

Guía de aprendizaje para estudiantes

Herramientas Matemáticas de Software Libre

Guía de aprendizaje para estudiantes

Herramientas Matemáticas de Software Libre



Miguel Ángel Vilela García
Carlos Alberto Morales Díaz

Copyright © 2001 – 2003 GRUPO DE USUARIOS DE LINUX DE CANARIAS

Se da permiso para copiar, distribuir o modificar este documento en los términos que establece la GNU Free Documentation License, Versión 1.1 o cualquier otra versión posterior publicada por la Free Software Foundation; las secciones que no pueden modificarse son “Presentación” y “Agradecimientos”, y no existen textos cubierta ni contracubierta. Se incluye una copia de la licencia en la sección “GNU Free Documentation License”. Este libro es © de sus autores.

La forma original de este documento es código fuente en \LaTeX . La compilación de este código fuente en \LaTeX tiene el efecto de generar una representación del documento independiente del dispositivo, que puede convertirse a otros formatos o imprimirse.

La GNU Free Documentation License está disponible en www.gnu.org o soliciándola por escrito a the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Este documento lo han maquetado sus autores utilizando \LaTeX y The GIMP, que son programas abiertos y libres.

A los novatos y novatas en el mundo de GNU/Linux,
y a sus familiares, amistades y parejas
(ellas y ellos saben por qué)

Presentación

Historia

Desde el curso académico 2001–2002 la FACULTAD DE MATEMÁTICAS de la UNIVERSIDAD DE LA LAGUNA dispone de sistemas Debian GNU/Linux en las aulas de ordenadores destinadas a impartir clases prácticas. Estas aulas están destinadas a que los alumnos realicen sus prácticas académicas, tales como programación en Pascal, Fortran, C o Java, análisis estadísticos o cálculos simbólicos. Sin embargo, los alumnos no disponían en su mayoría de los conocimientos mínimos necesarios para utilizar un equipo bajo sistema operativo GNU/Linux.

Ante tal panorama, y con el objetivo urgente de proporcionar a los alumnos la destreza mínima que necesitaban para utilizar los ordenadores de la facultad con Debian GNU/Linux, MIGUEL ÁNGEL VILELA (miembro del GRUPO DE USUARIOS DE LINUX DE CANARIAS y administrador de los sistemas Debian GNU/Linux de las aulas) propuso paralelamente la realización de un **Curso de Introducción a Linux para Alumnos** a SERGIO ALONSO (Vicedecano de Estadística) y a otros miembros del el GRUPO DE USUARIOS DE LINUX DE CANARIAS. En ambas partes la idea tuvo buena aceptación y en sólo 10 días el GRUPO DE USUARIOS DE LINUX DE CANARIAS y la Facultad de Matemáticas, con el patrocinio de IDE System Canarias S.L., organizaron la primera edición del Curso de Introducción a Linux para Alumnos.

Tras el éxito del Curso de Introducción a Linux para Alumnos (30 inscritos, 27 asistentes y más de 50 alumnos en lista de espera) se organizó la primera edición de la **Party de Instalación de Linux para Alumnos**, a la que no asistieron tantos alumnos debido (suponemos) a que la asistencia implicaba traer el ordenador personal. Sin embargo, el balance final de ambas actividades se consideró muy positivo, por lo que el GRUPO DE USUARIOS DE LINUX DE CANARIAS se embarcó en el siguiente reto: ampliar el Curso de Introducción a Linux para Alumnos para aumentar el temario (incluyendo la Party de Instalación de Linux para Alumnos).

El Curso de Introducción a Linux para Alumnos se ha realizado ya dos años consecutivos en la FACULTAD DE MATEMÁTICAS y para los próximos años pretendemos seguir realizándolo, ampliando

su radio de acción a otras facultades de la UNIVERSIDAD DE LA LAGUNA. Este libro es una puerta abierta para que el Curso de Introducción a Linux para Alumnos sea realizado en otras universidades y centros docentes.

Parte de este proyecto ha sido reformar los apuntes del Curso de Introducción a Linux para Alumnos, inicialmente escritos en SGML (con DTD de LinuxDoc), para traducirlos a L^AT_EX y así generar este libro, que constituye los apuntes del curso. Este libro es una publicación de “contenido abierto”, lo que significa que está abierto a las aportaciones que pueda hacer cualquier persona y seguirá evolucionando con el tiempo.

Objetivos

El objetivo del Curso de Introducción a Linux para Alumnos y de este libro es el aprendizaje del alumno (o lector), a ser posible de una forma amena. En adelante trataremos de tú tanto al alumno como al lector. Esperamos que no te sientas ofendido.

Este libro está dirigido principalmente a usuarios, con un enfoque fuertemente académico. Si hace poco que llegaste al maravilloso mundo de GNU/Linux y necesitas una guía práctica para empezar a trabajar, este libro puede resultarte de utilidad. Si además eres estudiante y necesitas hacer tus prácticas en ordenadores con GNU/Linux pero no te orientas aún, este libro puede ser tu salvación. Si ya tienes experiencia con GNU/Linux (no eres un novato) no encontrarás muchas cosas nuevas aquí, aunque tampoco te vendrá mal leerlo.

La finalidad del Curso de Introducción a Linux para Alumnos es que los alumnos adquieran los conocimientos necesarios para utilizar los ordenadores con GNU/Linux en sus prácticas académicas, y tal vez se animen a instalar GNU/Linux en sus propios ordenadores personales. Como se ha mencionado anteriormente, el Curso de Introducción a Linux para Alumnos consta de unas clases teórico-prácticas, en las que los alumnos toman apuntes mientras prueban lo que van aprendiendo con un ordenador delante, y unas jornadas práctico-teóricas en la que cada alumno es invitado a traer su propio ordenador personal para instalar GNU/Linux por sí mismo, contando siempre con la ayuda de los profesores.

Dada la posible variedad de los alumnos, la cantidad de materia que se desea impartir y las limitaciones de disponibilidad de puestos en las aulas de informática en cuanto a número y tiempo, el Curso de Introducción a Linux para Alumnos se ha dividido en varios módulos, todos ellos opcionales.

Las tareas rutinarias de administración y mantenimiento de un equipo bajo sistema operativo GNU/Linux no se tratan en este libro, entre otras razones porque suelen depender mucho de la máquina en la que se practiquen y de las necesidades de su(s) usuario(s). Esta parte del aprendizaje de GNU/Linux es demasiado práctica para un libro como éste. Para esto es mejor comprar algún libro de iniciación a Linux que incluya una distribución de GNU/Linux para instalar. Este tipo de libros se pueden encontrar en casi cualquier librería técnica en variedad para todos los gustos, desde pequeñas guías de 10 minutos con CD de instalación rápida hasta libros gruesos de sabiduría condensada, elige el que más te guste y reserva grandes ratos en tu agenda.

Uno de los motivos por los que publicamos este libro bajo la licencia GNU Free Documentation Licence es porque queremos que tenga toda la divulgación que sus lectores quieran. Este libro no es copyright de ninguna editorial, sólo de los autores, y damos nuestro permiso para que este material sea fotocopiado y redistribuido libremente. Nuestro objetivo es acercar GNU/Linux y sus herramientas a cuantos más alumnos y usuarios sea posible.

Contenidos

Este libro está dividido en módulos atendiendo a la división de contenidos que hacemos a la hora de impartir las clases, los cuales a su vez están divididos en temas, cuyos contenidos resumimos a continuación.

Capítulo 1: GNU Octave

Para casi cualquier científico se hace muy necesaria la computación rápida de cálculos matemáticos aplicados mediante un lenguaje que permita programar los cálculos. Octave te dará la solución para los cálculos que necesites para física, ingeniería, matemáticas y otras disciplinas.

Capítulo 2: GNUplot

La elaboración de gráficas elaboradas es otra tarea fundamental en el campo de las ciencias, y GNUplot tiene amplias posibilidades para esto.

Capítulo 3: GNU R

Otro lenguaje de cálculo y representación de datos, pero esta vez con capacidades especiales para cálculos estadísticos.

Capítulo 4: Yacas

Este lenguaje es diferente, para cálculo simbólico. Esto es especialmente útil en Matemáticas, donde a veces es necesario manipular las expresiones matemáticas sin evaluarlas.

Convenciones tipográficas

En la escritura de este libro hemos utilizado algunas convenciones para facilitar su lectura. Usaremos los diferentes tipos de letra para fines determinados:

- **sin adornos:** nombres de programas, paquetes, protocolos, lenguajes, etc. También para menús y botones.
- **máquina de escribir:** comandos, código fuente, pulsaciones y/o combinaciones de teclas; todo aquello que haya que teclear o leer de la pantalla.
- *itálica:* términos nuevos o que haya que resaltar.
- **negrita:** elementos de una descripción o aquello que necesite especial atención.
- **VERSALITA:** para nombres de personas, organizaciones y entidades en general.

Cuando hagamos referencia a combinaciones de teclas utilizaremos la siguiente notación:

- “C-” indica la tecla **Control**.
- “S-” indica la tecla **Shift**.
- “A-” indica la tecla **Alt** (a veces llamada **Meta**).

- “F1” ... “F1” son las teclas de función.
- “ESC” indica la tecla de Escape.
- “TAB” indica la tecla del tabulador.
- “Enter” indica la tecla del retorno de carro.

Además, cuando tengamos que mostrar comandos y/o la salida que éstos produzcan lo haremos de la siguiente forma:

```
$ comando
salida
del
comando
```

El símbolo del dólar \$ es lo que comúnmente se denomina el *prompt* del sistema, lo que el intérprete muestra a la espera de nuestras órdenes. En las distribuciones de GNU/Linux actuales no es normal que este prompt sea únicamente el símbolo del dólar, sino que suele contener también el nombre de la máquina, el nombre del usuario, o el directorio en el que nos encontramos. Estos detalles serán explicados en el tema “El intérprete de comandos”.

En los temas referentes a lenguajes de programación, documentación o cálculo incluiremos algunos ejemplos de código fuente de programas, scripts o documentos. Estos ejemplos están en el directorio `ejemplos` del paquete que contiene el código fuente de este libro, cuya localización en internet te facilitamos en la siguiente sección. Estos ficheros se utilizan en las clase teórico/prácticas para aprender el manejo de compiladores y herramientas de cálculo o documentación.

Fichero `ejemplo.txt`

Ejemplo 0.1: Éste es un ejemplo de cómo te mostraremos los ejemplos de código fuente. Éstos pueden ser programas, scripts o documentos escritos en diferentes lenguajes como C/C++, Fortran, Pascal, HTML, PHP, \LaTeX , Octave, GNUplot, R, Yacas, etc. Los ejemplos que aparezcan de este modo estarán disponibles por separado en sendos ficheros cuyo nombre se indica en la cabecera del ejemplo. Así te ahorras tener que teclear los ejemplos a la hora de probar los lenguajes. Cuando necesites buscar uno de estos ejemplos puedes encontrar su página en el índice de ejemplos de la página XIX.

Sobre este documento

Este libro ha sido escrito gracias a la iniciativa de algunos miembros y amigos del GRUPO DE USUARIOS DE LINUX DE CANARIAS, para el Curso de Introducción a Linux para Alumnos organizado conjuntamente por el GRUPO DE USUARIOS DE LINUX DE CANARIAS y la FACULTAD DE MATEMÁTICAS de la UNIVERSIDAD DE LA LAGUNA, en Tenerife (España). Mira en la página XIII para saber quiénes participaron en la elaboración de este libro.

Como indica la nota de copyright, estos apuntes son libremente distribuibles bajo los términos de la **Licencia de Documentación Libre de GNU**, lo que a grosso modo significa que se permite su copia, redistribución y modificación siempre que se nombre a los autores originales, no se reclame la autoría de trabajo ajeno ni se pierda la libertad de los contenidos.

En el apéndice de la página 75 encontrarás una traducción **no oficial** de la **Licencia de Documentación Libre GNU**. Para leer la versión original de la **GNU Free Documentation License** acude a la página oficial del proyecto GNU, <http://www.gnu.org/licenses/fdl.html>

La última versión de estos apuntes se mantendrá en <http://cila.gulic.org>, tanto su código fuente como las versiones en PostScript y PDF, además de un paquete con los ejemplos por separado.

Este libro es un proyecto **abierto** y en desarrollo, por eso no está completo y posiblemente tarde en estarlo (siempre hay algo que añadir, mejorar, corregir, actualizar, etc.). Si quieres colaborar en él ponte en contacto con nosotros enviando un correo electrónico a la dirección cila@listas.gulic.org. Nos gustaría recibir tus opiniones, correcciones y sugerencias críticas constructivas. Las críticas destructivas serán redireccionadas, como siempre, a `/dev/null`

Agradecimientos

Este libro lo hemos escrito miembros y amigos del GRUPO DE USUARIOS DE LINUX DE CANARIAS; informáticos, estudiantes y/o aficionados con entrega, entre todos unimos nuestros esfuerzos para escribir sobre aquello de lo que mejor entendemos.

La FACULTAD DE MATEMÁTICAS de la UNIVERSIDAD DE LA LAGUNA ha colaborado apasionadamente (y lo sigue haciendo) aportando sus aulas de ordenadores para realizar las sucesivas ediciones del Curso de Introducción a Linux para Alumnos . Desde el GRUPO DE USUARIOS DE LINUX DE CANARIAS agradecemos además la amabilidad con la que han colaborado en todo momento.

Pero este libro no habría llegado a ser lo que es de no ser por las siguientes personas. A todos ellos; muchas gracias.

- SERGIO ALONSO, Vicedecano de Estadística en la FACULTAD DE MATEMÁTICAS de la UNIVERSIDAD DE LA LAGUNA. Acogió amablemente la propuesta del Curso de Introducción a Linux para Alumnos y puso las aulas de ordenadores a disposición del curso. También ha coordinado la publicación de este libro a través del Servicio de Publicaciones de la UNIVERSIDAD DE LA LAGUNA.
- MANOLO GARCÍA ROMÁN, Secretario de la FACULTAD DE MATEMÁTICAS de la UNIVERSIDAD DE LA LAGUNA. Brindó una inestimable ayuda en las cuestiones \TeX nicas de la elaboración de estos apuntes. También se entregó de modo ejemplar en la organización de las distintas ediciones del curso en el ámbito de la UNIVERSIDAD DE LA LAGUNA. Ha coordinado la publicación de este libro a través del SERVICIO DE PUBLICACIONES UNIVERSIDAD DE LA LAGUNA.
- MIGUEL ÁNGEL VILELA GARCÍA escribió los temas “Yacas” y “R”.
- CARLOS ALBERTO MORALES DÍAZ escribió los temas de “Octave” y “GNUplot”.

También agradecemos a todas las personas que nos han animado a seguir con este proyecto: alumnas y alumnos de la FACULTAD DE MATEMÁTICAS y otras facultades de la UNIVERSIDAD DE LA LAGUNA, compañeros del GRUPO DE USUARIOS DE LINUX DE CANARIAS, Maribel C. Salgado, Pedro Reina, gente de HISPALINUX y alguien más que se nos queda en el tintero (o en el teclado).

Índice general

1. GNU Octave	1
1.1. Entorno	1
1.2. Tipos de datos	4
1.3. Variables y expresiones	6
1.4. Control de flujo	8
1.5. Funciones	10
1.6. Representación gráfica	13
1.7. Matrices	17
1.8. Ecuaciones diferenciales	20
1.9. Polinomios	21
1.10. Teoría de control	22
1.11. Procesamiento de señales	22
1.12. Tratamiento de imágenes	25
2. GNUplot	31
2.1. Representación de expresiones analíticas	32
2.2. Representación de archivos de datos	33

3. GNU R	35
3.1. Introducción	35
3.2. El entorno R	35
3.3. El lenguaje R	38
3.4. Vectores	39
3.5. Arrays y matrices	42
3.6. Factores: clasificación de datos	44
3.7. Listas	45
3.8. Hojas de datos	46
3.9. Funciones	47
3.10. Distribuciones de probabilidad	50
3.11. Estadística descriptiva	51
3.12. Inferencia estadística	52
3.13. Gráficas	55
4. Yacas	59
4.1. ¿Qué es Yacas?	59
4.2. Empezando con Yacas	59
4.3. Variables y funciones	63
4.4. Listas	65
4.5. Álgebra Lineal	66
4.6. Control de flujo	67
4.7. Gráficas	69
4.8. Programando con Yacas	70
4.9. Un ejemplo real	71
A. Licencia de Documentación Libre GNU	75
A.1. Aplicabilidad y definiciones	76
A.2. Copia literal	77
A.3. Copiado en cantidades	77
A.4. Modificaciones	78
A.5. Combinando documentos	79
A.6. Colecciones de documentos	80
A.7. Agregación con trabajos independientes	80
A.8. Traducción	80
A.9. Terminación	81
A.10. Futuras revisiones de esta licencia	81
A.11. Addendum	81

Índice de figuras

1.1. Múltiples líneas por gráfica.	14
1.2. Diagramas en coordenadas polares	15
1.3. Representación de histogramas	16
1.4. Gráficas en tres dimensiones	18
1.5. Resolución numérica de EDO	22
1.6. Gradiente visualizado a varios niveles de gris	26
1.7. Imagen en color cambiando las componentes RGB	26
1.8. Imagen tras aplicarle un filtro pasa-baja y pasa-alta	28
1.9. Imagen tras aplicar detector de bordes.	29
3.1. Las distintas formas de la función <code>plot()</code>	56
3.2. Gráficos Q-Q para dos variables	57
3.3. Gráficas para representar matrices	58
4.1. Proteus, interfaz gráfica para Yacas	61
4.2. Representación gráfica con GNUplot desde Yacas	69

Índice de ejemplos

0.1. [ejemplo.txt] Ejemplo de ejemplo	x
1.1. [vmax.m] Función en octave para devolver el máximo de un vector	10
1.2. [oregonator.cc] Función en c++ que compilaremos a formato nativo de octave	12
1.3. [sinus.m] Presentación gráfica cuatro sinusoidales.	14
1.4. [polares.m] Presentación gráfica en polares.	15
1.5. [histo.m] Presentación histograma.	16
1.6. [meshplot.m] Gráfica en tres dimensiones.	17
1.7. [ode.m] Resuelve una EDO ordinaria	20
1.8. [fftview.m] Muestra la FFT de dos funciones	23
3.1. [iodemo.R] Uso de sink()	39
3.2. [curtosis.R] Fichero de funciones	48
4.1. [lagrange.js] Script en Yacas que interpola una función	70
4.2. [sturm.js] Script en Yacas que implementa el algoritmo de Sturm	71

Capítulo 1

GNU Octave

Octave se puede definir como un lenguaje de alto nivel inspirado en un software comercial llamado MATLAB® (MATrix LABoratory). MATLAB® estuvo pensado inicialmente para álgebra numérica lineal (matrices, vectores y sus operaciones), y con el tiempo se le ha sacado partido a esta forma de trabajo. De la misma forma, Octave empezó siendo un software para que los alumnos de Ingeniería Química de las UNIVERSIDADES DE WISCONSIN-MADISON y TEXAS calcularan reacciones químicas.

A partir de ese momento, las contribuciones de los usuarios han hecho evolucionar este software y han añadido librerías y funcionalidades. Ahora, las aplicaciones de Octave ya no se limitan a simple trabajo con matrices y vectores, como una mera calculadora, sino que ahora aparte de aplicaciones puramente matemáticas o numéricas, es válido para otros campos de ciencias e ingenierías. Entre ellos, el procesamiento de señales (sonido), de imágenes (filtrados, análisis, etc), estadística, geometría, redes neuronales, sistemas de control realimentados y hasta dibujo vectorial. Intentaremos poner ejemplos de cada una de estas aplicaciones en la medida de lo posible para mostrar la versatilidad de Octave.

Estas librerías se pueden programar de forma interpretada, usando el propio lenguaje de octave, o de forma binaria, usando cualquiera de los lenguajes que soporte gcc como C/C++, pascal o fortran (recordemos que todo el código objeto era intercambiable). Además, también se puede hacer a la inversa, es decir, traducir programas de octave a c++ usando una librería llamada liboctave. Con esto se elimina la etapa de interpretación al ejecutarlo con lo que se gana velocidad cuando ésta sea determinante. Parece que hay bastantes cosas por ver, así que vamos a empezar.

1.1. Entorno

Octave tiene una filosofía de uso semejante a la de muchas otras aplicaciones de este libro: una interfaz en forma de shell, con una línea de comandos potente con muchos atajos y facilidades, para problemas sencillos, y la posibilidad de poder agrupar muchos comandos en ficheros de scripts, organizados en funciones, para enfrentarse a problemas complejos o para realizar automatizaciones.

Para comenzar a ver el manejo básico vamos a ejecutar Octave de manera interactiva. Con este método de trabajo, si cometemos un error al entrar una línea, podremos corregirlo sobre la marcha.

Para ejecutarlo, abre un terminal y en la línea de comandos teclea `octave`. Tras un mensaje de bienvenida, Octave te muestra un prompt que indica que está preparado y a la espera de comandos. En algunas distribuciones, Octave puede tener su icono en uno de los menús del sistema, en la zona de aplicaciones matemáticas. Teclear `octave` en la consola es más rápido y funciona el 100 % de las veces. Esto es lo que se nos muestra:

```
$ octave
GNU Octave, version 2.1.34 (i386-pc-linux-gnu).
Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001 John W. Eaton.
This is free software with ABSOLUTELY NO WARRANTY.
For details, type 'warranty'.
```

```
octave:1>
```

Cuando quieras salir de Octave teclea `exit`, `quit` o `C-D` y volverás al shell de partida.

La ayuda completa de `octave` la puedes obtener desde el prompt tecleando `help -i`. También puedes visualizar la misma ayuda desde el shell tecleando `info octave`. Luego, la documentación para cada función y variables se obtienen tecleando `help nombredelafuncion`. Por ejemplo:

```
octave:9> help coth
coth is the user-defined function from the file
/usr/share/octave/2.1.34/m/elfun/coth.m

- Mapping Function:  coth (X)
  Compute the hyperbolic cotangent of each element of X.
```

La mayoría de los comandos de Octave disponen de esta ayuda. Vemos que se nos dice una descripción de los parámetros y lo que realiza la función, lo cual es suficiente para que podamos utilizarla.

Cuando se invoca sin argumentos se obtiene un listado de todas las operaciones, funciones y variables incorporadas definidas en el sistema. Para conseguir esta información, Octave rastrea por los directorios donde están instaladas las funciones; de ahí su peculiar forma de organizar esta salida, que nos muestra las funciones clasificadas por temas, lo que puede ayudarnos a mirar y probar funciones. Además, aquí podemos encontrarnos funciones que no están pasadas a la documentación.

Octave usa la librería *GNU readline* para la edición en línea de comandos, al igual que *bash* y otros programas *GNU*. Contiene un historial que se puede leer con las flechas arriba y abajo, muchas combinaciones de teclas para hacer muchas cosas. Para más información y explicación sobre estas características teclea desde un shell `info rluserman`. No entraremos más en este tema.

Cada vez que te equivoques en la sintaxis, `octave` te indicará la posición donde cree que está el fallo con un angulillo `^`. A veces no se puede fiar uno completamente, y sólo te ayuda a saber más o menos donde se localiza. Veámoslo aquí:

```
octave:13> function y = f (x) y = x^2; endfunction
parse error:
```

```
>>> function y = f (x) y = x^2; endfunction
```

Otro tipo de errores pueden ocurrir dentro de funciones. En este caso, son errores en tiempo de ejecución, porque ocurren por un fallo en la ejecución del programa. En este caso, lo que aparece es la línea y posición dentro de la función y en la función que lo llamó y en las siguientes. Por ejemplo, en este hipotético caso:

```
octave:13> f ()
error: 'x' undefined near line 1 column 24
error: evaluating expression near line 1, column 24
error: evaluating assignment expression near line 1, column 22
error: called from 'f'
```

En este caso, la función `f` se compone de unas funciones que se llaman a otras. El error está en la línea 1 con una `x` mal definida. La función que contiene este error, según octave, formaba parte de una expresión en la línea 1, que a su vez formaba parte de una asignación también en la línea 1. Obsérvese que la función es la que definimos en el anterior ejemplo y el error es que hace falta pasarle un parámetro a la función.

Los comentarios dentro del código de octave se preceden con el carácter `#` o `%` y abarcan desde ese carácter hasta el final de la línea. Si ponemos en octave un código como éste:

```
function xdot = f (x, t)

# usage: f (x, t)
#
# This function defines the right hand
# side functions for a set of nonlinear
# differential equations.

    r = 0.25;
    ...
endfunction
```

Octave interpreta los comentarios después de la función como el texto de ayuda. De esta forma, cuando hagamos `help xdot` nos mostrará como ayuda el texto que hemos definido en el ejemplo.

Cuando una línea se hace demasiado larga se puede añadir al final de la línea una barra invertida `\` o unos puntos suspensivos `...` y continuar en la siguiente línea.

```
x = long_variable_name ...
    + longer_variable_name \
    - 42
```

Otra cosa interesante es que por defecto se muestra el resultado de la operación al realizarla salvo cuando se añade un `;` al final de la operación. También se pueden hacer varias operaciones en una misma línea separándolas con `;`.

```
octave:1> a=sqrt(3)
a = 1.7321
octave:2> b=sqrt(5);
octave:3> b
b = 2.2361
octave:4> sqrt(7)
ans = 2.6458
```

Como última nota, hay que añadir que cuando no capturamos el valor de retorno aparece la palabra `ans`, que representa el último resultado, y que se puede usar como variable en la siguiente línea. Continuando el ejemplo anterior:

```
octave:5> ans*ans
ans = 7.0000
octave:6> ans*ans
ans = 49.000
```

1.2. Tipos de datos

En Octave hay tres tipos de datos: numéricos (escalares, vectores y matrices), cadenas (strings) y estructuras. Como es de suponer, los numéricos son los más usados, mientras que cadenas sólo se usan para presentar mensajes. Las estructuras son algo que nos pueden dar una buena base para organizar tareas más complicadas. A continuación se muestra la forma de definir cada uno de ellos.

```
octave:13> a=[1 2 3]
a =
  1  2  3

octave:14> b=[1,2,3;4,5,6]
b =
  1  2  3
  4  5  6

octave:15> c="hola mundo"
c = hola mundo

octave:16> d.vector=[1;2];
octave:17> d.matriz=[1 2; 3 4];
octave:18> d.texto="titulo";
octave:19> d
d =
{
  texto = titulo
  vector =
    1
    2
```

```

matriz =
  1  2
  3  4

}
octave:20> d.vector
d.vector =
  1
  2

```

Se observará que las definiciones de matrices o vectores, las filas van separadas por comas (,) o espacios () mientras que las columnas se separan por punto y coma (;). También observamos que una estructura no es más que un *paquete* donde se pueden almacenar varias variables relacionadas juntas. Se usan principalmente para reducir el número de argumentos de las funciones agrupando todos los argumentos relacionados en estructuras y pasando éstas.

En relación con matrices y vectores, las siguientes funciones nos informan sobre las dimensiones de estos elementos. Por ejemplo:

```

octave:25> length(a)
ans = 3
octave:26> columns(b)
ans = 3
octave:27> rows(b)
ans = 2
octave:28> size(b)
ans =

  2  3

```

Donde `length` da la dimensión de un vector fila o columna, `columns` y `rows` dan las columnas o filas de una matriz y `size` devuelve un vector fila con las dimensiones.

Ya hemos visto la forma más simple de definir una matriz. También se puede definir una matriz en base a otras matrices existentes. Por ejemplo, continuando de lo anterior, podemos construir una matriz concatenando un vector fila junto a otro o encima de otro:

```

octave:34> g=[a a]
g =

  1  2  3  1  2  3

octave:35> g=[a;a]
g =

  1  2  3
  1  2  3

```

Como última nota sobre vectores y matrices, sólo queda comentar que hay una forma taquigráfica de definir un vector secuencial. `m:n` devuelve un vector de números consecutivos desde `m` hasta `n` de uno en uno, ambos inclusive. Por ejemplo:

```
octave:36> -2:3
ans =
-2 -1 0 1 2 3
```

De igual manera, `m:s:n` devuelve un vector de números consecutivos que van desde `m` hasta `n` de `s` en `s`. Por ejemplo:

```
octave:37> 7:-2:1
ans =
7 5 3 1
```

De las operaciones con cadenas diremos que se tratan como vectores y que hay muchas funciones para su manejo pero que no nombraremos aquí. Las operaciones con matrices y vectores son las usuales. De resto, comentar que también existen tipos de datos booleanos.

1.3. Variables y expresiones

Octave te permite llamar a las variables con secuencias de cualquier longitud de letras, números y subrayados, pero sin empezar en dígito. Los nombres son sensibles a mayúsculas/minúsculas. Ya hemos tenido ejemplos anteriormente.

Una vez que ha sido definida una variable, puede ser útil conocer el comando `who` que nos da una lista de variables definidas, y `whos` que nos muestra más información.

```
octave:15> who

*** local user variables:

a b c d g

octave:16> whos

*** local user variables:

prot  type                rows  cols  name
====  ====                =====  =====  =====
rwd  matrix                1     3    a
rwd  matrix                2     3    b
rwd  string                1    10    c
rwd  struct                -     -    d
rwd  matrix                2     3    g
```

En caso de que queramos eliminar una variable de memoria usaremos la orden `clear nombrevariable`. En caso de que queramos borrar todas las variables simplemente teclearemos `clear` sin argumentos.

La primera expresión que podemos aprender es la *extracción* de valores desde una matriz. En el caso más sencillo de extraer un elemento, por ejemplo, de la fila 1 y columna 2, escribimos `a(1,2)`. El operador `:` (dos puntos) nos vale de comodín, y nos valdrá para extraer, por ejemplo, toda la fila 1 escribiendo `a(1,:)` o toda la columna 2 escribiendo `a(:,2)`. También podemos incluir rangos en los argumentos, por ejemplo, para sacar las columnas 2 y 3 sería `a(:, [2 3])`.

Cabe señalar que las matrices en octave se crean dinámicamente, bajo demanda, es decir, que no hay que asignarles previamente un tamaño prefijado. Por ejemplo, este código al final contendrá un vector de 10 elementos:

```
for i = 1:10
    a(i) = sqrt(i);
endfor
```

Aunque no debemos hacer uso de esta creación dinámica salvo en casos irremediables, pues en el anterior ejemplo hemos reasignado memoria 10 veces, pues hemos creado primero un vector de tamaño 1, luego otro de tamaño 2, etc. En cambio, si hubiésemos previamente creado un vector de ceros de tamaño 10, con `zeros(1,10)`, el bucle no necesitaría cambiar el tamaño del vector. Aunque la mejor medida para conseguir velocidad es aprovechar la notación matricial de octave, y en vez del anterior ejemplo, escribir `a=sqrt(1:10)`; que hace lo mismo en una sola línea y es aún más rápido. Dejemos el inciso y sigamos.

Las expresiones más usadas son las aritméticas, que detallamos como:

X+Y y **X-Y** La suma y la resta. Si ambos operandos son matrices, el número de filas y columnas deben coincidir. Si uno es un número, su valor es añadido o restado respectivamente a todos los elementos del otro operando.

X.+Y y **X.-Y** Suma o resta elemento a elemento. Son equivalentes a las anteriores.

X*Y Producto de matrices. El número de columnas de X debe coincidir con el de filas de Y.

X.*Y Producto de dos elementos. Si ambos operandos son matrices, sus dimensiones deben coincidir.

X/Y División por la derecha. Esto es conceptualmente equivalente a la expresión $(y^{-1}) * x^{-1}$ usada para resolver sistemas lineales, pero que se calcula sin hacer la inversa de y^{-1} .

X./Y División por la derecha elemento a elemento.

X\Y División por la izquierda. Esto es conceptualmente equivalente a la expresión $inverse(x) * y$ pero que se calcula sin hacer la inversa de x .

X.\Y División por la izquierda elemento a elemento.

XY o **X**Y** Potencia. Si X e Y son escalares, devuelve X elevado a la potencia de Y . Si X es escalar e Y es una matriz cuadrada, se calcula usando autovalores. Si X es una matriz cuadrada e Y es un escalar, la matriz se calcula con repetidas multiplicaciones si Y es entero y usando autovalores si no es entero.

X.Y o **X.**Y** Potencia elemento a elemento. Si ambos sin matrices, sus dimensiones deben coincidir.

X.´ Traspuesta.

X´ Traspuesta compleja conjugada. Para valores reales es equivalente a la traspuesta. Para valores complejos, es equivalente a la expresión `conj(X.´)`.

Luego tenemos los operadores de comparación, que pueden ser `<` menor, `<=` menor o igual, `=` igual, `>=` mayor o igual, `>` mayor y cualquiera de estos `!=`, `~=` o `<>` distinto. Aplicados entre matrices devuelven una matriz de unos y ceros, con unos en los elementos donde la condición se cumpla y cero donde no se cumpla. También tenemos operadores booleanos y de incrementación y decrementación, pero que no los vamos a nombrar.

1.4. Control de flujo

Las sentencias para el control de flujo son las típicas de cualquier lenguaje de programación. Pondremos un simple ejemplo de cada una para que sirva de referencia en el caso que tuvieras que usarlas. Todas las sentencias se caracterizan por finalizar con una sentencia `end*`.

1.4.1. if

El caso más general de `if` es la estructura `if-elseif-else-endif`, que se muestra aquí.

```
if (rem (x, 2) == 0)
  printf ("x es par\n");
elseif (rem (x, 3) == 0)
  printf ("x es impar y divisible por 3\n");
else
  printf ("x es impar\n");
endif
```

1.4.2. switch

Es de reciente implantación así que se considera experimental (con respecto a la versión 2.0.5). Es una mera traducción de un bloque `if`, así que las condiciones siempre se miran de arriba a abajo y se ejecuta el código de la primera que sea verdadera y luego se sale. La forma de uso es la siguiente:

```
switch x
  case (x>=5)
    printf("x es mayor o igual que 5\n");
  case (x>=2)
    printf("x es mayor o igual que 2 y menor que 5\n");
  otherwise
    printf("x es menor que 2\n");
endswitch
```

1.4.3. while

Primero se comprueba la condición, y si es válida se ejecuta el cuerpo y el proceso se repite. Si no es válida se sale.

```
fib = ones (1, 10);
i = 3;
while (i <= 10)
  fib (i) = fib (i-1) + fib (i-2);
  i++;
endwhile
```

1.4.4. do-until

Es el mismo caso que el `while`, pero comprobando la condición al final, después de haber ejecutado una vez el cuerpo. Es decir, se ejecuta el cuerpo y se comprueba la condición, y si es válida se ejecuta el cuerpo de nuevo y el proceso se repite. Si no es válida se sale.

```
fib = ones (1, 10);
i = 2;
do
  i++;
  fib (i) = fib (i-1) + fib (i-2);
until (i == 10)
```

1.4.5. for

La variable del bucle se asigna consecutivamente a todos los elementos de un vector. En el caso de ejemplo, la variable `i` toma los valores 3, 4, ..., 9 y 10, ejecutando luego el cuerpo.

```
fib = ones (1, 10);
for i = 3:10
  fib (i) = fib (i-1) + fib (i-2);
endfor
```

1.4.6. break/continue

La sentencia `break` permite salir de un bucle `for` o `while` y seguir la ejecución del programa en la sentencia siguiente al bucle. La sentencia `continue` simplemente se salta todo el cuerpo y realiza la siguiente iteración del bucle sin salirse.

1.4.7. unwind_protect/try

Octave permite dos formas limitadas de manejo de excepciones. Más información en el manual.

1.5. Funciones

Pongamos un ejemplo de como se define una función en octave. En este ejemplo, a la función se le pasa un argumento que debe ser un vector, y se devuelve la media de sus componentes:

```
function retval = avg (v)
  # ayuda: la funcion avg(v) devuelve el promedio
  # de las componentes del vector v

  retval = sum (v) / length (v);
endfunction
```

Como se puede apreciar, es una función con un argumento y un valor de retorno. En octave, los argumentos de la función son locales, es decir, que los argumentos son copias de los originales y si se modifican desde dentro de la función, los cambios no son vistos por el llamante. Esto implica que la única forma que tenemos de devolver valores al llamante sea usando valores de retorno como en el ejemplo. Podemos hacer que devuelva más de un valor de retorno, disponiéndolos entre corchetes, como en este otro ejemplo que calcula el máximo de un vector.

Fichero vmax.m

```
function [max, idx] = vmax (v)
  # ayuda: vmax(v) devuelve el máximo valor de un vector
  # y la posición que ocupa

  idx = 1;
  max = v (1);
  for i = 2:length (v)
    if (v (i) > max)
      max = v (i);
      idx = i;
    endif
  endfor
endfunction
```

Ejemplo 1.1: La función recibe un vector y devuelve dos valores, el máximo valor encontrado en el vector y la posición donde se encuentra.

Excepto para casos muy simples, no es práctico teclear las funciones en la línea de comandos. Las funciones se suelen guardar en ficheros, que se pueden editar fácilmente y guardarlas para su uso posterior. Existe una pauta que hay que seguir, y es que el nombre del archivo debe ser el nombre de la función con la extensión `.m` en el directorio de trabajo. Por ejemplo, la función `vmax` que acabamos de definir la tendríamos que guardar en un fichero llamado `vmax.m` para que funcionase.

De igual forma se pueden colocar en ficheros instrucciones sueltas sin formar funciones. Esto se denomina *ficheros script*. Cuando se ejecuta uno de ellos, las instrucciones que contiene se ejecutan como si se escribieran una a una por la línea de comandos.

Cuando se le ordena a `octave` que ejecute una función el proceso que realiza es el siguiente: la lee y la analiza sintácticamente; en caso que no existan errores, la compila en un formato interno y pasa a ejecutarla. En caso de que el usuario le ordene a `octave` ejecutarla otra vez, si el fichero no ha sido modificado, `octave` utiliza la función ya compilada ahorrando el tiempo de lectura y análisis.

Veamos esto con un ejemplo. Cojamos el código fuente de la función anterior, guardémoslo en un fichero llamado `vmax.m` y tecleemos lo siguiente:

```
octave:1> clear
octave:2> who
octave:3> help vmax
vmax is the user-defined function from the file
/home/alberto/cvs/Libro\CILA/ejemplos/vmax.m

ayuda: vmax(v) devuelve el máximo valor de un vector
y la posición que ocupa
...

octave:3> [max idx]=vmax([5 4 3 2 7 5 4 9 6])
max = 9
idx = 8
octave:4> who

*** currently compiled functions:

vmax

*** local user variables:

idx  max
```

Como estamos viendo, efectivamente `who` al principio, después de limpiar la memoria, no nos decía nada, pero después de ejecutar la función, nos informa que la tiene ya compilada y almacenada para posteriores usos.

Donde se hagan usos más exigentes de octave, este esquema, aunque es inteligente, resulta poco óptimo, pues lo que se hace es traducir la función a un lenguaje intermedio que luego octave interpreta cuando se le manda a ejecutar. Si nuestro uso requiere de la máxima potencia de cálculo, viene bien saber que octave es capaz de ejecutar código objeto compilado con gcc, es decir, código de c, c++, fortran o pascal. Veamos un ejemplo en c++ para darnos cuenta lo sencillo que es y de las posibilidades que nos puede abrir.

Fichero `oregonator.cc`

```
#include <octave/oct.h>

DEFUN_DLD(oregonator, args, "El 'oregonador'.")
{
  ColumnVector dx(3);

  ColumnVector x(args(0).vector_value());

  dx(0) = 77.27 * (x(1) - x(0) * x(1) + x(0) - 8.375e-06 * pow(x(0), 2));
  dx(1) = (x(2) - x(0) * x(1) - x(1)) / 77.27;
  dx(2) = 0.161 * (x(0) - x(2));

  return octave_value(dx);
}
```

Ejemplo 1.2: Esta función recibe un vector de tres elementos como argumento de entrada y hace un cálculo con esos valores para devolver un resultado.

No vamos a analizar este ejemplo en profundidad, sino solamente dar unas pistas sobre como está hecho. En primer lugar vemos un `include <octave/oct.h>`. Lo que hace es definir todos los tipos de datos de octave y las funciones nativas como clases y métodos de C++. Con esto queremos decir que si en octave podíamos hacer `length(vector)` para obtener las dimensiones de un vector, en C++ podremos hacer lo mismo usando `vector.length`. Entendido esto, el resto son meras particularidades sintácticas específicas de C++ en las que no entraremos. Para compilar este código, vayámonos a un shell y escribamos lo siguiente:

```
alberto@baifito:ejemplos\$ ls oregonator.*
oregonator.cc
alberto@baifito:ejemplos\$ mkoctfile oregonator.cc
alberto@baifito:ejemplos\$ ls oregonator.*
oregonator.cc oregonator.o oregonator.oct
```

El fichero `oregonator.o` es un código objeto como cualquier otro que se obtiene con gcc. El fichero `oregonator.oct` es la función recompilada para octave. Para probarla, entremos en octave y tecleemos lo siguiente:

```
octave:1> help oregonator
oregonator is the dynamically-linked function from the file
```

```
/home/alberto/cvs/Libro_CILA/ejemplos/oregonator.oct
```

```
El ‘oregonador’.
```

```
...
octave:2> oregonator([1 2 3])
ans =

    77.269353
   -0.012942
   -0.322000
```

Como verás, en la ayuda de la función aparece el texto que definimos en el código fuente, así como una advertencia de que la función está dinámicamente linkada a las librerías de octave. En el segundo paso vemos que la ejecución de la función también funciona. Eso sí, si no pasamos los parámetros correctamente veremos como un *Segmentation Fault* cierra nuestro octave. El problema es que no comprobamos el tipo de dato en el código de C++, pero añadiendo las comprobaciones pertinentes podremos manejar estos casos excepcionales y no se dará este problema.

La ejecución de código compilado como éste puede llegar a ser hasta 10 veces más rápido que el código interpretado, en fichero con extensión `.m`. Además, podemos enlazar (link) funciones de otros lenguajes simplemente escribiendo una capa tal como hemos visto que se encargue de leer/escribir los datos en estructuras de octave. Y como en esos otros lenguajes se pueden hacer ventanas gráficas o acceder a periféricos, eso significa que en octave también se podrá. Las rutinas de procesamiento de imágenes que veremos (o las de sonido, que no veremos) son un ejemplo de ello. Para más ejemplos, consultar los ficheros con extensión `.cc` de la distribución de octave.

1.6. Representación gráfica

En los siguientes ejemplos entraremos en el campo de la representación gráfica, que también es sencillo (NOTA: no olvidarse los puntos y comas al final de línea pues los vectores son algo largos para estarlos visualizando, y pulsar `q` para cerrar las gráficas).

Para hacer representaciones gráficas deberás haber ejecutado Octave desde un shell dentro de las X puesto que la representación gráfica se realiza usando Gnuplot, cuya forma de funcionar por defecto es en entorno X-Window. No nos adentraremos demasiado pues describiremos Gnuplot en otro capítulo.

Presentación en una dimensión

La función `plot(vector)` o `plot(x,y)` es muy sencilla de usar. La diferencia entre ambas llamadas es que cuando presentamos un vector, el eje x se numera automáticamente de 1 en adelante, mientras que la segunda forma de llamarla, el valor del eje x está definido por nosotros. Veamos el siguiente ejemplo que presenta un período de una senoidal.

```
octave:25> x=[0:0.01:1];
```

```
octave:26> y=sin(2*pi*x);
octave:27> plot(x) # presentamos una recta
octave:28> plot(y) # presentamos una senoidal
octave:29> plot(x,y) # senoidal, pero con eje x bien puesto
```

Se pueden presentar varias gráficas en una usando la opción `hold on` y se pueden añadir títulos a las gráficas con un tercer parámetro a la función `plot`. Veamos un ejemplo:

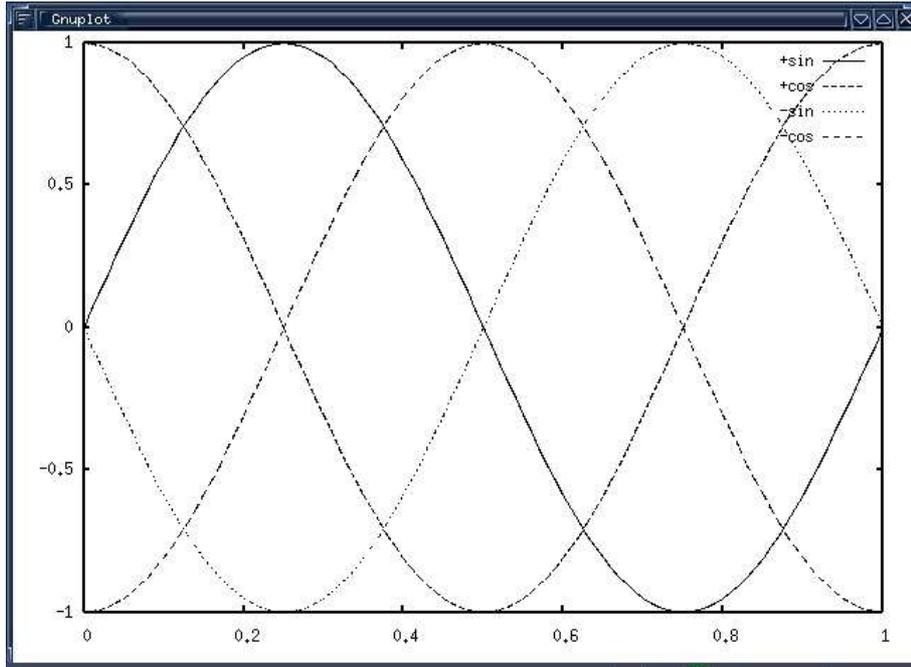


Figura 1.1: Múltiples líneas por gráfica.

Fichero sinus.m

```
#!/usr/bin/octave -qi
x=[0:0.01:1];      # rango variacion eje x
clearplot;        # borramos grafica
subplot(1,1,1);    # un unico plot centrado
axis("auto","normal"); # ejes automaticos
hold on;          # superpone siguientes graficos
plot(x, +sin(2*pi*x), '+sin;')
plot(x, +cos(2*pi*x), '+cos;')
plot(x, -sin(2*pi*x), '-sin;')
plot(x, -cos(2*pi*x), '-cos;')
```

Ejemplo 1.3: Mostramos gráficas sobreimpresas entre ellas y les asignamos títulos.

Octave y Gnuplot permiten diferentes estilos en las gráficas. Por ejemplo, en polares, de la forma polar(θ, ρ):

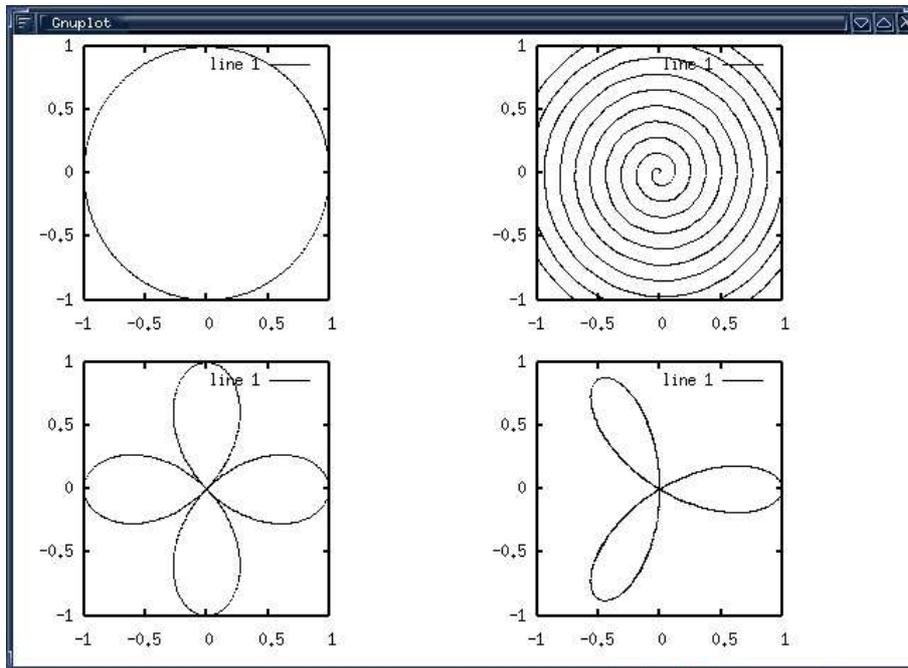


Figura 1.2: Diagramas en coordenadas polares

Fichero polares.m

```
#!/usr/bin/octave -qf
x=[0:0.01:2*pi]; # rango variacion angulo
clearplot;      # borra grafico
axis([-1 1 -1 1], "square"); # ejes manuales
hold off;       # no superpone siguientes graficos
subplot(2,2,1); # cuadrante 1 de 4
polar(x,ones(size(x))); # circulo
subplot(2,2,2); # cuadrante 2 de 4
polar(10*x,x/5); # espiral de arquimedes
subplot(2,2,3); # cuadrante 3 de 4
polar(x,cos(2*x));# rosa de cuatro petalos
subplot(2,2,4); # cuadrante 4 de 4
polar(x,cos(3*x));# rosa de tres petalos
```

Ejemplo 1.4: Mostramos cuatro gráficas diferentes en polares en cuatro cuadrantes.

También en forma de histograma, adornando las gráficas con títulos.

Fichero histo.m

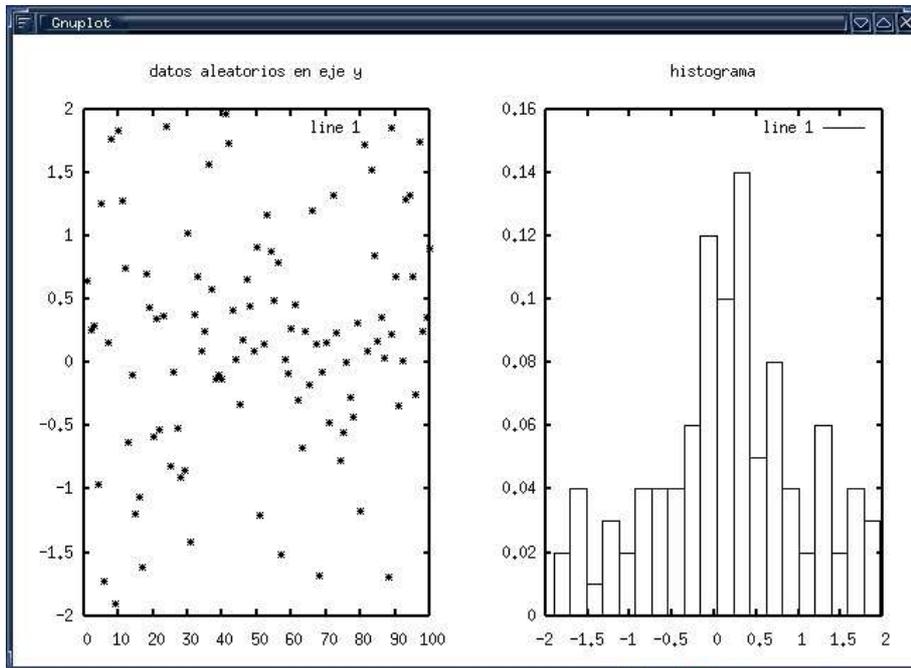


Figura 1.3: Representación de histogramas

```
#!/usr/bin/octave -qf
y=randn(100,1); # matriz num aleatorios normal
clearplot; # borra grafico
axis("auto","normal"); # ejes automáticos
hold off; # no superpone siguientes graficos
subplot(1,2,1); # cuadrante 1 de 2
title("datos aleatorios en eje y");
plot(y,'*'); # plotea los datos aleatorios con *
subplot(1,2,2); # cuadrante 2 de 2
title("histograma");
hist(y,20,1); # histograma de 20 barras normalizado a 1
```

Ejemplo 1.5: Mostramos un histograma y sus datos de partida, con títulos sobre las gráficas.

Representación en 3D

La función `mesh(x,y,z)` hace una representación 3D dados dos vectores `x` e `y` para los ejes `x` y `y` y una matriz bidimensional `z` que será la coordenada `Z` en un espacio tridimensional. Otra función llamada `contour(x,y,z)` con los mismos argumentos que `mesh()`, dibujará las curvas de nivel de la superficie. En este ejemplo, lo más complicado será generar una matriz `z` bonita. Una vez tenemos la matriz y los ejes, las dos llamadas son directas.

Fichero `meshplot.m`

```
#!/usr/bin/octave -qf
1; # limpia memoria

function configura
    hold off;           # no superpone siguientes graficos
    clearplot();       # limpiamos
    axis("auto","normal"); # ejes automaticos
    subplot(1,1,1);     # nos ponemos en una sola ventana
    xlabel("eje x");    #
    ylabel("eje y");    # ponemos etiquetas
    zlabel("eje z");    #
endfunction

x=[-10:0.5:10];       # vector del eje x
y=[-10:0.5:10];       # vector del eje y
[mx,my]=meshgrid(x,y); # genera matrices de ejes
mr=sqrt(mx.^2+my.^2); # matriz que contiene el radio
mz=sin(mr)./mr;       # funcion z=sin(r)/r
configura();
subplot(1,2,1);       # cuadrante 1 de 2
title("plano eje x creciente, eje y constante");
mesh(x,y,mx);
subplot(1,2,2);       # cuadrante 2 de 2
title("plano eje x constante, eje y creciente");
mesh(x,y,my);
pause(5);             # pausar 5 segundos
configura();
subplot(1,2,1);       # cuadrante 1 de 2
title("superficie 3d");
mesh(x,y,mz);
subplot(1,2,2);       # cuadrante 2 de 2
title("curvas de nivel");
contour(x,y,mz);
```

Ejemplo 1.6: Mostramos una sinc en 3 dimensiones, ası como los planos usados para generarla.

1.7. Matrices

Octave tiene una amplia coleccion de funciones para trabajar con matrices y vectores. Las veremos en un ejemplo.

```
octave:20> c=diag([1,2,3,4]) # creacion de matrices diagonales
c =
    1  0  0  0
    0  2  0  0
```

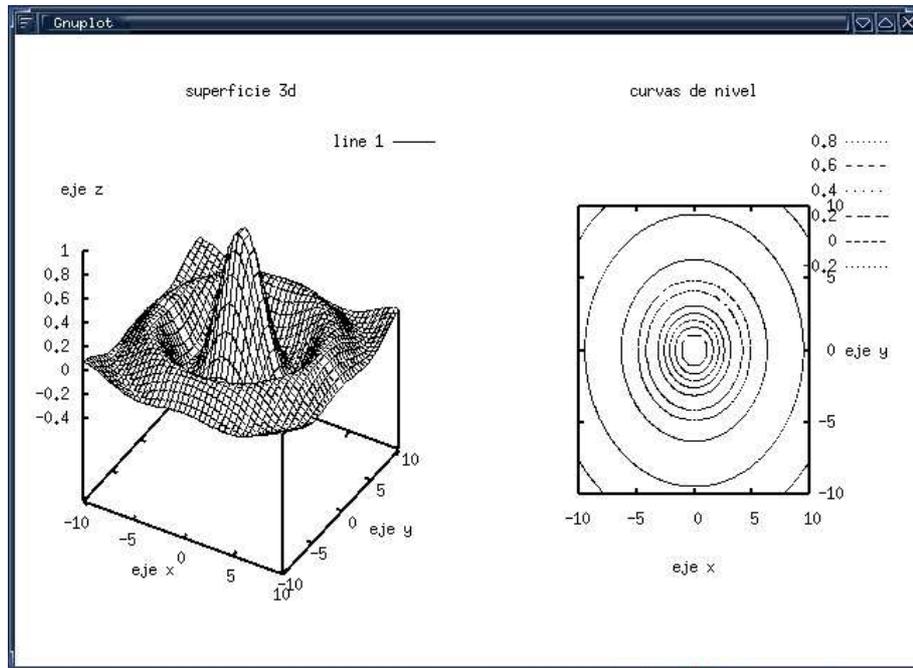


Figura 1.4: Gráficas en tres dimensiones

```
0 0 3 0
0 0 0 4
```

```
octave:21> inv(c) # inversa de una matriz
```

```
ans =
  1.00000  0.00000  0.00000  0.00000
  0.00000  0.50000  0.00000  0.00000
  0.00000  0.00000  0.33333  0.00000
  0.00000  0.00000  0.00000  0.25000
```

```
octave:22> det(c) # determinante de una matriz
```

```
ans = 24
```

```
octave:23> eye(4) # matriz identidad de dimensión 4
```

```
ans =
  1 0 0 0
  0 1 0 0
  0 0 1 0
  0 0 0 1
```

```
octave:15> ones(2,3) # matriz 2x3 repleta de unos
```

```
ans =
```

```

1 1 1
1 1 1

octave:16> zeros(1,7) # matriz 1x7 repleta de ceros
ans =

0 0 0 0 0 0 0

octave:24> rand(4,3) # matriz de números aleatorios 4x3
ans =
0.85927 0.43700 0.85462
0.88050 0.27016 0.52905
0.58098 0.54402 0.29237
0.41791 0.73324 0.45943

octave:5> randn(3,2) # matriz de números aleatorios gaussiana
ans =

0.64262 -1.03740
0.31010 -1.38565
-0.64096 0.58650

# resta cada elemento con su anterior
octave:7> diff([1 2 4 5 7 9 11 14])
ans =

1 2 1 2 2 2 3

# encuentra índices elementos no nulos
octave:8> find ([0 0 0 0 0 0 1 0 0 0 5 0])
ans =

7 11

# subdivide linealmente el intervalo [1,10] en 8 puntos.
octave:9> linspace (1, 10, 8)
ans =

1.0000 2.2857 3.5714 4.8571 6.1429 7.4286 8.7143 10.0000

# subdivide logarítmicamente el intervalo [10^0,10^3] en 6 puntos.
octave:13> logspace (0, 3, 6)
ans =

1.0000 3.9811 15.8489 63.0957 251.1886 1000.0000

# factorización lu de una matriz

```

```
octave:26> [a,b,c]=lu([1,3,2;3,2,1;3,2,6])
```

```
a =
```

```
1.00000 0.00000 0.00000
0.33333 1.00000 0.00000
1.00000 0.00000 1.00000
```

```
b =
```

```
3.00000 2.00000 1.00000
0.00000 2.33333 1.66667
0.00000 0.00000 5.00000
```

```
c =
```

```
0 1 0
1 0 0
0 0 1
```

```
octave:27> [a,b,c]=qr([1,3,2;3,2,1;3,2,6]) # factorización QR
```

```
a =
```

```
-0.312348 -0.752156 -0.580259
-0.156174 -0.561851 0.812362
-0.937043 0.344361 0.058026
```

```
b =
```

```
-6.40312 -3.12348 -3.59200
0.00000 -2.69145 -1.40463
0.00000 0.00000 2.03091
```

```
c =
```

```
0 0 1
0 1 0
1 0 0
```

1.8. Ecuaciones diferenciales

Fichero ode.m

```
#!/usr/bin/octave

clear;

function xdot = f (x, t)
    xdot = zeros (3,1);
    xdot(1) = 77.27 * (x(2) - x(1)*x(2) + x(1) - 8.375e-06*x(1)^2);
    xdot(2) = (x(3) - x(1)*x(2) - x(2)) / 77.27;
    xdot(3) = 0.161*(x(1) - x(3));
endfunction

# condicion inicial
x0 = [ 4; 1.1; 4 ];
# generaci3n del eje t para la simulacion
t = linspace (0, 50, 100);
# simulacion
y = lsode ("f", x0, t);
hold off;
subplot(3,1,1);
plot(t,y(:,1),',x(1);');
subplot(3,1,2);
plot(t,y(:,2),',x(2);');
subplot(3,1,3);
plot(t,y(:,3),',x(3);');
```

Ejemplo 1.7: Resuelve una EDO ordinaria de tercer orden y muestra por pantalla la evoluci3n de los $x(t)$.

1.9. Polinomios

Un polinomio de grado r en octave se presenta como un vector de dimensiones $r + 1$. A partir de 3l se pueden realizar operaciones.

```
octave:10> a=[1,2,3]; # a(x)=x^2+2x+3
octave:11> b=[3,2,3,2]; #b(x)=3x^3+2x^2+3x+2
octave:19> polyout(a)
1*s^2 + 2*s^1 + 3
octave:18> polyout(b)
3*s^3 + 2*s^2 + 3*s^1 + 2
octave:20> polyout(conv(a,b)) # producto de polinomios
3*s^5 + 8*s^4 + 16*s^3 + 14*s^2 + 13*s^1 + 6
octave:13> [coc, resto]=deconv([3 8 16 14 13 6],b); # divisi3n
octave:24> polyout (coc) # el cociente de la divisi3n
1*s^2 + 2*s^1 + 3
octave:25> polyout (resto) # el resto de la divisi3n
0*s^5 + 0*s^4 + 0*s^3 + 0*s^2 + 0*s^1 + 0
```

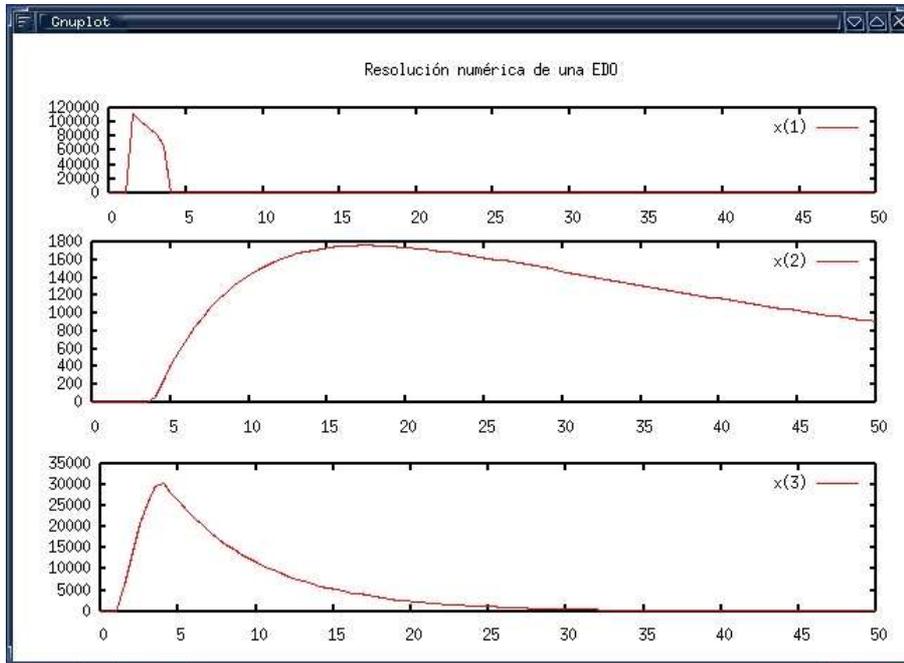


Figura 1.5: Resolución numérica de EDO

```
octave:26> polyout(polyderiv (b)) # la derivada de b(x)
9*s^2 + 4*s^1 + 3
octave:27> polyout(polyinteg (b)) # la integral de b(x)
0.75*s^4 + 0.666667*s^3 + 1.5*s^2 + 2*s^1 + 0
octave:17> polyval (b,2) # b(x) evaluado en x=2
ans = 40
```

1.10. Teoría de control

La versión 2.1 de octave incorpora una Toolbox de Control. El control es una rama de la ingeniería dedicada a modelar sistemas mediante las ecuaciones diferenciales que relacionan su salida con su entrada y predecir su comportamiento frente a diferentes entradas. En la toolbox se utilizan estructuras para abstraer al usuario el concepto de función de transferencia de un sistema.

1.11. Procesamiento de señales

En esta categoría están todas las funciones dedicadas a trabajar con espectros de señales. Tanto la *FFT*, como filtros digitales *FIR* e *IIR*, diferentes tipos de ventanas, o cálculo de modelos ARMA.

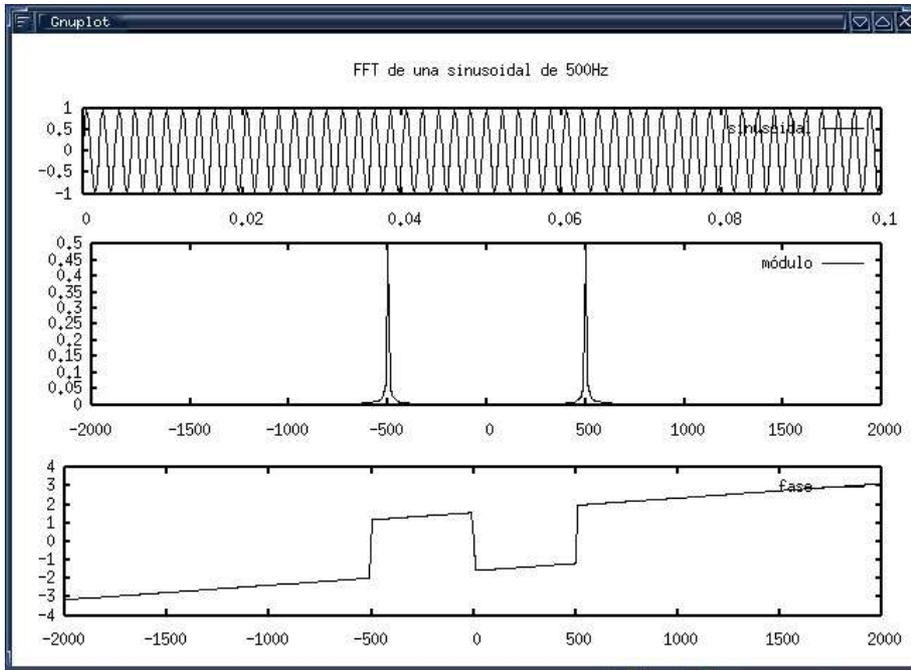
La transformada de Fourier discreta, más conocida por el nombre de su algoritmo FFT (**Fast Fourier Transform**), es la versión discreta y periódica de la transformada exponencial de Fourier.

Es una función muy usada en ingeniería y en la vida real. La tomaremos como la función para la ingeniería por antonomasia, y en este ejemplo veremos qué sencillo resulta obtener la transformada de Fourier discreta de una senoidal y un pulso.

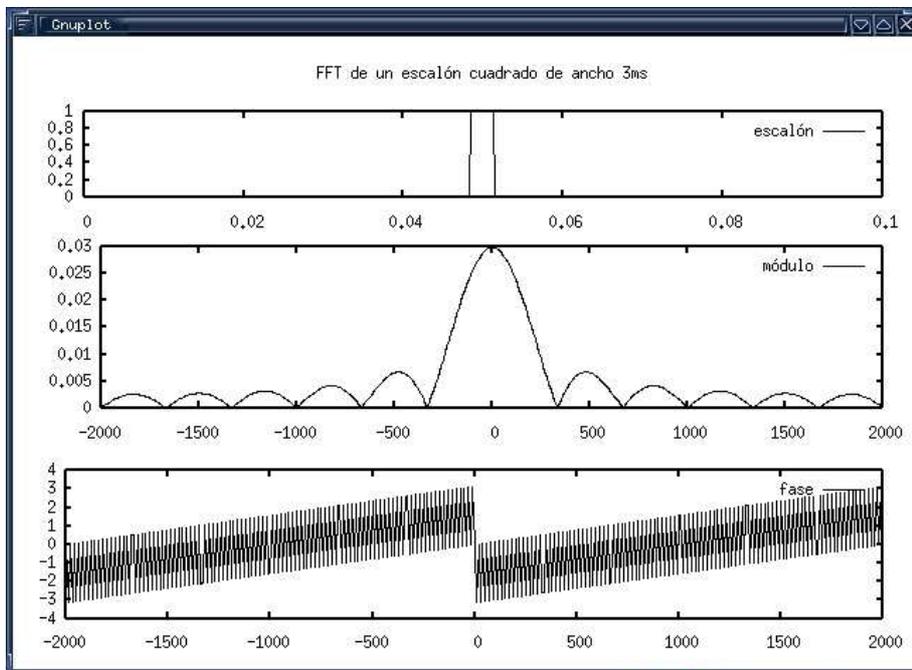
Fichero fftview.m

```
#!/usr/bin/octave
clear;
hold off;
T=0.1;           # periodo de muestreo
N=100;          # numero de muestras
W=3*(2*pi/T);   # frecuencia de la senoidal
A=20*T;         # ancho del escalon
t=T*[0:N];     # escala temporal
f=(2*pi/T)*[-N/2:N/2]/N; # escala en frecuencias
# la primera es la fft de un seno de 20Hz
title("Seno de 20Hz");
x1=sin(2*pi*W*t);
y1=fft(x1)/length(t);
y1=fftshift(y1);
subplot(3,1,1);
clearplot;
plot(t,x1,';sinusoidal;');
subplot(3,1,2);
clearplot;
plot(f,abs(y1),';modulo;');
subplot(3,1,3);
clearplot;
plot(f,arg(y1),';fase;');
pause(5);
# la segunda es la de un escalon centrado de ancho A sg.
x2=abs(t-T*N/2)<(A/2);
y2=fft(x2)/length(t);
y2=fftshift(y2);
subplot(3,1,1);
clearplot;
plot(t,x2,';escalon;');
subplot(3,1,2);
clearplot;
plot(f,abs(y2),';modulo;');
subplot(3,1,3);
clearplot;
plot(f,arg(y2),';fase;');
```

Ejemplo 1.8: Muestra por pantalla el módulo y la fase de dos funciones: una sinusoidal (que debe dar dos deltas de dirac) y un pulso (que debe dar una sinc).



(a) Ejemplo de FFT aplicado a una sinusoidal



(b) Ejemplo de FFT aplicado a una función escalón

1.12. Tratamiento de imágenes

Por último veamos un ejemplo de cómo se pueden cargar, realizar modificaciones, visualizar y salvar imágenes usando las rutinas de octave. Con esto podremos realizar muchas operaciones útiles en procesado, realzado y análisis de imágenes, útiles en muchas áreas de las ciencias.

En primer lugar suponemos sabido que una imagen se puede entender como una matriz donde cada punto tiene un color diferente. Cada punto se llama *píxel*. Hay muchas formas de representar píxeles, y octave maneja tres de ellas, que son las siguientes:

escala de grises En una imagen en escala de grises cada píxel se representa por un valor de punto flotante comprendido entre 0, que significa negro, y 1, que significa blanco. Por tanto, la representación de una imagen es una matriz llena de valores comprendidos entre 0 y 1. En este caso, un valor de 0.5 representaría un color gris que tiene partes iguales de blanco y negro mientras que un valor de 0.2 representaría un color que tiene 20% de negro y el resto de blanco.

color RGB Cualquier color en la naturaleza se puede aproximar (no es una correspondencia exacta) como suma de sus tres componentes de color, a saber: rojo (R, red), verde (G, green) y azul (B, blue). De esta forma, cada píxel de una imagen en color se puede representar como un terna de valores de sus tres componentes. Por tanto, una imagen en formato RGB son tres matrices del mismo tamaño, donde cada una almacena los valores de una componente.

color indexado Como en una imagen suele haber colores repetidos, esta representación lista por un lado los colores, de manera consecutiva y sin repetirse, y por otro lado los píxeles, cuyo color se almacena buscando el valor en la tabla de colores y almacenando el índice correspondiente en el vector.

La función usada para representar imágenes en pantalla es `imshow()`. En primer lugar la usaremos para representar valores en escala de grises. Vamos a probar creando una matriz que tenga variedad de valores y presentándola. Veamos el ejemplo de uso de esta función:

```
octave:15> a=0:0.01:1;
octave:16> ma=a'*a;
octave:17> imshow(ma,2)
octave:18> imshow(ma,16)
octave:19> imshow(ma,256)
```

Observando la matriz `ma` observamos que todos sus valores están en el rango $[0, 1]$. Si vemos su tamaño, vemos que es 100x100. Las diferentes llamadas a `imshow` se diferencian en el segundo parámetro que representa el número de colores discretos en el que presentará la imagen continua. En el primer caso veremos solamente dos colores, en la segunda 16 y en la tercera 256.

Probemos ahora a trabajar con una imagen en color. La función `loadimage()` se utiliza para cargar imágenes de un archivo. Las imágenes se devuelven en formato indexado, donde `c` representará la matriz y `cmap` representará los colores. Veamos el ejemplo:

```
octave:26> [c,cmap]=loadimage("tux.img");
octave:27> imshow(c,cmap)
```

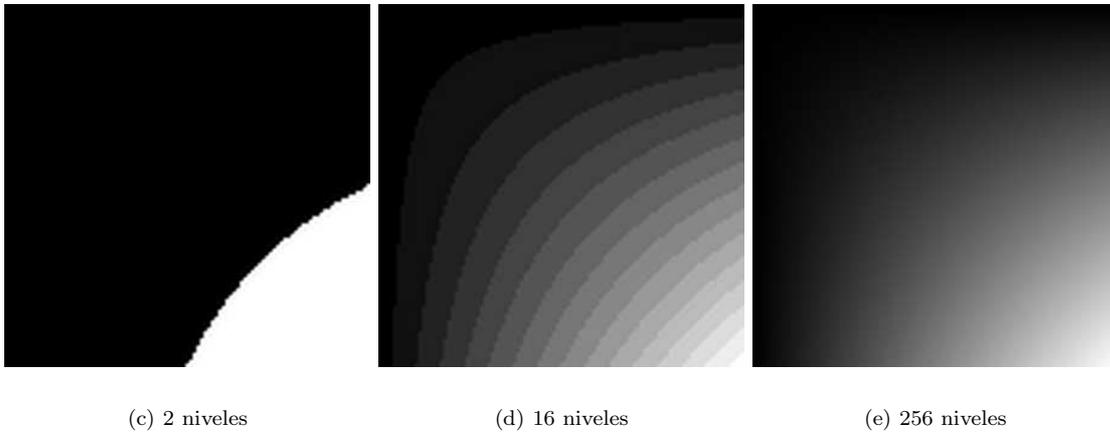


Figura 1.6: El gradiente anterior visualizado a varios niveles de gris

Como verás, se visualiza con la misma función. Ahora, usemos esta imagen como base para jugar un poco. Primero, transformémosla a un formato más manejable. Empezaremos pasándola a RGB con la función `ind2rgb` transforma la imagen de indexada a RGB. Luego, con `imshow` la representaremos.

```
octave:28> [r,g,b]=ind2rgb (c,cmap);
octave:29> imshow(r,g,b)
octave:30> imshow(g,r,b)
octave:31> imshow(b,g,r)
```



Figura 1.7: Los colores de la imagen vienen representados por sus componentes RGB. Si se invierten, los colores cambian.

Observamos que cada canal contiene sus valores, y cambiando el orden hacemos creer a `imshow` que los valores son diferentes y nos muestra imágenes con los colores trasladados. Podemos probar también sustituyendo las matrices `r`, `g` o `b` por unos o ceros y comprobar el efecto de quitar o añadir canales. El trabajo con imágenes RGB es complejo, así que usaremos una imagen en escala de grises para seguir efectuando nuestras pruebas.

```
octave:32> g=ind2gray(c,cmap);
octave:35> imshow(g,2)
octave:36> imshow(g,4)
octave:37> imshow(g,16)
octave:65> imshow(g,256)
```

Comenzaremos los análisis con un filtrado/realzado. Estas operaciones se realizan barriendo todos los píxeles de la imagen y creando una nueva imagen donde cada píxel es una ponderación de los valores de cada píxel y sus vecinos según unas matrices preestablecidos. Las matrices

$$\frac{1}{5} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Filtro pasa-baja Filtro pasa-alta Detector de bordes

Ahora veamos como se aplicaría un filtro. Este filtro es el primero, que genera una imagen donde cada píxel es un promedio entre el correspondiente en la imagen original y sus cuatro vecinos por cada lado.

```
octave:41> h=zeros(256,256);
octave:42> for i=2:255;
>     for j=2:255;
>         h(i,j)=(g(i,j)+g(i-1,j)+g(i+1,j)+ \
>                 g(i,j-1)+g(i,j+1))/5;
>     endfor;
> endfor;
octave:43> imshow(g,256)
octave:44> imshow(h,256)
```

Mostrando la imagen original y retocada comprobamos que el efecto que tiene es el de suavizar los bordes de la imagen, el de emborronarla.

Habrás notado que has tenido que esperar un rato (según la velocidad de tu máquina) durante un buen rato para ver el resultado. Este procedimiento es optimizable atendiendo a las especiales características de manejo de matrices de octave. En vez de hacer las operaciones elemento a elemento con sus superiores, inferiores, izquierdo y derecho, podemos hacerlas globalmente, simplemente trabajando con las matrices completas y operando con ellas con sus desplazadas una posición hacia arriba, abajo, izquierda y derecha. El ejemplo esta aquí y comprobarás que el resultado es notablemente más rápido.

```
octave:44> i=2:255;
octave:45> h=(g(i,i)+g(i-1,i)+g(i+1,i)+g(i,i-1)+g(i,i+1))/5;
octave:46> imshow(g,256)
octave:47> imshow(h,256)
```

Comprobemos ahora los otros dos filtros. El siguiente filtro pasa-alta se implementaría así. El resultado, al contrario que el anterior, acentúa los bordes en la imagen, resultando los pequeños detalles más destacados a simple vista.

```
octave:44> i=2:255;
octave:45> h=5*g(i,j)-g(i-1,j)-g(i+1,j)-g(i,j-1)-g(i,j+1);
octave:46> imshow(g,256)
octave:47> imshow(h,256)
```



Figura 1.8: Cuando aplicamos un filtro pasa-baja se suaviza la imagen y si aplicamos un pasa-alta, sus detalles se acentúan.

Y por último, el detector de bordes, que lo que hace es presentar de color blanco las zonas que en la imagen original tienen cambios más rápidos, mientras que las zonas más homogéneas las presenta en negro. Con esto, conseguimos detectar bordes acusados en la imagen.

```
octave:44> i=2:255;
octave:45> h=5*g(i,j)-g(i-1,j)-g(i+1,j)-g(i,j-1)-g(i,j+1);
octave:46> imshow(g,256)
octave:47> imshow(h,256)
```



Figura 1.9: Imagen tras aplicar detector de bordes.

Capítulo 2

GNUplot

Gnuplot es el programa encargado de hacer las gráficas 2D y 3D que se visualizaban en Octave. Gnuplot es un programa independiente de Octave, que usado por sí mismo te permite hacer representaciones de funciones continuas y de tablas de datos. Octave sólo usa un subconjunto de las funcionalidades de Gnuplot.

La primera característica de Gnuplot es que es muy similar a Octave en funcionamiento, es decir, que posee una interfaz de comandos muy poderosa que también puedes utilizar escribiendo scripts. Esta forma de trabajar tiene sus desventajas y sus ventajas. Las desventajas es que necesitas una curva de aprendizaje más lenta, donde tienes que haberte mirado por lo menos la descripción de uno de los comandos (`plot`) para poder empezar a hacer algo. Cuando estás tanteando datos mejor que uses otro programa que te permita hacer las cosas más interactivamente. Pero cuando ya tienes claro lo que tienes que hacer, por ejemplo, sobre una tabla de datos, y tienes 100 tablas de datos a las que hacer lo mismo, poder hacer un script puede ser de una gran ayuda.

La otra característica destacable de Gnuplot es la variedad de formatos de salida de que dispone, que se pueden seleccionar en el script. Te permite exportar a formatos vectoriales (`xfig`, `TEX`, `postscript`), formatos bitmap (`png`, `pbm`), o formatos de impresora (`epson`, `hp`, etc). Con esto puedes tener tu gráfica retocada por `xfig` en tu publicación en `LaTeX`, o bien puesta en tu página web (`png`) y o bien impresa directamente en una impresora.

Al ejecutar `gnuplot` en un shell entramos a su línea de comandos:

```
alberto@mencey:~$ gnuplot
```

```
G N U P L O T
Linux version 3.7
patchlevel 1
last modified Fri Oct 22 18:00:00 BST 1999
```

```
Copyright(C) 1986 - 1993, 1998, 1999
Thomas Williams, Colin Kelley and many others
```

Type ‘help’ to access the on-line reference manual
 The gnuplot FAQ is available from
 <<http://www.ucc.ie/gnuplot/gnuplot-faq.html>>

Send comments and requests for help to <info-gnuplot@dartmouth.edu>
 Send bugs, suggestions and mods to <submit@bugs.debian.org>

Terminal type set to ‘x11’
 gnuplot>

Como fuente de ayuda teclea **help** desde dentro del programa y después de una pantalla introductoria te saldrá un prompt sobre el que podrás escribir, o bien un nombre que elegirás de los topics que se te presentan, o bien un nombre de comando si quieres conocer su sintaxis.

Como has visto, el formato de salida es x11 (visualizar en las X). Para ver un listado de los diferentes tipos de salida disponibles usa **set terminal**.

2.1. Representación de expresiones analíticas

La parte más sencilla y práctica de Gnuplot es la presentación de funciones continuas, tanto en forma explícita $y=f(x)$ o $z=f(x,y)$, como puede ser en forma paramétrica: curvas 2D $(x,y)=f(t)$, curvas 3D $(x,y,z)=f(u)$, superficies 3D $(x,y,z)=f(u,v)$.

Con **help functions** tenemos un listado de las funciones que admite. Una gran desventaja que tiene es que muestrea las funciones a intervalos regulares, por tanto, no hace ningún análisis de discontinuidades (lo que se nota en, por ejemplo, la función **floor**), aunque sí se puede configurar para que reduzca el intervalo. Si queremos imponer cual será el rango del eje X o el Y lo ponemos entre corchetes antes de la función. Algunos ejemplos:

```
gnuplot> plot x                # identidad
gnuplot> plot abs(x)          # valor absoluto
gnuplot> plot x**2            # parábola
gnuplot> plot [-1:1] sqrt(1-x**2) # semicircunferencia
gnuplot> plot [] [-0.1:1.1] exp(-x**2) # gaussiana
gnuplot> plot [-1:4] gamma(x) # función gamma
gnuplot> plot floor(x)        # función redondeo hacia abajo
gnuplot> plot x-floor(x)      # diente de sierra
gnuplot> splot x**2+y**2      # plot en 3D
gnuplot> splot sqrt(1-x**2+y**2)
gnuplot> set isosamples 20,20 # cambia la resolución
gnuplot> replot
gnuplot> set isosamples 50,50 # cambia la resolución
gnuplot> set contour          # activa líneas de nivel
gnuplot> replot
gnuplot> set parametric       # modo paramétrico
```

```

dummy variable is t for curves, u/v for surfaces
gnuplot> set samples 500                # mejor resolución (+lento)
gnuplot> plot sin(7*t),cos(5*t)         # lissajous en 2D
gnuplot> splot sin(5*u),sin(6*u),sin(7*u) # lissajous en 3D
gnuplot> set samples 100               # menor resolución (+rápido)
gnuplot> splot cos(u)*cos(v),cos(u)*sin(v),sin(u) # esfera en 3D

```

2.2. Representación de archivos de datos

Gnuplot también tiene un modo para trabajar con archivos de datos con múltiples columnas. Cuando los archivos de datos tienen 1 ó 2 columnas se presentan directamente. Si un archivo tiene más de 2 columnas se pueden presentar columnas arbitrariamente, hacer operaciones matemáticas sencillas entre columnas. Veamos esto en un ejemplo real (bastante prolijo) donde un servidor genera una línea de log de load, logins y carga de cpu, a cada hora y queremos obtener gráficas que muestren la evolución en el tiempo

```
# Ejemplo para la monitorización de carga de un servidor en el tiempo
```

```

set title "Convex      November 1-7 1989      Circadian"
set key left box
set xrange[-1:24]
plot 'gnuplot.dat' using 2:4 title "Logged in" with impulses,\
     'gnuplot.dat' using 2:4 title "Logged in" with points
pause -1 "Hit return to continue"

set xrange [1:8]
#set xdtic
set title "Convex      November 1-7 1989"
set key below
set label "(Weekend)" at 5,25 center
plot 'gnuplot.dat' using 3:4 title "Logged in" with impulses,\
     'gnuplot.dat' using 3:5 t "Load average" with points,\
     'gnuplot.dat' using 3:6 t "%CPU used" with lines
set nolabel
pause -1 "Hit return to continue"
reset

```

Como último ejemplo, vamos a probar un script donde se hacen ajustes por el método de mínimos cuadrados con Gnuplot. En el ejemplo se realizan ajustes a una recta variando los pesos, pero el método de ajuste que utiliza Gnuplot permite poner cualquier función de ajuste, simplemente definiendo las variables y constantes y dando unos valores iniciales a las constantes.

```
# ajustes por mínimos cuadrados en Gnuplot
```

```

y(x) = a*x + b  # función a la que se ajustará
a = 0.0         # valores iniciales

```

```
b = 0.0          # de los parámetros

fit y(x) 'gnuplot-fit.dat' via a, b
set title 'Ajuste sin pesar'
plot 'gnuplot-fit.dat', y(x)
pause -1 "Pulsa enter para continuar"

fit y(x) 'gnuplot-fit.dat' using 1:2:3 via a, b
set title 'Ajuste con mayor peso en bajas temperaturas'
plot 'gnuplot-fit.dat', y(x)
pause -1 "Pulsa enter para continuar"

fit y(x) 'gnuplot-fit.dat' using 1:2:4 via a, b
set title 'Ajuste con mayor peso a altas temperaturas'
plot 'gnuplot-fit.dat', y(x)
pause -1 "Pulsa enter para continuar"

fit y(x) 'gnuplot-fit.dat' using 1:2:5 via a, b
set title 'Ajuste con peso correspondiente a error experimental'
plot 'gnuplot-fit.dat' using 1:2:5 with errorbars, y(x)
pause -1 "Pulsa enter para continuar"
```

Capítulo 3

GNU R

3.1. Introducción

R es a la vez un entorno y un lenguaje de programación para realizar cálculos y gráficos estadísticos. Es un proyecto GNU similar al sistema S desarrollado en Bell Laboratories (formalmente AT&T, ahora Lucent Technologies) por John Chambers y sus colegas. R puede considerarse como una implementación diferente de S. Hay diferencias importantes entre ambos, pero mucho código escrito para el sistema S puede ejecutarse en R sin modificarlo.

R proporciona una amplia variedad de técnicas gráficas y estadísticas (regresiones lineales y no lineales, tests estadísticos clásicos, análisis de series temporales, clasificaciones, clustering, etc.) y además es muy extensible. El lenguaje S suele ser utilizado para la investigación en metodología estadística, y R proporciona una alternativa de código abierto para esta actividad.

Uno de los puntos fuertes de R es la facilidad con la que se pueden producir gráficas de buen diseño y calidad de imprenta, incluyendo símbolos y fórmulas matemáticas donde sean necesarias. Aunque R pone un gran cuidado en las opciones por defecto para el diseño de las gráficas, el usuario puede tener control total sobre éstas.

R es Software Libre, disponible bajo los términos de la GNU General Public License de la Free Software Foundation, en forma de código fuente. Se puede compilar y ejecutar en una amplia variedad de plataformas UNIX (incluyendo Linux y FreeBSD), MacOS y Windows 9x/NT/2000.

El código fuente de R se puede descargar del sitio web del proyecto R (<http://www.r-project.org>), así como su documentación. Además de la documentación oficial (en inglés) existen otros documentos *contribuidos* entre los que se encuentra la traducción al español de “An Introduction to R” y “Gráficos Estadísticos con R”. Estos y más manuales se encuentran en el CRAN (Comprehensive R Archive Network), concretamente en <http://cran.r-project.org/other-docs.html>

3.2. El entorno R

R es un conjunto integrado de utilidades para manipulación, cálculo y representación de datos. Decimos que R es un *entorno* porque es un sistema diseñado para ser completamente coherente. El

entorno de R proporciona facilidades para manipulación y almacenamiento de datos, operaciones con variables indexadas (como vectores y matrices), análisis y representación de datos, un lenguaje de programación bien desarrollado (con todo lo que cabe esperar de un lenguaje de programación) y una amplia colección integrada y coherente de utilidades para análisis de datos.

Aunque mucha gente utiliza R como un sistema estadístico, sus autores prefieren considerarlo como "un entorno en el que se han implementado muchas técnicas estadísticas, clásicas y modernas". La mayoría de la estadística clásica y muchas de los últimos métodos están disponibles en R, aunque posiblemente tendrás que buscar un rato para encontrarlas.

La forma de trabajar con R es distinta a la de otros programas como SPSS. En R, un análisis estadístico se realiza en una serie de pasos, con unos resultados intermedios que se van almacenando en objetos, para ser observados o analizados posteriormente, produciendo unas salidas mínimas. En SPSS se obtendría de modo inmediato una salida copiosa para cualquier análisis. Esto puede parecer a primera vista una terrible incomodidad, pero si tuviéramos que trabajar en una máquina poco potente rápidamente nos daríamos cuenta de que puede resultar muy ventajosa la sencillez del entorno de R (un entorno de comandos) y la posibilidad de ver en cada momento exactamente lo que se necesita, sin excesos que desperdicien recursos del sistema.

Aún así, la mejor manera de trabajar con R es en un entorno gráfico, con un sistema de ventanas como X-Window, de forma que puedas ver las gráficas en el momento de generarlas.

Veamos cómo se trabaja con R usándolo. En primer lugar conviene crear un directorio y entrar en él antes de comenzar una sesión con R, que en éste almacena siempre en el directorio donde se ejecutan unos ficheros donde almacena los objetos, datos, funciones y comandos ejecutados. Esto puede sernos muy útil en trabajos largos, podemos interrumpir la sesión con R en cualquier momento y recuperarla luego donde mismo la dejamos.

Hemos considerado a lo largo de este curso que el símbolo del sistema Linux/UNIX es \$. Vamos a considerar ahora que el símbolo del prompt R es >. Abre un emulador de terminal dentro del entorno gráfico y ejecuta los siguientes comandos:

```
$ mkdir sesion_R
$ cd sesion_R
$ R
```

```
R : Copyright 2002, The R Development Core Team
Version 1.5.1 (2002-06-17)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
```

```
R is a collaborative project with many contributors.
Type 'contributors()' for more information.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.
```

```
>
```

El prompt de R indica el entorno que está esperando tus órdenes para ejecutarlas. Para salir del entorno de R puedes utilizar la función `q()` o pulsar `C-d`, entonces R te preguntará si deseas guardar el *espacio de trabajo*, que es toda la información sobre la sesión que quieres cerrar:

```
> q()
Save workspace image? [y/n/c]:
```

Para salir guardando la sesión pulsa `y`, para salir sin guardarla pulsa `n` y para para cancelar la salida pulsa `c`. Si guardas la sesión al salir quedará almacenada en el directorio en el que ejecutaste R, y será automáticamente recuperada la próxima vez que ejecutes R dentro del directorio en cuestión.

R dispone de un sistema de ayuda similar a las páginas de manual de Linux/UNIX. Cuando necesites información sobre una función, por ejemplo `solve`, ejecuta la función `help()` pasándole como parámetro el nombre de la función que quieras consultar. También existe una forma más corta: `?función`:

```
> help (solve)
> ?solve
```

Normalmente puedes ver la documentación también en el navegador web. Si ejecutas `help.start()` debería abrirse una ventana de navegador con la documentación en formato HTML por la cual puedes navegar para buscar lo que necesites.

Otra forma muy interesante de buscar ayuda dentro del entorno de R es pasarle una palabra a la función `help.search()`. Por ejemplo, para buscar una función que resuelva sistemas de ecuaciones puedes empezar por buscar la palabra `solve`.

```
> help.search ("solve")

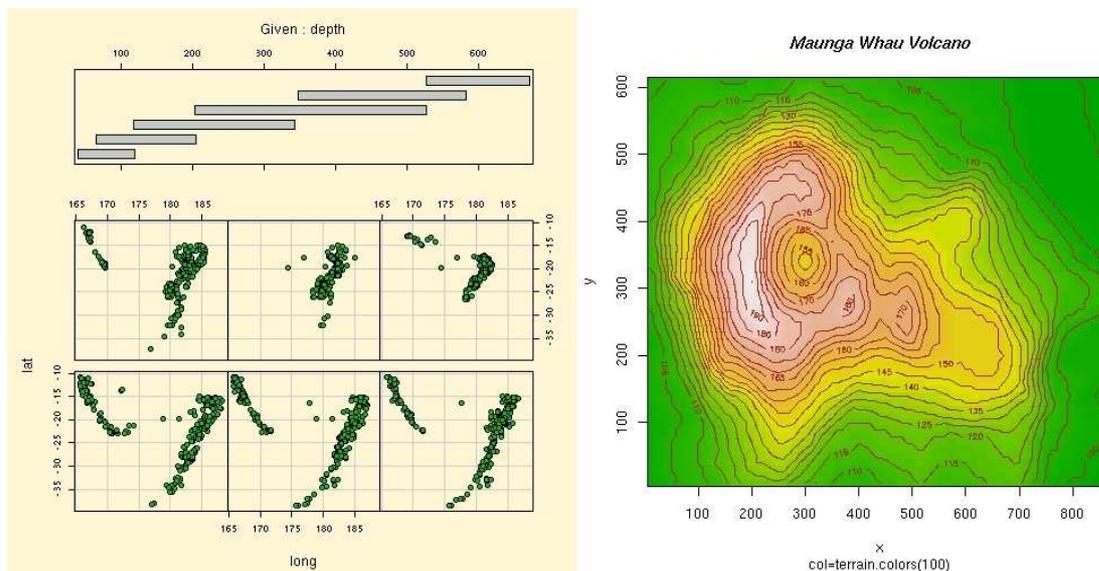
Help files with alias or title matching 'solve',
type 'help(FOO, package = PKG)' to inspect entry 'FOO(PKG) TITLE':

backsolve(base)          Solve an Upper or Lower Triangular System
qr(base)                 The QR Decomposition of a Matrix
solve(base)              Solve a System of Equations
```

De un primer golpe ya sabes que el paquete `base` de R tiene funciones para resolver sistemas triangulares (superiores o inferiores), descomponer matrices en la forma QR y resolver sistemas de ecuaciones (cualesquiera).

R proporciona algunas demostraciones sobre sus capacidades. Para ir abriendo el apetito ejecuta la funciones que muestran las demostraciones con gráficos e imágenes:

```
> demo (graphics)
> demo (images)
```



Demostraciones sobre gráficos e imágenes

3.3. El lenguaje R

R es también lenguaje de programación en toda regla, con el que puedes escribir programas que procesen ficheros de datos y generen análisis y representaciones de los datos.

Al igual que la mayoría de lenguajes basados en UNIX, el lenguaje de R distingue entre mayúsculas y minúsculas. Esto es, `esto` y `Esto` son símbolos distintos y pueden referirse a variables distintas. En R los nombres de variables pueden tener letras y números (como en casi cualquier lenguaje), aunque **no** pueden tener el subrayado (`_`), pero en su lugar sí pueden tener el punto. De hecho se suele utilizar el punto para separar palabras en los nombres de las variables en R, por ejemplo `hoja.de.datos`.

Las órdenes (o comandos) elementales son expresiones o asignaciones. Si ejecutas una expresión como comando ésta se evalúa y se imprime, pero su valor se pierde. En una asignación la expresión se evalúa y su valor se almacena en la variable, pero no se imprime.

Los comandos se separan con el caracter de punto y coma (`;`) o simplemente con una salto de línea. Puedes agrupar una serie de comandos encerrándolos entre llaves, de esta forma puedes definir funciones como veremos más adelante. También puedes poner comentarios, casi donde quieras¹, poniendo una almohadilla (`#`) y todo lo que la siga hasta el final de línea será obviado por el intérprete de R.

Si dejas un comando a medias R se dará cuenta, y en lugar de mostrarte el prompt `>` te mostrará `+` para avisarte de que está esperando por el final del comando.

Otra característica interesante de R (al menos en plataformas Linux/UNIX) es la posibilidad de editar la línea de comandos. Esto incluye el típico historial de comandos, que te permite repetir

¹No puedes meter un comentario dentro de una cadena de caracteres (formaría parte de la cadena), ni en medio de la lista de argumentos en la definición de una función

los comandos sin tener que volver a teclearlos. Tan solo utiliza los cursores arriba y abajo para recuperar los comandos que hayas tecleado. Además este historial de comando queda almacenado en la sesión cuando sales del entorno R, concretamente en un fichero `.Rhistory` en el directorio donde ejecutaste en entorno R.

Pero si necesitas usar una serie un poco larga de comandos seguramente te canses de tener que darle a los cursores. Para estos casos lo que necesitas es un script en el que escribes los comandos a modo de programa. Luego para ejecutar los comandos escritos en él utilizas la función `source()` del entorno pasándole el nombre del fichero donde escribiste los comandos:

```
> source ("script.R")
```

Del mismo modo que puedes tomar la entrada de un fichero también puedes redireccionar la salida hacia otro fichero, con la función `sink()`. Para almacenar la salida en el fichero `salida.txt` ejecuta el comando:

```
> sink ("salida.txt")
```

Para devolver la salida al intérprete utiliza la misma función `sink()` pero sin pasarle ningún parámetro. En el fichero `iodemo.R` tienes un ejemplo de esto.

Fichero `iodemo.R`

```
sink ("salida.txt")
hoja.datos = read.table ("muestra.dat")
attach (hoja.datos)
print (summary (Edad))
sink ()
print (summary (Ingresos))
```

Ejemplo 3.1: Este script lee una tabla que está almacenada en un fichero de texto llano (`muestra.dat`, que usaremos para los ejemplos) y extrae dos variables de ella. Para ambas variables muestra un breve resumen estadístico, pero guarda uno en el fichero `salida.txt` y el otro lo muestra en el entorno R.

Entra en el directorio donde tengas el fichero y ejecuta en el entorno R

```
> source ("iodemo.R")
```

3.4. Vectores

R realiza las operaciones sobre las llamadas *estructuras de datos*, de las cuales la más simple es el vector numérico. Para crear un vector con los 5 primeros números enteros positivos utilizamos la función `c()` que combina los objetos recibidos para formar un vector:

```
> x <- c (1, 2, 3, 4, 5)
> x
[1] 1 2 3 4 5
```

En efecto, el operador de asignación no se parece para nada a un signo de igualdad. De hecho es una flecha, que está indicando que el valor de lo que hay a su derecha debe asignarse a lo que hay a su izquierda. Si le das la vuelta a la flecha funciona al revés:

```
> c (6, 7, 8, 9, 10) -> y
> y
[1] 6 7 8 9 10
```

Cuando ejecutas el nombre de un objeto como comando, R interpreta que deseas ver su contenido y entonces ejecuta la función `print()` sobre el objeto en cuestión. Esto funciona sólo cuando estás dentro del entorno R y tecleas el nombre del objeto. Cuando quieras imprimir el objeto desde un fichero de comandos (como hacemos en `iodemo.R`) has de utilizar la función `print()`.

Dado que la función `c()` combina objetos para formar vectores, también puede combinar varios vectores para formar un nuevo vector que es el resultado de concatenar los anteriores:

```
> z <- c (x, y)
> z
[1] 1 2 3 4 5 6 7 8 9 10
```

Las operaciones usuales entre vectores se definen ‘‘elemento a elemento’’:

```
> x + 2
[1] 3 4 5 6 7
> 2 * x
[1] 2 4 6 8 10
> x + y
[1] 7 9 11 13 15
> x - y
[1] -5 -5 -5 -5 -5
> x * y
[1] 6 14 24 36 50
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444 0.5000000
> y / x
[1] 6.000000 3.500000 2.666667 2.250000 2.000000
> x ^ y
[1] 1 128 6561 262144 9765625
> y ^ x
[1] 6 49 512 6561 100000
```

Pero entonces ¿cómo se multiplican escalarmente dos vectores? Para esto hay una función llamada `crossprod`, que permite multiplicar dos vectores (o matrices). El operador `%*%` es una forma abreviada de este producto. Formalmente `crossprod(x,y)` es equivalente a `t(x)%*% y`, pero más rápido. La función `t()` es la trasposición de matrices:

```
> x
[1] 1 2 3 4
> y
[1] 4 5 6 7
> crossprod(x,y)
      [,1]
[1,]    60
> x %*% y
      [,1]
[1,]    60
```

Además de para guardar datos, los vectores suelen usarse para definir series o secuencias. Para definir una secuencia de números enteros consecutivos basta con poner el primer y el último separados por un caracter de dos puntos:

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
```

Pero para generar secuencias la mejor manera es utilizar la función `seq()`. Si le pasas tres valores numéricos le estarás especificando el primer y el último elemento de la secuencia, y la diferencia que debe haber entre cada dos elementos consecutivos:

```
> y <- seq (-1, 1, .2)
> y
[1] -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0
```

Para replicar un objeto varias veces, por ejemplo para generar una secuencia periódica, utiliza la función `rep()` pasándole el objeto y el número de veces que quieres replicarlo:

```
> z <- rep (seq (-1, 1, 1), 5)
> z
[1] -1  0  1 -1  0  1 -1  0  1 -1  0  1 -1  0  1
```

Además de vectores numéricos, R permite operar con vectores *lógicos*. Los valores válidos para los elementos de los vectores lógicos son los de la *lógica triestada*: `TRUE`, `FALSE` y `NA` (“Not Available”, no disponible). Los operadores lógicos para manejar estos valores son `&` para “and”, `|` para “or” y `!` para “not”.

Una forma de generar un vector de valores lógicos es efectuar una comparación entre un vector numérico y un valor numérico:

```
> z <- rep (seq (-1, 1, 1), 3)
> z
[1] -1  0  1 -1  0  1 -1  0  1
> z > 0
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Al contrario que en la mayoría de lenguajes de programación, en R los vectores se indexan comenzando por 1. Para acceder a un elemento de un vector utiliza la notación usual de los corchetes. También puedes acceder a un subconjunto del vector especificando como índice un vector con las posiciones de los elementos:

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> x[2:5]
[1] 2 3 4 5
> x[8:2]
[1] 8 7 6 5 4 3 2
```

También puedes acceder a un subconjunto de un vector utilizando un vector lógico:

```
> x[x > 4]
[1] 5 6 7 8 9 10
```

Incluso puedes indexar un vector no palabras, poniendo nombres a las posiciones dentro del vector mediante la función `names()`. Luego puedes acceder a los elementos mediante los nombres de las posiciones:

```
fruta = c (5, 10, 1, 20)
> fruta <- c (5, 10, 1, 20)
> names (fruta) <- c ("naranja", "plátano", "manzana", "ñame")
> fruta[c ("manzana", "naranja")]
manzana naranja
      1      5
> fruta
naranja plátano manzana   ñame
      5      10      1      20
```

3.5. Arrays y matrices

Un *array*² es una variable indexada multidimensional. Un vector es un array unidimensional y una matriz es un array bidimensional. R proporciona un buen manejo de arrays, sobretodo para matrices.

La dimensión de un array es un vector cuya longitud es la dimensión del array, y cada uno de sus elementos es la ‘‘longitud’’ de cada dimensión en el array. Por ejemplo, un array con dimensión (3, 5, 100) sería como un cubo de 1500 elementos con 3 celdas de largo, 5 de ancho y 100 de alto.

En la mayoría de lenguajes los elementos de un array se almacenan ‘‘por filas’’, lo que significa que el índice que avanza más rápido es el último. Sin embargo, en R la ordenación se hace al estilo

²a veces traducido por ‘‘arreglo’’

de FORTRAN, ordenando los elementos ‘‘por columnas’’, lo que significa que el índice que avanza más rápido es el primero.

Es importante tener esto en cuenta porque a la hora de crear una matriz hay que proporcionar los elementos agrupados por columnas, no por filas. Por ejemplo:

```
> a <- array (c (1:3,-3:-1), dim = c (3,2))
> a
      [,1] [,2]
[1,]    1  -3
[2,]    2  -2
[3,]    3  -1
```

Para acceder a los elementos de una matriz (o array) ponemos los subíndices entre corchetes y separados por comas. Si omitimos el subíndice de las filas (el primero) obtenemos la columna señalada por el segundo subíndice, y análogamente obtenemos las filas omitiendo el segundo subíndice.

```
> a[1,2]
[1] -3
> a[,2]
[1] -3 -2 -1
> a[1,]
[1] 1 -3
```

De la misma forma que podemos extraer elementos, filas o columnas de una matriz también podemos asignarles valores:

```
> a[,2] <- 3:5
> a[,2]
[1] 3 4 5
> a
      [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    3    5
```

Para multiplicar matrices (o arrays en general) utiliza el operador `%*%` que vimos antes:

```
> a
      [,1] [,2]
[1,]    1    3
[2,]    2    4
[3,]    3    5
> b = array (1:2, 2:1, dim = c (2,2))
> b
      [,1] [,2]
[1,]    1    2
[2,]    3    4
```

```
[1,] 1 1
[2,] 2 2
> a %*% b
      [,1] [,2]
[1,] 7 7
[2,] 10 10
[3,] 13 13
```

Vimos antes que si multiplicamos dos vectores x y y (de igual longitud) utilizando el operador `%*%` el resultado es el producto escalar de los vectores. En realidad la operación `x%*%y` es ambigua, porque podría significar $x'x$ o xx' (considerando x un vector columna). En estos casos de ambigüedad se considera implícitamente que la interpretación deseada es aquella de la que resulte la matriz más paqueña, por lo que se obtiene el producto escalar $x'x$.

Para calcular la matriz xx' puedes utilizar las funciones `cbind()` y `rbind()`, ya que éstas siempre devuelven matrices. Las funciones `cbind` y `rbind` toman una serie de vectores o números y los agrupan por columnas o filas en una matriz.

```
> x
[1] 1 2 3 4
> y
[1] 4 5 6 7
> x %*% y
      [,1]
[1,] 60
> cbind(x,y)
      x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
[4,] 4 7
> rbind(x,y)
      [,1] [,2] [,3] [,4]
x     1   2   3   4
y     4   5   6   7
> cbind(x) %*% rbind(y)
      [,1] [,2] [,3] [,4]
[1,] 4 5 6 7
[2,] 8 10 12 14
[3,] 12 15 18 21
[4,] 16 20 24 28
```

3.6. Factores: clasificación de datos

Un *factor* es un vector que utilizamos para especificar una clasificación discreta (agrupamiento) de los componentes de otros vectores del mismo tamaño. Para entender lo que son los factores nada mejor que un ejemplo claro y sencillo. En el entorno R y ejecuta lo siguiente:

```
> edad <- c (13, 16, 15, 17, 17, 18, 16, 16, 15, 16)
> sexo <- c ("hombre", "hombre", "mujer", "hombre", "mujer", "mujer",
+          "hombre", "mujer", "hombre", "mujer")
> edad
[1] 13 16 15 17 17 18 16 16 15 16
> sexo
[1] "hombre" "hombre" "mujer" "hombre" "mujer" "mujer" "hombre"
[9] "mujer" "hombre" "mujer"
```

Ahora tienes una variable `edad` que contiene las edades de 10 personas, y una variable `sexo` que define el sexo de cada una de las 10 personas anteriores (en el mismo orden) y que toma únicamente los valores "hombre" y "mujer". Vamos a calcular la media y la varianza muestrales de la edad de estas personas clasificándolas según el sexo, i.e. la edad media de los hombres por un lado y la de las mujeres por otro.

Necesitamos separar (agrupar) los elementos del vector `edad` según los valores que toma el vector `sexo`. Para esto utilizamos un factor. El factor en realidad es como el vector original, pero tiene un atributo añadido llamado `levels` (niveles) que son los distintos valores que toman los elementos del vector original. En el caso del vector `sexo` los niveles son "hombre" y "mujer". La función `levels` devuelve la lista de niveles que tenga el factor que le pases como parámetro.

```
> sexo.factor = factor (sexo)
> sexo.factor
[1] hombre hombre mujer hombre mujer mujer hombre mujer hombre
Levels: hombre mujer
> levels (sexo.factor)
[1] "hombre" "mujer"
```

Ahora queremos aplicar unas funciones, en este caso `mean()` y `var()`, a un vector de datos de manera que los datos sean separados según un factor. La función apropiada para esta tarea es `tapply`, y se usa del siguiente modo:

```
> tapply (edad, sexo.factor, mean)
hombre mujer
 15.4  16.4
> tapply (edad, sexo.factor, var)
hombre mujer
  2.3  1.3
```

Así obtenemos que dentro de esta muestra de 10 personas, la edad media de los hombres es 15,4 y la de las mujeres 16,4. Volveremos sobre los factores más adelante, así que asegúrate de entender al menos este uso de los mismos. Para mayores detalles consulta la ayuda de R.

3.7. Listas

En un array todos los elementos deben ser del mismo tipo, i.e. no puede ser que un array contenga a la vez números y palabras. Sin embargo la mayoría de muestras contienen datos de diferentes tipos: números, palabras, valores lógicos, intervalos, etc.

Una *lista* en R es una colección ordenada de objetos de cualquier tipo. Es como un vector, pero con la diferencia de que sus elementos pueden ser de distintos tipos.

Los elementos de una lista están siempre numerados por un subíndice y puedes referirte a ellos como con un vector, pero usando corchetes dobles. Así si *L* es una lista `L[[1]]` es su primer elemento. Pero también, al igual que en los vectores, puedes indexar los elementos de una lista dándoles nombres. En caso de nombrar los elementos de una lista hay una forma más cómoda de referirse a ellos: en lugar de `L[["nombre"]]` puedes usar `L$nombre`.

```
> L = list (nombre = "Pepe", esposa = "Pepa", hijos = 3,
+         edades.hijos = c (34,6,12))
> L
$nombre
[1] "Pepe"

$esposa
[1] "Pepa"

$hijos
[1] 3

$edades.hijos
[1] 34 6 12

> L[["nombre"]]
[1] "Pepe"
> L$nombre
[1] "Pepe"
```

Para concatenar listas recuerda que la función `c()` sirve para ello.

3.8. Hojas de datos

Una "hoja de datos" (en inglés "data frame") es básicamente una lista de vectores, con algunas restricciones. Los vectores de palabras son convertidos en factores.

Piensa en una hoja de datos como su nombre sugiere: una hoja o tabla en la que tienes varios datos sobre varios individuos. Las columnas de la hoja de datos son los vectores que la forman, y cada fila es una lista.

En el fichero `muestra.dat` tienes una hoja de datos preparada para ser leída con la función `read.table()`. La primera fila son los nombres de los vectores columna, y luego cada línea del fichero introduce una lista de valores, un valor para cada vector. Para entender esto con claridad lee la tabla del fichero `muestra.dat` y mantenla en una variable:

```
> hoja.de.datos = read.table ("muestra.dat")
> names (hoja.de.datos)
[1] "Sexo"      "Edad"      "Habitat"   "Ingresos"  "Lectura"   "TV"
```

Esta tabla contiene los datos (ficticios) de 50 jóvenes: su sexo, su edad, su hábitat, sus ingresos familiares (en miles de pesetas mensuales), el número de libros leídos anualmente y sus horas de televisión diarias.

Normalmente para acceder al vector `Edad` de la hoja de datos utilizaríamos la expresión `hoja.de.datos$Edad`, pero hay una forma más cómoda. Consiste en “conectar” la hoja de datos para que puedas referirte a sus vectores directamente por su nombre:

```
> attach (hoja.de.datos)
> summary (Edad)
  Min. 1st Qu.  Median    Mean 3rd Qu.   Max.
12.00  15.00   16.00   15.64  17.00   20.00
```

Ten en cuenta que este atajo es sólo para *leer* los datos de la hoja, no para modificarlos. Si quieres modificar un vector de la hoja de datos tienes que usar la notación normal:

```
> hoja.de.datos$Edad <- Edad + 10
```

De hecho, esto modifica el vector en la hoja de datos, pero no verás los cambios en los atajos hasta que desconectes la hoja de datos y la vuelvas a conectar. Para desconectar una hoja de datos utiliza la función `detach()`.

Cuando quieras almacenar una hoja de datos en un fichero utiliza la función `write.table()`. Su uso básico es darle la hoja de datos y el nombre del fichero, pero admite varias opciones para personalizar la forma en la que escribirá los datos en el fichero. Para ver estos detalles consulta la ayuda sobre la función ejecutando `?write.table`.

3.8.1. Valores perdidos

En ocasiones te encontrarás con hojas de datos en las que alguna celda no tiene el dato. Esto puede suceder porque no haya sido posible averiguar el dato, o porque éste no tenga sentido. En estos casos se utiliza para esa celda el valor `NA`, que significa que el valor “no está disponible” (`NA` es abreviatura de “Not Available”).

En otros casos sucede que tras efectuar una serie de operaciones sobre un conjunto de datos algunos de los resultados no tengan sentido matemático, como por ejemplo una división por cero (recuerda que la precisión de los procesadores es limitada). En caso de hacer una operación así `R` no se quejará en absoluto, sino que devolverá el valor `NaN` que significa que eso “no es un número” (`NaN` es abreviatura de “Not a Number”).

3.9. Funciones

Como todo buen lenguaje de programación, `R` te permite definir funciones para tu uso propio. Para definir una función asignas al objeto (que será tu función) el valor devuelto por la función `function`. Esta función particular recibe los mismos parámetros que recibirá tu función, y a continuación una *agrupación* de comandos, encerrados entre llaves y separados con `;` o saltos de línea. El valor devuelto por tu función será el valor de la última expresión de la agrupación.

Por ejemplo si quieres una función que calcule la curtosis de una muestra:

```

> curtosis <- function (x) {
+ n <- length (x)
+ c <- ( (n * (n - 1) * sum((x - mean(x))^4) ) /
+       ( (n - 1) * (n - 2) * (n - 3) * (var(x))^4 ) ) -
+       ( (3 * (n - 1)^2) / ( (n - 2) * (n - 3) ) )
+ c
+ }
> hoja.de.datos = read.table ("muestra.dat")
> attach (hoja.de.datos)
> curtosis (Edad)
[1] -2.921993
> curtosis (Lectura)
[1] -3.191575
> curtosis (TV)
[1] -3.192770

```

Normalmente las funciones no quedan bien escritas a la primera, por lo que tendrás que corregirlas una y otra vez. O tal vez quieras definir más funciones, modificar las que ya tengas escritas, etc. Todo esto puedes hacerlo más cómodamente si escribes las funciones en un fichero y lo importas con la función `source()` que vimos antes.

Así por ejemplo tienes escrita la función `curtosis` en el fichero `curtosis.R`. Si quieres modificarla y volver a usarla después de modificada no hay problema. Edita el fichero `curtosis.R` para modificar la función y luego vuelve a importarlo con la función `curtosis`. Recuerda que la función `source()` ejecuta los comandos que encuentra en el fichero que le digas, por lo que las funciones que tengas escritas en el fichero serán redefinidas y estarán lista para usar al instante.

Fichero `curtosis.R`

```

curtosis <- function (x) {
n <- length (x)
c <- ( (n * (n - 1) * sum((x - mean(x))^4) ) /
      ( (n - 1) * (n - 2) * (n - 3) * (var(x))^4 ) ) -
      ( (3 * (n - 1)^2) / ( (n - 2) * (n - 3) ) )
c
}

```

Ejemplo 3.2: Este fichero contiene la definición de una función, y cada vez que lo lees con la función `source()` será ejecutado. Por lo tanto las funciones que contiene son redefinidas, lo que te permite modificar las funciones y hacer efectivos los cambios sin tener que salir del entorno.

3.9.1. Control de flujo

R proporciona la sintaxis necesaria para construir funciones que mantengan en control del flujo del programa. Nos referimos a los condicionales y los bucles:

Ejecución condicional

La forma de contruir una sentencia condicional en R es `if (expresion_logica) sentencia_1 else sentencia_2`. Si la `expresion_logica` es cierta (su valor es `TRUE`) se ejecutará la `sentencia_1`, en otro caso se ejecutará la `sentencia_2`. `sentencia_1` y `sentencia_2` pueden ser también agrupaciones de comandos y/o expresiones.

```
> x = 1
> y = 2
> if (x > y) x else y
[1] 2
```

En las expresiones booleanas puedes emplear los operadores `|` (OR) `&` (AND) y `!` (NOT) normales, o si prefieres ahorrar un poco de tiempo los operadores ‘‘cortocircuitados’’ `||` (OR) y `&&`. Estos operadores lógicos tienen la ventaja de que sólo evalúan la segunda expresión cuando es necesario.

Para las comparaciones numéricas se utilizan los mismos comparadores binarios que en el lenguaje C: `<`, `>`, `<=`, `>=`, `!=` y `==`. Ten cuidado de no utilizar el operador `=` para comparaciones.

Existe además una versión *vectorizada* de la construcción condicional. Se trata de la función `ifelse()`, que recibe tres vectores `condicion`, `a`, `b` y devuelve un vector del tamaño del más largo. En este vector el elemento `i`-ésimo es `a[i]` si `condicion[i]` es cierto (su valor es `TRUE`) o bien `b[i]` en caso contrario.

```
> condicion = c (TRUE, FALSE)
> a = c (1, 2)
> b = c (3, 4)
> ifelse (condicion, a, b)
[1] 1 4
```

Ejecución repetitiva

Para la ejecución repetitiva de comandos dispones de la construcción `for (variable in valores) comandos`, donde `variable` es una variable vacía que irá cambiando de valor en cada iteración tomando consecutivamente los valores del vector `valores`. Para cada uno de estos valores se ejecutará la secuencia `comandos`.

```
> for (i in 1:3) print (c (i^2, i^3, i^4, i^5))
[1] 1 1 1 1
[1] 4 8 16 32
[1] 9 27 81 243
```

Para ejecutar una secuencia *mientras* se cumpla una condición (podría no ejecutarse la primera vez) utiliza la construcción `while (condicion) comandos`. Así mientras el valor de la expresión `condicion` sea `TRUE` se seguirá ejecutando la secuencia `comandos`.

```

x = rnorm (1)
> while (x > -2) {
+   print (x)
+   x = rnorm (1)
+ }
[1] -0.2199842
[1] -0.1615499
[1] 2.580791
[1] -0.1202099
[1] -0.794566
[1] -1.413912
[1] 0.4829018
[1] 0.2780835
[1] -0.5475556
[1] 2.974901
[1] -0.1805834

```

La construcción `repeat comandos` se mantiene ejecutando la secuencia `comandos` hasta que se ejecute (dentro de la secuencia) el comando `break`. Esta es la única forma de interrumpir la ejecución de un `repeat`. En iteración del bucle la instrucción `next` hace que la se termine la iteración y se comience con una nueva (no sale del bucle).

3.10. Distribuciones de probabilidad

R tiene un conjunto de funciones para evaluar las funciones de distribución, densidad y probabilidad de las distribuciones que están tabuladas. Para cada distribución la función que lleva su nombre devuelve el valor de su función de distribución, i.e. $P(X \leq x)$ ³.

Las distribuciones para las que R proporciona estas funciones son, con sus nombres en R: beta (`beta`), binomial (`binom`), Cauchy (`cauchy`), χ^2 (`chisq`), exponencial (`exp`), F de Fisher (`f`), gamma (`gamma`), geométrica (`geom`), hipergeométrica (`hyper`), log-normal (`lnorm`), logística (`logis`), binomial negativa (`nbinom`), normal (`norm`), Poisson (`pois`), t de Student (`t`), uniforme (`unif`), Weibull (`weibull`) y Wilcoxon (`wilcox`).

Cada una de estas distribuciones tabuladas proporciona cuatro funciones, cuyos nombres se obtienen precediendo las letras `d`, `p`, `q` o `r` al nombre en R de la distribución, y que son respectivamente las funciones de densidad, distribución, cuantiles y simulación (generadoras aleatorias).

Cada una de estas funciones puede requerir argumentos obligatorios para especificar los parámetros de la distribución o bien permitir argumentos opcionales para modificar los parámetros por defecto. Por ejemplo, `rnorm(10)` genera un vector de 10 números aleatorios que siguen una distribución normal con media 0 y varianza 1, pero estos parámetros pueden modificarse con las opciones oportunas:

```
> rnorm(5)
```

³En algunas tablas encontrarás que los valores son para la probabilidad complementaria a esta, i.e. $P(X > x) = 1 - P(X \leq x)$

```
[1] 0.5849966 2.6217292 -0.9060517 1.1373629 1.4008370
> rnorm(5, mean = 100, sd = 10)
[1] 109.8315 106.1667 94.7198 116.6163 109.8492
```

Por otra parte, para evaluar la función de densidad (o cualquier otra) de la distribución χ_n^2 es necesario saber el número de *grados de libertad* n (en inglés *degrees of freedom*). Este parámetro es obligatorio:

```
> rchisq (5)
Error in rchisq(5) : Argument "df" is missing, with no default
> rchisq (5, 3)
[1] 3.019664 3.335430 6.537741 4.534492 1.843734
> rchisq (5, 30)
[1] 29.76932 41.46605 25.69897 25.35080 33.47488
```

3.11. Estadística descriptiva

R te proporciona todas las funciones que necesites para hacer un estudio estadístico, empezando por una descripción de la muestra a base de medidas de posición centrales, no centrales, de dispersión y de forma.

Como siempre, cada una de estas funciones tiene sus propios parámetros (además de la muestra), así que no dudes en consultar la ayuda de R sobre cada función que necesites. Puedes encontrar opciones realmente interesantes. Las funciones más comunes para estos casos son:

mean(x) Calcula la media aritmética de los valores de un vector numérico x .

median(x) Calcula la mediana de los valores de un vector numérico x .

var(x) Calcula la varianza de los valores x , que puede ser un vector o una matriz.

cov(x,y) Calcula la covarianza de x y y , que pueden ser dos vectores o dos matrices.

cor(x,y) Calcula la correlación entre de x y y , que pueden ser dos vectores o dos matrices.

sd(x) Calcula la desviación estándar de los valores x , que puede ser un vector o una matriz.

range(x) Devuelve un vector con los valores mínimo y máximo encontrados en x , que puede ser un vector o una matriz.

fivenum(x) Devuelve un vector con el mínimo, el máximo y los cuartiles del vector x .

summary(x) Como **fivenum()** pero insertar la media (**mean()**) entre la mediana y el tercer cuartil. Además define los nombres de los resultados.

```
> x = rnorm (100)
> mean (x)
[1] -0.06673138
> median (x)
```

```

[1] -0.01544592
> var (x)
[1] 1.081379
> sd (x)
[1] 1.039894
> range (x)
[1] -3.122787  2.667167
> fivenum (x)
[1] -3.12278656 -0.76220579 -0.01544592  0.58888639  2.66716657
> summary (x)
      Min.  1st Qu.  Median    Mean 3rd Qu.    Max.
-3.12300 -0.75680 -0.01545 -0.06673  0.57860  2.66700

```

3.12. Inferencia estadística

Si no tienes conocimientos de inferencia estadística será mejor que te saltes este apartado. Vamos a ver que también para tareas de inferencia estadística R cuenta con multitud de funciones.

Veamos algunos ejercicios de inferencia estadística, utilizando como fuente de datos la tabla contenida en el fichero `muestra.dat`. No te vamos a dar una referencia amplia ni profunda, pero debería ser suficiente para que aprendas cómo debes buscar las funcionalidades que necesites para tus problemas.

Tests de normalidad

Consideremos la variable `Lectura` y vamos si se puede decir que se distribuye según una distribución normal $N(\mu, \sigma)$. Para ello queremos utilizar los tests de normalidad de Kolmogorov-Smirnov y Shapiro-Wilk. Considera un nivel de significación del 99%.

No sabemos qué funciones proporciona R para estos tests, así que ejecutamos `help.search ("test")` y buscamos los tests deseados. Encontramos entre éstos:

```

ks.test(ctest)           Kolmogorov-Smirnov Tests
shapiro.test(ctest)     Shapiro-Wilk Normality Test

```

Las funciones que necesitas no se encuentran en el paquete `base`, por lo que tendrás que cargar el paquete en el que se encuentran. Para ello utiliza la función `library()` como verás más adelante.

Para el test de Kolmogorov-Smirnov necesitas estimadores los parámetros de la distribución con la que quieras comparar la muestra. En esta caso has de estimar la media y la desviación típica, y una buena forma es utilizar los estimadores insesgados por analogía que son la media muestral y la cuasi-desviación típica muestral.

Utilizaremos en el ejemplo la desviación típica, para dejarte como ejercicio sustituirla por la cuasi-desviación típica muestral definiendo tu propia función que la calcule.

```
> hoja.de.datos = read.table ("muestra.dat")
> attach (hoja.de.datos)
> library (ctest)
> shapiro.test (Lectura)
```

Shapiro-Wilk normality test

```
data: Lectura
W = 0.9577, p-value = 0.0714
```

```
> ks.test (Lectura, "pnorm", mean = mean (Lectura), sd = sd (Lectura))
```

One-sample Kolmogorov-Smirnov test

```
data: Lectura
D = 0.0907, p-value = 0.8053
alternative hypothesis: two.sided
```

Este ejemplo, dado que los p-valores de los tests son $0,0714, 0,8053 > 0,01 = \alpha = 1 - 0,99$ se puede decir que el número de libros leídos anualmente se distribuye según una normal.

Contraste de hipótesis

Una vez comprobada la normalidad de una variable hay una serie de tests interesantes que puedes utilizar. Aprovechando esto vamos a plantear el siguiente un contraste de hipótesis para la media de la variable `Lectura`. La hipótesis nula es que la media es 15. Considera un nivel de confianza del 99 % ($1 - \alpha = 0,99$)

$$\left. \begin{array}{l} H_0: \mu = 15 \\ H_1: \mu \neq 15 \end{array} \right\}$$

Para este contraste tiramos del test T de Student para una muestra, que buscando como antes encontrarás que lo proporciona la función `t.test()`. Con una mirada a la ayuda de esta función verás que debes especificar la media con la que quieres contrastar (`mu = 15`) y el nivel de confianza (`conf.level = 0.99`).

```
> t.test (Lectura, mu = 15, conf.level = 0.99)
```

One Sample t-test

```
data: Lectura
t = -1.8956, df = 49, p-value = 0.06392
alternative hypothesis: true mean is not equal to 15
99 percent confidence interval:
 10.89657 15.70343
```

```
sample estimates:
mean of x
  13.3
```

Dado el p-valor del contraste, $0,06392 > 0,01 = \alpha$, no se rechaza la hipótesis nula; se puede considerar que el número medio de libros leídos anualmente es 15, con un nivel de confianza del 99 %.

Independencia de variables

Considera ahora la dos variables **Ingresos** y **Habitat**. Ambas son vectores de caracteres y por lo tanto al estar en una hoja de datos han sido convertidas a factores. Vamos a contrastar si estas variables son independientes, utilizando la tabla de contingencia y el test χ^2 . Considera un nivel de confianza del 95 %.

De nuevo para averiguar qué funciones nos proporciona R buscamos en la ayuda: ejecutando `help.search ("contingency")` encontramos

```
fable(base)          Flat Contingency Tables
```

Con la misma forma de búsqueda encontramos la función para el test χ^2 con `help.search ("test")`

```
chisq.test(ctest)    Pearson's Chi-squared Test for Count Data
```

Aplicamos el test χ^2 a la tabla de contingencia de las variables **Ingresos** y **Habitat**:

```
> t = ftable(Ingresos, Habitat)
> t
      Habitat Rural Urbano
Ingresos
<100      2      0
100-200   13      9
200-300    5     10
300-400    1      3
400-500    0      4
>500      0      2
> chisq.test(t)
```

Pearson's Chi-squared test

```
data:  t
X-squared = 10.6105, df = 5, p-value = 0.05967
> qchisq (0.95, 5)
[1] 11.07050
```

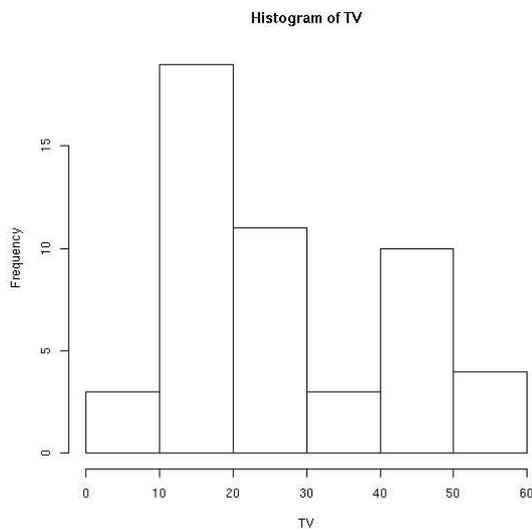
La región crítica para este test es $W = \chi^2 > \chi_{5,0,05}^2$, y dado que el estadístico χ^2 es $10,6105 < 11,07050 = \chi_{5,0,05}^2$ podemos considerar que las variables son independientes.

3.13. Gráficas

La función más frecuentemente usada para crear gráficas es `plot()`. Esta función representa uno o dos vectores numéricos como una nube de puntos, un factor mediante un diagrama de barras (`barplot`), un factor y un vector numérico con un diagrama de cajas (`boxplot`), y un par de factores como un histograma con cada barras dividida según el factor.

Para verlo más claro ejecuta la función `plot()` con las variables de la tabla del fichero `muestra.dat`. Prueba las siguientes combinaciones: `plot (Lectura)`, `plot (Lectura, TV)`, `plot (Ingresos)`, `plot (Ingresos, Habitat)`

Para representar vectores numéricos agrupando sus valores en intervalos utilizamos un histograma. La función `hist()` hace exactamente eso.



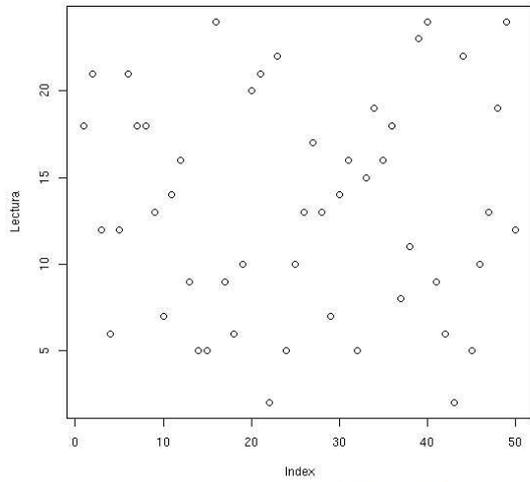
Histograma obtenido con `hist (TV)`

Otras gráficas interesantes son las de comparación de distribuciones, basadas en los cuantiles. La función `qqnorm()` toma un vector numérico y lo representa comparando sus cuantiles con los cuantiles teóricos (los que se esperarían en una muestra que siguiera una distribución normal). La función `qqline()` añade a la gráfica la recta que pasa por los cuantiles de los datos y la distribución.

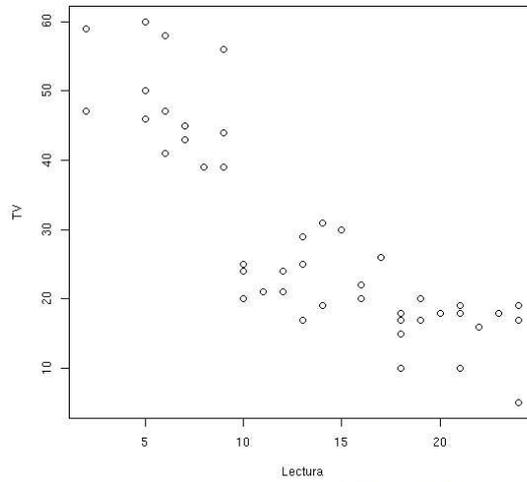
Para representar matrices puedes optar por ver sus valores como imágenes. La función `image()` lo hace a modo de una rejilla de rectángulos de diferentes colores para representar el valor de las celdas. Para verlo en forma de contorno usa `contour()`, y para verlo en perspectiva utiliza `persp()`.

Los gráficos de sectores te dan una impresión clara a primera vista de los porcentajes que tienen cada “clase” en una clasificación de datos. Por ejemplo para ver los porcentajes de personas según las edades representamos en un gráfico de sectores el número de individuos que tiene cada edad. Una forma de hacerlo (seguramente no es la mejor) sería:

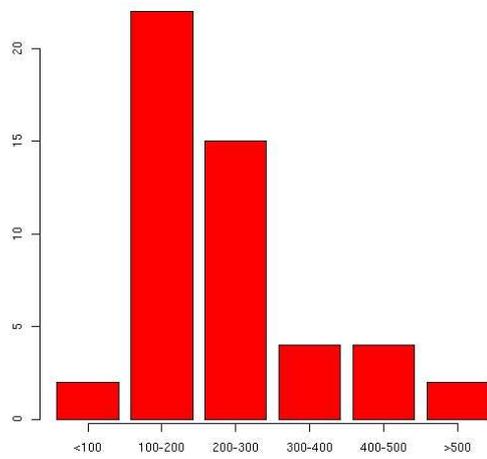
```
> pie (tapply (Edad, fedad, length))
```



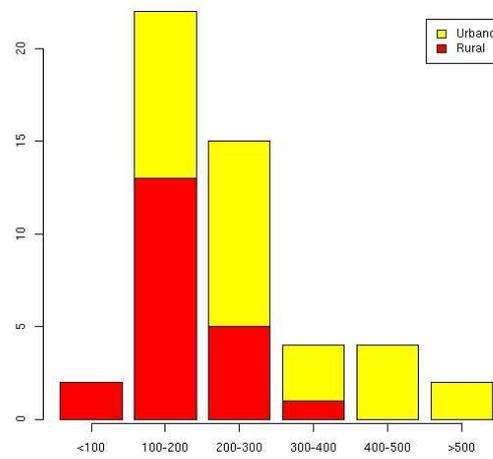
(a) plot (Lectura)



(b) plot (Lectura, TV)

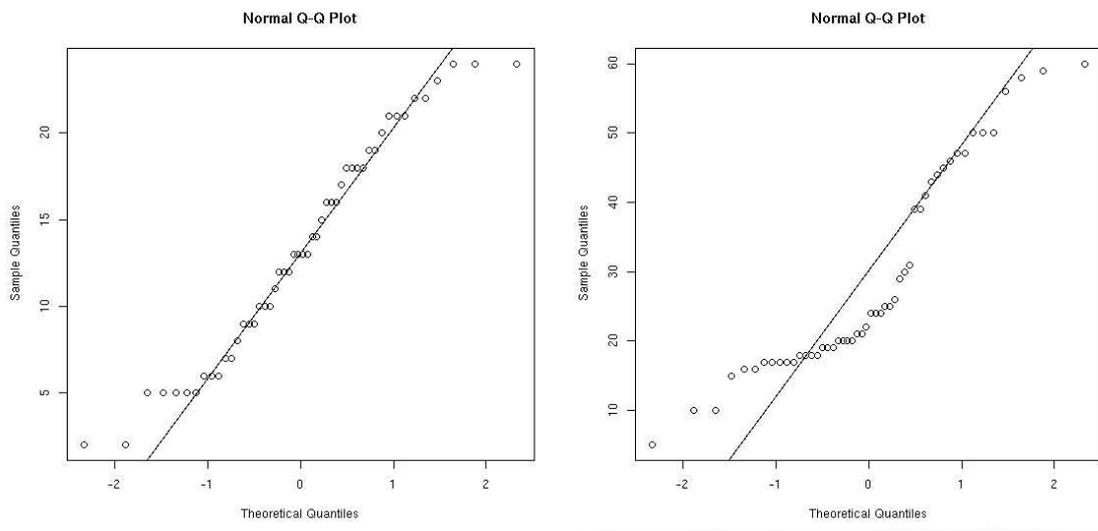


(c) plot (Ingresos)



(d) plot (Ingresos, Habitat)

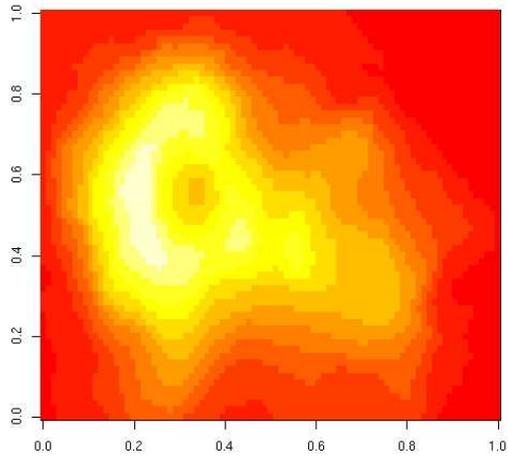
Figura 3.1: Las distintas formas de la función plot()



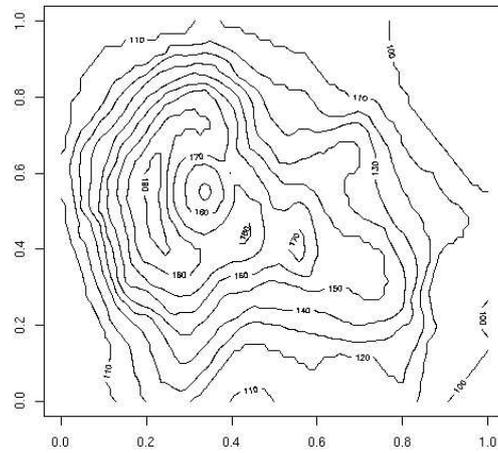
(a) Gráfico Q-Q para Lectura

(b) Gráfico Q-Q para TV

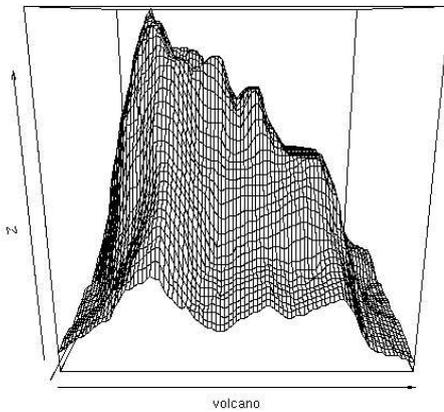
Figura 3.2: Gráficos Q-Q para dos variables, una que se comporta como una normal y otra que no



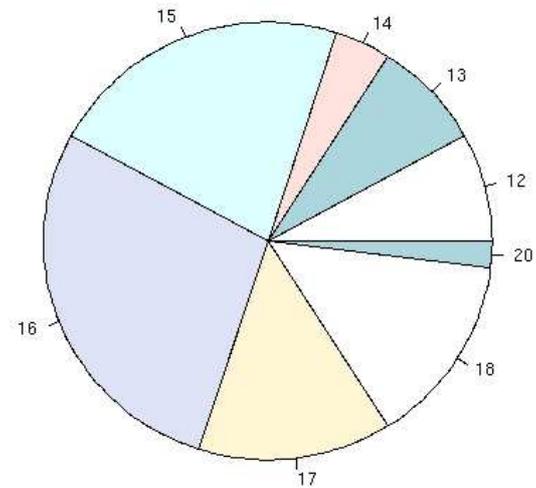
(a) `image(volcano)`



(b) `contour(volcano)`



(c) `persp(volcano)`



(d) Gráfica de sectores para la Edad

Figura 3.3: Gráficas para representar matrices. El objeto `volcano` lo obtenemos al ejecutar `data(volcano)`

Capítulo 4

Yacas

En este tema aprenderás a utilizar el programa Yacas para relizar cálculos simbólicos. Este tipo de cálculos resulta de gran utilidad puesto que funciona de la misma forma que harías tú mismo a mano, operando con expresiones matemáticas sin calcular el valor numérico de cada expresión.

4.1. ¿Qué es Yacas?

Yacas es un Sistema de Álgebra Computacional (en inglés CAS, de Computer Algebra System) de propósito general fácil de usar. Un Sistema de Álgebra Computacional (SAC) es un programa que permite manipulaciones simbólicas sobre expresiones matemáticas, reduciendo el tiempo necesario para realizar cálculos embarazosos pero triviales. Esto se hace no mediante números, sino mediante símbolos, por lo que el resultado de una operación con expresiones matemáticas es una nueva expresión matemática.

Yacas está construido sobre su propio lenguaje de programación diseñado para este propósito, en el que se pueden implementar fácilmente nuevos algoritmos. Además, incorpora una documentación extensa sobre las funcionalidades que implementa y los métodos usados para implementarlas.

Entre las características que Yacas implementa encontramos precisión arbitraria, números racionales, vectores, números complejos, cálculos con matrices (incluyendo inversas, determinantes y resolución de sistemas lineales), derivación, series de Taylor, resolución numérica (método de Newton), y un montón más de algoritmos no matemáticos. Tiene también soporte básico para polinomios en una variable, integración de funciones y cálculo tensorial.

En la web del proyecto Yacas (<http://yacas.sourceforge.net>) encontrarás el código fuente del programa y toda su documentación: desde un tutorial básico hasta una guía de programación en Yacas. Parte del texto de este tema son traducciones de trozos de la documentación oficial de Yacas que se distribuye junto con el programa bajo licencia GPL.

4.2. Empezando con Yacas

Yacas tiene un intérprete de comandos que nos permite ejecutar funciones para probar lo que queremos programar, para luego escribir un script (un fichero de comandos a modo de guión). Para

entrar al intérprete de Yacas basta con ejecutar el comando `yacas` en una consola o terminal. Para salir del intérprete puedes pulsar `Control-C` o bien ejecutar la función `Exit()`; Pulsar `Control-C` también es útil para detener inmediatamente la ejecución del intérprete de Yacas (o de un script escrito para Yacas).

```
$ yacas
[editvi.ys] [unix.ys]
True;
Numeric mode: "Gmp"
To exit Yacas, enter Exit(); or quit or Ctrl-c. Type ?? for help.
Or type ?function for help on a function.
Type 'restart' to restart Yacas.
To see example commands, keep typing Example();
In>
```

También existe una interfaz gráfica llamada Proteus. Si la tienes instalada deberías poder iniciarla ejecutando el comando `proteusworksheet`. Esta utilidad te proporciona un entorno más cómodo e integrado en el que puedes probar código en el intérprete e ir escribiendo un script en el editor que incluye.

La última línea (`In>`) es el prompt de entrada de Yacas, que nos indica que el intérprete está esperando nuestras órdenes. Todas las órdenes de Yacas deben terminar con un punto y coma (`;`), aunque el intérprete suele añadirlo si no lo ponemos. Sin embargo a la hora de programar en Yacas veremos que el punto y coma hay que escribirlo siempre, de modo que es mejor acostumbrarse a ponerlo para evitar posteriores dolores de cabeza.

Para ir abriendo el apetito sigue la sugerencia de Yacas: ejecuta la función `Example()`; varias veces. Por supuesto no tienes que teclear el comando todas las veces, ya que Yacas cuenta con edición de línea. Esto incluye que todos los comandos que ejecutes en Yacas se almacenarán en un historial, y para repetirlos puedes recuperarlos pulsando la tecla de cursor hacia arriba. Este historial queda guardado en el fichero `~/yacas_history`. Veamos la salida que produce Yacas tras varias ejecuciones de `Example()`;

```
In> Example();
Current example : 40!;

Simple factorial of a number.

Out> 815915283247897734345611269596115894272000000000;
In> Example();
Current example : D(x)Sin(x);

Taking the derivative of a function (the derivative of Sin(x) with
respect to x in this case).

Out> Cos(x);
In> Example();
Current example : Taylor(x,0,5)Sin(x);
```

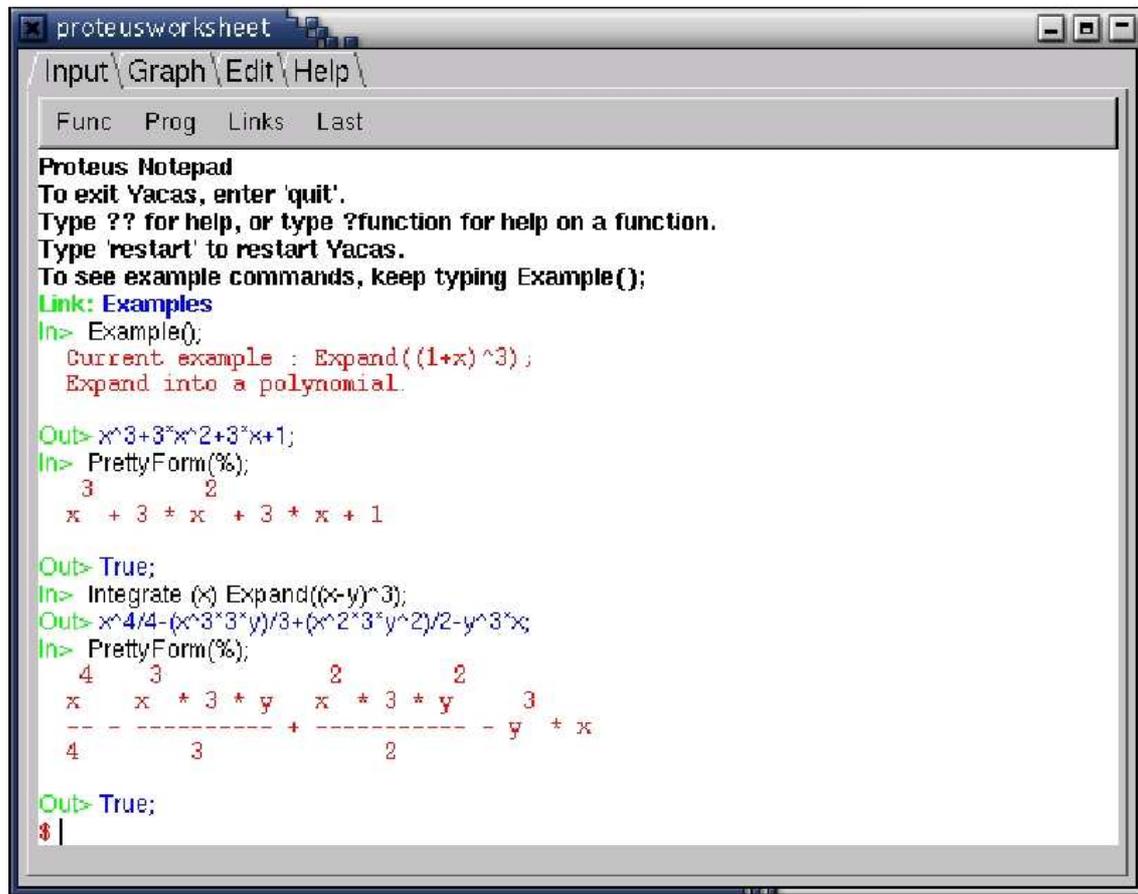


Figura 4.1: Proteus, interfaz grfica para Yacas

Expanding a function into a Taylor series.

```

Out> x-x^3/6+x^5/120;
In> Example();
Current example : Integrate(x,a,b)Sin(x);
    
```

Integrate a function.

```

Out> Cos(a)-Cos(b);
In> Example();
Current example : Solve(a+x*y==z,x);
    
```

Solve a function for a variable.

```
Out> (z-a)/y;
In> Example();
Current example : Limit(x,0)Sin(x)/x;
```

Take a limit.

```
Out> 1;
```

En cualquier momento puedes acceder a la documentación de una función ejecutando la función precedida por un signo de interrogación, p.ej. `?IsFreeOf`. Un detalle importante de Yacas es que, como muchos lenguajes del mundo de `Un*x`, es sensible a las mayúsculas; esto es, no es lo mismo `esto` que `Esto` ni que `eSTO`.

Estos ejemplos ilustran lo fácil que es obtener buenos resultados en cálculos simbólicos con Yacas. Sin embargo es probable que el resultado del siguiente ejemplo no te resulte fácil de interpretar:

```
In> Integrate(x) Expand((x-y)^3);
Out> x^4/4-(x^3*3*y)/3+(x^2*3*y^2)/2-y^3*x;
```

la forma en que Yacas recibe nuestra orden de integrar $\int (x - y)^3 dx$ no es nada críptica, pero su respuesta no es precisamente fácil de leer. Esta forma de mostrar las expresiones matemáticas es la normal en Yacas, pero no la única. Podemos pedir a Yacas que muestre las expresiones de forma más “bonita” con la función `PrettyForm()`.

```
In> PrettyForm(%)
```

$$\frac{x^4}{4} - \frac{x^3 * 3 * y}{3} + \frac{x^2 * 3 * y^2}{2} - y^3 * x$$

```
Out> True;
```

El signo `%` es una referencia que apunta siempre al último valor devuelto por el intérprete, es decir lo que aparezca a la derecha del último `Out>`. Esta referencia sólo está disponible cuando usamos Yacas como intérprete interactivo, no se puede utilizar desde un script.

Otras formas de mostrar una expresión son las utilizadas en el lenguaje de programación `C` y en el lenguaje tipográfico `TeX`. Para mostrar una expresión de estas formas están las funciones `CForm()` y `TeXForm()` respectivamente. Fíjate que estas funciones no devuelven la expresión que muestran, por lo que no podemos utilizar la referencia `%` dos veces seguidas para mostrar una expresión (la segunda mostraría la expresión lógica `True`). Por eso escribimos `Integrate(x) Expand((x-y)^3)` en cada línea:

```
In> CForm (Integrate(x) Expand((x-y)^3))
Out> "pow(x, 4.) / 4. - ( pow(x, 3.) * 3. * y) / 3. + ( pow(x, 2.) *
```

```

3. * pow(y, 2.) / 2. - pow(y, 3.) * x";
In> TeXForm (Integrate(x) Expand((x-y)^3))
Out> "$\frac{x ^{4}}{4} - \frac{x ^{3} 3 y}{3} + \frac{x ^{2} 3 y ^{
2}}{2} - y ^{3} x$";

```

La función `CForm()` devuelve la expresión escrita en lenguaje C, de forma que podemos incluirla en el código de un programa en C. La función `TeXForm()` devuelve la expresión escrita en el lenguaje tipográfico `TEX`, de forma que podemos incluirla en un documento escrito en `TEX` o `LATEX`. Por ejemplo, este libro está escrito en `LATEX`, lo que me permite incluir la salida de `TeXForm()` en este mismo párrafo con sólo copiar y pegar. De esta forma puedo escribir:

$$\int (x - y)^3 dx = \frac{x^4}{4} - \frac{x^3 3y}{3} + \frac{x^2 3y^2}{2} - y^3 x$$

En ocasiones nos interesa conocer el valor numérico de una expresión determinada, y posiblemente sólo nos interese un número determinado de decimales. Yacas es un lenguaje de precisión arbitraria, lo que significa que puedes pedirle toda la precisión que quieras, pero ten cuidado no le pidas demasiada precisión si tu máquina no es realmente potente.

La función `Precision()` establece el número de cifras decimales con las que se aproximarán los valores. Estas aproximaciones se realizan con la función `N()`, que además permite saltarse la precisión establecida por `Precision()` pasándosela como segundo parámetro. En el siguiente ejemplo se ilustra esto:

```

In> alpha := Sqrt (Pi ())
Out> Sqrt(3.1415926535897932384626433832795028841971694);
In> Precision (2);
Out> True;
In> N (alpha)
Out> 1.77;
In> Precision (20);
Out> True;
In> N (alpha)
Out> 1.77245385090551602729;
In> N (alpha, 50)
Out> 1.77245385090551602729816748334114518279754945629866;

```

Normalmente no hay problema por pedirle a Yacas unos cientos de dígitos de precisión, pero siempre que vayas a trabajar con números o precisiones enormes recuerda que el tiempo necesario para los cálculos aumentará rápidamente.

4.3. Variables y funciones

En Yacas se entiende por variable un objeto al que se puede asignar un valor. Las variables pueden llamarse como quieras siempre que el nombre empiece por una letra y continúe con letras y números. Para asignar un valor a una variable se utiliza el operador de asignación `:=` en lugar

de = (este último se utiliza como comparador). En cualquier momento puedes ver el valor de una variable tecleando su nombre como si fuera un comando de Yacas, y esto también es válido en un script. Asignaciones válidas son:

```
In> n := 10;
Out> 10;
In> m := n!;
Out> 3628800;
In> n
Out> 10;
In> m
Out> 3628800;
```

Si en algún momento quieres que una variable pierda su valor no tienes más que “limpiarla” con la función `Clear()`:

```
In> m
Out> 3628800;
In> Clear (m);
Out> True;
In> m
Out> m;
```

Las funciones son como las variables, con el añadido de que pueden depender de una o más variables. Además con Yacas podemos sobrecargar una función, esto es, definir dos funciones con el mismo nombre pero distinto número de variables de forma que se evalúe una u otra según el número de parámetros que reciba.

Como ejemplo definimos la función `Area` de dos formas: si recibe un parámetro devuelve el área del círculo con el radio igual a ese parámetro, pero si recibe dos parámetros devuelve el área de la elipse que tenga esos dos parámetros como radios.

```
In> Area (r) := Pi () * r^2;
Out> True;
In> Area (a, b) := Pi () * a * b;
Out> True;
In> Area (3);
Out> 28.2743338823;
In> Area (3, 5);
Out> 47.1238898035;
```

Al definir una función, Yacas no evalúa la parte de la derecha sino que se asigna como expresión simbólica. Esto puede no interesarte en casos como el que ilustra el siguiente ejemplo:

```
In> f(x) := x^5;
Out> True;
In> g(x) := D(x) f(x);
```

```

Out> True;
In> g(x)
Out> 5*x^4;
In> g(2)
Out> 5*x^4;

```

Sería deseable que $g(2)$ devolviera el valor de la función $g(x)$ cuando x vale 2, pero no lo hace porque $g(x)$ almacena la expresión $5*x^4$ sin ninguna referencia a la necesidad de evaluarla en un valor determinado. Sin embargo si indicamos a Yacas que evalúe la expresión antes de asignarla a la función (mediante `Eval()`) podremos pedirle que evalúe la función $g(x)$ en valores determinados. Esto que parece un follón es en realidad simple, el siguiente ejemplo muestra el código que funciona como es de esperar:

```

In> g(x) := Eval (D(x) f(x));
Out> True;
In> g(x)
Out> 5*x^4;
In> g(2)
Out> 80;

```

4.4. Listas

Uno de los elementos más importantes del lenguaje de Yacas son las listas, que como era de esperar son grupos de objetos ordenados. Yacas representa las listas con sus elementos entre llaves y separados por comas. Los vectores son listas, y las matrices son listas de listas. De hecho, cualquier expresión matemática en Yacas puede ser transformada en una lista. Para acceder a los elementos de una lista puedes usar la notación habitual en los corchetes, y no sólo con números sino también con secuencias de ellos. Los elementos de la lista pueden ser cualquier tipo de objetos.

```

In> lista := {a, b, c, d, e, f};
Out> {a,b,c,d,e,f};
In> lista[2];
Out> b;
In> lista[2 .. 4];
Out> {b,c,d};

```

Fíjate que los espacios a ambos lados del operador `..` son necesarios para distinguir estos puntos de los que podrían formar parte de un número.

También puedes indexar una lista con palabras, no sólo con números. Esto te permite tener pequeñas bases de datos en forma de listas asociativas con parejas clave – valor.

```

In> boy := {};
Out> {};
In> boy["nombre"] := "Eric";
Out> True;

```

```
In> boy["apellido"] := "Cartman";
Out> True;
In> boy["edad"] := 7.34;
Out> True;
In> boy["educado"] := False;
Out> True;
In> boy
Out> {"educado",False},{"edad",7.34},{"apellido","Cartman"},
{"nombre","Eric"};
```

4.5. Álgebra Lineal

Los vectores de dimensión fija se representan mediante listas. La lista $\{1,2,3\}$ es el vector $(1,2,3)$. Las matrices se representan como vectores de vectores. Dado que los vectores son realmente listas, sus elementos pueden asignarse igual que los de las listas:

```
In> l := ZeroVector (3);
Out> {0,0,0};
In> l;
Out> {0,0,0};
In> l[2] := 2;
Out> True;
In> l;
Out> {0,2,0};
```

Yacas puede multiplicar vectores, matrices y números del modo usual en álgebra lineal:

```
In> v := {1, 0, 0, 0}
Out> {1,0,0,0};
In> E4 := { {0, u1, 0, 0}, \
           {d0, 0, u2, 0}, \
           {0, d1, 0, 0}, \
           {0, 0, d2, 0} }
Out> {{0,u1,0,0},{d0,0,u2,0},{0,d1,0,0},{0,0,d2,0}};
In> PrettyForm (%)
```

```
/
| ( 0 ) ( u1 ) ( 0 ) ( 0 ) |
|
| ( d0 ) ( 0 ) ( u2 ) ( 0 ) |
|
| ( 0 ) ( d1 ) ( 0 ) ( 0 ) |
|
| ( 0 ) ( 0 ) ( d2 ) ( 0 ) |
\
/
```

```

Out> True;
In> CharacteristicEquation (E4, x)
Out> x^4-x*u2*d1*x-u1*d0*x^2;
In> Expand (% , x)
Out> x^4-(u2*d1+u1*d0)*x^2;
In> PrettyForm (%)

      4                2
x  - ( u2 * d1 + u1 * d0 ) * x

Out> True;
In> v + E4 * v + E4 * E4 * v + E4 * E4 * E4 * v
Out> {u1*d0+1,d0+(d0*u1+u2*d1)*d0,d1*d0,d2*d1*d0};
In> PrettyForm (%)

/                               \
| u1 * d0 + 1                   |
|                               |
| d0 + ( d0 * u1 + u2 * d1 ) * d0 |
|                               |
| d1 * d0                       |
|                               |
| d2 * d1 * d0                   |
\                               /

Out> True;

```

La librería estándar de Yacas incluye también cálculo del determinante y la inversa de una matriz, autovectores y autovalores (en casos simples) y resolución de sistemas de ecuaciones lineales del tipo $Ax = b$ donde A es una matriz y x y b son vectores.

4.6. Control de flujo

El lenguaje de Yacas incluye algunas construcciones y funciones para el control de flujo. Para ello necesitas saber también que Yacas te permite agrupar bloques de instrucciones de forma que aparezcan como una sola instrucción. Para ello simplemente encierra las instrucciones del bloque entre corchetes (`[y]`), o bien utiliza la función `Prog()` pasándole las instrucciones separadas por `;`.

Para los bucles disponemos de las funciones `ForEach()` y `While()`. La función `ForEach(x, list) body` ejecuta el bloque de instrucciones `body` para cada elemento de la lista `list` asignando el valor de cada elemento a la variable `x` en cada interacción. La función `While(predicate) body` repite el bloque `body` hasta que la condición `predicate` devuelva `False`.

Para los condicionales está la función `If(predicate, body1, body2)`, en la que se ejecuta el bloque `body1` si `predicate` devuelve `True` o ejecuta el bloque `body2` si `predicate` devuelve `False`,

y en ambos casos devuelve el valor devuelto por el bloque ejecutado. El segundo bloque es opcional. Si llamas a `If (predicate, body1)` se ejecutará el bloque `body1` si `predicate` devuelve `True` y devolverá el valor que devuelva `body1`, o bien se devolverá `False` si así lo hace `predicate`.

Como ejemplo de control de flujo construimos una lista con los números enteros pares de 2 a 20 y calculamos el producto de los que no sean divisibles por 3. Luego definimos el factorial de un número de forma recursiva.

```
In> L := {};
Out> {};
In> i := 2;
Out> 2;
In> While (i <= 20) [ \
In>   L := Append (L, i); \
In>   i := i + 2; \
In> ];
Out> True;
In> L;
Out> {2,4,6,8,10,12,14,16,18,20};
In> answer := 1;
Out> 1;
In> ForEach (i, L) [ \
In>   If ( Mod (i, 3) != 0, \
In>     answer := answer * i \
In>   ); \
In> ];
Out> True;
In> answer;
Out> 2867200;
In> Factorial (x) := [ \
In>   If ( IsInteger (x) And x >= 0, \
In>     If (x = 0, 1, \
In>       x * Eval (Factorial (x-1)) ), \
In>   False ); \
In> ];
In> Factorial (5)
Out> 120;
In> Factorial (25)
Out> 15511210043330985984000000;
```

Una barra invertida `\` al final de una línea indica al intérprete de Yacas que la línea no está terminada, sino que continúa después del salto de carro. En este ejemplo hemos utilizado esto descaradamente. Esto se puede utilizar en el intérprete para hacer que nuestros comandos sean más cómodos de leer, pero no es necesario al escribir un script.

4.7. Gráficas

Yacas incorpora la posibilidad de representar gráficas bidimensionales utilizando GNUplot, si bien esta capacidad depende de que tengas instalado también el programa GNUplot. Para saber si tu versión de Yacas tiene soporte para GNUplot fíjate en la primera línea que imprime el intérprete cuando arranca, si aparece `[gnuplot.y]` es que tu versión de Yacas soporta para gráficas con GNUplot¹.

```
$ yacas
[editvi.y] [gnuplot.y] [unix.y]
```

La función para representar gráficas es `Plot2D()` y recibe dos parámetros: la expresión (o lista de expresiones) para representar y el intervalo de la variable en la forma `valor_minimo : valor_maximo`.

En anteriores versiones de Yacas, la función para representar gráficas era `GnuPlot()` y recibía cuatro parámetros: el valor mínimo de la variable, el valor máximo de la variable, el número de puntos utilizados para la representación y la función para representar.

En los siguientes ejemplos calculamos el el polinomio de Taylor de orden 5 para la función $\sin(x)$ en el origen y la representamos (en rojo) junto con la función $\sin(x)$ (en verde).

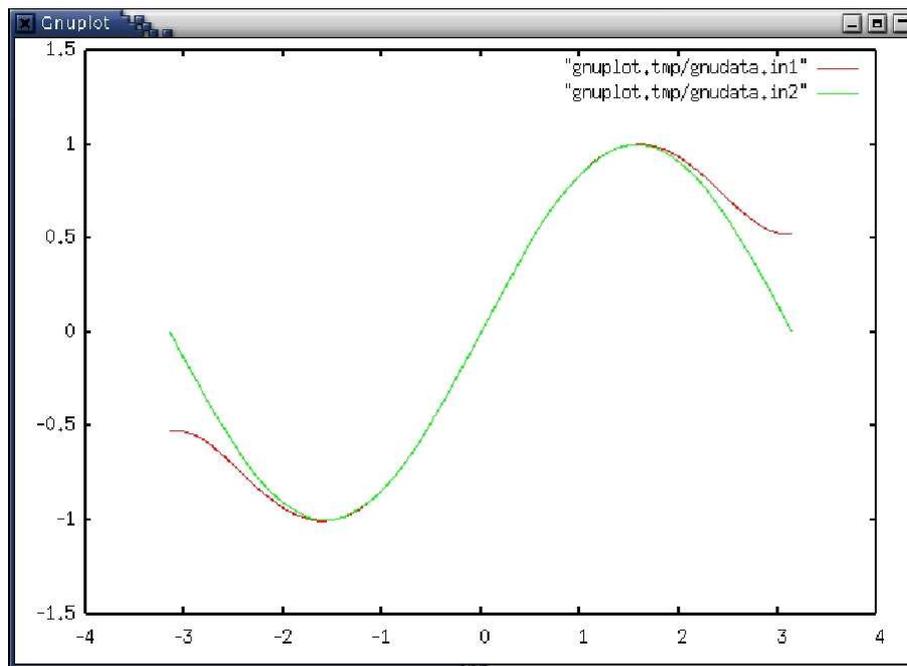


Figura 4.2: Representación gráfica con GNUplot desde Yacas

¹Esto era cierto hasta la versión 1.0.50 de Yacas, pero en el momento de escribir esta nota la versión 1.0.51 de mi máquina no aparece `[gnuplot.y]` pero puede hacer las gráficas con la función `Plot2D`

```
In> f(x) := Eval (Taylor (x, 0, 5) Sin(x) )
Out> True;
In> Plot2D ({f(x), Sin (x)}, -Pi:Pi)
Out> True;
In> GnuPlot (-Pi(), Pi(), 50, {f(x), Sin(x)})
GnuPlot: created file gnuplot.tmp/gnudata.in1
GnuPlot: created file gnuplot.tmp/gnudata.in2
Out> True;
```

4.8. Programando con Yacas

Si bien el intérprete de Yacas proporciona una interfaz cómoda para ejecutar cálculos, cuando éstos se complican resulta más práctico escribir un “script” y ejecutarlo con Yacas. Para hacer un programa con Yacas escribe en un fichero la secuencia de comandos que componen el programa, incluyendo definiciones de funciones. Los ficheros de script de Yacas suelen tener la extensión `.ys` (no es obligatorio).

Para ejecutar el script simplemente ejecuta Yacas dándole el nombre del fichero como parámetro. La opción `-c` del comando `yacas` hace que Yacas no muestre los prompts `In>` y `Out>` (útil para sesiones no interactivas, como el caso de ejecutar un script). El siguiente ejemplo muestra como ejecutar un script en Yacas, en este caso el script es el fichero de ejemplo `lagrange.ys`

```
$ yacas -c lagrange.ys
```

Fichero `lagrange.ys`

```
f(x) := 1.0 / (1.0 + x^2);
x1 := {-5, -2, 0, 3, 5};
y1 := {};
n := Length (x1);
For ( i := 1 , i <= n , i++ ) [
  y1 := Append (y1, Eval (f (x1[i])));
];
pol (x) := Eval (LagrangeInterpolant (x1, y1, x));
Plot2D ({f(x), pol(x)}, x1[1]:x1[n]);
```

Ejemplo 4.1: Script en Yacas que muestra un interpolante de Lagrange para una nube de cinco puntos tomados de la función $f(x) = \frac{1}{1+x^2}$. En este ejemplo el interpolante de Lagrange no es el adecuado, pero nuestra intención no es interpolar $f(x)$

Otra forma de ejecutar un script es desde el intérprete, mediante las funciones `Load()` y `Use()`. La función `Load("script.ys")` lee el fichero `script.ys` y evalúa todas las expresiones que encuentre en él. Siempre devuelve `True`. `Use()` hace lo mismo que `Load()` con la salvedad de que sólo lee el fichero si no ha sido leído antes por otra llamada `Load()` o `Use()`.

Un uso común de `Use()` es cargar un fichero con funciones con las que queremos trabajar desde el intérprete, de modo que no tenemos que teclear las funciones cada vez ni tampoco hay que ejecutar el script. La función `Load()` resulta útil para ejecutar un script desde el intérprete cuando queremos hacerlo varias veces modificando el script sin salir del intérprete.

4.9. Un ejemplo real

A continuación explicamos un ejemplo del uso de Yacas ‘‘en la vida real’’: calcular la sucesión de Sturm asociada a un polinomio real de una variable real. Esta sucesión se utiliza en el teorema de Sturm para calcular el número de raíces reales de un polinomio en un intervalo de la recta real. El algoritmo para calcular esta sucesión de polinomios no es complicado, pero por si no lo conoces aquí tienes una breve explicación:

El algoritmo de Sturm parte de un polinomio con coeficientes reales $p(x)$ y su derivada $p'(x)$. Tomando $f_0(x) = p(x)$ como primer polinomio y $f_1(x) = p'(x)$ como segundo polinomio, se realizan sucesivas divisiones euclídeas utilizando en cada división el divisor de la anterior como dividendo y el opuesto del resto de la anterior como divisor. Al cabo de un número finito de iteraciones, en las que $f_j(x)$ es el resto de la división $j - 1$, se obtiene un resto nulo en la iteración $m + 1$.

$$\begin{aligned} f_0(x) &= f_1(x)q_1(x) - f_2(x) \\ f_1(x) &= f_2(x)q_2(x) - f_3(x) \\ &\vdots \\ f_{m-2}(x) &= f_{m-1}(x)q_{m-1}(x) - f_m(x) \\ f_{m-1}(x) &= f_m(x)q_m(x) \end{aligned}$$

A partir de los polinomios f_j obtenidos en estas divisiones se construye la sucesión de Sturm asociada a $p(x)$, que viene dada por

$$\left\{ \hat{f}_j(x) = \frac{f_j(x)}{f_m(x)} \right\}_{j=0}^n$$

El teorema de Sturm garantiza que el número de raíces del polinomio $p(x)$ en el intervalo $[a, b]$ es exactamente $V(a) - V(b)$, donde $V(x_0)$ es el número de variaciones de la sucesión de Sturm evaluada en el punto x_0 , i.e. el número de cambios de signo en la sucesión $\{\hat{f}_0(x_0), \hat{f}_1(x_0), \dots, \hat{f}_m(x_0)\}$.

Para implementar esto en Yacas definimos la función `Sturm()` que recibe el polinomio `P` junto con la variable en la que éste está expresado `x` y genera una lista `L` con el polinomio, su derivada y los sucesivos restos de las divisiones euclídeas. Para ahorrar esfuerzo computacional acotamos el número de términos de la lista con el grado del polinomio $P(x)$, ya que éste es el número máximo de divisiones necesarias para encontrar el máximo común divisor del polinomio y su derivada. Luego busca el último polinomio no nulo en `L`, que es el máximo común divisor del polinomio y su derivada, y construye la sucesión de Sturm en una nueva lista `S` que finalmente devuelve como valor de retorno de la función. La función `Simplify` simplifica cualquier expresión que reciba.

`Fichero sturm.js`

```
Sturm (P, x) := [
  Local (L, S, m, i);

  L := List ();
```

```

L := Append (L, P(x));
L := Append (L, D(x) P(x));

/* El número de términos en la sucesión de Sturm asociada a P(x)
 * es siempre no superior al grado del polinomio P(x) */
For ( i := 3 , i <= Degree(P(x)) , i++ )
  L := Append (L, - Mod ( L[i-2] , L[i-1] ) );

m := 0;
While ( m < Length (L) And
  Not IsZero (L[m+1]) ) m++;

S := List ();
For (i := 1, i <= m, i++)
  S := Append (S, Simplify (Div ( L[i] , L[m] ) ) );

S;
];

P(x) := RandomPoly (x, 3, 1, 10);
PrettyForm (P(x));
S := Sturm (P, x);
PrettyForm (S);

```

Ejemplo 4.2: Script en Yacas que implementa el algoritmo de Sturm.

En el script escribimos esta función y añadimos los comandos necesarios para ejecutar un test. La función `RandomPoly()` genera un polinomio aleatorio, lo mostramos con `PrettyForm()`, calculamos su sucesión de Sturm y la mostramos con `PrettyForm()` de nuevo. El resultado de ejecutar el script es el siguiente:

$$7 * x^3 + 10 * x^2 + 3 * x + 3$$

$$\frac{9 * \sqrt{-96605 * x^2 + 73253 * x - 181452}}{2685619} \cdot \frac{81 * (-139 * x - 117)}{19321} + 1$$

Este ejemplo real ha sido extraído del programa presentado por unos alumnos como trabajo para asignatura “Álgebra Computacional” que se imparte en la Facultad de Matemáticas de la Universidad de La Laguna. En teoría debieron realizar el trabajo utilizando Maple V, pero propusieron al profesor la alternativa de programarlo en Yacas y éste aceptó con curiosidad. El programa aproxima las raíces reales de un polinomio utilizando el algoritmo de Sturm para aislar las raíces en intervalos y aproximándolas con el método de la bisección (raíces de multiplicidad impar) o con el método de cristalización simulada (raíces de multiplicidad par). El trabajo fue expuesto en clase y tuvo la calificación de sobresaliente, el código y las transparencias están disponibles en <http://www.fmat.u11.es/~miguev/algcomp/>

Apéndice A

Licencia de Documentación Libre GNU

Versión 1.1, Marzo de 2000

Ésta es la GNU Free Document License (GFDL), versión 1.1 (de marzo de 2000), que cubre manuales y documentación para el software de la Free Software Foundation, con posibilidades en otros campos. La traducción ¹ no tiene ningún valor legal, ni ha sido comprobada de acuerdo a la legislación de ningún país en particular. Vea el original en <http://www.gnu.org/copyleft/fdl.html>

Los autores de esta traducción son:

- Igor Támara (ikks@bigfoot.com)
- Pablo Reyes (reyes_pablo@hotmail.com)
- Revisión: Vladimir Támara P. (vtamara@gnu.org)

Copyright © 2000 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Se permite la copia y distribución de copias literales de este documento de licencia, pero no se permiten cambios.

Preámbulo

El propósito de esta licencia es permitir que un manual, libro de texto, u otro documento escrito sea “libre” en el sentido de libertad: asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera comercial o no. En segundo término, esta licencia preserva para el autor o para quien publica una manera de obtener reconocimiento por su trabajo, al tiempo que no se consideran responsables de las modificaciones realizadas por terceros.

¹N. del T. Derechos Reservados en el sentido de GNU <http://www.gnu.org/copyleft/copyleft.es.html>

Esta licencia es una especie de “copyleft” que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Esto complementa la Licencia Pública General GNU, que es una licencia de copyleft diseñada para el software libre.

Hemos diseñado esta Licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: Un programa libre debe venir con los manuales que ofrezcan la mismas libertades que da el software. Pero esta licencia no se limita a manuales de software; puede ser usada para cualquier trabajo textual, sin tener en cuenta su temática o si se publica como libro impreso. Recomendamos esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

A.1. Aplicabilidad y definiciones

Esta Licencia se aplica a cualquier manual u otro documento que contenga una nota del propietario de los derechos que indique que puede ser distribuido bajo los términos de la Licencia. El “Documento”, en adelante, se refiere a cualquiera de dichos manuales o trabajos. Cualquier miembro del público es un licenciataria, y será denominado como “Usted”.

Una “Versión Modificada” del Documento significa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones y/o traducciones a otro idioma.

Una “Sección Secundaria” es un apéndice titulado o una sección preliminar al prólogo del Documento que tiene que ver exclusivamente con la relación de quien publica, o los autores del Documento, o el tema general del Documento(o asuntos relacionados) y cuyo contenido no entra directamente en este tema general. (Por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar matemáticas.) La relación puede ser un asunto de conexión histórica, o de posición legal, comercial, filosófica, ética o política con el tema o la materia del texto.

Las “Secciones Invariantes” son ciertas Secciones Secundarias cuyos títulos son denominados como Secciones Invariantes, en la nota que indica que el documento es liberado bajo esta licencia.

Los “Textos de Cubierta” son ciertos pasajes cortos de texto que se listan, como Textos de Portada o Textos de Contra Portada, en la nota que indica que el documento está liberado bajo esta Licencia.

Una copia “Transparente” del Documento, significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público general, cuyos contenidos pueden ser vistos y editados directamente con editores de texto genéricos o (para imágenes compuestas por píxeles) de programas genéricos de dibujo o (para dibujos) algún editor gráfico ampliamente disponible, y que sea adecuado para exportar a formateadores de texto o para traducción automática a una variedad de formatos adecuados para ingresar a formateadores de texto. Una copia hecha en un formato de un archivo que no sea Transparente, cuyo formato ha sido diseñado para impedir o dificultar subsecuentes modificaciones posteriores por parte de los lectores no es Transparente. Una copia que no es “Transparente” es llamada “Opaca”.

Como ejemplos de formatos adecuados para copias Transparentes están el ASCII plano sin formato, formato de Texinfo, formato de LaTeX, SGML o XML usando un DTD disponible ampliamente, y HTML simple que sigue los estándares, diseñado para modificaciones humanas. Los

formatos Opacos incluyen PostScript, PDF, formatos propietarios que pueden ser leídos y editados únicamente en procesadores de palabras propietarios, SGML o XML para los cuáles los DTD y/o herramientas de procesamiento no están disponibles generalmente, y el HTML generado por máquinas producto de algún procesador de palabras sólo para propósitos de salida.

La “Portada” en un libro impreso significa, la propia portada junto con las páginas siguientes necesarias para mantener la legibilidad del material, que esta Licencia requiere que aparezca en la portada. Para trabajos en formatos que no tienen Portada como tal, “Portada” significa el texto junto a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del trabajo.

A.2. Copia literal

Puede copiar y distribuir el Documento en cualquier medio, sea en forma comercial o no, siempre y cuando esta Licencia, las notas de derecho de autor, y la nota de licencia que indica que esta Licencia se aplica al Documento se reproduzca en todas las copias, y que usted no añada ninguna otra condición a las expuestas en esta Licencia. No puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

También puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias públicamente.

A.3. Copiado en cantidades

Si publica copias impresas del Documento que sobrepasen las 100, y la nota de Licencia del Documento exige Textos de Cubierta, debe incluir las copias con cubiertas que lleven en forma clara y legible, todos esos textos de Cubierta: Textos Frontales en la cubierta frontal, y Textos Posteriores de Cubierta en la Cubierta Posterior. Ambas cubiertas deben identificarlo a Usted clara y legiblemente como quien publica tales copias. La Cubierta Frontal debe mostrar el título completo con todas las palabras igualmente prominentes y visibles. Además puede añadir otro material en la cubierta. Las copias con cambios limitados en las cubiertas, siempre que preserven el título del Documento y satisfagan estas condiciones, puede considerarse como copia literal.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros (tantos como sea razonable colocar) en la cubierta real, y continuar el resto en páginas adyacentes.

Si publica o distribuye copias Opacas del Documento cuya cantidad exceda las cien, debe incluir una copia Transparente que pueda ser leída por una máquina con cada copia Opaca, o entregar en o con cada copia Opaca una dirección en red de computador públicamente-accesible conteniendo una copia completa Transparente del Documento, sin material adicional, a la cual el público en general de la red pueda acceder a bajar anónimamente sin cargo usando protocolos de standard público. Si usted hace uso de la última opción, deberá tomar medidas necesarias, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio por lo menos un año después de su última distribución de copias Opacas (directamente o a través de sus agentes o distribuidores) de esa edición al público.

Se solicita, aunque no es requisito, que contacte con los autores del Documento antes de redistribuir cualquier número de copias, para permitirle la oportunidad de que le suministren una versión del Documento.

A.4. Modificaciones

Puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las secciones 2 y 3 anteriores, siempre que Usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada haciendo el rol del Documento, por lo tanto licenciando la distribución y modificación de la Versión Modificada a quien quiera que posea una copia de éste. Además, debe hacer lo siguiente en la Versión Modificada:

- Uso en la Portada (y en las cubiertas, si hay alguna) de un título distinto al del Documento, y de versiones anteriores (que deberían, si hay alguna, estar listados en la sección de Historia del Documento). Puede usar el mismo título que el de las versiones anteriores al original siempre que quién publicó la primera versión lo permita.
- Listar en la Portada, como autores, una o más personas, o entidades responsables por la autoría o las modificaciones en la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (Todos sus autores principales, si son inferiores a cinco).
- Estado en la Portada del nombre de quien publica la Versión Modificada, como quien publica.
- Preservar todas las notas de derechos de autor del Documento.
- Añadir una nota de derecho de autor apropiada a sus modificaciones adyacentes a las otras notas de derecho de autor.
- Incluir, inmediatamente después de la nota de derecho de autor, una nota de licencia dando el permiso público para usar la Versión Modificada bajo los términos de esta Licencia, de la forma mostrada en la Adición (LEGAL) abajo.
- Preservar en esa nota de licencia el listado completo de Secciones Invariantes y en los Textos de las Cubiertas que sean requeridos como se especifique en la nota de Licencia del Documento
- Incluir una copia sin modificación de esta Licencia.
- Preservar la sección llamada “Historia”, y su título, y añadir a esta una sección estableciendo al menos el título, el año, los nuevos autores, y quién publicó la Versión Modificada como reza en la Portada. Si no hay una sección titulada “Historia” en el Documento, crear una estableciendo el título, el año, los autores y quién publicó el Documento como reza en la Portada, añadiendo además un artículo describiendo la Versión Modificada como se estableció en el punto anterior.
- Preservar la localización en red, si hay, dada en la Documentación para acceder públicamente a una copia Transparente del Documento, tanto como las otras direcciones de red dadas en el Documento para versiones anteriores en las cuáles estuviese basado. Éstas pueden ubicarse en la sección “Historia”. Se puede omitir la ubicación en red para un trabajo que sea publicado por lo menos 4 años antes que el mismo Documento, o si quien publica originalmente la versión da permiso explícitamente.

- En cualquier sección titulada “Agradecimientos” o “Dedicatorias”, preservar el título de la sección, y preservar en la sección toda la sustancia y el tono de los agradecimientos y/o dedicatorias de cada contribuyente que estén incluidas.
- Preservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Los números de sección o el equivalente no son considerados parte de los títulos de la sección. M. Borrar cualquier sección titulada “Aprobaciones”. Tales secciones no pueden estar incluidas en las Versiones Modificadas.
- Borrar cualquier sección titulada “Aprobaciones”. Tales secciones no pueden estar incluidas en las Versiones Modificadas.
- No retitular ninguna sección existente como “Aprobaciones” o conflictuar con título con alguna Sección Invariante.

Si la Versión Modificada incluye secciones, apéndices nuevos o preliminares al prólogo que califican como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, añada sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.

Puede añadir una sección titulada “Aprobaciones”, siempre que contenga únicamente aprobaciones de su Versión Modificada por varias fuentes. Por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como un standard.

Puede añadir un pasaje de hasta cinco palabras como un Texto de Cubierta Frontal, y un pasaje de hasta 25 palabras como un texto de Cubierta Posterior, al final de la lista de Textos de Cubierta en la Versión Modificada. Solamente un pasaje de Texto de Cubierta Frontal y un Texto de Cubierta Posterior puede ser añadido por (o a manera de arreglos hechos por) una entidad. Si el Documento ya incluye un texto de cubierta para la misma cubierta, previamente añadido por usted o por arreglo hecho por la misma entidad, a nombre de la cual está actuando, no puede añadir otra, pero puede reemplazar la anterior, con permiso explícito de quien publicó anteriormente tal cubierta.

El(los) autor(es) y quien(es) publica(n) el Documento no da(n) con esta Licencia permiso para usar sus nombres para publicidad o para asegurar o implicar aprobación de cualquier Versión Modificada.

A.5. Combinando documentos

Puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, y listadas todas como Secciones Invariantes del trabajo combinado en su nota de licencia.

El trabajo combinado necesita contener solamente una copia de esta Licencia, y múltiples Secciones Invariantes Idénticas que pueden ser reemplazadas por una sola copia. Si hay múltiples Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único añadiéndole al final de éste, en paréntesis, el nombre del autor o de

quien publicó originalmente esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes en la nota de licencia del trabajo combinado.

En la combinación, debe combinar cualquier sección titulada “Historia” de los varios documentos originales, formando una sección titulada “Historia”; de la misma forma combine cualquier sección titulada “Agradecimientos”, y cualquier sección titulada “Dedicatorias”. Debe borrar todas las secciones tituladas “Aprobaciones”.

A.6. Colecciones de documentos

Puede hacer una colección consistente del Documento y otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en los varios documentos con una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para una copia literal de cada uno de los documentos en cualquiera de todos los aspectos.

Puede extraer un solo documento de una de tales colecciones, y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraído, y siga esta Licencia en todos los otros aspectos concernientes a la copia literal de tal documento.

A.7. Agregación con trabajos independientes

Una recopilación del Documento o de sus derivados con otros documentos o trabajos separados o independientes, en cualquier tipo de distribución o medio de almacenamiento, no como un todo, cuenta como una Versión Modificada del Documento, teniendo en cuenta que ninguna compilación de derechos de autor sea clamada por la recopilación. A tal recopilación se le llama “agregado”, y esta Licencia no se aplica a los otros trabajos auto-contenidos y por lo tanto compilados con el Documento, o a cuenta de haber sido compilados, si no son ellos los mismos trabajos derivados del Documento.

Si el requerimiento de la sección 3 del Texto de la Cubierta es aplicable a estas copias del Documento, entonces si el Documento es menor que un cuarto del agregado entero, Los Textos de la Cubierta del Documento pueden ser colocados en cubiertas que enmarquen solamente el Documento entre el agregado. De otra forma deben aparecer en cubiertas enmarcando todo el agregado.

A.8. Traducción

Se considera a la Traducción como una clase de modificación. Así que puede distribuir traducciones del Documento bajo los términos de la sección 4. Reemplazar las Secciones Invariantes con traducciones requiere permiso especial de los dueños de derecho de autor, pero puede incluir traducciones de algunas o todas las Secciones Invariantes adicionalmente a las versiones originales de las Secciones Invariantes. Puede incluir una traducción de esta Licencia siempre que incluya también la versión Inglesa de esta Licencia. En caso de un desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la versión original en Inglés prevalecerá.

A.9. Terminación

No se puede copiar, modificar, sublicenciar, o distribuir el Documento excepto por lo permitido expresamente bajo esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y sus derechos serán automáticamente retirados de esa licencia. De todas maneras, los terceros que hayan recibido copias o derechos de su parte bajo esta Licencia no tendrán por terminadas sus licencias siempre que tales personas o entidades se encuentren en total conformidad con la licencia original.

A.10. Futuras revisiones de esta licencia

La Free Software Foundation puede publicar nuevas versiones o revisadas de la Licencia de Documentación Libre GNU cada cierto tiempo. Tales versiones serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar problemas o intereses. Vea <http://www.gnu.org/copyleft/>

Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que una versión numerada particularmente de esta licencia o “cualquier versión posterior” se aplica a ésta, tiene la opción de seguir los términos y condiciones de la versión especificada o cualquiera posterior que haya sido publicada (no como un borrador) por la Free Software Foundation. Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada (no como un borrador) por la Free Software Foundation.

A.11. Addendum

Para usar esta licencia en un documento que usted haya escrito, incluya una copia de la Licencia en el documento y ponga el siguiente derecho de autor y nota de licencia justo después del título de la página:

©Año Su Nombre.

Permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre GNU, Versión 1.1 o cualquier otra versión posterior publicada por la Free Software Foundation; con las Secciones Invariantes siendo LISTE SUS TÍTULOS, siendo LISTE el texto de la Cubierta Frontal, y siendo LISTELO el texto de la Cubierta Posterior.

Se incluye una copia de la licencia en la sección titulada “Licencia de Documentación Libre GNU”.

Si no tiene Secciones Invariantes, escriba “Sin Secciones Invariantes” en vez de decir cuáles son invariantes. Si no tiene Texto de Cubierta Frontal, escriba “Sin Texto de Cubierta Frontal” en vez de “siendo LISTE el texto de la Cubierta Frontal”; así como para la Cubierta Posterior.

Si su documento contiene ejemplos de código de programa no triviales, recomendamos liberar estos ejemplos en paralelo bajo su elección de licencia de software libre, tal como la Licencia de Público General GNU, para permitir su uso en software libre.

Índice alfabético

- Álgebra Computacional, 59
- arreglo, 42
- cálculo simbólico, 59
- CAS, 59
- estadística descriptiva, 51
- GNU R, 35
- inferencia estadística, 52
- lógica triestada, 41
- Octave, 1
 - control de flujo, 8
 - detección de bordes, 28
 - ecuaciones diferenciales, 20
 - entorno, 1
 - expresiones, 6
 - filtrado de imágenes, 27
 - funciones, 10
 - gráficas, 13
 - matrices, 17
 - polinomios, 21
 - procesamiento de señales, 22
 - teoría de control, 22
 - tipos de datos, 4
 - tratamiento de imágenes, 25
 - variables, 6
- proteus, 61
- R, 35
 - &&, 49
 - órdenes, 38
 - arrays, 42
 - índices, 43
 - dimensión, 42
 - indexado, 43
 - por columnas, 43
 - por filas, 43
 - producto, 43
 - subíndices, 43
 - asignación, operador de, 40
 - ayuda, 37
 - búsqueda, 37
 - clasificación de datos, 44
 - comandos, 38
 - comentarios, 38
 - contraste de hipótesis, 53
 - cor(), 51
 - correlación, 51
 - cov(), 51
 - covarianza, 51
 - data frame, 46
 - demos, 38
 - desviación típica, 51

- distribuciones de probabilidad, 50
 - densidad, 50
 - distribución, 50
 - quantiles, 50
 - simulación, 50
 - distribuciones tabuladas, 50
 - documentación, 35
 - entorno, 35
 - estadística descriptiva, 51
 - factores, 44
 - `fivenum()`, 51
 - frame, 46
 - funciones, 47
 - `break`, 50
 - control de flujo, 48
 - `for`, 49
 - `if`, 49
 - `ifelse()`, 49
 - leer desde fichero, 48
 - `next`, 50
 - `repeat`, 50
 - `while`, 49
 - gráficas, 55
 - `contour()`, 55
 - de sectores, 55
 - `hist()`, 55
 - `image()`, 55
 - `persp()`, 55
 - `pie()`, 55
 - `plot()`, 55
 - `qqplot()`, 55
 - `help()`, 37
 - `help.search()`, 37
 - historial, 39
 - hojas de datos, 46
 - conectar, 47
 - desconectar, 47
 - escribir en fichero, 47
 - leer desde fichero, 46
 - independencia de variables, 54
 - inferencia estadística, 52
 - leer comandos de un fichero, 39
 - lenguaje, 38
 - `library()`, 52
 - listas, 45
 - índices, 46
 - nombres, 46
 - matrices, 42
 - `cbind()`, 44
 - multiplicar, 43
 - `rbind()`, 44
 - `mean()`, 51
 - media, 51
 - `median()`, 51
 - mediana, 51
 - NA, 47
 - NaN, 47
 - `print()`, 40
 - prompt, 36
 - `range()`, 51
 - rango, 51
 - `read.table()`, 46
 - `sd()`, 51
 - sesiones, 37
 - `sink()`, 39
 - `source()`, 39
 - `summary()`, 51
 - tabla de contingencia, 54
 - tablas, 46
 - `tapply()`, 45
 - tests de normalidad, 52
 - valores perdidos, 47
 - `var()`, 51
 - varianza, 51
 - vectores, 39
 - índices, 42
 - indixado, 42
 - `names()`, 42
 - nombres, 42
 - operaciones elementales, 40
 - producto escalar, 40
 - secuencias, 41
 - subíndices, 42
 - vectores lógicos, 41
 - volcar en un fichero, 39
- Yacas, 59
- Álgebra Lineal, 66
 - CFrom, 62
 - control de flujo, 67
 - ejemplo real, 71
 - funciones, 63
 - gráficas, 69
 - listas, 65

matrices, 66
presición, 63
PrettyFrom, 62
programación, 70
proteus, 61
script, 70
TeXFrom, 62
variables, 63
vectores, 66