



Antonia M. Delgado
Juanjo Nieto
Aureliano M. Robles
Óscar Sánchez

Métodos numéricos básicos con Octave

Antonia M. Delgado
Juanjo Nieto
Aureliano M. Robles
Óscar Sánchez

Granada, septiembre 2018



Está permitido copiar y distribuir este texto en cualquier medio o formato bajo los siguientes términos: se ha de dar crédito de manera adecuada (atribución), no se puede hacer uso del material con propósitos comerciales, si remezcla, transforma o crea a partir del material no podrá distribuir el material modificado.

Antonia M. Delgado Amaro
amdelgado@ugr.es
Dpto. de Matemática Aplicada
Universidad de Granada

Juanjo Nieto Muñoz
jjmnieto@ugr.es
Dpto. de Matemática Aplicada
Universidad de Granada

Aureliano M. Robles Pérez
arobles@ugr.es
Dpto. de Matemática Aplicada
Universidad de Granada

Óscar Sánchez Romero
ossanche@ugr.es
Dpto. de Matemática Aplicada
Universidad de Granada

Los autores autorizan expresamente a la Editorial Técnica AVICAM a distribuir copias de este texto. En este sentido la renuncia expresa de los autores a cualquier beneficio económico producto de su venta compromete a esta Editorial a que este hecho repercuta únicamente en una reducción del precio final, que lo haga más asequible a los estudiantes (y resto de usuarios) interesados en el mismo.

Se hace constar que los códigos listados en las páginas 181 y 182 se distribuyen por separado bajo modalidad de licencia. Todos los códigos contenidos están disponibles en el sitio web:

<http://www.ugr.es/local/jjmnieto/MNBOctave.html>

ISBN: 978-84-16535-79-8

Depósito Legal: GR 1213-2016

A nuestras familias

Índice general

	Pág.
Prólogo	XI
Presentación	XIII
1 Primeros pasos con Octave	1
1.1 El entorno Octave, uso básico	1
1.1.1 Manejando datos: matrices, polinomios, cadenas, variables . .	3
1.1.2 Aritmética básica	9
1.1.3 Cómo guardar nuestro trabajo	12
1.2 Creación de funciones y <i>scripts</i> en Octave	13
1.2.1 Primeros pasos en programación: condicionales y bucles	17
1.3 Algunos apuntes finales	24
1.3.1 Octave vs MatLab	26
1.3.2 Números de punto flotante vs números reales	27
2 Ecuaciones no lineales	29
2.1 Fundamentos teóricos	29
2.1.1 Existencia y unicidad de soluciones	29
2.1.2 Método de bisección	30
2.1.3 Métodos de regula-falsi y secante	31
2.1.4 Método de Newton-Raphson	34
2.2 Localización de raíces de funciones en Octave	35
2.3 Implementación de algunos métodos	37
2.3.1 Método de bisección	37
2.3.2 Método de la secante	41
2.3.3 Newton-Raphson	43
3 SEL I: métodos directos	45
3.1 Fundamentos teóricos	45
3.1.1 Resolución de sistemas triangulares	45
3.1.2 Método de Gauss	46

3.1.3	Factorización de matrices	48
3.1.4	Condicionamiento de una matriz	50
3.2	Implementación de algunos métodos	51
3.2.1	Sustitución regresiva y sustitución progresiva	52
3.2.2	Método de Gauss	53
3.2.3	Factorización LU	54
4	SEL II: métodos iterativos	57
4.1	Fundamentos teóricos	57
4.1.1	Métodos iterativos clásicos: Jacobi, Gauss-Seidel y relajación	57
4.1.2	Métodos de descenso: rápido y gradiente conjugado	60
4.2	Implementación de algunos métodos	62
4.2.1	Método de Gauss-Seidel	62
4.2.2	Método del gradiente conjugado	64
5	Valores y vectores propios	67
5.1	Fundamentos teóricos	67
5.1.1	Método de las potencias	71
5.1.2	El método QR	72
5.2	Implementación de algunos métodos	73
5.2.1	Los comandos <i>eig</i> , <i>poly</i> y <i>expm</i>	73
5.2.2	Método de las potencias	75
5.2.3	Método QR	76
6	Interpolación	79
6.1	Fundamentos teóricos	79
6.1.1	Problema general de interpolación de datos lagrangianos	80
6.1.2	Interpolación polinomial para datos de tipo Hermite	85
6.1.3	Error en la interpolación polinomial	87
6.1.4	Funciones polinómicas a trozos: <i>splines</i>	88
6.2	Implementación de algunos métodos	89
6.2.1	Interpolación polinomial	89
6.2.2	Interpolación <i>spline</i>	94
7	Mínimos cuadrados	101
7.1	Fundamentos teóricos	101
7.1.1	Ajuste por mínimos cuadrados discreto mediante polinomios	103
7.1.2	Solución generalizada de sistemas de ecuaciones lineales	106
7.1.3	Ajuste discreto mediante funciones cualesquiera	107
7.1.4	Ajuste de funciones por mínimos cuadrados continuo	109
7.2	Implementación de algunos métodos	110
7.2.1	Aproximación por mínimos cuadrados discreta	110
7.2.2	Aproximación por mínimos cuadrados continua	113

8	Derivación y cuadratura numérica	115
8.1	Fundamentos teóricos	115
8.1.1	Derivación numérica	116
8.1.2	Integración numérica	118
8.1.3	Reglas compuestas de integración numérica. Error de cuadratura	120
8.2	Implementación de algunos métodos	122
8.2.1	Fórmulas de derivación numérica	122
8.2.2	Fórmulas de cuadratura o integración numérica	125
8.2.3	Fórmulas compuestas	128
9	Ecuaciones diferenciales I: PVI	131
9.1	Fundamentos teóricos	131
9.1.1	Algunos métodos de integración numérica	133
9.2	Resolución numérica de PVI's	135
9.2.1	El comando <code>lsode</code>	137
9.2.2	Implementación del método Runge-Kutta de orden 4	140
10	Ecuaciones diferenciales II: PVF	145
10.1	Fundamentos teóricos	146
10.2	Resolución numérica de PVFs	147
10.2.1	Métodos en diferencias finitas para problemas lineales	147
10.2.2	Métodos de tiro simple	152
	Ejercicios	155
1	Uso básico de Octave	155
2	Resolución de ecuaciones no lineales	157
3	Métodos directos para SEL	160
4	Métodos iterativos para SEL	163
5	Valores y vectores propios	165
6	Interpolación	166
7	Mínimos cuadrados	168
8	Derivación y cuadratura numérica	170
9	Problemas de valores iniciales	171
10	Problemas de valores en la frontera	172
	Bibliografía	175
	Índice alfabético	177
	Lista de programas	181

Prólogo

Desde el último tercio del siglo pasado, la imperiosa y continuamente creciente necesidad de cálculo intensivo en múltiples ámbitos de la actividad humana ha impuesto el desarrollo de medios para llevarlo a cabo de forma eficiente y barata. Se fabrican nuevos procesadores cada vez más potentes, y se diseñan lenguajes de programación orientados al objeto. En cuanto a la resolución de problemas numéricos, el usuario típico, casi nunca un programador profesional, quiere resolver el suyo sin perder demasiado tiempo escribiendo largas rutinas, depurando errores, o tratando de entender qué significan los ininteligibles mensajes de error que casi nunca aclaran su verdadera causa. Lo ideal es que un lenguaje orientado al cálculo numérico y a la resolución de problemas mediante métodos numéricos se escriba yendo al grano, es decir, con la menor cantidad posible de lo que se podría calificar de “código farfolla” que, sin dejar de ser necesario, podría ser ocultado como parte interna del lenguaje, permitiendo de este modo la fácil lectura y visibilidad de lo que realmente importa. Lenguajes no específicamente orientados como `C++` necesitan, por lo general, 10 líneas de código farfolla por cada línea neta de método numérico. Por otro lado, un lenguaje interpretado permite una interacción mucho más ágil que uno compilado, y las diferencias en tiempos de ejecución que antes existían entre ambos tipos están en la actualidad prácticamente superadas.

Octave es un lenguaje orientado exclusivamente al cálculo numérico, con especial indicación en el tratamiento masivo de cantidades numéricas, principalmente vectores y matrices. La mayor parte de las funciones, operadores y órdenes, que suelen existir en todos los lenguajes para un argumento simple, admiten en Octave un argumento múltiple, vector o matriz, lo que permite ahorrar la escritura de estructuras de repetición y simplificar enormemente la lectura y la depuración. Además, Octave es parte del proyecto GNU, por lo que su uso es libre, y es compatible en gran medida con MatLab®.

Los autores de este libro son profesores de larga experiencia en la docencia de métodos numéricos en matemáticas y diversas ingenierías, en el empleo de dichas técnicas en su labor de investigación, así como en el uso de software de cálculo científico. En los contenidos de este libro, escogidos con gran acierto, se ofrece al lector todo lo necesario para iniciarse en Octave mediante las técnicas de métodos numéricos más comunmente empleadas, que suelen enseñarse en un primer o segundo curso de análisis numérico de cualquier carrera de ciencias o ingeniería. Para un aprendizaje inicial, esta obra no requiere del aporte de otros manuales o textos, ni de Octave ni de análisis numérico,

pues contiene todo lo necesario, desde los fundamentos teóricos hasta su implementación práctica mediante un programa en Octave, con ilustrativos ejemplos. Por supuesto, no pretende sustituir ni constituye un curso de métodos numéricos, pues no contiene profundos teoremas ni largas demostraciones, ni muchos otros aspectos de índole teórica, ni tampoco es un manual del lenguaje Octave, ya que no se usan más que las funciones necesarias para la resolución de los problemas planteados. Pero con toda seguridad es un texto autosuficiente para la iniciación y entrenamiento en la programación de métodos numéricos, y un valioso manual de consulta a la hora de “rescatar” del olvido una solución a un problema.

Granada, junio de 2016

José Martínez Aroza

Presentación

Este libro está dirigido a estudiantes (y profesores) de primeros cursos de carreras científicas que van a desarrollar e implementar métodos numéricos para resolver problemas matemáticos elementales. En él, pretendemos proporcionar las técnicas y herramientas básicas para la implementación de algoritmos numéricos usando el programa de cálculo científico Octave, lenguaje de libre distribución desarrollado por John W. Eaton y colaboradores [8]. Para ello, en cada capítulo nos centraremos en un determinado problema matemático cuya resolución requiera de herramientas numéricas, comenzando con una breve introducción a los fundamentos matemáticos inherentes al problema y pasando directamente a la implementación de los métodos que conduzcan a su resolución; cuando proceda, indicaremos los comandos directos que proporcionan la resolución del problema y, en algunos los casos, describiremos e implementaremos los algoritmos que usualmente el estudiante debe programar. Dado que pretendemos desarrollar un manual básico de implementación de métodos numéricos orientado a los estudiantes, hemos elegido los problemas matemáticos a estudiar en el orden en que normalmente aparecen en los temarios de las asignaturas, que suele ser en orden creciente de dificultad, proponiendo abundantes ejercicios de desarrollo cuya complejidad va en aumento, pero manteniendo, en la medida de lo posible, la consistencia de cada tema, de modo que pueda ser consultado de modo independiente. Además, debemos hacer notar que algunos de los ejemplos incluidos a lo largo del texto, así como varios de los ejercicios propuestos, provienen de problemas originados en otras áreas de la ciencia y la tecnología, de manera que a través de ellos podemos percibir la utilidad práctica de los contenidos aquí expuestos.

Por todo lo hasta aquí dicho, la introducción matemática de cada capítulo, pretende sólo establecer los resultados básicos necesarios para comprender y desarrollar los algoritmos numéricos propuestos, sin hacer una descripción exhaustiva del análisis matemático inherente que, en cada capítulo, se deja al lector interesado como bibliografía complementaria a desarrollar. Con ello, buscamos más sencillez en los contenidos de este libro, enfatizando en todo momento el contenido aplicado sobre el desarrollo de métodos numéricos. De igual modo, tampoco pretendemos desarrollar un manual completo de Octave, ya que existen numerosos disponibles [2, 5, 19]. No obstante, el libro dispone de un índice alfabético exhaustivo que incluye comandos de Octave referenciando las páginas donde han sido empleados. De esta forma el texto puede ser utilizado como

complemento a dichos manuales ya que contiene numerosos ejemplos de uso, muchas veces no trivial, de dichos comandos.

Por último, y como no podía ser de otro modo, animamos a los lectores a que nos remitan todas las correcciones y sugerencias que consideren oportunas para una mejora del presente manual.

Granada, septiembre de 2016

Los autores

Capítulo 1

Primeros pasos con Octave

En este primer capítulo introductorio pretendemos realizar un acercamiento elemental al programa Octave, como lenguaje científico de programación, que nos permita un uso directo y auto-dirigido y nos proporcione recursos suficientes para afrontar las dificultades específicas matemáticas y/o de programación que vayamos planteando en cada capítulo posterior. Al igual que en el resto de capítulos no pretendemos un análisis exhaustivo de contenidos matemáticos, en este tema no vamos a hacer una descripción completa de Octave, sino una introducción práctica orientada a los contenidos posteriores, que podrá ser complementada con las referencias bibliográficas que vamos a ir dando en cada caso. En primer lugar, para familiarizarnos con el programa Octave, vamos a tratar algunas nociones básicas sobre el tipo de datos con los que trabaja y su aritmética y control de flujo.

1.1. El entorno Octave, uso básico

Octave, como tal programa, se puede dirigir desde un terminal en el que se ejecutan los comandos escritos tras el *prompt* “*octave :>*”, al que nos referiremos como “*>*” a partir de este momento. La siguientes líneas muestran algunos comandos básicos que podemos ejecutar desde un terminal:

<i>></i> 2+7	operaciones usuales; 2+7 realiza la suma indicada;
<i>></i> pwd	comandos típicos de Unix; <i>pwd</i> para directorio actual de trabajo;
<i>></i> ls	<i>ls</i> lista los ficheros y directorios en el directorio de trabajo;
<i>></i> exit	salir del programa Octave.

Esta forma de trabajar en un terminal hace que ciertas tareas sean tediosas, por lo que normalmente emplearemos un entorno gráfico para usuario (*Graphical User Interface*, GUI) que haga más agradable el manejo de Octave. En este sentido Octave proporciona una GUI oficial en http://en.wikipedia.org/wiki/GNU_Octave aunque existen ya diversas opciones de libre distribución (y algunas privativas) para los sistemas operativos usuales

(Linux, MacOS y Windows). En todas ellas, con leves diferencias, podemos identificar los siguientes elementos o zonas de trabajo (en la figura 1.1 vemos un ejemplo de GUI: el *OctaveGUI*):

- Una barra de menús, que dependerá del GUI utilizado.
- Una ventana de comandos (donde se ejecutarán propiamente todas las órdenes de Octave).
- Un editor de textos (donde editaremos nuestros “programas”).
- Un historial de comandos (listado de órdenes ejecutadas).
- Un “espacio de trabajo” donde se listan las variables definidas en memoria.
- Un explorador (para navegar en el árbol de directorios).
- Una ventana gráfica (que se abrirá sólo cuando la orden ejecutada demande una salida gráfica, y que suele depender de un programa gráfico externo: GNUplot, Aqua, etc.).

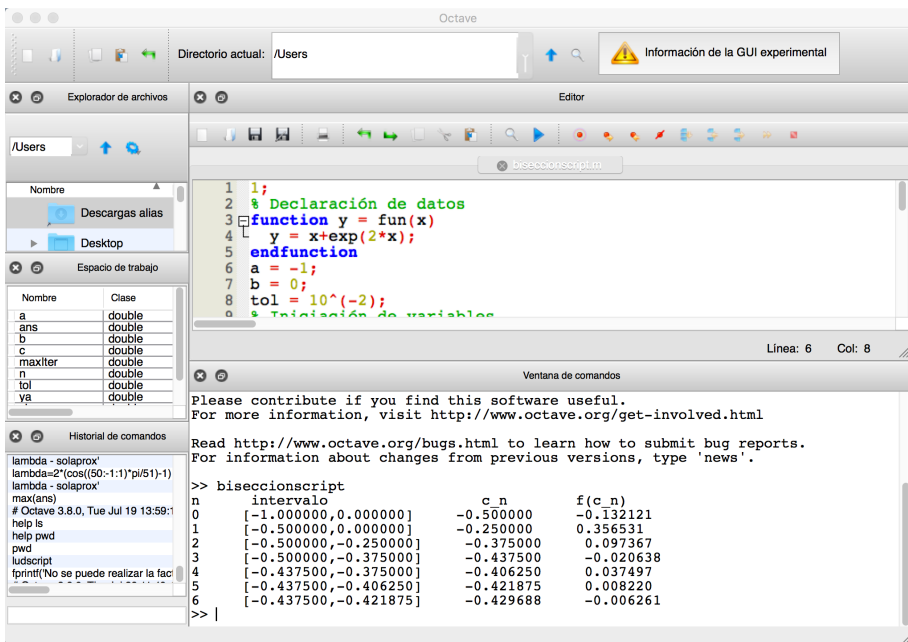



Figura 1.1: Entorno gráfico del programa OctaveGUI.

Como decíamos, podemos hacer un uso básico de Octave desde la línea de comandos del terminal, escribiendo directamente las órdenes y obteniendo los resultados en el terminal. Aunque esta no sea la manera usual de trabajar, es muy útil conocer su funcionamiento antes de pasar a crear programas más complejos. Es conveniente saber que:

- Para que Octave interprete los comandos escritos hay que pulsar la tecla *enter* .
- El *atajo de teclado* *Ctrl+C* interrumpe la ejecución de un comando y podemos usarlo cuando no se produce respuesta alguna en un tiempo razonable (¡se ha quedado colgado!).
- Si al final de un comando escribimos un **punto y coma** “;” Octave no mostrará el resultado de dicho comando (¡aunque sí se haya ejecutado! y por tanto puede haber realizado cambios en los datos almacenados en la memoria del ordenador).
- En el caso de que un comando sea excesivamente largo se puede escribir en varias líneas si se emplean **puntos suspensivos** “...” entre una línea y la siguiente (véase, por ejemplo, las líneas 6 y 7 del código de la página 124).

1.1.1. Manejando datos

Octave es capaz de trabajar con distintas clases de datos, entre las que nosotros destacamos las siguientes: **números** (tanto reales como complejos), **matrices** (que incluyen como casos particulares los **vectores** y los **polinomios**) y las **cadenas de caracteres** (que pueden ser usadas como **variables**). Es necesario tener claro en cada momento qué dato tenemos entre manos, ya que hay operadores y funciones que, dependiendo de la clase de dato sobre el que actúen lo hacen de distinta manera, o incluso puede que no estén bien definidas. La naturaleza de cada dato se puede consultar en el “Espacio de trabajo”.

Números (reales y complejos). Los números reales se representan mediante dígitos donde la coma decimal, al igual que en las calculadoras, se representa con un punto. Los números complejos se escriben del modo usual teniendo en cuenta que, en su parte imaginaria, la unidad compleja se representa mediante el símbolo “i” que es conveniente acompañar siempre de su coeficiente real yuxtapuesto, incluso cuando sea un uno: “1i”, “2.5i”, etc. (véase el ejercicio §1.2). Más adelante, cuando aprendamos a almacenar datos en variables, comprenderemos por qué conviene añadir el coeficiente. Comprobemos, mediante algunas operaciones elementales, la dinámica básica para escribir y trabajar con números reales y complejos, así como el uso de los paréntesis para señalar el orden en el que deseamos ejecutar los comandos, y el hecho de que **dejar o no espacios entre operaciones es irrelevante**.

```
> (1.5 + 2.5i)*1i
> 1.5 + 2.5i*1i
> 3 + 5.5 * (4.1 - 7)
> 3+5.5*(4.1-7)
> a = (2 + 1i)*(2 - 1i)
```

En el último comando ejecutado, hemos creado la *variable* “a” a la que asignamos (mediante el símbolo =) el valor de la operación realizada (en este caso $(2 + i) * (2 - i) = 5$);

en adelante, cada vez que aparezca la variable `a`, su valor será computado como 5. Más adelante, en esta misma sección, abordaremos el tema de las variables.

Matrices (y vectores). Octave está orientado al uso del cálculo vectorial y matricial, por lo que podemos adivinar que la sintaxis de estos elementos será un punto clave para su manejo. En lo que sigue hablaremos únicamente de matrices pues los vectores son, en particular, matrices con una única fila o columna. No obstante, al tratar la aritmética con estos elementos, hay que señalar que algunas funciones de Octave no trabajan igual cuando actúan sobre un vector fila, sobre un vector columna, o sobre una matriz. Empezaremos viendo diferentes formas de definir matrices. La definición “elemento a elemento” de una matriz o de un vector se hace introduciendo sus componentes entre corchetes, de forma que los elementos de una fila se separan mediante *comas* (o espacios), y las distintas filas se separan con *punto y coma* (o retornos de carro). De esta forma, por ejemplo:

```
> b = [1, 6]           asigna a la variable b el vector fila (1,6);
> b = [1 6]           y lo mismo, usando espacios.
> A = [1, 4; 3, 2]     Asigna a la variable A la matriz cuadrada  $\begin{pmatrix} 1 & 4 \\ 3 & 2 \end{pmatrix}$ ;
> A = [1 4
3 2]                 y lo mismo, usando espacios y salto de carro.
> c = [2; 7]          Asigna a la variable c el vector columna  $\begin{pmatrix} 2 \\ 7 \end{pmatrix}$ ;
> c = [2
7]                  y lo mismo, usando saltos de carro.
```

Observamos que, aunque podemos separar componentes mediante espacios y saltos de carro en lugar de comas y punto y coma, respectivamente, no recomendamos el empleo de espacios en blanco como delimitadores dentro de una fila, ya que puede dar lugar a confusión (véase el ejercicio §1.4). En este texto, usaremos siempre el punto y coma para cambiar de columna.

Veamos otra forma de definir vectores mediante **rangos**. Un rango es una forma simple de escribir un vector fila cuyos elementos están equiespaciados. La sintaxis de un vector fila definido mediante un rango es la siguiente (también válido sin corchetes),

[inicio : paso : tope]

Aquí `inicio` es el primero de los valores del vector, `paso` es el incremento que sufrirán los siguientes valores, y `tope` es la cantidad que no pueden superar dichos valores. En el caso de que el `paso` se omita, el valor por defecto es 1. Con esta definición no sabremos, *a priori*, el número de componentes de un vector (aunque la podríamos obtener mediante la orden `length(vector)`), ya que lo que fijamos es la distancia entre componentes, el `paso`. Alternativamente podemos usar

linspace(inicio, fin, num)

que genera un vector fila con exactamente `num` valores equiespaciados, siendo `inicio` el primero de ellos y `fin` el último. Observamos las siguientes definiciones:

```
> 3:7                                genera el vector (3,4,5,6,7),
> [3:2:12]                           genera el vector (3,5,7,9,11),
> 1:0.2:2                             genera el vector (1,1.2,1.4,1.6,1.8,2),
> linspace(0,1,5)                     genera el vector (0,0.25,0.5,0.75,1).
```

Los elementos que se combinan dentro de una matriz pueden ser, a su vez, matrices. De esta forma podemos crear matrices yuxtaponiendo matrices o vectores, siempre que las dimensiones de los elementos a combinar sea apropiada. Lo comprobamos mediante algunos ejemplos:

```
> [A, A]
> [A, -A; A, A]
> [b; b]
> [A, b]
> [A, c]
```

Por lo general, Octave devolverá un **mensaje diagnóstico** cuando uno trata de llevar a cabo operaciones imposibles o mal definidas, como ha ocurrido en uno de los ejemplos anteriores.

Existen rutinas para definir **matrices específicas**. Algunas de ellas son:

el comando...	crea ...
<code>zeros(m, n)</code>	una matriz de orden m por n con componentes 0
<code>ones(m, n)</code>	una matriz de orden m por n llena de unos
<code>eye(n)</code>	la matriz identidad de orden n
<code>eye(m, n)</code>	la matriz de orden $m \times n$ con 1 en la diagonal principal y 0 fuera
<code>rand(m, n)</code>	una matriz de orden $m \times n$ con componentes entre 0 y 1 aleatorias
<code>diag(v, k)</code>	una matriz de ceros cuadrada con el vector v en la diagonal k -ésima
<code>diag(A, k)</code>	el vector de la diagonal k -ésima de la matriz A

Existen otros comandos específicos para definir matrices “conocidas”, como son las matrices de Vandermonde, Hilbert, Toeplitz, etc., que iremos viendo a lo largo del libro.

A continuación presentamos comandos para **seleccionar y manipular** los elementos de una matriz o vector. La dimensión de una matriz se obtiene mediante el comando `size(A)`, que devuelve un vector con el número de filas y columnas de la matriz A . El comando `length(A)` devuelve el valor más grande entre el número de filas y columnas de la matriz A , por lo que se emplea usualmente para obtener la longitud de un vector. En ocasiones, es necesario extraer una cierta componente de una matriz; la forma de seleccionar el elemento situado en la fila k -ésima y columna m -ésima de una matriz almacenada en la variable A es mediante la orden `A(k, m)`. Así, por ejemplo:

> `A(1, 2)` es el elemento situado en la primera fila y segunda columna de A .

Análogamente, si queremos cambiar este valor por un 8, por ejemplo, no tenemos más que ejecutar:

> `A(1, 2) = 8`

También podemos **aumentar el tamaño** de una matriz añadiendo las componentes que necesitemos; Octave completará automáticamente los huecos con ceros. Por ejemplo:

> `M = []` iniciamos una matriz M vacía,
 > `M(2, 3) = 8` la convertimos en una matriz 2×3 con $M_{2,3} = 8$ y 0 el resto,
 > `M(4, 1) = -2` y ahora en una matriz 4×3 con $M_{2,3} = 8$, $M_{4,1} = -2$ y 0 el resto.

Si queremos **seleccionar la última componente** de una matriz o vector de tamaño desconocido, podemos echar mano de los comandos `size` y `length` para determinar primero su dimensión, pero es más rápido usar directamente el comando `end`. Por ejemplo,

> `A(1, end)` es el último elemento de la primera fila.

De forma similar podemos **extraer una submatriz** empleando los rangos de filas y columnas que nos interesen como se indica a continuación. Tomamos, por ejemplo, la matriz 4×3 definida por `D = eye(4, 3)`. Se puede extraer una submatriz utilizando

`D(rango filas, rango columnas)`

donde los rangos de filas y columnas a considerar se indican mediante vectores que contienen sus números de orden dentro de la matriz, usando **dos puntos para abarcar todo el rango de valores**. Así, por ejemplo:

> `D([1, 2], [1, 3])` entradas de las dos primeras filas y columnas primera y tercera;
 > `D(:, [1, 3])` columnas primera y tercera completas;
 > `D([1:3], :)` las tres primeras filas completas;
 > `D(:, end-1)` la penúltima columna completa.

Podemos apreciar la facilidad que aporta esta sintaxis en la **permuta y eliminación** de filas/columnas de una matriz, operaciones que vamos a emplear a menudo. Por ejemplo:

> `D([3, 1], :) = D([1, 3], :)` permuta las filas primera y tercera de la matriz D ;
 > `D(1, :) = []` elimina la primera fila de la matriz D ;
 > `D(:, 2) = []` elimina la segunda columna de la matriz D .

Casos particularmente interesantes son los comandos `triu(A)` y `tril(A)` que extraen, respectivamente, la **parte triangular superior e inferior** de una matriz A . Por defecto, ambos comandos incluyen la diagonal de la matriz A , aunque se puede añadir un “-1” como segundo argumento opcional en caso de que no se desee. Además, en la

sección 1.2.1 veremos cómo seleccionar los elementos de una matriz que verifiquen una condición. Para más detalles véase [5, Data types. Object Sizes y Matrix Manipulation].

Polinomios. La representación en Octave de una función polinómica de la forma

$$p(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$$

se realiza **mediante el vector fila** `p = [a_n, ..., a_1, a_0]`, cuyas componentes son los coeficientes del poninomio. Por ejemplo:

```
> p1 = [2, -1, 3]           representa al polinomio p1(x) = 2x^2 - x + 3;
> p2 = [1, 0, 0]           Representa a p2(x) = x^2; nótese la necesidad de 0.
```

Por lo tanto su tratamiento como funciones de `x`, y no como simples vectores, requiere órdenes específicas en Octave. Veamos algunos ejemplos.

el comando...	crea ...
<code>polyval(p1, x)</code>	las evaluaciones del polinomio <code>p1</code> sobre las componentes de <code>x</code>
<code>polyderiv(p1)</code>	el polinomio resultante de derivar <code>p1</code>
<code>polyint(p1)</code>	el polinomio resultante de integrar <code>p1</code> y con valor 0 en 0
<code>roots(p1)</code>	un vector columna con las raíces complejas del polinomio <code>p1</code>
<code>poly(v)</code>	un polinomio cuyas raíces son las componentes del vector <code>v</code>
<code>poly(A)</code>	el polinomio característico de la matriz cuadrada <code>A</code>
<code>polyout(p, 'x')</code>	la expresión estándar del polinomio <code>p</code>
<code>conv(p1, p2)</code>	el producto de los polinomios <code>p1</code> y <code>p2</code>
<code>[c, r] = deconv(p1, p2)</code>	el cociente <code>c</code> y el resto <code>r</code> de dividir <code>p1</code> entre <code>p2</code>

Resaltamos que el comando `polyval` realiza las evaluaciones mediante el algoritmo de Horner-Ruffini, que es mucho más estable numéricamente que la evaluación directa.

En los capítulos que requieran el uso de polinomios, desarrollaremos más los comandos asociados. Para profundizar, podemos consultar [5, Polynomial Manipulations].

Cadenas de caracteres. Las cadenas de caracteres o *strings* se definen en Octave como *texto delimitado entre comillas (simples o dobles)* de la siguiente forma:

```
> 'cualquier texto entre las comillas incluyendo espacios'
```

Al igual que una matriz o un vector, es posible **seleccionar uno de los caracteres** de una cadena apelando a su posición. El siguiente ejemplo,

```
> cadena = "pkhatewcous"
> cadena([1 4 8 end-2])
```

produce como salida `paco`, es decir, las componentes que ocupan las posiciones primera, cuarta, octava y antepenúltima de la cadena que hemos llamado `cadena`. Por cierto, es

importante en todo código diferenciar entre los datos que son numéricos, los comandos y las cadenas de caracteres, para evitar posibles errores posteriores. Para profundizar sobre las cadenas de caracteres (comparación, concatenación, etc.) véase [5, Strings].

Variables. Como hemos ido viendo a lo largo de esta sección, podemos almacenar cualquier objeto de Octave (números, vectores, cadenas de caracteres, etc.) en una *variable* mediante el operador de **asignación** “=” del siguiente modo:

```
nombre_de_la_variable = objeto_asignado
```

Los nombres de las variables pueden estar compuestos por letras, números o guiones de subrayado, siempre y cuando **no empiecen por un número**, ya que sería confundido con un coeficiente, y teniendo siempre en cuenta que Octave **distingue entre mayúsculas y minúsculas**. Ejecutamos como muestra las siguientes instrucciones:

```
> a = 5; b = 3;                                definimos sendas variables a y b,
> b = a + b                                     podemos operar con ellas e incluso redefinirlas.
b = 8
```

En ocasiones es posible que necesitemos definir varias variables **al mismo tiempo** (por ejemplo para permutar los valores de dos de ellas) y, en este caso, podemos (¡debemos!) usar el comando de reparto `deal` como sigue:

```
> [a, b] = deal(5, 3);                         definimos sendas variables a y b a la vez,
> [a, b] = deal(b, a)                          y las modificamos usando sus valores previos.
a = 3
b = 5
```

Notemos que si intentamos hacer la permutación anterior escribiendo

```
> [a, b] = [b, a]
```

se produciría un error. Esto es debido a que, para Octave, `[b, a]` es un único objeto, un vector, y no dos objetos que puedan ser guardados por separado, como pretendía nuestra orden; de ahí la necesidad del comando `deal`.

En cada sesión de trabajo, Octave guarda en memoria las distintas variables que se han definido, por lo que es conveniente tener en mente sus nombres y qué hemos almacenado en cada una. Es por esto que el programa ofrece la posibilidad de **visualizar el listado de variables definidas** hasta el momento mediante el comando `who`. Al usar un entorno gráfico GUI concreto, es habitual que exista una ventana (esto es, el espacio de trabajo) que contenga tanto el nombre de las variables creadas como el tipo de objeto que son cada una de ellas. Cuando alguna variable deja de ser útil se puede **eliminar** mediante el comando `clear` del siguiente modo:

```
> clear('nombre1', 'nombre2')
```

es decir, usando como argumentos las cadenas de caracteres que definen a las variables a eliminar. Si usamos `clear()` (esto es, sin argumentos) eliminaremos todas las variables que haya en memoria. Hemos de tener en cuenta que existen diversas variables (y comandos) predefinidas por defecto en el programa, como por ejemplo `pi` o `e` (donde están almacenados los valores del número $\pi = 3.141592\dots$ y del número $e = 2.718281\dots$ respectivamente), y que no deben ser alteradas, aunque Octave permite hacerlo. Si accidentalmente “pisamos” una variable predefinida podemos provocar errores indeseados, pero es fácil subsanarlos limpiando la variable creada, lo que nunca eliminará la que contiene Octave. Lo observamos en este ejemplo:

```
> pi = 7; roots = 5;      redenominamos la variable pi y el comando roots;
> roots([1, 0, 1])       produce un mensaje de error;
> sin(pi)                devuelve 0.65699, el seno de 7, y no el esperado seno de pi;
> clear('pi', 'roots')   devuelve a su estado inicial tanto pi como roots.
```

Por último, una variable interesante en Octave es la variable `ans`, donde se guarda el último valor obtenido como respuesta y no guardado expresamente en alguna variable.

1.1.2. Aritmética básica

En este punto vamos a introducir en primer lugar la aritmética con números y en segundo lugar la de matrices y vectores, para facilitar su comprensión.

Operar con números. Octave tiene una serie de operadores y funciones predefinidas, algunas de las cuales se indican en la siguiente tabla (véase [5, Arithmetic] para un listado más completo).

los comandos...	realizan operaciones de tipo...
<code>+, -, *, /</code>	aritmética básica (suma, diferencia, producto, división)
<code>^, sqrt, exp, log</code>	potencia, raíz cuadrada, exponencial, logaritmo natural
<code>sin, cos, tan, cot, sec, csc</code>	trigonométricas: seno, coseno, tangente, cotangente, secante, cosecante
<code>asin, acos, atan...</code>	versiones arco de las anteriores (anteponiendo una a)
<code>hsin, hcos, htan...</code>	hiperbólicas (anteponiendo una h)
<code>abs, arg, real, imag, conj</code>	módulo, argumento, parte real, parte imaginaria y conjugado

Así, por ejemplo:

```
> pi ^ (2*4)                realiza la operación pi^8;
> sqrt(4)                  notamos la estructura: función con argumentos entre paréntesis;
> acos(1) + asin(1)        es posible introducir espacios entre función y argumento;
> rand()                   o rand, algunas funciones pueden ir sin argumentos.
```

La mayor parte de las funciones predefinidas en Octave están diseñadas para actuar sobre escalares y, además, sobre matrices y vectores, en cuyo caso lo hacen componente a componente (véase el ejercicio §1.14). Esta información, y cualquier otra relevante acerca de las funciones predefinidas de Octave, podemos obtenerla mediante el comando `help` seguido de un espacio y el nombre de la función deseada. Por ejemplo, si solicitamos la ayuda de una función cualquiera `paquito` mediante el comando `help paquito`, obtendremos un mensaje (en inglés) similar al siguiente:

```
-- La funcion seleccionada: paquito (X)
calcula 'paquito' para cada elemento de X.
```

En otras ocasiones, dependiendo del tipo de argumento, un mismo comando actúa de distinto modo, como vimos, por ejemplo, en el apartado de vectores y matrices con los comandos `diag` o `poly`. En la ayuda de un comando no sólo encontraremos su funcionamiento sino que, a menudo, Octave proporciona una lista de comandos *afines*, lo que a menudo nos facilita la resolución del problema que estamos trabajando. En este sentido, destacamos algunas operaciones útiles que, vía `help`, nos pueden conducir a otras numerosas funciones relacionadas con **números enteros** o con **estadística**.

el comando...	calcula...
<code>gcd(a,b)</code>	el máximo común divisor de a y b
<code>lcm(a,b)</code>	el mínimo común múltiplo de a y b
<code>rem(a,b)</code>	el resto de dividir a entre b (es decir, a módulo b)
<code>round, ceil, floor, fix</code>	redondeo normal, por arriba, por abajo, cercano a 0
<code>sum(v), prod(v)</code>	suma y producto de las componentes del vector v
<code>cumsum(v), cumprod(v)</code>	la suma y el producto anteriores, pero acumulados
<code>mean, median, mode, std</code>	media aritmética, mediana, moda y desviación típica

Operar con matrices y vectores. En este apartado veremos cómo operar con matrices y vectores. Hemos de tener mucho cuidado con las operaciones **producto, división y potenciación**. Estas operaciones (**precedidas de un punto**: `.*`, `./`, `.^`) actúan sobre las matrices y los vectores **elemento a elemento**, esto es,

operación	devuelve...
<code>A.*B</code>	una matriz del mismo orden que $A = (a_{i,j})$ y $B = (b_{i,j})$ cuyos elementos vienen dados por $a_{i,j} * b_{i,j}$
<code>A./B</code>	igual que el anterior pero con elementos $a_{i,j}/b_{i,j}$
<code>A.^3</code>	una matriz del mismo orden que A con elementos $a_{i,j}^3$

Por otro lado, encontramos **operaciones puramente matriciales** o vectoriales. Los símbolos sin punto, (`*`, `\`, `^`) se usan para definir las siguientes operaciones matriciales. Esto es,

operación	devuelve...
$A * B$	el producto (matricial) de las matrices A y B
$A * v$	el producto (matriz \times vector) de la matriz A y el vector columna v
$A \setminus b$	la solución del sistema de ecuaciones lineales $Ax = b$ (es decir $A^{-1} * b$)
$A^{\wedge} p$	la potencia matricial de A

En el caso de las operaciones **suma y resta**, $(+, -)$, si ambos operandos son matrices del mismo orden se realizan las correspondientes operaciones matriciales. Sin embargo, si uno de los operandos es un número, entonces el escalar se suma o resta a cada uno de los elementos de la matriz.

Hemos de destacar que el comando $A \setminus b$, también escrito como `mldivide` (A, b) y llamado **división izquierda** (no confundir con el operador de división entre números, $/$), no realiza internamente el cálculo de A^{-1} . En cuanto a la operación potencia, $A^{\wedge} p$, en el caso en que p sea un número entero, calcula $|p|$ sucesivas multiplicaciones matriciales de A o A^{-1} , dependiendo del signo de p . Sin embargo, si p no es entero, realiza un desarrollo en términos de valores propios (que escapa del objetivo de este texto).

Otras funciones de marcado caracter matricial son:

operación	calcula...
<code>det</code> (A)	el determinante de la matriz A
<code>inv</code> (A)	la matriz inversa de la matriz A
A'	la matriz traspuesta conjugada de la matriz A
<code>transpose</code> (A)	la matriz traspuesta de la matriz A (equivalente a $A.'$)
<code>vec</code> (A)	un vector columna formado por las columnas de la matriz A
<code>fliplr</code> , <code>flipud</code>	invierten el orden de las columnas o las filas respectivamente
<code>compan</code> (p)	una matriz cuyo polinomio característico es p
<code>eig</code> (A)	los valores propios de la matriz A
<code>reshape</code> (A, m, n)	construye una matriz $m \times n$ a partir de A

En Octave también podemos encontrar muchas funciones, de caracter esencialmente vectorial, usadas habitualmente en matemáticas. Citamos algunas.

la orden...	calcula...
<code>max</code> (v)	el máximo (y su posición) de los elementos de v
<code>min</code> (v)	el mínimo (y su posición) de los elementos de v
<code>sort</code> (v)	los elementos de v ordenados de menor a mayor
<code>dot</code> (u, v)	el producto escalar de u y v
<code>norm</code> (v) o <code>norm</code> ($v, 2$)	la norma euclídea del vector v
<code>norm</code> ($v, 1$), <code>norm</code> (v, inf)	la norma 1 y la norma del máximo de v

Nótese que estos comandos **no actúan del mismo modo sobre matrices**. Por ejemplo, el comando `norm`, al actuar sobre matrices, calcula las respectivas normas matriciales subordinadas a las vectoriales.

1.1.3. Cómo guardar nuestro trabajo

Octave permite almacenar, total o parcialmente, tanto los comandos ejecutados en una sesión y sus salidas, como las variables creadas u otros diversos objetos que iremos describiendo. Antes de comentar distintas formas de guardar datos, para hacer uso de ellos en el futuro, hemos de tener claro dónde se van a crear los archivos en los que los guardamos.

¿Dónde guardamos los ficheros? Siempre que se arranque Octave tendremos un *directorio de trabajo activo*, es decir, un directorio donde el programa deposita los archivos que creamos y donde busca los comandos que invocamos (siempre y cuando estos no estén predefinidos). El **directorio de trabajo** se puede ver en todo momento mediante el comando `pwd` o en la ventana del GUI que estemos usando. Igualmente, podemos movernos por los directorios mediante `cd directorio` o usando dicha ventana de navegación. Es importante que en dicho directorio tengamos permisos de lectura y escritura, que es lo habitual si trabajamos en nuestro propio ordenador, pero no necesariamente si usamos máquinas públicas o de centros educativos con acceso regulado.

Una vez tenemos claro dónde se alojarán nuestros ficheros, veamos cómo guardar variables, sesiones completas o partes de ellas, salidas gráficas y, por último, una forma particular de crear/guardar sendos tipos particulares de ficheros: las *funciones* y los *scripts* que, por su especial relevancia y utilidad, comentaremos en una sección aparte.

Guardar sesiones. Si lo que queremos es guardar una copia de las instrucciones introducidas en la línea de comandos y los resultados obtenidos es necesario **comenzar la sesión**, o el momento a partir del cual queremos empezar a guardar, mediante el comando

```
> diary on
```

lo que iniciará el proceso de copia tanto de los comandos introducidos como de sus correspondientes salidas, y que no se detendrá hasta que ejecutemos el comando

```
> diary off
```

momento en el que se creará en el directorio de trabajo un archivo de texto denominado *diary.txt* conteniendo los datos anteriores. Esto nos permite, por ejemplo, guardar la resolución de ejercicios numéricos que se puedan plantear y que podamos ir resolviendo paso a paso con Octave.

Guardar variables. En ocasiones, queremos guardar los valores almacenados en variables que hemos construido durante una sesión, bien porque son demasiados y el uso

de la memoria empieza a ralentizar las operaciones, o bien para ser tratados de forma externa por otro programa o usarlos en el futuro. Para ello ejecutaremos

```
> save nombrefichero.mat nombre_variable  
> save nombrefichero.mat
```

con lo que crearemos un fichero de nombre “nombrefichero.mat” en el que almacenaremos, respectivamente, la variable `nombre_variable` o todas las de la sesión en curso. Posteriormente podemos volver a cargarlas en memoria para ser usadas mediante

```
> load nombrefichero.mat
```

Guardar salidas gráficas y animaciones. Octave permite mostrar y guardar objetos gráficos de manera vectorial (no pierde nitidez al ser reescalado, por lo que es muy útil para ser manipulado posteriormente), utilizando para ello *software* externo (*Gnuplot*, *Aqua*,...) y, aunque iremos presentando las distintas opciones para generar gráficas a lo largo del texto, aquí queremos mostrar la forma en que podemos guardar en un fichero vectorial cualquiera de esas gráficas creadas. Una vez que hayamos generado alguna gráfica con Octave, por ejemplo

```
> x = 0:0.1:pi;  
> plot(x, sin(x), "@")
```

esta aparecerá en una ventana aparte con el número 1 (la primera vez; si no se indica lo contrario, las sucesivas gráficas aparecerán en esta misma ventana, sustituyendo a la anterior). Si queremos guardar en un fichero pdf (por defecto, aunque admite otras extensiones) la última de las gráficas generadas en la ventana con el número 1, ejecutaremos:

```
> saveas(1, 'nombre-archivo.pdf');
```

siendo 'nombre-archivo.pdf' el nombre del archivo PDF en el que queremos guardar la salida gráfica. Para ver cómo guardar varias gráficas en un único fichero creando una animación, remitimos al ejemplo 9.10.

1.2. Creación de funciones y *scripts* en Octave

Como hemos mencionado, existen dos tipos particulares de ficheros muy útiles y relevantes para programar con Octave: las **funciones** y los ***scripts***. Dedicaremos esta sección a su creación, funcionamiento básico y utilidad.

Funciones. Desde un punto de vista informático, una **función** es un comando almacenado con un cierto *nombre* que realiza un cálculo o tarea determinada. La función, por lo general, necesita unos datos o *argumentos* sobre los que actuar, a partir de los cuales realizar el cálculo o tarea y, al ejecutarse, devuelve los resultados de su acción. Además de

las funciones que Octave tiene predefinidas, como hemos visto en la sección anterior, es posible implementar nuestras propias funciones, según nuestras necesidades. Hay varias formas para definir funciones y la elección de una u otra dependerá del uso que queramos darle y de la complejidad de las tareas que deba llevar a cabo. Hemos de recalcar que el concepto de función va mucho más allá del concepto matemático de función, cuya única tarea es la evaluación de una cierta fórmula matemática sobre una o más variables.

En general, la sintaxis para definir una función en Octave es la siguiente

```
function [resultados] = nombre_función (lista_argumentos)
    cuerpo: cálculos y tareas a realizar
endfunction
```

donde `lista_argumentos` es la lista de datos o argumentos, separados por comas, sobre los que la función actuará, `nombre_función` es la cadena de caracteres que nosotros asignamos al comando creado, y al cual **tendremos que apelar para ejecutarlo posteriormente**, y `cuerpo` constituye propiamente la función, y consiste en el conjunto de todos cálculos o tareas a realizar sobre los argumentos y que dan lugar a `resultados`, es decir, el listado de los valores, separados por comas, que devolverá como resultado la función. Es habitual en `cuerpo` dejar espacios a la izquierda para las líneas que lo componen (indentación), para lo cual usaremos siempre la barra espaciadora, **¡nunca el tabulador!** La lista `resultados` se puede omitir en aquellas funciones que no devuelvan valores o devuelvan sólo uno. Por último, `endfunction` determina que hemos concluido la definición de nuestra función. Por ejemplo, $f(x) = x^2 - 1 + e^x$ se puede definir **directamente en la línea de comandos** como sigue:

```
> function y = f(x) y = x.^2-1+exp(x); endfunction
```

Posteriormente podemos usar esta función para otros menesteres, como por ejemplo

```
> f(7)                                evaluarla en un punto (o en un vector).
> feval('f',7)                        evaluarla usando el comando feval.
> quad('f',0,1)                       integrarla entre 0 y 1 usando quad.
```

Más adelante, en el capítulo 8, trataremos la integración numérica y el comando `quad`. Remarcamos aquí que, aunque `f(7)` y `feval('f',7)` realizan la evaluación de la función, al ejecutar `f(7)` la función `f` tiene el **rol de comando**, mientras que al ejecutar `feval('f',7)` tiene el **rol de argumento**, y es llamada por la cadena de caracteres `'f'` que la define, es decir, entre comillas.

Precisamente en los casos en que necesitamos definir una función muy simple, sin demasiada relevancia en nuestro trabajo, y que va a ser usada básicamente como argumento de otro comando, es práctico utilizar la siguiente estructura:

```
nombre_de_la_función = @ (lista_de_argumentos) cuerpo
```

llamada **función anónima**. Con esta nueva estructura, podríamos repetir el ejemplo y cálculos anteriores mediante

```

> f = @(x) x.^2-1+exp(x);           creamos la función anónima,
> f(7)                               y la evaluamos,
> feval(f,7)                         o la evaluamos de otra forma (nótese la ausencia de comillas),
> quad(f,0,1)                       o la integramos,
> quad(@(x) x.^2-1+exp(x),0,1)       incluso sin definirla previamente.

```

Como hemos visto en estos ejemplos, la función creada es muy simple y consiste únicamente en evaluaciones de números. Sin embargo, en algunas ocasiones, las tareas que componen el cuerpo de una función son bastante más complejas e involucran varias líneas de código. En estos casos puede ser conveniente **guardar nuestra función en un archivo** “nombre_función.m” (**¡con el mismo nombre que la función!**) para poder usarla o editarla cuando la necesitemos. Para ello, y siguiendo el ejemplo dado, primero **creamos un fichero** que contenga el siguiente código (es habitual para crear ficheros usar un **editor de textos** que suele venir integrado con el GUI asociado):

```

1 function y = f(x)
2 % Comentarios que aparecerán en la ayuda de la función.
3 y = x.^2-1+exp(x); % otros comentarios
4 endfunction

```

y lo guardamos, en este caso, en un archivo denominado “f.m”. Vamos a destacar unos cuantos **detalles importantes en la construcción de funciones**:

- La primera palabra del fichero ha de ser `function` y la última `endfunction`.
- El código contenido en este fichero es texto plano.
- En cada línea, todo lo escrito tras un `%` se considera un **comentario**, es decir, un texto que Octave **no leerá**.
- En particular, los comentarios que coloquemos en las líneas inmediatamente posteriores a los argumentos (la segunda línea en el ejemplo anterior) serán los que aparezcan cuando pidamos a Octave **información sobre la función** creada mediante:

```

> help f
'f' is a function from the file /Users/juan/ejemplos/f.m
Comentarios que aparecerán en la ayuda de la función.

```

por lo que podemos incorporar cualquier información relevante que no queramos olvidar sobre cada función creada (sobre la fecha, el autor, su utilidad, etc.).

- Cada vez que intentemos ejecutar dicha función en un terminal de Octave, bastará con que el fichero que hemos creado esté colocado, bien en el **directorio de trabajo**, bien en las carpetas que Octave tenga predefinidas. Pueden consultarse los comandos `path`, `addpath`, `savepath`, `rmpath` o `restoredefaultpath` para manipular estos directorios.

- Las variables auxiliares que pudiesen definirse y usarse dentro del cuerpo de una función (construida mediante el comando `function`) se consideran **variables locales**, es decir, sólo existen dentro de la función y sólo cambian durante la evaluación de la función, aunque su nombre pueda coincidir con el de alguna variable que tengamos definida fuera. De hecho, tras ejecutar la función, ni se guarda ni altera el valor de ninguna otra variable. Para utilizar dentro del cuerpo de una función una variable que esté definida fuera (**variable global**) consúltase la sección 1.3.

Normalmente, el editor de textos nos ayudará en la labor de escritura, destacando las palabras propias de la sintaxis de Octave, como `function` o `endfunction`, indicándonos dónde se abren y cierran los paréntesis pareados, etc. No obstante, esta labor de editar y guardar ficheros se puede hacer mediante cualquier editor de texto plano. En lo sucesivo, los códigos contenidos en un archivo con extensión `.m` se presentarán enmarcados y con sus líneas numeradas, igual que el ejemplo anterior.

Scripts. Un *script* no es más que un archivo de texto con extensión `.m` que contiene comandos de Octave ordenados del mismo modo que los ejecutaríamos uno a uno en línea de comandos. De hecho, Octave lo lee como si cada una de sus líneas se estuviese escribiendo en la misma línea de comandos, y las variables que se definen dentro conservan su valor mientras la sesión no caduque o se eliminen mediante `clear`.

Un detalle importante es que, a diferencia de los archivos de funciones, **un script no puede empezar por `function`**, precisamente para que Octave no lo interprete como un archivo de tipo función¹. Ilustremos todo esto con un ejemplo, calcular la letra del DNI español para obtener el NIF.

Ejemplo 1.1. Primero hemos de conocer el proceso que queremos programar, en este caso, el mecanismo de obtención del NIF. Este proceso es simple: se calcula el resto de dividir el número del DNI entre 23 y al resultado (un número entero entre 0 y 22) se le asigna una letra mediante la siguiente tabla:

0 → T	1 → R	2 → W	3 → A	4 → G	5 → M	6 → Y	7 → F
8 → P	9 → D	10 → X	11 → B	12 → N	13 → J	14 → Z	15 → S
16 → Q	17 → V	18 → H	19 → L	20 → C	21 → K	22 → E	

que corresponderá a su letra del NIF. El siguiente *script* está creado para, al ser ejecutado, calcular el NIF asociado al DNI núm: 77 777 777, obteniendo la letra B.

```
1 % primero creamos una lista con las letras ordenadas
2 letras = 'TRWAGMYFPDXBNJZSQVHLCKE';
3 dni = 77777777; % introducimos el DNI al que calcular el NIF
4 n = rem(dni,23); % calculamos el resto de dividir entre 23
5 letras(n+1) % y extraemos la letra
```

¹Si casualmente queremos que la primera tarea de un *script* sea precisamente la definición de alguna función, basta con anteponer otra tarea inocua en la primera línea, como por ejemplo `1+1;`.

Sobre este ejemplo, observamos la utilidad de crear una función “nif.m” que calcule la letra de cada DNI sin tener que modificar cada vez el *script*. Sería la siguiente:

```
1 function y = nif(dni)
2 % Esta función calcula la letra del NIF para el DNI introducido
3   letras = 'TRWAGMYFPDXBNJZSQVHLCKE';
4   n = rem(dni,23)+1;
5   y = letras(n);
6 endfunction
```

Como vemos en estos ejemplos simples, es en el cuerpo de una función, o en el contenido de un *script*, donde realmente vamos a tener que aprender a programar los métodos numéricos objeto de estudio en este libro. Por ello, dedicaremos esta última parte de la introducción a Octave al uso de los comandos elementales de programación.

1.2.1. Primeros pasos en programación

En muchos algoritmos numéricos, y en la mayoría de los contenidos en este libro, aparecen **estructuras recurrentes** (cada paso depende del anterior) en las que es necesario verificar ciertas **condiciones** antes de avanzar en el algoritmo. Por ello, es imprescindible saber implementar con Octave tales estructuras recurrentes o **bucles** y dichas **condiciones**. Estos dos elementos son muy similares en cualquier lenguaje de programación; en esta introducción haremos una presentación muy simple de los mismos.

Operadores lógicos, condiciones. Para poder hablar de condicionales antes hemos de saber expresar *condiciones*. El tipo de condiciones que vamos a emplear en este libro se van a poder expresar mediante los operadores de comparación de valores:

menor	mayor	menor o igual	mayor o igual	igual	distinto
$x < y$	$x > y$	$x \leq y$	$x \geq y$	$x == y$	$x \neq y$ o $x \sim y$

Todos estos operadores devuelven 1 en caso de repuesta afirmativa y 0 en caso contrario. En el caso de comparación de matrices del mismo orden, comparan elemento a elemento y devuelven una matriz del mismo orden con ceros y unos. Si uno de los argumentos es un escalar, la comparación se hace entre dicho escalar y cada uno de los elementos de la matriz, y devuelve el resultado en una matriz. Veamos algunos ejemplos, constatando que sólo se pueden comparar matrices de igual tamaño:

```
> v = [0:4]; v == 3*ones(1,5)
ans = 0 0 0 1 0
> v < 3
ans = 1 1 1 0 0
> v == [7,9]
error: nonconformant arguments (op1 is 1x5, op2 is 1x2)
```

Las comparaciones también se pueden combinar empleando los operadores booleanos (véase el ejercicio §1.5):

disyunción “uno u otro”	unión “ambos a la vez”	negación “lo contrario”
	&	!

Por último indicamos una lista reducida de comandos que pueden ser de utilidad:

la orden...	devuelve...
<code>any(v)</code>	1 si alguna entrada del vector v no es cero; 0 si son todas ceros,
<code>all(v)</code>	1 si todas las entradas de v son no nulas; 0 si alguna es cero,
<code>find(v)</code>	las posiciones de las componentes no nulas del vector v ,

y que también admiten condiciones lógicas como argumentos, por ejemplo, `any(v==2)` devuelve 1 cuando alguna entrada del vector v es igual a 2, `all(v>5)` devuelve 1 cuando todas las entradas de v son mayores que 5, y `find(v==3)` devuelve las posiciones de las entradas del vector v que sean 3. Como aplicación directa de este último comando, podríamos, por ejemplo, seleccionar las componentes no nulas de un vector o matriz como sigue:

```
> posic = find(v); v(posic)
```

Condicionales `if` y `switch`. Combinando estos operadores lógicos construimos los argumentos (condiciones) de la sentencia condicional `if`, cuya sintaxis genérica es:

```
if (condición1)
    código1
elseif (condición2)
    código2
else
    código3
endif
```

Este comando, si la `condición1` resulta ser cierta (**una condición cierta no es más que una matriz o vector de unos**), ejecuta los comandos descritos en `código1`. Si no, chequea si `condición2` es cierta y, en caso afirmativo, ejecuta `código2`. Y en caso de que ninguna de las dos condiciones anteriores resulten ciertas, entonces se ejecutan los comandos contenidos en `código3`. Si cualquiera de las condiciones devuelve un vector o matriz, el comando `if` sólo tomará la condición como cierta si dicha condición es cierta sobre todas las componentes del vector. Hay que tener en cuenta que en la sentencia anterior se pueden introducir tantos bloques

```
elseif (condiciónN)
    códigoN
```


como se quiera (condicional múltiple) o bien no introducir ninguno (condicional doble). Caso de querer programar una instrucción condicional simple, es decir, que no realice ninguna acción si `condición1` no es cierta, hemos de suprimir tanto los bloques `elseif` como el último bloque `else`.

Ejemplo 1.2. Para practicar, vamos a definir la función a trozos

$$f(x) = \begin{cases} 5x & \text{si } x \leq 0, \\ 7x^2 & \text{si } x > 0. \end{cases}$$

y a guardarla en un fichero de tipo función llamado `atrozos.m`.

```
1 function y = atrozos(x)
2   if (x <= 0)
3       y = 5*x;
4   else
5       y = 7*x.^2;
6   endif
7 endfunction
```

y ya podremos evaluarla en distintos valores ejecutando simplemente

```
> atrozos(-4)
> atrozos(pi/2)
```

No obstante, hemos de advertir que esta función así definida no es del todo apropiada en un entorno como Octave. Por ejemplo, al evaluarla sobre el vector `x = [-1 2]` deseáramos obtener $f(-1)$ y $f(2)$, esto es, `ans = -5 28`; sin embargo, ocurre que

```
> x = [-2,2]; atrozos(x)
ans = 7 28
```

Esto se debe a que la condición `x<=0` produce la salida `0 1`, que no es cierta, por lo que la función asigna directamente el valor $7x^2$ en ambos casos. Para solventar este problema, podemos definir la función a trozos de forma **vectorizada**², para que al ser aplicada sobre vectores actúe de forma componente a componente.

```
1 function y = atrozosvect(x)
2   y = 5*x.*(x<=0)+7*x.^2.*(x>0);
3 endfunction
```

Y, ahora sí,

```
> atrozosvect(x)
ans = -5 28
```

²En la sección 1.3 ampliaremos el concepto de vectorización.

La segunda sentencia condicional es `switch`, cuya sintaxis genérica es la siguiente:

```
switch (pregunta)
  case (respuestas1)
    código1
  case (respuestas2)
    código2
  otherwise
    códigofinal
endswitch
```

Esta sentencia evalúa primero `pregunta` (que puede ser o no una condición lógica), cuyo resultado es un número escalar o una cadena de caracteres, después compara el resultado con los posibles casos contenidos en `respuestas1` y, si es alguno de ellos, ejecuta `código1`. Si no, compara el resultado con los posibles casos contenidos en `respuestas2` y, si coincide con alguno, ejecuta `código2`, y así sucesivamente. Si el resultado a `pregunta` no está contenido en ningún caso, ejecuta `códigofinal` que aparece tras `otherwise` (bloque opcional). Análogamente a `if`, se pueden introducir tantos bloques

```
case (respuestasN)
  códigoN
```

como se necesiten. Para ilustrarlo, veamos un ejemplo.

Ejemplo 1.3. Creamos una función que proporciona el número de días de un mes dado incorporando, en este caso, tanto una **advertencia** para el mes de febrero como un **mensaje final de error**.

```
1 function y = diasmes(x)
2 % Número de días del mes x-ésimo.
3 switch x
4   case ({1 3 5 7 8 10 12})
5     y = 31;
6   case {4,6,9,11}
7     y = 30;
8   case (2)
9     y = 28;
10    warning('29 si es bisiesto');
11    otherwise
12      error('x no parece ser un número entre 1 y 12');
13  endswitch
14 endfunction
```

Notamos que, cuando el conjunto de repuestas posibles descrito en un `case` tiene un único elemento (línea 8), lo podemos poner aislado o entre paréntesis, pero cuando

tiene más de un elemento (líneas 4 y 6), estos deben ir separados por espacios o comas y encerrados entre llaves (consúltese la parte dedicada a *arrays* de la sección 1.3). Notamos también que encerrar o no entre paréntesis este conjunto de respuestas es irrelevante y solo afecta a la apariencia, a la facilidad para visualizarlos como conjunto.

Estructuras repetitivas: bucles `for`, `while` y `do–until`. El primer **bucle** que vamos a ver es el comando `for`, de aplicación en aquellas situaciones en las que conocemos exactamente la cantidad de veces que necesitamos repetir un conjunto de sentencias. Su sintaxis queda reflejada mediante la siguiente estructura:

```
for variable = valores
    código
endfor
```

Aquí `código` representa el conjunto de instrucciones que se van a ejecutar para cada valor que toma `variable`, que juega el papel de contador. El conjunto de valores que va a tomar esta `variable` viene dado por `valores` que, *a priori*, puede ser una expresión cualquiera, aunque por lo general emplearemos un vector con números, de modo que los valores que va tomando la `variable` son las distintas componentes de ese vector.

En el siguiente ejemplo, calculamos la suma de los diez primeros términos de la sucesión $\{\frac{1}{2^n}\}$.

```
> suma = 0;
> for n = 1:10
> suma = suma+1/2^n;
> endfor
> suma
```

Iniciamos la variable `suma` con el valor 0;
la variable `n` recorre los valores 1,2,3,..., 10.;
en cada paso `suma` se incrementa en $1/2^n$;

pedimos el resultado.

Por supuesto, el comando `for` es uno de los más versátiles y empleados en programación y podemos usarlo también, como iremos viendo a lo largo de este libro, para construir matrices específicas, implementar algoritmos recursivos, etc.

Ejemplo 1.4. Ilustramos este bucle con la programación de la Ley d’Hont (electoral) que se utiliza para asignar de forma “proporcional” el número de representantes de cada partido tras unas elecciones. El funcionamiento básico es el siguiente: si llamamos (P_1, P_2, \dots, P_m) a la lista de m partidos que concurren a una elección en la que se eligen n representantes en total, y (x_1, x_2, \dots, x_m) al número de votos recibidos por cada partido respectivamente, se crea una tabla del tipo

partido	votos	votos/2	votos/3	...	votos/ n
P_1	x_1	$x_1/2$	$x_1/3$...	x_1/n
P_2	x_2	$x_2/2$	$x_2/3$...	x_2/n
\vdots	\vdots	\vdots	\vdots		\vdots
P_m	x_m	$x_m/2$	$x_m/3$...	x_m/n

A continuación se eligen ordenadamente de mayor a menor los n mayores valores de la tabla y se asignan al partido de la fila correspondiente un representante por cada uno de estos n valores. El caso de empate lo dejamos para el ejercicio §1.16, por lo que este programa básico (que no contempla empates) no debe ser usado en casos reales.

```

1 function [y,valor] = dhont(votos,n)
2 % Reparto proporcional de n escaños según la Ley d'Hont
3 % votos = vector fila conteniendo los votos de cada partido
4 % n = número de escaños a repartir
5 % y = número de escaños asignados a cada partido en el mismo
6 %   orden que aparecen en votos.
7 % valor = número de votos resultante para obtener un escaño.
8 % NOTA: votos ha de ser >=0
9 % Este programa básico NO contempla los casos de empates
10 a = 1./(1:n);
11 x = vec(votos'*a);
12 for i = 1:n
13     [valor,p] = max(x);
14     x(p) = -1;
15 endfor
16 m = length(votos);
17 x = reshape(x,m,n)';
18 x = (x<0);
19 y = sum(x);
20 endfunction

```

En este programa, tras construir la tabla³ x , hemos usado el bucle `for` para que, ordenadamente, sustituya el máximo de la tabla por -1 , de modo que al concluir sólo hemos de contar el número de -1 en cada fila (líneas 18 y 19). Además, hemos añadido una salida adicional, `valor`, que corresponde con el número final de votos que han sido necesarios para obtener un representante.

El segundo **bucle** que vamos a comentar es `while`, un bucle controlado por una **condición previa a la ejecución**. Una estructura del tipo

```

while (condición)
    código
endwhile

```

ejecutará los comandos recogidos en código **mientras que** la condición sea cierta. Este tipo de bucle es por tanto más apropiado para aquellos algoritmos en los que no sepamos *a priori* el número exacto de iteraciones que hemos de realizar, sino que el proceso se detiene en base a que se cumpla o no una cierta condición de parada. Para ilustrar este bucle, usamos la sucesión de Fibonacci y su relación con la *razón áurea* o *número de oro*: $\gamma = \frac{1+\sqrt{5}}{2}$.

³Hemos puesto la tabla en forma de vector mediante `vec` para que el comando `max` no actúe sobre cada columna. Posteriormente la reconstruimos mediante `reshape`.

Ejemplo 1.5. Una sucesión de Fibonacci $\{x_n\}$ es una secuencia de números generada de forma que cada uno se obtiene como suma de los dos anteriores, $x_{n+2} = x_{n+1} + x_n$. Si fijamos los dos primeros valores como $x_0 = 0$ y $x_1 = 1$, obtenemos

n	0	1	2	3	4	5	6	7	8	9	...
x_n	0	1	1	2	3	5	8	13	21	34	...

Esta sucesión aparece en diversos ámbitos: la espiral de Fibonacci, el triángulo de Pascal, el crecimiento de poblaciones, etc. En este caso vamos a usar que la *razón aurea* puede ser obtenida como límite del cociente de dos iteraciones sucesivas, es decir: $\gamma = \lim_{n \rightarrow \infty} \frac{x_{n+1}}{x_n}$, para cualquier par de números reales distintos x_0 y x_1 . Vamos a calcular, dando dos valores iniciales aleatorios, el primer valor n para el cual el cociente $\frac{x_{n+1}}{x_n}$ aproxima el valor $\gamma = \frac{1+\sqrt{5}}{2}$ con un error inferior a 10^{-6} , es decir, no pararemos mientras que se cumpla la condición $|\gamma - \frac{x_{n+1}}{x_n}| > 10^{-6}$.

```
> ga = (1+sqrt(5))/2; n = 1;           Iniciamos  $\gamma$  y el contador  $n$ ;
> x0 = rand(); x1 = rand();           iniciamos aleatoriamente  $x_0$  y  $x_1$ ;
> while(abs(ga-x1./x0) > 10^(-6))      mientras el error sea grande,
> n++; [x0,x1] = deal(x1,x1+x0); n      da un paso más y muestra  $n$ .
> endwhile
```

Resaltamos un comando habitual, `n++`, que hace aumentar el valor de n en una unidad, es decir, es equivalente a `n = n+1`.

Por último, presentamos el **bucle do-until**, muy similar a `while` salvo porque este realizará siempre, al menos, una iteración ya que tiene una **condición posterior a la ejecución**. Responde a una estructura del tipo:

```
do código
until (condición)
```

que ejecutará los comandos recogidos en **código hasta que** condición deje de ser cierta. El programa descrito en el ejemplo 1.5 podrá ahora ser escrito como

```
> ga = (1+sqrt(5))/2; n = 1;           iniciamos  $\gamma$  y el contador  $n$ ;
> x = rand(1,2);                       iniciamos aleatoriamente  $x = [x_0, x_1]$ ;
> do n++; x = [x(2), x(1)+x(2)]; n      da un paso y muestra  $n$ ,
> until(abs(ga-x(2)./x(1)) < 10^(-6))  hasta que el error sea pequeño.
```

Aquí hemos añadido un pequeño cambio en la construcción de las iteraciones: las hemos ido guardando en el vector `x`, no requiriendo el uso de `deal`.

Detención de un bucle. Podemos detener cualquiera de los bucles anteriores `while`, `for` o `do-until`, al igual que en C, C++ o Java, introduciendo en su código (no en

las condiciones) el comando `break`. Caso de tener bucles anidados (uno dentro de otro), dicho comando únicamente detiene el bucle más interno. Del mismo modo, la sentencia `continue` dentro de un bucle, concluye automáticamente la iteración actual, pasando a la siguiente, pero no detiene el bucle, que es lo que haría el `break`.

1.3. Algunos apuntes finales

Esta breve sección final está dedicada a algunos apuntes y comandos de interés que, aunque no se encuadran explícitamente en ninguno de los métodos matemáticos de este libro, pueden ser de utilidad en cualquier momento.

Los comandos `tic-toc`. En ocasiones, sobre todo para medir la efectividad de un comando o un *script* que supone una mejora respecto de otro, es útil conocer su **tiempo de ejecución**. Para saberlo simplemente ejecutamos `tic` justo antes del comando o *script* cuya duración de ejecución queremos conocer, para ejecutar `toc` justo después. La salida, independientemente de la producida por el comando o *script*, será el tiempo transcurrido desde que ejecutamos `tic` hasta que ejecutamos `toc`.

Variables locales y variables globales. Como hemos visto en la sección 1.2, las eventuales variables que usa el cuerpo de una función, se llaman **variables locales**, sólo existen dentro de la función y, aunque su nombre coincida con el de alguna variable existente fuera de la misma, esta no se ve alterada por la ejecución de dicha función. Pero, en ocasiones, precisamente se necesita que una función use en su cuerpo una variable cuyo valor proviene del “exterior” de la función. En este caso se pueden usar dos alternativas: o bien pasamos dicha variable como argumento a la función, o bien ejecutamos el comando `global` tanto al definir la variable requerida, como al usarla dentro de la función, lo que la convierte en una **variable global** y permite el uso deseado (podemos encontrar sendos ejemplos en las secciones 9.2.2 y 10.2.2).

Argumentos opcionales y valor por defecto. Como hemos visto en la sección 1.2, al definir una función debemos indicar cuales son los argumentos sobre los que actúa de manera que, para evaluarla, debemos proporcionarle los valores de cada uno de esos argumentos para que el cuerpo de la función los utilice y produzca una respuesta. En ocasiones, desharemos crear un programa mediante el comando `function` que dependa de un cierto argumento de forma **opcional**, es decir, que tome un **valor por defecto** dado cuando ejecutemos la función sin introducir dicho argumento. Esto es muy fácil de implementar, como mostramos en el siguiente ejemplo,

```
function y = f(x, n = 2)
    y = x.^n;
endfunction
```

que define una función de dos variables x y n , pero que si se ejecuta sin dar valor alguno a n , esta toma el valor por defecto $n = 2$:

```
> f(5, 3)                                da como salida  $125 = 5^3$ , pero
> f(5)                                    dará como salida  $25 = 5^2$ .
```

Sirvan también como ejemplos las funciones de la sección 8.2.3.

Vectorización de operaciones. A la hora de crear nuestros *scripts* y nuestras funciones para implementar algoritmos, a menudo usaremos alguno de los bucles presentados en la sección anterior para realizar operaciones recursivas y no hay lenguaje de programación que no posea estas estructuras tan necesarias. Pero en algunas ocasiones abusamos de estos bucles, perdiendo eficacia en nuestros códigos y desaprovechando la estructura vectorial que tiene Octave (y otros programas) para realizar algunas operaciones sin necesidad de usar estas estructuras recurrentes, disminuyendo la eficiencia del código resultante y haciéndolo más lento.

La vectorización, que pretende disminuir esta ineficiencia, no es una técnica concreta sino una idea general: *evitar el uso de bucles y condicionales cuando estos puedan ser sustituido por operaciones vectoriales*. Vamos a intentar ilustrar esta idea mediante un ejemplo sencillo.

Ejemplo 1.6. Crea un programa que convierta en 0 todas las componentes pares de un vector v definido previamente.

Un primera resolución *no vectorizada* sería

```
> n = length(v);                                definimos la longitud del vector;
> for k = 2:2:n v(k) = 0; endfor
```

es decir, observamos que el contador k ha de recorrer todas las componentes pares del vector, una a una. Sin embargo, la siguiente orden

```
> v(2:2:end)=0;
```

realiza la misma operación *vectorizada*, y emplea un tiempo (que podemos calcular usando `tic-toc`) entre 3 y 10 veces inferior, dependiendo de la máquina y de la longitud del vector.

A lo largo de este texto insistiremos en esta idea. Como caso sencillo podemos citar también el ejemplo 1.2 anteriormente presentado, donde se redefine la función `atrozos` de forma vectorizada, evitando el uso de condicionales, o, a un nivel más elevado, los ejercicios §3.33 y §6.63, y las secciones que involucra.

Arrays. En la sección 1.1.1 hemos presentado los objetos de Octave sobre los cuales se ejecutan las operaciones básicas. Pero existe otro tipo de objeto más complejo que, en ocasiones, es útil para manejar conceptos que requieren precisamente de esta complejidad. Además del ejemplo 1.3, en el capítulo 6 encontraremos un caso típico de objeto

complejo: las funciones *splines* o funciones polinómicas a trozos. Por su naturaleza, cada uno de estos *splines* requiere, para ser manejado, del troceado que se hace del intervalo, del polinomio que lo define en cada trozo, de las condiciones de unión, etc., es decir, de varios objetos de diferente naturaleza y utilidad, y que puedan ser manejados de forma conjunta para su posterior uso. Un *array* permite almacenar esta especie de **multiobjeto**, guardando los diferentes objetos que lo componen entre llaves y separados por comas. Veamos un ejemplo ilustrativo de cómo crear un array que contenga tres objetos diferentes (una cadena de caracteres, un vector y una matriz) y de cómo seleccionar cada uno de sus objetos internos.

```
> a = {"abcdefg", linspace(0,2,3) , [1,9;4,6]};
> a{2}
ans = 0 1 2
```

Algunos comandos de utilidad relacionados con los *arrays* son `isscalar`, `isvector` o `ismatrix`, que permiten discernir sobre la naturaleza de un objeto en cuestión.

1.3.1. Octave vs MatLab

Es conocido que el programa de pago *MatLab*® es altamente compatible con Octave, aunque no completamente. Hacemos un breve inciso en esta introducción para describir algunas diferencias que, aunque no son esenciales o son fáciles de resolver, pueden producir malfunciones de los códigos Octave en MatLab o viceversa. En particular, la adaptación de los programas que en este libro se proporcionan requeriría tener en cuenta al menos los siguientes ítems.

Funciones en MatLab; comando `inline`. En MatLab, las funciones definidas con el comando `function` no pueden ser introducidas directamente en la línea de comandos, sino sólo a través de un fichero `.m`. Para funciones sencillas es posible usar también las funciones anónima descritas por medio del operador `@()`, pero para funciones más complejas se puede usar el comando `inline`, que existe en Octave para buscar mayor compatibilidad con MatLab, pero que está en desuso. Para más detalles, véase [5, Functions and Scripts].

Finalización de comandos: `end-comando`. Hemos visto diversos comandos, como `function`, `for` o `while`, que en Octave se pueden terminar bien como hemos descrito, con `endfunction`, `endfor`, y `endwhile`, o simplemente usando `end`. En MatLab sólo puede ser usada esta última forma. En este texto hemos optado por usar `endfunction`, `endfor` y `endwhile`, así como otros similares, para ayudar, dentro de un código, a visualizar dónde acaba cada comando, aunque también usaremos un criterio estándar de tabulaciones para clarificar esta visualización.

Cadenas de caracteres. Las cadenas de caracteres en MatLab van escritas entre dobles comillas, "cadena", mientras que en Octave se pueden usar simples o dobles.

Para un análisis detallado de las diferencias entre ambos lenguajes recomendamos leer detenidamente [22].

1.3.2. Números de punto flotante vs números reales

Debido a la limitación de recursos de los ordenadores, la representación y el tratamiento que hacen de los números reales no es, en absoluto, fiel. En primer lugar hay que señalar que Octave trabaja internamente con los números reales con mucha más precisión de la que nos muestra en pantalla por defecto. De hecho, aunque nos muestre en pantalla sólo cuatro cifras decimales (que, además, están redondeadas), podemos cambiar esta forma de presentación mediante el comando `format`. Por ejemplo, tras ejecutar

```
> format long
```

Octave mostrará las salidas numéricas con 15 decimales. Mediante `help format` podemos ver otras posibilidades de formatos de salida.

Pero el verdadero problema que se pretende abordar en este punto es que, debido a la limitación de memoria de cualquier ordenador, internamente sólo se pueden representar un **conjunto finito de números** reales denominados **números de punto flotante**. En particular, el valor real positivo más pequeño que podemos representar con Octave (dependiente del sistema) es aproximadamente 2.2×10^{-308} y el mayor 1.8×10^{308} , cosa que podemos comprobar ejecutando `help realmin` y `help realmax`. Podemos constatar que, si escribimos números inferiores o superiores a estos, la respuesta es 0 o `Inf`, respectivamente. Además, esta cantidad finita de números representables provoca que un número real, x , y su representación en el ordenador, que denotaremos por $fl(x)$, no tengan necesariamente que coincidir, aunque sí se cumple que el error relativo cometido es pequeño, es decir,

$$\frac{|x - fl(x)|}{|fl(x)|} \leq \frac{1}{2}\epsilon,$$

donde ϵ es típicamente un valor pequeño que dependerá del punto x . En Octave se puede estimar el valor de ϵ mediante el comando `eps`⁴ o `eps(1)`, que es del orden de 10^{-16} en el caso particular de una máquina que soporte aritmética de tipo flotante con doble precisión, según el estándar IEEE (véase detalles en [24]).

Pero esta cantidad finita de números de punto flotante no sólo produce errores en la representación de los números, sino que además hace que **su aritmética no se corresponda con la de los números reales**, pudiendo generar errores aún mayores. Convenzámonos de estos hechos mediante algunos ejemplos.

⁴Concretamente `eps(x)` devuelve la distancia relativa entre los dos puntos flotantes consecutivos más próximos al número real x .

```

> 1-1+10^(-16)           produce 1.0000e-16, pero, si reordenamos operaciones,
> 1-(1+10^(-16))         la respuesta es 0.
> x = 10^(-15); (1+x)/x-1/x produce 1.1250, pero
> x = 10^(-15); ((1+x)-1)/x produce 1.1102.

```

El efecto de dichos errores, en un proceso numérico que involucre miles de operaciones, puede ser catastrófico. Sin entrar en detalles, notemos que al considerar números de órdenes muy distintos (es decir, el cociente entre el mayor y el menor excede 10^{16}) el menor de ellos (en valor absoluto) queda reducido a cero si se suman o restan.

A modo de ejemplo, citamos los problemas de inestabilidad numérica en las fórmulas de derivación descritos en la sección 8.2.1 o el ejemplo 3.4 del capítulo 3, donde veremos que un algoritmo teóricamente exacto se puede volver numéricamente inestable.

Capítulo 2

Resolución de ecuaciones no lineales escalares

2.1. Fundamentos teóricos

Nuestro propósito en este capítulo es resolver ecuaciones de la forma

$$f(x) = 0, \tag{2.1}$$

donde f es una función real de variable real que se anula en uno o varios puntos de su dominio. Llamaremos *raíces* de f a los valores reales de x para los que se anula dicha función, es decir, se verifica (2.1). A la hora de resolver una ecuación, pueden presentarse diversas situaciones:

- que no tenga soluciones reales (aunque sí complejas): $x^2 + 1 = 0$;
- que tenga una única solución real: $x + e^{2x} = 0$;
- que tenga soluciones reales múltiples: $(x + 1)^2(x - 2)x^3 = 0$

En ciertos casos, como el de los polinomios de grado menor o igual que 4, son conocidas fórmulas que permiten calcular de manera exacta sus raíces (reales o complejas). Sin embargo, no hay fórmulas para ecuaciones no lineales cualesquiera. De hecho, Niels Henrik Abel demostró que no es posible encontrar fórmulas generales (que involucren radicales) para todos los polinomios de grado prefijado mayor o igual que 5. Esta, y otras razones, nos llevan a la resolución numérica de ecuaciones no lineales.

2.1.1. Existencia y unicidad de soluciones

Empezamos con varios resultados que nos permitirán aplicar los algoritmos con cierta seguridad sobre la posibilidad de resolver el problema planteado en este capítulo.

Teorema 2.1 (Teorema de Bolzano: condiciones suficientes de existencia de solución).

- *Hipótesis:* Sea $f : [a, b] \rightarrow \mathbb{R}$ una función continua tal que $f(a)f(b) < 0$, esto es, que tome valores con distinto signo en a y b .
- *Tesis:* existe, al menos, un punto $c \in]a, b[$ tal que $f(c) = 0$.

Teorema 2.2 (Condición suficiente de unicidad de solución).

- *Hipótesis:* Sea $f : [a, b] \rightarrow \mathbb{R}$ estrictamente monótona.
- *Tesis:* $f(x) = 0$ admite a lo sumo una solución (real) en el intervalo $]a, b[$.

Veamos cómo podemos aplicar estos resultados en un par de ejemplos.

Ejemplo 2.3. Comprobemos que $f(x) = x + e^{2x}$ tiene una única raíz.

- $f(-1)f(0) = (-1 + \frac{1}{e^2}) \cdot 1 < 0 \Rightarrow$ gracias al teorema 2.1, existe, al menos, una solución de $f(x) = 0$ en el intervalo $] -1, 0[$.
- $f'(x) = 1 + 2e^{2x} > 0, \forall x \in \mathbb{R} \Rightarrow f(x)$ es estrictamente creciente en todo $\mathbb{R} \Rightarrow$ gracias al teorema 2.2, la solución de $f(x) = 0$ es única.

Ejemplo 2.4. $f(x) = e^x$ es estrictamente creciente pero no tiene raíces.

2.1.2. Método de bisección

Es el método más simple para buscar soluciones de una ecuación. Es consecuencia directa del teorema 2.1 (Teorema de Bolzano).

Algoritmo 2.5 (Bisección).

Supongamos que $f : [a, b] \rightarrow \mathbb{R}$ es una función continua tal que $f(a)f(b) < 0$.

- Tomamos $I_0 = [a_0, b_0] = [a, b]$ y definimos $c_0 = \frac{a_0 + b_0}{2}$.
- Entonces tenemos tres situaciones posibles:
 - si $f(b_0)f(c_0) = 0$, entonces c_0 es una raíz de f .
 - si $f(b_0)f(c_0) > 0 \Rightarrow$ tomamos $I_1 = [a_1, b_1] = [a_0, c_0]$ como nuevo intervalo.
 - si $f(b_0)f(c_0) < 0 \Rightarrow$ tomamos $I_1 = [a_1, b_1] = [c_0, b_0]$ como nuevo intervalo.
- Repetimos el proceso anterior y construimos la sucesión de intervalos I_2, I_3, I_4, \dots , hasta obtener
 - una solución exacta de $f(x) = 0$ (esto no será lo más frecuente);
 - una aproximación “suficientemente buena” de una solución exacta de $f(x) = 0$.

En la sección 2.3 veremos una implementación de este método y lo aplicaremos a la función del ejemplo 2.3.

Son varias las ventajas destacables del método de bisección. Observemos en primer lugar que es aplicable, sin problema ninguno, bajo hipótesis muy básicas. Además se verifica que, si fijamos *a priori* el máximo error absoluto permitido, podemos conocer

el número de iteraciones necesarias para aproximar una raíz. En efecto, si tenemos una función $f : [a, b] \rightarrow \mathbb{R}$ continua tal que $f(a)f(b) < 0$, entonces sabemos que existe $\alpha \in]a, b[$ una raíz de f .

- Consideramos $I_0 = [a_0, b_0] = [a, b]$ y definimos $c_0 = \frac{a_0+b_0}{2}$ como la aproximación inicial.

El error absoluto cometido está acotado por

$$|\alpha - c_0| < \frac{b_0 - a_0}{2} = \frac{b - a}{2}.$$

- En el siguiente paso de bisección el error absoluto cometido está acotado por

$$|\alpha - c_1| < \frac{b_1 - a_1}{2} = \frac{1}{2} \frac{b_0 - a_0}{2} = \frac{b - a}{2^2}.$$

- Al aplicar el n -ésimo paso, el error absoluto cometido está acotado por

$$|\alpha - c_n| < \frac{b_n - a_n}{2} = \frac{1}{2} \frac{b_{n-1} - a_{n-1}}{2} = \frac{1}{2} \frac{b - a}{2^n} = \frac{b - a}{2^{n+1}}.$$

Por consiguiente, si queremos estar seguros de que el error absoluto cometido sea menor o igual que cierto valor prefijado ε , entonces

$$\frac{b - a}{2^{n+1}} \leq \varepsilon \Leftrightarrow \frac{b - a}{\varepsilon} \leq 2^{n+1} \Leftrightarrow \ln(2^{n+1}) \geq \ln\left(\frac{b - a}{\varepsilon}\right) \Leftrightarrow n + 1 \geq \frac{\ln\left(\frac{b - a}{\varepsilon}\right)}{\ln(2)}. \quad (2.2)$$

Ejemplo 2.6. Siguiendo con el ejemplo 2.3, si queremos calcular la solución con un error absoluto menor o igual que 10^{-10} , partiendo del intervalo $[-1, 0]$, el número de iteraciones necesarias será

$$n + 1 \geq \frac{\ln\left(\frac{0 - (-1)}{10^{-10}}\right)}{\ln(2)} = \frac{10 \ln(10)}{\ln(2)} \approx 33.2 \Leftrightarrow n \geq 33,$$

es decir, con 33 iteraciones podemos asegurar que el error será menor o igual que 10^{-10} .

Como conclusión, podemos señalar que el método de bisección siempre converge, que permite tener localizada la raíz en todo momento, y que podemos determinar *a priori* el número de iteraciones necesarias para alcanzar el máximo error absoluto permitido. Sin embargo veremos que, en comparación con otros métodos, es un método lento.

2.1.3. Métodos de regula-falsi y secante

En el método de bisección sólo hemos usado el signo de la función en los extremos de los sucesivos intervalos. Ahora vamos a tener en cuenta los valores de la función en

dichos extremos para calcular las sucesivas iteraciones. Como podemos ver en la figura 2.1, la idea de estos nuevos métodos que vamos a presentar aquí consiste en calcular los sucesivos ceros de la rectas que pasan por los puntos $(a_n, f(a_n))$ y $(b_n, f(b_n))$, con $n = 0, 1, 2, \dots$, para regula-falsi y por los puntos $(x_n, f(x_n))$ y $(x_{n+1}, f(x_{n+1}))$, con $n = 0, 1, 2, \dots$, para secante.

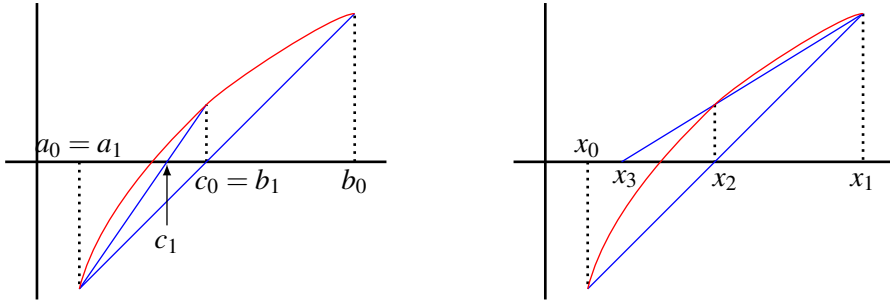


Figura 2.1: Métodos de regula-falsi (izquierda) y secante (derecha).

Método de regula- falsi. En el método de regula-falsi mantenemos la idea de tomar intervalos $I_n = [a_n, b_n]$ tales que $f(a_n)f(b_n) < 0$, pero ahora, en lugar del “punto medio”, se calcula el cero de la recta que pasa por los puntos $(a_n, f(a_n))$ y $(b_n, f(b_n))$.

Algoritmo 2.7 (Regula-falsi).

Supongamos que $f : [a, b] \rightarrow \mathbb{R}$ es una función continua tal que $f(a)f(b) < 0$.

- Tomamos $I_0 = [a_0, b_0] = [a, b]$ y definimos¹ $c_0 = \frac{a_0 f(b_0) - b_0 f(a_0)}{f(b_0) - f(a_0)}$.
- Entonces puede ocurrir que
 - si $f(b_0)f(c_0) = 0$, entonces c_0 es una raíz de f .
 - si $f(b_0)f(c_0) > 0 \Rightarrow$ tomamos $I_1 = [a_1, b_1] = [a_0, c_0]$ como nuevo intervalo.
 - si $f(b_0)f(c_0) < 0 \Rightarrow$ tomamos $I_1 = [a_1, b_1] = [c_0, b_0]$ como nuevo intervalo.
- Repetimos el proceso anterior y construimos una sucesión de intervalos I_2, I_3, I_4, \dots , hasta obtener
 - una solución exacta de $f(x) = 0$;
 - una aproximación “suficientemente buena” de una solución exacta de $f(x) = 0$.

Método de la secante. Al programar el método de regula-falsi, en cada paso debemos hacer una comprobación de signos. La idea del método de la secante es suprimir dicha comprobación de signos y usar siempre las dos últimas iteraciones calculadas. Para ello

¹El valor de c_0 se calcula a partir de la expresión que determina el cero de la recta que pasa por los puntos $(a_0, f(a_0))$ y $(b_0, f(b_0))$.

no determinaremos intervalos y, en su lugar, construiremos una sucesión de puntos a partir de dos puntos iniciales dados.

Algoritmo 2.8 (Secante).

Supongamos² que $f : [a, b] \rightarrow \mathbb{R}$ es una función continua tal que $f(a)f(b) < 0$.

- Tomamos $x_0 = a, x_1 = b$.
- Definimos la sucesión $\{x_2, x_3, x_4, x_5, \dots\}$ de forma iterativa mediante la fórmula

$$x_{n+2} = \frac{x_n f(x_{n+1}) - x_{n+1} f(x_n)}{f(x_{n+1}) - f(x_n)}, \quad n = 0, 1, 2, \dots,$$

o la expresión equivalente

$$x_{n+2} = x_{n+1} - \Delta x_n, \quad \text{donde } \Delta x_n = \frac{f(x_{n+1})(x_{n+1} - x_n)}{f(x_{n+1}) - f(x_n)}, \quad n = 0, 1, 2, \dots$$

Es de esperar que la sucesión $\{x_n\}$ esté bien definida y converja a una raíz de f .

Regula-falsi y secante: conclusiones y resultado sobre convergencia

El método de regula-falsi siempre converge a una raíz, la cual está localizada en cada iteración. Además, en general, es más rápido que bisección pero no podemos determinar, *a priori*, el número de iteraciones necesarias para alcanzar el máximo error absoluto permitido. Entonces, ¿cuándo paramos de hacer cuentas?

Por su parte, el método de la secante no siempre converge a una raíz y, si lo hace, no está localizada en cada iteración. Además, al igual que en regula-falsi, no podemos determinar, *a priori*, el número de iteraciones necesarias para alcanzar el máximo error absoluto permitido. Sin embargo, en general, es más rápido que bisección y menos costoso computacionalmente que regula-falsi.

Acabamos enunciando un resultado que garantiza la convergencia del método de la secante (véase [6]).

Teorema 2.9 (Convergencia del método de la secante).

- *Hipótesis:* sea $f : [a, b] \rightarrow \mathbb{R}$ tal que
 - es continua en $[a, b]$ con $f(a)f(b) < 0$;
 - es derivable en $]a, b[$ con $f'(x) \neq 0$ en $]a, b[$;
 - no cambia de convexidad³ en $[a, b]$.
- *Tesis:* el método de la secante converge a la única solución de la ecuación $f(x) = 0$.

²Esta condición nos garantiza la existencia de solución en el intervalo aunque, a diferencia de los casos anteriores, no es necesaria para hacer los cálculos.

³Si existe f'' , esto equivale a exigir que $f''(x)$ no cambie de signo en el intervalo $]a, b[$.

2.1.4. Método de Newton-Raphson

Gráficamente, en el método de la secante utilizamos rectas secantes a una curva. Por contra, si empleamos rectas tangentes, entonces obtenemos el método de Newton-Raphson. Esto es, vamos calculando sucesivamente los ceros de las rectas que pasan por los puntos $(x_n, f(x_n))$ con pendiente $f'(x_n)$, para $n = 0, 1, 2, \dots$. La idea gráfica del método puede verse en la figura 2.2.

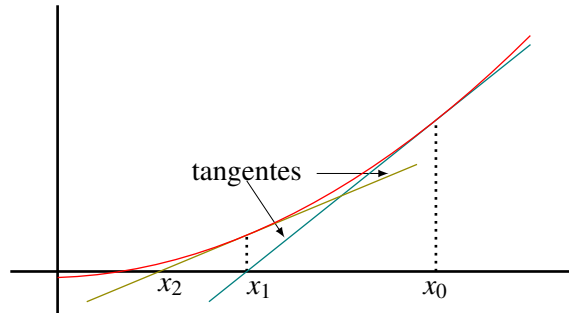


Figura 2.2: Método de Newton-Raphson.

Como en el caso de la secante, construiremos una sucesión de puntos, aunque en Newton-Raphson partimos de un único punto inicial.

Algoritmo 2.10 (Newton-Raphson).

Supongamos² que $f : [a, b] \rightarrow \mathbb{R}$ es una función derivable tal que $f(a)f(b) < 0$.

- Tomamos $x_0 \in [a, b]$ como aproximación inicial.
- Definimos la sucesión $\{x_1, x_2, x_3, x_4, \dots\}$ mediante la expresión

$$x_{n+1} = x_n - \Delta x_n, \text{ donde } \Delta x_n = \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \dots$$

Es de esperar que la sucesión $\{x_n\}$ exista y converja a una raíz de f .

Sobre el método de Newton-Raphson podemos comentar que, ni la sucesión $\{x_n\}$ ha de estar bien definida, ya que f' puede anularse, ni ha de converger a una solución, y que si lo hace, no la tendremos localizada (véanse los ejercicios §2.26 y §2.27). Tampoco podemos determinar, *a priori*, el número de iteraciones necesarias para alcanzar el máximo error absoluto permitido, por lo que tendremos que establecer un criterio de parada. Además, tenemos que calcular derivadas para realizar las sucesivas iteraciones. Sin embargo, cuando converge, es bastante más rápido que bisección, regla-falsi y secante.

Para acabar, vemos un resultado sobre la convergencia de este método que se puede consultar en [6].

Teorema 2.11 (Convergencia del método de Newton-Raphson).

- *Hipótesis:* Sea $f : [a, b] \rightarrow \mathbb{R}$ tal que
 - es continua en $[a, b]$ con $f(a)f(b) < 0$;
 - es derivable en $]a, b[$ con $f'(x) \neq 0$ en $]a, b[$;
 - no cambia de convexidad³ en $[a, b]$.
- *Tesis:*
 - La ecuación $f(x) = 0$ admite una única solución en $[a, b]$.
 - El método de Newton-Raphson es convergente si tomamos, como aproximación inicial, cualquier $x_0 \in]a, b[$ tal que $f(x_0)f''(x_0) > 0$.

El resultado de convergencia, en el caso $f(a) < 0$, está representado esquemáticamente en la figura 2.3.

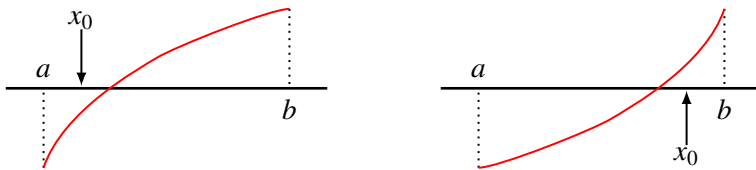


Figura 2.3: Convergencia en Newton-Raphson.

2.2. Localización de raíces de funciones en Octave

Para definir funciones con Octave usaremos tanto el comando `function` como funciones anónimas (véase la sección 1.2). El caso específico de las funciones polinómicas, las cuales se pueden representar de una manera mucho más cómoda y versátil, usando notación vectorial, se puede consultar en la sección 1.1.1. Ilustramos esta sección mediante la resolución de la ecuación $x^2 - 1 + e^x = 0$.

```
> function y = f(x) y = x.^2-1+exp(x); endfunction
```

Es conveniente que observemos el uso del comando “.” en lugar de “^”, para elevar al cuadrado. De esta forma “habilitamos” la función anterior para que sea evaluada sobre vectores, que es el mecanismo habitual de Octave. Ahora podemos trabajar con la función en el sentido que nos interese. Por ejemplo, para localizar gráficamente las posibles raíces podemos emplear cualquiera de los siguientes comandos:

```
> fplot('f', [-1, 1])
> x = -1:0.01:1; plot(x, f(x))
```

En estas situaciones suele ser de utilidad activar el mallado de la ventana gráfica mediante el comando `grid on`, obteniéndose así la gráfica de la figura 2.4. A partir de ella

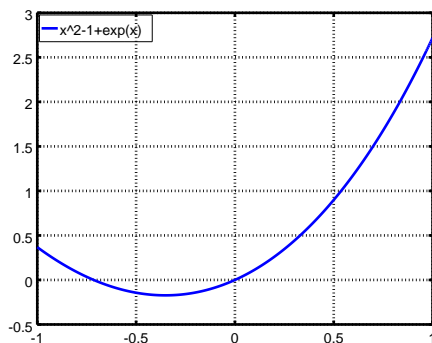


Figura 2.4: $f(x) = x^2 - 1 + e^x$ en el intervalo $[-1, 1]$.

podemos observar que hay dos raíces, y localizarlas en intervalos disjuntos:

```
> f(-1)
ans = 0.36788
> f(-0.1)
ans = -0.085163
> feval('f', 1)
ans = 2.7183
```

Por tanto, es fácil comprobar, aplicando el Teorema de Bolzano, que tiene una raíz en el intervalo $[-1, -0.1]$ y otra en $[-0.1, 1]$. Para calcular (de manera aproximada) la primera de ellas, podemos emplear el comando `fzero` de la siguiente forma.

```
> fzero('f', -0.8)
ans = -0.71456
```

Observemos que `fzero` requiere dos argumentos: el nombre de la función y un valor “cercano” a la raíz.

En los ejemplos anteriores hemos visto cómo debemos usar una función `function` como argumento de otro comando: tenemos que “llamarla” como una cadena de caracteres formada con su nombre entre comillas ('f', en los dos últimos casos).

Igualmente se puede trabajar con funciones anónimas, aunque no se emplearía la cadena de caracteres para pasarla como argumento, sino el nombre de la función, como ya se comentó en la sección 1.2. Veamos, empleando una definición anónima de la misma función, cómo localizar y aproximar la raíz que se encontraba en el intervalo $[-0.1, 1]$ (que obviamente es $x = 0$).

```
> g = @(x) x.^2-1+exp(x);
> x = -0.1:0.01:1; plot(x,g(x))
> g(-0.1)
> feval(g,1)
```

Finalmente, constatamos que mediante `fzero` obtenemos un valor aproximado de la raíz.

```
> fzero(g,0.1)
ans = -2.9412e-18
```

2.3. Implementación de algunos métodos

En esta sección presentamos implementaciones en Octave de los métodos de bisección, secante y Newton-Raphson. Pretendemos que la dificultad sea progresiva, de forma que la comprensión de un código sirva de ayuda para el siguiente.

2.3.1. Método de bisección

A continuación mostramos una implementación sencilla del algoritmo de bisección. Llevaremos a cabo este proceso en dos etapas. La primera corresponde a un *script* en el que aparecen los comandos necesarios para aproximar la raíz de la ecuación $x + e^{2x} = 0$ en el intervalo $[-1, 0]$ (véase el ejemplo 2.3). En la segunda desarrollamos una función aplicable a cualquier problema escalar para que, de esta forma, la rutina resultante esté disponible en nuestra biblioteca de métodos numéricos implementados en Octave para poder usarla en cualquier momento.

En ambos casos consideramos un criterio de parada basado en un número máximo de iteraciones que, tal y como vimos en la sección 2.1, determinaremos a partir de una cota superior para el máximo error absoluto cometido. Dicha cota vendrá preestablecida por el valor de la variable de tolerancia `tol`, según (2.2).

```
1 1;
2 % Declaración de datos
3 function y = fun(x)
4     y = x+exp(2*x);
5 endfunction
6 a = -1;
7 b = 0;
8 tol = 10^(-2);
9 % Iniciación de variables
10 maxIter = ceil((log(b-a)-log(tol))/log(2))-1;
11 n = 0;
12 c = (a+b)/2;
13 yc = fun(c);
```

```

14 fprintf('n          intervalo          c_n          f(c_n) \n')
15 fprintf('%i          [%f,%f]          %f          %f \n', n, a, b, c, yc)
16 ya = fun(a);
17 yb = fun(b);
18 % Bucle para n>=1
19 for n = 1:maxIter
20     if yc == 0 % La raíz es c
21         a = c;
22         b = c;
23         fprintf('%i          [%f,%f]          %f          %f \n', n, a, b, c, yc)
24         break
25     elseif yb*yc>0 % La raíz está en [a,c]
26         b = c;
27         yb = yc;
28     else % La raíz está en [c,b]
29         a = c;
30         ya = yc;
31     endif
32     c = (a+b)/2;
33     yc = fun(c);
34     fprintf('%i          [%f,%f]          %f          %f \n', n, a, b, c, yc)
35 endfor

```

Guardaremos este *script* en un archivo con el nombre “biseccionscript.m” y, para ejecutarlo, bastará con escribir en la línea de comandos el nombre del archivo (sin la extensión). Así, cuando ejecutamos el código

```
> biseccionscript
```

obtenemos una tabla de resultados similar a la siguiente,

n	intervalo	c_n	f(c_n)
0	[-1.000000,0.000000]	-0.500000	-0.132121
1	[-0.500000,0.000000]	-0.250000	0.356531
2	[-0.500000,-0.250000]	-0.375000	0.097367
3	[-0.500000,-0.375000]	-0.437500	-0.020638
4	[-0.437500,-0.375000]	-0.406250	0.037497
5	[-0.437500,-0.406250]	-0.421875	0.008220
6	[-0.437500,-0.421875]	-0.429688	-0.006261

En este momento podemos asegurar que $f(x)$ tiene una raíz α que dista de la última iteración calculada, -0.429688 , menos que 0.01 , es decir, $|\alpha - (-0.429688)| < 10^{-2}$. Por tanto hemos cometido un error menor que una centésima. Si quisiéramos conocer las dos primeras cifras decimales (exactas) de la raíz entonces tendríamos que realizar alguna iteración más.

Analicemos el código presentado con más detalle. Los comandos del *script* están agrupados en tres bloques: declaración de datos, iniciación de variables y, por último, el bucle que contiene las instrucciones del algoritmo 2.5 (para $n \geq 1$).

- En la declaración de datos se definen las funciones y los valores que necesitarán los comandos posteriores. Esto es,
 - la función `fun(x)`, de la que queremos calcular la raíz (líneas 3–5);
 - los valores `a` y `b` (líneas 6–7), que delimitan una raíz de `fun` (suponemos que `fun(a)` y `fun(b)` tienen signo distinto);
 - y un valor de tolerancia `tol` para el error absoluto (línea 8).

Como indicamos en la página 16, puesto que el primer dato es una función definida con el comando `function`, debemos añadir un comando neutro en la primera línea del *script* para que Octave no confunda este fichero con el de una función declarada con `function`. En nuestro caso hemos añadido el comando `1;`.

- En la iniciación de variables
 - definimos el número máximo de iteraciones, `maxIter` (línea 10), de acuerdo con la tolerancia indicada (ver fórmula (2.2)),
 - calculamos los valores iniciales `c0`, `fun(c0)` y asociamos los valores correspondientes a las variables `c` e `yc`, respectivamente (líneas 12 y 13),
 - mediante el comando `fprintf`⁴ imprimimos la cadena de caracteres que hace de cabecera de la tabla que se muestra cuando ejecutamos el *script* (línea 14) y la primera línea de la tabla, correspondiente a la iteración inicial (línea 15),
 - por último, guardamos los valores de la función en los extremos del intervalo con las variables `ya` e `yb` (líneas 16 y 17), que usaremos en la comprobación de cambio de signo para la asignación del intervalo en la siguiente iteración.
- En el bucle `for` se identifica en qué parte del intervalo queda el cambio de signo y se redefine el nuevo intervalo para la nueva iteración (líneas 25 a 30). El cálculo del punto medio y lo que sigue es análogo a lo realizado en la iniciación de variables.

A continuación adaptamos y simplificamos los comandos del *script* anterior para crear una función llamada `biseccion`, que guardaremos en un archivo con el nombre “biseccion.m” y en un directorio en el que Octave la pueda localizar, tal y como se explicó en la sección 1.2. Además, haremos que la función creada sólo devuelva información sobre la última iteración realizada.

⁴Podemos resaltar que, al emplear `fprintf` en las líneas 14, 15, 23 y 34, el comando `\n` realiza un retorno de carro. Además, `fprintf` puede contener especificadores de formato (al igual que los comandos de igual nombre de C++), los cuales nos permiten mostrar los distintos valores según sus características. En nuestro ejemplo, el primer especificador corresponde a un número entero (valor de la variable `n`, que está asociada a un contador) y, puesto que pretendemos que `fprintf` lo muestre como tal, incluimos `%i` (de *integer* en inglés). El resto de variables (esto es, `a`, `b`, `c`, `yc`) las mostramos en formato de punto flotante, para lo que usamos el especificador `%f`.

```

1 function [caprox,err,numiter] = biseccion(f,a,b,tol)
2 % Esta función aproxima una raíz c de la función continua f(x),
3 % localizada en [a,b], mediante el método de bisección.
4 % Datos necesarios para llamar a la función:
5 % f, expresión de f(x);
6 % a y b, extremos del intervalo donde sabemos que existe
7 % una raíz por el Teorema de Bolzano ya que f(a)*f(b)<0;
8 % tol, tolerancia máxima en la aproximación de la raíz,
9 % La función devuelve como respuesta tres números:
10 % caprox = valor aproximado de la raíz;
11 % err = |f(caprox)|;
12 % numiter = número de iteraciones realizadas.
13
14 % Iniciación de variables
15 n = 0;
16 c = (a+b)/2;
17 yc = feval(f,c);
18 maxIter = ceil((log(b-a)-log(tol))/log(2))-1;
19 ya = feval(f,a);
20 yb = feval(f,b);
21 % Bucle para n>=1
22 for n = 1:maxIter
23     if yc == 0 % La raíz es c
24         a = c;
25         b = c;
26         fprintf('Se ha alcanzado el cero exacto \n');
27         break
28     elseif yb*yc>0 % La raíz está en [a,c]
29         b = c;
30         yb = yc;
31     else % La raíz está en [c,b]
32         a = c;
33         ya = yc;
34     endif
35     c = (a+b)/2;
36     yc = feval(f,c);
37 endfor
38 % Definición de la respuesta
39 caprox = c; err = abs(yc); numiter = n;
40 endfunction

```

Veamos cómo se aplica la función recién definida en el ejemplo 2.3.

```

> function y = fun(x) y = x+exp(2*x); endfunction
> [cerol, valorf1, n1] = biseccion('fun',-1,0,10^(-6))
cerol = -0.42630
valorf1 = 2.9428e-07
n1 = 19

```

En el paso del *script* a la función, son significativas las siguientes diferencias.

- En el código de la función se añade, a continuación de la declaración de la función, una prolija **ayuda** que explica cómo se han de introducir los datos que requiere, así como la respuesta que va a proporcionar una vez ejecutada la función. Esta ayuda es útil para compartir los códigos o, incluso, si nosotros mismos prevemos usarlos en más de una ocasión ya que, aunque sea desconocida o la olvidemos, podemos acceder a ella mediante el comando `help`. Para ello recordamos que es preciso teclear en la línea de comandos la secuencia

```
> help biseccion
```

- El bloque de declaración de datos ha desaparecido, ya que dichos datos (f , a , b , tol) serán proporcionados como **argumentos** al usar la función.
- Cuando se pasa una **función como argumento** a otra función, se ha de tener en cuenta cómo se ha definido. Si es una función `function`, se ha de hacer mediante la cadena de caracteres con su nombre (entre comillas), mientras que si es una función anónima se emplea directamente el nombre. Esto implica que, para evaluarla y que funcione en cualquier caso, debemos emplear el comando `feval` (líneas 17, 19, 20 y 36), cosa innecesaria en el *script*.

Por otra parte, en una función, los resultados proporcionados están recogidos en unas pocas variables, las cuales resumen todo el proceso llevado a cabo. En este caso, hemos elegido los valores del valor aproximado, del error y de la última iteración calculados, desestimando los valores de las iteraciones intermedias. Por cierto, recordamos que si queremos ver estos tres valores (y no sólo el primero) al ejecutar la función, debemos declararlos al realizar la llamada a dicha función. Podemos comprobar todo esto ejecutando los siguientes comandos.

```
> biseccion('fun', -1, 0, 10^(-6))
ans = -0.42630
> [cerol] = biseccion('fun', -1, 0, 10^(-6))
cerol = -0.42630
> [cerol, valorf1] = biseccion('fun', -1, 0, 10^(-6))
cerol = -0.42630
valorf1 = 2.9428e-07
```

2.3.2. Método de la secante

El segundo de los métodos que vamos a programar en este capítulo es el de la secante. Son varios los comentarios que debemos tener en cuenta.

- Como *a priori* no sabemos el número de iteraciones que nos van a proporcionar una aproximación con una tolerancia prefijada, optamos por usar el bucle `while`.

- En la implementación tenemos que incluir algún criterio de parada, una condición que detenga el bucle. En esta ocasión optamos por un criterio de tolerancia sobre la diferencia entre iteraciones sucesivas, es decir, $|x_{k+1} - x_k| < tol$. También podríamos establecer un criterio de parada sobre el valor de la función, esto es, $|f(x_k)| < tol$ (véase el ejercicio §2.22).
- Además, como no tenemos asegurada la convergencia del método, fijaremos un número máximo de iteraciones. Si este criterio es el que provoca la finalización del bucle, indicaremos que no se ha alcanzado la precisión deseada.

Bajo estas premisas, presentamos la siguiente implementación del algoritmo 2.8 (método de la secante).

```

1 function [c,err,numiter] = secante(f,x0,x1,tol,maxIter)
2 % Esta función genera una aproximación de una raíz c de la
3 % función continua f(x) mediante el método de la secante.
4 % Datos necesarios para llamar a la función:
5 %   f, expresión de f(x);
6 %   x0 y x1, aproximaciones iniciales de la raíz;
7 %   tol, tolerancia máxima entre aproximaciones sucesivas,
8 %   es decir,  $|x_k - x_{kmenos1}| < tol$ ;
9 %   maxIter-1, número máximo de iteraciones a realizar.
10 % La función devuelve como respuesta tres valores:
11 %   c = valor aproximado de la raíz;
12 %   err =  $|f(c)|$ ;
13 %   numiter = número de iteraciones realizadas.
14
15 % Iniciación de variables
16 n = 1;
17 difsuc = x1-x0; % Diferencia entre iteraciones sucesivas
18 xnmenos1 = x0;
19 xn = x1;
20 fxnmenos1 = feval(f,xnmenos1);
21 fxn = feval(f,xn);
22 % Bucle
23 while abs(difsuc)>=tol && n<=maxIter
24     n = n+1;
25     difsuc = fxn*(xn-xnmenos1)/(fxn-fxnmenos1);
26     xnmenos1 = xn; fxnmenos1 = fxn; % Se guardan datos para
27     xn = xn-difsuc; % la próxima iteración
28     fxn = feval(f,xn);
29 endwhile
30 if n>maxIter
31     warning('Secante ha llegado al máximo de iteraciones \n');
32 endif
33 % Definición de respuestas
34 c = xn; err = abs(fxn); numiter = n-1;
35 endfunction

```


Veamos cómo funciona este código cuando lo aplicamos a la ecuación del ejemplo 2.3, aunque esta vez crearemos la ecuación mediante una función anónima.

```
> funAnon = @(x) x+exp(2*x);
> [cero2, error2, n2] = secante(funAnon,-1,0,10^(-6),10)
cero2 = -0.42630
valorf2 = 6.0230e-14
n2 = 6
```

2.3.3. Newton-Raphson

Finalizamos con una implementación del método de Newton-Raphson. Desde el punto de vista computacional, la función asociada a este método no presenta diferencias esenciales con respecto a la definida para la secante, salvo la necesidad de pasar como argumento la derivada de la función a la que se le busca raíz.

```
1 function [c,err,numiter] = newton(f,fdx,x0,tol,maxIter)
2 % Esta función genera una aproximación de la raíz c de la
3 % función derivable f(x) mediante el método Newton-Raphson.
4 % Datos necesarios para llamar a la función:
5 % f y fdx, expresiones de f(x) y f'(x);
6 % x0, aproximación inicial de la raíz c;
7 % tol, tolerancia máxima entre aproximaciones sucesivas;
8 % maxIter, número máximo de iteraciones a realizar.
9 % La función devuelve como respuesta tres valores:
10 % c = valor aproximado de la raíz;
11 % err = |f(c)|;
12 % numiter = número de iteraciones realizadas.
13 % Iniciación de variables
14 n = 0; xn = x0;
15 fxn = feval(f,xn);
16 fdxxn = feval(fdx,xn);
17 difsuc = fxn/fdxxn; % Diferencia entre iteraciones sucesivas
18 % Bucle
19 while abs(difsuc)>=tol && n<=maxIter
20     n = n+1;
21     xn = xn-difsuc;
22     fxn = feval(f,xn);
23     fdxxn = feval(fdx,xn);
24     difsuc = fxn/fdxxn; % Guarda datos para la próxima iteración
25 endwhile
26 if n>maxIter
27     warning('Newton-Raphson llegó al máximo de iteraciones \n');
28 endif
29 % Definición de respuestas
30 c = xn; err = abs(fxn); numiter = n;
31 endfunction
```

Como hemos dicho, si queremos aplicar este algoritmo al ejemplo 2.3, hay que pasar como argumentos tanto la función como su derivada a la hora de llamar al comando, lo que en este caso vamos a hacer directamente mediante funciones anónimas

```
> [cero3, valorf3, n3] = ...
  newton(@(x) x+exp(2*x), @(x) 1+2*exp(2*x), -1, 10^(-6), 10)
```

cuya salida es

```
cero3 = -0.42630
valorf3 = 1.6120e-10
n3 = 4
```

Comparativas. Los comentarios que hemos hecho en las secciones anteriores sobre la velocidad de convergencia de estos tres métodos se ven reflejados en su aplicación a la ecuación del ejemplo 2.3.

Para obtener un error menor o igual que 10^{-6} con bisección, secante y Newton-Raphson han sido necesarias $n_1=19$, $n_2=6$ y $n_3=4$ iteraciones respectivamente. Esta diferencia se refleja, en ejemplos más complejos, en diferencias relevantes en tiempo de computación (véase el ejercicio §2.23).

El resultado que nos devuelve Octave en los tres casos es, aparentemente, el mismo: $\text{cero1}=-0.42630$, $\text{cero2}=-0.42630$ y $\text{cero3}=-0.42630$. Sin embargo, si ejecutamos el comando `format long` (tal y como se comentó en la sección 1.3.2), podremos observar las diferencias existentes entre las aproximaciones:

```
> cero1
ceroBis = -0.426302909851074
> cero2
ceroSec = -0.426302751013404
> cero3
ceroNR = -0.426302750919851
```

Capítulo 3

Resolución de sistemas de ecuaciones lineales con métodos directos

3.1. Fundamentos teóricos

El objetivo de este capítulo es resolver sistemas de ecuaciones lineales (SEL) mediante métodos directos. La idea básica de los métodos directos que vamos a estudiar consiste en transformar el sistema original

$$Ax = b \tag{3.1}$$

en uno o varios sistemas equivalentes que sean más fáciles de resolver, por ejemplo los de la forma $Tx = c$, donde T es una matriz cuadrada triangular cualquiera. Los algoritmos que vamos a ver en este capítulo presupondrán siempre que A es una matriz cuadrada regular, por lo que (3.1) será un sistema compatible determinado. Para ver en qué sentido podemos “resolver” sistemas incompatibles o compatibles indeterminados, véase la sección 7.1.2.

En el resto de este capítulo, para fijar notación, denotaremos por a_{ij} (o $a_{i,j}$), con $1 \leq i, j \leq n$, a la entrada de una matriz cuadrada A , de orden n , correspondiente a la fila i y la columna j .

3.1.1. Resolución de sistemas triangulares

Antes de comenzar con la presentación de los métodos, recordemos que un SEL triangular, esto es, un SEL de la forma $Tx = c$ donde T es una matriz triangular (superior o inferior). Cuando T es regular se resuelve mediante una simple sustitución regresiva, si T es triangular superior, o sustitución progresiva, si T es triangular inferior. A continuación mostramos los algoritmos de estos dos procedimientos.

Algoritmo 3.1 (Sustitución regresiva).

Supongamos que T es una matriz triangular superior y regular, esto es, $t_{ij} = 0$ para $1 \leq j < i \leq n$, y $t_{ii} \neq 0$, para $1 \leq i \leq n$.

- De la última ecuación, despejamos la última incógnita: $x_n = c_n/t_{nn}$.
- De la penúltima ecuación, usando el valor de x_n ya obtenido, despejamos la penúltima incógnita:

$$x_{n-1} = \frac{c_{n-1} - t_{n-1,n}x_n}{t_{n-1,n-1}}.$$

- Y así regresivamente, para $i = n-2, n-3, \dots, 1$, de la i -ésima ecuación despejamos la i -ésima incógnita:

$$x_i = \frac{c_i - t_{i,i+1}x_{i+1} - t_{i,i+2}x_{i+2} - \dots - t_{i,n}x_n}{t_{ii}}. \quad (3.2)$$

Algoritmo 3.2 (Sustitución progresiva).

Supongamos que T es una matriz triangular inferior y regular, esto es, $t_{ij} = 0$ para $1 \leq i < j \leq n$, y $t_{ii} \neq 0$, para $1 \leq i \leq n$.

- De la primera ecuación, despejamos la primera incógnita: $x_1 = c_1/t_{11}$.
- De la segunda ecuación, usando el valor de x_1 ya calculado, despejamos la segunda incógnita:

$$x_2 = \frac{c_2 - t_{21}x_1}{t_{22}}.$$

- Y así de manera progresiva, para $i = 3, 4, \dots, n$, de la i -ésima ecuación, despejamos la i -ésima incógnita:

$$x_i = \frac{c_i - t_{i1}x_1 - t_{i2}x_2 - \dots - t_{i,i-1}x_{i-1}}{t_{ii}}. \quad (3.3)$$

En la sección 3.2.1 implementaremos ambos métodos y, además, los aplicaremos en los códigos de los métodos de resolución directa que veremos a continuación.

3.1.2. Método de Gauss

La idea del método de Gauss es, usando operaciones elementales por filas, transformar el sistema original en otro sistema equivalente que sea triangular superior.

Las operaciones elementales por filas son:

- permuta, o intercambio, de filas ($F_i \leftrightarrow F_j$).
- multiplicación de una fila por un escalar λ no nulo ($F_i \rightarrow \lambda F_i$).
- adición de filas ($F_i \rightarrow F_i + F_j$).

Así, realizando operaciones elementales, iremos haciendo ceros por debajo de la diagonal de la matriz ampliada del sistema $\bar{A} = (A|b)$.

Algoritmo 3.3 (Método de Gauss).

- Dado el sistema $Ax = b$ a resolver, construimos y actuamos sobre $\bar{A} = (A|b)$.
- Para $j = 1, 2, 3, \dots, n$:
 - **Pivoteo** (selección y colocación del pivote): buscamos el primer **elemento no nulo** de la j -ésima columna bajando a partir de la diagonal y llamamos $i_j \geq j$ a la fila en la que se encuentra dicho pivote.
 - Si $i_j > j$, intercambiamos la j -ésima fila con la i_j -ésima: $F_j \leftrightarrow F_{i_j}$.
 - Si $i_j = j$, no hacemos nada.
 - **Anulación de los elementos bajo la diagonal**: para $i \geq j + 1$, le restamos a la fila i -ésima la j -ésima fila multiplicada por $\frac{a_{ij}}{a_{jj}}$: $F_i \rightarrow F_i - \frac{a_{ij}}{a_{jj}}F_j$.
- Una vez terminado este proceso, la matriz (ampliada) obtenida es triangular superior y el SEL asociado es equivalente al original (esto es, tiene las mismas soluciones). Resolvemos dicho sistema por el algoritmo 3.1 de sustitución regresiva.

Inestabilidad del método de Gauss. El método de Gauss descrito presenta una gran inestabilidad frente a errores de redondeo. Veámoslo sobre un ejemplo.

Ejemplo 3.4. Consideremos el siguiente SEL,

$$\begin{pmatrix} 10^{-16} & -2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \end{pmatrix},$$

cuya solución exacta es $x = \frac{2}{2+10^{-16}}$, $y = 2 - \frac{2}{2+10^{-16}}$, valores que son aproximadamente iguales a la unidad. Como el primer elemento de la primera columna es distinto de cero, no hay intercambio de filas, y tenemos que multiplicar la primera fila por 10^{16} y restársela a la segunda, con lo que obtenemos la matriz ampliada

$$\left(\begin{array}{cc|c} 10^{-16} & -2 & -2 \\ 0 & 2 \times 10^{16} + 1 & 2 \times 10^{16} + 2 \end{array} \right).$$

Puesto que la máquina con la que vamos a resolver el sistema trabaja con precisión finita podría ocurrir que $fl(2 \times 10^{16} + 1) = fl(2 \times 10^{16} + 2) = fl(2 \times 10^{16})$, donde $fl(x)$ indica representación en punto flotante de x , tal y como se indicó en la sección 1.3.2. Debido a este redondeo, en ese caso la matriz realmente obtenida durante el cálculo sería

$$\left(\begin{array}{cc|c} 10^{-16} & -2 & -2 \\ 0 & 2 \times 10^{16} & 2 \times 10^{16} \end{array} \right).$$

De esta forma, obtenemos $x = 0$, $y = 1$ como solución numérica del sistema, que está muy lejos de ser la solución exacta. Podemos constatar que, efectivamente, Octave actúa

así ejecutando los comandos:

```
> Ab = [10^(-16), -2, -2; 1, 1, 2];
> Ab(2, :) = Ab(2, :) - Ab(1, :) / Ab(1, 1);
> [y, x] = deal(Ab(2, 3) / Ab(2, 2), (Ab(1, 3) - Ab(1, 2) * y) / Ab(1, 1))
y = 1
x = 0
```

Pero este no es un problema únicamente achacable a Octave, pues es común a cualquier cálculo siempre que se emplee aritmética discreta. Una buena forma de solventarlo es utilizar algún criterio en la selección del pivote que ayude a evitar estos errores; son las conocidas como **técnicas de pivoteo** (véase [4, 28]). Nosotros vamos a proponer, en la sección 3.2.2, la variación del método de Gauss con pivoteo parcial. Dicha estrategia consiste en elegir como pivote de cada columna al elemento de mayor valor absoluto entre los elementos que se encuentran en o por debajo de la diagonal.

3.1.3. Factorización de matrices

En esta sección planteamos la idea de escribir la matriz A factorizada como producto de otras dos matrices, esto es, $A = BC$. Dicha factorización nos permitirá resolver el sistema (3.1) en dos pasos sucesivos, ya que:

$$Ax = b \iff B(Cx) = b \iff By = b; \quad Cx = y.$$

La idea es, por tanto, encontrar factorizaciones para las que dichos sistemas sean más sencillos de resolver que el sistema original. Este es el espíritu de las factorizaciones que veremos a continuación, en las que B y C serán matrices triangulares u ortogonales¹. Veamos, en primer lugar, un resultado que nos proporciona las condiciones necesarias para obtener diversas factorizaciones de una matriz cuadrada A y que desarrollaremos a continuación. En él aparecen las submatrices principales A_k de una matriz A , que no son más que las submatrices cuadradas formadas por sus primeras k filas y k columnas.

Teorema 3.5. *Sea A una matriz real cuadrada de orden n , y A_k , para $k = 1, \dots, n$, sus submatrices principales. Se tienen los siguientes resultados:*

- **(Factorización LU)** Si $\det(A_k) \neq 0 \forall k$, entonces existen sendas matrices L y U , triangular inferior y superior respectivamente, tales que $A = LU$.
- **(Factorización PALU)** Existe una matriz P de permutación² y sendas matrices L y U , triangular inferior y superior respectivamente, tales que $PA = LU$.
- **(Factorización de Cholesky)** Si A es simétrica ($A = A^t$) y $\det(A_k) > 0 \forall k$, entonces existe una única matriz R , triangular superior con diagonal positiva, tal que $A = R^t R$.
- **(Factorización QR)** Existe una única matriz ortogonal Q y una única matriz triangular superior R tales que $A = QR$.

¹ Q es ortogonal si verifica $Q^{-1} = Q^t$, por lo que el sistema $Qy = b$ se resuelve como $y = Q^t b$.

²Una matriz de permutación no es más que la matriz identidad con sus filas permutadas.

Factorizaciones LU y PALU. La descomposición LU de una matriz A consiste en encontrar dos matrices, L triangular inferior y U triangular superior, tales que $A = LU$. En realidad (véase [25]), esta factorización es equivalente al método de Gauss sin realizar intercambio de filas, por lo que, del mismo modo que no siempre es posible aplicar Gauss sin permutar alguna fila (por ejemplo, si algún $a_{jj} = 0$), no siempre es posible realizar directamente la factorización LU de una matriz. Este hecho da una idea aproximada de las dos primeras descomposiciones descritas en el teorema 3.5, ya que si los menores principales $\det(A_k)$ son no nulos, los pivotes siempre se encuentran en la diagonal y no hacen falta permutaciones, mientras que, en el caso general, lo que podemos asegurar es que, haciendo ciertas permutaciones previas en la matriz A (esto es PA), se puede aplicar Gauss, y obtener la descomposición PALU, esto es $PA = LU$ (véase el ejercicio §3.39).

Para obtener un algoritmo que dé lugar a una descomposición $A = LU$ (sin permutaciones) basta con despejar en dicha igualdad componente a componente. Si fijamos los elementos de la diagonal de L o de U , esta factorización es única. Mostramos aquí cómo quedaría el algoritmo de factorización sin fijar ninguno de estos elementos, para barrer todos los casos posibles.

Algoritmo 3.6 (Factorización LU).

- Para $r = 1, \dots, n$;
 - $u_{rr} l_{rr} = \left(a_{rr} - \sum_{k=1}^{r-1} l_{rk} u_{kr} \right)$, (entre u_{rr} y l_{rr} se elige uno y se despeja el otro),
 - Para $s = r + 1, \dots, n$ se construyen:

$$u_{rs} = \frac{1}{l_{rr}} \left(a_{rs} - \sum_{k=1}^{r-1} l_{rk} u_{ks} \right), \quad l_{s,r} = \frac{1}{u_{r,r}} \left(a_{sr} - \sum_{k=1}^{r-1} l_{sk} u_{kr} \right).$$

Cuando se fijan los valores de la diagonal de L iguales a 1, $l_{ii} = 1$, la factorización se conoce como *factorización de Doolittle*, mientras que si se fijan los valores de la diagonal de U , $u_{ii} = 1$, entonces se conoce como *factorización de Crout*.

Factorización de Cholesky. En el caso en que A es una matriz simétrica, podemos buscar un tipo especial de Factorización LU, llamada *factorización de Cholesky*, en el que L sea la traspuesta de $U = R$, esto es, $A = LU = R^t R$. La condición descrita en el teorema 3.5 para la existencia de tal descomposición (también llamado criterio de Sylvester) equivale a que A sea **definida positiva**. En este caso, dejamos como ejercicio al lector el desarrollo del algoritmo correspondiente.

Factorización QR. Por último, la factorización QR consiste en descomponer la matriz A como producto de una matriz ortogonal Q por una matriz triangular superior R . Entonces, resolver (3.1) equivale a resolver el sistema triangular equivalente $Rx = Q^t b$ mediante sustituciones regresivas. Además, en el capítulo 5 emplearemos esta factorización en la implementación del método QR de cálculo de valores propios.

3.1.4. Condicionamiento de una matriz

Antes de continuar, debemos poner un “pero” a los métodos directos vistos. En efecto, aunque teóricamente todos ellos nos dan la solución exacta en un número finito de pasos, en la práctica los cálculos son realizados por un ordenador que, como ya hemos comentado alguna vez, comete “pequeños” errores. Es por esto que debemos tener en cuenta el efecto acumulado de estos errores en la resolución del sistema.

Si al resolver el sistema de ecuaciones $Ax = b$ cometemos errores de redondeo, en realidad obtenemos una **solución aproximada** $x + e$ (donde e es un vector conteniendo los errores cometidos). Por lo tanto, en realidad el sistema que hemos resuelto es

$$A(x + e) = b + Ae,$$

que se puede ver como una modificación de nuestro sistema original, donde se ha cambiado el término independiente b por el término perturbado $b + Ae$ (donde $r = Ae$ se denomina **residuo**). En primer lugar vamos a convencernos que hay algunos casos que pueden ser muy sensibles a modificaciones como las anteriormente citadas.

Ejemplo 3.7. Consideremos la matriz A y el vector u dados por

$$A = \begin{pmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & 1 \end{pmatrix}_{n \times n}, \quad u = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix},$$

y el SEL $Ax = b$ con vector de términos independientes arbitrario b . Veamos entonces que una pequeña modificación de b puede suponer que la solución del sistema varíe mucho. Concretamente, si tomamos como término independiente $b + \varepsilon u$, entonces e tendría que verificar la igualdad $Ae = \varepsilon e_n$, de donde $e = \varepsilon(2^{n-1}, 2^{n-2}, \dots, 2, 1, 1)^t$ (tras emplear sustitución regresiva), por lo que, para n suficientemente grande, el error e puede ser enorme aunque ε sea pequeño.

Esta **sensibilidad ante perturbaciones** de los datos, que es algo intrínseco del sistema a resolver, aparece en otras muchas situaciones. Por ejemplo, cuando calculamos aproximaciones por mínimos cuadrados, donde la matriz de coeficientes es una matriz de Hilbert ($H_{i,j} = 1/(i+j-1)$) (véase el ejercicio §7.72).

El siguiente resultado nos proporciona una herramienta que permite estimar el efecto de los errores de redondeo que se puedan cometer al resolver el sistema de ecuaciones $Ax = b$ por cualquier método de los vistos hasta el momento.

Teorema 3.8. Si A es regular, $Ax = b$ y $A(x + e) = b + r$ (es decir: $r = Ae$), entonces

$$\frac{\|e\|}{\|x\|} \leq v(A) \frac{\|r\|}{\|b\|}, \quad \text{siendo } v(A) = \|A\| \|A^{-1}\|.$$

Según este resultado, podemos considerar que la cantidad $\nu(A)$ nos da una idea de la sensibilidad del problema en sí frente a errores de redondeo, independientemente del método que empleemos para resolver el sistema. Al valor $\nu(A)$ se le denomina **condicionamiento de la matriz** A . Se puede probar que $\nu(A) \geq 1$, por lo que diremos que un sistema de ecuaciones está mejor condicionado cuanto más próximo a 1 esté dicho valor y, si $\nu(A)$ es “muy grande” diremos que es un sistema mal condicionado.

De manera similar se puede ver que el condicionamiento controla los errores al perturbar A , esto es, cuando resolvemos el sistema $(A + \delta A)(x + e) = b + \delta b$. Para el lector interesado, en [24] se deducen cotas para los errores cometidos en tal situación. En Octave el condicionamiento de una matriz se puede calcular mediante el comando `cond`.

Llegados a este punto nos podemos hacer la siguiente pregunta. Si, aún al aplicar métodos directos, finalmente vamos a obtener una aproximación de la solución, ¿por qué no buscamos directamente soluciones aproximadas? Esta cuestión motiva los contenidos del capítulo siguiente.

3.2. Implementación de algunos métodos

En esta sección vamos a presentar la implementación con Octave de algunos de los métodos descritos anteriormente.

Antes de comenzar, haremos un breve comentario sobre la resolución con Octave de sistemas de ecuaciones lineales. Para resolver el sistema compatible determinado $Ax = b$ podemos emplear el comando directo de Octave `mldivide(A, b)`, o simplemente,

```
> A \ b
```

Este comando, dependiendo del sistema en cuestión, seleccionará una estrategia “apropiada” para resolverlo, es decir, aplicará un determinado método numérico que será diferente según cada caso (véase [25, tema 5] para un análisis en detalle). No obstante, hay que tener en cuenta que este comando devuelve resultados aunque el sistema no sea compatible determinado; este punto será analizado y aclarado en la sección 7.1.2.

Aunque no sea un objetivo concreto de nuestro estudio, debemos advertir que, si el sistema no es compatible determinado, entonces podemos discutir su carácter mediante el comando `rref` aplicado a la matriz ampliada, que proporciona un sistema escalonado reducido por filas. Más allá de la aplicación directa de este comando (que siempre estará a nuestra disposición), el objetivo principal de este capítulo es que comprendamos la implementación de algunos de los métodos directos de resolución de sistemas de ecuaciones lineales.

Por último, comentar que podemos realizar la factorización LU, Cholesky y QR directamente con comandos ya implementados en Octave. En efecto, la sintaxis

```
> [L, U, P] = lu(A)
```

proporciona la factorización LU de la matriz PA , siendo P la matriz de permutación que

son necesarias sobre A para realizar esta operación (es decir, la factorización PALU de A). Por otra parte, la sintaxis

```
> R = chol(A)
```

da como respuesta la matriz triangular superior R de la descomposición de Cholesky, esto es $A = R^T R$, para el caso en que A sea una matriz simétrica y definida positiva.

Finalmente, podemos calcular la factorización QR directamente con el comando `qr`,

```
> [Q,R] = qr(A)
```

obteniendo una matriz ortogonal Q y una matriz triangular superior R tales que $A = QR$.

Para más detalles sobre estos comandos, consúltase [5, 18.3 Matrix Factorizations].

3.2.1. Sustitución regresiva y sustitución progresiva

Para empezar, vamos a programar los algoritmos 3.1 y 3.2 (sustitución regresiva y progresiva) para la resolución de sistemas triangulares. Observemos que, por el modo en que los hemos implementado, permiten resolver varios sistemas que tengan la misma matriz de coeficientes y distintos términos independientes, poniendo como argumento c la matriz cuyas columnas sean los distintos términos independientes (véanse los ejercicios §3.35 y §3.36).

```
1 function x = sustregr(T,c)
2 % Esta función resuelve el SEL  $Tx = c$ , con  $T$  triangular superior
3 % y regular, mediante sustituciones regresivas.
4 [dim,nc] = size(c);
5 x = zeros(dim,nc);
6 for i = dim:-1:1
7     x(i,:) = (c(i,:)-T(i,:)*x)/T(i,i);
8 endfor
9 endfunction
```

La implementación del método de sustitución progresiva se obtiene de forma sencilla a partir del código anterior.

```
1 function x = sustprogr(T,c)
2 % Esta función resuelve un SEL  $Tx = c$ , con  $T$  triangular inferior
3 % y regular, mediante sustituciones progresivas.
4 [dim,nc] = size(c);
5 x = zeros(dim,nc);
6 for i = 1:dim
7     x(i,:) = (c(i,:)-T(i,:)*x)/T(i,i);
8 endfor
9 endfunction
```

Observemos que los códigos anteriores son exactamente iguales, salvo en el sentido en que se recorre el índice i , descendente o ascendente (línea 6). De hecho, las líneas 7 de ambas funciones son idénticas cuando implementan las fórmulas (3.2) y (3.3), aunque en realidad dichas fórmulas son distintas. Esto es posible gracias a que la variable x es dinámica (esto es, en cada iteración cambia de valor) y a que se han usado expresiones vectorizadas.

Notemos que la variable x empieza siendo un vector/matriz de la misma dimensión que el término independiente con componentes nulas y que, en cada iteración, se cambia el valor de las componentes en una de sus filas. Por simplicidad, podemos pensar que el término independiente es un vector columna. En ese caso, la instrucción $T(i, :) * x$ multiplica la i -ésima fila de la matriz T por el vector columna x . En ese momento, el vector x tendrá, a lo sumo, $i - 1$ componentes no nulas. En la sustitución regresiva dichas componentes serán las últimas, mientras que en la progresiva serán las primeras. De esta forma comprobamos que, efectivamente, hemos implementando las fórmulas (3.2) y (3.3), respectivamente.

3.2.2. Método de Gauss

A continuación implementaremos el método de Gauss con estrategia de pivoteo parcial. Concluiremos el programa con la resolución del sistema triangular superior obtenido usando la función `sustreg` que hemos implementado en la sección anterior.

```

1 function [x,p] = gausspiv(A,b)
2 % Esta función resuelve el sistema Ax=b mediante Gauss.
3 % x, solución del SEL (vector/es columna).
4 % p, vector con las permutaciones realizadas en el proceso.
5 [dim,ncb] = size(b);
6 Ab = [A b];
7 p = 1:dim;
8 for j = 1:dim-1
9     % Pivoteo parcial dentro de la columna j
10    [piv,pos] = max(abs(Ab(j:dim,j)));
11    Ab([j,j+pos-1],:) = Ab([j+pos-1,j],:);
12    p([j,j+pos-1]) = p([j+pos-1,j]);
13    % Anulación de los elementos bajo el pivote
14    Ab(j+1:dim,j:dim+ncb) = Ab(j+1:dim,j:dim+ncb) - ...
15        Ab(j+1:dim,j) * Ab(j,j:dim+ncb) / Ab(j,j);
16 endfor
17 % Resolvemos por sustitución regresiva
18 x = sustregr(Ab(1:dim,1:dim),Ab(1:dim,dim+1:dim+ncb));
19 endfunction

```

En este código aparece el comando `max` que, cuando actúa sobre un vector, devuelve el valor máximo y la primera posición donde lo alcanza. Esta es, justamente, la información que requiere el proceso de pivoteo, ya que el valor máximo será el valor del pivote,

pero lo que se necesita para permutar las filas es, precisamente, la posición que dicho pivote ocupa. Conviene reseñar en este punto que `pos`, (esto es, la posición que devuelve `max`) es con respecto a la columna `abs(Ab(j:dim, j))`, que es la j -ésima columna de la matriz ampliada sin los primeros $j-1$ términos. Por este motivo la posición de dicho pivote, en la matriz completa, viene dada por `pos + j-1`. Además, para guardar un registro de las permutaciones realizadas, se define el vector `p`, con valores iniciales desde 1 hasta la dimensión de A , y se le aplican exactamente las mismas permutaciones que a la matriz.

Finalmente, observemos que, en el proceso de anulación de los elementos bajo la diagonal, el cálculo de las filas $\frac{a_{i,j}}{a_{j,j}}F_j$ se hace directamente (línea 15) para todo $i \geq j+1$, esto es, vectorizando (véase el ejercicio §3.33).

Ejemplo 3.9. Apliquemos este método al ejemplo 3.4. Primero definimos la matriz de coeficientes y el vector de términos independientes

```
> A = [10^-16 -2 ; 1 1];
> b = [-2 ; 2];
```

y aplicamos la función `gausspiv`

```
> [x,p] = gausspiv(A,b)
x = [1.00000 ; 1.00000]
p = [2 1]
```

Observamos que se ha realizado un intercambio de filas para colocar el pivote de máximo valor absoluto en la primera fila, lo que nos permite obtener una solución correcta del sistema. Recordemos que, como ya vimos en el ejemplo 3.4, sin intercambio de filas no obtenemos la solución correcta.

3.2.3. Factorización LU

Ahora desarrollaremos la implementación del algoritmo de factorización LU, aunque hemos de avisar que peca de los mismos defectos que el algoritmo de Gauss sin pivotaje: no existencia de descomposición como e inestabilidad numérica (véanse, por ejemplo, los ejercicios §3.38, §3.39 y §3.40). No obstante, como se indica en [28], los algoritmos de Gauss y LU, aún con pivote, presentan inestabilidades en casos concretos, aunque son aceptablemente estables en la práctica.

Ejemplo 3.10. El siguiente código implementa la descomposición LU (sin permuta de filas) y resuelve el sistema

$$\begin{pmatrix} 3 & 1 & 2 \\ 1 & 4 & 3 \\ 3 & 3 & 2 \end{pmatrix} x = \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}.$$

```

1 % Datos del ejemplo
2 A = [3 1 2 ; 1 4 3 ; 3 3 2];
3 b = [1 ; 3 ; 4];
4 % Realizamos la factorización LU de Doolittle de A
5 n = size(A) (1);
6 aux = zeros(n,n);
7 % Calculamos la primera fila de U y la primera columna de L
8 aux(1,1) = A(1,1);
9 if aux(1,1) == 0
10     error('No se puede realizar la factorización LU');
11 endif;
12 for s = 2:n
13     aux(1,s) = A(1,s);
14     aux(s,1) = A(s,1)/aux(1,1);
15 endfor
16 % Filas y columnas siguientes, partiendo desde la diagonal
17 for r = 2:n
18     aux(r,r) = A(r,r)-(aux(r,1:r-1)*aux(1:r-1,r));
19     if aux(r,r) == 0
20         error('No se puede realizar la factorización LU')
21     endif;
22     for s = r+1:n
23         aux(r,s) = A(r,s)-aux(r,1:r-1)*aux(1:r-1,s));
24         aux(s,r) = (A(s,r)-(aux(s,1:r-1)*aux(1:r-1,r)))/aux(r,r);
25     endfor
26 endfor
27 L = tril(aux,-1)+eye(n);
28 U = triu(aux);
29 % Resolvemos el sistema a partir de dicha descomposición
30 y = sustprogr(L,b);
31 x = sustregr(U,y)

```

En el ejercicio §3.41 analizaremos el algoritmo de Thomas, que no es más que la descomposición LU para el caso en que la matriz de coeficientes es tridiagonal³.

Por otra parte, ya que la implementación del algoritmo de Cholesky sigue una filosofía muy similar, dejamos al lector su realización.

³Matriz cuyas componentes fuera de las tres diagonales centrales son nulas.

Capítulo 4

Resolución de sistemas de ecuaciones lineales con métodos iterativos

4.1. Fundamentos teóricos

En general, los sistemas de ecuaciones lineales que aparecen en la práctica suelen ser de grandes dimensiones y con una matriz de coeficientes, A , que contiene muchos ceros (**matriz dispersa**). En tales casos, los métodos de resolución directa son costosos y no aprovechan adecuadamente esa ventaja, pues a lo largo del proceso muchos de los coeficientes nulos de A dejan de serlo. En este sentido, los métodos iterativos que estudiaremos en este capítulo sólo hacen uso de los elementos de la matriz original A .

La idea básica de los métodos iterativos para resolver sistemas de ecuaciones lineales es la misma que la empleada en los métodos vistos en el capítulo 2 de resolución de ecuaciones no lineales. Esto es, son métodos que consisten en calcular una sucesión de vectores $\{x^{(k)}\}_{k \geq 0}$ que converja a la solución del sistema. No obstante, esta idea es bastante razonable desde el momento en que hemos comprobado que, en la práctica, los errores de redondeo hacen que un método directo no nos devuelva la solución exacta.

4.1.1. Métodos iterativos clásicos: Jacobi, Gauss-Seidel y relajación

Dado un SEL

$$Ax = b, \quad (4.1)$$

un método iterativo básico consiste en, a partir de un vector inicial $x^{(0)}$ cualquiera, construir una sucesión de forma iterativa según una fórmula del tipo

$$x^{(k+1)} = Bx^{(k)} + c, \quad (4.2)$$

donde B es una matriz cuadrada del mismo orden que la matriz del SEL (4.1) y c un vector dado. Tanto B como c serán elegidos de forma que el método sea convergente y consistente, esto es, que exista el límite y que este límite resuelva el sistema (4.1):

$$\lim_{k \rightarrow \infty} x^{(k)} = x = A^{-1}b.$$

Podemos caracterizar estos métodos mediante el siguiente resultado (véase [1, 6]).

Teorema 4.1. *Sea el sistema (4.1) y un método iterativo de la forma (4.2).*

- **(Convergencia):** *Si existe una norma matricial inducida tal que $\|B\| < 1$, entonces el método es convergente¹: $x^{(k)} \xrightarrow{k \rightarrow \infty} x \in \mathbb{R}^n$ (para cualquier $x^{(0)}$).*
- **(Consistencia):** *Para que el límite x de (4.2) resuelva el sistema $Ax = b$, es preciso que existan sendas matrices M (regular) y N tales que:*

$$A = M - N, \quad \text{con} \quad M^{-1}N = B \text{ y } M^{-1}b = c.$$

De hecho, se cumple $A^{-1}b = x = Bx + c = M^{-1}Nx + M^{-1}b$.

En esta sección deduciremos los **métodos iterativos clásicos** que responden a este esquema general (procurando que la matriz M sea fácil de invertir: diagonal o triangular, por ejemplo), aunque los presentaremos desde un enfoque más próximo a la fórmula de punto fijo $Mx = Nx + b$, equivalente, y desde las propias ecuaciones, no desde la formulación matricial (que sin embargo es más útil para probar resultados de convergencia).

Método de Jacobi. Consideremos el SEL $Ax = b$, que desmenuzamos elemento a elemento, en la forma

$$\sum_{j=1}^n a_{ij}x_j = b_i, \quad i = 1, \dots, n.$$

Supongamos que todos los elementos de la **diagonal de A son no nulos**. Entonces, una opción sencilla para generar un método iterativo del tipo (4.2) sería despejar en cada ecuación su correspondiente incógnita. De este modo se obtiene el denominado *método iterativo de Jacobi*, cuya expresión componente a componente es

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j^{(k)} \right), \quad i = 1, \dots, n.$$

En la forma matricial descrita en el teorema 4.1, el método de Jacobi se corresponde con la elección

$$B = D^{-1}(L + U), \quad c = D^{-1}b,$$

siendo $A = M - N = D - L - U$, con D la matriz diagonal formada por los elementos de la diagonal principal de A , y L, U las matrices triangulares, con diagonal nula, formadas por la parte triangular inferior y superior de $-A$, respectivamente.

¹En realidad, la convergencia **equivale** a que $\rho(B) < 1$, esto es, que el **radio espectral** de la matriz B sea menor que 1. Recordemos que el radio espectral de una matriz es el máximo, en módulo, de sus valores propios. Para más detalles, véanse el capítulo 5 y el ejercicio §5.49

Método de Gauss-Seidel. Si en la iteración $k \mapsto (k+1)$ del método de Jacobi, para calcular cada componente $x_i^{(k+1)}$ usásemos las componentes previas (es decir, $x_j^{(k+1)}$ con $j < i$) ya calculadas, en lugar de las provenientes de la iteración anterior ($x_j^{(k)}$ con $j < i$), el método que obtenemos es el llamado *método de Gauss-Seidel*. Concretamente, en este nuevo método, las iteraciones son de la forma

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right), \quad i = 1, \dots, n.$$

En forma matricial, este método corresponde a tomar $M = D - L$ y $N = U$, esto es,

$$B = (D - L)^{-1}U, \quad c = (D - L)^{-1}b,$$

siendo D , L y U las mismas matrices que las definidas en el método de Jacobi.

Método de relajación. La idea del método de relajación es, a partir del método de Gauss-Seidel, introducir un parámetro $\omega \geq 0$ para “promediar con peso”, en cada paso, la iteración k y la $(k+1)$, obteniendo una combinación “convexa” entre ambas. Concretamente, el método se escribe como

$$x_i^{(k+1)} = \overbrace{\omega \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right)}^{\text{iteración } (k+1) \text{ de Gauss-Seidel}} + (1 - \omega) x_i^{(k)}, \quad i = 1, \dots, n.$$

La forma matricial del método corresponde a tomar $M = \frac{D}{\omega} - L$ y $N = A - M$, quedando

$$B = \left(\frac{D}{\omega} - L \right)^{-1} \left(U + \frac{1 - \omega}{\omega} D \right), \quad c = \left(\frac{D}{\omega} - L \right)^{-1} b,$$

siendo, una vez más, D , L y U las matrices definidas en el método de Jacobi.

Teoremas de convergencia. Presentamos a continuación dos resultados que garantizan la convergencia de los métodos anteriores (véanse más detalles en [1]).

Teorema 4.2. Si A es *estrictamente diagonal dominante*², entonces los métodos de Jacobi y de Gauss-Seidel convergen a la solución del sistema (4.1).

Teorema 4.3. Si A es *simétrica definida positiva* y $0 < \omega < 2$, entonces el método de relajación converge a la solución del sistema (4.1).

Los métodos iterativos clásicos de Jacobi y Gauss-Seidel adolecen de una lenta convergencia, lo que dará paso a la mejora que supone en este aspecto el método de relajación (véase el ejercicio §4.46). Sin embargo, hay que aclarar que, para problemas complicados, es difícil estimar el valor óptimo del parámetro de relajación, por lo que usualmente se prefieren otras opciones, entre las cuales destacaremos el **método del gradiente conjugado**, que presentamos en la siguiente sección.

²Lo es cuando A cumple, para cada fila i , que $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$.

4.1.2. Métodos de descenso: descenso rápido y gradiente conjugado

Cuando la matriz A es **simétrica y definida positiva**, entonces podemos aplicar los métodos de descenso, que se basan en el siguiente hecho: la forma cuadrática

$$\phi : \mathbb{R}^n \rightarrow \mathbb{R}, \quad \phi(u) = \frac{1}{2} \langle x^t, Ax \rangle - \langle b, x \rangle,$$

posee un **mínimo que coincide con la solución** de (4.1). De hecho, se pueden calcular³ $\nabla \phi(x) = (Ax - b)$ y $Hess(\phi(x)) = A$ (definida positiva), por lo que ϕ resulta ser estrictamente convexo, su mínimo⁴ es único y será un punto crítico y, por la expresión del gradiente, deducimos que resuelve el sistema $Ax = b$. Por ello, la idea básica de los algoritmos de descenso que permiten minimizar ϕ es:

“Dado un punto $x \in \mathbb{R}^n$, encontrar una dirección r sobre la que ϕ descienda y encontrar su mínimo sobre la recta $x + \langle r \rangle$.”

Así, a partir de una aproximación inicial $x^{(0)}$, construiremos iterativamente una sucesión $x^{(k)}$ de forma que, a partir de cada $x^{(k)}$, se elige una dirección r_k en la que se asegure un decrecimiento de ϕ y se toma $x^{(k+1)} = x^{(k)} + \alpha_k r_k$ tal que $\phi(x^{(k+1)}) < \phi(x^{(k)})$. Recordamos⁵ que una condición suficiente para que r_k sea una dirección de descenso en $x^{(k)}$ es que $\langle r_k, \nabla \phi(x^{(k)}) \rangle \neq 0$.

El esquema genérico para los algoritmos de descenso será el siguiente.

Algoritmo 4.4 (Métodos de descenso).

- Dada una aproximación inicial $x^{(0)} \in \mathbb{R}^n$.
- Para $k = 0, 1, 2, 3, \dots$, (iteración de $x^{(k)}$ a $x^{(k+1)}$):
 - Elegir la dirección r_k y deducir α_k para que $\varphi(\alpha) = \phi(x^{(k)} + \alpha r_k)$ sea mínimo.
 - Construir la iteración $x^{(k+1)} = x^{(k)} + \alpha_k r_k$.
 - Establecer un criterio de parada; si se verifica, parar, y si no, seguir.

Método de descenso rápido. El *método de descenso rápido* consiste en elegir como dirección de descenso la del gradiente cambiado de signo,

$$r_k = -\nabla \phi(x^{(k)}) = b - Ax^{(k)},$$

que es⁵ la dirección de máximo decrecimiento de ϕ en $x^{(k)}$. Ahora tenemos que determinar el paso α_k para que $x^{(k+1)}$ sea un mínimo de $\varphi(\alpha) = \phi(x^{(k)} + \alpha r_k)$ (nótese que φ es un polinomio de grado 2 en α). Haciendo un simple cálculo, podemos ver que

$$\alpha_k = \frac{\langle r_k, b - Ax^{(k)} \rangle}{\langle r_k, Ar_k \rangle} = \frac{\langle r_k, r_k \rangle}{\langle r_k, Ar_k \rangle}. \quad (4.3)$$

³Como es usual, ∇ denota al operador gradiente y $Hess$ al hessiano.

⁴En el caso en que A fuese simétrica pero definida negativa, x sería un máximo de ϕ , y los métodos serán aplicables tal cual, aunque serían métodos de “ascenso”.

⁵El gradiente de una función siempre indica la dirección de máximo crecimiento de la misma, y es perpendicular a la curva de nivel, esto es, la dirección en que ϕ no varía.

El vector de descenso obtenido, $r_k = b - Ax^{(k)}$, también se denomina **vector residuo** y su norma puede ser empleado en el criterio de parada del método.

Este método puede converger muy lentamente según la elección de la aproximación inicial $x^{(0)}$ (véanse [18] y el ejercicio §4.48) y de los subespacios propios de la matriz A . Para solventar este problema se pueden emplear diferentes modificaciones del método aquí expuesto. Quizás la más sencilla corresponde al denominado *método del gradiente conjugado*.

Método de gradiente conjugado. A continuación veremos el método del gradiente conjugado, donde se persigue exactamente el mismo fin que antes, pero eligiendo una dirección de descenso que, para medir perpendicularidad, use la métrica generada por la matriz A en lugar del producto escalar, lo que mejorará el resultado. La formalización matemática del anterior comentario nos lleva a la noción de direcciones A -conjugadas.

En primer lugar, observamos que, en el método del descenso rápido, las direcciones de descenso que se eligen son perpendiculares entre sí. En efecto, es fácil comprobar a partir de (4.3) que

$$\langle r_k, r_{k+1} \rangle = \langle r_k, b - Ax^{(k)} \rangle = \langle r_k, r_k \rangle - \alpha_k \langle r_k, Ar_k \rangle = 0.$$

De hecho, una vez escogida la aproximación inicial $x^{(0)}$, ya siempre nos moveremos en direcciones perpendiculares. Pues bien, la idea del método del gradiente conjugado es, calculado $x^{(k)}$, elegir una dirección de descenso $p_k \neq 0$ que, en lugar de ser perpendicular a las direcciones previas, sea A -conjugada con ellas, es decir, $\langle p_k, Ap_j \rangle = 0$, $k \neq j$.

El método del gradiente conjugado se concreta en el siguiente teorema.

Teorema 4.5 (Método de gradiente conjugado).

- *Hipótesis:* sean A , una matriz cuadrada de orden n simétrica y definida positiva, y $b \in \mathbb{R}^n$. Además, sean $x^{(0)} \in \mathbb{R}^n$, un vector cualquiera, $r_0 = b - Ax^{(0)}$, $p_0 = r_0$ y las sucesiones construidas de forma iterativa,

- $x^{(k+1)} = x^{(k)} + \alpha_k p_k$, donde $\alpha_k = \langle p_k, r_k \rangle / \langle p_k, Ap_k \rangle$;
- $r_{k+1} = b - Ax^{(k+1)}$;
- $p_{k+1} = r_{k+1} + \beta_k p_k$, donde $\beta_k = -\langle r_{k+1}, Ap_k \rangle / \langle p_k, Ap_k \rangle$.

para $k = 0, 1, 2, 3, \dots$ mientras que $r_{k+1} \neq 0$.

- *Tesis:* existe $\kappa \leq n$ tal que $Ax^{(\kappa)} = b$. Además, para todo $k \leq \kappa - 1$, se verifica que
 - la dirección p_{k+1} es A -conjugada con todas las direcciones p_j previas, es decir, $\langle p_{k+1}, Ap_j \rangle = 0$ para todo $j \leq k$;
 - el residuo r_{k+1} es ortogonal a todos los residuos r_j previamente calculados, esto es, $\langle r_{k+1}, r_j \rangle = 0$ para todo $j \leq k$;
 - $\langle p_k, r_k \rangle = \langle r_k, r_k \rangle$.

Observemos que $x^{(k+1)}$ se propone para que sea mínimo en la recta $x^{(k)} + \langle p_k \rangle$ y que, además, p_{k+1} se busca como una combinación lineal del residuo r_{k+1} y p_k para que sea A -conjugado con este último. Después, como regalo, obtenemos que es A -conjugado con todos los anteriores.

Del teorema se deduce que $r_k = b - Ax^{(k)} = 0$, es decir, teóricamente, el algoritmo determinará la solución exacta del SEL $Ax = b$ en, a lo sumo, n iteraciones. Este resultado, totalmente válido en aritmética exacta, nos podría llevar a definir el método del gradiente conjugado como un **método directo**, aunque a efectos prácticos es conveniente verlo como un **método iterativo** por dos factores esenciales.

- El primer motivo es que suele alcanzar aproximaciones muy eficientes bastante antes de llegar a la última iteración. Esto es debido a que al elegir, en cada paso, las direcciones A -conjugadas, se minimiza sobre un espacio mayor que la recta $x^{(k)} + \langle p_k \rangle$, dependiendo de las dimensiones de los subespacios propios de la matriz A . En este sentido, se puede establecer un criterio de parada atendiendo a la norma del residuo.
- Por otro lado, es posible que, en las sucesivas iteraciones, tanto los errores debidos a la aritmética de punto flotante como la aparición de residuos extremadamente pequeños puedan alterar el resultado exacto. En particular, la aparición de denominadores casi nulos en el cálculo de los coeficientes α_k y β_k , puede dar lugar a “rupturas” del método, que son evitables mediante versiones estabilizadas (véase [24]).

4.2. Implementación de algunos métodos

En esta sección, amén de la resolución directa con `A\b` o `mldivide(A,b)`, veremos la implementación en Octave del método de Gauss-Seidel y del método del gradiente conjugado. La implementación del resto de métodos aquí vistos y de versiones optimizadas quedarán como ejercicio para el lector.

Observemos que, tal y como hemos descrito ambos métodos, además de la matriz de coeficientes A y el vector de términos independientes b , necesitamos una variable de entrada extra que corresponde al valor de la aproximación inicial $x^{(0)}$. En los códigos siguientes nosotros hemos optado por introducir este valor inicial como argumento, para poder modificarlo a discreción en algunos ejercicios (véas), aunque podría ser introducido como un vector aleatorio dentro del cuerpo de la función.

4.2.1. Método de Gauss-Seidel

La convergencia de este método no está asegurada *a priori*, por lo que introduciremos criterios de parada. En concreto, en el siguiente código se implementa un criterio de parada doble. Por un lado, fijamos un número máximo de iteraciones y, por otro, establecemos un criterio de parada sobre la norma del residuo. Este segundo criterio es del tipo

$$\frac{\|Ax^{(n)} - b\|}{\|b\|} \leq tol.$$

```

1 function [x,iter] = gaussseidel(A,b,x0,tol = 10^-12,maxiter = 25)
2 % Esta función calcula una solución aproximada del sistema
3 % A x = b mediante el método de Gauss-Seidel.
4 % Datos necesarios para llamar a la función:
5 %   A, matriz de coeficientes del SEL (regular).
6 %   b, vector de términos independientes (vector columna).
7 %   x0, aproximación inicial de la raíz (vector columna).
8 %   tol, tolerancia preestablecida sobre la norma del residuo.
9 %   maxiter, número máximo de iteraciones del método.
10 % La función devuelve como respuesta:
11 %   x, solución aproximada del SEL.
12 %   iter, número de iteraciones realizadas para obtener x
13 %   (si coincide con maxiter es que x no verifica la condición
14 %   de tolerancia impuesta).
15 %
16 n = length(A);
17 x = x0; % se inicializa x
18 normb = norm(b);
19 iter = 0;
20 while (norm(A*x-b)>tol*normb) && (iter<maxiter)
21     iter++;
22     for j=1:n
23         x(j) = (b(j)-A(j,[1:j-1,j+1:n])*x([1:j-1,j+1:n]))/A(j,j);
24     endfor
25 endwhile
26 endfunction

```

En este código observamos como, efectivamente, en cada paso las componentes ya calculadas del vector x se usan en el cálculo de las componentes siguientes. En [24] se puede encontrar un código que, siguiendo la estructura matricial (4.2), implementa entre otros este mismo algoritmo.

A continuación veamos cómo se aplicaría, en un ejemplo concreto, la función que acabamos de crear.

Ejemplo 4.6. Consideremos el sistema de ecuaciones

$$\left. \begin{array}{rrcr} 4x & - & y & + & 2z & = & 0 \\ -2x & - & 8y & + & z & = & 3 \\ x & + & 3y & + & 5z & = & 9 \end{array} \right\}.$$

Para resolverlo simplemente ejecutaríamos los siguientes comandos.

```

> A = [4,-1,2;-2,-8,1;1,3,5];
> b = [0;3;9];
> x0 = [0;0;0];
> gaussseidel(A,b,x0)
ans = [-0.93855;0.10056;1.92737]

```

Si queremos obtener el número de iteraciones realizadas, deberemos ejecutar la sentencia

```
> [x,iter] = gaussseidel(A,b,x0)
x = [-0.93855;0.10056;1.92737]
iter = 12
```

4.2.2. Método del gradiente conjugado

Para terminar el capítulo implementaremos el método del gradiente conjugado, en su versión básica a partir del Teorema 4.5. Observemos que, para nuestro propósito, necesitamos tres entradas: la matriz de coeficientes A (que debe ser simétrica y definida positiva o negativa), el vector de términos independientes b y el valor de la aproximación inicial $x^{(0)}$. Además, en este caso, el número de iteraciones máximo, para establecer el criterio de parada, se fija en dos veces la dimensión de A (y no se introduce como argumento de la función).

```
1 function [x,iter] = gradconj(A,b,x0,tol = 10^-5)
2 % Esta función calcula una solución aproximada del sistema
3 % A x = b mediante el método del gradiente conjugado.
4 % Datos necesarios para llamar a la función:
5 % A, matriz de coeficientes (simétrica y definida positiva o
6 % negativa).
7 % b, vector de términos independientes (vector columna).
8 % x0, aproximación inicial de la raíz (vector columna).
9 % tol, tolerancia preestablecida sobre el módulo del residuo.
10 % La función devuelve como respuesta:
11 % x, solución aproximada del SEL.
12 % iter, número de iteraciones realizadas para obtener x.
13 %
14 n = length(A);
15 x = x0;
16 r = b-A*x;
17 p = r;
18 iter = 0;
19 while (norm(r)>tol) && (iter<2*n)
20     iter++;
21     alpha = (p'*r)/(p'*(A*p));
22     x = x+alpha*p;
23     r = b-A*x;
24     beta = -(r'*(A*p))/(p'*(A*p));
25     p = r+beta*p;
26 endwhile
27 endfunction
```

Observemos que, tanto en `gaussseidel.m` como en `gradconj`, no se verifican las hipótesis que han de satisfacer las matrices de coeficientes.

Ejemplo 4.7. Vamos a aplicar este método al sistema $Ax = b$, con

$$A = \begin{pmatrix} 1.8 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.3 & -0.2 & 1.3 & -0.2 \\ 0.2 & -0.2 & 1.3 & -0.2 & 0.3 \\ 0.2 & 1.3 & -0.2 & 0.3 & -0.2 \\ 0.2 & -0.2 & 0.3 & -0.2 & 1.3 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ -2 \\ 5 \\ 0 \\ 3 \end{pmatrix}.$$

Para ello ejecutamos los siguientes comandos.

```
> A = [1.8, 0.2, 0.2, 0.2, 0.2;
      0.2, 0.3, -0.2, 1.3, -0.2;
      0.2, -0.2, 1.3, -0.2, 0.3;
      0.2, 1.3, -0.2, 0.3, -0.2;
      0.2, -0.2, 0.3, -0.2, 1.3];
> b = [1; -2; 5; 0; 3];
> x0 = [0; 0; 0; 0; 0];
> [x, iter] = gradconj(A, b, x0)
```

La respuesta obtenida es

```
x = [2.2204e-16; 1.0000e+00; 3.5000e+00; -1.0000e+00;
     1.5000e+00]
iter = 3
```

Observamos que la solución se alcanza en sólo 3 iteraciones, a pesar de que el sistema es de orden 5. Esto se debe a que, como hemos comentado, el método de gradiente conjugado optimiza ϕ , en cada paso, no sólo en una recta sino en todo un subespacio mayor que aumenta con las dimensiones de los subespacios propios de la matriz A y, en este caso, la matriz A tiene dos subespacios propios dobles, correspondientes a dos valores propios dobles ($\lambda = 2$ y $\lambda = 1$), mientras que el otro subespacio es simple, y asociado al valor propio simple $\lambda = -1$.

Capítulo 5

Cálculo y aplicaciones de los valores y vectores propios

5.1. Fundamentos teóricos

En este capítulo nos dedicaremos al cálculo aproximado de los valores y vectores propios de una matriz cuadrada A . Un número λ es **valor propio** de la matriz A si es posible encontrar un vector no nulo x verificando la identidad

$$Ax = \lambda x.$$

En tal caso se dice que x es un **vector propio** (o autovector) asociado a λ . En general, al conjunto de todos los valores propios de una matriz, también llamados autovalores, lo denotaremos por $\sigma(A)$ y lo llamaremos **espectro de A** . La primera caracterización de los valores propios de una matriz viene dada a través de las raíces de un cierto polinomio, debido al siguiente hecho:

$$Ax = \lambda x, \quad x \neq 0 \Leftrightarrow \text{el sistema } (A - \lambda I)x = 0, \text{ tiene soluciones } x \text{ no nulas,}$$

donde I es la matriz identidad del mismo orden que A . Dado que el sistema anterior siempre posee la solución trivial $x = 0$, afirmar que tiene soluciones no nulas (esto es, vectores propios) equivale a decir que es compatible indeterminado, o bien, a decir que su matriz de coeficientes no tiene rango máximo, por lo que su determinante ha de ser nulo:

$$Ax = \lambda x, \quad x \neq 0 \Leftrightarrow \det(A - \lambda I) = 0.$$

Lo interesante de esta caracterización es que la expresión $\det(A - \lambda I)$ es *un polinomio en la variable λ* (cuyo grado coincide con la dimensión de la matriz A), que se llama **polinomio característico de la matriz A** . Por lo tanto, los valores propios de una matriz son, exactamente, las raíces de su polinomio característico. Así, aunque la matriz A sea real, sus valores propios pueden ser complejos. En este capítulo, nos centraremos en el

cálculo de valores (y vectores) propios reales. *A priori*, este hecho nos lleva a pensar que el cálculo de valores propios se reduce al cálculo de ceros de polinomios, pero en la práctica no es así, dado que cuando la dimensión aumenta, el coste computacional de calcular determinantes aumenta de manera factorial, por lo que optaremos por otras técnicas.

Por otro lado, notemos que cualquier combinación lineal de vectores propios, asociados a un mismo valor propio λ , vuelve a ser vector propio (del mismo valor propio), de donde se deduce que este conjunto es un subespacio vectorial que llamaremos **subespacio propio** asociado al valor propio λ .

Ejemplo 5.1. Así por ejemplo,

- la matriz $\begin{pmatrix} -2 & 1 \\ 1 & -2 \end{pmatrix}$ tiene como polinomio característico $p(\lambda) = (-2 - \lambda)^2 - 1$ y sus valores propios son $\lambda_1 = -1$ y $\lambda_2 = -3$;
- los valores propios de una matriz cualquiera triangular (superior o inferior) coinciden con los elementos de la diagonal de dicha matriz.

Semejanza, diagonalización, potencias y exponencial de una matriz. Para concluir esta breve introducción teórica, vamos a repasar varios conceptos matemáticos ligados a los valores y los vectores propios.

Dos matrices A y B se denominan **semejantes** cuando entre ellas existe una relación del tipo $A = PBP^{-1}$ para cierta matriz regular P . En tal caso A y B tendrán los mismos valores propios, ya que poseen el mismo polinomio característico. En efecto,

$$\det(A - \lambda I) = \det(P(B - \lambda I)P^{-1}) = \det(P) \det(B - \lambda I) \frac{1}{\det(P)} = \det(B - \lambda I).$$

En el caso particular en que A sea semejante a una matriz D diagonal, entonces diremos que A es **diagonalizable** y, en esa situación, es fácil verificar que los elementos de la diagonal de D son los valores propios de A y las columnas de la matriz P son los vectores propios asociados. Esta semejanza permite calcular fácilmente potencias de la matriz A y su exponencial (ambas operaciones de utilidad en algunas aplicaciones que veremos a continuación) ya que

$$A = PDP^{-1} \Rightarrow A^k = PD^kP^{-1} \quad \text{y} \quad e^A = Pe^DP^{-1},$$

y tanto D^k como e^D se calculan de forma trivial (componente a componente sobre su diagonal).

Origen y aplicaciones. Los valores y vectores propios asociados a matrices aparecen, y son especialmente relevantes, en un gran número de campos de la ciencia contemporánea. Por citar sólo algunos de ellos, en **economía**, el modelo input-output de Leontief conlleva el estudio de la matriz de insumo-producto, relacionada con la oferta y la demanda entre sectores productivos. El buen planteamiento de este modelo (que permite

conocer su evolución) depende exclusivamente de que $\lambda = 1$ no sea valor propio de esa matriz. Además, el mayor de sus valores propios (al que llamaremos *dominante*) determinará la tasa máxima de crecimiento del sistema (véase [11]). Encontramos otro ejemplo en **biología**, en modelos de crecimiento poblacional, donde el valor propio dominante tiene el significado de una tasa de crecimiento, e informa sobre el comportamiento de la población bajo estudio una vez su crecimiento se ha estabilizado (véase el ejercicio §5.55). En esta misma línea, las potencias de la matriz, que antes hemos relacionado con sus valores propios, proporcionan información sobre la dinámica del modelo biológico estudiado y permiten establecer *a priori* su comportamiento a largo plazo (véase [10]). En **matemáticas** encontramos, por ejemplo, exponenciales de matrices (que ya hemos relacionado con sus valores propios) cuando resolvemos **ecuaciones diferenciales** lineales y, en el mismo capítulo anterior, hemos visto como el **radio espectral** de una matriz, que se calcula a través de sus valores propios, es determinante a la hora de establecer métodos adecuados de resolución de sistemas de ecuaciones lineales (véase [10]). En **física**, a través de las ecuaciones diferenciales que rigen el movimiento de las partículas y sus implementaciones numéricas (que estudiaremos en los capítulos 9 y 10), son casi innumerables los ejemplos en los que los valores propios dan información relevante sobre el modelo estudiado.

Para tener un ejemplo de referencia en lo que sigue, vamos a presentar uno que se enmarca en esta última línea, un problema de valores propios que se obtiene al discretizar una ecuación diferencial. El ejemplo es debido a Euler (siglo XVIII).

Ejemplo 5.2 (Columnas de Euler). Al estudiar la deformación de una columna vertical al ser sometida a una fuerza de compresión, también vertical, se obtiene la siguiente ecuación diferencial:

$$\left. \begin{aligned} u''(y) &= f u(y), \quad y \in [0, L], \\ u(0) &= 0, \quad u(L) = 0, \end{aligned} \right\}$$

siendo $u(y)$ el desplazamiento horizontal respecto a la posición original de un punto a altura y , L la altura de la columna y f un valor negativo directamente proporcional a la componente vertical de la fuerza que se aplica. La solución $u(y) = 0$ corresponde con el caso en que la columna no se deforma. El problema interesante es buscar los valores de f (fuerza aplicada) para los cuales sí hay una deformación $u \neq 0$, es decir, un problema de valores propios. En este caso, el valor propio más interesante es el más pequeño (en módulo), ya que se corresponderá con la mínima fuerza necesaria para que la columna se deforme. Si discretizamos la ecuación (proceso que desarrollaremos en los capítulos 9 y 10), obtenemos el sistema

$$\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & 1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \lambda \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}, \quad (5.1)$$

siendo $\lambda = \frac{fL^2}{(n+1)^2}$ y $x_i = u\left(i\frac{L}{n+1}\right)$, $i = 1, \dots, n$. Por lo tanto, tener deformaciones equivale a tener vectores x no nulos resolviendo este problema, y los valores propios λ asociados se corresponderán, salvo un factor constante, con la fuerza necesaria para generar esa deformación. Este ejemplo tiene una virtud adicional: los valores propios de la matriz son conocidos analíticamente,

$$\lambda_i = 2\left(\cos\left(k\frac{\pi}{n+1}\right) - 1\right), \quad i = 1, \dots, n, \quad (5.2)$$

lo que nos permitirá medir la bondad de los métodos que vamos a emplear.

Algunos teoremas importantes. Resaltamos dos resultados de especial importancia que son muy fáciles de verificar y que proporcionan, sin embargo, una información muy relevante respecto a los valores propios de una matriz. El primero de ellos nos garantiza que los valores propios sean reales (véase [1, Teorema 2.11] o [9, Teorema 8.1.1]).

Teorema 5.3.

- *Hipótesis:* sea A una matriz simétrica (esto es, $A = A^t$).
- *Tesis:* todos sus valores propios son reales y es diagonalizable, de manera que $A = PDP^t$, siendo P una matriz ortogonal ($P^{-1} = P^t$) y D una matriz diagonal con los valores propios de A en su diagonal.

El segundo resultado permite acotar en el plano complejo \mathbb{C} los valores propios (reales o complejos) de cualquier matriz (véase [9, Teorema 7.2.1] o [15, Teorema 3, Sección 5.2]).

Teorema 5.4 (Teorema de los discos de Gerschgorin).

- *Hipótesis:* sea A una matriz cuadrada cualquiera con entradas a_{ij} , $i, j = 1, \dots, n$.
- *Tesis:* todos sus valores propios están en alguno de los discos

$$D_i := \{z \in \mathbb{C} : |z - a_{ii}| \leq r_i\}, \text{ siendo el radio } r_i := \sum_{j \neq i} |a_{ij}|, \quad i = 1, \dots, n.$$

La aplicación directa de estos resultados a la matriz del ejemplo 5.2 también nos asegura que sus valores propios son números reales negativos.

Métodos de cálculo. En el ejemplo 5.2 el valor propio interesante era el más pequeño aunque, como ya hemos indicado en la introducción, en muchos casos lo habitual es que no sea necesario conocer todos los valores propios de una matriz sino sólo el mayor de ellos (en módulo), al que llamaremos *valor propio dominante*. Para este supuesto emplearemos un **método de potencias** que permitirá estimar únicamente este mayor valor propio y su vector propio asociado. Cabe también remarcar que la misma técnica

permitirá calcular, en dos pasos, el menor valor propio, al que hacíamos referencia en dicho ejemplo. Esto es debido a las siguientes propiedades:

$$\sigma(\alpha I + \beta A) = \{\alpha + \beta\lambda : \lambda \in \sigma(A)\} \text{ y } \sigma(A^2) = \{\lambda^2 : \lambda \in \sigma(A)\},$$

por lo que podemos manipular la matriz A adecuadamente para que su valor propio dominante sea el que nosotros queremos. Veamos cómo resolver el ejemplo 5.2.

Ejemplo 5.5. Para el sistema del ejemplo 5.2, en primer lugar vamos a calcular el valor propio dominante de A que denotaremos λ_{\max} . Sabiendo que todos los valores propios de A son negativos, observamos que $\lambda_{\max} < \dots < \lambda_{\min} < 0$, de donde:

$$\sigma(A - \lambda_{\max}I) = \{0 < \dots < (\lambda_{\min} - \lambda_{\max})\},$$

por lo que si ahora calculamos $\bar{\lambda} = \lambda_{\min} - \lambda_{\max}$, que es el valor propio dominante de $A - \lambda_{\max}I$, tendremos que el valor propio más pequeño vendrá dado por $\lambda_{\min} = \bar{\lambda} + \lambda_{\max}$ (véase el ejercicio §5.54).

En los casos en los que nos interese conocer todos y cada uno de los valores propios, podemos emplear un método que usa la semejanza entre matrices como vía de aproximación, el **método QR**. En este capítulo, nos centraremos exclusivamente en estos dos métodos de cálculo de valores propios: el método de las potencias para estimar el valor propio dominante, y el método QR para estimar todos los valores propios a la vez.

5.1.1. Método de las potencias

Supongamos que A es una matriz real cuadrada de dimensión n **diagonalizable** y que posee un **valor propio dominante** λ_1 , esto es, sus valores propios verifican la cadena de desigualdades

$$|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|.$$

En este caso se puede asegurar que λ_1 es real y su subespacio de vectores propios es una recta. El método se basa en que, en esta situación, la aplicación $x \mapsto Ax$ tiende a mover x hacia la dirección marcada por la recta de vectores propios de λ_1 , para *casi todo* x . De hecho, es posible probar que si iteramos el proceso, (esto es, Ax, A^2x, \dots, A^kx) nos acercaremos cada vez más a la recta de vectores propios asociados al valor propio λ_1 (véase [15]). En este caso, podemos estimar λ_1 como sigue:

$$v_k := A^k x \approx \text{autovector} \Rightarrow Av_k \approx \lambda_1 v_k \Rightarrow \frac{\langle Av_k, v_k \rangle}{\langle v_k, v_k \rangle} \approx \frac{\lambda_1 \langle v_k, v_k \rangle}{\langle v_k, v_k \rangle} = \lambda_1.$$

Este proceso tiene una pega, y es que las componentes de $A^k x$ pueden crecer o decrecer con k y esto hace que “explote” numéricamente, es decir, que no se puedan representar dichas cantidades numéricamente. Pero como al normalizar un vector no se cambia su dirección, si normalizamos el vector $A^k x$ obtenido en cada paso, podemos crear un algoritmo alternativo que evite dicho inconveniente.

Algoritmo 5.6 (Método de las potencias normalizado).

- Supongamos que A es una matriz real cuadrada de dimensión n **diagonalizable** con un **valor propio dominante** λ_1 .
- Tomamos un vector inicial $x_0 \in \mathbb{R}^n$ no nulo cualquiera.
 - Normalizamos $y_0 = \frac{x_0}{\|x_0\|_2}$ y redefinimos $x_1 = Ay_0$.
 - Aproximamos $\lambda_1 \approx \langle x_1, y_0 \rangle$ (no hay que dividir por $\langle y_0, y_0 \rangle$, ya que vale 1.)
 - Repetimos estos dos pasos calculando, y_k , x_{k+1} y la correspondiente aproximación, hasta que se cumpla algún criterio de parada.
- Al detener el bucle, tendremos las aproximaciones del valor propio $\lambda_1 \approx \langle x_{k+1}, y_k \rangle$ y de un vector propio asociado $v_1 \approx x_{k+1}$.

Más adelante, al implementar el método, discutiremos sobre posibles criterios de parada. En cualquier caso, si existe un valor propio dominante, se puede demostrar la convergencia descrita para (casi) cualquier vector inicial $x_0 \in \mathbb{R}^n$ no nulo (véase [28, Teorema 27.1]).

5.1.2. El método QR

Para los casos en que sea necesario calcular todo el espectro de la matriz A , vamos a presentar un método basado en la factorización¹ QR introducida en la sección 3.1.3. El algoritmo consiste en generar una sucesión de matrices A_k , todas semejantes a A , y que sea convergente a una matriz triangular. Recordamos que las matrices semejantes tienen los mismos valores propios y que los valores propios de una matriz triangular son precisamente los números que aparecen en su diagonal.

Algoritmo 5.7 (Método QR para calcular valores propios).

- Supongamos que $A_0 = A$ donde A es una matriz real cuadrada de dimensión n .
- Factorizamos $A_0 = Q_0 R_0$ como producto de Q_0 , ortogonal, y R_0 , triangular superior.
 - Definimos $A_1 = R_0 Q_0$, que es una matriz semejante a A_0 ya que $Q_0^T A_0 Q_0 = R_0 Q_0$.
 - Factorizamos de nuevo $A_1 = Q_1 R_1$.
 - Repetimos el proceso anterior hasta que se cumpla algún criterio de parada.
- Las matrices obtenidas, $\{A_k\}$, son semejantes a A y, bajo ciertas condiciones, tienden a ser triangulares, por lo que su diagonal contiene los valores propios aproximados de la matriz original.

Por último, comentamos que se puede demostrar (véase [4, Lema 4.3]) que si los valores propios de A verifican $|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$, entonces el límite de la sucesión de matrices construida es una matriz T triangular superior (véase el ejercicio §5.52).

¹La factorización QR consiste en descomponer una matriz cuadrada A como producto de una ortogonal Q y una triangular superior R .

De nuevo, cuando implementemos el método, discutiremos sobre posibles criterios de parada. En cualquier caso, vendrán dados por un control sobre “lo que le sobra” a una matriz para ser triangular superior.

5.2. Implementación de algunos métodos

Comenzamos esta sección construyendo la matriz A que nos apareció en el ejemplo 5.2 de la deformación de una columna vertical. Dicha matriz la vamos a definir (de tamaño 50×50) de dos formas distintas para recordar el uso de diferentes comandos.

```
> aux = ones(49,1);
> A = (-2)*eye(50)+diag(aux,1)+diag(aux,-1);
> aux = [-2,1,zeros(1,48)]; A = toeplitz(aux);
```

Esta matriz nos servirá de referencia para verificar los comandos y algoritmos que vamos a implementar. Presentamos en primer lugar algunos comandos directos de Octave relacionados con el cálculo de valores y vectores propios. Como siempre, recomendamos usar `help` comando para ampliar datos y conocer comandos afines.

5.2.1. Los comandos `eig`, `poly` y `expm`

El principal comando que debemos conocer es `eig`, que calcula directamente los autovalores y autovectores de una matriz. Así,

```
> eig(A);           devuelve (no necesariamente ordenados) los autovalores de A,
> [P,D] = eig(A);   matrices  $P$  de autovectores y  $D$  diagonal con los autovalores.
```

En el ejemplo 5.2, la matriz A es diagonalizable y podemos verificar la identidad $A = PDP^{-1}$ calculando la norma² de la diferencia $PD - AP$, que debería ser cero.

```
> norm(P*D-A*P)
ans = 4.7988e-15
```

Como siempre, no podemos esperar un cero absoluto debido a las aproximaciones numéricas realizadas internamente, pero sí algo muy cercano a cero, del orden de 10^{-15} .

El segundo comando que queremos presentar es `poly(A)`, que permite calcular el polinomio característico asociado a la matriz A , es decir, el polinomio cuyas raíces son los valores propios de A . En este punto es muy importante destacar que el comando `poly` es uno de esos comandos que, como comentamos en el capítulo 1, actúa de forma diferente si se aplica a vectores o a matrices. De hecho, si ejecutamos `poly(v)`, siendo v un vector, Octave nos devuelve un polinomio cuyas raíces son las componentes de v , por lo que podría ocasionar confusión si pensásemos que sobre matrices actúa columna a columna.

²Recordamos que el comando `norm(A)` calcula la norma matricial inducida por la norma euclídea.

Ya comentamos en la introducción del capítulo que, numéricamente, calcular los autovectores a partir del polinomio característico puede no ser una buena idea. Vamos a ilustrar este hecho con un ejemplo sencillo. Si partimos de la matriz identidad de orden 3 y calculamos por esta vía sus valores propios (que son todos iguales a 1), ejecutaríamos

```
> roots(poly(eye(3)))
```

obteniendo $1.00001+0.00001i$, $1.00001+0.00001i$, $0.99999+0.00000i$, con un error de 10^{-5} , a pesar de ser una matriz tan simple. Si chequeamos el error para nuestra matriz de referencia ejecutando `roots(poly(A))`, comparando con el resultado dado por `eig(A)`, observaremos un error mucho mayor (aunque, curiosamente, insignificante en los valores propios más pequeños, que son los que nos interesan en ese caso).

Por último, presentaremos el comando `expm(A)`, que calcula la matriz exponencial de A , y que puede ser útil en la resolución de ecuaciones diferenciales lineales con coeficientes constantes.

Antes de pasar a la implementación de los métodos descritos en la introducción del capítulo, vamos a resolver el ejemplo 5.2 usando los comandos directos de Octave que acabamos de presentar. Recordamos que se trata de estudiar la fuerza vertical hacia abajo mínima necesaria para deformar una columna vertical de longitud L (que vamos a fijar en $L = 5$ metros). Al discretizar la ecuación diferencial que rige esta deformación $u'' = fu$ (siendo $(-f)$ la fuerza necesaria por kilo de viga y por metro de viga) usando $n = 50$, nos aparece un problema de valores propios $Ax = \lambda x$ siendo A la matriz de referencia en esta sección y $\lambda = fL^2/(n+1)^2 = 0.0096117f$. El siguiente *script* nos da la respuesta y representa la viga vertical deformada y la fuerza aplicada, lo que nos permite observar que su máxima deformación se produce en el centro.

```
1 % Definiciones: longitud, número de nodos interiores y matriz
2 L = 5; n = 50;
3 v = [-2,1,zeros(1,n-2)]; A = toeplitz(v);
4 % Cálculo de valores propios y localización del menor en |.|
5 [P,D] = eig(A);
6 [lambda,posicion] = max(diag(D)); % Valores propios negativos
7 vector = P(:,posicion);
8 % Representación gráfica de la deformación
9 y = linspace(0,L,n+2);
10 deformacion = [0,vector',0];
11 plot(deformacion,y)
12 texto = sprintf("fuerza = %f", lambda*(n+1)^2/L^2);
13 xlabel(texto);
14 axis([-3,3,0,L]);
```

Este *script* nos devuelve la gráfica mostrada en la figura 5.1. Observamos que hemos incluido en dicha gráfica el valor de la fuerza necesaria mediante `xlabel("string")`, una opción que lo coloca como etiqueta para el eje OX . La cadena que muestra se ha

generado con el comando `sprintf`, que funciona de manera análoga a `fprintf`, aunque el resultado lo guarda en una cadena de caracteres. El rectángulo donde se hace la representación gráfica se fija mediante el comando `axis`.

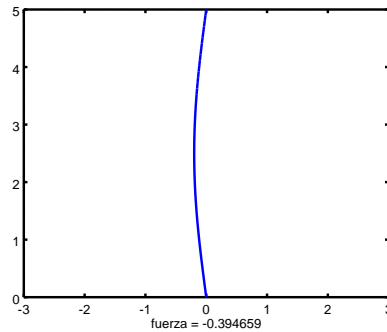


Figura 5.1: Deformación de una viga

5.2.2. Método de las potencias

Vamos a implementar el método de las potencias descrito en el algoritmo 5.6.

```

1 function [lambda,y] = potencias(A,x,tol,maxiter)
2 % Calcula una aproximación del valor propio dominante de A,
3 % matriz cuadrada, mediante el método de potencias a partir
4 % de un vector inicial x (columna).
5 % maxiter determina el número máximo de iteraciones del método.
6 % tol es la tolerancia sobre el error relativo entre iteraciones.
7 lambdaviejo = rand();
8 k = 0;
9 error = 1000;
10 while k<=maxiter && error>=tol
11     y = x/norm(x);
12     x = A*y;
13     lambda = y'*x;
14     error = abs((lambda-lambdaviejo)./lambda);
15     lambdaviejo = lambda;
16     k = k+1;
17 endwhile
18 if k>maxiter
19     warning('No converge tras %i iteraciones \n',maxiter);
20 else
21     fprintf('El método converge en %i iteraciones \n',k);
22 endif
23 endfunction

```

El criterio de parada elegido en este código es: detener el proceso cuando la distancia relativa entre dos aproximaciones sucesivas del valor propio dominante (es decir, $|\lambda^{(k)} - \lambda^{(k-1)}|/|\lambda^{(k)}|$, donde $\lambda^{(k)}$ es la k -ésima aproximación) sea inferior a la tolerancia prescrita.

Para completar el ejemplo 5.2, y usando la idea expuesta en el ejemplo 5.5, vamos a calcular el menor valor propio (en valor absoluto) de la matriz del sistema (5.1), es decir, la mínima fuerza necesaria para que la viga se deforme, usando la función `potencias` recién definida. Teniendo en cuenta que ya habíamos definido la matriz A al inicio de la sección 5.2, los comandos a implementar serían los siguientes:

```
> x = rand(50,1);
> bar1 = potencias(A,x,10^(-10),3000);
> bar2 = potencias(A-bar1*eye(50),x,10^(-10),3000);
> lambda = bar2+bar1;
> fuerza = -lambda*51^2/5^2;
```

lo que nos proporciona una estimación del menor valor propio $\lambda = -0.003793$ y de la fuerza mínima necesaria $f = 0.39466$ bastante similar a la obtenida usando `eig` en el *script* anterior: $f = 0.39467 N/(kgm)$. Además, usando los valores analíticos de los valores propios, y tomando $k = 1$ en (5.2), obtenemos el menor de ellos (en valor absoluto), de modo que ejecutando

```
> 2*(cos(pi/51)-1)
ans = -0.0037933
```

confirmamos que nuestro algoritmo funciona bastante bien.

5.2.3. Método QR

Por último, vamos a implementar el método QR descrito en la introducción. En cada paso utilizaremos el comando propio de Octave `qr` (A) para calcular la descomposición QR de una matriz, que ya fue comentado en el capítulo 3.

```
1 function d = qrvp(A,tol,maxiter)
2 % Esta función calcula una aproximación de los valores propios
3 % de una matriz cuadrada A mediante el método QR.
4 % nmax determina el número máximo de iteraciones a realizar.
5 % tol es una tolerancia sobre lo que le sobra a las matrices
6 % para ser triangulares.
7 T = A;
8 k = 0;
9 while k<=maxiter && norm(tril(T,-1),inf)>=tol
10     [Q,R] = qr(T);
11     T = R*Q;
```

```
12     k = k+1;
13 endwhile
14 if k>maxiter
15     warning('No converge tras %i iteraciones \n',maxiter);
16 else
17     fprintf('El método converge en %i iteraciones \n',k);
18 endif
19 d = diag(T);
20 end
```

Observamos en las línea 9 del código el criterio de parada establecido: además de usual tope en el número de iteraciones a realizar, el algoritmo se detendrá cuando la norma matricial de su parte inferior sea más pequeña que una tolerancia prefijada, puesto que las matrices obtenidas convergen a una triangular superior.

Así, por ejemplo, podemos comprobar que dicho algoritmo, aplicado a la matriz A del ejemplo 5.2, converge a una matriz triangular semejante, salvo tolerancias del orden de 10^{-6} , tras 3522 iteraciones mediante el comando

```
> qrvp(A,10^(-6),10000)
```

En este caso particular, se puede comprobar que los valores de esta matriz diagonal son una aproximación de los valores propios de A con un error menor que 10^{-10} .

Capítulo 6

Interpolación

6.1. Fundamentos teóricos

En este capítulo pretendemos abordar el problema de interpolación más simple, que no obstante posee múltiples aplicaciones, algunas de las cuales iremos viendo a lo largo de los siguientes capítulos. En general, consiste en encontrar una función regular muy simple, por ejemplo un polinomio, que pueda ser utilizada para diversos fines (evaluar, integrar, derivar, ...) y cuyos valores en ciertos puntos vengan prescritos de antemano por el problema bajo resolución. Ilustramos esta idea con un ejemplo.

Ejemplo 6.1. En la tabla de la distribución normal tipificada $N_{(0,1)}$ encontramos los siguientes valores:

x	0.50	0.51	0.52	0.53	0.54
$N_{(0,1)}(x)$	0.69146	0.69497	0.69847	0.70194	0.70540

(6.1)

Si necesitásemos el valor de dicha normal en $x = 0.513$, ¿qué podríamos hacer? Quizás la opción más sencilla sería calcular la recta $y(x) = ax + b$ que pasa por los puntos más cercanos al nuestro (es decir, $(0.51, 0.69497)$ y $(0.52, 0.69847)$), y tomar $y(0.513)$ como una aproximación del valor buscado. Otra vía, algo más complicada, sería buscar otra función sencilla $f(x)$, que satisfaga las cinco condiciones sobre los valores dados, y tomar $f(0.513)$ como una aproximación del valor buscado. Cualquiera que sea la opción elegida, estamos ante un problema de *interpolación de datos*.

En general, la interpolación de “datos” consiste en el siguiente proceso.

1. Consideramos una serie de datos obtenidos a partir de un experimento, a partir de la evaluación de una cierta función (no conocida completamente o difícil de evaluar), o de cualquier otra forma.
2. Construimos una función “simple” que satisfaga exactamente los datos considerados.

3. A la función construida la llamaremos **función interpoladora** (o **interpolante**) y la consideraremos como la función “real” asociada al experimento o a la función que desconocíamos.

Vamos a formalizar y concretar esta idea.

6.1.1. Problema general de interpolación de datos lagrangianos

Consideremos $(n+1)$ datos lagrangianos, esto es, $(n+1)$ datos de la forma $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$, con $x_i \neq x_j$ si $i \neq j$. El problema general de interpolación consiste en hallar una función interpoladora $f(x)$ tal que $f(x_i) = y_i$, para todo $0 \leq i \leq n$.

Es deseable que la función interpoladora cumpla una serie de características tales como ser fácil de hallar, fácil de evaluar, “suave” (es decir, suficientemente derivable), etc. Según la familia de funciones que usemos, tenemos distintos tipos de interpolación.

- Polinomial: se buscan polinomios (el caso más simple y usual).
- Trigonométrica: se buscan combinaciones de senos y cosenos (para el caso de datos oscilantes y/o periódicos).
- Por *splines*: se buscan funciones polinómicas a trozos (cuando hay “muchos” nodos).

Interpolación polinomial para datos lagrangianos

A partir de ahora, denotaremos por $\mathbb{P}_n[x]$ el espacio vectorial de los polinomios, en una variable, de grado menor o igual que n . Consideramos los $(n+1)$ datos lagrangianos de la siguiente tabla,

$$\begin{array}{c|c|c|c|c} x_i & x_0 & x_1 & \cdots & x_n \\ \hline y_i & y_0 & y_1 & \cdots & y_n \end{array}, \quad \text{con } x_i \neq x_j \text{ si } i \neq j, \quad (6.2)$$

y queremos hallar un polinomio $p(x) \in \mathbb{P}_n[x]$ tal que $p(x_i) = y_i$ para todo $0 \leq i \leq n$. Destacamos en este punto que $p(x)$ **no tiene que ser de grado exacto n** , ya que en $\mathbb{P}_n[x]$ también están los polinomios de grado inferior. Antes de resolver este problema, necesitamos saber si hay solución y, en caso de existir, si es única.

Teorema 6.2.

- *Hipótesis*: sean los $(n+1)$ datos lagrangianos dados en la tabla (6.2)
- *Tesis*: existe un único polinomio $p(x) \in \mathbb{P}_n[x]$ tal que $p(x_i) = y_i$, $0 \leq i \leq n$.

Demostración. Incluimos la demostración de este resultado porque es simple y, sobre todo, porque es de tipo constructivo, esto es, proporciona un camino para encontrar $p(x)$.

- Llamamos $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ al polinomio cuyos coeficientes debemos determinar.

- Imponiendo que $p(x)$ pase por los $(n+1)$ datos dados ($p(x_i) = y_i$), obtenemos el siguiente sistema de $(n+1)$ ecuaciones con $(n+1)$ incógnitas, que son los coeficientes del polinomio $p(x)$,

$$\left. \begin{array}{cccccc} a_n x_0^n & + & a_{n-1} x_0^{n-1} & + & \dots & + & a_1 x_0 & + & a_0 & = & y_0 \\ a_n x_1^n & + & a_{n-1} x_1^{n-1} & + & \dots & + & a_1 x_1 & + & a_0 & = & y_1 \\ & & & & & & & & & \vdots & \\ a_n x_n^n & + & a_{n-1} x_n^{n-1} & + & \dots & + & a_1 x_n & + & a_0 & = & y_n \end{array} \right\}. \quad (6.3)$$

- La matriz de coeficientes asociada al sistema (6.3) es una **matriz de Vandermonde** en los nodos $\{x_0, x_1, \dots, x_n\}$, cuyo determinante se sabe que es no nulo si, y sólo si, los nodos son distintos dos a dos (que es, justamente, nuestro caso).
- Concluimos que el sistema (6.3) tiene solución única y, por tanto, existe un único polinomio $p(x)$ que es solución del problema de interpolación propuesto. \square

Veamos a continuación varios métodos que nos permitirán calcular $p(x)$.

Método de los coeficientes indeterminados. Consiste en aplicar directamente la demostración que acabamos de ver, de forma que lo único que debemos hacer es resolver el sistema (6.3). Vamos a aplicarlo a un ejemplo concreto.

Ejemplo 6.3. Queremos calcular el polinomio $p(x) = a_2 x^2 + a_1 x + a_0$ tal que interpole los datos $\{(1, 0), (2, 3), (3, 9)\}$. El sistema que debemos resolver es

$$\left. \begin{array}{cccc} a_2 & + & a_1 & + & a_0 & = & 0 \\ 4a_2 & + & 2a_1 & + & a_0 & = & 3 \\ 9a_2 & + & 3a_1 & + & a_0 & = & 9 \end{array} \right\}.$$

Ya sabemos, por lo visto en la demostración del teorema, que el sistema es compatible determinado (de hecho, el determinante de la matriz de coeficientes es (-2)), y la solución es $a_2 = \frac{3}{2}$, $a_1 = -\frac{3}{2}$, $a_0 = 0$. Por tanto, el polinomio buscado es $p(x) = \frac{3}{2}x^2 - \frac{3}{2}x$.

Método de Lagrange. La idea de este método es la construcción de una base “adecuada” del espacio vectorial $\mathbb{P}_n[x]$, a la que llamaremos **base de Lagrange**.

En concreto, para los datos de la tabla (6.2), la base de Lagrange es el conjunto de polinomios $\{\ell_0(x), \ell_1(x), \dots, \ell_n(x)\}$, dados por

$$\ell_i(x) = \frac{(x-x_0) \cdots (x-x_{i-1})(x-x_{i+1}) \cdots (x-x_n)}{(x_i-x_0) \cdots (x_i-x_{i-1})(x_i-x_{i+1}) \cdots (x_i-x_n)}, \quad 0 \leq i \leq n.$$

La particularidad que caracteriza a estos polinomios es que cada $\ell_i(x)$ es un polinomio de grado n que vale 1 en x_i y 0 en los demás nodos. Por ello, la solución del problema de interpolación (es decir, el único polinomio $p(x)$ que interpola los datos de la tabla (6.2)) viene ahora dada por la expresión

$$p(x) = y_0 \ell_0(x) + y_1 \ell_1(x) + \dots + y_n \ell_n(x).$$

Ejemplo 6.4. Para los datos del ejemplo 6.3, la base de Lagrange es

$$\ell_0(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)}, \quad \ell_1(x) = \frac{(x-1)(x-3)}{(2-1)(2-3)}, \quad \ell_2(x) = \frac{(x-1)(x-2)}{(3-1)(3-2)}.$$

Por tanto, el polinomio interpolador es

$$p(x) = 0 \cdot \frac{(x-2)(x-3)}{(1-2)(1-3)} + 3 \cdot \frac{(x-1)(x-3)}{(2-1)(2-3)} + 9 \cdot \frac{(x-1)(x-2)}{(3-1)(3-2)} = \frac{3}{2}x^2 - \frac{3}{2}x.$$

Observamos que, como no podría ser de otro modo, este polinomio es el mismo que obteníamos en la resolución del ejemplo por el método de los coeficientes indeterminados.

Método de Newton. En este caso la idea es construir recursivamente el polinomio interpolador, es decir, construir el polinomio conforme vamos obteniendo los datos. Veamos primero el proceso sobre un ejemplo sencillo para entender la forma en la que se va construyendo el polinomio.

Ejemplo 6.5. De nuevo, para los datos del ejemplo 6.3, construimos el polinomio interpolador, pero en tantos pasos como datos tenemos. En este caso se realizarán tres pasos.

- Polinomio (de grado 0) que interpola al primer dato, $(1, 0)$: $p_0(x) = 0$.
- Polinomio de la forma $p_1(x) = p_0(x) + \alpha_1(x-1)$ que interpola los dos primeros datos, $(1, 0)$ y $(2, 3)$:
 - Por su forma, interpola al primer dato: $p_1(1) = 0$.
 - Para el dato adicional: $p_1(2) = 3 \Rightarrow \alpha_1 = 3$ y $p_1(x) = 3(x-1)$.
- Polinomio de la forma $p_2(x) = p_1(x) + \alpha_2(x-1)(x-2)$ que interpola todos los datos:
 - Por su forma, interpola los dos datos anteriores: $p_2(1) = 0$ y $p_2(2) = 3$.
 - Para el dato adicional: $p_2(3) = 9 \Rightarrow \alpha_2 = \frac{3}{2}$ y $p_2(x) = \frac{3}{2}(x-1)(x-2) + 3(x-1)$.

Si simplificamos el último polinomio calculado, obtenemos $p_2(x) = \frac{3}{2}x^2 - \frac{3}{2}x$ que, por supuesto, coincide con el obtenido mediante los métodos expuestos en los dos ejemplos anteriores.

Ahora enunciamos el resultado general que nos proporcionará el método de Newton para calcular el polinomio que interpola un conjunto de datos del tipo (6.2).

Teorema 6.6.

- *Hipótesis:*
 - Sean los $(k+1)$ primeros datos lagrangianos $\{(x_0, y_0), (x_1, y_1), \dots, (x_k, y_k)\}$, con $k \geq 0$, del problema general (6.2).
 - Sea, para $k \geq 1$, $p_{k-1}(x) \in \mathbb{P}_{k-1}[x]$ el polinomio interpolador de los k primeros datos, $\{(x_0, y_0), (x_1, y_1), \dots, (x_{k-1}, y_{k-1})\}$.

- *Tesis: el polinomio $p_k(x) \in \mathbb{P}_k[x]$ interpolador de $\{(x_0, y_0), (x_1, y_1), \dots, (x_k, y_k)\}$ es*

$$p_k(x) = p_{k-1}(x) + \alpha_k(x - x_0)(x - x_1) \cdots (x - x_{k-1}),$$

donde el coeficiente α_k viene determinado por

$$\alpha_k = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})}.$$

Del teorema anterior deducimos que el polinomio que interpola todos los datos de la tabla (6.2) es

$$\begin{aligned} p(x) = p_n(x) = & \alpha_0 + \alpha_1(x - x_0) + \alpha_2(x - x_0)(x - x_1) + \dots \\ & + \alpha_n(x - x_0)(x - x_1) \cdots (x - x_{n-1}), \end{aligned} \quad (6.4)$$

cuyos coeficientes α_k se obtienen de forma recursiva según el esquema

$$\alpha_0 = y_0; \quad \alpha_k = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})}, \quad 1 \leq k \leq n.$$

Para terminar este apartado, indicar que al conjunto de polinomios $\{1, (x - x_0), (x - x_0)(x - x_1), \dots, (x - x_0)(x - x_1) \cdots (x - x_{n-1})\}$ normalmente se le denomina **base de Newton** de $\mathbb{P}_n[x]$ asociada a los nodos $\{x_0, \dots, x_{n-1}\}$.

Diferencias divididas. Sería interesante, cuando interpolamos siguiendo la idea de Newton, poder calcular fácil y rápidamente los coeficientes α_i del polinomio interpolador. Para ello, vamos a introducir una nueva notación, considerando que los datos lagrangianos de la tabla (6.2) vienen dados a partir de una función, es decir, son de la forma $y_i = f(x_i)$, y se trata de buscar el polinomio interpolador $p(x)$ dado por (6.4). A partir de lo dicho, tenemos la siguiente definición.

Definición 6.7. Llamamos *diferencia dividida de orden k* (de f en los nodos $\{x_0, \dots, x_k\}$) al coeficiente α_k definido en el teorema 6.6, y la denotamos por

$$\alpha_k = f[x_0, x_1, \dots, x_k], \quad 0 \leq k \leq n.$$

Con esta notación, la expresión del polinomio interpolador en la forma de Newton (6.4) viene dada por

$$\begin{aligned} p(x) = & f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \dots \\ & + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}). \end{aligned} \quad (6.5)$$

La utilidad de esta notación la veremos tras el siguiente resultado, en el cual se enuncian ciertas propiedades de las diferencias divididas que nos permitirán calcularlas de forma fácil y rápida (véase [7, §2.6]).

Teorema 6.8.■ *Hipótesis:*

- Para $k \geq 0$, sea $\{(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_k, f(x_k))\}$ un conjunto con $(k+1)$ datos lagrangianos.
- Sean $f[x_i] = f(x_i)$, $0 \leq i \leq k$, las diferencias divididas de orden cero.

■ *Tesis:*

- Para $k \geq 1$, la diferencia dividida de orden k viene dada por la expresión

$$f[x_0, x_1, \dots, x_k] = \frac{f[x_1, x_2, \dots, x_k] - f[x_0, x_1, \dots, x_{k-1}]}{x_k - x_0}. \quad (6.6)$$

- $f[x_0, x_1, \dots, x_k]$ no depende del orden en que escribamos sus argumentos.

Gracias a la fórmula recurrente dada en (6.6), podemos calcular las diferencias divididas que necesitamos construyendo la siguiente tabla de izquierda a derecha.

x_0	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	\dots	$f[x_0, x_1, \dots, x_n]$
x_1	$f[x_1]$	$f[x_1, x_2]$	\vdots	\ddots	
x_2	$f[x_2]$	\vdots	$f[x_{n-2}, x_{n-1}, x_n]$		
\vdots	\vdots	$f[x_{n-1}, x_n]$			
x_n	$f[x_n]$				

De esta forma, los coeficientes del polinomio interpolador (6.5) los encontramos en la primera fila de la tabla anterior.

Ejemplo 6.9. Para los datos del ejemplo 6.3, la tabla de diferencias divididas correspondiente es la siguiente.

1	0	$\frac{3-0}{2-1} = 3$	$\frac{6-3}{3-1} = \frac{3}{2}$
2	3	$\frac{9-3}{3-2} = 6$	
3	9		

De donde, de nuevo, obtenemos $p(x) = 0 + 3(x-1) + \frac{3}{2}(x-1)(x-2) = \frac{3}{2}x^2 - \frac{3}{2}x$.

Ejemplo 6.10. En este otro ejemplo, vamos a considerar los datos $\{(0, 1), (1, 0), (2, 3), (3, 9)\}$, con los que obtenemos la siguiente tabla de diferencias divididas.

1	0	$\frac{3-0}{2-1} = 3$	$\frac{6-3}{3-1} = \frac{3}{2}$	$\frac{\frac{5}{3}-\frac{3}{2}}{0-1} = -\frac{1}{6}$
2	3	$\frac{9-3}{3-2} = 6$	$\frac{\frac{8}{3}-6}{0-2} = \frac{5}{3}$	
3	9	$\frac{1-9}{0-3} = \frac{8}{3}$		
0	1			

Por consiguiente, el polinomio que interpola los datos dados es $p(x) = 0 + 3(x - 1) + \frac{3}{2}(x - 1)(x - 2) - \frac{1}{6}(x - 1)(x - 2)(x - 3) = -\frac{1}{6}x^3 + \frac{5}{2}x^2 - \frac{10}{3}x + 1$.

Debemos observar que, en este ejemplo, ha bastado con añadir un nuevo nodo a los datos del ejemplo 6.9, lo que implica una nueva fila y un término más en cada fila de la tabla de diferencias divididas de dicho ejemplo. Además, también observamos que los nodos (colocados en la primera columna) no tienen por qué estar necesariamente ordenados.

Vistos estos dos ejemplos, debemos hacer notar que, en general, no es recomendable simplificar la expresión del polinomio para escribirlo en términos de la base clásica de monomios, pues la evaluación del polinomio usando el algoritmo de Horner-Ruffini, cuando este está expresado en la base de Newton, es muy eficiente.

Comparación de los métodos. Si empleamos el método de coeficientes indeterminados, hallar la expresión del polinomio interpolador involucra la resolución de un sistema y, por tanto, necesitaremos realizar bastante cálculos. Además, el sistema cambia cada vez que se modifican los datos. Sin embargo, al tener los coeficientes del polinomio en la base usual, es fácil de evaluar pues podemos aplicar el algoritmo de Horner-Ruffini (véase [15] y el ejercicio §1.11). Dicho algoritmo es el que utiliza internamente el comando `polyval`.

En segundo lugar, usando la idea de Lagrange no necesitamos ningún cálculo para dar la expresión del polinomio interpolador y, además, dicha expresión varía poco si sólo cambian las ordenadas y_i de los datos. Sin embargo, la evaluación del polinomio exige muchos cálculos y la base debe ser modificada cada vez que cambien las abscisas x_i de los datos o cuando se añadan nuevos datos. Debemos tener también en cuenta que esta manera de proceder es inestable computacionalmente.

Por último, mediante Newton necesitamos elaborar la tabla de diferencias divididas para dar la expresión del polinomio interpolador, lo cual implica un volumen intermedio de cálculos. Además, si se añaden nuevos datos, se puede aprovechar la tabla ya realizada. Por otra parte, por la manera de construir el polinomio, el algoritmo de Horner-Ruffini correspondiente a la base de Newton asociada al problema permite una fácil y eficiente evaluación del polinomio (véase [15]).

6.1.2. Interpolación polinomial para datos de tipo Hermite

Decimos que los datos de un problema de interpolación son de **tipo Hermite** cuando, además de los valores de la función en los nodos, aparecen valores en las derivadas sucesivas. En particular, hablamos del problema de **interpolación de Hermite clásico** cuando conocemos el valor de la función y de la primera derivada en todos los nodos, es

decir, los datos son del tipo,

$$\begin{array}{c|c|c|c|c} x_i & x_0 & x_1 & \dots & x_n \\ \hline y_i^0 = f(x_i) & y_0^0 & y_1^0 & \dots & y_n^0 \\ y_i^1 = f'(x_i) & y_0^1 & y_1^1 & \dots & y_n^1 \end{array} \quad \text{con } x_i \neq x_j \text{ si } i \neq j. \quad (6.7)$$

Otro caso particular es el de **interpolación tipo Taylor**, en el que tenemos los valores de la función y sus sucesivas derivadas en un único nodo x_0 ,

$$\begin{array}{c|c|c|c|c} \text{nodo: } x_0 & f(x_0) & f'(x_0) & \dots & f^{(n)}(x_0) \\ \hline \text{valores:} & y_0^0 & y_0^1 & \dots & y_0^n \end{array} \quad (6.8)$$

El siguiente resultado nos garantiza la existencia de un único polinomio interpolador para este tipo de datos.

Teorema 6.11.

- *Hipótesis:* sean $(N + 1)$ datos de tipo Hermite de forma que, si aparece la derivada j -ésima en un nodo, entonces también aparecen todas las derivadas de orden inferior en dicho nodo.
- *Tesis:* existe un único polinomio $p(x)$, de grado menor o igual que N , que interpola los datos de Hermite dados.

En particular, el problema de Hermite clásico (6.7) con $(n + 1)$ nodos, que tiene $(2n + 2)$ datos, es unisolvante en $\mathbb{P}_{2n+1}[x]$, y el problema de interpolación de tipo Taylor (6.8) con n derivadas, que tiene $(n + 1)$ datos, es unisolvante en $\mathbb{P}_n[x]$.

Para resolver estos problemas de interpolación que involucran derivadas, la opción más simple es generalizar la idea de Newton. Para ello necesitamos extender el concepto de diferencia dividida. Gracias a la fórmula recurrente (6.6) podemos “intuir” lo siguiente:

$$f'(x_0) = \lim_{x_1 \rightarrow x_0} \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \lim_{x_1 \rightarrow x_0} \frac{f[x_1] - f[x_0]}{x_1 - x_0} \stackrel{(6.6)}{=} \lim_{x_1 \rightarrow x_0} f[x_0, x_1] \stackrel{?}{=} f[x_0, x_0],$$

lo que permitiría relacionar las derivadas con las diferencias divididas, pero con argumentos repetidos. En realidad, podemos hacer la siguiente definición, de modo que el teorema 6.8 siga siendo válido aún cuando se repita algún nodo (véase [7, §2.7]).

Definición 6.12 (Diferencias divididas con argumentos repetidos). Sea f una función k veces derivable en x_0 , entonces

$$f[\overbrace{x_0, x_0, \dots, x_0}^{k+1 \text{ veces}}] := \frac{f^{(k)}(x_0)}{k!}.$$

Por lo tanto, podremos construir una tabla de diferencias divididas también para datos de tipo Hermite. En este caso, los nodos repetidos deberán colocarse en la primera columna de forma consecutiva, de modo que en las primeras columnas de la tabla se usará esta definición cuando aparezcan todos los nodos repetidos, y la fórmula (6.6) en los demás casos. Veámoslo en un ejemplo.

Ejemplo 6.13. Supongamos que queremos interpolar los datos

$$f(2) = 1; f'(2) = 3; f(3) = 3; f'(3) = 2; f(4) = 6.$$

La tabla de diferencias divididas es

$$\begin{array}{llll} 2 & 1 & f'(2) = 3 & \frac{2-3}{3-2} = -1 \quad \frac{0-(-1)}{3-2} = 1 \quad \frac{\frac{1}{2}-1}{4-2} = \frac{-1}{4} \\ 2 & 1 & \frac{3-1}{3-2} = 2 & \frac{2-2}{3-2} = 0 \quad \frac{1-0}{4-2} = \frac{1}{2} \\ 3 & 3 & f'(3) = 2 & \frac{3-2}{4-3} = 1 \\ 3 & 3 & \frac{6-3}{4-3} = 3 & \\ 4 & 6 & & \end{array}$$

y, por tanto, el polinomio interpolador en términos de la base de Newton correspondiente será $p(x) = 1 + 3(x-2) - (x-2)^2 + (x-2)^2(x-3) - \frac{1}{4}(x-2)^2(x-3)^2$.

Como vemos en el anterior ejemplo, cuando resolvemos un problema de interpolación con datos de tipo Hermite siguiendo la idea de Newton, al construir la base, cada abscisa aparece tantas veces como valores (de la función y sus derivadas) se den en ella.

6.1.3. Error en la interpolación polinomial

Consideremos un problema de interpolación para $(N+1)$ datos, que pueden ser de tipo lagrangiano o de tipo Hermite, correspondientes a cierta función $f(x)$.

Teorema 6.14 (Véase [7]).

■ *Hipótesis:*

- Sea $f(x)$ una función con derivada de orden $(N+1)$ continua en $[a, b]$;
- Sea $p(x) \in \mathbb{P}_N[x]$ el polinomio interpolador (en $(N+1)$ datos dados);
- Sea $x^* \in [a, b]$.

■ *Tesis:* el error cometido al aproximar $f(x^*)$ por $p(x^*)$ viene dado por

$$f(x^*) - p(x^*) = \frac{f^{(N+1)}(\xi)}{(N+1)!} \prod_{k=0}^N (x^* - x_k), \quad (6.9)$$

donde x_0, x_1, \dots, x_N son los nodos de interpolación (que podrían estar repetidos si hay datos sobre las derivadas) y $\xi \in]a, b[$ es algún punto que depende de x^* .

Debemos observar que, si tenemos datos lagrangianos, entonces los nodos serán distintos entre sí, mientras que si son datos de tipo Hermite, entonces aparecerán nodos repetidos en (6.9). Por otra parte, como ξ no es fácil de determinar, se suele usar la acotación

$$|f(x^*) - p(x^*)| \leq \frac{M}{(N+1)!} \prod_{k=0}^N |x^* - x_k|, \quad \text{donde } M = \max_{x \in [a, b]} \left\{ |f^{(N+1)}(x)| \right\}. \quad (6.10)$$

A partir de las expresiones (6.9) o (6.10), podemos sospechar que el error será mayor cuanto mayor sea el intervalo $[a, b]$. Por otra parte, es frecuente que los polinomios de interpolación ajusten bien en el centro del intervalo y mal en los extremos. Además, aumentar el número de nodos no garantiza una mejor “aproximación” mediante polinomios de interpolación. En este sentido, Carl Runge demostró en 1900 que, si interpolamos la función $f(x) = \frac{1}{1+x^2}$ en el intervalo $[-5, 5]$ tomando nodos equiespaciados, entonces la sucesión de polinomios $p_n(x) \in \mathbb{P}_n[x]$ no converge a $f(x)$ si $|x| > 3.6$ (véase 93). Pudiera parecer que este resultado contradice a Karl Weierstrass, quien en 1885 probó que, si $f: [a, b] \rightarrow \mathbb{R}$ es una función continua y $\varepsilon > 0$, entonces existe un polinomio $q(x)$ tal que $|f(x) - q(x)| < \varepsilon, \forall x \in [a, b]$ (con palabras: una función continua se puede aproximar, en un intervalo cerrado y acotado, tanto como deseemos por un polinomio). Sin embargo, no existe tal contradicción. Lo que ocurre es que, si queremos interpolar y aproximar bien una función de manera simultánea, entonces debemos tomar los nodos “adecuadamente”, y lo que el fenómeno de Runge dice es, precisamente, que hacerlo de modo equiespaciado no es lo más apropiado.

6.1.4. Funciones polinómicas a trozos: *splines*

Para paliar el problema de la interpolación cuando hay muchos nodos, recurriremos al uso de funciones polinómicas a trozos o, como se les denomina usualmente, funciones *spline*. La idea de la interpolación por funciones *spline* es “pegar” adecuadamente polinomios, esto es,

- usamos polinomios de grado bajo para interpolar pequeños grupos de datos (interpolación a trozos),
- e imponemos, si es necesario, condiciones adicionales (valores de la función o sus derivadas en los nodos ya existentes o en nodos auxiliares).

En lo que sigue, consideraremos los $(n+1)$ datos lagrangianos de la tabla (6.2) que consideramos ordenados de modo que $a = x_0 < x_1 < \dots < x_n = b$, y definiremos las diferentes familias de funciones *spline* que usaremos para buscar la función interpoladora.

Spline lineal. Un *spline* lineal (de clase cero) es una función continua formada por trozos de rectas (polinomios de grado menor o igual que uno en cada intervalo $[x_{i-1}, x_i]$, con $1 \leq i \leq n$). El *spline* lineal es la función interpoladora $s(x)$ dada por

$$s(x) = \begin{cases} y_{i-1} + \frac{y_i - y_{i-1}}{x_i - x_{i-1}}(x - x_{i-1}), & \forall x \in [x_{i-1}, x_i], \quad 1 \leq i \leq n. \end{cases}$$

Como vemos, hay una única forma de calcular dicho *spline* lineal, y no hay que añadir condiciones adicionales; esta unisolvencia es debida a que el espacio vectorial de los *splines* lineales sobre $(n + 1)$ nodos tiene precisamente dimensión igual a $(n + 1)$.

Spline cuadrático. Un *spline* cuadrático de clase uno es una función que está formada por trozos de parábolas (un polinomio de grado menor o igual que dos en cada subintervalo $[x_{i-1}, x_i]$, para $1 \leq i \leq n$) y que admite derivada primera continua.

En este caso, la dimensión del espacio vectorial de los *splines* cuadráticos de clase uno sobre $(n + 1)$ nodos es igual a $(n + 2)$. Por tanto, hay que añadir una condición adicional. Esa condición adicional puede ser, por ejemplo, el valor de la derivada en uno de los nodos o el valor de la función en un nodo auxiliar.

Spline cúbico de clase dos. Un *spline* cúbico de clase dos es una función formada por trozos de cúbicas (un polinomio de grado menor o igual que tres en cada subintervalo $[x_{i-1}, x_i]$, para $1 \leq i \leq n$) y que admite derivada segunda continua.

Ahora, la dimensión del espacio vectorial de los *splines* cúbicos de clase dos sobre $(n + 1)$ nodos es igual a $(n + 3)$. Por tanto, en este caso hay que añadir dos condiciones adicionales. Según qué condiciones adicionales se impongan, los tipos de *splines* cúbicos más habituales son

- **natural:** si imponemos derivada segunda nula en $x = a$ y en $x = b$;
- **periódico:** cuando imponemos que las derivadas primera y segunda coincidan en $x = a$ y en $x = b$ (se suele usar cuando los datos son periódicos, es decir, $y_0 = y_n$);
- **sujeto:** cuando prefijamos los valores de la primera derivada en $x = a$ y en $x = b$.

En ocasiones se usa el **spline cúbico de clase uno**, cuyo espacio vectorial tiene dimensión $(2n + 2)$. Este tipo de *spline* está asociado, en general, al problema de interpolación de Hermite clásico (6.7) en el que, para $(n + 1)$ nodos, tenemos exactamente $(2n + 2)$ datos. Por tanto, no hay que añadir ninguna condición adicional en este caso.

6.2. Implementación de algunos métodos

En este capítulo nos centraremos en el problema general de interpolación para datos lagrangianos.

6.2.1. Interpolación polinomial

En esta sección vamos a ver diferentes modos de resolver el problema asociado a los datos (6.1) del ejemplo 6.1.

Método de los coeficientes indeterminados (sistema de Vandermonde)

Si escribimos el polinomio interpolador, que será de grado menor o igual que 4, en la forma $p(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$, e imponemos las condiciones de interpolación, tenemos que sus coeficientes se corresponden con la solución del sistema

$$\begin{pmatrix} 0.50^4 & 0.50^3 & 0.50^2 & 0.50 & 1 \\ 0.51^4 & 0.51^3 & 0.51^2 & 0.51 & 1 \\ 0.52^4 & 0.52^3 & 0.52^2 & 0.52 & 1 \\ 0.53^4 & 0.53^3 & 0.53^2 & 0.53 & 1 \\ 0.54^4 & 0.54^3 & 0.54^2 & 0.54 & 1 \end{pmatrix} \begin{pmatrix} a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} 0.69146 \\ 0.69497 \\ 0.69847 \\ 0.70194 \\ 0.70540 \end{pmatrix}.$$

Para encontrar la solución de este sistema podemos emplear directamente los comandos que ya vienen implementados en Octave. La resolución del sistema se muestra en la siguiente secuencia de comandos.

```
> x = [0.50, 0.51, 0.52, 0.53, 0.54];
> y = [0.69146, 0.69497, 0.69847, 0.70194, 0.70540];
> A = x'.^[4:-1:0];
> p = (A\y)';
> polyout(p, 'x')
166.67*x^4 - 346.67*x^3 + 270.23*x^2 - 93.217*x^1 + 12.658
```

Observamos cómo se aprovecha la forma vectorizada de la operación potencia ($.$ $^$) para calcular la matriz de coeficientes. Recordamos además que la matriz de coeficientes del

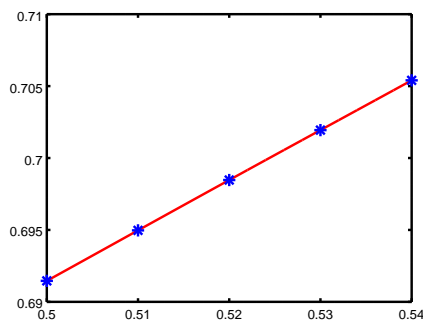


Figura 6.1: Resultado de la interpolación de los datos de la tabla (6.1).

sistema anterior es una *matriz de Vandermonde*, por lo que el comando directo de Octave, `vander(x, length(x))`, también nos proporcionaría la matriz de coeficientes. En la figura 6.1 vemos representados los puntos que hemos interpolado junto con el polinomio interpolador en el intervalo $[0.50, 0.54]$. Dicha gráfica se ha obtenido ejecutando los comandos


```
> z = 0.50:0.002:0.54;
> pz = polyval(p, z);
> plot(z, pz, 'r', x, y, '.*b')
```

Cálculo de las diferencias divididas

Si usamos una base de Newton de $\mathbb{P}_4[x]$ para los datos de la tabla (6.1), el polinomio interpolador viene dado por una expresión como en (6.5), es decir

$$\begin{aligned}
 p(x) = & f[0.50] + f[0.50, 0.51](x - 0.50) \\
 & + f[0.50, 0.51, 0.52](x - 0.50)(x - 0.51) \\
 & + f[0.50, 0.51, 0.52, 0.53](x - 0.50)(x - 0.51)(x - 0.52) \\
 & + f[0.50, 0.51, 0.52, 0.53, 0.54](x - 0.50)(x - 0.51)(x - 0.52)(x - 0.53),
 \end{aligned} \tag{6.11}$$

donde los coeficientes $f[*]$ son las diferencias divididas correspondientes.

En el siguiente código mostramos cómo obtener las diferencias divididas y, además, manipular la expresión (6.11) para devolver el polinomio interpolador en forma de vector, siguiendo la sintaxis usual de Octave para manejar polinomios (línea 11).

```
1 function p = difdiv(x, y)
2 % Esta función calcula el polinomio interpolador p mediante
3 % diferencias divididas para datos lagragianos.
4 n = length(y)-1; % n+1 es el número de nodos
5 d = y;          % Columna con diferencias divididas de orden 0
6 p = d(1);       % Polinomio que interpola en el primer nodo
7 for k = 1:n
8   % Columna con diferencias divididas de orden k
9   d = (d(2:end)-d(1:end-1))./(x(k+1:end)-x(1:end-k));
10  % Polinomio que interpola los k+1 primeros datos.
11  p = [0, p]+d(1)*poly(x(1:k));
12 endfor
13 endfunction
```

Observemos que el comando “+” funciona en Octave solo con vectores de la misma longitud. Por este motivo, para sumar polinomios de grado distinto, en la línea 11 hemos añadido un cero a la representación vectorial de un cierto polinomio (véase el ejercicio §1.13). La función `difdiv` se puede adaptar para realizar interpolación con datos tipo Hermite (véase el ejercicio §6.59).

Para los datos de la tabla (6.1), obtenemos el resultado de la siguiente forma.

```
> p = difdiv(x, y);
> polyout(p, 'x')
166.67*x^4 - 346.67*x^3 + 270.23*x^2 - 93.217*x^1 + 12.658
```

El comando `polyfit`

Para calcular el polinomio de grado menor o igual que n que interpola unos datos de tipo lagrangiano también podemos usar el comando `polyfit`. Para aplicarlo a nuestro ejemplo, solo tenemos que ejecutar

```
> p = polyfit(x, y, 4);
```

y obtenemos exactamente la misma respuesta.

Debemos tener en cuenta que, como veremos en el capítulo 7, el comando `polyfit` en realidad proporciona el polinomio (del grado que se especifique en su tercer argumento) que mejor aproxima unos datos en el sentido de los mínimos cuadrados. Por este motivo, `polyfit` no es válido para otros tipos de interpolación (Hermite, *splines*, etc.). No obstante, como podemos ver en [5, Interpolation] existen otros comandos directos para estos casos.

Limitaciones de la interpolación polinomial en la aproximación de funciones

Vamos a visualizar el fenómeno de Runge, que ya hemos comentado anteriormente en la sección 6.1.3, para comprobar que la interpolación con polinomios no siempre nos proporcionará los resultados deseados. Para comprender el *script* que ejecutaremos más adelante, comenzamos calculando el polinomio que interpola la función $f(x) = \frac{1}{1+25x^2}$ en cuatro nodos equidistantes del intervalo $[-1, 1]$.

```
> x = linspace(-1, 1, 4); y = 1 ./ (1+25*x.^2);
> p = polyfit(x, y, 3);
```

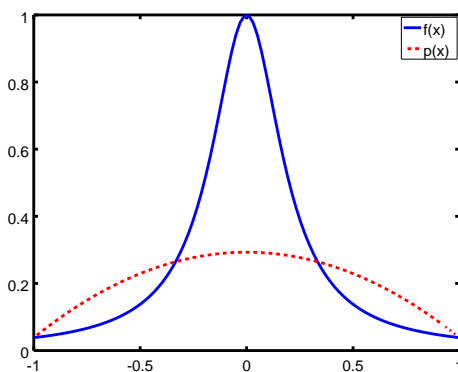


Figura 6.2: Interpolación de la función de Runge en 4 nodos equiespaciados.

Ahora representamos la función de Runge $f(x)$ y el polinomio interpolador $p(x)$ para comparar ambas gráficas. El resultado obtenido puede verse en la figura 6.2.

```
> z = linspace(-1,1,100);
> pz = polyval(p,z);
> plot(z,1./(1+25*z.^2),'b',z,pz,':r')
```

Ya estamos preparados para observar el conocido como fenómeno de Runge, esto es, vamos a ver que, aunque tengamos un mayor número de nodos, no siempre conseguiremos una mejora en el resultado de la interpolación. Muy al contrario, veremos que el error crece ilimitadamente. Para ello usaremos el siguiente *script*.

```
1 % Función de Runge
2 f = @(x) 1./(1+25*x.^2);
3 % Función que evalúa el polinomio interpolador de grado n.
4 function v = pz(z,n)
5     x = linspace(-1,1,n+1); % n+1 nodos equiespaciados
6     y = 1./(1+25*x.^2);
7     p = polyfit(x,y,n);
8     v = polyval(p,z);
9 endfunction
10 z = linspace(-1,1,100);
11 plot(z,f(z),'g',z,pz(z,8),'—b',z,pz(z,14),'-r')
```

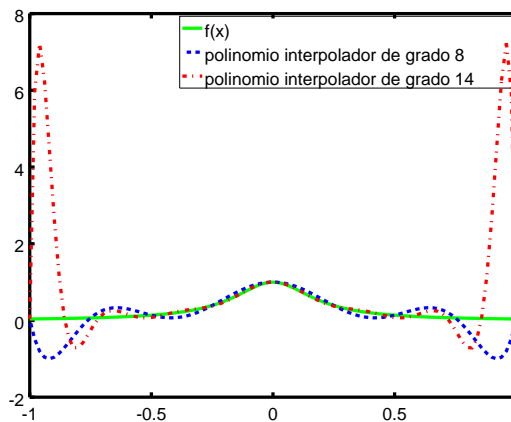


Figura 6.3: Fenómeno de Runge

Vemos la salida correspondiente en la figura 6.3, donde observamos las gráficas de $f(x)$, del polinomio interpolador de grado 8 y del polinomio interpolador de grado 14.

Como podemos comprobar, la diferencia entre el valor de la función y sus polinomios interpoladores se va haciendo muy grande en algunos puntos cercanos a los extremos del intervalo. Esta situación se puede mejorar usando aproximación en vez de interpolación, empleando nodos adecuados que no estén equiespaciados o mediante interpolación *spline*.

6.2.2. Interpolación *spline*

A continuación vamos a ver cómo programar la resolución de algunos problemas de interpolación *spline* en dimensión uno con Octave. Aunque existen comandos predefinidos para hacer esta labor, y que se pueden consultar en [5, *Polynomial Manipulations. Polynomial interpolation*], nosotros construiremos nuestras propias funciones y *scripts*.

Como ya hemos comentado, una función *spline* no es más que una función a trozos constituida por polinomios de cierto grado máximo prefijado, y de forma que, globalmente, la función tenga cierta regularidad.

Para visualizar un primer ejemplo de este tipo de funciones no tenemos más que invocar al comando `plot`. Dicho comando nos proporciona, justamente, la representación de un *spline* lineal que pasa por los puntos indicados. Así, si ejecutamos

```
> x = linspace(-1,1,9);
> y = 1./(1+25*x.^2);
> plot(x,y, '.*b', x,y, 'r');
```

obtenemos la gráfica del *spline* lineal que aproxima la función de Runge $f(x) = \frac{1}{1+25x^2}$ en nueve nodos equiespaciados del intervalo $[-1, 1]$, cuyo resultado gráfico puede verse

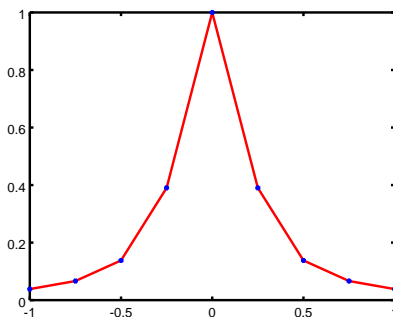


Figura 6.4: Interpolación mediante el *spline* lineal

en la figura 6.4. El cálculo concreto del *spline* lineal está propuesto en el ejercicio §6.60.

Interpolación *spline* cuadrática (de clase uno)

Vamos a suponer que, además de los valores de la función a interpolar en los nodos $x_0 < x_1 < \dots < x_n$ (que suponemos distintos y ordenados), conocemos el valor de la derivada de dicha función en el primer nodo, $f'(x_0) = d_0$. Para construir el *spline* cuadrático de clase uno correspondiente podemos seguir diversos caminos, dependiendo de cómo decidamos expresar dicho *spline*. Por ejemplo, podemos usar una base del espacio de *splines* cuadráticos mediante unas funciones *spline* especiales llamadas potencias truncadas,

$$(x-a)_+^k := \begin{cases} (x-a)^k & \text{si } x \geq a, \\ 0 & \text{si } x < a. \end{cases}$$

Es fácil ver que la potencia truncada de orden k es una función de clase $k-1$.

Usando la base de potencias truncadas, el *spline* cuadrático de clase uno puede ser expresado como (véase [16]),

$$s(x) = \alpha + \beta(x-x_0) + \sum_{i=0}^{n-1} \gamma_i (x-x_i)_+^2. \quad (6.12)$$

Los coeficientes de (6.12) se determinarán imponiendo las condiciones de interpolación $s(x_i) = y_i, 0 \leq i \leq n$, así como el dato extra introducido, esto es, $s'(x_0) = d_0$.

Una segunda opción es definir el *spline* directamente, trozo a trozo, tal como sigue

$$s(x) = \begin{cases} s_i(x), & \text{si } x \in [x_{i-1}, x_i], \quad i = 1, \dots, n, \end{cases} \quad (6.13)$$

con $s_i(x) = y_{i-1} + d_{i-1}(x-x_{i-1}) + z_i(x-x_{i-1})^2$, de modo que se verifica $s_i(x_{i-1}) = y_{i-1}$ y $s'_1(x_0) = d_0$. A continuación, tendremos que imponer las condiciones de interpolación, $s_i(x_i) = y_i$ para $1 \leq i \leq n$, y de regularidad sobre los nodos intermedios, $s'_i(x_i) = d_i$ para $1 \leq i \leq n-1$, para obtener los valores de los coeficientes d_1, \dots, d_{n-1} y z_1, \dots, z_n .

Construcción con base de potencias truncadas. De la expresión (6.12) deducimos directamente los valores de α y β , ya que $s(x_0) = y_0 = \alpha$ y $s'(x_0) = d_0 = \beta$. Por otro lado, los coeficientes γ_j los obtendremos recursivamente, evaluando el *spline* en cada nodo x_{j+1} , y deduciendo que, para $0 \leq j \leq n-1$,

$$y_{j+1} = s(x_{j+1}) = \alpha + \beta(x_{j+1} - x_0) + \gamma_j(x_{j+1} - x_j)^2 + \sum_{i=0}^{j-1} \gamma_i(x_{j+1} - x_i)^2,$$

lo que permite despejar γ_j una vez conocidos $\gamma_0, \dots, \gamma_{j-1}$.

En el siguiente *script* aplicamos esta construcción para mejorar el resultado de la interpolación de la función $f(x) = \frac{1}{1+25x^2}$ en $[-1, 1]$ tomando veinte nodos equiespaciados. Para guardar y, posteriormente, representar el *spline* interpolador, hemos optado por usar una matriz, que denotaremos s , de tamaño 19×3 y cuyas filas contienen los coeficientes del polinomio correspondiente a cada subintervalo.

```

1 f = @(x) 1./(1+25*x.^2); % Definimos la función de Runge
2 n = 19; % Número de subintervalos (n+1 nodos)
3 x = linspace(-1,1,n+1); % Nodos equiespaciados
4 y = f(x); % Valores de la función en los nodos
5 d0 = 25/338; % Derivada en el primer nodo
6 p = [0,0,y(1)]+[0,d0*poly(x(1))]; % Término lineal del spline
7 s = zeros(n,3); % Matriz que guarda, por filas, los coeficientes
8 % del polinomio de cada trozo
9 gam = []; % Vector que guarda los valores de las gammas
10 for j = 1:n % Determinamos las n gammas
11     pot = poly([x(j),x(j)]); % Potencia truncada cuadrática en x(j)
12     gam(j) = (y(j+1)-polyval(p,x(j+1)))/polyval(pot,x(j+1));
13     p = p+gam(j)*pot;
14     s(j,:) = p;
15 endfor
16 plotspline(s,x)
17 hold on
18 plot(x,y,'.*r');
19 hold off

```

Recordemos que el comando `hold on` fija como ventana gráfica activa la que, en ese momento, está en pantalla. A continuación, todos los gráficos que realicemos se incluirán en dicha ventana. El efecto de este comando se deshace ejecutando `hold off`.

Para representar el *spline* en el *script* anterior, usamos la función `plotspline`, que hemos construido nosotros y cuyo código mostramos a continuación.

```

1 function [] = plotspline(p,x)
2 % Esta función representa en cada intervalo [x(i),x(i+1)]
3 % el polinomio p(i,:) empleando el comando plot.
4 n = length(x)-1; % Número de subintervalos
5 for i = 1:n
6     z = linspace(x(i),x(i+1),20);
7     plot(z,polyval(p(i,:),z));
8     hold on
9 endfor
10 hold off
11 endfunction

```

Con todo ello, observamos en la figura 6.5 la salida del *script* anterior, que muestra tanto los datos usados para la interpolación como el *spline* cuadrático interpolador. No obstante, aunque los resultados son satisfactorios, tenemos que señalar que trabajar con bases de potencias truncadas puede ser numéricamente inestable cuando el *spline* no se puede construir paso a paso, pues para el cálculo de los γ_j necesitaríamos la resolución de un sistema de ecuaciones lineales que, normalmente, está mal condicionado (véase [16]). Por este motivo, es común usar el enfoque que presentamos a continuación.

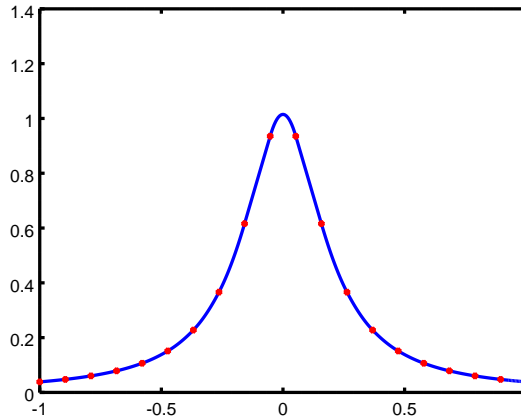


Figura 6.5: Interpolación mediante *spline* cuadrático.

Construcción trozo a trozo. En este segundo enfoque vamos a proponer una construcción directa, como una función a trozos, de la forma (6.13). En este caso, si definimos los valores $h_i = x_i - x_{i-1}$ y $w_i = \frac{y_i - y_{i-1}}{h_i}$, para $1 \leq i \leq n$, e imponemos las condiciones de interpolación y de regularidad, deducimos una expresión recursiva de los coeficientes d_i y z_i del *spline*. En efecto,

$$\left. \begin{array}{l} s_i(x_i) = s_{i+1}(x_i) = y_i \\ s'_i(x_i) = s'_{i+1}(x_i) = d_i \end{array} \right\} \Rightarrow \left. \begin{array}{l} y_{i-1} + d_{i-1}h_i + z_i h_i^2 = y_i \\ d_{i-1} + 2z_i h_i = d_i \end{array} \right\} \Rightarrow \left. \begin{array}{l} z_i = \frac{w_i - d_{i-1}}{h_i} \\ d_i = 2w_i - d_{i-1} \end{array} \right\}$$

para $i = 1, \dots, n-1$. Observamos que, por la condición de interpolación en el último nodo, $s_n(x_n) = y_n$, tenemos que la expresión anterior es también cierta para z_n . Guardaremos los resultados en los vectores $h = (h_1, \dots, h_n)$, $w = (w_1, \dots, w_n)$, $d = (d_0, \dots, d_{n-1})$ y $z = (z_1, \dots, z_n)$, con el fin de calcular, en una segunda etapa, el *spline* correspondiente. Para ello haremos uso del comando `diff` (véase capítulo 8), que es muy útil para implementar los vectores que involucran diferencias como $h_i = x_i - x_{i-1}$; de hecho, `diff(x)` realiza precisamente la operación $(x(2:\text{end}) - x(1:\text{end}-1))$. La función considerada y los valores de los nodos son los mismos que el caso anterior.

```
> n = length(x)-1;
> h = diff(x);
> w = diff(y)./h;
> d = [d0];
> for i = 1:n-1 d = [d, 2*w(i)-d(i)]; endfor
> z = (w-d)./h
```

Una vez calculados estos vectores podemos hallar la función *spline*, que guardaremos de nuevo en una matriz de orden $n \times 3$, cuya fila i -ésima contendrá los coeficientes del polinomio $s_i(x)$. Dicho polinomio podemos reescribirlo como

$$s_i(x) = (y_{i-1} - d_{i-1}x_{i-1} + z_i x_{i-1}^2) + (d_{i-1} - 2z_i x_{i-1})x + z_i x^2,$$

por lo que la programación vectorizada del *spline* cuadrático de clase uno, que mostramos a continuación, es muy simple.

```
> ax = x(1:end-1);
> ay = y(1:end-1);
> s = [z ; d-2*z.*ax ; ay-d.*ax+z.*ax.^2]';
```

Interpolación con *spline* cúbico sujeto

Siguiendo la misma idea que hemos empleado en la construcción directa, trozo a trozo, del *spline* cuadrático, nos proponemos interpolar los valores $\{(x_0, y_0), \dots, (x_n, y_n)\}$ mediante un *spline* cúbico de clase dos. Suponemos aquí también que los nodos son distintos dos a dos y están ordenados, $x_0 < \dots < x_n$. En cada intervalo $[x_{i-1}, x_i]$, $1 \leq i \leq n$, este *spline* viene dado por un polinomio de grado tres que puede ser escrito como

$$s_i(x) = y_{i-1} + d_{i-1}(x - x_{i-1}) + z_i(x - x_{i-1})^2 + t_i(x - x_{i-1})^2(x - x_i). \quad (6.14)$$

Al igual que en el apartado anterior, denotamos por $h_i = x_i - x_{i-1}$ y $w_i = \frac{y_i - y_{i-1}}{h_i}$, para $1 \leq i \leq n$. Ahora, imponiendo tanto la continuidad del *spline* como la continuidad de su derivada primera en los nodos intermedios, se deducen las relaciones

$$z_i = \frac{w_i - d_{i-1}}{h_i}, \quad t_i = \frac{d_{i-1} - 2w_i + d_i}{h_i^2}, \quad i = 1, \dots, n.$$

De esta forma, el *spline* cúbico sujeto quedará determinado en cuanto calculemos los valores de las derivadas primeras en los nodos intermedios, esto es, los coeficientes d_1, \dots, d_{n-1} . Observemos que las condiciones de *spline* sujeto, $s'(x_0) = \alpha_0$ y $s'(x_n) = \alpha_n$, determinan directamente el primero y el último de los coeficientes, $d_0 = \alpha_0$ y $d_n = \alpha_n$. A diferencia del caso anterior, ahora los valores d_i no se pueden calcular de forma recurrente, sino que se obtienen como solución del siguiente sistema de ecuaciones lineales tridiagonal simétrico

$$\begin{pmatrix} \frac{2}{h_1} + \frac{2}{h_2} & \frac{1}{h_2} & & & \\ \frac{1}{h_2} & \frac{2}{h_2} + \frac{2}{h_3} & \frac{1}{h_3} & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \frac{1}{h_{n-1}} \\ & & & \frac{1}{h_{n-1}} & \frac{2}{h_{n-1}} + \frac{2}{h_n} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{pmatrix} = \begin{pmatrix} \frac{3w_1}{h_1} + \frac{3w_2}{h_2} - \frac{d_0}{h_1} \\ \frac{3w_2}{h_2} + \frac{3w_3}{h_3} \\ \vdots \\ \frac{3w_{n-2}}{h_{n-2}} + \frac{3w_{n-1}}{h_{n-1}} \\ \frac{3w_{n-1}}{h_{n-1}} + \frac{3w_n}{h_n} - \frac{d_n}{h_n} \end{pmatrix}, \quad (6.15)$$

que proviene de imponer clase dos en los nodos intermedios, esto es, $s_i''(x_i) = s_{i+1}''(x_i)$, para $1 \leq i \leq n-1$.

Para mejorar el resultado de la interpolación polinomial en el ejemplo de Runge, debemos considerar los valores $\alpha_0 = 0.073964497$ y $\alpha_n = -0.073964497$. Procedemos pues a introducir el sistema (6.15):

```
> n = 8;
> x = linspace(-1,1,n+1);
> y = 1./(1+25*x.^2);
> h = diff(x);
> w = diff(y)./h;
> A = diag(2./h(1:n-1)+2./h(2:n),0)...
+diag(1./h(2:n-1),1)+diag(1./h(2:n-1),-1);
> b = 3*(w(1:n-1)./h(1:n-1)+w(2:n)./h(2:n));
> d0 = 0.073964497;
> dn = -0.073964497;
> b(1) = b(1)-d0./h(1);
> b(end) = b(end)-dn./h(end);
```

y a resolverlo, obteniendo el vector vector de derivadas $d = (d_0, \dots, d_n)$. A continuación, generamos los coeficientes $z = (z_1, \dots, z_n)$ y $t = (t_1, \dots, t_n)$ (por cierto, debemos tener en cuenta que, en este caso, el vector d tiene una componente más que en el caso cuadrático, pues ahora necesitamos el valor d_n para el cálculo de todos los coeficientes).

```
> d = A\b';
> d = [d0,d',dn];
> z = (w-d(1:n))./h;
> t = (d(1:n)-2*w+d(2:n+1))./(h.^2);
```

Finalmente, definimos el *spline* cúbico a partir de (6.14). En este caso el cálculo del *spline* no lo hacemos de forma vectorizada, como sí hicimos con el cuadrático (véase el ejercicio §6.63).

```
> s = zeros(n,4);
> for i = 1:n
    s(i,:) = [0,0,0,y(i)]+[0,0,d(i)*poly([x(i)])]...
    +[0,z(i)*poly([x(i),x(i)])]...
    +t(i)*poly([x(i),x(i),x(i+1)]);
endfor
```

Para terminar, podemos representar el *spline* con nuestro comando `plotspline`, junto con los puntos a interpolar.

```
> plotspline(s,x)
> hold on
```

```
> plot(x,y, '.*r')  
> hold off
```

Obtenemos así la siguiente gráfica.

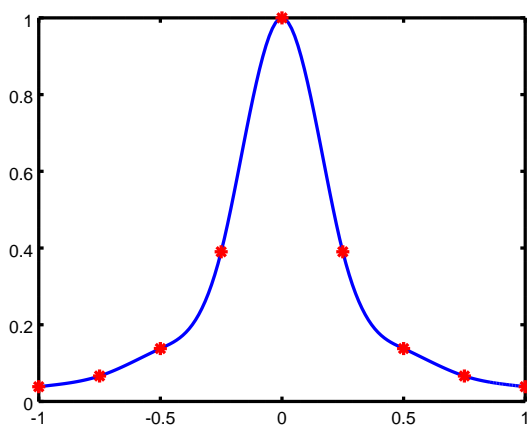


Figura 6.6: Interpolación mediante *spline* cúbico sujeto.

Capítulo 7

Aproximación por mínimos cuadrados discreta y continua

7.1. Fundamentos teóricos

La idea básica de la aproximación es obtener, a partir de una serie de datos (bien una nube de muchos puntos, bien datos provenientes de una cierta relación funcional), una función cuya gráfica pase “lo más cerca posible” de tales datos.

En principio, el problema planteado es parecido al de interpolación, pero hay una diferencia importante que se refleja justamente en la primera exigencia realizada. A saber,

- Interpolación: la gráfica de la función interpoladora pasa exactamente por los datos.
- Aproximación: la gráfica de la función aproximante pasa “cerca” de los datos.

Veamos cómo, en el ejemplo representado en la figura 7.1 y realizado con Octave, aparecen los siguientes elementos:

- 11 datos marcados con aspas y el polinomio interpolador de grado 10 correspondiente. Como vemos, parece “no respetar” la función “sugerida” por los datos.
- El *spline* lineal que interpola los datos y sí parece ajustarse al perfil de los datos, pero con “picos”.
- Por último, el polinomio de grado 3 que mejor aproxima a los datos, en el sentido que veremos es este capítulo. Es cierto que no pasa exactamente por los datos, pero sí parece representar, mejor que los anteriores, a la función sugerida por los puntos. Además, es sencillo y regular.

Como hemos dicho, los datos pueden provenir de una relación funcional o simplemente ser una serie de valores discretos. Atendiendo a esta naturaleza, tenemos dos posibilidades a la hora de abordar el problema de aproximación.

- Aproximación discreta: el conjunto de datos es finito.

- Aproximación continua: el dato es una función conocida (pero complicada de manejar) en todo un intervalo.

En ambos casos, nuestro objetivo será encontrar una función que sea fácil de calcular, fácil de evaluar, suficientemente derivable, etc.

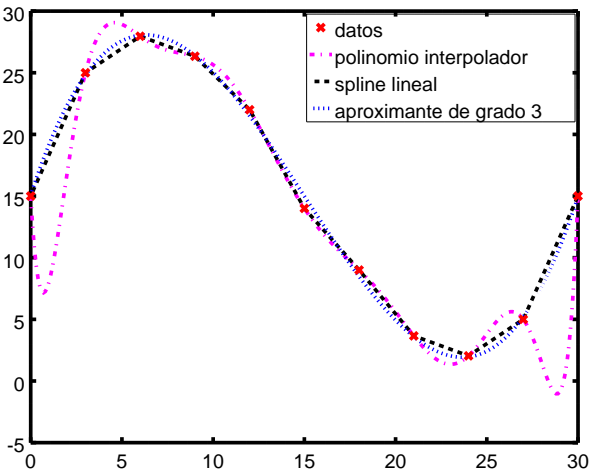


Figura 7.1: Aproximación vs interpolación.

La principal cuestión que surge es cómo decidir cuándo una función aproxima mejor que otra un determinado conjunto de datos. A tal fin, definiremos una función error, de manera que el problema planteado equivalga a medir cuánto nos equivocamos con cada posible función y, así, escoger la que tiene el error “más pequeño”. En la siguiente tabla se recogen varias posibilidades para definir la función error, tanto en el caso discreto como en el continuo. En ambos casos, llamamos $u(x)$ a la función aproximante.

		Aproximación discreta	Aproximación continua
Datos		$\{(x_0, y_0), \dots, (x_n, y_n)\}$	$f : [a, b] \rightarrow \mathbb{R}$
Función error		$e_i = u(x_i) - y_i $	$e(x) = u(x) - f(x) $
Tipos	Mínima suma	$\sum_{i=0}^n e_i$	$\int_a^b e(x) \, dx$
	Mínimos cuadrados	$\sum_{i=0}^n e_i^2$	$\int_a^b e^2(x) \, dx$
	Minimax	$\max_{0 \leq i \leq n} e_i$	$\sup_{x \in [a, b]} e(x)$

Por cuestiones de simplicidad, en este manual sólo estudiaremos el caso del ajuste por mínimos cuadrados.

7.1.1. Ajuste por mínimos cuadrados discreto mediante polinomios

Para entender mejor cómo se define el error cuadrático y cómo funciona la aproximación por mínimos cuadrados discreta, introduciremos varios conceptos a través de un ejemplo concreto.

Consideremos la serie de cinco datos $\{(1, 1.15), (2, 0.4), (3, 2), (4, 2.6), (5, 1.9)\}$. La nube de puntos para estos datos viene dada en la figura 7.2, en la que también aparece representada la recta $f(x) = 0.5x$, que pretende ser una aproximación de estos datos. Al

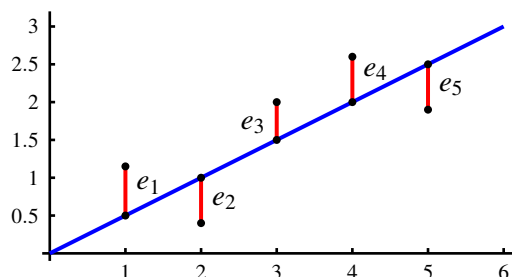


Figura 7.2: Ejemplo de aproximación por mínimos cuadrados discreta.

usar una recta arbitraria cualquiera de la forma $y = a + bx$, el error cuadrático cometido, expresado en función de los coeficientes a y b de la recta, vendría dado por

$$\begin{aligned} E_2(a, b) &= e_1^2 + e_2^2 + e_3^2 + e_4^2 + e_5^2 \\ &= (a + b - 1.15)^2 + (a + 2b - 0.4)^2 + (a + 3b - 2)^2 \\ &\quad + (a + 4b - 2.6)^2 + (a + 5b - 1.9)^2. \end{aligned}$$

De esta forma, hemos transformado el problema “encontrar la recta que ajusta los datos del ejemplo con el menor error cuadrático” en el problema “buscar el mínimo de la función de dos variables $E_2(a, b)$ ”. Por ejemplo, al tomar la función $f(x) = 0.5x$ como aproximación de estos datos, esto es, tomar $a = 0.5$ y $b = 0$, tenemos que el error cuadrático cometido es $E_2(0.5, 0) = 1.7525$.

Puesto que $E_2(a, b)$ es una función polinómica de dos variables y de grado dos, y además tiene coeficientes positivos en los términos a^2 y b^2 (es decir, es un “paraboloide hacia arriba”, algo así como una “parábola tridimensional hacia arriba”), sabemos que tiene un mínimo absoluto, que será por lo tanto un extremo relativo y se calculará como punto crítico. Si somos capaces de calcularlo, habremos dado con los coeficientes de la recta que mejor ajusta los datos.

Para calcular este punto crítico (sólo habrá uno en la mayoría de los casos) recurrimos a las derivadas parciales de la función $E_2(a, b)$. Así, tenemos que resolver el sistema

$$\left. \begin{aligned} \frac{\partial E_2}{\partial a}(a, b) &= 0 \\ \frac{\partial E_2}{\partial b}(a, b) &= 0 \end{aligned} \right\}. \quad (7.1)$$

En nuestro ejemplo, tras realizar las operaciones y simplificaciones adecuadas, el sistema (7.1) se reduce a

$$\left. \begin{aligned} 5a + 15b &= 8.05 \\ 15a + 55b &= 27.85 \end{aligned} \right\}. \quad (7.2)$$

O sea, hemos obtenido un sistema lineal, de dos ecuaciones con dos incógnitas, cuya solución es $a = 0.5$, $b = 0.37$. Por consiguiente, la recta que mejor ajusta por mínimos cuadrados los datos del ejemplo está dada por la expresión $y = 0.5 + 0.37x$. Para esta recta, el error cuadrático cometido es $E_2(0.5, 0.37) = 1.5230$. Por supuesto, este error es menor que el correspondiente a la recta $y = 0.5x$.

Ahora que hemos comprendido este sencillo ejemplo, vamos a describir este procedimiento de modo general. Como hemos visto, cuando buscamos la recta $y = ax + b$ que mejor ajusta por mínimos cuadrados a una serie de datos, todo se reduce a plantear (y resolver) el sistema que han de verificar los coeficientes a y b de dicha recta. Hay varias formas de obtener fácilmente este sistema y que, además, se generalizan de manera muy intuitiva cuando pretendemos aproximar por mínimos cuadrados discretos usando cualquier otra familia de funciones que dependa linealmente de ciertos parámetros. En particular, para aproximar con polinomios de cualquier grado prefijado.

Primer procedimiento: uso de tablas

Consiste en construir el sistema a partir de una tabla cuyas columnas corresponden a los valores de x , y , x^2 y, por último, xy , y calcular la suma de los elementos de cada columna. El porqué de estas tablas se deduce de observar, en detalle, el sistema que se obtiene para un caso general. Si escribimos para un conjunto genérico de datos $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ el sistema (7.2) obtenido en el ejemplo, vemos que viene dado por

$$\left. \begin{aligned} (n+1)a + (\sum x_i)b &= \sum y_i \\ (\sum x_i)a + (\sum x_i^2)b &= \sum (x_i y_i) \end{aligned} \right\}, \quad (7.3)$$

por lo que observamos que las sumas que proporcionan los coeficientes del sistema son, precisamente, las que hemos mencionado anteriormente. Aquí, los índices de las sumatorias barren los $n+1$ datos del problema. En nuestro ejemplo, la tabla resultante sería la siguiente,

	x_i	y_i	x_i^2	$x_i y_i$
	1	1.15	1	1.15
	2	0.4	4	0.8
	3	2	9	6
	4	2.6	16	10.4
	5	1.9	25	9.5
Suma	15	8.05	55	27.85

cuyos valores se corresponden con los acaecidos en el sistema (7.2).

Como hemos dicho anteriormente, si, en lugar de con una recta, queremos **ajustar mediante una parábola** $y = a + bx + cx^2$, obtendremos un sistema 3×3 para calcular valores de los coeficientes a , b y c . Operando de forma análoga a como hemos hecho en el caso previo, este sistema puede ser fácilmente descrito como

$$\left. \begin{aligned} (n+1)a + (\sum x_i)b + (\sum x_i^2)c &= \sum y_i, \\ (\sum x_i)a + (\sum x_i^2)b + (\sum x_i^3)c &= \sum (x_i y_i) \\ (\sum x_i^2)a + (\sum x_i^3)b + (\sum x_i^4)c &= \sum (x_i^2 y_i) \end{aligned} \right\}$$

por lo que, para calcular ahora todos los coeficientes, simplemente tenemos que construir una tabla con siete columnas correspondientes a los valores de x , y , x^2 , x^3 , x^4 , xy , x^2y .

A partir de lo visto para la recta y la parábola, es relativamente fácil generalizar estos resultados para calcular el sistema y la tabla necesarios para ajustar un conjunto de datos discretos mediante polinomios de cualquier grado por mínimos cuadrados. Dicha generalización queda propuesta en el ejercicio §7.67.

Segundo procedimiento: uso de matrices

Para entender este procedimiento, primero supongamos que los datos de nuestro ejemplo estuvieran completamente alineados. Entonces existiría una recta $y = a + bx$ para la que serían ciertas las siguientes relaciones de igualdad,

$$\left. \begin{aligned} a + b \times 1 &= 1.15 \\ a + b \times 2 &= 0.4 \\ a + b \times 3 &= 2 \\ a + b \times 4 &= 2.6 \\ a + b \times 5 &= 1.9 \end{aligned} \right\} \Leftrightarrow \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1.15 \\ 0.4 \\ 2 \\ 2.6 \\ 1.9 \end{pmatrix} \Leftrightarrow A \begin{pmatrix} a \\ b \end{pmatrix} = y. \quad (7.4)$$

Sin embargo, visto como un sistema de ecuaciones, (7.4) ¡no tiene solución! (el par $(a, b) = (1.9, 0.75)$ es el único que satisface las dos primeras igualdades y no cumple las tres últimas). Por ello, la idea de este procedimiento consiste buscar la “mejor de las no-soluciones” de (7.4). Observemos que la matriz de coeficientes A es una **matriz de Vandermonde** que ya nos apareció anteriormente en el capítulo de interpolación, y que el vector de términos independientes se corresponde con los valores y_i de los datos.

En este caso el proceso consiste en multiplicar por la izquierda, en ambos lados del sistema (7.4), por A^t , la matriz transpuesta de la matriz de coeficientes, obteniendo así un sistema 2×2 que, en nuestro ejemplo, queda como

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \begin{pmatrix} 1.15 \\ 0.4 \\ 2 \\ 2.6 \\ 1.9 \end{pmatrix},$$

con lo que obtenemos la ecuación matricial

$$\begin{pmatrix} 5 & 15 \\ 15 & 55 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 8.05 \\ 27.85 \end{pmatrix},$$

que resulta ser, justamente, el sistema (7.2) en representación matricial. Por lo tanto, matricialmente, el sistema (7.3) a resolver puede ser obtenido como

$$A^t A \begin{pmatrix} a \\ b \end{pmatrix} = A^t y, \quad (7.5)$$

siendo A una matriz de Vandermonde con dos columnas.

Para el caso de la parábola que mejor se ajusta a los datos, se puede proceder de forma completamente análoga; entonces el sistema a resolver será

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 9 & 16 & 25 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 9 & 16 & 25 \end{pmatrix} \begin{pmatrix} 1.15 \\ 0.4 \\ 2 \\ 2.6 \\ 1.9 \end{pmatrix}.$$

En este caso, es un sistema 3×3 , pero obtenido del mismo modo matricial (7.5), siendo en este caso A una matriz de Vandermonde con tres columnas.

De nuevo es un ejercicio determinar, usando esta misma idea, que el sistema que permite calcular el polinomio de grado m que mejor aproxima por mínimos cuadrados es, precisamente, (7.5), siendo A en este caso una matriz de Vandermonde con $m + 1$ columnas. Asimismo es bastante fácil llegar al sistema que debemos resolver en el caso más general de aproximación mediante una familia de funciones que constituya un espacio vectorial cualquiera.

7.1.2. Solución generalizada de sistemas de ecuaciones lineales

La misma idea desarrollada en la sección previa nos puede servir de motivación para intentar “resolver”, cometiendo el mínimo error posible, un sistema de ecuaciones

lineales $Ax = b$ que, *a priori*, pueda ser incompatible o compatible indeterminado. En el primer caso, podríamos buscar el vector x que minimizase el error cometido al aproximar b por Ax , esto es, por ejemplo, minimizar $E(x) := \|Ax - b\|_2^2$; mientras que en el segundo caso se trataría de determinar, de entre todas las soluciones del sistema $Ax = b$, aquella que tuviese, por ejemplo, norma mínima. Veamos el resultado que sustenta la existencia de dichos mínimos (véase, por ejemplo, [20]).

Teorema 7.1.

■ *Hipótesis:*

- Sea una matriz A de orden $n \times m$ y de rango igual a m (su número de columnas);
- Sea un vector $b \in \mathbb{R}^n$ cualquiera.

■ *Tesis:*

- El sistema $(A^t A)x = A^t b$ tiene una única solución, x_0 .
- Si $Ax = b$ es incompatible, entonces x_0 minimiza $E(x) := \|Ax - b\|_2^2$.
- Si $Ax = b$ es compatible indeterminado, entonces x_0 es su solución con norma mínima.

Lo que hemos comentado en esta sección tiene también aplicación directa al ajuste por mínimos cuadrados discreto. De hecho, las hipótesis del teorema se satisfacen siempre que se verifiquen las dos siguientes condiciones.

C1. El número de datos es igual o superior al número de parámetros que determinan a cada función de la familia considerada (es decir, $n \geq m$).

C2. Al menos hay tantos valores distintos de abscisas como número de parámetros que determinan a la familia (es decir, el rango de A es m).

Por ejemplo, si queremos determinar la recta de mínimos cuadrados, entonces necesitaremos al menos dos datos con diferente abscisa, ya que una recta viene dada por dos parámetros. Y para una parábola (que está determinada por tres parámetros) necesitaremos al menos tres datos con abscisas diferentes.

Para concluir esta sección, es necesario decir que esto es precisamente lo que realiza internamente el comando `A\b` o `mldivide(A, b)` de Octave.

7.1.3. Ajuste discreto mediante funciones cualesquiera

Lo visto en las secciones 7.1.1 y 7.1.2 sólo podemos aplicarlo cuando, al realizar las derivadas parciales de la función error cuadrático, llegamos a un sistema lineal. Esto ocurre siempre que la familia de funciones para aproximar constituya un espacio vectorial. Sin embargo, si usamos otras familias de funciones que no dependan linealmente de los parámetros a ajustar, entonces el sistema resultante puede ser no lineal y, en general, imposible de resolver de manera exacta. Ante este problema planteamos dos posibles soluciones.

1. Resolver de manera aproximada el sistema.
2. Transformar los datos originales y las funciones de ajuste, mediante cambios de variable, de manera que el ajuste (por mínimos cuadrados) a los datos transformados sí resulte lineal.

El primer caso corresponde a los llamados **métodos de descenso** (por ejemplo, los de descenso rápido o los de gradiente conjugado), que consisten en calcular iterativamente una solución aproximada de forma que se reduzca el error asociado en cada iteración, de manera análoga a los métodos estudiados en la sección 4.1.2 para la resolución de sistemas de ecuaciones lineales. Una explicación más detallada de estos métodos excede los límites de estas notas, por lo que sólo desarrollaremos la segunda de las ideas.

La segunda vía se conoce como el **método de linealización** y es bastante más simple a la hora de comprender los cálculos a realizar, aunque proporciona un peor resultado (esto es, un error cuadrático mayor) que si usamos un método de descenso. De todas formas, la aproximación obtenida es, en la práctica, suficientemente buena. Veamos algunos de los ejemplos más habituales.

Aproximación exponencial. Consideremos que $\{(x_i, y_i) : i = 0, \dots, n\}$ (con $y_i \geq 0$) es el conjunto de datos que queremos aproximar mediante la familia de funciones exponenciales de la forma

$$\{y = ae^{bx} : a, b \in \mathbb{R}\}.$$

Para “eliminar” la no linealidad exponencial, introducimos el logaritmo neperiano y operamos como vemos a continuación.

$$y = ae^{bx} \Rightarrow \ln(y) = \ln(ae^{bx}) = \ln(a) + \ln(e^{bx}) = \ln(a) + bx \Rightarrow Y = A + bx,$$

donde $Y = \ln(y)$ y $A = \ln(a)$. Por lo tanto, cambiaremos el problema de ajustar los datos originales mediante exponenciales, al problema de ajustar los datos modificados $\{(x_i, Y_i) = (x_i, \ln(y_i)) : i = 0, \dots, n\}$ mediante una recta $Y = Ax + b$. Así, deshaciendo el cambio, el ajuste de los datos originales vendrá dado por $y = e^{A+bx}$.

Aproximación potencial. Si queremos aproximar los datos $\{(x_i, y_i) : i = 0, \dots, n\}$ (con $x_i, y_i \geq 0$) mediante la familia de funciones potenciales de la forma $\{y = ax^b : a, b \in \mathbb{R}\}$, usamos de nuevo la transformación dada por el logaritmo. Por lo tanto, debemos ajustar los datos transformados $\{(X_i, Y_i) = (\ln(x_i), \ln(y_i)) : i = 0, \dots, n\}$ por medio de una recta $Y = A + bX$ y, deshaciendo el cambio, el ajuste de los datos originales vendrá dado por $y = e^{A+b\ln(x)} = e^A x^b$.

Aproximación hiperbólica. Si aproximamos los datos $\{(x_i, y_i) : i = 0, \dots, n\}$ (con $x_i > 0$) mediante la familia de funciones hiperbólicas $\{y = a + \frac{b}{x} : a, b \in \mathbb{R}\}$, usamos la transformación $X = \frac{1}{x}$. En este caso, primero debemos ajustar, mediante una recta $y = a + bX$, los datos transformados $\{(X_i, y_i) = (\frac{1}{x_i}, y_i) : i = 0, \dots, n\}$. Deshaciendo el cambio, el ajuste de los datos originales vendrá dado por $y = a + \frac{b}{x}$.

Aproximación logística. Para aproximar el conjunto de datos $\{(x_i, y_i) : i = 0, \dots, n\}$ (con $0 < y_i < 1$) mediante la familia de funciones logísticas $\{y = \frac{1}{1+ae^{bx}} : a > 0, b \in \mathbb{R}\}$, combinamos las dos transformaciones anteriores y hacemos el cambio $Y = \ln(\frac{1}{y} - 1)$. Así, tomamos $\{(x_i, Y_i) = (x_i, \ln(\frac{1}{y_i} - 1)) : i = 0, \dots, n\}$ y ajustamos con $Y = A + bx$. En este caso, tras deshacer el cambio, queda el ajuste $y = \frac{1}{1+e^{A+bx}}$.

7.1.4. Ajuste de funciones por mínimos cuadrados continuo

Supongamos que el objeto que queremos aproximar ahora no es un conjunto discreto de puntos, sino que es una función $f : [a, b] \rightarrow \mathbb{R}$, y que queremos hacerlo mediante una combinación lineal de m funciones linealmente independientes $\phi_1(x), \phi_2(x), \dots, \phi_m(x)$, es decir,

$$f(x) \approx c_1\phi_1(x) + c_2\phi_2(x) + \dots + c_m\phi_m(x).$$

Entonces debemos minimizar la función de error cuadrático dada por:

$$E(c_1, c_2, \dots, c_m) = \int_a^b (c_1\phi_1(x) + c_2\phi_2(x) + \dots + c_m\phi_m(x) - f(x))^2 dx,$$

por lo que, de nuevo, buscamos los extremos relativos de dicha función. Tomando derivadas parciales y simplificando, el sistema de ecuaciones lineales que nos permite calcular los puntos críticos de $E(c_1, c_2, \dots, c_m)$ es

$$\left. \begin{aligned} c_1 \int_a^b \phi_1(x)\phi_1(x) dx + \dots + c_m \int_a^b \phi_1(x)\phi_m(x) dx &= \int_a^b \phi_1(x)f(x) dx \\ c_1 \int_a^b \phi_2(x)\phi_1(x) dx + \dots + c_m \int_a^b \phi_2(x)\phi_m(x) dx &= \int_a^b \phi_2(x)f(x) dx \\ &\vdots \\ c_1 \int_a^b \phi_m(x)\phi_1(x) dx + \dots + c_m \int_a^b \phi_m(x)\phi_m(x) dx &= \int_a^b \phi_m(x)f(x) dx \end{aligned} \right\}.$$

A la matriz de coeficientes asociada se le denomina **matriz de Gramm** respecto de la base $\{\phi_1(x), \phi_2(x), \dots, \phi_m(x)\}$.

Ejemplo 7.2. Queremos calcular el polinomio $p(x) = c_1 + c_2x + c_3x^2$ que mejor aproxima, mediante mínimos cuadrados continuo, la función $f(x) = |x|$ en el intervalo $[-1, 1]$. El sistema que debemos resolver es

$$\left. \begin{aligned} c_1 \int_{-1}^1 1 dx + c_2 \int_{-1}^1 x dx + c_3 \int_{-1}^1 x^2 dx &= \int_{-1}^1 |x| dx \\ c_1 \int_{-1}^1 x dx + c_2 \int_{-1}^1 x^2 dx + c_3 \int_{-1}^1 x^3 dx &= \int_{-1}^1 x|x| dx \\ c_1 \int_{-1}^1 x^2 dx + c_2 \int_{-1}^1 x^3 dx + c_3 \int_{-1}^1 x^4 dx &= \int_{-1}^1 x^2|x| dx \end{aligned} \right\} \Rightarrow \left. \begin{aligned} 2c_1 + 0c_2 + \frac{2}{3}c_3 &= 1 \\ 0c_1 + \frac{2}{3}c_2 + 0c_3 &= 0 \\ \frac{2}{3}c_1 + 0c_2 + \frac{2}{5}c_3 &= \frac{1}{2} \end{aligned} \right\}.$$

En este ejemplo, el único extremo relativo es, justamente, el mínimo absoluto de la función error, y viene dado por $p(x) = \frac{3+15x^2}{16}$, $\forall x \in [-1, 1]$. En la figura 7.3 representamos el resultado obtenido.

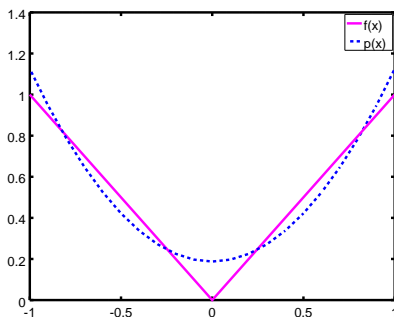


Figura 7.3: $p(x) = \frac{3+15x^2}{16}$ como aproximación en $[-1, 1]$ de $f(x) = |x|$.

7.2. Implementación de algunos métodos

7.2.1. Aproximación por mínimos cuadrados discreta

Consideremos el conjunto de datos dado por la siguiente tabla.

x	1	2	3	4	5	6	7
y	11	15	20	23	26	27	30

Para calcular la recta que mejor ajusta por mínimos cuadrados, empezamos definiendo los vectores x e y y construyendo la matriz de coeficientes. En este caso usaremos la función `vander` para definir la matriz de coeficientes.

```
> x = [1, 2, 3, 4, 5, 6, 7];
> y = [11, 15, 20, 23, 26, 27, 30];
> A = vander(x, 2);
```

A continuación resolvemos el sistema de ecuaciones lineales, con matriz de coeficientes A y término independiente y' , mediante el operador `\` y representamos gráficamente el polinomio obtenido junto con los datos aproximados. Para ello ejecutamos las siguientes órdenes.

```
> p = (A \ y')';
> polyout(p, 't')
3.1071*t^1 + 9.2857
```

```
> z = 0:0.1:8; pz = polyval(p,z);  
> plot(x,y,'xb',z,pz,'cr')
```

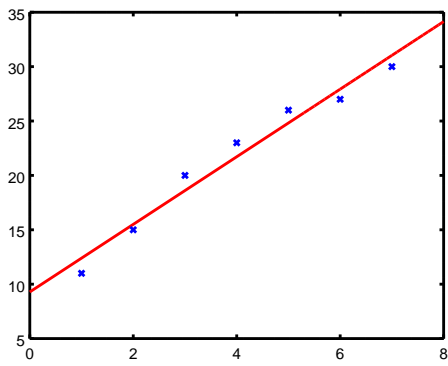


Figura 7.4: Aproximación lineal

Tal como comentamos en el capítulo 6, el comando `polyfit` también sirve para generar el polinomio (del grado que se especifique) que mejor aproxima unos datos en el sentido de los mínimos cuadrados. Concretamente, para datos x e y de longitud $n + 1$ (en este caso es preciso que todos los nodos sean distintos: $x_i \neq x_j$ si $i \neq j$), el comando `polyfit(x,y,n)` proporciona el polinomio de interpolación (de grado n), pero `polyfit(x,y,m)` con $m < n$, genera el polinomio de grado menor o igual que m que mejor aproxima (en el sentido de los mínimos cuadrados) el conjunto de pares de puntos (x_i,y_i) . De este modo, podemos corroborar el resultado obtenido anteriormente mediante la siguiente ejecución.

```
> p = polyfit(x,y,1);  
> polyout(p,'t')  
3.1071*t^1 + 9.2857
```

A continuación, para ilustrar el ajuste discreto mediante funciones no necesariamente polinómicas, usamos un ejemplo del campo de la demografía, en el que se desean ajustar, mediante una curva logística, los datos relativos a la población española en los últimos años, con vistas a predecir su evolución.

Fecha	Población	Fecha	Población	Fecha	Población	Fecha	Población
1857	15464340	1900	18616630	1950	28117873	2001	40499791
1860	15645072	1910	19990669	1960	30582936	2006	44708964
1877	16622175	1920	21388551	1970	33956047	2008	46063511
1887	17549608	1930	23677095	1981	37742561	2011	47190493
1897	18065635	1940	26014278	1991	39433942	2015	46449565

En este caso, vamos a realizar una aproximación logística, para lo que necesitamos escribir dichas cantidades como porcentajes de una población máxima estimada que vamos a tomar como 100 millones. Este ajuste no pretende ser realista, ya que en la población real influyen diversos factores que hacen que la tasa de crecimiento no sea necesariamente logística, pero sí es un ejemplo ilustrativo adecuado para este manual. El ajuste logístico sería el siguiente:

```
> x = [1857,1860,1877,1887,1897,1900,1910,1920,1930,1940,...
1950,1960,1970,1981,1991,2001,2006,2008,2011,2015];
> y = [15464340,15645072,16622175,17549608,18065635,...
18616630,19990669,21388551,23677095,26014278,28117873,...
30582936,33956047,37742561,39433942,40499791,44708964,...
46063511,47190493,46449565];
> y = y/10^8;
> Y = log(1./y-1);
> A = vander(x,2);
> p = (A\Y)';
p = [-0.010695, 21.725782]
```

Cambio de variables para aplicar ajuste lineal.

Ajuste lineal.

por lo que la función ajustada es $y = \frac{1}{1+e^{21.725782}e^{-0.010695x}}$. A continuación representamos gráficamente el ajuste obtenido junto con los datos aproximados.

```
> z = linspace(1820,2050,80);
> logi = @(z) 1./(1+exp(p(2)+p(1).*z));
> plot(x,y*100,'xb',z,logi(z)*100,'cr')
> ylabel('Millones de habitantes')
```

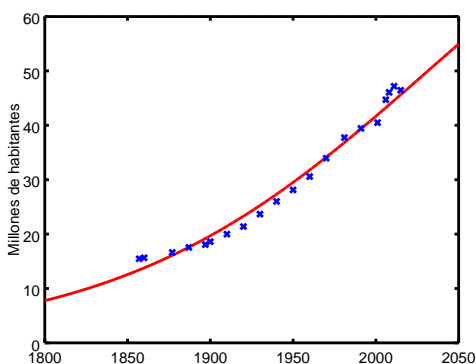


Figura 7.5: Aproximación logística

7.2.2. Aproximación por mínimos cuadrados continua

Comenzamos observando que, a diferencia del caso discreto, Octave no dispone de un comando predefinido para calcular aproximaciones por mínimos cuadrados continua.

Consideremos el siguiente *script* que calcula el polinomio de grado tres que mejor aproxima, en el sentido de los mínimos cuadrados continuo, la función seno en $[0, \pi]$.

```

1 % Script para calcular la aproximación por mínimos cuadrados
2 % continua de la función seno, en [0,pi], mediante polinomios
3 % de grado menor o igual que un valor prefijado.
4 % Definición del polinomio t^i (tomando i como parámetro).
5 1;
6 function y = pol(x,i) y = x.^i; endfunction;
7 gr = 3; % Elección del grado del polinomio.
8 vgr = gr:-1:0; % Vector de grados en el polinomio.
9 % Construcción de la matriz de coeficientes del sistema.
10 gramm = quadv(@(x) pol(x,vgr) '*pol(x,vgr),0,pi)
11 % Construcción del vector de términos independientes.
12 b = quadv(@(x) pol(x,vgr) '*sin(x),0,pi)
13 % Resolución del sistema y expresión del polinomio obtenido.
14 p = gramm\b;
15 polyout(p, 'x')
16 % Gráfica comparativa del seno y el polinomio obtenido.
17 x = linspace(0,pi,100);
18 plot(x, sin(x), 'g', x, polyval(p, x), 'k');
19 legend('Seno', 'Aproximacion');
20 axis([0 pi 0 1.05]);

```

Aunque el propio *script* contiene comentarios breves sobre cada una de sus acciones, quizás sea conveniente explicar con más detalle algunas partes.

Empezamos definiendo en la línea 6 los monomios de grado i (nótese que tanto t como i pueden ser vectores). Posteriormente, para construir la matriz de coeficientes del sistema `mcoef` y el vector de términos independientes `b`, hacemos uso del comando `quadv` (que realiza la integral definida de funciones vectoriales) y de una función anónima (`@(t)`) para poder pasarla como argumento (en las líneas 10 y 12).

En este gráfico, mediante el comando `legend`, se introducen leyendas que permiten identificar las distintas curvas representadas.

Por cierto, aunque hemos indicado que el polinomio a calcular sería el de grado tres, la simetría de la función seno hace que el polinomio obtenido sea solo de grado dos.

Presentamos ahora un segundo *script* para ajustar por mínimos cuadrados continuo una función, pero utilizando funciones trigonométricas en lugar de polinomios. Concretamente, tomaremos la parábola $f(x) = x^2$ en el intervalo $[-1, 1]$ y la aproximaremos mediante una función del tipo $c_1 + c_2 \sin(x) + c_3 \cos(x)$.

```

1 % Script para calcular la aproximación trigonométrica por
2 % mínimos cuadrados continua de la función  $x^2$  en  $[-1,1]$ .
3 % Definición de una función vectorial, que contiene a la base,
4 % evaluable sobre vectores columna.
5 1+1;
6 function y = basetrig(x)
7     y = [1+0*x, sin(x), cos(x)];
8 endfunction;
9 % Construcción del sistema.
10 gramm = quadv(@ (x) basetrig(x)'*basetrig(x), -1, 1);
11 % Construcción del vector de términos independientes.
12 b = quadv(@ (x) basetrig(x)'*x.^2, -1, 1);
13 % Resolución del sistema y expresión de la aprox obtenida.
14 c = gramm\b
15 % Gráfica comparativa de la identidad y su aprox trigonométrica.
16 x = linspace(-1,1,100)';
17 plot(x, x.^2, 'g', x, basetrig(x)*c, ':k');
18 legend('Parabola', 'Aproximacion');
19 axis([-1 1 -0.05 1.05]);

```

Nótese que la base de funciones usada se introduce, en las líneas 6–8, mediante una función vectorial. Por cierto, para que la función constantemente igual a 1 pueda ser evaluada adecuadamente sobre un vector x , hemos usado el truco $1+0*x$.

En la siguiente figura aparecen las salidas gráficas de los dos *scripts* anteriores.

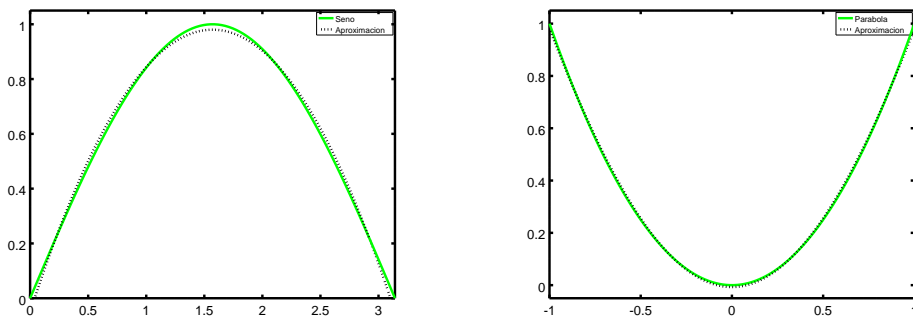


Figura 7.6: Aproximación del seno mediante polinomios de grado menor o igual que tres (izqda.) y aproximación de x^2 por funciones trigonométricas (dcha.).

Capítulo 8

Fórmulas de derivación y cuadratura numéricas

8.1. Fundamentos teóricos

Antes de comenzar, debemos recordar que tanto el cálculo de derivadas como el de primitivas son operaciones de tipo simbólico y que Octave, *a priori*, no está diseñado para operar de este modo. No obstante, podríamos cargar el paquete “symbolic”, que permite realizar estas y otras operaciones de carácter simbólico como, por ejemplo, manipular expresiones o resolver ecuaciones. Este y otros muchos paquetes para Octave se pueden encontrar en [21].

Así pues, el objetivo de este capítulo es obtener y programar fórmulas que permitan estimar, tanto el valor de la(s) derivada(s) de una función en un punto, como su integral definida en un cierto intervalo. Por tanto, no trataremos de calcular de forma simbólica la expresión de la función derivada, y tampoco buscaremos la expresión de una primitiva, aun cuando existan fórmulas para ello.

Aprovechando que en el capítulo 6 hemos trabajado con fórmulas de interpolación, usaremos este marco de trabajo para explicar, de forma simple e inmediata, las ideas de la derivación e integración numéricas. Lo enunciaremos a modo de teorema.

Teorema 8.1. *Si mediante el proceso genérico de interpolación podemos obtener un polinomio $p(x)$ que, intuitivamente, se parece a la función $f(x)$ que hemos interpolado, podemos pensar que la derivada en un punto (o la integral definida) del polinomio $p(x)$ también se parecerá a la derivada en un punto (o a la integral definida) de $f(x)$.*

Además, como el proceso de interpolación proporciona cierto control teórico sobre el error cometido al interpolar, la derivada (o la integral) de ese mismo error de interpolación nos proporcionará un control sobre la aproximación de la derivada (o de la integral).

Conviene decir que existen otros caminos que permiten resolver la cuestión planteada y que se obtienen por vías diferentes, pero son menos habituales. Nosotros, con esta idea básica en mente, pasamos a la primera parte: la derivación numérica.

8.1.1. Derivación numérica

Si partimos de una función derivable $f(x)$, para aproximar su primera derivada en un punto $x = a$ tomamos un parámetro pequeño y positivo $h > 0$ y hacemos las siguientes posibles deducciones.

- Fórmula de diferencias progresivas: se calcula el polinomio $p(x)$ que interpola a la función $f(x)$, en los puntos a y $a + h$, y aproximamos $f'(a)$ por $p'(a)$, obteniendo:

$$f(x) \approx p(x) = f(a) + \frac{f(a+h) - f(a)}{h}(x-a) \Rightarrow$$

$$f'(a) \approx p'(a) = \frac{f(a+h) - f(a)}{h}.$$

- Fórmula de diferencias regresivas: se calcula el polinomio $p(x)$ que interpola a $f(x)$, en los puntos $a - h$ y a , y aproximamos $f'(a)$ por $p'(a)$, obteniendo (omitimos la expresión de $p(x)$):

$$f'(a) \approx p'(a) = \frac{f(a) - f(a-h)}{h}.$$

- Fórmula de diferencias centradas: se calcula el polinomio $p(x)$ que interpola a $f(x)$, en los puntos $a - h$, a y $a + h$, y aproximamos $f'(a)$ por $p'(a)$, obteniendo:

$$f'(a) \approx p'(a) = \frac{f(a+h) - f(a-h)}{2h}.$$

Observamos que en esta tercera fórmula no aparece $f(a)$. De hecho, si interpolamos la función $f(x)$ sólo en los puntos $a - h$ y $a + h$, obtendremos un polinomio diferente pero que produce la misma fórmula de aproximación. Por ello, como veremos después, aunque las tres fórmulas usan sólo dos evaluaciones de $f(x)$, esta última será un poco *mejor*.

Es interesante que veamos la relación existente entre estas fórmulas y la definición de derivada. En efecto, en los tres casos tenemos que

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h} = \lim_{h \rightarrow 0} \frac{f(a) - f(a-h)}{h} = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a-h)}{2h}.$$

Por último, usaremos estas mismas ideas para aproximar la segunda derivada. La fórmula más usual es la siguiente.

- **Fórmula de diferencias centradas para la segunda derivada:** si calculamos el polinomio $p(x)$ que interpola a $f(x)$ en los puntos $a-h$, a y $a+h$, y usamos $p''(a)$ para aproximar $f''(a)$, obtendremos:

$$f''(a) \approx p''(a) = \frac{f(a-h) - 2f(a) + f(a+h)}{h^2}.$$

Como anunciamos en la introducción, el error cometido puede ser estimado a través del conocimiento del error de interpolación. Lo mostramos mediante el siguiente teorema (véase [15]).

Teorema 8.2.

- **Hipótesis:** sea $f : (a-h, a+h) \rightarrow \mathbb{R}$ una función suficientemente derivable.
- **Tesis:** existen valores $\xi_1, \xi_2, \xi_3, \xi_4 \in (a-h, a+h)$ tales que
 - $f'(a) = \frac{f(a+h) - f(a)}{h} - \frac{1}{2}h f''(\xi_1),$
 - $f'(a) = \frac{f(a) - f(a-h)}{h} - \frac{1}{2}h f''(\xi_2),$
 - $f'(a) = \frac{f(a+h) - f(a-h)}{2h} - \frac{1}{6}h^2 f'''(\xi_3),$
 - $f''(a) = \frac{f(a-h) - 2f(a) + f(a+h)}{h^2} - \frac{1}{12}h^2 f^{iv}(\xi_4).$

Inestabilidad numérica. El resultado anterior nos dice que, teóricamente, al aproximar las derivadas de una función suficientemente derivable mediante las fórmulas descritas, el error cometido tiende a cero conforme h tiende a cero y, de hecho, con orden cuadrático en los dos últimos casos. Sin embargo, lo teórico no siempre se ve reflejado en lo numérico. Así, en la sección de implementación numérica, veremos que estas fórmulas de derivación son inestables, en el sentido de que al hacer h menor que un cierto umbral, el valor de la aproximación, en general, se aleja cada vez más del valor correcto en lugar de converger al mismo.

Coefficientes indeterminados, exactitud. También podríamos enfocar directamente el diseño de fórmulas de tipo lineal

$$f'(a) \approx \sum_{k=0}^n \alpha_k f(x_k),$$

para aproximar $f'(a)$ a partir del conocimiento de los valores de $f(x)$ en $n+1$ nodos diferentes x_0, x_1, \dots, x_n . De este modo, para determinar el valor de los $n+1$ parámetros

α_k , le pediríamos a la fórmula que fuese exacta sobre una base de $\mathbb{P}_n[x]$, por ejemplo, su base usual. En tal caso, obtendríamos que

$$\begin{aligned} \text{usando } p_0(x) = 1 &\Rightarrow p'_0(a) = 0 = \alpha_0 + \dots + \alpha_n, \\ \text{usando } p_1(x) = x &\Rightarrow p'_1(a) = 1 = \alpha_0 x_0 + \dots + \alpha_n x_n, \\ \text{usando } p_2(x) = x^2 &\Rightarrow p'_2(a) = 2a = \alpha_0 x_0^2 + \dots + \alpha_n x_n^2, \\ &\vdots \\ \text{usando } p_n(x) = x^n &\Rightarrow p'_n(a) = na^{n-1} = \alpha_0 x_0^n + \dots + \alpha_n x_n^n. \end{aligned}$$

Es decir, llegaríamos a un sistema lineal de $n+1$ ecuaciones y $n+1$ incógnitas cuya matriz de coeficientes ya nos ha aparecido en el capítulo 6: es una matriz de Vandermonde. Por este motivo, ya sabemos que el sistema anterior posee una única solución. Este modo de obtener fórmulas de derivación numérica es completamente equivalente al realizado mediante interpolación. De hecho tenemos el siguiente resultado.

Teorema 8.3. *Una fórmula lineal de derivación numérica en $n+1$ nodos es exacta en $\mathbb{P}_n[x]$ si y sólo si es de tipo interpolatorio.*

También puede ocurrir que, aunque consideremos sólo $n+1$ nodos, obtengamos una fórmula con grado de exactitud mayor que n . Esto no contradice el resultado, simplemente indica que la fórmula obtenida es mejor de lo esperado *a priori*. Por ejemplo, esto es lo que ocurre con la fórmula de diferencias centradas (para la derivada de primer orden). En dicha fórmula obtenemos exactitud en $\mathbb{P}_2[x]$ aunque se usan sólo dos nodos ($n=1$) y, por esta razón, el error es cuadrático en lugar de lineal, como ocurre con las fórmulas progresivas y regresivas, que también usaban dos nodos.

8.1.2. Integración numérica

De manera análoga a la derivación numérica, podemos basarnos en el polinomio que interpola a una función, en varios nodos, para aproximar la integral de dicha función por la integral de su polinomio interpolador. Con esta idea, si partimos de una función continua $f(x)$ en un intervalo $[a, b]$, para aproximar su integral sobre ese intervalo, hacemos las siguientes deducciones.

- **Fórmula del rectángulo usando un punto del intervalo:** una vez calculado el polinomio $p(x)$ que interpola a $f(x)$ en un punto $m \in [a, b]$, aproximamos la integral de la siguiente forma:

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx = (b-a)f(m).$$

Normalmente se eligen $m = a$, $m = (a+b)/2$ ó $m = b$ obteniendo, respectivamente, las reglas del rectángulo a la izquierda, del punto medio y del rectángulo a la derecha.

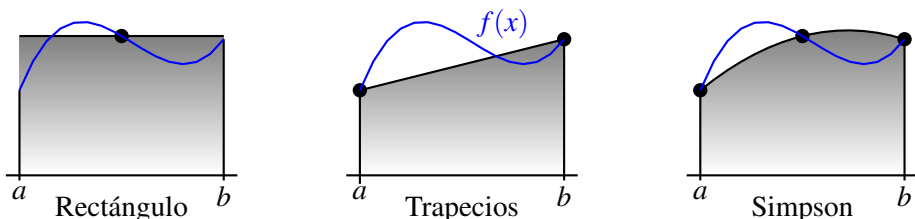
- **Fórmula del trapecio:** se calcula el polinomio $p(x)$ que interpola a $f(x)$ en los dos extremos del intervalo y aproximamos la integral mediante la expresión siguiente:

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx = (b-a) \frac{f(a) + f(b)}{2}.$$

- **Fórmula de Simpson:** se calcula el polinomio $p(x)$ que interpola a $f(x)$ en los dos extremos del intervalo y en el punto medio $m = (a+b)/2$, aproximando la integral de la forma siguiente:

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx = (b-a) \frac{f(a) + 4f(m) + f(b)}{6}.$$

En la siguiente figura podemos visualizar estas tres aproximaciones, ya que la integral definida se corresponde con el área que deja bajo sí la función integrada.



Como podemos observar en los tres casos, los errores cometidos corresponden a las áreas que, por exceso o por defecto, quedan entre la gráfica de la función y el área sombreada. Por tanto, los errores cometidos al usar estas reglas simples de integración no parecen muy aceptables. Para reducir tales errores podemos tener la tentación de usar polinomios que interpolen cada vez en más nodos pero, al igual que la interpolación se vuelve inestable al aumentar los nodos (véase el fenómeno de Runge descrito en el capítulo 6), la integral obtenida podría alejarse cada vez más de la buscada.

Para resolver el problema planteado en el párrafo anterior, podemos usar dos caminos diferentes (o una combinación de ambos): bien buscar nodos que, aunque no sean equiespaciados, “mejoren” en cierto sentido la aproximación, bien trocear el intervalo en subintervalos de longitud pequeña y usar fórmulas simples sobre cada uno de estos trocitos. En el primer caso, obtendremos las llamadas *fórmulas de cuadratura gaussianas*, que enseguida trataremos, y en el segundo caso, y dado que la integral sobre el intervalo total es igual a la suma de las integrales sobre todos y cada uno los subintervalos, podemos aproximar la integral total mediante una fórmula que llamaremos *compuesta* y que también estudiaremos posteriormente.

Coefficientes indeterminados, exactitud y fórmulas gaussianas. Al igual que en el caso de la derivación numérica, es posible buscar directamente una fórmula de cuadratura mediante fórmulas de tipo lineal

$$\int_a^b f(x) dx \approx \sum_{k=0}^n \alpha_k f(x_k), \quad (8.1)$$

a partir del conocimiento de los valores de f en $n+1$ nodos x_0, x_1, \dots, x_n prefijados, de modo que, para determinar los coeficientes $\alpha_0, \alpha_1, \dots, \alpha_n$, imponemos que dicha fórmula sea exacta en $\mathbb{P}_n[x]$. También en este caso obtendríamos las fórmulas descritas anteriormente (véase el ejercicio §8.83) y un teorema análogo al de derivación numérica.

Teorema 8.4. *Una fórmula lineal de integración numérica en $n + 1$ nodos es exacta en $\mathbb{P}_n[x]$ si y sólo si es de tipo interpolatorio.*

Usando la exactitud, podemos intentar minimizar el error haciéndonos la siguiente pregunta: fijado el número n , ¿es posible determinar algunos nodos x_0, x_1, \dots, x_n tales que la fórmula (8.1) obtenida sea aún mejor? La respuesta es afirmativa, y los nodos x_i (y los coeficientes asociados α_i) se determinarán imponiendo exactitud sobrejuntos con $2(n + 1)$ polinomios, dado que ahora hemos duplicado el número de incógnitas. A la fórmula obtenida se le llama gaussiana y los nodos correspondientes se denominan nodos gaussianos. Además, vienen caracterizadas por el siguiente resultado (véase [15]).

Teorema 8.5. *Una fórmula de integración numérica de tipo interpolatorio en $n + 1$ nodos es gaussiana si y sólo si es exacta en $\mathbb{P}_{2n+1}[x]$.*

A modo de ejemplo, la fórmula gaussiana con tres nodos ($n = 2$) sobre el intervalo $[-1, 1]$ viene dada por la expresión

$$\int_{-1}^1 f(x) dx \approx \frac{5}{9} f(-\sqrt{3/5}) + \frac{8}{9} f(0) + \frac{5}{9} f(\sqrt{3/5}),$$

y es exacta para polinomios de grado menor o igual a 5. La particularidad más interesante de estas fórmulas, aparte de su interés matemático, es que dan mayor precisión en la integración numérica cuando se aplican a funciones regulares, por lo que aparecen habitualmente en los algoritmos numéricos.

8.1.3. Reglas compuestas de integración numérica. Error de cuadratura

Como hemos dicho antes, la aplicación de las reglas simples de integración numérica no parecen ofrecer buenos resultados. De hecho, al igual que en el caso de la derivación numérica y usando el conocimiento sobre el error de interpolación, podemos estimar el error cometido mediante el siguiente teorema (véase [27]).

Teorema 8.6.

- *Hipótesis: sea $f(x)$ una función suficientemente derivable en el intervalo $[a, b]$.*
- *Tesis: existen valores $\xi_1, \xi_2, \xi_3, \xi_4 \in [a, b]$ tales que*
 - $\int_a^b f(x) dx = (b - a) f(a) + \frac{(b - a)^2}{2} f'(\xi_1),$
 - $\int_a^b f(x) dx = (b - a) f\left(\frac{a + b}{2}\right) + \frac{(b - a)^3}{24} f''(\xi_2),$
 - $\int_a^b f(x) dx = (b - a) \frac{f(a) + f(b)}{2} - \frac{(b - a)^3}{12} f''(\xi_3),$
 - $\int_a^b f(x) dx = (b - a) \frac{f(a) + 4f(\frac{a+b}{2}) + f(b)}{6} - \frac{(b - a)^5}{2880} f^{iv}(\xi_4).$

Constatamos de nuevo que los errores cometidos no parecen garantizar la bondad de las aproximaciones. Veámoslo con un ejemplo.

Ejemplo 8.7. Consideremos la integral definida

$$\int_1^{50} x^{-1} dx = 3.912023005\dots$$

Si estimamos esta integral mediante las reglas del trapecio y de Simpson, obtenemos los valores 24.99 y 9.6110457516 respectivamente, confirmando que ambas aproximaciones son bastante malas.

Por todo ello, para construir fórmulas efectivas, que no requieran interpolar en muchos nodos y que den errores pequeños, podemos proceder como sigue.

- Tomamos un natural $n \in \mathbb{N}$ suficientemente grande, troceamos $[a, b]$ en n subintervalos de longitud $\frac{b-a}{n}$ y creamos los nodos

$$a = x_0, x_1 = a + h, \dots, x_k = a + kh, \dots, x_n = b.$$

- Sobre cada subintervalo, aplicamos alguna de las anteriores reglas simples.
- Aproximamos la integral total mediante la suma de las n aproximaciones sobre cada subintervalo. A la fórmula así obtenida la llamamos *fórmula compuesta*.

Para las fórmulas simples vistas en la sección 8.1.2, las correspondientes fórmulas compuestas serían las siguientes.

- Fórmulas compuestas de los rectángulos:

$$\text{(izquierda)} \quad \int_a^b f(x) dx \approx h \sum_{k=1}^n f(a + (k-1)h),$$

$$\text{(punto medio)} \quad \int_a^b f(x) dx \approx h \sum_{k=1}^n f(a + (k-1/2)h),$$

$$\text{(derecha)} \quad \int_a^b f(x) dx \approx \sum_{k=1}^n f(a + kh).$$

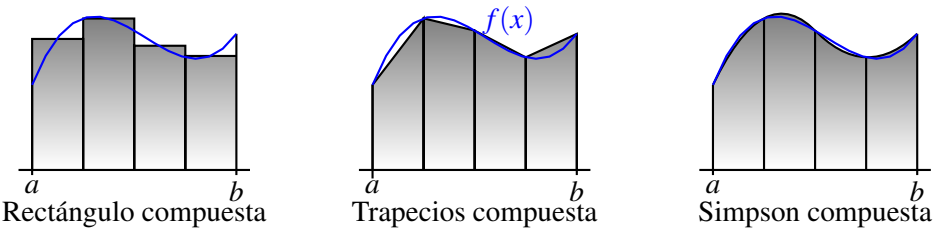
- Fórmula compuesta de los trapecios:

$$\int_a^b f(x) dx \approx \frac{h}{2} \left(f(a) + 2 \sum_{k=1}^{n-1} f(a + kh) + f(b) \right).$$

- Fórmula compuesta de Simpson: (requiere un número par de subintervalos)

$$\int_a^b f(x) dx \approx \frac{h}{3} \left(f(a) + 2 \sum_{m=1}^{n/2-1} f(a + 2mh) + 4 \sum_{m=1}^{n/2} f(a + (2m-1)h) + f(b) \right).$$

De este modo, utilizando simplemente $n = 4$ subintervalos, y como se observa visualmente en el siguiente dibujo, los errores cometidos se reducen sustancialmente.



Recuperando el ejemplo 8.7, si aproximamos la integral de $1/x$ sobre el intervalo $[1, 50]$, dividiendo el intervalo en subintervalos de longitud 1, mediante trapezios compuesta obtenemos el valor 3.9892..., mientras que con Simpson compuesta el valor aproximado es 3.9134.... En ambos casos, mejoramos significativamente los resultados obtenidos aplicando las reglas simples.

Para justificar teóricamente los buenos resultados para las fórmulas compuestas, tenemos el siguiente resultado (véase [27]).

Teorema 8.8.

- *Hipótesis:* sea f una función suficientemente derivable en el intervalo $[a, b]$.
- *Tesis:* los errores cometidos al aproximar $\int_a^b f(x) dx$ mediante las fórmulas compuestas, dadas anteriormente, son

Fórmula compuesta	Rect. izqd.	Rect. dcha.	Pto. medio	Trapezios	Simpson
cota de error	Ch	Ch	Ch^2	Ch^2	Ch^4

donde $h = \frac{b-a}{n}$ y C es una constante que sólo depende de $b - a$ y las derivadas de $f(x)$.

8.2. Implementación de algunos métodos

8.2.1. Fórmulas de derivación numérica

La programación de diferencias finitas, ya sean centradas, regresivas o progresivas, es muy simple en Octave. Por ejemplo, si queremos calcular y dibujar la derivada de la función seno¹ en el intervalo $[0, \pi]$, aproximando su derivada de forma numérica mediante diferencias progresivas y centradas, usando 50 puntos del intervalo, podríamos ejecutar el siguiente código.

```
> x = linspace(0, pi, 50);  
> y = sin(x);  
> progresivas = (y(2:end) - y(1:end-1)) ./ (pi/49);
```

¹Por supuesto, no queremos emplear que la función coseno es tal derivada.


```
> centradas = (y(3:end)-y(1:end-2))/(2*pi/49);
> plot(x(1:end-1),progresivas,'2',x(2:end-1),centradas,'3')
```

Nótese que, para las progresivas, obtenemos la derivada numérica en las 49 primeras abscisas y, para las centradas, en las 48 abscisas centrales.

Por último, podemos crear un programa que calcule la derivada numérica de una función cualquiera. Por ejemplo, que calcule las diferencias centradas sobre una cantidad de nodos y un intervalo que sean argumentos del propio comando.

```
1 function [x,df] = derinum(f,a,b,N)
2 % df es la derivada primera numérica de la función f sobre el
3 % vector x de N valores equiespaciados en el intervalo [a,b].
4 h = (b-a)/(N-1);
5 x = linspace(a,b,N);
6 df = (feval(f,a+h*(1:N))-feval(f,a+h*(-1:(N-2))))/(2*h);
7 endfunction
```

Como ejemplo ilustrativo, calculamos la derivada numérica de la función seno (usando 20 nodos) y la representamos en la figura 8.1 de la página 128, junto al coseno, que sabemos que es su derivada real.

```
> [x,df] = derinum('sin',0,pi,20);
> plot(x,df,['*',';derivada aproximada;'],...
x,cos(x),';derivada real;');
```

El comando `diff`. En realidad, la única operación que hemos empleado para realizar la derivada numérica ha sido extraer, a partir de un vector dado, el vector de las diferencias entre dos elementos consecutivos. Pues bien, Octave posee el comando `diff` (`y`) que realiza precisamente la operación $y(2:\text{end}) - y(1:\text{end}-1)$, que resulta muy útil para definir directamente las diferencias progresivas o regresivas, aunque no para las centradas. De hecho, si x representa un conjunto de abscisas ordenadas, aunque no necesariamente equiespaciadas, e y representa los valores de una cierta función sobre dichas abscisas (el seno, en el ejemplo anterior), la orden

```
> diff(y)./diff(x)
```

nos proporcionará directamente el vector de diferencias progresivas que aproxima la derivada de dicha función.

Diferencias centradas para la derivada segunda. Por último, cuando las abscisas x sí están equiespaciadas, podríamos implementar la fórmula de diferencias centradas para aproximar la derivada segunda de la función con valores y . En este caso, cualquiera de las siguientes tres órdenes darían el mismo resultado.

```
> (y(1:end-2)-2*y(2:end-1)+y(3:end))/h^2;
```

```
> diff(diff(y))/h^2;
> diff(y,2)/h^2;
```

siendo h la distancia entre dos nodos consecutivos cualesquiera de x . Al igual que antes, podemos crear un programa que calcule la derivada segunda numérica de una función cualquiera. Veamos un ejemplo.

```
1 function[x,d2f] = derinum2(f,a,b,N)
2 % d2f es la segunda derivada numérica de la función f sobre el
3 % vector x de N valores equiespaciados en el intervalo [a,b].
4 h = (b-a)/(N-1);
5 x = linspace(a,b,N);
6 d2f = (feval(f,a+h*(-1:(N-2)))-2*feval(f,a+h*(0:N-1))+...
7 feval(f,a+h*(1:N)))/h^2;
8 endfunction
```

Nótese que ambas funciones creadas, `derinum.m` y `derinum2.m`, requieren que la función argumento pueda ser evaluada un “poquito” a la izquierda de a y un “poquito” a la derecha de b .

Inestabilidad en la derivación numérica

Como adelantamos en la primera sección de este capítulo, las fórmulas de derivación numérica son computacionalmente inestables, es decir, al hacer el parámetro h cada vez más pequeño, existe un umbral a partir del cual la aproximación empeora en lugar de converger. Vamos a constatar este hecho con un ejemplo.

Si aproximamos la derivada de la función arcotangente en el punto $\sqrt{2}$ (cuyo valor exacto es $1/3$) mediante diferencias progresivas, y hacemos h tender a cero, al principio parece converger; sin embargo, a partir de cierto instante, la convergencia empieza a fallar y, poco después, el resultado obtenido es igual a 0, tal como muestra el siguiente código en el que h toma sucesivamente los valores 10^{-5} , 10^{-6} , ..., 10^{-16} (véase el ejercicio §8.82).

```
> pot = -5:-1:-16;
> h = 10.^pot;
> a = sqrt(2);
> format long
> [h; ((atan(a+h)-atan(a))./h)] ' en dos columnas, h y la aproximación.
ans =
0.0000100000000000 0.333331761992461
0.0000010000000000 0.333333176172346
0.0000001000000000 0.333333317614759
0.0000000100000000 0.333333327606766
0.0000000010000000 0.333333360913457
```

```

0.0000000000100000 0.333333360913457
0.0000000000010000 0.333333360913457
0.0000000000001000 0.333399974294934
0.0000000000000100 0.333066907387547
0.0000000000000010 0.333066907387547
0.0000000000000001 0.444089209850062
0.0000000000000000 0.000000000000000
0.0000000000000000 0.000000000000000
0.0000000000000000 0.000000000000000

```

Es por esto que el uso directo de las fórmulas de diferencias finitas debe hacerse con cierta cautela, aunque son indispensables para deducir métodos de resolución de ecuaciones diferenciales, como veremos en los dos siguientes capítulos.

8.2.2. Fórmulas de cuadratura o integración numérica

En lo que resta usaremos el siguiente ejemplo ilustrativo de función cuya integral definida es conocida:

$$\int_{-1}^1 2\sqrt{1-x^2} dx = \pi.$$

Definiremos pues la función f , por ejemplo, mediante una función anónima:

```
> f = @(x) 2*sqrt(1-x.^2);
```

y lo primero que vamos a hacer es calcular esta integral mediante un comando directo que ya viene implementado en Octave.

El comando **quad**

Para calcular la integral anterior ejecutaremos simplemente

```
> quad(f, -1, 1);
```

que produce la salida esperada, `ans = 3.1416`, esto es, el número π . Observemos que, si hubiésemos escrito `function y=f(x) y=2*sqrt(1-x.^2); endfunction` para definir la función f , entonces tendríamos que llamar a su cadena de caracteres para calcular su integral: `quad('f', -1, 1)`.

Internamente, el comando `quad` usa una fórmula de cuadratura compuesta que, en cada subintervalo, utiliza los nodos gaussianos que comentamos en la sección 8.1.2. El comando determina internamente el número necesario de subintervalos para alcanzar una precisión estimada de 10^{-8} . En realidad el comando `quad` permite modificar esta precisión, y otras opciones, para calcular integrales que puedan ser, *a priori*, muy singulares, y puede dar más información, además del valor de la integral. Su sintaxis completa, que podemos averiguar mediante `help quad`, es de la forma

```
> [V, IER, NFUN, ERR] = quad(F, A, B, TOL, SING)
```

y permite establecer, opcionalmente como hemos dicho, cierta precisión (TOL) relativa y/o absoluta, así como una lista de puntos (SING) en los que la función posea cierta singularidad. Asimismo, IER, NFUN, ERR son, respectivamente, un código de error (IER= 0 indica un resultado óptimo), el número de evaluaciones que ha usado y una estimación del error cometido.

Integrales impropias. El comando `quad` también permite calcular integrales impropias, bien porque el integrando no es continuo en alguno de los extremos del intervalo, bien porque alguno de estos extremos es infinito. Como ejemplo, mostramos las siguientes integrales (conocidas),

$$\int_0^1 \frac{1}{\sqrt{x}} dx = 2, \quad \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{x^2}{2}} dx = 1, \quad \int_0^{\infty} x^3 e^{-x} dx = \Gamma(4) = 3! = 6.$$

que calcularemos respectivamente mediante los comandos:

```
> quad(@(x) 1./sqrt(x), 0, 1)
> quad(@(x) 1./sqrt(2*pi)*exp(-x.^2./2), -Inf, Inf)
> quad(@(x) x.^3.*exp(-x), 0, Inf)
```

Aquí es conveniente resaltar que las funciones anónimas han sido definidas directamente dentro del comando `quad`, y que no es preciso hacerlo con anterioridad.

Variantes del comando `quad`. Octave tiene implementados internamente hasta cinco algoritmos para calcular integrales definidas, ya que en general no existe un algoritmo óptimo para cualquier tipo de función. El comando `quad`, como ya hemos dicho, usa la cuadratura gaussiana (compuesta) que presentamos en la introducción, por lo que tiene muy buena precisión cuando integramos funciones regulares, pero cuando el integrando no es regular empeora su comportamiento. Por ello, existen otros cuatro algoritmos que pueden proporcionar mayor precisión dependiendo del tipo de función o si se trata de una integral impropia. Aunque en este texto introductorio no vamos a entrar a fondo en los algoritmos, dejamos una indicación sobre los comandos y su aplicabilidad y dirigimos al lector al manual [5, Numerical Integration] para ampliar información.

Comando.	Características.
<code>quadv</code>	Versión vectorial de <code>quad</code> usando Simpson compuesto.
<code>quadl</code>	Mejor que <code>quad</code> cuando la función no es muy regular.
<code>quadgk</code>	Mejora el anterior en funciones oscilatorias y/o intervalos infinitos.
<code>quadcc</code>	El más robusto de todos, pero con más coste computacional.

Integración conociendo sólo evaluaciones. Primitiva numérica. Hasta ahora disponíamos de la expresión de la función $f(x)$ que queríamos integrar, y cada comando

Como ejemplo ilustrativo, en la figura 8.1 dibujamos la integral numérica de la función coseno junto con la función seno, que sabemos que es su primitiva.

```
> x = linspace(0,pi,20);
> y = cos(x);
> pva = cumtrapz(x,y);
> plot(x,pva, ['*',';primitiva aproximada;'], ...
x, sin(x), ';primitiva real;');
```

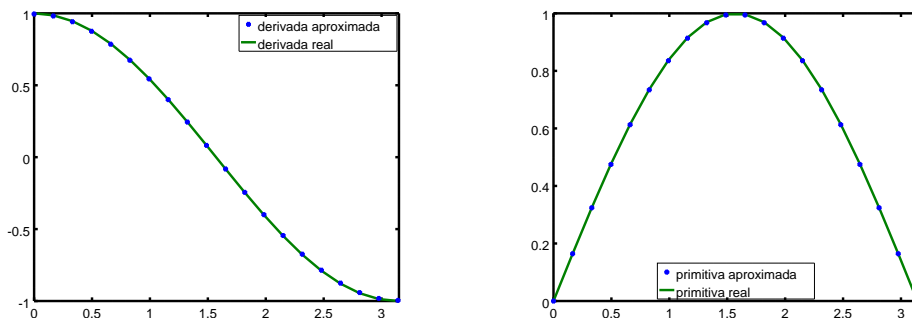


Figura 8.1: Derivada numérica del seno (izqda.) y primitiva numérica del coseno (dcha.) usando 20 nodos.

Para terminar esta sección, nos remitimos al manual [5] para el uso de comandos tales como `dblquad` o `triplequad`, relacionados con el cálculo de integrales dobles y triples. Estos comando pueden ser útiles para calcular áreas, superficies y volúmenes.

8.2.3. Fórmulas compuestas

Por último, vamos a implementar nosotros mismos algunas de las fórmulas de cuadratura compuestas vistas en la sección 8.1.3. Las siguientes funciones, que hemos llamado `rectang.m` y `simpson.m`, podrían ser una primera versión de la programación de las fórmulas compuestas de los rectángulos (punto medio) y Simpson, respectivamente, para aproximar la integral definida de una función dada f que pueda ser evaluada mediante `fval`, y utilizando nodos equiespaciados.

```
1 function int = rectang(f,a,b,N = 150)
2 % Función que aproxima la integral de f en el
3 % intervalo [a,b] mediante la fórmula compuesta de
4 % rectángulos (punto medio) con N subintervalos
5 %
6 h = (b-a)/N;
7 int = h*sum(feval(f,a+h*(1:N)-h/2));
8 endfunction
```

```

1 function int = simpson(f,a,b,N = 75)
2 % Función que aproxima la integral de f en el
3 % intervalo [a,b] mediante la fórmula compuesta de
4 % Simpson con 2N+1 nodos
5 %
6 h = (b-a)/N;
7 x = a+h*(0:(N-1));
8 int = (sum(feval(f,x)+4*feval(f,x+h/2)+feval(f,x+h)))*h/6;
9 endfunction

```

Remarquemos que el número de nodos ha sido introducido como argumento pero con un valor por defecto (véase la sección 1.3). Por ejemplo en la función `rectang.m` hemos tomado $N = 150$, lo que indica que podemos ejecutar esta función escribiendo `rectang(f,a,b,400)` para emplear 400 nodos, o simplemente `rectang(f,a,b)`, en cuyo caso se utilizarán los 150 que hemos dado por defecto.

Si usamos las funciones anteriores para aproximar la integral de referencia dada al inicio de la sección 8.2.2, ejecutándolas con 400 evaluaciones (las mismas que usó `quad`), obtenemos

```

> rectang(f,-1,1,400)
ans = 3.1417
> simpson(f,-1,1,200)
ans = 3.1414

```

siendo ambos resultados peores que el dado por `quad`. Esto es debido a que hemos usado nodos equiespaciados, mientras que los comandos de Octave producen métodos “adaptativos”, es decir, que usan más nodos allí donde la función es más singular u oscilante, hecho que detectan numéricamente y que no es siempre fácil de programar de forma eficiente.

Para concluir, y dado que el comando `trapz` proporciona esencialmente la fórmula compuesta de los trapecios, lo utilizaremos para definir una función que implemente la fórmula compuesta de los trapecios, pero actuando sobre funciones evaluables, del mismo modo que `rectang.m` y `simpson.m`.

```

1 function int = trapecios(f,a,b,N = 150)
2 % Función que aproxima la integral de f en el
3 % intervalo [a,b] mediante la fórmula compuesta de
4 % trapecios con N subintervalos
5 %
6 h = (b-a)/N;
7 x = a+h*(0:N);
8 y = feval(f,x);
9 int = h*trapz(y);
10 endfunction

```


Capítulo 9

Ecuaciones diferenciales I: Problemas de valores iniciales

En este capítulo vamos a presentar métodos numéricos para aproximar soluciones de problemas de valores iniciales asociados a ecuaciones diferenciales ordinarias. Empezaremos presentando este tipo de problemas y su tratamiento analítico.

9.1. Fundamentos teóricos

Una ecuación diferencial ordinaria (EDO) es un problema que consiste en encontrar las funciones reales $x(t)$ (de variable real) verificando una identidad que involucra una o varias de sus derivadas; veamos algunos ejemplos.

Ejemplo 9.1. Hallar, en cada caso, las funciones $x(t)$ tales que:

- $x'(t) = \lambda x(t)$, cuyas soluciones son las funciones de la forma $x(t) = Ke^{\lambda t}$, con $K \in \mathbb{R}$.
- $x''(t) = -x(t)$, cuyas soluciones son de la forma $x(t) = K_1 \cos(t) + K_2 \sin(t)$, con $K_1, K_2 \in \mathbb{R}$.
- $x'(t) = 1 + (x(t))^2$, cuyas soluciones son $x(t) = \tan(t + K)$, con $K \in \mathbb{R}$.
- $x'(t) = x(t)^{-1/2}$, con soluciones de la forma $x(t) = (\frac{3}{2}t + K)^{2/3}$, donde $K \in \mathbb{R}$.
- $x'(t) = \sqrt{-x(t)^2 - 1}$, que no admite solución.

Diremos que una EDO es de **orden** d si la derivada d -ésima de x es la de mayor orden que aparece en la ecuación. Como observamos en los ejemplos anteriores, las EDOs no han de tener una única solución, por lo que es habitual añadirles una o varias condiciones adicionales, que suelen ser valores de la función x o de algunas de sus derivadas en ciertos puntos distinguidos. Cuando la ecuación diferencial es de orden d y nos proporcionan el valor de x y de sus derivadas hasta orden $d - 1$ en un mismo punto t_0 , esto es, los valores de:

$$x(t_0), x'(t_0), \dots, x^{(d-1)}(t_0),$$

diremos que tenemos un problema de valores iniciales (PVI)¹. Así por ejemplo, las EDOs del ejemplo 9.1 anterior, completadas con las condiciones iniciales que se indican, producen los siguientes PVIs.

Ejemplo 9.2.

- $x'(t) = \lambda x(t)$ y $x(5) = 1$, tiene como solución $x(t) = e^{\lambda t - 5}$.
- $x''(t) = -x(t)$, con $x(0) = 0$ y $x'(0) = 1$, tiene como única solución $x(t) = \sin(t)$.
- $x'(t) = 1 + (x(t))^2$ y $x(0) = 0$, tiene como única solución $x(t) = \tan(t)$.
- $x'(t) = x(t)^{-1/2}$, con $x(0) = 0$, tiene infinitas soluciones que podemos escribir como:
 $x(t) = (\frac{3}{2}t + K)_+^{2/3}$, con $K \leq 0$, donde $(\cdot)_+$ denota la función parte positiva.

Vectorización de ecuaciones de orden superior. Por lo general, un PVI de orden $d > 1$ se puede reescribir, de forma equivalente, como una EDO de orden 1 vectorial, es decir, de la forma

$$u'(t) = f(u(t), t), \quad u(a) = u_0 \in \mathbb{R}^d, \quad t \in [a, b], \quad (9.1)$$

donde $u(t)$ es una función vectorial. Así, el PVI $x'' = -x$, con $x(0) = 0$ y $x'(0) = 1$, puede ser reescrito como

$$\begin{pmatrix} u_1(t) \\ u_2(t) \end{pmatrix}' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} u_1(t) \\ u_2(t) \end{pmatrix}, \quad \text{con } (u_1(0), u_2(0)) = (0, 1),$$

donde la incógnita es la función vectorial $u(t) = (u_1(t), u_2(t)) = (x(t), x'(t))$.

Existencia y unicidad de solución. Antes de plantearnos la aproximación de las soluciones de un PVI, es recomendable asegurarnos que tenga solución y que esta sea única. En este sentido, el siguiente resultado nos puede ser de gran utilidad (este, y otros concernientes al tratamiento analítico de PVIs, se pueden encontrar en [3, 13], así como numerosas aplicaciones de modelado en [26, 29]).

Teorema 9.3 (Picard-Lindelöf).

- *Hipótesis:* sea una función $f : \mathbb{R}^d \times [a, b] \rightarrow \mathbb{R}^d$ tal que
 - es continua en ambas variables,
 - tiene derivada continua y acotada respecto de la primera variable².
- *Tesis:* existe $u : [a, b] \rightarrow \mathbb{R}^d$ única solución de (9.1).

¹En el capítulo 10 trataremos EDOs con condiciones en los extremos de un cierto intervalo. Estos problemas se denominarán problemas de valores en la frontera o problemas de contorno.

²En realidad basta que f sea *lipschitziana* con respecto a la primera variable.

En el ejemplo 9.2 encontramos diferentes casos sobre existencia y unicidad. Mientras que la solución del tercer ejemplo únicamente existe como mucho para valores de t en el intervalo $]-\frac{\pi}{2}, \frac{\pi}{2}[$, en el cuarto ejemplo no se tiene unicidad de solución, aún imponiendo una condición inicial. Por lo tanto, señalamos que a lo largo de este capítulo supondremos que los PVI estudiados tienen siempre existencia y unicidad de solución. De hecho, si no hay solución, ¿qué sentido tendría aproximar algo que no existe? Y si la solución no es única, ¿cuál de ellas aproximaría nuestro método?

9.1.1. Algunos métodos de integración numérica

Una vez supuesta la existencia y unicidad de la solución $u : [a, b] \rightarrow \mathbb{R}^d$ de (9.1), nos planteamos obtener métodos para su aproximación numérica³. Concretamente, nos centraremos en esquemas en diferencias finitas, cuyo objetivo básico es calcular valores aproximados de $u(t_n)$, que denotaremos por y_n , sobre ciertos valores de la variable $t_n \in [a, b]$. Concretamente, dada una **partición del intervalo** $[a, b]$, $a = t_0 < t_1 < \dots < t_N = b$, se dice que una aproximación numérica de u en dicha partición viene dada por los pares (t_n, y_n) , para $n = 0, \dots, N$. Por **método numérico** entenderemos un **algoritmo para calcular los valores y_n , que aproximan a $u(t_n)$** .

Veamos, por ejemplo, la deducción de uno de los métodos más elementales y clásicos, el método de Euler explícito. A partir del PVI (9.1) obtenemos las identidades

$$u'(t_n) = f(u(t_n), t_n), \text{ para cada } t_n, n = 0, 1, \dots, N, \quad (9.2)$$

o bien, integrando,

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} u'(s) ds = \int_{t_n}^{t_{n+1}} f(u(s), s) ds. \quad (9.3)$$

Empleando la fórmula de diferencias progresivas para aproximar la derivada en (9.2), o bien la fórmula de rectángulos a izquierda para aproximar la integral en (9.3) (véase capítulo 8), se deduce que

$$u(t_{n+1}) \approx u(t_n) + (t_{n+1} - t_n)f(u(t_n), t_n), \quad 0 \leq n \leq N-1.$$

Imponiendo igualdad en esta expresión y sustituyendo la solución exacta por sus valores aproximados obtendremos una regla de cálculo, para los valores aproximados y_n , que se denomina **método de Euler explícito**.

Algoritmo 9.4 (Euler explícito para el PVI (9.1)).

- $y_0 = u_0$,
- $y_{n+1} = y_n + (t_{n+1} - t_n)f(y_n, t_n)$, $n \geq 0$.

³La resolución de ecuaciones también suele denominarse *integración de ecuaciones*, por lo que las técnicas de resolución se denominan *métodos de integración numérica*. Es preciso no confundirlas con las fórmulas de cuadratura estudiadas en el capítulo 8 para calcular integrales numéricamente.

Es decir, a partir del valor $y_0 = u_0$ (que viene dado por la condición inicial $u(t_0) = u_0$) el método permite calcular y_1 ; después, a partir de y_1 permite calcular y_2 , y así sucesivamente el resto de los valores aproximados. En la siguiente sección veremos su funcionamiento al aproximar la solución de un problema concreto.

Este ejemplo basta para entender la **idea general** de construcción de gran cantidad de métodos: en primer lugar se realiza una **partición uniforme**⁴ del intervalo $[a, b]$, es decir, que los nodos estén todos a la misma distancia y dados por la expresión

$$t_n = a + nh, \text{ donde } n = 0, \dots, N \text{ y } h = \frac{b-a}{N}, \text{ siendo } N \in \mathbb{N} \text{ un natural no nulo.}$$

Al número h se le llama **tamaño de la partición**. En segundo lugar se **discretiza la ecuación**, bien empleando fórmulas de aproximación para la derivada⁵ que aparece en (9.2) en cada uno de los nodos, o bien fórmulas de cuadratura para aproximar la integral en (9.3) usando algunos de los nodos de la partición. Ambas técnicas estudiadas en el capítulo 8. Por último, se **despeja** la incógnita y_{n+1} (la de mayor índice) en función de las anteriores aproximaciones. La diferencia esencial entre un método y otro proviene de la forma en que se discretice la ecuación diferencial.

Así, de forma análoga al método de Euler explícito, podemos deducir los siguientes **métodos de un paso**, es decir, en los que y_{n+1} sólo depende de la aproximación inmediatamente anterior y_n .

$$\text{Euler implícito: } y_{n+1} = y_n + hf(y_{n+1}, t_{n+1}).$$

$$\text{Trapezoidal o de Crank-Nicolson: } y_{n+1} = y_n + \frac{h}{2}(f(y_n, t_n) + f(y_{n+1}, t_{n+1})).$$

$$\text{Taylor de orden 2: } y_{n+1} = y_n + h\left(f(y_n, t_n) + \frac{h}{2}\left(\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial y}\right)(y_n, t_n)\right).$$

$$\text{Euler mejorado: } y_{n+1} = y_n + hf\left(y_n + \frac{h}{2}f(y_n, t_n), t_n + \frac{h}{2}\right).$$

Observemos que los dos primeros métodos son **implícitos**, esto es, para despejar y_{n+1} en cada paso, es preciso **resolver una ecuación**, lo que aumenta el coste computacional del método.

Por otro lado, se pueden proponer métodos que involucren los valores aproximados en más de dos nodos, con lo que llegamos a los denominados **métodos multipaso**, donde los más conocidos son las familias de métodos Adams–Bashfort (explícitos) y Adams–Moulton (implícitos) y de diferenciación regresiva o BDF (Backward Differencing Formulas). A continuación mostramos casos particulares para dos pasos (es decir, para despejar y_{n+2} es preciso usar dos aproximaciones previas, y_{n+1} e y_n):

⁴En realidad, la partición no ha de ser uniforme. Por simplicidad nos restringimos a este caso.

⁵En ocasiones, además de la derivada, podemos discretizar cualquier otro término de la ecuación diferencial (9.2).

$$\text{Adams-Bashfort: } y_{n+2} - y_{n+1} = \frac{h}{2}(-f(y_n, t_n) + 3f(y_{n+1}, t_{n+1})).$$

$$\text{Adams-Moulton: } y_{n+2} - y_{n+1} = \frac{h}{12}(-f(y_n, t_n) + 8f(y_{n+1}, t_{n+1}) + 5f(y_{n+2}, t_{n+2})).$$

$$\text{Método BDF: } 3y_{n+2} - 4y_{n+1} + y_n = 2hf(y_{n+2}, t_{n+2}).$$

Como idea general, comentar que los métodos BDF se obtienen a partir de fórmulas de aproximación de la primera derivada aplicados a (9.2), mientras que las fórmulas de tipo Adams se obtienen a partir de fórmulas de cuadratura para aproximar la integral en (9.3). En ambos casos las aproximaciones usadas son de **tipo interpolatorio**.

Como ya hemos dicho, se pueden crear tantos esquemas numéricos asociados a (9.1) como discretizaciones hagamos de la EDO, y cada uno con sus propiedades específicas: explícitos o implícitos, consistencia, orden, estabilidad, convergencia, región de A-estabilidad, etc. El conocimiento de estas propiedades básicas permite discutir la idoneidad de unos métodos u otros ante un determinado problema, e incluso obtener estimadores del error cometido de forma que, por ejemplo, se puedan emplear métodos de paso adaptativo, esto es, métodos que ajustan el valor del paso, h , para que el error no sobrepase un determinado umbral. Remitimos a [18] para una revisión de estos conceptos, o bien a [7] para una exposición rigurosa de todos estos aspectos.

9.2. Resolución numérica de PVIs

Antes de presentar los comandos que Octave posee para la resolución de PVIs, vamos a hacer un breve ejercicio sobre un ejemplo al que aplicaremos con Octave el método más elemental, el método de Euler explícito descrito en el algoritmo 9.4.

Ejemplo 9.5. Consideramos el PVI $x'(t) = \lambda x(t)$, $x(5) = 1$, y pretendemos aproximar su solución en el intervalo $[5, 6]$. En primer lugar vamos a construir la partición uniforme del intervalo: $t_n = 5 + nh$, con $n = 0, \dots, N$, siendo $h = 1/N$ el paso y N un número natural cualquiera no nulo (que determina el número de subintervalos). Para estos datos, el algoritmo 9.4 da lugar a:

$$y_0 = 1, \quad y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n, \quad \text{para } n \geq 0. \quad (9.4)$$

Mediante el siguiente código, para $\lambda = 1$, comparamos la solución analítica $x(t) = e^{t-5}$ con las soluciones aproximadas obtenidas tomando 6 y 11 nodos (esto es, $N = 5$ y $N = 10$, respectivamente). El cálculo de los valores aproximados se hace añadiendo al vector y , que inicialmente contiene únicamente la condición inicial, cada una de las distintas aproximaciones a partir de la fórmula (9.4), una a una. Notamos que generar una partición uniforme en N trozos del intervalo $[5, 6]$ con Octave es posible tanto usando `linspace(a, b, N+1)`, como usando el rango `5:h:6` (siendo $h = 1/N$). Comentamos además que el comando `clf`, que aparece en el siguiente código, se utiliza para borrar la ventana gráfica que, eventualmente, podría estar abierta con algún gráfico anterior.

```

1 lambda = 1;
2 % Representamos la solución analítica
3 clf
4 hold on
5 t = 5:0.01:6;
6 plot(t,exp(lambda*t-5),'k');
7 % Representamos los valores aproximados con 6 nodos
8 N = 5;
9 h = 1/N;
10 y = [1];
11 for i = 1:N
12     y = [y,y(:,i)*(1+lambda*h)];
13 endfor
14 plot(5:h:6,y,'xr')
15 % Representamos los valores aproximados con 11 nodos
16 N = 10;
17 h = 1/N;
18 y = [1];
19 for i = 1:N
20     y = [y,y(:,i)*(1+lambda*h)];
21 endfor
22 plot(5:h:6,y,'ob')
23 xlabel("t");ylabel("y");
24 legend('Soluc. exacta','Aprox. 6 nodos','Aprox. 11 nodos')
25 hold off

```

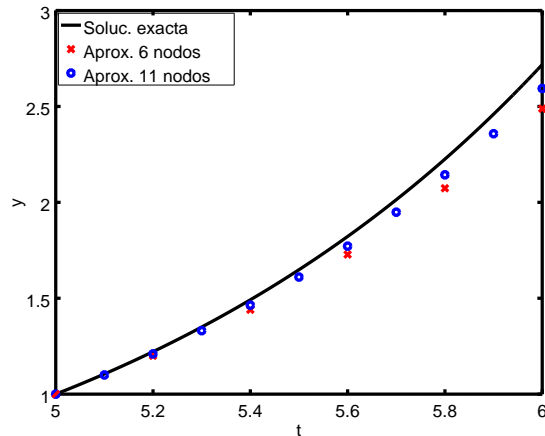


Figura 9.1: Aproximaciones numéricas mediante Euler explícito.

Podemos observar en la figura 9.1 que, efectivamente, los valores aproximados son cada vez mejores aproximaciones de x cuando se aumenta el número de nodos o, equivalentemente, cuando h tiende a cero. Esta **convergencia** es la propiedad básica que debemos exigir a cualquier método.

9.2.1. El comando `lsode`

El comando predeterminado de Octave para resolver PVIs del tipo (9.1) es `lsode`, aunque podemos emplear el paquete opcional `OdePkg` con multitud de comandos y métodos disponibles. Los argumentos básicos que requiere `lsode` son:

- una cadena de caracteres o una función anónima que llame a la función f que determina la ecuación diferencial,
- el vector columna con los valores iniciales u_0 ,
- un vector columna con los nodos t_n en los que se quiere calcular la aproximación.

Por defecto, `lsode` devuelve una matriz donde cada columna contiene los y_n , es decir, los valores aproximados del vector solución u sobre cada uno de los nodos t_n que se han indicado en el tercer argumento.

Ejemplo 9.6. Vamos a aproximar las soluciones del sistema de Lotka-Volterra, también llamado de presa-depredador,

$$\left. \begin{aligned} u_1'(t) &= a u_1(t) - b u_2(t) u_1(t) \\ u_2'(t) &= -c u_2(t) + d u_1(t) u_2(t) \end{aligned} \right\}$$

que modela la evolución temporal de sendas poblaciones u_1 y u_2 de presas y depredadores respectivamente, siendo a la tasa de natalidad de las presas y c la tasa de mortalidad de depredadores, mientras que la tasa de natalidad de depredadores es $d u_1(t)$ (proporcional al número de presas) y la tasa de mortalidad de presas es $b u_2(t)$ (proporcional a la cantidad de depredadores). Eligiendo $a = b = c = d = 1$, podemos obtener una aproximación de soluciones concretas de este sistema mediante siguientes los comandos:

```
> function udot = presadepre(u,t)
    a=1; b=1; c=1; d=1;
    udot(1,1) = a*u(1)-b*u(1)*u(2);
    udot(2,1) = -c*u(2)+d*u(1)*u(2);
endfunction
> t = linspace(0,10,100)';
> sol = lsode('presadepre',[0.5;0.5],t);
```

Primero hemos definido la función `presadepre` que determina la ecuación diferencial y el vector columna `t` con la partición del intervalo $[0, 10]$ con 100 valores equiespaciados. Por último, en la matriz `sol`, hemos guardado el resultado de la simulación numérica, que podemos ahora representar gráficamente mediante uno de los comandos

```
> plot(t,sol(:,1),t,sol(:,2))
> plot(t,sol')
```

Si queremos representar la solución obtenida sobre el **diagrama de fases** (esto es, la aproximación a la curva $(u_1(t), u_2(t))$), lo podemos hacer mediante el comando

```
> plot(sol(:,1), sol(:,2))
```

Observamos ambas salidas gráficas en la figura 9.2. Notamos, como particularidad de este modelo, que sus soluciones son periódicas.

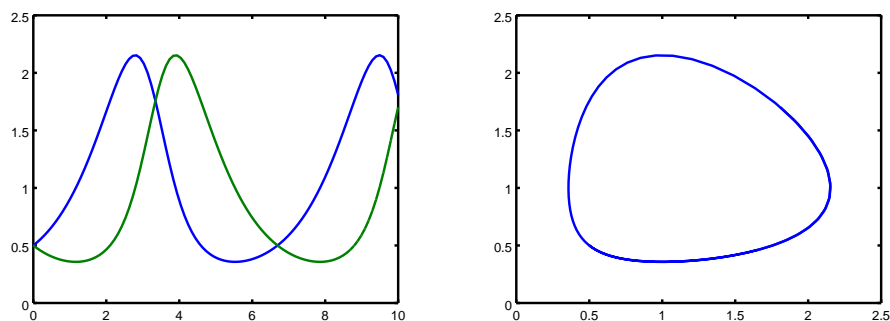


Figura 9.2: Lotka-Volterra: evolución (izda.) y diagrama de fases (dcha.)

El comando `lsode` determina, de forma automática, tanto el método a emplear como el tamaño de paso empleado en función de ciertas tolerancias preestablecidas, de manera que el usuario obtiene un resultado sin necesidad de saber nada sobre el proceso de generado. No obstante, debemos indicar que existen situaciones en las que **la simulación numérica puede no finalizar bien** y, en ese caso, el comando nos puede avisar de dos maneras: bien emitiendo mensajes de alerta, bien a través de un segundo argumento en la respuesta (`ystate`) que, en caso de ser distinto de 2, indica una finalización incorrecta del proceso. Visualizamos esto resolviendo el ejemplo $x' = 1 + x^2$ presentado en la introducción, cuya solución $x(t) = \tan(t)$ explota en $\pi/2$.

```
> f = @(x,t) 1+x.^2
> t = linspace(0, (pi/2)-10^(-7), 100);
> [x,ystate] = lsode(f,0,t);
> ystate
ystate = -5
```

devuelve múltiples WARNINGS

Algunos aspectos interesantes de la integración numérica son: el tipo y orden del método empleado (BDF, Adams, etc.), las tolerancias absolutas y relativas en el error cometido, los límites para el tamaño de paso h , y el número máximo de iteraciones N . Todos ellos pueden ser controlados mediante las opciones, `lsode_options`, del comando `lsode`, que se pueden consultar en [5, Differential Equations].

El comando `lsode` también admite que se le pase como dato la matriz Jacobiana de f , ya que esta puede ser de gran utilidad tanto para resolver ecuaciones, si el método es implícito, como para obtener estimaciones del error cometido.

Especialmente complicada es la simulación numérica de las ecuaciones denominadas **rígidas** (*stiff* en inglés). Estas se caracterizan por tener partes de la solución donde las velocidades son muy grandes (en módulo) frente a otras partes en las que, en comparación, son mucho más pequeñas. Esto puede provocar enormes errores en los métodos que no estén adaptados para esta característica. En particular, el comando `lsode` trabaja por defecto con un algoritmo válido para este tipo de problemas. Veamos un ejemplo asociado a la ecuación diferencial de un proceso químico, que da lugar a una ecuación rígida.

Ejemplo 9.7. Consideramos las siguientes reacciones químicas (reacción de Robertson),



A partir de las leyes de acción de masas, la EDO asociada al proceso anterior es

$$\left. \begin{aligned} u_1'(t) &= -k_1 u_1(t) + k_3 u_2(t) u_3(t) \\ u_2'(t) &= k_1 u_1(t) - k_2 u_2^2(t) - k_3 u_2(t) u_3(t) \\ u_3'(t) &= k_2 u_2^2(t) \end{aligned} \right\},$$

donde $u_1(t)$, $u_2(t)$ y $u_3(t)$ representan, respectivamente, las concentraciones de A , B y C en el instante de tiempo t . En el siguiente *script* se simula la ecuación diferencial anterior para valores concretos de las tasas k_i y condiciones iniciales $[1; 0; 0]$ (únicamente hay sustancia A); además se representan gráficamente las soluciones⁶.

```
1 b = 10^12; % Extremo superior de integración
2 t = [0, 10.^linspace(-4, 12, 100)];
3 function up = f(u, t) % definición de la EDO
4     n = 0; k1 = 0.04; k2 = 3*10^7; k3 = 10^4;
5     up(1) = -k1*u(1)+k3*u(2)*u(3);
6     up(2) = k1*u(1)-k2*u(2)^2-k3*u(2)*u(3);
7     up(3) = k2*u(2)^2;
8 endfunction
9 [sol, ystate] = lsode('f', [1; 0; 0], t); % resolución
10 % representación gráfica
11 subplot(2, 1, 1)
12 semilogx(t(2:end), sol(2:end, [1, 3])) % gráfica de u(1) y u(3)
13 subplot(2, 1, 2)
14 semilogx(t(2:end), sol(2:end, 2), 'r') % gráfica de u(2)
15 ystate
```

Como el proceso se ha simulado en el intervalo $[0, 10^{12}]$, para percibir claramente la variación de las variables, en la representación gráfica se ha usado una escala logarítmica en el eje x (mediante el comando `semilogx`). La elección de vector t (descrita en la

⁶Notamos en este código el uso del comando `subplot(n, m, r)`, que crea una matriz de gráficas con n filas y m columnas, colocando la gráfica ejecutada en la posición r de dicha tabla.

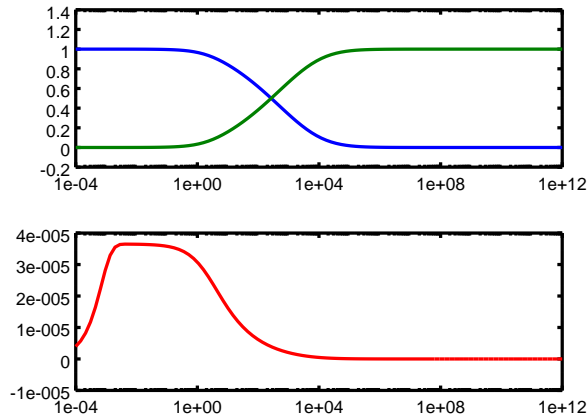


Figura 9.3: Ejemplo de ecuaciones rígidas: reacción de Robertson.

línea 2) que se pasa al comando `lsode` está motivada por este hecho. Para constatar el problema de rigidez de este sistema, observamos en la figura 9.3 que el rango de variación de u_2 es mucho menor que el de las otras dos variables (por eso lo hemos representado por separado).

En cualquier caso, una ventaja innegable de este comando de Octave con respecto a los análogos de programas comerciales es que, al ser de código abierto, tenemos acceso al funcionamiento íntegro del comando, que está descrito en [12], y podemos adaptarlo a nuestras necesidades específicas en cada caso.

9.2.2. Implementación del método Runge-Kutta de orden 4

Aunque el comando `lsode` será suficiente para nuestros propósitos, nos puede interesar programar nuestros propios métodos con Octave. Proponemos la implementación básica de un método que destaca por su sencillez y gran aplicabilidad para ecuaciones no rígidas, como es el método explícito de Runge-Kutta de cuatro etapas y orden 4.

Algoritmo 9.8 (Método de Runge-Kutta).

- $y_0 = u_0$;
- Para $n \geq 0$,
 - $K_1 = f(y_n, t_n)$ ◦ $K_2 = f(y_n + \frac{h}{2}K_1, t_n + \frac{h}{2})$,
 - $K_3 = f(y_n + \frac{h}{2}K_2, t_n + \frac{h}{2})$, ◦ $K_4 = f(y_n + hK_3, t_n + h)$,
 - $y_{n+1} = y_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4)$.

La implementación queda descrita en la siguiente función.

```

1 function [t,y] = rk4solver(f,t0,tfinal,u0,N)
2 % Resuelve el PVI: u'(t)=f(u(t),t) en [t0,tfinal]
3 % empleando el método de Runge-Kutta explícito de orden 4.
4 % En el caso n-dimensional, u0 es un vector columna.
5 % Devuelve:
6 %   t = vector con los nodos;
7 %   y(:,i) = vector con las aproximaciones de u(t(i)).
8   h = (tfinal-t0)/N;           % h tamaño de paso temporal
9   t = linspace(t0,tfinal,N+1); % vector de nodos
10  y = u0;                       % condición inicial
11  % Bucle con evolución temporal
12  for i = 1:N
13      K1 = feval(f,y(:,i),t(i));
14      K2 = feval(f,y(:,i)+h*K1/2,t(i)+h/2);
15      K3 = feval(f,y(:,i)+h*K2/2,t(i)+h/2);
16      K4 = feval(f,y(:,i)+h*K3,t(i)+h);
17      y = [y, (y(:,i)+h*(K1+2*K2+2*K3+K4)/6)];
18  endfor
19 endfunction

```

A continuación vamos a aplicar este método al conocido como problema balístico, esto es, el cálculo de la trayectoria que sigue un cuerpo lanzado desde un determinado punto y sobre el que, tras el impulso inicial, únicamente actúa el rozamiento y la fuerza de la gravedad. Aunque reciba un nombre tan bélico, puede ser aplicado a infinidad de situaciones particulares, como puede ser el juego de golf.

Ejemplo 9.9. Suponemos que una bola de golf es lanzada, desde el origen de coordenadas, en dirección a la parte positiva del eje OX . Si llamamos $(x(t), y(t))$ a su posición en el instante de tiempo t , es fácil deducir que se cumplen las ecuaciones

$$\left. \begin{aligned} mx''(t) &= -\mu x'(t), \\ my''(t) &= -\mu y'(t) - mg, \end{aligned} \right\} \quad \text{con } (x(0), y(0)) = (0, 0), \quad (x'(0), y'(0)) = (v_x, v_y),$$

donde μ es una constante positiva relacionada con el rozamiento entre el aire y la bola, m es su masa y $g = 9.8m/s^2$ la aceleración de la gravedad (en el caso de la bola de golf $\mu/m = 0.25s^{-1}$). Mediante un adecuado cambio de variable⁷ podemos describir la trayectoria de la bola mediante coordenadas de la forma $(x, y(x))$, de manera que $y(x)$ verifica el PVI

$$\ddot{y}(x) = \frac{-g}{(v_x - \frac{\mu}{m}x)^2}, \quad y(0) = 0, \quad \dot{y}(0) = v_y/v_x = \theta, \quad (9.5)$$

donde θ es el ángulo de lanzamiento y $\frac{mv_x}{\mu}$ el alcance máximo de la bola. La notación de este problema no coincide con la que venimos usando en el resto del capítulo ya que, como se puede observar, la que empleamos es totalmente natural por su significado físico.

⁷Para quien esté interesado en ello, el cambio aludido viene dado por $t = t(x) = \frac{-m}{\mu} \ln \left(1 - \frac{\mu}{mv_x} x \right)$.

Tomando $v = \sqrt{v_x^2 + v_y^2} = 50 \text{ km/h}$ como velocidad inicial (y por tanto $v_x = v \cos(\theta)$) y $\mu/m = 0.25 \text{ s}^{-1}$, vamos a aproximar posibles trayectorias de la bola, dependiendo del ángulo del lanzamiento inicial, θ en (9.5).

```

1 global vx;
2 v = 50*1000/3600; % velocidad expresada en m/s
3 hold on
4 axis([0,15,0,8]);
5 function [d] = bolafun(z,x)
6     global vx;
7     d = [z(2); -9.8/(vx-0.25*x).^2];
8 endfunction
9 for theta = pi/16:pi/16:6*pi/16
10     vx = v*cos(theta);
11     [xnum,solnum] = rk4solver('bolafun',0,15,[0;tan(theta)],100);
12     plot(xnum,solnum(1,:))
13 endfor
14 xlabel("x");
15 ylabel("y");
16 hold off

```

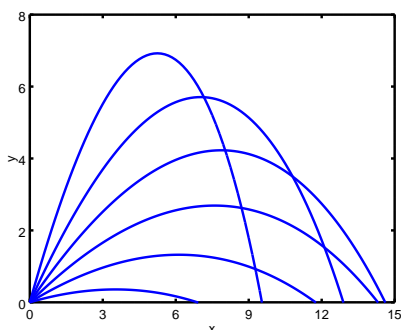


Figura 9.4: Trayectorias en función del ángulo de tiro inicial.

Son varias las observaciones que tenemos que hacer sobre este código. La velocidad inicial considerada (línea 2) se ha expresado en m/s para que sea compatible con las dimensiones del resto de las cantidades involucradas en el problema. La representación de las trayectorias, para distintos ángulos de tiro, se harán en una misma caja de dimensiones $[0, 15] \times [0, 8]$ (línea 4), ya que en caso contrario la representación muestra toda la trayectoria y puede ser poco ilustrativo. Antes de comenzar con las representaciones, fijamos la ventana gráfica mediante el comando `hold` (línea 3). La función `'bolafun'`, que define la ecuación diferencial (9.5), depende del parámetro v_x que cambia al variar el ángulo de tiro inicial, por lo que almacenamos dicha cantidad en la variable global

v_x (línea 1) y la actualizamos (línea 10) cada vez que cambiamos el valor del ángulo θ . Observemos que dicha variable es de carácter global y se ha de declarar también dentro de la función 'bolafun' (línea 6).

Ejemplo 9.10. Las trayectorias representadas en el ejemplo 9.9 necesitan de una pequeña explicación para que la persona que no está familiarizada con este tipo de técnicas las comprenda. En efecto, es lógico intentar presentar los resultados que queremos mostrar de la manera más intuitiva posible. Así, en el problema anterior, podríamos mostrar una película con la evolución temporal de la bola de golf, de manera que aparezca de forma natural el concepto de trayectoria. En este ejemplo vamos a ver cómo se puede generar una animación de este tipo. Para ello, vamos a emplear las gráficas que produce Octave a modo de fotogramas y las montaremos con otro programa apropiado para ello.

Con el siguiente código de Octave simulamos mediante Runge-Kutta la trayectoria de la bola de golf siguiendo el modelo anterior para un ángulo inicial de tiro fijo $\theta = \pi/4$.

```

1 1;
2 function [d] = bolafun(z,x)
3     vx = (50*1000/3600)*cos(pi/4);
4     d = [z(2); -9.8/(vx-0.25*x).^2];
5 endfunction
6 [xnum,solnum] = rk4solver('bolafun',0,15,[0;tan(pi/4)],300);
7 clf
8 axis([0,15,-1,6]);
9 hold on
10 for i = 0:100
11     plot(xnum(1+3*i),solnum(1,1+3*i),'r*');
12     saveas(1, strcat('./graftiro/imagen',num2str(i),'.png'),'png');
13     % Ojo! La carpeta graftiro ha de estar creada con anterioridad.
14 endfor
15 hold off

```

Por este motivo, no hace falta declarar la variable v_x , ya que es un dato que empleamos directamente en la definición de la función `bolafun`. Dicha aproximación numérica da información sobre la posición y velocidad de la bola para 301 valores x_i consecutivos y equiespaciados, que van desde $x(0) = 0$ hasta $x(300) = 15$. Para visualizar mejor la trayectoria, mostramos solo una de cada tres posiciones, generando así un total de 101 gráficas, de modo que en cada una de ellas se muestra un punto más que en la anterior. Para ello, tras limpiar la pantalla gráfica y fijar el recuadro $[0, 15] \times [-1, 6]$ (líneas 7 y 8), en el que se realizan todas las representaciones, se ejecuta `hold on` para que las sucesivas gráficas se superpongan. La generación de éstas se hace mediante un bucle `for` cuyo contador va de 0 a 100. Este contador sirve tanto para seleccionar el nuevo punto (línea 11), que se representa mediante un asterisco (rojo), como para darle nombre al fichero gráfico donde se guardará (línea 12). Estos ficheros se van a guardar en formato png en la subcarpeta “graftiro” del directorio de trabajo (que ha de estar creada previamente) y se denominarán “imagen#.png”, donde # será un número de 0 a 100.

Esto es posible gracias al comando `strcat` que concatena varias cadenas de caracteres: `'./graftiro/imagen'`, la salida de la función `num2str` y `'.png'`. Observemos que `num2str` toma un número y devuelve la cadena de caracteres que lo representa. Si no queremos mantener en las gráficas las posiciones anteriores, solo hay que dejar de usar el comando `hold`.

Una vez se tienen todas las gráficas en la carpeta indicada, lo único que queda es montar la animación, esto es, que las gráficas se muestren una tras otra de forma que parezca un movimiento continuo. Para ello se pueden emplear distintos programas de libre distribución, de entre los cuales nosotros hemos elegido FFmpeg, por estar disponible para una gran variedad de plataformas. Este programa, que ha de usarse desde un terminal externo a Octave, convierte los ficheros en una animación mediante la instrucción:

```
ffmpeg -i "imagen%01d.png" animacion.mpeg
```

que se ha de ejecutar una desde la carpeta “graftiro”, que es donde se encuentran las gráficas. En este caso devuelve un fichero de video denominado “animación” en formato mpeg, aunque se pueden escoger otros formatos de salida, como por ejemplo avi.

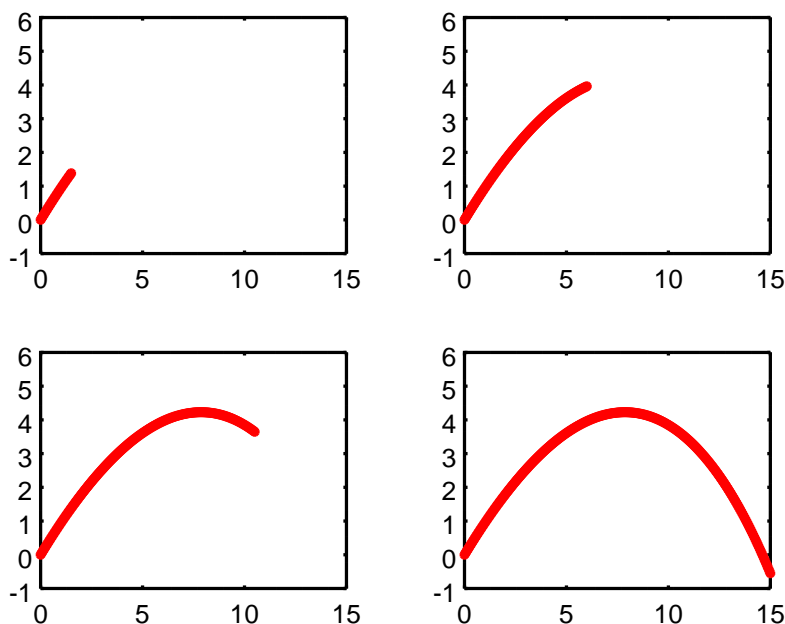


Figura 9.5: Cuatro instantáneas del recorrido de la bola.

Capítulo 10

Ecuaciones diferenciales II: Problemas de valores en la frontera

Este capítulo está dedicado a la simulación de soluciones de ecuaciones diferenciales ordinarias donde, en lugar de fijar ciertas condiciones iniciales (como hicimos en el capítulo 9), predeterminamos los valores que ha de tomar la solución en los extremos del intervalo donde se pretende aproximar la solución. Estas son las llamadas **condiciones de contorno o de frontera**. Dichos problemas se denominan genéricamente Problemas de Valores en la Frontera (PVF). Puesto que se imponen condiciones en dos puntos, las ecuaciones diferenciales asociadas suelen ser también de orden dos (aunque esto no es una regla). Veamos algunos ejemplos.

Ejemplo 10.1. La ecuación diferencial $x''(t) = -x(t)$, como vimos en el ejemplo 9.1, tiene por soluciones las funciones $x(t) = K_1 \cos(t) + K_2 \sin(t)$, con $K_1, K_2 \in \mathbb{R}$. Veamos qué ocurre con el PVF para distintas condiciones de contorno.

- Si imponemos $x(0) = 0$ y $x(\pi/2) = 1$, obtenemos la solución única $x(t) = \sin(t)$.
- Si pedimos $x(0) = 0$ y $x(\pi) = 0$, existen infinitas soluciones de la forma $x(t) = K \sin(t)$, con $K \in \mathbb{R}$.
- Por último, si imponemos las condiciones $x(0) = 0$ y $x(\pi) = 1$, observamos fácilmente que no existe solución.

A partir de este sencillo ejemplo, constatamos que la existencia y unicidad de los PVFs (así como su simulación numérica) requiere de un análisis particular, ya que una misma ecuación diferencial con distintas condiciones de contorno puede presentar situaciones bien diferentes.

10.1. Fundamentos teóricos

Para facilitar en la exposición del tema, adoptaremos la siguiente expresión estándar de un PVF unidimensional:

$$(PVF) \begin{cases} x''(t) = f(x(t), x'(t), t), & t \in [a, b], & (\text{EDO}) \\ x(a) = \alpha, x(b) = \beta. & & (\text{condiciones de frontera}) \end{cases} \quad (10.1)$$

Por simplicidad en este texto, hemos elegido condiciones de frontera de **tipo Dirichlet**. No obstante, las ideas que aquí se expongan se pueden generalizar a condiciones de frontera más generales, por ejemplo del tipo

$$a_0 x(a) - a_1 x'(a) = \alpha, \quad b_0 x(b) + b_1 x'(b) = \beta,$$

e, incluso, al caso de ecuaciones diferenciales de mayor orden.

Respecto al buen planteamiento de (10.1), hemos de comentar que son varias las ideas que se pueden usar para justificarlo. El enfoque que nosotros tomaremos viene motivado por las **técnicas de tiro**, que veremos en la última parte del tema. La idea básica es la siguiente, tomamos un valor s que podemos mover a discreción, creamos el PVI (que usa sólo la primera condición de frontera de (10.1), dejando una segunda condición inicial en función de un parámetro)

$$x''(t) = f(x(t), x'(t), t), \quad t \in [a, b], \quad x(a) = \alpha, x'(a) = s, \quad (10.2)$$

llamando $x(t; s)$ a su solución (suponemos pues que f cumple el teorema 9.3). Ahora se trata de encontrar algún valor de s tal que la correspondiente solución de (10.2) tome el valor β en $t = b$, es decir $x(b; s) = \beta$. De hecho, (10.1) tendrá tantas soluciones como ceros tenga la **función de tiro** $\phi(s) = x(b; s) - \beta$. A partir de esta idea se puede probar el siguiente resultado.

Teorema 10.2.

- *Hipótesis: sea una función $f : \mathbb{R} \times \mathbb{R} \times [a, b] \rightarrow \mathbb{R}$ tal que*
 - *$f(x, y, t)$ es continua en $\mathbb{R} \times \mathbb{R} \times [a, b]$,*
 - *tanto $\partial_x f$ como $\partial_y f$ son continuas y acotadas en $\mathbb{R} \times \mathbb{R} \times [a, b]$,*
 - *$0 < \partial_x f$ en $\mathbb{R} \times \mathbb{R} \times [a, b]$.*
- *Tesis: el problema (10.1) tiene una única solución $x(t; s_0)$ correspondiente al único cero de la función de tiro (es decir, $\phi(s_0) = 0$).*

Varios son los comentarios que merece este resultado, pudiéndose consultar sus detalles en [7]. Destacamos que la positividad de $\partial_x f$ permite demostrar que la función de tiro cumple la condición

$$\phi'(s) \geq c > 0, \quad \forall s \in \mathbb{R}.$$

Así podemos concluir que tiene un único cero y, por consiguiente, la existencia y unicidad de solución de (10.1). Pero lo más interesante para nuestros propósitos es que obtenemos valiosa información que nos permitirá analizar los métodos de tiro en la sección 10.2.2. Además, como vemos en el ejemplo 10.1 (donde $f(x, y, t) = -x$), si dicha hipótesis no se cumple, el resultado no es cierto en general.

10.2. Resolución numérica de PVFs

Octave no incluye por defecto funciones para resolver PVFs, si bien el paquete `OdePkg` sí contiene una función, `bvp4c`, para tal fin. En cualquier caso, pasamos directamente a ver cómo podemos resolver nosotros estos problemas, bien mediante **esquemas en diferencias finitas**, bien mediante **métodos de tiro**. Para el uso de `OdePkg` y `bvp4c`, nos remitimos a la ayuda de Octave [5].

10.2.1. Métodos en diferencias finitas para problemas lineales

Antes de introducir esta técnica de modo general, veamos cómo usarla a través de un ejemplo sencillo, aunque no poco interesante, que permitirá entender intuitivamente su funcionamiento.

Ejemplo 10.3. Comenzamos viendo un PVF asociado a la denominada ecuación de Poisson unidimensional,

$$x''(t) = r(t), \quad t \in [a, b], \quad x(a) = \alpha, \quad x(b) = \beta, \quad (10.3)$$

donde $r(t)$ es una función dada. Para resolverlo usaremos diferencias finitas.

En primer lugar tomamos la siguiente **partición uniforme**¹ del intervalo $[a, b]$:

$$t_n = a + nh \quad n = 0, \dots, (N+1), \quad h = \frac{1}{N+1}, \quad N \in \mathbb{N}. \quad (10.4)$$

A continuación **discretizamos las derivadas**, que aparecen en la ecuación diferencial, mediante la fórmula de diferencias centradas para la segunda derivada (descrita en la sección 8.1.1) de manera que, en los **nodos interiores** (los distintos de a y b), tenemos

$$\frac{x(t_{n+1}) - 2x(t_n) + x(t_{n-1}))}{h^2} \approx r(t_n), \quad n = 1, \dots, N,$$

mientras que en los nodos extremos, $t_0 = a$ y $t_{N+1} = b$, imponemos obviamente las condiciones de frontera, esto es $x(t_0) = \alpha$ y $x(t_{N+1}) = \beta$. Así, esta expresión nos proporciona un método para calcular el conjunto de los **valores aproximados** $y_n \approx x(t_n)$ como solución del **sistema de ecuaciones lineales** dado por

$$\left. \begin{aligned} \frac{1}{h^2} (y_{n+1} - 2y_n + y_{n-1}) &= r(t_n), \quad n = 1, \dots, N, \\ y_0 &= \alpha, \quad y_{N+1} = \beta. \end{aligned} \right\} \quad (10.5)$$

De este modo, la resolución numérica del PVF (10.3) se concreta en la resolución del sistema (10.5), cuya expresión matricial está determinada por una **matriz tridiagonal**:

¹A diferencia de los PVIs resueltos en la sección 9.1.1, ahora tomamos $N+1$ subintervalos, es decir, $N+2$ nodos equiespaciados, cuyo objetivo es obtener un sistema de ecuaciones $N \times N$.

$$\frac{1}{h^2} \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix}_{N \times N} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix} = \begin{pmatrix} r_1 - \frac{\alpha}{h^2} \\ r_2 \\ \vdots \\ r_{N-1} \\ r_N - \frac{\beta}{h^2} \end{pmatrix}, \quad (10.6)$$

donde $r_i = r(t_i)$ y las incógnitas del sistema son los valores y_i para $i = 1, \dots, N$. De esta forma la solución numérica obtenida viene dada por los valores $(\alpha, y_1, y_2, \dots, y_N, \beta)$. La implementación en Octave de dichos cálculos para el PVF (10.3) se puede hacer mediante el código recogido en el siguiente *script*, que ha sido adaptado de [18].

```

1 % Calcula una aproximación por diferencias finitas centradas del
2 % PVF: x''(t) = r(t), t en [a,b], x(a) = alpha, x(b) = beta
3 % Datos del problema
4 a = 0; b = 4;
5 alpha = 1; beta = 2;
6 r = @(t) -cos(t); % Función r(t) (evaluable sobre vectores)
7 % Definimos la discretización
8 N = 30; % Número de nodos interiores (ha de ser >2)
9 h = (b-a)/(N+1);
10 t = linspace(a,b,N+2); % Nodos, incluidos a y b
11 % Definición de la matriz de coeficientes
12 aux = ones(N,1)/h^2;
13 A = spdiags([aux -2*aux aux],[-1 0 1],N,N);
14 % Definición del vector de términos independientes
15 tint = (t(2:N+1))'; % Lista de N nodos interiores
16 rhs = r(tint); % rhs será el término independiente
17 rhs(1) = rhs(1)-alpha/h^2;
18 rhs(N) = rhs(N)-beta/h^2;
19 % Resolución del sistema
20 y = A\rhs;
21 % Representación de la solución numérica y la exacta
22 y = [alpha; y; beta];
23 plot(t,y,'*',t,alpha -1+(beta-cos(4))*t/4+cos(t))

```

A continuación analizamos con cierto detalle el código de este *script*.

- Tras la definición de los datos del problema, construimos la discretización del dominio $[a, b]$ mediante el comando `linspace` (línea 10).
- En la definición de la matriz de coeficientes del sistema (10.6) explotamos su estructura tridiagonal, esto es, de sus N^2 componentes, únicamente los $3N - 2$ valores de sus 3 diagonales principales son no nulos. Este es un ejemplo de lo que se conoce como **matriz hueca o dispersa** (*sparse matrix* en inglés). Por tanto, para que la memoria que consume Octave al resolver el sistema sea mucho menor, esta matriz puede representarse de forma especial, de manera que en su definición se indican

únicamente aquellas componentes de la matriz que son no nulas. Así pues, emplearemos el comando `spdiags` (línea 13), que requiere, en primer lugar, que se le proporcione una matriz cuyas columnas sean los valores que aparezcan en las diagonales para recolocarlos debidamente (en el ejemplo siguiente, página 150, se explica completamente el funcionamiento de `spdiags`).

- Para definir el término independiente del SEL, que guardamos en la variable `rhs`, primero evaluamos la función r sobre los nodos interiores (línea 16) y seguidamente modificamos la primera y la última componente de dicho vector restando α/h^2 y β/h^2 respectivamente, tal y como aparece en (10.6).
- Una vez definido el sistema (10.6), resolvemos con el comando `directo \` (línea 20) y mostramos el resultado junto con la solución exacta, que vemos en la figura 10.1.

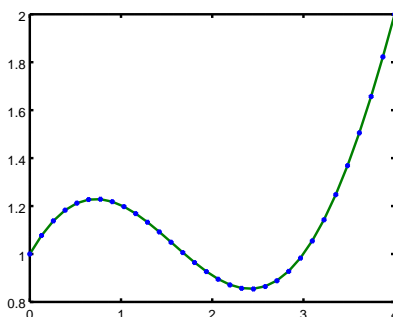


Figura 10.1: Soluciones aproximada y exacta del problema de Poisson.

A través de este caso particular, hemos podido apreciar claramente el proceso empleado: partición del intervalo, discretización de la ecuación, planteamiento y resolución del SEL con matriz tridiagonal (dispersa), y representación gráfica. Pasamos pues a presentar el método de diferencias finitas aplicado a un PVF cuya EDO asociada sea lineal de orden 2, esto es,

$$x''(t) = p(t)x(t) + q(t)x'(t) + r(t), \quad t \in [a, b], \quad x(a) = \alpha, \quad x(b) = \beta. \quad (10.7)$$

Gracias al teorema 10.2, si las funciones p , q y r son continuas y, además, $p = \partial_x f$ es positiva, entonces podemos afirmar que existe una única solución de (10.7). En este caso la notación de la **partición del intervalo** es la dada en (10.4). La **discretización de la ecuación diferencial**, empleando fórmulas de aproximación en diferencias finitas centradas en los nodos interiores, nos lleva a la expresión

$$\frac{x(t_{n+1}) - 2x(t_n) + x(t_{n-1}))}{h^2} - q(t_n) \frac{x(t_{n+1}) - x(t_{n-1}))}{2h} - p(t_n)x(t_n) \approx r(t_n), \quad n = 1, \dots, N.$$

Por tanto, en este caso **las soluciones aproximadas resuelven el SEL** siguiente:

$$\begin{cases} \frac{1}{h^2} \left(\left(1 + \frac{hq(t_n)}{2}\right) y_{n-1} - (2 + h^2 p(t_n)) y_n + \left(1 - \frac{hq(t_n)}{2}\right) y_{n+1} \right) = r_n, & n = 1, \dots, N, \\ y_0 = \alpha, & y_{N+1} = \beta, \end{cases}$$

que, matricialmente, se reescribe de nuevo mediante una matriz tridiagonal,

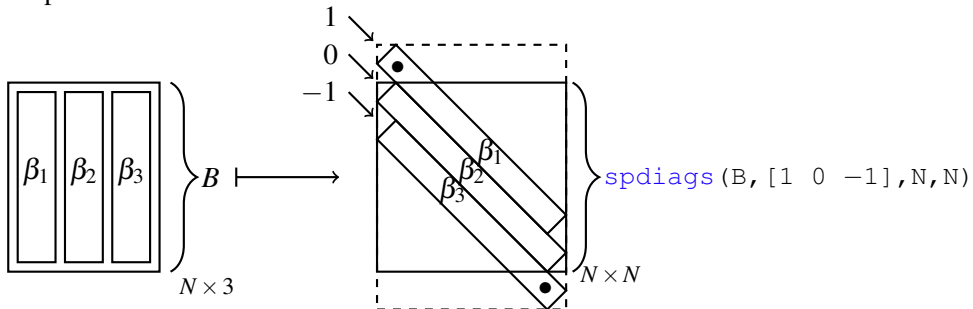
$$\begin{pmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{N-1} & a_{N-1} & c_{N-1} \\ & & & b_N & a_N \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix} = \begin{pmatrix} r_1 - \alpha b_1 \\ r_2 \\ \vdots \\ r_{N-1} \\ r_N - \beta c_N \end{pmatrix}, \quad (10.8)$$

donde $a_n = \frac{1}{h^2}(-2 - h^2 p(t_n))$, $b_n = \frac{1}{h^2}(1 + \frac{hq(t_n)}{2})$ y $c_n = \frac{1}{h^2}(1 - \frac{hq(t_n)}{2})$.

Para adaptar el *script* del ejemplo 10.3, añadiremos la declaración de las funciones p y q y construiremos el SEL (10.8) mediante el comando `spdiags` (véanse las líneas 16 y 17 del código que sigue) a partir de una matriz B cuyas columnas sean los coeficientes b_i , a_i y c_i . Concretamente `A=spdiags(B, [1 0 -1], N, N) ' crea`

$$B = \begin{pmatrix} b_1 & a_1 & c_1 \\ b_2 & a_2 & c_2 \\ \vdots & \vdots & \vdots \\ b_N & a_N & c_N \end{pmatrix} \longrightarrow A = \begin{pmatrix} a_1 & c_1 & & \\ b_2 & a_2 & c_2 & \\ & \ddots & \ddots & \ddots \\ & & b_N & a_N \end{pmatrix}.$$

Veamos en el siguiente esquema el funcionamiento de `spdiags`. Notamos que el primer valor del vector colocado en la superdiagonal y el último de la subdiagonal (notados por ●) desaparecen, por lo que primero los colocamos en orden `[1 0 -1]` y luego trasponemos.



Ejemplo 10.4. En el siguiente *script* se implementa el método propuesto y se aplica al PVF de tipo (10.7) (cuya solución exacta es $x(t) = \sqrt[3]{x} + \log(x)$) dado por

$$x''(t) = -\frac{11}{3t}x'(t) + \frac{1}{t^2}x(t) + \frac{8}{3t^2} - \frac{\ln(t)}{t^2}, \quad x(1) = 1, \quad x(4) = \sqrt[3]{4} + \ln(4).$$

```

1 % Calcula una aproximación por diferencias finitas centradas del
2 % PVF lineal: x''(t) = q(t)x(t) + p(t)x'(t) + r, t en [a,b],
3 %           x(a)=alpha, x(b)=beta
4 % Definimos los datos del problema
5 a = 1; alpha = 1;
6 b = 4; beta = 4^(1/3)+log(4);
7 p = @(t) 1./(t.^2);           % Funciones p(t), q(t) y r(t)
8 q = @(t) -11./(3*t) ;         % (evaluables sobre vectores)
9 r = @(t) 8./(3*t.^2)-log(t)./(t.^2);
10 % Definimos la discretización
11 N = 30;                       % Número de nodos interiores (ha de ser >2)
12 h = (b-a)/(N+1);
13 t = linspace(a,b,N+2); % Nodos, incluidos a y b
14 tint = (t(2:N+1))'; % Vector columna de N nodos interiores
15 % Definición de la matriz A de coeficientes
16 B = [1+q(tint)*h/2, -2-p(tint)*h^2, 1-q(tint)*h/2]./h^2;
17 A = spdiags(B, [1 0 -1], N, N)';
18 % Definición del vector de términos independientes
19 rhs = r(tint);
20 rhs(1) = rhs(1)-alpha*B(1,1);
21 rhs(N) = rhs(N)-beta*B(N,3);
22 % Resolución del sistema
23 y = A\rhs;
24 % Representación de las soluciones numérica y exacta
25 soln = [alpha; y; beta];
26 plot(t, soln, 'r', t, t.^(1/3)+log(t))

```

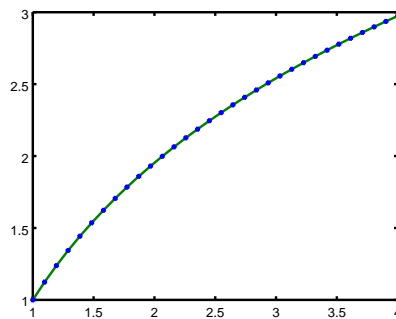


Figura 10.2: Soluciones exacta y aproximada del ejemplo 10.4.

Nótese que, si intentamos aplicar esta técnica al PVF general descrito en (10.1), el sistema de ecuaciones que aparece al discretizar la ecuación sería **no lineal**:

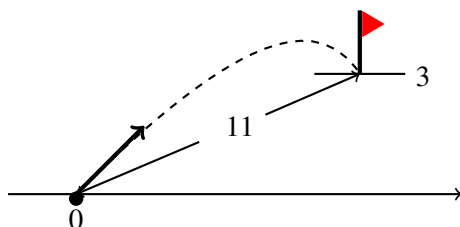
$$\frac{x(t_{n+1}) - 2x(t_n) + x(t_{n-1}))}{h^2} = f\left(x(t_n), \frac{x(t_{n+1}) - x(t_{n-1}))}{2h}, t_n\right),$$

por lo que puede ser complicado de resolver. Por ello vamos a proponer en el siguiente apartado los métodos de tiro como alternativa para PVFs no lineales.

10.2.2. Métodos de tiro simple

Este método numérico, en su versión más simple, es totalmente análogo al esquema comentado al principio del capítulo, cambiando las soluciones exactas $x(t;s)$ por sus aproximaciones numéricas, que denominaremos “tiros” por la analogía con el problema balístico visto en la sección 9.2.2. De hecho, para ilustrarlo, seguiremos con el ejemplo 9.9 de la bola de golf introducido en dicha sección.

Ejemplo 10.5. Supongamos ahora que nuestro objetivo es que la bola de golf entre, sin



tocar el *green*, en un hoyo que se encuentra 3 metros por encima del punto de tiro y a 11 metros de distancia. En primer lugar, tendríamos que ser capaces de deducir (con la ayuda del Teorema de Pitágoras: $\sqrt{11^2 - 3^2} = 10.583$) que las coordenadas de dicho punto son $(10.583, 3)$. Así, en este caso, lo que queremos es resolver el PVF dado por

$$y''(x) = \frac{-g}{(v_x - \frac{\mu}{m}x)^2}, \quad y(0) = 0, \quad y(10.583) = 3,$$

donde suponemos que aplicamos siempre la misma velocidad inicial

$$v = \sqrt{v_x^2 + v_y^2} = 50 \text{ km/h}, \quad \mu/m = 0.25 \text{ s}^{-1}, \quad v_x = v \cos(\theta),$$

siendo $\theta = y'(0)$ el ángulo a ajustar.

El objetivo del siguiente código es aproximar la solución del problema anterior, o lo que es lo mismo, encontrar un ángulo de tiro apropiado para que, al lanzar la bola, esta alcance el punto donde se encuentra el hoyo. Para ello seguiremos la siguiente estrategia.

- Realizamos dos lanzamientos de prueba tomando dos ángulos diferentes $\theta = \frac{\pi}{8}$ y $\theta = \frac{\pi}{4}$. En la figura 10.3 representamos, mediante una línea discontinua, estos dos tiros de prueba que, en este caso, han quedado uno por debajo del punto deseado de impacto y el otro por encima. Para realizar la simulación de todos los lanzamientos empleamos el algoritmo Runge-Kutta de orden 4 introducido en el capítulo 9.
- Definimos la función `errortiro`, análoga a la función de tiro $\phi(s)$ presentada en la introducción del capítulo, cuyo cero (no necesariamente único) corresponde al ángulo buscado para acertar en el hoyo.

- Para buscar el ángulo de tiro correcto, usamos el algoritmo de la secante (programado en el capítulo 2) para hallar el cero de la función 'errortiro' y mostramos, en línea continua, la solución numérica obtenida.

Como se puede observar, en el *script* diseñado, hemos empleado más variables globales pues las funciones 'bolafun' y 'errortiro' las requieren para su definición.

```

1 clear();
2 global vx v ejex ejey;
3 v = 50*1000/3600; % velocidad en m/s
4 % Definición de la ecuación diferencial
5 function [d] = bolafun(z,x)
6     global vx;
7     d = [z(2);-9.8/(vx-0.25*x).^2];
8 endfunction
9 ejey = 3;
10 ejex = sqrt(11^2-3^2);
11 % Representamos el punto donde se encuentra el objetivo
12 clf;
13 plot(ejex,ejey,'k*')
14 axis([0,12,0,5]);
15 hold on
16 % Primer tiro de prueba
17 theta = pi/8;
18 vx = v*cos(theta);
19 [xn,soln] = rk4solver("bolafun",0,ejex,[0;tan(theta)],100);
20 plot(xn,soln(1,:),':b')
21 printf("\n Diferencia tras primer tiro %f \n",ejey-soln(1,end))
22 % Segundo tiro de prueba
23 theta = pi/4;
24 vx = v*cos(theta);
25 [xn,soln] = rk4solver("bolafun",0,ejex,[0;tan(theta)],100);
26 plot(xn,soln(1,:),':b')
27 printf("Diferencia tras segundo tiro %f\n",3-soln(1,end))
28 % Función auxiliar
29 function [dif,soln] = errortiro(z)
30     global vx v ejex ejey
31     theta = z;
32     vx = v*cos(theta);
33     [xn,soln] = rk4solver("bolafun",0,ejex,[0;tan(theta)],100);
34     dif = ejey-soln(1,end);
35 endfunction
36 % Resolvemos mediante el método de la secante
37 [sol,errSec,numiter] = secante("errortiro",pi/8,pi/4,0.01,10)
38 % Representamos la solución
39 [t,soln] = rk4solver("bolafun",0,ejex,[0;tan(sol)],100);
40 plot(t,soln(1,:),':r')
41 hold off

```

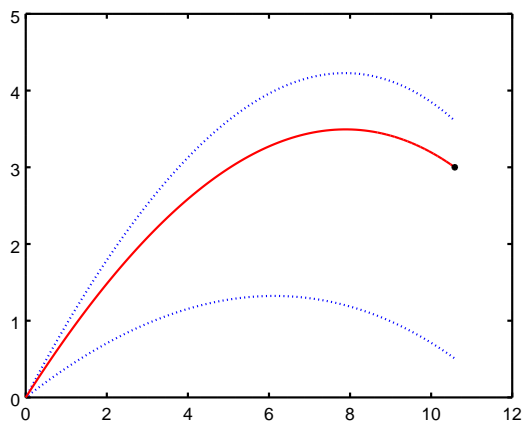


Figura 10.3: Cálculo de la trayectoria que alcanza el punto prescrito.

Para quien esté interesado en ver generalizaciones y resultados teóricos acerca de la convergencia de este tipo de técnicas, nos remitimos a los libros [7] y [14].

Ejercicios

Capítulo 1: primeros pasos con Octave

§1.1. Usando variables para evitar largos comandos, evalúa con Octave:

$$a) \frac{3\sqrt{32+e^{24}}+12}{\sin(32)^2} \quad b) \frac{\frac{(\log(27))^2}{2-2^4}}{2+\tan(4)} \quad c) \frac{\frac{\log(5^2)-4}{2x+4y}}{\frac{3(12+24x)+12}{12\cos(y)}}$$

para $(x,y) = (1,1), (2,3)$ y $(4,7)$.

§1.2. Ejecuta los siguientes comandos en el mismo orden e interpreta los resultados obtenidos; observa la relevancia del número real antes de la letra `i` al trabajar con números complejos.

```
> i = 5      > (1 + 2i)*i      > (1 + 2i)*1i
```

§1.3. Detecta errores sintácticos en los siguientes comandos de Octave.

```
> 9,5 +3      > 3 + 4 i      > clear(a)      > 2 1 - 3
```

§1.4. Ejecuta los siguientes comandos y observa la influencia de los espacios en blanco.

```
> [1 - 1]      > [1 -4i]      > [sin(pi)]
> [1 -1]      > [1 - 4i]      > [sin (pi)].
```

§1.5. Ejecuta los siguientes comandos e interpreta las diferencias en los resultados.

```
> x=1; 2 < x < 4      > x=1; 4 > x > 2      > x=1; (2< x & x< 4)
```

§1.6. Construye dos vectores columna, u y v , de igual longitud, realiza las operaciones `dot(u,v)`, `u.*v`, `u'*v`, `u*u'`, `u*v`, y compara los resultados.

§1.7. Dado un cierto vector $x = (x_0, x_1, \dots, x_n)$ necesitamos calcular $h = (h_1, \dots, h_n)$ donde $h_i = x_i - x_{i-1}$. Indica los comandos de Octave necesarios para realizar esta operación y aplícalos al vector $x = (1, 3, 5, 7, \dots, 99, 101, 103)$. Después consulta la página 123.

§1.8. Escribe una función que determine si un número x es o no natural.

- §1.9. Compara los resultados de sumar los 20 y los 30 primeros números de la sucesión $\{\frac{1}{2^n}\}$ y analiza los resultados jugando con el comando `format`.
- §1.10. Crea una matriz 100×80 cuyas componentes sean $a_{i,j} = \theta 2^i / 3^j$, siendo $\theta = 1$ en las filas impares y $\theta = -1$ en las filas pares.
- §1.11. El algoritmo de Horner-Ruffini es un algoritmo para la evaluación eficiente de un polinomio que, como hemos comentado en el capítulo 1, es el que realiza internamente el comando `polyval`. Consiste en reescribir el polinomio $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ de la forma

$$p(x) = a_0 + x(a_1 + x(a_2 + \dots x(a_{n-1} + a_n) \dots)),$$

y usar este orden de operaciones para la evaluación de p . Además, como vemos en el siguiente algoritmo, proporciona una técnica para calcular el cociente y el resto de dividir el polinomio entre el monomio $x - r$.

Algoritmo 11.1 (Horner-Ruffini). Sea un número real o complejo r y un polinomio $p(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$.

- Tomamos $b_n = a_n$
- Para $k = (n-1), (n-2), \dots, 0$ construimos $b_k = b_{k+1}r + a_k$.

Como resultado se obtienen dos cosas:

- **Evaluación** (Horner): $b_0 = p(r)$
- **División y raíces** (Ruffini): $p(x) = (x - r)q(x) + b_0$ donde $q(x)$ es el polinomio $q(x) = b_nx^{n-1} + b_{n-1}x^{n-2} + \dots + a_2x + b_1$.

En la literatura, este algoritmo suele representarse de forma gráfica como:

	a_n	a_{n-1}	a_1	a_0
	\downarrow	$+$	$+$	$+$
		b_nr	b_2r	b_1r
r	a_n	$b_nr + a_{n-1}$	$b_2r + a_1$	$b_1r + a_0$
	$= b_n$	$= b_{n-1}$	$= b_1$	$= b_0$
	coeficientes de $q(x)$			valor de $p(r)$.

Crea una función cuyos argumentos sean un polinomio p (recuerda que en Octave, p no es más que un vector con los coeficientes del polinomio ordenados) y un número r y cuya salida sea $p(r)$. Hazlo de manera que r pueda ser un vector y, por lo tanto, la salida sea el correspondiente vector de evaluaciones.

- §1.12. Modifica el programa anterior, para que produzca como segunda salida el polinomio $q(x)$.
- §1.13. Comprueba que no puedes sumar directamente con Octave los vectores que representan a los polinomios $p(x) = x^2 + 6x - 7$ y $q(x) = 2x^3 + 13x^2 - 8x - 7$. ¿Sabrías indicar el motivo? Define una función que, de manera automática, sume dos polinomios cualesquiera en Octave (independientemente de su grado).

§1.14. Comprueba que la siguiente función

```
> function [y] = f(x) y = 1; endfunction
```

al ser evaluada sobre vectores no actúa componente a componente. Ejecuta $1+0*x$, siendo x un número, un vector y una matriz. ¿Cuál es el resultado? Emplea, esta idea para definir la función constantemente igual a 1 de manera que actúe elemento a elemento sobre matrices o vectores.

§1.15. Una persona, profundamente embriagada y algo impertinente, se ha visto abandonada por un taxista en un puente iluminado por farolas equidistantes. Debido a su estado de embriaguez, su camino sigue las siguientes premisas:

- a) Únicamente realiza trayectos de una farola a una de las dos farolas contiguas (a la izquierda o la derecha) y, una vez que ha llegado, se sujeta a ella y descansa un ratito.
- b) Tras el descanso no recuerda el camino que llevaba, así que tira una moneda y se desplaza a la farola de su izquierda, si sale cara, o a la de su derecha, si sale cruz, y si queda de canto, descansa otro ratito.

Intenta diseñar una función que reproduzca este paseo aleatorio y nos indique, tras 20 tiros de moneda, a qué distancia estará este señor del punto donde lo dejó el taxista.

§1.16. Reconstruye la función `dhont.m` del ejemplo 1.4 para que, en caso de empate al asignar el último escaño, actúe como sigue:

- primera regla: se asigna el escaño al partido que haya obtenido más votos;
- segunda regla: en caso de que los dos (o más) partidos involucrados hayan obtenido los mismos votos, se asignará por sorteo.

Para aplicar la primera regla, se puede aprovechar que `[valor,p] = max(x)` asigna a `p` sólo la primera posición en la se se alcanza el valor máximo, por lo que combinado con `sort`, puede resolver este problema.

§1.17. Usando el comando `polyint`, calcula la primitiva de un polinomio que en 0 vale 10.

Capítulo 2: ecuaciones no lineales

§2.18. Usando exclusivamente el comando `fplot`, localiza un intervalo de longitud 0.1 donde se encuentre la única raíz de la función $g(x) = x + e^x$. Quizás te sea de ayuda activar el mallado de la ventana gráfica mediante el comando `grid on` o buscar en la ayuda de `fplot` posibles argumentos opcionales.

§2.19. Dada la función $f(x) = 2x + e^x$, ¿cómo podríamos constatar, con comandos de Octave, que tiene una raíz en el intervalo $[-5, 5]$?

- §2.20. Comprueba que el código `biseccion.m` produce un resultado equivocado si se le proporcionan datos que no cumplan la condición de cambio de signo (es decir, que no se verifica la condición $f(a) * f(b) < 0$). Para ello usa la función $f(x) = x^2 - 1$ en el intervalo inicial $[0, 0.5]$. Basándote en el ejercicio §2.19, añade al código un filtro para comprobar inicialmente si se verifica esta condición y que, en caso contrario, aborte el proceso e informe del problema acaecido (para lo que puedes emplear el comando `error`).
- §2.21. Comprueba que se puede programar el mismo algoritmo de bisección substituyendo el bucle `for` por un `while`. ¿Cual sería la condición a verificar en cada iteración?
- §2.22. Completa el algoritmo de la secante con un criterio de parada, sobre el valor del residuo, del tipo $|f(x_k)| < \delta$.
- §2.23. Compara el rendimiento de los tres métodos de aproximación de raíces en un ejemplo concreto. Para llevar a cabo esta comparación, puedes observar la exactitud obtenida frente al tiempo de cómputo, que se puede determinar mediante `tic toc` o el comando `cputime`¹.
- §2.24. Cuando pedimos una hipoteca por una cantidad C , a pagar durante n años en mensualidades fijas, a un interés nominal anual fijo i (porcentual), el pago mensual se obtiene mediante la fórmula²

$$\text{Letra} = L(i) = \frac{Ci}{1200} \frac{\left(1 + \frac{i}{1200}\right)^{12n}}{\left(1 + \frac{i}{1200}\right)^{12n} - 1}.$$

Supongamos que solicitamos una hipoteca de 120 000,00 euros y que queremos pagarla, en no más de 20 años, mediante pagos mensuales que no superen los 700 euros. ¿Cuál es el máximo interés que se ajusta a nuestras necesidades? En este caso, da una aproximación de i de manera que la cuota mensual correspondiente no difiera más de 2 euros de la deseada. Para ello realizaremos los siguientes pasos.

- Define en Octave la función $f(x) = L(x) - 700$, según los datos que se indican.
- Comprueba que dicha función tiene una raíz en el intervalo $(2, 6)$.
- Aproxima el valor de la raíz de $f(x)$ mediante el método de la secante.

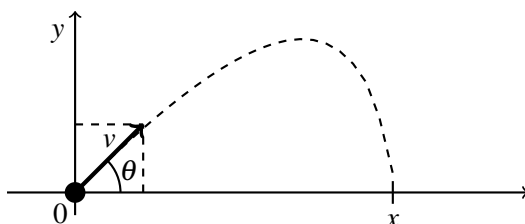
¹El comando `cputime` devuelve el tiempo de la CPU, en segundos, empleado en cada sesión de Octave, por lo que el tiempo dedicado a un proceso en concreto se puede obtener como la diferencia del tiempo de CPU antes y después de ejecutar dicho proceso.

²En realidad, la fórmula está calculada para meses ideales de 30 días, por lo que en préstamos reales, siempre hay una leve variación de la cuota resultante, que se ve incrementada aproximadamente en un factor $365/360$.

d) Escribe a continuación la respuesta al problema planteado a partir de los resultados obtenidos.

Si el interés lo fija el prestamista al 3 % y aún queremos que nuestros pagos mensuales no superen los 700 Euros, ¿a cuántos años debemos pedir la hipoteca?

§2.25. Una bola de golf es lanzada hacia la derecha desde un punto determinado. Podemos suponer que dicho punto coincide con el origen de coordenadas. Si llamamos $y(x)$ a la altura de la bola cuando ha recorrido x metros (en horizontal),



se puede deducir que la expresión de dicha trayectoria viene dada por

$$y(x; \theta) = \operatorname{tg}(\theta)x - \frac{g}{2v^2 \cos^2(\theta)}x^2,$$

donde $g = 9.8m/s^2$ es la aceleración de la gravedad y, además, suponemos que es la única fuerza que actúa sobre la bola una vez que ha sido impulsada inicialmente a una velocidad $v = 10m/s$ con un ángulo de tiro θ .

Busca los posibles ángulos con los que se puede hacer el lanzamiento para que la bola llegue a un agujero que dista del punto de lanzamiento 20 metros (en horizontal) y está a 3 metros sobre él.

§2.26. Considera la ecuación $x^3 - 9x^2 + 24x - 18 = 0$. Aplica a mano el método de Newton-Raphson con $x_0 = 3 + \sqrt{3/5}$ y verifica que se obtiene una sucesión de aproximaciones que va alternando los valores $3 + \sqrt{3/5}$ y $3 - \sqrt{3/5}$. Si ahora implementamos Newton-Raphson tomando $x_0 = 3.774597... = 3 + \sqrt{3/5}$, ¿qué ocurre? ¿Por qué?

§2.27. Considera la ecuación $x^3 - 8 = 0$ en el intervalo $[0, 2]$. ¿Qué ocurre si aplicas el método de bisección? ¿Y si empleas Newton-Raphson con $x_0 = 0$? Modifica adecuadamente los códigos de ambos métodos para que casos como el propuesto den respuestas coherentes.

§2.28. Programa el método de Regula-Falsi a partir de los códigos presentados para la resolución aproximada de ecuaciones no lineales. ¿Qué código tomarías como punto de partida?

Capítulo 3: SEL: métodos directos

§3.29. Un cierto problema de ecuaciones diferenciales (véase el ejercicio §10.92) se reduce a resolver el sistema lineal

$$\begin{pmatrix} a & 1 & 0 & \cdots & 0 & 1 \\ 1 & a & 1 & 0 & \cdots & 0 \\ 0 & 1 & a & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & a & 1 \\ 1 & 0 & \cdots & 0 & 1 & a \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix} = h^2 \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix},$$

donde $a = -2 - h^2$ y $f_n = f(nh)$ con $h = \frac{1}{N+1}$. Para $N = 9$, define la matriz de coeficientes A , usando los comandos `diag` y `eye`, y el vector de términos independientes del sistema anterior tomando $f(x) = 10\cos(2\pi x)$. Resuelve el sistema multiplicando A^{-1} por el término independiente y mediante `mldivide` o `\`.

§3.30. Aplica el programa `gausspiv` al sistema con coeficientes $a_{i,j} = \alpha^i/(\alpha-1)^j$ y término independiente $b_i = \sqrt[i]{\alpha}$, con $\alpha = 100$ e $i, j = 1, \dots, 100$. Comprueba que se han permutado varias filas. Ahora repite el proceso, pero realiza previamente esas mismas permutaciones en la matriz ampliada $(A|b)$ y verifica que, al volver a aplicar el método de Gauss, no se realiza permutación alguna.

§3.31. Debemos tener en cuenta que el programa `gausspiv` no funciona correctamente cuando la matriz de coeficientes es singular. En efecto, la solución devuelta está expresada en función de `NaN` (es decir Not a Number). Esto es debido a que, en estos casos, alguno de los pivotes es nulo y se produce una división por cero. Modifica el código de `gausspiv` para que, en caso de que la matriz sea singular, avise de este hecho y detenga el proceso, pudiéndose emplear el comando `error` para ello.

§3.32. A partir del código `gausspiv`, programa el método Gauss eliminando el pivoteo y observa, resolviéndolo, que el SEL del ejemplo 3.9 de la página 54 es altamente inestable.

§3.33. En las líneas 14 y 15 del código `gausspiv` se ha implementado de forma vectorizada el proceso de anulación de elementos bajo la diagonal. Una forma no vectorizada, aunque tal vez más intuitiva, de hacer el mismo proceso es

```
1   for i = j+1:dim
2       m = Ab(i,j)/Ab(j,j);
3       Ab(i,j:dim+ncb) = Ab(i,j:dim+ncb)-m*Ab(j,j:dim+ncb);
4   endfor
```

que recorre una a una las filas bajo el pivote y realiza las transformaciones elementales también una a una. Crea un nuevo programa usando este fragmento de código en lugar del contenido en las citadas líneas y aplica ambos al sistema del ejercicio §3.29, comparando los tiempos de resolución mediante `tic-toc`.

§3.34. En algunas ocasiones, incluso el método de Gauss con pivote parcial no proporciona una respuesta correcta. Compruébalo con el ejemplo $Ax = b$, donde

$$A = \begin{pmatrix} 10^{-16} & -1 \\ 10^{-16} & 10^{-16} \end{pmatrix} \text{ y } b = \begin{pmatrix} -1 \\ 2 \times 10^{-16} \end{pmatrix}.$$

Para solventar dichas inestabilidades, se propone otra estrategia, el *pivoteo parcial con escalado*, que consiste en escalar (es decir, multiplicar por un factor no nulo) cada ecuación antes de elegir pivote, de forma que el máximo de los elementos de cada fila de la matriz de coeficientes sea siempre 1 (en valor absoluto).

Programa el método de eliminación gaussiana usando pivoteo parcial con escalado y aplícalo al sistema anterior.

§3.35. Observa que el código `sustregr` está preparado para recibir en la variable `c` una matriz cuyas columnas serán tratadas como términos independientes. Usando como matriz `c` la matriz identidad, emplea esta funcionalidad para calcular la inversa de la matriz

$$A = \begin{pmatrix} 3 & 1 & 2 \\ 0 & 4 & 3 \\ 0 & 0 & 2 \end{pmatrix}.$$

§3.36. Al igual que el código `sustregr`, se verifica que `gausspiv` está preparado para recibir en la variable `c` una matriz, por lo que, eligiendo de nuevo como término independiente la matriz identidad, podemos obtener la inversa de una matriz cualquiera A de un modo bastante eficiente.

Otro cálculo bastante costoso de realizar, a partir de su definición, es el del determinante de una matriz $n \times n$, cuyo coste computacional es $(n!n - 1)$. De nuevo el método de Gauss se puede ver como una herramienta eficiente para calcularlo³. Modifica el código `gausspiv` para crear uno que calcule simultáneamente el determinante y la inversa de una matriz dada A .

§3.37. Modifica el código `sustregr` para que detecte si la matriz de entrada es efectivamente triangular inferior y se detenga, dando un mensaje de error, en caso contrario. Emplea para ello los comandos `tril` y `error`.

³Recordamos que al permutar las filas de una matriz, su determinante cambia de signo y que al multiplicar una fila por un número, su determinante también queda multiplicado por ese mismo número.

§3.38. Comprueba que el sistema

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 4 & 6 & 8 \end{pmatrix} x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad (11.1)$$

no se puede resolver mediante el algoritmo de Gauss sin hacer permutaciones por filas. Aplica el algoritmo `gausspiv` y observa las permutaciones que se han empleado para su resolución. Nos proponemos obtener, con comandos de Octave, la matriz de coeficientes y vector de términos independientes del sistema permutado mediante dos caminos distintos. La aproximación más directa es realizar dicha permutación tal y cómo se explicó en el primer capítulo.

```
> Aperm=A(p, :)
> bperm=b(p)
```

donde p es el vector de permutaciones devuelto por `gausspiv(A,b)`, y A, b son, respectivamente, la matriz de coeficientes y vector de términos independientes del sistema (11.1). Comprueba que se obtiene el mismo resultado multiplicando A y b por la izquierda por la llamada matriz de permutación, la cual se obtiene mediante el comando $P = \text{eye}(\text{size}(A)) (p, :)$, o bien el comando $P = \text{diag}(p, 0)$.

§3.39. Comprueba, con el código implementado en la sección 3.2.3, que la matriz de coeficientes del sistema (11.1) no admite una descomposición LU directa, es decir, sin permutación de filas. Realiza la factorización LU para la matriz A permutada (calculada en el ejercicio anterior), adaptando el *script* de la descomposición LU. Compara el resultado con el devuelto por `gausspiv`.

§3.40. El objetivo del siguiente ejercicio es comprobar que la resolución del sistema

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2+\varepsilon & 5 \\ 4 & 6 & 8 \end{pmatrix} x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

se puede hacer mediante descomposición LU directa, aunque su cálculo es inestable. Para ello podemos comprobar con lápiz y papel que la descomposición de Doolittle de esta matriz es

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & \frac{2}{\varepsilon} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 1 & 1 \\ 0 & \varepsilon & 3 \\ 0 & 0 & 4 - \frac{6}{\varepsilon} \end{pmatrix}$$

y, en consecuencia,

$$x = \frac{1}{2\varepsilon - 3} \begin{pmatrix} 4\varepsilon - 7 \\ 2 \\ -2\varepsilon + 2 \end{pmatrix}.$$

Comprueba que, sin embargo, el código de descomposición LU devuelve resultados diferentes dependiendo del valor de ε . Así, si $\varepsilon = 10^{-16}$, el código indica que no se puede obtener descomposición LU. Sin embargo, con $\varepsilon = 0.5 * 10^{-15}$, el código sí devuelve descomposición LU, pero la solución del sistema obtenida a partir de esta descomposición es inexacta. ¿Sabrías cómo hacer que dicho código funcione correctamente?

§3.41. Cuando la matriz de coeficientes es tridiagonal, el algoritmo de descomposición LU (de Doolittle) puede ser simplificado, ya que tanto L como U resultan ser también matrices tridiagonales, reduciendo de forma drástica el número de operaciones a realizar. El método resultante es la primera parte del llamado **método de Thomas** y viene dado por

$$A := \begin{pmatrix} a_1 & c_1 & & 0 \\ b_2 & a_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & b_n & a_n \end{pmatrix} = \begin{pmatrix} 1 & 0 & & 0 \\ \beta_2 & 1 & \ddots & \\ & \ddots & \ddots & 0 \\ 0 & & \beta_n & 1 \end{pmatrix} \begin{pmatrix} \alpha_1 & c_1 & & 0 \\ 0 & \alpha_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & 0 & \alpha_n \end{pmatrix} = LU,$$

donde $\alpha_1 = a_1$ y $\beta_i = \frac{b_i}{\alpha_{i-1}}$, $\alpha_i = a_i - \beta_i c_{i-1}$ para $i = 2, \dots, n$.

Programa dicho algoritmo y aplícalo a la matriz del ejercicio §3.29 con $N = 30$. Compara el tiempo de cómputo entre este algoritmo y el algoritmo normal usando el comando `tic toc`.

§3.42. Para completar el **método de Thomas**, podemos resolver el sistema $Ax = b$ mediante la factorización obtenida y teniendo en cuenta que los sistemas $Ly = b$ y $Ux = y$ se pueden simplificar bastante en el caso tridiagonal.

$Ly = b$ se resuelve como: $y_1 = b_1$, $y_i = b_i - \beta_i y_{i-1}$, $i = 2, \dots, n$,

$Ux = y$ se resuelve como: $x_n = \frac{y_n}{\alpha_n}$, $x_i = \frac{y_i - c_i x_{i+1}}{\alpha_i}$, $i = (n-1), \dots, 1$.

Programa un método combinado con el ejercicio anterior, cuyas entradas sean A y b , y que, tras descomponer la matriz, resuelva el sistema.

Capítulo 4: SEL: métodos iterativos

§4.43. Modifica el programa `gausseidel` de forma adecuada y crea una nueva función que implemente el método de Jacobi.

§4.44. Genera un SEL 20×20 , estrictamente diagonal dominante y con entradas aleatorias, mediante los comandos

```
> n=20; A=rand(n)+n*eye(n); b=n*rand(n,1);
```

y resuélvelo usando los métodos de Jacobi y de Gauss-Seidel. Verifica que la convergencia del segundo es más rápida.

§4.45. Repite varias veces el ejercicio anterior tomando sucesivamente como matriz de coeficientes la obtenida mediante el comando `A=rand(n)`. ¿Convergen siempre los métodos empleados? ¿Qué hemos cambiado cualitativamente en el sistema a resolver para que la convergencia en este caso no esté asegurada?

§4.46. Modifica el programa `gausseidel` de forma adecuada y crea una nueva función que implemente el método de relajación, añadiendo como entrada el peso ω . Aplícalo al sistema del ejercicio §3.29 con $x^{(0)} = (0, 0, 0, 0, 0, 0)^t$, tolerancia 10^{-2} y para distintos valores de ω . Compara la velocidad de convergencia de los métodos de Gauss-Seidel, de Jacobi y de relajación (con distintas elecciones de ω) para el presente sistema.

§4.47. En este ejercicio pretendemos realizar una mejora del algoritmo de gradiente conjugado.

a) Comprueba que las siguientes igualdades son correctas.

$$r_{k+1} = r_k - \alpha_k A p_k, \quad \alpha_k = \langle r_k, r_k \rangle / \langle p_k, A p_k \rangle, \quad \beta_k = \langle r_{k+1}, r_{k+1} \rangle / \langle r_k, r_k \rangle.$$

b) A partir de la información proporcionada en el apartado anterior, programa una versión mejorada del método de gradiente conjugado. Observa que podemos guardar los cálculos que se repiten, $w_k = A p_k$, $c_k = \langle r_k, r_k \rangle$ y $d_{k+1} = \langle r_{k+1}, r_{k+1} \rangle$ para ahorrar tiempo de cómputo.

Algoritmo 11.2 (Gradiente conjugado mejorado).

- $x^{(0)}$; $r_0 = b - A x^{(0)}$; $p_0 = r_0$; $c_0 = \langle r_0, r_0 \rangle$.
- Para $k = 0, 1, 2, \dots$
 - $w_k = A p_k$
 - $\alpha_k = c_k / \langle p_k, w_k \rangle$
 - $x^{(k+1)} = x^{(k)} + \alpha_k p_k$
 - $r_{k+1} = r_k - \alpha_k w_k$
 - $d_{k+1} = \langle r_{k+1}, r_{k+1} \rangle$
 - $\beta_k = d_{k+1} / c_k$
 - $p_{k+1} = r_{k+1} + \beta_k p_k$
 - $c_{k+1} = d_{k+1}$

c) Compara este algoritmo mejorado con el algoritmo de gradiente conjugado dado en la sección 4.2.2, comprobando el ahorro de tiempo en el cálculo mediante el uso del comando `tic-toc`. Considera el sistema que aparece en el ejercicio §3.29, con $N = 100$, para realizar las comparaciones.

§4.48. Programa el algoritmo de descenso rápido y compáralo con el de gradiente conjugado aplicando ambos al sistema del ejercicio §3.29 con $N = 100$.

Capítulo 5: valores y vectores propios

§5.49. La matriz:

$$A = \begin{pmatrix} -2 & 1 & \cdots & 0 \\ 1 & -2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \cdots & 1 & -2 \end{pmatrix}_{N \times N}$$

aparece en numerosos problemas relacionados con ecuaciones diferenciales como el descrito en el ejemplo 5.2 de la página 69. Por fijar ideas tomaremos $N = 20$.

- Sabiendo que es definida negativa... ¿Cómo podríamos aplicar el método de Cholesky para resolver este sistema?
- Calcula el radio espectral de la matriz de Gauss-Seidel.
- Verifica que, aunque no se cumple la hipótesis del teorema 4.1 (la matriz A no es estrictamente diagonal dominante), tanto el método de Jacobi como el de Gauss-Seidel sí convergen en este caso. (Nota: usa que un método es convergente si el radio espectral de su matriz B asociada es menor que 1.)

§5.50. Considera el sistema del ejercicio §3.29 tomando $f(x) = 10 \cos(2\pi x)$, y $N = 20$.

- Justifica mediante el teorema 5.4 (de los discos de Gerschgorin) que la matriz A es definida negativa.
- Obtén dichos valores propios y compáralos con la información que tenemos del apartado anterior.
- Por ser estrictamente diagonal dominante el método de Jacobi sabemos que es convergente. No obstante calcula el radio espectral de la matriz de Jacobi, B_J , empleando el código anterior.
- Justifica la convergencia de Gauss-Seidel.

§5.51. Observa que un polinomio genérico $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ se puede ver como el polinomio característico de la matriz

$$C = \begin{pmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & -\frac{a_{n-3}}{a_n} & \cdots & -\frac{a_0}{a_n} \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix}_{n \times n}.$$

Verifica con algunos ejemplos que el comando `compan(p)` descrito en el Capítulo 1 produce esta misma matriz y define una función que devuelva las raíces de $p(x)$ calculándolas como los valores propios de la matriz C .

§5.52. Comprueba con la matriz

$$A = \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}$$

que si la matriz tiene dos valores propios del mismo módulo los algoritmos de potencias y QR pueden no converger.

§5.53. Aplica el método de potencias a la matriz del ejercicio §5.49, de dimensión 20×20 , tomando como vector inicial $(1, \dots, 1)^t$. Observa lo que ocurre. Indica si, a la vista de la teoría expuesta, es razonable lo que obtienes.

Este ejercicio justifica el porqué, en la aplicación del método de potencias, se ha propuesto tomar el vector inicial como aleatorio.

§5.54. Sin utilizar que sabemos que los valores propios de la matriz del ejemplo 5.1 son negativos, y siguiendo las ideas de la página 71, crea un *script* para calcular el valor propio más pequeño en modulo.

§5.55. En la referencia [23] se publican los resultados de un estudio sobre la evolución de la población de osos *grizzly* en el parque Yellowstone, siguiendo el modelo generalizado de Leslie [17]. Parte de los datos recogidos se recopilan en la siguiente matriz

$$\begin{pmatrix} 0 & 0 & 0 & 0.40 & 0.41 & 0 & 0 & 0 & 0 & 0 \\ 0.78 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.87 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.87 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.87 & 0.89 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.02 & 0.01 & 0 & 0 & 0 & 0.39 & 0.39 \\ 0.10 & 0 & 0 & 0 & 0 & 0.79 & 0 & 0 & 0 & 0 \\ 0 & 0.04 & 0 & 0 & 0 & 0 & 0.84 & 0 & 0 & 0 \\ 0 & 0 & 0.04 & 0 & 0 & 0 & 0 & 0.84 & 0 & 0 \\ 0 & 0 & 0 & 0.04 & 0.03 & 0 & 0 & 0 & 0.84 & 0.84 \end{pmatrix},$$

cuyo valor propio dominante y vector propio asociado tienen una importante interpretación biológica. Calcúlos. Si te interesa profundizar en este tipo de modelado y su interpretación te recomendamos leer [10, 17, 23].

Capítulo 6: interpolación

§6.56. Calcula el polinomio que interpola la tabla de datos

x	1	2	4	8	16
y	10	20	40	50	100

Además, representa el polinomio y los datos de la tabla en una misma gráfica.

§6.57. Calcula, empleando comandos de Octave, los polinomios de la base de Lagrange asociados a los nodos $\{0, 2, 4, 6, 8\}$.

§6.58. En algunas situaciones puede ser que, más que obtener la expresión concreta del polinomio interpolador, nos interese saber cuánto vale este sobre ciertos valores. Por ejemplo, si queremos representar un polinomio interpolador mediante el comando `plot` no nos hace falta su expresión, si no, cuánto vale al evaluarlo sobre una serie de valores de abscisas. Existen algoritmos específicos para tal fin, como el *Algoritmo de Neville* (véase [15]), pero en este caso hemos optado por adaptar el código `difdiv` de una forma eficiente para que, a partir de los datos de interpolación, (x_i, y_i) , y las abscisas, z , donde queremos que se evalúe el polinomio interpolador, $p(x)$, nos devuelva un vector con dichas evaluaciones, esto es $p(z)$. Para ello, una vez obtenidas las diferencias divididas (¡y guardadas adecuadamente!), podemos ir evaluando paso a paso el polinomio interpolador en su forma de Newton (6.5) aplicando las ideas del algoritmo de Horner-Ruffini (véase el ejercicio §1.11), de la siguiente manera:

- Tomamos $b_n = f[x_0, x_1, \dots, x_n]$
- Para $k = (n-1), (n-2), \dots, 0$ construimos $b_k = b_{k+1}(z - x_k) + f[x_0, x_1, \dots, x_k]$.

obteniendo $p(z) = b_0$.

Asegúrate de la veracidad de este algoritmo y usa estas ideas para crear un código similar a `difdiv` que incluya z como argumento y devuelva $p(z)$ directamente, sin calcular explícitamente p .

§6.59. Adapta el código `difdiv` (dado en la página 91 para calcular el polinomio interpolador de Newton) de forma que permita realizar la interpolación de Hermite con la siguiente tabla de datos.

x	1	2	4
y	10	20	40
y'	10	20	40

§6.60. Adapta el código del *spline* cuadrático de clase 1 para calcular el *spline* lineal de clase cero que interpola datos (x_i, y_i) , $i = 0, \dots, n$.

§6.61. Aplica el *script* que dió lugar a la figura 6.5, pero en este caso usa un número impar de nodos. ¿Qué ocurre? Si aumentamos el número (impar) de nodos, ¿mejora la interpolación? Modifica de nuevo el *script* para aplicarlo al caso de un número impar de nodos y en el que conozcamos la derivada en $x = 0$ en lugar de $x = -1$. Observa que quizás debamos hacer *scripts* diferentes dependiendo de si el número de nodos es par o impar. ¿Ha mejorado el resultado con respecto al *script* original?

§6.62. Define una función `function` que permita calcular el *spline* cuadrático de clase uno, suponiendo que el dato extra dado es el valor de la derivada en cualquier nodo.

§6.63. Teniendo en cuenta que la expresión (6.14) para el *spline* cúbico puede ser reescrita como

$$s_i(x) = (y_{i-1} - d_i x_{i-1} + z_i x_{i-1}^2 - t_i x_i x_{i-1}^2) + (d_{i-1} - 2z_i x_{i-1} + t_i x_{i-1}^2 + 2t_i x_i x_{i-1})x \\ + (z_i - t_i x_i - 2t_i x_{i-1})x^2 + t_i x^3, \quad \text{para } i = 1, \dots, n,$$

escribe en forma vectorizada (análogamente a como se hizo con el *spline* cuadrático) la construcción del *spline* $s(x)$ como una matriz $n \times 4$ para no tener que usar el comando `for`.

Compara, usando el comando `tic-toc`, el tiempo de cómputo empleado con el comando vectorizado y sin vectorizar.

§6.64. Para calcular un ***spline* cúbico sujeto** que interpole los datos $x = (x_0, x_1, \dots, x_n)$ e $y = (y_0, y_1, \dots, y_n)$ se ha de resolver el sistema de ecuaciones lineales (6.15). Define una función que admita como argumentos los vectores x e y , y los valores y'_0, y'_n y que devuelva la solución del sistema de ecuaciones anterior. Comprueba su funcionamiento para los datos $x = (1, 2, 4, 5, 7, 8)$, $y = (1, 4, 7, 4, 7, 2)$, $y'_0 = y'_5 = 0$.

§6.65. Para calcular un ***spline* cúbico natural** que interpole los datos $x = (x_0, x_1, \dots, x_n)$ e $y = (y_0, y_1, \dots, y_n)$, y que cumple las condiciones extra $s''(x_0) = s''(x_n) = 0$, se ha de resolver el sistema de ecuaciones lineales

$$2d_0 + d_1 = 3w_1, \\ \frac{1}{h_i} d_{i-1} + 2 \left(\frac{1}{h_i} + \frac{1}{h_{i+1}} \right) d_i + \frac{1}{h_{i+1}} d_{i+1} = 3 \left(\frac{w_i}{h_i} + \frac{w_{i+1}}{h_{i+1}} \right), \quad i = 1, \dots, n-1, \\ d_{n-1} + 2d_n = 3w_n,$$

donde $h_i = x_i - x_{i-1}$, $w_i = \frac{y_i - y_{i-1}}{h_i}$ para $i = 1, \dots, n$. Los valores d_i para $i = 0, \dots, n$ son las incógnitas. Define una función que admita como argumentos los vectores x e y y que devuelva la solución del sistema de ecuaciones anterior. Comprueba su funcionamiento para los datos $x = (1, 2, 4, 5, 7, 8)$ y $y = (1, 4, 7, 4, 7, 2)$.

§6.66. Propón un *script* que permita calcular el ***spline* cúbico periódico**, y aplícalo a los datos $x = (1, 2, 4, 5, 7, 8, 10)$ y $y = (1, 4, 7, 4, 7, 2, 1)$.

Capítulo 7: mínimos cuadrados

§7.67. A partir de lo visto para la recta y la parábola, determinar cuál es el sistema necesario (y, por tanto, la tabla necesaria) para calcular el polinomio de grado menor o igual que m que mejor aproxima, por mínimos cuadrados discreto, un conjunto de $n + 1$ datos, siendo $m \leq n$.

§7.68. Determinar cómo se construye el sistema de ecuaciones, usando la regla de las matrices, que permite calcular el polinomio de grado m que mejor aproxima, por mínimos cuadrados discreto, un conjunto de $n + 1$ datos, con $m \leq n$.

§7.69. Calcula el polinomio de grado menor o igual que 3 que mejor aproxima (en el sentido de los mínimos cuadrados) la tabla de datos

x	1	2	4	8	16
y	10	20	40	50	100

Corroborra tu resultado con el resultado dado por `polyfit` y representa gráficamente el polinomio y los datos de la tabla.

§7.70. Calcula y representa el polinomio de grado menor o igual que 4 que mejor aproxima (en el sentido de los mínimos cuadrados discreto) a la función $f(x) = e^{-x^2}$ en una partición del intervalo $[-1, 1]$ con 1000 nodos equidistantes.

§7.71. Calcula la mejor aproximación por mínimos cuadrados discreta de los datos de la siguiente tabla

x	10	20	40	50	100
y	1	2	4	8	16

mediante una función de la forma $p(t) = a_2 \sin(t) + a_1 \cos(t) + a_0$. Lo primero que hay que observar en este caso es que el sistema a resolver no es el mismo que en el caso de que $p(t)$ sea un polinomio, y en consecuencia habría que determinar el sistema de ecuaciones lineales a resolver, de forma análoga a como se ha hecho en la sección 7.2.1. Con estas indicaciones, determina el SEL a resolver y calcula $p(t)$.

§7.72. Cuando se pretende calcular el polinomio de grado menor o igual a $n - 1$ que mejor aproxima, mediante mínimos cuadrados continuo, una función cualquiera $f(x)$ en el intervalo $[0, 1]$, se puede comprobar que la matriz de Gramm obtenida tiene como coeficientes $a_{i,j} = \frac{1}{i+j-1}$, donde $i, j = 1, \dots, n$. Esta matriz, que se conoce como matriz de Hilbert, se puede definir de manera muy sencilla en Octave mediante el comando `hilb`. Comprueba este hecho a partir de los códigos mostrados en este capítulo para $n = 10$. No obstante, el sistema de ecuaciones lineales que habría que resolver para realizar dicho ajuste tiene un serio problema numérico: estas matrices están muy mal condicionadas, por lo que su resolución numérica está sujeta a muchísima incertidumbre. Asegúrate de esto aplicando el comando `cond` a la matriz anterior.

§7.73. Repite la aproximación por mínimos cuadrados continua de la función seno (hecha en la sección 7.2.2) pero empleando un polinomio de grado diez. Observa que aparece un mensaje de advertencia. ¿Sabes a qué es debido?

§7.74. Repite la aproximación por mínimos cuadrados continua de la función seno, pero usando como medida el siguiente producto escalar con peso,

$$\langle g, h \rangle = \int_{-1}^1 \frac{g(x)h(x)}{\sqrt{1-x^2}} dx.$$

§7.75. Modifica el *script* de la página 114 para ajustar, mediante mínimos cuadrados continuo, la función $f(x) = 3x + 5x^2$, en el intervalo $[0, 3]$, mediante una expresión del tipo $c_1 + c_2 \exp(x) + c_3 \exp(-x)$.

Capítulo 8: derivación y cuadratura numérica

§8.76. Al medir f en una serie de puntos x_i , se ha obtenido la siguiente tabla de valores.

x_i	0	1	2	3
f_i	0	1	8	27

- Aproxima de distintas formas la derivada de f en $x = 2$.
- Aproxima la derivada de f en $x = 2.5$.
- Aproxima la integral de f en $[0, 3]$.

§8.77. Imponiendo exactitud, obtén de nuevo las fórmulas progresiva y regresiva de derivación numérica.

§8.78. Usando los comandos `flip` y `diff`, crea una función para calcular diferencias regresivas.

§8.79. En este ejercicio pretendemos calcular una **fórmula centrada para la segunda derivada** de una función $f(x)$ en el punto $x = a$ a partir de los valores $f(a-h)$, $f(a)$ y $f(a+h)$, donde $h \neq 0$. Para ello:

- a partir del polinomio de interpolación, justifica que la fórmula buscada es

$$f''(a) \approx \frac{f(a-h) - 2f(a) + f(a+h)}{h^2}.$$

- Obtén la misma fórmula imponiendo exactitud y determina el grado de exactitud de la misma.

§8.80. Construye de nuevo la función `derinum.m` (de la página 123) programando las diferencias finitas, que aparecen en las líneas 6, usando directamente el comando `diff`. ¿Podrías hacer algo parecido con las líneas 6 y 7 de `derinum2.m`?

§8.81. Repite los cálculos realizados en el ejemplo visto en la página 124 sobre la inestabilidad en la derivación numérica para calcular la derivada del seno, del coseno y de la función exponencial en los puntos $x = 0$ y en $x = \pi/2$. ¿Qué conclusión sacas de los resultados obtenidos?

§8.82. Relaciona los resultados obtenidos al aproximar la derivada de la función arco-tangente (véase página 124) con el comando `eps(sqrt(2))`.

- §8.83. Imponiendo exactitud, deduce de nuevo la fórmula del rectángulo usando un punto cualquiera del intervalo, la fórmula del trapecio y la de Simpson.
- §8.84. Al crear la función `simpson.m` en la página 129, observamos en la línea 8 que duplicamos algunas de las evaluaciones de `f`, elevando el coste computacional. Modifica el código para no duplicar ninguna evaluación.
- §8.85. Si definimos por $\bar{f} = \frac{1}{b-a} \int_a^b f(x)dx$ el valor medio, en el intervalo $[a, b]$, de la función continua f , podemos demostrar que, tomando valores aleatorios x_i uniformemente distribuidos en el intervalo $[a, b]$, se verifica la siguiente propiedad de convergencia:

$$\lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(x_i) = \bar{f}.$$

Corroborar numéricamente esta propiedad para la función $f(x) = e^{-x^2}$ en el intervalo $x \in [0, 1]$, comparando los resultados obtenidos con el proporcionado por el comando `quad`. Constata que esta idea puede servir para aproximar el valor de una integral $\int_a^b f(x)dx$ (**método de Montecarlo**) aunque su integral no sea calculable de forma analítica.

Capítulo 9: ecuaciones diferenciales I: PVI

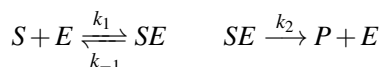
- §9.86. A partir del código mostrado para la implementación del algoritmo de Runge-Kutta de orden 4, obtén un código que implemente el algoritmo de Euler Explícito.
- §9.87. A partir del código desarrollado en el ejemplo 9.9, trata de resolver el ejercicio §2.25 sin usar la solución explícita que allí se da.
- §9.88. Implementa un código para el algoritmo explícito de Adam-Bashfort

$$y_{n+2} - y_{n+1} = \frac{h}{2} (-f(y_n, t_n) + 3f(y_{n+1}, t_{n+1})).$$

Para ello, nota que para arrancar la iteración y calcular y_2 , necesitas no sólo el dato inicial y_0 sino también la primera aproximación y_1 , que debes calcular mediante otro método (de un paso). Usa Runge-Kutta de orden 4 para dar este paso inicial.

- §9.89. Para describir el crecimiento de una bacteria mediante un modelo matemático, se supone que el paso de nutrientes al interior de la misma está regulado por un sistema de receptores enzimáticos localizados en su membrana (**Michaelis-Menten** 1913), de modo que el nutriente externo es capturado por uno de los receptores que hay sobre la membrana bacteriana, y entonces es, bien introducido como alimento en la bacteria por un mecanismo de transporte, o bien devuelto al exterior, dejando de

nuevo libre al receptor. Describimos este esquema molecular mediante las leyes de acción de masa siguientes,



donde S son las moléculas de nutriente externo, E los receptores enzimáticos no ocupados, SE los receptores ocupados y P el producto o nutriente que ha pasado al interior dejando de nuevo libre la molécula de enzima. Aplicando la ley de acción de masas a esta reacción (y haciendo ciertas simplificaciones⁴) obtenemos el siguiente sistema de dos ecuaciones diferenciales no lineales,

$$\left. \begin{aligned} \frac{ds}{dt} &= -k_1 e_0 s + (k_1 s + k_{-1})c \\ \frac{dc}{dt} &= k_1 e_0 s - (k_1 s + k_{-1} + k_2)c \end{aligned} \right\}$$

y con condiciones iniciales $s(0) = s_0$ dado y $c(0) = 0$. En este sistema, $s = [S]$ y $c = [SE]$ son concentraciones de producto, mientras que $e_0 > 0$ es la concentración inicial de enzimas. Por otra parte, la concentración de nutriente $p = [P]$ queda determinada *a posteriori* por la expresión integral

$$p(t) = k_2 \int_0^t c(s) ds.$$

Dándole valores (cualesquiera) a los ratios k_i de las reacciones y a e_0 , resuelve el sistema para s_0 arbitrario y verifica, mediante fórmulas de cuadratura numérica, que $p(t) \rightarrow s_0$ cuando $t \rightarrow \infty$, es decir, con el paso del tiempo la bacteria se come todo el nutriente que hay.

Capítulo 10: ecuaciones diferenciales II: PVF

§10.90. Usa el método de diferencias finitas descrito en la sección 10.2.1 para resolver el problema de valores en la frontera

$$\left. \begin{aligned} x''(t) &= (t^2 - 1)x'(t) + \frac{1}{t}x(t), \quad t \in]1, 2[\\ x(1) &= 7 \\ x(2) &= 9 \end{aligned} \right\},$$

con $N = 20$ nodos intermedios.

§10.91. En el ejemplo 5.2 se presenta un problema que se deduce a partir del PVF:

$$\left. \begin{aligned} u''(x) &= \lambda u(x), \quad x \in [0, 1] \\ u(0) &= 0 \\ u(1) &= 0 \end{aligned} \right\},$$

⁴Básicamente, se reescribe $e(t) = e_0 - c(t)$

donde λ es un valor real. Justifica el problema de valores propios que allí se resolvió empleando para ello la técnica de discretización mediante diferencias finitas.

§10.92. Se considera el problema de aproximación numérica de un PVF con **condiciones periódicas**:

$$\left. \begin{aligned} x''(t) - x(t) &= f(t), & t \in [0, 1], \\ x(0) &= x(1) \\ x'(0) &= x'(1) \end{aligned} \right\}.$$

Demuestra que, usando las aproximaciones

$$x'(0) \approx \frac{y_0 - y_{-1}}{h}, \quad x'(1) \approx \frac{y_{N+1} - y_N}{h}$$

para las condiciones de frontera, el cálculo de la solución numérica se reduce a resolver el sistema lineal $(N+1) \times (N+1)$

$$\frac{1}{h^2} \begin{pmatrix} a & 1 & 0 & \cdots & 0 & 1 \\ 1 & a & 1 & 0 & \cdots & 0 \\ 0 & 1 & a & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & a & 1 \\ 1 & 0 & \cdots & 0 & 1 & a \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}$$

donde $a = -2 - h^2$ y $f_n = f(nh)$, con $h = \frac{1}{N+1}$. Las incógnitas son $y_n \approx u(x_n)$, para $n = 0, 1, \dots, N$.

Resuelve numéricamente el problema anterior para $f(x) = \cos(\pi x)$ y compara gráficamente el resultado con la solución exacta $u(x) = \frac{e^{1-x} - e^x - (-1+e)\cos(\pi x)}{(e-1)(1+\pi^2)}$.

§10.93. Analiza la aplicación del método en diferencias finitas al siguiente PVF:

$$x''(t) = f(t), \quad x'(0) = \sigma, \quad x(1) = \beta, \quad t \in [0, 1],$$

empleando diferencias centradas tanto para la segunda derivada como para la primera, que aparece en las condiciones de frontera.

Bibliografía

- [1] O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, 1994.
- [2] G. Borrel, *Introducción a Matlab y Octave*, <http://iimyo.forja.rediris.es/>
- [3] E.A. Coddington, N. Levinson, *Theory of ordinary defferential equations*, Mac Graw-Hill, 1985.
- [4] J.W. Demmel, *Applied Numerical Linear Algebra*, SIAM Philadelphia, 1997.
- [5] J.W. Eaton, D. Bateman, S. Hauberg, R. Wehbring, *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations*, 2015. URL <http://www.gnu.org/software/octave/doc/interpreter/>
- [6] M. Gasca, *Cálculo numérico: Resolución de ecuaciones y sistemas*, Mira editores, Zaragoza, 1999.
- [7] W. Gautschi, *Numerical analysis*, Birkhäuser-Boston, 2012.
- [8] GNU Octave, página principal del proyecto, <http://www.octave.org>.
- [9] G.H. Golub and C.F. Van Loan, *Matrix computations*, third edition, Johns Hopkins University Press, Baltimore, 1996.
- [10] S.I. Grossman, *Álgebra Lineal con aplicaciones*, cuarta edición, Mc. Graw-Hill, México, 1992.
- [11] S. Gutiérrez. *Álgebra lineal para la Economía*, Thomson-Editorial A.C., Madrid, 1986.
- [12] A.C. Hindmarsh, *ODEPACK, A systematized collection of ODE solvers*, in Scientific Computing, R.S. Stepleman *et al.* (eds.), North-Holland, Amsterdam, 1983.
- [13] M. Hirsch, S. Smale, *Ecuaciones diferenciales, sistemas dinámicos y álgebra lineal*, Alianza Editorial, Madrid, 1983.
- [14] H.B. Keller, *Numerical Methods For Two-Point Boundary-Value Problems*, Dover, New York, 1992.

- [15] D. Kincaid and W. Cheney, *Análisis numérico. Las matemáticas del cálculo científico*, Editorial Addison-Wesley Iberoamericana, 1994.
- [16] R. Kress, *Numerical Analysis*, Graduate Texts in Mathematics, vol. 181, Springer-Verlag New York, 1998.
- [17] P.H. Leslie, *On the use of matrices in certain population mathematics*, *Biometrika*, **33**(3), 1945, 183–212.
- [18] R.J. Leveque, *Finite Difference Methods for Ordinary and Partial Differential Equations*, SIAM, Philadelphia, 2007.
- [19] P.J.G. Long, *Introduction to Octave*.
<http://www-mdp.eng.cam.ac.uk/web/CD/engapps/octave/octavetut.pdf>.
- [20] L. Merino, E. Santos, *Álgebra lineal con métodos elementales*, Ed. Thomson-Paraninfo, Madrid, 2006.
- [21] Octave-Forge, repositorio de paquetes externos,
<http://octave.sourceforge.net/index.html>
- [22] Wikis donde se analizan las diferencias entre los lenguajes de Octave y MatLab.
https://en.wikibooks.org/wiki/MATLAB_Programming/
http://wiki.octave.org/FAQ#Porting_programs_from_Matlab_to_Octave
- [23] C.M. Pease, D.J. Mattson, *Demography of the Yellowstone Grizzly Bears*, *Ecology*, **80**(3), 1999, 957–975.
- [24] A. Quarteroni, R. Sacco, F. Saleri, *Numerical Mathematics*, Springer-Verlag, New-York, 2000.
- [25] A. Quarteroni, F. Saleri, *Cálculo científico con MATLAB y Octave*, Springer-Verlag Italia, Milano, 2006.
D.O.I.: 10.1007/978-88-470-0504-4
- [26] E.D. Rainville, P.E. Bedient, R.E. Bedient, *Ecuaciones Diferenciales*, Prentice-Hall, México, 1998.
- [27] J.M. Sanz-Serna, *Diez lecciones de cálculo numérico (Segunda edición)*, Universidad de Valladolid (Secretariado de Publicaciones e Intercambio Editorial), 2010.
- [28] LL.N. Trefethen, D. Bau, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997.
- [29] D.G. Zill, *Ecuaciones diferenciales con aplicaciones de modelado*, sexta edición, Thomson, 1997.

Índice alfabético

[abs](#), 9, 23, 40, 43, 44, 53, 76

[all](#), 18

animación, 143

[ans](#), 9, 17, 19

[any](#), 18

[arg](#), 9

array, 25

[asin](#), [acos](#), [atan](#), 9, 124

áurea, razón, 22

[axis](#), 75, 113, 142, 143, 153

[break](#), 24, 38, 40

bucles, 21

cadena de caracteres, 7, 14, 36, 39, 75, 144

[ceil](#), 10, 38

[clear](#), 8, 16, 153, 155

[clf](#), 135, 143, 153

comentario, 15

[compan](#), 11, 165

[cond](#), 51, 169

condicional, sentencia, 18

condicionamiento de una matriz, 50

condiciones, 17

[conj](#), 9

[continue](#), 24

[conv](#), 7

[cos](#), 9, 123, 148

[cot](#), [csc](#), [sec](#), 9

[cputime](#), 158

[cumprod](#), 10

[cumsum](#), 10, 127

[cumtrapz](#), 127

[dblquad](#), 128

[deal](#), 8, 23, 48

[deconv](#), 7

[det](#), 11

d'Hont, Ley electoral, 21

[diag](#), 5, 73, 77, 99, 162

[diary](#), 12

[diff](#), 97, 123, 170

directorio de trabajo, 12, 15, 143

división izquierda “\”, 10, 51, 107

[do-until](#), 23

[eig](#), 11, 73

[end](#), 6, 91, 122, 127, 139, 153

[eps](#), 27

[error](#), 20, 55, 158, 160, 161

estadística, 10

[exit](#), 1

[exp](#), 9, 14

[expm](#), 74

[eye](#), 5, 55, 73

factorización

de Cholesky, 49

LU, 49, 55

QR, 49, 52, 72

[feval](#), 14, 36, 37, 41, 128, 141

[ffmpeg](#), 144

[find](#), 18

[fix](#), 10

[flip](#), 170

[fliplr](#), 11

[flipud](#), 11

[floor](#), 10

[for](#), 21, 25, 38, 52

- [format](#), 27, 44, 124
- fórmula de
 - rectángulos, 118, 121, 128
 - Simpson, 119, 121, 128
 - trapecios, 118, 121, 129
- [fplot](#), 35
- [fprintf](#), 38, 39, 74, 76, 77
- función, 13
 - valor por defecto, 24, 129
 - anónima, 14, 26, 36, 41, 44
 - function, 14, 26, 39–41
 - inline, 26
 - predefinida, 9
- [function](#), 14, 24, 35
- [fzero](#), 36
- [global](#), 24, 143, 153
- [grid](#), 36
- guardar
 - animación, 143
 - funciones, 13
 - gráficas, 13
 - sesiones, 12
 - variables, 12
- [help](#), 10, 15, 41
- [hilb](#), 169
- hipoteca, 158
- [hold](#), 96, 99, 136, 142, 143, 153
- [if](#), 18, 38, 43, 55, 76
- [imag](#), 9
- [Inf](#), 27, 126
- [inline](#), 26
- interpolación
 - de Hermite, 85, 167
 - Lagrangiana, 80, 89
 - spline, 88, 94
- [inv](#), 11, 73
- [legend](#), 113, 136
- [length](#), 4, 5, 22, 63, 64, 91, 96
- [linspace](#), 4, 26, 74, 92, 113, 122, 128, 135, 137, 139, 141, 148, 151
- [load](#), 13
- [log](#), 38, 151
- [ls](#), 1
- [lsode](#), 137
- MatLab, 26
- matriz, 4
 - de Hilbert, 50, 169
 - de permutación, 48, 51, 162
 - de Vandermonde, 81, 105, 118
 - definida positiva, 49, 52, 59, 60
 - dispersa, 57, 148
 - estrictamente diagonal dominante, 59, 165
 - ortogonal, 48, 52, 70, 72
 - simétrica, 48, 59, 60, 70
 - triangular, 46, 48
 - tridiagonal, 55
- [max](#), 11, 22, 53, 74
- [mean](#), 10
- [median](#), 10
- método
 - Adams, 134, 138
 - BDF, 134, 138
 - de bisección, 30, 37
 - de Crank-Nicolson, 134
 - de descenso rápido, 60
 - de diferencias finitas, 147
 - de Euler explícito, 133, 135
 - de Euler implícito, 134
 - de Euler mejorado, 134
 - de Gauss, 46, 47, 160
 - de Gauss-Seidel, 59, 62
 - de gradiente conjugado, 61, 64
 - de gradiente conjugado mejorado, 164
 - de Horner-Ruffini, 7, 85, 156
 - de Jacobi, 58, 163
 - de las potencias, 71, 75
 - de Montecarlo, 171
 - de Newton-Raphson, 34, 43
 - de regula-falsi, 32
 - de relajación, 59
 - de Runge-Kutta, 141, 152

- de secante, 32, 42, 118, 153
- de sustitución progresiva, 46, 53
- de sustitución regresiva, 46, 52
- de Taylor de orden 2, 134
- de Thomas, 163
- de tiro, 153
- QR-valores propios, 72, 76
- [min](#), 11
- [mldivide](#), 11, 51, 107
- [mode](#), 10
- número, 3
 - complejo, 3, 155
 - de Euler, 9
 - de punto flotante, 27
 - entero, 10
 - pi, 9, 19, 126
 - real, 3, 27
- [NaN](#), 160
- [norm](#), 63, 64, 73, 76, 77
- [num2str](#), 144
- OdePkg, 137, 147
- [ones](#), 5, 17, 73, 148
- operadores lógicos, 17
- permutar filas, 6, 54
- pivoteo, 47, 53, 160
- [plot](#), 13, 35, 37, 74, 91, 93, 94
- polinomios, 7
- [poly](#), 7, 73, 91, 96, 99
- [polyderiv](#), 7
- [polyfit](#), 92, 111
- [polyint](#), 7
- [polyout](#), 7, 90, 91
- [polyval](#), 7, 85, 91, 96, 111
- potencias truncadas, 95
- [prod](#), 10
- [pwd](#), 1, 12
- [qr](#), 52
- [quad](#), 14, 125
- [quadcc](#), [quadl](#), [quadgk](#), 126
- [quadv](#), 113, 114, 126
- radio espectral, 58, 69, 165
- [rand](#), 5, 9, 23, 76
- rango, 4, 6, 135
- [real](#), 9
- [realmax](#), 27
- [realmin](#), 27
- [rem](#), 10, 17
- [reshape](#), 11, 22
- [roots](#), 7, 74
- [round](#), 10
- [rref](#), 51
- Runge, fenómeno de, 88, 92, 99, 119
- [save](#), 13
- [saveas](#), 13, 143
- script*, 13, 16, 38
- [sec](#), [csc](#), [cot](#), 9
- [semilogx](#), 139
- [sin](#), 9, 113, 122, 155
- sistemas triangulares, 45, 52
- [size](#), 5, 52, 162
- [sort](#), 11, 157
- [spdiags](#), 149, 150
- [sprintf](#), 75
- [sqrt](#), 9, 23, 126
- [std](#), 10
- [strcat](#), 143
- string*, 7
- [subplot](#), 139
- [sum](#), 10, 22, 127
- [switch](#), 20
- symbolic, 115
- [tan](#), 9, 143
- tic-toc*, 24, 25
- [toeplitz](#), 73
- [transpose](#), 11
- [traspuesta-conjugada](#) ['], 11
- [trapz](#), 127, 129
- [tril](#), 6, 55, 77, 161
- [triplequad](#), 128
- [triu](#), 6, 55
- [vander](#), 90, 110

variable, 8

 global, 16, 24

 local, 16, 24

[vec](#), 11, 22

vector, 4

[warning](#), 20, 43, 44

[while](#), 22, 23, 43, 44, 63, 64

[who](#), 8

[xlabel](#), 74

[ylabel](#), 112

[zeros](#), 5, 52, 55, 74, 96, 113

Lista de programas

Programas disponibles en: <http://www.ugr.es/local/jjmnieto/MNBOctave.html>

1.1	nif.m : Programa para calcular la tetra del NIF	17
1.2	diasmes.m : Función que indica los días de cada mes del año	20
1.3	dhont.m : Reparto proporcional mediante la Ley d'Hont	22
2.4	<i>Script</i> del método de bisección	38
2.5	biseccion.m : Método de bisección	40
2.6	secante.m : Método de la secante	43
2.7	newton.m : Método de Newton-Raphson	44
3.8	sustreg.m : Método de sustituciones regresivas	52
3.9	sustprogr.m : Método de sustituciones progresivas	53
3.10	gausspiv.m : Método de Gauss con estrategia de pivote parcial	53
3.11	<i>Script</i> de factorización LU (Doolittle)	55
4.12	gaussseidel.m : Método de Gauss-Seidel	63
4.13	gradconj.m : Método del gradiente conjugado	64
5.14	<i>Script</i> deformación de una viga vertical	74
5.15	potencias.m : Método de las potencias	76
5.16	qrvp.m : Método QR para el cálculo de valores propios	77
6.17	difdiv.m : Cálculo del polinomio interpolador con diferencias divididas . . .	91
6.18	<i>Script</i> para el fenómeno de Runge (limitaciones de la interpolación)	93
6.19	<i>Script</i> para interpolar con <i>spline</i> cuadrático	96
6.20	plotspline.m : Representa gráficamente funciones <i>spline</i>	96
7.21	<i>Script</i> del ajuste polinómico por mínimos cuadrados continuo	113
7.22	<i>Script</i> del ajuste trigonométrico por mínimos cuadrados continuo	114
8.23	derinum.m : Derivada numérica de una función con diferencias centradas .	123
8.24	derinum2.m : Derivada segunda numérica mediante diferencias centradas .	124
8.25	rectang.m : Fórmula compuesta de rectángulos	129
8.26	simpson.m : Fórmula compuesta de Simpson	129
8.27	trapecios.m : Fórmula compuesta de trapecios	129
9.28	rk4solver.m : Algoritmo de Runge-Kutta de orden 4	141
9.29	<i>Script</i> que simula, mediante Runge-Kutta, la trayectoria de una bola con distintas inclinaciones iniciales	142

9.30	<i>Script</i> que genera múltiples gráficos con distintas aproximaciones de las posiciones de una bola lanzada.	143
10.31	<i>Script</i> con la aproximación del PVF de la ecuación de Poisson	148
10.32	<i>Script</i> con diferencias finitas para resolver PVFs lineales	151
10.33	<i>Script</i> que aproxima, mediante un método de tiro, la trayectoria de una bola para que alcance un punto prescrito	153