tab character &.`

Un `&` aparece fuera de un `tabular` o `array`.

`! Missing $ inserted.`

Errores c), d), f), h) de la Sec. 5.14.1.

`! Missing {` (o `}`) `inserted.`

Paréntesis cursivos no apareados.

`! Missing \begin{document}.`

Falta `\begin{document}` o hay algo incorrecto en el preámbulo.

`! Missing number, treated as zero.`

Falta un número donde LaTeX lo espera: `\hspace{}`, `\vspace cm`, `\setlength{\textwidth}{a}`, etc.

`! Something's wrong -- perhaps a missing \item.`

Posiblemente la primera palabra después de un `\begin{enumerate}` o `\begin{itemize}` no es `\item`.

`! Undefined control sequence.`

Aparece una secuencia `\<palabra>`, donde `<palabra>` no es un comando.

5.14.2. Advertencias.

La estructura de una advertencia de LaTeX es:

`LaTeX warning. <mensaje>.`

Algunos ejemplos:

`Label '...' multiply defined.`

Dos `\label` tienen el mismo argumento.

`Label(s) may have changed. Rerun to get cross-references right.`

Los números impresos por `\ref` y `\pageref` pueden ser incorrectos, pues los valores correspondientes cambiaron respecto al contenido del `aux` generado en la compilación anterior.

`Reference '...' on page ... undefined.`

El argumento de un `\ref` o un `\pageref` no fue definido por un `\label`.

TeX también envía advertencias. Se reconocen porque no comienzan con `TeX warning`. Algunos ejemplos.

`Overfull \hbox ...`

TeX no encontró un buen lugar para cortar una línea, y puso más texto en ella que lo conveniente.

`Overfull \vbox ...`

TeX no encontró un buen lugar para cortar una página, y puso más texto en ella que lo conveniente.

`Underfull \hbox ...`

TeX construyó una línea con muy poco material, de modo que el espacio entre palabras puede ser excesivo.

`Underfull \vbox ...`

TeX construyó una página con muy poco material, de modo que los espacios verticales (entre párrafos) pueden ser excesivos.

Las advertencias de LaTeX siempre deben ser atendidas. Una referencia doblemente definida, o no compilar por segunda vez cuando LaTeX lo sugiere, generará un resultado incorrecto en el `dvi`. Una referencia no definida, por su parte, hace aparecer un signo **??** en el texto final. Todos resultados no deseados, por cierto.

Las advertencias de TeX son menos decisivas. Un overfull o underfull puede redundar en que alguna palabra se salga del margen derecho del texto, que el espaciado entre palabras en una línea sea excesivo, o que el espacio vertical entre párrafos sea demasiado. Los estándares de calidad de TeX son altos, y por eso envía advertencias frecuentemente. Pero generalmente los defectos en el resultado final son imperceptibles a simple vista, o por lo menos no son suficientes para molestarnos realmente. A veces sí, por supuesto, y hay que estar atentos. Siempre conviene revisar el texto y prestar atención a estos detalles, aunque ello sólo tiene sentido al preparar la versión definitiva del documento.

Parte II

Métodos Numéricos.

Capítulo 6

Preliminares.

versión 3.03, 18 de Noviembre del 2003 [1].

6.1. Programas y funciones.

Programa de ortogonalidad en Octave

En esta sección nosotros escribiremos algunos programas simples usando Octave y C++. En nuestro primer ejemplo, llamado `orthog`, probaremos si dos vectores son ortogonales calculando su producto punto. Este simple programa lo bosquejamos a continuación

- Inicializamos los vectores $\vec{a}$ y $\vec{b}$.
- Evaluamos el producto punto como $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$.
- Imprimir el producto punto y establecer si los vectores son ortogonales.

Primero consideremos la versión en Octave del programa que llamaremos `orthog.m`. Las primeras líneas de `orthog` son:

```
% orthog - Programa para probar si un par de vectores es ortogonal.
% Supondremos vectores en 3D.
clear all; % Borra la memoria
```

Las primeras dos líneas son comentarios; si tipeamos `help orthog` desde la línea de comandos, Octave desplegará estas líneas. El comando `clear all` en la tercera línea borra la memoria. Las próximas líneas del programa

```
%* Inicializa los vectores a y b
a= input('Entre el primer vector: ');
b= input('Entre el segundo vector: ');
```

Los vectores entran usando el comando `input` en estas líneas. Los comentarios que comienzan con `%*` son aquellos que corresponden al bosquejo del programa que hicimos. En las líneas siguientes se evalúa el producto punto.

[1] Este capítulo está basado en el primer capítulo del libro: *Numerical Methods for Physics, second edition* de Alejandro L. Garcia, editorial Prentice Hall

```
%* Evalua el producto punto como la suma sobre el producto de los elementos
adotb=0;
for i=1:3
  adotb=adotb+a(i)*b(i);
end
```

El ciclo `for`, usando el índice `i`, recorre las componentes de los vectores. Una manera hábil de hacer lo mismo podría ser usar

```
%* Evalua el producto punto como el producto de dos vectores
adotb=a*b' ;
```

En este caso, hemos usado la multiplicación de matrices de Octave para calcular el producto punto como el producto vectorial del vector fila `a` y el columna `b'` (donde hemos usado el operador Hermítico conjugado `'`). Las últimas líneas del programa

```
%* Imprime el producto punto y si los vectores son ortogonales
if(adotb==0)
  disp('Los vectores son ortogonales');
else
  disp('Los vectores no son ortogonales');
  printf('Producto punto = %g \n', adotb);
end
```

De acuerdo al valor de `adotb` el programa despliega una de las dos posibles respuestas. A continuación la salida al ejecutar el `help` del programa

```
octave> help orthog
orthog is the file: /home/jrogan/orthog.m

 orthog - Programa para probar si un par de vectores es ortogonal.
 Supondremos vectores en 3D.


Additional help for builtin functions, operators, and variables
is available in the on-line version of the manual.  Use the command
‘help -i <topic>’ to search the manual index.

Help and information about Octave is also available on the WWW
at http://www.che.wisc.edu/octave/octave.html and via the
help-octave@bevo.che.wisc.edu mailing list.
```

Ahora ejecutamos el programa con diferentes vectores.

```
octave> orthog
Entre el primer vector: [1 1 1]
Entre el segundo vector: [1 -2 1]
Los vectores son ortogonales
```

```
octave> orthog
Entre el primer vector: [1 1 1]
Entre el segundo vector: [2 2 2]
Los vectores no son ortogonales
Producto punto = 6
```

Programa de ortogonalidad en C++.

Ahora consideremos la versión en C++ del programa `orthog`, el cual prueba si dos vectores son ortogonales mediante el cálculo de su producto punto. Primero escribiremos una clase muy básica de vectores en tres dimensiones. Las declaraciones están en `vector3d.h`

```
#ifndef _vector_3d_h
#define _vector_3d_h
//
// Clase basica de vectores 3d
//
#include <iostream>

class Vector{
private:
  double c_x;
  double c_y;
  double c_z;
public:
  Vector():c_x(0),c_y(0),c_z(0) {} ;
  Vector(double x, double y, double z):c_x(x),c_y(y),c_z(z) {} ;
  ~Vector() {} ;
  double x() const {return c_x;};
  double y() const {return c_y;};
  double z() const {return c_z;};
};

double operator * (const Vector &, const Vector &) ;
istream & operator >> (istream &, Vector &) ;

#endif
```

La pequeña implementación necesaria para esta clase la escribimos en `vector3d.cc` el cual es listado a continuación

```
#include "vector3d.h"

double operator * (const Vector & v1, const Vector &v2) {
  return v1.x()*v2.x()+v1.y()*v2.y()+ v1.z()*v2.z() ;
```

```
}
istream & operator >> (istream & is, Vector & v) {
  double x,y,z ;
  is >> x >> y>>z ;
  v= Vector(x,y,z) ;
  return is;
}
```

Ahora estamos en condiciones de escribir el programa propiamente tal. Las primeras líneas son

```
//  orthog - Programa para probar si un par de vectores es ortogonal.
// Supondremos vectores en 3D.
#include ‘‘vector3d.h’’
using namespace std;
```

Las primeras líneas son comentarios que nos recuerdan lo que el programa hace. La línea siguiente incluye las definiciones de nuestra recién creada clase. Luego incluímos una línea dice que vamos a usar el `namespace std`. A continuación comienza el programa

```
//  orthog - Programa para probar si un par de vectores es ortogonal.
//  Usaremos vectores en 3D.

#include "vector3d.h"

using namespace std;

int main()
{
  Vector a, b;
  cout << "Ingrese el primer vector : ";
  cin >> a ;
  cout << "Ingrese el segundo vector : ";
  cin >> b ;
  if(a*b==0) {
    cout <<"Son ortogonales" << endl;
  } else {
    cout << "No son ortogonales"<< endl ;
  }
  return 0;
}
```

Declaramos dos objetos tipo `Vector`, `a` y `b` para almacenar los vectores que entran. La instrucción de salida despliega sobre la pantalla:

```
Ingrese el primer vector :
```

La instrucción de entrada lee el `Vector` en `a` y luego, de la misma manera, el `Vector b`. Las próximas líneas hacen el trabajo

```
if(a*b==0) {
  cout <<"Son ortogonales" << endl;
} else {
  cout << "No son ortogonales"<< endl ;
}
```

y despliega el resultados. Aquí los comandos de compilación y una salida típica del programa.

```
jrogan@pucon:~$ g++ -Wall -o orthog orthog.cc vector3d.cc
jrogan@pucon:~$ orthog
Ingrese el primer vector : 1 0 0
Ingrese el segundo vector : 0 1 0
Son ortogonales
```

Programa de interpolación en Octave.

Es bien sabido que dados tres pares (x, y), se puede encontrar una cuadrática que pasa por los puntos deseados. Hay varias maneras de encontrar el polinomio y varias maneras de escribirlo. La forma de Lagrange del polinomio es

$$p(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}y_1 + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}y_2 + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}y_3 \ , \qquad (6.1)$$

donde (x_1, y_1), (x_2, y_2), (x_3, y_3), son los tres puntos por los que queremos pasar. Comúnmente tales polinomios son usados para interpolar entre los puntos dados. A continuación el bosquejo de un programa simple de interpolación, que llamaremos `interp`

- Inicializa los puntos (x_1, y_1), (x_2, y_2) y (x_3, y_3) para ser ajustados por el polinomio.
- Establece el intervalo de la interpolación (desde $x_{\text{mín}}$ hasta $x_{\text{máx}}$)
- Encuentra y^* para los valores deseados de x^*, usando la función `intrpf`.
- Finalmente, podemos graficar la curva dada por (x^*, y^*), y marcar los puntos originales para comparar.

Las primeras líneas del programa

```
% interp - Programa para interpolar datos usando
% el polinomio de Lagrange cuadratico para tres puntos dados.
clear all;
%* Inicializa los puntos a ser ajustados con una cuadratica
disp('Entre los puntos como pares x,y (e.g., [1 2])');
for i=1:3
  temp =input('Ingrese el punto: ');
  x(i)=temp(1);
```

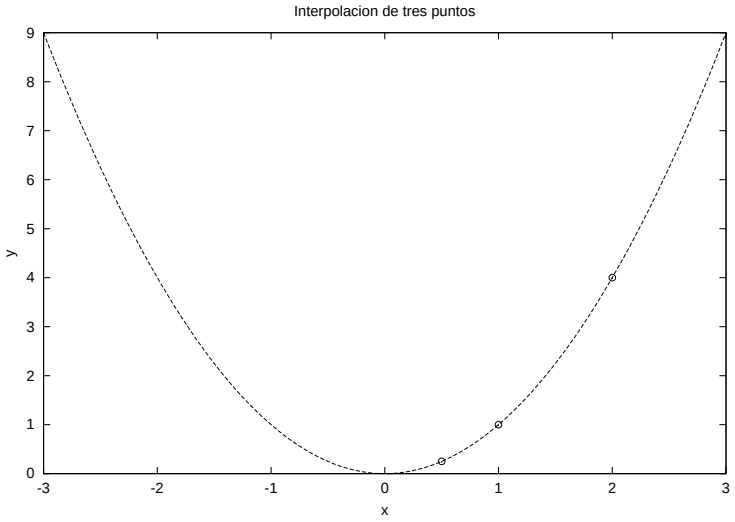


Figura 6.1: Salida gráfica del programa `interp`.

```
  y(i)=temp(2) ;
end
%* Establece el intervalo de interpolacion (desde x_min a x_max)
xr = input ('Ingrese el intervalo de valores de x como [x_min x_max]: ');
```

Aquí el programa lee los tres pares (x, y) y el intervalo de valores entre los cuales será interpolado.

Los valores interpolados $y^* = p(x^*)$ son calculados por la función `intrpf` desde $x^* = x_{\text{mín}}$ a $x^* = x_{\text{máx}}$. Estos valores de y^* (`yi`) son calculados en el ciclo.

```
%* Encontrar yi para los valores deseados de interpolacion xi
% usando la funcion intrpf
nplot= 100; % Numero de puntos para la curva interpolada
for i=1:nplot
  xi(i) = xr(1)+(xr(2)-xr(1))*(i-1)/(nplot-1);
  yi(i) = intrpf(xi(i), x, y); % Usando intrpf para interpolar
end
```

Finalmente, los resultados son graficados usando las funciones gráficas de Octave.

```
%* Grafica la curva dada por (xi,yi) y marca los puntos originales
xlabel('x');
ylabel('y');
title('Interpolacion de tres puntos');
gset nokey;
plot(x,y, '*',xi,yi,'-');
```

Los puntos de la interpolación (x^*, y^*) son graficados con línea segmentada y los datos originales con círculos, ver figura (6.1). El trabajo real es hecho por la función `intrpf`. Un bosquejo de lo que queremos que haga esta función a continuación.

- Entrada: $\vec{x} = [x_1\ x_2\ x_3]$, $\vec{y} = [y_1\ y_2\ y_3]$, y x^*.
- Salida: y^*.
- Cálculo de $y^* = p(x^*)$ usando el polinomio de Lagrange (6.1).

Las funciones en Octave están implementadas como en la mayoría de los lenguajes, excepto que aquí cada función tiene que ir en un archivo separado. El nombre del archivo debe coincidir con el nombre de la función (la función `intrpf` está en el archivo `intrpf.m`).

```
function yi=intrpf(xi,x,y)
% Funcion para interpolar entre puntos
% usando polinomio de Lagrange (cuadratico)
% Entradas
%   x   Vector de las coordenadas x de los puntos dados (3 valores)
%   y   Vector de las coordenadas y de los puntos dados (3 valores)
% Salida
%   yi  El polinomio de interpolacion evaluado en xi
```

La función `intrpf` tiene tres argumentos de entrada y uno de salida. El resto de la función es directa, sólo evalúa el polinomio definido en (6.1). El cuerpo de la función a continuación

```
yi = (xi-x(2))*(xi-x(3))/((x(1)-x(2))*(x(1)-x(3)))*y(1) ...
   + (xi-x(1))*(xi-x(3))/((x(2)-x(1))*(x(2)-x(3)))*y(2) ...
   + (xi-x(1))*(xi-x(2))/((x(3)-x(1))*(x(3)-x(2)))*y(3);
return;
```

Programa de interpolación en C++.

Las primeras líneas de la versión en C++ del programa `interp` son

```
// interp - Programa para interpolar datos usando
// el polinomio de Lagrange cuadratico para tres puntos dados.
#include "NumMeth.h"

double intrpf(double xi, double x[], double y[]);
int main()
{
```

Comentarios más una instrucción para incluir el archivo de encabezamiento `NumMeth.h`, listado a continuación

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>

using namespace std;
```

La declaración `double intrpf(..)` afirma que el programa pretende llamar una función `intrpf`, la cual tiene tres argumentos de tipo `double` y devuelve un `double`. Las próximas líneas del programa

```
//* Inicializa los puntos a ser ajustados con una cuadratica
double x[3+1], y[3+1] ;
cout << "Entre los puntos como pares x,y (e.g., [1 2])" << endl ;
for(int i=1; i<=3; i++) {
  cout << "x["<<i<<"] = ";
  cin>>  x[i];
  cout << "y["<<i<<"] = ";
  cin >> y[i];
}

//* Establece el intervalo de interpolacion (desde x_min a x_max)
double xmin, xmax;
cout <<"Entre el valor minimo de x: "; cin >> xmin ;
cout <<"Entre el valor maximo de x: "; cin >> xmax ;
```

El programa pregunta por los puntos para ajustar el polinomio de Lagrange (6.1) y por el intervalo de interpolación. Lo siguiente, los arreglos `xi` y `yi` son declarados:

```
//* Encontrar yi para los valores deseados de interpolacion xi
// usando la funcion intrpf
int nplot= 100; // Numero de puntos para la curva interpolada
double * xi = new double[nplot+1] ; // Reserva memoria para
double * yi = new double[nplot+1] ; // estos arreglos.
```

Estas líneas también podrían reemplazarse por

```
const int nplot =100; // Numero de puntos para la curva interpolada
double xi[nplot+1], yi[nplot+1] ;
```

En el primer caso hay asignamiento dinámico de memoria, `nplot` podría ser una entrada del programa. En el segundo caso `nplot` debe ser constante y para modificar el número de puntos debemos recompilar el programa, asignación estática.

Los valores interpolados son calculados en un `for`

```
for(int i=1; i<=nplot;i++) {
 xi[i] = xmin+(xmax-xmin)*double(i-1)/double(nplot-1);
 yi[i] = intrpf(xi[i], x, y); // Usando intrpf para interpolar
}
```

Notemos que `xi[1]`$=x_{\text{mín}}$, `xi[nplot]`$=x_{\text{máx}}$, con valores equiespaciados entre ellos. Los valores de `yi` ($y^* = p(x^*)$) son evaluados usando la ecuación (6.1) en la función `intrpf`. La salida del programa

```
//* Imprime las variables para graficar: x, y, xi, yi
ofstream xOut("x.txt"), yOut("y.txt"), xiOut("xi.txt"), yiOut("yi.txt");
for(int i =1; i <=3; i++) {
  xOut << x[i] << endl;
  yOut << y[i] << endl;
}
for(int i =1; i <=nplot; i++) {
  xiOut << xi[i] << endl;
  yiOut << yi[i] << endl;
}
XOut.close() ; yOut.close() ; xiOut.close(); yiOut.close() ;
```

Estos cuatro archivos de datos (`x.txt`, `y.txt`, etc.) son creados.

Desgraciadamente, C++ carece de una biblioteca gráfica estándar, así que necesitamos una aplicación gráfica adicional para graficar la salida. También podemos usar un pequeño *script* en Octave:

```
#!/usr/bin/octave
load x.txt; load y.txt; load xi.txt; load yi.txt;
%* Grafica la curva dada por (xi,yi) y marca los puntos originales
xlabel('x');
ylabel('y');
title('Interpolacion de tres puntos');
gset nokey;
plot(x,y, '*',xi,yi,'-');
pause
```

Al cual, incluso, podemos llamar desde el mismo programa mediante

```
system( "grafica.m" ) ;
```

La última línea del programa

```
delete [] xi, yi ; // Libera la memoria pedida con "new"
return 0;
}
```

Esta línea no es absolutamente necesaria, por que al salir el programa liberará la memoria de todas maneras. Sin embargo, se considera como buen estilo de programación, limpiar uno la memoria que requirió durante la ejecución del programa.

La función `intrpf`, la cual evalúa el polinomio de Lagrange, comienza por las siguientes líneas

```
double intrpf( double xi, double x[], double y[])
{
  //  Funcion para interpolar entre puntos
  // usando polinomio de Lagrange (cuadratico)
  // Entradas
```

```
//   x   Vector de las coordenadas x de los puntos dados (3 valores)
//   y   Vector de las coordenadas y de los puntos dados (3 valores)
// Salida
//   yi  El polinomio de interpolacion evaluado en xi
```

Especifica los argumentos de llamada y lo que devuelve. Todas las variables dentro de la función son *locales*. El C++ pasa las variables por valor, por defecto, la función recibe una copia que se destruye cuando termina la función, si se desea pasar una variable `double a` por referencia debemos anteponerle el signo &, es decir, pasarla como `double &a`. De esta manera la función puede modificar el valor que tenía la variable en el programa principal.

El resto de la función

```
//* Calcula yi=p(xi) usando Polinomio de Lagrange

double yi = (xi-x[2])*(xi-x[3])/((x[1]-x[2])*(x[1]-x[3]))*y[1]
  + (xi-x[1])*(xi-x[3])/((x[2]-x[1])*(x[2]-x[3]))*y[2]
  + (xi-x[1])*(xi-x[2])/((x[3]-x[1])*(x[3]-x[2]))*y[3];
return yi ;
```

Estas líneas evalúan el polinomio. Inicialmente pondremos esta función en el mismo archivo, luego la podemos separar en otro archivo y escribir un `Makefile` que genere el ejecutable.

6.2. Errores numéricos.

6.2.1. Errores de escala.

Un computador almacena números de punto flotante usando sólo una pequeña cantidad de memoria. Típicamente, a una variable de precisión simple (un `float` en C++) se le asignan 4 bytes (32 bits) para la representación del número, mientras que una variable de doble precisión (`double` en C++, por defecto en Octave) usa 8 bytes. Un número de punto flotante es representado por su mantisa y su exponente (por ejemplo, para 6.625×10^{-27} la mantisa decimal es 6.625 y el exponente es -27). El formato IEEE para doble precisión usa 53 bits para almacenar la mantisa (incluyendo un bit para el signo) y lo que resta, 11 bit para el exponente. La manera exacta en que el computador maneja la representación de los números no es tan importante como saber el intervalo máximo de valores y el número de cifras significativas.

El intervalo máximo es el límite sobre la magnitud de los números de punto flotante impuesta por el número de bit usados para el exponente. Para precisión simple un valor típico es $2^{\pm 127} \approx 10^{\pm 38}$; para precisión doble es típicamente $2^{\pm 1024} \approx 10^{\pm 308}$. Exceder el intervalo de la precisión simple no es difícil. Consideremos, por ejemplo, la evaluación del radio de Bohr en unidades SI,

$$a_0 = \frac{4\pi\varepsilon_0\hbar^2}{m_e e^2} \approx 5.3 \times 10^{-11} \text{ [m]} . \tag{6.2}$$

Mientras sus valores caen dentro del intervalo de un número de precisión simple, el intervalo es excedido en el cálculo del numerador ($4\pi\varepsilon_0\hbar^2 \approx 1.24 \times 10^{-78}$ [kg C^2 m]) y del denominador

($m_e e^2 \approx 2.34 \times 10^{-68}$ [kg C^2]). La mejor solución para lidiar con este tipo de dificultades de intervalo es trabajar en un conjunto de unidades naturales al problema (*e.g.* para problemas atómicos se trabaja en las distancias en Ångstroms y la carga en unidades de la carga del electrón).

Algunas veces los problemas de intervalo no son causados por la elección de las unidades, sino porque los números en el problema son inherentemente grandes. Consideremos un importante ejemplo, la función factorial. Usando la definición

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \ , \tag{6.3}$$

es fácil evaluar $n!$ en C++ como

```
double nFactorial=1;
for(int i=1; i <=n; i++) nFactorial *=i ;
```

donde `n` es un número dado.

En Octave, usando el operador dos puntos este cálculo puede ser realizado como

```
nFactorial = prod(1:n);
```

donde `prod(x)` es el producto de los elementos del vector `x` y `1:n=[1,2...,n]`. Infortunadamente, debido a problemas de intervalo, no podemos calcular $n!$ para $n > 170$ usando estos métodos directos de evaluación (6.3).

Una solución común para trabajar con números grandes es usar su logaritmo. Para el factorial

$$\log(n!) = \log(n) + \log(n-1) + \ldots + \log(3) + \log(2) + \log(1) \ . \tag{6.4}$$

En Octave, esto puede ser evaluado como

```
log_nFactorial = sum( log(1:n) ) ;
```

donde `sum(x)` es la suma de los elementos del vector `x`. Sin embargo, este esquema es computacionalmente pesado si n es grande. Una mejor estrategia es combinar el uso de logaritmos con la fórmula de Stirling[2]

$$n! = \sqrt{2n\pi} n^n e^{-n} \left(1 + \frac{1}{12n} + \frac{1}{288n^2} + \cdots \right) \tag{6.5}$$

o

$$\log(n!) = \frac{1}{2}\log(2n\pi) + n\log(n) - n + \log\left(1 + \frac{1}{12n} + \frac{1}{288n^2} + \cdots \right) \ . \tag{6.6}$$

Esta aproximación puede ser usada cuando n es grande ($n > 30$), de otra manera es preferible la definición original.

Finalmente, si el valor de $n!$ necesita ser impreso, podemos expresarlo como

$$n! = (\text{mantisa}) \times 10^{(\text{exponente})} \ , \tag{6.7}$$

donde el exponente es la parte entera de $\log_{10}(n!)$, y la mantisa es 10^a donde a es la parte fraccionaria de $\log_{10}(n!)$. Recordemos que la conversión entre logaritmo natural y logaritmo en base 10 es $\log_{10}(x) = \log_{10}(e)\log(x)$.

[2] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions* (New York: Dover 1972).

6.2.2. Errores de redondeo.

Supongamos que deseamos calcular numéricamente $f'(x)$, la derivada de una función conocida $f(x)$. En cálculo se aprendió que la fórmula para la derivada es

$$f'(x) = \frac{f(x+h) - f(x)}{h} \ , \tag{6.8}$$

en el límite en que $h \to 0$. ¿Qué sucede si evaluamos el lado derecho de esta expresión, poniendo $h = 0$?. Como el computador no entiende que la expresión es válida sólo como un límite, la división por cero tiene varias posibles salidas. El computador puede asignar el valor, `Inf`, el cual es un número de punto flotante especial reservado para representar el infinito. Ya que el numerador es también cero el computador podría evaluar el cociente siendo indefinido (Not-a-Number), `NaN`, otro valor reservado. O el cálculo podría parar y arrojar un mensaje de error.

Claramente, poniendo $h = 0$ para evaluar (6.8) no nos dará nada útil, pero ¿si le ponemos a h un valor muy pequeño, digamos $h = 10^{-300}$, usamos doble precisión? La respuesta aún será incorrecta debido a la segunda limitación sobre la representación de números con punto flotante: el número de dígitos en la mantisa. Para precisión simple, el número de dígitos significantes es típicamente 6 o 7 dígitos decimales; para doble precisión es sobre 16 dígitos. Así, en doble precisión, la operación $3 + 10^{-20}$ retorna una respuesta 3 por el redondeo; usando $h = 10^{-300}$ en la ecuación (6.8) casi con seguridad regresará 0 cuando evaluemos el numerador.

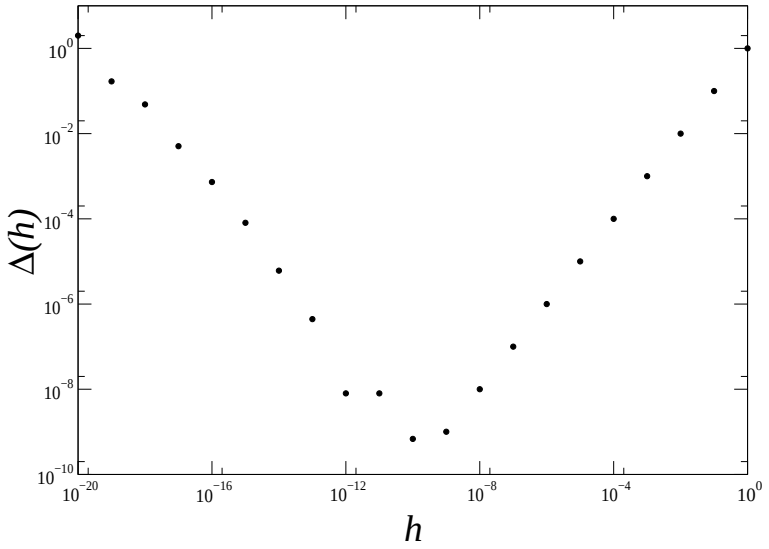


Figura 6.2: Error absoluto $\Delta(h)$, ecuación (6.9), versus h para $f(x) = x^2$ y $x = 1$.

La figura 6.2 ilustra la magnitud del error de redondeo en un cálculo típico de derivada. Definimos el error absoluto

$$\Delta(h) = \left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \ . \tag{6.9}$$

Notemos que $\Delta(h)$ decrece cuando h se hace más pequeño, lo cual es esperado, dado que la ecuación (6.8) es exacta cuando $h \to 0$. Por debajo de $h = 10^{-10}$, el error comienza a incrementarse debido a efectos de redondeo. En valores menores de h el error es tan grande que la respuesta carece de sentido. Volveremos en el próximo capítulo a la pregunta de cómo mejorar el cálculo de la derivada numéricamente.

Para testear la tolerancia del redondeo, definimos ε_Γ como el más pequeño número que, cuando es sumado a uno, regresa un valor distinto de 1. En Octave, la función integrada `eps` devuelve $\varepsilon_\Gamma \approx 2.22 \times 10^{-16}$. En C++, el archivo de encabezamiento `<cfloat>` define `DBL_EPSILON` (2.2204460492503131e-16) como ε_Γ para doble precisión.

Debido a los errores de redondeo la mayoría de los cálculos científicos usan doble precisión. Las desventajas de la doble precisión son que requieren más memoria y que algunas veces (no siempre) es más costosa computacionalmente. Los procesadores modernos están construidos para trabajar en doble precisión, tanto que puede ser más lento trabajar en precisión simple. Usar doble precisión algunas veces sólo desplaza las dificultades de redondeo. Por ejemplo, el cálculo de la inversa de una matriz trabaja bien en simple precisión para matrices pequeñas de 50×50 elementos, pero falla por errores de redondeo para matrices más grandes. La doble precisión nos permite trabajar con matrices de 100×100, pero si necesitamos resolver sistemas aún más grandes debemos usar un algoritmo diferente. La mejor manera de trabajar es usar algoritmos robustos contra el error de redondeo.

Capítulo 7

Ecuaciones diferenciales ordinarias: Métodos básicos.

versión final 2.2-031903[1].

En este capítulo resolveremos uno de los primeros problemas considerados por un estudiante de física: el vuelo de un proyectil y, en particular, el de una pelota de *baseball*. Sin la resistencia del aire el problema es fácil de resolver. Sin embargo, incluyendo un arrastre realista, nosotros necesitamos calcular la solución numéricamente. Para analizar este problema definiremos primero la diferenciación numérica. De hecho antes de aprender Física uno aprende cálculo así que no debemos sorprendernos si este es nuestro punto de partida. En la segunda mitad del capítulo nos ocuparemos de otro viejo conocido, el péndulo simple, pero sin la aproximación a ángulos pequeños. Los problemas oscilatorios, tales como el péndulo, nos revelarán una falla fatal en algunos de los métodos numéricos de resolver ecuaciones diferenciales ordinarias.

7.1. Movimiento de un proyectil.

7.1.1. Ecuaciones básicas.

Consideremos el simple movimiento de un proyectil, digamos un pelota de *baseball*. Para describir el movimiento, nosotros debemos calcular el vector posición $\vec{r}(t)$ y el vector velocidad $\vec{v}(t)$ del proyectil. Las ecuaciones básicas de movimiento son

$$\frac{d\vec{v}}{dt} = \frac{1}{m}\vec{F}_a(\vec{v}) - g\hat{y} \ , \quad \frac{d\vec{r}}{dt} = \vec{v} \ , \tag{7.1}$$

donde m es la masa del proyectil. La fuerza debido a la resistencia del aire es $\vec{F}_a(\vec{v})$, la aceleración gravitacional es g, e $\hat{y}$ es un vector unitario en la dirección y. El movimiento es bidimensional, tal que podemos ignorar la componente z y trabajar en el plano xy.

La resistencia del aire se incrementa con la velocidad del objeto, y la forma precisa para $\vec{F}_a(\vec{v})$ depende del flujo alrededor del proyectil. Comúnmente, esta fuerza es aproximada por

$$\vec{F}_a(\vec{v}) = -\frac{1}{2}C_d\rho A \,|\, v \,|\, \vec{v} \ , \tag{7.2}$$

[1]Este capítulo está basado en el segundo capítulo del libro: *Numerical Methods for Physics, second edition* de Alejandro L. Garcia, editorial Prentice Hall

donde C_d es el coeficiente de arrastre, ρ es la densidad del aire, y A es el área de la sección transversal del proyectil. El coeficiente de arrastre es un parámetro adimensional que depende de la geometría del proyectil, entre más aerodinámico el objeto, el coeficiente es menor.

Para una esfera suave de radio R moviéndose lentamente a través del fluido, el coeficiente de arrastre es dado por la Ley de Stokes,

$$C_d = \frac{12\nu}{Rv} = \frac{24}{\mathrm{Re}} \ , \tag{7.3}$$

donde ν es la viscosidad del fluido ($\nu \approx 1.5 \times 10^{-5}$ [m^2/s] para el aire) y $\mathrm{Re} = 2Rv/\nu$ es el adimensional *número de Reynolds*. Para un objeto del tamaño de una pelota de *baseball* moviéndose a través del aire, la ley de Stokes es válida sólo si la velocidad es menor que $0.2\tilde{[}$mm/s] (Re$\approx$ 1).

A velocidades altas (sobre 20 [cm/s], Re$>$ 10^3), la estela detrás de la esfera desarrolla vórtices y el coeficiente de arrastre es aproximadamente constante ($C_d \approx 0.5$) para un amplio intervalo de velocidades. Cuando el número de Reynolds excede un valor crítico, el flujo en la estela llega a ser turbulento y el coeficiente de arrastre cae dramáticamente. Esta reducción ocurre porque la turbulencia rompe la región de bajas presiones en la estela detrás de la esfera[2]. Para una esfera suave este número crítico de Reynolds es aproximadamente 3×10^5. Para una pelota de *baseball*, el coeficiente de arrastre es usualmente más pequeño que el de una esfera suave, porque las costuras rompen el flujo laminar precipitando el inicio de la turbulencia. Nosotros podemos tomar $C_d = 0.35$ como un valor promedio para el intervalo de velocidades típicas de una pelota de *baseball*.

Notemos que la fuerza de arrastre, ecuación (7.2), varía como el cuadrado de la magnitud de la velocidad ($F_a \propto v^2$) y, por supuesto, actúa en la dirección opuesta a la velocidad. La masa y el diámetro de una pelota de *baseball* son 0.145 [kg] y 7.4 [cm]. Para una pelota de *baseball*, el arrastre y la fuerza gravitacional son iguales en magnitud cuando $v \approx 40$ [m/s].

Nosotros sabemos cómo resolver las ecuaciones de movimiento si la resistencia del aire es despreciable. La trayectoria es

$$\vec{r}(t) = \vec{r}_1 + \vec{v}_1 t - \frac{1}{2} g t^2 \hat{y} \ , \tag{7.4}$$

donde $\vec{r}_1 \equiv \vec{r}(0)$ y $\vec{v}_1 \equiv \vec{v}(0)$ son la posición y la velocidad inicial. Si el proyectil parte del origen y la velocidad inicial forma un ángulo θ con la horizontal, entonces

$$x_{\mathrm{máx}} = \frac{2v_1^2}{g} \operatorname{sen}\theta \cos\theta \ , \quad y_{\mathrm{máx}} = \frac{v_1^2}{2g} \operatorname{sen}^2\theta \ , \tag{7.5}$$

son el alcance horizontal y la altura máxima. El tiempo de vuelo es

$$t_{fl} = \frac{2v_1}{g} \operatorname{sen}\theta \ . \tag{7.6}$$

Nuevamente estas expresiones son válidas sólo cuando no hay resistencia con el aire. Es fácil demostrar que el máximo alcance horizontal se obtiene cuando la velocidad forma un ángulo de 45° con la horizontal. Deseamos mantener esta información en mente cuando construyamos nuestra simulación. Si se sabe la solución exacta para un caso especial, se debe comparar constantemente que el programa trabaje bien para este caso.

[2] D.J. Tritton, Physical Fluid Dynamics, 2d ed. (Oxford: Clarendon Press, 1988).

7.1.2. Derivada avanzada.

Para resolver las ecuaciones de movimiento (7.1) necesitamos un método numérico para evaluar la primera derivada. La definición formal de la derivada es

$$f'(t) \equiv \lim_{\tau \to 0} \frac{f(t+\tau) - f(t)}{\tau} \ , \tag{7.7}$$

donde τ es el incremento temporal o paso en el tiempo. Como ya vimos en el capítulo pasado esta ecuación debe ser tratada con cuidado. La figura 6.2 ilustra que el uso de un valor extremadamente pequeño para τ causa un gran error en el cálculo de $(f(t+\tau) - f(t))/\tau$. Específicamente, los errores de redondeo ocurren en el cálculo de $t+\tau$, en la evaluación de la función f y en la sustracción del numerador. Dado que τ no puede ser elegido arbitrariamente pequeño, nosotros necesitamos estimar la diferencia entre $f'(t)$ y $(f(t+\tau) - f(t))/\tau$ para un τ finito.

Para encontrar esta diferencia usaremos una expansión de Taylor. Como físicos usualmente vemos las series de Taylor expresadas como

$$f(t+\tau) = f(t) + \tau f'(t) + \frac{\tau^2}{2} f''(t) + \ldots \tag{7.8}$$

donde el símbolo $(\ldots)$ significa términos de más alto orden que son usualmente despreciados. Una alternativa, forma equivalente de la serie de Taylor usada en análisis numérico es

$$f(t+\tau) = f(t) + \tau f'(t) + \frac{\tau^2}{2} f''(\zeta) \ . \tag{7.9}$$

donde ζ es un valor entre t y $t+\tau$. No hemos botado ningún término, esta expansión tiene un número finito de términos. El teorema de Taylor garantiza que existe *algún* valor ζ para el cual (7.9) es cierto, pero no sabemos cuál valor es este.

La ecuación previa puede ser rescrita

$$f'(t) = \frac{f(t+\tau) - f(t)}{\tau} - \frac{1}{2}\tau f''(\zeta) \ , \tag{7.10}$$

donde $t \leq \zeta \leq t+\tau$. Esta ecuación es conocida como la fórmula de la *derivada derecha* o *derivada adelantada*. El último término de la mano derecha es el *error de truncamiento*; este error es introducido por cortar la serie de Taylor.

En otras palabras, si mantenemos el último término en (7.10), nuestra expresión para $f'(t)$ es exacta. Pero no podemos evaluar este término porque no conocemos ζ, todo lo que conocemos es que ζ yace en algún lugar entre t y $t+\tau$. Así despreciamos el término $f''(\zeta)$ (truncamos) y decimos que el error que cometemos por despreciar este término es el error de truncamiento. No hay que confundir éste con el error de redondeo discutido anteriormente. El error de redondeo depende del *hardware*, el error de truncamiento depende de la aproximación usada en el algoritmo. Algunas veces veremos la ecuación (7.10) escrita como

$$f'(t) = \frac{f(t+\tau) - f(t)}{\tau} + \mathcal{O}(\tau) \tag{7.11}$$

donde el error de truncamiento es ahora especificado por su orden en τ, en este caso el error de truncamiento es lineal en τ. En la figura 6.2 la fuente de error predominante en estimar $f'(x)$ como $[f(x+h) - f(x)]/h$ es el error de redondeo cuando $h < 10^{-10}$ y es el error de truncamiento cuando $h > 10^{-10}$.

7.1.3. Método de Euler.

Las ecuaciones de movimiento que nosotros deseamos resolver numéricamente pueden ser escritas como:

$$\frac{d\vec{v}}{dt} = \vec{a}(\vec{r},\vec{v}) \ , \quad \frac{d\vec{r}}{dt} = \vec{v} \ , \tag{7.12}$$

donde $\vec{a}$ es la aceleración. Notemos que esta es la forma más general de las ecuaciones. En el movimiento de proyectiles la aceleración es sólo función de $\vec{v}$ (debido al arrastre), en otros problemas (e.g., órbitas de cometas) la aceleración dependerá de la posición.

Usando la derivada adelantada (7.11), nuestras ecuaciones de movimiento son

$$\frac{\vec{v}(t+\tau)-\vec{v}(t)}{\tau} + \mathcal{O}(\tau) = \vec{a}(\vec{r}(t),\vec{v}(t)) \ , \tag{7.13}$$

$$\frac{\vec{r}(t+\tau)-\vec{r}(t)}{\tau} + \mathcal{O}(\tau) = \vec{v}(t) \ , \tag{7.14}$$

o bien

$$\vec{v}(t+\tau) = \vec{v}(t) + \tau\vec{a}(\vec{r}(t),\vec{v}(t)) + \mathcal{O}(\tau^2) \ , \tag{7.15}$$

$$\vec{r}(t+\tau) = \vec{r}(t) + \tau\vec{v}(t) + \mathcal{O}(\tau^2) \ . \tag{7.16}$$

notemos que $\tau\mathcal{O}(\tau) = \mathcal{O}(\tau^2)$. Este esquema numérico es llamado el *método de Euler*. Antes de discutir los méritos relativos de este acercamiento, veamos cómo sería usado en la práctica.

Primero, introducimos la notación

$$f_n = f(t_n) \ , \quad t_n = (n-1)\tau \ , \quad n = 1,2,\ldots \tag{7.17}$$

tal que $f_1 = f(t=0)$. Nuestras ecuaciones para el método de Euler (despreciando el término del error) ahora toma la forma

$$\vec{v}_{n+1} = \vec{v}_n + \tau\vec{a}_n \ , \tag{7.18}$$

$$\vec{r}_{n+1} = \vec{r}_n + \tau\vec{v}_n \ , \tag{7.19}$$

donde $\vec{a}_n = \vec{a}(\vec{r}_n,\vec{v}_n)$. El cálculo de la trayectoria podría proceder así:

1. Especifique las condiciones iniciales, $\vec{r}_1$ y $\vec{v}_1$.
2. Elija un paso de tiempo τ.
3. Calcule la aceleración dados los actuales $\vec{r}$ y $\vec{v}$.
4. Use las ecuaciones (7.18) y (7.19) para calcular los nuevos $\vec{r}$ y $\vec{v}$.
5. Vaya al paso 3 hasta que suficientes puntos de trayectoria hayan sido calculados.

El método calcula un conjunto de valores para $\vec{r}_n$ y $\vec{v}_n$ que nos da la trayectoria, al menos en un conjunto discreto de valores. La figura 7.1 ilustra el cálculo de la trayectoria para un único paso de tiempo.

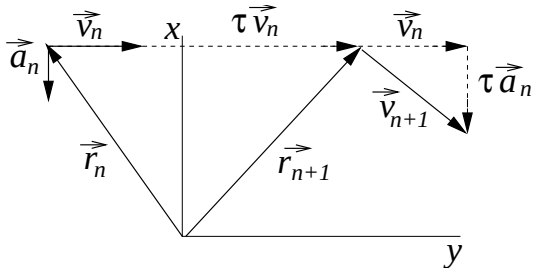


Figura 7.1: Trayectoria de una partícula después de un único paso de tiempo con el método de Euler. Sólo para efectos ilustrativos τ es grande.

7.1.4. Métodos de Euler-Cromer y de Punto Medio.

Una simple (y por ahora injustificada) modificación del método de Euler es usar las siguientes ecuaciones:

$$\vec{v}_{n+1} = \vec{v}_n + \tau\vec{a}_n \ , \tag{7.20}$$
$$\vec{r}_{n+1} = \vec{r}_n + \tau\vec{v}_{n+1} \ . \tag{7.21}$$

Notemos el cambio sutil: La velocidad actualizada es usada en la segunda ecuación. Esta fórmula es llamada *método de Euler-Cromer*[3]. El error de truncamiento es aún del orden de $\mathcal{O}(\tau^2)$, no parece que hemos ganado mucho. Veremos que esta forma es marcadamente superior al método de Euler en algunos casos.

En el *método del punto medio* usamos

$$\vec{v}_{n+1} = \vec{v}_n + \tau\vec{a}_n \ , \tag{7.22}$$
$$\vec{r}_{n+1} = \vec{r}_n + \tau\frac{\vec{v}_{n+1} + \vec{v}_n}{2} \ . \tag{7.23}$$

Notemos que hemos promediado las dos velocidades. Usando la ecuación para la velocidad en la ecuación de la posición, vemos que

$$\vec{r}_{n+1} = \vec{r}_n + \tau\vec{v}_n + \frac{1}{2}\vec{a}_n\tau^2 \ , \tag{7.24}$$

lo cual realmente hace esto lucir atractivo. El error de truncamiento es aún del orden de τ^2 en la ecuación velocidad, pero para la posición el error de truncamiento es ahora τ^3. Realmente, para el movimiento de proyectiles este método trabaja mejor que los otros dos. Infortunadamente, en otros sistemas físicos este método da pobres resultados.

7.1.5. Errores locales, errores globales y elección del paso de tiempo.

Para juzgar la precisión de estos métodos necesitamos distinguir entre errores de truncamiento locales y globales. Hasta ahora, el error de truncamiento que hemos discutido ha

[3] A. Cromer, "Stable solutions using the Euler approximation", *Am. J. Phys.*, **49** 455-9 (1981).

sido el error local, el error cometido en un único paso de tiempo. En un problema típico nosotros deseamos evaluar la trayectoria desde $t = 0$ a $t = T$. El número de pasos de tiempo es $N_T = T/\tau$; notemos que si reducimos τ, debemos tomar más pasos. Si el error local es $\mathcal{O}(\tau^n)$, entonces estimamos el error global como

$$\begin{aligned} \text{error global} &\propto N_T \times (\text{error local}) \\ &= N_T \mathcal{O}(\tau^n) = \frac{T}{\tau}\mathcal{O}(\tau^n) = T\mathcal{O}(\tau^{n-1}) \ . \end{aligned} \tag{7.25}$$

Por ejemplo, el método de Euler tiene un error local de truncamiento de $\mathcal{O}(\tau^2)$, pero un error global de truncamiento de $\mathcal{O}(\tau)$. Por supuesto, este análisis nos da sólo una estimación ya que no sabemos si los errores locales se acumularán o se cancelarán (*i.e.* interferencia constructiva o destructiva). El verdadero error global para un esquema numérico es altamente dependiente del problema que se está estudiando.

Una pregunta que siempre aparece es ¿cómo elegir el τ? Tratemos de responderla. Primero, supongamos que los errores de redondeo son despreciables tal que sólo debemos preocuparnos por los errores de truncamiento. Desde (7.10) y (7.16), el error local de truncamiento en el cálculo de la posición usando el método de Euler es aproximadamente $\tau^2 r'' = \tau^2 a$. Usando sólo estimaciones del orden de magnitud, tomamos $a \approx 10$ [m/s^2], el error en un solo paso en la posición es de 10^{-1} [m], cuando $\tau = 10^{-1}$ [s]. Si el tiempo de vuelo $T \approx 10^0$ [s], entonces el error global es del orden de metros. Si un error de esta magnitud es inaceptable entonces debemos disminuir el paso en el tiempo. Finalmente usando un paso de tiempo 10^{-1} [s] no introduciríamos ningún error significativo de redondeo dada la magnitud de los otros parámetros del problema.

En el mundo real, a menudo no podemos hacer un análisis tan elegante por una variedad de razones (ecuaciones complicadas, problemas con el redondeo, flojera, etc.). Sin embargo, a menudo podemos usar la intuición física. Respóndase usted mismo "¿en qué escala de tiempo el movimiento es casi lineal?". Por ejemplo, para la trayectoria completa de una pelota de *baseball* que es aproximadamente parabólica, el tiempo en el aire son unos pocos segundos, entonces el movimiento es aproximadamente lineal sobre una escala de tiempo de unos pocos centésimos de segundo. Para revisar nuestra intuición, nosotros podemos comparar los resultados obtenidos usando $\tau = 10^{-1}$ [s] y $\tau = 10^{-2}$ [s] y, si ellos son suficientemente cercanos, suponemos que todo está bien. A veces automatizamos la prueba de varios valores de τ; el programa es entonces llamado *adaptativo* (construiremos un programa de este tipo más adelante). Como con cualquier método numérico, la aplicación ciega de esta técnica es poco recomendada, aunque con sólo un poco de cuidado ésta puede ser usada exitosamente.

7.1.6. Programa de la pelota de *baseball*.

La tabla 7.1 bosqueja un simple programa, llamado `balle`, que usa el método de Euler para calcular la trayectoria de una pelota de *baseball*. Antes de correr el programa, establezcamos algunos valores razonables para tomar como entradas. Una velocidad inicial de $|\vec{v}_1|$ =15 [m/s] nos da una pelota que le han pegado débilmente. Partiendo del origen y despreciando la resistencia del aire, el tiempo de vuelo es de 2.2 [s], y el alcance horizontal es sobre los 23 [m] cuando el ángulo inicial $\theta = 45°$. A continuación, mostramos la salida a pantalla del programa `balle` en C++ cuando es corrido bajo estas condiciones:

- Fijar la posición inicial $\vec{r}_1$ y la velocidad inicial $\vec{v}_1$ de la pelota de *baseball*.
- Fijar los parámetros físicos (m, C_d, etc.).
- Iterar hasta que la bola golpée en el piso o el máximo número de pasos sea completado.
  - Grabar posición (calculada y teórica) para graficar.
  - Calcular la aceleración de la pelota de *baseball*.
  - Calcular la nueva posición y velocidad, $\vec{r}_{n+1}$ y $\vec{v}_{n+1}$, Usando el método de Euler, (7.18) y (7.19).
  - Si la pelota alcanza el suelo ($y < 0$) para la iteración.
- Imprimir el alcance máximo y el tiempo de vuelo.
- Graficar la trayectoria de la pelota de *baseball*.

Cuadro 7.1: Bosquejo del programa `balle`, el cual calcula la trayectoria de una pelota de *baseball* usando el método de Euler.

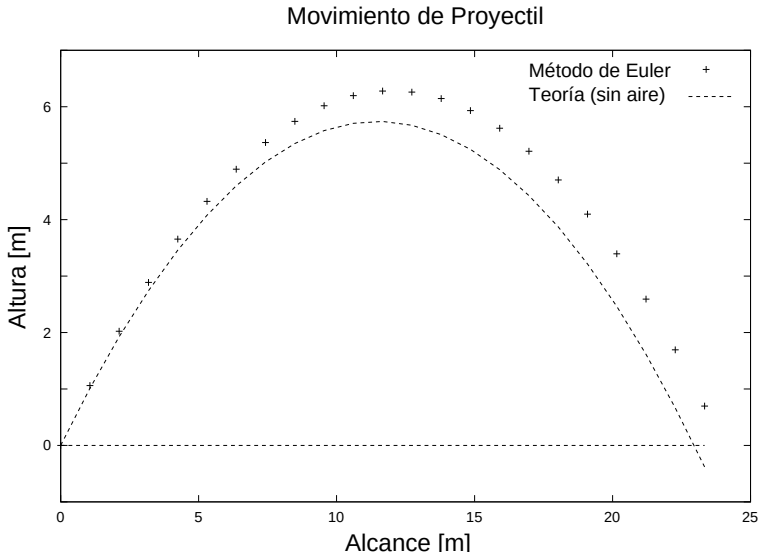


Figura 7.2: Salida del programa `balle` para una altura inicial de 0 [m], una velocidad inicial de 15 [m/s], y un paso de tiempo τ =0.1 [s]. No hay resistencia del aire. La línea continua es la teórica y los puntos son los calculados, la diferencia se debe a errores de truncamiento.

```
jrogan@huelen:~/programas$ balle
Ingrese la altura inicial [m] : 0
Ingrese la velocidad inicial [m/s]: 15
Ingrese angulo inicial (grados): 45
```

```
Ingrese el paso en el tiempo, tau en [s]: 0.1
Tiempo de vuelo: 2.2
Alcance: 24.3952
```

La salida en Octave debiera ser muy similar.

La trayectoria calculada por el programa es mostrada en la figura 7.2. Usando un paso de $\tau = 0.1$ [s], el error en el alcance horizontal es sobre un metro, como esperábamos del error de truncamiento. A velocidades bajas los resultados no son muy diferentes si incluimos la resistencia con el aire, ya que $|\vec{F}_a(\vec{v}_1)|/m \approx g/7$.

Ahora tratemos de batear un cuadrangular. Consideremos una velocidad inicial grande $|\, v_1 \,| = 50$ [m/s]. Debido a la resistencia, encontramos que el alcance es reducido a alrededor de 125 [m], menos de la mitad de su máximo teórico. La trayectoria es mostrada figura 7.3, notemos cómo se aparta de la forma parabólica.

En nuestras ecuaciones para el vuelo de una pelota de *baseball* no hemos incluido todos los factores en el problema. El coeficiente de arrastre no es constante sino más bien una complicada función de la velocidad. Además, la rotación de la pelota ayuda a levantar la pelota (efecto Magnus).

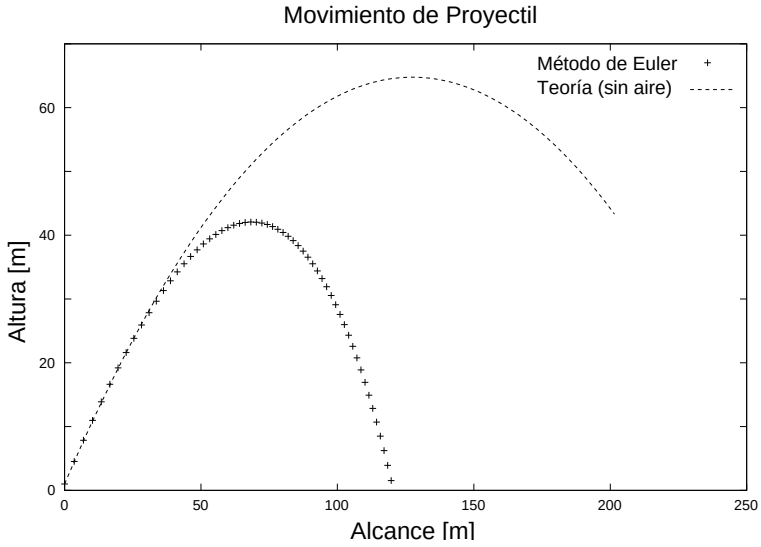


Figura 7.3: Salida del programa `balle` para una altura inicial de 1 [m], una velocidad inicial de 50 [m/s], y un paso de tiempo τ =0.1 [s]. Con resistencia del aire.

7.2. Péndulo simple.

7.2.1. Ecuaciones básicas.

El movimiento de los péndulos ha fascinado a físicos desde que Galileo fue hipnotizado por la lámpara en la Catedral de Pisa. El problema es tratado en los textos de mecánica básica pero antes de apresurarnos a calcular con el computador, revisemos algunos resultados

básicos. Para un péndulo simple es más conveniente describir la posición en términos del desplazamiento angular, $\theta(t)$. La ecuación de movimiento es

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\,\mathrm{sen}\,\theta \ , \tag{7.26}$$

donde L es la longitud del brazo y g es la aceleración de gravedad. En la aproximación para ángulo pequeño, $\mathrm{sen}\,\theta \approx \theta$, la ecuación (7.26) se simplifica a

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\theta \ . \tag{7.27}$$

Esta ecuación diferencial ordinaria es fácilmente resuelta para obtener

$$\theta(t) = C_1 \cos\left(\frac{2\pi t}{T_s} + C_2\right) \ , \tag{7.28}$$

donde las constantes C_1 y C_2 están determinadas por los valores iniciales de θ y $\omega = d\theta/dt$. El período para ángulos pequeños, T_s es

$$T_s = 2\pi\sqrt{\frac{L}{g}} \ . \tag{7.29}$$

Esta aproximación es razonablemente buena para oscilaciones con amplitudes menores o iguales a 20°.

Sin la aproximación para ángulos pequeños, la ecuación de movimiento es más difícil de resolver. Sin embargo, sabemos de la experiencia que el movimiento es todavía periódico. En efecto, es posible obtener una expresión para el período sin resolver explícitamente $\theta(t)$. La energía total es

$$E = \frac{1}{2}mL^2\omega^2 - mgL\cos\theta \ , \tag{7.30}$$

donde m es la masa de la lenteja. La energía total es conservada e igual a $E = -mgL\cos\theta_m$, donde θ_m es el ángulo máximo. De lo anterior, tenemos

$$\frac{1}{2}mL^2\omega^2 - mgL\cos\theta = mgL\cos\theta_m \ , \tag{7.31}$$

o

$$\omega^2 = \frac{2g}{L}\left(\cos\theta - \cos\theta_m\right) \ . \tag{7.32}$$

Ya que $\omega = d\theta/dt$,

$$dt = \frac{d\theta}{\sqrt{\dfrac{2g}{L}\left(\cos\theta - \cos\theta_m\right)}} \ . \tag{7.33}$$

En un período el péndulo se balancea de $\theta = \theta_m$ a $\theta = -\theta_m$ y regresa a $\theta = \theta_m$. Así, en medio período el péndulo se balancea desde $\theta = \theta_m$ a $\theta = -\theta_m$. Por último, por el mismo argumento,

en un cuarto de período el péndulo se balancea desde $\theta = \theta_m$ a $\theta = 0$, así integrando ambos lados de la ecuación (7.33)

$$\frac{T}{4} = \sqrt{\frac{L}{2g}} \int_0^{\theta_m} \frac{d\theta}{\sqrt{(\cos\theta - \cos\theta_m)}} \ . \tag{7.34}$$

Esta integral podría ser reescrita en términos de funciones especiales usando la identidad $\cos 2\theta = 1 - 2\,\mathrm{sen}^2\,\theta$, tal que

$$T = 2\sqrt{\frac{L}{g}} \int_0^{\theta_m} \frac{d\theta}{\sqrt{(\mathrm{sen}^2\,\theta_m/2 - \mathrm{sen}^2\,\theta/2)}} \ . \tag{7.35}$$

Introduciendo $K(x)$, la integral elíptica completa de primera especie,[4]

$$K(x) \equiv \int_0^{\pi/2} \frac{dz}{\sqrt{1 - x^2\,\mathrm{sen}^2\,z}} \ , \tag{7.36}$$

podríamos escribir el período como

$$T = 4\sqrt{\frac{L}{g}} K(\mathrm{sen}\,\theta_m/2) \ , \tag{7.37}$$

usando el cambio de variable $\mathrm{sen}\,z = \mathrm{sen}(\theta/2)/\,\mathrm{sen}(\theta_m/2)$. Para valores pequeños de θ_m, podríamos expandir $K(x)$ para obtener

$$T = 2\pi\sqrt{\frac{L}{g}} \left(1 + \frac{1}{16}\,\theta_m^2 + \ldots\right) \ . \tag{7.38}$$

Note que el primer término es la aproximación para ángulo pequeño (7.29).

7.2.2. Fórmulas para la derivada centrada.

Antes de programar el problema del péndulo miremos un par de otros esquemas para calcular el movimiento de un objeto. El método de Euler está basado en la formulación de la derivada derecha para df/dt dado por (7.7). Una definición equivalente para la derivada es

$$f'(t) = \lim_{\tau\to 0} \frac{f(t+\tau) - f(t-\tau)}{2\tau} \ . \tag{7.39}$$

Esta fórmula se dice centrada en t. Mientras esta fórmula parece muy similar a la ecuación (7.7), hay una gran diferencia cuando τ es finito. Nuevamente, usando la expansión de Taylor,

$$f(t+\tau) = f(t) + \tau f'(t) + \frac{1}{2}\tau^2 f''(t) + \frac{1}{6}\tau^3 f^{(3)}(\zeta_+) \ , \tag{7.40}$$

$$f(t-\tau) = f(t) - \tau f'(t) + \frac{1}{2}\tau^2 f''(t) - \frac{1}{6}\tau^3 f^{(3)}(\zeta_-) \ , \tag{7.41}$$

[4] I.S. Gradshteyn and I.M. Ryzhik, *Table of Integral, Series and Products* (New York: Academic Press, 1965)

donde $f^{(3)}$ es la tercera derivada de $f(t)$ y $\zeta_\pm$ es un valor ente t y $t \pm \tau$. Restando las dos ecuaciones anteriores y reordenando tenemos,

$$f'(t) = \frac{f(t+\tau) - f(t-\tau)}{2\tau} - \frac{1}{6}\tau^2 f^{(3)}(\zeta) \ , \tag{7.42}$$

donde $f^{(3)}(\zeta) = (f^{(3)}(\zeta_+) + f^{(3)}(\zeta_-))/2$ y $t - \tau \leq \zeta \leq t + \tau$. Esta es la *aproximación en la primera derivada centrada*. El punto clave es que el error de truncamiento es ahora cuadrático en τ, lo cual es un gran progreso sobre la aproximación de las derivadas adelantadas que tiene un error de truncamiento $\mathcal{O}(\tau)$.

Usando las expansiones de Taylor para $f(t+\tau)$ y $f(t-\tau)$ podemos construir una fórmula centrada para la segunda derivada. La que tiene la forma

$$f''(t) = \frac{f(t+\tau) + f(t-\tau) - 2f(t)}{\tau^2} - \frac{1}{12}\tau^2 f^{(4)}(\zeta) \ , \tag{7.43}$$

donde $t - \tau \leq \zeta \leq t + \tau$. De nuevo, el error de truncamiento es cuadrático en τ. La mejor manera de entender esta fórmula es pensar que la segunda derivada está compuesta de una derivada derecha y de una derivada izquierda, cada una con incrementos de $\tau/2$.

Usted podría pensar que el próximo paso sería preparar fórmulas más complicadas que tengan errores de truncamiento aún más pequeños, quizás usando ambas $f(t \pm \tau)$ y $f(t \pm 2\tau)$. Aunque tales fórmulas existen y son ocasionalmente usadas, las ecuaciones (7.10), (7.42) y (7.43) sirven como el "caballo de trabajo" para calcular las derivadas primera y segunda.

7.2.3. Métodos del "salto de la rana" y de Verlet.

Para el péndulo, las posiciones y velocidades generalizadas son θ y ω, pero para mantener la misma notación anterior trabajaremos con $\vec{r}$ y $\vec{v}$. Comenzaremos de las ecuaciones de movimiento escritas como

$$\frac{d\vec{v}}{dt} = \vec{a}(\vec{r}(t)) \ , \tag{7.44}$$

$$\frac{d\vec{r}}{dt} = \vec{v}(t) \ . \tag{7.45}$$

Note que explícitamente escribimos la aceleración dependiente solamente de la posición. Discretizando la derivada temporal usando la aproximación de derivada centrada da,

$$\frac{\vec{v}(t+\tau) - \vec{v}(t-\tau)}{2\tau} + \mathcal{O}(\tau^2) = \vec{a}(\vec{r}(t)) \ , \tag{7.46}$$

para la ecuación de la velocidad. Note que aunque los valores de velocidad son evaluados en $t + \tau$ y $t - \tau$, la aceleración es evaluada en el tiempo t.

Por razones que pronto serán claras, la discretización de la ecuación de posición estará centrada entre $t + 2\tau$ y t,

$$\frac{\vec{r}(t+2\tau) - \vec{r}(t)}{2\tau} + \mathcal{O}(\tau^2) = \vec{v}(t+\tau) \ . \tag{7.47}$$

De nuevo usamos la notación $f_n \equiv f(t = (n-1)\tau)$, en la cual las ecuaciones (7.47) y (7.46) son escritas como,

$$\frac{\vec{v}_{n+1} - \vec{v}_{n-1}}{2\tau} + \mathcal{O}(\tau^2) = \vec{a}(\vec{r}_n) \ , \tag{7.48}$$

$$\frac{\vec{r}_{n+2} - \vec{r}_n}{2\tau} + \mathcal{O}(\tau^2) = \vec{v}_{n+1} \ . \tag{7.49}$$

Reordenando los términos para obtener los valores futuros a la izquierda,

$$\vec{v}_{n+1} = \vec{v}_{n-1} + 2\tau\vec{a}(\vec{r}_n) + \mathcal{O}(\tau^3) \ , \tag{7.50}$$

$$\vec{r}_{n+2} = \vec{r}_n + 2\tau\vec{v}_{n+1} + \mathcal{O}(\tau^3) \ . \tag{7.51}$$

Este es el *método del "salto de la rana" (leap frog)*. Naturalmente, cuando el método es usado en un programa, el término $\mathcal{O}(\tau^3)$ no va y por lo tanto constituye el error de truncamiento para el método.

El nombre "salto de la rana" es usado ya que la solución avanza en pasos de 2τ, con la posición evaluada en valores impares ($\vec{r}_1$, $\vec{r}_3$, $\vec{r}_5$, ...), mientras que la velocidad está calculada en los valores pares ($\vec{v}_2$, $\vec{v}_4$, $\vec{v}_6$, ...). Este entrelazamiento es necesario ya que la aceleración, la cual es una función de la posición, necesita ser evaluada en un tiempo que está centrado entre la velocidad nueva y la antigua. Algunas veces el esquema del "salto de la rana" es formulado como

$$\vec{v}_{n+1/2} = \vec{v}_{n-1/2} + \tau\vec{a}(\vec{r}_n) \ , \tag{7.52}$$

$$\vec{r}_{n+1} = \vec{r}_n + \tau\vec{v}_{n+1/2} \ , \tag{7.53}$$

con $\vec{v}_{n\pm1/2} \equiv \vec{v}(t = (n-1\pm1/2)\tau)$. En esta forma, el esquema es funcionalmente equivalente al método de Euler-Cromer.

Para el último esquema numérico de este capítulo tomaremos una aproximación diferente y empezaremos con,

$$\frac{d\vec{r}}{dt} = \vec{v}(t) \ , \tag{7.54}$$

$$\frac{d^2\vec{r}}{dt^2} = \vec{a}(\vec{r}) \ . \tag{7.55}$$

Usando las fórmulas diferenciales centradas para la primera y segunda derivada, tenemos

$$\frac{\vec{r}_{n+1} - \vec{r}_{n-1}}{2\tau} + \mathcal{O}(\tau^2) = \vec{v}_n \ , \tag{7.56}$$

$$\frac{\vec{r}_{n+1} + \vec{r}_{n-1} - 2\vec{r}_n}{\tau^2} + \mathcal{O}(\tau^2) = \vec{a}_n \ , \tag{7.57}$$

donde $\vec{a}_n \equiv \vec{a}(\vec{r}_n)$. Reordenando términos,

$$\vec{v}_n = \frac{\vec{r}_{n+1} - \vec{r}_{n-1}}{2\tau} + \mathcal{O}(\tau^2) \ , \tag{7.58}$$

$$\vec{r}_{n+1} = 2\vec{r}_n - \vec{r}_{n-1} + \tau^2\vec{a}_n + \mathcal{O}(\tau^4) \ . \tag{7.59}$$

Estas ecuaciones, conocidas como el *método de Verlet* [5], podrían parecer extrañas a primera vista, pero ellas son fáciles de usar. Suponga que conocemos $\vec{r}_0$ y $\vec{r}_1$; usando la ecuación (7.59), obtenemos $\vec{r}_2$. Conociendo $\vec{r}_1$ y $\vec{r}_2$ podríamos ahora calcular $\vec{r}_3$, luego usando la ecuación (7.58) obtenemos $\vec{v}_2$, y así sucesivamente.

Los métodos del "salto de la rana" y de Verlet tienen la desventaja que no son "autoiniciados". Usualmente tenemos las condiciones iniciales $\vec{r}_1 = \vec{r}(t=0)$ y $\vec{v}_1 = \vec{v}(t=0)$, pero no $\vec{v}_0 = \vec{v}(t=-\tau)$ [necesitado por el "salto de la rana" en la ecuación (7.50)] o $\vec{r}_0 = \vec{r}(t=-\tau)$ [necesitado por Verlet en la ecuación (7.59)]. Este es el precio que hay que pagar para los esquemas centrados en el tiempo.

Para lograr que estos métodos partan, tenemos una variedad de opciones. El método de Euler-Cromer, usando la ecuación (7.53), toma $\vec{v}_{1/2} = \vec{v}_1$, lo cual es simple pero no muy preciso. Una alternativa es usar otro esquema para lograr que las cosas partan, por ejemplo, en el "salto de la rana" uno podría tomar un paso tipo Euler para atrás, $\vec{v}_0 = \vec{v}_1 - \tau\vec{a}_1$. Algunas precauciones deberían ser tomadas en este primer paso para preservar la precisión del método; usando

$$\vec{r}_0 = \vec{r}_1 - \tau\vec{v}_1 + \frac{\tau^2}{2}\,\vec{a}(\vec{r}_1)\ , \tag{7.60}$$

es una buena manera de comenzar el método de Verlet.

Además de su simplicidad, el método del "salto de la rana" a menudo tiene propiedades favorables (*e.g.* conservación de la energía) cuando resuelve ciertos problemas. El método de Verlet tiene muchas ventajas. Primero, la ecuación de posición tiene un error de truncamiento menor que otros métodos. Segundo, si la fuerza es solamente una función de la posición y si nos preocupamos sólo de la trayectoria de la partícula y no de su velocidad (como en muchos problemas de mecánica celeste), podemos saltarnos completamente el cálculo de velocidad. El método es popular para el cálculo de las trayectorias en sistemas con muchas partículas, por ejemplo, el estudio de fluidos a nivel microscópico.

7.2.4. Programa de péndulo simple.

Las ecuaciones de movimiento para un péndulo simple son

$$\frac{d\omega}{dt} = \alpha(\theta) \quad \frac{d\theta}{dt} = \omega\ , \tag{7.61}$$

donde la aceleración angular $\alpha(\theta) = -g\,\mathrm{sen}\,\theta/L$. El método de Euler para resolver estas ecuaciones diferenciales ordinarias es iterar las ecuaciones:

$$\theta_{n+1} = \theta_n + \tau\omega_n\ , \tag{7.62}$$

$$\omega_{n+1} = \omega_n + \tau\alpha_n\ . \tag{7.63}$$

Si estamos interesados solamente en el ángulo y no la velocidad, el método de Verlet sólo usa la ecuación

$$\theta_{n+1} = 2\theta_n - \theta_{n-1} + \tau^2\alpha_n\ . \tag{7.64}$$

[5] L.Verlet, "Computer experiments on classical fluid I. Thermodynamical properties of Lennard-Jones molecules", *Phys. Rev.* **159**, 98-103 (1967).

- Seleccionar el método a usar: Euler o Verlet.
- Fijar la posición inicial θ_1 y la velocidad $\omega_1 = 0$ del péndulo.
- Fijar los parámetros físicos y otras variables.
- Tomar un paso para atrás para partir Verlet; ver ecuación (7.60).
- Iterar sobre el número deseado de pasos con el paso de tiempo y método numérico dado.
  - Grabar ángulo y tiempo para graficar.
  - Calcular la nueva posición y velocidad usando el método de Euler o de Verlet.
  - Comprobar si el péndulo a pasado a través de $\theta = 0$; si es así usar el tiempo transcurrido para estimar el período.
- Estima el período de oscilación, incluyendo barra de error.
- Graficar las oscilaciones como θ versus t.

Cuadro 7.2: Bosquejo del programa `péndulo`, el cual calcula el tiempo de evolución de un péndulo simple usando el método de Euler o Verlet.

En vez de usar las unidades SI, usaremos las unidades adimensionales naturales del problema. Hay solamente dos parámetros en el problema, g y L y ellos siempre aparecen en la razón g/L. Fijando esta razón a la unidad, el período para pequeñas amplitudes $T_s = 2\pi$. En otras palabras, necesitamos solamente una unidad en el problema: una escala de tiempo. Ajustamos nuestra unidad de tiempo tal que el período de pequeñas amplitudes sea 2π.

La tabla 7.2 presenta un bosquejo del programa `pendulo`, el cual calcula el movimiento de un péndulo simple usando o el método de Euler o el de Verlet. El programa estima el período por registrar cuando el ángulo cambia de signo; esto es verificar si θ_n y θ_{n+1} tienen signos opuestos probando si $\theta_n * \theta_{n+1} < 0$. Cada cambio de signo da una estimación para el período, $\tilde{T}_k = 2\tau(n_{k+1} - n_k)$, donde n_k es el paso de tiempo en el cual el k-ésimo cambio de signo ocurre. El período estimado de cada inversión de signo es registrado, y el valor medio calculado como

$$\left\langle \tilde{T} \right\rangle = \frac{1}{M} \sum_{k=1}^{M} \tilde{T}_k \ , \tag{7.65}$$

donde M es el número de veces que $\tilde{T}$ es evaluado. La barra de error para esta medición del período es estimada como $\sigma = s/\sqrt{M}$, donde

$$s = \sqrt{\frac{1}{M-1} \sum_{k=1}^{M} \left(\tilde{T}_k - \left\langle \tilde{T} \right\rangle \right)^2} \ , \tag{7.66}$$

es la desviación estándar de la muestra $\tilde{T}$. Note que cuando el número de medidas se incre-

menta, la desviación estándar de la muestra tiende a una constante, mientras que la barra de error estimado decrece.

Para comprobar el programa `pendulo`, primero tratamos con ángulos iniciales pequeños, θ_m, ya que conocemos el período $T \approx 2\pi$. Tomando $\tau = 0.1$ tendremos sobre 60 puntos por oscilación; tomando 300 pasos deberíamos tener como cinco oscilaciones. Para $\theta_m = 10°$, el método de Euler calcula un período estimado de $\langle \tilde{T} \rangle = 6.375 \pm 0.025$ sobre un 1.5 % mayor que el esperado $T = 2\pi(1.002)$ dado por la ecuación (7.38). Nuestro error estimado para el período es entorno a $\pm\tau$ en cada medida. Cinco oscilaciones son 9 medidas de $\tilde{T}$, así que nuestro error estimado para el período debería ser $(\tau/2)/\sqrt{9} \approx 0.02$. Notemos que la estimación está en buen acuerdo con los resultados obtenidos usando la desviación estándar. Hasta aquí todo parece razonable.

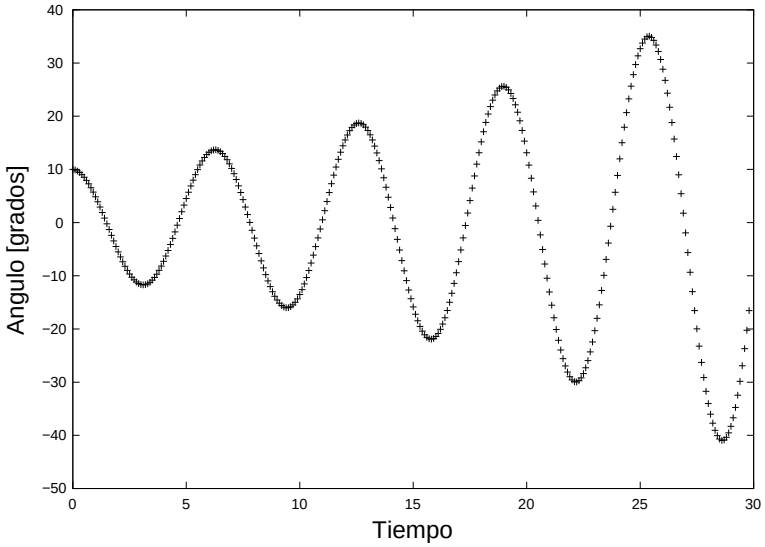


Figura 7.4: Salida del programa `péndulo` usando el método de Euler. El ángulo inicial es $\theta_m = 10°$, el paso en el tiempo es $\tau = 0.1$, y 300 iteraciones fueron calculadas.

Infortunadamente si miramos el gráfico 7.4 nos muestra los problemas del método de Euler. La amplitud de oscilación crece con el tiempo. Ya que la energía es proporcional al ángulo máximo, esto significa que la energía total se incrementa en el tiempo. El error global de truncamiento en el método de Euler se acumula en este caso. Para pasos de tiempos pequeños $\tau = 0.05$ e incrementos en el número de pasos (600) podemos mejorar los resultados, ver figura 7.5, pero no eliminamos el error. El método del punto medio tiene la misma inestabilidad numérica.

Usando el método de Verlet con $\theta_m = 10°$, el paso en el tiempo $\tau = 0.1$ y 300 iteraciones obtenemos los resultados graficados en 7.5. Estos resultados son mucho mejores; la amplitud de oscilación se mantiene cerca de los 10° y $\langle \tilde{T} \rangle = 6.275 \pm 0.037$. Afortunadamente el método de Verlet, el del "salto de rana" y el de Euler-Cromer no sufren de la inestabilidad numérica encontrada al usar el método de Euler.

Para $\theta_m = 90°$, la primera corrección para la aproximación de ángulo pequeño, ecuación (7.38), da $T = 7.252$. Usando el método de Verlet, el programa da un período estimado de $\langle \tilde{T} \rangle = 7.414 \pm 0.014$, lo cual indica que (7.38) es una buena aproximación (alrededor de un

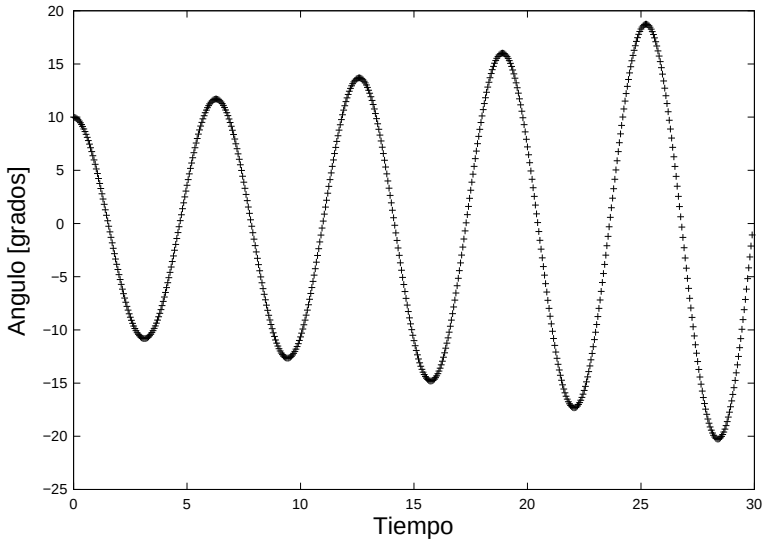


Figura 7.5: Salida del programa `péndulo` usando el método de Euler. El ángulo inicial es $\theta_m = 10°$, el paso en el tiempo es $\tau = 0.05$ y 600 iteraciones fueron calculadas.

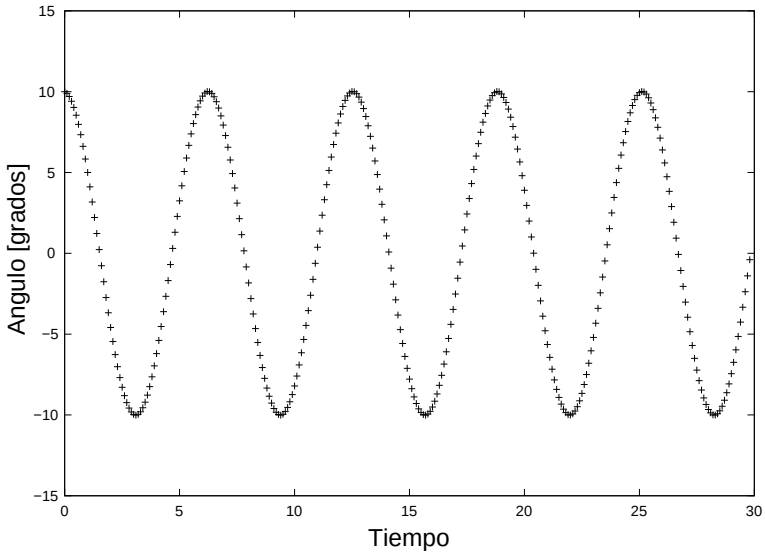


Figura 7.6: Salida del programa `péndulo` usando el método de Verlet. El ángulo inicial es $\theta_m = 10°$, el paso en el tiempo es $\tau = 0.1$ y 300 iteraciones fueron calculadas.

2 % de error), aún para ángulos grandes. Para el ángulo muy grande de $\theta_m = 170°$, vemos la trayectoria en la figura 7.6. Notemos como la curva tiende a aplanarse en los puntos de retorno. En este caso el período estimado es $\langle \tilde{T} \rangle = 15.3333 \pm 0.0667$, mientras que (7.38) da $T = 9.740$, indicando que esta aproximación para (7.37) deja de ser válida para este ángulo tan grande.

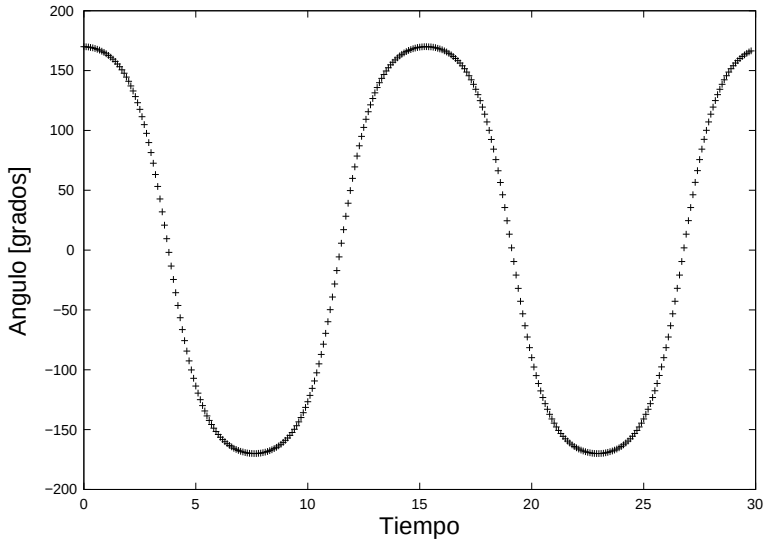


Figura 7.7: Salida del programa `péndulo` usando el método de Verlet. El ángulo inicial es $\theta_m = 170°$, el paso en el tiempo es $\tau = 0.1$ y 300 iteraciones fueron calculadas.

7.3. Listado de los programas.

7.3.1. `balle.cc`

```
#include "NumMeth.h"

int main()
{
  const double Cd=0.35;
  const double rho=1.293;                // [kg/m^3]
  const double radio=0.037;                      // [m]
  double A= M_PI*radio*radio ;
  double m=0.145;                                // [kg]
  double g=9.8;                  // [m/s^2]
  double a = -Cd*rho*A/(2.0e0*m) ;

  double v0, theta0, tau;
  ofstream salida ("salida.txt") ;
  ofstream salidaT ("salidaT.txt") ;

  double x0, y0;
  x0=0.0e0 ;
  cout << "Ingrese la altura inicial [m] : ";
  cin >> y0;
  cout << "Ingrese la velocidad inicial [m/s]: ";
  cin >> v0;
```

```
  cout <<"Ingrese angulo inicial (grados): ";
  cin >> theta0;

  int flagRA = 2 ;
  while (flagRA!=0 && flagRA !=1) {
    cout <<"Con resistencia del aire, Si= 1, No= 0: ";
    cin >> flagRA;
  }
  cout <<"Ingrese el paso en el tiempo, tau en [s]: ";
  cin >> tau ;
  double vxn=v0*cos(M_PI*theta0/180.0) ;
  double vyn=v0*sin(M_PI*theta0/180.0) ;
  double xn=x0 ;
  double yn=y0 ;
  double tiempo = -tau;
  while( yn >= y0) {
    tiempo +=tau ;
    salidaT << x0+v0*cos(M_PI*theta0/180.0) *tiempo <<" " ;
    salidaT << y0+v0*sin(M_PI*theta0/180.0) *tiempo -g*tiempo*tiempo/2.0e0<< endl;
    salida << xn << " " << yn << endl;
    if(flagRA==0) a=0.0e0 ;
    double v=sqrt(vxn*vxn+vyn*vyn) ;
    double axn= a*v*vxn ;
    double ayn= a*v*vyn -g ;
    double xnp1 = xn + tau*vxn ;
    double ynp1 = yn + tau*vyn ;
    double vxnp1 = vxn + tau*axn;
    double vynp1 = vyn + tau*ayn;
    vxn=vxnp1;
    vyn=vynp1;
    xn=xnp1 ;
    yn=ynp1 ;
  }
  cout << "Tiempo de vuelo: " << tiempo<< endl;
  cout << "Alcance: " << xn<<endl;
  salida.close();
  return 0;
}
```

7.3.2. pendulo.cc

```
#include "NumMeth.h"

int main()
{
```

```
int respuesta=2 ;
while(respuesta != 0 && respuesta !=1 ) {
  cout << "Eliga el metodo: Euler=0 y Verlet=1: " ;
  cin >> respuesta ;
}
double theta1 ;
double omega1 = 0.0e0;
cout << "Ingrese el angulo inicial (grados): ";
cin >> theta1 ;
theta1*=M_PI/180.0e0 ;
double tau ;
cout << "Ingrese el paso de tiempo: ";
cin >> tau ;
int pasos ;
cout << "Ingrese el numero de pasos: ";
cin >> pasos ;

double * periodo = new double[pasos] ;

ofstream salida ("salidaPendulo.txt");

double theta0= theta1-tau*omega1-tau*tau*sin(theta1) ;

double thetaNm1=theta0 ;
double thetaN=theta1 ;
double omegaN=omega1;

double thetaNp1, omegaNp1 ;

int nK=1;
int M=0 ;

for(int i=1; i< pasos; i++) {
  double alphaN=-sin(thetaN);
  if(respuesta==0) {          // Euler
    thetaNp1=thetaN+tau*omegaN ;
    omegaNp1=omegaN+tau*alphaN ;
  } else {
    thetaNp1=2.0e0*thetaN-thetaNm1+tau*tau*alphaN ;
  }
  salida << (i-1)*tau<<" " <<thetaNp1*180/M_PI<< endl ;
  if (thetaNp1*thetaN<0) {
    if(M==0) {
      periodo[M++]=0.0e0;
      nK=i ;
```

```
      } else {
        periodo[M++] = 2.0e0*tau*double(i-nK);
        nK=i ;
      }
    }

    thetaNm1=thetaN ;
    thetaN=thetaNp1 ;
    omegaN=omegaNp1 ;
  }
  double Tprom=0.0e0;
  for (int i=1; i < M; i++) Tprom+=periodo[i] ;
  Tprom/=double(M-1) ;
  double ssr=0.0 ;
  for (int i=1; i < M; i++) ssr+=(periodo[i]-Tprom)* (periodo[i]-Tprom);
  ssr/=double(M-2);
  double sigma =sqrt(ssr/double(M-1)) ;
  cout <<" Periodo = "<< Tprom << "+/-"<< sigma << endl ;
  salida.close() ;
  delete [] periodo;
  return 0;
}
```

Capítulo 8

Ecuaciones Diferenciales Ordinarias II: Métodos Avanzados.

versión 3.2 16 Diciembre 2003[1]

En el capítulo anterior aprendimos cómo resolver ecuaciones diferenciales ordinarias usando algunos métodos simples. En este capítulo haremos algo de mecánica celeste básica comenzando con el problema de Kepler. Al calcular la órbita de un satélite pequeño alrededor de un cuerpo masivo (*e.g* un cometa orbitando el Sol), descubriremos que métodos mucho más sofisticados son necesarios para manipular sistemas simples de dos cuerpos.

8.1. Órbitas de cometas.

8.1.1. Ecuaciones básicas.

Considere el problema de Kepler en el cual un pequeño satélite, tal como un cometa, orbita el Sol. Usamos un sistema de coordenadas Copernicano y fijamos el Sol en el origen. Por ahora, consideremos solamente la fuerza gravitacional entre el cometa y el Sol, y despreciemos todas las otras fuerzas (e.g., fuerzas debidas a los planetas, viento solar). La fuerza sobre el cometa es

$$\vec{F} = -\frac{GmM}{|\vec{r}|^3}\vec{r}\,, \tag{8.1}$$

donde $\vec{r}$ es la posición del cometa, m es su masa, $M = 1.99 \times 10^{30}$ [kg] es la masa del Sol, y $G = 6.67 \times 10^{-11}$ [m^3/kg s^2] es la constante gravitacional.

Las unidades naturales de longitud y tiempo para este problema no son metros ni segundos. Como unidad de distancia usaremos la unidad astronómica [AU], 1 AU=1.496×10^{11} [m], la cual es igual a la distancia media de la Tierra al Sol. La unidad de tiempo será el [año] AU (el período de una órbita circular de radio 1 [AU]). En estas unidades, el producto $GM = 4\pi^2$ [AU^3/año^2]. Tomaremos la masa del cometa, m, como la unidad; en unidades MKS la masa típica de un cometa es $10^{15\pm3}$ [kg].

Ahora tenemos suficiente para ensamblar nuestro programa, pero antes hagamos una rápida revisión de lo que sabemos de órbitas. Para un tratamiento completo podemos recurrir

[1]Este capítulo está basado en el tercer capítulo del libro: *Numerical Methods for Physics, second edition* de Alejandro L. Garcia, editorial Prentice Hall.

a algunos textos de mecánica estándar, tales como Symon[2]o Landau y Lifshitz[3]. La energía total del satélite es

$$E = \frac{1}{2}mv^2 - \frac{GMm}{r} \ , \tag{8.2}$$

donde $r = |\vec{r}|$ y $v = |\vec{v}|$. Esta energía total es conservada, tal como el momento angular,

$$\vec{L} = \vec{r} \times (m\vec{v}) \ . \tag{8.3}$$

Ya que este problema es bidimensional, consideraremos el movimiento en el plano x-y. El único componente distinto de cero del momento angular está en la dirección z.

Cuando la órbita es circular, en el sistema de referencia que gira con el satélite, la fuerza centrífuga es compensada por la fuerza gravitacional,

$$\frac{mv^2}{r} = \frac{GMm}{r^2} \ , \tag{8.4}$$

o

$$v = \sqrt{\frac{GM}{r}} \ . \tag{8.5}$$

Por colocar algunos valores, en una órbita circular en $r = 1$ [AU] la velocidad orbital es $v = 2\pi$ [AU/año] (cerca de 30.000 [km/h]). Reemplazando la ecuación (8.5) en (8.2), la energía total en una órbita circular es

$$E = -\frac{GMm}{2r} \ . \tag{8.6}$$

En una órbita elíptica, los semiejes mayores y menores, a y b, son desiguales (figura 8.1). La

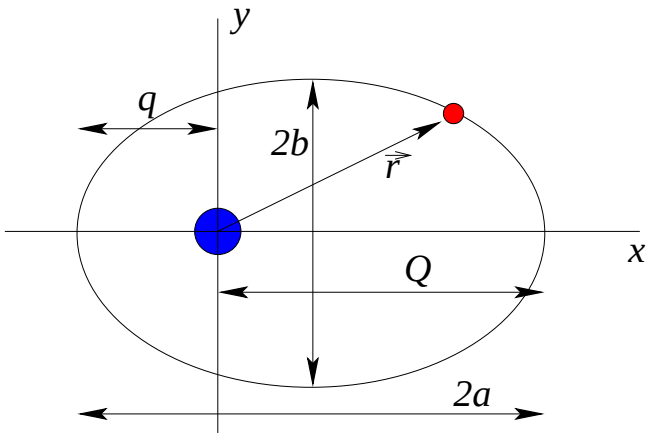


Figura 8.1: Órbita elíptica alrededor del Sol.

excentricidad, e, está definida como

$$e = \sqrt{1 - \frac{b^2}{a^2}} \ . \tag{8.7}$$

[2]K. Symon, *Mechanics* (Reading Mass.: Addison-Wesley, 1971).
[3]L. Landau and E. Lifshitz, *Mechanics* (Oxford: Pergamon, 1976).

Nombre del Cometa	T [años]	e	q [AU]	i	Primera pasada
Encke	3.30	0.847	0.339	12.4°	1786
Biela	6.62	0.756	0.861	12.6°	1772
Schwassmann-Wachmann 1	16.10	0.132	5.540	9.5°	1925
Halley	76.03	0.967	0.587	162.2°	239 A.C.
Grigg-Mellish	164.3	0.969	0.923	109.8°	1742
Hale-Bopp	2508.0	0.995	0.913	89.4°	1995

Cuadro 8.1: Datos orbitales de algunos cometas.

La excentricidad de la Tierra es $e = 0.017$, por lo tanto esta órbita está muy cercana de ser circular. La distancia del Sol al perihelio (punto de mayor aproximación) es $q = (1 - e)a$; la distancia del Sol al afelio es $Q = (1 + e)a$.

La ecuación (8.6) también se mantiene para una órbita elíptica, si reemplazamos el radio con el semieje mayor; por lo tanto la energía total es

$$E = -\frac{GMm}{2a} \ . \tag{8.8}$$

Note que $E \leq 0$. De las ecuaciones (8.2) y (8.8), encontramos que la velocidad orbital como función de la distancia radial es

$$v = \sqrt{GM\left(\frac{2}{r} - \frac{1}{a}\right)} \ . \tag{8.9}$$

La velocidad es máxima en el perihelio y mínima en el afelio, la razón entre las velocidades está dada por Q/q. Finalmente, usando la conservación de momento angular, podríamos derivar la tercera ley de Kepler,

$$T^2 = \frac{4\pi^2}{GM}a^3 \ , \tag{8.10}$$

donde T es el período de la órbita.

Los datos orbitales para unos pocos cometas bien conocidos están dados en la tabla 8.1. La inclinación, i, es el ángulo entre el plano orbital del cometa y el plano eclíptico (el plano de la órbita de los planetas). Cuando la inclinación es menor que los 90°, se dice que la órbita es directa, cuando es mayor que 90°, se dice que la órbita es retrógrada (*i.e.*, orbita el Sol en la dirección opuesta a la de los planetas).

8.1.2. Programa `orbita`.

Un programa simple, llamado `orbita`, que calcula las órbitas para el problema de Kepler usando varios métodos numéricos es propuesto en la tabla 8.2. El método de Euler, descrito en el capítulo anterior, calcula la trayectoria del cometa como

$$\vec{r}_{n+1} = \vec{r}_n + \tau\vec{v}_n \ , \tag{8.11}$$

$$\vec{v}_{n+1} = \vec{v}_n + \tau\vec{a}(\vec{r}_n) \ , \tag{8.12}$$

- Fijar la posición y velocidad inicial del cometa.
- Fijar los parámetros físicos (m, G, etc.).
- Iterar sobre el número deseado de pasos usando el método numérico especificado.
  - Grabar posición y la energía para graficar.
  - Calcular la nueva posición y velocidad usando:
    - Método de Euler (8.11), (8.12) o;
    - Método de Euler-Cromer (8.13), (8.14) o;
    - Método Runge-Kutta de cuarto orden (8.30), (8.31) o;
    - Método de Runge-Kutta adaptativo.
- Graficar la trayectoria del cometa.
- Graficar la energía del cometa versus el tiempo.

Cuadro 8.2: Bosquejo del programa `orbita`, el cual calcula la trayectoria de un cometa usando varios métodos numéricos.

donde $\vec{a}$ es la aceleración gravitacional. De nuevo, discretizamos el tiempo y usamos la notación $f_n \equiv f(t = (n-1)\tau)$, donde τ es el paso tiempo.

El caso de prueba más simple es una órbita circular. Para un radio orbital de 1 [AU], la ecuación (8.5) da una velocidad tangencial de 2π [AU/año]. Unos 50 puntos por revolución orbital nos daría una suave curva, tal que $\tau = 0.02$ [años] (o cercano a una semana) es un paso de tiempo razonable. Con esos valores, el programa `orbita` usando el método de Euler, da los resultados mostrados en la figura 8.2. Inmediatamente vemos que la órbita no es circular, pero una espiral hacia fuera. La razón es clara desde el gráfico de energía; en vez de ser constante, la energía total aumenta continuamente. Este tipo de inestabilidad se observa, también, en el método de Euler para el péndulo simple. Afortunadamente hay una solución simple a este problema: el método Euler-Cromer para calcular la trayectoria

$$\vec{v}_{n+1} = \vec{v}_n + \tau\vec{a}(\vec{r}_n) \ , \tag{8.13}$$

$$\vec{r}_{n+1} = \vec{r}_n + \tau\vec{v}_{n+1} \ . \tag{8.14}$$

Note que el sólo cambio del método de Euler en que primero calculamos la nueva velocidad, $\vec{v}_{n+1}$, y luego la usamos en el cálculo de la nueva posición. Para las mismas condiciones iniciales y paso de tiempo, el método de Euler-Cromer da resultados mucho mejores, como los mostrados en la figura 8.3. La órbita es casi circular, y la energía total se conserva. Las energías potencial y cinética no son constantes, pero este problema podría ser mejorado usando un paso de tiempo pequeño. El programa órbita también da la opción de usar el método de Runge-Kutta, el cual es descrito en las próximas dos secciones.

Aunque el método de Euler-Cromer hace un buen trabajo para bajas excentricidades, tiene problemas con órbitas más elípticas, como se muestra en la figura 8.4. Note que si la

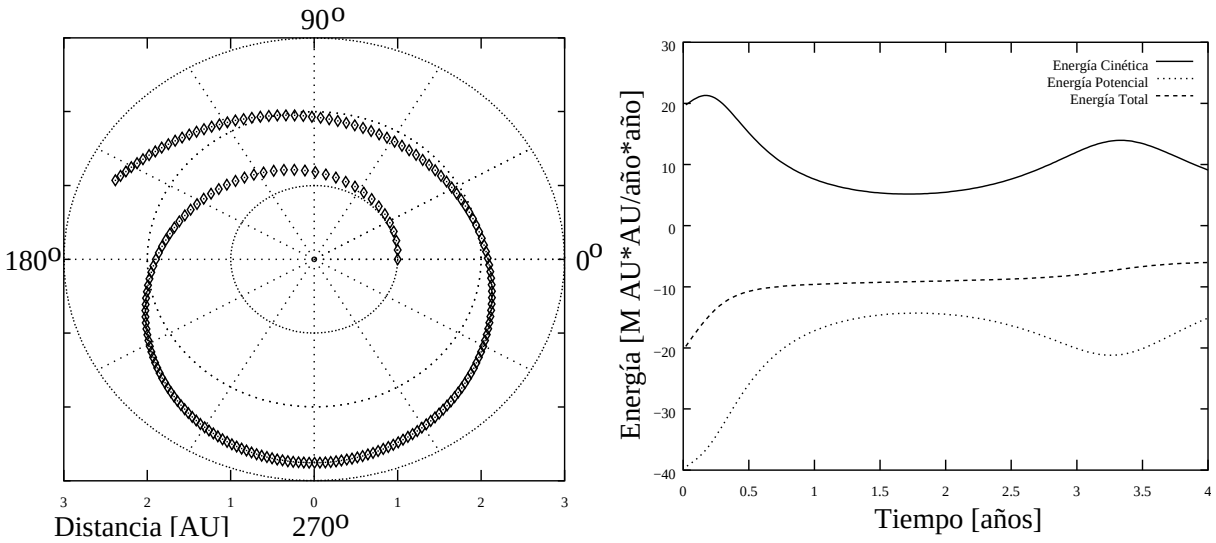


Figura 8.2: Gráfico de la trayectoria y la energía desde el programa `orbita` usando el método de Euler. La distancia radial inicial es 1 [AU] y la velocidad tangencial inicial es 2π [AU/año]. El paso en el tiempo es $\tau = 0.02$ [años]; y 200 pasos son calculados. Los resultados están en desacuerdo con la predicción teórica de una órbita circular con energía total constante.

energía llega a ser positiva; el satélite alcanza la velocidad de escape. Si bajamos el paso de tiempo desde $\tau = 0.02$ [años] a $\tau = 0.005$ [años] obtenemos mejores resultados, como los mostrados en la figura 8.5. Estos resultados no son del todo perfectos; la órbita puede ser una elipse cerrada, pero todavía tiene una notable deriva espúria.

En este punto usted se podría estar preguntando, "¿Por qué estamos estudiando este problema?, si la solución analítica es bien conocida". Es verdad que hay problemas mecánicos celestes más interesantes (*e.g.*, el efecto de perturbaciones sobre la órbita, problema de tres cuerpos). Sin embargo, antes de hacer los casos complicados podríamos, siempre, chequear los algoritmos de problemas conocidos. Suponga que introducimos una pequeña fuerza de arrastre sobre el cometa. Podríamos pecar de inocentes creyendo que la precisión de la figura 8.5 fue un fenómeno físico más que un artefacto numérico.

Claramente, el método de Euler-Cromer hace un trabajo inaceptable de rastreo de las órbitas más elípticas. Los resultados mejoran si achicamos el paso de tiempo, pero entonces sólo podemos rastrear unas pocas órbitas. Suponga que deseamos rastrear cometas para posibles impactos con la Tierra. Un gran cometa impactando sobre la Tierra sería más destructivo que una guerra nuclear. Muchos cometas tienen órbitas extremadamente elípticas y períodos de cientos de años. Esta amenaza desde el espacio exterior motiva nuestro estudio de métodos más avanzados para resolver ecuaciones diferenciales ordinarias.

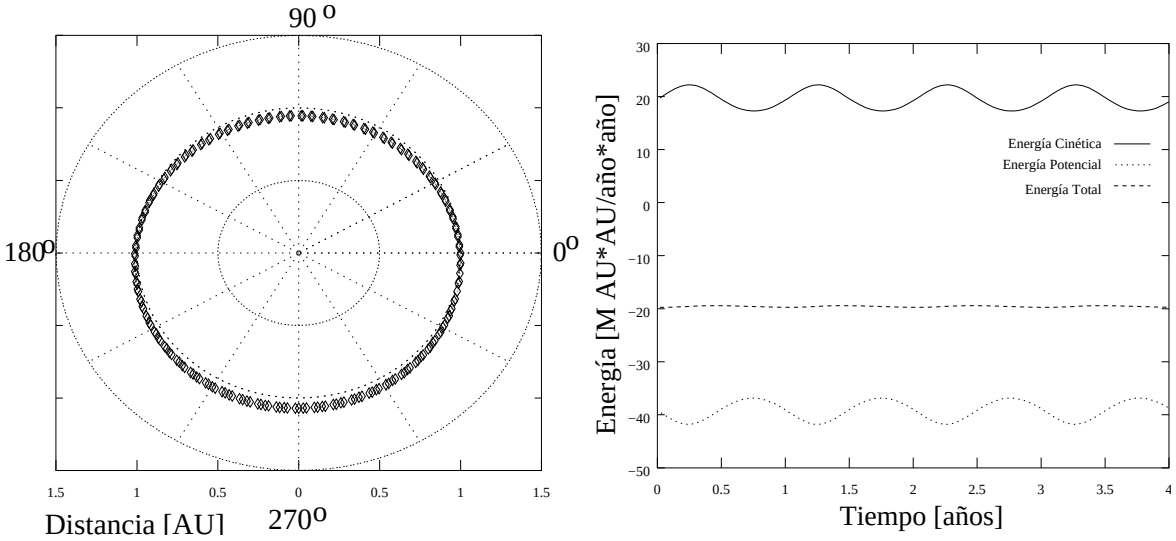


Figura 8.3: Gráfico de la trayectoria y la energía desde el programa `orbita` usando el método de Euler-Cromer. Los parámetros son los mismos que en la figura 8.2. Los resultados están en un acuerdo cualitativo al menos con la predicción teórica de una órbita circular con energía total constante.

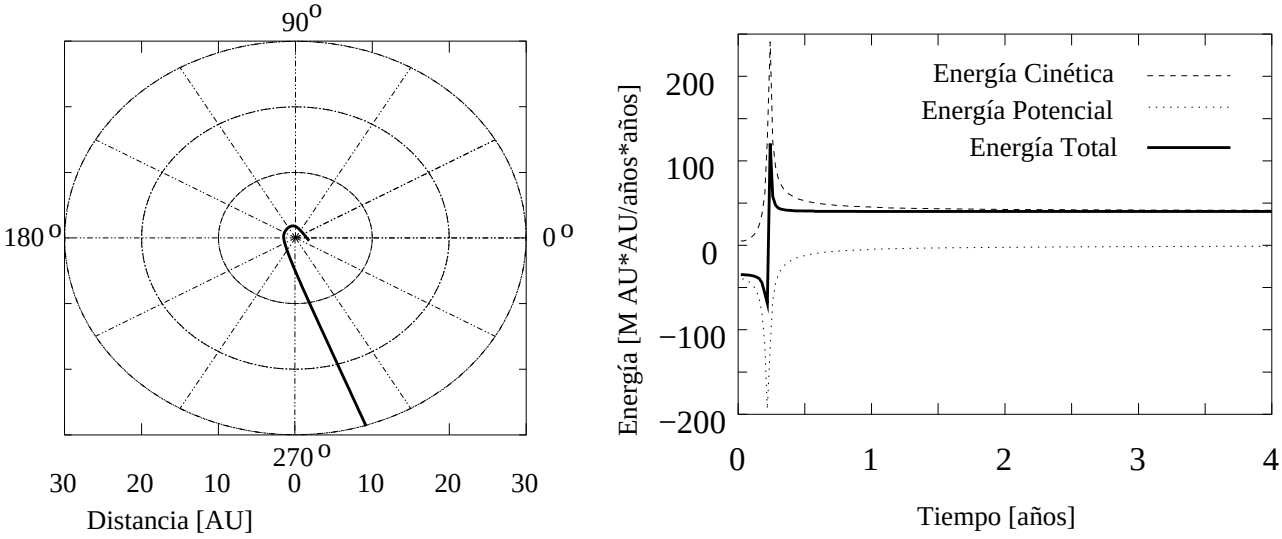


Figura 8.4: Gráfico de la trayectoria y la energía desde el programa `orbita` usando el método de Euler-Cromer. La distancia radial inicial es 1 [AU] y la velocidad tangencial inicial es π [AU/año]. El paso en el tiempo es $\tau = 0.02$ [años]; y 200 pasos son calculados. Debido al error numérico el cometa alcanza la velocidad de escape, la posición final es 35 [AU] y la energía total es positiva.

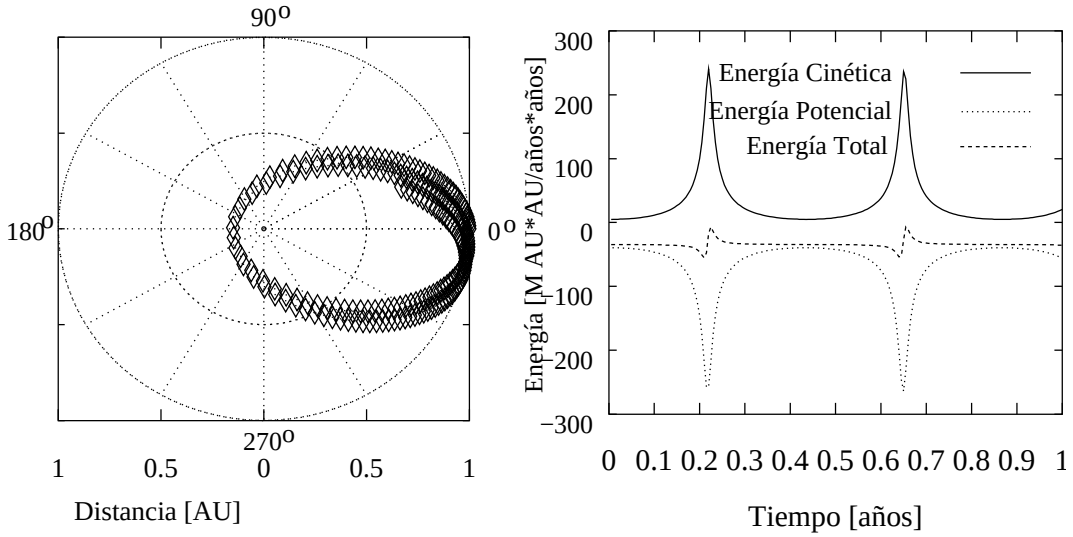


Figura 8.5: Gráfico de la trayectoria y la energía desde el programa `orbita` usando el método de Euler-Cromer. Los parámetros son los mismos que en la figura 8.4 excepto que el paso de tiempo es más pequeño $\tau = 0.005$ [años]. Los resultados son mejores, pero aún presenta una precesión espúria.

8.2. Métodos de Runge-Kutta.

8.2.1. Runge-Kutta de segundo orden.

Ahora miremos uno de los métodos más populares para resolver numéricamente las ecuaciones diferenciales ordinarias: Runge-Kutta. Primero trabajaremos las fórmulas generales de Runge-Kutta y luego las aplicaremos específicamente a nuestro problema del cometa. De esta manera será fácil usar el método Runge-Kutta para otros sistemas físicos. Nuestra ecuación diferencial ordinaria general toma la forma

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}(t), t) \ , \tag{8.15}$$

donde el vector de estado $x(t) = [x_1(t), x_2(t), \ldots x_N(t)]$ es la solución deseada. En el problema de Kepler tenemos

$$\vec{x}(t) = [r_x(t),\ r_y(t),\ v_x(t),\ v_y(t)] \ , \tag{8.16}$$

y

$$\begin{aligned} \vec{f}(\vec{x}(t), t) &= \left[\frac{dr_x}{dt},\ \frac{dr_y}{dt},\ \frac{dv_x}{dt},\ \frac{dv_y}{dt}\right] \ , \\ &= [v_x(t),\ v_y(t),\ F_x(t)/m,\ F_y(t)/m] \ , \end{aligned} \tag{8.17}$$

donde r_x, v_x, y F_x son las componentes x de la posición, la velocidad y la fuerza respectivamente (y lo mismo para la componente y). Note que en el problema de Kepler, la función f no depende explícitamente del tiempo sino que sólo depende de $\vec{x}(t)$.

Nuestro punto de partida es el método simple de Euler; en forma vectorial podría ser escrito como

$$\vec{x}(t+\tau) = \vec{x}(t) + \tau \vec{f}(\vec{x}, t) \ . \tag{8.18}$$

Consideremos la primera fórmula de Runge-Kutta:

$$\vec{x}(t+\tau) = \vec{x}(t) + \tau \vec{f}\left(\vec{x}^{\,*}\left(t+\frac{1}{2}\tau\right), t+\frac{1}{2}\tau\right) \ , \tag{8.19}$$

donde

$$\vec{x}^{\,*}\left(t+\frac{1}{2}\tau\right) \equiv \vec{x}(t) + \frac{1}{2}\tau \vec{f}(\vec{x}, t) \ . \tag{8.20}$$

Para ver de dónde viene esta fórmula, consideremos por el momento el caso de una variable. Sabemos que la expansión de Taylor

$$\begin{aligned} x(t+\tau) &= x(t) + \tau \frac{dx(\zeta)}{dt} \ , \\ &= x(t) + \tau f(x(\zeta), \zeta) \ , \end{aligned} \tag{8.21}$$

es exacta para algún valor de ς entre t y $t+\tau$, como se vio en la ecuación (7.10). La fórmula de Euler toma $\varsigma = t$; Euler-Cromer usa $\varsigma = t$ en la ecuación de velocidad y $\varsigma = t+\tau$ en la ecuación de posición. Runge-Kutta usa $\varsigma = t + \frac{1}{2}\tau$, lo cual pareciera ser una mejor estimación. Sin embargo, $x\left(t+\frac{1}{2}\tau\right)$ no es conocida, podemos aproximarla de la manera simple: usando un paso de Euler calculamos $x^*\left(t+\frac{1}{2}\tau\right)$ y usando esta como nuestra estimación de $x\left(t+\frac{1}{2}\tau\right)$.

A continuación un ejemplo simple usando la fórmula Runge-Kutta. Consideremos la ecuación

$$\frac{dx}{dt} = -x \ , \quad x(t=0) = 1 \ . \tag{8.22}$$

La solución de la ecuación (8.22) es $x(t) = e^{-t}$. Usando el método de Euler con un paso de tiempo de $\tau = 0.1$, tenemos

$$\begin{aligned} x(0.1) &= 1 + 0.1(-1) = 0.9 \ , \\ x(0.2) &= 0.9 + (0.1)(-0.9) = 0.81 \ , \\ x(0.3) &= 0.81 + 0.1(-0.81) = 0.729 \ , \\ x(0.4) &= 0.729 + 0.1(-0.729) = 0.6561 \ . \end{aligned}$$

Ahora tratemos con Runge-Kutta. Para hacer una correcta comparación usaremos un paso de tiempo mayor para Runge-Kutta $\tau = 0.2$ porque hace el doble de evaluaciones de $f(x)$. Por la fórmula de Runge-Kutta presentada arriba,

$$\begin{aligned} x^*(0.1) &= 1 + 0.1(-1) = 0.9 \ , \\ x(0.2) &= 1 + 0.2(-0.9) = 0.82 \ , \\ x^*(0.3) &= 0.82 + 0.1(-0.82) = 0.738 \\ x(0.4) &= 0.82 + 0.2(-0.738) = 0.6724 \ . \end{aligned}$$

Podemos comparar esto con la solución exacta $x(0.4) = \exp(-0.4) \approx 0.6703$. Claramente, Runge-Kutta lo hace mucho mejor que Euler; los errores porcentuales absolutos son 0.3 % y 2.1 % respectivamente.

8.2.2. Fórmulas generales de Runge-Kutta.

La fórmula discutida arriba no es la única posible para un Runge-Kutta de segundo orden. Aquí hay una alternativa:

$$\vec{x}(t+\tau) = \vec{x}(t) + \frac{1}{2}\tau[\vec{f}(\vec{x}(t),t) + \vec{f}(\vec{x}^{\,*}(t+\tau),t+\tau)] \ , \tag{8.23}$$

donde

$$\vec{x}^{\,*}(t+\tau) \equiv \vec{x}(t) + \tau\vec{f}(\vec{x}(t),t) \ . \tag{8.24}$$

Para entender este esquema, consideremos nuevamante el caso en una variable. En nuestra fórmula original, estimamos que $f(x(\varsigma),\varsigma)$ como $\frac{1}{2}[f(x,t) + f(x^{\,*}(t+\tau),t+\tau)]$.

Estas expresiones pueden ser deducidas usando la expansión de Taylor con dos variables,

$$f(x+h,t+\tau) = \sum_{n=0}^{\infty} \frac{1}{n!}\left(h\frac{\partial}{\partial x} + \tau\frac{\partial}{\partial t}\right)^n f(x,t) \ , \tag{8.25}$$

donde todas las derivadas son evaluadas en $(x,\, t)$. Para una fórmula general de Runge-Kutta de segundo orden queremos obtener una expresión de la siguiente forma

$$x(t+\tau) = x(t) + w_1\tau f(x(t),t) + w_2\tau f(x^*, t+\alpha\tau) \ , \tag{8.26}$$

donde

$$x^* \equiv x(t) + \beta\tau f(x(t),t) \ . \tag{8.27}$$

Hay cuatro coeficientes no especificados: α, β, w_1 y w_2. Note que cubrimos las ecuaciones (8.19) y (8.20) eligiendo los valores

$$w_1 = 0 \ , \quad w_2 = 1 \quad \alpha = \frac{1}{2} \ , \quad \beta = \frac{1}{2} \ , \tag{8.28}$$

y las ecuaciones (8.23) y (8.24) eligiendo

$$w_1 = \frac{1}{2} \ , \quad w_2 = \frac{1}{2} \ , \quad \alpha = 1 \ , \quad \beta = 1 \ . \tag{8.29}$$

Deseamos seleccionar cuatro coeficientes tal que tengamos una precisión de segundo orden; esto es deseamos calzar la serie de Taylor a través de los términos de la segunda derivada. Los detalles del cálculo se proponen como un ejercicio, pero cualquier grupo de coeficientes satisfacen las relaciones siguientes $w_1 + w_2 = 1$, $\alpha w_2 = 1/2$ y $\alpha = \beta$ darán un esquema Runge-Kutta de segundo orden. El error de truncamiento local es $\mathcal{O}(\tau^3)$, pero la expresión explícita no tiene una forma simple. No está claro que un esquema sea superior al otro ya que el error de truncamiento, siendo una función complicada de $f(x,t)$, variará de problema a problema.

8.2.3. Runge-Kutta de cuarto orden.

Presentamos las fórmulas de Runge-Kutta de segundo orden porque es fácil de comprender su construcción. En la práctica, sin embargo, el método más comúnmente usado es la siguiente fórmula de cuarto orden:

$$\vec{x}(t+\tau) = \vec{x}(t) + \frac{1}{6}\tau\left[\vec{F}_1 + 2\vec{F}_2 + 2\vec{F}_3 + \vec{F}_4\right] \ , \tag{8.30}$$

donde

$$\begin{aligned}
\vec{F}_1 &= \vec{f}(\vec{x}, t) \ , \\
\vec{F}_2 &= \vec{f}\left(\vec{x} + \frac{1}{2}\tau\vec{F}_1, t + \frac{1}{2}\tau\right) \ , \\
\vec{F}_3 &= \vec{f}\left(\vec{x} + \frac{1}{2}\tau\vec{F}_2, t + \frac{1}{2}\tau\right) \ , \\
\vec{F}_4 &= \vec{f}(\vec{x} + \tau\vec{F}_3, t + \tau) \ .
\end{aligned} \tag{8.31}$$

El siguiente extracto del *Numerical Recipes*[4] resume mejor el estado que las fórmulas de arriba tienen en el mundo del análisis numérico:

> Para muchos usuarios científicos, el método de Runge-Kutta de cuarto orden no es sólo la primera palabra en esquemas de integración para ecuaciones diferenciales ordinarias, si no que es la última también. De hecho, usted puede ir bastante lejos con este viejo caballito de batalla, especialmente si los combina con un algorítmo de paso adaptativo... Bulirsch-Stoer o los métodos predictor-corrector pueden ser mucho más eficientes para problemas donde se require una alta precisión. Estos métodos son los finos caballos de carrera mientras que Runge-Kutta es el fiel caballo de tiro.

Usted se preguntará, ¿por qué fórmulas de cuarto orden y no de orden superior? Bien, los métodos de orden superior tienen un error de truncamiento mejor, pero también requieren más cálculo, esto es, más evaluaciones de $f(x,t)$. Hay dos opciones, hacer más pasos con un τ pequeño usando un método de orden inferior o hacer pocos pasos con un τ más grande usando un método de orden superior. Ya que los métodos de Runge-Kutta de órdenes superiores son muy complicados, el esquema de cuarto orden dado anteriormente es muy conveniente. Entre paréntesis, el error de truncamiento local para Runge-Kutta de cuarto orden es $\mathcal{O}(\tau^5)$.

- *Entradas:* $\vec{x}(t)$, t, τ, $\vec{f}(\vec{x}, t; \lambda)$, y λ.
- *Salidas:* $\vec{x}(t+\tau)$.
- Evaluación $\vec{F}_1$, $\vec{F}_2$, $\vec{F}_3$ y $\vec{F}_4$ usando ecuación (8.31).
- Cálculo de $\vec{x}(t+\tau)$ usando Runge-Kutta de cuarto orden, usando ecuación (8.30).

Cuadro 8.3: Bosquejo de la función `rk4`, la cual evalúa un paso simple usando el método Runge-Kutta de cuarto orden.

Para implementar métodos de cuarto orden para nuestro problema de la órbita, usaremos la función `rk4` (tabla 8.3). Esta función toma como datos: el estado actual del sistema, $\vec{x}(t)$; el

[4] W. Press, B. Flannery, S. Tukolsky and W. Vetterling, *Numerical Recipes in* FORTRAN, 2nd ed. (Cambridge: Cambridge University Press 1992).

- *Entradas:* $\vec{x}(t)$, t (no se usa), GM.
- *Salidas:* $d\vec{x}(t)/dt$.
- Evalúa la aceleración $\vec{a} = -(GM\vec{r}/\,|\,\vec{r}\,|^3)$.
- *Retorno:* $d\vec{x}(t)/dt = [v_x, v_y, a_x, a_y]$.

Cuadro 8.4: Bosquejo de la función `gravrk`, la cual es usada por la función Runge-Kutta para evaluar las ecuaciones de movimiento para el problema de Kepler.

paso de tiempo para ser usado, τ; el tiempo actual, t; la función $\vec{f}(\vec{x}(t), t; \lambda)$; donde λ es una lista de parámetros usados por $\vec{f}$. La salida es el nuevo estado del sistema, $\vec{x}(t+\tau)$, calculado por el método de Runge-Kutta. Usando Runge-Kutta de cuarto orden da los resultados mostrados en la figura 8.6, la cual es mucho mejor que las obtenidas usando el método de Euler-Cromer (figura 8.5).

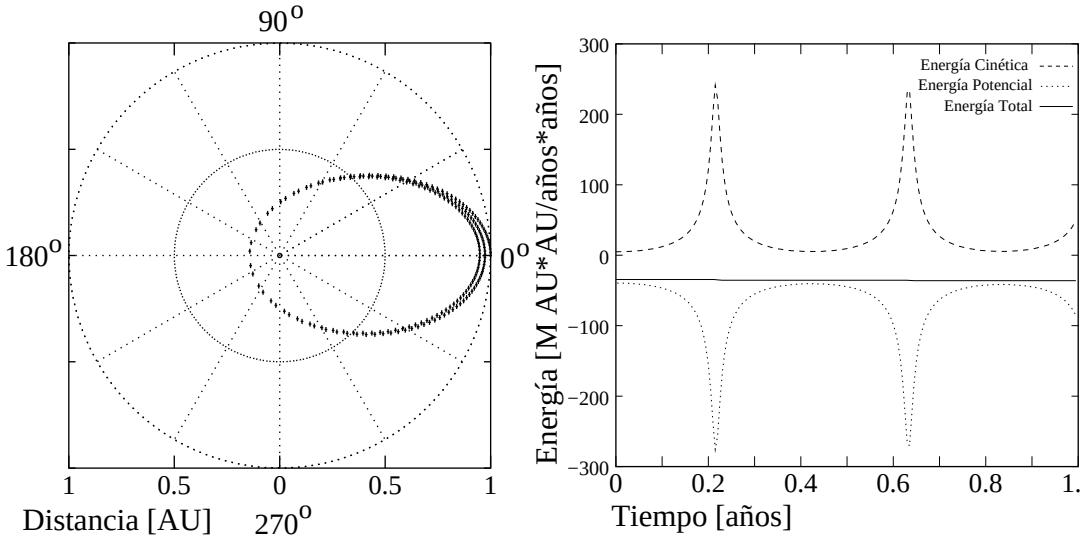


Figura 8.6: Gráfico de la trayectoria y la energía desde el programa `orbita` usando el método de Runge-Kutta. La distancia radial inicial es 1 [AU] y la velocidad tangencial inicial es π [AU/año]. El paso en el tiempo es $\tau = 0.005$ [años]; y 200 pasos son calculados. Comparemos con la figura 8.5.

8.2.4. Pasando funciones a funciones.

La función de Runge-Kutta `rk4` es muy simple, pero introduce un elemento de programación que no hemos usado antes. La función $\vec{f}(\vec{x}, t; \lambda)$ está introducida como un parámetro de entrada a `rk4`. Esto nos permite usar `rk4` para resolver diferentes problemas cambiando

simplemente la definición de $\vec{f}$ (como lo haremos en la última sección). Para el problema de Kepler, la función `gravrk` (tabla 8.4) define la ecuación de movimiento devolviendo dx/dt, ecuación (8.17).

En C++ el puntero a la función $\vec{f}(\vec{x}, t; \lambda)$ es pasado como parámetro a `rk4`. El programa `orbita` llama a `rk4` como

```
rk4( state, nState, time, tau, gravrk, param) ;
```

donde el vector de estado es $\vec{x} = [r_x, r_y, v_x, v_y]$. En el inicio del archivo, la función `gravrk` es declarada con el prototipo

```
void gravrk( double * x, double t, double param, double * deriv );
```

La primera línea de `rk4` es

```
void rk4(double * x, int nX, double t, double tau,
    void (*derivsRK) (double *, double, double, double *) , double param)
```

Cuando es llamado por `orbita`, esta función recibe un puntero a `gravrk` en la variable `derivsRK`. Dentro de `rk4`, la sentencia

```
(*derivsRK)( x, t, param, F1 ) ;
```

es equivalente a

```
gravrk( x, t, param, F1 ) ;
```

ya que `dervsRK` apunta a `gravrk`.

8.3. Métodos adaptativos

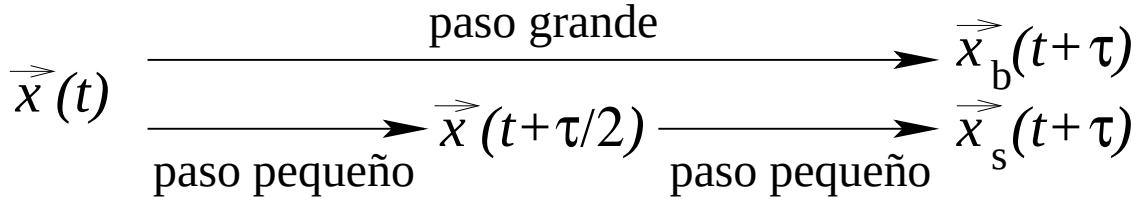
8.3.1. Programas con paso de tiempo adaptativo.

Ya que el método de Runge-Kutta de cuarto orden es más preciso (errores de truncamiento pequeños), hace un mejor trabajo dentro de una órbita altamemente elíptica. Aún para una distancia inicial al afelio de 1 [AU] y una velocidad inicial en el afelio de $\pi/2$ [AU/año] usando un paso tiempo tan pequeño como $\tau = 0.0005$ [años] ($\approx 4\frac{1}{2}$ [hrs]), la energía total varía sobre el 7 % por órbita. Si pensamos la física, llegamos a que realizar una integración con un paso pequeño es sólo necesaria cuando el cometa haga su acercamiento más próximo, punto en el cual su velocidad es máxima. Cualquier error pequeño en la trayectoria cuando rodea al Sol causa una gran desviación en la energía potencial.

La idea ahora es diseñar un programa que use un paso de tiempo pequeño cuando el cometa está cerca del Sol y pasos de tiempo grande cuando está lejos. Tal como está, normalmente tenemos sólo una idea apróximada de lo que τ pudiera ser; ahora tenemos que seleccionar un $\tau_{\text{mín}}$ y un $\tau_{\text{máx}}$ y una manera de intercambiar entre ellos. Si tenemos que hacer esto por prueba y error manual, podría ser peor que haciéndolo por la fuerza bruta calculando con un paso de tiempo pequeño toda la trayectoria. Idealmente, deseamos estar completamente liberados de tener que especificar un paso de tiempo. Deseamos tener una trayectoria calculada de la

misma posición inicial hasta algún tiempo final con la seguridad de que la solución es correcta a una precisión especificada

Los programas adaptativos continuamente monitorean la solución y modifican el paso de tiempo para asegurar que se mantenga la precisión especificada por el usuario. Esos programas pueden hacer algunos cálculos extras para optimizar la elección de τ, en muchos casos este trabajo extra vale la pena. Aquí está una manera para implementar esta idea: dado el estado actual $\vec{x}(t)$, el programa calcula $\vec{x}(t+\tau)$ como siempre, y luego repite el cálculo haciendolo en dos pasos, cada uno con paso de tiempo $\frac{\tau}{2}$. Visualmente, esto es



La diferencia entre las dos respuestas, $\vec{x}_b(t+\tau)$ y $\vec{x}_s(t+\tau)$, estima el error de truncamiento local. Si el error es tolerable, el valor calculado es aceptado y un valor mayor de τ es usado en la próxima iteración. Por otra parte, si el error es muy grande, la respuesta es rebotada, el paso de tiempo es reducido y el procedimiento es repetido hasta que se obtenga una respuesta aceptable. El error de truncamiento estimado para el actual paso de tiempo puede guiarnos en seleccionar en nuevo paso de tiempo para la próxima iteración.

8.3.2. Función adaptativa de Runge-Kutta.

Aquí mostramos cómo una iteración adaptativa puede ser implementada para un esquema de Runge-Kutta de cuarto orden: llamemos Δ al error de truncamiento; sabemos que $\Delta \propto \tau^5$ para un esquema Runge-Kutta de cuarto orden. Supongamos que el paso de tiempo actual τ_{ant} da un error de $\Delta_c = |\,\vec{x}_b - \vec{x}_s\,|$; esta es nuestra estimación para el error de truncamiento. Dado que deseamos que el error sea menor o igual que el error ideal especificado por el usuario, le llamamos Δ_i; luego, el nuevo paso de tiempo estimado es

$$\tau_{\text{est}} = \tau \left| \frac{\Delta_i}{\Delta_c} \right|^{1/5} . \tag{8.32}$$

Ya que esto es sólo una estimación, el nuevo paso de tiempo es $\tau_{\text{nuevo}} = S_1 \tau_{\text{est}}$, donde $S_1 < 1$. Esto nos hace sobreestimar el cambio cuando disminuimos τ y subestimar el cambio cuando lo aumentamos. Malogramos los esfuerzos computacionales cada vez que rebotamos una respuesta y necesitamos reducir el paso de tiempo, por lo tanto es mejor ajustar $\tau_{\text{nuevo}} < \tau_{\text{est}}$.

Podríamos poner un segundo factor de seguridad, $S_2 < 1$, para asegurarse que el programa no sea demasiado entusiasta en aumentar o disminuir precipitadamente el paso de tiempo. Con ambas precauciones, el nuevo paso de tiempo es

$$\tau_{\text{nuevo}} = \begin{cases} S_2 \tau_{\text{ant}} & \text{si } S_1 \tau_{\text{est}} > S_2 \tau_{\text{ant}} \\ \tau / S_2 & \text{si } S_1 \tau_{\text{est}} < \tau_{\text{ant}} / S_2 \\ S_1 \tau_{\text{est}} & \text{en otro caso} \end{cases} . \tag{8.33}$$

- *Entradas:* $\vec{x}(t)$, t, τ, Δ_i, $\vec{f}(\vec{x}, t; \lambda)$, y λ.
- *Salidas:* $x(t')$, t', τ_{nuevo}.
- Fijar las variables iniciales
- Iterar sobre el número deseado de intentos para satisfacer el error límite.
  - Tomar dos pequeños pasos de tiempo.
  - Tomar un único paso grande de tiempo.
  - Calcule el error de truncamiento estimado.
  - Estime el nuevo valor de τ (incluyendo factores de seguridad).
  - Si el error es aceptable, regresar los valores calculados.

  Mostrar un mensaje de error si el error límite nunca es satisfecho.

Cuadro 8.5: Bosquejo de la función `rka`, la cual evalúa un único paso usando un método adaptativo de Runge-Kutta de cuarto orden.

Esto obliga a asegurar que nuestro nueva estimación para τ nunca aumente o decrezca por más que un factor S_2. Por supuesto, este nuevo τ podría ser insuficientemente pequeño, y tendríamos que continuar reduciendo el paso de tiempo; pero al menos sabríamos que no ocurrirá de un modo incontrolado.

Este procedimiento no es "a prueba de balas", los errores de redondeo llegan a ser significativos en pasos de tiempos muy pequeños. Por esta razón la iteración adaptativa podría fallar para encontrar un paso de tiempo que de la precisión deseada. Debemos mantener esta limitación en mente cuando especifiquemos el error aceptable. Una función de Runge-Kutta adaptativa, llamada `rka`, es esbozada en la tabla 8.5. Note que los datos de entrada en la secuencia de llamada son los mismos que para `rk4`, excepto por la suma de Δ_i, el error ideal especificado. Las salidas del `rka` son el nuevo estado del sistema, $\vec{x}(t')$; el tiempo nuevo, t' y el nuevo paso de tiempo, τ nuevo, el cual podría ser usado la próxima vez que sea llamada la `rka`.

Usando el método de Runge-Kutta adaptativo, el programa `orbita` da los resultados en la figura 8.7 para una órbita altamente elíptica. Notemos que el programa toma muchos más pasos en el perihelio que en el afelio. Podemos comparar con los resultados usando el método de Runge-Kutta no adaptativo (figura 8.6) en el cual los pasos en el perihelio son ampliamente espaciados. Una gráfica de los pasos de tiempo versus la distancia radial (figura 8.8) muestra que τ varía casi tres órdenes de magnitud. Interesantemente esta gráfica revela una relación exponencial aproximada de la forma $\tau \propto \sqrt{r^3}$. Por supuesto esta dependencia nos recuerda la tercera ley de Kepler, ecuación (8.10). Esperamos alguna dispersión en los puntos, ya que nuestra rutina adaptada solamente estima el paso de tiempo óptimo.

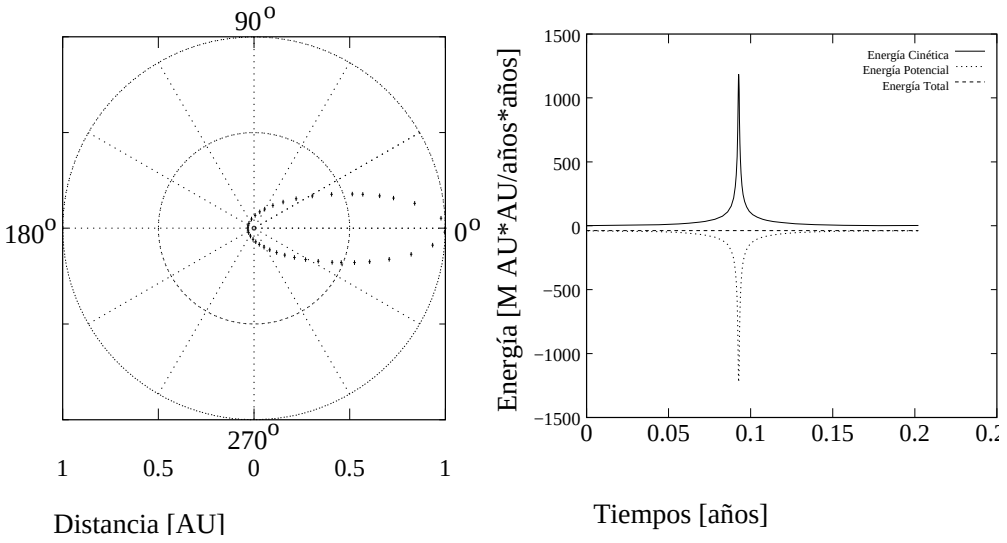


Figura 8.7: Gráfico de la trayectoria y la energía desde el programa `orbita` usando el método de Runge-Kutta adaptativo. La distancia radial inicial es 1 [AU] y la velocidad tangencial inicial es $\pi/2$ [AU/año]. El paso inicial en el tiempo es $\tau = 0.1$ [años]; y 40 pasos son calculados.

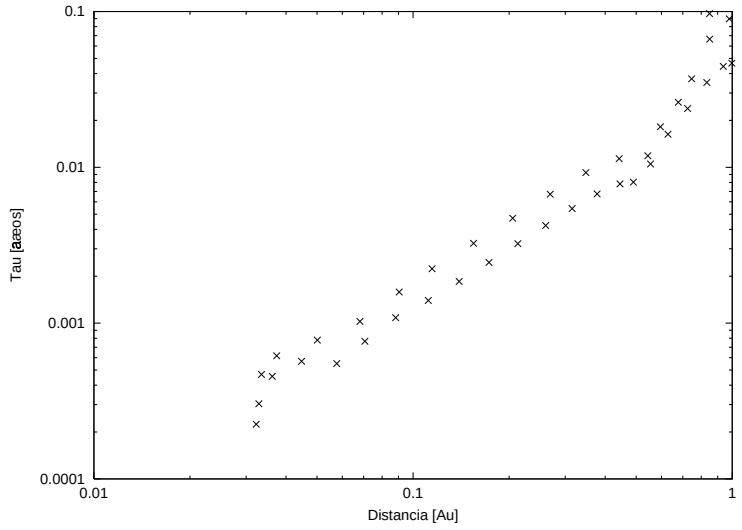


Figura 8.8: Paso de tiempo como función de la distancia radial desde el programa `orbita` usando el método de Runge-Kutta adaptativo. Los paramétros son los mismos de la figura 8.7.

8.4. Listados del programa.

8.4.1. `orbita.cc`

```
#include "NumMeth.h"
```

```
const double GM=4.0e0*M_PI*M_PI ;
const double masaCometa = 1.0e0 ;
const double adaptErr = 1.0e-3;

void rk4(double * x, int nX, double t, double tau,
         void(*derivsRK)(double *, double, double, double *),
         double param)
{
  double * F1=new double [nX] ;
  double * F2=new double [nX] ;
  double * F3=new double [nX] ;
  double * F4=new double [nX] ;
  double * xtemp=new double [nX] ;

  // Evaluemos F1=f(x,t)
  (*derivsRK) (x, t, param, F1) ;

  double half_tau = tau/2.0e0;
  double t_half = t+half_tau ;
  double t_full = t+tau ;

  // Evaluamos F2=f(x+tau*F1/2, t+tau/2)
  for(int i=0; i<nX; i++) xtemp[i]=x[i]+half_tau*F1[i] ;
  (*derivsRK) (xtemp, t_half, param, F2) ;

  // Evaluamos F3=f(x+tau*F2/2, t+tau/2)
  for(int i=0; i<nX; i++) xtemp[i]=x[i]+half_tau*F2[i] ;
  (*derivsRK) (xtemp, t_half, param, F3) ;

  // Evaluamos F4=f(x+tau*F3, t+tau)
  for(int i=0; i<nX; i++) xtemp[i]=x[i]+tau*F3[i] ;
  (*derivsRK) (xtemp, t_full, param, F4) ;

  // Retornamos x(t+tau)

  for(int i=0; i<nX; i++) x[i] += tau*(F1[i]+F4[i]+2.0e0*(F2[i]+F3[i]))/6.0e0 ;

  delete [] F1; delete [] F2; delete [] F3; delete [] F4;
  delete [] xtemp ;
}

void rka ( double * x, int nX, double  & t, double & tau, double erro,
           void(*derivsRK)(double *, double, double, double *),
           double param)
{
```

```
  double tSave = t ;
  double safe1 = 0.9, safe2 = 4.0 ; //factores de seguridad

  double * xSmall = new double[nX] ;
  double * xBig = new double[nX] ;
  int maxTray=100 ;
  for (int iTray=0; iTray<maxTray; iTray++) {
    // Tomemos dos peque{\~n}os pasos en el tiempo
    double half_tau = 0.5*tau ;
    for (int i =0; i < nX; i++) xSmall[i]=x[i] ;
    rk4( xSmall, nX, tSave, half_tau, derivsRK, param) ;
    t= tSave + half_tau ;
    rk4( xSmall, nX, t, half_tau, derivsRK, param) ;
   // Tomemos un solo tiempo grande
    for (int i =0; i < nX; i++) xBig[i]=x[i] ;
    rk4( xBig, nX, tSave, tau, derivsRK, param) ;

    // Calculemos el error de truncamiento estimado
    double erroRatio = 0.0e0  ;
    double eps = 1.0e-16 ;
    for (int i = 0 ; i < nX; i++) {
      double scale = erro * (fabs(xSmall[i]) + fabs(xBig[i]))/2.0e0 ;
      double xDiff = xSmall[i] - xBig[i] ;
      double ratio = fabs(xDiff)/(scale+eps) ;
      erroRatio = (erroRatio > ratio ) ? erroRatio:ratio ;
    }
    // Estimamos el nuevo valor de tau (incluyendo factores de seguridad)
    double tau_old= tau ;
    tau = safe1*tau_old*pow(erroRatio, -0.20) ;
    tau = (tau > tau_old/safe2) ? tau:tau_old/safe2 ;
    tau = (tau < safe2*tau_old) ? tau:safe2*tau_old ;

    // Si el error es aceptable regrese los valores computados
    if ( erroRatio < 1 ) {
      for (int i =0 ; i < nX; i++) x[i]=xSmall[i] ;
      return ;
    }
  }
  cout << "Error: Runge-Kutta adaptativo fallo" << endl ;
  exit(-1) ;
}


void gravrk( double * x, double t, double param, double * deriv)
{
```

```
  double gm=param ;
  double rX=x[0], rY=x[1];
  double vX=x[2], vY=x[3] ;
  double mod_r= sqrt(rX*rX+rY*rY) ;
  double aX= -gm*rX/(mod_r*mod_r*mod_r) ;
  double aY= -gm*rY/(mod_r*mod_r*mod_r) ;

  // Retorna la derivada

  deriv[0] = vX;
  deriv[1] = vY;
  deriv[2] = aX;
  deriv[3] = aY;
}

int main()
{
  ofstream salida0 ("Orbita.txt") ;
  ofstream salidaE ("Energia.txt") ;
  ofstream salidaT ("Tau.txt") ;

  double r0 ;
  cout << "Ingrese la distancia radial inicial [AU]: " ;
  cin >> r0 ;
  double vT ;
  cout << "Ingrese la velocidad tangencial inicial [AU/a{\~n}os]: " ;
  cin >> vT ;
  double x0=r0 ;
  double y0=0.0e0;
  double vx0=0.0e0 ;
  double vy0=vT;
  //
  // Suponemos angulo inicial nulo
  //
  int metodo = 0 ;
  while( metodo < 1|| metodo > 4 ) {
    cout << "Ingrese el m{\'e}todo num{\'e}rico a usar :" << endl ;
    cout << "\t Metodo de Euler \t\t\t[1]" << endl;
    cout << "\t Metodo de Euler-Cromer \t\t[2]" << endl;
    cout << "\t Metodo de Runge-Kutta 4 orden \t\t[3]" << endl;
    cout << "\t Metodo de Runge-Kutta adaptativo \t[4]" << endl;
    cout << "elija: " ;
    cin >> metodo ;
  }
  double tau ;
```

```
cout << "Ingrese paso en el tiempo: " ;
cin >> tau ;
int numPasos ;
cout << "Ingrese el numero de pasos: " ;
cin >> numPasos ;

double param=GM ;
const int dimX= 4;
double * x = new double[dimX] ;

double xN= x0;
double yN= y0;
double vxN=vx0;
double vyN=vy0;
double vxNp1, vyNp1, xNp1, yNp1;
double tiempo = 0.0e0 ;

for(int pasos=0; pasos < numPasos; pasos++) {
  double r =sqrt(xN*xN+yN*yN) ;
  double v2 =vxN*vxN+vyN*vyN ;
  double theta= atan2(yN, xN) ;
  salida0 << theta << " " << r << endl ;

  double Ep = -GM*masaCometa/r ;
  double Ek = masaCometa*v2/2.0e0 ;
  double ET= Ep+Ek ;

  salidaE<< tiempo << " " << Ek<< " " << Ep<<" " << ET << endl ;

  double modr3=pow(xN*xN+yN*yN, 3.0e0/2.0e0) ;
  double axN= -GM*xN/modr3 ;
  double ayN= -GM*yN/modr3 ;
  switch( metodo ) {
  case 1: {                    // Euler
    vxNp1=vxN+tau*axN ;
    vyNp1=vyN+tau*ayN ;
    xNp1= xN+tau* vxN ;
    yNp1= yN+tau* vyN ;
    tiempo += tau ;
  }
  break ;
  case 2: {                    // Euler-Cromer
    vxNp1=vxN+tau*axN ;
    vyNp1=vyN+tau*ayN ;
    xNp1= xN+tau* vxNp1 ;
```

```
      yNp1= yN+tau* vyNp1 ;
      tiempo += tau ;
    }
    break ;
    case 3: {                    // Runge-Kutta 4to Orden
      x[0] = xN;
      x[1] = yN;
      x[2] = vxN;
      x[3] = vyN;
      rk4( x, dimX, tiempo, tau, gravrk, param);
      xNp1=x[0] ;
      yNp1=x[1];
      vxNp1=x[2];
      vyNp1=x[3];
      tiempo += tau ;
    }
    break ;
    case 4: {
      x[0] = xN;
      x[1] = yN;
      x[2] = vxN;
      x[3] = vyN;
      rka( x, dimX, tiempo, tau, adaptErr, gravrk, param);
      double distancia = sqrt( x[0]*x[0]+x[1]*x[1]) ;
      salidaT<< distancia << " " <<tau << endl ;
      xNp1=x[0] ;
      yNp1=x[1];
      vxNp1=x[2];
      vyNp1=x[3];
    }
    }
    xN=xNp1 ;
    yN=yNp1 ;
    vxN=vxNp1 ;
    vyN=vyNp1 ;
  }
  salidaO.close() ;
  salidaE.close() ;
  salidaT.close() ;
  delete [] x;
  return 0;
}
```

Capítulo 9

Resolviendo sistemas de ecuaciones.

versión final 2.20-030407[1]

En este capítulo aprenderemos a cómo resolver sistemas de ecuaciones de ambos tipos, lineal y no lineal. Ya conocemos los algoritmos básicos para los sistemas lineales: eliminar variables hasta que tengamos una sola ecuación con una sola incógnita. Para sistemas no-lineales desarrollamos un esquema iterativo que en cada paso resuelve una versión linealizada de estas ecuaciones. Para mantener la continuidad, la discusión será motivada por el cálculo del estado estacionario, un importante tema en ecuaciones diferenciales ordinarias.

9.1. Sistemas de ecuaciones lineales.

9.1.1. Estado estacionario de EDO.

En el capítulo pasado, nosotros vimos cómo resolver ecuaciones diferenciales ordinarias de la forma

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}, t) \ , \tag{9.1}$$

donde $\vec{x} = [x_1, x_2, \ldots, x_N]$. Dada una condición inicial para las N variables $x_i(t) = 0$, nosotros podemos calcular la serie de tiempo $x_i(t)$ por una variedad de métodos (*e.g.* Runge-Kutta).

Los ejemplos que hemos estudiado hasta aquí han sido aquellos conocidos como sistemas autónomos donde $\vec{f}(\vec{x}, t) = \vec{f}(\vec{x})$, esto es, $\vec{f}$ no depende explícitamente del tiempo. Para sistemas autónomos, a menudo existe una importante clase de condiciones iniciales para las cuales $x_i(t) = x_i(0)$ para todo i y t. Estos puntos, en el espacio N-dimensional de nuestras variables, son llamados *estado estacionario*. Si nosotros partimos de un estado estacionario nos quedamos ahí para siempre. Localizar los estados estacionarios para ecuaciones diferenciales ordinarias es importante, ya que ellos son usados en el análisis de bifurcaciones.[2]

Es fácil ver que $\vec{x}^* = [x_1^*, x_2^*, \ldots, x_N^*]$ es un estado estacionario si y sólo si

$$\vec{f}(\vec{x}^*) = 0 \ , \tag{9.2}$$

o

$$f_i(x_1^*, x_2^*, \ldots, x_N^*) = 0 \ , \quad \text{para todo } i, \tag{9.3}$$

[1] Este capítulo está basado en el cuarto capítulo del libro: *Numerical Methods for Physics, second edition* de Alejandro L. Garcia, editorial Prentice Hall.

[2] R. Seydel, *From Equilibrium to Chaos, Practical Bifurcation and Stability Analysis* (New York: Elsevier, 1988).

ya que esto implica que $d\vec{x}^*/dt = 0$. Localizar los estados estacionarios se reduce al problema de resolver N ecuaciones con N incógnitas ${x_i}^*$. Este problema es también llamado "encontrar las raices de $f(x)$".

Como un ejemplo, consideremos el péndulo simple; el estado está descrito por el ángulo θ y la velocidad angular ω. Los estados estacionarios se encuentran al resolver las ecuaciones no lineales

$$-\frac{g}{L}\,\mathrm{sen}\,\theta^* = 0\ , \quad \omega^* = 0\ . \tag{9.4}$$

Las raices son $\theta^* = 0, \pm\pi, \pm 2\pi, \ldots$ y $\omega^* = 0$. Por supuesto que no todos los sistemas son tan fáciles de resolver.

Debería ser claro que encontrar raices tiene muchas más aplicaciones que calcular los estados estacionario. En este capítulo consideraremos una variedad de maneras para resolver ambos sistemas lineales y sistemas no lineales. En esta sección y en la próxima consideraremos sistemas lineales, dejando los sistemas no lineales para la parte final del capítulo.

9.1.2. Eliminación Gaussiana.

El problema de resolver $f_i(\{x_j\}) = 0$ se divide en dos importantes clases. En esta sección consideramos el caso fácil cuando $f_i(\{x_j\})$ es una función lineal. El problema en ese caso se reduce a resolver un sistema de N ecuaciones con N incógnitas.

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N - b_1 = 0 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N - b_2 = 0 \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \\ a_{N1}x_1 + a_{N2}x_2 + \ldots + a_{NN}x_N - b_N = 0\ , \end{array} \tag{9.5}$$

o en forma matricial

$$\check{A}\,\vec{x} - \vec{b} = 0\ , \tag{9.6}$$

donde

$$\check{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots \\ a_{21} & a_{22} & \ldots \\ \vdots & \vdots & \ddots \end{bmatrix}\ , \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}\ , \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \end{bmatrix}\ . \tag{9.7}$$

Sabemos cómo resolver este conjunto de ecuaciones. Combinando las ecuaciones para eliminar variables hasta que tengamos una sola ecuación con una sola incógnita. Hagamos un ejemplo simple para revisar cómo el procedimiento trabaja. Tomemos las ecuaciones

$$\begin{array}{rcrcrcl} x_1 & + & x_2 & + & x_3 & = & 6\ , \\ -x_1 & + & 2x_2 & & & = & 3\ , \\ 2x_1 & & & + & x_3 & = & 5\ . \end{array} \tag{9.8}$$

Deseamos eliminar x_1 de la segunda y de la tercera ecuación. Para llevar a cabo esto, primero sumamos la primera y la segunda y después restamos dos veces la primera a la tercera. Esto nos da

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & x_3 & = & 6\ , \\ & & 3x_2 & + & x_3 & = & 9\ , \\ & & -2x_2 & - & x_3 & = & -7\ . \end{array} \tag{9.9}$$

Ahora, eliminamos x_2 de la última ecuación multiplicando la segunda ecuación por $-\frac{2}{3}$ y la restamos de la tercera, dándonos

$$\begin{aligned} x_1 + x_2 + x_3 &= 6\,, \\ 3x_2 + x_3 &= 9\,, \\ -\tfrac{1}{3}x_3 &= -1\,. \end{aligned} \tag{9.10}$$

Este procedimiento es llamado *eliminación hacia adelante*. Si se tienen N ecuaciones, eliminamos x_1 de las ecuaciones 2 hasta la N, luego eliminamos x_1 y x_2 de las ecuaciones 3 hasta la N, y así sucesivamente. La última ecuación sólo contendrá la variable x_N.

Volviendo al ejemplo, es ahora trivial resolver la tercera ecuación resultando $x_3 = 3$. Podemos ahora sustituir este valor en la segunda ecuación obteniendo $3x_2 + 3 = 9$, tal que $x_2 = 2$. Finalmente, introduciendo los valores de x_2 y x_3 en la primera ecuación obtenemos $x_1 = 1$. Este segundo procedimiento es llamado *sustitución hacia atrás*. Debería ser claro cómo esto trabaja con sistemas grandes de ecuaciones. Usando la última ecuación obtenemos x_N, éste es usado en la penúltima ecuación para obtener x_{N-1} y así seguimos.

Este método de resolver sistemas de ecuaciones lineales por eliminación hacia adelante y luego sustitución hacia atrás es llamado *eliminación Gaussiana*.[3] Esta es una secuencia rutinaria de pasos que es simple para un computador realizar sistemáticamente. Para N ecuaciones con N incógnitas, el tiempo de computación para eliminación Gaussiana va como N^3. Afortunadamente, si el sistema es esparcido o ralo[4] (la mayoría de los coeficientes son cero), el tiempo de cálculo puede ser considerablemente reducido.

9.1.3. Pivoteando.

La eliminación Gaussiana es un procedimiento simple, sin embargo, se debe estar conciente de sus riesgos. Para ilustrar la primera fuente de problemas, consideremos el conjunto de ecuaciones

$$\begin{aligned} \epsilon x_1 + x_2 + x_3 &= 5\,, \\ x_1 + x_2 &= 3\,, \\ x_1 + x_3 &= 4\,. \end{aligned} \tag{9.11}$$

En el límite $\epsilon \to 0$ la solución es $x_1 = 1$, $x_2 = 2$, $x_3 = 3$. Para estas ecuaciones, el primer paso de la eliminación hacia adelante podría partir por multiplicar la primera ecuación por $(1/\epsilon)$ y sustrayéndola de la segunda y tercera ecuaciones, lo que da

$$\begin{aligned} \epsilon x_1 + x_2 + x_3 &= 5\,, \\ (1-1/\epsilon)x_2 - (1/\epsilon)x_3 &= 3-5/\epsilon\,, \\ -(1/\epsilon)x_2 + (1-1/\epsilon)x_3 &= 4-5/\epsilon\,. \end{aligned} \tag{9.12}$$

Por supuesto, si $\epsilon = 0$ tenemos grandes problemas, ya que el factor $1/\epsilon$ estalla. Aún si $\epsilon \neq 0$, pero es pequeño, vamos a tener serios problemas de redondeo. Supongamos que $1/\epsilon$ es tan

[3] G.E. Forsythe and C.B. Moler, *Computer Solution of Linear Algebraic System* (Upper Saddle River, N.J.: Prentice-Hall, 1967).

[4] *sparse*

grande que $(C - 1/\epsilon) \to -1/\epsilon$, donde C es del orden de la unidad. Nuestras ecuaciones, después del redondeo, llegan a ser

$$\begin{aligned} \epsilon x_1 + x_2 + x_3 &= 5\,, \\ -(1/\epsilon)x_2 - (1/\epsilon)x_3 &= -5/\epsilon\,, \\ -(1/\epsilon)x_2 - (1/\epsilon)x_3 &= -5/\epsilon\,. \end{aligned} \tag{9.13}$$

En este punto está claro que no podemos proceder ya que la segunda y la tercera ecuación en (9.13) son ahora idénticas; no tenemos más que tres ecuaciones independientes. El próximo paso de eliminación sería transformar la tercera ecuación en (4.13) en la tautología $0 = 0$.

Afortunadamente, hay una solución simple: intercambiar el orden de las ecuaciones antes de realizar la eliminación. Cambiando las ecuaciones primera y segunda en (9.11),

$$\begin{aligned} x_1 + x_2 &= 3\,, \\ \epsilon x_1 + x_2 + x_3 &= 5\,, \\ x_1 + x_3 &= 4\,. \end{aligned} \tag{9.14}$$

El primer paso de la eliminación hacia adelante nos da las ecuaciones

$$\begin{aligned} x_1 + x_2 &= 3\,, \\ (1-\epsilon)x_2 + x_3 &= 5 - 3\epsilon\,, \\ -x_2 + x_3 &= 4 - 3\,. \end{aligned} \tag{9.15}$$

El redondeo elimina los términos proporcionales a ϵ, dando

$$\begin{aligned} x_1 + x_2 &= 3\,, \\ x_2 + x_3 &= 5\,, \\ -x_2 + x_3 &= 1\,. \end{aligned} \tag{9.16}$$

El segundo paso de eliminación hacia adelante elimina x_2 de la tercera ecuación en (9.16) usando la segunda ecuación,

$$\begin{aligned} x_1 + x_2 &= 3\,, \\ x_2 + x_3 &= 5\,, \\ 2x_3 &= 6\,. \end{aligned} \tag{9.17}$$

Se puede comprobar fácilmente que la sustitución hacia atrás da $x_1 = 1$, $x_2 = 2$, y $x_3 = 3$, la cual es la respuesta correcta en el límite $\epsilon \to 0$.

Los algoritmos que reordenan las ecuaciones cuando ellas sitúan elementos pequeños en la diagonal son llamados *pivotes*. A menudo si todos los elementos de una matriz son inicialmente de una magnitud comparable, el procedimiento de eliminación hacia adelante podría producir pequeños elementos sobre la diagonal principal. El precio de pivotear es sólo un par de líneas extras en el programa, pero es esencial usar pivoteo para todas, no sólo para las matrices más pequeñas. Aún con el pivoteo, no podemos garantizar estar a salvo de problemas de redondeo si se trata de matrices muy grandes.

9.1.4. Determinantes.

Es fácil obtener el determinante de una matriz usando la eliminación Gaussiana. Después de completar la eliminación hacia adelante, uno simplemente calcula el producto de los coeficientes de los elementos diagonales. Tomamos nuestro ejemplo original, ecuación (9.8). La matriz es

$$\check{A} = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix} . \tag{9.18}$$

Completada la eliminación hacia adelante, ecuación (9.10), el producto de los coeficientes de los elementos diagonales es $(1)(3)(-\frac{1}{3}) = -1$, el cual, usted puede comprobar, es el determinante de $\check{A}$. Este método es sutílmente más complicado cuando se usa el pivoteo. Si el número de pivoteos es impar, el determinante es el producto negativo de los coeficientes de los elementos de la diagonal. Ahora debería ser obvio que la regla de Cramer es computacionalmente una manera ineficiente para resolver el conjunto de ecuaciones lineales.

9.1.5. Eliminación Gaussiana en Octave.

No necesitamos escribir un programa en Octave para realizar la eliminación Gaussiana con pivoteo. En vez de esto nosotros podemos usar las capacidades de manipulación de matrices que viene con Octave. En Octave, la eliminación Gaussiana es una rutina primitiva, tal como las funciones seno o raíz cuadrada. Como con cualquier rutina "enlatada", usted debería entender, en general, como trabaja y reconocer posibles problemas, especialmente los computacionales (*e.g.*, ¿poner `sqrt(-1)` retorna un número imaginario o un error?).

Octave implementa la eliminación de Gauss usando los operadores *slash* `/` y *backslash* `\`. El sistema de ecuaciones lineales $\vec{x}\,\check{A} = \vec{b}$ donde $\vec{x}$ y $\vec{b}$ son vectores fila, se resuelve usando el operador *slash* como $\vec{x} = \vec{b}/\check{A}$. El sistema de ecuaciones lineales $\check{A}\,\vec{x} = \vec{b}$ donde $\vec{x}$ y $\vec{b}$ son vectores columna, se resuelve usando el operador *backslash* como $\vec{x} = \vec{b}\backslash\check{A}$. Como un ejemplo, tomemos la ecuación (9.11) con $\epsilon = 0$, escrito en forma matricial

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 4 \end{bmatrix} \tag{9.19}$$

Usando Octave en forma interactiva la solución es ilustrada a continuación:

```
octave> A=[0 1 1; 1 1 0; 1 0 1];
octave> b=[5;3;4];
octave> x=A\b;
octave> disp(x);
  1
  2
  3
```

Claramente, Octave usa pivoteo en este caso. Los operadores *slash* y *backslash* pueden ser usados de la misma manera en programas. El comando en Octave para el determinante de una matriz es `det(A)`.

9.1.6. Eliminación Gaussiana con C++ de objetos matriciales.

El uso de arreglos multidimensionales en C++ no es del todo comodo. Las maneras usuales de declarar un arreglo de $M \times N$ de números con punto flotante en doble precisión son:

```
const int M=3, N=3;
double A[M][N] ;
```

para asignación estática de memoria y

```
int M, N
. . .  // Se fijan valores a M y N
double * * A = new double * [M] ; // asignacion de un vector de punteros
for(int i=0;i<M;i++) {
     A[i] = new double [N] ; // asignacion de memoria para cada fila
}
```

para un reserva dinámica (no olvide desasignar cada fila con un `delete[]`). Una asignación estática es simple pero rígida, ya que las dimensiones son constantes fijas. Cuando el arreglo estático es pasado a una función, esta función debería declarar algún arreglo con el mismo número de columnas de acuerdo al arreglo a ser indexado apropiadamente. Una asignación dinámica es flexible pero es difícil de manejar, aunque los detalles pueden ser escondidos dentro de funciones separadas. Accesar elementos fuera de los contornos del arreglo es un error de programación común (*bug*) el cual es difícil de rastrear con arreglos dinámicos.

C++ nos permite corregir estas deficiencias creando nuestros propios tipos de variables. Estas variables definidas por el usuario son llamadas *clases de objetos*. De aquí en adelante usaremos la clase `Matrix` para declarar arreglos de una o dos dimensiones de números de punto flotante. Esta clase está enteramente declarada dentro del archivo de cabecera `Matrix.h` e implementada dentro de `Matrix.cc`. Algunos ejemplos de declaración de objetos de `Matrix` son

```
int M=3, N=2 ;
Matrix A(M,N), b(N), x(3) ;
```

Aunque esto se parece a una asignación de arreglo estático, note que las dimensiones no son constantes fijas. Tanto vectores unidimensionales como matrices bidimensionales pueden ser declarados; las anteriores son tratadas como matrices de una sola columna. Los valores en estas variables pueden ser fijados por la declaración de asignamiento

```
A(0,0)=0;          A(0,1)=1;          A(0,2)=1;
A(1,0)=1;          A(1,1)=1;          A(1,2)=0;
A(2,0)=1;          A(2,1)=0;          A(2,2)=1;
b(1)=5;            b(2)=3;            b(3)=4.
```

El formato para el objeto `Matrix` es $A(i, j)$ en vez de $A[i][j]$, para distinguirlos de los arreglos C++ convencionales.

Un objeto `Matrix` conoce sus dimensiones, y usted puede obtenerlas usando las funciones miembros `nRow()` y `nCol()`. Por ejemplo:

- *Entradas:* $\check{A}$, $\vec{b}$.
- *Salidas:* $\vec{x}$, $\det \check{A}$.
- Fija el factor de escala, $s_i = \max_j(|\, A_{i,j}\,|)$ para cada fila.
- Itera sobre las filas $k = 1, \ldots, (N-1)$.
  - Selecciona la columna a pivotear a partir de $\max_j(|\, A_{i,j}\,|)/s_i$.
  - Realiza pivoteos usando la lista de índice de columnas.
  - Realiza la eliminación hacia adelante.
- Calcula el determinante $\det \check{A}$, como producto de los elementos de la diagonal.
- Realiza la sustitución hacia atrás.

Cuadro 9.1: Bosquejo de la función `ge`, la cual resuelve un sistema de ecuaciones lineales $\check{A}\,\vec{x} = \vec{b}$ por eliminación Gaussiana. La función también devuelve el determinante de $\check{A}$.

```
int m = A.nRow(), n = A.nCol() ;
```

asigna `m` y `n` a las dimensiones de `A`. La verificación de los contornos se realiza cada vez que un objeto `Matrix` es indexado. Las líneas

```
Matrix newA(3, 7) ;  // Una matriz no cuadrada de 3 por 7
newA(4,5) = 0;        // Fuera de los limites
```

produce el mensage de error: `Indice de fila fuera de los limites` cuando el programa es corrido. Esto significa que al menos uno de los índices i no satisface la condición $0 < i \leq N$. Los objetos `Matrix` automáticamente liberan la memoria que les fue asignada cuando ellos exceden su alcance (*scope*), por lo tanto no es necesario invocar `delete`.

Para asignar todos los elementos de un objeto `Matrix` a un valor dado usamos la función miembro `set(double x )`, por ejemplo `A.set(1.0)`. Una matriz entera puede ser asignada a otra del mismo tamaño (*e.g.*, `Matrix C(3,3); C=A;`). Sin embargo, a diferencia de las matrices de Octave, operaciones aritméticas como `2*A` no están aún implementadas. Operaciones de este tipo deberían ser llevadas a cabo elemento por elemento, típicamente usando ciclos `for`.

La rutina de eliminación Gaussiana `ge`, la cual usa los objetos tipo `Matrix`, es descrita en la tabla 9.1. Declarando y asignando la matriz `A` y los vectores `b` y `x` como arriba, un programa puede llamar esta rutina como

```
double determ = ge(A, b, x) ; // Eliminacion Gaussiana
cout << x(1) << ", " << x(2) << ", " << x(3) << endl ;
cout   << "Determinante = " << determ << endl ;
```

para resolver el sistema lineal $\check{A}\,\vec{x} = \vec{b}$ y calcular el determinante de $\check{A}$.

9.2. Matriz inversa.

9.2.1. Matriz inversa y eliminación Gaussiana.

En la sección previa hemos revisado cómo resolver un conjunto de ecuaciones simultáneas por eliminación de Gauss. Sin embargo, si usted está familiarizado con el álgebra lineal, probablemente podría escribir la solución de

$$\check{A}\,\vec{x} = \vec{b}\ , \tag{9.20}$$

como

$$\vec{x} = \check{A}^{-1}\,\vec{b}\ , \tag{9.21}$$

donde $\check{A}^{-1}$ es la matriz inversa de $\check{A}$. No nos debería sorprender que el cálculo de la matriz inversa esté relacionada al algoritmo para resolver un conjunto de ecuaciones lineales. Usualmente la inversa de una matriz es calculada por aplicaciones repetidas de eliminaciones de Gauss (o una variante de descomposición llamada LU).

La inversa de una matriz está definida por la ecuación

$$\check{A}\,\check{A}^{-1} = \check{I}\ , \tag{9.22}$$

donde $\check{I}$ es la matriz identidad

$$\check{I} = \begin{bmatrix} 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}\ . \tag{9.23}$$

Definiendo los vectores columna

$$\hat{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix}\ , \quad \hat{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix}\ , \quad \dots\ , \quad \hat{e}_N = \begin{bmatrix} \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}\ , \tag{9.24}$$

podríamos escribir la matriz identidad como una vector fila de los vectores columna

$$\check{I} = \begin{bmatrix} \hat{e}_1\ \hat{e}_2\ \dots\ \hat{e}_N \end{bmatrix}\ . \tag{9.25}$$

Si resolvemos el conjunto de ecuaciones lineales,

$$\check{A}\,\vec{x}_1 = \hat{e}_1\ . \tag{9.26}$$

El vector solución $\vec{x}_1$ es la primera columna de la inversa $\check{A}^{-1}$. Si procedemos de esta manera con los otros $\hat{e}$ calcularemos todas las columnas de $\check{A}^{-1}$. En otras palabras, nuestra ecuación para la matriz inversa $\check{A}\check{A}^{-1} = \check{I}$ es resuelta escribiéndola como

$$\check{A}\ \begin{bmatrix} \hat{x}_1\ \hat{x}_2\ \dots\ \hat{x}_N \end{bmatrix} = \begin{bmatrix} \hat{e}_1\ \hat{e}_2\ \dots\ \hat{e}_N \end{bmatrix}\ . \tag{9.27}$$

- *Entradas:* $\check{A}$.
- *Salidas:* $\check{A}^{-1}$, $\det \check{A}$.
- Matriz $\check{b}$ es inicializada a la matriz identidad.
- Fija el factor de escala, $s_i = \max_j(|\, A_{i,j} \,|)$ para cada fila.
- Itera sobre las filas $k = 1, \ldots, (N-1)$.
  - Selecciona la columna a pivotear a partir de $\max_j(|\, A_{i,j} \,|)/s_i$.
  - Realiza pivoteos usando la lista de índice de columnas.
  - Realiza la eliminación hacia adelante.
- Calcula el determinante $\det\check{A}$, como producto de los elementos de la diagonal.
- Realiza la sustitución hacia atrás.

Cuadro 9.2: Bosquejo de la función `inv`, la cual calcula el inverso de una matriz y su determinante.

Después de calcular los $\vec{x}$, construimos $\check{A}^{-1}$ como

$$\check{A}^{-1} \left[\hat{x}_1 \; \hat{x}_2 \; \ldots \; \hat{x}_N\right] = \begin{bmatrix} (\vec{x}_1)_1 & (\vec{x}_2)_1 & \ldots & (\vec{x}_N)_1 \\ (\vec{x}_1)_2 & (\vec{x}_2)_2 & \ldots & (\vec{x}_N)_2 \\ \vdots & \vdots & \ddots & \vdots \\ (\vec{x}_1)_N & (\vec{x}_2)_N & \ldots & (\vec{x}_N)_N \end{bmatrix} \tag{9.28}$$

donde$(\vec{x}_i)_j$ es el elemento j de $\vec{x}_i$.

La tabla 9.2 muestra la función `inv` para calcular la inversa de una matriz. En Octave, `inv(A)` es una función internamente construida que regresa a la matriz inversa de `A` . Es posible resolver un sistema de ecuaciones lineales usando la matriz inversa, pero hacer esto es usualmente sobrepasarse. Una excepción podría ser el caso donde deseamos resolver un número de problemas similares en el cual la matriz $\check{A}$ es fija pero el vector $\vec{b}$ toma muchos valores diferentes. Finalmente, aquí hay una fórmula manual para tener presente: la inversa de una matriz 2×2 es

$$\check{A}^{-1} = \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} . \tag{9.29}$$

Para matrices mayores las fórmulas rápidamente llegan a ser muy desordenadas.

9.2.2. Matrices singulares y patológicas.

Antes que regresemos a la Física, discutamos otro posible riesgo en resolver sistemas de ecuaciones lineales. Consideremos las ecuaciones

$$\begin{array}{rcrcl} x_1 & + & x_2 & = & 1 \\ 2x_1 & + & 2x_2 & = & 2 \end{array} . \tag{9.30}$$

Note que realmente no tenemos dos ecuaciones independientes, ya que la segunda es sólo el doble de la primera. Estas ecuaciones no tienen una solución única. Si tratamos de hacer una eliminación hacia adelante, obtenemos

$$\begin{array}{rcrcl} x_1 & + & x_2 & = & 1 \\ & & 0 & = & 0 \end{array} , \tag{9.31}$$

y no se puede hacer nada más.

Otra manera de mirar este problema es ver que la matriz

$$\check{A} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \tag{9.32}$$

no tiene inversa. Una matriz sin inversa se dice que es *singular*. Una matriz singular también tiene un determinante nulo. Las matrices singulares no siempre son evidentes. ¿Podría adivinar si esta matriz

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

es singular?

Aquí está lo que pasa en Octave cuando tratamos de resolver (9.30)

```
octave:1> A=[1 1; 2 2];
octave:2> b=[1; 2];
octave:3> x=A\b;
warning: matrix singular to machine precision, rcond = 0
```

Dependiendo de su compilador, la rutina `inv` de C++ probablemente retorne infinito o arroje un mensage de error. En algunos casos la rutina calcula una inversa para una matriz singular, pero naturalmente la respuesta es espuria.

Algunas veces la matriz no es singular pero está tan cerca de serlo que los errores de redondeo podrían "empujarla al abismo". Un ejemplo trivial sería

$$\begin{bmatrix} 1+\epsilon & 1 \\ 2 & 2 \end{bmatrix} , \tag{9.33}$$

donde ϵ es un número muy pequeño. A una matriz se le dice *patológica* cuando está muy cerca de ser singular.

Si usted sospecha que está tratrando con una matriz patológica cuando resuelve $\check{A}\vec{x} = \vec{b}$, entonces calcule el error absoluto, $\left| \check{A}\vec{x} - \vec{b} \right| / \left| \vec{b} \right|$ para comprobar si $\vec{x}$ es una solución precisa.

Formalmente, la condición numérica está definida como la "distancia" normalizada entre una matriz y la matriz singular más cercana[5]. Hay una variedad de maneras para definir la distancia, dependiendo del tipo de norma usada. La función de Octave `cond(A)` regresa la condición numérica de una matriz `A`. Una matriz con una gran condición numérica es patológica. Un pequeño determinante puede algunas veces advertirnos que la matriz podría ser patológica, pero la condición numérica es el criterio real.[6]

9.2.3. Osciladores armónicos acoplados.

En el comienzo de este capítulo, discutimos el problema de encontrar los estados estacionarios de ecuaciones diferenciales ordinarias. Un ejemplo canónico de un sistema con interacciones lineales es el caso de osciladores armónicos acoplados. Consideremos el sistema mostrado en la figura 9.1; las constantes de resorte son $k_1, \ldots, k_4$. La posición de los bloques, relativas a la pared izquierda, son x_1, x_2 y x_3. La distancia entre las paredes es L_P, y las longitudes naturales de los resortes son $L_1, \ldots, L_4$. Los bloques son de ancho despreciable.

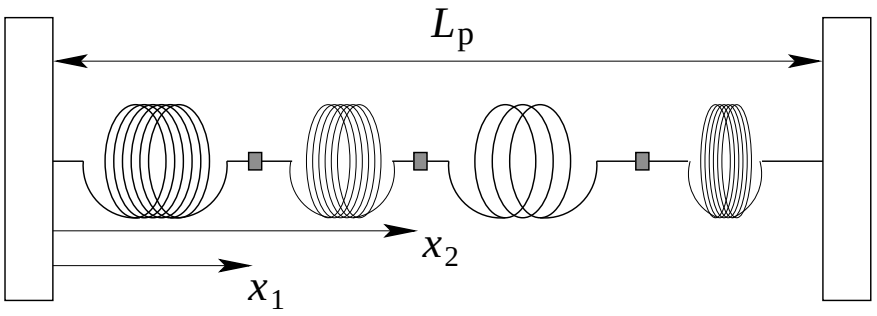


Figura 9.1: Sistema de bloques acoplados por resortes anclados entre paredes.

La ecuación de movimiento para el bloque i es

$$\frac{dx_i}{dt} = v_i \ , \qquad \frac{dv_i}{dt} = \frac{1}{m_i} F_i \ , \tag{9.34}$$

donde F_i es la fuerza neta sobre el bloque i. En el estado estacionario, las velocidades v_i, son nulas y las fuerzas netas, F_i son cero. Esto es justo el caso de equilibrio estático. Nuestro trabajo ahora es encontrar las posiciones en reposo de las masas. Explícitamente las fuerzas netas son

$$\begin{aligned} F_1 &= -k_1(x_1 - L_1) + k_2(x_2 - x_1 - L_2) \\ F_2 &= -k_2(x_2 - x_1 - L_2) + k_3(x_3 - x_2 - L_3) \\ F_3 &= -k_3(x_3 - x_2 - L_3) + k_4(L_p - x_3 - L_4) \end{aligned} \tag{9.35}$$

o en forma matricial,

$$\begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} -k_1 - k_2 & k_2 & 0 \\ k_2 & -k_2 - k_3 & k_3 \\ 0 & k_3 & k_3 - k_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} -k_1 L_1 + k_2 L_2 \\ -k_2 L_2 + k_3 L_3 \\ -k_3 L_3 + k_4(L_4 - L_p) \end{bmatrix} . \tag{9.36}$$

[5] S. Conte and C. de Boor, *Elementary Numerical Analysis* (New York: McGraw-Hill, 1980).

[6] G.H. Golub and C.F. van Loan, *Matrix computation* 2d ed. (Baltimore Johns Hopkins University Press, 1989).

Por conveniencia, abreviaremos la ecuación de arriba como $\vec{F} = \check{K}\vec{x} - \vec{b}$. En forma matricial las simetrías son claras, y vemos que no sería difícil extender el problema a un sistema más grande de osciladores acoplados. En el estado de equilibrio estático las fuerzas netas son iguales a cero, así obtenemos el resto de las posiciones de las masas resolviendo $\check{K}\vec{x} - \vec{b}$.

9.3. Sistemas de ecuaciones no lineales.

9.3.1. Método de Newton en una variable.

Ahora que sabemos cómo resolver sistemas de ecuaciones lineales, procederemos al caso más general (y más desafiante) de resolver sistemas de ecuaciones no lineales. Este problema es difícil, tanto que consideraremos primero el caso de una variable. Deseamos resolver para x^* tal que

$$f(x^*) = 0 \ . \tag{9.37}$$

donde $f(x^*)$ es ahora una función general. Hay un número de métodos disponibles para encontrar raices en una variable. Quizás usted conozca algunos métodos como el de bisección o de la secante u otros algoritmos. También hay algoritmos especializados para cuando $f(x)$ es un polinomio (*e.g.*, función de `roots` en Octave). En vez de utilizar todos estos esquemas, nos concentraremos en el más simple y útil de los métodos para el caso general de N variables.

El *método de Newton*[7] está basado en la expansión de Taylor de $f(x)$ entorno a la raíz x^*. Supongamos que hacemos una estimación acerca de la localización de la raíz; llamamos a esta estimación x_1. Nuestro error podría ser escrito como $\delta x = x_1 - x^*$ o $x^* = x_1 - \delta x$. Escribiendo la expansión de Taylor de $f(x^*)$,

$$f(x^*) = f(x_1 - \delta x) = f(x_1) - \delta x \frac{df(x_1)}{dx} + \mathcal{O}(\delta x^2) \ . \tag{9.38}$$

Note que si x^* es una raíz, $f(x^*) = 0$, así podemos resolver para δx

$$\delta x = \frac{f(x_1)}{f'(x_1)} + \mathcal{O}(\delta x^2) \ . \tag{9.39}$$

Despreciamos el término $\mathcal{O}(\delta x^2)$ (este será nuestro error de truncamiento) y usamos la expresión resultante de δx para corregir nuestra estimación inicial. La nueva estimación es

$$x_2 = x_1 - \delta x = x_1 - \frac{f(x_1)}{f'(x_1)} \ . \tag{9.40}$$

Este procedimiento podría ser iterado como

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \ , \tag{9.41}$$

para mejorar nuestra estimación hasta obtener la precisión deseada.

El procedimiento iterativo descrito más arriba puede entenderse mejor gráficamente (figura 9.2). Note que en cada paso usamos la derivada de $f(x)$ para dibujar la línea tangente a

[7] F.S. Acton, *Numerical Methods that Work* (New York: Harper&Row, 1970).

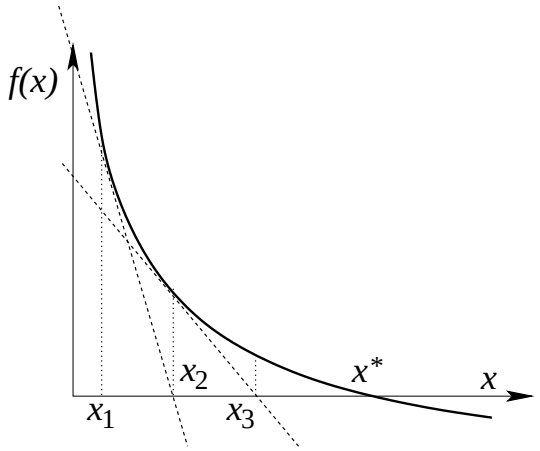


Figura 9.2: Representación gráfica del método de Newton.

la función. Donde esta línea tangente intersecta el eje x es nuestra siguiente estimación de la raíz. Efectivamente, estamos linealizando $f(x)$ y resolviendo el problema lineal. Si la función es bien comportada, pronto será aproximadamente lineal en alguna vecindad cerca de la raíz x^*.

Unas pocas notas acerca del método de Newton: primero, cuando converge, encuentra una raíz muy rápidamente; pero, desafortunadamente, algunas veces diverge (*e.g.*, $f'(x_n) \approx 0$) y falla. Para funciones bien comportadas, este método es garantizado que converger si estamos suficientemente cerca de la raíz. Por esta razón algunas veces está combinada con un algoritmo más lento pero que asegura encontrar la raíz, tal como la bisección. Segundo, si hay varias raices, la raíz a la cual el método converge depende de la estimación inicial (que podría no ser la raíz deseada). Hay procedimientos (*e.g.*, "*deflation*") para encontrar múltiples raices usando el método de Newton. Finalmente, el método es más lento cuando se encuentran raices tangentes, tales como para $f(x) = x^2$.

9.3.2. Método de Newton multivariable.

No es difícil generalizar el método de Newton a problemas de N variables. Ahora nuestra incógnita $\vec{x} = [x_1\ ,\ x_2\ ,\ \ldots\ ,\ x_N]$ es un vector fila, y deseamos encontrar los ceros (raices) de la función vector fila

$$\vec{f}(\vec{x}) = [f_1(\vec{x})\ ,\ f_2(\vec{x})\ ,\ \ldots\ ,\ f_N(\vec{x})]\ . \tag{9.42}$$

De nuevo, hacemos una estimación inicial para ubicar la raíz, llamamos a esta estimación $\vec{x}_1$. Nuestro error podría ser esrito como $\delta\vec{x} = \vec{x}_1 - \vec{x}^*$ o $\vec{x}^* = \vec{x}_1 - \delta\vec{x}$. Usando la expansión de Taylor de $\vec{f}(\vec{x}_1)$,

$$\vec{f}(\vec{x}^*) = \vec{f}(\vec{x}_1 - \delta\vec{x}) = \vec{f}(\vec{x}_1) - \delta\vec{x}\,\check{D}(\vec{x}_1) + \mathcal{O}(\delta\vec{x}^2)\ , \tag{9.43}$$

donde $\check{D}$ es el Jacobiano, definido como

$$D_{ij}(\vec{x}) = \frac{\partial f_j(\vec{x})}{\partial x_i}\ . \tag{9.44}$$

Ya que $\vec{x}^*$ es la raíz, $\vec{f}(\vec{x}^*) = 0$, como antes podríamos resolverla para $\delta\vec{x}$. Despreciando el término de error, podemos escribir la ecuación (9.43) como

$$\vec{f}(\vec{x}_1) = \delta\vec{x}\,\check{D}(\vec{x}_1)\ , \tag{9.45}$$

o

$$\delta\vec{x} = \vec{f}(\vec{x}_1)\,\check{D}^{-1}(\vec{x}_1)\ . \tag{9.46}$$

Nuestra nueva estimación es

$$\vec{x}_2 = \vec{x}_1 - \delta\vec{x} = \vec{x}_1 - \vec{f}(\vec{x}_1)\,\check{D}^{-1}(\vec{x}_1)\ . \tag{9.47}$$

Este procedimiento puede ser iterado para mejorar nuestra estimación hasta que sea obtenida la precisión deseada. Ya que $\check{D}$ cambia en cada iteración, podría ser desgastador calcular su inversa. En cambio, usamos la eliminación Gaussiana sobre la ecuación (9.45) para resolver para $\delta\vec{x}$, y calcular la nueva estimación como $\vec{x}_2 = \vec{x}_1 - \delta\vec{x}$.

9.3.3. Programa del método de Newton.

Para demostrar el método de Newton, lo usaremos para calcular estados estacionarios del modelo de Lorenz.

A principios de la década de 1960 el meteorólogo del MIT, Ed Lorenz, encontró que el clima es intrínsecamente impredecible, no por su complejidad, sino por la naturaleza no lineal de las ecuaciones que lo gobiernan. Lorenz formuló un modelo simple del clima global, reduciendo el problema a un sistema de ecuaciones diferenciales ordinarias de 12 variables. Él observó que este sistema tenía un comportamiento aperiódico y que era extremadamente sensible a las condiciones iniciales. Estudiemos un modelo de Lorenz simplificado a tres variables

$$\begin{aligned}
\frac{dx}{dt} &= \sigma(y - x)\ , \\
\frac{dy}{dt} &= rx - y - xz\ , \\
\frac{dz}{dt} &= xy - bz\ ,
\end{aligned} \tag{9.48}$$

donde σ, r y b son contantes positivas. Estas ecuaciones simples fueron originalmente desarrolladas como un modelo para un fluido con convección. La variable x mide la razón de convección, y y z miden los gradientes de temperaturas horizontales y verticales. Los parámetros σ y b dependen de las propiedades del fluido y de la geometría del contenedor; comúnmente, los valores $\sigma = 10$ y $b = 8/3$ son usados. El parámetro r es proporcional al gradiente de temperatura aplicado.

Un programa que encuentra las raices de este sistema de ecuaciones usando el método de Newton, al que llamamos `newtn`, es esbozado en la tabla 9.3. Este programa llama la función `fnewt` (tabla 9.4), la cual, dado un $\vec{x} = [x\ ,\ y\ ,\ z]$ regresa

$$\vec{f} = \begin{bmatrix} \sigma(y - x) \\ rx - y - xz \\ xy - bz \end{bmatrix}\ , \quad \check{D} = \begin{bmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{bmatrix}\ . \tag{9.49}$$

- Conjunto inicial de estimaciones $\vec{x}_1$ y parámetros λ_k.
- Itera sobre el número de pasos deseados.
  - Evalúa la función $\vec{f}(\vec{x}_n; \lambda_k)$ y su Jacobiano $\check{D}$.
  - Encuentra $\delta\vec{x}$ por la eliminación Gaussiana [ver (9.45)].
  - Actualiza la estimación para la raíz usando $\vec{x}_{n+1} = \vec{x}_n - \delta\vec{x}$.
- Imprime la estimación final de la raíz.

Cuadro 9.3: Descripción del programa `newtn`, el cual encuentra una raíz para un conjunto de ecuaciones.

Para $r = 28$, $\sigma = 10$ y $b = 8/3$, las tres raíces son $[x, y, z] = [0, 0, 0]$, $[6\sqrt{2}, 6\sqrt{2}, 27]$ y $[-6\sqrt{2}, -6\sqrt{2}, 27]$.

Un ejemplo de la salida de `newtn` está dada más abajo; note que el programa obtiene la raíz $[6\sqrt{2}, 6\sqrt{2}, 27]$.

```
octave:1> newtn
newtn is the file: ~/cursos/MFM1.apuntes2/programas/newtn.m

 newtn - Programa para resolver un sistema de ecuaciones no lineales
 usando el metodo de Newton. Las ecuaciones estan definidas en fnewt

Entre la estimacion inicial (vector columna): [50; 50; 50]
Entre los parametros a: [28 10 8/3]
Despues de  10 iteraciones la raiz es
     8.4853   8.4853  27.0000
```

Otras condiciones iniciales convergen a otras raices. Por ejemplo, comenzando por $[2, 2, 2]$, convergemos a la raíz $[0, 0, 0]$; si comenzamos en $[5, 5, 5]$, converge a $[-6\sqrt{2}, -6\sqrt{2}, 27]$. Comenzando por el punto $[4, 4, 15]$, el método converge a una raíz sólo después de hacer una excursión increíblemente distante del origen.

9.3.4. Continuación.

La primera dificultad con el método de Newton es la necesidad de dar una buena estimación inicial para la raíz. A menudo nuestro problema es de la forma

$$\vec{f}(\vec{x}^*; \lambda) = 0 \ , \tag{9.50}$$

donde λ es algún parámetro en el problema. Supongamos que conocemos una raíz $\vec{x}_0^*$ para el valor λ_0 pero necesitamos encontrar una raíz para un valor diferente de λ_a. Intuitivamente, si $\lambda_a \approx \lambda_0$, entonces $\vec{x}_0^*$ debería ser una buena estimación inicial para encontrar $\vec{x}_a^*$ usando

- *Entradas:* $\vec{x} = [x, y, z]$, $\lambda = [r, \sigma, b]$.
- *Salidas:* $\vec{f}, \check{D}$.
- Evalúa $\vec{f}$ por el modelo de Lorenz [(9.49)].
- Evalúa el Jacobiano $\check{D}$ para el modelo de Lorenz.

Cuadro 9.4: Descripción de la función `fnewt`, la cual es usada por `newtn` para encontrar los estados estables de la ecuación de Lorenz.

el método de Newton. Pero si $\lambda_a \not\approx \lambda_0$, ¿hay alguna manera para hacer uso de nuestra raíz conocida?

La respuesta es sí y la técnica es conocida como *continuación*. La idea es acercarse poco a poco a λ_a definiendo la siguiente secuencia de λ :

$$\lambda_i = \lambda_0 + (\lambda_a - \lambda_0)\frac{i}{N} \ , \tag{9.51}$$

para $i = 1, \ldots\ldots, N$. Usando el método de Newton, resolvemos $f(\vec{x}_1^*; \lambda_1) = 0$ con $\vec{x}_0^*$ como la estimación inicial. Si N es suficientemente grande, entonces $\lambda_1 \approx \lambda_0$ y el método podría converger rápidamente. Luego usamos $\vec{x}_1^*$ como una estimación inicial para resolver $f(\vec{x}_2^*; \lambda_2) = 0$; la secuencia continúa hasta alcanzar nuestro valor deseado en λ_a. La técnica de continuación tiene un beneficio adicional que podemos usar cuando estamos interesados en conocer no sólo una raíz simple, sino conocer como $\vec{x}^*$ varía con λ.

9.4. Listados del programa.

9.4.1. Definición de la clase `Matrix`.

```
//
// Clase de Matrices doubles
//
#ifndef _Matrix_h
#define _Matrix_h

#include <iostream>
#include <string>

class Matrix {
 private:
  int nFilP ;
  int nColP ;
  int dimP ;
  double * dataP ;
  void copy(const Matrix &) ;
 public:
  Matrix() ;
  Matrix( int, int = 1, double = 0.0e0) ;
  Matrix( const Matrix &) ;
  ~Matrix() ;
  Matrix & operator = (const Matrix &) ;
  double & operator () (int, int =0) ;
  int nRow () const;
  int nCol () const ;
  void set(double) ;
  double maximoF(int) const ;
  void pivot(int,int) ;
  void multi(double, int, int ) ;
};

ostream & operator << (ostream &, Matrix );

void error ( int, string="Error") ;

Matrix operator * (Matrix, Matrix) ;
Matrix operator + (Matrix, Matrix) ;
Matrix operator - (Matrix, Matrix) ;

#endif
```

9.4.2. Implementación de la clase `Matrix`.

```
//
// Implementacion de la clase Matrix

#include "Matrix.h"
#include <cassert>
#include <cstdlib>

// Funcion privada de copia
void Matrix::copy(const Matrix & mat) {
  nFilP=mat.nFilP ;
  nColP=mat.nColP ;
  dimP = mat.dimP ;
  dataP = new double[dimP] ;
  for(int i =0 ; i< dimP; i++) dataP[i]=mat.dataP[i] ;
}

// Constructor default Crea una matriz de 1 por 1 fijando su valor a cero
Matrix::Matrix() {
  Matrix(1) ;
}

// Constructor normal Crea una matriz de N por M
// y fija sus valores a cero o a un valor ingresado
Matrix::Matrix(int nF, int nC, double v) {
  error(nF>0 && nC >0, "Una de las dimensiones no es positiva" ) ;
  nFilP= nF ;
  nColP = nC ;
  dimP = nFilP*nColP ;
  dataP = new double[dimP] ;
  assert(dataP != 0) ;          // Habia memoria para reservar
  for (int i=0; i < dimP; i++) dataP[i]=v ;
}

// Constructor de copia
Matrix::Matrix( const Matrix & mat) {
  this->copy(mat) ;
  // this permite acceder a la misma vartiable como un todo
  // *this representa la variable, y this un puntero a ella
  // -> nos permite elegir miembros de la clase pero cuando tenemos el puntero
  // a la variable y no su valor
}

// Destructor
```

```
Matrix::~Matrix() {
  delete [] dataP ;             // Libera la memoria
}

// Operador Asignacion, sobrecarga del = para que funcione con Matrix
Matrix & Matrix::operator= (const Matrix & mat) {
  if( this == &mat ) return *this ; // Si ambos lados son iguales no hace nada
  delete [] dataP ;             //  borra la data del lado izquierdo
  this-> copy(mat) ;
  return * this ;
}

// Funciones simples que retornan el numero de filas y de columnas
// el const al final indica que no modifican ningun miembro de la clasa
int Matrix::nRow() const {return nFilP; }
int Matrix::nCol() const {return nColP;}

// operador Parentesis
// Permite accesar los valores de una matriz via el par (i,j)
// Ejemplo a(1,1)=2*b(2,3)
// Si no se especifica la columna la toma igual a 0
double & Matrix::operator() (int indF, int indC) {
  error(indF>-1 && indF< nFilP, "Indice de fila fuera de los limites") ;
  error(indC>-1 && indC<nColP, "Indice de columna fuera de los limites" ) ;
  return dataP[indC+nColP*indF] ;
}

void Matrix::set(double value) {
  for(int i=0; i<dimP; i++) dataP[i]=value ;
}

double Matrix::maximoF(int indF) const {
  error(indF>-1 && indF< nFilP, "Indice de fila fuera  en maximoF") ;
  double max= dataP[nColP*indF];
  for(int i=nColP*indF; i<nColP*(indF+1); i++ ) {
    max = (max > fabs(dataP[i]) ) ? max : fabs(dataP[i]) ;
  }
  return max ;
}

void Matrix::pivot(int indF1, int indF2) {
  error(indF1>-1 && indF1< nFilP, "Primer indice de fila fuera en Pivot") ;
  error(indF2>-1 && indF2< nFilP, "Segundo indice de fila fuera en Pivot") ;
```

```
  if(indF1==indF2) return ;
  double paso ;
  int ind1, ind2 ;
  for (int i = 0; i < nColP; i++ ) {
    ind1 = nColP*indF1+i ;
    ind2 = nColP*indF2+i ;
    paso = dataP[ind1] ;
    dataP[ind1] = dataP[ind2] ;
    dataP[ind2] = paso ;
  }
}

void Matrix::multi(double fact, int indF1, int indF2 ) {
  int ind1, ind2 ;
  for (int i = 0; i < nColP; i++ ) {
    ind1 = nColP*indF1+i ;
    ind2 = nColP*indF2+i ;
    dataP[ind2] += fact*dataP[ind1] ;
  }
}

//
// IOSTREAM <<
//
ostream & operator << (ostream & os, Matrix mat ) {
  for(int indF=0; indF < mat.nRow(); indF++) {
        os << "| " ;
    for(int indC=0; indC<mat.nCol(); indC++) {
      os << mat(indF,indC) << " ";
    }
    os << "|"<<endl ;
  }
  return os ;
}

void error ( int i, string s) {
  if(i) return ;
  cout << s<<"\a" <<endl ;
  exit(-1);
}

Matrix  operator * (Matrix A , Matrix B) {
  int nColA = A.nCol() ;
  int nColB = B.nCol() ;
  int nFilA = A.nRow() ;
```

```
  int nFilB = B.nRow() ;
  error(nColA == nFilB, "No se pueden multiplicar por sus dimensiones") ;
  Matrix R(nFilA   , nColB ) ;
  for (int indF = 0; indF<nFilB; indF++){
    for (int indC=0; indC<nColA; indC++){
      double sum = 0.0 ;
      for (int k=0; k < nColA; k++ ) sum += A(indF, k) * B(k, indC) ;
      R(indF, indC) = sum  ;
    }
  }
  return R ;
}

Matrix  operator + (Matrix A , Matrix B) {
  int nColA = A.nCol() ;
  int nColB = B.nCol() ;
  int nFilA = A.nRow() ;
  int nFilB = B.nRow() ;
  error(nFilA == nFilB && nColA== nColB, "tienen dimensiones distintas") ;
  Matrix R(nFilA   , nColA ) ;
  for (int indF = 0; indF<nFilB; indF++){
    for (int indC=0; indC<nColA; indC++){
      R(indF, indC) = A( indF, indC) + B(indF, indC);
    }
  }
  return R ;
}

Matrix  operator - (Matrix A , Matrix B) {
  int nColA = A.nCol() ;
  int nColB = B.nCol() ;
  int nFilA = A.nRow() ;
  int nFilB = B.nRow() ;
  error(nFilA == nFilB && nColA== nColB, "tienen dimensiones distintas") ;
  Matrix R(nFilA   , nColA ) ;
  for (int indF = 0; indF<nFilB; indF++){
    for (int indC=0; indC<nColA; indC++){
      R(indF, indC) = A( indF, indC) - B(indF, indC);
    }
  }
  return R ;
}
```

9.4.3. Función de eliminación Gaussiana ge.

```
//
// Eliminacion Gaussiana.
//

#include "NumMeth.h"
#include "Matrix.h"

double ge( Matrix A, Matrix  b, Matrix & x)
{
  int nFil = A.nRow() ;
  int nCol = A.nCol() ;
  Matrix si (nFil) ;

  for ( int indF= 0; indF < nFil ; indF++)  si(indF)= A.maximoF(indF) ;
  double determinante=1.0e0 ;
  for (int indC= 0; indC< nCol-1; indC++) {
    // pivoteo
    double max = A(indC,indC)/si(indC) ;
    int indicePivot = indC ;
    for (int i = indC+1; i < nCol ; i++){
      if( A(i, indC)/si(i) > max ) {
        max = A(i,indC)/si(i) ;
        indicePivot = i ;
      }
    }
    if(indicePivot != indC)  {
      A.pivot(indC, indicePivot) ;
      b.pivot(indC, indicePivot) ;
      determinante *= -1.0e0 ;
    }
    double Ad= A(indC,indC) ;
    for (int indF=indC+1; indF< nFil; indF++ ) {
      double factor =  - A(indF, indC)/Ad ;
      A.multi(factor, indC, indF);
      b.multi(factor, indC, indF);
    }
  }
  for (int indF= nFil-1; indF >=0; indF--) {
    double sum =0.0e0 ;
    for (int i = indF+1; i < nCol; i++) sum += A(indF, i)*x(i) ;
    x(indF) =  b(indF)/A(indF, indF) - sum /A(indF, indF);
  }
  double determ = 1.0e0 ;
```

```
  for (int i = 0; i < nCol; i++) determ *= A(i,i) ;

  return determinante*determ  ;
}
```

9.4.4. Función para inversión de matrices inv.

```
//
// Invierte una matriz
//

#include "NumMeth.h"
#include "Matrix.h"

double inv( Matrix A, Matrix & invA) {
  int nFil = A.nRow() ;
  int nCol = A.nCol() ;
  error(nFil==nCol, "Matriz no es cuadrada");
  Matrix e(nFil) ;
  Matrix x(nFil) ;
  double deter = 0.0e0 ;
  invA.set(0.0e0) ;
  for(int indC=0; indC < nCol; indC++) {
    e.set(0.0e0) ;
    e(indC) = 1.0e0;
    deter = ge( A, e, x) ;
    error(fabs(deter)>1.0e-10, " Determinante Singular" );
    for (int indF=0; indF< nFil; indF++)   invA(indF, indC) = x(indF) ;
  }
  return deter ;
}
```

9.4.5. Programa newtn en Octave.

```
% newtn - Programa para resolver un sistema de ecuaciones no lineales
% usando el metodo de Newton. Las ecuaciones estan definidas en fnewt
suppress_verbose_help_message=1;
clear all; help  newtn;

x0=input('Entre la estimacion inicial (vector columna): ');
x=x0;
a=input('Entre los parametros a: ');

nStep =10;
```

```
for iStep=1:nStep

[f D] = fnewt(x,a) ;
dx=D\f ;
x=x-dx;
end
fprintf('Despues de  %g iteraciones la raiz es \n', nStep);
disp(x');
```

Función `fnewt` en Octave.

```
function [f,D] = fnewt(x,a)

f(1)=a(2)*(x(2)-x(1));
f(2)= a(1)*x(1)-x(2)-x(1)*x(3);
f(3)=x(1)*x(2)-a(3)*x(3) ;

D(1,1)=-a(2);
D(2,1)=a(1)-x(3);
D(3,1)=x(2);
D(1,2)=a(2);
D(2,2)=-1;
D(3,2)=x(1);
D(1,3)=0;
D(2,3)=-x(1);
D(3,3)=-a(3);

return
```

9.4.6. Programa newtn en c++.

```
#include <iostream>
#include "NumMeth.h"
#include "Matrix.h"

void fnewt( Matrix & f, Matrix & D, Matrix xN, Matrix lambda )
{
  double r= lambda(0) ;
  double s= lambda(1) ;
  double b=lambda(2) ;
  double x=xN(0) ;
  double y=xN(1) ;
  double z=xN(2) ;
  f(0) = s*(y-x) ;
  f(1) = r*x-y-x*z ;
  f(2) = x*y-b*z ;
  D(0,0) = -s ;
  D(1,0) = r-z ;
  D(2,0) = y ;
  D(0,1) = s ;
  D(1,1)= -1.0e0 ;
  D(2,1) = x ;
  D(0,2)=0.0e0 ;
  D(1,2)= -x ;
  D(2,2) =-b ;
}

double ge( Matrix A, Matrix  b, Matrix & x)
{
  int nFil = A.nRow() ;
  int nCol = A.nCol() ;
  Matrix si (nFil) ;

  for ( int indF= 0; indF < nFil ; indF++)  si(indF)= A.maximoF(indF) ;
  double determinante=1.0e0 ;
  for (int indC= 0; indC< nCol-1; indC++) {
    // pivoteo
    double max = A(indC,indC)/si(indC) ;
    int indicePivot = indC ;
    for (int i = indC+1; i < nCol ; i++){
      if( A(i, indC)/si(i) > max ) {
        max = A(i,indC)/si(i) ;
        indicePivot = i ;
      }
```

```
    }
    if(indicePivot != indC)  {
      A.pivot(indC, indicePivot) ;
      b.pivot(indC, indicePivot) ;
      determinante *= -1.0e0 ;
    }
    double Ad= A(indC,indC) ;
    for (int indF=indC+1; indF< nFil; indF++ ) {
      double factor =  - A(indF, indC)/Ad ;
      A.multi(factor, indC, indF);
      b.multi(factor, indC, indF);
    }
  }
  for (int indF= nFil-1; indF >=0; indF--) {
    double sum =0.0e0 ;
    for (int i = indF+1; i < nCol; i++) sum += A(indF, i)*x(i) ;
    x(indF) =  b(indF)/A(indF, indF) - sum /A(indF, indF);
  }
  double determ = 1.0e0 ;
  for (int i = 0; i < nCol; i++) determ *= A(i,i) ;

  return determinante*determ  ;
}


int main()
{
  Matrix D(3,3) ;
  Matrix f(3) ;
  Matrix lambda(3), xN(3), xNp1(3), dx(3) ;

  cout <<"Ingrese r , Sigma, b : " ;
  cin >> lambda(0) >> lambda(1) >> lambda(2) ;
  cout << "Ingrese estimacion inicial : " ;
  cin>>   xN(0) >>  xN(1) >>   xN(2) ;

  int iteraciones ;
  cout << "Ingrese numero de iteraciones : " ;
  cin >> iteraciones ;

  for (int itera = 0; itera < iteraciones; itera ++ ) {
    f.set(0.0e0) ;
    D.set(0.0e0) ;
    dx.set(0.0e0) ;
    fnewt(f, D, xN, lambda ) ;
```

```
    ge(D, f, dx) ;
    xNp1=xN-dx ;
    xN=xNp1 ;
    cout << xN(0)<< " " << xN(1) << " " << xN(2) << endl ;
  }
  cout << xN <<endl ;
  return 0;
}
```

Capítulo 10

Análisis de datos.

versión 3.0, 02 de Diciembre 2003[1]

A menudo se destaca que los sistemas físicos simulados sobre un computador son similares al trabajo experimental. La razón de esta analogía es hecha ya que la simulación computacional produce datos en muchas maneras similares a los experimentos de laboratorio. Sabemos que en el trabajo experimental uno a menudo necesita analizar los resultados y esto es lo mismo que con la simulación numérica. Este capítulo cubre parcialmente algunos tópicos en el análisis de datos.

10.1. Ajuste de curvas.

10.1.1. El calentamiento global.

En el presente, parece que predicciones sobre el tiempo de largo alcance y precisas nunca se llevarán a cabo. La razón es a causa de las ecuaciones gobernantes que son altamente no lineales y sus soluciones son extremadamente sensibles a las condiciones iniciales (ver el modelo de Lorenz). Por otra parte, las predicciones generales acerca del clima terrestre son posibles. Se pueden predecir si habrá o no condiciones de sequía en África en los próximos años, aunque no la cantidad de precipitaciones de un día en particular.

El calentamiento global es un importante tópico actualmente debatido en los foros de investigación sobre clima. El calentamiento es culpa de los gases de invernadero, tal como es el dióxido de carbono en la atmósfera. Estos gases calientan la Tierra porque son transparentes a la radiación de onda corta que llega desde el Sol, pero opacas a la radiación infrarroja desde la tierra. Científicos y legisladores todavía están debatiendo la amenaza del calentamiento global. Sin embargo, nadie se pregunta qué concentraciones de gases invernadero están en aumento. Específicamente, los niveles de dióxido de carbono han estado firmemente aumentados desde la revolución industrial. La figura 10.1 muestra el aumento en la concentración de dióxido de carbono durante los años ochenta, medidos en Mauna Loa, Hawai.

El estudio del calentamiento global ha producido una vasta cantidad de datos, tanto mediciones prácticas, como datos de las simulaciones computacionales. En este capítulo estudiaremos algunas técnicas básicas para analizar y reducir tales conjuntos de datos. Por ejemplo, para los datos mostrados en la figura 10.1, ¿cuál es la razón estimada de crecimiento

[1]Este capítulo está basado en el quinto capítulo del libro: *Numerical Methods for Physics, second edition* de Alejandro L. Garcia, editorial Prentice Hall.

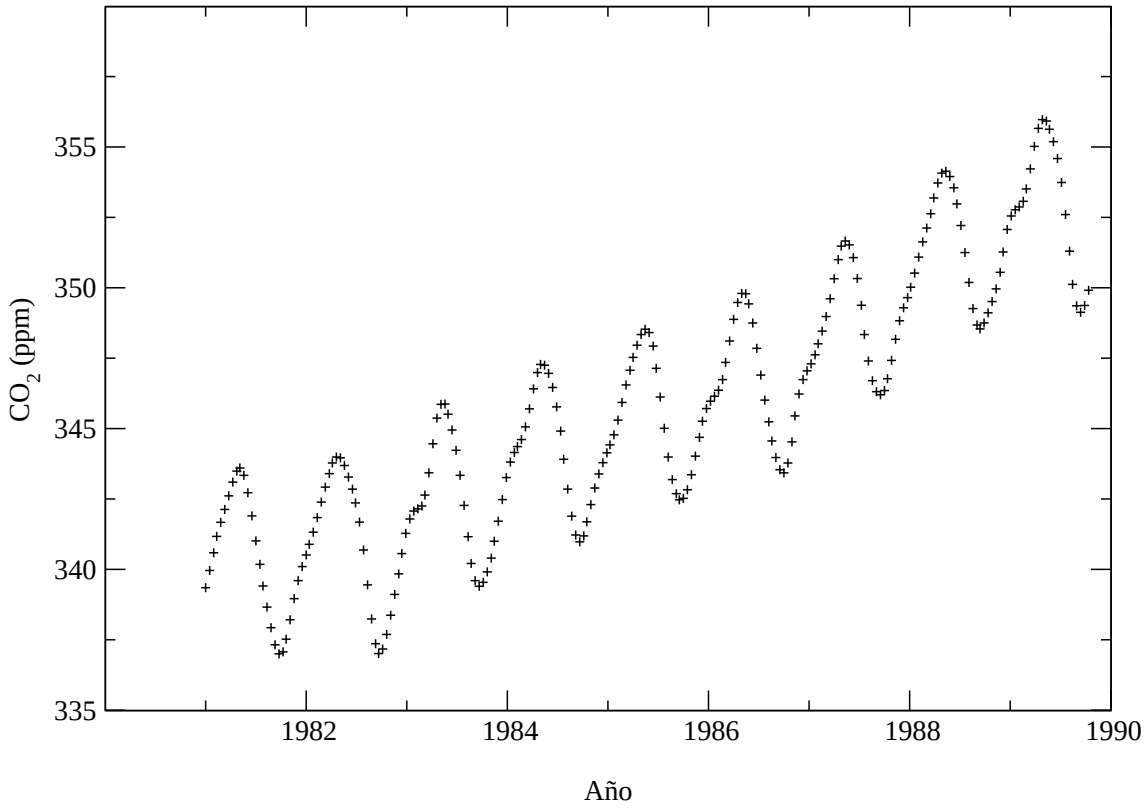


Figura 10.1: Dióxido de carbono (en partes por millón) medida en Mauna Loa, Hawai desde 1981 a 1990. Barra de error estimada es $\sigma_0 = 0.16$ [p.p.m.] .

de la concentración de CO_2 por año?. Esta es la primera pregunta que motiva nuestro estudio de ajustes de curvas.

10.1.2. Teoría general.

El tipo más simple de análisis de datos es ajustar una curva. Suponga que tenemos un conjunto de datos de N puntos (x_i, y_i). Deseamos ajustar estos datos a una función $Y(x; \{a_j\})$, donde $\{a_j\}$ es un conjunto de M parámetros ajustables. Nuestro objetivo es encontrar los valores de esos parámetros, para los cuales la función ajusta mejor los datos. Intuitivamente, esperamos que si nuestro ajuste de la curva es bueno, un gráfico del conjunto de datos (x_i, y_i) y la función $Y(x; \{a_j\})$ mostrarán la curva pasando cerca de los puntos (ver figura 10.2). Podemos cuantificar esta sentencia midiendo la distancia entre un punto y la curva.

$$\Delta_i = Y(x_i; \{a_j\}) - y_i \ . \tag{10.1}$$

Nuestro criterio de ajuste para la curva será que la suma de los cuadrados de los errores sea un mínimo; esto es, necesitamos encontrar $\{a_j\}$ que minimice la función

$$D(\{a_j\}) = \sum_{i=1}^{N} \Delta_i^2 = \sum_{i=1}^{N} \left[Y(x_i; \{aj\}) - y_i\right]^2 \ . \tag{10.2}$$

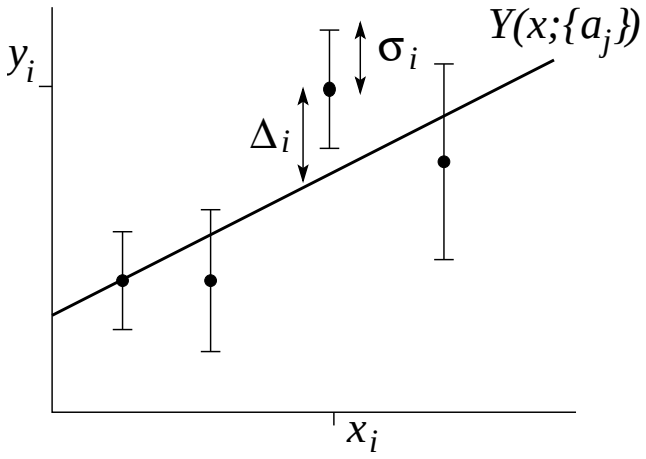


Figura 10.2: Ajuste de datos a una curva.

Esta técnica nos dará el *ajuste de los cuadrados mínimos*; esta no es la única manera de obtener un ajuste de la curva, pero es la más común. El método de los mínimos cuadrados fue el primero usado por Gauss para determinar las órbitas de los cometas a partir de los datos observados.

A menudo, nuestros datos tienen una barra de error estimada (o intervalo de seguridad), el cual escribimos como $y_i \pm \sigma_i$. En este caso podríamos modificar nuestro criterio de ajuste tanto como para dar un peso menor a los puntos con más error. En este espíritu definimos

$$\chi^2(\{a_j\}) = \sum_{i=1}^{N} \left(\frac{\Delta_i}{\sigma_i} \right)^2 = \sum_{i=1}^{N} \frac{[Y(x_i;\{a_j\}) - y_i]^2}{\sigma_i^2} \ . \tag{10.3}$$

La función chi-cuadrado es la función de ajuste más comúnmente usada, porque si los errores son distribuidos gaussianamente, podemos hacer sentencias estadísticas concernientes a la virtud del ajuste.

Antes de continuar, debemos destacar que discutiremos brevemente la validación de la curva de ajuste. Usted puede ajustar cualquier curva para cualquier conjunto de datos, pero esto no significa que los resultados sean significativos. Para establecer un significado tenemos que preguntarnos lo siguiente: ¿cuál es la probabilidad de que los datos, dado el error experimental asociado con cada uno de los puntos, sean descritos por la curva?. Desafortunadamente, las hipótesis de prueba ocupan una porción significativa de un curso estadístico y están fuera de los objetivos de este libro.[2]

10.1.3. Regresión lineal.

Primero consideremos ajustar el conjunto de datos con una línea recta,

$$Y(x;\{a_1, a_2\}) = a_1 + a_2 x \ . \tag{10.4}$$

[2]W. Mendenhall, R.L. Scheaffer, and D.D. Wackerly, *Mathematical Statistics with Applications* (Boston: Duxbury Press, 1981).

Este tipo de ajuste de curva es también conocido como regresión lineal. Deseamos determinar a_1 y a_2 tal que

$$\chi^2(a_1, a_2) = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left(a_1 + a_2 x_i - y_i\right)^2 \ , \tag{10.5}$$

es minimizada. El mínimo es encontrado diferenciando (10.5) y ajustando la derivada a cero:

$$\frac{\partial \chi^2}{\partial a_1} = 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left(a_1 + a_2 x_i - y_i\right) = 0 \ , \tag{10.6}$$

$$\frac{\partial \chi^2}{\partial a_2} = 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left(a_1 + a_2 x_i - y_i\right) x_i = 0 \ , \tag{10.7}$$

o

$$a_1 S + a_2 \Sigma x - \Sigma y = 0 \tag{10.8}$$

$$a_1 \Sigma x + a_2 \Sigma x^2 - \Sigma xy = 0 \ , \tag{10.9}$$

donde

$$\begin{aligned} S &\equiv \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \ , \quad \Sigma x \equiv \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \ , \quad \Sigma y \equiv \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \ , \\ \Sigma x^2 &\equiv \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \ , \quad \Sigma y^2 \equiv \sum_{i=1}^{N} \frac{y_i^2}{\sigma_i^2} \ , \quad \Sigma xy \equiv \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \ . \end{aligned} \tag{10.10}$$

Puesto que las sumas pueden ser calculadas directamente de los datos, ellas son constantes conocidas. De este modo tenemos un conjunto lineal de dos ecuaciones simultáneas en las incógnitas a_1 y a_2. Estas ecuaciones son fáciles de resolver:

$$a_1 = \frac{\Sigma y \Sigma x^2 - \Sigma x \Sigma xy}{S\Sigma x^2 - (\Sigma x)^2} \ , \qquad a_2 = \frac{S\Sigma xy - \Sigma x \Sigma y}{S\Sigma x^2 - (\Sigma x)^2} \ . \tag{10.11}$$

Note que si σ_i es una constante, esto es, si el error es el mismo para todos los datos, los σ se cancelan en las ecuaciones (10.11). En este caso, los parámetros, a_1 y a_2, son independientes de la barra de error. Es común que un conjunto de datos no incluya las barras de error asociadas. Todavía podríamos usar las ecuaciones (10.11) para ajustar los datos si tenemos que los σ_i son constantes (σ_i=1 siendo la elección más simple).

Lo siguiente es obtener una barra de error asociado, σ_{aj}, para el parámetro a_j de la curva de ajuste. Usando la ley de la propagación de errores,[3]

$$\sigma_{a_j}^2 = \sum_{i=1}^{N} \left(\frac{\partial a_j}{\partial y_i}\right) \sigma_i^2 \ , \tag{10.12}$$

[3]P. Bevington, *Data Reduction an Error Analysis for the Physical Sciences* 2d ed. (New York: McGraw-Hill, 1992).

e insertando las ecuaciones (10.11), después de un poco de álgebra obtenemos

$$\sigma_{a_1} = \sqrt{\frac{\Sigma x^2}{S\Sigma x^2 - (\Sigma x)^2}} \ , \qquad \sigma_{a_2} = \sqrt{\frac{S}{S\Sigma x^2 - (\Sigma x)^2}} \ . \tag{10.13}$$

Note que σ_{a_j} es independiente de y_i. Si las barras de error de los datos son constantes ($\sigma_i = \sigma_0$), el error en los parámetros es

$$\sigma_{a_1} = \frac{\sigma_0}{\sqrt{N}}\sqrt{\frac{\langle x^2\rangle}{\langle x^2\rangle - \langle x\rangle^2}} \ , \qquad \sigma_{a_2} = \frac{\sigma_0}{\sqrt{N}}\sqrt{\frac{1}{\langle x^2\rangle - \langle x\rangle^2}} \ , \tag{10.14}$$

donde

$$\langle x\rangle = \frac{1}{N}\sum_{i=1}^{N} x_i \ , \quad \langle x^2\rangle = \frac{1}{N}\sum_{i=1}^{N} x_i^2 \ . \tag{10.15}$$

Finalmente, si nuestro conjunto de datos no tiene un conjunto de barras de error asociadas, podríamos estimar σ_0 a partir de la diferencia muestral de los datos,

$$\sigma_0^2 \approx s^2 = \frac{1}{N-2}\sum_{i=1}^{N}\left[y_i - (a_1 + a_2 x_i)\right]^2 \ , \tag{10.16}$$

donde s es la desviación estándar de la muestra. Note que esta varianza muestral está normalizada por $N-2$, puesto que hemos extraído dos parámetros a_1 y a_2, de los datos.

Muchos problemas de ajuste de curva no lineales, pueden ser transformados en problemas lineales por un simple cambio de variable. Por ejemplo,

$$Z(x;\{\alpha,\beta\}) = \alpha e^{\beta x} \ , \tag{10.17}$$

podría ser reescrita como las ecuaciones (10.4) usando el cambio de variable

$$\ln Z = Y \ , \quad \ln\alpha = a_1 \ , \quad \beta = a_2 \ . \tag{10.18}$$

Similarmente, para ajustar una potencia de la forma

$$Z(t;\{\alpha,\beta\}) = \alpha t^{\beta} \ , \tag{10.19}$$

usamos el cambio de variable

$$\ln Z = Y \ , \quad \ln t = x \ , \quad \ln\alpha = a_1 \ , \quad \beta = a_2 \ . \tag{10.20}$$

Estas transformaciones deberían ser familiares debido a que se usan para graficar datos en escala semi logarítmica o escalas log-log.

10.1.4. Ajuste general lineal de mínimos cuadrados.

El procedimiento de ajuste de mínimos cuadrados es fácil de generalizar para funciones de la forma

$$Y(x;\{a_j\}) = a_1 Y_1(x) + a_2 Y_2(x) + \ldots + a_M Y_M(x) = \sum_{j=1}^{M} a_j Y_j(x) \ . \tag{10.21}$$

Para encontrar los óptimos parámetros procedemos como antes encontrando el mínimo de χ^2,

$$\frac{\partial \chi^2}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left\{ \sum_{k=1}^{M} a_k Y_k(x) - y_i \right\}^2 = 0 \ , \tag{10.22}$$

o

$$\sum_{i=1}^{N} \frac{1}{\sigma_i^2} Y_j(x_i) \left\{ \sum_{k=1}^{M} a_k Y_k(x) - y_i \right\} = 0 \ , \tag{10.23}$$

por lo tanto

$$\sum_{i=1}^{N} \sum_{k=1}^{M} \frac{Y_j(x_i) Y_k(x_i)}{\sigma_i^2} a_k = \sum_{i=1}^{N} \frac{Y_j(x_i) y_i}{\sigma_i^2} \ , \tag{10.24}$$

para $j = 1, \ldots, M$. Este conjunto de ecuaciones es conocida como las *ecuaciones normales* del problema de los mínimos cuadrados.

Las ecuaciones normales son más fáciles de trabajar en forma matricial. Primero, definamos la matriz de diseño $\check{A}$, como

$$\check{A} = \begin{bmatrix} Y_1(x_i)/\sigma_1 & Y_2(x_i)/\sigma_1 & \ldots \\ Y_1(x_i)/\sigma_2 & Y_2(x_i)/\sigma_2 & \ldots \\ \vdots & \vdots & \ddots \end{bmatrix} \ , \qquad A_{ij} = \frac{Y_j(x_i)}{\sigma_i} \ . \tag{10.25}$$

Note que la matriz de diseño no depende de y_i, los valores de datos, pero depende de x_i, esto es, sobre el diseño del experimento (donde las mediciones son tomadas). Usando la matriz de diseño, podemos escribir (10.24) como

$$\sum_{i=1}^{N} \sum_{k=1}^{M} A_{ij} A_{ik} a_k = \sum_{i=1}^{N} A_{ij} \frac{y_i}{\sigma_i^2} \ , \tag{10.26}$$

o en forma matricial,

$$(\check{A}^T \check{A}) \vec{a} = \check{A}^T \vec{b} \ , \tag{10.27}$$

donde el vector $\vec{b}$ está definido como $b_i = y_i/\sigma_i$ y $\check{A}^T$ es la traspuesta de la matriz de diseño. Esta ecuación es fácil de resolver para el parámetro vector $\vec{a}$,

$$\vec{a} = (\check{A}^T \check{A})^{-1} \check{A}^T \vec{b} \ . \tag{10.28}$$

Usando este resultado y la ley de propagación de errores, en la ecuación (10.12), encontramos que el error estimado en el parámetro a_j es

$$\sigma_{aj} = \sqrt{C_{jj}} \tag{10.29}$$

donde $\check{C} = (\check{A}^T \check{A})^{-1}$.

Una aplicación común de esta formulación general lineal de ajuste de mínimos cuadrados es el ajuste de los datos polinomiales,

$$Y(x; \{a_j\}) = a_1 + a_2 x + a_3 x^2 + \ldots + a_M x^{M-1} \ . \tag{10.30}$$

La matriz de diseño tiene elementos $A_{ij} = a_j x_I^{j-1}/\sigma_i$, el cual es un tipo de matriz de Vandermonde. Desafortunadamente estas matrices son notoriamente patológicas, por lo tanto sea cuidadoso para $M > 10$.

10.1.5. Bondades del ajuste.

Podemos fácilmente ajustar cada punto de dato si el número de parámetros, M, es igual al número de puntos de datos, N. En este caso ya sea que hemos construido una teoría trivial o no hemos tomado suficientes datos. En vez de eso, supongamos el escenario más común en el cual $N \gg M$. Porque cada dato de punto tiene un error, no esperemos que la curva pase exactamente a través de los datos. Sin embargo, preguntemos, con las barras de error dadas, ¿cuan probable es que la curva en efecto describa los datos?. Por supuesto, si no se tienen barras de error no hay nada que se pueda decir acerca de la bondad del ajuste.

El sentido común sugiere que si el ajuste es bueno, entonces el promedio de la diferencia debería ser aproximadamente igual a la barra de error,

$$| \, y_i - Y(x_i) \, | \approx \sigma_i \ . \tag{10.31}$$

Colocando esto dentro de nuestra definición para χ^2, la ecuación (10.3), da $\chi^2 \approx N$. Sin embargo, sabemos que entre más parámetros usamos, más fácil es ajustar la curva; el ajuste puede ser perfecto si $M = N$. Esto sugiere que nuestra regla práctica para que un ajuste sea bueno es

$$\chi^2 \approx N - M \ . \tag{10.32}$$

Por supuesto, este es sólo un crudo indicador, pero es mejor que una simple ojeada a la curva. Un análisis completo podría usar χ estadístico para asignar una probabilidad de que los datos sean ajustados por la curva.

Si encontramos que $\chi^2 \gg N - M$, entonces una de dos: no hemos usado una función apropiada, $Y(x)$, para nuestro ajuste de curva o las barras de errores, σ_I, son demasiado pequeñas. Por otra parte, si $\chi^2 \ll N - M$, el ajuste es tan espectacularmente bueno que podemos esperar que las barras de error sean realmente demasiado grandes.

Habitualmente la calidad de un ajuste se mide por el coeficiente de correlación adimensional que es un número que varía entre 0 y 1, correspondiendo a 1 un mejor ajuste.

$$r^2 = \frac{S(a_1 \sum y - a_2 \sum xy) - (\sum y)^2}{S \sum y^2 - (\sum y)^2} \ . \tag{10.33}$$

Capítulo 11

Integración numérica básica

versión 4.1, 4 de Diciembre del 2004

En este capítulo veremos algunos métodos básicos para realizar cuadraturas (evaluar integrales numéricamente).

11.1. Definiciones

Consideremos la integral

$$I = \int_a^b f(x)\, dx \ . \tag{11.1}$$

Nuestra estrategia para estimar I es evaluar $f(x)$ en unos cuantos puntos, y ajustar una curva simple a través de estos puntos. Primero, subdividamos el intervalo $[a, b]$ en N subintervalos. Definamos los puntos x_i como

$$x_0 = a, x_N = b, x_0 < x_1 < x_2 < \cdots < x_{N-1} < x_N \ .$$

La función $f(x)$ sólo se evalúa en estos puntos, de modo que usamos la notación $f_i \equiv f(x_i)$, ver figura 11.1.

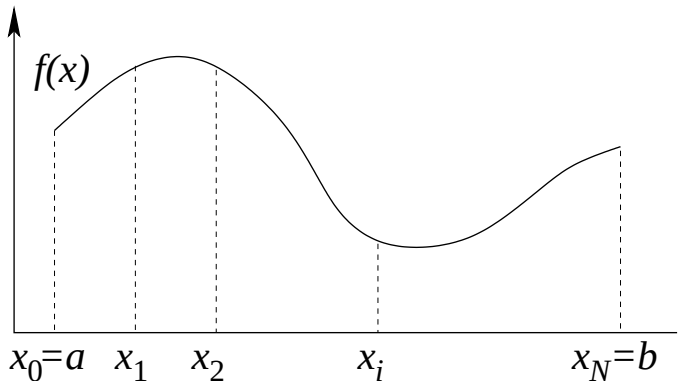


Figura 11.1: Subintervalos.

11.2. Regla trapezoidal

El método más simple de cuadratura es la *regla trapezoidal*. Conectamos los puntos f_i con líneas rectas, y la función lineal formada por la sucesión de líneas rectas es nuestra curva aproximante. La integral de esta función aproximante es fácil de calcular, pues es la suma de

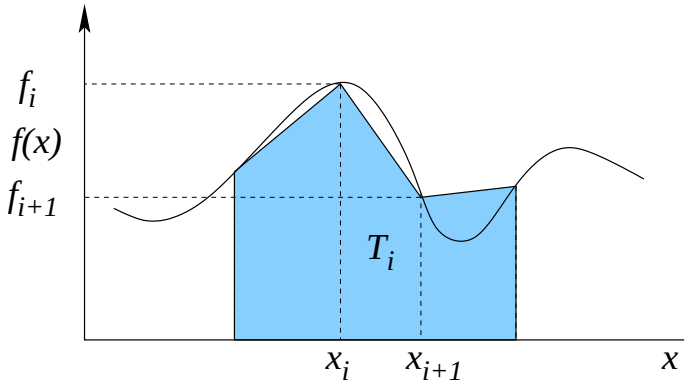


Figura 11.2: Sucesión de líneas rectas como curva aproximante.

las áreas de trapezoides. El área de un trapezoide es

$$T_i = \frac{1}{2}(x_{i+1} - x_i)(f_{i+1} + f_i) \ .$$

La integral es entonces calculada como la suma de las áreas de los trapezoides:

$$I \simeq I_T = T_0 + T_1 + \cdots + T_{N-1} \ .$$

Las expresiones son más simples si tomamos puntos equidistantes. En este caso el espaciado entre cada par de puntos es

$$h = \frac{b-a}{N} \ ,$$

de modo que $x_i = a + ih$. El área de un trapezoide es entonces

$$T_i = \frac{1}{2}h(f_{i+1} + f_i) \ ,$$

y la regla trapezoidal para puntos equiespaciados queda

$$\begin{aligned} I_T(h) &= \frac{1}{2}hf_0 + hf_1 + hf_2 + \cdots + hf_{N-1} + \frac{1}{2}hf_N \\ &= \frac{1}{2}h(f_0 + f_N) + h\sum_{i=1}^{N-1} f_i \end{aligned} \tag{11.2}$$

11.3. Interpolación con datos equidistantes.

Para el caso de puntos $x_i = a + ih$ que están igualmente espaciados podemos plantear un formalismo más general. Primero que todo, ajustemos un polinomio de grado N a un conjunto de $N+1$ datos x_i, f_i, no necesariamente equidistantes, de la forma:

$$P_N(x) = b_0 + b_1(x - x_0) + \cdots + b_N(x - x_0)(x - x_1)\cdots(x - x_{N-1}) \ . \tag{11.3}$$

Determinemos los coeficientes usando los datos:

$$\begin{aligned}
b_0 &= f_0 = f(x_0) \\
b_1 &= \frac{f_1 - f_0}{x_1 - x_0} \equiv f[x_1, x_0] \\
b_2 &= \left(\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0} \right) / (x_2 - x_0) = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0} \equiv f[x_2, x_1, x_0] \\
&\vdots \\
b_N &= f[x_N, x_{N-1}, x_{N-2}, \ldots, x_1, x_0] \ ,
\end{aligned}$$

donde hemos usado la definición recurrente

$$f[x_N, x_{N-1}, x_{N-2}, \ldots, x_1, x_0] \equiv \frac{f[x_N, x_{N-1}, x_{N-2}, \ldots, x_1] - f[x_{N-1}, x_{N-2}, \ldots, x_1, x_0]}{x_N - x_0} \ .$$

La expresión anterior es conocida como la n-ésima diferencia dividida finita. Usando estas definiciones podemos escribir el polinomio de interpolación (11.3) de la siguiente manera:

$$\begin{aligned}
P_N(x) = f_0 + f[x_1, x_0](x - x_0) + f[x_2, x_1, x_0](x - x_1)(x - x_0) + \cdots \\
+ f[x_N, x_{N-1}, x_{N-2}, \ldots, x_1, x_0](x - x_0)(x - x_1)\cdots(x - x_{N-1}) \ ,
\end{aligned}$$

Este polinomio es conocido como *el polinomio de interpolación por diferencias divididas de Newton*. Analicemos estos $f[\ldots]$ para el caso equiespaciado,

$$f[x_1, x_0] = \frac{f_1 - f_0}{x_1 - x_0} = \frac{\Delta f_0}{1! h} \ .$$

El segundo

$$f[x_2, x_1, x_0] = \frac{\dfrac{f_2 - f_1}{x_2 - x_1} - \dfrac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0} = \frac{f(x_2) - 2f(x_1) + f(x_0)}{2h^2} = \frac{\Delta^2 f_0}{2! h^2} \ .$$

En general tenemos (puede probarse por inducción) que

$$f[x_N, x_{N-1}, x_{N-2}, \ldots, x_1, x_0] = \frac{\Delta^n f_0}{n! h^n} \ .$$

Luego podemos escribir la llamada *fórmula hacia adelante de Newton-Gregory*, definiendo $\alpha = (x - x_0)/h$ de la siguiente manera

$$f_N(x) = f(x_0) + \Delta f(x_0)\alpha + \frac{\Delta^2 f(x_0)}{2!}\alpha(\alpha - 1) + \cdots + \frac{\Delta^n f(x_0)}{n!}\alpha(\alpha - 1)\cdots(\alpha - n + 1) + R_N \ , \tag{11.4}$$

el error de truncamiento viene dado por

$$R_N = \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1} \alpha(\alpha-1)(\alpha-2)\cdots(\alpha-n) \ ,$$

para algún $\xi \in [a,b]$.

11.4. Reglas de cuadratura

La integral (11.1) puede ser evaluada separándola en una suma de integrales sobre pequeños intervalos que contengan dos o tres puntos, luego usando el polinomio de interpolación adecuado para el número de puntos escogidos podemos evaluar aproximadamente cada integral. Para terminar el proceso debemos sumar todas las integrales parciales. Usemos sólo dos puntos, y entre esos dos puntos aproximamos la función por el polinomio de interpolación de dos puntos, *i.e.* por una recta, figura 11.3 (a).

$$I_i(h) = \int_{x_i}^{x_{i+1}} f(x)\,dx \approx \int_{x_i}^{x_{i+1}} \left(f(x_i) + \Delta f(x_i) \frac{x-x_i}{h} \right) dx = f(x_i)h + \frac{\Delta f(x_i)}{h} \frac{h^2}{2} \ ,$$

usando que $\Delta f(x_i) = f(x_{i+1}) - f(x_i)$ podemos dar una expresión para la integral

$$I_i(h) \approx \frac{f(x_{i+1}) + f(x_i)}{2}\, h \ . \tag{11.5}$$

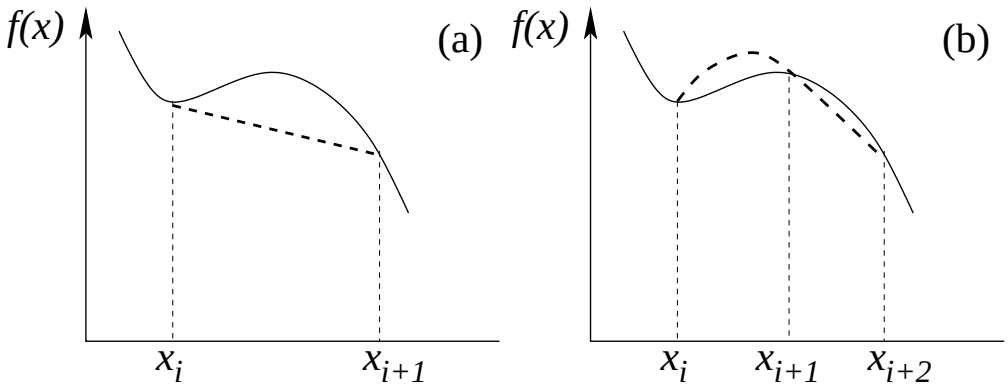


Figura 11.3: Aproximaciones, (a) lineal, (b) cuadrática.

Si consideramos intervalos que contengan tres puntos podemos usar la aproximación a segundo orden, es decir esta vez aproximamos nuestra función por un polinomio de segundo grado, figura 11.3 (b).

$$I_i(h) = \int_{x_i}^{x_{i+2}} f(x)\,dx \approx \int_{x_i}^{x_{i+2}} \left(f(x_i) + \Delta f(x_i) \frac{(x-x_i)}{h} + \frac{\Delta^2 f(x_i)}{2} \frac{(x-x_i)}{h} \frac{(x-x_i-h)}{h} \right) dx \ ,$$

integrando obtenemos

$$I_i(h) \approx 2hf(x_i) + \frac{\Delta f(x_i)}{h}\frac{4h^2}{2} + \frac{\Delta^2 f(x_i)}{2h^2}\left(\frac{8h^3}{3} - \frac{4h^3}{2}\right) ,$$

usando la expresión para $\Delta f(x_i) = f(x_{i+1}) - f(x_i)$ y para $\Delta^2 f(x_i) = f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)$ tenemos

$$I_i(h) \approx \frac{h}{3}\left(f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right) . \tag{11.6}$$

Se puede continuar encontrando expresiones usando aproximaciones de orden mayor, sin embargo, las fórmulas (11.5) y (11.6) son, sin duda, las más usadas.

A continuación, para evaluar la integral finita (11.1) debemos sumar las integrales parciales para ambos casos. Partamos sumando (11.5)

$$I = \int_a^b f(x)\,dx = \sum_{i=0}^{N-1}\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \sum_{i=0}^{N-1}\frac{f(x_{i+1}) - f(x_i)}{2}h \equiv I_T(h) ,$$

haciendo la suma tenemos

$$I_T(h) = \frac{h}{2}[f(a) + f(b)] + h\sum_{i=1}^{N-1} f(x_i) , \qquad \text{regla trapezoidal.} \tag{11.7}$$

Ahora evaluemos la integral finita, (11.1), usando la descomposición en integrales de tres puntos ecuación (11.6)

$$I = \int_a^b f(x)\,dx = \sum_{i=0}^{N-2}\int_{x_i}^{x_{i+2}} f(x)\,dx \approx \sum_{i=0}^{N-2}\frac{h}{3}[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})] \equiv I_s(h) ,$$

haciendo la suma tenemos

$$I_s(h) = \frac{2h}{3}\left[\frac{f(a)}{2} + 2f(x_1) + f(x_2) + 2f(x_3) + f(x_4) + 2f(x_5) + \ldots + \frac{f(b)}{2}\right] . \tag{11.8}$$

La cual es conocida como la regla de Simpson. Notemos que se necesita un número impar de puntos equiespaciados para aplicar esta regla.

Se puede estimar el error cometido en cada caso integrando el término (11.3) que corresponde al error de truncamiento de la expansión. Evaluemos el error para el caso trapezoidal en que cortamos la expansión a primer orden, por lo tanto el error corresponde

$$\epsilon_T(h) = \int_{x_i}^{x_{i+1}} \frac{d^2 f(\xi)}{dx^2}\frac{h^2}{2!}\frac{(x - x_i)}{h}\frac{(x - x_i - h)}{h}\,dx ,$$

haciendo el cambio de variable $u = x - x_i$ tenemos

$$\epsilon_T(h) = \frac{1}{2}\frac{d^2 f(\xi)}{dx^2}\int_0^h u(u - h)\,du = \left(\frac{h^3}{3} - \frac{h^3}{2}\right)\frac{1}{2}\frac{d^2 f(\xi)}{dx^2} = -\frac{h^3}{12}\frac{d^2 f(\xi)}{dx^2}$$

este es el error en cada integral entre x_i y x_{i+1}, para obtener el error total en la integral entre a y b tenemos que multiplicar por el número de intervalos (N),

$$I - I_T(h) = N\epsilon_T(h) = -N\frac{h^3}{12}\frac{d^2f(\xi)}{dx^2} = -\frac{(b-a)h^2}{12}\frac{d^2f(\xi)}{dx^2} = -\frac{h^2}{12}\left(f'(b) - f'(a)\right) \ .$$

La expresión para el error en el método trapezoidal nos queda

$$I - I_T(h) = -\frac{1}{12}h^2[f'(b) - f'(a)] + \mathcal{O}(h^4) \ . \tag{11.9}$$

Vemos que el error es proporcional a h^2, y que la regla trapezoidal tendrá problemas cuando la derivada diverge en los extremos del intervalo. Por ejemplo, la integral $\int_0^b \sqrt{x}\,dx$ es problemática. En el caso de la integral de Simpson se puede probar que el error es proporcional a h^4.

Como ejemplo, consideremos la función error

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-y^2}\,dy \ . \tag{11.10}$$

Para $x = 1$, $\mathrm{erf}(1) \simeq 0.842701$. La regla trapezoidal con $N = 5$ da 0.83837, que es correcto en los dos primeros decimales. La regla de Simpson, con los mismos 5 puntos, da 0.84274, con cuatro decimales correctos . Por supuesto, el integrando en este ejemplo es muy suave y bien comportado lo cual asegura que los métodos funcionaran bien.

11.5. Integración de Romberg

Una pregunta usual es cuántas subdivisiones del intervalo realizar. Un modo de decidir es repetir el cálculo con un intervalo más pequeño. Si la respuesta no cambia apreciablemente, la aceptamos como correcta (sin embargo, esto no evita que podamos ser engañados por funciones patológicas o escenarios inusuales). Con la regla trapezoidal, si el número de paneles es una potencia de dos, podemos dividir el tamaño del intervalo por dos sin tener que recalcular todos los puntos.

Definamos la secuencia de tamaños de intervalo,

$$h_1 = (b-a) \ , h_2 = \frac{1}{2}(b-a) \ , \ldots \ , h_n = \frac{1}{2^{n-1}}(b-a) \ . \tag{11.11}$$

Para $n = 1$ sólo hay un panel, luego

$$I_T(h_1) = \frac{1}{2}(b-a)[f(a) + f(b)] = \frac{1}{2}h_1[f(a) + f(b)] \ .$$

Para $n = 2$ se añade un punto interior, luego

$$\begin{aligned} I_T(h_2) &= \frac{1}{2}h_2[f(a) + f(b)] + h_2 f(a + h_2) \\ &= \frac{1}{2}I_T(h_1) + h_2 f(a + h_2) \ . \end{aligned}$$

Hay una fórmula recursiva para calcular $I_T(h_n)$ usando $I_T(h_{n-1})$:

$$I_T(h_n) = \frac{1}{2} I_T(h_{n-1}) + h_n \sum_{i=1}^{2^{n-2}} f[a + (2i-1)h_n] \ . \tag{11.12}$$

El segundo término del lado derecho da la contribución de los puntos interiores que se han agregado cuando el tamaño del intervalo es reducido a la mitad.

Usando el método recursivo descrito, podemos agregar paneles hasta que la respuesta parezca converger. Sin embargo, podemos mejorar notablemente este proceso usando un método llamado *integración de Romberg*. Primero veamos la mecánica, y después explicaremos por qué funciona. El método calcula los elementos de una matriz triangular:

$$R = \begin{matrix} R_{1,1} & - & - & - \\ R_{2,1} & R_{2,2} & - & - \\ R_{3,1} & R_{3,2} & R_{3,3} & - \\ \vdots & \vdots & \vdots & \ddots \end{matrix} \tag{11.13}$$

La primera columna es simplemente la regla trapezoidal recursiva:

$$R_{i,1} = I_T(h_i) \ . \tag{11.14}$$

Las sucesivas columnas a la derecha se calculan usando la fórmula de extrapolación de Richardson:

$$R_{i+1,j+1} = R_{i+1,j} + \frac{1}{4^j - 1}[R_{i+1,j} - R_{i,j}] \ . \tag{11.15}$$

La estimación más precisa para la integral es el elemento $R_{N,N}$.

El programa `romberg.cc` calcula la integral erf(1) usando el método de Romberg. Con $N = 3$, se obtiene la tabla

```
0.771743        0        0
0.825263 0.843103        0
0.838368 0.842736 0.842712
```

$R_{3,3}$ da el resultado exacto con 4 decimales, usando los mismos 4 paneles que antes usamos con la regla trapezoidal (y que ahora reobtenemos en el elemento $R_{3,1}$).

Es útil que el programa entregue toda la tabla y no sólo el último término, para tener una estimación del error. Como en otros contextos, usar una tabla demasiado grande puede no ser conveniente pues errores de redondeo pueden comenzar a degradar la respuesta.

Para entender por qué el esquema de Romberg funciona, consideremos el error para la regla trapezoidal, $E_T(h_n) = I - I_T(h_n)$. Usando (11.9),

$$E_T(h_n) = -\frac{1}{12} h_n^2 [f'(b) - f'(a)] + \mathcal{O}(h_n^4) \ .$$

Como $h_{n+1} = h_n/2$,

$$E_T(h_{n+1}) = -\frac{1}{48} h_n^2 [f'(b) - f'(a)] + \mathcal{O}(h_n^4) \ .$$

Consideremos ahora la segunda columna de la tabla de Romberg. El error de truncamiento para $R_{n+1,2}$ es:

$$\begin{aligned} I - R_{n+1,2} &= I - \left\{ I_T(h_{n+1}) + \frac{1}{3}[I_T(h_{n+1}) - I_T(h_n)] \right\} \\ &= E_T(h_{n+1}) + \frac{1}{3}[E_T(h_{n+1}) - E_T(h_n)] \\ &= -\left[\frac{1}{48} + \frac{1}{3}\left(\frac{1}{48} - \frac{1}{12} \right) \right] h_n^2 [f'(b) - f'(a)] + \mathcal{O}(h_n^4) \\ &= \mathcal{O}(h_n^4) \end{aligned}$$

Notemos cómo el término h_n^2 se cancela, dejando un error de truncamiento de orden h_n^4. La siguiente columna (la tercera) de la tabla de Romberg cancela este término, y así sucesivamente.

11.6. Cuadratura de Gauss.

Consideremos el problema de cuadratura sin considerar puntos fijos equidistantes, sino que pensamos en hacer una elección adecuada de puntos y pesos para cada punto, tal que nuestra integral pueda aproximarse como:

$$\int_a^b f(x)\,dx \approx w_1 f(x_1) + \ldots + w_N f(x_n)\ , \tag{11.16}$$

aquí los x_i son un conjunto de puntos elegidos de manera inteligente tal que disminuyan el error y los w_i son sus pesos respectivos, no necesariamente iguales unos con otros.

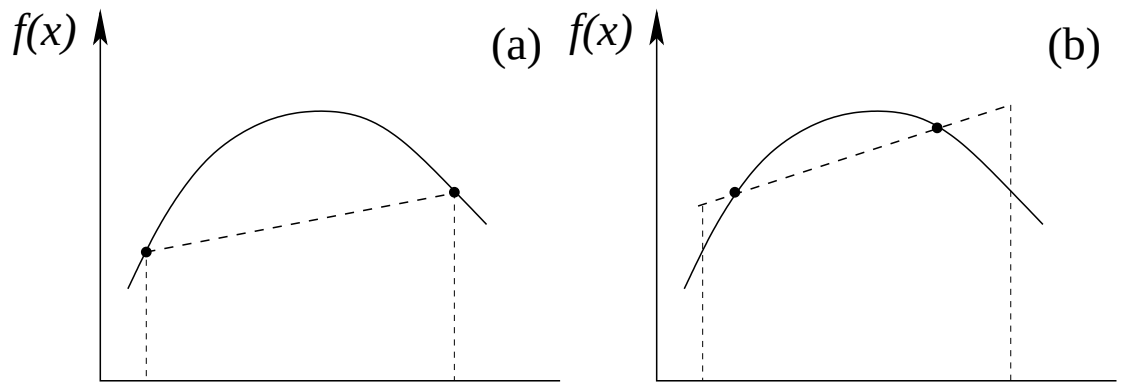


Figura 11.4: (a) Ilustración de la regla trapezoidal que une los dos puntos extremos por una recta. (B) Estimación mejorada al elegir inteligentemente los puntos.

El nombre que reciben esta clase de técnicas es *Cuadratura de Gauss*, la más común de ellas es la conocida como de Gauss-Legendre, y es útil para un intervalo finito, el cual es mapeado mediante un cambio de variables al intervalo $[-1, 1]$. Existen otras cuadraturas como la de Gauss-Laguerre óptima para integrales de la forma $\int_0^\infty e^{-x} f(x)\,dx$.

$\pm x_i$	w_i	$\pm x_i$	w_i
N=2		N=8	
0.57735 02692	1.00000 00000	0.18343 46425	0.36268 37834
N=3		0.52553 24099	0.31370 66459
0.00000 00000	0.88888 88889	0.79666 64774	0.22238 10345
0.77459 66692	0.55555 55556	0.96028 98565	0.10122 85363
N=4		N=12	
0.33998 10436	0.65214 51549	0.12523 34085	0.24914 70458
0.86113 63116	0.34785 48451	0.36783 14990	0.23349 25365
N=5		0.58731 79543	0.20316 74267
0.00000 00000	0.56888 88889	0.76990 26742	0.16007 83285
0.53846 93101	0.47862 86705	0.90411 72564	0.10693 93260
0.90617 98459	0.23692 68850	0.98156 06342	0.04717 53364

Cuadro 11.1: Puntos y pesos para integración de Gauss-Legendre.

Para evaluar (11.1), debemos mapear el intervalo $[a, b]$ en el intervalo $[-1, 1]$ mediante el cambio de variable $x = \frac{1}{2}(b + a) + \frac{1}{2}(b - a)z$, quedándonos

$$\int_a^b f(x)\, dx = \frac{b-a}{2} \int_{-1}^{1} f(z)\, dz \ , \tag{11.17}$$

esta última integral puede ser evaluada usando diferentes conjuntos de puntos y pesos,

$$\int_{-1}^{1} f(z)\, dz \approx w_1 f(x_1) + \ldots + w_N f(x_n) \ . \tag{11.18}$$

En la tabla 11.1 presentamos los conjuntos con $N = 2, 3, 4, 8$ y 12 puntos. Los puntos x_i corresponden a los ceros del Polinomio de Legendre P_N de grado N y los pesos vienen dados por

$$w_i = \frac{2}{(1 - x_i^2)[(d/dx)P_N(x_i)]^2} \quad 1 = 1, 2, \ldots, N$$

Como ejemplo, consideremos la función error

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-y^2}\, dy \ , \tag{11.19}$$

para $x = 1$, es decir,

$$\mathrm{erf}(1) = \int_0^1 \frac{2}{\sqrt{\pi}} e^{-y^2}\, dy \ . \tag{11.20}$$

Haciendo el cambio de variable sugerido $y = \frac{1}{2}(b + a) + \frac{1}{2}(b - a)z = \frac{1}{2} + \frac{z}{2}$ y $dy = dz/2$, la integral queda

$$\mathrm{erf}(1) \simeq 0.842701 = \int_{-1}^{1} \frac{1}{\sqrt{\pi}} e^{-(1+z)^2/4}\, dz = \int_{-1}^{1} g(z)\, dz \ . \tag{11.21}$$

Evaluemos con el conjunto de dos puntos

$$\mathrm{erf}(1) = \int_{-1}^{1} g(z)\,dz \simeq g(-0.57735\,02692) + g(+0.57735\,02692) = 0.84244189252125179\ , \tag{11.22}$$

tres decimales correctos. Ahora con el conjunto de tres puntos,

$$\begin{aligned}\mathrm{erf}(1) = \int_{-1}^{1} g(z)\,dz \simeq\ & 0.55555\,55556 \times g(-0.77459\,66692) \\ & + 0.88888\,88889 \times g(0.00000\,00000) \\ & + 0.55555\,55556 \times g(+0.77459\,66692) = 0.84269001852936587\ ,\end{aligned} \tag{11.23}$$

tres decimales correctos. Finalmente con el conjunto de cuatro puntos

$$\begin{aligned}\mathrm{erf}(1) = \int_{-1}^{1} g(z)\,dz \simeq\ & 0.34785\,48451 \times g(-0.86113\,63116) \\ & + 0.65214\,51549 \times g(-0.33998\,10436) \\ & + 0.65214\,51549\,10436 \times g(+0.33998\,10436) \\ & + 0.34785\,48451 \times g(+0.86113\,63116) = 0.84270117131761124\ ,\end{aligned} \tag{11.24}$$

los seis decimales correctos.

11.7. Bibliografía

- *Numerical Methods for Physics, second edition* de Alejandro L. Garcia, editorial Prentice Hall.
- *Métodos Numéricos Aplicados en Ingeniería* de Jean-Marie Ledanois, Aura López de Ramos, José Antonio Pimentel M. y Filipo F. Pironti Lubrano, editorial Mc Graw Hill.
- *Métodos Numéricos para Ingenieros* Steven C. Chapra y Raymond P. Canale, editorial Mc Graw Hill.

11.8. Listados del programa.

11.8.1. trapecio.cc

```
#include "NumMeth.h"

double integrando(double);

int main(){
  double a=0, b=1,x;
  int N;
  double h;
  double I, suma=0;
  cout << "Regla trapezoidal para erf(1)" << endl;
  cout << "Numero de puntos: " ;
  cin >> N;

  h = (b-a)/(N-1);
  for (int i=2;i<N;i++){
    x = a+(i-1)*h;
    suma+=integrando(x);
  }
  I = h*((integrando(a)+integrando(b))/2.0 + suma);
  cout << "Valor aproximado: erf(1) ~= " << I << endl;
  cout << "Valor exacto:     erf(1)  = 0.842701" << endl;
}

double integrando(double x){
  return (2.0/sqrt(M_PI))*exp(-x*x);
}
```

11.8.2. romberg.cc

```
#include "NumMeth.h"
#include <iomanip.h>

double integrando(double);

int main(){
  double a=0, b=1,x;
  int N,np;
  double h;
  double I, suma=0;
  cout << "Integracion de Romberg para erf(1)" << endl;
```

```
  cout << "Dimension de tabla: " ;
  cin >> N;
  double ** R = new double * [N];
  for (int i=0;i<N;i++){
    R[i] = new double[N];
  }

  for (int i=0;i<N;i++){
    for (int j=0;j<N;j++){
      R[i][j] = 0 ;
    }
  }

//   for (int i=0;i<N;i++){
//     for (int j=0;j<N;j++){
//       cout << R[i][j] << " ";
//     }
//     cout << endl;
//   }

  h = (b-a); // Ancho del intervalo
  np = 1;    // Numero de paneles
  for (int i=0;i<N;i++,h/=2,np*=2){

    if (i==0){
      R[0][0] = h*(integrando(a)+integrando(b))/2.0;
    }
    else {
      suma = 0;
      for (int j=1;j<=np-1;j+=2){
        x = a+j*h;
        suma += integrando(x);
      }
      R[i][0] = R[i-1][0]/2.0 + h*suma;
    }
  }

  int m = 1;
  for (int j=1;j<N;j++){
    m *= 4;
    for (int i=j;i<N;i++){
      R[i][j] = R[i][j-1] + (R[i][j-1]-R[i-1][j-1])/(m-1);
    }
  }
```

```
  cout << "Tabla R" << endl << endl;
  for (int i=0;i<N;i++){
    for (int j=0;j<N;j++){
      cout << /*setprecision(6) << */ setw(8) <<  R[i][j] << " ";
    }
    cout << endl;
  }

  cout << endl;
  cout << "Valor aproximado: erf(1) ~= " << R[N-1][N-1] << endl;
  cout << "Valor exacto:     erf(1)  = 0.842701" << endl;

  for (int i=0;i<N;i++){
    delete [] R[i];
  }
  delete [] R;
}

  double integrando(double x){
  return (2.0/sqrt(M_PI))*exp(-x*x);
}
```

Parte III

Apéndices.

Apéndice A

Transferencia a diskettes.

La filosofía de diferentes unidades (A:, B:,...) difiere de la estructura única del sistema de archivos que existe en UNIX. Son varias las alternativas que existen para la transferencia de información a diskette.

- Una posibilidad es disponer de una máquina WIN9X con ftp instalado y acceso a red. Empleando dicha aplicación se pueden intercambiar archivos entre un sistema y el otro.

- Existe un conjunto de comandos llamados `mtools` disponible en multitud plataformas, que permiten el acceso a diskettes en formato WIN9X de una forma muy eficiente.

  `mdir a:` Muestra el contenido de un diskette en `a:`.

  `mcopy file a:` Copia el archivo `file` del sistema de archivos UNIX en un diskette en `a:`.

  `mcopy a:file file` Copia el archivo `a:file` del diskette en el sistema de archivos UNIX con el nombre `file`.

  `mdel a:file` Borra el archivo `a:file` del diskette.

  Con `a:` nos referimos a la primera diskettera `/dev/fd0` y luego al archivo que se encuentra en el diskette. Su nombre se compone de `a:filename`. Si se desea emplear el caracter comodín para un conjunto de archivos del diskette, estos deben rodearse de dobles comillas para evitar la actuación del *shell* (p.e. `mcopy ''a:*.dat''`). La opción `-t` realiza la conversión necesaria entre UNIX y WIN9X, que se debe realizar **sólo** en archivos de texto.

- Una alternativa final es montar el dispositivo `/dev/fd0` en algún directorio, típicamente `/floppy`, considerando el tipo especial de sistema de archivos que posee `vfat` y luego copiar y borrar usando comandos UNIX. Esta forma suele estar restringida sólo a `root`, el comando: `mount -t vfat /dev/fd0 /floppy` no puede ser dado por un usuario. Sin embargo, el sistema aceptará el comando `mount /floppy` de parte del usuario. Una vez terminado el trabajo con el floppy éste debe ser desmontado, antes de sacarlo, mediante el comando: `umount /floppy`.

Apéndice B

Editores tipo emacs.

Los editores tipo emacs se parecen mucho y en su mayoría sus comandos son los mismos. Para ejemplificar este tipo de editores nos centraremos en `XEmacs`, pero los comandos y descripciones se aplican casi por igual a todos ellos. Los editores tipo emacs constan de tres zonas:

- La zona de edición: donde aparece el texto que está siendo editado y que ocupa la mayor parte de la pantalla.
- La zona de información: es una barra que esta situada en la penúltima línea de la pantalla.
- La zona de introducción de datos: es la última línea de la pantalla.

Emacs es un editor que permite la edición visual de un archivo (en contraste con el modo de edición de `vi`). El texto se agrega o modifica en la zona de edición, usando las teclas disponibles en el teclado.

Además, existen una serie de comandos disponibles para asistir en esta tarea.

La mayoría de los comandos de emacs se realizan empleando la tecla de `CONTROL` o la tecla `META`[1]. Emplearemos la nomenclatura: `C-key` para indicar que la tecla `key` debe de ser pulsada junto con `CONTROL` y `M-key` para indicar que la tecla `META` debe de ser pulsada junto a `key`. En este último caso NO es necesario pulsar simultáneamente las teclas `ESC` y `key`, pudiendo pulsarse secuencialmente `ESC` y luego `key`, sin embargo, si se usa `ALT` como `META` deben ser pulsadas simultáneamente. Observemos que en un teclado normal hay unos 50 caracteres (letras y números). Usando `SHIFT` se agregan otros 50. Así, usando `CONTROL` y `META`, hay unos $50 \cdot 4 = 200$ comandos disponibles. Además, existen comandos especiales llamados *prefijos*, que modifican el comando siguiente. Por ejemplo, `C-x` es un prefijo, y si `C-s` es un comando (de búsqueda en este caso), `C-x C-s` es otro (grabar archivo). Así, a través de un prefijo, se duplican el número de comandos disponibles sólo con el teclado, hasta llegar a unos $200 \cdot 2 = 400$ comandos en total.

Aparte de estos comandos accesibles por teclas, algunos de los cuales comentaremos a continuación, existen comandos que es posible ejecutar por nombre, haciendo así el número de comandos disponibles virtualmente infinito.

[1] Dado que la mayoría de los teclados actuales no poseen la tecla `META` se emplea ya sea `ESC` o `ALT`.

Revisemos los comandos más usuales, ordenados por tópico.

Abortar y deshacer

En cualquier momento, es posible abortar la operación en curso, o deshacer un comando indeseado:

C-g	abortar
C-x u	deshacer

Archivos

C-x C-f	cargar archivo
C-x i	insertar archivo
C-x C-s	grabar archivo
C-x C-w	grabar con nombre
C-x C-c	salir

Ventanas

`Emacs` permite dividir la pantalla en varias ventanas. En cada ventana se puede editar texto e ingresar comandos independientemente. Esto es útil en dos situaciones: a) si necesitamos editar un solo archivo, pero necesitamos ver su contenido en dos posiciones distintas (por ejemplo, el comienzo y el final de archivos muy grandes); y b) si necesitamos editar o ver varios archivos simultáneamente. Naturalmente, aunque son independientes, sólo es posible editar un archivo a la vez. A la ventana en la cual se encuentra el cursor en un momento dado le llamamos la "ventana actual".

C-x 2	dividir ventana actual en 2 partes, con línea horizontal
C-x 3	dividir ventana actual en 2 partes, con línea vertical
C-x 1	sólo 1 ventana (la ventana actual, eliminando las otras)
C-x 0	elimina sólo la ventana actual
C-x o	cambia el cursor a la siguiente ventana

El cambio del cursor a una ventana cualquiera se puede hacer también rápidamente a través del *mouse*.

Comandos de movimiento

Algunos de estos comandos tienen dos teclas asociadas, como se indica a continuación.

C-b o ←	izquierda un carácter	**C-f** o →	derecha un carácter
C-p o ↑	arriba una línea	**C-n** o ↓	abajo una línea
C-a o `Home`	principio de la línea	**C-e** o `End`	fin de la línea
M-< o **C-Home**	principio del documento	**M->** o **C-End**	fin del documento
M-f o **M-→**	avanza una palabra	**M-b** o **M-←**	retrocede una palabra
C-v o `Page Up`	avanza una página	**M-v** o `Page Down`	retrocede una página
M-g (número)	salta a la línea (número)	**C-l**	refresca la pantalla

Comandos de inserción y borrado

Al ser un editor en modo visual, las modificaciones se pueden hacer en el texto sin necesidad de entrar en ningún modo especial.

C-d o `Delete`	borra un carácter después del cursor
`Backspace`	borra un carácter antes del cursor
C-k	borra desde la posición del cursor hasta el fin de línea (no incluye el cambio de línea)
M-d	borra desde el cursor hacia adelante, hasta que termina una palabra
M-Backspace	borra desde el cursor hacia atrás, hasta que comienza una palabra
C-o	Inserta una línea en la posición del cursor

Mayúsculas y minúsculas

M-u	Cambia a mayúscula desde la posición del cursor hasta el fin de la palabra
M-l	Cambia a minúscula desde la posición del cursor hasta el fin de la palabra
M-c	Cambia a mayúscula el carácter en la posición del cursor y a minúscula hasta el fin de la palabra

Por ejemplo, veamos el efecto de cada uno de estos comandos sobre la palabra `EmAcS`, si el cursor está sobre la letra `E` (¡el efecto es distinto si está sobre cualquier otra letra!):

M-u : `EmAcS` ⟶ `EMACS`
M-l : `EmAcS` ⟶ `emacs`
M-c : `EmAcS` ⟶ `Emacs`

Transposición

Los siguientes comandos toman como referencia la posición actual del cursor. Por ejemplo, **C-t** intercambia el carácter justo antes del cursor con el carácter justo después.

C-t	Transpone dos caracteres
M-t	Transpone dos palabras
C-x C-t	Transpone dos líneas

Búsqueda y reemplazo

C-s	Búsqueda hacia el fin del texto
C-r	Búsqueda hacia el inicio del texto
M- %	Búsqueda y sustitución (pide confirmación cada vez)
M-&	Búsqueda y sustitución (sin confirmación)

Definición de regiones y reemplazo

Uno de los conceptos importantes en `emacs` es el de región. Para ello, necesitamos dos conceptos auxiliares: el *punto* y la *marca*. El punto es simplemente el cursor. Específicamente, es el punto donde *comienza* el cursor. Así, si el cursor se encuentra sobre la letra `c` en `emacs`, el punto está entre la `a` y la `c`. La marca, por su parte, es una señal que se coloca en algún punto del archivo con los comandos apropiados. La *región* es el espacio comprendido entre el punto y la marca.

Para colocar una marca basta ubicar el cursor en el lugar deseado, y teclear **C-**`Space` o **C-@**. Esto coloca la marca donde está el punto (en el ejemplo del párrafo anterior, quedaría entre las letras `a` y `c`. Una vez colocada la marca, podemos mover el cursor a cualquier otro lugar del archivo (hacia atrás o hacia adelante respecto a la marca). Esto define una cierta ubicación para el punto, y, por tanto, queda definida la región automáticamente.

La región es una porción del archivo que se puede manipular como un todo. Una región se puede borrar, copiar, pegar en otro punto del archivo o incluso en otro archivo; una región se puede imprimir, grabar como un archivo distinto; etc. Así, muchas operaciones importantes se pueden efectuar sobre un bloque del archivo.

Por ejemplo, si queremos duplicar una región, basta con definir la región deseada (poniendo la marca y el punto donde corresponda) y teclear **M-w**. Esto copia la región a un buffer temporal (llamado *kill buffer*). Luego movemos el cursor al lugar donde queremos insertar el texto duplicado, y hacemos **C-y**. Este comando toma el contenido del *kill buffer* y lo inserta en el archivo. El resultado final es que hemos duplicado una cierta porción del texto.

Si la intención era mover dicha porción, el procedimiento es el mismo, pero con el comando **C-w** en vez de **M-w**. **C-w** también copia la región a un *kill buffer*, pero borra el texto de la pantalla.

Resumiendo:

C-`Space` o **C-@**	Comienzo de región
M-w	Copia región
C-w	Corta región
C-y	Pega región

El concepto de *kill buffer* es mucho más poderoso que lo explicado recién. En realidad, muchos comandos, no sólo **M-w** y **C-w**, copian texto en un *kill buffer*. En general, cualquier comando que borre más de un carácter a la vez, lo hace. Por ejemplo, **C-k** borra una línea. Lo que hace no es sólo borrarla, sino además copiarla en un *kill buffer*. Lo mismo ocurre con los comandos que borran palabras completas (**M-d**, **M-**`Backspace`), y muchos otros. Lo

interesante es que **C-y** funciona también en todos esos casos: **C-y** lo único que hace es tomar el último texto colocado en un *kill buffer* (resultado de la última operación que borró más de un carácter a la vez), y lo coloca en el archivo. Por lo tanto, no sólo podemos copiar o mover "regiones", sino también palabras o líneas. Más aún, el *kill buffer* no es borrado con el **C-y**, así que ese mismo texto puede ser duplicado muchas veces. Continuará disponible con **C-y** mientras no se ponga un nuevo texto en el *kill buffer*.

Además, `emacs` dispone no de uno sino de muchos *kill buffers*. Esto permite recuperar texto borrado hace mucho rato. En efecto, cada vez que se borra más de un carácter de una vez, se una un nuevo *kill buffer*. Por ejemplo, consideremos el texto:

```
La primera linea del texto,
la segunda linea,
y finalmente la tercera.
```

Si en este párrafo borramos la primera línea (con **C-k**), después borramos la primera palabra de la segunda (con **M-d**, por ejemplo), y luego la segunda palabra de la última, entonces habrá tres *kill buffers* ocupados:

buffer 1 : `La primera linea del texto,`
buffer 2 : `la`
buffer 3 : `finalmente`

Al colocar el cursor después del punto final, **C-y** toma el contenido del último *kill buffer* y lo coloca en el texto:

```
 segunda linea,
y  la tercera. finalmente
```

Si se teclea ahora **M-y**, el último texto recuperado, `finalmente`, es reemplazado por el penúltimo texto borrado, y que está en el *kill buffer* anterior:

```
 segunda linea,
y  la tercera. la
```

Además, la posición de los *kill buffers* se rota:

buffer 1 : `finalmente`
buffer 2 : `La primera linea del texto,`
buffer 3 : `la`

Sucesivas aplicaciones de **M-y** después de un **C-y** rotan sobre todos los *kill buffers* (que pueden ser muchos). El editor, así, conserva un conjunto de las últimas zonas borradas durante la edición, pudiendo recuperarse una antigua a pesar de haber seleccionado una nueva zona, o borrado una nueva palabra o línea. Toda la información en los *kill buffers* se pierde al salir de `emacs` (**C-c**).

Resumimos entonces los comandos para manejo de los *kill buffers*:

C-y	Copia el contenido del último *kill buffer* ocupado
M-y	Rota los *kill buffers* ocupados

Definición de macros

La clave de la configurabilidad de `emacs` está en la posibilidad de definir nuevos comandos que modifiquen su comportamiento o agreguen nuevas funciones de acuerdo a nuestras necesidades. Un modo de hacerlo es a través del archivo de configuración `$HOME/.emacs`, para lo cual se sugiere leer la documentación disponible en la distribución instalada. Sin embargo, si sólo necesitamos un nuevo comando en la sesión de trabajo actual, un modo más simple es definir una *macro*, un conjunto de órdenes que son ejecutados como un solo comando. Los comandos relevantes son:

C-x (	Comienza la definición de una macro
C-x)	Termina la definición de una macro
C-x e	Ejecuta una macro definida

Todas las sucesiones de teclas y comandos dados entre **C-x (** y **C-x)** son recordados por `emacs`, y después pueden ser ejecutados de una vez con **C-x e**.

Como ejemplo, consideremos el siguiente texto, con los cinco primeros lugares del ránking ATP (sistema de entrada) al 26 de marzo de 2002:

```
1 hewitt, lleyton (Aus)
2 kuerten, gustavo (Bra)
3 ferrero, juan (Esp)
4 kafelnikov, yevgeny (Rus)
5 haas, tommy (Ger)
```

Supongamos que queremos: (a) poner los nombres y apellidos con mayúscula (como debería ser); (b) poner las siglas de países sólo en mayúsculas.

Para definir una macro, colocamos el cursor al comienzo de la primera línea, en el `1`, y damos **C-x (**. Ahora realizamos todos los comandos necesarios para hacer las tres tareas solicitadas para el primer jugador solamente: **M-f** (avanza una palabra, hasta el espacio antes de `hewitt`; **M-c M-c** (cambia a `Hewitt, Lleyton`); **M-u** (cambia a `AUS`); `Home` (vuelve el cursor al comienzo de la línea); ↓ (coloca el cursor al comienzo de la línea siguiente, en el `2`). Los dos últimos pasos son importantes, porque dejan el cursor en la posición correcta para ejecutar el comando nuevamente. Ahora terminamos la definición con **C-x)**. Listo. Si ahora ejecutamos la macro, con **C-x e**, veremos que la segunda línea queda modificada igual que la primera, y así podemos continuar hasta el final:

```
1 Hewitt, Lleyton (AUS)
2 Kuerten, Gustavo (BRA)
3 Ferrero, Juan (ESP)
4 Kafelnikov, Yevgeny (RUS)
5 Haas, Tommy (GER)
```

Comandos por nombre

Aparte de los ya comentados existen muchas otras órdenes que no tienen necesariamente una tecla asociada (*bindkey*) asociada. Para su ejecución debe de teclearse previamente:

`M-x`

y a continuación en la zona inferior de la pantalla se introduce el comando deseado. Empleando el `TAB` se puede completar dicho comando (igual que en `bash`).

De hecho, esto sirve para cualquier comando, incluso si tiene tecla asociada. Por ejemplo, ya sabemos **M-g** `n` va a la línea n del documento. Pero esto no es sino el comando `goto-line`, y se puede también ejecutar tecleando: `M-x goto-line n`.

Repetición

Todos los comandos de `emacs`, tanto los que tienen una tecla asociada como los que se ejecutan con nombre, se pueden ejecutar más de una vez, anteponiéndoles un argumento numérico con

`M-(number)`

Por ejemplo, si deseamos escribir 20 letras `e`, basta teclear `M-20 e`. Esto es particularmente útil con las macros definidos por el usuario. En el ejemplo anterior, con el ránking ATP, después de definir la macro quedamos en la línea 2, y en vez de ejecutar **C-x e** 4 veces, podemos teclear `M-4 C-x e`, con el mismo resultado, pero en mucho menos tiempo.

Para terminar la discusión de este editor, diremos que es conveniente conocer las secuencias de control básico de `emacs`:
`C-a, C-e, C-k, C-y, C-w, C-t, C-d`, etc.,
porque funcionan para editar la línea de comandos en el *shell*, como también en muchos programas de texto y en ventanas de diálogo de las aplicaciones *X Windows*. A su vez, los editores `jed`, `xjed`, `jove` también usan por defecto estas combinaciones.

Apéndice C

Una breve introducción a Octave/Matlab

C.1. Introducción

Octave es un poderoso software para análisis numérico y visualización. Muchos de sus comandos son compatibles con Matlab. En estos apuntes revisaremos algunas características de estos programas. En realidad, el autor de este capítulo ha sido usuario durante algunos años de Matlab, de modo que estos apuntes se han basado en ese conocimiento, considerando los comandos que le son más familiares de Matlab. En la mayoría de las ocasiones he verificado que los comandos descritos son también compatibles con Octave, pero ocasionalmente se puede haber omitido algo....

Matlab es una abreviación de *Matrix Laboratory*. Los elementos básicos con los que se trabaja con matrices. Todos los otros tipos de variables (vectores, texto, polinomios, etc.), son tratados como matrices. Esto permite escribir rutinas optimizadas para el trabajo con matrices, y extender su uso a todos los otros tipos de variables fácilmente.

C.2. Interfase con el programa

Con Octave/Matlab se puede interactuar de dos modos: un modo interactivo, o a través de *scripts*. Al llamar a Octave/Matlab (escribiendo `octave` en el prompt, por ejemplo), se nos presenta un prompt. Si escribimos `a=1`, el programa responderá `a=1`. Alternativamente, podemos escribir `a=3;` (con punto y coma al final), y el programa no responderá (elimina el eco), pero almacena el nuevo valor de `a`. Si a continuación escribimos `a`, el programa responderá `a=3`. Hasta este punto, hemos usado el modo interactivo.

Alternativamente, podemos introducir las instrucciones anteriores en un archivo, llamado, por ejemplo, `prueba.m`. En el prompt, al escribir `prueba`, y si nuestro archivo está en el path de búsqueda del programa, las líneas de `prueba.m` serán ejecutadas una a una. Por ejemplo, si el archivo consta de las siguientes cuatro líneas:

```
a=3;
a
```

```
a=5
a
```

el programa responderá con

```
a=3
a=5
a=5
```

`prueba.m` corresponde a un *script*. Todas las instrucciones de Octave/Matlab pueden ejecutarse tanto en modo interactivo como desde un *script*. En Linux se puede ejecutar un archivo de comandos Octave de modo *stand-alone* incluyendo en la primera línea:
`#!/usr/bin/octave -q`.

C.3. Tipos de variables

C.3.1. Escalares

A pesar de que éstos son sólo un tipo especial de matrices (ver subsección siguiente), conviene mencionar algunas características específicas.

- Un número sin punto decimal es tratado como un entero exacto. Un número con punto decimal es tratado como un número en doble precisión. Esto puede no ser evidente en el output. Por *default*, `8.4` es escrito en pantalla como `8.4000`. Tras la instrucción `format long`, sin embargo, es escrito como `8.40000000000000`. Para volver al formato original, basta la instrucción `format`.
- Octave/Matlab acepta números reales y complejos. La unidad imaginaria es `i`: `8i` y `8*i` definen el mismo número complejo. Como `i` es una varible habitualmente usada en iteraciones, también está disponible `j` como un sinónimo. Octave/Matlab distinguen entre mayúsculas y minúsculas.
- Octave/Matlab representa de manera especial los infinitos y cantidades que no son números. `inf` es infinito, y `NaN` es un no-número (Not-a-Number). Por ejemplo, escribir `a=1/0` no arroja un error, sino un mensaje de advertencia, y asigna a `a` el valor `inf`. Análogamente, `a=0/0` asigna a `a` el valor `NaN`.

C.3.2. Matrices

Este tipo de variable corresponde a escalares, vectores fila o columna, y matrices convencionales.

Construcción

Las instrucciones:

```
a = [1 2 ; 3 4]
```

ó

```
a = [1, 2; 3, 4]
```

definen la matriz $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. Las comas (opcionales) separan elementos de columnas distintas, y los punto y coma separan elementos de filas distintas. El vector fila $(1 \quad 2)$ es

```
b = [1 2]
```

y el vector columna $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ es

```
c = [1;2]
```

Un número se define simplemente como `d = [3]` ó `d = 3`.

Nota importante: Muchas funciones de Octave/Matlab en las páginas siguientes aceptan indistintamente escalares, vectores filas, vectores columnas, o matrices, y su output es un escalar, vector o matriz, respectivamente. Por ejemplo, `log(a)` es un vector fila si `a` es un vector fila (donde cada elemento es el logaritmo natural del elemento correspondiente en `a`), y un vector columna si `a` es un vector columna. En el resto de este manual *no se advertira* este hecho, y se pondrán ejemplos con un solo tipo de variable, en el entendido que el lector está conciente de esta nota.

Acceso y modificación de elementos individuales

Accesamos los elementos de cada matriz usando los índices de filas y columnas, que parten de uno. Usando la matriz `a` antes definida, `a(1,2)` es 2. Para modificar un elemento, basta escribir, por ejemplo, `a(2,2) = 5`. Esto convierte a la matriz en $\begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}$. En el caso especial de vectores filas o columnas, basta un índice. (En los ejemplos anteriores, `b(2)` = `c(2)` = 2.)

Una característica muy importante del programa es que toda matriz es redimensionada automáticamente cuando se intenta modificar un elemento que sobrepasa las dimensiones actuales de la matriz, llenando con ceros los lugares necesarios. Por ejemplo, si `b = [1 2]`, y en seguida intentamos la asignación `b(5) = 8`, `b` es automáticamente convertido al vector fila de 5 elementos `[1 2 0 0 8]`.

Concatenación de matrices

Si $a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, $b = \begin{pmatrix} 5 & 6 \end{pmatrix}$, $c = \begin{pmatrix} 7 \\ 8 \end{pmatrix}$, entonces

```
d = [a c]
```

$d = \begin{pmatrix} 1 & 2 & 7 \\ 3 & 4 & 8 \end{pmatrix}$

```
d = [a; b]
```

$d = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$

```
d = [a [0; 0] c]
```

$$d = \begin{pmatrix} 1 & 2 & 0 & 7 \\ 3 & 4 & 0 & 8 \end{pmatrix}$$

C.3.3. Strings

Las cadenas de texto son casos particulares de vectores fila, y se construyen y modifican de modo idéntico.

Construcción

Las instrucciones

```
t = ['un buen texto']
t = ["un buen texto"]
t = 'un buen texto'
t = "un buen texto"
```

definen el mismo string `t`.

Acceso y modificación de elementos individuales

```
r = t(4)
```

```
r = 'b'
t(9) = 's'

texto = 'un buen sexto'
```

Concatenación

```
t = 'un buen texto';
t1 = [t ' es necesario']

t1 = 'un buen texto es necesario'
```

C.3.4. Estructuras

Las estructuras son extensiones de los tipos de variables anteriores. Una estructura consta de distintos campos, y cada campo puede ser una matriz (es decir, un escalar, un vector o una matriz), o una string.

Construcción

Las líneas

```
persona.nombre = 'Eduardo'
persona.edad = 30
persona.matriz_favorita = [2 8;10 15];
```

definen una estructura con tres campos, uno de los cuales es un string, otro un escalar, y otro una matriz:

```
persona =
{
nombre = 'Eduardo';
edad = 30;
matriz_favorita = [2 8; 10 15];
}
```

Acceso y modificación de elementos individuales

```
s = persona.nombre
```

```
 s = 'Eduardo'
```

```
persona.nombre = 'Claudio'
persona.matriz_favorita(2,1) = 8

persona =
{
nombre = 'Claudio';
edad = 30;
matriz_favorita = [2 8; 8 15];
}
```

C.4. Operadores básicos

C.4.1. Operadores aritméticos

Los operadores `+`, `-`, `*` corresponden a la suma, resta y multiplicación convencional de matrices. Ambas matrices deben tener la misma dimensión, a menos que una sea un escalar. Un escalar puede ser sumado, restado o multiplicado de una matriz de cualquier dimensión.

`.*` y `./` permiten multiplicar y dividir elemento por elemento. Por ejemplo, si

$$a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad b = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

entonces

```
c = a.*b
```

$$c = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

```
c = a./b
```

$$c = \begin{pmatrix} 0.2 & 0.3333 \\ 0.42857 & 0.5 \end{pmatrix}$$

Si `b` es un escalar, `a.*b` y `a./b` equivalen a `a*b` y `a/b`.

`a^b` es `a` elevado a `b`, si `b` es un escalar. `a.^b` eleva cada elemento de `a` a `b`.

`a'` es la matriz $\mathtt{a}^\dagger$ (traspuesta y conjugada)

`a.'` es la matriz traspuesta de `a`.

C.4.2. Operadores relacionales

Los siguientes operadores están disponibles:

`<  <=  >  >=  ==  ~=`

El resultado de estas operaciones es 1 (verdadero) ó 0 (falso). Si uno de los operandos es una matriz y el otro un escalar, se compara el escalar con cada elemento de la matriz. Si ambos operandos son matrices, el test se realiza elemento por elemento; en este caso, las matrices deben ser de igual dimensión. Por ejemplo,

```
a = [1 2 3];
b = [4 2 1];
c = (a<3);
d = (a>=b);
```

$c = (1, 1, 0)$
$d = (0, 1, 1)$

C.4.3. Operadores lógicos

Los siguientes símbolos corresponden a los operadores `AND`, `OR` y `NOT`:

`&  |  ~`

El resultado de estas operaciones es 1 (verdadero) ó 0 (falso).

C.4.4. El operador :

Es uno de los operadores fundamentales. Permite crear vectores y extraer submatrices.

: crea vectores de acuerdo a las siguientes reglas:

`j:k`	es lo mismo que `[j,j+1,...,k]`, si `j<=k`.
`j:i:k`	es lo mismo que `[j,j+i,j+2*i,...,k]`, si `i>0` y `j<k`, o si `i<0` y `j>k`.

: extrae submatrices de acuerdo a las siguientes reglas:

`A(:,j)`	es la `j`-ésima columna de `A`.
`A(i,:)`	es la `i`-ésima fila de `A`.
`A(:,:)`	es `A`.
`A(:,j:k)`	es `A(:,j)`, `A(:,j+1)`, ..., `A(:,k)`.
`A(:)`	son todos los elementos de `A`, agrupados en una única columna.

C.4.5. Operadores de aparición preferente en scripts

Los siguientes operadores es más probable que aparezcan durante la escritura de un *script* que en modo interactivo.

`%` : Comentario. El resto de la línea es ignorado.

`...` : Continuación de línea. Si una línea es muy larga y no cabe en la pantalla, o por alguna otra razón se desea dividir una línea, se puede usar el operador `...` . Por ejemplo,

```
m = [1 2 3 ...
4 5 6];
```

es equivalente a

```
m = [1 2 3 4 5 6];
```

C.5. Comandos matriciales básicos

Antes de revisar una a una diversas familias de comandos disponibles, y puesto que las matrices son el elemento fundamental en Octave/Matlab, en esta sección reuniremos algunas de las funciones más frecuentes sobre matrices, y cómo se realizan en Octave/Matlab.

Op. aritmética	`+`, `-`, `*`, `.*`, `./`	(ver subsección C.4.1)
Conjugar	`conj(a)`	
Trasponer	`a.'`	
Trasponer y conjugar	`a'`	
Invertir	`inv(a)`	
Autovalores, autovectores	`[v,d]=eig(a)`	(ver subsección C.6.5)
Determinante	`det(a)`	
Extraer elementos	`:`	(ver subsección C.4.4)
Traza	`trace(a)`	
Dimensiones	`size(a)`	
Exponencial	`exp(a)`	(elemento por elemento)
	`expm(a)`	(exponencial matricial)

C.6. Comandos

En esta sección revisaremos diversos comandos de uso frecuente en Octave/Matlab. Esta lista no pretende ser exhaustiva (se puede consultar la documentación para mayores detalles), y está determinada por mi propio uso del programa y lo que yo considero más frecuente debido a esa experiencia. Insistimos en que ni la lista de comandos es exhaustiva, ni la lista de ejemplos o usos de cada comando lo es. Esto pretende ser sólo una descripción de los aspectos que me parecen más importantes o de uso más recurrente.

C.6.1. Comandos generales

`clear` Borra variables y funciones de la memoria

`clear`	Borra todas las variables en memoria
`clear a`	Borra la variable `a`

`disp` Presenta matrices o texto

`disp(a)` presenta en pantalla los contenidos de una matriz, sin imprimir el nombre de la matriz. `a` puede ser una string.

```
disp('     c1       c2');
disp([.3 .4]);
```

$c1$	$c2$
0.30000	0.40000

`load, save` Carga/Guarda variables desde el disco

`save fname a b`	Guarda las variables `a` y `b` en el archivo `fname`
`load fname`	Lee el archivo `fname`, cargando las definiciones de variables en él definidas.

`size,length` Dimensiones de una matriz/largo de un vector

Si a es una matrix de $n \times m$:

`d = size(a)`	`d = [m,n]`
`[m,n] = size(a)`	Aloja en `m` el número de filas, y en `n` el de columnas

Si b es un vector de n elementos, `length(b)` es n.

`who` Lista de variables en memoria

`quit` Termina Octave/Matlab

C.6.2. Como lenguaje de programación

Control de flujo

`for`

```
n=3;                                    a=[1 4 9]
for i=1:n
    a(i)=i^2;
end
```

Para Octave el vector resultante es columna en vez de fila.

Observar el uso del operador `:` para generar el vector `[1 2 3]`. Cualquier vector se puede utilizar en su lugar: `for i=[2 8 9 -3]`, `for i=10:-2:1` (equivalente a `[10 8 6 4 2]`), etc. son válidas. El ciclo `for` anterior se podría haber escrito en una sola línea así:

```
for i=1:n, a(i)=i^2; end
```

`if, elseif, else`

Ejemplos:

a)
```
if a~=b, disp(a); end
```

b)
```
if a==[3 8 9 10]
   b = a(1:3);
end
```

c)
```
if a>3
   clear a;
elseif a<0
   save a;
else
   disp('Valor de a no considerado');
end
```

Naturalmente, `elseif` y `else` son opcionales. En vez de las expresiones condicionales indicadas en el ejemplo pueden aparecer cualquier función que dé valores `1` (verdadero) ó `0` (falso).

`while`

```
while s
    comandos
end
```

Mientras `s` es `1`, se ejecutan los `comandos` entre `while` y `end`. `s` puede ser cualquier expresión que dé por resultado `1` (verdadero) ó `0` (falso).

`break`

Interrumpe ejecución de ciclos `for` o `while`. En *loops* anidados, `break` sale del más interno solamente.

Funciones lógicas

Además de expresiones construidas con los operadores relacionales `==`, `<=`, etc., y los operadores lógicos `&`, `|` y `~`, los comandos de control de flujo anteriores admiten cualquier función cuyo resultado sea `1` (verdadero) ó `0` (falso). Particularmente útiles son funciones como las siguientes:

`all(a)`	`1` si todos los elementos de `a` son no nulos, y `0` si alguno es cero
`any(a)`	`1` si alguno de los elementos de `a` es no nulo
`isempty(a)`	`1` si `a` es matriz vacía (`a=[]`)

Otras funciones entregan matrices de la misma dimensión que el argumento, con unos o ceros en los lugares en que la condición es verdadera o falsa, respectivamente:

`finite(a)`	1 donde `a` es finito (no `inf` ni `NaN`)
`isinf(a)`	1 donde `a` es infinito
`isnan(a)`	1 donde `a` es un `NaN`

Por ejemplo, luego de ejecutar las líneas

```
x = [-2 -1 0 1 2];
y = 1./x;
a = finite(y);
b = isinf(y);
c = isnan(y);
```

se tiene

```
a = [1 1 0 1 1]
b = [0 0 1 0 0]
c = [0 0 0 0 0]
```

Otra función lógica muy importante es `find`:

`find(a)`	Encuentra los índices de los elementos no nulos de `a`.

Por ejemplo, si ejecutamos las líneas

```
x=[11 0 33 0 55];
z1=find(x);
z2=find(x>0 & x<40);
```

obtendremos

```
z1 = [1 3 5]
z2 = [1 3]
```

`find` también puede dar dos resultados de salida simultáneamente (más sobre esta posibilidad en la sección C.6.2), en cuyo caso el resultado son los pares de índices (índices de fila y columna) para cada elemento no nulo de una matriz

```
y=[1 2 3 4 5;6 7 8 9 10];
[z3,z4]=find(y>8);
```

da como resultado

```
z3 = [2;2];
z4 = [4;5];
```

`z3` contiene los índice de fila y `z4` los de columna para los elementos no nulos de la matriz `y>8`. Esto permite construir, por ejemplo, la matriz `z5=[z3 z4]` $= \begin{pmatrix} 2 & 4 \\ 2 & 5 \end{pmatrix}$, en la cual cada fila es la posición de `y` tal que la condición `y>8` es verdadera (en este caso, es verdadera para los elementos `y(2,4)` e `y(2,5)`).

Funciones definidas por el usuario

Octave/Matlab puede ser fácilmente extendido por el usuario definiendo nuevas funciones que le acomoden a sus propósitos. Esto se hace a través del comando `function`.

Podemos definir (en modo interactivo o dentro de un *script*), una función en la forma

```
function nombre (argumentos)
   comandos
endfunction
```

`argumentos` es una lista de argumentos separados por comas, y `comandos` es la sucesión de comandos que serán ejecutados al llamar a `nombre`. La lista de argumentos es opcional, en cuyo caso los paréntesis redondos se pueden omitir.

A mediano y largo plazo, puede ser mucho más conveniente definir las funciones en archivos especiales, listos para ser llamados en el futuro desde modo interactivo o desde cualquier *script*. Esto se hace escribiendo la definición de una función en un *script* con extensión `.m`. Cuando Octave/Matlab debe ejecutar un comando o función que no conoce, por ejemplo, `suma(x,y)`,busca en los archivos accesibles en su path de búsqueda un archivo llamado `suma.m`, lo carga y ejecuta la definición contenida en ese archivo.

Por ejemplo, si escribimos en el *script* `suma.m` las líneas

```
function s=suma(x,y)
s = x+y;
```

el resultado de `suma(2,3)` será `5`.

Las funciones así definidas pueden entregar más de un argumento si es necesario (ya hemos visto algunos ejemplos con `find` y `size`). Por ejemplo, definimos una función que efectúe un análisis estadístico básico en `stat.m`:

```
function [mean,stdev] = stat(x)
n = length(x);
mean = sum(x)/n;
stdev = sqrt(sum((x-mean).^2/n));
```

Al llamarla en la forma `[m,s] = stat(x)`, si `x` es un vector fila o columna, en `m` quedará el promedio de los elementos de `x`, y en `s` la desviación estándard.

Todas las variables dentro de un *script* que define una función son locales, a menos que se indique lo contrario con `global`. Por ejemplo, si un *script* `x.m` llama a una función `f`, y dentro de `f.m` se usa una variable `a` que queremos sea global, ella se debe declarar en la forma `global a` tanto en `f.m` como en el *script* que la llamó, `x.m`, y en todo otro *script* que pretenda usar esa variable global.

C.6.3. Matrices y variables elementales

Matrices constantes importantes

Las siguientes son matrices que se emplean habitualmente en distintos contextos, y que es útil tener muy presente:

`eye(n)`	Matriz identidad de $n \times n$
`ones(m,n)`	Matriz de $m \times n$, con todos los elementos igual a 1.
`rand(m,n)`	Matriz de $m \times n$ de números al azar, distribuidos uniformemente.
`randn(m,n)`	Igual que `rand`, pero con distribución normal (Gaussiana).
`zeros(m,n)`	Igual que `ones`, pero con todos los elementos 0.

Matrices útiles para construir ejes o mallas para graficar

Las siguientes son matrices se emplean habitualmente en la construcci'on de gráficos:

`v = linspace(min,max,n)`	Vector cuyo primer elemento es `min`, su último elemento es `max`, y tiene `n` elementos equiespaciados.
`v = logspace(min,max,n)`	Análogo a `linspace`, pero los `n` elementos están espaciados logarítmicamente.
`[X,Y] = meshgrid(x,y)`	Construye una malla del plano x-y. Las filas de `X` son copias del vector `x`, y las columnas de `Y` son copias del vector `y`.

Por ejemplo:

```
x = [1 2 3];
y = [4 5];
[X,Y] = meshgrid(x,y);
```

da

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \qquad Y = \begin{pmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \end{pmatrix}.$$

Notemos que al tomar sucesivamente los pares ordenados (`X(1,1),Y(1,1)`), (`X(1,2),Y(1,2)`), (`X(1,3),Y(1,3)`), etc., se obtienen todos los pares ordenados posibles tales que el primer elemento está en `x` y el segundo está en `y`. Esta característica hace particularmente útil el comando `meshgrid` en el contexto de gráficos de funciones de dos variables (ver secciones C.6.7, C.6.7).

Constantes especiales

Octave/Matlab proporciona algunos números especiales, algunos de los cuales ya mencionamos en la sección C.3.1.

`i`, `j`	Unidad imaginaria ($\sqrt{-1}$)
`inf`	Infinito
`NaN`	Not-A-Number
`pi`	El número π ($= 3.1415926535897\ldots$)

Funciones elementales

Desde luego, Octave/Matlab proporciona todas las funciones matemáticas básicas. Por ejemplo:

a) Funciones sobre números reales/complejos

`abs`	Valor absoluto de números reales, o módulo de números imaginarios
`angle`	Ángulo de fase de un número imaginario
`conj`	Complejo conjugado
`real`	Parte real
`imag`	Parte imaginaria
`sign`	Signo
`sqrt`	Raíz cuadrada

b) Exponencial y funciones asociadas

`cos`, `sin`, etc.	Funciones trigonométricas
`cosh`, `sinh`, etc.	Funciones hiperbólicas
`exp`	Exponencial
`log`	Logaritmo

c) Redondeo

`ceil`	Redondear hacia $+\infty$
`fix`	Redondear hacia cero
`floor`	Redondear hacia $-\infty$
`round`	Redondear hacia el entero más cercano

Funciones especiales

Además, Octave/Matlab proporciona diversas funciones matemáticas especiales. Algunos ejemplos:

`bessel`	Función de Bessel
`besselh`	Función de Hankel
`beta`	Función beta
`ellipke`	Función elíptica
`erf`	Función error
`gamma`	Función gamma

Así, por ejemplo, `bessel(alpha,X)` evalúa la función de Bessel de orden `alpha`, $J_\alpha(x)$, para cada elemento de la matriz X.

C.6.4. Polinomios

Octave/Matlab representa los polinomios como vectores fila. El polinomio

$$p = c_n x^n + \cdots + c_1 x + c_0$$

es representado en Octave/Matlab en la forma

```
p = [c_n, ..., c1, c0]
```

Podemos efectuar una serie de operaciones con los polinomios así representados.

`poly(x)`	Polinomio cuyas raíces son los elementos de `x`.
`polyval(p,x)`	Evalúa el polinomio `p` en `x` (en los elementos de `x` si éste es un vector)
`roots(p)`	Raíces del polinomio `p`

C.6.5. Álgebra lineal (matrices cuadradas)

Unos pocos ejemplos, entre los comandos de uso más habitual:

`det`	Determinante
`rank`	Número de filas o columnas linealmente independientes
`trace`	Traza
`inv`	Matriz inversa
`eig`	Autovalores y autovectores
`poly`	Polinomio característico

Notar que `poly` es la misma función de la sección C.6.4 que construye un polinomio de raíces dadas. En el fondo, construir el polinomio característico de una matriz es lo mismo, y por tanto tiene sentido asignarles la misma función. Y no hay confusión, pues una opera sobre vectores y la otra sobre matrices cuadradas.

El uso de todos estos comandos son autoexplicativos, salvo `eig`, que se puede emplear de dos modos:

```
d = eig(a)
[V,D] = eig(a)
```

La primera forma deja en `d` un vector con los autovalores de `a`. La segunda, deja en `D` una matriz diagonal con los autovalores, y en `V` una matiz cuyas columnas son los autovalores, de modo que `A*V = V*D`. Por ejemplo, si `a =[1 2; 3 4]`, entonces

$$d = \begin{pmatrix} 5.37228 \\ -0.37228 \end{pmatrix}$$

y

$$D = \begin{pmatrix} 5.37228\ldots & 0 \\ 0 & -0.37228\ldots \end{pmatrix}, \qquad V = \begin{pmatrix} 0.41597\ldots & -0.82456\ldots \\ 0.90938\ldots & 0.56577\ldots \end{pmatrix}.$$

La primera columna de `V` es el autovector de `a` asociado al primer autovalor, $5.37228\ldots$.

C.6.6. Análisis de datos y transformada de Fourier

En Octave/Matlab están disponibles diversas herramientas para el análisis de series de datos (estadística, correlaciones, convolución, etc.). Algunas de las operaciones básicas son:

a) Máximos y mínimos
   Si `a` es un vector, `max(a)` es el mayor elemento de `a`. Si es una matriz, `max(a)` es un vector fila, que contiene el máximo elemento para cada columna.

```
a = [1 6 7; 2 8 3; 0 4 1]
b = max(a)
```
$b = (2 \quad 8 \quad 7)$

Se sigue que el mayor elemento de la matriz se obtiene con `max(max(a))`.

`min` opera de modo análogo, entregando los mínimos.

b) Estadística básica
Las siguientes funciones, como `min` y `max`, operan sobre vectores del modo usual, y sobre matrices entregando vectores fila, con cada elemento representando a cada columna de la matriz.

`mean`	Valor promedio
`median`	Mediana
`std`	Desviación standard
`prod`	Producto de los elementos
`sum`	Suma de los elementos

c) Orden
`sort(a)` ordena los elementos de `a` en orden ascendente si `a` es un vector. Si es una matriz, ordena cada columna.

```
b = sort([1 3 9; 8 2 1; 4 -3 0]);
```
$$b = \begin{pmatrix} 1 & -3 & 0 \\ 4 & 2 & 1 \\ 8 & 3 & 9 \end{pmatrix}$$

d) Transformada de Fourier

Por último, es posible efectuar transformadas de Fourier directas e inversas, en una o dos dimensiones. Por ejemplo, `fft` y `ifft` dan la transformada de Fourier y la transformada inversa de `x`, usando un algoritmo de *fast Fourier transform* (FFT). Específicamente, si `X=fft(x)` y `x=ifft(X)`, y los vectores son de largo `N`:

$$X(k) = \sum_{j=1}^{N} x(j)\omega_N^{(j-1)(k-1)} \ ,$$
$$x(j) = \frac{1}{N}\sum_{k=1}^{N} X(k)\omega_N^{-(j-1)(k-1)} \ ,$$

donde $\omega_N = e^{-2\pi i/N}$.

C.6.7. Gráficos

Una de las características más importantes de Matlab son sus amplias posibilidades gráficas. Algunas de esas características se encuentran también en Octave. En esta sección revisaremos el caso de gráficos en dos dimensiones, en la siguiente el caso de tres dimensiones, y luego examinaremos algunas posibilidades de manipulación de gráficos.

Gráficos bidimensionales

Para graficar en dos dimensiones se usa el comando `plot`. `plot(x,y)` grafica la ordenada `y` versus la abscisa `x`. `plot(y)` asume abscisa `[1,2,...n]`, donde `n` es la longitud de `y`.

Ejemplo: Si `x=[2 8 9]`, `y=[6 3 2]`, entonces
`plot(x,y)`

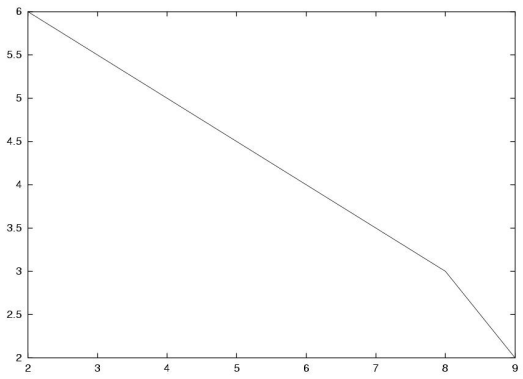


Figura C.1: Gráfico simple.

Por *default*, Octave utiliza `gnuplot` para los gráficos. Por *default*, los puntos se conectan con una línea roja en este caso. El aspecto de la línea o de los puntos puede ser modificado. Por ejemplo, `plot(x,y,'ob')` hace que los puntos sean indicados con círculos (`'o'`) azules (`'b'`, *blue*). Otros modificadores posibles son:

`-`	línea (*default*)	`r`	red
`.`	puntos	`g`	green
`@`	otro estilo de puntos	`b`	blue
`+`	signo más	`m`	magenta
`*`	asteriscos	`c`	cyan
`o`	círculos	`w`	white
`x`	cruces		

Dos o más gráficos se pueden incluir en el mismo output agregando más argumentos a `plot`. Por ejemplo: `plot(x1,y1,'x',x2,y2,'og',x3,y3,'.c')`.

Los mapas de contorno son un tipo especial de gráfico. Dada una función $z = f(x, y)$, nos interesa graficar los puntos (x, y) tales que $f = c$, con c alguna constante. Por ejemplo, consideremos

$$z = xe^{-x^2-y^2} \ , \quad x \in [-2, 2], \ y \in [-2, 3] \ .$$

Para obtener el gráfico de contorno de z, mostrando los niveles $z = -.3$, $z = -.1$, $z = 0$, $z = .1$ y $z = .3$, podemos usar las instrucciones:

```
x = -2:.2:2;
y = -2:.2:3;
[X,Y] = meshgrid(x,y);
```

```
Z = X.*exp(-X.^2-Y.^2);
contour(Z.',[-.3 -.1 0 .1 .3],x,y);   # Octave por default (gnuplot)
contour(x, y, Z.',[-.3 -.1 0 .1 .3]); # Octave con plplot y Matlab
```

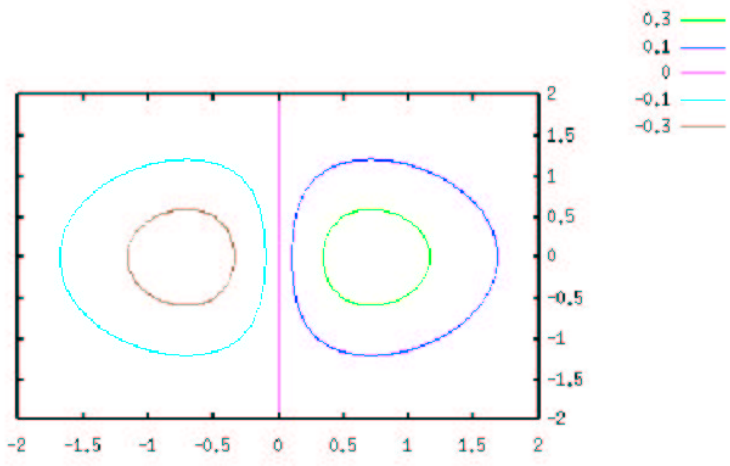


Figura C.2: Curvas de contorno.

Las dos primeras líneas definen los puntos sobre los ejes x e y en los cuales la función será evaluada. En este caso, escojimos una grilla en que puntos contiguos están separados por `.2`. Para un mapa de contorno, necesitamos evaluar la función en todos los pares ordenados (x, y) posibles al escoger x en `x` e y en `y`. Para eso usamos `meshgrid` (introducida sin mayores explicaciones en la sección C.6.3). Luego evaluamos la función [`Z` es una matriz, donde cada elemento es el valor de la función en un par ordenado (x, y)], y finalmente construimos el mapa de contorno para los niveles deseados.

Gráficos tridimensionales

También es posible realizar gráficos tridimensionales. Por ejemplo, la misma doble gaussiana de la sección anterior se puede graficar en tres dimensiones, para mostrarla como una superficie $z(x, y)$. Basta reemplazar la última instrucción, que llama a `contour`, por la siguiente:

```
mesh(X,Y,Z)
```

Observar que, mientras `contour` acepta argumentos dos de los cuales son vectores, y el tercero una matriz, en `mesh` los tres argumentos son matrices de la misma dimensión (usamos `X`, `Y`, en vez de `x`, `y`).

Nota importante: Otro modo de hacer gráficos bi y tridimensionales es con `gplot` y `gsplot` (instrucciones asociadas realmente no a Octave sino a `gnuplot`, y por tanto no equivalentes a instrucciones en Matlab). Se recomienda consultar la documentación de Octave para los detalles.

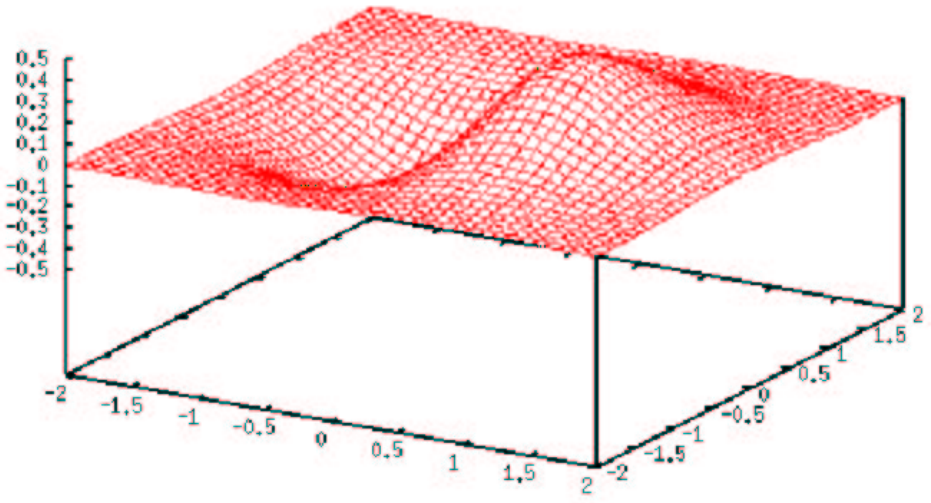


Figura C.3: Curvas de contorno.

Manipulación de gráficos

Los siguientes comandos están disponibles para modificar gráficos construidos con Octave/Matlab:

a) Ejes

`axis([x1 y1 x2 y2])` Cambia el eje x al rango $(x1, x2)$, y el eje y al rango $(y1, y2)$.

b) Títulos

`title(s)`	Título (`s` es un string)
`xlabel(s)`	Título del eje x, y, z.
`ylabel(s)`	
`zlabel(s)`	

c) Grillas

`grid` Incluye o borra una grilla de referencia en un gráfico bidimensional. `grid ''on''` coloca la grilla y `grid ''off''` la saca. `grid` equivale a `grid ''on''`.

Al usar `gnuplot`, el gráfico mostrado en pantalla no es actualizado automáticamente. Para actualizarlo y ver las modificaciones efectuadas, hay que dar la instrucción `replot`.

Los siguientes comandos permiten manipular las ventanas gráficas:

`hold`	Permite "congelar" la figura actual, de modo que sucesivos comandos gráficos se superponen sobre dicha figura (normalmente la figura anterior es reemplazada por la nueva). `hold on` activa este "congelamiento", y `hold off` lo desactiva. `hold` cambia alternativamente entre el estado `on` y `off`.
`closeplot`	Cierra la ventana actual.

Finalmente, si se desea guardar un gráfico en un archivo, se puede proceder del siguiente modo si Octave está generando los gráficos con `gnuplot` y se trabaja en un terminal con XWindows. Si se desea guardar un gráfico de la función $y = x^3$, por ejemplo:

```
x = linspace(1,10,30);
y = x.^3;
plot(x,y);
gset term postscript color
gset output ‘‘xcubo.ps’’
replot
gset term x11
```

Las tres primeras líneas son los comandos de Octave/Matlab convencionales para graficar. Luego se resetea el terminal a un terminal postscript en colores (`gset term postscript` si no deseamos los colores), para que el output sucesivo vaya en formato postscript y no a la pantalla. La siguiente línea indica que la salida es al archivo `xcubo.ps`. Finalmente, se redibuja el gráfico (con lo cual el archivo `xcubo.ps` es realmente generado), y se vuelve al terminal XWindows para continuar trabajando con salida a la pantalla.

Debemos hacer notar que no necesariamente el gráfico exportado a *Postscript* se verá igual al resultado que `gnuplot` muestra en pantalla. Durante la preparación de este manual, nos dimos cuenta de ello al intentar cambiar los estilos de línea de `plot`. Queda entonces advertido el lector.

C.6.8. Strings

Para manipular una cadena de texto, disponemos de los siguientes comandos:

`lower`	Convierte a minúsculas
`upper`	Convierte a mayúsculas

Así, `lower('Texto')` da `'texto'`, y `upper('Texto')` da `'TEXTO'`.

Para comparar dos matrices entre sí, usamos `strcmp`:

`strcmp(a,b)`	1 si `a` y `b` son idénticas, 0 en caso contrario

Podemos convertir números enteros o reales en strings, y strings en números, con los comandos:

`int2str`	Convierte entero en string
`num2str`	Convierte número en string
`str2num`	Convierte string en número

Por ejemplo, podemos usar esto para construir un título para un gráfico:

```
s = ['Intensidad transmitida vs. frecuencia, n = ', num2str(1.5)];
title(s);
```

Esto pondrá un título en el gráfico con el texto:
`Intensidad transmitida vs. frecuencia, n = 1.5`.

C.6.9. Manejo de archivos

Ocasionalmente nos interesará grabar el resultado de nuestros cálculos en archivos, o utilizar datos de archivos para nuevos cálculos. El primer paso es abrir un archivo:

```
archivo = fopen('archivo.dat','w');
```

Esto abre el archivo `archivo.dat` para escritura (`'w'`), y le asigna a este archivo un número que queda alojado en la variable `archivo` para futura referencia.

Los modos de apertura posibles son:

`r`	Abre para lectura
`w`	Abre para escritura, descartando contenidos anteriores si los hay
`a`	Abre o crea archivo para escritura, agregando datos al final del archivo si ya existe
`r+`	Abre para lectura y escritura
`w+`	Crea archivo para lectura y escritura
`a+`	Abre o crea archivo para lectura y escritura, agregando datos al final del archivo si ya existe

En un archivo se puede escribir en modo binario:

`fread`	Lee datos binarios
`fwrite`	Escribe datos binarios

o en modo texto

`fgetl`	Lee una línea del archivo, descarta cambio de línea
`fgets`	Lee una línea del archivo, preserva cambio de línea
`fprintf`	Escribe datos siguiendo un formato
`fscanf`	Lee datos siguiendo un formato

Referimos al lector a la ayuda que proporciona Octave/Matlab para interiorizarse del uso de estos comandos. Sólo expondremos el uso de `fprintf`, pues el formato es algo que habitualmente se necesita tanto para escribir en archivos como en pantalla, y `fprintf` se puede usar en ambos casos.

La instrucción

```
fprintf(archivo,'formato',A,B,...)
```

imprime en el archivo asociado con el identificador `archivo` (asociado al mismo al usar `fopen`, ver más arriba), las variables `A`, `B`, etc., usando el formato `'formato'`. `archivo=1` corresponde a la pantalla; si `archivo` se omite, el *default* es `1`, es decir, `fprintf` imprime en pantalla si `archivo=1` o si se omite el primer argumento.

`'formato'` es una string, que puede contener caracters normales, caracteres de escape o especificadores de conversión. Los caracteres de escape son:

`\n`	New line
`\t`	Horizontal tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\\`	Backslash
`\'`	Single quote

Por ejemplo, la línea

```
fprintf('Una tabulacion\t y un \'\'original\'\' cambio de linea\n aqui\n')
```

da como resultado

```
Una tabulacion       y un ''original'' cambio de linea
aqui
```

Es importante notar que por *default*, el cambio de línea al final de un `fprintf` no existe, de modo que, si queremos evitar salidas a pantalla o a archivo poco estéticas, siempre hay que terminar con un `\n`.

Los especificadores de conversión permiten dar formato adecuado a las variables numéricas `A`, `B`, etc. que se desean imprimir. Constan del caracter `%`, seguido de indicadores de ancho (opcionales), y caracteres de conversión. Por ejemplo, si deseamos imprimir el número π con 5 decimales, la instrucción es:

```
fprintf('Numero pi = %.5f\n',pi)
```

El resultado:

```
Numero pi = 3.14159
```

Los caracteres de conversión pueden ser

`%e`	Notación exponencial (Ej.: `2.4e-5`)
`%f`	Notación con punto decimal fijo (Ej.: `0.000024`)
`%g`	`%e` o `%f`, dependiendo de cuál sea más corto (los ceros no significativos no se imprimen)

Entre `%` y `e`, `f`, o `g` según corresponda, se pueden agregar uno o más de los siguientes caracteres, en este orden:

- Un signo menos (`-`), para especificar alineamiento a la izquierda (a la derecha es el *default*).

- Un número entero para especificar un ancho mínimo del campo.
- Un punto para separar el número anterior del siguiente número.
- Un número indicando la precisión (número de dígitos a la derecha del punto decimal).

En el siguiente ejemplo veremos distintos casos posibles. El output fue generado con las siguientes instrucciones, contenidas en un *script*:

```
a = .04395;
fprintf('123456789012345\n');
fprintf('a = %.3f.\n',a);
fprintf('a = %10.2f.\n',a);
fprintf('a = %-10.2f.\n',a);
fprintf('a = %4f.\n',a);
fprintf('a = %5.3e.\n',a);
fprintf('a = %f.\n',a);
fprintf('a = %e.\n',a);
fprintf('a = %g.\n',a);
```

El resultado:

```
12345678901234567890
a = 0.044.
a =       0.04.
a = 0.04      .
a = 0.043950.
a = 4.395e-02.
a = 0.043950.
a = 4.395000e-02.
a = 0.04395.
```

En la primera línea, se imprimen tres decimales. En la segunda, dos, pero el ancho mínimo es 10 caracteres, de modo que se alínea a la derecha el output y se completa con blancos. En la tercera línea es lo mismo, pero alineado a la izquierda. En la cuarta línea se ha especificado un ancho mínimo de 4 caracteres; como el tamaño del número es mayor, esto no tiene efecto y se imprime el número completo. En la quinta línea se usa notación exponencial, con tres decimal (nuevamente, el ancho mínimo especificado, 5, es menor que el ancho del output, luego no tiene efecto). Las últimas tres líneas comparan el output de `%f`, `%e` y `%g`, sin otras especificaciones.

Si se desean imprimir más de un número, basta agregar las conversiones adecuadas y los argumentos en `fprintf`. Así, la línea

```
fprintf('Dos numeros arbitrarios: %g y %g.\n',pi,exp(4));
```

da por resultado

```
Dos numeros arbitrarios: 3.14159 y 54.5982.
```

Si los argumentos numéricos de `fprintf` son matrices, el formato es aplicado a cada columna hasta terminar la matriz. Por ejemplo, el *script*

```
x = 1:5;
y1 = exp(x);
y2 = log(x);
a = [x; y1; y2];
fprintf = ('%g  %8g %8.3f\n',a);
```

da el output

```
1   2.71828    0.000
2   7.38906    0.693
3   20.0855    1.099
4   54.5982    1.386
5   148.413    1.609
```