

Introducción a C *y a métodos numéricos*

Héctor Manuel
Mora Escobar

colección **textos**



UNIVERSIDAD
NACIONAL
DE COLOMBIA
Sede Bogotá

Héctor Manuel Mora Escobar

es Matemático e Ingeniero Civil de la Universidad Nacional de Colombia. En la Université de Nancy en Francia obtuvo el DEA, Diplôme d'Études Approfondies, y el Doctorat de 3^{ème} Cycle en Matemáticas Aplicadas. Desde 1975 es docente de la Universidad Nacional de Colombia, actualmente se desempeña como Profesor Titular del Departamento de Matemáticas. Ha publicado varios libros y artículos

sobre optimización, métodos numéricos y prospectiva. Es miembro de las siguientes sociedades: Sociedad Colombiana de Matemáticas, Society for Industrial and Applied Mathematics, Société de Mathématiques Appliquées et Industrielles, Mathematical Programming Society y Sociedad Colombiana de Investigación de Operaciones.





UNIVERSIDAD
NACIONAL
DE COLOMBIA
Sede Bogotá

colección textos

HÉCTOR MANUEL MORA ESCOBAR

es Matemático e Ingeniero Civil de la Universidad Nacional de Colombia. En la Université de Nancy en Francia obtuvo el DEA, Diplôme d'Études Approfondies, y el Doctorat de 3ème Cycle en Matemáticas Aplicadas.

Desde 1975 es docente de la Universidad Nacional de Colombia, actualmente se desempeña como Profesor Titular del Departamento de Matemáticas. Ha publicado varios libros y artículos sobre optimización, métodos numéricos y prospectiva. Es miembro de las siguientes sociedades: Sociedad Colombiana de Matemáticas, Society for Industrial and Applied Mathematics, Société de Mathématiques Appliquées et Industrielles, Mathematical Programming Society y Sociedad Colombiana de Investigación de Operaciones.

Introducción a C y a métodos numéricos

Héctor Manuel Mora Escobar

Profesor del Departamento de Matemáticas
Facultad de Ciencias, Universidad Nacional de Colombia

Introducción a C y a métodos numéricos

Universidad Nacional de Colombia

FACULTAD DE CIENCIAS
BOGOTÁ

© Universidad Nacional de Colombia
Facultad de Ciencias
Departamento de Matemáticas y Estadística
© Héctor Manuel Mora Escobar

Primera edición, 2004
Bogotá, Colombia, 2004

UNIBIBLOS

Director general
Ramón Fayad Naffah

Coordinación editorial
Dora Inés Perilla Castillo

Revisión editorial
Óscar Torres

Preparación editorial e impresión
Universidad Nacional de Colombia, Unibiblos
dirunibiblo_bog@unal.edu.co

Carátula
Camilo Umaña

ISBN 958-701-363-8
ISBN 958-701-138-4
(obra completa)

Catalogación en la publicación Universidad Nacional de Colombia
Mora Escobar, Héctor Manuel, 1953-

Introducción a C y a métodos numéricos / Héctor Manuel Mora Escobar —
Bogotá : Universidad Nacional de Colombia. Facultad de Ciencias, 2004
408 p.

ISBN : 958-701-363-8

1. Análisis numérico 2.C (Lenguaje de programación para computadores)
3. Matemáticas aplicadas

CDD-21 519.4 / M827in / 2004

A Hélène, Nicolás y Sylvie

Índice general

Prólogo	ix
1. Introducción	1
2. Generalidades	5
2.1. El primer programa	6
2.2. Editar, compilar, ejecutar	9
2.2.1. g++	10
2.2.2. bcc32	10
2.2.3. Turbo C++	11
2.2.4. Visual C++	12
2.2.5. Dev-C++	14
2.3. Comentarios	15
2.4. Identificadores	16
2.5. Tipos de datos	19
2.6. Operador de asignación	21
2.7. Operadores aritméticos	22
2.8. Prioridad de los operadores aritméticos	24
2.9. Abreviaciones usuales	26
2.10. Funciones matemáticas	27
2.11. Entrada de datos y salida de resultados	30
2.12. Conversiones de tipo en expresiones mixtas	35
2.13. Moldes	36

3. Estructuras de control	41
3.1. <code>if</code>	41
3.2. Operadores relacionales y lógicos	45
3.3. <code>for</code>	48
3.4. <code>while</code>	51
3.5. <code>do while</code>	54
3.6. <code>switch</code>	57
3.7. <code>break</code>	58
3.8. <code>continue</code>	60
3.9. <code>goto</code> y <code>exit</code>	61
4. Funciones	67
4.1. Generalidades	67
4.2. Funciones recurrentes	75
4.3. Parámetros por valor y por referencia	76
4.4. Parámetros por defecto	81
4.5. Variables locales y variables globales	82
4.6. Sobrecarga de funciones	83
4.7. Biblioteca estándar	85
5. Arreglos	87
5.1. Arreglos unidimensionales	87
5.2. Arreglos multidimensionales	94
5.3. Cadenas	98
5.4. Inicialización de arreglos	103
6. Apuntadores	109
6.1. Apuntadores y arreglos unidimensionales	111
6.2. Apuntadores a apuntadores	116
6.3. Apuntadores y arreglos bidimensionales	116
6.4. Matrices y arreglos unidimensionales	117
6.5. Arreglos aleatorios	129
6.6. Asignación dinámica de memoria	131

ÍNDICE GENERAL

6.7. Matrices y apuntadores dobles	135
6.8. Arreglos a partir de 1	139
7. Lectura y escritura en archivos	143
7.1. <code>fopen</code> , <code>fscanf</code> , <code>fclose</code> , <code>fprintf</code>	144
7.2. <code>feof</code>	148
7.3. Algunos ejemplos	150
8. Temas varios	157
8.1. <code>sizeof</code>	157
8.2. <code>const</code>	159
8.3. <code>typedef</code>	159
8.4. <code>include</code>	161
8.5. <code>define</code>	164
8.6. Apuntadores a funciones	165
8.7. Funciones en línea	169
8.8. Argumentos de la función <code>main</code>	171
9. Estructuras	175
9.1. Un ejemplo con complejos	175
9.2. Un ejemplo típico	181
10. Algunas funciones elementales	183
10.1. Código de algunas funciones	184
10.2. Versiones con saltos	190
10.3. Método burbuja	194
11. Solución de sistemas lineales	199
11.1. Notación	199
11.2. Métodos ingenuos	200
11.3. Sistema diagonal	202
11.4. Sistema triangular superior	202
11.4.1. Número de operaciones	203
11.4.2. Implementación en C	204

ÍNDICE GENERAL

11.5. Sistema triangular inferior	206
11.6. Método de Gauss	206
11.6.1. Número de operaciones	211
11.6.2. Implementación en C	213
11.7. Factorización LU	215
11.8. Método de Gauss con pivoteo parcial	217
11.9. Factorización $LU=PA$	223
11.10. Método de Cholesky	227
11.10.1. Matrices definidas positivas	227
11.10.2. Factorización de Cholesky	229
11.10.3. Número de operaciones de la factorización	235
11.10.4. Solución del sistema	237
11.11. Método de Gauss-Seidel	239
11.12. Solución por mínimos cuadrados	245
11.12.1. Derivadas parciales	246
11.12.2. Ecuaciones normales	247
12. Solución de ecuaciones	255
12.1. Método de Newton	257
12.1.1. Orden de convergencia	262
12.2. Método de la secante	263
12.3. Método de la bisección	267
12.4. Método de Regula Falsi	268
12.5. Modificación del método de Regula Falsi	270
12.6. Método de punto fijo	272
12.6.1. Método de punto fijo y método de Newton	278
12.7. Método de Newton en \mathbb{R}^n	279
12.7.1. Matriz jacobiana	280
12.7.2. Fórmula de Newton en \mathbb{R}^n	280
13. Interpolación y aproximación	285
13.1. Interpolación	287
13.2. Interpolación de Lagrange	289

ÍNDICE GENERAL

13.2.1. Algunos resultados previos	289
13.2.2. Polinomios de Lagrange	290
13.2.3. Existencia, unicidad y error	291
13.3. Diferencias divididas de Newton	293
13.3.1. Tabla de diferencias divididas	296
13.3.2. Cálculo del valor interpolado	298
13.4. Diferencias finitas	302
13.4.1. Tabla de diferencias finitas	302
13.4.2. Cálculo del valor interpolado	304
13.5. Aproximación por mínimos cuadrados	306
14. Integración y diferenciación	313
14.1. Integración numérica	313
14.2. Fórmula del trapecio	314
14.2.1. Errores local y global	317
14.3. Fórmula de Simpson	319
14.3.1. Errores local y global	320
14.4. Otras fórmulas de Newton-Cotes	325
14.4.1. Fórmulas de Newton-Cotes abiertas	325
14.5. Cuadratura de Gauss	326
14.5.1. Polinomios de Legendre	331
14.6. Derivación numérica	332
15. Ecuaciones diferenciales	337
15.1. Método de Euler	338
15.2. Método de Heun	341
15.3. Método del punto medio	344
15.4. Método de Runge-Kutta	347
15.5. Deducción de RK2	350
15.6. Control del paso	353
15.7. Orden del método y orden del error	359
15.7.1. Verificación numérica del orden del error	360
15.8. Métodos multipaso explícitos	362

ÍNDICE GENERAL

15.9. Métodos multipaso implícitos	366
15.10. Sistemas de ecuaciones diferenciales	371
15.11. Ecuaciones diferenciales de orden superior	373
15.12. Ecuaciones diferenciales con condiciones de frontera	376
15.13. Ecuaciones lineales con condiciones de frontera	379
A. Estilo en C	389
A.1. Generalidades	389
A.2. Ejemplo	393
A.3. Estructuras de control	395

Prólogo

El propósito de este libro es presentar los conceptos más importantes del lenguaje C y varios temas de métodos numéricos. También están algunos temas sencillos y muy útiles de C++.

Está dirigido principalmente, pero no de modo exclusivo, para el curso *Programación y Métodos Numéricos* que deben tomar los estudiantes de las carreras de Matemáticas y de Estadística en la Universidad Nacional. Estos estudiantes ya han visto cálculo diferencial y están cursando, de manera simultánea, cálculo integral y álgebra lineal. Además se supone que los lectores tienen nociones elementales sobre los computadores, en particular sobre el manejo de un editor de texto.

Los primeros 14 capítulos se pueden cubrir en un semestre. En tiempo, esto corresponde a cinco horas semanales durante quince semanas aproximadamente. Obtener un nivel medio de programación no es difícil pero requiere de manera indispensable bastante tiempo de práctica frente a un computador.

El capítulo 15, sobre solución numérica de ecuaciones diferenciales ordinarias, no corresponde al programa del curso, pero está incluido para no dejar por fuera un tema tan importante. Sin embargo, otro tema muy útil, solución numérica de las ecuaciones diferenciales parciales, no está incluido. Tampoco se trata el tema del cálculo de valores y vectores propios.

En la página electrónica del autor, se encontrará una lista de erratas del libro, que se irá completando a medida que los errores sean detectados. En esta página también está el código para algunos métodos y otros documentos relacionados con el tema. Actualmente la dirección es:

www.matematicas.unal.edu.co/~hmora/

Si hay reorganización de las páginas de la Universidad, será necesario entrar a la página

www.unal.edu.co

dirigirse a Sede de Bogotá, Facultad de Ciencias, Departamento de Matemáticas y página del autor.

Quiero agradecer especialmente a los profesores Fabio González, Fernando Bernal, Luis Eduardo Giraldo, Félix Soriano, Humberto Sarria y Héctor López quienes tuvieron la amabilidad y la paciencia de leer la versión inicial de este libro. También a los estudiantes del curso Programación y Métodos Numéricos de las carreras de Matemáticas y Estadística, en especial a Claudia Chica y Marcela Ewert. Las sugerencias, comentarios y correcciones de todos ellos fueron muy útiles. También doy gracias al profesor Gustavo Rubiano, director de Publicaciones de la Facultad de Ciencias, quien facilitó la publicación preliminar, en la colección Notas de Clase, de la primera versión de este documento.

El autor estará muy agradecido por los comentarios, sugerencias y correcciones enviados a:

hmmorae@unal.edu.co

hectormora@yahoo.com

Deseo agradecer a la Universidad Nacional por haberme permitido destinar parte de mi tiempo de trabajo a esta obra, este tiempo fue una parte importante del necesario para la realización del libro.

El texto fue escrito en L^AT_EX. Quiero agradecer al profesor Rodrigo De Castro quien amablemente me ayudó a resolver las inquietudes y los problemas presentados.

Finalmente, y de manera muy especial, agradezco a Hélène, Nicolás y Sylvie. Sin su apoyo, comprensión y paciencia no hubiera sido posible escribir este libro.

1

Introducción

El lenguaje de programación más utilizado para cálculos científicos es el Fortran, bien sea en sus versiones originales, en Fortran-77 o en Fortran-90. Inicialmente Fortran fue el único lenguaje empleado para cálculo científico. Posteriormente se utilizaron también otros lenguajes, como Algol, PL-1, Pascal, C, C++...

Actualmente se sigue utilizando ampliamente Fortran en laboratorios de investigación del más alto nivel pues, además de sus buenas cualidades, hay muchos programas y bibliotecas hechos en ese lenguaje. Estos programas han sido probados muchas veces y por tanto su confiabilidad es muy grande. No son pues creíbles las opiniones, salidas de algunos sectores informáticos, sobre la obsolescencia del Fortran.

Sin embargo, actualmente una parte importante de programas para cálculo científico se hace en C y C++. Entre sus ventajas, algunas compartidas también por Fortran, se pueden citar:

- Disponibilidad: es posible encontrar fácilmente diferentes compiladores de C y C++, para microcomputadores, minicomputadores, estaciones de trabajo y computadores grandes (*mainframes*).
- Portabilidad: los programas pueden moverse fácilmente entre diferentes tipos de computadores; por ejemplo, un programa en C hecho en un microcomputador debe poder pasar sin problemas a una estación de trabajo.
- Eficiencia.

- **Bibliotecas:** C y C++ permiten crear fácilmente interfaces para usar las bibliotecas hechas en Fortran.

En el medio estudiantil colombiano, donde fundamentalmente se usan microcomputadores con plataforma Windows, es mucho más común encontrar compiladores de C y C++ que de Fortran. Esta situación no tiene que ver con las bondades o defectos de los lenguajes, pero hace que, en la práctica, en muchos casos, sea más fácil pensar en hacer un programa en C o C++ que en Fortran. Actualmente está creciendo el uso de Linux, en el cual, casi siempre, todas las distribuciones tienen compilador de C, C++ y Fortran.

Este libro, de alcance no muy extenso, es una introducción al lenguaje C, a algunas de las características sencillas de C++ y al tema de los métodos numéricos. No contiene dos temas fundamentales de C++, que están relacionados entre sí: las clases y la programación orientada a objetos.

El libro no pretende ser de referencia, es decir, los temas no se tratan de manera exhaustiva; solamente están los detalles y características más importantes o más usados.

Se invita al lector a profundizar y complementar los temas tratados, y a continuar con el estudio, aprendizaje y práctica de algunos nuevos, en particular clases y programación orientada a objetos, en libros como [BaN94], [Buz93], [Cap94], [DeD99], [Els90], [Sch92].

C es considerado como un lenguaje de nivel medio; en cambio Pascal, Basic o Fortran se consideran de más alto nivel. Esto no es mejor ni peor, simplemente quiere decir que C utiliza al mismo tiempo características de lenguajes de alto nivel y propiedades cercanas al lenguaje de máquina. En C hay algunas funcionalidades parecidas al lenguaje interno del computador, pero esto implica que el programador debe ser mucho más cuidadoso. Por ejemplo, C no controla si los subíndices de un arreglo (algo semejante a un vector) salen fuera del rango de variación previsto.

Se puede considerar que C++ es como una ampliación de C; por tanto C++ acepta casi todo lo que es válido en C. A su vez, las órdenes y características específicas de C++ no son válidas en C.

En cuanto a métodos numéricos, sólo están algunos de los temas más importantes. Para cada tema o problema hay muchos métodos. De nuevo este libro no es, y no podría serlo, exhaustivo. Los métodos trata-

dos presentan un equilibrio entre eficiencia, facilidad de presentación y popularidad. Hay muy buenos libros de referencia para métodos numéricos y análisis numérico. Algunos de ellos son [GoV96], [IsK66], [YoG72], [DaB74], [BuF85], [LaT87], [StB93], [Atk78].

También hay libros que al mismo tiempo tratan los dos temas: C (o C++) y métodos numéricos; por ejemplo: [Pre93], [Kem87], [OrG98], [Flo95], [ReD90].

Los ejemplos de programas de este libro han sido probados en varios compiladores. El autor espera que funcionen en la mayoría. Para algunos ejemplos, aparecen los resultados producidos por el programa. Estos resultados pueden variar ligeramente de un compilador a otro; por ejemplo, en el número de cifras decimales desplegadas en pantalla.

En algunos ejemplos no están todos los detalles ni instrucciones necesarias, aparecen únicamente las instrucciones más relevantes para ilustrar el tema en estudio. Usualmente aparecerán puntos suspensivos indicando que hacen falta algunas instrucciones; por ejemplo, las instrucciones donde se leen, se definen o se les asigna un valor a las variables.

2

Generalidades

Hacer un programa en C (o en C++) significa escribir o crear un archivo texto o archivo ASCII (American Standard Code for Information Interchange) que esté de acuerdo con la sintaxis de C y que haga lo que el programador desea. Este archivo se denomina el **programa fuente** o el **código**. Generalmente, tiene la extensión **.c** o **.cpp**, es decir, si el programa se va a llamar **progr01**, entonces el programa fuente se llamará **progr01.c** o **progr01.cpp**.

El programa fuente se puede escribir con cualquier editor de texto; por ejemplo, Edit de DOS, Notepad y emacs en Windows, vi, pico, emacs en Unix y Linux, y muchos otros. También puede ser escrito por medio del editor del compilador, cuando lo tiene.

Algunos compiladores vienen con ambiente integrado de desarrollo, IDE (*Integrated Development Environment*), es decir, un ambiente en el que están disponibles, al mismo tiempo, un editor, el compilador, el ejecutor del programa, un depurador (*debugger*) y otras herramientas. Por ejemplo, Visual C++ (para Windows), Turbo C (para DOS), Dev-C++ (para Windows, software libre). En otros casos el compilador no tiene IDE. Por ejemplo, el compilador bcc32 (software libre de la casa Borland) o el compilador g++ (gcc) para Linux.

El programa fuente debe estar conforme a la sintaxis de C y debe dar las órdenes adecuadas para hacer lo que el programador desea. Una vez construida una primera versión del programa fuente, es necesario utilizar el compilador. El compilador revisa si el programa fuente está de acuerdo con las normas de C. Si hay errores, el compilador produce mensajes

indicando el tipo de error y la línea donde han sido detectados. Entonces el programador debe corregir el programa fuente y volver a compilar (utilizar el compilador). Este proceso se repite hasta eliminar todos los errores.

Si no hay errores de compilación, el compilador crea un programa ejecutable, generalmente con la extensión .exe (en Windows o DOS); por ejemplo, progr01.exe. Este programa ejecutable está en lenguaje de máquina y es el que realmente hace lo que el programador escribió en el programa fuente.

En el mejor de los casos, el programa no presenta errores durante la ejecución y además efectivamente hace lo que el programador quiere.

También puede suceder que el programa no presenta errores de ejecución, pero no hace lo que el programador quería, es decir, el programa hace lo que está en el programa fuente, pero no corresponde a lo que deseaba el programador. Entonces el programador debe revisar y modificar el programa fuente y después compilar de nuevo.

En un tercer caso hay errores durante la ejecución del programa; por ejemplo, se presenta el cálculo de la raíz cuadrada de un número negativo o una división por cero o el programa nunca termina (permanece en un ciclo sin fin). Como en el caso anterior, el programador debe revisar y modificar el programa fuente y después compilar de nuevo.

Por el momento hemos pasado por alto el proceso de enlace o encadenamiento (*link*). De manera muy esquemática se puede decir que mediante este proceso se concatenan en un único bloque varios módulos que han sido compilados por separado.

2.1 El primer programa

Uno de los programas más sencillos y pequeños puede ser el siguiente:

```
#include <stdio.h>
int main()
{
    printf(" Este es mi primer programa.\n");
    return 0;
}
```

La primera línea, `#include ...`, indica que se va a utilizar una biblioteca cuyo archivo de cabecera es `stdio.h`, es decir, la biblioteca *standard input output*. En un archivo de cabecera hay información y declaraciones necesarias para el uso correcto de las funciones de la biblioteca correspondiente.

En un programa en C hay una o varias funciones, pero siempre tiene que estar la función `main`, es decir, la función principal. La palabra `int`, que precede a `main`, indica que la función `main` devuelve un valor entero. Justamente la función `main`, en este ejemplo, devuelve 0 por medio de la instrucción `return 0`. Para la función `main` se usa la siguiente convención: devuelve 0 si no hay errores; devuelve 1 u otro valor no nulo si hubo errores.

Dentro de los paréntesis después de `main` no hay nada; esto quiere decir que la función `main` no tiene parámetros o argumentos.

Poco a poco el lector entenderá el significado del valor devuelto por una función, de los parámetros o argumentos de una función. Por el momento puede suponer que siempre se debe escribir `int main()`.

El corchete izquierdo, `{`, indica el comienzo de la función `main` y el corchete derecho, `}`, indica el final.

La función `printf` es propia de C y sirve para “escribir” o desplegar en la pantalla algún resultado. Esta función está definida en el archivo de cabecera `stdio.h`. En este ejemplo, el programa ordena escribir en la pantalla la cadena de caracteres `Este es mi primer programa`. La cadena empieza con `"` y termina con `"`. Dentro de la cadena también está `\n` que sirve para crear una nueva línea dentro de lo que aparece en la pantalla. Observe lo que pasa, durante la ejecución del programa, al cambiar la línea por una de las siguientes:

```
printf(" Este es mi primer programa.");
printf(" Este es mi primer programa.\n\n\n");
printf(" Este es mi primer \n\n programa.\n");
printf("\n\n\n\n Este es mi \n primer programa.\n");
```

Al final de las instrucciones u órdenes de C siempre hay un punto y coma. Este signo determina donde acaba la instrucción.

Este primer programa también se puede escribir con una función específica de C++.

```
#include <iostream.h>
```

```

int main()
{
    cout<<" Este es mi primer programa en C++."<<endl;
    return 0;
}

```

Aquí se utiliza el archivo de cabecera `iostream.h`, en el cual está la función `cout`. Observe la diferencia de sintaxis entre `printf` y `cout`. La orden para cambiar de línea está dada por `endl`. También se puede meter, dentro de la cadena, `\n` y se obtienen los mismos resultados.

```
cout<<" Este es mi primer programa.\n";
```

Para obtener cambios de línea adicionales se puede utilizar varias veces `endl`, por ejemplo:

```

cout<<"Este es mi primer programa."<<endl<<endl;
cout<<"Este es mi primer"<<endl<<"programa."<<endl;
cout<<endl<<"Este es mi"<<endl<<"primer programa.";
```

Algunos compiladores muy recientes no aceptan de buen gusto el anterior programa en C++; por ejemplo, el `g++` (`gcc`) 3.2 de Linux. Lo consideran anticuado (*deprecated or antiquated*). El programa se debería escribir así:

```

#include <iostream>
using namespace std;
int main()
{
    cout<<" Este es mi primer programa en C++."<<endl;
    return 0;
}

```

Observe que aparece simplemente `iostream` y no `iostream.h`. La segunda línea, `using ...`, sirve para que no sea necesario escribir

```
std::cout<<...
```

sino simplemente

```
cout<<...
```

Cuando hay varios espacios seguidos, es lo mismo que si hubiera uno solo. Las líneas en blanco no se tienen en cuenta, simplemente sirven para

la presentación del programa fuente. El programa del ejemplo también se hubiera podido escribir como sigue, pero su lectura no es fácil.

```
#include <stdio.h>
    int
main(){printf
    (   " Este es mi primer programa.\n")
        ;    return
    0;}
```

2.2 Editar, compilar, ejecutar

En esta sección hay algunas pautas generales para poder efectuar el proceso de edición, compilación y ejecución de un programa. Estas pautas pueden variar de computador en computador o de configuración en configuración; además, sirven para los casos más sencillos. Poco a poco, el programador deberá aprender a utilizar otras opciones y formas de compilación.

Suponga que el programa fuente se llama `prog01.cpp`. También se supone que el compilador que se va a usar ha sido instalado correctamente.

Para facilitar la escritura se utilizará la siguiente notación:

`>orden`

indica que se activa el botón (o parte de un menú) llamado *orden*. Esto se logra, generalmente, picando con el *mouse* dicho botón. En algunos casos se logra el mismo objetivo mediante el desplazamiento con las flechas y se finaliza oprimiendo la tecla Enter. Por ejemplo, `>File` indica que se activa el botón *File*.

La notación `Alt-f` indica que se mantiene oprimida la tecla Alt y después se pulsa ligeramente la tecla f. Finalmente se suelta la tecla Alt. De manera análoga se utiliza la misma notación para la tecla Ctrl o para la tecla \uparrow que aquí se denominará por May (mayúsculas); por ejemplo, `Ctrl-F5` o `May-F2`.

2.2.1 g++

Este compilador está en la mayoría de las distribuciones de Linux, o posiblemente en todas. Es en realidad el mismo compilador `gcc`.

Para editar el programa fuente, utilice el editor de su preferencia, por ejemplo, Emacs, vi, pico. No olvide guardar los cambios hechos.

Se puede compilar mediante la orden

```
g++ prog01.cpp
```

En este caso, si no hay errores de compilación, se produce un archivo ejecutable llamado siempre `a.out`.

Para correr el programa se debe dar la orden

```
./a.out
```

Si se da la orden

```
g++ prog01.cpp -o algo
```

el compilador produce un archivo ejecutable llamado `algo`. Es más diligente que el nombre del archivo ejecutable sea semejante al del programa fuente, por ejemplo,

```
g++ prog01.cpp -o prog01
```

Así, para correr el programa, se da la orden

```
./prog01
```

Si desea que no aparezcan las advertencias que indican que el programa está escrito en un lenguaje anticuado, digite

```
g++ prog01.cpp -o prog01 -Wno-deprecated
```

Para obtener información sobre `g++`

```
man g++
```

Para salir de la información, oprima `q`.

2.2.2 bcc32

Una manera sencilla de utilizar este compilador es en ambiente DOS, es decir, estando en Windows es necesario abrir una ventana de DOS o

ventana de sistema.

La edición del archivo se hace con cualquier editor de texto de Windows (bloc de notas, Wordpad, Emacs para Windows...) o de DOS (edit...).

Una vez editado y guardado el archivo, se compila mediante

```
bcc32 prog01.cpp
```

La orden anterior produce, si no hay errores, el archivo **prog01.exe**. Éste es un archivo ejecutable. Generalmente no es necesario especificar la extensión .cpp, el compilador la presupone, o sea, bastaría con

```
bcc32 prog01
```

Para correr el programa basta con digitar

```
prog01
```

Como se observa, no es necesario escribir la extensión .exe.

2.2.3 Turbo C++

Este compilador para DOS viene con IDE (ambiente integrado). Sin embargo también se puede utilizar únicamente el compilador de la misma forma como se usa el compilador **bcc32**. Basta con cambiar **bcc32** por **tcc**.

Para utilizar el IDE, es necesario, como primer paso, disponer de una ventana de DOS. Desde allí, el compilador y su ambiente se activan por medio de la orden

```
tc
```

Ya dentro de Turbo C++, el ambiente muestra directamente la ventana del editor. Es necesario escribir el programa o hacer las modificaciones deseadas.

Se tiene acceso a la barra de menú, en la parte superior de la ventana, por medio del *mouse* o por medio de la tecla **Alt** acompañada simultáneamente de la primera letra de la orden o submenú. Por ejemplo Alt-F para el submenú *File*.

Para grabar el archivo o los cambios:

```
>File >Save
```

Para algunas de estas secuencias de operaciones hay una tecla que reemplaza la secuencia. La secuencia *File Save* se puede reemplazar por F2.

Para compilar:

► *Compile* ► *Compile* o Alt-F9

Para correr el programa

► *Run* ► *Run* o Ctrl-F9

La secuencia anterior, cuando el programa (la última versión) no ha sido compilado, en un primer paso lo compila y después lo corre. Entonces, en muchos casos, no es necesaria la secuencia *Compile Compile*.

Después de correr el programa, el ambiente Turbo C++, vuelve inmediatamente a la pantalla donde está el editor. Si el programa muestra algunos resultados en pantalla, éstos no se ven, pues la pantalla de resultados queda inmediatamente oculta por la pantalla del editor. Para ver la pantalla de resultados:

Alt-F5

Para volver a la pantalla del editor, oprima cualquier tecla.

Para salir de Turbo C++

► *File* ► *Quit* o Alt-x

2.2.4 Visual C++

En Visual C++, los programas son proyectos. Un proyecto puede abarcar varios archivos, algunos .cpp y algunos archivos de cabecera .h. Supongamos que el proyecto se llama *prog50*, y que dentro del proyecto *prog50*, hay únicamente un archivo .cpp, el archivo *parte_a.cpp*.

La **primera vez** es necesario crear el proyecto y agregarle archivos.

► *File* ► *New* ► *Projects* ► *Win32 Console Application* ► *Project Name* *prog50* • *Create a new workspace* ► *OK* • *An empty project* ► *Finish* ► *OK*

Hasta aquí, se creó un proyecto con nombre *prog50*; esto quiere decir que hay una carpeta, con el nombre *prog50*, con la siguiente ruta (posiblemente):

```
c:\Archivos de Programa\Microsoft Visual Studio\
    My Projects\prog50\
```

Dentro de esa carpeta, Visual C++ creó la carpeta Debug y cuatro archivos: prog50.dsp, prog50.dsw, prog50.ncb y prog50.opt. Ahora es necesario agregar por lo menos un archivo fuente, un .cpp. También se pueden agregar otros archivos .cpp y archivos de cabecera .h:

*>Project >Add To Project >New >C++ Source File
 >File name parte_a >OK*

Como se utilizó la opción de un archivo nuevo, inmediatamente Visual C++ abre el editor para escribir en el archivo **parte_a.cpp**

Al finalizar la escritura en el archivo, la secuencia

>Build >Compile o Ctrl-F7

guarda el archivo **parte_a.cpp**, lo compila y, si no hay errores, crea el archivo **parte_a.obj** en la carpeta **Debug**.

La secuencia

>Build >Build o F7

hace la edición de enlaces (link) y, si no hay errores, crea el ejecutable **prog50.exe** en la carpeta **Debug**. Si es necesario, esta secuencia también guarda y compila.

La secuencia

>Build >Execute o Ctrl-F5

activa el ejecutable **prog50.exe**

Las **otras veces** ya no es necesario crear el proyecto, basta con abrir el archivo adecuado. La secuencia

>File >Recent Workspaces

permite escoger el proyecto **prog50** y hacer las modificaciones necesarias. Para compilar, enlazar y correr se utilizan las mismas secuencias anteriores.

En Visual C++ y en Dev-C++, varias de estas secuencias se pueden reemplazar por íconos que aparecen en una de las barras superiores; por ejemplo el ícono del disquete para guardar, la carpeta semiabierta para abrir un documento...

2.2.5 Dev-C++

Este compilador para Windows es software libre, viene con IDE. Su dirección es www.bloodshed.net. Después de instalado, para un programa nuevo, se puede utilizar inicialmente la siguiente secuencia:

▷File ▷New Source file

Entonces Dev-C++ abre un archivo predefinido que el programador debe completar.

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
    system("PAUSE");
    return 0;
}
```

Hay dos líneas “nuevas”: se incluye otro archivo de cabecera, el archivo **stdlib.h** (*standard library*), precisamente para poder usar la orden del sistema operativo **system("PAUSE")**. Esto sirve simplemente para que, cuando el programa haya hecho todo lo que debe hacer, no cierre inmediatamente la pantalla de resultados, sino que haga una pausa.

Una vez que el programador ha escrito su programa, la secuencia

▷File ▷Save unit o Ctrl-s

permite guardar el archivo. Si es la primera vez, es necesario dar el nombre, para nuestro ejemplo, **prog01**. Las veces posteriores, la secuencia anterior simplemente graba los cambios realizados en el archivo.

La secuencia

▷Execute ▷Compile and Run o Ctrl-F10

encadena las tres acciones importantes: guardar, compilar y ejecutar. Cuando en una ocasión anterior ya se hizo una parte del programa y se desea modificarlo, se puede utilizar la secuencia

▷File ▷Open project or file o Ctrl-o

Con la secuencia anterior se puede escoger el archivo que se desea abrir. También se puede utilizar

►File ►Reopen

La orden `system("pause")` también puede ser usada en Turbo C para no salir de la pantalla de resultados, así no es necesario utilizar Alt-F5 para regresar a la pantalla de resultados. Con los compiladores bcc32, Visual C++ y g++, no es necesaria.

2.3 Comentarios

Un comentario en C es una porción de texto que está en el programa fuente con el objetivo de hacerlo más claro, pero no influye en nada en el programa ejecutable.

```
#include <stdio.h>
// Comentario de una linea, especifico de C++
/* Comentario en C. */
/*
    Puede ser
    de varias
    lineas
*/
}

int main()
{
    printf(" Este es mi primer programa.\n");
    return 0;
}
```

Un comentario en C empieza con `/*` y acaba con `*/`. Un comentario en C++ empieza con `//` y termina donde acaba la línea. Un comentario en C puede abarcar una parte de una línea, una línea completa o varias líneas. Un comentario en C++ no puede abarcar más de una línea, pero puede haber varios comentarios seguidos. Los comentarios pueden estar después de una instrucción.

```
// Primera linea de un comentario en C++
```

```
// Segunda linea de un comentario en C++
// Tercera linea de un comentario en C++
// ...
printf(" HOLA.\n"); // escribe HOLA
printf(" HOLA.\n"); /* escribe HOLA */
```

Cuando el programador está haciendo el programa tiene muy claro qué significa cada parte del programa. Sin embargo, al cabo de unas semanas, en especial si ha estado haciendo otros programas, el significado de cada parte del programa empieza a hacerse difuso. Más aún, el programa fuente debe ser suficientemente claro para que pueda ser leído por otras personas. La inclusión de comentarios adecuados permite hacer más claro un programa y facilita enormemente su corrección y actualización posterior. Los comentarios tienen que ver con varios aspectos, por ejemplo: objetivos del programa, datos, resultados, método empleado, significado de las variables...

2.4 Identificadores

Los identificadores sirven para el nombre de variables o de funciones. La longitud máxima de un identificador depende del compilador utilizado y del lenguaje (C o C++). Generalmente no es inferior a 32; más aún, puede ser muy grande. Los identificadores sólo usan símbolos alfanuméricos, es decir, letras minúsculas, letras mayúsculas, dígitos y el símbolo de subrayado:

```
a b c... x y z A B C... X Y Z 0 1 2 3 4 5 6 7 8 9 _
```

El primer carácter no puede ser un dígito. No puede haber espacios dentro de un identificador. Las minúsculas se consideran diferentes de las mayúsculas. Un identificador no puede ser una palabra clave. Las palabras clave están reservadas para C o C++. Usualmente la lista de palabras clave es la siguiente:

asm	auto	break	case	catch
char	class	const	continue	default
delete	do	double	else	enum
extern	float	for	friend	goto
if	inline	int	long	new

```
operator  private   protected  public    register
return    short     signed      sizeof    static
struct    switch    template   this      throw
try      typedef   union      unsigned  virtual
void      volatile while
```

De los siguientes identificadores posibles, los cuatro primeros son correctos, los cinco últimos son inadecuados. El primer identificador es diferente del segundo.

```
a1
A1
peso_especifico_del_mercurio
b1j
peso especifico
peso-especifico
ia
peso'esp
float
```

Además, para los identificadores, es preferible no utilizar los nombres de las funciones usuales de C o de C++.

En los siguientes dos ejemplos, para los identificadores se utilizan nombres de funciones de entrada y salida. En el primer ejemplo no hay conflicto, pero de todas maneras es aconsejable no usar `printf` para un identificador.

```
double printf = 4.5;

cout<<printf<<endl;
```

En el segundo ejemplo hay conflicto entre la función `printf`, usada en la segunda línea, y el identificador `printf`.

```
double printf = 4.5;

printf(" %lf\n", printf);
```

Los identificadores deben ser explicativos con respecto a la variable que representan. El tercero, `peso_especifico_del_mercurio`, es un buen

ejemplo. Sin embargo es un poco largo. Si un identificador es muy largo, no hay error; simplemente es ineficiente cuando hay que escribir varias, o muchas, veces el identificador.

Buscar que sean muy cortos puede llevar a que sean poco o nada explicativos; por ejemplo, usar

pem

es corto, y tal vez eficiente, pero dice poco con respecto al peso específico del mercurio.

Usualmente, cuando es necesario, se emplean las tres o cuatro primeras letras de cada palabra. Para diferenciar visualmente las palabras, éstas se separan por el símbolo de subrayado o se escribe la primera letra de cada palabra (a partir de la segunda) en mayúsculas; por ejemplo, un identificador para el peso específico del mercurio puede ser

pesoEspHg
peso_esp_hg

Obviamente, si por tradición las variables tienen nombres muy cortos, entonces no se justifica utilizar identificadores largos. Por ejemplo, de manera natural, los identificadores para los coeficientes de una ecuación cuadrática pueden ser

a b c

Adicionalmente, uno o varios comentarios adecuados pueden explicar lo que hace un programa y el significado de cada identificador.

```
// Programa para la solucion de la ecuacion cuadratica
//
//  a x x + b x + c = 0
//
```

En los comentarios es posible, dependiendo del sistema operativo, utilizar letras tildadas, ñ y Ñ. Sin embargo, estos caracteres se pueden perder al pasar de un sistema operativo a otro, por ejemplo, al pasar de DOS a Windows. Entonces **es preferible usar letras sin tildes, no importa que sea un error de ortografía.**

En un programa hay, o debe haber, una correspondencia biunívoca entre las variables y los identificadores. Por esta razón, de manera un poco abusiva, algunas veces se utilizan indistintamente. Por ejemplo, en lugar de decir la variable peso específico del mercurio con identificador `pesoEspHg`, se utilizará con frecuencia: la variable `pesoEspHg`.

2.5 Tipos de datos

Los principales tipos de datos son:

```
char
int
float
double
```

El primero, `char`, se usa para los caracteres, no sólo los alfanuméricos, también para los otros caracteres usuales como + (= ? [* , etc. El tipo `int` se usa para números enteros. Para los números reales (tienen una parte entera y una parte fraccionaria no necesariamente nula) se usan los tipos `float` y `double`.

Este último es de mayor tamaño y permite almacenar más información, pero a su vez necesita más espacio para su almacenamiento y un poco más de tiempo para las operaciones. En general, para hacer cálculos numéricos, si no hay restricciones muy fuertes de disponibilidad de memoria, es preferible utilizar `double` para los reales. Generalmente se conoce el tipo `float` con el nombre de punto flotante en precisión sencilla y `double` con el nombre de punto flotante en doble precisión o simplemente doble precisión.

Antes de su uso en el programa, generalmente al comienzo de las funciones, es necesario declarar el tipo de las variables, por ejemplo:

```
double a, b, c;
int i, denom;
int j;
```

La siguiente tabla muestra los tamaños usuales en bytes (un byte, u octeto, es una unidad de información compuesta por 8 bits; un bit, acrónimo de binary digit, es la unidad más sencilla de información y puede tomar

los valores 0 o 1) de los tipos anteriores, y el rango de variación.

Tipo	Tamaño (bytes)	Rango
char	2	
int	2	-32768 a 32767
float	4	3.4E-38 a 3.4E38
double	8	1.7E-308 a 1.7E308

La tabla anterior puede variar con el computador o con el compilador. Con algunos compiladores el tipo entero usa cuatro bytes y tiene un rango de variación mucho más amplio

La notación usada no es la del idioma español. Aquí la parte entera se separa de la fraccionaria con punto y no con coma, como es lo correcto en español; es decir, aquí 1.7 representa una unidad y siete décimas. Consecuentemente no se utiliza punto para separar las unidades de mil de las centenas, ni se utiliza la comilla para separar los millones. También se está utilizando una convención para la notación científica, a saber,

$$\begin{aligned} 3.45\text{E}42 &:= \frac{345}{100} \times 10^{42}, \\ 0.345\text{E}-38 &:= \frac{345}{1000} \times 10^{-38}. \end{aligned}$$

En la notación científica de C también se puede utilizar la e minúscula en lugar de la mayúscula; por ejemplo, `3.45e42`.

Para los tipos `float` y `double`, en la tabla está escrito únicamente el rango positivo. Es necesario sobrentender el cero y el rango negativo; por ejemplo, el tipo `float` también puede ser cero o variar entre -3.4E38 y -3.4E-38.

Mediante los modificadores

`unsigned long`

se obtienen nuevos tipos. Los más usuales están en la siguiente tabla.

Tipo	Tamaño (bytes)	Rango
<code>unsigned int</code>	2	0 a 65535
<code>long int</code>	4	-2147483648 a 2147483647
<code>long double</code>	10	3.4E-4932 a 3.4E4932

Los valores de la tabla anterior también pueden variar con el tipo de computador o con el compilador.

El tipo `long double` debe ser manejado con mucho cuidado, pues en varios compiladores aparentemente funciona bien la compilación de programas con variables `long double`, pero puede haber errores en los resultados. Consulte detalladamente la documentación del compilador que usa, en lo referente al manejo de `long double`.

2.6 Operador de asignación

Es el operador más utilizado. Mediante su uso se le asigna a una variable un nuevo valor. La forma general es:

variable = expresión;

Lo anterior indica que la variable, que siempre está a la izquierda del signo igual, toma el valor dado por la expresión de la derecha. La asignación no se debe confundir con el significado de una igualdad en una expresión matemática. Algunos ejemplos permiten aclarar el uso y significado de la asignación.

```
int a;
double x, y;
char c;

a = 100;
y = 9.0/4.0;
x = 5.0 + y;
a = a+3;
c = 'A';
```

Antes de la instrucción `a = 100` la variable `a` tenía un valor determinado, o tal vez ningún valor específico. De todas maneras, después de la instrucción `a = 100` la variable entera `a` vale cien. Después de la instrucción `y = 9.0/4.0` la variable `y` vale 2.25. Como la variable `y` corresponde a un número real, es preferible que la expresión de la derecha sea hecha explícitamente entre números reales; por esto en este caso es preferible escribir `9.0/4.0` y no simplemente `9/4`. Como se verá más

adelante, el valor de $9/4$ es el entero 2. Después de la instrucción `x = 5.0 + y`, la variable `x` vale 7.25.

La instrucción `a = a + 3` sería un error desde el punto de vista matemático, trabajando en el conjunto de los números reales. En cambio en C tiene un sentido preciso, toma el valor que tiene `a`, le adiciona 3 y el resultado es el nuevo valor de `a`. O sea, antes de la instrucción, `a` vale 100. Después de la instrucción, `a` vale 103.

En el momento en que se especifica el tipo de una variable, se puede hacer una asignación inicial. Por ejemplo, la instrucción

```
int i = 3, j = 5;
```

dice que las variables `i`, `j` son enteras y además a `i` se le asigna el valor 3 y a `j` el valor 5.

2.7 Operadores aritméticos

Los principales operadores aritméticos son:

```
+ - / * % ++ --
```

Como era de esperarse `+` `-` `/` indican la adición, la sustracción y la división. Para la multiplicación entre dos números se emplea `*` (el asterisco). Por ejemplo

```
int i, j, k;
double x, y;

i = 27;
j = 4;
k = i+j;
cout<<k<<endl;
x = 10.0;
y = 3.0;
cout<<x/y<<endl;
cout<<x*y<<endl;
cout<<i/j<<endl;
```

produce la escritura en pantalla de los resultados

```

31
3.33333
30
6

```

Es importante observar que al efectuar la división entre dos números enteros i, j ($j \neq 0$), C da como resultado la parte entera de i/j cuando i/j es positivo y menos la parte entera de $-i/j$ en caso contrario. En resumen, el resultado es

$$\text{signo}(i/j) \lfloor |i/j| \rfloor .$$

Por ejemplo,

```

int i, j;
i = 27;
j = -4;
cout<<i/j<<endl;

```

produce la escritura en pantalla de -6 .

El operador $-$ además de representar la sustracción, operación binaria, sirve para indicar el cambio de signo. La instrucción

```
x = -y;
```

asigna a la variable x el inverso aditivo del valor de y .

El operador $\%$ se aplica únicamente a enteros e indica el residuo entero de la división. Por ejemplo,

```

int i, j, k;
i = 25;
j = 7;
k = i%j;
cout<<k<<endl;

```

producirá como resultado la escritura en pantalla del valor 4.

El operador $++$ es un operador “unario”, o sea, tiene un solo argumento. Se aplica a variables enteras. Su efecto es aumentar en una unidad. Hay dos formas de usarlo, antes o después de la variable. El resultado final, para la variable a la que se le aplica, es el mismo, pero los valores intermedios pueden ser diferentes. En el ejemplo

```

int i = 4, j = 4, k;

k = i++;
cout<<i<<" "<<k<<endl;
k = ++j;
cout<<j<<" "<<k<<endl;

```

el resultado será la escritura de

```

5 4
5 5

```

Tanto para *i* como para *j* el valor inicial resultó incrementado en una unidad. En la primera asignación, la variable *k* tomó el valor de *i* antes del incremento. En la segunda asignación, la variable *k* tomó el valor de *j* después del incremento.

En caso de duda, en lugar de utilizar una sola instrucción, se puede realizar exactamente lo mismo mediante dos instrucciones que no presentan posibilidad de mala interpretación.

<i>k</i> = <i>i</i> ++;	es equivalente a	<i>k</i> = <i>i</i> ;
		<i>i</i> ++;

<i>k</i> = ++ <i>j</i> ;	es equivalente a	<i>j</i> ++;
		<i>k</i> = <i>j</i> ;

también es equivalente a	<i>+j</i> ;
	<i>k</i> = <i>j</i> ;

El operador *--* es el operador de decremento. Su uso es análogo al de *++*.

2.8 Prioridad de los operadores aritméticos

Cuando en una expresión hay varios operadores aritméticos, existen reglas precisas de prioridad, llamada también **precedencia**, que indican el orden en que el computador debe efectuar las operaciones.

- En una expresión aritmética puede haber paréntesis redondos, es decir (,). No están permitidos, como paréntesis, los rectangulares

[,], ni los corchetes { , }. Como se verá posteriormente, los paréntesis rectangulares [,] se utilizan para los subíndices.

- Los paréntesis tienen prioridad sobre todos los operadores y deben cumplir las reglas usuales de los paréntesis, por ejemplo: deben ir por parejas (uno izquierdo y uno derecho); en el orden usual, de izquierda a derecha, no puede aparecer un paréntesis derecho si no va precedido de su correspondiente paréntesis izquierdo; cuando hay paréntesis anidados, unos dentro de otros, los paréntesis internos tienen prioridad sobre los paréntesis externos.
- Cuando en una expresión hay dos operadores iguales, sin paréntesis, es necesario conocer la clase de asociatividad, es decir, se requiere saber si se asocia utilizando el orden de izquierda a derecha o el orden de derecha a izquierda.

Al encontrar la expresión $a/b/c$, dado que la división es una operación binaria, habría dos posibilidades. La primera consiste en efectuar primero el cociente a/b y dividir el resultado por c . La segunda consiste en efectuar primero el cociente b/c y enseguida dividir el valor a por el resultado. Utilizando paréntesis, las dos posibilidades se pueden expresar así:

$$\begin{array}{l} (a/b)/c \\ a/(b/c) \end{array}$$

La primera correspondería a asociatividad de izquierda a derecha y la segunda de derecha a izquierda. Obviamente, cuando una operación es asociativa, por ejemplo la adición, no importa en qué orden se haga la asociación. Para la división, el lenguaje C, como está indicado en la siguiente tabla, asocia de izquierda a derecha, entonces $a/b/c$ es lo mismo que $(a/b)/c$.

La siguiente tabla muestra la precedencia, de mayor a menor, y la asociatividad de los operadores aritméticos más usados.

Operador		Asociatividad	Precedencia
++	posincremento	izq. a derecha	mayor
--	posdecremento	izq. a derecha	
++	preincremento	derecha a izq.	
--	predecremento	derecha a izq.	
-	cambio de signo	derecha a izq.	
*	multiplicación	izq. a derecha	
/	división	izq. a derecha	
%	resto de la div. entera	izq. a derecha	
+	adición	izq. a derecha	
-	sustracción	izq. a derecha	
=	asignación	derecha a izq.	menor

Por medio de paréntesis, que tienen prioridad superior a los operadores aritméticos, los siguientes ejemplos muestran el orden en que se realizan algunas operaciones.

```
double a = 1.2, b = 3.1, c = -2.5, d = 5.1, x;

x = a/b/c;           // x = (a/b)/c;
x = a*b+c/d-c*b;   // x = ( (a*b) + (c/d) ) - (c*b);
x = a/-c;           // x = a/(-c);
```

De todas maneras, en caso de duda o para facilitar la lectura, se recomienda el uso adecuado de paréntesis o de espacios.

```
double a = 1.2, b = 3.1, c = -2.5, d = 5.1, x;

x = (a/b)/c;
x = a*b + c/d - c*b;
x = a/(-c);
```

2.9 Abreviaciones usuales

Una de las instrucciones frecuentes en los programas de computador es la siguiente: se toma el valor de una variable y se le suma una constante o una variable y el resultado se convierte en el nuevo valor de la variable. Por ejemplo:

```
a = a + 2.0;
b = b + c;
```

En C, las dos asignaciones anteriores se pueden escribir de manera abreviada así:

```
a += 2.0;
b += c;
```

Al principio, parece un poco raro o difícil de leer, pero finalmente se adquiere la costumbre. De manera análoga, se puede abbreviar cuando hay multiplicaciones, restas o divisiones. O sea, las instrucciones

```
a = a*3.0;
i = i - 2;
b = b/c;
```

se pueden abbreviar por

```
a *= 3.0;
i -= 2;
b /= c;
```

2.10 Funciones matemáticas

C tiene predefinidas en sus bibliotecas varias funciones matemáticas cuyo archivo de cabecera es `math.h`. Por ejemplo, la función `sqrt` permite obtener la raíz cuadrada. Su argumento es tipo `double` y su resultado también.

El siguiente programa permite calcular las raíces de una ecuación cuadrática, solamente cuando éstas son reales. Más adelante se verá el caso general. Además, hay otro inconveniente: este programa sirve únicamente para los valores definidos en el programa. También se verá posteriormente cómo hacer para que el programa reciba otros valores y trabaje con ellos.

```
// Calculo de las raices de una ecuacion cuadratica,
// cuando son reales.
```

```
#include <iostream.h>
#include <math.h>

int main()
{
    double a = 7.0, b = 3.1, c = -5.72;
    double raiz1, raiz2;

    raiz1 = (-b + sqrt(b*b-4.0*a*c) )/(2.0*a);
    raiz2 = (-b - sqrt(b*b-4.0*a*c) )/(2.0*a);

    cout<<" Las raices son: "<<raiz1<<" "<<raiz2<<endl;
    return 0;
}
```

El programa anterior producirá el siguiente resultado.

Las raices son: 0.709256 -1.15211

Al hacer un programa, la primera preocupación es que funcione. La segunda es que sea eficiente. En el programa anterior hay varios cálculos repetidos: $b*b-4.0*a*c$ y $2.0*a$. Hacer el programa anterior más eficiente tendrá una ganancia imperceptible, milésimas de segundo o menos. Pero cuando se hace más eficiente un proceso que se realiza millones de veces, la mejoría puede ser notable.

```
// Calculo de las raices de una ecuacion cuadratica,
// cuando son reales.

#include <iostream.h>
#include <math.h>

int main()
{
    double a = 7.0, b = 3.1, c = -5.72;
    double raiz1, raiz2, d, a2;

    a2 = 2.0*a;
    d = sqrt(b*b-4.0*a*c);
```

```

raiz1 = (-b + d )/a2;
raiz2 = (-b - d )/a2;

cout<<"\n\n "<<a<< " x*x + "<<b<< " x + "<<c<<endl;
cout<<"Las raices son: "<<raiz1<< " "<<raiz2<<endl;
return 0;
}

```

En la siguiente tabla están las principales funciones matemáticas de C. Para todas, el resultado es tipo **double**. Todas, salvo **pow** , tienen un solo argumento. **pow(x,y)** evalúa x^y , ambos argumentos son tipo **double** y el resultado también lo es.

		Argumentos	Resultado
acos	arco coseno	double	double
asin	arco seno	double	double
atan	arco tangente	double	double
ceil	parte entera superior	double	double
cos	coseno	double	double
cosh	coseno hiperbólico	double	double
exp	exponencial: e^x	double	double
fabs	valor absoluto	double	double
floor	parte entera inferior	double	double
log	logaritmo neperiano	double	double
log10	logaritmo en base 10	double	double
pow	x^y	double, double	double
sin	seno	double	double
sinh	seno hiperbólico	double	double
sqrt	raíz cuadrada	double	double
tan	tangente	double	double
tanh	tangente hiperbólica	double	double

Si se utiliza alguna de estas funciones con un argumento fuera de su conjunto de definición, se producirá un error durante la ejecución del programa, este se detendrá y producirá un pequeño aviso. Por ejemplo, si se utiliza **sqrt** para un número negativo, el aviso producido, dependiendo del compilador, puede ser:

sqrt: DOMAIN error

Es responsabilidad del programador verificar, antes del llamado a la función, que se va a utilizar un argumento adecuado. De manera análoga, cuando hay divisiones, se debe estar seguro de que el divisor no es nulo. En caso contrario, debe haber un control previo para evitar una terminación anormal del programa.

2.11 Entrada de datos y salida de resultados

La manera natural de entrar datos al programa, cuando hay pocos, es por medio del teclado. En C, se hace mediante la función `scanf`. Los siguientes ejemplos ilustran su uso.

```
// Calculo de las raices de una ecuacion cuadratica,
// cuando son reales.

#include <stdio.h>
#include <math.h>

int main()
{
    double a, b, c, raiz1, raiz2, d, a2;

    printf("\n Raices de  a*x*x + b*x + c = 0.\n\n");

    printf(" a = ");
    scanf("%lf", &a);

    printf(" b = ");
    scanf("%lf", &b);

    printf(" c = ");
    scanf("%lf", &c);

    printf("\n %lf x*x + %lf x + %lf = 0\n\n", a, b, c);

    a2 = 2.0*a;
    d = sqrt(b*b-4.0*a*c);
    raiz1 = (-b + d )/a2;
```

```

raiz2 = (-b - d )/a2;

printf(" Raices: %12.4lf  %15.6lf\n", raiz1, raiz2);
return 0;
}

```

Después de que el programa escribe en la pantalla `a =`, sigue la orden `scanf`. Entonces el usuario, por medio del teclado, deberá digitar el valor de `a`. Para finalizar esta orden el usuario deberá oprimir la tecla Return o Enter.

Allí, lo que está entre comillas es la cadena de formato. Para este caso, `%lf` es el formato para números tipo `double`. La siguiente lista muestra otros tipos de formato. Algunos corresponden a conceptos todavía no presentados en este libro. Consulte la documentación del compilador que usa, sobre los formatos para números `long int` y `long double`.

<code>%d</code>	<code>int</code>
<code>%f</code>	<code>float</code>
<code>%lf</code>	<code>double</code>
<code>%u</code>	<code>unsigned int</code>
<code>%e</code>	<code>double, float</code> en notación científica
<code>%i</code>	<code>int</code>
<code>%c</code>	<code>char</code>
<code>%s</code>	cadena de caracteres
<code>%p</code>	puntero o apuntador

Es importante recalcar que en la lectura mediante `scanf`, la variable debe ir precedida de `&`. Es un error muy común, especialmente en programadores conocedores de otros lenguajes (Pascal, Fortran u otros), no colocar este signo. No es un error de compilación, pero el resultado no será el deseado en la lectura.

Después de la lectura de `a`, de manera análoga, el programa lee los otros dos valores, el de `b` y el de `c`.

En la orden:

```
printf("\n %lf x*x + %lf x + %lf = 0\n\n", a, b, c);
```

la cadena de formato tiene tres veces el formato `%lf`, correspondientes a las tres variables que aparecen después de la cadena. De acuerdo con la

cadena de formato, el resultado producido será:

un cambio de línea,

un espacio,

el valor de **a**,

x*x +

el valor de **b**,

x +

el valor de **c**,

= 0

dos cambios de línea.

Dicho de otra manera, el resultado es semejante a escribir la cadena

"\n **x*x + x + = 0\n\n"**

insertando en el lugar adecuado los valores de **a**, **b**, **c**.

En la orden **printf**, las variables que van a ser escritas no van precedidas de signo **&**, como era el caso para **scanf**.

Es una buena costumbre escribir los datos, o una parte de ellos, tan pronto han sido leídos, antes de efectuar los cálculos, para verificar que la lectura se desarrolló de manera adecuada.

En los casos anteriores, al utilizar simplemente **%lf**, el programador no tiene control sobre el número de cifras decimales ni sobre la longitud del campo (número de columnas utilizadas para la escritura del número). En la última orden de escritura, hay modificadores para el formato. Por ejemplo, el formato **%12.4lf** indica que el valor de **raiz1** ocupará 12 columnas, de las cuales las cuatro últimas son para escribir cuatro cifras decimales. Por defecto, las salidas siempre están ajustadas a la derecha, entonces los espacios sobrantes aparecerán a la izquierda en blanco. Si **raiz1** valiera **-10.0/3.0**, entonces aparecerían en pantalla cinco espacios seguidos de **-3.3333**. Cuando el número no cabe en la longitud de campo prevista, C toma el espacio necesario para la escritura del número.

Los modificadores también se pueden utilizar para otros tipos de formato; por ejemplo:

%15.4f

```
%10.4e
%5d
```

La lectura de datos no tiene que hacerse necesariamente uno por uno, como en el programa anterior. La lectura de las tres variables se podría hacer también mediante:

```
printf(" Entre los coeficientes a b c : ");
scanf("%lf%lf%lf", &a, &b, &c);
```

En este caso, durante la ejecución del programa, después de que sale en pantalla el aviso “ Entre los coeficientes a b c : ”, el usuario debe digitar los tres valores, separados entre sí por uno o más espacios en blanco, y finalizar oprimiendo la tecla Enter.

En C++, las entradas y salidas se hacen mediante `cin` y `cout`. El programa anterior podría quedar de la siguiente manera:

```
// Calculo de las raices de una ecuacion cuadratica,
// cuando son reales.

#include <iostream.h>
#include <math.h>

int main()
{
    double a, b, c, raiz1, raiz2, d, a2;

    cout<<"\n Raices de a*x*x + b*x + c = 0.\n\n";

    cout<<" a = ";
    cin>>a;

    cout<<" b = ";
    cin>>b;

    cout<<" c = ";
    cin>>c;

    cout<<"\n "<<a<<" x*x + "<<b<<" x + "<<c<<" = 0.\n\n";
```

```

a2 = 2.0*a;
d = sqrt(b*b-4.0*a*c);
raiz1 = (-b + d )/a2;
raiz2 = (-b - d )/a2;

cout<<" Las raices son: "<<raiz1<<" "<<raiz2<<endl;
return 0;
}

```

En la entrada de datos mediante **cin** no se requiere colocar el signo & antes de la variable. Tampoco hay que especificar el formato, no importa que sea entero, punto flotante o doble precisión.

La utilización de modificadores de formato como longitud de campo y número de cifras decimales con las órdenes **cin** y **cout**, está fuera del alcance de este libro.

También es posible hacer una sola orden para entrar varios datos al tiempo. La entrada de los tres coeficientes se podría hacer mediante:

```

cout<<" Entre los coeficientes a b c : ";
cin>>a>>b>>c;

```

C no controla explícitamente las entradas, ni por teclado ni desde archivo. Por ejemplo, si el programa está esperando leer un número y el usuario se equivoca y en lugar de digitar un cero digita la letra o, entonces el programa hace alguna conversión y continúa con el proceso. En otros lenguajes, cuando esto sucede, el programa se detiene e informa sobre el error.

C puede controlar si la lectura se efectuó de manera adecuada ya que la función **scanf** devuelve un valor entero que indica el número de campos bien leídos y almacenados. Si no hubo almacenamiento, **scanf** devuelve 0, en otros casos de error devuelve EOF (generalmente EOF es lo mismo que -1). El control entonces se puede hacer con un **if**, instrucción que se verá en el siguiente capítulo. Una vez que el lector conozca el manejo del **if**, puede volver a este párrafo para entender el esquema siguiente que muestra cómo se podría hacer el control.

```
int result;
```

```

...
result = scanf("%lf", &a);
if( result <= 0 ){
    ...
}
else {
    ...
}

```

A lo largo de este libro se supondrá que la entrada de datos se hace con valores adecuados y por tanto no se controla su funcionamiento. Pero, en un nivel más avanzado, un programador experimentado y cuidadoso debería controlar siempre la entrada de datos.

2.12 Conversiones de tipo en expresiones mixtas

Cuando en una expresión hay constantes o variables de distintos tipos, C hace conversiones temporales de tipo, operación (binaria) por operación, buscando el tipo que permita un mejor manejo de los operandos, es decir, el tipo del operador de mayor nivel. Por ejemplo, en una operación binaria (como adición, multiplicación...)

```

double      y  float   → double,
double      e  int     → double,
float       e  int     → float,
short int   e  int     → int,
long double y  double  → long double,
long double y  float   → long double,
long double e  int     → long double.

```

En el ejemplo

```

int i = 8, j = 3;
double x = 1.5, u, z;

```

```

u = i/j*x;
z = i*x/j;
cout<<u<<endl;
cout<<z<<endl;

```

aparentemente *u* y *z* tienen el mismo valor, puesto que, según el orden en que se efectúan las operaciones, en ambos casos se tiene $(i \cdot x)/j$. Pero en realidad *u* vale 3.0 y *z* vale 4.0. En cada expresión hay dos operaciones de igual prioridad, entonces el orden es de izquierda a derecha. Para *u* se hace primero la división entera, con resultado 2, y enseguida el producto doble precisión, y da 3.0. Para *z* se hace primero la multiplicación doble precisión, con resultado 12.0, y enseguida la división doble precisión, y da 4.0.

2.13 Moldes

Algunas veces es necesario utilizar temporalmente el valor de una variable en otro tipo diferente de aquel con que fue definida. Supongamos que la variable *i* se va a utilizar muchas veces como entera y por tanto ha sido definida como entera, pero en algún momento se requiere usar su valor como doble precisión. Esto se hace en C, anteponiendo el molde (**double**). Por ejemplo:

```

int i, j;
double x, y;

i = 3;
j = 4;
y = (double)i;
x = y*y/(double)j;

```

En el ejemplo anterior, **no se requiere colocar el molde** a *j*, puesto que la multiplicación *y*y* da doble precisión, y al hacer la división de un valor doble precisión por un entero, se hace previamente la conversión del valor entero a doble precisión.

De igual manera se pueden utilizar los moldes (**float**) e (**int**). Al utilizar el molde (**int**) para valores **float** o **double**, C trunca al entero más cercano comprendido entre 0 y el valor, o sea,

$$(\text{int})x = \text{signo}(x)\lfloor |x| \rfloor .$$

En C++ los moldes tienen forma funcional, visualmente más clara. Por ejemplo, el molde `double()` es una función cuyo argumento es de cualquier tipo y el resultado es doble precisión. Así, son ejemplos de moldes en C++:

```
int i;
double y;

i = 3;
y = sqrt(double(i));
//y = sqrt(i);
cout<<y<< " <<int(y)<<endl;
```

En este ejemplo tampoco se requiere el molde para `i`.

Ejercicios

Para cada uno de los enunciados siguientes, defina cuáles son los datos necesarios. Haga un programa que lea los datos, realice los cálculos y muestre los resultados.

- 2.1** Resuelva un sistema de dos ecuaciones lineales con dos incógnitas. Suponga que tiene una única solución.
- 2.2** Resuelva un sistema de tres ecuaciones lineales con tres incógnitas. Suponga que tiene una única solución.
- 2.3** Resuelva una ecuación polinómica (o polinomial) de tercer grado por medio de la fórmula de Tartaglia y Cardano. Suponga que se tiene el caso más sencillo.
- 2.4** Calcule el coseno de un ángulo medido en grados, minutos y segundos.
- 2.5** Dado un ángulo en radianes, conviértalo a grados, minutos y segundos.
- 2.6** Un objeto ha recorrido una distancia d (metros) en un tiempo t (horas, minutos, segundos y milésimas de segundo). Calcule su velocidad promedio en km/h y el número de segundos necesarios para recorrer un kilómetro.

- 2.7** Calcule la cuota mensual fija para un préstamo por un monto de M pesos, con una tasa de interés mensual i (en porcentaje) y con un plazo de n meses.
- 2.8** Dadas las longitudes de los lados de un triángulo, calcule, en grados, los ángulos.
- 2.9** Dadas las longitudes de los lados de un triángulo, calcule su área.
- 2.10** Dadas las coordenadas, en \mathbb{R}^2 , de los vértices de un triángulo, calcule, en grados, los tres ángulos.
- 2.11** Dadas las coordenadas, en \mathbb{R}^2 , de los vértices de un triángulo, calcule su área.
- 2.12** Dadas las coordenadas, en \mathbb{R}^2 , de los vértices de un triángulo, calcule el centro del círculo inscrito.
- 2.13** Dadas las coordenadas, en \mathbb{R}^2 , de los vértices de un triángulo, calcule el centro del círculo circunscrito.
- 2.14** Dadas las coordenadas, en \mathbb{R}^2 , de los vértices de un triángulo, calcule el ortocentro.
- 2.15** Dadas las coordenadas, en \mathbb{R}^2 , de los vértices de un triángulo, calcule el baricentro.
- 2.16** Dadas las coordenadas, en \mathbb{R}^3 , de los vértices de un triángulo, calcule, en grados, los 3 ángulos.
- 2.17** Dadas las coordenadas, en \mathbb{R}^3 , de los vértices de un triángulo, calcule su área.
- 2.18** Dadas las coordenadas de puntos puntos en \mathbb{R}^3 , calcule el volumen del tetraedro y la área total (suma de las cuatro áreas).
- 2.19** Dadas las coordenadas de los vértices de un cuadrilátero convexo, calcule su área.
- 2.20** Dadas las coordenadas de los vértices de un cuadrilátero no convexo, calcule su área.
- 2.21** Calcule la distancia de un punto a una recta y encuentre el punto de la recta más cercano al punto dado.

- 2.22** Calcule la distancia de un punto a un plano y halle el punto del plano más cercano al punto dado.
- 2.23** Dada una parábola convexa (hacia arriba) y un punto \bar{x} , encuentre el punto de la parábola más cercano a \bar{x} .
- 2.24** Dado un plano y la recta que pasa por dos puntos u y v , halle el punto de intersección de la recta y el plano.
- 2.25** Dado un plano y la recta que pasa por un punto u y es paralela a d , obtenga el punto de intersección de la recta y el plano.
- 2.26** Dados dos vectores en \mathbb{R}^3 , calcule el producto interno (escalar), el ángulo entre ellos y el producto vectorial.
- 2.27** Hay dos maneras usuales de medir la eficiencia o la economía en el consumo de combustible de un vehículo: número de litros necesarios para 100 kilómetros, y número de kilómetros por galón. Dado el número de litros para 100 kilómetros, calcule el número de kilómetros por galón.
- 2.28** Dado el número de kilómetros por galón, calcule número de litros para 100 kilómetros.

)

3

Estructuras de control

En los ejemplos vistos hasta ahora, los programas tienen una estructura lineal, es decir, primero se efectúa la primera orden o sentencia, enseguida la segunda, después la tercera y así sucesivamente hasta la última. Por el contrario, la mayoría de los programas, dependiendo del cumplimiento o no de alguna condición, toman decisiones diferentes. También es muy usual repetir una o varias instrucciones mientras sea cierta una condición. Estas bifurcaciones y repeticiones se logran mediante las sentencias o estructuras de control.

(

3.1 if

En uno de los casos más sencillos el **if** tiene la siguiente forma:

```
sentencia_1;  
if( condicion ) sentencia_a;  
else sentencia_b;  
sentencia_2;
```

Las líneas primera y última no hacen parte del **if**, pero aparecen para ver mejor la secuencia de realización de las órdenes. Tal como está presentado, las órdenes de estas dos líneas siempre se realizan. Si la condición se cumple, se realiza *sentencia_a*, y si no se cumple, se realiza *sentencia_b*. En resumen, dependiendo de que se cumpla o no la condición, se realiza

una de las dos secuencias siguientes:

<i>sentencia_1;</i> <i>sentencia_a;</i> <i>sentencia_2;</i>	<i>sentencia_1;</i> <i>sentencia_b;</i> <i>sentencia_2;</i>
---	---

El siguiente ejemplo, muy sencillo, permite asignar a *c* el máximo entre *a* y *b*.

```
double a, b, c;
...
printf(" a = %lf ,  b =  %lf\n", a, b);
if( a >= b ) c = a;
else c = b;
printf(" c = %lf\n", c);
```

En este ejemplo no están todas las instrucciones necesarias para el funcionamiento correcto; aparecen únicamente las instrucciones más relevantes para ilustrar el tema en estudio. Los puntos suspensivos indican que hacen falta algunas instrucciones, por ejemplo, las instrucciones donde se leen, se definen o se les asigna un valor a las variables *a* y *b*. Mientras no se indique lo contrario, se usará la anterior convención a lo largo del libro.

En la estructura *if* no es necesaria la parte *else*. Se puede tener una estructura de la siguiente forma.

```
sentencia_1;
if( condicion ) sentencia_a;
sentencia_2;
```

En este caso, si la condición se cumple, se realiza *sentencia_a*, y si no se cumple, el programa no hace nada y salta a la siguiente sentencia. Así, las dos posibles secuencias de realización son:

<i>sentencia_1;</i> <i>sentencia_a;</i> <i>sentencia_2;</i>	<i>sentencia_1;</i> <i>sentencia_2;</i>
---	--

En el siguiente ejemplo, también se asigna a *c* el máximo entre *a* y *b*.

```
int a, b, c;
```

```

...
cout<<" a = "<<a<<" ,   b =  "<<b<<endl;
c = a;
if( b > a ) c = b;
cout<<" c = "<<c<<endl;

```

En el **if**, tanto *sentencia_a* como *sentencia_b* se pueden reemplazar por bloques de sentencias. Un **bloque de sentencias**, o simplemente un **bloque**, es una sucesión de sentencias encerradas entre llaves {}, { }.

```

sentencia_1;
if( condicion ) {
    sentencia_a1;
    sentencia_a2;
    ...
}
else {
    sentencia_b1;
    sentencia_b2;
    ...
}
sentencia_2;

```

Si la condición se cumple, se realiza el primer bloque; si no se cumple, se realiza el segundo bloque. En resumen, dependiendo de que se cumpla o no la condición, se realiza una de las dos secuencias siguientes:

<i>sentencia_1;</i>	<i>sentencia_1;</i>
<i>sentencia_a1;</i>	<i>sentencia_b1;</i>
<i>sentencia_a2;</i>	<i>sentencia_b2;</i>
:	:
<i>sentencia_2;</i>	<i>sentencia_2;</i>

Un **if** puede anidar (dentro de él), uno o varios **if** u otras estructuras de control, que a su vez pueden anidar otras estructuras de control.

El siguiente programa considera las raíces reales y las raíces complejas de una ecuación cuadrática, dependiendo del valor del discriminante $b^2 - 4ac$.

```
// Raices de una ecuacion cuadratica.

//#include <iostream.h>
#include <math.h>
#include <stdio.h>

int main()
{
    double a, b, c, raiz1, raiz2, d, a2, pReal, pImag;

    printf("\n Raices de  a*x*x + b*x + c = 0.\n\n");

    printf(" Entre los coeficientes  a  b  c : ");
    scanf("%lf%lf%lf", &a, &b, &c);

    printf("\n %lf x*x + %lf x + %lf = 0.\n\n", a, b, c);

    a2 = 2.0*a;
    d = b*b-4.0*a*c;
    if( d >= 0.0 ){
        // raices reales
        d = sqrt(d);
        raiz1 = (-b + d )/a2;
        raiz2 = (-b - d )/a2;
        printf("\n Raices reales: %lf , %lf\n", raiz1,
               raiz2);
    }
    else{
        // raices complejas
        d = sqrt(-d);
        pReal = -b/a2;
        pImag = d/a2;
        printf("\n Raices complejas: ");
        printf("%lf + %lf i , %lf - %lf i\n",
               pReal, pImag, pReal, pImag);
    }
    return 0;
}
```

3.2 Operadores relacionales y lógicos

Los operadores relacionales —mayor, menor, mayor o igual, menor o igual, igual, diferente— permiten comparar dos variables o valores generalmente numéricos:

<	menor
>	mayor
<=	menor o igual
>=	mayor o igual
==	igual
!=	diferente

Los operadores lógicos son:

&&	y
 	o
!	no

Sus tablas de verdad son exactamente las mismas de la lógica elemental. Sean p, q dos proposiciones. La proposición p y q es verdadera únicamente cuando ambas son verdaderas; en los demás casos es falsa. La proposición p o q es verdadera cuando alguna de las dos es verdadera; es falsa únicamente cuando las dos son falsas. La negación de una proposición verdadera es falsa; la negación de una proposición falsa es verdadera.

Como en C no existe explícitamente un tipo de dato lógico o booleano, entonces se usan los enteros. Así, falso se hace equivalente a 0 y verdadero a cualquier valor no nulo, usualmente 1. Consecuentemente en C, más que mirar si una condición es verdadera o falsa, se mira si la expresión es no nula o nula.

Por ejemplo, es lo mismo escribir

```
if( i != 0 )...
```

que escribir

```
if( i )...
```

La primera expresión es, tal vez, más fácil de entender, pues concuerda con nuestra manera lógica de razonar. Es más claro decir “si i es diferente de cero...” que decir “ si $i \neq 0$ ”.

También se puede escribir como condición del **if** (y de otras estructuras de control) expresiones menos simples.

```
if( a*c-b*d )...
```

Más aún, esa expresión puede ser de tipo doble precisión. Esto es lícito aunque no muy usual. La condición sería falsa únicamente cuando $a*c-b*d$ fuera exactamente 0.0. Para otros valores, aún muy cercanos a cero, como 1.0E-20, la condición es verdadera. Lo anterior es un poco inexacto, en realidad depende del compilador, del computador y del tipo de la expresión. Para un valor doble precisión, en algunos compiladores, los valores positivos menores a 2.4E-323 se consideran nulos.

Como las comparaciones dan como resultado un valor entero 0 o 1, se pueden hacer asignaciones con este valor. Esto quita un poco de claridad, pero es completamente lícito. Por ejemplo, se puede escribir

```
n = x >= 0.0;
```

Si x es no negativo, entonces n toma el valor 1. Si x es negativo, entonces n toma el valor 0. El programa anterior, *prog04a*, hacía el cálculo de las raíces reales o complejas de una ecuación cuadrática, pero no consideraba algunos casos muy raros, pero posibles. Por ejemplo, a puede ser cero, entonces la ecuación no solamente no es cuadrática, sino que el programa presenta una división por cero (situación que un programador siempre debe evitar). El siguiente programa usa estructuras **if** anidadas y considera todos los posibles casos.

```
// Raices de una ecuacion cuadratica.
// Considera los casos extremos: a=0, a=b=0...

##include <iostream.h>
#include <math.h>
#include <stdio.h>

int main()
{
```

```

double a, b, c, raiz1, raiz2, d, a2, pReal, pImag;

printf("\n Raices de a*x*x + b*x + c = 0.\n\n");

printf(" Entre los coeficientes a b c : ");
scanf("%lf%lf%lf", &a, &b, &c);

printf("\n %lf x*x + %lf x + %lf = 0.\n\n", a, b, c);

if( a != 0.0 ){
    // ecuacion realmente cuadratica
    a2 = 2.0*a;
    d = b*b-4.0*a*c;
    if( d >= 0.0 ){
        // raices reales
        d = sqrt(d);
        raiz1 = (-b + d )/a2;
        raiz2 = (-b - d )/a2;
        printf("\n Raices reales: ");
        printf(" %lf , %lf\n", raiz1, raiz2);
    }
    else{
        // raices complejas
        d = sqrt(-d);
        pReal = -b/a2;
        pImag = d/a2;
        printf("\n Raices complejas: ");
        printf("%lf +- %lf i\n", pReal, pImag);
    }
}
else{
    // ecuacion no cuadratica
    if( b != 0.0 ){
        // ecuacion lineal
        printf("La ecuacion es lineal. Su raiz es: %lf\n",
               -c/b);
    }
    else{

```

```

    if( c == 0.0 )printf(" Todo real es solucion.\n");
    else printf(" No hay solucion.\n");
}
}
return 0;
}

```

3.3 for

La estructura **for** permite repetir varias veces una sentencia o una secuencia de sentencias. Esta estructura repetitiva recibe el nombre de **bucle**, lo mismo que las estructuras **while** y **do while** que se verán más adelante.

La aplicación clásica del **for** consiste en hacer algo para **i** variando desde 1 hasta **n**. En el siguiente ejemplo, se calcula la suma de 1 hasta 15 de $1/i$ con escritura de las sumas parciales.

```

double suma;
int i;

suma = 0.0;
for( i=1; i <= 15; i++){
    suma += 1.0/i;
    printf("%5d %12.6lf\n", i, suma);
}

```

Inicialmente **i** vale 1. Si **i** es menor o igual que 15, entonces se actualiza **suma** y se hace la escritura. Enseguida se incrementa **i** en una unidad; si todavía **i** es menor o igual que 15 se repite el proceso. Al final se hace el proceso para **i** con valor de 15. De nuevo se incrementa el valor de **i**, su nuevo valor es 16. Ahora **i** no cumple la condición, entonces se detiene el proceso.

La forma general del **for** es:

```
sentencia_a;
for(INIC; CDC; MODIF) {
    sentencia_1;
    sentencia_2;
    :
}
sentencia_b;
```

Si se necesita repetir una sola instrucción, la forma del **for** es:

```
sentencia_a;
for(INIC; CDC; MODIF) sentencia_1;
sentencia_b;
```

Dentro de los paréntesis del **for** hay tres partes:

- inicialización,
- condición de continuación,
- modificación.

Las partes están separadas entre sí por punto y coma. Usualmente, pero no es obligatorio, hay una variable llamada de control; en el ejemplo anterior es la variable *i*.

En la inicialización se asigna a la variable de control su valor inicial. Esto se hace una sola vez, no se vuelve a hacer en el **for**. Es lo primero que se hace antes del resto del **for**.

Si la condición de continuación se cumple (si la expresión es no nula), entonces se realiza el cuerpo del **for** (lo que está entre las llaves o la única instrucción presente). Si la primera vez no se cumple la condición, entonces nunca se realiza el cuerpo del **for**.

Cada vez que se realiza el cuerpo del **for**, después de realizar la última instrucción, se efectúa la modificación que esté indicada en la tercera parte. En muchos casos lo que se hace es simplemente incrementar en una unidad la variable de control.

Ninguna de las tres partes es indispensable. Lo único realmente indispensable es la presencia de los dos signos punto y coma. El siguiente ejemplo no presenta errores de sintaxis, pero tiene un defecto muy desagradable, nunca acaba.

```
for( ; ; ) cout<<"Buenos dias\n";
```

En el siguiente ejemplo se calcula el factorial de n , cuando n es pequeño. El resultado es un valor entero. Para valores de n mayores que 12 es necesario que la variable **nFact** sea doble precisión.

```
int n, i, nFact;

cout<<" n = ";
cin>>n;
nFact = 1;
for( i = 2; i <= n; i++) nFact *= i;
cout<<nFact;
```

En la parte de inicialización del **for** puede haber más de una inicialización. En este caso deben estar separadas por una coma. De manera análoga, puede haber más de una modificación y también es necesario separar con una coma. En el siguiente ejemplo, **i** empieza en 1 y **j** empieza en 20; al final de cada repetición **i**, aumenta en una unidad y **j** disminuye en 2 unidades.

```
int n, i, j, suma;

n = 20;
suma = 0;
for( i=1, j=n; i<j && i*i<j*j-10*j ; i++, j -= 2){
    suma += i+j;
    printf(" %d %d \n",i,j);
}
printf(" suma = %d \n", suma);
```

El resultado producido será la escritura de los valores de **i** y de **j** y al final el valor de **suma**. Entonces escribirá:

```
1 20
2 18
3 16
4 14
suma = 78
```

Después de realizar el **for** con **i=4** y **j=14**, se hacen las modificaciones indicadas; **i** pasa a valer 5 y **j** valdrá 12. Se cumple la condición **i<j**, pero no la condición **i*i<j*j-10*j**. Entonces se acaba el **for** y se pasa a la siguiente instrucción, es decir, la escritura de **suma**.

3.4 while

La estructura **while** permite repetir varias veces una sentencia o una secuencia de sentencias. Es como una simplificación del **for**. Aquí no hay variable de control, no hay una parte explícita de inicialización ni de modificación. Usualmente en el **for** se conocía por anticipado el número de veces que se iba a efectuar el cuerpo del **for**. En el **while**, simplemente, cuando no se cumpla la condición se acaba el **while**. La forma general del **while** es:

```
sentencia_a;
while( condicion ) {
    sentencia_1;
    sentencia_2;
    :
}
sentencia_b;
```

Si se necesita repetir una sola instrucción, la forma del **while** es:

```
sentencia_a;
while( condicion ) sentencia_1;
sentencia_b;
```

Después de *sentencia_a*, para empezar el **while** se verifica la condición de continuación. Si se cumple, se realizan las sentencias dentro del cuerpo del **while**: *sentencia_1* , *sentencia_2*... Cada vez que se realiza la última sentencia dentro del cuerpo del **while**, se pasa a verificar la condición, y si se cumple, se vuelve a realizar el grupo de sentencias del **while**. Si la primera vez no se cumple la condición, entonces nunca se efectúan las sentencias del **while**.

El desarrollo en serie de Taylor para la función exponencial está dado por:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

La fórmula anterior es exacta, pero irrealizable para hacer una evaluación numérica, puesto que no se puede efectuar un número infinito de operaciones (n varía desde 1 hasta infinito). Lo que se hace usualmente es aplicar la fórmula hasta que el término $x^n/n!$ sea muy pequeño comparado con la suma parcial. Sea

$$a_k = \frac{x^k}{k!}, \quad s_n = \sum_{k=0}^n a_k.$$

Dado un valor positivo ε cercano a cero, por ejemplo 0.00000001, el cálculo aproximado de e^x se realiza hasta que

$$\frac{|a_n|}{1 + |s_n|} \leq \varepsilon.$$

El 1 en el denominador se colocó simplemente para asegurar que el denominador no sea nulo. Una salida más elegante y precisa sería evaluar una cota para el resto y utilizar este valor para detener el proceso.

```
// Programa para calcular una aproximacion de exp(x).

#include <math.h>
#include <stdio.h>

int main()
{
    double x, sn, an, nFact, eps;
    int n;

    printf("\n Calculo aproximado de exp(x).\n\n x = ");
    scanf("%lf", &x);

    eps = 1.0e-8;
    sn = 1.0 + x;
    nFact = 1;
    n = 1;
    an = x;
    while( fabs(an)/( 1.0 + fabs(sn) ) > eps ){
        n++;
        nFact *= n;
        an = x;
        for( int i = 1; i < n; i++ )
            an *= x / i;
        sn += an;
    }
}
```

```

    an = pow(x,n)/nFact;
    sn += an;
}
printf(" exp(x) = %lf\n", sn);
return 0;
}

```

Se puede lograr más eficiencia, se evita el uso de `pow` y el cálculo explícito de $n!$, si se tiene en cuenta que

$$a_n = a_{n-1} \frac{x}{n}.$$

```

eps = 1.0e-8;
sn = 1.0 + x;
n = 1;
an = x;
while( fabs(an)/( 1.0 + fabs(sn) ) > eps ){
    n++;
    an *= x/n;
    sn += an;
}

```

Para la función coseno, su serie de Taylor, alrededor de cero, es:

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \dots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

La (una) fórmula recursiva es:

$$a_n = -a_{n-1} \frac{x^2}{2n(2n-1)}.$$

La porción de programa siguiente calcula una aproximación del valor de $\cos x$.

```

double x, sn, an, eps, xx;
int n2; // 2*n

printf("\n Calculo aproximado de cos(x).\n\n x = ");
scanf("%lf", &x);

```

```

eps = 1.0e-8;
xx = x*x;
an = -xx/2.0;
sn = 1.0 + an;
n2 = 2;
while( fabs(an)/( 1.0 + fabs(sn) ) > eps ){
    n2 += 2;
    an *= -xx/n2/(n2-1.0);
    sn += an;
}
printf(" cos(x) = %lf\n", sn);

```

3.5 do while

Es muy parecida a la estructura `while`, pero el control se hace al final. Esto hace que el cuerpo del `do while` siempre se realice por lo menos una vez. La forma general del `do while` es:

```

sentencia_a;
do {
    sentencia_1;
    sentencia_2;
    :
} while( condicion );
sentencia_b;

```

Si se necesita repetir una sola instrucción, la forma del `do-while` es:

```

sentencia_a;
do sentencia_1;
while( condicion );
sentencia_b;

```

Consideremos dos enteros positivos p y q . Su máximo común divisor se puede calcular por el algoritmo de Euclides. Este es un proceso iterativo en el que en cada iteración, dados dos enteros positivos $p_k \geq q_k$, se halla el cociente c_k y el residuo r_k al dividir p_k por q_k :

$$p_k = c_k q_k + r_k.$$

Para pasar a la nueva iteración, el divisor pasa a ser dividendo y el residuo pasa a ser divisor.

$$\begin{aligned} p_{k+1} &= q_k, \\ q_{k+1} &= r_k. \end{aligned}$$

El proceso se repite hasta que el residuo sea nulo. El máximo común divisor estará dado por el último divisor q . Por ejemplo, consideremos $p = 60$, $q = 18$. La siguiente tabla muestra los pasos intermedios del algoritmo

p_k	q_k	c_k	r_k
60	18	3	6
18	6	3	0

Entonces el máximo común divisor es 6. Para $p = 34$, $q = 15$, los siguientes son los resultados parciales.

p_k	q_k	c_k	r_k
34	15	2	4
15	4	3	3
4	3	1	1
3	1	3	0

Entonces el máximo común divisor es 1, es decir, 34 y 15 son primos relativos.

Este algoritmo es un ejemplo adecuado para ver la utilización de la estructura de control **do-while**. Es necesario efectuar por lo menos una división. Si ese primer residuo fuera nulo, el m.c.d. sería el menor de los dos números.

```
// Maximo comun divisor de dos numeros

//#include <iostream.h>
#include <math.h>
#include <stdio.h>

int main()
{
    int p, q, c, r, temp;

    printf("\n Maximo comun divisor de 2 enteros positivos");
```

```

printf("\n p = ");
scanf("%d", &p);
printf(" q = ");
scanf("%d", &q);
printf("\n\n p = %d , q = %d\n", p, q);

if( p < 1 || q < 1 ){
    printf("\n\n Numeros inadecuados\n\n");
    return;
}
// intercambio, si necesario, para que p > q
if( p < q ){
    temp = p;
    p = q;
    q = temp;
}
do{
    c = p/q;
    r = p%q;
    printf("%5d%5d%5d%5d\n", p, q, c, r);
    p = q;
    q = r;
} while( r!= 0 );
printf(" m.c.d. = %d\n", p);
return 0;
}

```

Observe que, para que dos variables intercambien sus valores, se necesita utilizar una variable intermedia que almacene temporalmente uno de los dos valores.

La instrucción **return** permite salir inmediatamente de la función, en este caso de la función **main**, es decir, permite salir del programa. Más adelante se verá que **return** también sirve para que las funciones devuelvan un valor.

3.6 switch

El `if` permite la bifurcación en dos vías, dependiendo de la verdad o falsedad de una expresión. Algunas veces se requiere que la bifurcación se haga en más de dos vías, dependiendo de que una variable de control tome un valor, otro valor, un tercer valor... Esto se podría hacer mediante varios `if`, pero es poco elegante y pierde claridad. La estructura `switch` permite las bifurcaciones en varias vías dependiendo de una variable de control, cuyo tipo debe ser necesariamente `int` o `char`. La forma general es la siguiente:

```
switch( variable ){
    case valor1 :
        sentencia_1_1
        sentencia_1_2
        ...
        break;
    case valor2 :
        sentencia_2_1
        ...
        break;
    ...
    default :
        sentencia_d_1
        ...
}
sentencia_b;
```

Si la variable de control tiene el primer valor, se efectúan las sentencias del primer grupo y después el programa pasa a realizar `sentencia_b`. Si la variable de control tiene el segundo valor, se efectúan las sentencias del segundo grupo y después el programa pasa a realizar `sentencia_b`. Si la variable de control no tiene ninguno de los valores previstos, el programa realiza las sentencias de la parte `default` y en seguida efectúa `sentencia_b`. La parte `default` no es indispensable desde el punto de vista de la sintaxis, pero en la práctica puede ser útil o a veces necesaria.

Cuando para dos valores diferentes de la variable de control se tiene que hacer el mismo grupo de sentencias, en lugar de hacer `case valor_i`: grupo de sentencias y `case valor_j`: grupo de sentencias, se coloca simplemente `case valor_i: case valor_j: grupo de sentencias`.

En el siguiente ejemplo, a partir de una **nota**, valor entero entre 0 y 50, se escribe un aviso:

“Aprueba. Muy bien.” si $40 \leq \text{nota} \leq 50$,
 “Aprueba.” si $30 \leq \text{nota} \leq 39$,
 “Puede habilitar.” si $20 \leq \text{nota} \leq 29$,
 “Reprueba.” si $0 \leq \text{nota} \leq 19$,

La siguiente porción de programa hace la implementación, basada en la cifra de las decenas.

```
int nota, dec;

printf(" nota = ");
scanf("%d", &nota);
if( 0 <= nota && nota <= 50 ){
    dec = nota/10;
    switch( dec ){
        case 5: case 4:
            printf(" %2d Aprueba. Muy bien.\n", nota);
            break;
        case 3:
            printf(" %2d Aprueba.\n", nota);
            break;
        case 2:
            printf(" %2d Puede habilitar.\n", nota);
            break;
        default:
            printf(" %2d Reprueba.\n", nota);
            break;
    }
}
else printf(" %d Nota inadecuada.\n", nota);
```

3.7 break

Además del uso en la estructura **switch**, la instrucción **break** se puede utilizar en las estructuras **for**, **while**, **do while**. En estos bucles el control se hace siempre al comienzo o al final del cuerpo del bucle. En

algunos casos, en programas bastante complejos puede ser útil salir del bucle, mediante un **if**, en un sitio intermedio del bucle. Para esto se usa la instrucción **break**.

```

sentencia_a;
for( INIC; CDC; MODIF ){
    sentencia_1;
    sentencia_2;
    ...
    if( condicion_2 ){
        sentencia_c_1;
        sentencia_c_2;
        ...
        break;
    }
    ...
}
sentencia_b;
```

Si en una pasada del programa por el cuerpo del bucle, se cumple la condición del **if**, *condicion_2*, entonces el programa realiza las instrucciones *sentencia_c_1*, *sentencia_c_2*, ..., y salta a la instrucción *sentencia_b*.

La siguiente porción de programa calcula la suma de los enteros entre *n1* inclusive y *n2* inclusive, menores que el mínimo múltiplo de 10 en ese mismo intervalo. Si en el conjunto {*n1*, *n1*+1..., *n2*} no hay múltiplos de 10, calcula la suma de todos ellos. También calcula la suma de los cuadrados de esos enteros.

```

int n1, n2, i, div, res;
double sx, sxx;

div = 10;
printf(" n1 n2 : ");
scanf("%d%d", &n1, &n2);

sx = 0.0;
sxx = 0.0;
for( i = n1; i <= n2; i++ ){
    res = i%div;
    if( !res ) break; // if( res == 0 ) break;
```

```

    sx += i;
    sxx += i*i;
    printf("%5d\n", i);
}
printf(" suma x = %lf ,  suma x*x = %lf\n", sx, sxx);

```

Si hay varios bucles anidados, la instrucción **for** permite salir de un solo bucle, del bucle más interno donde se encuentra el **break**.

3.8 continue

La instrucción **continue** permite saltar la parte restante del cuerpo de un bucle —**for**, **while** o **do while**— y pasar a empezar una nueva iteración. Su uso en una parte intermedia de un bucle va asociado generalmente a un **if**.

```

sentencia_a;
for(INIC; CDC; MODIF) {
    sentencia_1;
    sentencia_2;
    ...
    if( condicion_2 ){
        sentencia_c_1;
        sentencia_c_2;
        ...
        continue;
    }
    sentencia_3;
    sentencia_4;
    ...
}
sentencia_b;

```

Si en una pasada del programa por el cuerpo del bucle, se cumple la condición del **if**, *condicion_2*, entonces el programa realiza las instrucciones *sentencia_c_1*, *sentencia_c_2*, ..., y vuelve a empezar una nueva iteración sin ejecutar las sentencias restantes del bloque **for** (*sentencia_3*, *sentencia_4*...). Como en el esquema anterior se trata de un **for**, entonces realiza la modificación indicada en **MODIF**.

La siguiente porción de programa calcula la suma de los enteros no múltiplos de 10, entre **n1** inclusive y **n2** inclusive. También calcula la suma de los cuadrados de esos enteros.

```
int n1, n2, i, div, res;
double sx, sxx;

div = 10;
printf(" n1 n2 : ");
scanf("%d%d", &n1, &n2);

sx = 0.0;
sxx = 0.0;
for( i = n1; i <= n2; i++){
    res = i%div;
    if( !res ) continue; // if( res == 0 ) continue;
    sx += i;
    sxx += i*i;
    printf("%5d\n", i);
}
printf(" suma x = %lf ,  suma x*x = %lf\n", sx, sxx);
```

En el ejemplo anterior, cuando **i** es un múltiplo de 10, el programa pasa a efectuar una nueva iteración; entonces, antes de calcular **res**, como está indicado en este **for**, incrementa en una unidad el valor de **i**.

3.9 goto y exit

La instrucción **goto** no debería usarse en los programas llamados bien estructurados, pues permite salidas múltiples de un bucle y además puede transferir la secuencia del programa a instrucciones diferentes de la siguiente al bucle. Sin embargo, usado con moderación y en casos de real necesidad, puede ser útil, por ejemplo, para salir de varios bucles anidados mediante una sola instrucción.

El **goto** debe estar seguido del nombre de una etiqueta. Una etiqueta es simplemente un identificador seguido de dos puntos. La etiqueta debe existir dentro de la misma función donde está el **goto**. Hasta ahora sólo se ha visto la función **main**. El siguiente capítulo corresponde justamente

al tema de las funciones.

```
{
    ...
    ... goto final1;
    ...
final1:
    ...
}
```

La función `exit()`, en la biblioteca estándar, cuyo archivo de cabecera es `stdlib.h`, permite salir anticipadamente de un programa. Generalmente se usa `exit(0)` para indicar que se trata de una terminación normal y `exit(1)` para indicar algún problema durante el programa (también podría ser otro valor diferente de 0).

Una de las formas para saber si un entero n , mayor que 1, es primo, consiste en averiguar si n es divisible por los enteros entre 2 y la raíz cuadrada de n . El siguiente programa implementa este sencillo algoritmo.

```
// Programa para saber si un entero mayor que 1 es primo

//#include <iostream.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i, nPrimo, res;

    printf("\n Averigua si el entero n > 1 es primo.\n");
    printf("\n n = ");
    scanf("%d", &n);
    printf("\n\n n = %d", n);

    if( n <= 1 ){
        printf("\n\n Valor inadecuado.\n\n");
        exit(1);
    }
```

```

}

nPrimo = 1;
// 1 indica que n es primo, 0 indica que no.
// Inicialmente se supone que n es primo.
for( i=2; i*i <= n; i++){
    res = n%i; // residuo
    if( !res ){
        nPrimo = 0;
        break;
    }
}
if( nPrimo ) printf(" es primo.\n");
else printf(" no es primo, es divisible por %d\n", i);
return 0;
}

```

Ejercicios

Para cada uno de los enunciados siguientes, defina cuáles son los datos necesarios. Haga un programa que lea los datos, verifique que no hay inconsistencias (por ejemplo, distancias negativas o nulas; dos vértices de un triángulo iguales; recta que pasa por un punto paralela a $d = 0$...), estudie todas las posibilidades relajando algunas suposiciones (por ejemplo: solución única, sin solución, número infinito de soluciones...), realice los cálculos y muestre los resultados.

- 3.1** Considere los ejercicios del capítulo anterior.
- 3.2** Dadas las longitudes de los lados de un triángulo, averigüe si éste es acutángulo, rectángulo u obtusángulo.
- 3.3** Escriba la expresión binaria, en orden inverso, de un entero no negativo.
- 3.4** Escriba la expresión en base p (p entero, $2 \leq p \leq 9$), en orden inverso, de un entero no negativo.
- 3.5** Escriba la expresión hexadecimal, en orden inverso, de un entero no negativo.

- 3.6** Escriba en letras un entero n , $31 \leq n \leq 99$.
- 3.7** Escriba en letras un entero n , $1 \leq n \leq 99$.
- 3.8** Escriba en letras un entero n , $1 \leq n \leq 999$.
- 3.9** Calcule una aproximación del valor de e utilizando parte de la serie

$$\sum_{k=0}^{\infty} \frac{1}{k!}.$$

- 3.10** Calcule por medio de la serie de Taylor truncada un valor aproximado de $\sin(x)$. Muestre el número de términos de la serie necesarios para que el valor absoluto de la diferencia entre el valor obtenido y el valor producido por la función `sin` sea menor que `1.0E-12`.
- 3.11** Convierta coordenadas cartesianas en polares.
- 3.12** Convierta coordenadas cartesianas en esféricas.
- 3.13** Convierta coordenadas polares en cartesianas.
- 3.14** Convierta coordenadas esféricas en cartesianas.
- 3.15** Dados tres números, escríbalos en orden creciente.
- 3.16** Dados tres números, escríbalos en orden decreciente.
- 3.17** Dados cuatro números, escríbalos en orden creciente.
- 3.18** Dados cuatro números, escríbalos en orden decreciente.
- 3.19** Calcule el mínimo común múltiplo de dos enteros positivos.
- 3.20** Simplifique un racional.
- 3.21** Haga las cuatro operaciones entre dos racionales.
- 3.22** Haga las cuatro operaciones entre dos complejos.
- 3.23** Calcule z^n , donde z es un complejo y n un entero positivo.
- 3.24** Dado un entero positivo n , escriba los primos menores o iguales a n .

- 3.25** Dado un entero positivo n , calcule el número de primos menores o iguales a n .
- 3.26** Dado un entero positivo n , calcule su descomposición canónica, $n = p_1^{m_1} p_2^{m_2} \dots p_k^{m_k}$, donde los p_i son primos diferentes y los m_i son enteros positivos.
- 3.27** Dado un entero positivo n , calcule la función de Möbius $\mu(n)$. Recuérdese que $\mu(1) = 1$, $\mu(n) = 0$ si n es divisible por un cuadrado distinto de 1; $\mu(n) = (-1)^k$ si 1 es el único cuadrado divisor de n y n tiene k divisores primos.
- 3.28** Dado un entero positivo n , calcule por conteo la función de Euler $\varphi(n)$. Recuérdese que $\varphi(n)$ indica el número de primos relativos con n en el conjunto $\{0, 1, 2, \dots, n - 1\}$. Por convención $\varphi(1) = 1$.
- 3.29** Dado un entero positivo n , calcule la función de Euler $\varphi(n)$. Recuérdese que $\varphi(1) = 1$ y para $n > 1$, si la descomposición canónica es $n = p_1^{m_1} p_2^{m_2} \dots p_k^{m_k}$, entonces

$$\varphi(n) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_k}\right).$$

- 3.30** Dados tres enteros a , b y m positivos, halle todas las soluciones de la congruencia $ax \equiv b$ (mód. m).
- 3.31** Dados un número x y un entero no negativo n , calcule una aproximación de la función de Bessel de primera clase y de grado n ,

$$\begin{aligned} J_n(x) &= x^n \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m}}{2^{2m+n} m! (n+m)!}, \\ &= \sum_{m=0}^{\infty} \frac{(-1)^m}{m! (n+m)!} \left(\frac{x}{2}\right)^{2m+n}. \end{aligned}$$

- 3.32** Halle, para el compilador y el computador que usted usa, el mayor entero y el menor entero utilizables con variables tipo `int`. Compare con los valores de `limits.h`

- 3.33** Halle, para el compilador y el computador que usted usa, el mayor entero y el menor entero utilizables con variables tipo `long int`. Compare con los valores de `limits.h`
- 3.34** Halle, para el compilador y el computador que usted usa, el mayor entero y el menor entero utilizables con variables tipo `unsigned int`. Compare con los valores de `limits.h`
- 3.35** Dado un entero positivo n , escriba los primeros n números de Fibonacci. Recuérdese que $u_1 = 1$, $u_2 = 1$, $u_k = u_{k-1} + u_{k-2}$, $k \geq 3$.
- 3.36** Dado un número x y un entero no negativo n , calcule el polinomio de Legendre de grado n evaluado en x . Recuérdese que $P_0(x) = 1$, $P_1(x) = x = 1$,

$$P_{k+1}(x) = \frac{2k+1}{k+1} x P_k(x) - \frac{k}{k+1} P_{k-1}(x).$$

4

Funciones

Como se mencionó anteriormente, de manera muy esquemática, se puede decir que un programa fuente en C es una sucesión de funciones que se llaman (se utilizan) entre sí. Siempre tiene que estar la función `main`.

4.1 Generalidades

La forma general de una función en C es la que se presenta a continuación. Posteriormente hay varios ejemplos que permiten un mejor manejo de las las funciones.

```
tipo nombre_funcion(tipo argumento, tipo argumento, ...)  
{  
    cuerpo de la funcion  
}
```

Considere el siguiente programa. En él hay una función que calcula el factorial de un número entero. Esta función es llamada varias veces desde el programa principal.

```
// Factorial de los enteros entre 0 y 15.  
// Usa una funcion.  
//-----  
#include <math.h>  
#include <stdio.h>
```

```
#include <stdlib.h>
//-----
double fact( int k );
//=====
int main()
{
    int n;

    printf("\n Factorial de los enteros en [0,15]\n");

    printf("\n\n      n          n!\n\n");
    for( n = 0; n <= 15; n++){
        printf("%5d%14.0lf\n", n, fact(n));
    }
    return 0;
}
//=====
double fact( int k )
{
    // calcula el factorial de k >= 0
    // devuelve 0 si k < 0.

    int n;
    double kFact = 1.0;

    if( k < 0 ){
        printf(" %d : valor inadecuado para factorial\n");
        return 0.0;
    }
    for( n = 2; n <= k; n++) kFact *= n;
    return kFact;
}
```

Antes del comienzo de la función `main` está el prototipo de la función `fact`. Este prototipo es casi igual a la primera línea del sitio donde realmente se define la función, después del final de la función `main`. La única diferencia es que al final del prototipo hay un punto y coma. El prototipo informa que la función `fact` devuelve un valor doble precisión y tiene un único parámetro (o argumento) tipo entero.

El cuerpo de la función debe estar entre dos corchetes (entre { y }). La función usa dos variables, **n** de tipo entero y **kFact** de tipo doble precisión. Esta variable **n** de la función **fact** es completamente distinta de la variable **n** de la función **main**.

Como la función **fact** devuelve un valor (hay funciones que no devuelven ningún valor), en alguna parte del cuerpo de la función, por lo menos una vez, debe estar **return** seguido de un valor adecuado. Además, siempre la última instrucción en una salida normal de la función, antes de que el programa regrese al sitio donde fue llamada la función, debe ser un **return**. Por ejemplo, la siguiente modificación produce, durante su compilación, advertencias (*warnings*) sobre posibles errores.

```
double factMal( int k )
{
    int n;
    double kFact = 1.0;

    if( k < 0 ){
        printf(" %d : valor inadecuado para factorial\n");
        return 0.0;
    }
    return kFact;
    for( n = 2; n <= k; n++) kFact *= n;
}
```

La línea **for** nunca se realizaría pues el programa al pasar por **return kFact** sale de la función.

La función del siguiente ejemplo tiene un único parámetro entero y no devuelve ningún valor. Esto se indica por medio de **void**. Esta función simplemente escribe en la pantalla el número indicado de líneas en blanco (en realidad hace *n* saltos de línea). Su llamado puede ser **lineas(2);** o también **lineas(j);** .

```
void lineas( int n )
{
    // escribe n lineas en blanco.

    int i;
```

```

    for( i = 1; i <= n; i++) printf("\n");
}

```

La siguiente función tiene dos parámetros doble precisión y devuelve un valor doble precisión. Devuelve el valor dominante entre dos valores. El valor dominante es aquel cuyo valor absoluto es mayor.

```

double dominante( double a, double b )
{
    // calcula el valor dominante.

    if( fabs(a) >= fabs(b) ) return a;
    else return b;
}

```

La función del siguiente ejemplo no tiene parámetros ni devuelve un valor. Su oficio es simplemente crear una detención hasta que el usuario oprima la tecla Enter. Su llamado es simplemente `pausa();`

```

void pausa(void)
{
    char c;

    printf(" Oprime ENTER para continuar ");
    scanf("%c",&c);
}

```

En el ejemplo que sigue, también hay una pausa hasta que el usuario oprima cualquier tecla. Requiere el uso de la función `getch()`, que no hace parte de la norma ANSI C. Su archivo de cabecera es `conio.h`. Viene con algunos compiladores, por ejemplo, los de la casa Borland.

```

void pausa2(void)
{
    // necesita conio.h

    printf(" Oprime una tecla para continuar ");
    getch();
}

```

Si no se ha encontrado una raíz, el proceso continúa con uno de los dos intervalos $[a, m]$ o $[m, b]$. En cada iteración el tamaño del intervalo se reduce a la mitad. Entonces, o se encuentra la raíz, o se llega a un intervalo muy pequeño donde hay una raíz y cualquiera de los dos extremos del intervalo se puede considerar como raíz.

Por ejemplo, considere la ecuación $f(x) = x^5 - 3x^4 + 10x - 8 = 0$.

$$f(2) = -4, \quad f(3) = 22,$$

además, f es continua en el intervalo $[2, 3]$, luego se puede iniciar el método de bisección. Hay por lo menos una raíz entre 2 y 3.

El punto medio es $m = 2.5$, $f(2.5) = -2.531$, $f(a)f(m) > 0$, luego el nuevo intervalo será $[2.5, 3]$.

El punto medio es $m = 2.75$, $f(2.75) = 5.202$, $f(a)f(m) < 0$, luego el nuevo intervalo será $[2.5, 2.75]$.

El punto medio es $m = 2.625$, $f(2.625) = 0.445$, $f(a)f(m) < 0$, luego el nuevo intervalo será $[2.5, 2.625]$.

El punto medio es $m = 2.562$, $f(2.562) = -1.239$, $f(a)f(m) > 0$, luego el nuevo intervalo será $[2.562, 2.625]$.

El punto medio es $m = 2.594$, $f(2.594) = -0.449$, $f(a)f(m) > 0$, luego el nuevo intervalo será $[2.594, 2.625]$.

Los valores anteriores y otros más están en la siguiente tabla. El proceso se detuvo cuando el tamaño del intervalo fue menor que 0.0005 .

k	a	b	$f(a)$	$f(b)$	m	$f(m)$
0	2.000	3.000	-4.000	22.000	2.500	-2.531
1	2.500	3.000	-2.531	22.000	2.750	5.202
2	2.500	2.750	-2.531	5.202	2.625	0.445
3	2.500	2.625	-2.531	0.445	2.562	-1.239
4	2.562	2.625	-1.239	0.445	2.594	-0.449
5	2.594	2.625	-0.449	0.445	2.609	-0.016
6	2.609	2.625	-0.016	0.445	2.617	0.211
7	2.609	2.617	-0.016	0.211	2.613	0.097
8	2.609	2.613	-0.016	0.097	2.611	0.040
9	2.609	2.611	-0.016	0.040	2.610	0.012
10	2.609	2.610	-0.016	0.012	2.610	-0.002
11	2.610	2.610				

También se puede usar la función `getchar()`, de C estándar, con archivo de cabecera `stdio.h`. Sin embargo, algunos autores no son muy partidarios de su uso con compiladores como Turbo C. Ver [Sch92].

```
void pausa3(void)
{
    int c;

    printf(" Oprima ENTER para continuar. ");
    c = getchar();
    if( c*c >= 0 ) printf(" ");
}
```

La última línea de la función es realmente tonta, ya que la condición siempre es cierta. Si no existiera, el compilador haría una advertencia (*warning*) indicando que a `c` se le asigna un valor y no se usa para nada.

Una función de pausa puede ser útil para hacer paradas en programas que producen bastantes resultados en pantalla, pues los resultados desfilan rápidamente por la pantalla y el usuario no los alcanza a leer. También es útil cuando se usan ambientes integrados de edición, compilación y ejecución. En estos casos, la mayor parte del tiempo el editor está activo; mediante un botón se activa la ejecución y, acabada la ejecución del programa, aparece de nuevo el editor y el usuario no alcanza a ver los resultados del programa.

En C, como en casi todos o tal vez todos los lenguajes de programación, una función puede llamar a otra y a su vez ésta a otra, etc.

El siguiente programa permite hallar una raíz de la ecuación $f(x) = 0$, mediante el método de la bisección o de dicotomía. El método es muy sencillo y seguro. Si se conocen a y b , $a < b$, tales que f es continua en $[a, b]$ y $f(a)f(b) < 0$ (f cambia de signo por lo menos una vez en $[a, b]$), entonces f tiene por lo menos una raíz en $[a, b]$. Partiendo de este intervalo se calcula m , el punto medio, y se tienen tres posibilidades excluyentes:

$f(m) = 0$, en este caso m es una raíz;

$f(a)f(m) < 0$, en este caso hay una raíz en el intervalo $[a, m]$;

$f(a)f(m) > 0$, en este caso hay una raíz en el intervalo $[m, b]$.

Entonces $x = 2.610$ es una raíz (aproximadamente).

```
// Calcula una raiz de la funcion definida en f
// por el metodo de biseccion
//-----
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//-----
double f( double x);
double bisec0(double a, double b, double eps);
//=====
int main()
{
    double a, b, raiz;

    printf("\n Calcula una solucion de f(x) = 0\n");
    printf("Se empieza con a, b tales que f(a)f(b) < 0\n");
    do{
        printf(" Entre los valores a b : ");
        scanf( "%lf%lf", &a, &b);
    } while ( a >= b || f(a)*f(b) >= 0.0 );
    printf("\n raiz = %lf\n", bisec0(a, b, 0.000001) );
    return 0;
}
//=====
double f( double x )
{
    // f(x) = x^5 - 3 x^4 + 10 x - 8

    return x*x*x*x*x - 3.0*x*x*x*x*x + 10.0*x - 8.0;
}
//-----
double bisec0(double a, double b, double eps)
{
    double fa, fb, m, fm;

    fa = f(a);
    fb = f(b);
```

```

while( b-a > eps){
    m = (a+b)/2.0;
    fm = f(m);
    if( fm == 0.0 ) return m;
    if( fa*fm < 0.0 ){
        // nuevo intervalo  [a,m]
        b = m;
        fb = fm;
    }
    else{
        // nuevo intervalo  [m,b]
        a = m;
        fa = fm;
    }
}
if( fabs(fa) <= fabs(fb) ) return a;
else return b;
}

```

Con respecto al programa anterior y a las funciones, es conveniente hacer varios comentarios. El programa halla una raíz de la función definida en *f*. Si se desea otra función, es necesario modificar el programa fuente y compilar de nuevo.

La función *bisec0* no controla si los parámetros están bien o mal. Tal como aparece, la función supone que los parámetros están bien. Pero es mejor que la función *bisec0* controle que los parámetros estén bien:

$$a < b,$$

$$f(a)f(b) < 0,$$

$$\text{eps} > 0 \text{ pero no muy grande.}$$

Además, desde el sitio donde se hace el llamado a *bisec0*, se debe saber si hubo errores y de qué clase. Más adelante se verá, utilizando parámetros por referencia, que una función puede devolver varios valores y así sobrepasar estos inconvenientes.

Ni en el programa principal ni en la función *bisec0* se consideró el caso en que $f(a)f(b) = 0$, caso en que no es necesario continuar la búsqueda de una raíz puesto que *a* o *b* son raíces.

El programa principal llama de manera adecuada la función `bisec0` pero se podrían presentar casos problemáticos, por ejemplo, si en la evaluación de f fuera necesario calcular e^t para algún valor muy grande de t , se produciría *overflow*.

El método de la bisección es seguro para funciones continuas. El programador o el usuario deben verificar que la función definida en `f` sea realmente continua. El programa mismo no puede controlar la continuidad.

4.2 Funciones recurrentes

En C una función puede llamarse a sí misma, se dice entonces que la función es recurrente, a veces se usa el anglicismo “recursiva” (proveniente de *recursive*). Esto puede ser una solución elegante pero debe usarse con mucho cuidado. Un programa que usa funciones recurrentes puede ser ineficiente (más lento). Además la recurrencia crea una pila (*stack*), que puede ser muy grande si la recurrencia es muy profunda.

Para enteros no negativos, el factorial puede definirse de manera recurrente:

$$n! = \begin{cases} 1 & \text{si } n = 0, 1 \\ n(n-1)! & \text{si } n \geq 2. \end{cases}$$

A continuación aparece una versión recurrente para el cálculo de $n!$.

```
double factRec( int n )
{
    // devuelve 0 si n es negativo,
    // en los demás casos devuelve n!
    // Version recurrente.

    double i;

    if( n < 0 ){
        printf(" n = %d inadecuado.\n", n);
        return 0.0;
    }
    if( n < 2 ) return 1.0;
    else return( n*factRec(n-1) );
}
```

4.3 Parámetros por valor y por referencia

Consideré el siguiente programa y los resultados producidos por él. Aunque aparentemente la función `interc0` intercambia dos valores, finalmente no lo hace. Aunque parezca una contradicción, lo hace pero no lo hace.

```
// Funcion que intercambia dos valores pero
// no lo hace definitivamente.

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void interc0( int i, int j);
//=====
int main()
{
    int m, n;

    m = 2;
    n = 5;
    printf(" MAIN1:  m = %d ,  n = %d\n", m, n);
    interc0(m, n);
    printf(" MAIN2:  m = %d ,  n = %d\n", m, n);
    return 0;
}
//=====
void interc0( int i, int j)
{
    int t;

    printf(" FUNC1:  i = %d ,  j = %d\n", i, j);
    t = i;
    i = j;
    j = t;
    printf(" FUNC2:  i = %d ,  j = %d\n", i, j);
}
```

Los resultados son:

```
MAIN1: m = 2 , n = 5
FUNC1: i = 2 , j = 5
FUNC2: i = 5 , j = 2
MAIN2: m = 2 , n = 5
```

Las tres primeras líneas concuerdan con lo esperado. Lo único raro es la cuarta línea: los valores de `m` y de `n`, después del llamado de la función, no han sido intercambiados. Sin embargo, en la tercera línea de los resultados, se ve que en la función hubo intercambio. Entonces, ¿dónde está el error?

La razón es sencilla. En C y C++, por defecto, los **parámetros** se pasan **por valor**. Esto quiere decir que, al hacer el llamado a la función, se manda una copia de los valores de los parámetros. Entonces la función `interc0` intercambia una copia de los valores de los parámetros, pero no intercambia los originales.

Si se desea mandar el parámetro “original”, esto se llama **parámetros por referencia**. En C se hace como en el siguiente ejemplo.

```
// Funcion que intercambia dos valores.
// Version a la manera de C.
//-----
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//-----
void interc1( int *i, int *j);
//=====
int main()
{
    int m, n;

    m = 2;
    n = 5;
    printf(" MAIN1: m = %d , n = %d\n", m, n);
    interc1(&m, &n);
    printf(" MAIN2: m = %d , n = %d\n", m, n);
    return 0;
}
```

```

}

//=====
void interc1( int *i, int *j)
{
    int t;

    t = *i;
    *i = *j;
    *j = t;
}

```

En C++ el paso de parámetros por referencia es mucho más sencillo. Por eso a continuación hay únicamente explicaciones muy someras sobre la forma de pasar parámetros por referencia en C:

- En el llamado a la función, el parámetro pasado por referencia debe ir precedido de &; más adelante, cuando se vea el tema de apuntadores, el lector entenderá que eso significa que se pasa la dirección del parámetro.
- En la definición de la función, el parámetro pasado por referencia debe estar precedido de un asterisco. Esto indica (tema posterior) que se trata de un apuntador.

A continuación se presenta el ejemplo análogo, a la manera de C++.

```

// Funcion que intercambia dos valores.
// Version a la manera de C++.

```

```

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void interc( int &i, int &j);
//=====

int main()
{
    int m, n;

```

```

m = 2;
n = 5;
printf(" MAIN1: m = %d , n = %d\n", m, n);
interc(m, n);
printf(" MAIN2: m = %d , n = %d\n", m, n);
return 0;
}
//=====================================================================
void interc( int &i, int &j)
{
    int t;

    t = i;
    i = j;
    j = t;
}

```

Observe lo sencillo que es pasar parámetros por referencia en C++. Basta con colocar el signo `&` precediendo al parámetro en la primera línea de definición de la función. Consecuentemente debe hacerse lo mismo donde está el prototipo de la función.

En la función del siguiente ejemplo se utiliza el método de bisección visto anteriormente. Hay algunas diferencias con la función `bisec0`. La función `bisec` devuelve un valor que posiblemente corresponde a la raíz; además, en un parámetro por referencia, se tiene información sobre el buen o mal desarrollo del proceso.

```

double bisec(double a, double b, int &indic,
             double epsX, double epsF)
{
    // Metodo de biseccion para f(x) = 0.
    // f debe ser continua en [a,b] y f(a)f(b) < 0.

    // Esta funcion calcula dos valores: r , indic
    // Devuelve r, posiblemente una raiz.
    // indic     es un parametro por referencia.

    // indic valdra:
    //           1   si se obtuvo una raiz,

```

```

//          0   si f(a)f(b) > 0.0
//          -1  si a >= b
//          -2  si epsX  <= 0  o  epsF < 0.0

// Si indic = 1, el valor devuelto sera una raiz.
// En los demas casos el valor devuelto sera  0.0
// Sea x una raiz exacta; r es raiz (aproximada)
// si |f(r)| <= epsF  o  si  |r-x| <= epsX

double fa, fb, m, fm;

if( epsX <= 0.0 || epsF < 0.0 ){
    indic = -2;
    return 0.0;
}
fa = f(a);
fb = f(b);
if( fabs(fa) <= epsF ){
    indic = 1;
    return a;
}
if( fabs(fb) <= epsF ){
    indic = 1;
    return b;
}
if( a >= b ){
    indic = -1;
    return 0.0;
}
if( fa*fb > 0.0 ){
    indic = 0;
    return 0.0;
}
while( b-a > epsX){
    m = (a+b)/2.0;
    fm = f(m);
    if( fabs(fm) <= epsF ){
        indic = 1;
    }
}

```

```

        return m;
    }
    if( fa*fm < 0.0 ){
        // nuevo intervalo [a,m]
        b = m;
        fb = fm;
    }
    else{
        // nuevo intervalo [m,b]
        a = m;
        fa = fm;
    }
}
indic = 1;
if( fabs(fa) <= fabs(fb) ) return a;
else return b;
}

```

El llamado a la función `bisec` puede ser de la forma siguiente:

```

raiz = bisec(a, b, result, 1.0e-4, 1.0e-6);
if( result == 1 ) printf("\n raiz = %15.10lf\n", raiz);
else printf("result=%d: ERROR en llamado a bisec\n",
           result);

```

4.4 Parámetros por defecto

En C++ es posible pasar parámetros por defecto. Esto quiere decir que en el prototipo de la función se define para el último parámetro (o los últimos parámetros) un valor predeterminado. El llamado de la función se puede hacer con un valor específico o utilizar el valor predeterminado. En el siguiente ejemplo se retoma la función `bisec` y se definen los dos últimos parámetros por defecto.

```

double bisec(double a, double b, int &indic,
             double epsX = 1.0E-6, double epsF = 1.0E-8);
int main()
{
    ...

```

```

}

//-----
int bisec(double a, double b, double &raiz, double epsX,
          double epsF)
{
    ...
    ...
    ...
}

```

Si se hace el llamado `bisec(a, b, ind, 1.0E-9, 1.0e-10)`, entonces la función trabajará con esos dos valores pasados explícitamente, es decir, `epsX = 1.0E-9`, y `epsF = 1.0E-10`.

Si se hace el llamado `bisec(a, b, ind, 1.0E-9)`, entonces la función tomará para el parámetro faltante, el último, el valor predeterminado, o sea, `epsX = 1.0E-9, epsF = 1.0E-8`.

Si se hace el llamado `bisec(a, b, ind)`, entonces la función tomará para los parámetros faltantes, los dos últimos, los dos valores predeterminados, o sea, `epsX = 1.0E-6, epsF = 1.0E-8`.

4.5 Variables locales y variables globales

En los ejemplos de programas vistos hasta ahora, todas las variables usadas eran propias a cada función (recuerde que `main` también es una función), es decir, son **variables locales**. También se dice que tienen **ámbito local**. Si hay variables con el mismo nombre en diferentes funciones, son cosas completamente distintas.

A veces es conveniente tener variables que se puedan usar en varias funciones; en realidad se podrán usar en todas las funciones. Estas variables se llaman **globales**, pues tienen ámbito global. Esto va un poco en contravía de la filosofía de las funciones de tipo general, que deben usar únicamente variables locales. Sin embargo, algunas veces puede ser útil. Para esto basta con declararlas antes de todas las funciones.

```

double PI = 3.141592654, RADIANT = 57.29577951;
int nLlamadasF;

... func1( ... );

```

```

int main(void)
{
    ...
    ...
    nLlamadasF = 0;
    ...
    ... = sin(t/RADIAN);
}

//-----
... func1( ... )
{
    ...
    nLlamadasF++;
    ...
    ... = 2.0*PI;
}

```

En el esquema anterior, los valores de las variables PI y RADIAN pueden ser utilizados en cualquier función, sin tener que redefinirlos en cada función. La variable `nLlamadasF` permite saber, en cualquier parte, el número de veces que la función `func1` ha sido llamada. En algunos métodos numéricos, la eficiencia se mide por el número de veces que se realice una operación (sumas o multiplicaciones) o por el número de veces que se haga un llamado a una función.

4.6 Sobrecarga de funciones

En C, dos funciones no pueden tener el mismo nombre. En C++ está permitido; esto se llama la sobrecarga de funciones. Pero, para poder reconocer una u otra función, se necesita que el número de parámetros sea diferente o que los tipos de los parámetros sean diferentes. Dicho de otra forma, en C++ no puede haber dos funciones con el mismo nombre, con el mismo número de parámetros y con el mismo tipo de parámetros. El siguiente ejemplo es completamente lícito.

```

double maximo( double a, double b);
double maximo( double a, double b, double c);

```

```
int maximo( int a, int b);
int maximo( int a, int b, int c);
//=====
int main(void)
{
    double x, y, z;
    int i, j;
    ...
    ... = max(x, y);
    ... = max(i, j);
    ... = max(x, y, z);
    ... = max(i, j, 4);
    ...
}
//=====
double maximo( double a, double b)
{
    if( a >= b ) return a;
    else return b;
}
//-----
double maximo( double a, double b, double c)
{
    if( a >= b && a >= c) return a;
    if( b >= c ) return b;
    else return c;
}
//-----
int maximo( int a, int b)
{
    if( a >= b ) return a;
    else return b;
}
//-----
int maximo( int a, int b, int c)
{
    if( a >= b && a >= c) return a;
    if( b >= c ) return b;
```

```

    else return c;
}

```

En cambio, el siguiente ejemplo no está permitido:

```

int factorial( int n );
double factorial( int n );
//=====
int main(void)
{
    ...
}

//=====
int factorial( int n )
{
    ...
}

//-----
double factorial( int n )
{
    ...
}

```

4.7 Biblioteca estándar

alloc.h	Asignación dinámica de memoria.
complex.h	Números complejos. C++. No es estándar.
ctype.h	Manejo de caracteres.
errno.h	Códigos de error.
float.h	Definición de constantes para punto flotante.
iostream.h	Flujo de entrada y salida. C++.
limits.h	Límites de enteros, depende de la implementación.
math.h	Matemáticas.
stddef.h	Constantes de uso común.
stdio.h	Entrada y salida (Input/Output) estándar.
stdlib.h	Declaraciones varias.
string.h	Manejo de cadenas de caracteres.
time.h	Funciones de tiempo del sistema.

El estándar ANSI C provee varias funciones listas para usar. También hay funciones exclusivas de C++. Los principales archivos de cabecera para funciones y valores predeterminados de ANSI C y de C++ son los de la tabla anterior.

Dependiendo del compilador, existen muchos más archivos de cabecera para C++. En [DeD99] hay una lista bastante completa de archivos de cabecera para C++, los cuales no se utilizan en este libro.

Ejercicios

Para cada uno de los enunciados siguientes, defina cuáles son los datos necesarios. Haga un programa que lea los datos, llame la **función que realiza los cálculos y devuelve los resultados** y, finalmente, en la función `main`, muestre los resultados. La función debe analizar los parámetros, estudiar su consistencia o inconsistencia e informar sobre ello por medio de los resultados devueltos. También, cuando se justifica, haga una función para la lectura de los datos. Puede ser útil, cuando el proceso es bastante complejo, hacer funciones más simples que hacen partes específicas del proceso y son llamadas desde otra función que coordina los cálculos.

- 4.1 La gran mayoría de los ejercicios de los dos capítulos anteriores pueden ser resueltos por medio de funciones.
- 4.2 Dado un número x y un entero no negativo n , calcule x^n . Hágalo mediante un `for`. Verifique comparando con el resultado producido por la función `pow`.
- 4.3 Dado un número x y un entero no negativo n , calcule x^n . Hágalo mediante una función recurrente, $x^n = x \cdot x^{n-1}$.
- 4.4 Dado un número x y un entero no negativo n , calcule x^n , tratando de efectuar pocas multiplicaciones. Por ejemplo para calcular x^8 se requiere calcular $x^2 = xx$, $x^4 = x^2x^2$ y finalmente $x^8 = x^4x^4$, o sea, tres multiplicaciones. El cálculo de x^6 se puede hacer con tres multiplicaciones. Aparentemente, para calcular x^7 es necesario hacer cuatro multiplicaciones. Además de calcular x^n , muestre el número de multiplicaciones realizadas entre números `double`.

5

Arreglos

Los arreglos (*arrays*) permiten almacenar vectores y matrices. Los **arreglos unidimensionales** sirven para manejar vectores y los **arreglos bidimensionales** para matrices. Sin embargo, las matrices también se pueden almacenar mediante arreglos unidimensionales y por medio de apuntadores a apuntadores, temas que se verán en el capítulo siguiente.

La palabra unidimensional no indica que se trata de vectores en espacios de dimensión uno; indica que su manejo se hace mediante un subíndice. El manejo de los arreglos bidimensionales se hace mediante dos subíndices.

5.1 Arreglos unidimensionales

El siguiente ejemplo muestra la definición de tres arreglos, uno de 80 elementos doble precisión, otro de 30 elementos enteros y uno de 20 elementos tipo carácter.

```
double x[80];  
int factores[30];  
char codSexo[20];
```

Los nombres deben cumplir con las normas para los identificadores. La primera línea indica que se han reservado 80 posiciones para números doble precisión. Estas posiciones son contiguas. Es importante recalcar

que en C, a diferencia de otros lenguajes, el primer elemento es `x[0]`, el segundo es `x[1]`, el tercero es `x[2]`, y así sucesivamente; el último elemento es `x[79]`.

En `x` hay espacio reservado para 80 elementos, pero esto no obliga a trabajar con los 80; el programa puede utilizar menos de 80 elementos.

C no controla si los subíndices están fuera del rango previsto; esto es responsabilidad del programador. Por ejemplo, si en algún momento el programa debe utilizar `x[90]`, lo usa sin importar si los resultados son catastróficos.

Cuando un parámetro de una función es un arreglo, se considera implícitamente que es un parámetro por referencia. O sea, si en la función se modifica algún elemento del arreglo, entonces se modificó realmente el valor original y no una copia. Pasar un arreglo como parámetro de una función y llamar esta función es muy sencillo. Se hace como en el esquema siguiente.

```
... funcion(..., double x[], ...); // prototipo
//-----
int main(void)
{
    double v[30];
    ...
    ... funcion(..., v, ...); // llamado a la funcion
    ...
}
//-----
... funcion(..., double x[],...)// definicion de la funcion
{
    // cuerpo de la funcion
    ...
}
```

En el esquema anterior, el llamado a la función se hizo desde la función `main`. Esto no es ninguna obligación; el llamado se puede hacer desde cualquier función donde se define un arreglo o donde a su vez llega un arreglo como parámetro.

También se puede hacer el paso de un arreglo como parámetro de la siguiente manera. Es la forma más usual. Tiene involucrada la noción de apuntador que se verá en el siguiente capítulo.

```

... funcion(..., double *x, ...); // prototipo
//-----
int main(void)
{
    double v[30];
    ...
    ... funcion(..., v, ...); // llamado a la funcion
    ...
}
//-----
... funcion(..., double *x, ...)// definicion de la funcion
{
    // cuerpo de la funcion
    ...
}

```

El programa del siguiente ejemplo lee el tamaño de un vector, lee los elementos del vector, los escribe y halla el promedio. Para esto utiliza funciones. Observe la manera como un arreglo se pasa como parámetro.

```

// Arreglos unidimensionales
// Lectura y escritura de un vector y calculo del promedio
//-----
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//-----
void lectX(double *x, int n, char c );
void escrX(double *x, int n );
double promX( double *x, int n );
//=====
int main()
{
    double v[40];
    int n;

    printf("\n Promedio de elementos de un vector.\n\n");

```

```
printf(" numero de elementos : ");
scanf( "%d", &n);
if( n > 40 ){
    printf("\n Numero demasiado grande\n\n");
    exit(1);
}
lectX(v, n, 'v');
printf(" v : \n");
escrX(v, n);
printf(" promedio = %lf\n", promX(v, n));
return 0;
}
//=====
void lectX(double *x, int n, char c )
{
    // lectura de los elementos de un "vector".
    int i;

    for( i = 0; i < n; i++){
        printf(" %c(%d) = ", c, i+1);
        scanf("%lf", &x[i] );
    }
}
//=====
void escrX(double *x, int n )
{
    // escritura de los elementos de un "vector".

    int i;
    int nEltosLin = 5; // numero de elementos por linea

    for( i = 0; i < n; i++){
        printf("%15.8lf", x[i]);
        if( (i+1)%nEltosLin == 0 || i == n-1) printf("\n");
    }
}
//=====
```

```

double promX( double *x, int n)
{
    // promedio de los elementos del 'vector' x

    int i;
    double s = 0.0;

    if( n <= 0 ){
        printf(" promX: n = %d inadecuado\n", n);
        return 0.0;
    }
    for( i = 0; i < n; i++) s += x[i];
    return s/n;
}

```

La función `lectX` tiene tres parámetros: el arreglo, el número de elementos y una letra. Esta letra sirve para el pequeño aviso que sale antes de la lectura de cada elemento. En el ejemplo, cuando se “llama” la función, el tercer parámetro es ‘v’; entonces en la ejecución aparecerán los avisos:

```

v(1) =
v(2) =
...

```

Observe que en el `printf` de la función `lectX` aparece `i+1`; entonces para el usuario el “vector” empieza en 1 y acaba en n . Internamente empieza en 0 y acaba en $n - 1$.

Es importante anotar que si durante la entrada de datos hay errores, es necesario volver a empezar para corregir. Suponga que $n = 50$, que el usuario ha entrado correctamente 40 datos, que en el dato cuadragésimo primero el usuario digitó mal algo y después oprimió la tecla Enter. Ya no puede corregir. Sólo le queda acabar de entrar datos o abortar el programa (parada forzada del programa desde el sistema operativo) y volver a empezar. Esto sugiere que es más seguro hacer que el programa lea los datos en un archivo. La entrada y salida con archivos se verá en un capítulo posterior.

Cuando un arreglo unidimensional es parámetro de una función, no importa que el arreglo haya sido declarado de 1000 elementos y se trabaje

con 20 o que haya sido declarado de 10 y se trabaje con 10. La función es de uso general siempre y cuando se controle que no va a ser llamada para usarla con subíndices mayores que los previstos. En la siguiente sección se trata el tema de los arreglos bidimensionales. Allí, el paso de parámetros no permite que la función sea completamente general.

En el siguiente ejemplo, dado un entero $n \geq 2$ (pero no demasiado grande), el programa imprime los factores primos. El algoritmo es muy sencillo. Se busca $d > 1$, el divisor más pequeño de n . Este divisor es necesariamente un primo. Se divide n por d y se continúa el proceso con el último cociente. El proceso termina cuando el cociente es 1. Si $n = 45$, el primer divisor es 3. El cociente es 15. El primer divisor de 15 es 3. El cociente es 5. El primer divisor de 5 es 5 y el cociente es 1.

```
// Arreglos unidimensionales
// Factores primos de un entero >= 2
//-----
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//-----
int primerDiv( int n );
int factoresPrimos( int n, int *fp, int &nf, int nfMax );
//=====
int main()
{
    int vFactPrim[40]; // vector con los factores primos
    int n;
    int nFact;        // numero de factores primos
    int i;

    printf("\n Factores primos de un entero >= 2.\n\n");

    printf(" n = ");
    scanf( "%d", &n );
    if( factoresPrimos(n, vFactPrim, nFact, 40 ) ){
        for(i = 0; i < nFact; i++) printf(" %d",
            vFactPrim[i]);
        printf("\n");
    }
}
```

```

    else printf(" ERROR\n");
    return 0;
}
//=====================================================================
int primerDiv( int n)
{
    // n debe ser mayor o igual a 2.
    // Calcula el primer divisor, mayor que 1, de n
    // Si n es primo, devuelve n.
    // Si hay error, devuelve 0.

    int i;

    if( n < 2 ){
        printf(" primerDiv: %d inadecuado.\n", n);
        return 0;
    }
    for( i = 2; i*i <= n; i++) if( n%i == 0 ) return i;
    return n;
}
//=====================================================================
int factoresPrimos( int n, int *fp, int &nf, int nfMax)
{
    // factores primos de n
    // devuelve 0 si hay error.
    // devuelve 1 si todo esta bien.
    // fp : vector con los factores primos
    // nf : numero de factores primos
    // nfMax : tamano del vector fp

    int d, indic;

    if( n < 2 ){
        printf(" factoresPrimos: %d inadecuado.\n", n);
        return 0;
    }
    nf = 0;
    do{

```

```

if( nf >= nfMax ){
    printf("factoresPrimos: demasiados factores.\n");
    return 0;
}
d = primerDiv(n);
fp[nf] = d;
nf++;
n /= d;
} while( n > 1 );
return 1;
}

```

5.2 Arreglos multidimensionales

La declaración de los arreglos bidimensionales, caso particular de los arreglos multidimensionales, se hace como en el siguiente ejemplo:

```

double a[3][4];
int pos[10][40];
char list[25][25];

```

En la primera línea se reserva espacio para $3 \times 4 = 12$ elementos doble precisión. El primer subíndice varía entre 0 y 2, y el segundo varía entre 0 y 3. Usualmente, de manera análoga a las matrices, se dice que el primer subíndice indica la fila y el segundo subíndice indica la columna. Un arreglo tridimensional se declararía así:

```
double c[20][30][10];
```

Los sitios para los elementos de *a* están contiguos en el orden fila por fila, o sea, *a*[0][0], *a*[0][1], *a*[0][2], *a*[0][3], *a*[1][0], *a*[1][1], *a*[1][2], *a*[1][3], *a*[2][0], *a*[2][1], *a*[2][2], *a*[2][3].

En el siguiente ejemplo, el programa sirve para leer matrices, escriirlas y calcular el producto. Lo hace mediante la utilización de funciones que tienen como parámetros arreglos bidimensionales.

```
// prog14
// Arreglos bidimensionales
```

```

// Lectura y escritura de 2 matrices y calculo del producto
//-----
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//-----
void lectA0(double a[][][40], int m, int n, char c );
void escrA0(double a[][][40], int m, int n );
int prodAB0(double a[][][40], int m, int n, double b[][][40],
            int p, int q, double c[][][40]);
//=====
int main()
{
    double a[50][40], b[20][40], c[60][40];
    int m, n, p, q;

    printf("\n Producto de dos matrices.\n\n");

    printf(" num. de filas de A : ");
    scanf( "%d", &m);
    printf(" num. de columnas de A : ");
    scanf( "%d", &n);
    // es necesario controlar que m, n no son muy grandes
    // ni negativos

    printf(" num. de filas de B : ");
    scanf( "%d", &p);
    printf(" num. de columnas de B : ");
    scanf( "%d", &q);
    // es necesario controlar que p, q no son muy grandes
    // ni negativos

    if( n != p ){
        printf(" Producto imposible\n");
        exit(1);
    }
    lectA0(a, m, n, 'A');
    printf(" A : \n");

```

```

escriA0(a, m, n);

lectA0(b, n, q, 'B');
printf(" B : \n");
escriA0(b, n, q);

if( prodAB0(a,m,n, b,p,q, c) ){
    printf(" C : \n");
    escriA0(c, m, q);
}
else printf("\ ERROR\n");
return 0;
}

//=====
void lectA0(double a[][][40], int m, int n, char c )
{
    // lectura de los elementos de una matriz.

    int i, j;

    for( i = 0; i < m; i++){
        for( j=0; j < n; j++){
            printf(" %c[%d] [%d] = ", c, i+1, j+1);
            scanf("%lf", &a[i][j] );
        }
    }
}

//=====
void escriA0(double a[][][40], int m, int n )
{
    // escritura de los elementos de una matriz

    int i, j;
    int nEltosLin = 5; // numero de elementos por linea

    for( i = 0; i < m; i++){
        for( j = 0; j < n; j++){
            printf("%15.8lf", a[i][j]);
        }
    }
}

```

```

        if((j+1)%nEltosLin == 0 || j==n-1)printf("\n");
    }
}
//-----
int prodAB0(double a[][][40], int m, int n, double b[][][40],
            int p, int q, double c[][][40])
{
    // producto de dos matrices, a  mxn,  b  pxq
    // devuelve  1  si se puede hacer el producto
    // devuelve  0  si no se puede

    int i, j, k;
    double s;

    if(m<0||n<0||p<0||q<0 || n!= p ) return 0;
    for( i=0; i < m; i++){
        for( j=0; j < q; j++){
            s = 0.0;
            for( k=0; k<n; k++) s += a[i][k]*b[k][j];
            c[i][j] = s;
        }
    }
    return 1;
}

```

Cuando en una función un parámetro es un arreglo bidimensional, la función debe saber, en su definición, el número de columnas del arreglo bidimensional. Por eso en la definición de las funciones está `a[][][40]`. Esto hace que las funciones del ejemplo sirvan únicamente para arreglos bidimensionales definidos con 40 columnas. Entonces estas funciones no son de uso general. Este inconveniente se puede resolver de dos maneras:

- Mediante apunadores y apunadores dobles. Este tema se verá en el siguiente capítulo.
- Almacenando las matrices en arreglos unidimensionales con la convención de que los primeros elementos del arreglo corresponden a la primera fila de la matriz, los que siguen corresponden a la segunda fila, y así sucesivamente. Esta modalidad es muy usada, tiene

algunas ventajas muy importantes. Se verá con más detalle más adelante.

En resumen, los arreglos bidimensionales no son muy adecuados para pasarlos como parámetros a funciones. Su uso debería restringirse a casos en que el arreglo bidimensional se usa únicamente en la función donde se define.

En el ejemplo anterior, en la función `lectA0`, antes de la lectura del elemento `a[i][j]`, el programa escribe los valores `i+1` y `j+1`, entonces para el usuario el primer subíndice empieza en 1 y acaba en `m`; el segundo empieza en 1 y acaba en `n`.

5.3 Cadenas

Los arreglos unidimensionales de caracteres, además de su manejo estándar como arreglo, pueden ser utilizados como cadenas de caracteres, siempre y cuando uno de los elementos del arreglo indique el fin de la cadena. Esto se hace mediante el carácter especial

`'\0'`

En el ejemplo

```
// Arreglo de caracteres como tal.
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char aviso[30];
    int i;

    aviso[0] = 'C';
    aviso[1] = 'o';
    aviso[2] = 'm';
    aviso[3] = 'o';
    aviso[4] = ' ';
    aviso[5] = 'e';
    aviso[6] = 's';
```

```

aviso[7] = 't';
aviso[8] = 'a';
aviso[9] = '?';
for(i=0; i<= 9; i++) printf("%c", aviso[i]);
return 0;
}

```

el arreglo **aviso** se consideró como un simple arreglo de caracteres. El programa escribe

Como esta?

En el siguiente ejemplo, el arreglo **aviso** es (o contiene) una cadena, *string*, pues hay un fin de cadena. Para la escritura se usa el formato **%s**. El resultado es el mismo.

```

// prog15b
// Cadena de caracteres
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char aviso[30];

    aviso[0] = 'C';
    aviso[1] = 'o';
    aviso[2] = 'm';
    aviso[3] = 'o';
    aviso[4] = ' ';
    aviso[5] = 'e';
    aviso[6] = 's';
    aviso[7] = 't';
    aviso[8] = 'a';
    aviso[9] = '?';
    aviso[10] ='\'\0';
    printf("%s", aviso);
    return 0;
}

```

Si se modifica ligeramente de la siguiente manera:

```
char aviso[30];

aviso[0] = 'C';
aviso[1] = 'o';
aviso[2] = 'm';
aviso[3] = 'o';
aviso[4] = '\0';
aviso[5] = 'e';
aviso[6] = 's';
aviso[7] = 't';
aviso[8] = 'a';
aviso[9] = '?';
aviso[10] = '\0';
printf("%s", aviso);
```

entonces únicamente escribe **Como**, ya que encuentra el fin de cadena (el primero) después de la segunda letra o.

La lectura de cadenas de hace mediante la función `gets()`. Su archivo de cabecera es `stdio.h`. Su único parámetro es precisamente la cadena que se desea leer.

```
char nombre[81];

printf(" Por favor, escriba su nombre : ");
gets(nombre);
printf("\n Buenos dias %s\n", nombre);
```

En C++ se puede utilizar `cin` para leer cadenas que no contengan espacios. Cuando hay un espacio, es reemplazado por fin de cadena.

```
char nombre[81];

cout<<" Por favor, escriba su nombre : ";
cin>>nombre;
cout<<endl<<" Buenos dias "<<nombre<<endl;
```

Si el usuario escribe **Juanito**, el programa (la parte de programa) anterior escribirá **Buenos dias Juanito**. Pero si el usuario escribe el nombre de dos palabras **Juan Manuel**, el programa escribirá **Buenos dias Juan**.

En C++ es posible leer cadenas de caracteres con espacios mediante `cin.getline()`. Este tema no se trata en este libro.

Para tener acceso a las funciones para el manejo de cadenas, se necesita el archivo de cabecera `string.h`. Las funciones más usuales son:

```
strcpy( , )
strcat( , )
strlen( )
```

El primer parámetro de `strcpy` (*string copy*) debe ser un arreglo de caracteres. El segundo parámetro debe ser una cadena constante o una cadena en un arreglo de caracteres. La función copia en el arreglo (primer parámetro) la cadena (el segundo parámetro). Se presentan problemas si el segundo parámetro no cabe en el primero. En un manual de referencia de C puede encontrarse información más detallada sobre estas y otras funciones relacionadas con las cadenas.

La función `strlen` (*string length*) da como resultado la longitud de la cadena sin incluir el fin de cadena.

Una cadena constante es una sucesión de caracteres delimitada por dos comillas dobles; por ejemplo: "Hola". No necesita explícitamente el signo de fin de cadena, ya que C lo coloca implícitamente. La cadena constante más sencilla es la cadena vacía: "". La cadena constante de un solo carácter es diferente del carácter. Por ejemplo, "x" es diferente de 'x'.

El programa `prog15b` se puede escribir más rápidamente así:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char aviso[30];

    strcpy(aviso, "Como esta?");
    printf("%s", aviso);
    printf("\n longitud = %d\n", strlen(aviso));
```

```
    return 0;
}
```

Como era de esperarse, el programa anterior escribe `Como esta?` y en la línea siguiente `longitud = 10`. Efectivamente las diez primeras posiciones del arreglo `aviso`, de la 0 a la 9, están ocupadas. La posición 10 está ocupada con el fin de cadena. El arreglo `aviso` puede contener cadenas de longitud menor o igual a 29, pues se necesita un elemento para el signo de fin de cadena.

La función `strcat` sirve para concatenar dos cadenas. El primer parámetro de `strcat` debe ser una cadena en un arreglo de caracteres. El segundo parámetro debe ser una cadena constante o una cadena en un arreglo de caracteres. La función pega la segunda cadena a la derecha de la primera cadena. Aparecen problemas si en el primer arreglo no cabe la concatenación de la primera y la segunda cadenas. La concatenación se hace de manera limpia: la función quita el fin de cadena en el primer arreglo y pega la segunda incluyendo su fin de cadena.

```
// funcion strcat

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char nombre[41], apell[41], Nombre[81];

    printf(" Por favor, escriba su nombre : ");
    gets(nombre);
    printf(" Por favor, escriba su apellido : ");
    gets(apell);
    strcpy(Nombre, nombre);
    strcat(Nombre, " ");
    strcat(Nombre, apell);
    printf("Buenos dias %s\n", Nombre);
    return 0;
}
```

5.4 Inicialización de arreglos

Los arreglos pequeños se pueden inicializar de la siguiente manera:

```
double x[4] = { 1.1, 1.2, 1.3, 1.4};
```

Esto es lo mismo que escribir:

```
double x[4];  
  
x[0] = 1.1;  
x[1] = 1.2;  
x[2] = 1.3;  
x[3] = 1.4;
```

Si dentro de los corchetes hay menos valores que el tamaño del arreglo, generalmente C asigna 0.0 a los faltantes. El ejemplo

```
double x[4] = { 1.1, 1.2};
```

produce el mismo resultado que

```
double x[4];  
  
x[0] = 1.1;  
x[1] = 1.2;  
x[2] = 0.0;  
x[3] = 0.0;
```

Si no se precisa el tamaño del arreglo en una inicialización, C le asigna el tamaño dado por el número de elementos. El ejemplo

```
double x[] = { 1.1, 1.2, 1.3};
```

es equivalente a

```
double x[3];  
  
x[0] = 1.1;  
x[1] = 1.2;  
x[2] = 1.3;
```

En este otro ejemplo, con una cadena en un arreglo de caracteres,

```
char saludo[] = "Buenos dias";
```

resulta lo mismo que escribir

```
char saludo[12] = {'B', 'u', 'e', 'n', 'o', 's', ' ', 'd', 'i', 'a', 's', '\0'};
```

o igual que escribir

```
char saludo[12];  
  
saludo[0] = 'B';  
saludo[1] = 'u';  
saludo[2] = 'e';  
saludo[3] = 'n';  
saludo[4] = 'o';  
saludo[5] = 's';  
saludo[6] = ' ';  
saludo[7] = 'd';  
saludo[8] = 'i';  
saludo[9] = 'a';  
saludo[10] = 's';  
saludo[11] = '\0';
```

Para arreglos bidimensionales, basta con recordar que primero están los elementos de la fila 0, enseguida los de la fila 1, y así sucesivamente. La inicialización

```
double a[2][3] = { 1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
```

produce el mismo resultado que

```
double a[2][3];  
  
a[0][0] = 1.1;  
a[0][1] = 1.2;  
a[0][2] = 1.3;  
a[1][0] = 1.4;  
a[1][1] = 1.5;  
a[1][2] = 1.6;
```

La siguiente inicialización también hubiera producido el mismo resultado anterior:

```
double a[][] = { 1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
```

En el ejemplo anterior, C sabe que las filas tienen tres elementos, entonces en el arreglo `a` debe haber dos filas. En el ejemplo que sigue, C asigna ceros a lo que queda faltando hasta obtener filas completas.

```
double a[][] = { 1.1, 1.2, 1.3, 1.4};
```

Lo anterior es equivalente a

```
double a[2][3];
```

```
a[0][0] = 1.1;
a[0][1] = 1.2;
a[0][2] = 1.3;
a[1][0] = 1.4;
a[1][1] = 0.0;
a[1][2] = 0.0;
```

En las siguientes inicializaciones hay errores. Para los arreglos bidimensionales, C necesita conocer el tamaño de las filas (el número de columnas).

```
double a[][] = { 1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
double b[2][] = { 1.1, 1.2, 1.3, 1.4, 1.5, 1.6};
```

Ejercicios

Para cada uno de los enunciados siguientes, defina cuáles son los datos necesarios. Elabore un programa que lea los datos, llame la función (o las funciones) que realiza los cálculos y devuelve los resultados, y finalmente, haga que el programa principal (la función `main`) muestre los resultados.

- 5.1** Intercambie los elementos de un vector: el primero pasa a la última posición y el último a la primera posición, el segundo pasa a la penúltima posición y viceversa...

- 5.2** Obtenga la expresión binaria de un entero no negativo.
- 5.3** Obtenga la expresión en base p (p entero, $2 \leq p \leq 9$), de un entero no negativo.
- 5.4** Obtenga la expresión hexadecimal de un entero no negativo.
- 5.5** Averigüe si una lista de números está ordenada de menor a mayor.
- 5.6** Averigüe si una lista de números está ordenada de manera estrictamente creciente.
- 5.7** Averigüe si una lista tiene números repetidos.
- 5.8** Ordenar, de menor a mayor, los elementos de una lista.
- 5.9** Averigüe si una lista ordenada de menor a mayor tiene números repetidos.
- 5.10** Dada una lista de n números, averigüe si el número t está en la lista.
- 5.11** Dada una lista de n números, ordenada de menor a mayor, averigüe si el número t está en la lista.
- 5.12** Halle el promedio de los elementos de un vector.
- 5.13** Halle la desviación estándar de los elementos de un vector.
- 5.14** Dado un vector con componentes (coordenadas) no negativas, halle el promedio geométrico.
- 5.15** Halle la moda de los elementos de un vector.
- 5.16** Halle la mediana de los elementos de un vector.
- 5.17** Dados un vector x de n componentes y una lista, v_1, v_2, \dots, v_m estrictamente creciente, averigüe cuantos elementos de x hay en cada uno de los $m + 1$ intervalos $(\infty, v_1]$, $(v_1, v_2]$, $(v_2, v_3]$, ..., $(v_{m-1}, v_m]$, (v_m, ∞) .
- 5.18** Dado un vector de enteros positivos, halle el m.c.d.
- 5.19** Dado un vector de enteros positivos, halle el m.c.m. (mínimo común múltiplo).

- 5.20** Dado un polinomio definido por el grado n y los $n + 1$ coeficientes, calcule el verdadero grado. Por ejemplo, si $n = 4$ y $p(x) = 5 + 0x + 6x^2 + 0x^3 + 0x^4$, su verdadero grado es 2.
- 5.21** Dado un polinomio halle su derivada.
- 5.22** Dado un polinomio p y un punto (a, b) halle su antiderivada q tal que $q(a) = b$.
- 5.23** Dados dos polinomios (pueden ser de grado diferente), halle su suma.
- 5.24** Dados dos polinomios, halle su producto.
- 5.25** Dados dos polinomios, halle el cociente y el residuo de la división.
- 5.26** Dados n puntos en \mathbb{R}^2 , P_1, P_2, \dots, P_n . Verifique que la línea poligonal cerrada $P_1P_2\dots P_nP_1$ sea de Jordan (no tiene “cruces”).
- 5.27** Dados n puntos en \mathbb{R}^2 , P_1, P_2, \dots, P_n , tal que la línea poligonal cerrada $P_1P_2\dots P_nP_1$ es de Jordan, averigüe si el polígono determinado es convexo.
- 5.28** Dados n puntos en \mathbb{R}^2 , P_1, P_2, \dots, P_n , tal que la línea poligonal cerrada $P_1P_2\dots P_nP_1$ es de Jordan, halle el área del polígono determinado.
- 5.29** Sea x un vector en \mathbb{R}^n y A una matriz de tamaño $m \times n$ definida por una lista de p triples de la forma (i_k, j_k, v_k) para indicar que $a_{i_k j_k} = v_k$ y que las demás componentes de A son nulas; calcule Ax .
- 5.30** Considere un conjunto A de n elementos enteros almacenados en un arreglo a . Considere una lista de m parejas en $A \times A$. Esta lista define una relación sobre A . Averigüe si la lista está realmente bien definida, si la relación es simétrica, antisimétrica, reflexiva, transitiva y de equivalencia.
- 5.31** Considere un conjunto A de n elementos y una matriz M de tamaño $n \times n$ que representa una operación binaria sobre A . Averigüe si la operación binaria está bien definida, si es conmutativa, si es asociativa, si existe elemento identidad, si existe inverso para cada elemento de A .

6

Apuntadores

Los apuntadores, llamados también **punteros**, son variables que guardan direcciones de memoria de variables doble precisión o de variables enteras o de variables de otro tipo. En el siguiente ejemplo, p y q están declaradas como apuntadores, p está disponible para guardar una dirección de una variable doble precisión. Usualmente se dice que p apunta a una variable doble precisión. El otro apuntador, q, apunta (puede apuntar) a una variable entera.

```
double *p, x = 0.5;  
int *q, i = 4;
```

Inicialmente estos dos apuntadores tienen un valor cualquiera. Para realmente asignarles una dirección, se debe escribir algo de la forma:

```
p = &x;  
q = &i;
```

Escribir p = &i es un error. El operador & es un operador unario (llamado algunas veces monario) que devuelve la dirección de memoria de una variable. El otro operador para apuntadores es *, que devuelve el valor contenido en una dirección. Estos dos operadores son inversos entre sí. Por un lado, *(&x) es exactamente lo mismo que x. Análogamente, &(*p) es exactamente igual a p.

Considere el siguiente ejemplo:

```
double *p, x = 0.5, y, *r;
```

```

int *q, i = 4, j, *s;

p = &x;
q = &i;

y = *p;
j = *q;
r = p+1;
s = q+3;
printf(" %p %d %lf\n", p, int(p), y);
printf(" %p %d %d\n", q, int(q), j);
printf(" %p %d\n", r, int(r));
printf(" %p %d\n", s, int(s));

```

Estos son los resultados:

```

0064FDFC 6618620 0.500000
0064FDFO 6618608 4
0064FE04 6618628
0064FDFC 6618620

```

Los dos primeros resultados de la primera línea corresponden al valor de *p*, en formato **%p** (para apuntadores) y convertido a entero en formato entero. El primer valor está en hexadecimal, es decir, en base 16. Recuerde de que se usan los dígitos 0, 1, ..., 9 y las letras *A*, *B*, *C*, *D*, *E* y *F*, cuyos valores son *A* = 10, *B* = 11, *C* = 12, *D* = 13, *E* = 14, *F* = 15. Entonces $0064FDFC = 6 \times 16^5 + 4 \times 16^4 + 15 \times 16^3 + 13 \times 16^2 + 15 \times 16 + 12 = 6618620$. Este es el segundo valor de la primera línea. Estos dos valores son simplemente la dirección de la variable doble precisión *x*. Esto quiere decir que en los 8 bytes que empiezan en la dirección de memoria 0064FDFC está el valor de *x*. O sea, en los bytes con dirección 0064FDFC, 0064FDFO, 0064FDFF, 0064FE00, 0064FE01, 0064FE02, 0064FE03, está el valor de *x*.

La orden *y = *p* asigna a *y* el valor contenido en la dirección apuntada por *p*. Como justamente *p* es la dirección de *x*, entonces a *y* se le asigna el valor de *x*, es decir, 0.5 que es precisamente el último valor de la primera línea.

En realidad, los dos primeros valores son válidos durante cada corrida del programa, pero pueden cambiar de corrida a corrida. Muy probable-

mente tambien al compilar y correr el programa en otro computador o con otro compilador. Obviamente el tercer valor siempre será el mismo, no importa el compilador ni el computador.

De manera análoga, en la segunda línea está el valor de `q`, en formato hexadecimal, y como entero. A la variable `j` se le asignó el valor apuntado por `q`, o sea, el valor de `i`, es decir, 4.

Los valores de la tercera línea son en apariencia sorprendentes. Simplemente están los valores del apuntador `r`. Pero se esperaría que apareciera 0064FDFD 6618621 ya que `r` fue definido como `p+1`. Lo que pasa es que se usa la **aritmética de apuntadores**. El apuntador `p` tiene una dirección de memoria de una variable doble precisión; `r` es un apuntador doble precisión y al hacer la asignación `r = p+1`, la aritmética de apuntadores le asigna lo que sería la dirección de la siguiente variable doble precisión. Como una variable doble precisión utiliza 8 bytes, entonces los valores se incrementaron en 8 unidades. Ahora sí los valores de la tercera línea son comprensibles.

En la cuarta línea aparece lo que sería la dirección de la tercera variable entera, enseguida de `i` (`q` es la dirección de `i`). En este compilador los enteros utilizan 4 bytes; entonces, el valor del apuntador `s` corresponde al valor de `q` más 12 unidades.

Las únicas operaciones permitidas a los apuntadores son:

- sumar un entero; por ejemplo, `p += 9;`
- restar un entero; por ejemplo, `p -= 2;`
- operador de incremento; por ejemplo, `q++;`
- operador de decremento; por ejemplo, `p--;`

En ellas se utiliza la aritmética de operadores.

6.1 Apuntadores y arreglos unidimensionales

Considere el siguiente ejemplo:

```
double x[156] = {1.1, 1.2, 1.3, 1.4};
double *p;

cout<<" x = "<<x<<", dir. de x[0] = "<<&x[0]<<endl;
p = x;
```

```
cout<<" p = "<<p<<endl;
```

El resultado es el siguiente:

```
x = 0x0064f924, dir. de x[0] = 0x0064f924
p = 0x0064f924
```

La variable **x** es un arreglo unidimensional de 156 elementos doble precisión, pero para ciertas cosas se considera como una dirección de memoria. Al escribir **x**, el resultado es una dirección de memoria. Observe también que es exactamente la dirección del primer elemento del arreglo, o sea, de **x[0]**. Usando **cout** la dirección aparece con minúsculas y hay 10 caracteres. En los ejemplos anteriores, con **printf** hay mayúsculas y 8 caracteres. Siendo **p** un apuntador, la asignación **p = x** es correcta. En la segunda línea de resultados, aparece efectivamente la misma dirección.

En el siguiente ejemplo, **a** es un arreglo bidimensional; hay varias coincidencias con el ejemplo anterior, pero la asignación **p = a** es incorrecta.

```
double a[10][20] = {1.1, 1.2, 1.3, 1.4};
double *p;

cout<<" a = "<<a<<" dir. de a[0][0] = "<<&a[0][0]<<endl;
// p = a;
// La instruccion anterior (en comentario) es incorrecta.
p = &a[0][0];
cout<<" p = "<<p<<endl;
```

El resultado es:

```
a = 0x0064f7c4 dir. de a[0][0] = 0x0064f7c4
p = 0x0064f7c4
```

En C los apuntadores y los arreglos están íntimamente ligados. En el siguiente ejemplo se ve que **x[i]** y ***(p+i)** son exactamente lo mismo.

```
double x[40] = {1.1, 1.2, 1.3}, *p, *q;
int i;

p = x;
```

```

q = x;
for( i=0; i <= 5; i++){
    printf("%6.2lf%6.2lf%6.2lf\n", x[i], *(p+i), *q);
    q++;
}

```

El resultado es el siguiente:

```

1.10  1.10  1.10
1.20  1.20  1.20
1.30  1.30  1.30
0.00  0.00  0.00
0.00  0.00  0.00
0.00  0.00  0.00

```

Es más fácil ver la equivalencia entre `x[i]` y `*(p+i)` si esta última expresión se escribe `*(&x[0]+i)`, o sea, el contenido de la dirección de `x[0]` incrementada en `i`.

En el ejemplo anterior, el uso del apuntador `q` sugiere el uso de apuntadores para el manejo de los arreglos. A continuación hay varias formas de una función que calcula el promedio de los primeros n elementos de un arreglo.

```

double promX1( double *x, int n)
{
    // promedio de los elementos del 'vector' x

    int i;
    double s = 0.0;

    // error si n <= 0

    for( i = 0; i < n; i++) s += x[i];
    return s/n;
}

//-----
double promX2( double *x, int n)
{
    // promedio de los elementos del 'vector' x

```

```
double *p, *pFin;
double s = 0.0;

// error si n <= 0

pFin = x+n;
for( p = x ; p < pFin; p++) s += *p;
return s/n;
}

//-----
double promX3( double *x, int n)
{
    // promedio de los elementos del 'vector' x

    double *p, *pFin;
    double s = 0.0;

    // error si n <= 0

    p = x;
    pFin = x+n;
    while( p < pFin){
        s += *p;
        p++;
    }
    return s/n;
}

//-----
double promX4( double *x, int n)
{
    // promedio de los elementos del 'vector' x

    double *p, *pFin;
    double s = 0.0;

    // error si n <= 0
```

```

p = x;
pFin = x+n;
while( p < pFin) s += *p++;
return s/n;
}

```

En ciertos casos el manejo de un arreglo con apuntadores es más eficiente que el manejo con subíndices. Sin embargo, para una persona no muy acostumbrada a la utilización de los apuntadores, es más fácil escribir o leer un programa donde los arreglos se tratan por medio de subíndices. A lo largo de este libro habrá ejemplos con los dos enfoques: con subíndices para tratar de ofrecer mayor claridad; con apuntadores para buscar, posiblemente, mayor eficiencia y para ir acostumbrando al lector a su comprensión y manejo. Un programador de nivel intermedio puede fácilmente “traducir” un programa con subíndices a un programa con apuntadores.

Considere ahora la siguiente porción de código donde se utiliza cualquiera de las cuatro funciones que calculan el promedio.

```

double x[100] = { 0.1, 0.2, 0.3};

printf("%lf\n", promX1(x,4) );
printf("%lf\n", promX1( &x[2], 10) );

```

El resultado es el siguiente:

```

0.150000
0.030000

```

Es claro que el promedio de los primeros cuatro valores del arreglo **x**: 0.1, 0.2, 0.3 y 0.0, es 0.15 . En la segunda llamada a la función **promX1**, el primer parámetro es la dirección de memoria del elemento **x[2]** y el segundo es el valor 10; entonces, la función **promX1**, que espera un arreglo, considera un arreglo que empieza exactamente en esa dirección y que tiene 10 elementos, o sea, los valores 0.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0; su promedio es 0.03. Esta manera de pasar como parámetro un arreglo que en realidad es parte de un arreglo más grande, es muy útil y eficiente para el manejo de vectores y matrices en métodos numéricos.

6.2 Apuntadores a apuntadores

Un apuntador a apuntador, a veces llamado apuntador doble (diferente de un apuntador a doble precisión), es una variable que contiene la dirección de memoria de un apuntador. Se declara colocando doble asterisco delante del nombre. Considere el siguiente ejemplo:

```
int i, *p, **q;

i = 16;
p = &i;
q = &p;
printf("%p %d %p %p %d\n", p, *p, q, *q, **q);
```

El resultado es el siguiente:

```
0064FE00 16 0064FDFC 0064FE00 16
```

p es un apuntador a entero; después de la asignación tiene la dirección de i. q es un apuntador a apuntador a entero; dicho de otra forma, un apuntador doble a entero. Después de la asignación q tiene la dirección del apuntador p. La orden printf escribe cinco valores:

- p, es decir, la dirección de i,
- el contenido apuntado por p, es decir, el valor de i,
- q, es decir, la dirección de p,
- el contenido apuntado por q, es decir, el valor de p,
- el contenido apuntado por lo apuntado por q, o sea, el valor de i.

El primero y el cuarto valor coinciden; también coinciden el segundo y el quinto valor.

6.3 Apuntadores y arreglos bidimensionales

Como los arreglos bidimensionales almacenan las matrices fila por fila, entonces se puede saber en qué posición, a partir del elemento a[0][0], está el elemento a[i][j]. Si n es el tamaño de una fila, es decir, el número de columnas, entonces el elemento a[i][j] está $i \cdot n + j$ posiciones después de a[0][0].

El siguiente ejemplo produce como resultado dos valores iguales:

```

double *p, c[4][3] = { .1,.2,.3,.4,.5,.6,
    .7,.8,.9, 1.0, 1.1, 1.2};
int n, i, j ;

n = 3;
i = 2;
j = 1;
p = &c[0][0];

printf("%lf %lf\n", c[i][j], *(p + i*n + j));

```

Teniendo en cuenta los valores del ejemplo, el resultado es:

0.800000 0.800000

6.4 Matrices y arreglos unidimensionales

Anteriormente se vio que el paso de arreglos bidimensionales le quitaba generalidad a las funciones. Una manera de resolver este inconveniente es almacenando la matriz en un vector, fila por fila. También se podría hacer columna por columna; es menos usual, pero en algunos casos específicos podría ser más conveniente. La fórmula para calcular la posición es la misma o análoga a la de los apuntadores y los arreglos bidimensionales.

Si se considera una matriz, con elementos a_{ij} , donde $1 \leq i \leq m$, $1 \leq j \leq n$, almacenada en un arreglo unidimensional v , entonces

$$a_{ij} = v[(i-1)*n + j-1].$$

Si se considera una matriz, con elementos a_{ij} , donde $0 \leq i \leq m-1$, $0 \leq j \leq n-1$, almacenada en un arreglo unidimensional v , entonces

$$a_{ij} = v[i*n + j].$$

En el siguiente ejemplo, como en el programa `prog14`, se hace la lectura y escritura de dos matrices, y luego el producto. El almacenamiento de las matrices se hace fila por fila en arreglos unidimensionales.

// Matrices en arreglos unidimensionales

```
// Lectura y escritura de 2 matrices y calculo del producto

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void escrX(double *x, int n );
void lectA1(double *a, int m, int n, char c );
void escrA1(double *a, int m, int n );
int prodABsub1(double *a, int m, int n, double *b, int p,
    int q, double *c);
//=====
int main()
{
    double a[1000], b[1000], c[1000];
    int m, n, p, q;

    printf("\n Producto de dos matrices.\n\n");

    printf(" num. de filas de A : ");
    scanf( "%d", &m);
    printf(" num. de columnas de A : ");
    scanf( "%d", &n);
    // es necesario controlar que m, n no son muy grandes
    // ni negativos

    printf(" num. de filas de B : ");
    scanf( "%d", &p);
    printf(" num. de columnas de B : ");
    scanf( "%d", &q);
    // es necesario controlar que p, q no son muy grandes
    // ni negativos

    if( n != p ){
        printf(" Producto imposible\n");
        exit(1);
    }
    lectA1(a, m, n, 'A');
```

```

printf(" A : \n");
escriA1(a, m, n);

lectA1(b, n, q, 'B');
printf(" B : \n");
escriA1(b, n, q);

if( prodABsub1(a,m,n, b,p,q, c) ){
    printf(" C : \n");
    escriA1(c, m, q);
}
else printf("\n ERROR\n");
return 0;
}

//=====
void lectA1(double *a, int m, int n, char c )
{
    // lectura de los elementos de una matriz.
    // almacenada en un "vector"

    int i, j;

    for( i = 0; i < m; i++){
        for( j=0; j < n; j++){
            printf(" %c[%d] [%d] = ", c, i+1, j+1);
            scanf("%lf", &a[i*n+j] );
        }
    }
}

//=====
void escriA1(double *a, int m, int n )
{
    // escritura de los elementos de una matriz
    // almacenada en un vector
    // utiliza escriX

    int i;
}

```

```

    for(i=0;i<m;i++) escrX( &a[i*n] , n );
}
//-----
void escrX(double *x, int n )
{
    // escritura de los elementos de un "vector"

    int j;
    int nEltosLin = 5; // numero de elementos por linea

    for( j = 0; j < n; j++){
        printf("%15.8lf", x[j]);
        if( (j+1)%nEltosLin == 0 || j == n-1) printf("\n");
    }
}
//-----
int prodABsub1(double *a, int m, int n, double *b, int p,
               int q, double *c)
{
    // producto de dos matrices, a mxn, b pxq
    // devuelve 1 si se puede hacer el producto
    // devuelve 0 si no se puede
    // Las matrices estan almacenadas en "vectores"
    // El manejo se hace mediante subindices

    int i, j, k;
    double s;

    if(m<0||n<0||p<0||q<0 || n!= p ) return 0;
    for( i=0; i < m; i++){
        for( j=0; j < q; j++){
            s = 0.0;
            for( k=0; k<n; k++) s += a[i*n+k]*b[k*q+j];
            c[i*q+j] = s;
        }
    }
    return 1;
}

```

Observe que la función que escribe la matriz, `escriA1`, lo hace fila por fila. Para esto llama varias veces la función `escriX` que escribe los elementos de un vector. O sea, cada fila de la matriz se considera como un vector de n elementos que empieza en la dirección de `a[i*n]`.

En el programa anterior, como en un ejemplo ya visto, durante la lectura de la matriz, aparecen los subíndices $i+1$ y $j+1$. Entonces, para el usuario, aparentemente los subíndices empiezan en 1 y acaban en m o en n .

A manera de ejemplo del paso de manejar los arreglos con subíndices al tratamiento mediante apuntadores, a continuación hay una versión de la función que calcula el producto entre matrices, almacenadas en vectores, mediante el uso de apuntadores.

```
int prodABap1(double *a, int m, int n, double *b, int p,
              int q, double *c)
{
    // producto de dos matrices, a mxn, b pxq
    // devuelve 1 si se puede hacer el producto
    // devuelve 0 si no se puede
    // Las matrices estan almacenadas en "vectores"
    // el producto se hace por medio de apuntadores

    double *cij, *ai0, *bkj; // apuntadores a: Cij, Aik, Bkj
    double *cmq; // apuntador a la dirección de c[m*q]
    double *ciq; // apunt. a C(i+1,0) : c[i*q]
    double *ai0; // apuntador a Ai0
    double *ain; // apunt. a A(i+1,0) : a[i*n]
    double *b0j; // apuntador a B(0,j)

    if(m<0||n<0||p<0||q<0 || n!= p ) return 0;
    cij = c;
    cmq = c+m*q;
    ai0 = a;
    while( cij < cmq ){
        cij = cij + q;
        b0j = b;
        while( cij < cij ){
            aik = ai0;
```

```

    ain = ai0+n;
    bkj = b0j;
    *cij = 0.0;
    while( aik < ain ){
        *cij += (*aik++)*(*bkj);
        bkj += q;
    }
    cij++;
    b0j++;
}
ai0 = ain;
}
return 1;
}

```

La elaboración de la función `prodABsub1` es más fácil que la de la función `prodABap1`. Su lectura, por otro programador, también es más fácil. Sin embargo, la función `prodABsub1` tiene algunas ineficiencias claras.

Dentro de un bucle `for` (y dentro de los otros bucles) no se deben realizar operaciones independientes de las variables que se modifican dentro del `for`; es más eficiente usar una variable a la que se le asigna, fuera del `for`, el valor de la operación y, en lugar de efectuar la operación dentro del `for`, se usa la variable suplementaria. Por ejemplo, en la función `prodABsub1`, dentro del bucle `for(k=...)` se realiza varias veces, quizás muchas, la operación `i*n+k`. Como las variables `i`, `n` no se modifican dentro de este `for`, entonces es preferible utilizar una variable entera adicional, por ejemplo llamada `in`, fuera de este `for` hacer la asignación `in = i*n;` y dentro del `for(k=...)` utilizar `a[in+k]`.

La siguiente función, `prodABsub2`, es una versión más eficiente de la función `prodABsub1`.

```

int prodABsub2(double *a, int m, int n, double *b, int p,
               int q, double *c)
{
    // ...
    int i, j, k, in, iq;
    double s;

```

```

if(m<0||n<0||p<0||q<0 || n!= p ) return 0;

for( i=0; i < m; i++){
    in = i*n;
    iq = i*q;
    for( j=0; j < q; j++){
        s = 0.0;
        for( k=0; k<n; k++) s += a[in+k]*b[k*q+j];
        c[iq+j] = s;
    }
}
return 1;
}

```

En la función anterior se observa que la variable de control para el primer **for** es la variable **i**, pero ella no se utiliza sola dentro del **for**, sirve para calcular **in** e **iq**. Se puede pensar que el **for** puede estar controlado directamente por una de las dos variables. También se puede pensar en cambiar el **for** por **while** o viceversa. A manera de ilustración, aparecen a continuación otras dos funciones que también realizan el producto de dos matrices almacenadas en arreglos unidimensionales.

```

int prodABsub3(double *a, int m, int n, double *b, int p,
               int q, double *c)
{
    // ...

    int i, j, k, in, iq, kq, ink, iqj;
    double s;

    if(m<0||n<0||p<0||q<0 || n!= p ) return 0;

    iq = 0;
    for( in=0; in < n*m; in += n){
        iqj = iq;
        for( j=0; j < q; j++){
            s = 0.0;
            ink = in;
            for( kq=0; kq<n*q; kq += q){

```

```

        s += a[ink]*b[kq+j];
        ink++;
    }
    c[iqj] = s;
    iqj++;
}
iq += q;
}
return 1;
}
//-----
int prodABap2(double *a, int m, int n, double *b, int p,
               int q, double *c)
{
    // ...

    double *cij, *aik, *bkj;// apuntadores a: Cij, Aik, Bkj
    double *ai0; // apuntador a Ai0
    double *ain; // apunt. a A(i+1,0) : a[i*n]
    double *b0j; // apuntador a B(0,j)
    double *amn;
    double *b0q;

    if(m<0||n<0||p<0||q<0 || n!= p ) return 0;

    amn = a + m*n;
    b0q = b+q;
    cij = c;
    ain = a+n;
    for( ai0 = a; ai0 < amn; ai0 += n){
        for( b0j = b; b0j < b0q; b0j++){
            bkj = b0j;
            *cij = 0.0;
            for(aik = ai0; aik < ain; aik++ ){
                *cij += (*aik)*(*bkj);
                bkj += q;
            }
            cij++;
        }
    }
}

```

```

    }
    ain += n;
}
return 1;
}

```

Esta funciones pueden ser mejoradas dependiendo, parcialmente, de la plataforma y del compilador. Todas tienen una eficiencia más o menos semejante, salvo la primera, `prodABsub1`, que es un poco menos eficiente.

La siguiente tabla muestra los tiempos para el producto de matrices 500×500 y 1000×1000 . Estos resultados corresponden a un compilador Borland C++ 5.2 para Win32 en un microcomputador con procesador Intel Pentium III MMX de 450 Mhz con 128 Mb de memoria RAM.

El número de operaciones para el producto de dos matrices $n \times n$ es proporcional a n^3 . Es interesante comprobar que, al duplicar el tamaño de las matrices, el tiempo, aproximadamente, se multiplica por $2^3 = 8$.

		$n = 500$	$n = 1000$
subíndices	<code>prodABsub1</code>	8.8	67.8
subíndices	<code>prodABsub2</code>	8.0	63.3
subíndices	<code>prodABsub3</code>	7.7	60.9
apuntadores	<code>prodABap1</code>	7.8	63.1
apuntadores	<code>prodABap2</code>	7.9	63.4

Otra manera de hacer el producto de dos matrices almacenadas en vectores, mediante el uso de apuntadores, que permite más claridad y que brinda la posibilidad de tratar de optimizar cada parte por separado, consiste en utilizar el hecho de que el elemento c_{ij} de la matriz producto es simplemente el producto escalar de la fila i de A y la columna j de B . La fila i de A es simplemente el vector que empieza en la dirección del primer elemento de la fila y tiene n elementos, uno después del otro, de manera contigua. En la columna j de la matriz B hay un ligero inconveniente, los elementos están en un arreglo, pero no consecutivamente. Para pasar de un elemento al siguiente, hay que saltar q posiciones (suponiendo que B es $p \times q$).

A continuación hay dos funciones que hacen el producto escalar de dos vectores de n elementos. El primero está almacenado en el arreglo `x`; hay que saltar `saltox` posiciones para pasar de un elemento al siguiente. Es decir, son los elementos `x[0], x[saltox], x[2*saltox],`

..., $x[(n-1)*saltox]$. De manera análoga, los elementos del segundo vector son: $y[0]$, $y[saltoy]$, $y[2*saltoy]$, ..., $y[(n-1)*saltoy]$. La primera función `prodXYsub` se hace de manera casi natural; la segunda, `prodXY`, es posiblemente más eficiente, pero su programación requiere más tiempo. Se puede utilizar el siguiente criterio: es útil gastar más tiempo haciendo una función más eficiente si ésta se va a usar muchas veces. Como es más eficiente `i++` que `i += 1`, entonces esta última función considera por aparte el caso `salto == 1` y el caso `salto != 1`.

```
double prodXYsub( double *x, int saltox, double *y,
                  int saltoy, int n)
{
    // producto escalar de dos vectores
    // uso de subíndices
    // x[0], x[saltox], x[2*saltox], ..., x[(n-1)*saltox]
    // y[0], y[saltoy], y[2*saltoy], ..., y[(n-1)*saltoy]

    double s = 0.0;
    int i;

    if( n < 0 ){
        printf("prodXY: n negativo\n");
        return s;
    }
    if( n==0 ) return s;
    for(i=0; i<n; i++) s += x[i*saltox]*y[i*saltoy];
    return s;
}
//-----
double prodXY( double *x, int saltox, double *y,
               int saltoy, int n)
{
    // producto escalar de dos vectores
    // uso de apunadores
    // x[0], x[saltox], x[2*saltox], ..., x[(n-1)*saltox]
    // y[0], y[saltoy], y[2*saltoy], ..., y[(n-1)*saltoy]

    double s = 0.0;
    double *xi, *yi, *xn, *yn;
```

```

if( n < 0 ){
    printf("prodXY: n negativo\n");
    return 0.0;
}
if( n==0 ) return 0.0;

if( saltox == 1 ){
    if( saltoy == 1 ){
        // saltox = saltoy = 1
        xi = x;
        xn = x + n;
        yi = y;
        while( xi < xn ) s += (*xi++)*(*yi++);
    }
    else{
        // saltox = 1; saltoy != 1
        xi = x;
        xn = x + n;
        yi = y;
        while( xi < xn ){
            s += (*xi++)*(*yi++);
            yi += saltoy;
        }
    }
}
else{
    if( saltoy == 1 ){
        // saltox > 1; saltoy = 1
        yi = y;
        yn = y + n;
        xi = x;
        while( yi < yn ){
            s += (*xi)*(*yi++);
            xi += saltox;
        }
    }
    else{

```

```

// saltox != 1; saltoy != 1
xi = x;
xn = x + n*saltox;
yi = y;
while( xi < xn ){
    s += (*xi)*(*yi);
    xi += saltox;
    yi += saltoy;
}
}
return s;
}

```

Utilizando la función `prodXY`, la función que calcula el producto de dos matrices podría tener la siguiente forma:

```

int prodAB1(double *a, int m, int n, double *b, int p,
            int q, double *c)
{
    // Producto de dos matrices, A mxn, B pxq.
    // Devuelve 1 si se puede hacer el producto,
    // devuelve 0 si no se puede.
    // Las matrices estan almacenadas en "vectores".
    // El producto se hace por medio de apuntadores.
    // Utiliza una funcion de producto escalar.

    double *cij; // apuntador a Cij
    double *cmq; // apuntador a la direccion de c[m*q]
    double *ciq; // apunt. a C(i+1,0) : Ciq : c[i*q]
    double *ai0; // apuntador a Ai0
    double *b0j; // apuntador a B0j

    if(m<0||n<0||p<0||q<0 || n!= p ) return 0;
    cij = c;
    cmq = c+m*q;
    ai0 = a;
    while( cij < cmq ){
        ciq = cij + q;

```

```

b0j = b;
while( cij < ciq ){
    *cij = prodXY(ai0, 1, b0j, q, n);
    cij++;
    b0j++;
}
ai0 += n;
}
return 1;
}

```

Normalmente una función es un poco menos eficiente cuando llama otra función para hacer un proceso, que cuando directamente lo hace internamente. Pero, por otro lado, repartir el trabajo en funciones permite revisar, depurar y tratar de optimizar más fácilmente cada función por aparte.

En el ejemplo del producto de matrices, resultó aún más eficiente esta última función —`prodAB1`— que la función `prodABap1`. Esto se debe tal vez a que la función `prodXY` es bastante eficiente y que muy posiblemente la función `prodABsub1` tiene varias partes susceptibles de ser mejoradas. La siguiente tabla muestra los tiempos, en segundos, para el producto de dos matrices cuadradas, con $n = 500$ y con $n = 1000$.

		$n = 500$	$n = 1000$
subíndices	<code>prodABsub2</code>	8.0	63.3
apuntadores	<code>prodABap1</code>	7.8	63.1
apuntadores + prod. escalar	<code>prodAB1</code>	7.0	56.0

6.5 Arreglos aleatorios

El lector podrá preguntarse:

¿Cómo trabajar con matrices 500×500 ?

¿Entrando los valores por teclado? (el autor de este libro no lo hizo).

¿Leyendo los datos en un archivo? Podría ser, pero todavía no se ha visto este tema.

¿Creando la matriz con alguna fórmula específica? Podría ser, por ejemplo, $a_{ij} = 10000i + j$.

¿Creando la matriz aleatoriamente? ¿Porqué no?

C provee una función para crear números aleatorios: `rand()`. Esta función da como resultado enteros entre 0 y `RAND_MAX`, cuyo valor está definido en el archivo de cabecera `stdlib.h`. La siguiente función devuelve un valor doble precisión aleatorio, o seudoaleatorio, en el intervalo $[a, b]$. Se utiliza el cambio de variable

$$t = a + \frac{b - a}{\text{RAND_MAX}} r.$$

Cuando r vale 0, entonces t vale a . Cuando r vale `RAND_MAX`, entonces t vale b .

```
double aleat(double a, double b)
{
    // calculo de un numero aleatorio en [ a, b ]

    double r;

    r = a + rand()*(b-a)/RAND_MAX;
    return r;
}
```

Utilizando la anterior función se puede hacer una función que da valores aleatorios a los elementos de un arreglo unidimensional.

```
void aleatX( double *x, int n, double a, double b)
{
    // vector aleatorio, valores entre a y b

    int k;

    for( k=0; k<n; k++) x[k] = aleat(a,b);
}
```

En el siguiente trozo de programa, se hace el llamado a la función anterior, primero para un vector, luego para una matriz almacenada en un arreglo unidimensional.

```

double x[1000];
double a[250000]; // matriz A
int m, n;

...
m = 400;
n = 500;
...
aleatX(x, n, 0.0, 1.0);
aleatX(a, m*n, -1.0, 1.0);
...

```

6.6 Asignación dinámica de memoria

Cuando se declara un arreglo en una función, por ejemplo en `main`, con un tamaño determinado, C separa para él la memoria durante todo el tiempo que el control del programa esté en esa función (desde que entra hasta que sale). Buscando que el programa sea general, es necesario declarar el arreglo con un número muy grande de elementos. Sin embargo, la mayoría de las veces se va a usar ese arreglo con un número pequeño de elementos. La memoria reservada para ese arreglo no puede ser utilizada por otro arreglo que podría necesitarla. Como la memoria es un recurso finito, no debe desperdiciarse. Más aún: si se trata de utilizar arreglos demasiado grandes podría producirse un error, durante la compilación, por memoria insuficiente. En algunos sistemas operacionales, cuando se pide más memoria que la disponible, el sistema utiliza el disco duro para reemplazar la memoria faltante, pero hay un inconveniente: el acceso al disco duro es mucho menos rápido que el acceso a la memoria.

C permite trabajar con arreglos del tamaño exactamente necesario, mediante la asignación dinámica de memoria. Considere el siguiente ejemplo, que usa las funciones anteriores `escriX`, `aleat`, `aleatX`.

```

double *x;
int n;

printf(" n = ");
scanf("%d", &n );

```

```

x = malloc( n*sizeof(double) );
if( x == NULL ){
    printf(" Memoria insuficiente\n");
    exit(1);
}
aleatX(x, n, -1.0, 1.0);
escriX(x, n);
free(x);

```

A lo largo de este libro se supondrá que cuando un programa usa funciones definidas con anterioridad, para que funcione, es necesario agregar, al código presentado, el prototipo, la definición y los archivos de cabecera de cada una de las funciones utilizadas.

La función **sizeof**, del lenguaje C, da como resultado un entero que indica el número de bytes que utilizan las variables cuyo tipo es el indicado por el parámetro. En la mayoría de las implementaciones **sizeof(double)** es 8. La función **malloc** reserva el número de bytes indicado por el parámetro. Se hubiera podido escribir **x = malloc(n*8)**, pero podría no funcionar en un computador con un compilador de C donde los números doble precisión utilicen un número de bytes diferente de 8. Cuando no hay suficiente memoria disponible, la función **malloc** devuelve **NULL**, o sea, el apuntador valdrá **NULL**. Es muy conveniente controlar siempre si hubo suficiente memoria. La función **free** libera el espacio que había sido reservado para **x**.

El ejemplo anterior, hecho en C, y suponiendo que las funciones están estrictamente en C, puede presentar errores con algunos compiladores, si se compila como programa C++. Este hecho es muy poco frecuente, pero puede suceder.

El anterior inconveniente se puede solucionar colocando un molde para el uso de **malloc**:

```
x = (double *)malloc( n*sizeof(double) );
```

La asignación anterior es válida tanto en C como en C++.

La asignación de memoria en C++ es más sencilla; se hace por medio de **new**. La memoria que ya fue utilizada y que no se necesita más, se libera mediante **delete**. Aunque a veces no es absolutamente indispensable liberar la memoria al salir de una función, de todas maneras sí es una buena costumbre de programación el hacerlo.

Los ejemplos que siguen se refieren a la evaluación de un polinomio p en un valor dado t . Supongamos que en un arreglo p están almacenados los coeficientes de un polinomio de grado n . Las siguientes líneas permiten leer el grado del polinomio, asignar memoria dinámicamente para almacenar los coeficientes, dar valores aleatorios a los coeficientes e imprimirlos.

```
double *p;
int n;

printf(" n = ");
scanf("%d", &n);

p = new double[n+1];
if( p == NULL ){
    printf(" Memoria insuficiente\n");
    exit(1);
}
aleatX(p, n+1, -10.0, 10.0);
escrX(p, n+1);

delete p;
```

La siguiente función muestra una manera sencilla, pero ineficiente, de evaluar $p(t)$.

```
double evalPol00( double *p, int n, double t)
{
    // evaluacion simplista de p(t)
    // p(x) = p[0] + p[1]*x + p[2]*x*x + ... + p[n]*x^n

    int i;
    double pt = 0.0;

    for( i=0; i<= n; i++) pt += p[i]*pow(t,i);
    return pt;
}
```

Una manera más eficiente de evaluar $p(t) = a_0 + a_1t + a_2t^2 + \dots + a_nt^n$

consiste en observar la siguiente manera de agrupar.

$$p(t) = (\cdots (((a_n)t + a_{n-1})t + a_{n-2})t + \cdots + a_1)t + a_0.$$

Es decir, se empieza con el valor a_n . A continuación se repite el proceso: *se multiplica por t y se adiciona el siguiente coeficiente (en orden decreciente)*. Esta manera de calcular se conoce con el nombre de esquema de Hörner o también hace parte del proceso conocido como división sintética. La división sintética permite evaluar $p(t)$ y, al mismo tiempo, proporciona el polinomio cociente al efectuar la división $p(x)/(x - t)$. Como ejemplo, considere el polinomio $p(x) = 3 - 4x + 5x^2 - 16x^3 + 7x^4$ dividido por $x - 2$.

$$\begin{array}{r} 7 & -16 & 5 & -4 & 3 & | & 2 \\ & 14 & -4 & 2 & -4 & | \\ \hline 7 & -2 & 1 & -2 & -1 & | \end{array}$$

Entonces $p(2) = -1$ y $7x^4 - 16x^3 + 5x^2 - 4x + 3 = (7x^3 - 2x^2 + x - 2)(x - 2) + (-1)$.

La primera de las dos funciones que siguen, `evalPol`, evalúa el polinomio en un valor t usando el esquema de Hörner. La segunda función devuelve $p(t)$ (que también es el residuo) y en el arreglo apuntado por `q` están los coeficientes del polinomio cociente de la división. El arreglo `p` debe tener $n+1$ elementos; el cociente, de grado $n-1$, tiene n elementos.

```
double evalPol( double *p, int n, double t)
{
    // evaluacion de p(t) utilizando el esquema de Horner
    // p(x) = p[0] + p[1]*x + p[2]*x*x + ... + p[n]*x^n

    double pt, *pi, *pn;

    pi = p + n;
    pt = *pi;
    pi--;
    while( pi >= p ) pt = pt*t + *pi--;
    return pt;
}
//-----
double divSint( double *p, int n, double t, double *q)
```

```

{
    // division sintetica de p(x)/(x-t)
    // n es el grado del polinomio
    // p(x) = p[0] + p[1]*x + p[2]*x*x + ... + p[n]*x^n
    // devuelve el residuo
    // q contiene el cociente

    double pt; // p(t) : residuo
    double *pi, *pn, *qi;

    pi = p + n;
    pt = *pi;
    pi--;
    qi = q+n-1;
    while( pi >= p ){
        *qi-- = pt;
        pt = pt*t + *pi--;
    }
    return pt;
}

```

Al utilizar las funciones `evalPol00` y `evalPol` con polinomios creados aleatoriamente, se ve una gran gran diferencia entre la eficiencia de los dos métodos. La siguiente tabla muestra algunos tiempos en segundos.

n=	500000	1000000	5000000
<code>evalPol00</code>	0.33	0.65	3.56
<code>evalPol</code>	0.03	0.06	0.29

6.7 Matrices y apuntadores dobles

Otra de las maneras útiles para el manejo de matrices, que no quitan generalidad a las funciones, es mediante apuntadores dobles (apuntadores a apuntadores).

Consideremos una matriz de m filas y n columnas. Cada fila tiene n elementos y se puede considerar como un arreglo unidimensional de n elementos. Para el manejo de cada fila, basta con conocer la dirección del primer elemento de la fila. O sea, cada fila se maneja por medio de un

apuntador. Como hay m filas, entonces hay m apuntadores y se necesita un arreglo con los m apuntadores. A su vez este arreglo de apuntadores se puede manejar con la dirección del primer apuntador, o sea, con un apuntador a apuntador.

Si \mathbf{a} es un apuntador doble, entonces $\mathbf{a}[i]$ es la dirección de $\mathbf{a}[i][0]$ y \mathbf{a} es la dirección de $\mathbf{a}[0]$, es decir,

$$\begin{aligned}\mathbf{a}[i] &= \&\mathbf{a}[i][0], \\ \mathbf{a} &= \&\mathbf{a}[0].\end{aligned}$$

En el siguiente ejemplo se hace un asignación dinámica de memoria para una matriz, se imprimen ciertas direcciones y valores y se libera la memoria asignada.

```
double **a;
int m, n, i, j;

m = 4;
n = 3;
//-----
// asignacion de memoria

a = new double *[m] ;
if( a == NULL ){
    printf("Memoria insuficiente\n");
    exit(1);
}
for( i=0; i<m; i++ ) {
    a[i] = new double [n];
    if( a[i] == NULL ){
        printf("Memoria insuficiente\n");
        exit(1);
    }
}
//-----
// algunos resultados

for( i = 0; i < m; i++ ){
    for( j = 0; j < n; j++ )
```

```

        printf("&a[%d] [%d]=%d ", i, j, &a[i][j] );
        //printf("&a%d%d=%d ", i, j, &a[i][j]);
        printf("\n");
    }
    printf("\n");
    for(i=0; i<m; i++) printf("a[%d]=%d\n", i, a[i]);
    printf("\n");
    for(i=0; i<m; i++) printf("&a[%d]=%d\n", i, &a[i]);
    printf("\n");
    printf("a = %d\n", a);
//-----
// liberacion de memoria

for( i=0; i<m; i++) delete a[i];
delete a;

```

Los resultados pueden ser los siguientes:

```

&a[0][0]=6695108 &a[0][1]=6695116 &a[0][2]=6695124
&a[1][0]=6695136 &a[1][1]=6695144 &a[1][2]=6695152
&a[2][0]=6695164 &a[2][1]=6695172 &a[2][2]=6695180
&a[3][0]=6695192 &a[3][1]=6695200 &a[3][2]=6695208

```

```

a[0]=6695108
a[1]=6695136
a[2]=6695164
a[3]=6695192

```

```

&a[0]=6695088
&a[1]=6695092
&a[2]=6695096
&a[3]=6695100

```

```
a = 6695088
```

Observe, en el ejemplo anterior, los siguientes hechos:

- En la declaración, el nombre del apuntador doble está precedido por dos asteriscos.

- Primero se asigna memoria para un arreglo de apuntadores.
- En la asignación de memoria para un arreglo de apuntadores, se coloca un asterisco antes del paréntesis cuadrado izquierdo.
- A cada uno de los apuntadores del arreglo de apuntadores se le asigna memoria para los elementos de la fila correspondiente.
- La liberación de memoria se hace en el orden inverso, primero se libera la memoria apuntada por cada uno de los apuntadores del arreglo de apuntadores.
- Después se libera la memoria apuntada por el apuntador doble.

En los resultados se pueden comprobar varios hechos. El primer paquete muestra las direcciones de cada uno de los elementos de la matriz. Observe que en cada fila las direcciones son consecutivas (de 8 en 8 por tratarse de números doble precisión). El primer elemento de una fila no está en la dirección siguiente a la dirección del último elemento de la fila anterior. El segundo paquete muestra los valores del arreglo de apuntadores. Obviamente el valor de `a[i]` es exactamente la dirección de `a[i][0]`. El tercer paquete de resultados muestra las direcciones de los elementos del arreglo de apuntadores. Las direcciones son contiguas, pero no de 8 en 8. El valor de `a` es exactamente la dirección del primer elemento del arreglo de apuntadores.

La asignación dinámica de memoria para una matriz en un apuntador doble, y su liberación, pueden hacerse por medio de las funciones que aparecen a continuación.

```
double **creaA( int m, int n)
{
    // asigna dinamicamente memoria para una matriz  m x n
    // m >= 1,  n >= 1
    // devuelve un valor de apuntador a apuntador
    //      tipo doble precision

    double **p;
    int i;

    p = new double *[m] ;
```

```

if( p == NULL ) return( NULL );

for( i=0; i<m; i++ ) {
    p[i] = new double[n];
    if( p[i] == NULL ) return( NULL );
}
return p;
}

//-----
void libA( double **a, int m)
{
    // libera memoria asignada a una matriz
    // m : numero de filas

    int i;

    for( i = 0; i < m; i++) delete a[i];
    delete a;
}

```

A continuación aparece un ejemplo de la utilización de estas dos funciones.

```

double **a;
int m, n;

...
a = creaA(m, n);
...
libA( a, m);

```

Observe en la definición de la función, que **creaA** devuelve un apuntador doble, indicado por los dos asteriscos que preceden al nombre de la función. Por supuesto en el llamado se necesita un apuntador doble.

6.8 Arreglos a partir de 1

Como se ha visto, los arreglos en C empiezan en 0. A veces por mayor comodidad, o por necesidad, deben comenzar en el subíndice 1.

Mediante un desplazamiento del valor del apuntador, se puede manejar un apuntador con subíndices entre 1 y n . Más aún, es posible hacer variar el subíndice entre dos valores enteros $i_1 < i_2$.

Considere la siguiente porción de programa.

```
double *x; // arreglo entre 1 y n

int n, i;

n = 7;
x = new double[n];
x--;

for( i = 1; i <= n; i++) x[i] = i*i*i;

x++;
delete x;
```

Suponga que el valor de **x**, después de la asignación de memoria, es la dirección 10328. Entonces las 7 direcciones para los 7 elementos del arreglo son: $10328 + 0 \times 8$, $10328 + 1 \times 8$, $10328 + 2 \times 8$, ..., $10328 + 6 \times 8$. La orden **x--** hace que el valor de **x** se disminuya en una unidad de memoria doble precisión, entonces **x** pasa a valer 10320. Las posiciones de memoria para los 7 elementos siguen siendo las mismas, pero se pueden ver como los valores $10320 + 1 \times 8$, $10320 + 2 \times 8$, $10320 + 3 \times 8$, ..., $10320 + 7 \times 8$. Es decir, a partir de ese nuevo valor de **x**, los elementos del vector estarían en las posiciones 1 hasta 7.

Para liberar la memoria se requiere que **x** tenga el valor inicial, entonces es necesario incrementar su valor en una unidad (de memoria doble precisión).

La siguiente porción de programa muestra cómo podría ser el manejo de arreglos bidimensionales con asignación dinámica de memoria, comenzando en el subíndice 1.

```
double **a; // matriz con subindices en [1,m], [1,n]

int m, n, i, j;

m = 3;
```

```

n = 4;
a = new double *[m];
if( a == NULL ){
    printf(" Memoria insuficiente\n");
    exit(1);
}
a--;
for( i = 1; i <= m; i++){
    a[i] = new double[n];
    if( a[i] == NULL ){
        printf(" Memoria insuficiente\n");
        exit(1);
    }
    a[i]--;
}

for( i = 1; i <= m; i++){
    for( j = 1; j <= n; j++) a[i][j] = i*j;
}

// liberacion de la memoria

for( i = 1; i <= m; i++ ){
    a[i]++;
    delete a[i];
}
a++;
delete a;

```

Fue necesario disminuir en una unidad tanto `a`, como cada uno de los `a[i]`. Para liberar la memoria se requiere el proceso inverso.

A fin de obtener un arreglo con subíndice que varíe entre `i1` y `i2`, $i1 < i2$, se necesita asignar memoria para $i2 - i1 + 1$ elementos y disminuir el valor del apuntador en `i1`. El siguiente ejemplo ilustra lo anterior.

```

double *y; // arreglo entre i1 e i2, i1 < i2

int i, i1, i2;

```

```
i1 = -2;  
i2 = 7;  
  
y = new double[i2-i1+1];  
y -= i1;  
  
for( i =i1; i <= i2; i++) y[i] = rand();  
  
y += i1;  
delete y;
```

Ejercicios

Para cada uno de los enunciados siguientes, defina cuáles son los datos necesarios. Haga un programa que lea los datos, llame la función (o las funciones) que realiza los cálculos y devuelve los resultados, y finalmente que el programa principal (la función `main`) muestre los resultados. En todos los casos haga asignación dinámica de memoria. Cuando se trate de matrices, haga el programa de dos maneras, almacenando la matriz mediante apuntadores sencillos y mediante apuntadores dobles.

- 6.1** Considere los ejercicios del capítulo anterior.
- 6.2** Haga funciones que realicen las siguientes operaciones elementales sobre las filas de una matriz: intercambiar dos filas; multiplicar una fila por una constante; agregar a una fila un múltiplo de otra fila.
- 6.3** Construya la traspuesta de una matriz.
- 6.4** Dada una matriz A , calcule $A^T A$.
- 6.5** Averigüe si una matriz es simétrica.
- 6.6** Averigüe si una matriz es diagonal.
- 6.7** Averigüe si una matriz es triangular superior.

7

Lectura y escritura en archivos

Este capítulo presenta las nociones de C sobre la lectura y escritura en **archivos de texto**. La lectura de un archivo existente, se hace de manera secuencial, es decir, lo que está en el archivo se lee en el orden en que aparece. No se puede leer un dato al principio del archivo, después al final y después en la mitad. La escritura también es secuencial.

En realidad, de manera artificial, sí se podría, por ejemplo, leer el primer dato, leer los datos que siguen pero desechándolos hasta llegar al final, leer el dato deseado, devolverse hasta el principio, leer datos desechándolos y leer el dato deseado en un punto intermedio del archivo. Sin embargo, en este libro, siempre se supondrá que la lectura y escritura se hacen de manera natural, es decir, secuencial.

Las principales funciones que se utilizan para la entrada y salida con archivos están a continuación. La palabra entrada se refiere a lectura: la información entra, desde un archivo, al “programa”. La palabra salida se refiere a escritura: la información sale del “programa” hacia un archivo. Observe que por convención todas empiezan por la letra **f** (de *file*).

```
fopen,  
fscanf,  
fclose,  
fprintf,  
feof.
```

7.1 fopen, fscanf, fclose, fprintf

El siguiente programa lee en un archivo el tamaño de un vector, lee las componentes del vector, calcula el promedio y escribe este último resultado en otro archivo.

```
// lectura de un vector en un archivo
// escritura del promedio en otro archivo

#include...

int main()
{
    double *x, prom;
    FILE *archDat, *archRes;
    int n, i;

    archDat = fopen("datos1", "r");
    if( archDat == NULL ){
        printf("\n\n Archivo inexistente.\n\n");
        exit(1);
    }
    fscanf(archDat, "%d", &n);
    x = new double[n];
    if( x == NULL ){
        printf("\n\n Memoria insuficiente.\n\n");
        exit(1);
    }
    for( i=0; i<n; i++) fscanf(archDat, "%lf", &x[i]);
    fclose(archDat);

    archRes = fopen("ejemplo.res", "w");

    prom = promX(x, n);
    fprintf(archRes, " promedio = %12.4lf\n", prom);
    fclose(archRes);
    return 0;
}
```

El programa usa las variables **archDat** y **archRes**. Estas variables apuntan a un FILE, que es el tipo de archivo que se va a utilizar, es decir, un archivo de texto de acceso secuencial. Simplificando, se puede decir que **archDat** es el nombre de un archivo dentro del programa.

El nombre externo del archivo, en el sistema operacional, es **datos1**. La función **fopen** hace la correspondencia entre el nombre interno y el nombre externo, y abre el archivo. El segundo parámetro de **fopen**, o sea la cadena de caracteres "r", indica que se trata de un archivo para lectura.

Las maneras más usuales de manejo de archivos de texto mediante **fopen** son:

- "r" Para lectura.
- "w" Para escritura.
- "a" Para escritura, al final del archivo.
- "r+" Para lectura y escritura.

Al abrir un archivo para lectura, necesariamente debe existir. Si la función **fopen** no puede abrir el archivo, entonces devuelve NULL. Es una buena costumbre verificar si el programa pudo abrir el archivo.

La lectura en un archivo se puede hacer mediante la función **fscanf**. Su manejo es semejante al de **scanf**, pero tiene un parámetro adicional, antes de la cadena de formato. Este primer parámetro indica en qué archivo (nombre interno dentro del programa) va a hacerse la lectura. En el programa del ejemplo, la función **fscanf** se utilizó para leer **n** (tamaño del vector) y para leer cada una de las componentes del vector.

Supongamos que el vector tiene cuatro componentes y sus valores son 1.2, 1.8, 3 y 4. Como la lectura es secuencial, cualquiera de los siguientes tres ejemplos de archivo (hay muchos otros más) son adecuados.

Primer ejemplo del archivo **datos1**:

```
4
1.2 1.8 3 4
```

Segundo ejemplo del archivo **datos1**:

```
4      1.2
1.8
3          4
```

Tercer ejemplo del archivo **datos1**.

4

1.2

1.8

3 4 5.1

La función **fclose** permite cerrar los archivos de texto previamente abiertos. Algunas veces no es absolutamente necesario cerrar los archivos, pero es una buena costumbre de programación cerrar los archivos abiertos cuando finaliza su uso. El único parámetro para la función **fclose** es el nombre (interno) del archivo.

La escritura en un archivo se puede hacer con la función **fprintf**. Su manejo es semejante al de **printf**, pero tiene un parámetro adicional, antes de la cadena de formato. Este primer parámetro indica en qué archivo (nombre interno en el programa) va a hacerse la escritura.

Al tratar de abrir un archivo para escritura, en modo "w", pueden presentarse tres casos.

- Existe el archivo con ese nombre, entonces **fopen** borra el existente y crea uno nuevo, listo para empezar a escribir en él.
- El nombre de archivo es adecuado pero no existe, entonces **fopen** crea uno con ese nombre, listo para empezar a escribir en él.
- El nombre del archivo es inadecuado, entonces **fopen** devuelve el valor **NULL**.

El programa del ejemplo anterior hace uso de una función **promX**, que calcula el promedio de los primeros *n* elementos de un arreglo. Puede ser cualquiera de las cuatro funciones **promX** del capítulo 5. Obviamente hay que incluirla dentro del programa, lo mismo que su prototipo.

En el programa del ejemplo, el archivo de lectura se llama **datos1** y el archivo de escritura se llama **ejemplo.res**. Si se quiere modificar los datos, basta con editar el archivo **datos1**, efectuar los cambios necesarios y correr de nuevo el programa. En algunos casos esto puede ser útil.

Pero en otros casos puede ser más flexible dar al programa, durante su ejecución, los nombres de los archivos que van a ser usados.

El siguiente programa, modificación del anterior, permite que durante la ejecución reciba los nombres de los archivos.

```
// Lectura y escritura en archivos
// lectura de los nombres de los archivos

#include...

int main()
{
    // lectura de un vector en un archivo
    // escritura del promedio en otro archivo

    double *x, prom;
    FILE *archDat, *archRes;
    char nombre[41];
    int n, i;

    printf("\n Nombre del archivo con los datos: ");
    gets(nombre);
    archDat = fopen(nombre, "r");
    if( archDat == NULL ){
        printf("\n\n Archivo inexistente.\n\n");
        exit(1);
    }
    fscanf(archDat, "%d", &n);
    x = new double[n];
    if( x == NULL ){
        printf("\n\n Memoria insuficiente.\n\n");
        exit(1);
    }
    for( i=0; i<n; i++) fscanf(archDat, "%lf", &x[i]);
    fclose(archDat);

    printf("\n Nombre del archivo para resultados: ");
    gets(nombre);
    archRes = fopen(nombre, "w");
}
```

```

if( archRes == NULL ){
    printf("Nombre de archivo de resultados erroneo.\n");
    exit(1);
}
prom = promX(x, n);
fprintf(archRes, " promedio = %12.4lf\n", prom);
fclose(archRes);
return 0;
}

```

La función **fscanf**, de manera análoga a la función **scanf**, devuelve un valor entero que indica el número de campos bien leídos y almacenados. Si no hubo almacenamiento, **fscanf** devuelve 0; si encuentra el final del archivo, devuelve EOF (generalmente EOF es lo mismo que -1).

7.2 feof

En los dos ejemplos anteriores, el primer dato indicaba el número de valores que el programa debía leer. En algunos casos se sabe que un archivo contiene valores y es necesario leerlos, pero no se conoce por anticipado el número de valores. Entonces el programa debe leer hasta que encuentre el final del archivo. Para esto se utiliza la función **feof** (*end of file*). Esta función tiene como parámetro un archivo (un apuntador a tipo FILE). Devuelve un valor no nulo si se ha alcanzado el fin de archivo, y 0 en caso contrario.

Suponga que es necesario calcular el promedio y la desviación estándar de los valores contenidos en un archivo. La definición de desviación estándar

$$\sigma = \left(\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n} \right)^{1/2}$$

indicaría que es necesario conocer primero el promedio, \bar{x} , para poder calcular los valores $(x_i - \bar{x})^2$, es decir, se necesitaría leer dos veces el archivo, la primera para poder calcular el promedio y la segunda para los valores $(x_i - \bar{x})^2$. Otra posibilidad consistiría en almacenar todos los valores en un arreglo y después hacer todos los cálculos necesarios. Esto implicaría el uso de mucha memoria, si el archivo es muy grande. Finalmente, una solución mucho más eficiente consiste en utilizar otra

definición equivalente de desviación estándar:

$$\sigma = \left(\frac{\sum_{i=1}^n x_i^2 - n\bar{x}^2}{n} \right)^{1/2}.$$

El siguiente programa lee valores numéricos en un archivo, hasta que encuentra el fin de archivo o hasta que detecta problemas en la lectura mediante `fscanf`. Calcula el promedio y la desviación estándar de los valores leídos.

```
// Calculo de la desviacion estandar
// Utilizacion de feof

#include...

int main()
{
    double xi, prom, sumax, sumaxx, desvEst;
    FILE *archDat, *archRes;
    char nombre[41];
    int n, fin, res;

    printf("\n Nombre del archivo con los datos: ");
    gets(nombre);
    archDat = fopen(nombre, "r");
    if( archDat == NULL ){
        printf("\n\n Archivo inexistente.\n\n");
        exit(1);
    }

    printf("\n Nombre del archivo para resultados: ");
    gets(nombre);
    archRes = fopen(nombre, "w");
    if( archRes == NULL ){
        printf("Nombre de archivo de resultados erroneo.\n");
        exit(1);
    }
    sumax = 0.0;
    sumaxx = 0.0;
    n = 0;
```

```

fin = 0;
while( !fin ){
    res = fscanf(archDat, "%lf", &xi);
    if( feof(archDat) || res <= 0 ) fin = 1;
    else{
        fprintf(archRes, "%12.6lf\n", xi);
        n++;
        sumax += xi;
        sumaxx += xi*xi;
    }
}
fclose(archDat);

if( n > 0 ){
    prom = sumax/n;
    desvEst = sqrt( (sumaxx - n*prom*prom)/n );
    fprintf(archRes, "promedio = %12.4lf\n", prom);
    fprintf(archRes, "desviacion estandar = %12.4lf\n",
            desvEst);
}
else {
    fprintf(archRes, " No hay numeros.\n");
}
fclose(archRes);
return 0;
}

```

Si en el archivo solamente hay números bien escritos, el programa calcula su promedio y desviación estándar. Si antes del fin de archivo la función `fscanf` detecta errores en la lectura, entonces el programa calcula el promedio y desviación estándar de los valores leídos hasta ese momento.

7.3 Algunos ejemplos

A continuación aparecen varios ejemplos de funciones de lectura y de escritura en archivos, de vectores y de matrices almacenadas en arreglos unidimensionales o mediante el uso de apuntadores dobles.

```
int flectX(FILE *arch, double *x, int n)
```

```

{
    // Lectura en un archivo de los primeros n
    // elementos de un arreglo x.

    // Devuelve 1 si se efectuo la lectura normalmente,
    //          0 si se encontro fin de archivo antes
    //          de acabar la lectura,
    //         -1 si hubo un error durante la lectura
    //          con fscanf.

    int i, res;

    for(i=0; i< n; i++){
        res = fscanf(arch, "%lf", &x[i]);
        if( feof(arch) ) return 0;
        if( res <= 0 ) return -1;
    }
    return 1;
}
//-----
void fescrX(FILE *arch, double *x, int n )
{
    //escritura en un archivo de los elementos de un vector

    int j;
    int nEltosLin = 5; // numero de elementos por linea

    for( j = 0; j < n; j++){
        fprintf(arch, "%15.8lf", x[j]);
        if((j+1)%nEltosLin == 0 || j== n-1)
            fprintf(arch,"\\n");
    }
}

```

A continuación se presenta un ejemplo esquemático de la utilización de las dos funciones anteriores.

```

double *x;
FILE *archDat, *archRes;

```

```

int n, res;

...
archDat = fopen( ..., "r");
if( archDat == NULL ){
    printf("\n\n Archivo inexistente.\n\n");
    exit(1);
}
...
archRes = fopen( ..., "w");

fscanf(archDat, "%d", &n);
x = new double[n];
res = flectX(archDat, x, n);
switch( res ){
    case 1:
        fprintf(archRes," x :\n");
        fescrX(archRes, x, n);
        break;
    case 0:
        fprintf(archRes,"Fin de archivo imprevisto.\n");
        break;
    case -1:
        fprintf(archRes, "Error en lectura.\n");
        break;
    default:
        fprintf(archRes, "RARISIMO.\n");
        break;
}

```

Las siguientes funciones sirven para lectura y escritura en archivos de matrices en arreglos unidimensionales o mediante apuntadores dobles.

```

int flectA1(FILE *arch, double *a, int m, int n)
{
    // Lectura en un archivo de una matriz m x n
    // almacenada en un arreglo unidimensional

```

```

// Devuelve 1 si se efectuo la lectura normalmente,
//           0 si se encontro fin de archivo antes
//           de acabar la lectura,
//           -1 si hubo un error durante la lectura
//           con fscanf.

// usa : flectX

int i, res;

for(i=0; i< m; i++){
    res = flectX(arch, &a[i*n], n);
    if( res <= 0 ) return res;
}
return 1;
}

//-----
void fescrA1(FILE *arch, double *a, int m, int n)
{
    // Escritura en un archivo de una matriz m x n
    // almacenada en un arreglo unidimensional

    // usa : fescrX

    int i;

    for(i=0; i< m; i++) fescrX(arch, &a[i*n], n);
}

//-----
int flectA(FILE *arch, double **a, int m, int n)
{
    // Lectura en un archivo de una matriz m x n
    // almacenada mediante un apuntador doble.

    // Devuelve 1 si se efectuo la lectura normalmente,
    //           0 si se encontro fin de archivo antes
    //           de acabar la lectura,
    //           -1 si hubo un error durante la lectura
}

```

```

//           con fscanf.

// usa : flectX

int i, res;

for(i=0; i< m; i++){
    res = flectX(arch, a[i], n);
    if( res <= 0 ) return res;
}
return 1;
}

//-----
void fescrA(FILE *arch, double **a, int m, int n)
{
    // Escritura en un archivo de una matriz m x n
    // almacenada mediante un apuntador doble.

    // usa : fscrX

    int i;

    for(i=0; i< m; i++) fscrX(arch, a[i], n);
}

```

Ejercicios

Para cada uno de los enunciados siguientes, defina cuáles son los datos necesarios. Haga un programa que lea los datos en un archivo, llame la función que realiza los cálculos y finalmente escriba los resultados en un archivo.

- 7.1 Considere los ejercicios de los capítulos anteriores, especialmente aquellos en que el número de datos puede ser grande.
- 7.2 Considere un archivo que contiene únicamente números; por ejemplo:

3.141592 -342

```

1.5e-2

-32.1542
32.5E+02 +31 31.45 1000
0.31415E1

```

Haga un programa que detecte si hay errores. Si los hay, indique el número de la línea donde está el error (por lo menos el primero) y muestre el trozo de línea donde está el error. Si no hay errores, habiendo averiguado el número de números, asigne memoria dinámicamente y almacene los números. Sugerencia: lea cada línea del archivo como una cadena. En cada línea determine la subcadena correspondiente a un posible número y conviértala, si es posible, en un número.

- 7.3 Consideré un archivo de texto donde están los resultados de una encuesta. Defina la estructura del archivo; por ejemplo, de las columnas 1 a 10 hay un código, 11-12 edad, 13 sexo, 14-20 sueldo, 21-22 número de hijos, etc. Haga un programa que lea el archivo y encuentre las posibles inconsistencias, no solo de cada campo (edad, sueldo...) sino también entre campo y campo; por ejemplo, en las columnas 14-20 no debe haber letras, la edad no puede ser negativa, si la edad es 10 no puede tener 4 hijos...
- 7.4 Consideré un archivo con restricciones (igualdades o desigualdades) escrito de manera semejante al siguiente ejemplo:

```

2*escritorios + 3*sillas>=24
4.5*mesas+3.1*sillas <= 31.5

```

Defina las características que debe tener el archivo. Haga un programa que lea el archivo, detecte si está bien hecho y escriba en otro archivo el número de variables, el número de restricciones, la lista de las variables y las restricciones por medio de una matriz A , un vector de tipos de restricciones y un vector b de términos independientes.

8

Temas varios

Este capítulo contiene varios temas sueltos, que no son indispensables pero pueden ser muy útiles en algunos casos.

8.1 sizeof

La función `sizeof` sirve principalmente para conocer el número de bytes que utiliza un compilador específico para los diferentes tipos de datos. El siguiente ejemplo muestra su uso y los resultados con un compilador y en un computador específico. Con otro compilador o en otra clase de computador los resultados pueden ser diferentes.

```
// sizeof
#include...
//-----
int main()
{
    double *x, y[10], a[10][10], *p;

    x = new double[20];

    printf("Tama~nos(bytes) de los diferentes tipos.\n\n");
    printf(" int       : %d \n", sizeof(int));
    printf(" float     : %d \n", sizeof(float));
    printf(" double    : %d \n", sizeof(double));
```

```

printf(" char      : %d \n\n", sizeof(char));
printf(" short int : %d \n", sizeof(short int));
printf(" long int  : %d \n", sizeof(long int));
printf(" unsigned   : %d \n", sizeof(unsigned));
printf(" long double: %d \n\n", sizeof(long double));

p = y;
printf(" arreglo    y : %d\n", sizeof(y) );
printf(" arreglo    a : %d\n", sizeof(a) );
printf(" apuntador x : %d\n", sizeof(x));
printf(" apuntador p : %d\n", sizeof(p));
return 0;
}

```

El programa anterior produce los siguientes resultados:

Tama~nos (bytes) de los diferentes tipos.

```

int      : 4
float    : 4
double   : 8
char     : 1

short int : 2
long int  : 4
unsigned  : 4
long double: 10

arreglo   y : 80
arreglo   a : 800
apuntador x : 4
apuntador p : 4

```

En este caso, en el mismo computador con el mismo compilador, los tipos **int** y **long int** tienen el mismo tama~o; entonces no se justifica utilizar variables o constantes tipo **long int**. Al utilizar **sizeof** con un arreglo, la respuesta es el n~mero total de bytes utilizados por el arreglo; por ejemplo, para los arreglos **y** y **a**. Los arreglos y los apuntadores

tienen, en general, una manejo casi idéntico, pero son en realidad diferentes. Observe que utilizar `sizeof` para `x`, un apuntador al que se le ha asignado memoria dinámicamente, no da como resultado el total de memoria asignada. Tampoco devuelve el valor “imaginado”, es decir 80, la utilización de `sizeof` con el apuntador `p` al que se le asignó el valor de `y`.

8.2 const

Si en la declaración de algunas variables, el especificador `const` precede al tipo, entonces el compilador sabe que esas variables no pueden ser modificadas en el programa. Más específicamente, no puede haber asignaciones donde el lado izquierdo sea una de esas variables. Además, hace más legible el programa fuente, al indicar que se trata de una constante.

```
const double c = 2.998E8, velSon = 331.0;
// velocidades de la luz y del sonido en m/s
const int nMax = 100;
```

El compilador producirá un mensaje de error si posteriormente se trata de hacer una asignación. El mensaje de error puede ser análogo a “Error...: Cannot modify a `const` object in function `main()`”.

```
c *= 1.001;
nMax += 0;
```

Observe que realmente `nMax += 0` no trata de modificar la variable `nMax`, pero para el compilador es un error por estar en el lado izquierdo de una asignación.

8.3 typedef

La utilización de `typedef` simplemente introduce un sinónimo para un tipo de datos. El esquema de su uso es el siguiente:

`typedef tipo sinónimo;`

Por ejemplo,

```
typedef int entero;
```

permite usar **entero** en lugar de **int** en las declaraciones de tipo de las variables, como la siguiente declaración:

```
entero i, j = 4, m;
```

En el siguiente ejemplo, referente al épsilon de la máquina, se usa la palabra clave **typedef**. La representación de un número real en el computador tiene un número finito de cifras significativas. Para conocer un valor aproximado del número de cifras significativas se utiliza el épsilon de la máquina.

$$\varepsilon_{\text{maq}} = \min\{\varepsilon > 0 : 1 + \varepsilon > 1\}.$$

En la definición anterior, la comparación $1 + \varepsilon > 1$ no se hace en el sentido matemático estricto; se hace según la interpretación del computador, es decir, muy probablemente para el computador, en lenguaje C, $1 + .001 > 1$, pero $1 + 10^{-100}$ lo considera igual a 1. Una manera aproximada de calcular ε_{maq} consiste en empezar con $\varepsilon = 1$ e ir disminuyendo su valor hasta que no se detecte diferencia entre $1 + \varepsilon$ y 1. El siguiente ejemplo calcula una aproximación de ε_{maq} cuando se trabaja con números doble precisión.

```
// uso de typedef

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "misfunc.h"
#include <iostream.h>

typedef double numero;

numero epsMaq(void);
//=====
int main()
{
    cout<<" eps de la maquina = "<<epsMaq()<<endl;
    return 0;
}
```

```

}

//=====
numero epsMaq(void)
{
    // calculo aproximado del epsilon de la maquina

    const numero uno = 1.0;
    const numero divisor = 2.0;
    numero eps, unoEps;

    eps = 1.0;
    unoEps = uno+eps;
    while( unoEps > uno ){
        eps /= divisor;
        unoEps = uno+eps;
    }
    eps *= divisor;

    return eps;
}

```

El resultado puede ser `2.22045e-16`. Esto indica que los números doble precisión utilizan aproximadamente 16 cifras significativas.

Si se desea calcular el épsilon de la máquina usando números de precisión sencilla, basta con cambiar únicamente la línea `typedef`, para que quede

```
typedef float numero;
```

En este caso el resultado puede ser `1.19209e-07`, lo que indica que los números de precisión sencilla utilizan aproximadamente siete cifras significativas.

8.4 include

En un programa fuente, además de las órdenes y controles correspondientes al lenguaje C o C++, hay otras órdenes o directrices para el preprocesador, quien las realiza antes de la compilación. Las directrices para el preprocesador, una únicamente por línea, empiezan con `#`.

Cuando aparece en el programa fuente `#include <nombrearchivo>`, entonces el preprocesador reemplaza esta línea por el archivo completo. Su uso puede ser de dos formas:

```
#include <nombrearchivo>

#include "nombrearchivo"
```

En el primer caso, es un archivo de cabecera estándar o ha sido creado por el programador, pero debe estar donde se encuentran los archivos de cabecera.

En el segundo caso, el archivo debe estar en el directorio o carpeta de trabajo, es decir, donde está el programa fuente. Si no está en la misma carpeta donde está el programa fuente, entonces es necesario indicar la dirección exacta del archivo. Generalmente el nombre de estos archivos tiene al final `.h` o `.hpp`.

Cuando se trata de un proyecto pequeño, un archivo de cabecera creado por el programador tiene generalmente los prototipos de las funciones y después las definiciones de las funciones. También puede tener otras directrices para el preprocesador.

El siguiente es un ejemplo de un archivo de cabecera para una pequeña biblioteca creada por el programador. Supongamos que el nombre de este archivo es `mibibl.h`.

```
double min( double a, double b);
double max( double a, double b);

double min( double a, double b)
{
    // minimo de {a, b}

    if( a <= b ) return a;
    else return b;
}
//-----
double max( double a, double b)
{
    // maximo de {a, b}
```

```

    if( a >= b ) return a;
    else return b;
}

```

El siguiente ejemplo, muy sencillo, calcula el mínimo y el máximo de tres números. Usa el archivo de cabecera **mibibl.h**.

```

// minimo y maximo de 3 numeros

#include <iostream.h>
#include "mibibl.h"

int main()
{
    double a, b, c;

    cout<<"\n\n Minimo y maximo de 3 numeros.\n\n";
    cout<<" entre los 3 numeros : ";
    cin>>a>>b>>c;
    cout<<a<<" "<<b<<" "<<c<<endl;
    cout<<" minimo = "<<min(a, min(b, c) )<<endl;
    cout<<" maximo = "<<max(a, max(b, c) )<<endl;
    return 0;
}

```

Para proyectos de mayor tamaño, el archivo de cabecera, un archivo **.h**, tiene únicamente los prototipos de las funciones. En otro archivo, un archivo **.c** o **.cpp**, están las definiciones de las funciones. Este último, cuando ya está depurado, se compila una sola vez y está listo para el enlace o *link*. El resultado es un archivo **.obj**. El archivo donde está la función **main** tiene un **include** para el archivo **.h**, no para el archivo donde están las definiciones. Así, cuando se compila el programa principal (lo cual se hace muchas veces), éste incluye el archivo **.h**, muy pequeño, y después se hace el enlace con el **.obj** que está listo. Si el **include** abarcara, en un todo, los prototipos y las definiciones, cada vez que se compila el programa principal habría que recompilar las definiciones de las funciones.

8.5 define

La directriz **define** tiene dos usos. El primero sirve para definir constantes. Por tradición estas constantes se escriben en mayúsculas. Por ejemplo:

```
#define PI 3.14159265358979
```

Cuando el compilador encuentra el identificador definido por medio de la directriz **define**, lo reemplaza por la cadena asociada. En **define** se puede usar algo ya definido; por ejemplo,

```
#define PI 3.14159265358979
#define PI2 2.0*PI
```

Observe que, como en las otras directrices para el preprocesador, no hay punto y coma al final de la instrucción. Actualmente se prefiere el uso de **const** que explicita el tipo.

```
const double PI = 3.14159265358979;
```

El segundo uso de **define** permite definir macros. Su forma general es la siguiente:

```
#define identificador(identificador, ..., identificador) cadena
```

El anterior esquema es poco diciente. Los dos ejemplos siguientes muestran mejor el uso de **define**.

```
#include...
#define CUADRADO(x) (x)*(x)
#define CUADRADOB(x) x*x
#define NORMA(x, y) (sqrt((x)*(x)+(y)*(y)))

int main()
{
    double x = 3.0, y = 4.0;

    cout<<CUADRADO(x)<<endl;
    cout<<CUADRADOB(x)<<endl;
```

```

cout<<CUADRADO(x+y)<<endl;
cout<<CUADRADOB(x+y)<<endl;
cout<<NORMA(x, y)<<endl;
return 0;
}

```

Los resultados producidos son:

```

9
9
49
19
5

```

En la definición de CUADRADO, aparentemente, hay muchos paréntesis; parece más natural definir como se hace en CUADRADOB. Los dos primeros resultados coinciden, lo cual es absolutamente normal. Sin embargo, el tercero y el cuarto no coinciden, pero en apariencia debían coincidir. La razón es la siguiente: cuando el compilador encuentra CUADRADOB(x+y), lo reemplaza por $x+y*x+y$, diferente de lo esperado, es decir de $(x+y)*(x+y)$. Ahora es comprensible el resultado 19. Algunos autores recomiendan, por seguridad, empezar y terminar la cadena con paréntesis, como en NORMA. Entonces CUADRADO quedaría así:

```
#define CUADRADO(x) ((x)*(x))
```

8.6 Apuntadores a funciones

Antes de ver exactamente la utilidad y la manera de usar los apuntadores a funciones, veamos un ejemplo de una función que efectúa el cálculo aproximado de la integral definida por la fórmula del trapecio. Su uso se puede mejorar y generalizar por medio de apuntadores a funciones.

Sea

$$I = I(f, a, b) = \int_a^b f(x) dx.$$

El valor de I se puede aproximar así:

$$I \approx h \left(\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(b)}{2} \right), \quad \text{donde } h = \frac{b-a}{n}, \quad x_i = a + ih.$$

Consideré el siguiente programa:

```
// formula del trapecio para integrales,
// SIN apuntadores a funciones.

#include...

double trapecio0( double a, double b, int n);
double f( double x);
double g( double x);
=====
int main()
{
    double a, b;
    int n;

    cout<<"\n\n Calculo de la integral definida"
        << " por medio de la formula del trapecio\n\n";
    cout<<" entre a   b   n : ";
    cin>>a>>b>>n;
    cout<<"\n\n integral  ~=  "<<trapecio0(a, b, n);
    return 0;
}
=====
double trapecio0( double a, double b, int n)
{
    // Calculo de la integral definida de la funcion  f(x)
    // en el intervalo [a,b], utilizando  n  subintervalos.
    // f(x) esta definida en la funcion  f

    double s, h;
    int i;

    if( n <= 0 ){

```

```

cout<<"\n\n trapecio0: ERROR: n = "<<n<<endl;
return 0.0;
}
s = (f(a) + f(b))/2.0;
h = (b-a)/n;
for( i = 1; i <= n-1; i++) s += f( a + i*h );
return s*h;
}
//-----
double f( double x)
{
    return exp(-x*x);
}
//-----
double g( double x)
{
    const double DOSPI = 6.283185307179586;
    return exp(-x*x/2.0)/sqrt(DOSPI);
}

```

El programa de este ejemplo calcula una aproximación de I para la función definida en `double f(...)`, pero no puede calcular una aproximación de I para la función definida en `double g(...)`. De manera natural se desea que uno de los parámetros de `trapecio0` sea la función. Esto se logra mediante los apuntadores a funciones. Veamos, mediante una modificación del ejemplo anterior, su uso.

```

// formula del trapecio para integrales,
// CON apuntadores a funciones.

#include...

double trapecio( double (*f)(double x), double a, double b,
    int n);
double f1( double x);
double f2( double x);
//=====
int main()

```

```

{
    double a, b;
    int n;

    cout<<"\n\n Calculo de la integral definida"
        <<" por medio de la formula del trapecio\n\n";
    cout<<" entre a   b   n : ";
    cin>>a>>b>>n;
    cout<<"\n\n integral de f1 ~= "<<trapecio(f1, a, b, n);
    cout<<"\n\n integral de f2 ~= "<<trapecio(f2, a, b, n);
    return 0;
}

//=====
double trapecio(double (*f)(double x), double a, double b,
    int n)
{
    // Calculo de la integral definida de la funcion  f(x)
    // en el intervalo [a,b], utilizando  n  subintervalos.
    // f(x) esta definida en la funcion  f

    double s, h;
    int i;

    if( n <= 0 ){
        cout<<"\n\n trapecio: ERROR: n = "<<n<<endl;
        return 0.0;
    }
    s = ( (*f)(a) + (*f)(b) )/2.0;
    h = (b-a)/n;
    for( i = 1; i <= n-1; i++) s += (*f)( a + i*h );
    return s*h;
}
//-----
double f1( double x)
{
    return exp(-x*x);
}
//-----

```

```

double f2( double x)
{
    const double DOSPI = 6.283185307179586;

    return exp(-x*x/2.0)/sqrt(DOSPI);
}

```

Este ejemplo muestra las principales características del uso de los apuntadores a funciones. En la definición de la función `trapecio`, un parámetro es un apuntador a función. El identificador está precedido de asterisco y aparece entre paréntesis: `(*f)`. Además están el tipo devuelto por la función y los parámetros: `double (*f)(double x)`. Esto implica que `trapecio` se puede usar para cualquier función que devuelva un número doble precisión y que tenga un único parámetro tipo doble precisión. Cuando se utiliza el apuntador dentro de `trapecio` para evaluar la función en un valor específico, también está precedido de asterisco y encerrado entre paréntesis: `(*f)(a)`.

8.7 Funciones en línea

Cuando una función, por ejemplo la función `main`, llama a otra, hay un pequeño incremento en el tiempo de ejecución correspondiente al llamado en sí de la función y a la pasada de los parámetros. Este incremento de tiempo se conoce como *overhead*, que se puede traducir por gastos generales o sobrecosto. C++ permite utilizar funciones en línea cuyo objetivo es que el compilador coloque una copia de la función en línea en cada sitio del programa donde se utiliza la función. Así, no hay paso de parámetros y se gana en eficiencia, pero usualmente el programa ejecutable resulta más grande. Las funciones en línea son aconsejables para funciones pequeñas, de una o dos instrucciones.

El ejemplo siguiente tiene dos partes. En la primera se hace uso de la función `cosGrd`, función común y corriente, que calcula el coseno, cuando el ángulo está dado en grados. En la segunda parte se hace uso de una función en línea con el mismo objetivo.

```

// coseno de un angulo en grados,
// SIN funciones en linea.

```

```
#include <math.h>
#include <iostream.h>

double cosGrd( double x);
//=====
int main()
{
    double a;

    cout<<"\n\n Calculo de cos(a) , a en grados.\n\n";
    cout<<" a = ";
    cin>>a;
    cout<<"\n\n cos("<<a<<") = "<<cosGrd(a);
    return 0;
}
//=====
double cosGrd( double x)
{
    return cos(0.01745329251994*x);
}



---


// coseno de un angulo en grados,
// CON una funcion en linea.

#include <math.h>
#include <iostream.h>

inline double cosGrd(double x) return cos(.017453292519*x);}
//=====
int main()
{
    double a;

    cout<<"\n\n Calculo de cos(a) , a en grados.\n\n";
    cout<<" a = ";
    cin>>a;
    cout<<"\n\n cos("<<a<<") = "<<cosGrd(a);
```

```
    return 0;
}
```

Es usual que una función en línea esté escrita en una única línea, pero no es obligatorio. La función en línea `cosGrd` se hubiera podido escribir así:

```
inline double cosGrd(double x)
{
    // calculo del coseno de un angulo en grados

    return cos(0.0174532925199433*x);
}
```

Otro ejemplo de una función en línea puede ser el siguiente, que calcula el máximo entre dos números doble precisión.

```
inline double max(double x, double y)
{
    if( x >= y ) return x;
    else return y;
}
```

En varios libros el ejemplo clásico de funciones en línea corresponde a una función que hace el mismo trabajo que la anterior, pero está escrita de una forma mucho más compacta. El lector interesado debe consultar en un manual de C sobre el uso del operador ?, pues esta función hace uso de él.

```
inline double max(double x, double y){return((x>y) ? x:y);}
```

8.8 Argumentos de la función `main`

Hasta ahora se había supuesto que la función `main` no tenía argumentos. En realidad, sí puede tenerlos. Estos argumentos se usan en la línea de comandos donde se da la orden de ejecución al programa; por ejemplo en DOS o en Unix.

Supongamos que el programa se llama `prog28`. Normalmente, para comenzar la ejecución del programa, es necesario digitar `prog28` y

después oprimir la tecla Enter. Supongamos además que, empezando el programa, éste pregunta por el nombre del archivo de datos y el nombre del archivo para resultados. En lugar de hacer lo anterior, se puede escribir enseguida de **prog28**, antes de oprimir Enter, el nombre del archivo de datos y el nombre del archivo para los resultados; por ejemplo,

```
prog28 caso1.dat result
```

Los parámetros para **main** son **argc** y **argv**. El primero es de tipo entero y sirve para contar el número de palabras (o cadenas) en la línea de comandos. El nombre mismo del programa se cuenta como una palabra. En el ejemplo **prog28 caso1.dat result** hay tres palabras. El parámetro **argv** es un arreglo de cadenas donde se almacenan las cadenas dadas en la línea de comandos. El siguiente ejemplo muestra un esquema de su uso.

```
// prog28
// Parametros en la linea de comandos,
// uso de argc, argv

#include...

int main( int argc, char *argv[] )
{
    FILE *archDat, *archRes;

    if( argc != 3 ){
        cout<<"ERROR: el llamado debe ser:\n"
            <<"prog28 nombre_arch_datos nombre_arch_result"
            <<endl;
        exit(1);
    }

    archDat = fopen(argv[1], "r");
    if( archDat == NULL ){
        printf("\n\n Archivo inexistente.\n\n");
        exit(1);
    }
```

```
archRes = fopen(argv[2], "w");
if( archRes == NULL ){
    printf("Nombre de archivo de resultados erroneo.\n");
    exit(1);
}

...
fclose(archDat);
fclose(archRes);
return 0;
}
```

Como `argv` es un arreglo de cadenas, entonces empieza en el subíndice cero. Supongamos que el llamado se hace

```
prog28 caso1.dat result
```

entonces la primera palabra en la línea de comandos será `argv[0]`, o sea la cadena "prog28". La segunda palabra en la línea de comandos, la cadena "caso1.dat", quedará almacenada en `argv[1]` y finalmente la cadena "result" quedará almacenada en `argv[2]`.

Otros temas de C que no se presentan en este libro son: enumeraciones, uniones, el operador ?, `static`, `extern`. El lector interesado podrá encontrarlos en un manual de C y C++.

9

Estructuras

Una estructura es un agrupamiento de variables, posiblemente de diferentes tipos, que se denomina con un solo nombre y constituye un nuevo tipo de datos.

9.1 Un ejemplo con complejos

El siguiente ejemplo define una estructura para los números complejos. Tiene dos elementos: la parte real y la parte imaginaria.

```
struct complejo {  
    double pReal; // parte real  
    double pImag; // parte imaginaria  
};
```

La palabra clave **struct** dice que se trata de una estructura. La palabra **complejo** es el nombre de este nuevo tipo de estructura (o simplemente tipo de dato). Siguiendo con el ejemplo, ahora es permitido:

```
struct complejo z, w, c;
```

La instrucción anterior es de C y es reconocible por cualquier compilador de C o de C++. En compiladores modernos de C++ se puede escribir sencillamente:

```
complejo z, w, c;
```

Para tener acceso a un elemento de una variable del tipo de la nueva estructura es necesario dar el nombre de la variable seguido de punto y del nombre del elemento; por ejemplo, `z.pImag`. El siguiente programa muestra realiza el producto de dos complejos.

```
// Estructuras, ejemplo con complejos.
//-----
#include <stdlib.h>
#include <stdio.h>
//-----
struct complejo {
    double pReal; // parte real
    double pImag; // parte imaginaria
};
//=====
int main( void )
{
    complejo z, w, c;

    z.pReal = 1.0;
    z.pImag = 1.0;

    w.pReal = 1.0;
    w.pImag = -1.0;

    c.pReal = z.pReal*w.pReal - z.pImag*w.pImag;
    c.pImag = z.pReal*w.pImag + z.pImag*w.pReal;

    printf(" parte real = %lf  parte imag. = %lf\n",
           c.pReal, c.pImag);
    return 0;
}
```

Una vez definida una estructura, ésta puede servir para describir un parámetro de una función, para declarar arreglos cuyos elementos sean estructuras, para declarar apuntadores, para asignar memoria dinámicamente, para que una función devuelva una estructura.

El siguiente ejemplo usa la estructura `complejo` definida anteriormente. El programa lee en un archivo el valor de `n`. Asigna dinámicamente

mente memoria para n estructuras **complejo**, lee en el archivo n parejas de valores (correspondientes a la parte real y la parte imaginaria) y las guarda en el arreglo de complejos, busca en el arreglo el complejo de mayor norma y finalmente escribe el cuadrado de ese complejo.

```
// Estructuras pasadas como parametros,
// arreglos de estructuras,
// funciones que devuelven una estructura.
//-----
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
//-----
struct complejo {
    double pReal; // parte real
    double pImag; // parte imaginaria
};
//-----
complejo prodCompl( complejo z1, complejo z2);
int flectXCompl(FILE *arch, complejo *x, int n);
double normaCompl(complejo x);
complejo complMaxNorma(complejo *x, int n);
//=====
int main( void )
{
    // Lee en un archivo n.
    // Lee en el archivo n complejos. Para leer un
    // complejo lee la parte real y la parte imaginaria.
    // Busca el complejo de mayor norma y escribe su
    // cuadrado.

    complejo *z, w;
    int n, resLect;
    FILE *archDat;
    char nombre[51];

    printf("\n Nombre del archivo con los datos: ");
    gets(nombre);
```

```

archDat = fopen(nombre, "r");
if( archDat == NULL ){
    printf("\n\n Archivo inexistente.\n\n");
    exit(1);
}

fscanf(archDat, "%d", &n);
z = new complejo[n];
if( z == NULL ){
    printf("\n\n Memoria insuficiente.\n\n");
    exit(1);
}

resLect = flectXCompl(archDat, z, n);
if( resLect != 1 ){
    printf(" ERROR leyendo archivo.\n");
    exit(1);
}

w = complMaxNorma(z, n);
printf(" parte real = %lf  parte imag. = %lf\n",
       w.pReal, w.pImag);

w = prodCompl(w, w);
printf(" parte real = %lf  parte imag. = %lf\n",
       w.pReal, w.pImag);
return 0;
}
=====

complejo prodCompl( complejo z1, complejo z2)
{
    // producto de dos complejos

    complejo z3;

    z3.pReal = z1.pReal*z2.pReal - z1.pImag*z2.pImag;
    z3.pImag = z1.pReal*z2.pImag + z1.pImag*z2.pReal;
}

```

```

    return z3;
}
//-----
int flectXCompl(FILE *arch, complejo *x, int n)
{
    // Lectura en un archivo de los primeros n
    // elementos de un arreglo complejo x.

    // Devuelve 1 si se efectuo la lectura normalmente,
    //          0 si se encontro fin de archivo antes
    //          de acabar la lectura,
    //         -1 si hubo un error durante la lectura
    //          con fscanf.

    int i, res;

    for(i=0; i< n; i++){
        res = fscanf(arch, "%lf", &x[i].pReal);
        if( feof(arch) ) return 0;
        if( res <= 0 ) return -1;
        res = fscanf(arch, "%lf", &x[i].pImag);
        if( feof(arch) ) return 0;
        if( res <= 0 ) return -1;
    }
    return 1;
}
//-----
complejo complMaxNorma(complejo *x, int n)
{
    // Devuelve el complejo de maxima norma
    // en el arreglo de complejos x.

    double normaMax = 0.0, normai;
    int i, imax = -1;

    for( i=0; i < n; i++){
        normai = normaCompl( x[i] );
        if( normai > normaMax ){

```

```

        imax = i;
        normaMax = normai;
    }
}

return x[imax];
}
//-----
double normaCompl(complejo x)
{
    // Norma del complejo x

    return sqrt(x.pReal*x.pReal + x.pImag*x.pImag);
}

```

Una vez definida la estructura `complejo`, ésta se puede usar de varias maneras. En el programa anterior se observa:

- Los parámetros de las funciones pueden ser del tipo `complejo`: funciones `flectXCompl`, `normaCompl`, `complMaxNorma`, `prodCompl`.
- Las funciones pueden devolver un `complejo`: funciones `prodCompl`, `complMaxNorma`.
- La variable `z` en la función `main` es un apuntador a `complejo`. Un parámetro de la función `flectXCompl` es un apuntador a `complejo`.
- En la función `main`, utilizando `z`, apuntador a `complejo`, se hace una asignación dinámica de memoria para `n` elementos tipo `complejo`.
- En la función `flectXCompl` el segundo parámetro, `x`, es un “vector” de elementos tipo `complejo`. Para el `i`-ésimo elemento, el `complejo` `x[i]`, su parte real es `x[i].pReal` y su parte imaginaria es `x[i].pImag`.

Cuando se tiene un apuntador a una estructura, si se quiere acceder directamente a uno de sus campos, se usa el operador `->` de la siguiente manera: `apuntador->campo`.

A modo de ejemplo, la función `normaCompl` anterior, se reescribe a continuación utilizando `->`.

```

double normaComplb(complejo x)
{
    // Norma del complejo x

    complejo *w;

    w = &x;
    return sqrt(w->pReal*w->pReal + w->pImag*w->pImag);
}

```

9.2 Un ejemplo típico

En muchos libros de C el ejemplo típico de estructura es semejante al siguiente. Corresponde a la información de un empleado de una compañía.

```

struct empleado{
    char nombres[20];
    char apell[20];      // apellidos
    char docIdent;       // clase de documento de identidad
    int numIdent;        // numero de doc. identidad
    char dirLini[30];    // primera linea de la direccion
    char dirLin2[30];   // segunda linea de la direccion
    char CiudDept[30];  // ciudad y departamento
    char tel[10];        // numero de telefono
    char dirElec[30];   // direccion electronica: e-mail
    float sueldo;
    char fechaNac[8];   // fecha de nacimiento
    char fechaIng[8];   // fecha de ingreso
    char cargo[20];
};

```

Ejercicios

Los ejercicios de este capítulo se pueden hacer definiendo explícitamente una estructura por medio de `struct`, o sin su uso. Haga o estudie las dos alternativas y detecte ventajas y desventajas.

9.1 Defina una estructura para el manejo de números fraccionarios y haga funciones para:

- simplificar,
- sumar,
- multiplicar,
- restar,
- dividir,
- suma de las coordenadas (fraccionarias) de un vector,
- producto de las coordenadas (fraccionarias) de un vector,

En todos los casos controle que numerador y denominador permanecen dentro de los rangos permitidos.

9.2 Un grafo se puede representar por una lista de vértices y una lista de arcos. Una red es un grafo donde adicionalmente cada arco tiene un valor (costo o distancia). Defina una estructura para su manejo. Haga funciones para:

- agregar un arco con costo,
- quitar un arco,
- quitar un vértice de la lista de arcos,
- quitar un vértice de la lista de arcos y de la lista de vértices,
- modificar un costo,
- buscar si un arco existe,
- dada una sublista de vértices, averiguar si es un camino,
- calcular su costo.

10

Algunas funciones elementales

En este capítulo hay algunas funciones elementales que serán útiles especialmente para el manejo de matrices, en particular para la solución de sistemas de ecuaciones. Sean x , y vectores (cuando sea necesario se consideran como matrices columna), α , k escalares. La tabla siguiente muestra el nombre de la función y su efecto. Se usa la misma convención de C, el signo igual indica asignación.

<code>prodXY</code>	producto $x^T y$
<code>dist2</code>	$\ x - y\ _2 = \left(\sum_{i=1}^n (x_i - y_i)^2 \right)^{\frac{1}{2}}$
<code>alfaX</code>	$x = \alpha x$
<code>xMasAlfaY</code>	$x = x + \alpha y$
<code>xIglY</code>	$x = y$
<code>xIglAlfY</code>	$x = \alpha y$
<code>xIglK</code>	$x = k$, es decir, $x_i = k \forall i$
<code>intercXY</code>	intercambio de x , y
<code>maxX</code>	$\max_i\{x_i\}$
<code>maxXPos</code>	$\max_i\{x_i\}$ e indica la posición
<code>maxAbsX</code>	$\max_i\{ x_i \}$
<code>maxAbsXPos</code>	$\max_i\{ x_i \}$ e indica la posición
<code>ordBurb</code>	ordenar, de menor a mayor, los x_i

Para cada una de ellas, está la versión sin saltos y la versión con saltos. Utilizando la sobrecarga de funciones permitida por C++, las dos versiones tienen el mismo nombre.

10.1 Código de algunas funciones

A continuación se presenta el código en C (y algo de C++) de las funciones para vectores almacenados de manera consecutiva (sin salto). Para vectores almacenados con salto, es decir, en $x[0]$, $x[salto]$, $x[2*salto]$, ..., $x[(n-1)*salto]$, está el código completo de algunas de ellas. El código de las demás se puede obtener en la página del autor. Actualmente la dirección es: www.matematicas.unal.edu.co/~hmora/. Si hay reorganización de las páginas de la Universidad, será necesario entrar a la página de la Universidad www.unal.edu.co ir a Sede de Bogotá, Facultad de Ciencias, Departamento de Matemáticas y página del autor.

```
double prodXY(double *x, double *y, int n);
double prodXY(double *x, int saltox, double *y, int saltoy,
    int n);

double dist2(double *x, double *y, int n);
double dist2(double *x, int saltox, double *y, int saltoy,
    int n);

void alfaX(double alfa, double *x, int n);
void alfaX(double alfa, double *x, int salto, int n);

void xMasAlfaY(double *x, double alfa, double *y, int n);
void xMasAlfaY(double *x, int saltox, double alfa,
    double *y, int saltoy, int n);

void xIglK(double *x, int n, double k);
void xIglK(double *x, int salto, int n, double k);

void xIglY(double *x, double *y, int n);
void xIglY(double *x, int saltox, double *y, int saltoy,
    int n);

void xIglAlfY(double *x, double alfa, double *y, int n);
void xIglAlfY(double *x, int saltox, double alfa,
    double *y, int saltoy, int n);
```

10.1. CÓDIGO DE ALGUNAS FUNCIONES

```
void intercXY(double *x, double *y, int n);
void intercXY(double *x, int saltox, double *y, int saltoy,
              int n);

double maxX(double *x, int n);
double maxX(double *x, int salto, int n);

double maxXPos(double *x, int n, int &posi);
double maxXPos(double *x, int salto, int n, int &posi);

double maxAbsX(double *x, int n);
double maxAbsX(double *x, int salto, int n);

double maxAbsXPos(double *x, int n, int &posi);
double maxAbsXPos(double *x, int salto, int n, int &posi);

void ordBurb(double *x, int n);
void ordBurb(double *x, int salto, int n);

//-----
double prodXY(double *x, double *y, int n)
{
    // producto "interno"  x.y

    double s = 0.0;
    double *pxi, *pyi, *pxn;

    pxi = x;
    pyi = y;
    pxn = x + n;
    while( pxi < pxn ) s += (*pxi++)*(*pyi++);
    return s;
}
//-----
double dist2(double *x, double *y, int n)
{
    // distancia euclideana (Holder, orden 2) entre x y

    double s = 0.0;
```

```

double *pxi, *pyi, *pxn, xiyi;

pxi = x;
pyi = y;
pxn = x + n;
while( pxi < pxn ){
    xiyi = (*pxi++) - (*pyi++);
    s += xiyi*xiyi;
}
return sqrt(s);
}

//-----
void alfaX(double alfa, double *x, int n)
{
    // x = alfa * x

    double *pxi, *pxn;

    pxi = x;
    pxn = x + n;
    while( pxi < pxn ) *pxi++ *= alfa;
}

//-----
void xMasAlfaY(double *x, double alfa, double *y, int n)
{
    // x = x + alfa * y

    double *pxi, *pxn, *pyi;

    pxi = x;
    pyi = y;
    pxn = x + n;
    while( pxi < pxn ) *pxi++ += alfa*(*pyi++);
}

//-----
void xIglK(double *x, int n, double k)
{
    // x = k
}

```

```
double *pxi, *pxn;

pxi = x;
pxn = x + n;
while( pxi < pxn ) *pxi++ = k;
}

//-----
void xIglY(double *x, double *y, int n)
{
    // x = y

    double *pxi, *pxn, *pyi;

    pxi = x;
    pyi = y;
    pxn = x + n;
    while( pxi < pxn ) *pxi++ = *pyi++;
}

//-----
void xIglAlfY(double *x, double alfa, double *y, int n)
{
    // x = alfa * y

    double *pxi, *pxn, *pyi;

    pxi = x;
    pyi = y;
    pxn = x + n;
    while( pxi < pxn ) *pxi++ = alfa*(*pyi++);
}

//-----
void intercXY(double *x, double *y, int n)
{
    // intercambiar x y

    double *pxi, *pxn, *pyi, t;
```

```

pxi = x;
pyi = y;
pxn = x + n;
while( pxi < pxn ){
    t = *pxi;
    *pxi++ = *pyi;
    *pyi++ = t;
}
//-----
double maxX(double *x, int n)
{
    // maximo xi

    double *pxi, *pxn, maxi = -1.0E100, xi;

    pxi = x;
    pxn = x + n;
    while( pxi < pxn ){
        xi = *pxi++;
        if( xi > maxi ) maxi = xi;
    }
    return maxi;
}
//-----
double maxXPos(double *x, int n, int &posi)
{
    // maximo xi
    // posi contendra la posicion del maximo

    double *pxi, *pxn, maxi = -1.0E100, xi;
    int i;

    posi = -1;
    pxi = x;
    pxn = x + n;
    i = 0;
    while( pxi < pxn ){

```

```

xi = *pxi++;
if( xi > maxi ){
    maxi = xi;
    posi = i;
}
i++;
}
return maxi;
}
//-----
double maxAbsX(double *x, int n)
{
    // maximo |xi| 

    double *pxi, *pxn, maxi = 0.0, axi;

    pxi = x;
    pxn = x + n;
    while( pxi < pxn ){
        axi = fabs(*pxi++);
        if( axi > maxi ) maxi = axi;
    }
    return maxi;
}
//-----
double maxAbsXPos(double *x, int n, int &posi)
{
    // maximo |xi|
    // posi contendra la posicion del maximo valor abs.

    double *pxi, *pxn, maxi = 0.0, axi;
    int i;

    posi = -1;
    pxi = x;
    pxn = x + n;
    i = 0;
    while( pxi < pxn ){

```

```

    axi = fabs(*pxi++);
    if( axi > maxi ){
        maxi = axi;
        posi = i;
    }
    i++;
}
return maxi;
}

```

10.2 Versiones con saltos

```

double dist2(double *x, int saltox, double *y, int saltoy,
            int n)
{
    // distancia euclideana (Holder, orden 2 ) entre x, y
    //
    // x[0], x[saltox], x[2*saltox], ..., x[(n-1)*saltox]
    // y[0], y[saltoy], y[2*saltoy], ..., y[(n-1)*saltoy]

    double s = 0.0;
    double *pxi, *pyi, *pxn, *pyn, xiyi;

    if( n < 0 ){
        printf("dist2: n negativo\n");
        return 0.0;
    }
    if( n==0 ) return 0.0;

    if( saltox == 1 ){
        if( saltoy == 1 ){
            // saltox = saltoy = 1
            pxi = x;
            pxn = x + n;
            pyi = y;
            while( pxi < pxn ){
                xiyi = (*pxi++) - (*pyi++);
                s += xiyi*xiyi;
            }
        }
    }
}

```

```

        }
    }
else{
    // saltox = 1; saltoy != 1
    pxi = x;
    pxn = x + n;
    pyi = y;
    while( pxi < pxn ){
        xiyi = (*pxi++) - (*pyi);
        s += xiyi*xiyi;
        pyi += saltoy;
    }
}
else{
    if( saltoy == 1 ){
        // saltox > 1; saltoy = 1
        pyi = y;
        pyn = y + n;
        pxi = x;
        while( pyi < pyn ){
            xiyi = (*pxi) - (*pyi++);
            s += xiyi*xiyi;
            pxi += saltox;
        }
    }
else{
    // saltox != 1; saltoy != 1
    pxi = x;
    pxn = x + n*saltox;
    pyi = y;
    while( pxi < pxn ){
        xiyi = (*pxi) - (*pyi);
        s += xiyi*xiyi;
        pxi += saltox;
        pyi += saltoy;
    }
}
}

```

```

    }
    return sqrt(s);
}
//-----
void alfaX(double alfa, double *x, int salto, int n)
{
    // x = alfa * x
    // x en x[0], x[salto], x[2*salto], ..., x[(n-1)*salto]

    double *pxi, *pxn;

    if( salto == 1 ){
        pxi = x;
        pxn = x + n;
        while( pxi < pxn ) *pxi++ *= alfa;
    }
    else{
        pxi = x;
        pxn = x + n*salto;
        while( pxi < pxn ){
            *pxi *= alfa;
            pxi += salto;
        }
    }
}
//-----
void xMasAlfaY(double *x, int saltox, double alfa,
                double *y, int saltoy, int n)
{
    // x = x + alfa y
    // x[0], x[saltox], x[2*saltox], ..., x[(n-1)*saltox]
    // y[0], y[saltoy], y[2*saltoy], ..., y[(n-1)*saltoy]

    // uso de apuntadores

    double *pxi, *pyi, *pxn, *pyn;

    if( n < 0 ){

```

```

printf("xMasAlfaY: n negativo\n");
return;
}
if( n==0 ) return;

if( saltox == 1 ){
    if( saltoy == 1 ){
        // saltox = saltoy = 1
        pxi = x;
        pxn = x + n;
        pyi = y;
        while( pxi < pxn ) *pxi++ += alfa*(pyi++);
    }
    else{
        // saltox = 1; saltoy != 1
        pxi = x;
        pxn = x + n;
        pyi = y;
        while( pxi < pxn ){
            *pxi++ += alfa*(pyi);
            pyi += saltoy;
        }
    }
}
else{
    if( saltoy == 1 ){
        // saltox > 1; saltoy = 1
        pyi = y;
        pyn = y + n;
        pxi = x;
        while( pyi < pyn ){
            *pxi += alfa*(pyi++);
            pxi += saltox;
        }
    }
    else{
        // saltox != 1; saltoy != 1
        pxi = x;
    }
}

```

```

pxn = x + n*saltox;
pyi = y;
while( pxi < pxn ){
    *pxi += alfa*(*pyi);
    pxi += saltox;
    pyi += saltoy;
}
}
}
}

```

10.3 Método burbuja

Para ordenar un arreglo, supongamos en orden creciente, existen numerosos métodos. Uno de ellos es el método burbuja, el cual cuenta con una buena combinación de sencillez y eficiencia para arreglos no muy grandes.

Supongamos que se tiene un vector $x \in \mathbb{R}^n$. En una primera iteración (primera pasada) se mira si x_1, x_2 están ordenados; si no lo están se intercambian. Después se mira x_2 y x_3 ; si no están ordenados se intercambian. Después se mira x_3 y x_4 ; si no están ordenados se intercambian. Finalmente, si x_{n-1} y x_n no están ordenados, se intercambian. Por convención, x_i indica el i -ésimo elemento de x después de la última modificación. Después de esta primera pasada el mayor elemento queda en la última posición.

En una segunda pasada se hace un proceso semejante, pero la última pareja que se compara es x_{n-2}, x_{n-1} . Al final de esta segunda pasada los dos últimos elementos están bien ordenados (en sus posiciones definitivas).

En una tercera pasada se repite el proceso, pero la última pareja que se compara es x_{n-3}, x_{n-2} . Al final de esta tercera pasada los tres últimos elementos están bien ordenados.

Si en una pasada no hay intercambios, el vector está ordenado.

Considere

$$x = (6, -1, 14, 0, 1, 2)$$

Como 6 y -1 no están ordenados, se intercambian.

$$x = (-1, 6, 14, 0, 1, 2)$$

Como 6 y 14 están ordenados, no se hace nada.

$$x = (-1, 6, 14, 0, 1, 2)$$

Como 14 y 0 no están ordenados, se intercambian.

$$x = (-1, 6, 0, 14, 1, 2)$$

Como 14 y 1 no están ordenados, se intercambian.

$$x = (-1, 6, 0, 1, 14, 2)$$

Como 14 y 2 no están ordenados, se intercambian.

$$x = (-1, 6, 0, 1, 2, 14)$$

Al final de esta primera pasada el elemento mayor quedó en la última posición.

Segunda pasada. Como -1 y 6 están ordenados, no se hace nada.

$$x = (-1, 6, 0, 1, 2, 14)$$

Como 6 y 0 no están ordenados, se intercambian.

$$x = (-1, 0, 6, 1, 2, 14)$$

Como 6 y 1 no están ordenados, se intercambian.

$$x = (-1, 0, 1, 6, 2, 14)$$

Como 6 y 2 no están ordenados, se intercambian.

$$x = (-1, 0, 1, 2, 6, 14)$$

Finalizada esta segunda pasada, 6 y 14 están en sus posiciones definitivas.

Tercera pasada. Como -1 y 0 están ordenados, no se hace nada.

$$x = (-1, 0, 1, 2, 6, 14)$$

Como 0 y 1 están ordenados, no se hace nada.

$$x = (-1, 0, 1, 2, 6, 14)$$

Como 1 y 2 están ordenados, no se hace nada.

$$x = (-1, 0, 1, 2, 6, 14)$$

En esta tercera etapa no hubo cambios, luego el vector está ordenado.

```
void ordBurb(double *x, int n)
{
    // ordena, de manera creciente, el vector x
    // por el metodo burbuja

    int nCamb, m;
    double *pxi, *pxi1, *pxm, t;

    for( m = n-1; m >= 0; m--){
        nCamb = 0;
        pxi = x;
        pxi1 = x + 1;
        pxm = x + m;
        while( pxi < pxm ){
            if( *pxi > *pxi1 ){
                t = *pxi;
                *pxi++ = *pxi1;
                *pxi1++ = t;
                nCamb++;
            }
            else{
                pxi++;
                pxi1++;
            }
        }
        if( nCamb == 0 ) return;
    }
}
```

Ejercicios

Los ejercicios de este capítulo no están enfocados directamente a los temas tratados en él; más bien, corresponden a varios temas y aparecen aquí por tratarse del último capítulo sobre C.

- 10.1** Haga un programa que escriba en un archivo todas las combinaciones de enteros entre 1 y n .
- 10.2** Elabore un programa que escriba en un archivo todas las permutaciones de los enteros entre 1 y n .
- 10.3** Dado un entero $n > 0$ y $2n$ enteros $u_1 \leq v_1, u_2 \leq v_2, \dots, u_n \leq v_n$, escriba todas la n -uplas (i_1, i_2, \dots, i_n) tales que $u_k \leq i_k \leq v_k$ para todo k .
- 10.4** Utilice el ejercicio anterior para resolver por la “fuerza bruta” (estudio de todas las posibilidades) el problema del morral con cotas inferiores y superiores. Sean n un entero positivo; $C, p_1, p_2, \dots, p_n, b_1, b_2, \dots, b_n$ números positivos; $u_1 \leq v_1, u_2 \leq v_2, \dots, u_n \leq v_n$ enteros positivos. Encontrar enteros x_1, x_2, \dots, x_n para resolver el siguiente problema de maximización con restricciones:

$$\begin{aligned} \max \quad & b_1x_1 + b_2x_2 + \cdots + b_nx_n \\ \text{s.t. } & p_1x_1 + p_2x_2 + \cdots + p_nx_n \leq C \\ & u_i \leq x_i \leq v_i, \quad i = 1, 2, \dots, n. \end{aligned}$$

11

Solución de sistemas lineales

Uno de los problemas numéricos más frecuentes, o tal vez el más frecuente, consiste en resolver un sistema de ecuaciones de la forma

$$Ax = b \tag{11.1}$$

donde A es una matriz cuadrada, de tamaño $n \times n$, invertible. Esto quiere decir que el sistema tiene una única solución.

Se trata de resolver un sistema de ecuaciones de orden mucho mayor que 2. En la práctica se pueden encontrar sistemas de tamaño 20, 100, 1000 o mucho más grandes. Puesto que se trata de resolver el sistema con la ayuda de un computador, entonces las operaciones realizadas involucran errores de redondeo o truncamiento. La solución obtenida no es absolutamente exacta, pero se desea que la acumulación de los errores sea relativamente pequeña o casi despreciable.

11.1 Notación

Sean A una matriz $m \times n$, con elementos a_{ij} , $i = 1, \dots, m$, $j = 1, \dots, n$ y $x = (x_1, x_2, \dots, x_n)$. Para denotar filas o columnas, o partes de ellas, se

usará la notación de Matlab y Scilab.

- parte de un vector: $x(5 : 7) = (x_5, x_6, x_7)$,
- fila i -ésima: $A_{i\cdot} = A(i, :)$,
- columna j -ésima: $A_{\cdot j} = A(:, j)$,
- parte de la fila i -ésima: $A(i, 1 : 4) = [a_{i1} \ a_{i2} \ a_{i3} \ a_{i4}]$
- parte de la columna j -ésima: $A(2 : 4, j) = [a_{2j} \ a_{3j} \ a_{4j}]^T$
- submatriz: $A(3 : 6, 2 : 5)$.

Supongamos que se tiene una función, llamada **prodEsc**, que calcula el producto escalar entre dos vectores (del mismo tamaño), o entre dos filas de una matriz, o entre dos columnas, o entre una fila y una columna.

Supongamos que se tiene una función, llamada **xMasAlfaY**, que a un vector le suma α veces otro vector. Los valores del primer vector desaparecen los correspondientes al resultado

$$\text{xMasAlfaY}(u, 3.5, v) : u \leftarrow u + 3.5v$$

Esta función también trabaja para filas o columnas de una matriz.

11.2 Métodos ingenuos

Teóricamente, resolver el sistema $Ax = b$ es equivalente a la expresión

$$x = A^{-1}b.$$

Es claro que calcular la inversa de una matriz es mucho más dispendioso que resolver un sistema de ecuaciones; entonces, este camino sólo se utiliza en deducciones teóricas o, en muy raros casos, cuando A^{-1} se calcula muy fácilmente.

Otro método que podría utilizarse para resolver $Ax = b$ es la regla de Cramer. Para un sistema de orden 3 las fórmulas son:

$$x_1 = \frac{\det \begin{bmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{bmatrix}}{\det(A)}, \quad x_2 = \frac{\det \begin{bmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{bmatrix}}{\det(A)},$$

$$x_3 = \frac{\det \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{bmatrix}}{\det(A)}.$$

Supongamos ahora que cada determinante se calcula por medio de cofactores. Este cálculo se puede hacer utilizando cualquier fila o cualquier columna; por ejemplo, si A es 3×3 , utilizando la primera fila,

$$\det(A) = a_{11} \det \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12} \det \begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13} \det \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}.$$

En general, sea M_{ij} la matriz $(n-1) \times (n-1)$ obtenida al suprimir de A la fila i y la columna j . Si se calcula $\det(A)$ utilizando la primera fila,

$$\det(A) = a_{11} \det(M_{11}) - a_{12} \det(M_{12}) + \cdots + (-1)^{(1+n)} a_{1n} \det(M_{1n}).$$

Sea μ_n el número de multiplicaciones necesarias para calcular, por cofactores, el determinante de una matriz de orden n . La fórmula anterior nos indica que

$$\mu_n > n\mu_{n-1}.$$

Como a su vez $\mu_{n-1} > (n-1)\mu_{n-2}$ y $\mu_{n-2} > (n-2)\mu_{n-3}$, ..., entonces

$$\begin{aligned} \mu_n &> n(n-1)(n-2) \cdots \mu_2 = n(n-1)(n-2) \cdots 2, \\ \mu_n &> n!. \end{aligned}$$

Para resolver un sistema de ecuaciones por la regla de Cramer, hay que calcular $n+1$ determinantes, luego el número total de multiplicaciones necesarias para resolver un sistema de ecuaciones por la regla de Cramer, calculando los determinantes por cofactores, es superior a $(n+1)!$.

Tomemos un sistema, relativamente pequeño, $n = 20$,

$$21! = 5.1091E19.$$

Siendo muy optimistas (sin tener en cuenta las sumas y otras operaciones concomitantes), supongamos que un computador del año 2000 hace 1000 millones de multiplicaciones por segundo. Entonces, el tiempo necesario para resolver un sistema de ecuaciones de orden 20 por la regla de Cramer y el método de cofactores es francamente inmanejable:

$$\text{tiempo} > 5.1091E10 \text{ segundos} = 16.2 \text{ siglos.}$$

11.3 Sistema diagonal

El caso más sencillo de (11.1) corresponde a una matriz diagonal. Para matrices triangulares, en particular para las diagonales, el determinante es el producto de los n elementos diagonales. Entonces una matriz triangular es invertible si y solamente si todos los elementos diagonales son diferentes de cero.

La solución de un sistema diagonal se obtiene mediante

$$x_i = \frac{b_i}{a_{ii}}, \quad i = 1, \dots, n. \quad (11.2)$$

Como los elementos diagonales son no nulos, no hay ningún problema para efectuar las divisiones.

11.4 Sistema triangular superior

Resolver un sistema triangular superior (A es triangular superior) es muy sencillo. Antes de ver el algoritmo en el caso general, veamos, por medio de un ejemplo, cómo se resuelve un sistema de orden 4.

Ejemplo 11.1. Resolver el siguiente sistema:

$$\begin{aligned} 4x_1 + 3x_2 - 2x_3 + x_4 &= 4 \\ -0.25x_2 + 2.5x_3 + 4.25x_4 &= -11 \\ 45x_3 + 79x_4 &= -203 \\ 2.8x_4 &= -5.6 \end{aligned}$$

De la cuarta ecuación, se deduce que $x_4 = -5.6/2.8 = -2$. A partir de la tercera ecuación

$$\begin{aligned} 45x_3 &= -203 - (79x_4) \\ x_3 &= \frac{-203 - (79x_4)}{45}. \end{aligned}$$

Reemplazando x_4 por su valor, se obtiene $x_3 = -1$. A partir de la segunda ecuación

$$\begin{aligned} -0.25x_2 &= -11 - (2.5x_3 + 4.25x_4) \\ x_2 &= \frac{-11 - (2.5x_3 + 4.25x_4)}{-0.25}. \end{aligned}$$

Reemplazando x_3 y x_4 por sus valores, se obtiene $x_2 = 0$. Finalmente, utilizando la primera ecuación,

$$\begin{aligned} 4x_1 &= 4 - (3x_2 - 2x_3 + x_4) \\ x_3 &= \frac{4 - (3x_2 - 2x_3 + x_4)}{4}. \end{aligned}$$

Reemplazando x_2 , x_3 y x_4 por sus valores, se obtiene $x_1 = 1$. \diamond

En general, para resolver un sistema triangular, primero se calcula $x_n = b_n/a_{nn}$. Con este valor se puede calcular x_{n-1} , y así sucesivamente. Conocidos los valores x_{i+1} , x_{i+2} , ..., x_n , la ecuación i -ésima es

$$\begin{aligned} a_{ii}x_i + a_{i,i+1}x_{i+1} + a_{i,i+2}x_{i+2} + \dots + a_{in}x_n &= b_i, \\ a_{ii}x_i + \text{prodEsc}(A(i, i+1 : n), x(i+1 : n)) &= b_i, \\ x_i &= \frac{b_i - \text{prodEsc}(A(i, i+1 : n), x(i+1 : n))}{a_{ii}} \end{aligned}$$

Como se supone que A es regular (invertible o no singular), los elementos diagonales son no nulos y no se presentan problemas al efectuar la división.

El esquema del algoritmo es el siguiente:

```

 $x_n = b_n/a_{nn}$ 
para  $i = n-1, \dots, 1$ 
     $x_i = (b_i - \text{prodEsc}(A(i, i+1 : n), x(i+1 : n))) / a_{ii}$ 
fin-para

```

Si los subíndices de A y de x van desde 0 hasta $n-1$, entonces el esquema del algoritmo queda así:

```

 $x_{n-1} = b_{n-1}/a_{n-1,n-1}$ 
para  $i = n-2, \dots, 0$ 
     $x_i = (b_i - \text{prodEsc}(A(i, i+1 : n-1), x(i+1 : n-1))) / a_{ii}$ 
fin-para

```

11.4.1 Número de operaciones

Una de las maneras de medir la rapidez o lentitud de un método es mediante el conteo del número de operaciones. Usualmente se tienen

en cuenta las sumas, restas, multiplicaciones y divisiones entre números de punto flotante, aunque hay más operaciones fuera de las anteriores, por ejemplo las comparaciones y las operaciones entre enteros. Las cuatro operaciones se conocen con el nombre genérico de operaciones de punto flotante *flops* (floating point operations). Algunas veces se hacen dos grupos: por un lado sumas y restas, y por otro multiplicaciones y divisiones. Si se supone que el tiempo necesario para efectuar una multiplicación es bastante mayor que el tiempo de una suma, entonces se acostumbra a dar el número de multiplicaciones (o divisiones). El diseño de los procesadores actuales muestra tendencia al hecho de que los dos tiempos sean comparables. Entonces se acostumbra a evaluar el número de *flops*.

	Sumas y restas	Multiplicaciones y divisiones
cálculo de x_n	0	1
cálculo de x_{n-1}	1	2
cálculo de x_{n-2}	2	3
...		
cálculo de x_2	$n - 2$	$n - 1$
cálculo de x_1	$n - 1$	n
Total	$n^2/2 - n/2$	$n^2/2 + n/2$

Número total de operaciones de punto flotante: n^2 .

11.4.2 Implementación en C

Para la siguiente implementación en C es necesario tener en cuenta estos aspectos:

- La matriz está almacenada en un arreglo unidimensional. Por tanto, la información referente a la matriz A se pasa a la función por medio de la dirección del primer elemento de la matriz.
- Se usa la función `prodXY`, definida en el capítulo 10, que calcula el producto escalar de dos vectores. Su prototipo es

```
double prodXY(double *x, double *y, int n);
```

- Para utilizarla se requiere pasar las direcciones de cada uno de los primeros elementos de los vectores en consideración, o sea, las direcciones de $a_{i,i+1}$ y de x_{i+1} . El elemento $a_{i,i+1}$ es exactamente $a[i*n + i+1]$, o sea, con $a[i*(n+1) + 1]$.
- El número de elementos de $x(i + 1 : n - 1)$ es $n - 1 - i$
- Como `prodXY` devuelve 0 cuando n es menor o igual que cero, entonces el cálculo de x_{n-1} se puede incluir dentro del ciclo `for`.
- Para ahorrar memoria, la solución del sistema, es decir el vector x , estará a la salida de la función, en el arreglo `b`, que contenía inicialmente los términos independientes.

```

int solTrSup(double *a, double *b, int n, double eps)
{
    // Solucion del sistema triangular superior
    //      A x = b.

    // A almacenada en un arreglo unidimensional,
    // Aij esta en a[i*n+j].

    // Devuelve 1 si se resolvio el sistema.
    //          0 si la matriz es singular o casi.
    //          -1 si n es inadecuado
    //
    // Cuando hay solucion, esta quedara en b.

    // A se considera singular si
    // | Aii | <= eps para algun i.

    int i, ii, n1, n_1;

    if( n <= 0 ) return -1;

    n1 = n+1;
    n_1 = n-1;
    for( i = n_1; i >= 0; i--){
        ii = i*n1;
        if( fabs( a[ii] ) <= eps ) return 0;

```

```

    b[i] = (b[i]-prodXY(&a[ii+1], &b[i+1], n_1-i))/a[ii];
}
return 1;
}

```

11.5 Sistema triangular inferior

La solución de un sistema triangular inferior $Ax = b$, A triangular inferior, es análoga al caso de un sistema triangular superior. Primero se calcula x_1 , después x_2 , enseguida x_3 y así sucesivamente hasta x_n .

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}}. \quad (11.3)$$

El esquema del algoritmo es el siguiente:

```

para i = 1, ..., n
    xi = (bi - prodEsc(A(i, 1 : i - 1), x(1 : i - 1)))/aii
fin-para

```

El número de operaciones es exactamente el mismo del caso triangular superior.

11.6 Método de Gauss

El método de Gauss para resolver el sistema $Ax = b$ tiene dos partes; la primera es la triangularización del sistema, es decir, por medio de operaciones elementales, se construye un sistema

$$A'x = b', \quad (11.4)$$

equivalente al primero, tal que A' sea triangular superior. Que los sistemas sean equivalentes quiere decir que la solución de $Ax = b$ es exactamente la misma solución de $A'x = b'$. La segunda parte es simplemente la solución del sistema triangular superior.

Para una matriz, con índices entre 1 y n , el esquema de triangularización se puede escribir así:

para $k = 1, \dots, n - 1$

buscar ceros en la columna k , por debajo de la diagonal.

fin-para k

Afinando un poco más:

para $k = 1, \dots, n - 1$

para $i = k + 1, \dots, n$

buscar ceros en la posición de a_{ik} .

fin-para i

fin-para k

Ejemplo 11.2. Consideremos el siguiente sistema de ecuaciones:

$$\begin{array}{rcl} 4x_1 + 3x_2 - 2x_3 + x_4 & = & 4 \\ 3x_1 + 2x_2 + x_3 + 5x_4 & = & -8 \\ -2x_1 + 3x_2 + x_3 + 2x_4 & = & -7 \\ -5x_1 & + x_3 + x_4 & = -8 \end{array}$$

En forma matricial se puede escribir:

$$\left[\begin{array}{cccc} 4 & 3 & -2 & 1 \\ 3 & 2 & 1 & 5 \\ -2 & 3 & 1 & 2 \\ -5 & 0 & 1 & 1 \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \right] = \left[\begin{array}{c} 4 \\ -8 \\ -7 \\ -8 \end{array} \right]$$

Es usual trabajar únicamente con los números, olvidando temporalmente los x_i . Más aún, se acostumbra trabajar con una matriz ampliada, resultado de pegar a la derecha de A el vector b .

$$\left[\begin{array}{cccc|c} 4 & 3 & -2 & 1 & 4 \\ 3 & 2 & 1 & 5 & -8 \\ -2 & 3 & 1 & 2 & -7 \\ -5 & 0 & 1 & 1 & -8 \end{array} \right]$$

Inicialmente hay que buscar ceros en la primera columna. Para buscar cero en la posición $(2,1)$, fila 2 y columna 1, se hace la siguiente operación:

$$\text{fila2}_{\text{nueva}} \leftarrow \text{fila2}_{\text{vieja}} - (3/4) * \text{fila1}$$

Para hacer más sencilla la escritura la expresión anterior se escribirá simplemente:

$$\text{fila2} \leftarrow \text{fila2} - (3/4)*\text{fila1}$$

$$\begin{bmatrix} 4 & 3 & -2 & 1 & 4 \\ 0 & -0.25 & 2.5 & 4.25 & -11 \\ -2 & 3 & 1 & 2 & -7 \\ -5 & 0 & 1 & 1 & -8 \end{bmatrix}$$

Para buscar cero en la posición (3, 1) se hace la siguiente operación:

$$\text{fila3} \leftarrow \text{fila3} - (-2/4)*\text{fila1}$$

$$\begin{bmatrix} 4 & 3 & -2 & 1 & 4 \\ 0 & -0.25 & 2.5 & 4.25 & -11 \\ 0 & 4.5 & 0 & 2.5 & -5 \\ -5 & 0 & 1 & 1 & -8 \end{bmatrix}$$

Para buscar cero en la posición (4, 1) se hace la siguiente operación:

$$\text{fila4} \leftarrow \text{fila4} - (-5/4)*\text{fila1}$$

$$\begin{bmatrix} 4 & 3 & -2 & 1 & 4 \\ 0 & -0.25 & 2.5 & 4.25 & -11 \\ 0 & 4.5 & 0 & 2.5 & -5 \\ 0 & 3.75 & -1.5 & 2.25 & -3 \end{bmatrix}$$

Ahora hay que buscar ceros en la segunda columna. Para buscar cero en la posición (3, 2) se hace la siguiente operación:

$$\text{fila3} \leftarrow \text{fila3} - (4.5/(-0.25))*\text{fila2}$$

$$\begin{bmatrix} 4 & 3 & -2 & 1 & 4 \\ 0 & -0.25 & 2.5 & 4.25 & -11 \\ 0 & 0 & 45 & 79 & -203 \\ 0 & 3.75 & -1.5 & 2.25 & -3 \end{bmatrix}$$

Para buscar cero en la posición (4, 2) se hace siguiente operación:

$$\text{fila4} \leftarrow \text{fila4} - (3.75/(-0.25))*\text{fila2}$$

$$\begin{bmatrix} 4 & 3 & -2 & 1 & 4 \\ 0 & -0.25 & 2.5 & 4.25 & -11 \\ 0 & 0 & 45 & 79 & -203 \\ 0 & 0 & 36 & 66 & -168 \end{bmatrix}$$

Para buscar cero en la posición (4, 3) se hace la siguiente operación:

$$\text{fila4} \leftarrow \text{fila4} - (36/45)*\text{fila3}$$

$$\begin{bmatrix} 4 & 3 & -2 & 1 & 4 \\ 0 & -0.25 & 2.5 & 4.25 & -11 \\ 0 & 0 & 45 & 79 & -203 \\ 0 & 0 & 0 & 2.8 & -5.6 \end{bmatrix}$$

El sistema resultante ya es triangular superior. Entonces se calcula primero $x_4 = -5.6/2.8 = -2$. Con este valor, utilizando la tercera ecuación resultante, se calcula x_3 , después x_2 y x_1 .

$$x = (1, 0, -1, -2). \diamond$$

De manera general, cuando ya hay ceros por debajo de la diagonal, en las columnas 1, 2, ..., $k - 1$, para obtener cero en la posición (i, k) se hace la operación

$$\text{filai} \leftarrow \text{filai} - (a_{ik}/a_{kk})*\text{filak}$$

Lo anterior se puede reescribir así:

$$\begin{aligned} \text{lik} &= a_{ik}/a_{kk} \\ A(i, :) &= A(i, :) - \text{lik} * A(k, :) \\ b_i &= b_i - \text{lik} * b_k \end{aligned}$$

Como en las columnas 1, 2, ..., $k - 1$ hay ceros, tanto en la fila k como en la fila i , entonces $a_{i1}, a_{i2}, \dots, a_{i,k-1}$ seguirán siendo cero. Además, las operaciones se hacen de tal manera que a_{ik} se vuelva cero. Entonces a_{ik} no se calcula puesto que dará 0. Luego los cálculos se hacen en la fila i a partir de la columna $k + 1$.

$$\begin{aligned} \text{lik} &= a_{ik}/a_{kk} \\ a_{ik} &= 0 \\ A(i, k + 1 : n) &= A(i, k + 1 : n) - \text{lik} * A(k, k + 1 : n) \\ b_i &= b_i - \text{lik} * b_k \end{aligned}$$

En resumen, el esquema de la triangularización es:

```

para  $k = 1, \dots, n - 1$ 
  para  $i = k + 1, \dots, n$ 
    lik =  $a_{ik}/a_{kk}$ ,  $a_{ik} = 0$ 
     $A(i, k + 1 : n) = A(i, k + 1 : n) - \text{lik} * A(k, k + 1 : n)$ 
     $b_i = b_i - \text{lik} * b_k$ 
  fin-para  $i$ 
fin-para  $k$ 

```

Este esquema funciona, siempre y cuando no aparezca un **pivote**, a_{kk} , nulo o casi nulo. Cuando aparezca es necesario buscar un elemento no nulo en el resto de la columna. Si, en el proceso de triangulación, toda la columna $A(k : n, k)$ es nula o casi nula, entonces A es singular.

```

para  $k = 1, \dots, n - 1$ 
  para  $i = k + 1, \dots, n$ 
    si  $|a_{kk}| \leq \varepsilon$  ent
      buscar  $m$ ,  $k + 1 \leq m \leq n$ , tal que  $|a_{mk}| > \varepsilon$ 
      si no fue posible ent salir
      intercambiar( $A(k, k : n)$ ,  $A(m, k : n)$ )
      intercambiar( $b_k, b_m$ )
    fin-si
    lik =  $a_{ik}/a_{kk}$ ,  $a_{ik} = 0$ 
     $A(i, k + 1 : n) = A(i, k + 1 : n) - \text{lik} * A(k, k + 1 : n)$ 
     $b_i = b_i - \text{lik} * b_k$ 
  fin-para  $i$ 
fin-para  $k$ 
si  $|a_{nn}| \leq \varepsilon$  ent salir

```

Cuando en un proceso una variable toma valores enteros desde un límite inferior hasta un límite superior, y el límite inferior es mayor que el límite superior, el proceso no se efectúa.

Así, en el algoritmo anterior se puede hacer variar k , en el bucle externo, entre 1 y n , y entonces no es necesario controlar si $a_{nn} \approx 0$ ya que, cuando $k = n$, no es posible buscar m entre $n + 1$ y n .

```

para  $k = 1, \dots, n$ 
  para  $i = k + 1, \dots, n$ 
    si  $|a_{kk}| \leq \varepsilon$  ent
      buscar  $m$ ,  $k + 1 \leq m \leq n$ , tal que  $|a_{mk}| > \varepsilon$ 
      si no fue posible ent salir
      intercambiar( $A(k, k : n)$ ,  $A(m, k : n)$ )
      intercambiar( $b_k, b_m$ )
    fin-si
    lik =  $a_{ik}/a_{kk}$ ,  $a_{ik} = 0$ 
     $A(i, k + 1 : n) = A(i, k + 1 : n) - \text{lik} * A(k, k + 1 : n)$ 
     $b_i = b_i - \text{lik} * b_k$ 
  fin-para  $i$ 
fin-para  $k$ 

```

El anterior algoritmo se adapta fácilmente al caso de los subíndices, variando entre 0 y $n - 1$.

```

para  $k = 0, \dots, n - 1$ 
  para  $i = k + 1, \dots, n - 1$ 
    si  $|a_{kk}| \leq \varepsilon$  ent
      buscar  $m$ ,  $k + 1 \leq m \leq n - 1$ , tal que  $|a_{mk}| > \varepsilon$ 
      si no fue posible ent salir
      intercambiar( $A(k, k : n - 1)$ ,  $A(m, k : n - 1)$ )
      intercambiar( $b_k, b_m$ )
    fin-si
    lik =  $a_{ik}/a_{kk}$ ,  $a_{ik} = 0$ 
     $A(i, k + 1 : n - 1) = A(i, k + 1 : n - 1) - \text{lik} * A(k, k + 1 : n - 1)$ 
     $b_i = b_i - \text{lik} * b_k$ 
  fin-para  $i$ 
fin-para  $k$ 

```

11.6.1 Número de operaciones

En el método de Gauss hay que tener en cuenta el número de operaciones de cada uno de los dos procesos: triangularización y solución del sistema triangular.

Triangularización

Consideremos inicialmente la búsqueda de cero en la posición (2, 1). Para efectuar $A(2, 2 : n) = A(2, 2 : n) - \text{lik} * A(1, 2 : n)$ es necesario hacer

$n - 1$ sumas y restas. Para $b_2 = b_2 - \text{lik} * b_1$ es necesario una resta. En resumen n sumas (o restas). Multiplicaciones y divisiones: una división para calcular lik ; $n - 1$ multiplicaciones para $\text{lik} * A(1, 2 : n)$ y una para $\text{lik} * b_1$. En resumen, $n + 1$ multiplicaciones (o divisiones).

Para obtener un cero en la posición $(3, 1)$ se necesita exactamente el mismo número de operaciones. Entonces para la obtener ceros en la primera columna:

	Sumas y restas	Multiplicaciones y divisiones
cero en la posición de a_{21}	n	$n + 1$
cero en la posición de a_{31}	n	$n + 1$
...		
cero en la posición de a_{n1}	n	$n + 1$
Total para la columna 1	$(n - 1)n$	$(n - 1)(n + 1)$

Un conteo semejante permite ver que se requieren $n - 1$ sumas y n multiplicaciones para obtener un cero en la posición de a_{32} . Para buscar ceros en la columna 2 se van a necesitar $(n - 2)(n - 1)$ sumas y $(n - 2)n$ multiplicaciones.

	Sumas y restas	Multiplicaciones y divisiones
ceros en la columna 1	$(n - 1)n$	$(n - 1)(n + 1)$
ceros en la columna 2	$(n - 2)(n - 1)$	$(n - 2)n$
ceros en la columna 3	$(n - 3)(n - 2)$	$(n - 3)(n - 1)$
...		
ceros en la columna $n - 2$	$2(3)$	$2(4)$
ceros en la columna $n - 1$	$1(2)$	$1(3)$

Es necesario utilizar el resultado

$$\sum_{i=1}^m i^2 = \frac{m(m+1)(2m+1)}{6}.$$

Número de sumas y restas:

$$\sum_{i=1}^{n-1} i(i+1) = \sum_{i=1}^{n-1} (i^2 + i) = \frac{n^3}{3} - \frac{n}{3} \approx \frac{n^3}{3}.$$

Número de multiplicaciones y divisiones:

$$\sum_{i=1}^{n-1} i(i+2) = \sum_{i=1}^{n-1} (i^2 + 2i) = \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6} \approx \frac{n^3}{3}.$$

Número de operaciones:

$$\frac{n^3}{3} - \frac{n}{3} + \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6} = \frac{2n^3}{3} + \frac{n^2}{2} - \frac{7n}{6} \approx \frac{2n^3}{3}.$$

Proceso completo

El número de operaciones para las dos partes, triangularización y solución del sistema triangular, es

$$\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6} \approx \frac{2n^3}{3}.$$

Para valores grandes de n el número de operaciones de la solución del sistema triangular es despreciable con respecto al número de operaciones de la triangularización.

11.6.2 Implementación en C

La implementación hace uso de la función `xMasAlfaY`, definida en el capítulo 10, que adiciona a un vector un múltiplo de otro vector:

$$u \leftarrow u + 3.5v$$

Su prototipo es:

```
void xMasAlfaY(double *x, double alfa, double *y, int n);
```

La siguiente implementación en C es casi la traducción a C del algoritmo presentado. Hay algunas variables adicionales para no tener que efectuar varias veces la misma operación, por ejemplo, $n + 1$.

```
int gauss1( double *a, double *b, double *x, int n,
            double eps)
{
    // Metodo de Gauss SIN PIVOTEO parcial
    // para resolver A x = b
```

```

// Intercambio de filas solo si el pivote es casi nulo.

// A almacenada en un arreglo unidimensional
// Aij esta en a[i*n+j]

// Devuelve 1 si se resolvio el sistema.
// 0 si la matriz es singular o casi.
// -1 si n es inadecuado

int k, i, n1, ik, n_1, kk, in, j, ii, m;
double lik, s;

if( n <= 0 ) return -1;

n1 = n+1;
n_1 = n-1;

// triangularizacion

for( k = 0; k < n; k++){
    kk = k*n1;
    if( fabs( a[kk] ) <= eps ){
        for(m = k+1; m<n; m++)if(fabs(a[m*n+k]) >= eps)
            break;
        if( m >= n ) return 0;
        intercXY(&a[kk], &a[m*n+k], n-k);
        interc( b[k], b[m] );
    }
    for( i = k+1; i < n; i++){
        // anular Aik
        ik = i*n + k;
        lik = a[ik]/a[kk];
        xMasAlfaY( &a[ik+1], -lik, &a[kk+1], n_1-k );
        b[i] -= lik*b[k];
        a[ik] = 0.0;
    }
}
solTrSup(a, b, x, n, eps);

```

```

    return 1;
}

```

11.7 Factorización LU

Si durante el proceso del método de Gauss no fue necesario intercambiar filas, entonces se puede demostrar que se obtiene fácilmente la factorización $A = LU$, donde L es una matriz triangular inferior con unos en la diagonal y U es una matriz triangular superior. La matriz U es simplemente la matriz triangular superior obtenida al final del proceso.

Para el ejemplo anterior:

$$U = \begin{bmatrix} 4 & 3 & -2 & 1 \\ 0 & -0.25 & 2.5 & 4.25 \\ 0 & 0 & 45 & 79 \\ 0 & 0 & 0 & 2.8 \end{bmatrix}$$

La matriz L , con unos en la diagonal, va a estar formada simplemente por los coeficientes $l_{ik} = a_{ik}/a_{kk}$.

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix}$$

Siguiendo con el ejemplo:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -0.75 & 1 & 0 & 0 \\ -0.5 & -18 & 1 & 0 \\ -1.25 & -15 & 0.8 & 1 \end{bmatrix}$$

En este ejemplo, fácilmente se comprueba que $LU = A$. Esta factorización es útil para resolver otro sistema $Ax = \tilde{b}$, exactamente con la misma matriz de coeficientes, pero con diferentes términos independientes.

$$\begin{aligned} Ax &= \tilde{b}, \\ LUx &= \tilde{b}, \\ Ly &= \tilde{b}, \\ \text{donde } Ux &= y. \end{aligned}$$

En resumen:

- Resolver $Ly = \tilde{b}$ para obtener y .
- Resolver $Ux = y$ para obtener x .

Ejemplo 11.3. Resolver

$$\begin{aligned} 4x_1 + 3x_2 - 2x_3 + x_4 &= 8 \\ 3x_1 + 2x_2 + x_3 + 5x_4 &= 30 \\ -2x_1 + 3x_2 + x_3 + 2x_4 &= 15 \\ -5x_1 + x_3 + x_4 &= 2 \end{aligned}$$

Al resolver

$$\left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0.75 & 1 & 0 & 0 \\ -0.5 & -18 & 1 & 0 \\ -1.25 & -15 & 0.8 & 1 \end{array} \right] \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ y_4 \end{array} \right] = \left[\begin{array}{c} 8 \\ 30 \\ 15 \\ 2 \end{array} \right]$$

se obtiene $y = [8 \ 24 \ 451 \ 11.2]^T$. Al resolver

$$\left[\begin{array}{cccc} 4 & 3 & -2 & 1 \\ 0 & -0.25 & 2.5 & 4.25 \\ 0 & 0 & 45 & 79 \\ 0 & 0 & 0 & 2.8 \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \right] = \left[\begin{array}{c} 8.0 \\ 24.0 \\ 451.0 \\ 11.2 \end{array} \right]$$

se obtiene la solución final $x = [1 \ 2 \ 3 \ 4]^T$. ◇

Resolver un sistema triangular, con unos en la diagonal, requiere $n^2 - n \approx n^2$ operaciones. Entonces, para resolver un sistema adicional, con la misma matriz A , se requiere efectuar aproximadamente $2n^2$ operaciones, en lugar de $2n^3/3$ que se requerirían si se volviera a empezar el proceso.

La factorización $A = LU$ es un subproducto gratuito del método de Gauss; gratuito en tiempo y en requerimientos de memoria. No se requiere tiempo adicional puesto que el cálculo de los l_{ik} se hace dentro del método de Gauss. Tampoco se requiere memoria adicional puesto que los valores l_{ik} se pueden ir almacenando en A en el sitio de a_{ik} que justamente vale cero.

En el algoritmo hay únicamente un pequeño cambio:

$$\begin{array}{l} \vdots \\ \text{lik} = a_{ik}/a_{kk} \\ a_{ik} = \text{lik} \\ A(i, k+1 : n-1) = A(i, k+1 : n-1) - \text{lik} * A(k, k+1 : n-1) \\ b_i = b_i - \text{lik} * b_k \\ \vdots \end{array}$$

En la matriz final A estará la información indispensable de L y de U .

$$L = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ l_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ l_{31} & l_{32} & u_{31} & \cdots & u_{3n} \\ \vdots & & \ddots & & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & u_{nn} \end{bmatrix}$$

En el ejemplo anterior, la matriz final con información de L y de U es:

$$\begin{bmatrix} 4 & 3 & -2 & 1 \\ 0.75 & -0.25 & 2.5 & 4.25 \\ -0.5 & -18 & 45 & 79 \\ -1.25 & -15 & 0.8 & 2.8 \end{bmatrix}$$

11.8 Método de Gauss con pivoteo parcial

En el método de Gauss clásico, únicamente se intercambian filas cuando el pivote, a_{kk} , es nulo o casi nulo. Como el pivote (el elemento a_{kk} en la iteración k) va a ser divisor para el cálculo de lik , y como el error de redondeo o de truncamiento se hace mayor cuando el divisor es cercano a cero, entonces es muy conveniente buscar que el pivote sea grande en valor absoluto. Es decir, hay que evitar los pivotes que sin ser nulos son cercanos a cero.

En el método de Gauss con pivoteo parcial se busca el elemento dominante, o sea, el de mayor valor absoluto en la columna k de la diagonal hacia abajo, es decir, entre los valores $|a_{kk}|, |a_{k+1,k}|, |a_{k+2,k}|, \dots, |a_{kn}|$, y se intercambian la fila k y la fila del valor dominante. Esto mejora notablemente, en muchos casos, la precisión de la solución final obtenida.

Se dice que el pivoteo es total si en la iteración k se busca el mayor valor de $\{|a_{ij}| : k \leq i, j \leq n\}$. En este caso es necesario intercambiar dos filas y dos columnas. Así se consigue mejorar un poco la precisión con respecto al método de pivoteo parcial, pero a un costo nada despreciable. En el método de pivoteo parcial se busca el mayor valor entre $n - k + 1$ valores. En el pivoteo total se busca entre $(n - k + 1)^2$ valores. Si se busca, de manera secuencial, el máximo entre p elementos, entonces hay que hacer, además de operaciones de asignación, por lo menos $p - 1$ comparaciones. Estas operaciones no son de punto flotante y son más rápidas que ellas, pero para n grande, el tiempo utilizado no es despreciable. En el método de pivoteo parcial hay aproximadamente $n^2/2$ comparaciones, en el pivoteo total aproximadamente $n^3/6$. En resumen, con el pivoteo total se gana un poco de precisión, pero se gasta bastante más tiempo. El balance aconseja preferir el pivoteo parcial.

Ejemplo 11.4. Resolver por el método de Gauss con pivoteo parcial el siguiente sistema de ecuaciones.

$$\begin{aligned} 4x_1 + 3x_2 - 2x_3 + x_4 &= 4 \\ 3x_1 + 2x_2 + x_3 + 5x_4 &= -8 \\ -2x_1 + 3x_2 + x_3 + 2x_4 &= -7 \\ -5x_1 + x_3 + x_4 &= -8 \end{aligned}$$

La matriz aumentada es:

$$\left[\begin{array}{ccccc} 4 & 3 & -2 & 1 & 4 \\ 3 & 2 & 1 & 5 & -8 \\ -2 & 3 & 1 & 2 & -7 \\ -5 & 0 & 1 & 1 & -8 \end{array} \right]$$

El valor dominante de $A(1 : 4, 1)$ es -5 y está en la fila 4. Entonces se intercambian las filas 1 y 4.

$$\left[\begin{array}{ccccc} -5 & 0 & 1 & 1 & -8 \\ 3 & 2 & 1 & 5 & -8 \\ -2 & 3 & 1 & 2 & -7 \\ 4 & 3 & -2 & 1 & 4 \end{array} \right]$$

Buscar ceros en las posiciones de a_{21} , a_{31} , a_{41} se hace de la manera habitual usando los valores de $\text{lik} = 3/(-5) = -0.6$, 0.4 y -0.8 . Se

obtiene

$$\left[\begin{array}{ccccc} -5 & 0 & 1 & 1 & -8 \\ 0 & 2 & 1.6 & 5.6 & 12.8 \\ 0 & 3 & 0.6 & 1.6 & -3.8 \\ 0 & 3 & -1.2 & 1.8 & -2.4 \end{array} \right]$$

El valor dominante de $A(2 : 4, 2)$ es 3 y está en la fila 3 (o en la fila 4). Entonces se intercambian las filas 2 y 3.

$$\left[\begin{array}{ccccc} -5 & 0 & 1 & 1 & -8 \\ 0 & 3 & 0.6 & 1.6 & -3.8 \\ 0 & 2 & 1.6 & 5.6 & 12.8 \\ 0 & 3 & -1.2 & 1.8 & -2.4 \end{array} \right]$$

Buscar ceros en las posiciones de a_{32} , a_{42} se hace usando los valores de $\text{lik} = 2/3 = 0.6666$ y 1. Se obtiene

$$\left[\begin{array}{ccccc} -5 & 0 & 1 & 1 & -8 \\ 0 & 3 & 0.6 & 1.6 & -3.8 \\ 0 & 0 & 1.2 & 4.5333 & -10.2667 \\ 0 & 0 & -1.8 & 0.2 & 1.4 \end{array} \right]$$

Hay que intercambiar las filas 3 y 4.

$$\left[\begin{array}{ccccc} -5 & 0 & 1 & 1 & -8 \\ 0 & 3 & 0.6 & 1.6 & -3.8 \\ 0 & 0 & -1.8 & 0.2 & 1.4 \\ 0 & 0 & 1.2 & 4.5333 & -10.2667 \end{array} \right]$$

El valor de lik es $1.2/(-1.8) = -0.6667$. Se obtiene

$$\left[\begin{array}{ccccc} -5 & 0 & 1 & 1 & -8 \\ 0 & 3 & 0.6 & 1.6 & -3.8 \\ 0 & 0 & -1.8 & 0.2 & 1.4 \\ 0 & 0 & 0 & 4.6667 & -9.3333 \end{array} \right]$$

Al resolver el sistema triangular superior, se encuentra la solución:

$$x = (1, 0, -1, -2). \diamond$$

El ejemplo anterior sirve simplemente para mostrar el desarrollo del método de Gauss con pivoteo parcial, pero no muestra sus ventajas. El

ejemplo siguiente, tomado de [Atk78], se resuelve inicialmente por el método de Gauss sin pivoteo y después con pivoteo parcial. Los cálculos se hacen con cuatro cifras decimales.

$$\begin{aligned} 0.729x_1 + 0.81x_2 + 0.9x_3 &= 0.6867 \\ x_1 + x_2 + x_3 &= .8338 \\ 1.331x_1 + 1.21x_2 + 1.1x_3 &= 1 \end{aligned}$$

Con la solución exacta, tomada con cuatro cifras decimales, es

$$x = (0.2245, 0.2814, 0.3279).$$

Al resolver el sistema por el método de Gauss, con cuatro cifras decimales y sin pivoteo, resultan los siguientes pasos:

$$\left[\begin{array}{cccc} 0.7290 & 0.8100 & 0.9000 & 0.6867 \\ 1.0000 & 1.0000 & 1.0000 & 0.8338 \\ 1.3310 & 1.2100 & 1.1000 & 1.0000 \end{array} \right]$$

Con $\text{lik} = 1.3717$ y con $\text{lik} = 1.8258$ se obtiene

$$\left[\begin{array}{cccc} 0.7290 & 0.8100 & 0.9000 & 0.6867 \\ 0.0000 & -0.1111 & -0.2345 & -0.1081 \\ 0.0000 & -0.2689 & -0.5432 & -0.2538 \end{array} \right]$$

Con $\text{lik} = 2.4203$ se obtiene

$$\left[\begin{array}{cccc} 0.7290 & 0.8100 & 0.9000 & 0.6867 \\ 0.0000 & -0.1111 & -0.2345 & -0.1081 \\ 0.0000 & 0.0000 & 0.0244 & 0.0078 \end{array} \right]$$

La solución del sistema triangular da:

$$x = (0.2163, 0.2979, 0.3197).$$

Sea x^* la solución exacta del sistema $Ax = b$. Para comparar x^1 y x^2 , dos aproximaciones de la solución, se miran sus distancias a x^* :

$$\|x^1 - x^*\|, \quad \|x^2 - x^*\|.$$

Si $\|x^1 - x^*\| < \|x^2 - x^*\|$, entonces x^1 es, entre x^1 y x^2 , la mejor aproximación de x^* . Cuando no se conoce x^* , entonces se utiliza la norma del

vector residuo o resto, $r = Ax - b$. Si x es la solución exacta, entonces la norma de su resto vale cero. Entonces hay que comparar

$$\|Ax^1 - b\|, \quad \|Ax^2 - b\|.$$

Para la solución obtenida por el método de Gauss, sin pivoteo,

$$\|Ax - b\| = 1.0357e-004, \quad \|x - x^*\| = 0.0202.$$

En seguida está el método de Gauss con pivoteo parcial, haciendo cálculos con 4 cifras decimales.

$$\left[\begin{array}{cccc} 0.7290 & 0.8100 & 0.9000 & 0.6867 \\ 1.0000 & 1.0000 & 1.0000 & 0.8338 \\ 1.3310 & 1.2100 & 1.1000 & 1.0000 \end{array} \right]$$

Intercambio de las filas 1 y 3.

$$\left[\begin{array}{cccc} 1.3310 & 1.2100 & 1.1000 & 1.0000 \\ 1.0000 & 1.0000 & 1.0000 & 0.8338 \\ 0.7290 & 0.8100 & 0.9000 & 0.6867 \end{array} \right]$$

Con $\text{lik} = 0.7513$ y con $\text{lik} = 0.5477$ se obtiene

$$\left[\begin{array}{cccc} 1.3310 & 1.2100 & 1.1000 & 1.0000 \\ 0.0000 & 0.0909 & 0.1736 & 0.0825 \\ 0.0000 & 0.1473 & 0.2975 & 0.1390 \end{array} \right]$$

Intercambio de las filas 2 y 3.

$$\left[\begin{array}{cccc} 1.3310 & 1.2100 & 1.1000 & 1.0000 \\ 0.0000 & 0.1473 & 0.2975 & 0.1390 \\ 0.0000 & 0.0909 & 0.1736 & 0.0825 \end{array} \right]$$

Con $\text{lik} = 0.6171$ se obtiene

$$\left[\begin{array}{cccc} 1.3310 & 1.2100 & 1.1000 & 1.0000 \\ 0.0000 & 0.1473 & 0.2975 & 0.1390 \\ 0.0000 & 0.0000 & -0.0100 & -0.0033 \end{array} \right]$$

La solución del sistema triangular da:

$$x = (0.2267, 0.2770, 0.3300).$$

El cálculo del residuo y la comparación con la solución exacta da:

$$\|Ax - b\| = 1.5112e-004, \quad \|x - x^*\| = 0.0053.$$

Se observa que para este ejemplo la norma del residuo es del mismo orden de magnitud que la norma del residuo correspondiente a la solución obtenida sin pivoteo, aunque algo mayor. La comparación directa con la solución exacta favorece notablemente al método de pivoteo parcial: 0.0053 y 0.0202, relación de 1 a 4 aproximadamente. Además, “visualmente” se observa la mejor calidad de la solución obtenida con pivoteo.

A continuación aparece una versión de la función que implementa el método de Gauss con pivoteo parcial.

```
int gausspp( double *a, double *b, int n, double eps)
{
    // Metodo de Gauss CON PIVOTEO parcial
    // para resolver A x = b.

    // Intercambio real de las filas.

    // A almacenada en un arreglo unidimensional,
    // Aij esta en a[i*n+j].

    // Devuelve 1 si se resolvio el sistema.
    //          0 si la matriz es singular o casi.
    //         -1 si n es inadecuado

    // Cuando hay solucion, esta quedara en b.

    int k, i, n1, ik, n_1, kk, in, j, ii, m, m0;
    double lik, s, aamk, t;

    if( n <= 0 ) return -1;

    n1 = n+1;
    n_1 = n-1;

    // triangularizacion
```

```

for( k = 0; k < n_1; k++){
    kk = k*n1;
    aamk = maxAbsXPos( &a[kk], n, n-k, m0);
    if( aamk <= eps ) return 0;
    if( m0 != 0 ){
        m = k+m0;
        intercXY( &a[kk], &a[m*n+k], n-k);
        t = b[k]; b[k] = b[m]; b[m] = t;
    }

    for( i = k+1; i < n; i++){
        // anular Aik
        ik = i*n + k;
        lik = a[ik]/a[kk];
        xMasAlfaY( &a[ik+1], -lik, &a[kk+1], n_1-k);
        b[i] -= lik*b[k];
        a[ik] = 0.0;
    }
}
if( fabs(a[n*n-1]) <= eps) return 0;

solTrSup(a, b, n, eps);
return 1;
}

```

11.9 Factorización $LU=PA$

Si se aplica el método de Gauss con pivoteo parcial muy probablemente se hace por lo menos un intercambio de filas y no se puede obtener la factorización $A = LU$, pero sí se puede obtener la factorización

$$LU = PA.$$

Las matrices L y U tienen el mismo significado de la factorización LU . P es una matriz de permutación, es decir, se obtiene mediante permutación de filas de la matriz identidad I .

Si P y Q son matrices de permutación, entonces:

- PQ es una matriz de permutación.
- $P^{-1} = P^T$ (P es ortogonal).
- PA es una permutación de las filas de A .
- AP es una permutación de las columnas de A .

Una matriz de permutación P se puede representar de manera más compacta por medio de un vector $p \in \mathbb{R}^n$ con la siguiente convención:

$$P_i = I_{p_i}.$$

En palabras, la fila i de P es simplemente la fila p_i de I . Obviamente p debe cumplir:

$$\begin{aligned} p_i &\in \{1, 2, 3, \dots, n\} \quad \forall i \\ p_i &\neq p_j \quad \forall i \neq j. \end{aligned}$$

Por ejemplo, $p = (2, 4, 3, 1)$ representa la matriz

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

De la misma forma que en la factorización LU , los valores l_{ik} se almacenan en el sitio donde se anula el valor a_{ik} . El vector p inicialmente es $(1, 2, 3, \dots, n)$. A medida que se intercambian las filas de la matriz, se intercambian las componentes de p .

Ejemplo 11.5. Obtener la factorización $LU = PA$, donde

$$A = \begin{bmatrix} 4 & 3 & -2 & 1 \\ 3 & 2 & 1 & 5 \\ -2 & 3 & 1 & 2 \\ -5 & 0 & 1 & 1 \end{bmatrix}.$$

Inicialmente $p = (1, 2, 3, 4)$. Para buscar el mejor pivote, se intercambian las filas 1 y 4.

$$p = (4, 2, 3, 1), \quad \begin{bmatrix} -5 & 0 & 1 & 1 \\ 3 & 2 & 1 & 5 \\ -2 & 3 & 1 & 2 \\ 4 & 3 & -2 & 1 \end{bmatrix}.$$

Buscando ceros en la primera columna y almacenando allí los valores l_{ik} se obtiene:

$$\left[\begin{array}{cccc} -5 & 0 & 1 & 1 \\ -0.6 & 2 & 1.6 & 5.6 \\ 0.4 & 3 & 0.6 & 1.6 \\ -0.8 & 3 & -1.2 & 1.8 \end{array} \right].$$

Para buscar el mejor pivote, se intercambian las filas 2 y 3.

$$p = (4, 3, 2, 1), \quad \left[\begin{array}{cccc} -5 & 0 & 1 & 1 \\ 0.4 & 3 & 0.6 & 1.6 \\ -0.6 & 2 & 1.6 & 5.6 \\ -0.8 & 3 & -1.2 & 1.8 \end{array} \right].$$

Buscando ceros en la segunda columna y almacenando allí los valores l_{ik} se obtiene:

$$\left[\begin{array}{cccc} -5 & 0 & 1 & 1 \\ 0.4 & 3 & 0.6 & 1.6 \\ -0.6 & 0.6667 & 1.2 & 4.5333 \\ -0.8 & 1 & -1.8 & 0.2 \end{array} \right].$$

Para buscar el mejor pivote, se intercambian las filas 3 y 4.

$$p = (4, 3, 1, 2), \quad \left[\begin{array}{cccc} -5 & 0 & 1 & 1 \\ 0.4 & 3 & 0.6 & 1.6 \\ -0.8 & 1 & -1.8 & 0.2 \\ -0.6 & 0.6667 & 1.2 & 4.5333 \end{array} \right].$$

Buscando ceros en la tercera columna y almacenando allí los valores l_{ik} se obtiene:

$$\left[\begin{array}{cccc} -5 & 0 & 1 & 1 \\ 0.4 & 3 & 0.6 & 1.6 \\ -0.8 & 1 & -1.8 & 0.2 \\ -0.6 & 0.6667 & -0.6667 & 4.6667 \end{array} \right].$$

En esta última matriz y en el arreglo p está toda la información necesaria para obtener L , U , P . Entonces:

$$L = \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0.4 & 1 & 0 & 0 \\ -0.8 & 1 & 1 & 0 \\ -0.6 & 0.6667 & -0.6667 & 1 \end{array} \right].$$

$$U = \begin{bmatrix} -5 & 0 & 1 & 1 \\ 0 & 3 & 0.6 & 1.6 \\ 0 & 0 & -1.8 & 0.2 \\ 0 & 0 & 0 & 4.6667 \end{bmatrix}.$$

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \diamond$$

Si se desea resolver el sistema $Ax = b$ a partir de la descomposición $PA = LU$, se considera el sistema $P^{-1}LUx = b$, o sea, $P^T LUx = b$. Sean $z = LUx$ y $y = Ux$. La solución de $Ax = b$ tiene tres pasos:

- Resolver $P^T z = b$, o sea, $z = Pb$.
- Resolver $Ly = z$.
- Resolver $Ux = y$.

Ejemplo 11.6. Para la matriz A del ejemplo anterior, resolver $Ax = b$ con $b = [4 \ -8 \ -7 \ -8]^T$.

$$z = Pb = \begin{bmatrix} -8 \\ -7 \\ 4 \\ -8 \end{bmatrix}$$

$$Ly = z, \text{ entonces } y = \begin{bmatrix} -8 \\ -3.8 \\ 1.4 \\ -9.3333 \end{bmatrix}$$

$$Ux = y, \text{ entonces } x = \begin{bmatrix} 1 \\ 0 \\ -1 \\ -2 \end{bmatrix} \diamond$$

11.10 Método de Cholesky

Este método sirve para resolver el sistema $Ax = b$ cuando la matriz A es **definida positiva** (también llamada positivamente definida). Este tipo de matrices se presenta en problemas específicos de ingeniería y física, principalmente.

11.10.1 Matrices definidas positivas

Una matriz simétrica es definida positiva si

$$x^T Ax > 0, \quad \forall x \in \mathbb{R}^n, x \neq 0. \quad (11.5)$$

Para una matriz cuadrada cualquiera,

$$\begin{aligned} x^T Ax &= [x_1 \ x_2 \ \dots \ x_n] \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ &= [x_1 \ x_2 \ \dots \ x_n] \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n \end{bmatrix} \\ &= \sum_{i=1}^n \sum_{j=i}^n a_{ij}x_i x_j. \end{aligned}$$

Si A es simétrica,

$$x^T Ax = \sum_{i=1}^n a_{ii}x_i^2 + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n a_{ij}x_i x_j.$$

Ejemplo 11.7. Sea I la matriz identidad de orden n . Entonces $x^T I x = x^T x = \|x\|^2$. Luego la matriz I es definida positiva. \diamond

Ejemplo 11.8. Sea A la matriz nula de orden n . Entonces $x^T 0 x = 0$. Luego la matriz nula no es definida positiva. \diamond

Ejemplo 11.9. Sea

$$\begin{aligned} A &= \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}. \\ x^T Ax &= x_1^2 + 5x_2^2 + 4x_1x_2 \\ &= x_1^2 + 4x_1x_2 + 4x_2^2 + x_2^2 \\ &= (x_1 + 2x_2)^2 + x_2^2. \end{aligned}$$

Obviamente $x^T Ax \geq 0$. Además $x^T Ax = 0$ si y solamente si los dos sumandos son nulos, es decir, si y solamente si $x_2 = 0$ y $x_1 = 0$, o sea, cuando $x = 0$. Luego A es definida positiva. \diamond

Ejemplo 11.10. Sea

$$\begin{aligned} A &= \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}. \\ x^T Ax &= x_1^2 + 4x_2^2 + 4x_1x_2 \\ &= (x_1 + 2x_2)^2. \end{aligned}$$

Obviamente $x^T Ax \geq 0$. Pero si $x = (6, -3)$, entonces $x^T Ax = 0$. Luego A no es definida positiva. \diamond

Ejemplo 11.11. Sea

$$\begin{aligned} A &= \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}. \\ x^T Ax &= x_1^2 + 3x_2^2 + 4x_1x_2 \\ &= (x_1 + 2x_2)^2 - x_2^2. \end{aligned}$$

Si $x = (6, -3)$, entonces $x^T Ax = -9$. Luego A no es definida positiva. \diamond

Ejemplo 11.12. Sea

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Como A no es simétrica, entonces no es definida positiva. \diamond

Sean $\lambda_1, \lambda_2, \dots, \lambda_n$ los valores propios de A . Si A es simétrica, entonces todos sus valores propios son reales.

Sea δ_i el determinante de la submatriz de A , de tamaño $i \times i$, obtenida al quitar de A las filas $i + 1, i + 2, \dots, n$ y las columnas $i + 1, i + 2, \dots, n$. O sea,

$$\begin{aligned}\delta_1 &= \det([a_{11}]) = a_{11}, \\ \delta_2 &= \det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \\ \delta_3 &= \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{13} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \\ &\vdots \\ \delta_n &= \det(A).\end{aligned}$$

La definición 11.5 tiene relación directa con el nombre matriz definida positiva. Sin embargo, no es una manera fácil o práctica de saber cuándo una matriz simétrica es definida positiva, sobre todo si A es grande. El teorema siguiente presenta algunas de las caracterizaciones de las matrices definidas positivas. Para matrices pequeñas ($n \leq 4$) la caracterización por medio de los δ_i puede ser la de aplicación más sencilla. La última caracterización, llamada factorización de Cholesky, es la más adecuada para matrices grandes. En [Str86], [NoD88] y [Mor01] hay demostraciones y ejemplos.

Teorema 11.1. *Sea A simétrica. Las siguientes afirmaciones son equivalentes.*

- *A es definida positiva.*
- $\lambda_i > 0, \forall i.$
- $\delta_i > 0, \forall i.$
- *Existe U matriz triangular superior e invertible tal que $A = U^T U$.*

11.10.2 Factorización de Cholesky

Antes de estudiar el caso general, veamos la posible factorización para los ejemplos de la sección anterior.

La matriz identidad se puede escribir como $I = I^T I$, siendo I triangular superior invertible. Luego existe la factorización de Cholesky para la matriz identidad.

Si existe la factorización de Cholesky de una matriz, al ser U y U^T invertibles, entonces A debe ser invertible. Luego la matriz nula no tiene factorización de Cholesky.

Sea

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}.$$

Entonces

$$\begin{bmatrix} u_{11} & 0 \\ u_{12} & u_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}$$

$$u_{11}^2 = 1$$

$$u_{11}u_{12} = 2,$$

$$u_{12}^2 + u_{22}^2 = 5$$

Se deduce que

$$\begin{aligned} u_{11} &= 1 \\ u_{12} &= 2, \\ u_{22} &= 1, \\ U &= \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}. \end{aligned}$$

Entonces existe la factorización de Cholesky de A .

Cuando se calculó u_{11} se hubiera podido tomar $u_{11} = -1$ y se hubiera podido obtener otra matriz U . Se puede demostrar que si se escogen los elementos diagonales u_{ii} positivos, entonces la factorización, cuando existe, es única.

Sea

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}.$$

Entonces

$$\begin{bmatrix} u_{11} & 0 \\ u_{12} & u_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

$$u_{11}^2 = 1$$

$$u_{11}u_{12} = 2,$$

$$u_{12}^2 + u_{22}^2 = 4$$

Se deduce que

$$\begin{aligned} u_{11} &= 1 \\ u_{12} &= 2, \\ u_{22} &= 0, \\ U &= \begin{bmatrix} 1 & 2 \\ 0 & 0 \end{bmatrix}. \end{aligned}$$

Entonces, aunque existe U tal que $A = U^T U$, sin embargo no existe la factorización de Cholesky de A ya que U no es invertible.

Sea

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

Entonces

$$\begin{bmatrix} u_{11} & 0 \\ u_{12} & u_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$$

$$u_{11}^2 = 1$$

$$u_{11}u_{12} = 2,$$

$$u_{12}^2 + u_{22}^2 = 3$$

Se deduce que

$$\begin{aligned} u_{11} &= 1 \\ u_{12} &= 2, \\ u_{22}^2 &= -1. \end{aligned}$$

Entonces no existe la factorización de Cholesky de A .

En el caso general,

$$\begin{bmatrix} u_{11} & & & & \\ \vdots & & & & \\ u_{1k} & \cdots & u_{kk} & & \\ \vdots & & & & \\ u_{1j} & \cdots & u_{kj} & \cdots & u_{jj} \\ \vdots & & & & \\ u_{1n} & \cdots & u_{kn} & \cdots & u_{jn} & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & \cdots & u_{1k} & \cdots & u_{1j} & \cdots & u_{1n} \\ & & & & & & \\ & & & & & & \\ u_{kk} & \cdots & u_{kj} & \cdots & u_{kn} & & \\ & & & & & & \\ u_{jj} & \cdots & u_{jn} & & & & \\ & & & & & & \\ & & & & & & \\ u_{nn} & & & & & & \end{bmatrix}$$

El producto de la fila 1 de U^T por la columna 1 de U da:

$$u_{11}^2 = a_{11}.$$

Luego

$$u_{11} = \sqrt{a_{11}}. \quad (11.6)$$

El producto de la fila 1 de U^T por la columna j de U da:

$$u_{11}u_{1j} = a_{1j}.$$

Luego

$$u_{1j} = \frac{a_{1j}}{u_{11}}, \quad j = 2, \dots, n. \quad (11.7)$$

Al hacer el producto de la fila 2 de U^T por la columna 2 de U , se puede calcular u_{22} . Al hacer el producto de la fila 2 de U^T por la columna j de U , se puede calcular u_{2j} . Se observa que el cálculo de los elementos de U se hace fila por fila. Supongamos ahora que se conocen los elementos de las filas 1, 2, ..., $k - 1$ de U y se desea calcular los elementos de la fila k de U . El producto de la fila k de U^T por la columna k de U da:

$$\begin{aligned} \sum_{i=1}^k u_{ik}^2 &= a_{kk} \\ \sum_{i=1}^{k-1} u_{ik}^2 + u_{kk}^2 &= a_{kk}. \end{aligned}$$

Luego

$$u_{kk} = \sqrt{a_{kk} - \sum_{i=1}^{k-1} u_{ik}^2}, \quad k = 2, \dots, n. \quad (11.8)$$

El producto de la fila k de U^T por la columna j de U da:

$$\sum_{i=1}^k u_{ik} u_{ij} = a_{kj}.$$

Luego

$$u_{kj} = \frac{a_{kj} - \sum_{i=1}^{k-1} u_{ik} u_{ij}}{u_{kk}}, \quad k = 2, \dots, n, \quad j = k+1, \dots, n. \quad (11.9)$$

Si consideramos que el valor de la sumatoria es 0 cuando el límite inferior es más grande que el límite superior, entonces las fórmulas 11.8 y 11.9 pueden ser usadas para $k = 1, \dots, n$.

Ejemplo 11.13. Sea

$$A = \begin{bmatrix} 16 & -12 & 8 & -16 \\ -12 & 18 & -6 & 9 \\ 8 & -6 & 5 & -10 \\ -16 & 9 & -10 & 46 \end{bmatrix}.$$

$$u_{11} = \sqrt{16} = 4$$

$$u_{12} = \frac{-12}{4} = -3$$

$$u_{13} = \frac{8}{4} = 2$$

$$u_{14} = \frac{-16}{4} = -4$$

$$u_{22} = \sqrt{18 - (-3)^2} = 3$$

$$u_{23} = \frac{-6 - (-3)(2)}{3} = 0$$

$$u_{24} = \frac{9 - (-3)(-4)}{3} = -1$$

$$u_{33} = \sqrt{5 - (2^2 + 0^2)} = 1$$

$$u_{34} = \frac{-10 - (2(-4) + 0(-1))}{1} = -2$$

$$u_{44} = \sqrt{46 - ((-4)^2 + (-1)^2 + (-2)^2)} = 5.$$

Entonces,

$$U = \begin{bmatrix} 4 & -3 & 2 & -4 \\ 0 & 3 & 0 & -1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 5 \end{bmatrix}. \diamond$$

La factorización de Cholesky no existe cuando en la fórmula 11.8 la cantidad dentro del radical es negativa o nula. Utilizando el producto escalar, las fórmulas 11.8 y 11.9 se pueden reescribir así:

$$\begin{aligned} t &= a_{kk} - \text{prodEsc}(U(1:k-1,k) , U(1:k-1,k)), \\ u_{kk} &= \sqrt{t}, \\ u_{kj} &= \frac{a_{kj} - \text{prodEsc}(U(1:k-1,k) , U(1:k-1,j))}{u_{kk}} \end{aligned}$$

Para ahorrar espacio de memoria, los valores u_{kk} y u_{kj} se pueden almacenar sobre los antiguos valores de a_{kk} y a_{kj} . O sea, al empezar el algoritmo se tiene la matriz A . Al finalizar, en la parte triangular superior del espacio ocupado por A estará U .

$$t = a_{kk} - \text{prodEsc}(A(1:k-1,k) , A(1:k-1,k)), \quad (11.10)$$

$$a_{kk} = \sqrt{t}, \quad (11.11)$$

$$a_{kj} = \frac{a_{kj} - \text{prodEsc}(A(1:k-1,k) , A(1:k-1,j))}{a_{kk}} \quad (11.12)$$

El siguiente es el esquema del algoritmo para la factorización de Cholesky. Si acaba normalmente, la matriz A es definida positiva. Si en algún momento $t \leq \varepsilon$, entonces A no es definida positiva.

```

datos:  $A, \varepsilon$ 
para  $k = 1, \dots, n$ 
    cálculo de  $t$  según (11.10)
    si  $t \leq \varepsilon$  ent salir
     $a_{kk} = \sqrt{t}$ 
    para  $j = k+1, \dots, n$ 
        cálculo de  $a_{kj}$  según (11.12)
    fin-para  $j$ 
fin-para  $k$ 
```

La siguiente es la implementación en C, casi la traducción literal, del algoritmo anterior. Obviamente los subíndices de A varían entre 0 y $n - 1$.

```

int factChol(double *a, int n, double eps)
{
    // Factorizacion de Cholesky de la matriz simetrica A
    // almacenada por filas en el arreglo a.
    // A = Ut U, U es triangular sup., invertible.
    // Devuelve 0 si A no es definida positiva.
    //           1 si A es definida positiva
    //           en este caso U estara en
    //           la parte triangular superior de A.
    // Solamente se trabaja con la parte superior de A.

    int j, k, n1, kk, kj;
    double t;

    n1 = n+1;
    for( k = 0; k < n; k++){
        kk = k*n1;
        t = a[kk] - prodXY( &a[k], n, &a[k], n, k);
        if( t <= eps ) return 0;
        a[kk] = sqrt(t);
        for( j = k+1; j < n; j++){
            kj = kk + j-k;
            a[kj] = (a[kj]-prodXY(&a[k], n, &a[j], n, k))/a[kk];
        }
    }
    return 1;
}

```

11.10.3 Número de operaciones de la factorización

Para el cálculo del número de operaciones supongamos que el tiempo necesario para calcular una raíz cuadrada es del mismo orden de magnitud que el tiempo de una multiplicación.

	Sumas y restas	Multiplicaciones, divisiones y raíces
cálculo de u_{11}	0	1
cálculo de u_{12}	0	1
cálculo de u_{1n}	0	1
cálculo de u_{22}	1	2
cálculo de u_{23}	1	2
cálculo de u_{2n}	1	2
...		
cálculo de u_{nn}	$n - 1$	n

Agrupando por filas:

	Sumas y restas	Multiplicaciones, divisiones y raíces
cálculo de $U_1.$	$n(0)$	$n(1)$
cálculo de $U_2.$	$(n - 1)1$	$(n - 1)2$
cálculo de $U_3.$	$(n - 2)2$	$(n - 2)3$
...		
cálculo de $U_n.$	$1(n - 1)$	$1(n)$

Número de sumas y restas:

$$\sum_{i=1}^{n-1} (n - i)i = \frac{n^3 - n}{6} \approx \frac{n^3}{6}.$$

Número de multiplicaciones, divisiones y raíces:

$$\sum_{i=1}^n (n + 1 - i)i = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3} \approx \frac{n^3}{6}.$$

Número total de operaciones:

$$\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \approx \frac{n^3}{3}.$$

11.10.4 Solución del sistema

Una vez obtenida la factorización de Cholesky, resolver $Ax = b$ es lo mismo que resolver $U^T U x = b$. Al hacer el cambio de variable $Ux = y$, la solución del sistema $Ax = b$ se convierte en

$$\text{resolver} \quad U^T y = b, \quad (11.13)$$

$$\text{resolver} \quad Ux = y. \quad (11.14)$$

Resolver cada uno de los dos sistemas es muy fácil. El primero es triangular inferior, el segundo triangular superior. El número total de operaciones para resolver el sistema está dado por la factorización más la solución de dos sistemas triangulares.

$$\text{Número de operaciones} \approx \frac{n^3}{3} + 2n^2 \approx \frac{n^3}{3}.$$

Esto quiere decir que para valores grandes de n , resolver un sistema, con A definida positiva, por el método de Cholesky, gasta la mitad del tiempo requerido por el método de Gauss.

El método de Cholesky se utiliza para matrices definidas positivas. Pero no es necesario tratar de averiguar por otro criterio si la matriz es definida positiva. Simplemente se trata de obtener la factorización de Cholesky de A simétrica. Si fue posible, entonces A es definida positiva y se continúa con la solución de los dos sistemas triangulares. Si no fue posible obtener la factorización de Cholesky, entonces A no es definida positiva y no se puede aplicar el método de Cholesky para resolver $Ax = b$.

Ejemplo 11.14. Resolver

$$\begin{bmatrix} 16 & -12 & 8 \\ -12 & 18 & -6 \\ 8 & -6 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 76 \\ -66 \\ 46 \end{bmatrix}.$$

La factorización de Cholesky es posible (A es definida positiva):

$$U = \begin{bmatrix} 4 & -3 & 2 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}.$$

Al resolver $U^T y = b$ se obtiene

$$y = (19, -3, 4).$$

Finalmente, al resolver $Ux = y$ se obtiene

$$x = (3, -1, 2). \diamond$$

Para la implementación en C ya están todos los elementos, solamente basta ensamblarlos y tener en cuenta que no es necesario construir U^T . En la solución del sistema triangular inferior $U^T y = b$, es necesario trabajar con partes de las filas de U^T , o sea, con partes de las columnas de U .

```
int solChol(double *a, double *b, int n, double eps)
{
    // Solucion por el metodo de Cholesky
    // del sistema A x = b.

    // A debe ser simetrica.

    // Devuelve 0 si A no es definida positiva.
    //         1 si A es definida positiva.
    // En este caso la solucion estara en b.
    // La parte superior de A queda modificada.
    // Alli quedara U tal que U' U = A.

    // Solamente se trabaja con la parte superior de A.

    if( factChol(a, n, eps) == 0 ) return 0;
    solUTy(a, b, n);
    solTrSup(a, b, n, eps);
    return 1;
}

//-----
void solUTy(double *u, double *b, int n)
{
    // Solucion del sistema triangular inferior
    // U' y = b,
```

```

// donde U es triangular superior y esta
// en el arreglo u.
// Los elementos diagonales deben ser no nulos.

//  $U_{ij}$  esta en  $u[i*n+j]$ .

// La solucion quedara en b.

int i, n1;

n1 = n+1;
for( i = 0; i < n; i++){
    b[i] = (b[i] - prodXY( &u[i], n, b, 1, i))/u[i*n1];
}
}

```

11.11 Método de Gauss-Seidel

Los métodos de Gauss y Cholesky hacen parte de los métodos directos o finitos. Al cabo de un número finito de operaciones, en ausencia de errores de redondeo, se obtiene x^* solución del sistema $Ax = b$.

El método de Gauss-Seidel hace parte de los métodos llamados indirectos o iterativos. En ellos se comienza con $x^0 = (x_1^0, x_2^0, \dots, x_n^0)$, una aproximación inicial de la solución. A partir de x^0 se construye una nueva aproximación de la solución, $x^1 = (x_1^1, x_2^1, \dots, x_n^1)$. A partir de x^1 se construye x^2 (aquí el superíndice indica la iteración y no indica una potencia). Así sucesivamente se construye una sucesión de vectores $\{x^k\}$, con el objetivo, no siempre garantizado, de que

$$\lim_{k \rightarrow \infty} x^k = x^*.$$

Generalmente los métodos indirectos son una buena opción cuando la matriz es muy grande y dispersa o rala (*sparse*), es decir, cuando el número de elementos no nulos es pequeño comparado con n^2 , número total de elementos de A . En estos casos se debe utilizar una estructura de datos adecuada que permita almacenar únicamente los elementos no nulos.

En cada iteración del método de Gauss-Seidel, hay n subiteraciones. En la primera subiteración se modifica únicamente x_1 . Las demás coordenadas x_2, x_3, \dots, x_n no se modifican. El cálculo de x_1 se hace de tal manera que se satisfaga la primera ecuación.

$$\begin{aligned}x_1^1 &= \frac{b_1 - (a_{12}x_2^0 + a_{13}x_3^0 + \cdots + a_{1n}x_n^0)}{a_{11}}, \\x_i^1 &= x_i^0, \quad i = 2, \dots, n.\end{aligned}$$

En la segunda subiteración se modifica únicamente x_2 . Las demás coordenadas x_1, x_3, \dots, x_n no se modifican. El cálculo de x_2 se hace de tal manera que se satisfaga la segunda ecuación.

$$\begin{aligned}x_2^2 &= \frac{b_2 - (a_{21}x_1^1 + a_{23}x_3^1 + \cdots + a_{2n}x_n^1)}{a_{22}}, \\x_i^2 &= x_i^1, \quad i = 1, 3, \dots, n.\end{aligned}$$

Así sucesivamente, en la n -ésima subiteración se modifica únicamente x_n . Las demás coordenadas x_1, x_2, \dots, x_{n-1} no se modifican. El cálculo de x_n se hace de tal manera que se satisfaga la n -ésima ecuación.

$$\begin{aligned}x_n^n &= \frac{b_n - (a_{n1}x_1^{n-1} + a_{n3}x_3^{n-1} + \cdots + a_{nn}x_n^{n-1})}{a_{nn}}, \\x_i^n &= x_i^{n-1}, \quad i = 1, 2, \dots, n-1.\end{aligned}$$

Ejemplo 11.15. Resolver

$$\left[\begin{array}{cccc} 10 & 2 & -1 & 0 \\ 1 & 20 & -2 & 3 \\ -2 & 1 & 30 & 0 \\ 1 & 2 & 3 & 20 \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \right] = \left[\begin{array}{c} 26 \\ -15 \\ 53 \\ 47 \end{array} \right]$$

partiendo de $x^0 = (1, 2, 3, 4)$.

$$x_1^1 = \frac{26 - (2 \times 2 + (-1) \times 3 + 0 \times 4)}{10} = 2.5,$$

$$x^1 = (2.5, 2, 3, 4).$$

$$x_2^2 = \frac{-15 - (1 \times 2.5 + (-2) \times 3 + 3 \times 4)}{20} = -1.175,$$

$$x^2 = (2.5, -1.175, 3, 4).$$

$$x_3^3 = \frac{53 - (-2 \times 2.5 + 1 \times (-1.175) + 0 \times 4)}{30} = 1.9725,$$

$$x^3 = (2.5, -1.175, 1.9725, 4).$$

$$x_4^4 = \frac{47 - (1 \times 2.5 + 2 \times (-1.175) + 3 \times 1.9725)}{20} = 2.0466,$$

$$x^4 = (2.5, -1.175, 1.9725, 2.0466).$$

Una vez que se ha hecho una iteración completa (n subiteraciones), se utiliza el último x obtenido como aproximación inicial y se vuelve a empezar; se calcula x_1 de tal manera que se satisfaga la primera ecuación, luego se calcula x_2 ... A continuación están las iteraciones siguientes para el ejemplo anterior.

3.0323	-1.1750	1.9725	2.0466
3.0323	-1.0114	1.9725	2.0466
3.0323	-1.0114	2.0025	2.0466
3.0323	-1.0114	2.0025	1.9991
3.0025	-1.0114	2.0025	1.9991
3.0025	-0.9997	2.0025	1.9991
3.0025	-0.9997	2.0002	1.9991
3.0025	-0.9997	2.0002	1.9998
3.0000	-0.9997	2.0002	1.9998
3.0000	-1.0000	2.0002	1.9998
3.0000	-1.0000	2.0000	1.9998
3.0000	-1.0000	2.0000	2.0000
3.0000	-1.0000	2.0000	2.0000
3.0000	-1.0000	2.0000	2.0000
3.0000	-1.0000	2.0000	2.0000

Teóricamente, el método de Gauss-Seidel puede ser un proceso infinito. En la práctica el proceso se acaba cuando de x^k a x^{k+n} los cambios son muy pequeños. Esto quiere decir que el x actual es casi la solución x^* .

Como el método no siempre converge, entonces otra detención del proceso, no deseada pero posible, está determinada cuando el número de iteraciones realizadas es igual a un número máximo de iteraciones previsto.

El siguiente ejemplo no es convergente, ni siquiera empezando de una aproximación inicial muy cercana a la solución. La solución exacta es $x = (1, 1, 1)$.

Ejemplo 11.16. Resolver

$$\begin{bmatrix} -1 & 2 & 10 \\ 11 & -1 & 2 \\ 1 & 5 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 11 \\ 12 \\ 8 \end{bmatrix}$$

partiendo de $x^0 = (1.0001, 1.0001, 1.0001)$.

$$\begin{array}{ccc} 1.0012 & 1.0001 & 1.0001 \\ 1.0012 & 1.0134 & 1.0001 \\ 1.0012 & 1.0134 & 0.9660 \end{array}$$

$$\begin{array}{ccc} 0.6863 & 1.0134 & 0.9660 \\ 0.6863 & -2.5189 & 0.9660 \\ 0.6863 & -2.5189 & 9.9541 \end{array}$$

$$\begin{array}{ccc} 83.5031 & -2.5189 & 9.9541 \\ 83.5031 & 926.4428 & 9.9541 \\ 83.5031 & 926.4428 & -2353.8586 \end{array}$$

Algunos criterios garantizan la convergencia del método de Gauss-Seidel. Por ser condiciones suficientes para la convergencia son criterios demasiado fuertes, es decir, la matriz A puede no cumplir estos requisitos y sin embargo el método puede ser convergente. En la práctica, con frecuencia, es muy dispendioso poder aplicar estos criterios.

Una matriz cuadrada es de **diagonal estrictamente dominante por filas** si en cada fila el valor absoluto del elemento diagonal es mayor

que la suma de los valores absolutos de los otros elementos de la fila,

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|, \quad \forall i.$$

Teorema 11.2. Si A es de diagonal estrictamente dominante por filas, entonces el método de Gauss-Seidel converge para cualquier x^0 inicial.

Teorema 11.3. Si A es definida positiva, entonces el método de Gauss-Seidel converge para cualquier x^0 inicial.

Teóricamente el método de Gauss-Seidel se debería detener cuando $\|x^k - x^*\| < \varepsilon$. Sin embargo la condición anterior necesita conocer x^* , que es precisamente lo que se está buscando. Entonces, de manera práctica el método de GS se detiene cuando $\|x^k - x^{k+n}\| < \varepsilon$.

Dejando de lado los superíndices, las fórmulas del método de Gauss-Seidel se pueden reescribir para facilitar el algoritmo y para mostrar que $\|x^k - x^*\|$ y $\|x^k - x^{k+n}\|$ están relacionadas.

$$\begin{aligned} x_i &\leftarrow \frac{b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j}{a_{ii}}, \\ x_i &\leftarrow \frac{b_i - \sum_{j=1}^n a_{ij} x_j + a_{ii} x_i}{a_{ii}}, \\ x_i &\leftarrow x_i + \frac{b_i - A_i \cdot x}{a_{ii}}. \end{aligned}$$

Sean

$$\begin{aligned} r_i &= b_i - A_i \cdot x, \\ \delta_i &= \frac{r_i}{a_{ii}}. \end{aligned}$$

El valor r_i es simplemente el error, residuo o resto que se comete en la i -ésima ecuación al utilizar el x actual. Si $r_i = 0$, entonces la ecuación i -ésima se satisface perfectamente. El valor δ_i es la modificación que sufre x_i en una iteración.

Sean $r = (r_1, r_2, \dots, r_n)$, $\delta = (\delta_1, \delta_2, \dots, \delta_n)$. Entonces $x^{k+n} = x^k + \delta$. Además x^k es solución si y solamente si $r = 0$, o sea, si y solamente

$\delta = 0$. Lo anterior justifica que el método de GS se detenga cuando $\|\delta\| \leq \varepsilon$. La norma $\|\delta\|$ puede ser la norma euclíadiana o $\max |\delta_i|$ o $\sum |\delta_i|$.

Si en el criterio de parada del algoritmo se desea enfatizar sobre los errores o residuos, entonces se puede comparar $\|\delta\|$ con $\varepsilon / \|(a_{11}, \dots, a_{nn})\|$; por ejemplo,

$$\|\delta\| \leq \frac{\varepsilon}{\max |a_{ii}|}.$$

El esquema del algoritmo para resolver un sistema de ecuaciones por el método de Gauss-Seidel es:

```

datos: A, b, x0, ε, maxit
x = x0
para k = 1,...,maxit
    nrmD ← 0
    para i = 1,...,n
        δi = (bi - prodEsc(Ai.., x)) / aii
        xi ← xi + δi
        nrmD ← nrmD + δi
    fin-para i
    si nrmD ≤ ε ent x* ≈ x, salir
fin-para k

```

La siguiente es la implementación en C, casi la traducción literal, del algoritmo anterior.

```

int gaussSei(double *a, double *b, double *x, int n,
             double eps, int maxit)
{
    // Metodo de Gauss-Seidel para resolver A x = b

    // Devuelve:
    // 1 : si se obtuvo aproximadamente la solucion.
    // 0 : si hay un elemento diagonal nulo.
    // 2 : si hubo demasiadas iteraciones, mas de maxit.

    // Entrando a la funcion, x contiene la
    // aproximacion inicial de la solucion.
    // Saliendo x tendra la ultima aproximacion.

```

```

int i, n1, k;
double nrmD, di;

n1 = n+1;
for(i=0;i<n;i++) if(fabs(a[i*n1])<=1.e-40) return 0;

for( k=1; k <= maxit; k++){
    nrmD = 0.0;
    for( i=0; i<n; i++){
        di = (b[i] - prodXY(&a[i*n], x, n))/a[i*n1];
        nrmD += fabs(di);
        x[i] += di;
    }
    if( nrmD <= eps ) return 1;
}
return 2;
}

```

11.12 Solución por mínimos cuadrados

Consideremos ahora un sistema de ecuaciones $Ax = b$, no necesariamente cuadrado, donde A es una matriz $m \times n$ cuyas columnas son linealmente independientes. Esto implica que hay más filas que columnas, $m \geq n$, y que además el rango de A es n . Es muy probable que este sistema no tenga solución, es decir, tal vez no existe x que cumpla exactamente las m igualdades. Se desea que

$$\begin{aligned}
Ax &= b, \\
Ax - b &= 0, \\
\|Ax - b\| &= 0, \\
\|Ax - b\|_2 &= 0, \\
\|Ax - b\|_2^2 &= 0.
\end{aligned}$$

Es posible que lo deseado no se cumpla, entonces se quiere que el incumplimiento (el error) sea lo más pequeño posible. Se desea minimizar esa cantidad,

$$\min \|Ax - b\|_2^2. \quad (11.15)$$

El vector x que minimice $\|Ax - b\|_2^2$ se llama solución por mínimos cuadrados. Como se verá más adelante, tal x existe y es único (suponiendo que las columnas de A son linealmente independientes).

Con el ánimo de hacer más clara la deducción, supongamos que A es una matriz 4×3 . Sea $f(x) = \|Ax - b\|_2^2$,

$$f(x) = (a_{11}x_1 + a_{12}x_2 + a_{13}x_3 - b_1)^2 + (a_{21}x_1 + a_{22}x_2 + a_{23}x_3 - b_2)^2 + \\ (a_{31}x_1 + a_{32}x_2 + a_{33}x_3 - b_3)^2 + (a_{41}x_1 + a_{42}x_2 + a_{43}x_3 - b_4)^2.$$

11.12.1 Derivadas parciales

Este libro está dirigido principalmente a estudiantes de segundo semestre, quienes todavía no conocen el cálculo en varias variables. En este capítulo y en el siguiente se requiere saber calcular derivadas parciales. A continuación se presenta una breve introducción al cálculo (mecánico) de las derivadas parciales.

Sea φ una función de varias variables con valor real, $\varphi : \mathbb{R}^n \rightarrow \mathbb{R}$. Bajo ciertas condiciones de existencia, la derivada parcial de φ con respecto a x_i se obtiene considerando las otras variables como constantes y derivando $\varphi(x_1, x_2, \dots, x_n)$ únicamente con respecto a x_i . Esta derivada parcial se denota

$$\frac{\partial \varphi}{\partial x_i}.$$

Evaluada en un punto específico \bar{x} , se denota

$$\frac{\partial \varphi}{\partial x_i}(\bar{x}).$$

Por ejemplo, si $\varphi(x_1, x_2, x_3, x_4) = (4x_1^3 + 6x_4)^9 + 5x_1x_2 + 8x_4$,

$$\begin{aligned}\frac{\partial \varphi}{\partial x_1} &= 9(4x_1^3 + 6x_4)^8(12x_1^2) + 5x_2, \\ \frac{\partial \varphi}{\partial x_2} &= 5x_1, \\ \frac{\partial \varphi}{\partial x_3} &= 0, \\ \frac{\partial \varphi}{\partial x_4} &= 54(4x_1^3 + 6x_4)^8 + 8.\end{aligned}$$

11.12.2 Ecuaciones normales

Para obtener el mínimo de f se requiere que las tres derivadas parciales, $\partial f / \partial x_1$, $\partial f / \partial x_2$ y $\partial f / \partial x_3$, sean nulas.

$$\begin{aligned}\frac{\partial f}{\partial x_1} = & 2(a_{11}x_1 + a_{12}x_2 + a_{13}x_3 - b_1)a_{11} \\ & + 2(a_{21}x_1 + a_{22}x_2 + a_{23}x_3 - b_2)a_{21} \\ & + 2(a_{31}x_1 + a_{32}x_2 + a_{33}x_3 - b_3)a_{31} \\ & + 2(a_{41}x_1 + a_{42}x_2 + a_{43}x_3 - b_4)a_{41}.\end{aligned}$$

Escribiendo de manera matricial,

$$\begin{aligned}\frac{\partial f}{\partial x_1} = & 2(A_1 \cdot x - b_1)a_{11} + 2(A_2 \cdot x - b_2)a_{21} + 2(A_3 \cdot x - b_3)a_{31} \\ & + 2(A_4 \cdot x - b_4)a_{41}.\end{aligned}$$

Si B es una matriz y u un vector columna, entonces $(Bu)_i = B_i \cdot u$.

$$\begin{aligned}\frac{\partial f}{\partial x_1} &= 2 \left(((Ax)_1 - b_1)a_{11} + ((Ax)_2 - b_2)a_{21} + ((Ax)_3 - b_3)a_{31} \right. \\ &\quad \left. + ((Ax)_4 - b_4)a_{41} \right), \\ &= 2 \sum_{i=1}^4 (Ax - b)_i a_{i1}, \\ &= 2 \sum_{i=1}^4 (A \cdot 1)_i (Ax - b)_i, \\ &= 2 \sum_{i=1}^4 (A^T \cdot 1)_i (Ax - b)_i, \\ &= 2 A^T \cdot 1 \cdot (Ax - b), \\ &= 2 (A^T (Ax - b))_1\end{aligned}$$

De manera semejante

$$\begin{aligned}\frac{\partial f}{\partial x_2} &= 2 (A^T (Ax - b))_2, \\ \frac{\partial f}{\partial x_3} &= 2 (A^T (Ax - b))_3\end{aligned}$$

Igualando a cero las tres derivadas parciales y quitando el 2 se tiene

$$\begin{aligned}(A^T(Ax - b))_1 &= 0, \\ (A^T(Ax - b))_2 &= 0, \\ (A^T(Ax - b))_3 &= 0\end{aligned}$$

Es decir,

$$\begin{aligned}A^T(Ax - b) &= 0, \\ A^T Ax &= A^T b.\end{aligned}\tag{11.16}$$

Las ecuaciones (11.16) se llaman **ecuaciones normales** para la solución (o seudosolución) de un sistema de ecuaciones por mínimos cuadrados.

La matriz $A^T A$ es simétrica de tamaño $n \times n$. En general, si A es una matriz $m \times n$ de rango r , entonces $A^T A$ también es de rango r (ver [Str86]). Como se supuso que el rango de A es n , entonces $A^T A$ es invertible. Más aún, $A^T A$ es definida positiva.

Por ser $A^T A$ invertible, hay una única solución de (11.16), o sea, hay un solo vector x que hace que las derivadas parciales sean nulas. En general, las derivadas parciales nulas son simplemente una condición necesaria para obtener el mínimo de una función (también lo es para máximos o para puntos de silla), pero en este caso, como $A^T A$ es definida positiva, f es convexa, y entonces anular las derivadas parciales se convierte en condición necesaria y suficiente para el mínimo.

En resumen, si las columnas de A son linealmente independientes, entonces la solución por mínimos cuadrados existe y es única. Para obtener la solución por mínimos cuadrados se resuelven las ecuaciones normales.

Como $A^T A$ es definida positiva, (11.16) se puede resolver por el método de Cholesky. Si $m \geq n$ y al hacer la factorización de Cholesky resulta que $A^T A$ no es definida positiva, entonces las columnas de A son linealmente dependientes.

Si el sistema $Ax = b$ tiene solución exacta, ésta coincide con la solución por mínimos cuadrados.

Ejemplo 11.17. Resolver por mínimos cuadrados:

$$\begin{bmatrix} 2 & 1 & 0 \\ -1 & -2 & 3 \\ -2 & 2 & 1 \\ 5 & 4 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3.1 \\ 8.9 \\ -3.1 \\ 0.1 \end{bmatrix}.$$

Las ecuaciones normales dan:

$$\begin{bmatrix} 34 & 20 & -15 \\ 20 & 25 & -12 \\ -15 & -12 & 14 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4.0 \\ -20.5 \\ 23.4 \end{bmatrix}$$

La solución por mínimos cuadrados es:

$$x = (2.0252, -1.0132, 2.9728).$$

El error, $Ax - b$, es:

$$\begin{bmatrix} -0.0628 \\ 0.0196 \\ -0.0039 \\ 0.0275 \end{bmatrix}. \diamond$$

Ejemplo 11.18. Resolver por mínimos cuadrados:

$$\begin{bmatrix} 2 & 1 & 3 \\ -1 & -2 & 0 \\ -2 & 2 & -6 \\ 5 & 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 9 \\ -3 \\ 0 \end{bmatrix}.$$

Las ecuaciones normales dan:

$$\begin{bmatrix} 34 & 20 & 48 \\ 20 & 25 & 15 \\ 48 & 15 & 81 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -21 \\ 27 \end{bmatrix}$$

Al tratar de resolver este sistema de ecuaciones por el método de Cholesky; no se puede obtener la factorización de Cholesky, luego $A^T A$ no es definida positiva, es decir, las columnas de A son linealmente dependientes. Si se aplica el método de Gauss, se obtiene que $A^T A$ es singular y se concluye que las columnas de A son linealmente dependientes. \diamond

Ejemplo 11.19. Resolver por mínimos cuadrados:

$$\begin{bmatrix} 2 & 1 \\ -1 & -2 \\ -2 & 2 \\ 5 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ -6 \\ 6 \end{bmatrix}.$$

Las ecuaciones normales dan:

$$\begin{bmatrix} 34 & 20 \\ 20 & 25 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 48 \\ 15 \end{bmatrix}$$

La solución por mínimos cuadrados es:

$$x = (2, -1).$$

El error, $Ax - b$, es:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

En este caso, el sistema inicial tenía solución exacta y la solución por mínimos cuadrados coincide con ella. ◇

La implementación eficiente de la solución por mínimos cuadrados, vía ecuaciones normales, debe tener en cuenta algunos detalles. No es necesario construir toda la matriz simétrica $A^T A$ (n^2 elementos). Basta con almacenar en un arreglo de tamaño $n(n + 1)/2$ la parte triangular superior de $A^T A$.

Este almacenamiento puede ser por filas, es decir, primero los n elementos de la primera fila, enseguida los $n - 1$ elementos de la segunda fila a partir del elemento diagonal, después los $n - 2$ de la tercera fila a partir del elemento diagonal y así sucesivamente hasta almacenar un solo elemento de la fila n . Si se almacena la parte triangular superior de $A^T A$ por columnas, se almacena primero un elemento de la primera columna, enseguida dos elementos de la segunda columna y así sucesivamente. Cada una de las dos formas tiene sus ventajas y desventajas. La solución por el método de Cholesky debe tener en cuenta este tipo de estructura de almacenamiento de la información.

Otros métodos eficientes para resolver sistemas de ecuaciones por mínimos cuadrados utilizan matrices ortogonales de Givens o de Householder.

Ejercicios

11.1 Resuelva el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 5 & -1 & 2 & 1 \\ 0 & 4 & 1 & -2 \\ 0 & 0 & 2 & 4 \\ 0 & 0 & 0 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} -4 \\ 8 \\ 0 \\ -6 \end{bmatrix}.$$

11.2 Resuelva por el método de Gauss con pivoteo parcial el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -12 & -10 \\ -12 & 45 & 30 \\ -10 & 30 & 41 \end{bmatrix}, \quad b = \begin{bmatrix} 6 \\ -9 \\ -31 \end{bmatrix}.$$

11.3 Halle la factorización $LU = PA$ para la matriz A del ejercicio anterior. Resuelva el sistema $Ax = c$, con

$$c = \begin{bmatrix} -36 \\ 126 \\ 122 \end{bmatrix}.$$

11.4 Resuelva por el método de Gauss con pivoteo parcial el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -12 & -10 \\ -12 & 45 & 30 \\ -10 & 45 & 25 \end{bmatrix}, \quad b = \begin{bmatrix} -18 \\ 63 \\ 60 \end{bmatrix}.$$

11.5 Resuelva por el método de Gauss con pivoteo parcial el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -15 & -10 \\ -12 & 45 & 15 \\ -10 & 75/2 & 25 \end{bmatrix}, \quad b = \begin{bmatrix} -42 \\ 96 \\ 105 \end{bmatrix}.$$

11.6 Resuelva por el método de Cholesky el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -12 & -10 \\ -12 & 45 & 30 \\ -10 & 30 & 41 \end{bmatrix}, \quad b = \begin{bmatrix} 6 \\ -9 \\ -31 \end{bmatrix}.$$

11.7 Resuelva por el método de Cholesky el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & 2 & 4 & -2 \\ 2 & 5 & 6 & -3 \\ 4 & 6 & 8 & -4 \\ -2 & -3 & -4 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 8 \\ 10 \\ 14 \\ -6 \end{bmatrix}.$$

11.8 Resuelva por el método de Cholesky el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -2 & 2 \\ -2 & 5 & 1 \\ 2 & 1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 4 \\ 4 \\ 5 \end{bmatrix}.$$

11.9 Resuelva por el método de Gauss-Seidel el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 5 & 1 & -1 \\ 2 & 4 & 1 \\ 1 & 2 & 10 \end{bmatrix}, \quad b = \begin{bmatrix} 9 \\ 20 \\ 48 \end{bmatrix}.$$

11.10 Resuelva por el método de Gauss-Seidel el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -12 & -10 \\ -12 & 45 & 30 \\ -10 & 30 & 41 \end{bmatrix}, \quad b = \begin{bmatrix} 6 \\ -9 \\ -31 \end{bmatrix}.$$

11.11 Resuelva por el método de Gauss-Seidel el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 1 & 4 & 5 \\ 6 & 2 & 7 \\ 8 & 9 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 10 \\ 15 \\ 20 \end{bmatrix}.$$

11.12 Resuelva por mínimos cuadrados el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 12 & 15 \end{bmatrix}, \quad b = \begin{bmatrix} 10 \\ 15 \\ 20 \\ 25 \end{bmatrix}.$$

11.13 Resuelva por mínimos cuadrados el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}, \quad b = \begin{bmatrix} 10 \\ 15 \\ 20 \\ 25 \end{bmatrix}.$$

- 11.14** Utilizando apuntadores dobles para almacenar la matriz A , elabore un programa (y una función) que calcule el determinante de A .
- 11.15** Utilizando apuntadores dobles para almacenar la matriz A , haga un programa (y una función) para resolver un sistema por el método de Gauss con pivoteo parcial.
- 11.16** Utilizando apuntadores dobles para almacenar la matriz rectangular A , elabore un programa (y una función) que obtenga la forma escalonada reducida por filas.
- 11.17** Utilizando apuntadores dobles para almacenar la matriz A , haga un programa (y una función) para resolver un sistema por el método de Cholesky.
- 11.18** Utilizando apuntadores dobles para almacenar la matriz A , haga un programa (y una función) para resolver un sistema por el método de Gauss-Seidel.
- 11.19** La información indispensable de una matriz simétrica es la parte superior, o sea, $n(n+1)/2$ datos. Ésta se puede almacenar, fila por fila, en un vector

$$(a_{11}, a_{12}, a_{13}, \dots, a_{1n}, a_{22}, a_{23}, \dots, a_{2n}, \dots, a_{n-1,n-1}, a_{n-1,n}, a_{nn}),$$

que ocupa aproximadamente la mitad del espacio de los n^2 elementos de la matriz completa. Haga un programa para almacenar de A únicamente la parte superior, hacer la factorización de Cholesky y resolver el sistema $Ax = b$.

- 11.20** Resuelva por el método de Cholesky el sistema $Ax = b$, donde

$$A = \begin{bmatrix} 4 & -2 & 2 & 0 & 0 & 0 & 0 \\ -2 & 5 & -1 & 2 & 0 & 0 & 0 \\ 2 & -1 & 5 & 4 & 6 & 0 & 0 \\ 0 & 2 & 4 & 6 & 5 & -2 & 0 \\ 0 & 0 & 6 & 5 & 11 & 4 & 3 \\ 0 & 0 & 0 & -2 & 4 & 12 & 2 \\ 0 & 0 & 0 & 0 & 3 & 2 & 14 \end{bmatrix}, \quad b = \begin{bmatrix} -8 \\ 12 \\ 8 \\ 12 \\ 23 \\ 18 \\ -6 \end{bmatrix}.$$

- 11.21** Se dice que una matriz simétrica es una matriz banda de ancho $2m-1$ si $a_{ij} = 0$ cuando $|j-i| \geq m$. Demuestre que si A es definida

positiva y la factorización de Cholesky es $A = U^T U$, entonces U es una matriz triangular superior banda, de ancho m : $a_{ij} = 0$ si $i > j$ y $a_{ij} = 0$ si $j - i \geq m$. En el ejercicio anterior $m = 3$.

- 11.22** La información indispensable de una matriz simétrica de ancho de banda $2m - 1$ se puede almacenar en una matriz $m \times n$. Para la matriz del penúltimo ejercicio,

$$\begin{bmatrix} 4 & -2 & 2 \\ 5 & -1 & 2 \\ 5 & 4 & 6 \\ 6 & 5 & -2 \\ 11 & 4 & 3 \\ 12 & 2 & \\ 14 & & \end{bmatrix}.$$

Haga un programa para almacenar de A únicamente la banda superior, hacer la factorización de Cholesky y resolver el sistema $Ax = b$.

- 11.23** Se define la densidad de una matriz de tamaño $m \times n$ como el porcentaje de componentes no nulos de una matriz

$$\text{densidad}(A) = \frac{\text{número de } a_{ij} \text{ no nulos}}{mn} 100.$$

Una matriz, generalmente de tamaño muy grande, se llama dispersa o rala, (*sparse*), si su densidad es pequeña, es decir menor que 10%, o menor que 5%, o 1%. Estas matrices se presentan con frecuencia en problemas reales de física, ingeniería, economía, etc. La información indispensable, los elementos no nulos, se puede almacenar por una lista de triples

$$\begin{aligned} & (i_1, j_1, v_1) \\ & (i_2, j_2, v_2) \\ & \vdots \\ & (i_k, j_k, v_k) \\ & \vdots \end{aligned}$$

donde $a_{i_k, j_k} = v_k$. Haga un programa para almacenar de A únicamente los elementos no nulos y resolver por el método de Gauss-Seidel el sistema $Ax = b$.

12

Solución de ecuaciones

Uno de los problemas más corrientes en matemáticas consiste en resolver una ecuación, es decir, encontrar un valor $x^* \in \mathbb{R}$ que satisfaga

$$f(x) = 0,$$

donde f es una función de variable y valor real, o sea,

$$f : \mathbb{R} \rightarrow \mathbb{R}.$$

Este x^* se llama solución de la ecuación. A veces también se dice que x^* es una raíz. Algunos ejemplos sencillos de ecuaciones son:

$$\begin{aligned}x^5 - 3x^4 + 10x - 8 &= 0, \\ e^x - x^3 + 8 &= 0, \\ \frac{x^2 + x}{\cos(x-1) + 2} - x &= 0.\end{aligned}$$

En algunos casos no se tiene una expresión sencilla de f , sino que $f(x)$ corresponde al resultado de un proceso; por ejemplo:

$$\int_{-\infty}^x e^{-t^2} dt - 0.2 = 0.$$

Lo mínimo que se le exige a f es que sea continua. Si no es continua en todo \mathbb{R} , por lo menos debe ser continua en un intervalo $[a, b]$ donde se busca la raíz. Algunos métodos requieren que f sea derivable. Para la

aplicación de algunos teoremas de convergencia, no para el método en sí, se requieren derivadas de orden superior.

Los métodos generales de solución de ecuaciones sirven únicamente para hallar raíces reales. Algunos métodos específicos para polinomios permiten obtener raíces complejas.

Los métodos presuponen que la ecuación $f(x) = 0$ tiene solución. Es necesario, antes de aplicar mecánicamente los métodos, estudiar la función, averiguar si tiene raíces, ubicarlas aproximadamente. En algunos casos muy difíciles no es posible hacer un análisis previo de la función, entonces hay que utilizar de manera mecánica uno o varios métodos, pero sabiendo que podrían ser ineficientes o, simplemente, no funcionar.

La mayoría de los métodos parten de x_0 , aproximación inicial de x^* , a partir del cual se obtiene x_1 . A partir de x_1 se obtiene x_2 , después x_3 , y así sucesivamente se construye la sucesión $\{x_k\}$ con el objetivo, no siempre cumplido, de que

$$\lim_{k \rightarrow \infty} x_k = x^*.$$

El proceso anterior es teóricamente infinito, y obtendría la solución después de haber hecho un número infinito de cálculos. En la práctica el proceso se detiene cuando se obtenga una aproximación suficientemente buena de x^* . Esto querría decir que el proceso se detendría cuando

$$|x_k - x^*| \leq \varepsilon,$$

para un ε dado. El anterior criterio supone el conocimiento de x^* , que es justamente lo buscado. Entonces se utiliza el criterio, éste si aplicable,

$$|f(x_k)| \leq \varepsilon.$$

En la mayoría de los casos, cuanto más cerca esté x_0 de x^* , más rápidamente se obtendrá una buena aproximación de x^* .

Otros métodos parten de un intervalo inicial $[a_0, b_0]$, en el cual se sabe que existe una raíz x^* . A partir de él, se construye otro intervalo $[a_1, b_1]$, contenido en el anterior, en el que también está x^* y que es de menor tamaño. De manera análoga se construye $[a_2, b_2]$. Se espera que

la sucesión formada por los tamaños tienda a 0. Explícitamente,

$$\begin{aligned}x^* &\in [a_0, b_0], \\[a_{k+1}, b_{k+1}] &\subset [a_k, b_k], \quad k = 1, 2, \dots, \\x^* &\in [a_k, b_k], \quad k = 1, 2, \dots, \\\lim_{k \rightarrow \infty} (b_k - a_k) &= 0.\end{aligned}$$

En este caso, el proceso se detiene cuando se obtiene un intervalo suficientemente pequeño,

$$|b_k - a_k| \leq \varepsilon.$$

Cualquiera de los puntos del último intervalo es una buena aproximación de x^* .

12.1 Método de Newton

También se conoce como el método de Newton-Raphson. Dado x_0 , se construye la recta tangente en $(x_0, f(x_0))$. El valor de x donde esta recta corta el eje x es el nuevo valor x_1 . Ahora se construye la recta tangente en el punto $(x_1, f(x_1))$. El punto de corte entre la recta y el eje x determina x_2 ...

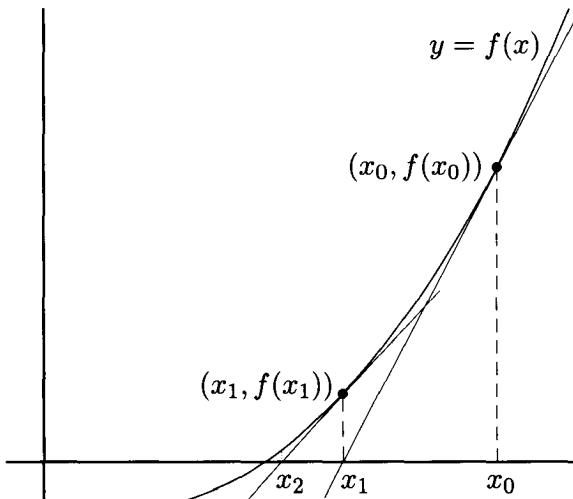


Figura 12.1 *Método de Newton.*

En el caso general, dado x_k , se construye la recta tangente en el punto $(x_k, f(x_k))$,

$$y = f'(x_k)(x - x_k) + f(x_k).$$

Para $y = 0$ se tiene $x = x_{k+1}$,

$$0 = f'(x_k)(x_{k+1} - x_k) + f(x_k).$$

Entonces

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}} \quad (12.1)$$

Ejemplo 12.1. Aplicar el método de Newton a la ecuación $x^5 - 3x^4 + 10x - 8 = 0$, partiendo de $x_0 = 3$.

k	x_k	$f(x_k)$	$f'(x_k)$
0	3.000000	2.200000E+01	91.000000
1	2.758242	5.589425E+00	47.587479
2	2.640786	9.381331E-01	32.171792
3	2.611626	4.892142E-02	28.848275
4	2.609930	1.590178E-04	28.660840
5	2.609924	1.698318E-09	28.660228
6	2.609924	-2.838008E-15	28.660227

Las raíces reales del polinomio $x^5 - 3x^4 + 10x - 8$ son: 2.6099, 1.3566, 1. Tomando otros valores iniciales el método converge a estas raíces. Si se toma $x_0 = 2.1$, se esperaría que el método vaya hacia una de las raíces cercanas, 2.6099 o 1.3566 . Sin embargo, hay convergencia hacia 1.

k	x_k	$f(x_k)$	$f'(x_k)$
0	2.100000	-4.503290e+00	-3.891500
1	0.942788	-1.974259e-01	3.894306
2	0.993484	-1.988663e-02	3.103997
3	0.999891	-3.272854e-04	3.001745
4	1.000000	-9.509814e-08	3.000001
5	1.000000	-7.993606e-15	3.000000

El método de Newton es muy popular por sus ventajas:

- Sencillez.
- Generalmente converge.
- En la mayoría de los casos, cuando converge, lo hace rápidamente.

También tiene algunas desventajas:

- Puede no converger.
- Presenta problemas cuando $f'(x_k) \approx 0$.
- Requiere poder evaluar, en cada iteración, el valor $f'(x)$.

La implementación del método de Newton debe tener en cuenta varios aspectos. Como no es un método totalmente seguro, debe estar previsto un número máximo de iteraciones, llamado por ejemplo `maxit`. Para una precisión ε_f , la detención deseada para el proceso iterativo se tiene cuando $|f(x_k)| \leq \varepsilon_f$. Otra detención posible se da cuando dos valores de x son casi iguales, es decir, cuando $|x_k - x_{k-1}| \leq \varepsilon_x$. Se acostumbra a utilizar el cambio relativo, o sea, $|x_k - x_{k-1}|/|x_k| \leq \varepsilon_x$. Para evitar las divisiones por cero, se usa $|x_k - x_{k-1}|/(1 + |x_k|) \leq \varepsilon_x$. Finalmente, siempre hay que evitar las divisiones por cero o por valores casi nulos. Entonces, otra posible parada, no deseada, corresponde a $|f'(x_k)| \leq \varepsilon_0$. El algoritmo para el método de Newton puede tener el siguiente esquema:

```

datos: x0, maxit,  $\varepsilon_f$ ,  $\varepsilon_x$ ,  $\varepsilon_0$ 
xk = x0
fx = f(xk), fpx = f'(xk)
para k=1,...,maxit
    si |fpx|  $\leq \varepsilon_0$  ent salir
     $\delta$  = fx/fpx
    xk = xk- $\delta$ 
    fx = f(xk), fpx = f'(xk)
    si |fx|  $\leq \varepsilon_f$  ent salir
    si  $|\delta|/(1+|x_k|) \leq \varepsilon_x$  ent salir
fin-para k

```

Para la implementación en C, es necesario determinar cómo se evalúa f y f' . Fundamentalmente hay dos posibilidades:

- Hacer una función de C para evaluar f y otra para evaluar f' .
- Hacer una función de C donde se evalúe al mismo tiempo f y f' .

Cada una de estas posibilidades puede hacerse con nombres fijos para las funciones o por medio de apuntadores a funciones. En el método de Newton, en cada iteración se calcula una vez $f(x)$ y una vez $f'(x)$. Cuando se cambia f es necesario cambiar f' . Por razones de sencillez en la escritura, la siguiente implementación en C utiliza un nombre fijo para una función donde al mismo tiempo se evalúa f y f' .

```
double newton(double x0, int maxit, double epsF,
              double epsX, double eps0, int &indic)
{
    // Metodo de Newton para resolver  f(x) = 0.
    //
    // f(x) esta definida en  fxpx.
    // Esta ultima funcion tambien calcula f'(x)
    //
    // indic  valdra:
    //      1  si  | f(xk) | <= epsF
    //      2  si  | xk-xk1 |/( 1+ |xk| ) <= epsX
    //      0  si  | f'(xk) | <= eps0
    //      3  si  en  maxit  iteraciones no se ha tenido
    //            convergencia
    //
    // Sea r el valor devuelto por newton.
    // Si indic= 1, r es una buena aproximacion de una raiz
    // Si indic= 2, r es una buena aproximacion de una raiz
    //           o el ultimo xk calculado.
    //           = 3, r es el ultimo xk calculado.
    //           = 0, r es el ultimo xk calculado.
    // x0  es una aproximacion inicial de la raiz.

    int k;
    double delta, xk, fx, fpx;

    xk = x0;
    fx = fxpx(xk, fpx);
```

```

for( k=1; k<= maxit; k++){
    if( fabs(fpx) <= eps0 ){
        indic = 0;
        return xk;
    }
    delta = fx/fpx;
    xk = xk - delta;
    fx = fxfpx(xk, fpx);
    if( fabs(fx) <= epsF ){
        indic = 1;
        return xk;
    }
    if( fabs(delta)/( 1.0+fabs(xk) ) <= epsX ){
        indic = 2;
        return xk;
    }
}
indic = 3;
return xk;
}
//-----
double fxfpx(double x, double &fpx)
{
    // Calculo de f(x) y f'(x)
    // Devuelve f(x)
    // La variable fpx tendra el valor f'(x)

    double fx;

    // f(x) = x^5 - 3x^4 + 10x - 8.

    fx = x*x*x*x*x - 3.0*x*x*x*x + 10.0*x - 8.0;
    fpx = 5.0*x*x*x*x - 12.0*x*x*x + 10.0;

    return fx;
}

```

La llamada a la función **newton** puede ser semejante a

```
x = newton(3.0, 20, 1.0E-10, 1.0E-15, 1.0E-30, resul);
if( resul == 1 ) ...
```

12.1.1 Orden de convergencia

Definición 12.1. Sea $\{x_k\}$ una sucesión de números reales con límite L . Se dice que la convergencia tiene **orden de convergencia p** , si p es el mayor valor tal que el siguiente límite existe y es finito.

$$\lim_{k \rightarrow \infty} \frac{|x_{k+1} - L|}{|x_k - L|^p} = \beta < \infty$$

En este caso se dice que β es la tasa de convergencia. Cuando el orden es 1, se dice que la convergencia es lineal.

Cuando el orden es 2, se dice que la convergencia es cuadrática. De acuerdo con la definición, lo ideal es tener órdenes altos con tasas pequeñas. Una convergencia lineal con tasa 1 es una convergencia muy lenta; por ejemplo $x_k = 1/k$. La convergencia de $x_k = 1/3^k$ es más rápida, es lineal pero con tasa $1/3$. La sucesión definida por $x_0 = 4/5$, $x_{k+1} = x_k^2$ tiene convergencia cuadrática.

Cuando se tiene una sucesión $\{x^k\}$ en \mathbb{R}^n , para la definición del orden de convergencia se usa

$$\lim_{k \rightarrow \infty} \frac{\|x_{k+1} - L\|}{\|x_k - L\|^p}.$$

Teorema 12.1. Sean $a < b$, $I =]a, b[$, $f : I \rightarrow \mathbb{R}$, $x^* \in I$, $f(x^*) = 0$, f' y f'' existen y son continuas en I , $f'(x^*) \neq 0$, $\{x_k\}$ la sucesión definida por 12.1. Si x_0 está suficientemente cerca de x^* , entonces

$$\lim_{k \rightarrow \infty} x_k = x^*, \tag{12.2}$$

$$\lim_{k \rightarrow \infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^2} = \frac{|f''(x^*)|}{2|f'(x^*)|} \tag{12.3}$$

La demostración de este teorema está en varios libros; por ejemplo, en [Atk78] o en [Sch91]. El primer resultado dice que la sucesión converge a x^* . El segundo dice que la convergencia es cuadrática o de orden superior. La frase “ x_0 está suficientemente cerca de x^* , entonces...” quiere decir que existe $\varepsilon > 0$ tal que si $x_0 \in [x^* - \varepsilon, x^* + \varepsilon] \subseteq I$, entonces...

A manera de comprobación, después de que se calculó una raíz, se puede ver si la sucesión muestra aproximadamente convergencia cuadrática. Sea $e_k = x_k - x^*$. La sucesión $|e_k|/|e_{k-1}|^2$ debería acercarse a $|f''(x^*)|/(2|f'(x^*)|)$. Para el ejemplo anterior $|f''(x^*)/(2|f'(x^*)|) = 16/(2 \times 3) = 2.6666$.

k	x_k	$ e_k $	$ e_k / e_{k-1} ^2$
0	2.1000000000000001	1.100000e+00	
1	0.9427881279712185	5.721187e-02	4.728254e-02
2	0.9934841559110774	6.515844e-03	1.990666e+00
3	0.9998909365826297	1.090634e-04	2.568844e+00
4	0.9999999683006239	3.169938e-08	2.664971e+00
5	0.999999999999973	2.664535e-15	2.651673e+00

12.2 Método de la secante

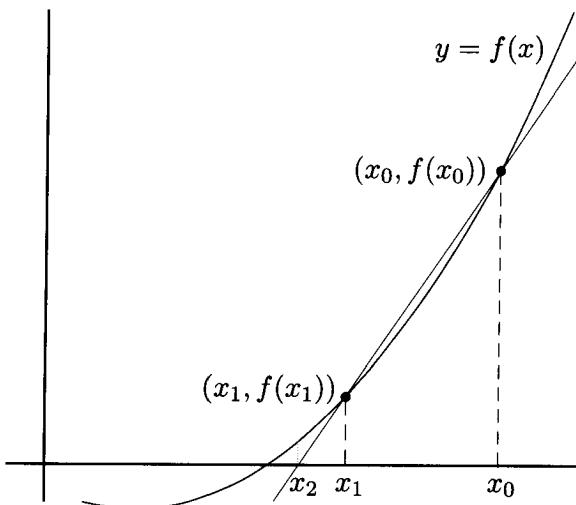
Uno de los inconvenientes del método de Newton es que necesita evaluar $f'(x)$ en cada iteración. Algunas veces esto es imposible o muy difícil. Si en el método de Newton se modifica la fórmula 12.1 reemplazando $f'(x_k)$ por una aproximación

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}},$$

entonces se obtiene

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \quad (12.4)$$

En el método de Newton se utilizaba la recta tangente a la curva en el punto $(x_k, f(x_k))$. En el método de la secante se utiliza la recta (secante) que pasa por los puntos $(x_k, f(x_k))$ y $(x_{k-1}, f(x_{k-1}))$.

Figura 12.2 *Método de la secante.*

Ejemplo 12.2. Aplicar el método de la secante a la ecuación $x^5 - 3x^4 + 10x - 8 = 0$, partiendo de $x_0 = 3$.

k	x_k	$f(x_k)$
0	3.000000	2.200000e+01
1	3.010000	2.292085e+01
2	2.761091	5.725624e+00
3	2.678210	2.226281e+00
4	2.625482	4.593602e-01
5	2.611773	5.317368e-02
6	2.609979	1.552812e-03
7	2.609925	5.512240e-06
8	2.609924	5.747927e-10
9	2.609924	-2.838008e-15

Mediante condiciones semejantes a las exigidas en el teorema 12.1 se muestra (ver [Sch91]), que el método de la secante tiene orden de convergencia

$$\frac{1 + \sqrt{5}}{2} \approx 1.618$$

Como el método de la secante es semejante al método de Newton, entonces tienen aproximadamente las mismas ventajas y las mismas desventajas, salvo dos aspectos:

- La convergencia del método de la secante, en la mayoría de los casos, es menos rápida que en el método de Newton.
- El método de la secante obvia la necesidad de evaluar las derivadas.

El esquema del algoritmo es semejante al del método de Newton. Hay varias posibles salidas, algunas deseables, otras no.

```
datos: x0, maxit,  $\varepsilon_f$ ,  $\varepsilon_x$ ,  $\varepsilon_0$ 
x1 = x0 + 0.1, f0 = f(x0), f1 = f(x1)
para k=1,...,maxit
    den = f1-f0
    si |den|  $\leq \varepsilon_0$  ent salir
     $\delta = f1 * (x1 - x0) / den$ 
    x2 = x1 -  $\delta$ , f2 = f(x2)
    si |f2|  $\leq \varepsilon_f$  ent salir
    si | $\delta$ |/(1+|x2|)  $\leq \varepsilon_x$  ent salir
    x0 = x1, f0 = f1, x1 = x2, f1 = f2
fin-para k
```

Para la implementación en C, la evaluación de $f(x)$ puede hacerse en una función con nombre fijo o por medio de un apuntador a función. En la siguiente implementación la evaluación de $f(x)$ se hace en la función con prototipo `double f(double x);`.

```
double secante(double x0, int maxit, double epsF,
               double epsX, double eps0, int &indic)
{
    // Metodo de la secante para resolver f(x) = 0.
    //
    // f(x) esta definida en la funcion f
    //
    // indic valdra:
    //      1 si |f(x2)| <= epsF
    //      2 si |x2-x1|/(1+|x2|) <= epsX
    //      0 si |f(x1) - f(x0)| <= eps0
    //      3 si en maxit iteraciones no se ha tenido
    //          convergencia
    //
```

```

// Sea r el valor devuelto por secante.
// Si indic= 1, r es una buena aproximacion de una raiz
// Si indic= 2, r es una buena aproximacion de una raiz
//           o el ultimo x2 calculado.
//           = 3, r es el ultimo x2 calculado.
//           = 0, r es el ultimo x2 calculado.
// x0 es una aproximacion inicial de la raiz.

int k;
double delta, x1, x2, f0, f1, f2, den;

x1 = x0 + 0.01;
f0 = f(x0);
f1 = f(x1);
for( k=1; k<= maxit; k++){
    den = f1 - f0;
    if( fabs(den) <= eps0 ){
        indic = 0;
        return x1;
    }
    delta = f1*(x1-x0)/den;
    x2 = x1 - delta;
    f2 = f(x2);
    if( fabs(f2) <= epsF ){
        indic = 1;
        return x2;
    }
    if( fabs(delta)/( 1.0+fabs(x2) ) <= epsX ){
        indic = 2;
        return x2;
    }
    x0 = x1;  x1 = x2;  f0 = f1;  f1 = f2;
}
indic = 3;
return x2;
}
//-----
double f(double x)

```

```

{
    // Calculo de f(x)

    // f(x) = x^5 - 3x^4 + 10x - 8.

    return x*x*x*x*x - 3.0*x*x*x*x*x + 10.0*x - 8.0;
}

```

12.3 Método de la bisección

Si la función f es continua en el intervalo $[a, b]$, $a < b$, y si $f(a)$ y $f(b)$ tienen signo diferente,

$$f(a)f(b) < 0,$$

entonces f tiene por lo menos una raíz en el intervalo. Este método ya se vio en el capítulo sobre funciones.

Usualmente se define el error asociado a una aproximación como

$$e_k = |x_k - x^*|.$$

En el método de la bisección, dado el intervalo $[a_k, b_k]$, $a_k < b_k$, no se tiene un valor de x_k . Se sabe que en $[a_k, b_k]$ hay por lo menos una raíz. Cualquiera de los valores en el intervalo podría ser x_k . Sea E_k el máximo error que puede haber en la iteración k ,

$$e_k \leq E_k = b_k - a_k.$$

Como el tamaño de un intervalo es exactamente la mitad del anterior

$$b_k - a_k = \frac{1}{2}(b_{k-1} - a_{k-1}),$$

entonces

$$b_k - a_k = \left(\frac{1}{2}\right) \left(\frac{1}{2}\right) (b_{k-2} - a_{k-2}).$$

Finalmente

$$b_k - a_k = \left(\frac{1}{2}\right)^k (b_0 - a_0).$$

Obviamente $E_k \rightarrow 0$ y

$$\frac{E_k}{E_{k-1}} = \frac{1}{2} \rightarrow \frac{1}{2}.$$

Esto quiere decir que la sucesión de cotas del error tiene convergencia lineal (orden 1) y tasa de convergencia 1/2.

En el método de la bisección se puede saber por anticipado el número de iteraciones necesarias para obtener un tamaño deseado,

$$\begin{aligned} b_k - a_k &\leq \varepsilon, \\ \left(\frac{1}{2}\right)^k (b_0 - a_0) &\leq \varepsilon, \\ \left(\frac{1}{2}\right)^k &\leq \frac{\varepsilon}{b_0 - a_0}, \\ 2^k &\geq \frac{b_0 - a_0}{\varepsilon}, \\ k \log 2 &\geq \log \frac{b_0 - a_0}{\varepsilon}, \\ k &\geq \frac{\log \frac{b_0 - a_0}{\varepsilon}}{\log 2}. \end{aligned}$$

Por ejemplo, si el tamaño del intervalo inicial es 3, si $\varepsilon = 1.0E - 6$, entonces en $k = 22$ (≥ 21.52) iteraciones se obtiene un intervalo suficientemente pequeño.

12.4 Método de Regula Falsi

Igualmente se conoce con el nombre de falsa posición. Es una modificación del método de la bisección. También empieza con un intervalo $[a_0, b_0]$ donde f es continua y tal que $f(a_0)$ y $f(b_0)$ tienen signo diferente.

En el método de bisección, en cada iteración, únicamente se tiene en cuenta el signo de $f(a_k)$ y de $f(b_k)$, pero no sus valores: no se está utilizando toda la información disponible. Además es de esperar que si $f(a_k)$ está más cerca de 0 que $f(b_k)$, entonces puede ser interesante considerar, no el punto medio, sino un punto más cercano a a_k . De manera análoga, si $f(b_k)$ está más cerca de 0 que $f(a_k)$, entonces puede ser interesante considerar, no el punto medio, sino un punto más cercano a b_k .

En el método de Regula Falsi se considera el punto donde la recta que pasa por $(a_k, f(a_k))$, $(b_k, f(b_k))$ corta el eje x . Como $f(a_k)$ y $f(b_k)$ tienen signo diferente, entonces el punto de corte c_k queda entre a_k y b_k .

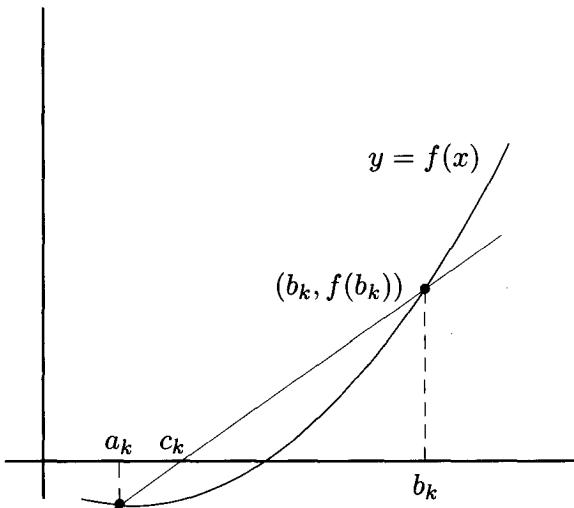


Figura 12.3 *Regula Falsi.*

La ecuación de la recta es:

$$y - f(a_k) = \frac{f(b_k) - f(a_k)}{b_k - a_k}(x - a_k)$$

Cuando $y = 0$ se tiene el punto de corte $x = c_k$,

$$c_k = a_k - \frac{f(a_k)(b_k - a_k)}{f(b_k) - f(a_k)} \quad (12.5)$$

Esta fórmula es semejante a la de la secante. Como $f(a_k)$ y $f(b_k)$ tienen signo diferente, entonces $f(b_k) - f(a_k)$ tiene signo contrario al de $f(a_k)$. Entonces $-f(a_k)/(f(b_k) - f(a_k)) > 0$. Usando de nuevo que $f(a_k)$ y $f(b_k)$ tienen signo diferente, entonces $|f(a_k)|/|f(b_k) - f(a_k)| < 1$. Luego $0 < -f(a_k)/(f(b_k) - f(a_k)) < 1$. Esto muestra que $a_k < c_k < b_k$.

Partiendo de un intervalo inicial $[a_0, b_0]$, en la iteración k se tiene el intervalo $[a_k, b_k]$ donde f es continua y $f(a_k), f(b_k)$ tienen diferente signo. Se calcula c_k el punto de corte y se tienen tres posibilidades excluyentes:

- $f(c_k) \approx 0$; en este caso c_k es, aproximadamente, una raíz;
- $f(a_k)f(c_k) < 0$; en este caso hay una raíz en el intervalo $[a_k, c_k] = [a_{k+1}, b_{k+1}]$;
- $f(a_k)f(c_k) > 0$; en este caso hay una raíz en el intervalo $[c_k, b_k] = [a_{k+1}, b_{k+1}]$.

Ejemplo 12.3. Aplicar el método de Regula Falsi a la ecuación $x^5 - 3x^4 + 10x - 8 = 0$, partiendo de $[2, 5]$.

k	a_k	b_k	$f(a_k)$	$f(b_k)$	c_k	$f(c_k)$
0	2.000000	5	-4.000000	1292	2.009259	-4.054857
1	2.009259	5	-4.054857	1292	2.018616	-4.108820
2	2.018616	5	-4.108820	1292	2.028067	-4.161744
3	2.028067	5	-4.161744	1292	2.037610	-4.213478
4	2.037610	5	-4.213478	1292	2.047239	-4.263862
5	2.047239	5	-4.263862	1292	2.056952	-4.312734
10	2.096539	5	-4.489666	1292	2.106594	-4.528370
20	2.198548	5	-4.739498	1292	2.208787	-4.744664
30	2.298673	5	-4.594020	1292	2.308244	-4.554769
335	2.609924	5	-0.000001	1292	2.609924	-0.000001

Como se ve, la convergencia es muy lenta. El problema radica en que en el método de Regula Falsi **no se puede garantizar** que

$$\lim_{k \rightarrow \infty} (b_k - a_k) = 0.$$

Esto quiere decir que el método no es seguro. Entonces, en una implementación, es necesario trabajar con un número máximo de iteraciones.

12.5 Modificación del método de Regula Falsi

Los dos métodos, bisección y Regula Falsi, se pueden combinar en uno solo de la siguiente manera. En cada iteración se calcula m_k y c_k . Esto define tres subintervalos en $[a_k, b_k]$. En por lo menos uno de ellos se tiene una raíz. Si los tres subintervalos sirven, se puede escoger cualquiera, o mejor aún, el de menor tamaño. En un caso muy especial, cuando m_k y c_k coinciden, se tiene simplemente una iteración del método de bisección.

En cualquiera de los casos

$$b_{k+1} - a_{k+1} \leq \frac{1}{2}(b_k - a_k),$$

entonces

$$b_k - a_k \leq \left(\frac{1}{2}\right)^k (b_0 - a_0),$$

lo que garantiza que

$$\lim_{k \rightarrow \infty} (b_k - a_k) = 0.$$

Ejemplo 12.4. Aplicar la modificación del método de Regula Falsi a la ecuación $x^5 - 3x^4 + 10x - 8 = 0$, partiendo de $[2, 5]$.

k	a	b	f(a)	f(b)	c	f(c)	m	f(m)
0	2.0000	5.0000	-4.00e+0	1.29e+3	2.0093	-4.0e+0	3.5000	1.0e+2
1	2.0093	3.5000	-4.05e+0	1.02e+2	2.0662	-4.4e+0	2.7546	5.4e+0
2	2.0662	2.7546	-4.36e+0	5.42e+0	2.3731	-4.2e+0	2.4104	-3.8e+0
3	2.4104	2.7546	-3.80e+0	5.42e+0	2.5523	-1.5e+0	2.5825	-7.4e-1
4	2.5825	2.7546	-7.44e-1	5.42e+0	2.6033	-1.9e-1	2.6686	1.9e+0
5	2.6033	2.6686	-1.87e-1	1.88e+0	2.6092	-2.0e-2	2.6360	7.8e-1
6	2.6092	2.6360	-2.00e-2	7.84e-1	2.6099	-9.7e-4	2.6226	3.7e-1
7	2.6099	2.6226	-9.73e-4	3.72e-1	2.6099	-2.3e-5	2.6162	1.8e-1
8	2.6099	2.6162	-2.33e-5	1.83e-1	2.6099	-2.8e-7	2.6131	9.1e-2
9	2.6099	2.6131	-2.81e-7	9.10e-2	2.6099	-1.7e-9		

La modificación es mucho mejor que el método de Regula Falsi. Además, el número de iteraciones de la modificación debe ser menor o igual que el número de iteraciones del método de bisección. Pero para comparar equitativamente el método de bisección y la modificación de Regula Falsi, es necesario tener en cuenta el número de evaluaciones de $f(x)$.

En el método de bisección, en k iteraciones, el número de evaluaciones de f está dado por:

$$n_{\text{bisec}} = 2 + k_{\text{bisec}}.$$

En la modificación de Regula Falsi,

$$n_{\text{modif}} = 2 + 2 k_{\text{modif}}.$$

12.6 Método de punto fijo

Los métodos vistos se aplican a la solución de la ecuación $f(x) = 0$. El método de punto fijo sirve para resolver la ecuación

$$g(x) = x. \quad (12.6)$$

Se busca un x^* tal que su imagen, por medio de la función g , sea el mismo x^* . Por tal motivo se dice que x^* es un punto fijo de la función g .

La aplicación del método es muy sencilla. A partir de un x_0 dado, se aplica varias veces la fórmula

$$x_{k+1} = g(x_k). \quad (12.7)$$

Se espera que la sucesión $\{x_k\}$ construida mediante (12.7) converja hacia x^* . En algunos casos el método, además de ser muy sencillo, es muy eficiente; en otros casos la eficiencia es muy pequeña; finalmente, en otros casos el método definitivamente no sirve.

Ejemplo 12.5. Resolver $x^3 + x^2 + 6x + 5 = 0$. Esta ecuación se puede escribir en la forma

$$x = -\frac{x^3 + x^2 + 5}{6}.$$

Aplicando el método de punto fijo a partir de $x_0 = -1$ se tiene:

$$\begin{aligned} x_0 &= -1 \\ x_1 &= -0.833333 \\ x_2 &= -0.852623 \\ x_3 &= -0.851190 \\ x_4 &= -0.851303 \\ x_5 &= -0.851294 \\ x_6 &= -0.851295 \\ x_7 &= -0.851295 \\ x_8 &= -0.851295 \end{aligned}$$

Entonces se tiene una aproximación de una raíz, $x^* \approx -0.851295$. En este caso el método funcionó muy bien. Utilicemos ahora otra expresión para $x = g(x)$:

$$x = -\frac{x^3 + 6x + 5}{x}.$$

Aplicando el método de punto fijo a partir de $x_0 = -0.851$, muy buena aproximación de la raíz, se tiene:

$$\begin{array}{ll} x_0 & = -0.8510 \\ x_1 & = -0.8488 \\ x_2 & = -0.8294 \\ x_3 & = -0.6599 \\ x_4 & = 1.1415 \\ x_5 & = -11.6832 \\ x_6 & = -142.0691 \\ x_7 & = -2.0190e+4 \end{array}$$

En este caso se observa que, aun partiendo de una muy buena aproximación de la solución, no hay convergencia. ◇

Antes de ver un resultado sobre convergencia del método de punto fijo, observemos su interpretación gráfica. La solución de $g(x) = x$ está determinada por el punto de corte, si lo hay, entre las gráficas $y = g(x)$ y $y = x$.

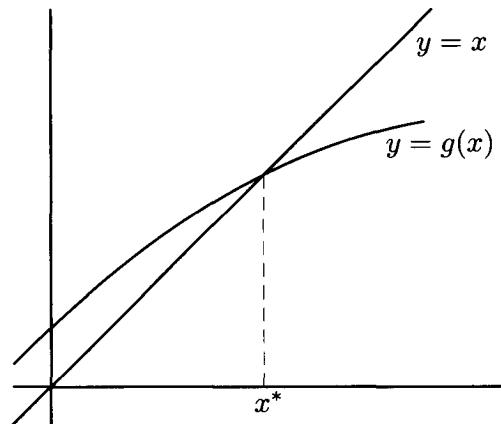


Figura 12.4 Punto fijo.

Después de dibujar las dos funciones, la construcción de los puntos $x_1, x_2, x_3\dots$ se hace de la siguiente manera. Despues de situar el valor x_0 sobre el eje x , para obtener el valor x_1 , se busca verticalmente la

curva $y = g(x)$. El punto obtenido tiene coordenadas $(x_0, g(x_0))$, o sea, (x_0, x_1) . Para obtener $x_2 = g(x_1)$ es necesario inicialmente resituar x_1 sobre el eje x , para lo cual basta con buscar horizontalmente la recta $y = x$ para obtener el punto (x_1, x_1) . A partir de este punto se puede obtener x_2 buscando verticalmente la curva $y = g(x)$. Se tiene el punto $(x_1, g(x_1))$, o sea, (x_1, x_2) . Con desplazamiento horizontal se obtiene (x_2, x_2) . En resumen, se repite varias veces el siguiente procedimiento: *a partir de (x_k, x_k) buscar verticalmente en la curva $y = g(x)$ el punto (x_k, x_{k+1}) , y a partir del punto obtenido buscar horizontalmente en la recta $y = x$ el punto (x_{k+1}, x_{k+1}) .* Si el proceso converge, los puntos obtenidos tienden hacia el punto $(x^*, g(x^*)) = (x^*, x^*)$.

Las figuras 12.5.a, 12.5.b, 12.5.c y 12.5.d muestran gráficamente la utilización del método; en los dos primeros casos hay convergencia; en los otros dos no hay, aun si la aproximación inicial es bastante buena.

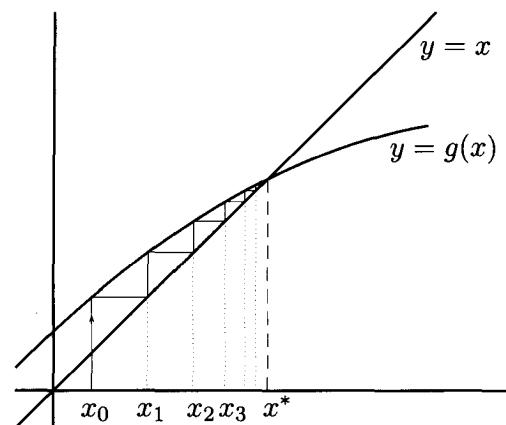


Figura 12.5.a *Método de punto fijo.*

12.6. MÉTODO DE PUNTO FIJO

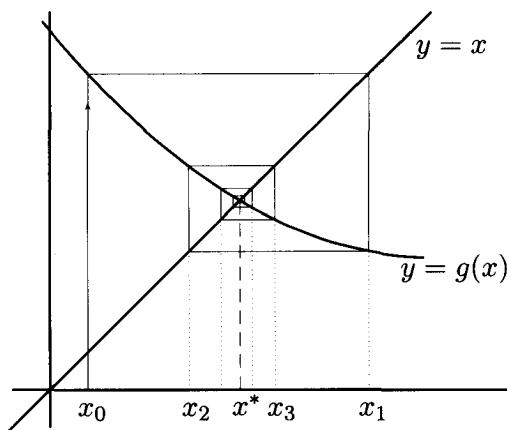


Figura 12.5.b *Método de punto fijo.*

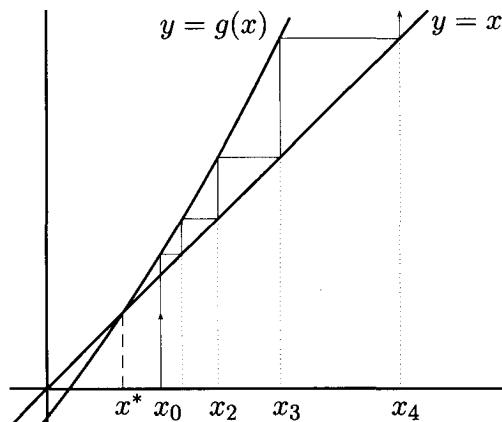


Figura 12.5.c *Método de punto fijo.*

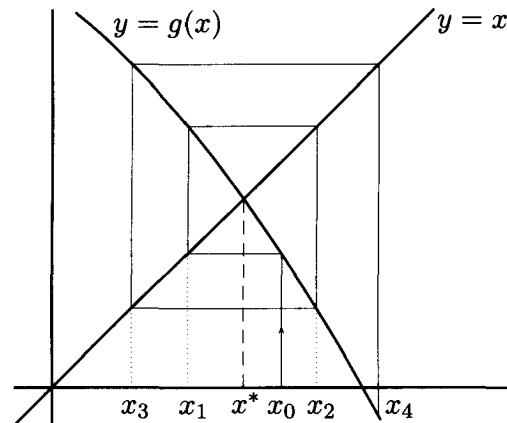


Figura 12.5.d Método de punto fijo.

En seguida se presentan dos teoremas (demostración en [Atk78]) sobre la convergencia del método de punto fijo; el primero es más general y más preciso, el segundo es una simplificación del primero, de más fácil aplicación para ciertos problemas.

Teorema 12.2. *Sea g continuamente diferenciable en el intervalo $[a, b]$, tal que*

$$\begin{aligned} g([a, b]) &\subseteq [a, b], \\ |g'(x)| &< 1, \text{ para todo } x \in [a, b]. \end{aligned}$$

Entonces existe un único x^ en $[a, b]$ solución de $x = g(x)$ y la iteración de punto fijo (12.7) converge a x^* para todo $x_0 \in [a, b]$.*

Teorema 12.3. *Sea x^* solución de $x = g(x)$, g continuamente diferenciable en un intervalo abierto I tal que $x^* \in I$, $|g'(x^*)| < 1$. Entonces la iteración de punto fijo (12.7) converge a x^* para todo x_0 suficientemente cerca de x^* .*

El caso ideal (la convergencia es más rápida) se tiene cuando $g'(x^*) \approx 0$.

En los dos ejemplos numéricos anteriores, para resolver $x^3 + x^2 + 6x + 5 = 0$, se tiene: $x = g(x) = -(x^3 + x^2 + 5)/6$, $g'(-0.8513) = -0.0786$. Si se considera $x = g(x) - (x^3 + 6x + 5)/x$, $g'(-0.8513) = 8.6020$. Estos resultados numéricos concuerdan con el último teorema.

Dos de los ejemplos gráficos anteriores muestran justamente que cuando $|g'(x^*)| < 1$ el método converge.

Ejemplo 12.6. Resolver $x^2 = 3$, o sea, calcular $\sqrt{3}$.

$$\begin{aligned} x^2 &= 3, \\ x^2 + x^2 &= x^2 + 3, \\ x &= \frac{x^2 + 3}{2x}, \\ x &= \frac{x + 3/x}{2}. \end{aligned}$$

$$\begin{aligned} x_0 &= 3 \\ x_1 &= 2 \\ x_2 &= 1.750000000000000 \\ x_3 &= 1.73214285714286 \\ x_4 &= 1.73205081001473 \\ x_5 &= 1.73205080756888 \\ x_6 &= 1.73205080756888 \end{aligned}$$

Se observa que la convergencia es bastante rápida. Este método es muy utilizado para calcular raíces cuadradas en calculadoras de bolsillo y computadores.

Aplicando el teorema 12.3 y teniendo en cuenta que $g'(x^*) = g'(\sqrt{3}) = 1/2 - 1.5/x^{*2} = 0$, se concluye rápidamente que si x^0 está suficientemente cerca de $\sqrt{3}$, entonces el método converge.

La aplicación del teorema 12.2 no es tan inmediata, pero se obtiene información más detallada. La solución está en el intervalo $[2, 3]$; consideremos un intervalo aún más grande: $I = [1 + \varepsilon, 4]$ con $0 < \varepsilon < 1$.

$$\begin{aligned} g(1) &= 2, \\ g(4) &= 2.375, \\ g'(x) &= \frac{1}{2} - \frac{3}{2x^2}, \\ g'(\sqrt{3}) &= 0, \\ g'(1) &= -1, \\ g'(4) &= \frac{13}{32}, \\ g''(x) &= \frac{3}{x^3}. \end{aligned}$$

Entonces $g''(x) > 0$ para todo x positivo. Luego $g'(x)$ es creciente para $x > 0$. Como $g'(1) = -1$, entonces $-1 < g'(1+\varepsilon)$. De nuevo por ser $g'(x)$ creciente, entonces $-1 < g'(x) \leq 13/32$ para todo $x \in I$. En resumen, $|g'(x)| < 1$ cuando $x \in I$.

Entre $1 + \varepsilon$ y $\sqrt{3}$ el valor de $g'(x)$ es negativo. Entre $\sqrt{3}$ y 4 el valor de $g'(x)$ es positivo. Luego g decrece en $[1 + \varepsilon, \sqrt{3}]$ y crece en $[\sqrt{3}, 4]$. Entonces $g([1 + \varepsilon, \sqrt{3}]) = [g(1 + \varepsilon), \sqrt{3}] \subseteq [2, \sqrt{3}]$ y $g([\sqrt{3}, 4]) = [\sqrt{3}, 2.375]$. En consecuencia $g(I) = [\sqrt{3}, 2.375] \subseteq I$. Entonces el método de punto fijo converge a $x^* = \sqrt{3}$ para cualquier $x_0 \in [1, 4]$. Este resultado se puede generalizar al intervalo $[1 + \varepsilon, b]$ con $b > \sqrt{3}$.

Si se empieza con $x_0 = 1/2$, no se cumplen las condiciones del teorema; sin embargo, el método converge. \diamond

12.6.1 Método de punto fijo y método de Newton

Supongamos que $c \neq 0$ es una constante y que x^* es solución de la ecuación $f(x) = 0$. Ésta se puede reescribir

$$\begin{aligned} 0 &= cf(x), \\ x &= x + cf(x) = g(x). \end{aligned} \tag{12.8}$$

Si se desea resolver esta ecuación por el método de punto fijo, la convergencia es más rápida cuando $g'(x^*) = 0$, o sea,

$$\begin{aligned} 1 + cf'(x^*) &= 0, \\ c &= -\frac{1}{f'(x^*)}. \end{aligned}$$

Entonces al aplicar el método de punto fijo a (12.8), se tiene la fórmula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x^*)}. \tag{12.9}$$

Para aplicar esta fórmula se necesitaría conocer $f'(x^*)$ e implícitamente el valor de x^* , que es precisamente lo que se busca. La fórmula del método de Newton, (12.1), puede ser vista simplemente como la utilización de (12.9) reemplazando $f'(x^*)$ por la mejor aproximación conocida en ese momento: $f'(x_k)$.

12.7 Método de Newton en \mathbb{R}^n

Consideremos ahora un sistema de n ecuaciones con n incógnitas; por ejemplo,

$$\begin{aligned} x_1^2 + x_1x_2 + x_3 - 3 &= 0 \\ 2x_1 + 3x_2x_3 - 5 &= 0 \\ (x_1 + x_2 + x_3)^2 - 10x_3 + 1 &= 0. \end{aligned} \tag{12.10}$$

Este sistema no se puede escribir en la forma matricial $Ax = b$; entonces no se puede resolver por los métodos usuales para sistemas de ecuaciones lineales. Lo que se hace, como en el método de Newton en \mathbb{R} , es utilizar aproximaciones de primer orden (llamadas también aproximaciones lineales). Esto es simplemente la generalización de la aproximación por una recta.

Un sistema de n ecuaciones con n incógnitas se puede escribir de la forma

$$\begin{aligned} F_1(x_1, x_2, \dots, x_n) &= 0 \\ F_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ F_n(x_1, x_2, \dots, x_n) &= 0, \end{aligned}$$

donde cada F_i es una función de n variables con valor real, o sea, $F_i : \mathbb{R}^n \rightarrow \mathbb{R}$. Denotemos $x = (x_1, x_2, \dots, x_n)$ y

$$F(x) = \begin{bmatrix} F_1(x) \\ F_2(x) \\ \vdots \\ F_n(x) \end{bmatrix}.$$

Así F es una función de variable vectorial y valor vectorial, $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$, y el problema se escribe de manera muy compacta:

$$F(x) = 0. \tag{12.11}$$

Este libro está dirigido principalmente a estudiantes de segundo semestre, quienes todavía no conocen el cálculo en varias variables, entonces no habrá una deducción (ni formal ni intuitiva) del método, simplemente se verá como una generalización del método en \mathbb{R} .

12.7.1 Matriz jacobiana

La matriz jacobiana de la función $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$, denotada por $J_F(\bar{x})$ o por $F'(\bar{x})$, es una matriz de tamaño $n \times n$, en la que en la i -ésima fila están las n derivadas parciales de F_i ,

$$J_F(x) = F'(x) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1}(x) & \frac{\partial F_1}{\partial x_2}(x) & \cdots & \frac{\partial F_1}{\partial x_n}(x) \\ \frac{\partial F_2}{\partial x_1}(x) & \frac{\partial F_2}{\partial x_2}(x) & \cdots & \frac{\partial F_2}{\partial x_n}(x) \\ \vdots & & \ddots & \\ \frac{\partial F_n}{\partial x_1}(x) & \frac{\partial F_n}{\partial x_2}(x) & \cdots & \frac{\partial F_n}{\partial x_n}(x) \end{bmatrix}$$

Para las ecuaciones (12.10), escritas en la forma $F(x) = 0$,

$$F'(x) = \begin{bmatrix} 2x_1 + x_2 & x_1 & 1 \\ 2 & 3x_3 & 3x_2 \\ 2(x_1 + x_2 + x_3) & 2(x_1 + x_2 + x_3) & 2(x_1 + x_2 + x_3) - 10 \end{bmatrix}$$

$$F'(2, -3, 4) = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 12 & -9 \\ 6 & 6 & -4 \end{bmatrix}.$$

12.7.2 Fórmula de Newton en \mathbb{R}^n

La fórmula del método de Newton en \mathbb{R} ,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

se puede reescribir con superíndices en lugar de subíndices:

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}.$$

De nuevo, es simplemente otra forma de escribir

$$x^{k+1} = x^k - f'(x^k)^{-1}f(x^k).$$

Esta expresión sí se puede generalizar

$$x^{k+1} = x^k - F'(x^k)^{-1}F(x^k). \quad (12.12)$$

Su interpretación, muy natural, aparece a continuación. Sea x^* , un vector de n componentes, solución del sistema (12.11). Dependiendo de la conveniencia se podrá escribir

$$x^* = (x_1^*, x_2^*, \dots, x_n^*) \quad \text{o} \quad x^* = \begin{bmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_n^* \end{bmatrix}.$$

El método empieza con un vector $x^0 = (x_1^0, x_2^0, \dots, x_n^0)$, aproximación inicial de la solución x^* . Mediante (12.12) se construye una sucesión de vectores $\{x^k = (x_1^k, x_2^k, \dots, x_n^k)\}$ con el deseo de que $x^k \rightarrow x^*$. En palabras, el vector x^{k+1} es igual al vector x^k menos el producto de la inversa de la matriz jacobiana $F'(x^k)$ y el vector $F(x^k)$. Para evitar el cálculo de una inversa, la fórmula se puede reescribir

$$\begin{aligned} d^k &= -F'(x^k)^{-1}F(x^k) \\ x^{k+1} &= x^k + d^k. \end{aligned}$$

Premultiplicando por $F'(x^k)$

$$\begin{aligned} F'(x^k)d^k &= -F'(x^k)F'(x^k)^{-1}F(x^k), \\ F'(x^k)d^k &= -F(x^k). \end{aligned}$$

En esta última expresión se conoce (o se puede calcular) la matriz $F'(x^k)$. También se conoce el vector $F(x^k)$. O sea, simplemente se tiene un sistema de ecuaciones lineales. La solución de este sistema es el vector d^k . Entonces las fórmulas para el método de Newton son:

$$\begin{aligned} \text{resolver } F'(x^k)d^k &= -F(x^k), \\ x^{k+1} &= x^k + d^k. \end{aligned} \quad (12.13)$$

Ejemplo 12.7. Resolver el sistema

$$\begin{aligned}x_1^2 + x_1x_2 + x_3 - 3 &= 0 \\2x_1 + 3x_2x_3 - 5 &= 0 \\(x_1 + x_2 + x_3)^2 - 10x_3 + 1 &= 0\end{aligned}$$

a partir de $x^0 = (2, -3, 4)$.

$$F(x^0) = \begin{bmatrix} -1 \\ -37 \\ -30 \end{bmatrix}, \quad F'(x^0) = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 12 & -9 \\ 6 & 6 & -4 \end{bmatrix}$$

$$\text{resolver } \begin{bmatrix} 1 & 2 & 1 \\ 2 & 12 & -9 \\ 6 & 6 & -4 \end{bmatrix} \begin{bmatrix} d_1^0 \\ d_2^0 \\ d_3^0 \end{bmatrix} = - \begin{bmatrix} -1 \\ -37 \\ -30 \end{bmatrix}, \quad d^0 = \begin{bmatrix} 2.5753 \\ 0.5890 \\ -2.7534 \end{bmatrix}$$

$$x^1 = \begin{bmatrix} 2 \\ -3 \\ 4 \end{bmatrix} + \begin{bmatrix} 2.5753 \\ 0.5890 \\ -2.7534 \end{bmatrix} = \begin{bmatrix} 4.5753 \\ -2.4110 \\ 1.2466 \end{bmatrix}$$

$$F(x^1) = \begin{bmatrix} 8.1494 \\ -4.8656 \\ 0.1689 \end{bmatrix}, \quad F'(x^1) = \begin{bmatrix} 6.7397 & 4.5753 & 1.0000 \\ 2.0000 & 3.7397 & -7.2329 \\ 6.8219 & 6.8219 & -3.1781 \end{bmatrix}$$

$$\begin{bmatrix} 6.7397 & 4.5753 & 1.0000 \\ 2.0000 & 3.7397 & -7.2329 \\ 6.8219 & 6.8219 & -3.1781 \end{bmatrix} \begin{bmatrix} d_1^1 \\ d_2^1 \\ d_3^1 \end{bmatrix} = - \begin{bmatrix} 8.1494 \\ -4.8656 \\ 0.1689 \end{bmatrix}, \quad d^1 = \begin{bmatrix} -4.4433 \\ 4.6537 \\ 0.5048 \end{bmatrix}$$

$$x^2 = \begin{bmatrix} 4.5753 \\ -2.4110 \\ 1.2466 \end{bmatrix} + \begin{bmatrix} -4.4433 \\ 4.6537 \\ 0.5048 \end{bmatrix} = \begin{bmatrix} 0.1321 \\ 2.2428 \\ 1.7514 \end{bmatrix}$$

A continuación se presentan los resultados de $F(x^k)$, $F'(x^k)$, d^k , x^{k+1} .
 $k = 2$

$$\begin{bmatrix} -0.9350 \\ 7.0481 \\ 0.5116 \end{bmatrix}, \quad \begin{bmatrix} 2.5069 & 0.1321 & 1.0000 \\ 2.0000 & 5.2542 & 6.7283 \\ 8.2524 & 8.2524 & -1.7476 \end{bmatrix}, \quad \begin{bmatrix} 0.6513 \\ -0.8376 \\ -0.5870 \end{bmatrix}, \quad \begin{bmatrix} 0.7833 \\ 1.4052 \\ 1.1644 \end{bmatrix}$$

$k = 3$

$$\begin{bmatrix} -0.1213 \\ 1.4751 \\ 0.5981 \end{bmatrix}, \quad \begin{bmatrix} 2.9718 & 0.7833 & 1.0000 \\ 2.0000 & 3.4931 & 4.2156 \\ 6.7057 & 6.7057 & -3.2943 \end{bmatrix}, \quad \begin{bmatrix} 0.1824 \\ -0.3454 \\ -0.1502 \end{bmatrix}, \quad \begin{bmatrix} 0.9658 \\ 1.0598 \\ 1.0141 \end{bmatrix}$$

$$k = 4$$

$$\begin{bmatrix} -0.0297 \\ 0.1557 \\ 0.0981 \end{bmatrix}, \begin{bmatrix} 2.9913 & 0.9658 & 1.0000 \\ 2.0000 & 3.0424 & 3.1793 \\ 6.0793 & 6.0793 & -3.9207 \end{bmatrix}, \begin{bmatrix} 0.0335 \\ -0.0587 \\ -0.0139 \end{bmatrix}, \begin{bmatrix} 0.9993 \\ 1.0011 \\ 1.0002 \end{bmatrix}$$

$$k = 5$$

$$\begin{bmatrix} -0.0008 \\ 0.0025 \\ 0.0015 \end{bmatrix}, \begin{bmatrix} 2.9997 & 0.9993 & 1.0000 \\ 2.0000 & 3.0006 & 3.0033 \\ 6.0012 & 6.0012 & -3.9988 \end{bmatrix}, \begin{bmatrix} 0.0007 \\ -0.0011 \\ -0.0002 \end{bmatrix}, \begin{bmatrix} 1.0000 \\ 1.0000 \\ 1.0000 \end{bmatrix}$$

$$F(x^6) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \text{ luego } x^* \approx \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \diamond$$

Ejercicios

Trate de resolver las ecuaciones propuestas, utilice métodos diferentes, compare sus ventajas y desventajas. Emplee varios puntos iniciales. Busque, si es posible, otras raíces.

12.1 $x^3 + 2x^2 + 3x + 4 = 0.$

12.2 $x^3 + 2x^2 - 3x - 4 = 0.$

12.3 $x^4 - 4x^3 + 6x^2 - 4x + 1 = 0.$

12.4 $x^4 - 4x^3 + 6x^2 - 4x - 1 = 0.$

12.5 $x^4 - 4x^3 + 6x^2 - 4x + 2 = 0.$

12.6

$$\frac{\frac{3x-6}{\cos(x)+2} - \frac{x-2}{x^2+1}}{\frac{x^2+x+10}{e^x+x^2}} + x^3 - 8 = 0.$$

12.7

$$10000000 i \frac{(1+i)^{12}}{(1+i)^{12} - 1} = 945560.$$

12.8 $x_1^2 - x_1x_2 + 3x_1 - 4x_2 + 10 = 0,$
 $-2x_1^2 + x_2^2 + 3x_1x_2 - 4x_1 + 5x_2 - 42 = 0.$

12.9 $x_1 + x_2 + 2x_1x_2 - 31 = 0,$
 $6x_1 + 5x_2 + 3x_1x_2 - 74 = 0.$

13

Interpolación y aproximación

En muchas situaciones de la vida real se tiene una tabla de valores correspondientes a dos magnitudes relacionadas; por ejemplo,

Año	Población
1930	3425
1940	5243
1950	10538
1960	19123
1970	38765
1980	82468
1985	91963
1990	103646
1995	123425

De manera más general, se tiene una tabla de valores

x_0	$f(x_0)$
x_1	$f(x_1)$
x_2	$f(x_2)$
\vdots	\vdots
x_n	$f(x_n)$

y se desea obtener una función \tilde{f} , sencilla y fácil de calcular, aproximación de f , o en otros casos, dado un \bar{x} , se desea obtener $\tilde{f}(\bar{x})$ valor aproximado de $f(\bar{x})$.

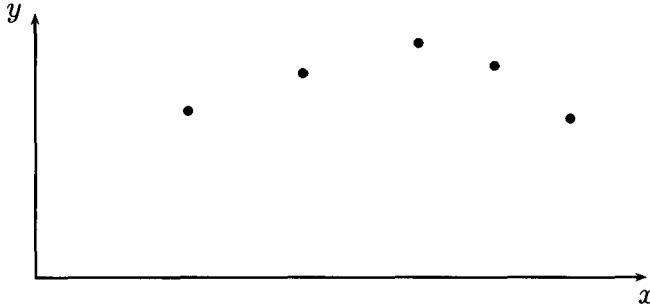


Figura 13.1

Los valores $f(x_i)$ pueden corresponder a:

- Datos o medidas obtenidos experimentalmente.
- Valores de una función f que se conoce pero tiene una expresión analítica muy complicada o de evaluación difícil o lenta.
- Una función de la que no se conoce una expresión analítica, pero se puede conocer $f(x)$ como solución de una ecuación funcional (por ejemplo, una ecuación diferencial) o como resultado de un proceso numérico.

Cuando se desea que la función \tilde{f} pase exactamente por los puntos conocidos,

$$\tilde{f}(x_i) = f(x_i) \forall i,$$

se habla de interpolación o de métodos de colocación.

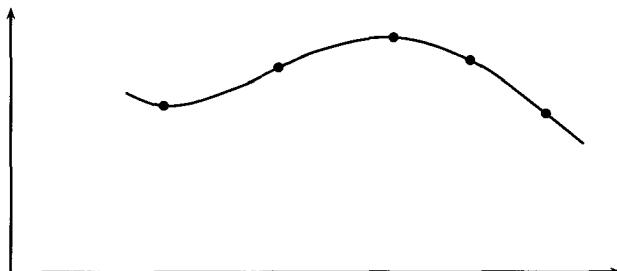


Figura 13.2

En los demás casos se habla de aproximación. En este capítulo se verá aproximación por mínimos cuadrados.

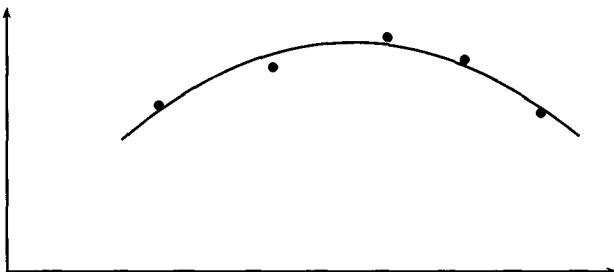


Figura 13.3

13.1 Interpolación

En el caso general de interpolación se tiene un conjunto de n puntos $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ con la condición de que los x_i son todos diferentes. Este conjunto se llama el soporte. La función \tilde{f} , que se desea construir, debe ser combinación lineal de n funciones llamadas funciones de la base. Supongamos que estas funciones son $\varphi_1, \varphi_2, \dots, \varphi_n$. Entonces,

$$\tilde{f}(x) = a_1\varphi_1(x) + a_2\varphi_2(x) + \cdots + a_n\varphi_n(x).$$

Como las funciones de la base son conocidas, para conocer \tilde{f} basta conocer los escalares a_1, a_2, \dots, a_n .

Las funciones de la base deben ser linealmente independientes. Si $n \geq 2$, la independencia lineal significa que no es posible que una de las funciones sea combinación lineal de las otras. Por ejemplo, las funciones $\varphi_1(x) = 4, \varphi_2(x) = 6x^2 - 20$ y $\varphi_3(x) = 2x^2$ no son linealmente independientes.

Los escalares a_1, a_2, \dots, a_n se escogen de tal manera que $\tilde{f}(x_i) = y_i$, para $i = 1, 2, \dots, n$. Entonces

$$\begin{aligned} a_1\varphi_1(x_1) + a_2\varphi_2(x_1) + \cdots + a_n\varphi_n(x_1) &= y_1 \\ a_1\varphi_1(x_2) + a_2\varphi_2(x_2) + \cdots + a_n\varphi_n(x_2) &= y_2 \\ &\vdots \\ a_1\varphi_1(x_n) + a_2\varphi_2(x_n) + \cdots + a_n\varphi_n(x_n) &= y_n \end{aligned}$$

Las m igualdades anteriores se pueden escribir matricialmente:

$$\begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \cdots & \varphi_n(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \cdots & \varphi_n(x_2) \\ \vdots & & & \\ \varphi_1(x_n) & \varphi_2(x_n) & \cdots & \varphi_n(x_n) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

De manera compacta se tiene

$$\Phi a = y. \quad (13.1)$$

La matriz Φ es una matriz cuadrada $n \times n$, a es un vector columna $n \times 1$, y es un vector columna $n \times 1$. Son conocidos la matriz Φ y el vector columna y . El vector columna a es el vector de incógnitas. Como las funciones de la base son linealmente independientes, entonces las columnas de Φ son linealmente independientes. En consecuencia, Φ es invertible y (13.1) se puede resolver (numéricamente).

Ejemplo 13.1. Dados los puntos $(-1, 1)$, $(2, -2)$, $(3, 5)$ y la base formada por las funciones $\varphi_1(x) = 1$, $\varphi_2(x) = x$, $\varphi_3(x) = x^2$, encontrar la función de interpolación.

Al plantear $\Phi a = y$, se tiene

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 5 \end{bmatrix}$$

Entonces

$$a = \begin{bmatrix} -4 \\ -3 \\ 2 \end{bmatrix}, \quad \tilde{f}(x) = -4 - 3x + 2x^2,$$

que efectivamente pasa por los puntos dados. \diamond

Ejemplo 13.2. Dados los puntos mismos $(-1, 1)$, $(2, -2)$, $(3, 5)$ y la base formada por las funciones $\varphi_1(x) = 1$, $\varphi_2(x) = e^x$, $\varphi_3(x) = e^{2x}$, encontrar la función de interpolación.

Al plantear $\Phi a = y$, se tiene

$$\begin{bmatrix} 1 & 0.3679 & 0.1353 \\ 1 & 7.3891 & 54.5982 \\ 1 & 20.0855 & 403.4288 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 5 \end{bmatrix}$$

Entonces

$$a = \begin{bmatrix} -1.2921 \\ -0.8123 \\ 0.0496 \end{bmatrix}, \quad \tilde{f}(x) = 1.2921 - 0.8123e^x + 0.0496e^{2x},$$

que efectivamente también pasa por los puntos dados. \diamond

13.2 Interpolación de Lagrange

En la interpolación de Lagrange la función \tilde{f} que pasa por los puntos es un polinomio, pero el polinomio se calcula sin resolver explícitamente un sistema de ecuaciones. Más precisamente, dados $n+1$ puntos

$$(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n),$$

donde $y_i = f(x_i) = f_i$, se desea encontrar un polinomio $p \in \mathcal{P}_n$ (el conjunto de polinomios de grado menor o igual a n), que pase exactamente por esos puntos, es decir,

$$p(x_i) = y_i, \quad i = 0, 1, 2, \dots, n. \quad (13.2)$$

Por ejemplo, se desea encontrar un polinomio de grado menor o igual a 2 que pase por los puntos

$$(-1, 1), (2, -2), (3, 5).$$

Los valores x_i deben ser todos diferentes entre sí. Sin perder generalidad, se puede suponer que $x_0 < x_1 < x_2 < \dots < x_n$.

El problema 13.2 se puede resolver planteando $n+1$ ecuaciones con $n+1$ incógnitas (los coeficientes del polinomio). Este sistema lineal se puede resolver y se tendría la solución. Una manera más adecuada de encontrar p es por medio de los polinomios de Lagrange.

13.2.1 Algunos resultados previos

Teorema 13.1. *Sea $p \in \mathcal{P}_n$. Si existen $n+1$ valores diferentes $x_0, x_1, x_2, \dots, x_n$ tales que $p(x_i) = 0 \forall i$, entonces $p(x) = 0 \forall x$, es decir, p es el polinomio nulo.*

Teorema 13.2. Teorema del valor medio. Sea f derivable en el intervalo $[a, b]$, entonces existe $c \in [a, b]$ tal que

$$\frac{f(b) - f(a)}{b - a} = f'(c).$$

Corolario 13.1. Si $f(a) = f(b) = 0$, entonces existe $c \in [a, b]$ tal que

$$f'(c) = 0.$$

13.2.2 Polinomios de Lagrange

Dados $n + 1$ valores diferentes $x_0, x_1, x_2, \dots, x_n$, se definen $n + 1$ polinomios de Lagrange $L_0, L_1, L_2, \dots, L_n$ de la siguiente manera:

$$L_k(x) = \frac{\prod_{i=0, i \neq k}^n (x - x_i)}{\prod_{i=0, i \neq k}^n (x_k - x_i)}. \quad (13.3)$$

La construcción de los polinomios de Lagrange, para los datos del último ejemplo $x_0 = -1, x_1 = 2, x_2 = 3$, da:

$$\begin{aligned} L_0(x) &= \frac{(x - 2)(x - 3)}{(-1 - 2)(-1 - 3)} = \frac{x^2 - 5x + 6}{12}, \\ L_1(x) &= \frac{(x - -1)(x - 3)}{(2 - -1)(2 - 3)} = \frac{x^2 - 2x - 3}{-3}, \\ L_2(x) &= \frac{(x - -1)(x - 2)}{(3 - -1)(3 - 2)} = \frac{x^2 - x - 2}{4}. \end{aligned}$$

Es claro que el numerador de (13.3) es el producto de n polinomios de grado 1; entonces el numerador es un polinomio de grado, exactamente, n . El denominador es el producto de n números, ninguno de los cuales es nulo, luego el denominador es un número no nulo. En resumen, L_k es un polinomio de grado n .

Reemplazando se verifica que L_k se anula en todos los x_i , salvo en x_k ,

$$L_k(x_i) = \begin{cases} 0 & \text{si } i \neq k, \\ 1 & \text{si } i = k. \end{cases} \quad (13.4)$$

En el ejemplo, $L_2(-1) = 0$, $L_2(2) = 0$, $L_2(3) = 1$.

Con los polinomios de Lagrange se construye inmediatamente p ,

$$p(x) = \sum_{k=0}^n y_k L_k(x). \quad (13.5)$$

Por construcción p es un polinomio en \mathcal{P}_n . Reemplazando, fácilmente se verifica 13.2.

Para el ejemplo,

$$p(x) = 1L_0(x) - 2L_1(x) + 5L_2(x) = 2x^2 - 3x - 4.$$

Ejemplo 13.3. Encontrar el polinomio, de grado menor o igual a 3, que pasa por los puntos

$$(-1, 1), (1, -5), (2, -2), (3, 5).$$

$$\begin{aligned} L_0(x) &= \frac{(x-1)(x-2)(x-3)}{(-1-1)(-1-2)(-1-3)} = \frac{x^3 - 6x^2 + 11x - 6}{-24}, \\ L_1(x) &= \frac{x^3 - 4x^2 + x + 6}{4}, \\ L_2(x) &= \frac{x^3 - 3x^2 - x + 3}{-3}, \\ L_3(x) &= \frac{x^3 - 2x^2 - x + 2}{8}, \\ p(x) &= 2x^2 - 3x - 4. \quad \diamond \end{aligned}$$

En la práctica se usa la interpolación de Lagrange de grado 2 o 3, máximo 4. Si hay muchos puntos, éstos se utilizan por grupos de 3 o 4, máximo 5 puntos.

13.2.3 Existencia, unicidad y error

El polinomio $p \in \mathcal{P}_n$ existe puesto que se puede construir. Sea $q \in \mathcal{P}_n$ otro polinomio tal que

$$q(x_i) = y_i, \quad i = 0, 1, 2, \dots, n.$$

Sea $r(x) = p(x) - q(x)$. Por construcción, $r \in \mathcal{P}_n$, además $r(x_i) = 0$, $i = 0, 1, 2, n$, o sea, r se anula en $n+1$ valores diferentes, luego $r(x) = 0$, de donde $q(x) = p(x)$.

Teorema 13.3. Sean $x_0, x_1, x_2, \dots, x_n$ reales distintos; t un real; I_t el menor intervalo que contiene a $x_0, x_1, x_2, \dots, x_n, t$; $f \in C_{I_t}^{n+1}$ (f tiene derivadas continuas de orden 0, 1, 2, ..., $n+1$); p_n el polinomio de grado menor o igual a n que pasa por los $n+1$ puntos $(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n))$. Entonces $E(t)$, el error en t , está dado por:

$$E(t) = f(t) - p_n(t) = (t - x_0)(t - x_1) \cdots (t - x_n) f^{(n+1)}(\xi) / (n+1)! \quad (13.6)$$

para algún $\xi \in I_t$.

Demostración. Si $t = x_i$ para algún i , entonces se tiene trivialmente el resultado. Supongamos ahora que $t \notin \{x_0, x_1, x_2, \dots, x_n\}$. Sean

$$\begin{aligned}\Phi(x) &= (x - x_0)(x - x_1) \cdots (x - x_n), \\ G(x) &= E(x) - \frac{\Phi(x)}{\Phi(t)} E(t).\end{aligned}$$

Entonces

$$\begin{aligned}G &\in C_{I_t}^{n+1}, \\ G(x_i) &= E(x_i) - \frac{\Phi(x_i)}{\Phi(t)} E(t) = 0, \\ G(t) &= E(t) - \frac{\Phi(t)}{\Phi(t)} E(t) = 0.\end{aligned}$$

Como G tiene por lo menos $n+2$ ceros en I_t , aplicando el corolario del teorema del valor medio, se deduce que G' tiene por lo menos $n+2-1$ ceros en I_t . Así sucesivamente se concluye que $G^{(n+1)}$ tiene por lo menos un cero en I_t . Sea ξ tal que

$$G^{(n+1)}(\xi) = 0.$$

De acuerdo con las definiciones

$$\begin{aligned}E^{(n+1)}(x) &= f^{(n+1)}(x) - p_n^{(n+1)}(x) = f^{(n+1)}(x), \\ \Phi^{(n+1)}(x) &= (n+1)!, \\ G^{(n+1)}(x) &= E^{(n+1)}(x) - \frac{\Phi^{(n+1)}(x)}{\Phi(t)} E(t), \\ G^{(n+1)}(x) &= f^{(n+1)}(x) - \frac{(n+1)!}{\Phi(t)} E(t), \\ G^{(n+1)}(\xi) &= f^{(n+1)}(\xi) - \frac{(n+1)!}{\Phi(t)} E(t) = 0.\end{aligned}$$

Teorema 13.3. Sean $x_0, x_1, x_2, \dots, x_n$ reales distintos; t un real; I_t el menor intervalo que contiene a $x_0, x_1, x_2, \dots, x_n, t$; $f \in C_{I_t}^{n+1}$ (f tiene derivadas continuas de orden 0, 1, 2, ..., $n+1$); p_n el polinomio de grado menor o igual a n que pasa por los $n+1$ puntos $(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n))$. Entonces $E(t)$, el error en t , está dado por:

$$E(t) = f(t) - p_n(t) = (t-x_0)(t-x_1) \cdots (t-x_n) f^{(n+1)}(\xi) / (n+1)! \quad (13.6)$$

para algún $\xi \in I_t$.

Demostración. Si $t = x_i$ para algún i , entonces se tiene trivialmente el resultado. Supongamos ahora que $t \notin \{x_0, x_1, x_2, \dots, x_n\}$. Sean

$$\begin{aligned}\Phi(x) &= (x-x_0)(x-x_1) \cdots (x-x_n), \\ G(x) &= E(x) - \frac{\Phi(x)}{\Phi(t)} E(t).\end{aligned}$$

Entonces

$$\begin{aligned}G &\in C_{I_t}^{n+1}, \\ G(x_i) &= E(x_i) - \frac{\Phi(x_i)}{\Phi(t)} E(t) = 0, \\ G(t) &= E(t) - \frac{\Phi(t)}{\Phi(t)} E(t) = 0.\end{aligned}$$

Como G tiene por lo menos $n+2$ ceros en I_t , aplicando el corolario del teorema del valor medio, se deduce que G' tiene por lo menos $n+2-1$ ceros en I_t . Así sucesivamente se concluye que $G^{(n+1)}$ tiene por lo menos un cero en I_t . Sea ξ tal que

$$G^{(n+1)}(\xi) = 0.$$

De acuerdo con las definiciones

$$\begin{aligned}E^{(n+1)}(x) &= f^{(n+1)}(x) - p_n^{(n+1)}(x) = f^{(n+1)}(x), \\ \Phi^{(n+1)}(x) &= (n+1)!, \\ G^{(n+1)}(x) &= E^{(n+1)}(x) - \frac{\Phi^{(n+1)}(x)}{\Phi(t)} E(t), \\ G^{(n+1)}(x) &= f^{(n+1)}(x) - \frac{(n+1)!}{\Phi(t)} E(t), \\ G^{(n+1)}(\xi) &= f^{(n+1)}(\xi) - \frac{(n+1)!}{\Phi(t)} E(t) = 0.\end{aligned}$$

Entonces

$$E(t) = \frac{\Phi(t)}{(n+1)!} f^{(n+1)}(\xi).$$

13.3 Diferencias divididas de Newton

Esta es una manera diferente de hacer los cálculos para la interpolación polinómica. En la interpolación de Lagrange se construye explícitamente p , es decir, se conocen sus coeficientes. Por medio de las diferencias divididas no se tiene explícitamente el polinomio, pero se puede obtener fácilmente el valor $p(x)$ para cualquier x .

Supongamos de nuevo que tenemos los mismos $n+1$ puntos,

$$(x_0, f_0), (x_1, f_1), (x_2, f_2), \dots, (x_{n-1}, f_{n-1}), (x_n, f_n).$$

Con ellos se obtiene $p_n \in \mathcal{P}_n$. Si se consideran únicamente los primeros n puntos

$$(x_0, f_0), (x_1, f_1), (x_2, f_2), \dots, (x_{n-1}, f_{n-1}),$$

se puede construir $p_{n-1} \in \mathcal{P}_{n-1}$. Sea $c(x)$ la corrección que permite pasar de p_{n-1} a p_n ,

$$p_n(x) = p_{n-1}(x) + c(x), \text{ es decir, } c(x) = p_n(x) - p_{n-1}(x).$$

Por construcción, c es un polinomio en \mathcal{P}_n . Además,

$$c(x_i) = p_n(x_i) - p_{n-1}(x_i) = 0, \quad i = 0, 1, 2, \dots, n-1.$$

La fórmula anterior dice que c tiene n raíces diferentes x_0, x_1, \dots, x_{n-1} , entonces

$$c(x) = \alpha_n(x - x_0)(x - x_1)(x - x_2) \cdots (x - x_{n-1}).$$

$$f(x_n) = p(x_n) = p_{n-1}(x_n) + c(x_n),$$

$$f(x_n) = p_{n-1}(x_n) + \alpha_n(x_n - x_0)(x_n - x_1)(x_n - x_2) \cdots (x_n - x_{n-1}).$$

De la última igualdad se puede despejar α_n . Este valor se define como la diferencia dividida de orden n de f en los puntos $x_0, x_1, x_2, \dots, x_n$. Se denota

$$\alpha_n = f[x_0, x_1, \dots, x_n] := \frac{f(x_n) - p_{n-1}(x_n)}{(x_n - x_0)(x_n - x_1)(x_n - x_2) \cdots (x_n - x_{n-1})}.$$

El nombre diferencia dividida no tiene, por el momento, un significado muy claro; éste se verá más adelante. Una de las igualdades anteriores se reescribe

$$p_n(x) = p_{n-1}(x) + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}). \quad (13.7)$$

Esta fórmula es la que se utiliza para calcular $p_n(x)$, una vez que se sepa calcular, de manera sencilla, $f[x_0, x_1, \dots, x_n]$.

- Para calcular $p(x)$, se empieza calculando $p_0(x)$.
- A partir de $p_0(x)$, con el valor $f[x_0, x_1]$, se calcula $p_1(x)$.
- A partir de $p_1(x)$, con el valor $f[x_0, x_1, x_2]$, se calcula $p_2(x)$.
- A partir de $p_2(x)$, con el valor $f[x_0, x_1, x_2, x_3]$, se calcula $p_3(x)$.
- \vdots
- A partir de $p_{n-1}(x)$, con el valor $f[x_0, x_1, \dots, x_n]$, se calcula $p_n(x)$.

Obviamente

$$p_0(x) = f(x_0). \quad (13.8)$$

Por definición, consistente con lo visto antes,

$$f[x_0] := f(x_0),$$

que se generaliza a

$$f[x_i] := f(x_i), \quad \forall i. \quad (13.9)$$

Las demás diferencias divididas se deducen de (13.7),

$$\begin{aligned} p_1(x) &= p_0(x) + f[x_0, x_1](x - x_0), \\ f[x_0, x_1] &= \frac{p_1(x) - p_0(x)}{x - x_0}. \end{aligned}$$

Para $x = x_1$,

$$\begin{aligned} f[x_0, x_1] &= \frac{p_1(x_1) - p_0(x_1)}{x_1 - x_0}, \\ f[x_0, x_1] &= \frac{f(x_1) - f(x_0)}{x_1 - x_0}, \\ f[x_0, x_1] &= \frac{f[x_1] - f[x_0]}{x_1 - x_0}. \end{aligned}$$

La anterior igualdad se generaliza a

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}. \quad (13.10)$$

Deducción de $f[x_0, x_1, x_2]$:

$$\begin{aligned} p_2(x) &= p_1(x) + f[x_0, x_1, x_2](x - x_0)(x - x_1), \\ f[x_0, x_1, x_2] &= \frac{p_2(x) - p_1(x)}{(x - x_0)(x - x_1)}, \\ x &= x_2, \\ f[x_0, x_1, x_2] &= \frac{p_2(x_2) - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)}, \\ &= \dots \\ f[x_0, x_1, x_2] &= \frac{f_0(x_2 - x_1) - f_1(x_2 - x_0) + f_2(x_1 - x_0)}{(x_2 - x_1)(x_2 - x_0)(x_1 - x_0)}. \end{aligned}$$

Por otro lado,

$$\begin{aligned} \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} &= \frac{\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0}, \\ &= \dots \\ \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} &= \frac{f_0(x_2 - x_1) - f_1(x_2 - x_0) + f_2(x_1 - x_0)}{(x_2 - x_1)(x_2 - x_0)(x_1 - x_0)}. \end{aligned}$$

Luego

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}.$$

Generalizando,

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}. \quad (13.11)$$

La generalización para diferencias divididas de orden j es:

$$f[x_i, x_{i+1}, \dots, x_{i+j}] = \frac{f[x_{i+1}, \dots, x_{i+j}] - f[x_i, \dots, x_{i+j-1}]}{x_{i+j} - x_i}. \quad (13.12)$$

Las fórmulas anteriores dan sentido al nombre diferencias divididas. Cuando no se preste a confusión, se puede utilizar la siguiente notación:

$$D^j f[x_i] := f[x_i, x_{i+1}, \dots, x_{i+j}]. \quad (13.13)$$

Entonces

$$D^0 f[x_i] := f(x_i), \quad (13.14)$$

$$Df[x_i] = D^1 f[x_i] = \frac{D^0 f[x_{i+1}] - D^0 f[x_i]}{x_{i+1} - x_i}, \quad (13.15)$$

$$D^2 f[x_i] = \frac{D^1 f[x_{i+1}] - D^1 f[x_i]}{x_{i+2} - x_i}, \quad (13.16)$$

$$D^j f[x_i] = \frac{D^{j-1} f[x_{i+1}] - D^{j-1} f[x_i]}{x_{i+j} - x_i}. \quad (13.17)$$

13.3.1 Tabla de diferencias divididas

Para ejemplos pequeños, hechos a mano, se acostumbra construir la tabla de diferencias divididas, la cual tiene el siguiente aspecto:

x_i	f_i	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$
x_0	f_0	$f[x_0, x_1]$		
x_1	f_1		$f[x_0, x_1, x_2]$	
x_2	f_2	$f[x_1, x_2]$		$f[x_0, x_1, x_2, x_3]$
x_3	f_3	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$	$f[x_1, x_2, x_3, x_4]$
x_4	f_4	$f[x_3, x_4]$	$f[x_2, x_3, x_4]$	

En la tabla anterior, dados 5 puntos, están las diferencias divididas hasta de orden 3. Claro está, se hubiera podido calcular también la diferencia dividida de orden 4, que estaría colocada en una columna adicional a la derecha.

La elaboración de la tabla es relativamente sencilla. Las dos primeras columnas corresponden a los datos. La tercera columna, la de las diferencias divididas de primer orden, $f[x_i, x_{i+1}]$, se obtiene mediante la resta de dos elementos consecutivos de la columna anterior dividida por la resta de los dos elementos correspondientes de la primera columna. Por ejemplo, $f[x_3, x_4] = (f_4 - f_3)/(x_4 - x_3)$. Obsérvese que este valor se coloca en medio de la fila de f_3 y de la fila de f_4 .

Para el cálculo de una diferencia dividida de segundo orden, cuarta columna, se divide la resta de dos elementos consecutivos de la columna anterior por la resta de dos elementos de la primera columna, pero dejando uno intercalado. Por ejemplo, $f[x_1, x_2, x_3] = (f[x_2, x_3] - f[x_1, x_2])/(x_3 - x_1)$.

Para el cálculo de una diferencia dividida de tercer orden, quinta columna, se divide la resta de dos elementos consecutivos de la columna anterior por la resta de dos elementos de la primera columna, pero dejando dos intercalados. Por ejemplo, $f[x_0, x_1, x_2, x_3] = (f[x_1, x_2, x_3] - f[x_0, x_1, x_2])/(x_3 - x_0)$.

Ejemplo 13.4. Construir la tabla de diferencias divididas, hasta el orden 3, a partir de los seis puntos siguientes:

$$(0, 0), (0.5, 0.7071), (1, 1), (2, 1.4142), (3, 1.7321), (4, 2).$$

x_i	f_i	$Df[x_i]$	$D^2f[x_i]$	$D^3f[x_i]$
0	0.0000			
.5	0.7071	1.4142		
1	1.0000	0.5858	-0.8284	0.3570
2	1.4142	0.4142	-0.1144	0.0265
3	1.7321	0.3179	-0.0482	0.0077
4	2.0000	0.2679	-0.0250	

El valor 1.4142 es simplemente $(0.7071 - 0)/(0.5 - 0)$. El valor 0.2679 es simplemente $(2 - 1.7321)/(4 - 3)$. El valor -0.1144 es simplemente $(0.4142 - 0.5858)/(2 - .5)$. El valor 0.0077 es simplemente $(-0.0250 - -0.0482)/(4 - 1)$. ◇

El esquema algorítmico para calcular la tabla de diferencias divididas hasta el orden m es el siguiente:

```

para  $i = 0, \dots, n$ 
     $D^0 f[x_i] = f(x_i)$ 
fin-para  $i$ 
para  $j = 1, \dots, m$ 
    para  $i = 0, \dots, n - j$ 
        calcular  $D^j f[x_i]$  según (13.17)
    fin-para  $i$ 
fin-para  $j$ 

```

Si los datos $f(x_i)$ corresponden a un polinomio, esto se puede deducir mediante las siguientes observaciones:

- Si para algún m todos los valores $f[x_k, x_{k+1}, \dots, x_{k+m}]$ son iguales (o aproximadamente iguales), entonces f es (aproximadamente) un polinomio de grado m .
- Si para algún r todos los valores $f[x_k, x_{k+1}, \dots, x_{k+r}]$ son nulos (o aproximadamente nulos), entonces f es (aproximadamente) un polinomio de grado $r - 1$.

13.3.2 Cálculo del valor interpolado

La fórmula (13.7) se puede reescribir a partir de un punto x_k , pues no siempre se debe tomar como valor de referencia x_0 ,

$$p_m(x) = p_{m-1}(x) + D^m f[x_k](x - x_k)(x - x_{k+1}) \cdots (x - x_{k+m-1}). \quad (13.18)$$

Si se calcula $p_{m-1}(x)$ de manera análoga, queda en función de $p_{m-2}(x)$ y así sucesivamente se obtiene:

$$p_m(x) = \sum_{i=0}^m \left[D^i f[x_k] \prod_{j=0}^{i-1} (x - x_{k+j}) \right]. \quad (13.19)$$

El proceso para el cálculo es el siguiente:

$$\begin{aligned}
 p_0(x) &= f_k \\
 p_1(x) &= p_0(x) + D^1 f[x_k](x - x_k) \\
 p_2(x) &= p_1(x) + D^2 f[x_k](x - x_k)(x - x_{k+1}) \\
 p_3(x) &= p_2(x) + D^3 f[x_k](x - x_k)(x - x_{k+1})(x - x_{k+2}) \\
 p_4(x) &= p_3(x) + D^4 f[x_k](x - x_k)(x - x_{k+1})(x - x_{k+2})(x - x_{k+3}) \\
 &\vdots
 \end{aligned}$$

Se observa que para calcular $p_j(x)$ hay multiplicaciones que ya se hicieron para obtener $p_{j-1}(x)$; entonces, no es necesario repetirlas sino organizar el proceso de manera más eficiente.

$$\begin{aligned}\gamma_0 &= 1, & p_0(x) &= f_k \\ \gamma_1 &= \gamma_0(x - x_k), & p_1(x) &= p_0(x) + D^1 f[x_k] \gamma_1 \\ \gamma_2 &= \gamma_1(x - x_{k+1}), & p_2(x) &= p_1(x) + D^2 f[x_k] \gamma_2 \\ \gamma_3 &= \gamma_2(x - x_{k+2}), & p_3(x) &= p_2(x) + D^3 f[x_k] \gamma_3 \\ \gamma_4 &= \gamma_3(x - x_{k+3}), & p_4(x) &= p_3(x) + D^4 f[x_k] \gamma_4 \\ &\vdots\end{aligned}$$

Únicamente queda por precisar la escogencia del punto inicial o de referencia x_k . Si se desea evaluar $p_m(\bar{x})$, ¿cuál debe ser x_k ? Recordemos que se supone que los puntos $x_0, x_1, x_2, \dots, x_n$ están ordenados y que m , orden del polinomio de interpolación, es menor o igual que n . Obviamente, aunque no es absolutamente indispensable, también se supone que $\bar{x} \notin \{x_0, x_1, \dots, x_n\}$.

Naturalmente se desea que $\bar{x} \in [x_k, x_{k+m}]$. Pero no siempre se cumple; esto sucede cuando $\bar{x} \notin [x_0, x_n]$. En estos casos se habla de **extrapolación** y se debe escoger $x_k = x_0$ si $\bar{x} < x_0$. En el caso opuesto se toma $x_k = x_{n-m}$.

En los demás casos, se desea que \bar{x} esté lo “más cerca” posible del intervalo $[x_k, x_{k+m}]$ o del conjunto de puntos $x_k, x_{k+1}, x_{k+2}, \dots, x_{k+m}$.

Ejemplo 13.5. Considere los datos del ejemplo anterior para calcular por interpolación cuadrática y por interpolación cúbica una aproximación de $f(1.69)$.

El primer paso consiste en determinar el x_k . Para ello únicamente se tienen en cuenta los valores x_i .

$$\begin{array}{c} x_i \\ 0 \\ .5 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}$$

Para el caso de la interpolación cuadrática, una simple inspección visual determina que hay dos posibilidades para x_k . La primera es $x_k =$

0.5, intervalo $[0.5, 2]$. La segunda es $x_k = 1$, intervalo $[1, 3]$. ¿Cuál es mejor?

Para medir la cercanía se puede usar la distancia de \bar{x} al promedio de los extremos del intervalo $(x_i + x_{i+2})/2$ (el centro del intervalo) o la distancia de \bar{x} al promedio de todos los puntos $(x_i + x_{i+1} + x_{i+2})/3$. En general

$$u_i = \frac{x_i + x_{i+m}}{2}, \quad (13.20)$$

$$v_i = \frac{x_i + x_{i+1} + x_{i+2} + \cdots + x_{i+m}}{m+1}, \quad (13.21)$$

$$|\bar{x} - u_k| = \min_i \{|\bar{x} - u_i| : \bar{x} \in [x_i, x_{i+m}] \}, \quad (13.22)$$

$$|\bar{x} - v_k| = \min_i \{|\bar{x} - v_i| : \bar{x} \in [x_i, x_{i+m}] \}. \quad (13.23)$$

Los valores u_i y v_i son, de alguna forma, indicadores del centro de masa del intervalo $[x_i, x_{i+m}]$. Con frecuencia, los dos criterios, (13.22) y (13.23), definen el mismo x_k , pero en algunos casos no es así. De todas formas son criterios razonables y para trabajar se escoge un solo criterio, lo cual da buenos resultados. Se puede preferir la utilización de v_i que, aunque requiere más operaciones, tiene en cuenta todos los x_j pertenecientes a $[x_i, x_{i+m}]$.

Los resultados numéricos para la interpolación cuadrática dan:

x_i	u_i	$ \bar{x} - u_i $	v_i	$ \bar{x} - v_i $
0				
.5	1.25	0.44	1.1667	0.5233
1	2.00	0.31✓	2.0000	0.3100✓
2				
3				
4				

Para la interpolación cúbica hay tres posibilidades para x_k : 0, 0.5 y 1.

x_i	u_i	$ \bar{x} - u_i $	v_i	$ \bar{x} - v_i $
0	1.00	0.69	0.875	0.815
.5	1.75	0.06 \checkmark	1.625	0.065 \checkmark
1	2.50	0.81	2.500	0.810
2				
3				
4				

Una vez escogido $x_k = 1$ para obtener la aproximación cuadrática de $f(1.69)$, los cálculos dan:

$$\begin{aligned}\gamma_0 &= 1, & p_0(x) &= 1, \\ \gamma_1 &= 1(1.69 - 1) = 0.69, & p_1(x) &= 1 + 0.4142(0.69) = 1.285798 \\ \gamma_2 &= 0.69(1.69 - 2) = -0.2139, & p_2(x) &= 1.285798 - 0.0482(-0.2139) \\ && p_2(x) &= 1.296097\end{aligned}$$

Para la interpolación cúbica, $x_k = 0.5$:

$$\begin{aligned}\gamma_0 &= 1, & p_0(x) &= 0.7071, \\ \gamma_1 &= 1(1.69 - 0.5) = 1.19, & p_1(x) &= 0.7071 + 0.5858(1.19) \\ && p_1(x) &= 1.404202 \\ \gamma_2 &= 1.19(1.69 - 1) = 0.8211, & p_2(x) &= 1.404202 - 0.1144(0.8211) \\ && p_2(x) &= 1.310268 \\ \gamma_3 &= 0.8211(1.69 - 2) = -0.254541, & p_3(x) &= 1.310268 + 0.0265(-0.254541) \\ && p_3(x) &= 1.303523. \quad \diamond\end{aligned}$$

El esquema del algoritmo para calcular $p_m(\bar{x})$, a partir de la tabla de diferencia divididas, es el siguiente:

```

determinar  $x_k$ 
px = f( $x_k$ )
gi = 1.0
para  $j = 1, \dots, m$ 
    gi = gi * ( $\bar{x} - x_{k+j-1}$ )
    px = px + gi *  $D^j f[x_k]$ 
fin-para  $j$ 

```

13.4 Diferencias finitas

Cuando los puntos $(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_n, f(x_n))$, están igualmente espaciados en x , es decir, existe un $h > 0$ tal que

$$x_i = x_0 + ih, \quad i = 1, \dots, n$$

entonces se pueden utilizar las diferencias finitas, definidas por

$$\Delta^0 f_i = f_i \quad (13.24)$$

$$\Delta f_i = f_{i+1} - f_i \quad (13.25)$$

$$\Delta^{k+1} f_i = \Delta^k(\Delta f_i) = \Delta^k f_{i+1} - \Delta^k f_i \quad (13.26)$$

Algunas de las propiedades interesantes de las diferencias finitas son:

$$\Delta^k f_i = \sum_{j=0}^k (-1)^j \binom{k}{j} f_{i+k-j}, \quad (13.27)$$

$$f_{i+k} = \sum_{j=0}^k \binom{k}{j} \Delta^j f_i. \quad (13.28)$$

Las demostraciones se pueden hacer por inducción. La primera igualdad permite calcular $\Delta^k f_i$ sin tener explícitamente los valores $\Delta^{k-1} f_j$. La segunda igualdad permite el proceso inverso al cálculo de las diferencias finitas (se obtienen a partir de los valores iniciales f_p), es decir, obtener un valor f_m a partir de las diferencias finitas.

Para valores igualmente espaciados, las diferencias finitas y las divididas están estrechamente relacionadas.

$$\begin{aligned} D^0 f[x_i] = f[x_i] &= f_i = \Delta^0 f_i \\ D^1 f[x_i] = f[x_i, x_{i+1}] &= \frac{f_{i+1} - f_i}{x_{i+1} - x_i} = \frac{\Delta^1 f_i}{h} \\ D^2 f[x_i] = f[x_i, x_{i+1}, x_{i+2}] &= \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i} = \dots = \frac{\Delta^2 f_i}{2h^2} \\ D^m f[x_i] = f[x_i, \dots, x_{i+m}] &= \frac{\Delta^m f_i}{m! h^m} \end{aligned} \quad (13.29)$$

13.4.1 Tabla de diferencias finitas

La tabla de diferencias finitas tiene una estructura análoga a la tabla de diferencias divididas. Se usa para ejemplos pequeños hechos a mano.

x_i	f_i	Δf_i	$\Delta^2 f_i$	$\Delta^3 f_i$
x_0	f_0			
		Δf_0		
x_1	f_1		$\Delta^2 f_0$	
		Δf_1		$\Delta^3 f_0$
x_2	f_2		$\Delta^2 f_1$	
		Δf_2		$\Delta^3 f_1$
x_3	f_3		$\Delta^2 f_2$	
		Δf_3		
x_4	f_4			

La elaboración de la tabla es muy sencilla. Las dos primeras columnas corresponden a los datos. A partir de la tercera columna, para calcular cada elemento se hace la resta de dos elementos consecutivos de la columna anterior. Por ejemplo, $\Delta f_3 = f_4 - f_3$. Obsérvese que este valor se coloca en medio de la fila de f_3 y de la fila de f_4 . Por ejemplo, $\Delta^2 f_1 = \Delta f_2 - \Delta f_1$. De manera semejante, $\Delta^3 f_0 = \Delta^2 f_1 - \Delta^2 f_0$.

Ejemplo 13.6. Construir la tabla de diferencias finitas, hasta el orden 3, a partir de los seis puntos siguientes: (0, 0), (0.5, 0.7071), (1, 1), (1.5, 1.2247), (2, 1.4142), (2.5, 1.5811).

x_i	f_i	Δf_i	$\Delta^2 f_i$	$\Delta^3 f_i$
0	0.0000			
		0.7071		
.5	0.7071		-0.4142	
		0.2929		0.3460
1	1.0000		-0.0682	
		0.2247		0.0330
1.5	1.2247		-0.0352	
		0.1895		0.0126
2	1.4142		-0.0226	
		0.1669		
2.5	1.5811			

El valor 0.1895 es simplemente $1.4142 - 1.2247$. El valor 0.0330 es simplemente $-0.0352 - -0.0682$. ◇

El esquema algorítmico para calcular la tabla de diferencias finitas hasta el orden m es el siguiente:

```

para  $i = 0, \dots, n$ 
     $\Delta^0 f_i = f(x_i)$ 
fin-para  $i$ 
para  $j = 1, \dots, m$ 
    para  $i = 0, \dots, n - j$ 
         $\Delta^j f_i = \Delta^{j-1} f_{i+1} - \Delta^{j-1} f_i$ 
    fin-para  $i$ 
fin-para  $j$ 

```

13.4.2 Cálculo del valor interpolado

Teniendo en cuenta la relación entre diferencias divididas y finitas (13.29), la igualdad (13.19) se puede escribir

$$p_m(x) = \sum_{i=0}^m \left[\frac{\Delta^i f_k}{i! h^i} \prod_{j=0}^{i-1} (x - x_{k+j}) \right].$$

El valor $i!$ se puede escribir $\prod_{j=0}^{i-1} (j + 1)$. Además, sea $s = (x - x_k)/h$, es decir, $x = x_k + sh$. Entonces, $x - x_{k+j} = x_k + sh - x_k - jh = (s - j)h$.

$$\begin{aligned} p_m(x) &= \sum_{i=0}^m \left[\frac{\Delta^i f_k}{i! h^i} \prod_{j=0}^{i-1} (s - j)h \right] \\ &= \sum_{i=0}^m \left[\frac{\Delta^i f_k}{i!} \prod_{j=0}^{i-1} (s - j) \right] \\ &= \sum_{i=0}^m \Delta^i f_k \prod_{j=0}^{i-1} \frac{s - j}{j + 1} \end{aligned}$$

Si a y b son enteros no negativos, $a \geq b$, el coeficiente binomial está definido por

$$\binom{a}{b} = \frac{a!}{(a - b)! b!}.$$

Desarrollando los factoriales y simplificando se tiene

$$\binom{a}{b} = \frac{a(a - 1)(a - 2) \cdots (a - b + 1)}{1 \times 2 \times 3 \times \cdots \times b} = \frac{a(a - 1)(a - 2) \cdots (a - b + 1)}{b!}$$

Esta última expresión sirve para cualquier valor real a y cualquier entero no negativo b , con la convención de que $\binom{a}{0} = 1$. Entonces,

$$\prod_{j=0}^{i-1} \frac{s-j}{j+1}$$

se puede denotar simplemente por $\binom{s}{i}$ y así

$$p_m(x) = \sum_{i=0}^m \Delta^i f_k \binom{s}{i}. \quad (13.30)$$

Este coeficiente $\binom{s}{i}$ guarda propiedades semejantes a las del coeficiente binomial, en particular

$$\binom{s}{i} = \binom{s}{i-1} \frac{s-i+1}{i}.$$

Esto permite su cálculo de manera recurrente

$$\begin{aligned} \binom{s}{0} &= 1, \\ \binom{s}{1} &= \binom{s}{0}s \\ \binom{s}{2} &= \binom{s}{1} \frac{s-1}{2} \\ \binom{s}{3} &= \binom{s}{2} \frac{s-2}{3} \\ \binom{s}{4} &= \binom{s}{3} \frac{s-3}{4} \\ &\vdots \end{aligned}$$

Escoger el x_k para interpolar por un polinomio de grado m , se hace como en las diferencias divididas. Como los valores x_i están igualmente espaciados los valores, u_i y v_i coinciden.

$$\begin{aligned} u_i &= \frac{x_i + x_{i+m}}{2}, \quad i = 0, \dots, n-m, \\ |x - u_k| &= \min\{|x - u_i| : i = 0, \dots, n-m\}. \end{aligned}$$

Definido el x_k , es necesario calcular s :

$$s = \frac{x - x_k}{h}.$$

El esquema de los cálculos es:

$$\begin{aligned}\gamma_0 &= 1, & p_0(x) &= f_k \\ \gamma_1 &= \gamma_0 s, & p_1(x) &= p_0(x) + \Delta^1 f_k \gamma_1 \\ \gamma_2 &= \gamma_1(s - 1)/2, & p_2(x) &= p_1(x) + \Delta^2 f_k \gamma_2 \\ \gamma_3 &= \gamma_2(s - 2)/3, & p_3(x) &= p_2(x) + \Delta^3 f_k \gamma_3 \\ \gamma_4 &= \gamma_3(s - 3)/4, & p_4(x) &= p_3(x) + \Delta^4 f_k \gamma_4 \\ &\vdots\end{aligned}$$

Ejemplo 13.7. Calcular $p_3(1.96)$ y $p_2(1.96)$ a partir de los puntos $(0, 0)$, $(0.5, 0.7071)$, $(1, 1)$, $(1.5, 1.2247)$, $(2, 1.4142)$, $(2.5, 1.5811)$.

La tabla de diferencias finitas es la misma del ejemplo anterior. Para calcular $p_3(1.96)$ se tiene $x_k = x_2 = 1$. Entonces $s = (1.96 - 1)/0.5 = 1.92$.

$$\begin{aligned}\gamma_0 &= 1, & p_0(x) &= f_2 = 1 \\ \gamma_1 &= 1(1.92) = 1.92, & p_1(x) &= 1 + .2247(1.92) = 1.431424 \\ \gamma_2 &= 1.92(1.92 - 1)/2 = .8832, & p_2(x) &= 1.431424 - .0352(.8832) \\ && p_2(x) &= 1.400335 \\ \gamma_3 &= \gamma_2(1.92 - 2)/3 = -.023552, & p_3(x) &= 1.400335 + .0126(-.023552) \\ && p_3(x) &= 1.400039\end{aligned}$$

Para calcular $p_2(1.96)$ se tiene $x_k = x_3 = 1.5$. Entonces $s = (1.96 - 1.5)/0.5 = 0.92$.

$$\begin{aligned}\gamma_0 &= 1, & p_0(x) &= f_3 = 1.2247 \\ \gamma_1 &= 1(0.92) = 0.92, & p_1(x) &= 1.2247 + .1895(.92) = 1.39904 \\ \gamma_2 &= 0.92(0.92 - 1)/2 = -.0368, & p_2(x) &= 1.39904 - .0226(-0.0368) \\ && p_2(x) &= 1.399872\end{aligned}$$

13.5 Aproximación por mínimos cuadrados

Cuando hay muchos puntos no es conveniente buscar un único polinomio o una función que pase exactamente por todos los puntos. Entonces hay dos soluciones: la primera, vista anteriormente, es hacer interpolación por grupos pequeños de puntos. Para muchos casos es una solución

muy buena. Sin embargo, en algunas ocasiones se desea una función que sirva para todos los puntos. La segunda solución consiste en obtener una sola función \tilde{f} que, aunque no pase por todos los puntos, pase relativamente cerca de todos. Este es el enfoque de la aproximación por mínimos cuadrados.

Se supone que hay m puntos $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ y que los x_i son todos diferentes. La función \tilde{f} , que se desea construir, debe ser combinación lineal de n funciones llamadas funciones de la base. Supongamos que estas funciones son $\varphi_1, \varphi_2, \dots, \varphi_n$. Entonces,

$$\tilde{f}(x) = a_1\varphi_1(x) + a_2\varphi_2(x) + \cdots + a_n\varphi_n(x).$$

Como las funciones de la base son conocidas, para conocer \tilde{f} basta conocer los escalares a_1, a_2, \dots, a_n .

Como se supone que hay muchos puntos (m grande) y como se desea que \tilde{f} sea sencilla, es decir, n es relativamente pequeño, entonces se debe tener que $m \geq n$.

Las funciones de la base deben ser linealmente independientes. Los escalares a_1, a_2, \dots, a_n se escogen de tal manera que $\tilde{f}(x_i) \approx y_i$, para $i = 1, 2, \dots, m$. Entonces,

$$\begin{aligned} a_1\varphi_1(x_1) + a_2\varphi_2(x_1) + \cdots + a_n\varphi_n(x_1) &\approx y_1 \\ a_1\varphi_1(x_2) + a_2\varphi_2(x_2) + \cdots + a_n\varphi_n(x_2) &\approx y_2 \\ a_1\varphi_1(x_3) + a_2\varphi_2(x_3) + \cdots + a_n\varphi_n(x_3) &\approx y_3 \\ &\vdots \\ a_1\varphi_1(x_m) + a_2\varphi_2(x_m) + \cdots + a_n\varphi_n(x_m) &\approx y_m. \end{aligned}$$

Las m igualdades (aproximadas) anteriores se pueden escribir de manera matricial:

$$\left[\begin{array}{cccc} \varphi_1(x_1) & \varphi_2(x_1) & \cdots & \varphi_n(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \cdots & \varphi_n(x_2) \\ \varphi_1(x_3) & \varphi_2(x_3) & \cdots & \varphi_n(x_3) \\ \vdots & & & \\ \varphi_1(x_m) & \varphi_2(x_m) & \cdots & \varphi_n(x_m) \end{array} \right] \left[\begin{array}{c} a_1 \\ a_2 \\ \vdots \\ a_n \end{array} \right] \approx \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{array} \right]$$

De manera compacta se tiene

$$\Phi a \approx y. \quad (13.31)$$

La matriz Φ es una matriz $m \times n$ rectangular alta ($m \geq n$), a es un vector columna $n \times 1$, y es un vector columna $m \times 1$. Son conocidos la matriz Φ y el vector columna y . El vector columna a es el vector de incógnitas. Como las funciones de la base son linealmente independientes, entonces las columnas de Φ son linealmente independientes. En consecuencia, (13.31) se puede resolver por mínimos cuadrados:

$$(\Phi^T \Phi) a = \Phi^T y. \quad (13.32)$$

Recordemos del capítulo 11 que para resolver por mínimos cuadrados el sistema $Ax = b$, se minimiza $\|Ax - b\|_2^2$. Traduciendo esto al problema de aproximación por mínimos cuadrados, se tiene

$$\min \sum_{i=1}^m \left(\sum_{j=1}^n a_j \varphi_j(x_i) - y_i \right)^2.$$

es decir,

$$\min \sum_{i=1}^m (\tilde{f}(x_i) - y_i)^2.$$

Esto significa que se está buscando una función \tilde{f} , combinación lineal de las funciones de la base, tal que minimiza la suma de los cuadrados de las distancias entre los puntos $(x_i, \tilde{f}(x_i))$ y (x_i, y_i) .

Ejemplo 13.8. Dadas las funciones $\varphi_1(x) = 1$, $\varphi_2(x) = x$, $\varphi_3(x) = x^2$, encontrar la función \tilde{f} que aproxima por mínimos cuadrados la función dada por los puntos $(0, 0.55)$, $(1, 0.65)$, $(1.5, 0.725)$, $(2, 0.85)$, $(3, 1.35)$.

Como las funciones de la base son 1 , x , x^2 , en realidad se está buscando aproximar por mínimos cuadrados por medio de un parábola. El sistema inicial es

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1.5 & 2.25 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \approx \begin{bmatrix} 0.55 \\ 0.65 \\ 0.725 \\ 0.85 \\ 1.35 \end{bmatrix}$$

Las ecuaciones normales dan:

$$\begin{bmatrix} 5 & 7.5 & 16.25 \\ 7.5 & 16.25 & 39.375 \\ 16.25 & 39.375 & 103.0625 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 4.1250 \\ 7.4875 \\ 17.8313 \end{bmatrix}$$

La solución es:

$$a = \begin{bmatrix} 0.56 \\ -0.04 \\ 0.10 \end{bmatrix}, \quad \tilde{f}(x) = 0.56 - 0.04x + 0.1x^2.$$

$$\begin{bmatrix} \tilde{f}(x_1) \\ \tilde{f}(x_2) \\ \tilde{f}(x_3) \\ \tilde{f}(x_4) \\ \tilde{f}(x_5) \end{bmatrix} = \Phi a = \begin{bmatrix} 0.56 \\ 0.62 \\ 0.725 \\ 0.88 \\ 1.34 \end{bmatrix}, \quad y = \begin{bmatrix} 0.55 \\ 0.65 \\ 0.725 \\ 0.85 \\ 1.35 \end{bmatrix} \diamond$$

Ejercicios

- 13.1** Halle, resolviendo el sistema de ecuaciones, el polinomio de interpolación que pasa por los puntos

$$(1, -5), \\ (2, -4), \\ (4, 4).$$

- 13.2** Halle, por medio de los polinomios de Lagrange, el polinomio de interpolación que pasa por los puntos del ejercicio anterior.

- 13.3** Halle el polinomio de interpolación que pasa por los puntos

$$(-1, -5), \\ (1, -5), \\ (2, -2), \\ (4, 40).$$

- 13.4** Halle el polinomio de interpolación que pasa por los puntos

$$(-1, 10), \\ (1, 8), \\ (2, 4), \\ (4, -10).$$

13.5 Considere los puntos

$$\begin{aligned} & (0.10, 11.0000), \\ & (0.13, 8.6923), \\ & (0.16, 7.2500), \\ & (0.20, 6.0000), \\ & (0.26, 4.8462), \\ & (0.40, 3.5000), \\ & (0.32, 4.1250), \\ & (0.50, 3.0000). \end{aligned}$$

Construya la tabla de diferencias divididas hasta el orden 3. Obtenga $p_2(0.11)$, $p_2(0.08)$, $p_2(0.25)$, $p_2(0.12)$, $p_2(0.33)$, $p_2(0.6)$, $p_3(0.25)$, $p_3(0.33)$, $p_3(0.6)$.

13.6 Considere los puntos

$$\begin{aligned} & (0.05, 21.0000), \\ & (0.10, 11.0000), \\ & (0.15, 7.6667), \\ & (0.20, 6.0000), \\ & (0.25, 5.0000), \\ & (0.30, 4.3333), \\ & (0.35, 3.8571), \\ & (0.40, 3.5000). \end{aligned}$$

Construya la tabla de diferencias divididas hasta el orden 3. Calcule $p_2(0.11)$, $p_2(0.08)$, $p_2(0.25)$, $p_2(0.12)$, $p_2(0.33)$, $p_2(0.6)$, $p_3(0.25)$, $p_3(0.33)$, $p_3(0.6)$.

13.7 Considere los mismos puntos del ejercicio anterior. Construya la tabla de diferencias finitas hasta el orden 3. Halle $p_2(0.11)$, $p_2(0.08)$, $p_2(0.25)$, $p_2(0.12)$, $p_2(0.33)$, $p_2(0.6)$, $p_3(0.25)$, $p_3(0.33)$, $p_3(0.6)$.

13.8 Considere los puntos

$$\begin{aligned} &(0.05, 2.0513), \\ &(0.10, 2.1052), \\ &(0.15, 2.1618), \\ &(0.20, 2.2214), \\ &(0.25, 2.2840), \\ &(0.30, 2.3499), \\ &(0.35, 2.4191), \\ &(0.40, 2.4918). \end{aligned}$$

Obtenga la recta de aproximación por mínimos cuadrados.

- 13.9** Considere los mismos puntos del ejercicio anterior. Obtenga la parábola de aproximación por mínimos cuadrados.
- 13.10** Considere los mismos puntos de los dos ejercicios anteriores. Use otra base y obtenga la correspondiente función de aproximación por mínimos cuadrados.

14

Integración y diferenciación

14.1 Integración numérica

Esta técnica sirve para calcular el valor numérico de una integral definida, es decir, para obtener el valor

$$I = \int_a^b f(x)dx.$$

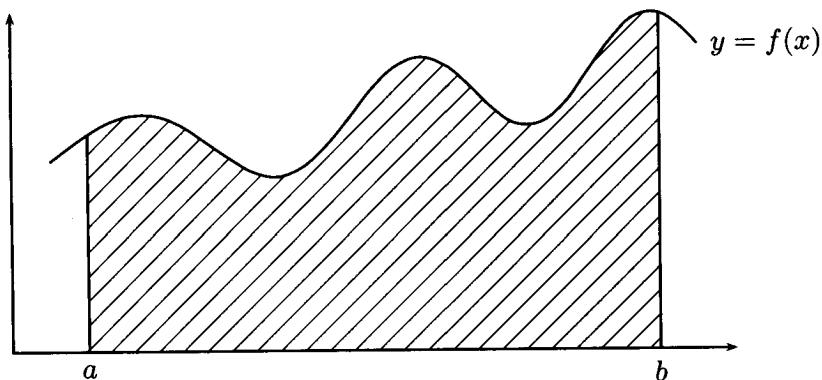


Figura 14.1

En la mayoría de los casos no se puede calcular el valor exacto I ; simplemente se calcula \tilde{I} aproximación de I .

De todas maneras primero se debe tratar de hallar la antiderivada. Cuando esto sea imposible o muy difícil, entonces se recurre a la

integración numérica. Por ejemplo, calcular una aproximación de

$$\int_{0.1}^{0.5} e^{x^2} dx.$$

En este capítulo hay ejemplos de integración numérica con funciones cuya antiderivada es muy fácil de obtener y para los que no se debe utilizar la integración numérica; se usan solamente para comparar el resultado aproximado con el valor exacto.

14.2 Fórmula del trapecio

La fórmula del trapecio, como también la fórmula de Simpson, hace parte de las fórmulas de Newton-Cotes. Sean $n + 1$ valores igualmente espaciados $a = x_0, x_1, x_2, \dots, x_n = b$, donde

$$x_i = a + ih, \quad i = 0, 1, 2, \dots, n, \quad h = \frac{b - a}{n},$$

y supongamos conocidos $y_i = f(x_i)$. Supongamos además que n es un múltiplo de m , $n = km$. La integral $\int_{x_0}^{x_n} f(x)dx$ se puede separar en intervalos más pequeños:

$$\int_{x_0}^{x_n} f(x)dx = \int_{x_0}^{x_m} f(x)dx + \int_{x_m}^{x_{2m}} f(x)dx + \cdots + \int_{x_{n-m}}^{x_n} f(x)dx.$$

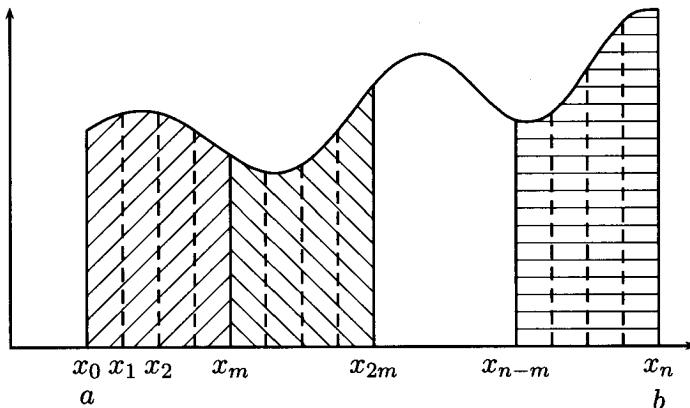


Figura 14.2

En el intervalo $[x_0, x_m]$ se conocen los puntos (x_0, y_0) , (x_1, y_1) , ..., (x_m, y_m) y se puede construir el polinomio de interpolación de Lagrange $p_m(x)$. Entonces la integral $\int_{x_0}^{x_m} f(x)dx$ se aproxima por la integral de p_m ,

$$\int_{x_0}^{x_m} f(x)dx \approx \int_{x_0}^{x_m} p_m(x)dx.$$

Para $m = 1$ se tiene la fórmula del trapecio. Su deducción es mucho más sencilla si se supone que $x_0 = 0$. Esto equivale a hacer el cambio de variable $x' = x - x_0$.

$$\begin{aligned} p_1(x) &= y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}, \\ p_1(x) &= y_0 \frac{x - h}{-h} + y_1 \frac{x}{h}, \\ p_1(x) &= y_0 + x \frac{y_1 - y_0}{h}. \end{aligned}$$

Entonces

$$\begin{aligned} \int_{x_0}^{x_1} p_1(x)dx &= \int_0^h (y_0 + x \frac{y_1 - y_0}{h}) dx \\ &= y_0 h + \frac{h^2}{2} \frac{y_1 - y_0}{h}, \\ &= h \left(\frac{y_0}{2} + \frac{y_1}{2} \right), \\ \int_{x_0}^{x_1} f(x)dx &\approx h \frac{y_0 + y_1}{2}. \end{aligned} \tag{14.1}$$

De la fórmula (14.1) o de la gráfica se deduce naturalmente el nombre de fórmula del trapecio.

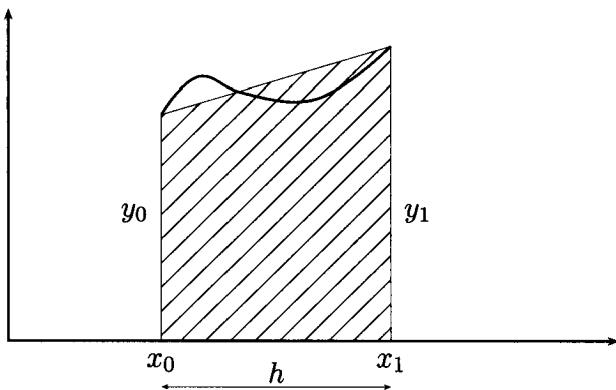


Figura 14.3

Ejemplo 14.1.

$$\int_0^{0.2} e^x dx \approx 0.2 \left(\frac{1}{2}e^0 + \frac{1}{2}e^{0.2} \right) = 0.22214028. \quad \diamond$$

Aplicando la fórmula del trapecio a cada uno de los intervalos $[x_{i-1}, x_i]$ se tiene:

$$\int_{x_0}^{x_1} f(x)dx \approx h\left(\frac{y_0}{2} + \frac{y_1}{2}\right),$$

$$\int_{x_1}^{x_2} f(x)dx \approx h\left(\frac{y_1}{2} + \frac{y_2}{2}\right),$$

$$\vdots \qquad \approx \qquad \vdots$$

$$\int_{x_{n-1}}^{x_n} f(x)dx \approx h\left(\frac{y_{n-1}}{2} + \frac{y_n}{2}\right).$$

$$\int_{x_0}^{x_n} f(x)dx \approx h\left(\frac{y_0}{2} + \frac{y_1}{2} + \frac{y_2}{2} + \frac{y_3}{2} + \cdots + \frac{y_{n-1}}{2} + \frac{y_n}{2}\right),$$

$$\int_{x_0}^{x_n} f(x)dx \approx h\left(\frac{y_0}{2} + y_1 + y_2 + \cdots + y_{n-2} + y_{n-1} + \frac{y_n}{2}\right), \quad (14.2)$$

$$\int_{x_0}^{x_n} f(x)dx \approx h\left(\frac{y_0}{2} + \sum_{i=1}^{n-1} y_i + \frac{y_n}{2}\right).$$

Ejemplo 14.2.

$$\int_0^{0.8} e^x dx \approx 0.2\left(\frac{1}{2}e^0 + e^{0.2} + e^{0.4} + e^{0.6} + \frac{1}{2}e^{0.8}\right) = 1.22962334. \quad \diamond$$

14.2.1 Errores local y global

El error local de la fórmula del trapecio es el error proveniente de la fórmula (14.1).

$$\begin{aligned} e_{\text{loc}} &= I_{\text{loc}} - \tilde{I}_{\text{loc}}, \\ e_{\text{loc}} &= \int_{x_0}^{x_1} f(x)dx - h\left(\frac{y_0}{2} + \frac{y_1}{2}\right) \\ &= \int_{x_0}^{x_1} f(x)dx - \int_{x_0}^{x_1} p_1(x)dx \\ &= \int_{x_0}^{x_1} (f(x) - p_1(x))dx. \end{aligned}$$

Utilizando la fórmula del error para la interpolación polinómica 13.6,

$$e_{\text{loc}} = \int_{x_0}^{x_1} \frac{(x - x_0)(x - x_1)}{2} f''(\xi_x)dx, \quad \xi_x \in [x_0, x_1].$$

El teorema del valor medio para integrales dice: *Sean f continua en $[a, b]$, g integrable en $[a, b]$, g no cambia de signo en $[a, b]$, entonces*

$$\int_a^b f(x)g(x)dx = f(c) \int_a^b g(x)dx$$

para algún c en $[a, b]$.

Teniendo en cuenta que $(x - x_0)(x - x_1) \leq 0$ en el intervalo $[x_0, x_1]$ y aplicando el teorema del valor medio para integrales, existe $z \in [x_0, x_1]$ tal que

$$e_{\text{loc}} = \frac{f''(z)}{2} \int_{x_0}^{x_1} (x - x_0)(x - x_1)dx, \quad z \in [x_0, x_1].$$

Mediante el cambio de variable $t = x - x_0$, $dt = dx$,

$$\begin{aligned} e_{\text{loc}} &= \frac{f''(z)}{2} \int_0^h t(t - h)dt, \quad z \in [x_0, x_1], \\ &= \frac{f''(z)}{2} \left(-\frac{h^3}{6}\right), \quad z \in [x_0, x_1], \\ e_{\text{loc}} &= -h^3 \frac{f''(z)}{12}, \quad z \in [x_0, x_1]. \end{aligned} \tag{14.3}$$

La fórmula anterior, como muchas de las fórmulas de error, sirve principalmente para obtener cotas del error cometido.

$$|e_{\text{loc}}| \leq \frac{h^3}{12} M, \quad M = \max\{|f''(z)| : z \in [x_0, x_1]\}. \quad (14.4)$$

En el ejemplo 14.1, $f''(x) = e^x$, $\max\{|f''(z)| : z \in [0, 0.2]\} = 1.22140276$, luego el máximo error que se puede cometer, en valor absoluto, es $(0.2)^3 \times 1.22140276/12 = 8.1427 \cdot 10^{-4}$. En este ejemplo, se conoce el valor exacto $I = e^{0.2} - 1 = 0.22140276$, luego $|e| = 7.3752 \cdot 10^{-4}$.

En algunos casos, la fórmula del error permite afinar un poco más. Si $f''(x) > 0$ (f estrictamente convexa) en $[x_0, x_1]$ y como $I = \tilde{I} + e_{\text{loc}}$, entonces la fórmula del trapecio da un valor aproximado pero superior al exacto.

En el mismo ejemplo, $f''(x)$ varía en el intervalo $[1, 1.22140276]$ cuando $x \in [0, 0.2]$. Luego

$$e_{\text{loc}} \in [-0.00081427, -0.00066667],$$

entonces

$$I \in [0.22132601, 0.22147361].$$

El error global es el error correspondiente al hacer la aproximación de la integral sobre todo el intervalo $[x_0, x_n]$, o sea, el error en la fórmula 14.2,

$$\begin{aligned} e_{\text{glob}} &= \int_{x_0}^{x_n} f(x) dx - h\left(\frac{y_0}{2} + y_1 + y_2 + \cdots + y_{n-2} + y_{n-1} + \frac{y_n}{2}\right) \\ &= \sum_{i=1}^n \left(-\frac{f''(z_i) h^3}{12}\right), \quad z_i \in [x_{i-1}, x_i] \\ &= -\frac{h^3}{12} \sum_{i=1}^n f''(z_i), \quad z_i \in [x_{i-1}, x_i] \end{aligned}$$

Sean

$$M_1 = \min\{f''(x) : x \in [a, b]\}, \quad M_2 = \max\{f''(x) : x \in [a, b]\}.$$

Entonces

$$\begin{aligned} M_1 &\leq f''(z_i) \leq M_2, \quad \forall i \\ nM_1 &\leq \sum_{i=1}^n f''(z_i) \leq nM_2, \\ M_1 &\leq \frac{1}{n} \sum_{i=1}^n f''(z_i) \leq M_2. \end{aligned}$$

Si $f \in C_{[a,b]}^2$, entonces, aplicando el teorema del valor intermedio a f'' , existe $\xi \in [a, b]$ tal que

$$f''(\xi) = \frac{1}{n} \sum_{i=1}^n f''(z_i).$$

Entonces

$$e_{\text{glob}} = -\frac{h^3}{12} n f''(\xi), \quad \xi \in [a, b].$$

Como $h = (b - a)/n$, entonces $n = (b - a)/h$.

$$e_{\text{glob}} = -h^2 \frac{(b - a)f''(\xi)}{12}, \quad \xi \in [a, b]. \quad (14.5)$$

14.3 Fórmula de Simpson

Es la fórmula de Newton-Cotes para $m = 2$,

$$\int_{x_0}^{x_2} f(x)dx \approx \int_{x_0}^{x_2} p_2(x)dx.$$

El polinomio de interpolación $p_2(x)$ se construye a partir de los puntos (x_0, y_0) , (x_1, y_1) , (x_2, y_2) . Para facilitar la deducción de la fórmula, supongamos que p_2 es el polinomio de interpolación que pasa por los puntos $(0, y_0)$, (h, y_1) , $(2h, y_2)$. Entonces

$$\begin{aligned} p_2(x) &= y_0 \frac{(x - h)(x - 2h)}{(0 - h)(0 - 2h)} + y_1 \frac{(x - 0)(x - 2h)}{(h - 0)(h - 2h)} + y_2 \frac{(x - 0)(x - h)}{(2h - 0)(2h - h)}, \\ &= \frac{1}{2h^2} (y_0(x - h)(x - 2h) - 2y_1 x(x - 2h) + y_2 x(x - h)), \\ &= \frac{1}{2h^2} (x^2(y_0 - 2y_1 + y_2) + hx(-3y_0 + 4y_1 - y_2) + 2h^2 y_0), \end{aligned}$$

$$\int_0^{2h} p_2(x)dx = \frac{1}{2h^2} \left(\frac{8h^3}{3}(y_0 - 2y_1 + y_2) + h \frac{4h^2}{2}(-3y_0 + 4y_1 - y_2) + 2h^2(2h)y_0 \right),$$

$$\int_0^{2h} p_2(x)dx = h\left(\frac{1}{3}y_0 + \frac{4}{3}y_1 + \frac{1}{3}y_2\right).$$

Entonces

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}(y_0 + 4y_1 + y_2) \quad (14.6)$$

Suponiendo que n es par, al aplicar la fórmula anterior a cada uno de los intervalos $[x_0, x_2]$, $[x_2, x_4]$, $[x_4, x_6]$, ..., $[x_{n-4}, x_{n-2}]$, $[x_{n-2}, x_n]$, se tiene:

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + \cdots + 4y_{n-1} + y_n) \quad (14.7)$$

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{3}\left(y_0 + 4 \sum_{j=1}^k y_{2j-1} + 2 \sum_{j=1}^{k-1} y_{2j} + y_n\right)$$

Ejemplo 14.3.

$$\int_0^{0.8} e^x dx \approx \frac{0.2}{3}(e^0 + 4(e^{0.2} + e^{0.6}) + 2e^{0.4} + e^{0.8}) = 1.22555177.$$

El valor exacto, con 8 cifras decimales, es 1.22554093, entonces el error es -0.00001084 . \diamond

14.3.1 Errores local y global

Para facilitar la deducción del error local, consideremos la integral entre $-h$ y h . Sea $f \in C_{[-h,h]}^4$.

$$\begin{aligned} e(h) &= e_{\text{loc}}(h) = \int_{-h}^h f(x) dx - \int_{-h}^h p_2(x) dx, \\ &= \int_{-h}^h f(x) dx - \frac{h}{3} (f(-h) + 4f(0) + f(h)). \end{aligned}$$

Sea F tal que $F'(x) = f(x)$, entonces $\int_{-h}^h f(x) dx = F(h) - F(-h)$. Al derivar con respecto a h se tiene $f(h) + f(-h)$.

$$e'(h) = f(h) + f(-h) - \frac{1}{3}(f(-h) + 4f(0) + f(h))$$

$$- \frac{h}{3}(-f'(-h) + f'(h)),$$

$$3e'(h) = 2f(h) + 2f(-h) - 4f(0) - h(f'(h) - f'(-h)).$$

$$3e''(h) = 2f'(h) - 2f'(-h) - f'(h) + f'(-h) - h(f''(h) + f''(-h)), \\ = f'(h) - f'(-h) - h(f''(h) + f''(-h)).$$

$$3e'''(h) = f''(h) + f''(-h) - (f''(h) + f''(-h)) - h(f'''(h) - f'''(-h)), \\ = -h(f'''(h) - f'''(-h)),$$

$$e'''(h) = -\frac{h}{3}(f'''(h) - f'''(-h)),$$

$$e'''(h) = -\frac{2h^2}{3} \frac{f'''(h) - f'''(-h)}{2h}.$$

De los resultados anteriores se ve claramente que $e(0) = e'(0) = e''(0) = e'''(0) = 0$. Además, como $f \in C^4$, entonces $f''' \in C^1$. Por el teorema del valor medio, existe $\beta \in [-h, h]$, $\beta = \alpha h$, $\alpha \in [-1, 1]$, tal que

$$\frac{f'''(h) - f'''(-h)}{2h} = f^{(4)}(\alpha h), \quad \alpha \in [-1, 1].$$

Entonces

$$e'''(h) = -\frac{2h^2}{3} f^{(4)}(\alpha h), \quad \alpha \in [-1, 1].$$

Sea

$$g_4(h) = f^{(4)}(\alpha h).$$

$$e'''(h) = -\frac{2h^2}{3} g_4(h).$$

$$e''(h) = \int_0^h e'''(t) dt + e''(0),$$

$$e''(h) = -\frac{2}{3} \int_0^h t^2 g_4(t) dt.$$

Como g_4 es continua, t^2 es integrable y no cambia de signo en $[0, h]$, se puede aplicar el teorema del valor medio para integrales,

$$\begin{aligned} e''(h) &= -\frac{2}{3} g_4(\xi_4) \int_0^h t^2 dt, \quad \xi_4 \in [0, h], \\ e''(h) &= -\frac{2}{9} h^3 g_4(\xi_4). \end{aligned}$$

Sea

$$g_3(h) = g_4(\xi_4) = f^{(4)}(\theta_3 h), \quad -1 \leq \theta_3 \leq 1,$$

entonces

$$e''(h) = -\frac{2}{9} h^3 g_3(h).$$

De manera semejante,

$$\begin{aligned} e'(h) &= \int_0^h e''(t) dt + e'(0), \\ e'(h) &= -\frac{2}{9} \int_0^h t^3 g_3(t) dt, \\ e'(h) &= -\frac{2}{9} g_3(\xi_3) \int_0^h t^3 dt, \quad \xi_3 \in [0, h], \\ e'(h) &= -\frac{1}{18} h^4 g_3(\xi_3). \end{aligned}$$

Sea

$$\begin{aligned} g_2(h) &= g_3(\xi_3) = f^{(4)}(\theta_2 h), \quad -1 \leq \theta_2 \leq 1, \\ e'(h) &= -\frac{1}{18} h^4 g_2(h). \end{aligned}$$

$$\begin{aligned}
e(h) &= \int_0^h e'(t) dt + e(0), \\
e(h) &= -\frac{1}{18} \int_0^h t^4 g_2(t) dt, \\
e(h) &= -\frac{1}{18} g_2(\xi_2) \int_0^h t^4 dt, \quad \xi_2 \in [0, h], \\
e(h) &= -\frac{1}{90} h^5 g_2(\xi_2), \\
e(h) &= -\frac{h^5}{90} f^{(4)}(\theta_1 h), \quad -1 \leq \theta_1 \leq 1, \\
e(h) &= -\frac{h^5}{90} f^{(4)}(z), \quad -h \leq z \leq h.
\end{aligned}$$

Volviendo al intervalo $[x_0, x_2]$,

$$e_{\text{loc}} = -h^5 \frac{f^{(4)}(z)}{90}, \quad z \in [x_0, x_2]. \quad (14.8)$$

La deducción del error global se hace de manera semejante al error global en la fórmula del trapecio. Sean $n = 2k$, $M_1 = \min\{f^{(4)}(x) : x \in [a, b]\}$, $M_2 = \max\{f^{(4)}(x) : x \in [a, b]\}$.

$$\begin{aligned}
e_{\text{glob}} &= \int_a^b f(x) dx - \left(\frac{h}{3} (y_0 + 4 \sum_{j=1}^k y_{2j-1} + 2 \sum_{j=1}^{k-1} y_{2j} + y_n) \right), \\
&= \sum_{j=1}^k \left(-h^5 \frac{f^{(4)}(z_j)}{90} \right), \quad z_j \in [x_{2j-2}, x_{2j}], \\
&= -\frac{h^5}{90} \sum_{j=1}^k f^{(4)}(z_j)
\end{aligned}$$

$$M_1 \leq f^{(4)}(z_j) \leq M_2, \quad \forall j$$

$$kM_1 \leq \sum_{j=1}^k f^{(4)}(z_j) \leq kM_2,$$

$$M_1 \leq \frac{1}{k} \sum_{j=1}^k f^{(4)}(z_j) \leq M_2,$$

Entonces, existe $\xi \in [a, b]$, tal que

$$\begin{aligned} \frac{1}{k} \sum_{j=1}^k f^{(4)}(z_j) &= f^{(4)}(\xi), \\ \sum_{j=1}^k f^{(4)}(z_j) &= k f^{(4)}(\xi), \\ \sum_{j=1}^k f^{(4)}(z_j) &= \frac{n}{2} f^{(4)}(\xi), \\ \sum_{j=1}^k f^{(4)}(z_j) &= \frac{b-a}{2h} f^{(4)}(\xi). \end{aligned}$$

Entonces

$$e_{\text{glob}} = -h^4 \frac{(b-a)f^{(4)}(\xi)}{180}, \quad \xi \in [a, b]. \quad (14.9)$$

La fórmula de Simpson es exacta para polinomios de grado inferior o igual a 3. El error global es del orden de h^4 .

Pasando de una interpolación lineal (fórmula del trapecio) a una interpolación cuadrática (fórmula de Simpson), el error global pasa de $O(h^2)$ a $O(h^4)$, es decir, una mejora notable. Se puede ver que al utilizar interpolación cúbica se obtiene

$$\int_{x_0}^{x_3} f(x) dx = \frac{h}{8} (3y_0 + 9y_1 + 9y_2 + 3y_3) - \frac{3}{80} h^5 f^{(4)}(z), \quad z \in [x_0, x_3],$$

llamada segunda fórmula de Simpson. Entonces el error local es $O(h^5)$ y el error global es $O(h^4)$. La fórmula anterior es exacta para polinomios de grado inferior o igual a 3. En resumen, la interpolación cúbica no mejora la calidad de la aproximación numérica, luego es preferible utilizar la fórmula (14.7), más sencilla y de calidad semejante.

Sin embargo, cuando se tiene una tabla fija con un número impar de subintervalos (n impar, número par de puntos), se puede aplicar la (primera) fórmula de Simpson sobre el intervalo $[x_0, x_{n-3}]$ y la segunda fórmula sobre el intervalo $[x_{n-3}, x_n]$.

14.4 Otras fórmulas de Newton-Cotes

Las fórmulas de Newton-Cotes se pueden clasificar en abiertas y cerradas. Las fórmulas del trapecio y de Simpson son casos particulares de las fórmulas cerradas. En ellas se aproxima la integral en el intervalo $[x_0, x_m]$ usando el polinomio de interpolación, de grado menor o igual a m , construido a partir de los puntos $(x_0, y_0), (x_1, y_1), \dots, (x_{m-1}, y_{m-1}), (x_m, y_m)$, igualmente espaciados en x .

$$\int_{x_0}^{x_m} f(x)dx \approx \int_{x_0}^{x_m} p_m(x)dx.$$

La siguiente tabla muestra las más importantes.

m		error
1	$\frac{h}{2}(y_0 + y_1)$	$-\frac{f''(z)}{12} h^3$
2	$\frac{h}{3}(y_0 + 4y_1 + y_2)$	$-\frac{f^{(4)}(z)}{90} h^5$
3	$\frac{3h}{8}(y_0 + 3y_1 + 3y_2 + y_3)$	$-\frac{3f^{(4)}(z)}{80} h^5$
4	$\frac{2h}{45}(7y_0 + 32y_1 + 12y_2 + 32y_3 + 7y_4)$	$-\frac{8f^{(6)}(z)}{945} h^7$

En todos los casos, $z \in [x_0, x_m]$.

14.4.1 Fórmulas de Newton-Cotes abiertas

En estas fórmulas el polinomio de interpolación se calcula sin utilizar los extremos del intervalo de integración,

$$\int_{x_0}^{x_{m+2}} f(x)dx \approx \int_{x_0}^{x_{m+2}} p_m(x)dx,$$

donde p_m , polinomio de grado menor o igual a m , se construye utilizando los puntos $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m), (x_{m+1}, y_{m+1})$, igualmente

espaciados en x .

m		error
0	$2h y_1$	$+\frac{f''(z)}{3} h^3$
1	$\frac{3h}{2}(y_1 + y_2)$	$+\frac{3f''(z)}{4} h^3$
2	$\frac{4h}{3}(2y_1 - y_2 + 2y_3)$	$+\frac{14f^{(4)}(z)}{45} h^5$
3	$\frac{5h}{24}(11y_1 + y_2 + y_3 + 11y_4)$	$+\frac{95f^{(4)}(z)}{144} h^5$

En todos los casos $z \in [x_0, x_{m+2}]$.

Ejemplo 14.4.

$$\int_0^{0.8} e^x dx \approx \frac{4 \times 0.2}{3} (2e^{0.2} - e^{0.4} + 2e^{0.6}) = 1.22539158.$$

El valor exacto, con 8 cifras decimales, es 1.22554093, entonces el error es 0.00014935. ◇

En general, las fórmulas cerradas son más precisas que las abiertas, entonces, siempre que se pueda, es preferible utilizar las fórmulas cerradas. Las fórmulas abiertas se usan cuando no se conoce el valor de la función f en los extremos del intervalo de integración; por ejemplo, en la solución numérica de algunas ecuaciones diferenciales ordinarias.

14.5 Cuadratura de Gauss

En las diferentes fórmulas de Newton-Cotes, los valores x_i deben estar igualmente espaciados. Esto se presenta con frecuencia cuando se dispone de una tabla de valores $(x_i, f(x_i))$. En la cuadratura de Gauss se calcula la integral en un intervalo fijo $[-1, 1]$ mediante valores precisos pero no igualmente espaciados. Es decir, no se debe disponer de una tabla de valores, sino que debe ser posible evaluar la función en valores específicos.

La fórmula de cuadratura de Gauss tiene la forma

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i). \quad (14.10)$$

Los valores w_i se llaman los pesos o ponderaciones y los x_i son las abscisas. Si se desea integrar en otro intervalo,

$$\int_a^b \varphi(\xi) d\xi$$

es necesario hacer un cambio de variable,

$$t = \frac{2}{b-a}(\xi - a) - 1, \quad \xi = \frac{b-a}{2}(t+1) + a, \quad d\xi = \frac{b-a}{2}dt$$

$$\begin{aligned} \int_a^b \varphi(\xi) d\xi &= \frac{b-a}{2} \int_{-1}^1 \varphi\left(\frac{b-a}{2}(t+1) + a\right) dt, \\ \int_a^b \varphi(\xi) d\xi &\approx \frac{b-a}{2} \sum_{i=1}^n w_i \varphi\left(\frac{b-a}{2}(x_i + 1) + a\right), \end{aligned} \quad (14.11)$$

$$\int_a^b \varphi(\xi) d\xi \approx \frac{b-a}{2} \sum_{i=1}^n w_i \varphi(\xi_i), \quad (14.12)$$

$$\xi_i = \frac{b-a}{2}(x_i + 1) + a. \quad (14.13)$$

En la cuadratura de Gauss se desea que la fórmula (14.10) sea exacta para los polinomios de grado menor o igual que $m = m_n$, y se desea que este valor m_n sea lo más grande posible. En particular,

$$\int_{-1}^1 f(x) dx = \sum_{i=1}^n w_i f(x_i), \quad \text{si } f(x) = 1, x, x^2, \dots, x^{m_n}.$$

La anterior igualdad da lugar a $m_n + 1$ ecuaciones con $2n$ incógnitas (los w_i y los x_i). De donde $m_n = 2n - 1$, es decir, la fórmula (14.10) debe ser exacta para polinomios de grado menor o igual a $2n - 1$.

Recordemos que

$$\int_{-1}^1 x^k dx = \begin{cases} 0 & \text{si } k \text{ es impar,} \\ \frac{2}{k+1} & \text{si } k \text{ es par.} \end{cases}$$

Para $n = 1$, se debe cumplir

$$w_1 = \int_{-1}^1 1 dx = 2,$$

$$w_1 x_1 = \int_{-1}^1 x dx = 0.$$

Se deduce inmediatamente que

$$w_1 = 2, \quad x_1 = 0. \quad (14.14)$$

Para $n \geq 2$, se puede suponer, sin perder generalidad, que hay simetría en los valores x_i y en los pesos w_i . Más específicamente, se puede suponer que:

$$x_1 < x_2 < \dots < x_n,$$

$$x_i = -x_{n+1-i},$$

$$w_i = w_{n+1-i}.$$

Para $n = 2$,

$$w_1 + w_2 = \int_{-1}^1 1 dx = 2,$$

$$w_1 x_1 + w_2 x_2 = \int_{-1}^1 x dx = 0,$$

$$w_1 x_1^2 + w_2 x_2^2 = \int_{-1}^1 x^2 dx = \frac{2}{3},$$

$$w_1 x_1^3 + w_2 x_2^3 = \int_{-1}^1 x^3 dx = 0.$$

Por suposiciones de simetría,

$$x_1 < 0 < x_2,$$

$$x_1 = -x_2,$$

$$w_1 = w_2.$$

Entonces

$$2w_1 = 2,$$

$$2w_1 x_1^2 = \frac{2}{3}.$$

Finalmente,

$$w_1 = 1, x_1 = -\sqrt{\frac{1}{3}},$$

$$w_2 = 1, x_2 = \sqrt{\frac{1}{3}}.$$

Para $n = 3$,

$$w_1 + w_2 + w_3 = 2,$$

$$w_1 x_1 + w_2 x_2 + w_3 x_3 = 0,$$

$$w_1 x_1^2 + w_2 x_2^2 + w_3 x_3^2 = \frac{2}{3},$$

$$w_1 x_1^3 + w_2 x_2^3 + w_3 x_3^3 = 0,$$

$$w_1 x_1^4 + w_2 x_2^4 + w_3 x_3^4 = \frac{2}{5},$$

$$w_1 x_1^5 + w_2 x_2^5 + w_3 x_3^5 = 0.$$

Por suposiciones de simetría,

$$x_1 < 0 = x_2 < x_3,$$

$$x_1 = -x_3,$$

$$w_1 = w_3.$$

Entonces

$$2w_1 + w_2 = 2,$$

$$2w_1 x_1^2 = \frac{2}{3},$$

$$2w_1 x_1^4 = \frac{2}{5}.$$

Finalmente,

$$w_1 = \frac{5}{9}, x_1 = -\sqrt{\frac{3}{5}},$$

$$w_2 = \frac{8}{9}, x_2 = 0,$$

$$w_3 = \frac{5}{9}, x_3 = \sqrt{\frac{3}{5}}.$$

La siguiente tabla contiene los valores w_i , x_i , para valores de n menores o iguales a 4.

n	w_i	x_i
1	2	0
2	1	± 0.577350269189626
3	0.555555555555555 0.888888888888889	± 0.774596669241483 0
4	0.347854845137454 0.652145154862546	± 0.861136311594053 ± 0.339981043584856

Tablas más completas se pueden encontrar en [Fro70] o en [AbS74].

Ejemplo 14.5. Calcular una aproximación de

$$\int_{0.2}^{0.8} e^x dx$$

por cuadratura de Gauss con $n = 3$.

$$\xi_1 = \frac{0.8 - 0.2}{2}(-0.774596669241483 + 1) + 0.2 = 0.26762099922756$$

$$\xi_2 = \frac{0.8 - 0.2}{2}(0 + 1) + 0.2 = 0.5$$

$$\xi_3 = \frac{0.8 - 0.2}{2}(0.774596669241483 + 1) + 0.2 = 0.73237900077244$$

$$\begin{aligned} \int_{0.2}^{0.8} e^x dx &\approx \frac{0.8 - 0.2}{2} \left(\frac{5}{9} e^{\xi_1} + \frac{8}{9} e^{\xi_2} + \frac{5}{9} e^{\xi_3} \right) \\ &\approx 1.00413814737559 \end{aligned}$$

El valor exacto es $e^{0.8} - e^{0.2} = 1.00413817033230$, entonces el error es $0.00000002295671 \approx 2.3 \cdot 10^{-8}$. Si se emplea la fórmula de Simpson, que también utiliza tres evaluaciones de la función, se tiene

$$\int_{0.2}^{0.8} e^x dx \approx \frac{0.3}{3} (e^{0.2} + 4e^{0.5} + e^{0.8}) = 1.00418287694532$$

El error es $-0.00004470661302 \approx 4.5 \cdot 10^{-5}$. \diamond

La fórmula del error para 14.10 es:

$$e_n = \frac{2^{2n+1}(n!)^4}{(2n+1)((2n)!)^3} f^{(2n)}(\xi), \quad -1 < \xi < 1. \quad (14.15)$$

Para 14.12 el error está dado por:

$$e_n = \frac{(b-a)^{2n+1}(n!)^4}{(2n+1)((2n)!)^3} f^{(2n)}(\xi), \quad a < \xi < b. \quad (14.16)$$

Comparemos el método de Simpson y la fórmula de cuadratura de Gauss con $n = 3$, para integrar en el intervalo $[a, b]$, con $h = (b-a)/2$. En los dos casos es necesario evaluar tres veces la función.

$$\begin{aligned} e_{\text{Simpson}} &= -\frac{h^5}{90} f^{(4)}(z), \\ e_{\text{Gauss3}} &= \frac{(2h)^7(3!)^4}{7(6!)^3} f^{(6)}(\xi) = \frac{h^7}{15750} f^{(6)}(\xi). \end{aligned}$$

Se observa que mientras que la fórmula de Simpson es exacta para polinomios de grado menor o igual a 3, la fórmula de Gauss es exacta hasta para polinomios de grado 5. Sea $0 < h < 1$. No sólo $h^7 < h^5$, sino que el coeficiente $1/15750$ es mucho menor que $1/90$.

En el ejemplo anterior, $h = 0.3$, y tanto $f^{(4)}$ como $f^{(6)}$ varían en el intervalo $[1.22, 2.23]$.

$$\begin{aligned} e_{\text{Simpson}} &= -2.7 \cdot 10^{-5} f^{(4)}(z), \\ e_{\text{Gauss3}} &= 1.39 \cdot 10^{-8} f^{(6)}(\xi). \end{aligned}$$

14.5.1 Polinomios de Legendre

Las fórmulas de cuadratura vistas son las fórmulas de Gauss-Legendre. En ellas están involucrados los polinomios ortogonales de Legendre. También hay cuadratura de Gauss-Laguerre, de Gauss-Hermite y de Gauss-Chebyshev, relacionadas con los polinomios de Laguerre, de Hermite y de Chebyshev.

Hay varias maneras de definir los polinomios de Legendre; una de ellas es:

$$P_0(x) = 1, \quad (14.17)$$

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n. \quad (14.18)$$

Por ejemplo,

$$P_0(x) = 1,$$

$$P_1(x) = x,$$

$$P_2(x) = \frac{1}{2}(3x^2 - 1),$$

$$P_3(x) = \frac{1}{2}(5x^3 - x),$$

$$P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3).$$

También existe una expresión recursiva:

$$P_0(x) = 1, \tag{14.19}$$

$$P_1(x) = x, \tag{14.20}$$

$$P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x). \tag{14.21}$$

Algunas de las propiedades de los polinomios de Legendre son:

- $\int_{-1}^1 x^k P_n(x) dx = 0, \quad k = 0, 1, 2, \dots, n-1,$ (14.22)

- $\int_{-1}^1 P_m(x) P_n(x) dx = 0, \quad m \neq n,$ (14.23)

- $\int_{-1}^1 (P_n(x))^2 dx = \frac{2}{2n+1}.$ (14.24)

Las abscisas de las fórmulas de cuadratura de Gauss-Legendre son exactamente las raíces de $P_n(x)$. Además,

- $w_i = \frac{1}{P'_n(x_i)} \int_{-1}^1 \frac{P_n(x)}{x - x_i} dx,$ (14.25)

- $w_i = \frac{1}{(P'_n(x_i))^2} \frac{2}{1 - x_i^2}.$ (14.26)

14.6 Derivación numérica

Dados los puntos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ igualmente espaciados en x , o sea, $x_i = x_0 + ih$, se desea tener aproximaciones de $f'(x_i)$ y $f''(x_i)$.

Como se vio anteriormente (13.6),

$$f(x) = p_n(x) + (x - x_0)(x - x_1) \cdots (x - x_n)f^{(n+1)}(\xi)/(n+1)!.$$

Sea $\Phi(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$. Como ξ depende de x , se puede considerar $F(x) = f^{(n+1)}(\xi(x))/(n+1)!$. Entonces

$$\begin{aligned} f(x) &= p_n(x) + \Phi(x)F(x) \\ f'(x) &= p'_n(x) + \Phi'(x)F(x) + \Phi(x)F'(x), \\ f'(x_i) &= p'_n(x_i) + \Phi'(x_i)F(x_i) + \Phi(x_i)F'(x_i), \\ f'(x_i) &= p'_n(x_i) + \Phi'(x_i)F(x_i). \end{aligned}$$

Para $n = 1$

$$p_1(x) = y_0 + \frac{(y_1 - y_0)}{h}(x - x_0), \quad p'_1(x) = \frac{(y_1 - y_0)}{h}.$$

$$\Phi(x) = (x - x_0)(x - x_1), \quad \Phi'(x) = 2x - 2x_0 - h$$

Entonces

$$f'(x_0) = \frac{(y_1 - y_0)}{h} + (2x_0 - 2x_0 - h)F(x_0) = \frac{(y_1 - y_0)}{h} - \frac{h}{2}f''(\xi(x_0)),$$

$$f'(x_1) = \frac{(y_1 - y_0)}{h} + (2x_1 - 2x_0 - h)F(x_1) = \frac{(y_1 - y_0)}{h} + \frac{h}{2}f''(\xi(x_1)).$$

En general,

$$f'(x_i) = \frac{(y_{i+1} - y_i)}{h} - \frac{h}{2}f''(\xi), \quad \xi \in [x_i, x_{i+1}] \quad (14.27)$$

$$f'(x_i) = \frac{(y_i - y_{i-1})}{h} + \frac{h}{2}f''(\zeta), \quad \zeta \in [x_{i-1}, x_i] \quad (14.28)$$

El primer término después del signo igual corresponde al valor aproximado. El segundo término es el error. Se acostumbra decir que el error es del orden de h . Esto se escribe

$$f'(x_i) = \frac{(y_{i+1} - y_i)}{h} + O(h),$$

$$f'(x_i) = \frac{(y_i - y_{i-1})}{h} + O(h).$$

Para $n = 2$, sea $s = (x - x_0)/h$,

$$\begin{aligned} p_2(x) &= y_0 + s\Delta f_0 + \frac{s(s-1)}{2} \frac{\Delta^2 f_0}{2}, \\ p_2(x) &= y_0 + \frac{x-x_0}{h} \Delta f_0 + \frac{x-x_0}{h} \frac{x-x_0-h}{h} \frac{\Delta^2 f_0}{2}, \\ p'_2(x) &= \frac{\Delta f_0}{h} + \frac{2x-2x_0-h}{h^2} \frac{\Delta^2 f_0}{2}, \\ p'_2(x_1) &= \frac{\Delta f_0}{h} + \frac{\Delta^2 f_0}{2h} = \dots \\ p'_2(x_1) &= \frac{y_2-y_0}{2h}. \end{aligned}$$

$$\Phi(x) = (x - x_0)(x - x_0 - h)(x - x_0 - 2h),$$

$$\Phi(x) = (x - x_0)^3 - 3h(x - x_0)^2 + 2h^2(x - x_0),$$

$$\Phi'(x) = 3(x - x_0)^2 - 6h(x - x_0) + 2h^2,$$

$$\Phi'(x_1) = 3h^2 - 6h^2 + 2h^2 = -h^2.$$

Entonces

$$f'(x_1) = \frac{y_2 - y_0}{2h} - \frac{h^2}{6} f'''(\xi), \quad \xi \in [x_0, x_2].$$

De manera general,

$$\begin{aligned} f'(x_i) &= \frac{y_{i+1} - y_{i-1}}{2h} - \frac{h^2}{6} f'''(\xi), \quad \xi \in [x_{i-1}, x_{i+1}], \\ f'(x_i) &= \frac{y_{i+1} - y_{i-1}}{2h} + O(h^2). \end{aligned} \tag{14.29}$$

En [YoG72], página 357, hay una tabla con varias fórmulas para diferenciación numérica. Para la segunda derivada, una fórmula muy empleada es:

$$\begin{aligned} f''(x_i) &= \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} - \frac{h^2}{12} f^{(4)}(\xi), \quad \xi \in [x_{i-1}, x_{i+1}], \\ f''(x_i) &= \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + O(h^2). \end{aligned} \tag{14.30}$$

La deducción de las fórmulas de derivación numérica se hizo a partir de una tabla de valores (x_i, y_i) , pero para el uso de éstas solamente se requiere conocer o poder evaluar f en los puntos necesarios. Por esta

razón, algunas veces las fórmulas aparecen directamente en función de h :

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h),$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2),$$

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2).$$

Ejemplo 14.6. Dada $f(x) = \sqrt{x}$, evaluar aproximadamente $f'(4)$ y $f''(4)$, utilizando $h = 0.2$.

$$f'(4) \approx \frac{2.0494 - 2}{0.2} = 0.2470$$

$$f'(4) \approx \frac{2 - 1.9494}{0.2} = 0.2532$$

$$f'(4) \approx \frac{2.0494 - 1.9494}{2 \times 0.2} = 0.2501$$

$$f''(4) \approx \frac{2.0494 - 2 \times 2 + 1.9494}{0.2^2} = -0.0313. \diamond$$

El error de las dos primeras aproximaciones no es el mismo, pero es del mismo orden de magnitud $O(h)$. La tercera aproximación es mejor que las anteriores; su error es del orden de $O(h^2)$. Los valores exactos son $f'(4) = 0.25$, $f''(4) = -0.03125$.

Ejercicios

14.1 Calcule

$$\int_{0.2}^1 e^x dx$$

utilizando la fórmula del trapecio y de Simpson, variando el número de subintervalos. También por medio de la cuadratura de Gauss variando el número puntos. Calcule los errores. Compare.

14.2 Calcule

$$\int_0^1 e^{-x^2} dx$$

utilizando la fórmula de Simpson. Utilice seis cifras decimales. Tome los valores $n = 2, 4, 8, 16, 32\dots$ hasta que no haya variación.

- 14.3** Haga un programa para calcular $\int_a^b f(x)dx$, siguiendo el esquema del ejercicio anterior.
- 14.4** Observe, por ejemplo, que para $n = 2$ se evalúa la función en a , $(a + b)/2$, b . Para $n = 4$ se evalúa la función en a , $a + (b - a)/4$, $(a + b)/2$, $a + 3(b - a)/4$, b . Haga el programa eficiente para que no evalúe la función dos veces en el mismo punto.
- 14.5** Haga un programa para calcular $\int_a^b f(x)dx$, partiendo $[a, b]$ en subintervalos y utilizando en cada subintervalo cuadratura de Gauss.

- 14.6** Considere los puntos

$$\begin{aligned} &(0.05, 2.0513), \\ &(0.10, 2.1052), \\ &(0.15, 2.1618), \\ &(0.20, 2.2214), \\ &(0.25, 2.2840), \\ &(0.30, 2.3499), \\ &(0.35, 2.4191), \\ &(0.40, 2.4918). \end{aligned}$$

Calcule de la mejor manera posible

$$\int_{0.05}^{0.35} f(x)dx, \quad \int_{0.05}^{0.40} f(x)dx, \quad \int_{0.05}^{0.45} f(x)dx.$$

- 14.7** Considere los mismos puntos del ejercicio anterior. Calcule una aproximación de $f'(0.25)$, $f'(0.225)$, $f''(0.30)$.
- 14.8** Combine integración numérica y solución de ecuaciones para resolver

$$\int_0^x e^{-t^2} dt = 0.1.$$

15

Ecuaciones diferenciales

Este capítulo se refiere únicamente a ecuaciones diferenciales **ordinarias**. Generalmente una ecuación diferencial ordinaria de primer orden con condiciones iniciales, EDO1CI, se escribe de la forma

$$\begin{aligned}y' &= f(x, y) \text{ para } a \leq x \leq b, \\y(x_0) &= y_0.\end{aligned}\tag{15.1}$$

Frecuentemente la condición inicial está dada sobre el extremo izquierdo del intervalo, o sea, $a = x_0$. Un ejemplo de EDO1CI es:

$$\begin{aligned}y' &= \frac{xy}{1 + x^2 + y^2} + 3x^2, \quad x \in [2, 4], \\y(2) &= 5.\end{aligned}$$

Temas importantísimos como existencia de la solución, unicidad o estabilidad, no serán tratados en este texto. El lector deberá remitirse a un libro de ecuaciones diferenciales. Aquí se supondrá que las funciones satisfacen todas las condiciones necesarias (continuidad, diferenciabilidad, condición de Lipschitz...) para que la solución exista, sea única...

Como en todos los otros casos de métodos numéricos, la primera opción para resolver una EDO1CI es buscar la solución analítica. Si esto no se logra, entonces se busca la solución numérica que consiste en encontrar valores aproximados y_1, y_2, \dots, y_n tales que

$$y_i \approx y(x_i), \quad i = 1, \dots, n, \quad \text{donde } a = x_0 < x_1 < x_2 < \dots < x_n = b.$$

En muchos casos los valores x_i están igualmente espaciados, o sea,

$$x_i = a + ih, \quad i = 0, 1, \dots, n, \quad \text{con } h = \frac{b - a}{n}.$$

En varios de los ejemplos siguientes se aplicarán los métodos numéricos para ecuaciones diferenciales con solución analítica conocida. Esto se hace simplemente para comparar la solución numérica con la solución exacta.

15.1 Método de Euler

Se aplica a una EDO1CI como en (15.1) utilizando puntos igualmente espaciados. Su deducción es muy sencilla.

$$y'(x_0) \approx \frac{y(x_0 + h) - y(x_0)}{h}.$$

Por otro lado

$$y'(x_0) = f(x_0, y_0).$$

Entonces

$$y(x_0 + h) \approx y_0 + hf(x_0, y_0).$$

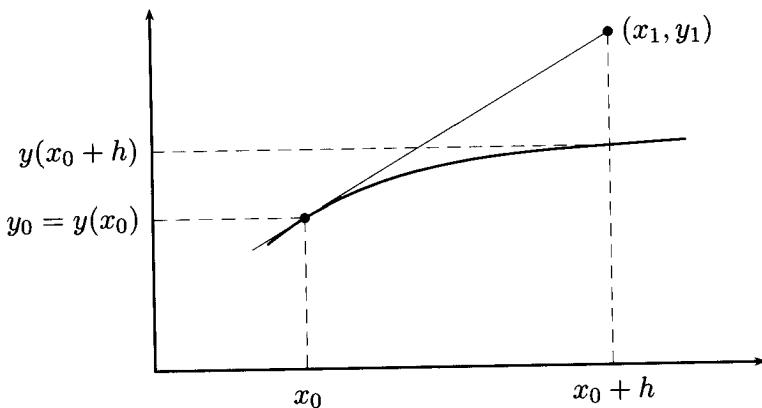
Si denotamos por y_1 la aproximación de $y(x_0 + h)$, entonces la fórmula del método de Euler es justamente

$$y_1 = y_0 + hf(x_0, y_0).$$

Aplicando varias veces el mismo tipo de aproximaciones, se tiene la fórmula general:

$$y_{i+1} = y_i + hf(x_i, y_i). \tag{15.2}$$

Gráficamente esto significa que $y(x_i + h) = y(x_{i+1})$ se aproxima por el valor obtenido a partir de la recta tangente a la curva en el punto (x_i, y_i) .

Figura 15.1 *Método de Euler.*

El valor y_1 es una aproximación de $y(x_1)$. A partir de y_1 , no de $y(x_1)$, se hace una aproximación de $y'(x_1)$. Es decir, al suponer que y_2 es una aproximación de $y(x_2)$, se han hecho dos aproximaciones consecutivas y el error pudo haberse acumulado. De manera análoga, para decir que y_3 es una aproximación de $y(x_3)$, se han hecho tres aproximaciones, una sobre otra.

Sea $\varphi(t, h)$ definida para $t_1 \leq t \leq t_2$ y para valores pequeños de h . Se dice que

$$\varphi(t, h) = O(h^p)$$

si para valores pequeños de h existe una constante c tal que

$$|\varphi(t, h)| \leq ch^p, \quad \forall t \in [t_1, t_2].$$

También se acostumbra decir que

$$\varphi(t, h) \approx ch^p.$$

El error local tiene que ver con el error cometido para calcular $y(x_{i+1})$ suponiendo que y_i es un valor exacto, es decir, $y_i = y(x_i)$. El error global es el error que hay al considerar y_n como aproximación de $y(x_n)$ (n indica el número de intervalos).

Los resultados sobre el error en el método de Euler son:

$$y_1 = y(x_1) + O(h^2) \tag{15.3}$$

$$y_n = y(x_n) + O(h). \tag{15.4}$$

Ejemplo 15.1. Resolver, por el método de Euler, la ecuación diferencial

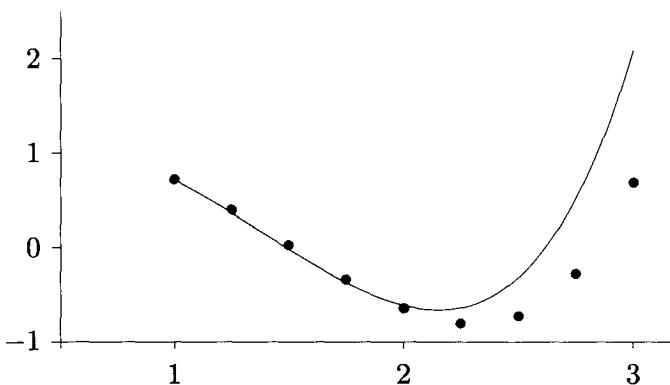
$$\begin{aligned}y' &= 2x^2 - 4x + y \\y(1) &= 0.7182818\end{aligned}$$

en el intervalo $[1, 3]$, con $h = 0.25$.

La primera observación es que esta ecuación diferencial se puede resolver analíticamente. Su solución es $y = e^x - 2x^2$. Luego no debería ser resuelta numéricamente. Sin embargo, el hecho de conocer su solución exacta permite ver el error cometido por el método numérico.

$$\begin{aligned}y_1 &= y_0 + hf(x_0, y_0) \\&= 0.7182818 + 0.25f(1, 0.7182818) \\&= 0.7182818 + 0.25(0.7182818 + 2 \times 1^2 - 4 \times 1) \\&= 0.3978523 \\y_2 &= y_1 + hf(x_1, y_1) \\&= 0.3978523 + 0.25f(1.25, 0.3978523) \\&= 0.3978523 + 0.25(0.3978523 + 2 \times 1.25^2 - 4 \times 1.25) \\&= 0.0285654 \\y_3 &= \dots\end{aligned}$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
1.00	0.7182818	0.7182818
1.25	0.3978523	0.3653430
1.50	0.0285654	-0.0183109
1.75	-0.3392933	-0.3703973
2.00	-0.6428666	-0.6109439
2.25	-0.8035833	-0.6372642
2.50	-0.7232291	-0.3175060
2.75	-0.2790364	0.5176319
3.00	0.6824545	2.0855369

Figura 15.2 *Método de Euler.*

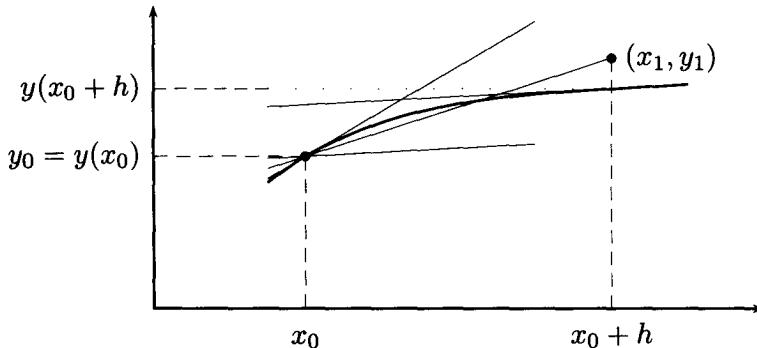
En los primeros valores se observa que el error es muy pequeño. A partir de $x = 2$ se empiezan a distanciar los valores $\tilde{y}(x)$ y $y(x)$. Si se trabaja con $h = 0.1$ se obtiene $\tilde{y}(3) = 1.4327409$; con $h = 0.01$ se obtiene $\tilde{y}(3) = 2.0133187$; con $h = 0.001$ se obtiene $\tilde{y}(3) = 2.0782381$. ◇

15.2 Método de Heun

Este método es una modificación o mejora del método de Euler y se utiliza para el mismo tipo de problemas. También se conoce con el nombre de método del trapecio. En el método de Euler se utiliza la aproximación

$$y(x + h) = y(x) + hy'(x).$$

En el método de Heun se busca cambiar, en la aproximación anterior, la derivada en x por un promedio de la derivada en x y en $x + h$.

Figura 15.3 *Método de Heun.*

$$y(x+h) \approx y(x) + h \frac{y'(x) + y'(x+h)}{2}$$

o sea,

$$y(x+h) \approx y(x) + h \frac{f(x, y(x)) + f(x+h, y(x+h))}{2}.$$

La fórmula anterior no se puede aplicar. Sirve para aproximar $y(x+h)$ pero utiliza $y(x+h)$. Entonces, en el lado derecho, se reemplaza $y(x+h)$ por la aproximación dada por el método de Euler

$$y(x+h) \approx y(x) + h \frac{f(x, y(x)) + f(x+h, y(x) + hf(x, y(x)))}{2}.$$

La anterior aproximación suele escribirse de la siguiente manera:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ K_2 &= hf(x_i + h, y_i + K_1) \\ y_{i+1} &= y_i + \frac{1}{2}(K_1 + K_2). \end{aligned} \tag{15.5}$$

Ejemplo 15.2. Resolver, por el método de Heun, la ecuación diferencial

$$\begin{aligned} y' &= 2x^2 - 4x + y \\ y(1) &= 0.7182818 \end{aligned}$$

en el intervalo $[1, 3]$, con $h = 0.25$.

$$\begin{aligned} K_1 &= hf(x_0, y_0) \\ &= 0.25f(1, 0.7182818) \\ &= -0.320430 \end{aligned}$$

$$\begin{aligned} K_2 &= hf(x_0 + h, y_0 + K_1) \\ &= 0.25f(1.25, 0.397852) \\ &= -0.369287 \end{aligned}$$

$$\begin{aligned} y_1 &= y_0 + (K_1 + K_2)/2 \\ &= 0.3734236 \end{aligned}$$

$$\begin{aligned} K_1 &= hf(x_1, y_1) \\ &= 0.25f(1.25, 0.3734236) \\ &= -0.375394 \\ K_2 &= hf(x_1 + h, y_1 + K_1) \\ &= 0.25f(1.5, -0.001971) \\ &= -0.375493 \\ y_2 &= y_1 + (K_1 + K_2)/2 \\ &= -0.0020198 \end{aligned}$$

$$K_1 = \dots$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
1.00	0.7182818	0.7182818
1.25	0.3734236	0.3653430
1.50	-0.0020198	-0.0183109
1.75	-0.3463378	-0.3703973
2.00	-0.5804641	-0.6109439
2.25	-0.6030946	-0.6372642
2.50	-0.2844337	-0.3175060
2.75	0.5418193	0.5176319
3.00	2.0887372	2.0855369

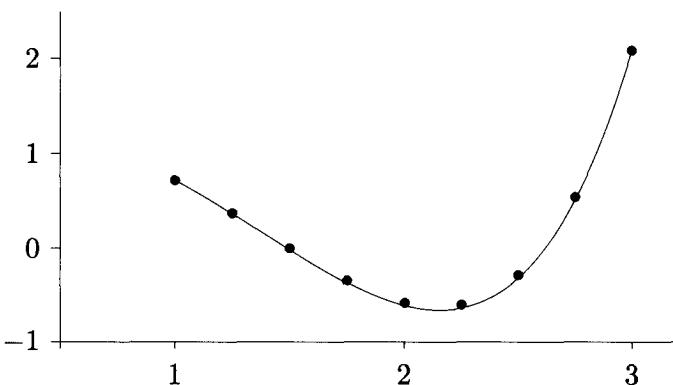


Figura 15.4 Método de Heun.

En este ejemplo los resultados son mucho mejores. Por un lado, el método es mejor, pero, por otro, es natural tener mejores resultados pues hubo que evaluar 16 veces la función $f(x, y)$, 2 veces en cada iteración. En el ejemplo del método de Euler hubo simplemente 8 evaluaciones de la función $f(x, y)$. Al aplicar el método de Heun con $h = 0.5$ (es necesario evaluar 8 veces la función) se obtiene $\tilde{y}(3) = 2.1488885$, resultado no tan bueno como 2.0887372, pero netamente mejor que el obtenido por el método de Euler. Si se trabaja con $h = 0.1$ se obtiene $\tilde{y}(3) = 2.0841331$; con $h = 0.01$ se obtiene $\tilde{y}(3) = 2.0855081$; con $h = 0.001$ se obtiene $\tilde{y}(3) = 2.0855366$. ◇

15.3 Método del punto medio

También este método es una modificación o mejora del método de Euler y se utiliza para el mismo tipo de problemas. En el método de Euler se utiliza la aproximación

$$y(x + h) = y(x) + hy'(x).$$

En el método del punto medio se busca cambiar, en la aproximación anterior, la derivada en x por la derivada en el punto medio entre x y $x + h$, o sea, por la derivada en $x + h/2$.

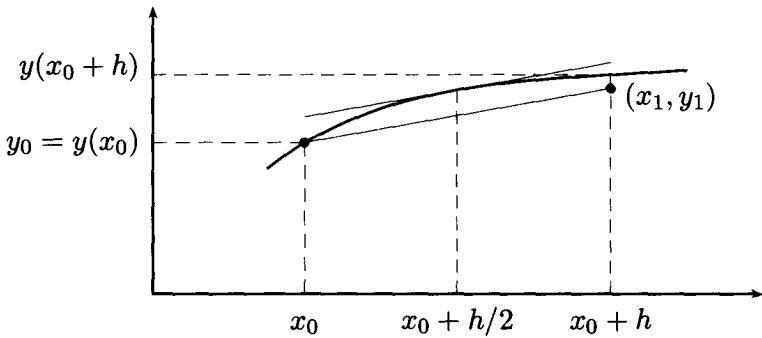


Figura 15.5 Método del punto medio.

$$y(x+h) \approx y(x) + h y'(x+h/2)$$

o sea,

$$y(x+h) \approx y(x) + h f(x+h/2, y(x+h/2)).$$

Como no se conoce $y(x+h/2)$, se reemplaza por la aproximación que daría el método de Euler con un paso de $h/2$.

$$\begin{aligned} y(x+h/2) &\approx y(x) + \frac{h}{2} f(x, y) \\ y(x+h) &\approx y(x) + h f(x+h/2, y(x) + \frac{h}{2} f(x, y)). \end{aligned}$$

La anterior aproximación suele escribirse de la siguiente manera:

$$\begin{aligned} K_1 &= h f(x_i, y_i) \\ K_2 &= h f(x_i + h/2, y_i + K_1/2) \\ y_{i+1} &= y_i + K_2. \end{aligned} \tag{15.6}$$

Ejemplo 15.3. Resolver, por el método del punto medio, la ecuación diferencial

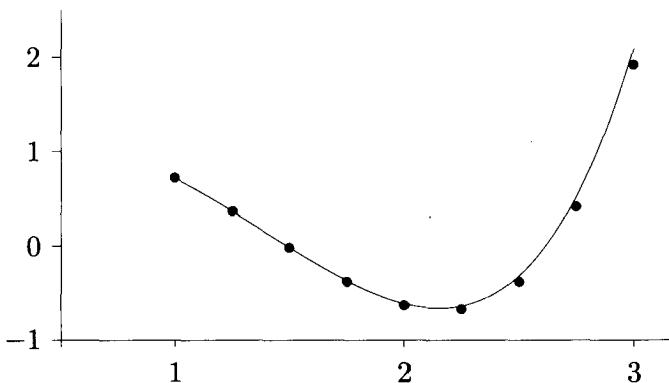
$$\begin{aligned} y' &= 2x^2 - 4x + y \\ y(1) &= 0.7182818 \end{aligned}$$

en el intervalo $[1, 3]$, con $h = 0.25$.

$$\begin{aligned}
 K_1 &= hf(x_0, y_0) \\
 &= 0.25f(1, 0.7182818) \\
 &= -0.320430 \\
 K_2 &= hf(x_0 + h/2, y_0 + K_1/2) \\
 &= 0.25f(1.125, 0.558067) \\
 &= -0.352671 \\
 y_1 &= y_0 + K_2 \\
 &= 0.3656111 \\
 \\
 K_1 &= hf(x_1, y_1) \\
 &= 0.25f(1.25, 0.3656111) \\
 &= -0.377347 \\
 K_2 &= hf(x_1 + h/2, y_1 + K_1/2) \\
 &= 0.25f(1.375, 0.176937) \\
 &= -0.385453 \\
 y_2 &= y_1 + K_2 \\
 &= -0.0198420
 \end{aligned}$$

$$K_1 = \dots$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
1.00	0.7182818	0.7182818
1.25	0.3656111	0.3653430
1.50	-0.0198420	-0.0183109
1.75	-0.3769851	-0.3703973
2.00	-0.6275434	-0.6109439
2.25	-0.6712275	-0.6372642
2.50	-0.3795415	-0.3175060
2.75	0.4121500	0.5176319
3.00	1.9147859	2.0855369

Figura 15.6 *Método del punto medio.*

También, en este ejemplo, los resultados son mucho mejores. De nuevo hubo que evaluar 16 veces la función $f(x, y)$, 2 veces en cada iteración. Al aplicar el método del punto medio con $h = 0.5$ (es necesario evaluar 8 veces la función) se obtiene $\tilde{y}(3) = 1.5515985$, resultado no tan bueno como 2.0887372, pero netamente mejor que el obtenido por el método de Euler. Si se trabaja con $h = 0.1$ se obtiene $\tilde{y}(3) = 2.0538177$; con $h = 0.01$ se obtiene $\tilde{y}(3) = 2.0851903$; con $h = 0.001$ se obtiene $\tilde{y}(3) = 2.0855334$. ◇

15.4 Método de Runge-Kutta

El método de Runge-Kutta o, más bien, los métodos de Runge-Kutta se aplican a una EDO1CI como en (15.1) utilizando puntos igualmente espaciados. La forma general del método RK de orden n es la siguiente:

$$\begin{aligned}
 K_1 &= hf(x_i, y_i) \\
 K_2 &= hf(x_i + \alpha_2 h, y_i + \beta_{21} K_1) \\
 K_3 &= hf(x_i + \alpha_3 h, y_i + \beta_{31} K_1 + \beta_{32} K_2) \\
 &\vdots \\
 K_n &= hf(x_i + \alpha_n h, y_i + \beta_{n1} K_1 + \beta_{n2} K_2 + \cdots + \beta_{n,n-1} K_{n-1}) \\
 y_{i+1} &= y_i + R_1 K_1 + R_2 K_2 + \dots + R_n K_n.
 \end{aligned} \tag{15.7}$$

Se ve claramente que los métodos vistos son de RK: el método de Euler es uno de RK de orden 1, el método de Heun y el del punto medio son métodos de RK de orden 2.

Método de Euler:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ y_{i+1} &= y_i + K_1. \end{aligned}$$

Método de Heun:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ K_2 &= hf(x_i + h, y_i + K_1) \\ y_{i+1} &= y_i + \frac{1}{2}K_1 + \frac{1}{2}K_2. \end{aligned}$$

Método del punto medio:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ K_2 &= hf\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}K_1\right) \\ y_{i+1} &= y_i + 0K_1 + K_2. \end{aligned}$$

Un método muy popular es el siguiente método RK de orden 4:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ K_2 &= hf(x_i + h/2, y_i + K_1/2) \\ K_3 &= hf(x_i + h/2, y_i + K_2/2) \\ K_4 &= hf(x_i + h, y_i + K_3) \\ y_{i+1} &= y_i + (K_1 + 2K_2 + 2K_3 + K_4)/6. \end{aligned} \tag{15.8}$$

Ejemplo 15.4. Resolver, por el método RK4 anterior, la ecuación diferencial

$$\begin{aligned} y' &= 2x^2 - 4x + y \\ y(1) &= 0.7182818 \end{aligned}$$

en el intervalo $[1, 3]$, con $h = 0.25$.

$$\begin{aligned}
 K_1 &= hf(x_0, y_0) \\
 &= 0.25f(1, 0.7182818) \\
 &= -0.320430 \\
 K_2 &= hf(x_0 + h/2, y_0 + K_1/2) \\
 &= 0.25f(1.125, 0.558067) \\
 &= -0.352671 \\
 K_3 &= hf(x_0 + h/2, y_0 + K_2/2) \\
 &= 0.25f(1.125, 0.541946) \\
 &= -0.356701 \\
 K_4 &= hf(x_0 + h, y_0 + K_3) \\
 &= 0.25f(1.25, 0.361581) \\
 &= -0.378355 \\
 y_1 &= y_0 + (K_1 + 2K_2 + 2K_3 + K_4)/6 \\
 &= 0.3653606
 \end{aligned}$$

$$\begin{aligned}
 K_1 &= hf(x_1, y_1) \\
 &= 0.25f(1.25, 0.3653606) \\
 &= -0.377410 \\
 K_2 &= hf(x_1 + h/2, y_1 + K_1/2) \\
 &= 0.25f(1.375, 0.176656) \\
 &= -0.385524
 \end{aligned}$$

$$\begin{aligned}
 K_3 &= hf(x_1 + h/2, y_1 + K_2/2) \\
 &= 0.25f(1.375, 0.172599) \\
 &= -0.386538 \\
 K_4 &= hf(x_1 + h, y_1 + K_3) \\
 &= 0.25f(1.5, -0.02117) \\
 &= -0.380294 \\
 y_2 &= y_1 + (K_1 + 2K_2 + 2K_3 + K_4)/6 \\
 &= -0.0182773
 \end{aligned}$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
1.00	0.7182818	0.7182818
1.25	0.3653606	0.3653430
1.50	-0.0182773	-0.0183109
1.75	-0.3703514	-0.3703973
2.00	-0.6108932	-0.6109439
2.25	-0.6372210	-0.6372642
2.50	-0.3174905	-0.3175060
2.75	0.5175891	0.5176319
3.00	2.0853898	2.0855369

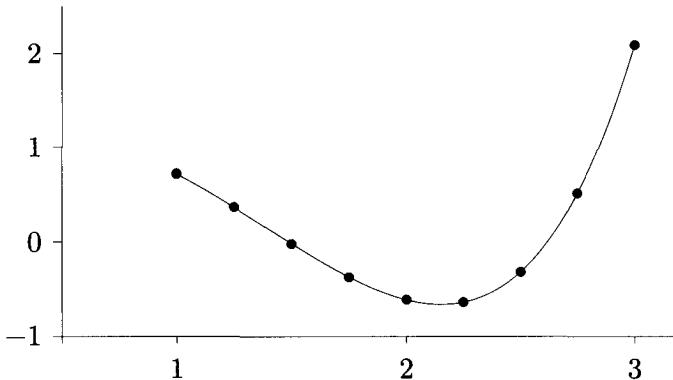


Figura 15.7 Método de Runge-Kutta 4.

En este ejemplo, los resultados son aún mejores. Hubo que evaluar 32 veces la función $f(x, y)$, 4 veces en cada iteración. Si se trabaja con $h = 0.1$ se obtiene $\tilde{y}(3) = 2.0855314$; con $h = 0.01$ se obtiene $\tilde{y}(3) = 2.0855369$; con $h = 0.001$ se obtiene $\tilde{y}(3) = 2.0855369$. \diamond

15.5 Deducción de RK2

En secciones anteriores se hizo la deducción, de manera más o menos intuitiva, de los métodos de Heun y del punto medio. Los dos resultan ser métodos de RK de orden 2. En esta sección veremos una deducción diferente y general de RK2.

El método RK2 tiene el siguiente esquema:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ K_2 &= hf(x_i + \alpha_2 h, y_i + \beta_{21} K_1) \\ y_{i+1} &= y_i + R_1 K_1 + R_2 K_2. \end{aligned}$$

Como hay un solo coeficiente α y un solo coeficiente β , utilicémoslos sin subíndices:

$$\begin{aligned} K_1 &= hf(x_i, y_i) \\ K_2 &= hf(x_i + \alpha h, y_i + \beta K_1) \\ y_{i+1} &= y_i + R_1 K_1 + R_2 K_2. \end{aligned}$$

Sea g una función de dos variables. Si g es diferenciable en el punto (\bar{u}, \bar{v}) , entonces se puede utilizar la siguiente aproximación de primer orden:

$$g(\bar{u} + \Delta u, \bar{v} + \Delta v) \approx g(\bar{u}, \bar{v}) + \Delta u \frac{\partial g}{\partial u}(\bar{u}, \bar{v}) + \Delta v \frac{\partial g}{\partial v}(\bar{u}, \bar{v}). \quad (15.9)$$

La aproximación de segundo orden para $y(x_i + h)$ es:

$$y(x_i + h) = y(x_i) + hy'(x_i) + \frac{h^2}{2}y''(x_i) + O(h^3) \quad (15.10)$$

$$y(x_i + h) \approx y(x_i) + hy'(x_i) + \frac{h^2}{2}y''(x_i). \quad (15.11)$$

En la aproximación anterior, podemos tener en cuenta que $y(x_i) = y_i$, y que $y'(x_i) = f(x_i, y_i)$. Además,

$$\begin{aligned} y''(x_i) &= \frac{d}{dx}y'(x_i) \\ &= \frac{d}{dx}f(x_i, y_i) \\ &= \frac{\partial f}{\partial x}f(x_i, y_i) + \frac{\partial f}{\partial y}f(x_i, y_i)\frac{\partial y}{\partial x}(x_i) \\ &= \frac{\partial f}{\partial x}f(x_i, y_i) + y'(x_i)\frac{\partial f}{\partial y}f(x_i, y_i). \end{aligned}$$

Para acortar la escritura utilizaremos la siguiente notación:

$$\begin{aligned} f &:= f(x_i, y_i) \\ f_x &:= \frac{\partial f}{\partial x} f(x_i, y_i) \\ f_y &:= \frac{\partial f}{\partial y} f(x_i, y_i) \\ y &:= y(x_i) \\ y' &:= y'(x_i) = f(x_i, y_i) = f \\ y'' &:= y''(x_i). \end{aligned}$$

Entonces

$$\begin{aligned} y'' &= f_x + f f_y \\ y(x_i + h) &\approx y + hf + \frac{h^2}{2} f_x + \frac{h^2}{2} f f_y. \end{aligned} \tag{15.12}$$

Por otro lado, el método RK2 se puede reescribir:

$$y_{i+1} = y_i + R_1 h f(x_i, y_i) + R_2 h f(x_i + \alpha h, y_i + \beta K_1).$$

Utilizando (15.9):

$$\begin{aligned} y_{i+1} &= y_i + R_1 h f(x_i, y_i) \\ &+ R_2 h \left(f(x_i, y_i) + \alpha h \frac{\partial f}{\partial x}(x_i, y_i) + \beta K_1 \frac{\partial f}{\partial y}(x_i, y_i) \right). \end{aligned}$$

Utilizando la notación se obtiene:

$$\begin{aligned} y_{i+1} &= y + R_1 h f + R_2 h (f + \alpha h f_x + \beta K_1 f_y) \\ y_{i+1} &= y + (R_1 + R_2) h f + R_2 h^2 \alpha f_x + R_2 h \beta K_1 f_y. \end{aligned}$$

Como $K_1 = hf$, entonces

$$y_{i+1} = y + (R_1 + R_2) h f + R_2 \alpha h^2 f_x + R_2 \beta h^2 f f_y. \tag{15.13}$$

Al hacer la igualdad $y(x_i + h) = y_{i+1}$, en las ecuaciones (15.12) y (15.13)

se comparan los coeficientes de hf , de h^2f_x y de h^2ff_y y se deduce:

$$R_1 + R_2 = 1,$$

$$R_2\alpha = \frac{1}{2},$$

$$R_2\beta = \frac{1}{2}.$$

Entonces

$$\beta = \alpha, \tag{15.14}$$

$$R_2 = \frac{1}{2\alpha}. \tag{15.15}$$

$$R_1 = 1 - R_2. \tag{15.16}$$

Si $\alpha = 1$, entonces $\beta = 1$, $R_2 = 1/2$ y $R_1 = 1/2$, es decir, el método de Heun. Si $\alpha = 1/2$, entonces $\beta = 1/2$, $R_2 = 1$ y $R_1 = 0$, es decir, el método del punto medio. Para otros valores de α se tienen otros métodos de RK de orden 2.

15.6 Control del paso

Hasta ahora se ha supuesto que para hallar la solución numérica de una ecuación diferencial, los puntos están igualmente espaciados, es decir, $x_i - x_{i-1} = h$ para $i = 1, 2, \dots, n$. Esta política no es, en general, adecuada. Es preferible utilizar valores de h pequeños cuando es indispensable para mantener errores relativamente pequeños, y utilizar valores grandes de h cuando se puede.

Hay varios métodos para el control de h . En uno de ellos, se supone conocido y_i , una muy buena aproximación de $y(x_i)$, y se aplica un método con un paso h para obtener \tilde{y} aproximación de $y(x_i + h)$. También se aplica el mismo método dos veces con el paso $h/2$ para obtener $\tilde{\tilde{y}}$, otra aproximación de $y(x_i + h)$. Con estos dos valores se puede acotar el error y así saber si es necesario trabajar con un paso más pequeño.

En otro enfoque, el que veremos en esta sección, se aplican dos métodos diferentes, con el mismo h y con estas dos aproximaciones se acota el error. Así se determina la buena o mala calidad de las aproximaciones.

Supongamos que tenemos dos métodos: el método A con error local $O(h^p)$ y el método B con error local $O(h^{p+1})$ (o con error local $O(h^q)$, $q \geq p + 1$). Partimos de y_i , muy buena aproximación de $y(x_i)$. Aplicando los dos métodos calculamos y_A y y_B , aproximaciones de $y(x_i + h)$. El control de paso tiene dos partes: en la primera se obtiene una aproximación del posible error obtenido.

$$|\text{error}| \approx e = \Phi_1(y_A, y_B, h, p).$$

Si e es menor o igual que un valor ε dado, entonces se acepta y_B como buena aproximación de $y(x + h)$. En caso contrario, es necesario utilizar un valor de h más pequeño. En ambos casos el valor de h se puede modificar, bien sea por necesidad ($e > \varepsilon$), bien sea porque, siendo h aceptable, es conveniente modificarlo para el siguiente paso. Para ello se calcula un coeficiente C_0 que sirve para obtener C coeficiente de h

$$\begin{aligned} C_0 &= \Phi_2(y_A, y_B, h, p) \\ C &= \varphi(C_0, \dots) \\ h' &= Ch. \end{aligned}$$

Los diferentes algoritmos difieren en la manera de calcular e , C_0 y C (las funciones Φ_1 , Φ_2 y φ). Más aún, para el mismo método A y el mismo método B hay diferentes algoritmos.

Un método muy popular es el de **Runge-Kutta-Fehlberg**, construido a partir de un método de RK de orden 5 (el método A) y de un método de RK de orden 6 (el método B). Una de sus ventajas está dada por el siguiente hecho: los valores K_1, K_2, K_3, K_4 y K_5 son los mismos para los dos métodos. Teniendo en cuenta la forma general (15.7) del método RK, basta con dar los valores α_i y β_{ij} . Recuérdese que siempre $K_1 = hf(x_i, y_i)$.

i	α_i	β_{i1}	β_{i2}	...
2	$\frac{1}{4}$	$\frac{1}{4}$		
3	$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$	
4	$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$	$\frac{7296}{2197}$
5	1	$\frac{439}{216}$	-8	$\frac{3680}{513} - \frac{845}{4104}$
6	$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565} \frac{1859}{4104} - \frac{11}{40}$

$$\begin{aligned} y_A &= y_i + \frac{25}{216}K_1 + 0K_2 + \frac{1408}{2565}K_3 + \frac{2197}{4104}K_4 - \frac{1}{5}K_5 \\ y_B &= y_i + \frac{16}{135}K_1 + 0K_2 + \frac{6656}{12825}K_3 + \frac{28561}{56430}K_4 - \frac{9}{50}K_5 + \frac{2}{55}K_6 \end{aligned} \quad (15.17)$$

Los errores locales son respectivamente $O(h^5)$ y $O(h^6)$. Realmente hay varias fórmulas RK5 y RK6; las anteriores están en [BuF85] y [EnU96]. Hay otras fórmulas diferentes en [ChC99].

La aproximación del error está dada por

$$|\text{error}| \approx e = \frac{|y_A - y_B|}{h}. \quad (15.18)$$

El coeficiente para la modificación del valor de h está dado por:

$$\begin{aligned} C_0 &= 0.84 \left(\frac{\varepsilon}{e} \right)^{1/4}, \\ C &= \min\{C_0, 4\}, \\ C &= \max\{C, 0.1\}. \end{aligned} \quad (15.19)$$

Las fórmulas anteriores buscan que C no sea muy grande ni muy pequeño. Más específicamente, C debe estar en el intervalo $[0.1, 4]$.

En la descripción del algoritmo usaremos la siguiente notación de Matlab y de Scilab. La orden

$$\mathbf{u} = [\mathbf{u}; \mathbf{t}]$$

significa que al vector columna \mathbf{u} se le agrega al final el valor \mathbf{t} y el resultado se llama de nuevo \mathbf{u} .

MÉTODO RUNGE-KUTTA-FEHLBERG

```

datos:  $x_0, y_0, b, h_0, \varepsilon, h_{min}$ 
 $x = x_0, y = y_0, h = h_0$ 
 $X = [x_0], Y = [y_0]$ 
mientras  $x < b$ 
     $h = \min\{h, b - x\}$ 
    hbien = 0
    mientras hbien = 0
        calcular  $y_A, y_B$  según (15.17)
         $e = |y_A - y_B|/h$ 
        si  $e \leq \varepsilon$ 
             $x = x + h, y = y_B$ 
            bienh = 1
             $X = [X; x], Y = [Y; y]$ 
        fin-si
         $C_0 = 0.84(\varepsilon/e)^{1/4}$ 
         $C = \max\{C_0, 0.1\}, C = \min\{C, 4\}$ 
         $h = Ch$ 
        si  $h < h_{min}$  ent parar
    fin-mientras
fin-mientras

```

La salida no deseada del algoritmo anterior se produce cuando h se vuelve demasiado pequeño. Esto se produce en problemas muy difíciles cuando, para mantener el posible error dentro de lo establecido, ha sido necesario disminuir mucho el valor de h , por debajo del límite deseado.

En una versión ligeramente más eficiente, inicialmente no se calcula y_A ni y_B . Se calcula directamente

$$e = \left| \frac{1}{360}K_1 - \frac{128}{4275}K_3 - \frac{2197}{75240}K_4 + \frac{1}{50}K_5 + \frac{2}{55}K_6 \right|.$$

Cuando el valor de h es adecuado, entonces se calcula y_B para poder hacer la asignación $y = y_B$.

Ejemplo 15.5. Resolver, por el método RKF con control de paso, la ecuación diferencial

$$\begin{aligned}y' &= 2x^2 - 4x + y \\y(1) &= 0.7182818\end{aligned}$$

en el intervalo $[1, 3]$, con $h_0 = 0.5$ y $\varepsilon = 10^{-6}$.

$$\begin{aligned}y_A &= -0.01834063 \\y_B &= -0.01830704 \\e &= 0.00006717\end{aligned}$$

$h = 0.5$ no sirve.

$$\begin{aligned}C_0 &= 0.29341805 \\C &= 0.29341805 \\h &= 0.14670902 \\y_A &= 0.51793321 \\y_B &= 0.51793329 \\e &= 0.00000057\end{aligned}$$

$h = 0.14670902$ sirve.

$$\begin{aligned}x &= 1.14670902 \\y &= 0.51793329 \\C_0 &= 0.96535578 \\C &= 0.96535578 \\h &= 0.14162640 \\y_A &= 0.30712817 \\y_B &= 0.30712821 \\e &= 0.00000029\end{aligned}$$

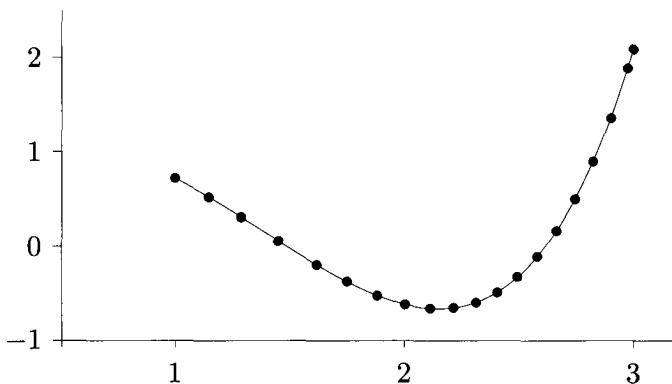
$h = 0.14162640$ sirve.

$$x = 1.28833543$$

$$y = 0.30712821$$

$$\vdots$$

x	h	$\tilde{y}(x)$	$y(x)$
1.0000000	0.1467090	0.7182818	0.7182818
1.1467090	0.1416264	0.5179333	0.5179333
1.2883354	0.1622270	0.3071282	0.3071282
1.4505624	0.1686867	0.0572501	0.0572501
1.6192491	0.1333497	-0.1946380	-0.1946380
1.7525988	0.1329359	-0.3736279	-0.3736279
1.8855347	0.1191306	-0.5206051	-0.5206051
2.0046653	0.1092950	-0.6137572	-0.6137571
2.1139603	0.1024064	-0.6566848	-0.6566847
2.2163666	0.0971218	-0.6506243	-0.6506241
2.3134884	0.0928111	-0.5948276	-0.5948275
2.4062996	0.0891591	-0.4877186	-0.4877184
2.4954587	0.0859853	-0.3273334	-0.3273332
2.5814440	0.0831757	-0.1114979	-0.1114977
2.6646196	0.0806534	0.1620898	0.1620900
2.7452730	0.0783639	0.4958158	0.4958160
2.8236369	0.0762674	0.8921268	0.8921270
2.8999043	0.0743333	1.3535162	1.3535164
2.9742376	0.0257624	1.8825153	1.8825156
3.0000000		2.0855366	2.0855369

Figura 15.8 *Método de Runge-Kutta-Fehlberg.*

15.7 Orden del método y orden del error

Para algunos de los métodos hasta ahora vistos, todos son métodos de RK, se ha hablado del orden del método, del orden del error local y del orden del error global.

El orden del método se refiere al número de evaluaciones de la función f en cada iteración. Así por ejemplo, el método de Euler es un método de orden 1 y el método de Heun es un método de orden 2.

El orden del error local se refiere al exponente de h en el error teórico cometido en cada iteración. Si la fórmula es

$$y(x+h) = y(x) + R_1 k_1 + R_2 K_2 + \cdots + R_n K_n + O(h^p),$$

se dice que el error local es del orden de h^p , o simplemente, el error local es de orden p .

El orden del error global se refiere al exponente de h en el error obtenido al aproximar $y(b)$ después de hacer $(b - x_0)/h$ iteraciones.

Hemos visto seis métodos, Euler, Heun, punto medio, un RK4, un RK5 y un RK6. La siguiente tabla presenta los órdenes de los errores.

Método	Fórmula	Orden del método	Error local
Euler	(15.2)	1	$O(h^2)$
Heun	(15.5)	2	$O(h^3)$
Punto medio	(15.6)	2	$O(h^3)$
RK4	(15.8)	4	$O(h^5)$
RK5	(15.17)	5	$O(h^5)$
RK6	(15.17)	6	$O(h^6)$

El orden del error global es generalmente igual al orden del error local menos una unidad. Por ejemplo, el error global en el método de Euler es $O(h)$.

A medida que aumenta el orden del método, aumenta el orden del error, es decir, el error disminuye. Pero al pasar de RK4 a RK5 el orden del error no mejora. Por eso es más interesante usar el RK4 que el RK5 ya que se hacen solamente 4 evaluaciones y se tiene un error semejante. Ya con RK6 se obtiene un error más pequeño, pero a costa de dos evaluaciones más.

15.7.1 Verificación numérica del orden del error

Cuando se conoce la solución exacta de una ecuación diferencial, en muchos casos, se puede verificar el orden del error de un método específico. Más aún, se podría obtener el orden del error si éste no se conociera.

Sea $O(h^p)$ el error local del método. Se puede hacer la siguiente aproximación:

$$\text{error} = e \approx ch^p.$$

Al tomar logaritmo en la aproximación anterior se obtiene

$$\log(e) \approx \log(c) + p \log(h) \quad (15.20)$$

Para diferentes valores de h se evalúa el error cometido y se obtienen así varios puntos de la forma $(\log(h_i), \log(e_i))$. Estos puntos deben estar, aproximadamente, sobre una recta. La pendiente de esta recta es precisamente p . El valor de p se puede obtener gráficamente o por mínimos cuadrados.

Ejemplo 15.6. Obtener numéricamente el orden del error local del método de Heun usando la ecuación diferencial

$$\begin{aligned}y' &= 2x^2 - 4x + y \\y(1) &= 0.7182818,\end{aligned}$$

con $h = 0.1, 0.12, 0.14, 0.16, 0.18$ y 0.2 .

h	$x_0 + h$	$\tilde{y}(x_0 + h)$	$y(x_0 + h)$	e	$\log(h)$	$\log(e)$
0.10	1.10	0.584701	0.584166	0.000535	-2.302585	-7.532503
0.12	1.12	0.556975	0.556054	0.000921	-2.120264	-6.989970
0.14	1.14	0.529024	0.527568	0.001456	-1.966113	-6.532007
0.16	1.16	0.500897	0.498733	0.002164	-1.832581	-6.135958
0.18	1.18	0.472641	0.469574	0.003067	-1.714798	-5.787212
0.20	1.20	0.444304	0.440117	0.004187	-1.609438	-5.475793

En la siguiente gráfica, $\log(h)$ en las abscisas y $\log(e)$ en las ordenadas, los puntos están aproximadamente en una recta.

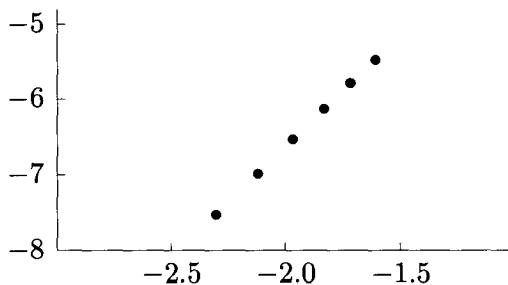


Figura 15.9 *Orden local.*

Al calcular numéricamente los coeficientes de la recta de aproximación por mínimos cuadrados, se obtiene

$$\begin{aligned}\log(e) &\approx 2.967325 \log(h) - 0.698893 \\e &\approx 0.497135h^{2.97}.\end{aligned}$$

Estos resultados numéricos concuerdan con el resultado teórico. ◇

15.8 Métodos multipaso explícitos

Los métodos RK son considerados como métodos monopaso (unipaso) por la siguiente razón. El valor y_{i+1} se calcula únicamente a partir del punto (x_i, y_i) . En los métodos multipaso se utilizan otros puntos anteriores, por ejemplo, para calcular y_{i+1} se utilizan los puntos (x_{i-2}, y_{i-2}) , (x_{i-1}, y_{i-1}) y (x_i, y_i) .

Veamos un caso particular. Supongamos que se conocen los valores $y_0 = y(x_0)$, $y_1 = y(x_1)$ y $y_2 = y(x_2)$. Por facilidad para la deducción, supongamos que $x_0 = h$, $x_1 = h$ y $x_2 = 2h$.

Sea $p_2(x)$ el polinomio de grado menor o igual a 2 que interpola a f en los valores 0 , h y $2h$, es decir, el polinomio pasa por los puntos $(0, f_0)$, (h, f_1) y $(2h, f_2)$, donde $f_i = f(x_i, y_i)$. Este polinomio se puede obtener utilizando polinomios de Lagrange:

$$p_2(x) = f_0 \frac{(x-h)(x-2h)}{(0-h)(0-2h)} + f_1 \frac{(x-0)(x-2h)}{(h-0)(h-2h)} + f_2 \frac{(x-0)(x-h)}{(2h-0)(2h-h)}.$$

Después de algunas factorizaciones se obtiene:

$$p_2(x) = \frac{1}{2h^2} ((f_0 - 2f_1 + f_2)x^2 + (-3f_0 + 4f_1 - f_2)hx + 2h^2 f_0).$$

Por otro lado, por el teorema fundamental del cálculo integral

$$\begin{aligned} \int_{x_2}^{x_3} y'(x)dx &= y(x_3) - y(x_2) \\ y(x_3) &= y(x_2) + \int_{x_2}^{x_3} y'(x)dx \\ y(x_3) &= y(x_2) + \int_{2h}^{3h} f(x, y)dx. \end{aligned}$$

Si se reemplaza $f(x, y)$ por el polinomio de interpolación, se tiene:

$$\begin{aligned}
 y(x_3) &\approx y(x_2) + \int_{2h}^{3h} p_2(x) dx \\
 y(x_3) &\approx y(x_2) + \int_{2h}^{3h} \frac{1}{2h^2} \left((f_0 - 2f_1 + f_2)x^2 + \right. \\
 &\quad \left. (-3f_0 + 4f_1 - f_2)hx + 2h^2 f_0 \right) dx \\
 y_3 &= y_2 + \frac{1}{2h^2} \left((f_0 - 2f_1 + f_2) \frac{19}{3} h^3 + \right. \\
 &\quad \left. (-3f_0 + 4f_1 - f_2) \frac{5}{2} h^3 + 2h^3 f_0 \right) \\
 y_3 &= y_2 + \frac{h}{12} (5f_0 - 16f_1 + 23f_2)
 \end{aligned} \tag{15.21}$$

La anterior igualdad se conoce con el nombre de **Adams-Bashforth de orden 2** (se utiliza un polinomio de orden 2). También recibe el nombre de método multipaso explícito o método multipaso abierto de orden 2.

Si los valores y_0, y_1 y y_2 son exactos, o sea, si $y_0 = y(x_0)$, $y_1 = y(x_1)$ y $y_2 = y(x_2)$, entonces los valores f_i son exactos, o sea, $f(x_i, y_i) = f(x_i, y(x_i))$ y el error está dado por

$$y(x_3) = y(x_2) + \frac{h}{12} (5f_0 - 16f_1 + 23f_2) + \frac{3}{8} y^{(3)}(z) h^4, \quad z \in [x_0, x_3]. \tag{15.22}$$

La fórmula (15.21) se escribe en el caso general

$$y_{i+1} = y_i + \frac{h}{12} (5f_{i-2} - 16f_{i-1} + 23f_i). \tag{15.23}$$

Para empezar a aplicar esta fórmula se requiere conocer los valores f_j anteriores. Entonces es indispensable utilizar un método RK el número de veces necesario. El método RK escogido debe ser de mejor calidad que el método de Adams-Bashforth que estamos utilizando. Para nuestro caso podemos utilizar RK4.

Ejemplo 15.7. Resolver, por el método de Adams-Bashforth de orden

2, la ecuación diferencial

$$\begin{aligned}y' &= 2x^2 - 4x + y \\y(1) &= 0.7182818\end{aligned}$$

en el intervalo $[1, 3]$, con $h = 0.25$.

Al aplicar el método RK4 dos veces se obtiene:

$$\begin{aligned}y_1 &= 0.3653606 \\y_2 &= -0.0182773.\end{aligned}$$

Entonces

$$\begin{aligned}f_0 &= f(x_0, y_0) = -1.2817182 \\f_1 &= f(x_1, y_1) = -1.5096394 \\f_2 &= -1.5182773 \\y_3 &= y_2 + h(5f_0 - 16f_1 + 23f_2)/12 \\&= -0.3760843 \\f_3 &= f(x_3, y_3) = -1.2510843 \\y_4 &= -0.6267238 \\&\vdots\end{aligned}$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
1.00	0.7182818	0.7182818
1.25	0.3653606	0.3653430
1.50	-0.0182773	-0.0183109
1.75	-0.3760843	-0.3703973
2.00	-0.6267238	-0.6109439
2.25	-0.6681548	-0.6372642
2.50	-0.3706632	-0.3175060
2.75	0.4320786	0.5176319
3.00	1.9534879	2.0855369

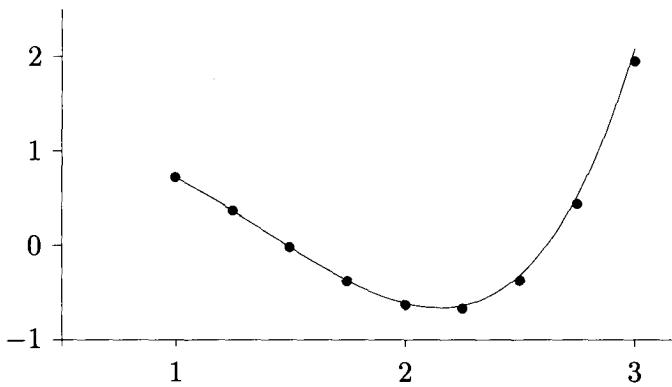


Figura 15.10 Método de Adams-Basforth 2.

En este caso hubo que evaluar 8 veces la función para los dos valores de RK4 y en seguida 6 evaluaciones para un total de 14 evaluaciones de la función f . ◇

MULTIPASO EXPLÍCITO: ADAMS-BASHFORTH

n		error
0	$y_{i+1} = y_i + hf_i$	$\frac{1}{2}y''(\xi)h^2$
1	$y_{i+1} = y_i + \frac{h}{2}(-f_{i-1} + 3f_i)$	$\frac{5}{12}y'''(\xi)h^3$
2	$y_{i+1} = y_i + \frac{h}{12}(5f_{i-2} - 16f_{i-1} + 23f_i)$	$\frac{3}{8}y^{(4)}(\xi)h^4$
3	$y_{i+1} = y_i + \frac{h}{24}(-9f_{i-3} + 37f_{i-2} - 59f_{i-1} + 55f_i)$	$\frac{251}{720}y^{(5)}(\xi)h^5$
4	$y_{i+1} = y_i + \frac{h}{720}(251f_{i-4} - 1274f_{i-3} + 2616f_{i-2} - 2774f_{i-1} + 1901f_i)$	$\frac{95}{288}y^{(6)}(\xi)h^6$

En la anterior tabla se muestran las principales fórmulas. Allí n indica el grado del polinomio de interpolación usado. En algunos libros,

n está asociado con número de puntos utilizados para la interpolación (igual al grado del polinomio más uno). Obsérvese que la primera fórmula es simplemente el método de Euler.

15.9 Métodos multipaso implícitos

En estos métodos se utiliza un polinomio de interpolación, el mismo de los métodos explícitos, pero el intervalo de integración varía.

Veamos un caso particular. Supongamos que se conocen los valores $y_0 = y(x_0)$, $y_1 = y(x_1)$ y $y_2 = y(x_2)$. Por facilidad para la deducción, supongamos que $x_0 = h$, $x_1 = h$ y $x_2 = 2h$.

Sea $p_2(x)$ el polinomio de grado menor o igual a 2 que interpola a f en los valores 0, h y $2h$, es decir, el polinomio pasa por los puntos $(0, f_0)$, (h, f_1) y $(2h, f_2)$, donde $f_i = f(x_i, y_i)$. Como se vio en la sección anterior,

$$p_2(x) = \frac{1}{2h^2} ((f_0 - 2f_1 + f_2)x^2 + (-3f_0 + 4f_1 - f_2)hx + 2h^2 f_0).$$

El teorema fundamental del cálculo integral se usa de la siguiente manera:

$$\begin{aligned} \int_{x_1}^{x_2} y'(x)dx &= y(x_2) - y(x_1) \\ y(x_2) &= y(x_1) + \int_{x_1}^{x_2} y'(x)dx \\ y(x_2) &= y(x_1) + \int_h^{2h} f(x, y)dx. \end{aligned}$$

Si se reemplaza $f(x, y)$ por el polinomio de interpolación se tiene:

$$\begin{aligned} y(x_2) &\approx y(x_1) + \int_h^{2h} p_2(x)dx \\ y(x_2) &\approx y(x_1) + \int_h^{2h} \frac{1}{2h^2} \left((f_0 - 2f_1 + f_2)x^2 + \right. \\ &\quad \left. (-3f_0 + 4f_1 - f_2)hx + 2h^2 f_0 \right) dx \end{aligned}$$

$$\begin{aligned}
 y_2 &= y_1 + \frac{1}{2h^2} \left((f_0 - 2f_1 + f_2) \frac{7}{3} h^3 + \right. \\
 &\quad \left. (-3f_0 + 4f_1 - f_2) \frac{3}{2} h^3 + 2h^3 f_0 \right) \\
 y_2 &= y_1 + \frac{h}{12} (-f_0 + 8f_1 + 5f_2).
 \end{aligned} \tag{15.24}$$

La anterior igualdad se conoce con el nombre de **fórmula de Adams-Moulton de orden 2** (se utiliza un polinomio de orden 2). También recibe el nombre de método multipaso implícito o método multipaso cerrado de orden 2.

Si los valores y_0 , y_1 y y_2 son exactos, o sea, si $y_0 = y(x_0)$, $y_1 = y(x_1)$ y $y_2 = y(x_2)$, entonces los valores f_i son exactos, o sea, $f(x_i, y_i) = f(x_i, y(x_i))$ y el error está dado por

$$y(x_2) = y(x_1) + \frac{h}{12} (-f_0 + 8f_1 + 5f_2) - \frac{1}{24} y^{(3)}(z)h^4, \quad z \in [x_0, x_2]. \tag{15.25}$$

La fórmula (15.24) se escribe en el caso general

$$y_{i+1} = y_i + \frac{h}{12} (-f_{i-1} + 8f_i + 5f_{i+1}). \tag{15.26}$$

Para empezar a aplicar esta fórmula es indispensable conocer los valores f_j anteriores. Entonces se requiere utilizar un método RK el número de veces necesario. El método RK escogido debe ser de mejor calidad que el método de Adams-Bashforth que estamos utilizando. Para nuestro caso podemos utilizar RK4.

Una dificultad más grande, y específica de los métodos implícitos, está dada por el siguiente hecho: para calcular y_{i+1} se utiliza f_{i+1} , pero este valor es justamente $f(x_{i+1}, y_{i+1})$. ¿Cómo salir de este círculo vicioso? Inicialmente se calcula y_{i+1}^0 , una primera aproximación, por el método de Euler. Con este valor se puede calcular $f_{i+1}^0 = f(x_{i+1}, y_{i+1}^0)$ y en seguida y_{i+1}^1 . De nuevo se calcula $f_{i+1}^1 = f(x_{i+1}, y_{i+1}^1)$ y en seguida y_{i+1}^2 . Este proceso iterativo acaba cuando dos valores consecutivos, y_{i+1}^k y y_{i+1}^{k+1} , son muy parecidos. Este método recibe también el nombre de método **predictor-corregidor**. La fórmula queda entonces así:

$$y_{i+1}^{k+1} = y_i + \frac{h}{12} (-f_{i-1} + 8f_i + 5f_{i+1}^k). \tag{15.27}$$

El criterio de parada puede ser:

$$\frac{|y_i^{k+1} - y_i^k|}{\max\{1, |y_i^{k+1}|\}} \leq \varepsilon.$$

Ejemplo 15.8. Resolver, por el método de Adams-Moulton de orden 2, la ecuación diferencial

$$\begin{aligned} y' &= 2x^2 - 4x + y \\ y(1) &= 0.7182818 \end{aligned}$$

en el intervalo $[1, 3]$, con $h = 0.25$ y $\varepsilon = 0.0001$.

Al aplicar el método RK4 una vez, se obtiene:

$$y_1 = 0.3653606$$

Entonces

$$\begin{aligned} f_0 &= f(x_0, y_0) = -1.2817182 \\ f_1 &= f(x_1, y_1) = -1.5096394 \end{aligned}$$

Aplicando Euler se obtiene una primera aproximación de y_2 :

$$\begin{aligned} y_2^0 &= -0.0120493 \\ f_2^0 &= -1.5120493 \end{aligned}$$

Empiezan las iteraciones:

$$\begin{aligned} y_2^1 &= -0.0170487 \\ f_2^1 &= -1.5170487 \\ y_2^2 &= -0.0175694 \\ f_2^2 &= -1.5175694 \\ y_2^3 &= -0.0176237 = y_2 \end{aligned}$$

Para calcular y_2 se utilizan los valores:

$$\begin{aligned} f_1 &= -1.5096394 \\ f_2 &= -1.5176237. \end{aligned}$$

Aplicando Euler se obtiene una primera aproximación de y_3 :

$$\begin{aligned}y_3^0 &= -0.3970296 \\f_3^0 &= -1.2720296\end{aligned}$$

Empiezan las iteraciones:

$$\begin{aligned}y_3^1 &= -0.3716132 \\f_3^1 &= -1.2466132 \\y_3^2 &= -0.3689657 \\f_3^2 &= -1.2439657 \\y_3^3 &= -0.3686899 \\f_3^3 &= -1.2436899 \\y_3^4 &= -0.3686612 = y_3 \\&\vdots\end{aligned}$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
1.00	0.7182818	0.7182818
1.25	0.3653606	0.3653430
1.50	-0.0176237	-0.0183109
1.75	-0.3686612	-0.3703973
2.00	-0.6076225	-0.6109439
2.25	-0.6315876	-0.6372642
2.50	-0.3084043	-0.3175060
2.75	0.5316463	0.5176319
3.00	2.1065205	2.0855369

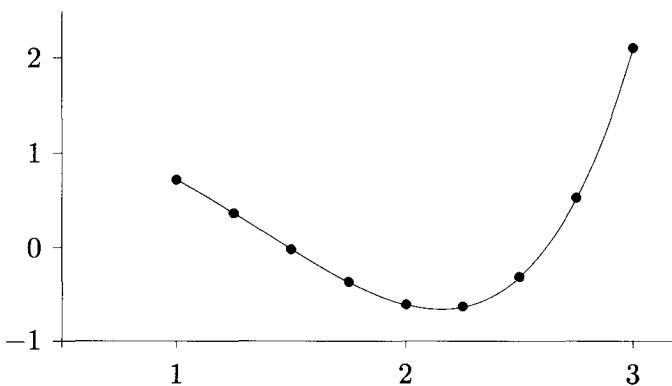


Figura 15.11 Método de Adams-Moulton 2.

En este caso hubo que evaluar 4 veces la función para el valor de RK4 y en seguida, en cada uno de los otros 7 intervalos, una evaluación fija más las requeridas al iterar. En este ejemplo hubo, en promedio, 4 por intervalo, para un total de 32 evaluaciones de f . El valor final y_8 es más exacto que el obtenido por Adams-Bashforth, pero a costa de más evaluaciones. ◇

Teóricamente, los dos métodos multipaso de orden 2 tienen un error local del mismo orden, $O(h^4)$, pero el coeficiente en el método multipaso explícito, $3/8$, es nueve veces el coeficiente en el error del método implícito, $1/24$.

MULTIPASO IMPLÍCITO: ADAMS-MOULTON

n		error
1	$y_{i+1} = y_i + \frac{h}{2}(f_i + f_{i+1})$	$-\frac{1}{12}y''(\xi)h^3$
2	$y_{i+1} = y_i + \frac{h}{12}(-f_{i-1} + 8f_i + 5f_{i+1})$	$-\frac{1}{24}y^{(3)}(\xi)h^4$
3	$y_{i+1} = y_i + \frac{h}{24}(f_{i-2} - 5f_{i-1} + 19f_i + 9f_{i+1})$	$-\frac{19}{720}y^{(4)}(\xi)h^5$
4	$y_{i+1} = y_i + \frac{h}{720}(-19f_{i-3} + 106f_{i-2} - 264f_{i-1} + 646f_i + 251f_{i+1})$	$-\frac{27}{1440}y^{(5)}(\xi)h^6$

La tabla anterior contiene las principales fórmulas. Allí n indica el grado del polinomio de interpolación usado. Obsérvese que el método de Heun corresponde a una iteración (una sola) del método multipaso implícito de orden 1.

15.10 Sistemas de ecuaciones diferenciales

Un sistema de m ecuaciones diferenciales de primer orden se puede escribir de la siguiente forma:

$$\begin{aligned}\frac{dy_1}{dx} &= f_1(x, y_1, y_2, \dots, y_m) \\ \frac{dy_2}{dx} &= f_2(x, y_1, y_2, \dots, y_m) \\ &\vdots \\ \frac{dy_m}{dx} &= f_m(x, y_1, y_2, \dots, y_m)\end{aligned}$$

para $x_0 \leq x \leq b$, con las condiciones iniciales

$$\begin{aligned}y_1(x_0) &= y_1^0 \\ y_2(x_0) &= y_2^0 \\ &\vdots \\ y_m(x_0) &= y_m^0.\end{aligned}$$

Utilicemos la siguiente notación:

$$\begin{aligned}y &= (y_1, y_2, \dots, y_m) \\ y^0 &= (y_1^0, y_2^0, \dots, y_m^0) \\ f(x, y) &= f(x, y_1, y_2, \dots, y_m) \\ &= (f_1(x, y_1, \dots, y_m), f_2(x, y_1, \dots, y_m), \dots, f_m(x, y_1, \dots, y_m)).\end{aligned}$$

De esta manera, el sistema se puede escribir así:

$$\begin{aligned}y' &= f(x, y), \quad x_0 \leq x \leq b \\ y(x_0) &= y^0.\end{aligned}$$

La solución numérica del sistema de ecuaciones consiste en un conjunto de vectores $y^0, y^1, y^2, \dots, y^n$,

$$y^i = (y_1^i, y_2^i, \dots, y_m^i),$$

donde cada y_j^i es una aproximación:

$$y_j^i \approx y_j(x_k).$$

Los métodos vistos anteriormente se pueden generalizar de manera inmediata. Si se trata de los método RK, entonces los K_i dejan de ser números y pasan a ser vectores K^i . Para y se utiliza un superíndice para indicar el intervalo, ya que los subíndices se usan para las componentes del vector. Por ejemplo, las fórmulas de RK4 se convierten en:

$$\begin{aligned} K^1 &= hf(x_i, y^i) \\ K^2 &= hf(x_i + h/2, y^i + K^1/2) \\ K^3 &= hf(x_i + h/2, y^i + K^2/2) \\ K^4 &= hf(x_i + h, y^i + K^3) \\ y^{i+1} &= y^i + (K^1 + 2K^2 + 2K^3 + K^4)/6. \end{aligned} \quad (15.28)$$

Ejemplo 15.9. Resolver el siguiente sistema de ecuaciones por RK4:

$$\begin{aligned} y'_1 &= \frac{2y_1}{x} + x^3y_2, \quad 1 \leq x \leq 2 \\ y'_2 &= -\frac{3}{x}y_2 \\ y_1(1) &= -1 \\ y_2(1) &= 1 \end{aligned}$$

con $h = 0.2$.

La solución (exacta) de este sencillo sistema de ecuaciones es:

$$\begin{aligned} y_1(x) &= -x \\ y_2(x) &= x^{-3}. \end{aligned}$$

Para la solución numérica:

$$K^1 = (-0.2, -0.6)$$

$$K^2 = (-0.2136600, -0.3818182)$$

$$K^3 = (-0.1871036, -0.4413223)$$

$$K^4 = (-0.2026222, -0.2793388)$$

$$y^1 = (-1.2006916, 0.5790634)$$

$$K^1 = (-0.2001062, -0.2895317)$$

$$K^2 = (-0.2093988, -0.2004450)$$

$$K^3 = (-0.1912561, -0.2210035)$$

$$K^4 = (-0.2011961, -0.1534542)$$

$$y^2 = (-1.4011269, 0.3647495)$$

$$\vdots$$

x_i	$\tilde{y}_1(x_i)$	$\tilde{y}_2(x_i)$	$y_1(x_i)$	$y_2(x_i)$
1.0	-1.0	1.0	-1.0	1.0
1.2	-1.2006916	0.5790634	-1.2	0.5787037
1.4	-1.4011269	0.3647495	-1.4	0.3644315
1.6	-1.6014497	0.2443822	-1.6	0.2441406
1.8	-1.8017156	0.1716477	-1.8	0.1714678
2.0	-2.0019491	0.1251354	-2.0	0.125

◇

15.11 Ecuaciones diferenciales de orden superior

Una ecuación diferencial ordinaria, de orden m , con condiciones iniciales, se puede escribir de la siguiente manera:

$$\begin{aligned}
 y^{(m)} &= f(x, y, y', y'', \dots, y^{(m-1)}), \quad x_0 \leq x \leq b \\
 y(x_0) &= y_0 \\
 y'(x_0) &= y'_0 \\
 y''(x_0) &= y''_0 \\
 &\vdots \\
 y^{(m-1)}(x_0) &= y_0^{(m-1)}.
 \end{aligned}$$

Esta ecuación diferencial se puede convertir en un sistema de ecuaciones diferenciales de primer orden, mediante el siguiente cambio de variables:

$$\begin{aligned}
 u_1 &= y \\
 u_2 &= y' \\
 u_3 &= y'' \\
 &\vdots \\
 u_m &= y^{(m-1)}
 \end{aligned}$$

Entonces la ecuación diferencial se convierte en el siguiente sistema:

$$\begin{aligned}
 u'_1 &= u_2 \\
 u'_2 &= u_3 \\
 u'_3 &= u_4 \\
 &\vdots \\
 u'_{m-1} &= u_m \\
 u'_m &= f(x, u_1, u_2, \dots, u_m) \\
 u_1(x_0) &= y_0 \\
 u_2(x_0) &= y'_0 \\
 u_3(x_0) &= y''_0 \\
 &\vdots \\
 u_m(x_0) &= y_0^{(m-1)}.
 \end{aligned}$$

De forma más compacta,

$$\begin{aligned}
 u' &= F(x, u), \quad x_0 \leq x \leq b \\
 u(x_0) &= \kappa_0,
 \end{aligned}$$

donde $\kappa_0 = [y_0 \ y'_0 \ y''_0 \ \dots \ y_0^{(m-1)}]^T$. Este sistema se puede resolver por los métodos para sistemas de ecuaciones diferenciales de primer orden.

Ejemplo 15.10. Resolver la ecuación diferencial

$$y'' = \frac{4y - xy'}{x^2}, \quad 1 \leq x \leq 2,$$

$$y(1) = 3$$

$$y'(1) = 10,$$

por el método RK4, con $h = 0.2$.

Sean $u_1 = y$, $u_2 = y'$.

$$u'_1 = u_2$$

$$u'_2 = \frac{4u_1 - xu_2}{x^2}, \quad 1 \leq x \leq 2,$$

$$u_1(1) = 3$$

$$u_2(1) = 10.$$

La solución exacta es $y = 4x^2 - x^{-2}$. Al aplicar el método RK4 se obtiene:

$$K^1 = (2, 0.4)$$

$$K^2 = (2.04, 0.7900826)$$

$$K^3 = (2.0790083, 0.7678437)$$

$$K^4 = (2.1535687, 1.0270306)$$

$$u^1 = (5.0652642, 10.7571472)$$

$$\vdots$$

x_i	$\tilde{u}_1(x_i)$	$\tilde{u}_2(x_i)$	$y(x_i)$
1.0	3.0	10.0	3.0
1.2	5.0652642	10.757147	5.0655556
1.4	7.3293797	11.928367	7.3297959
1.6	9.8488422	13.287616	9.849375
1.8	12.65069	14.742141	12.651358
2.0	15.749173	16.249097	15.75



15.12 Ecuaciones diferenciales con condiciones de frontera

Una ecuación diferencial de segundo orden con condiciones de frontera se puede escribir de la forma

$$\begin{aligned} y'' &= f(x, y, y'), \quad a \leq x \leq b, \\ y(a) &= y_a \\ y(b) &= y_b. \end{aligned} \tag{15.29}$$

Esta ecuación diferencial se puede convertir en un sistema de dos ecuaciones diferenciales, pero para obtener su solución numérica se presenta un inconveniente: se debería conocer el valor $y'_a = y'(a)$. Esta dificultad se supera mediante el **método del disparo** (*shooting*).

Como no se conoce y'_a , se le asigna un valor aproximado inicial. Puede ser

$$y'_a \approx \frac{y_b - y_a}{b - a}.$$

Con este valor inicial se busca la solución numérica, hasta obtener

$$\tilde{y}(b) = \tilde{y}(b, y'_a).$$

Este valor debería ser el valor conocido y_b . Si no coinciden, es necesario modificar la suposición de y'_a hasta obtener el resultado deseado. Si $\tilde{y}(b, y'_a) < y_b$, entonces se debe aumentar la pendiente inicial del disparo. De manera análoga, si $\tilde{y}(b, y'_a) > y_b$, se debe disminuir la pendiente inicial del disparo. Lo anterior se puede presentar como la solución de una ecuación:

$$\varphi(y'_a) = y_b - \tilde{y}(b, y'_a) = 0.$$

Esta ecuación se puede resolver, entre otros métodos, por el de la secante o el de bisección.

Para facilitar la presentación del método se considera el problema $P(v)$, donde:

v = aproximación de y'_a ,

n = número de intervalos para la solución numérica,

$\tilde{y} = (\tilde{y}_0, \tilde{y}_1, \dots, \tilde{y}_n)$ = solución numérica del siguiente problema:

$$y' = f(x, y), \quad a \leq x \leq b$$

$$y(a) = y_a \quad P(v)$$

$$y'(a) = v,$$

$$\varphi(v) = y_b - \tilde{y}_n = y_b - \tilde{y}(b, v). \quad (15.30)$$

Se desea encontrar v^* tal que $\varphi(v^*) = 0$. Entonces la solución numérica de $P(v^*)$ es la solución numérica de (15.29). Si se aplica el método de la secante para resolver la ecuación $\varphi(v) = 0$, el algoritmo es el siguiente:

MÉTODO DEL DISPARO

datos: $f, a, b, y_a, y_b, \varepsilon, \text{maxit}, \varepsilon_0$ $\varepsilon_r = \max\{1, y_b \} \varepsilon$ $v_0 = (y_b - y_a)/(b - a)$ \tilde{y} = solución numérica de $P(v_0)$ $\varphi_0 = y_b - \tilde{y}_n$ si $ \varphi_0 \leq \varepsilon_r$ ent parar $v_1 = v_0 + \varphi_0/(b - a)$ \tilde{y} = solución numérica de $P(v_1)$ $\varphi_1 = y_b - \tilde{y}_n$ si $ \varphi_1 \leq \varepsilon_r$ ent parar para $k = 1, \dots, \text{maxit}$ $\delta = \varphi_1 - \varphi_0$ si $ \delta \leq \varepsilon_0$ ent parar $v_2 = v_1 - \varphi_1(v_1 - v_0)/\delta$ \tilde{y} = solución numérica de $P(v_2)$ $\varphi_2 = y_b - \tilde{y}_n$ si $ \varphi_2 \leq \varepsilon_r$ ent parar $v_0 = v_1, v_1 = v_2, \varphi_0 = \varphi_1, \varphi_1 = \varphi_2$ fin-paro OJO: no hubo convergencia.

Ejemplo 15.11. Resolver la ecuación diferencial

$$y'' = \frac{2 \cos(2x) - y' - 4x^2y}{x^2}, \quad 0.2 \leq x \leq 0.7$$

$$y(0.2) = 0.3894183$$

$$y(0.7) = 0.9854497,$$

con $h = 0.1$ y utilizando RK4 para la solución del sistema de ecuaciones diferenciales asociado.

La primera aproximación de $y'(a)$ es

$$v_0 = (0.9854497 - 0.3894183)/(0.7 - 0.2) = 1.19206278$$

Al resolver numéricamente el problema P(1.19206278) se obtiene:

$$\tilde{y}_5 = 0.94935663.$$

El disparo resultó muy bajo.

$$\varphi_0 = 0.03609310$$

$$v_1 = 1.19206278 + 0.03609310/(0.7 - 0.5) = 1.26424897$$

Al resolver numéricamente el problema P(1.26424897) se obtiene:

$$\tilde{y}_5 = 0.95337713$$

$$\varphi_1 = 0.03207260$$

Primera iteración del método de la secante:

$$v_2 = 1.84009748$$

Al resolver numéricamente el problema P(1.84009748) se obtiene:

$$\tilde{y}_5 = 0.98544973$$

Este disparo fue preciso (no siempre se obtiene la solución con una sola iteración de la secante). El último vector \tilde{y} es la solución. La solución exacta es $y = \operatorname{sen}(2x)$.

x_i	$\tilde{y}(x_i)$	$y(x_i)$
0.2	0.3894183	0.3894183
0.3	0.5647741	0.5646425
0.4	0.7174439	0.7173561
0.5	0.8415217	0.8414710
0.6	0.9320614	0.9320391
0.7	0.9854497	0.9854497

◇

15.13 Ecuaciones diferenciales lineales con condiciones de frontera

Una ecuación diferencial lineal de segundo orden con condiciones de frontera se puede escribir de la forma

$$\begin{aligned} p(x)y'' + q(x)y' + r(x)y &= s(x), \quad a \leq x \leq b, \\ y(a) &= y_a \\ y(b) &= y_b. \end{aligned} \tag{15.31}$$

Obviamente esta ecuación se puede resolver por el método del disparo, pero, dada la linealidad, se puede resolver usando aproximaciones numéricas (diferencias finitas) para y' y y'' .

El intervalo $[a, b]$ se divide en $n \geq 2$ subintervalos de tamaño $h = (b - a)/n$. Los puntos x_i están igualmente espaciados ($x_i = a + ih$). Se utilizan las siguientes aproximaciones y la siguiente notación:

$$\begin{aligned} y''_i &\approx \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} \\ y'_i &\approx \frac{-y_{i-1} + y_{i+1}}{2h} \\ p_i &:= p(x_i) \\ q_i &:= q(x_i) \\ r_i &:= r(x_i) \\ s_i &:= s(x_i). \end{aligned}$$

Entonces:

$$p_i \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} + q_i \frac{-y_{i-1} + y_{i+1}}{2h} + r_i y_i = s_i, \quad i = 1, \dots, n - 1.$$

Es decir, se tiene un sistema de $n - 1$ ecuaciones con $n - 1$ incógnitas, y_1, y_2, \dots, y_{n-1} .

$$\begin{aligned} 2p_i \frac{y_{i-1} - 2y_i + y_{i+1}}{2h^2} + hq_i \frac{-y_{i-1} + y_{i+1}}{2h^2} + \frac{2h^2 r_i y_i}{2h^2} &= \frac{2h^2 s_i}{2h^2} \\ (2p_i - hq_i)y_{i-1} + (-4p_i + 2h^2 r_i)y_i + (2p_i + hq_i)y_{i+1} &= 2h^2 s_i \end{aligned}$$

Este sistema es tridiagonal.

$$\left[\begin{array}{ccc|ccc} d_1 & u_1 & & & y_1 & \beta_1 \\ l_1 & d_2 & u_2 & & y_2 & \beta_2 \\ & l_2 & d_3 & u_3 & y_3 & \beta_3 \\ & & l_{n-3} & d_{n-2} & u_{n-2} & \beta_{n-2} \\ & & & l_{n-2} & d_{n-1} & \beta_{n-1} \end{array} \right] = \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{array} \right], \quad (15.32)$$

donde

$$\begin{aligned} d_i &= -4p_i + 2h^2 r_i, & i &= 1, \dots, n-1, \\ u_i &= 2p_i + hq_i, & i &= 1, \dots, n-2, \\ l_i &= 2p_{i+1} - hq_{i+1}, & i &= 1, \dots, n-2, \\ \beta_1 &= 2h^2 s_1 - (2p_1 - hq_1)y_a, \\ \beta_i &= 2h^2 s_i, & i &= 2, \dots, n-2, \\ \beta_{n-1} &= 2h^2 s_{n-1} - (2p_{n-1} + hq_{n-1})y_b. \end{aligned}$$

Ejemplo 15.12. Resolver por diferencias finitas la ecuación diferencial

$$\begin{aligned} x^2 y'' + y' + 4x^2 y &= 2 \cos(2x), \quad 0.2 \leq x \leq 0.7 \\ y(0.2) &= 0.3894183 \\ y(0.7) &= 0.9854497, \end{aligned}$$

con $n = 5$, es decir, $h = 0.1$.

Al calcular los coeficientes del sistema tridiagonal se obtiene:

$$d_1 = -4p_1 + 2h^2r_1$$

$$d_1 = -4(0.3)^2 + 2(0.1)^2 4(0.3)^2 = -0.3528$$

$$u_1 = 2p_1 + hq_1$$

$$u_1 = 2(0.3)^2 + 0.1(1) = 0.28$$

$$l_1 = 2p_2 - hq_2$$

$$l_1 = 2(0.4)^2 - 0.1(1) = 0.22$$

$$d = (-0.3528, -0.6272, -0.98, -1.4112),$$

$$u = (0.28, 0.42, 0.6),$$

$$l = (0.22, 0.4, 0.62),$$

$$\beta = (0.00186, 0.0278683, 0.0216121, -0.7935745).$$

Su solución es

$$(y_1, y_2, y_3, y_4) = (0.5628333, 0.7158127, 0.8404825, 0.9315998).$$

x_i	$\tilde{y}(x_i)$	$y(x_i)$
0.2	0.3894183	0.3894183
0.3	0.5628333	0.5646425
0.4	0.7158127	0.7173561
0.5	0.8404825	0.8414710
0.6	0.9315998	0.9320391
0.7	0.9854497	0.9854497

◇

Ejercicios

Escoja varias ecuaciones diferenciales (o sistemas de ecuaciones diferenciales) de las que conozca la solución exacta. Fije el intervalo de trabajo. Determine qué métodos puede utilizar. Aplique varios de ellos. Compare los resultados. Cambie el tamaño del paso. Compare de nuevo.

Un procedimiento adecuado para obtener las ecuaciones diferenciales consiste en partir de la solución (una función cualquiera) y construir la ecuación diferencial.

La aplicación de los métodos se puede hacer de varias maneras: a mano con ayuda de una calculadora; parte a mano y parte con ayuda de software para matemáticas como Scilab o Matlab; haciendo un programa, no necesariamente muy sofisticado, para cada método.

A continuación se presentan algunos ejemplos sencillos.

15.1

$$\begin{aligned}y' &= e^x - \frac{y}{x} \\y(1) &= 0.\end{aligned}$$

Su solución es $y = e^x - \frac{e^x}{x}$.

15.2

$$\begin{aligned}y'_1 &= 2y_1 + y_2 + 3 \\y'_2 &= 4y_1 - y_2 + 9 \\y_1(0) &= -3 \\y_2(0) &= 5.\end{aligned}$$

Su solución es $y_1(t) = -e^{-2t} - 2$, $y_2(t) = 4e^{-2t} + 1$.

15.3

$$\begin{aligned}y'' &= \frac{2}{x(2-x)}y' \\y(1) &= -2 \\y'(1) &= 1.\end{aligned}$$

Su solución es $y = -2\ln(2-x) - x - 1$. Tenga especial cuidado con el intervalo de trabajo.

15.4

$$\begin{aligned}y''' + y'' + y' + y &= 4e^x \\y(0) &= 1 \\y'(0) &= 2 \\y''(0) &= 1 \\y'''(0) &= 0.\end{aligned}$$

Su solución es $y = e^x + \sin(x)$.

15.5

$$y''y = e^{2x} - \operatorname{sen}^2(x)$$

$$y(0) = 1$$

$$y(\pi) = e^\pi.$$

Su solución es $y = e^x + \operatorname{sen}(x)$.

15.6

$$y'' + e^{-x}y' + y = 2e^x + 1 + e^{-x}\cos(x)$$

$$y(0) = 1$$

$$y(\pi) = e^\pi.$$

Su solución es $y = e^x + \operatorname{sen}(x)$.

Bibliografía

- [AbS74] Abramowitz Milton. y Stegun Irene A. (eds.), *Handbook of Mathematical Functions*, Dover, New York, 1974.
- [Atk98] Atkinson Kendall E., *An Introduction to Numerical Analysis*, Wiley, New York, 1978.
- [BaN94] Barton John J. y Nackman Lee R., *Scientific and Engineering C++*, Addison-Wesley, Reading, 1994.
- [Ber01] Berryhill John R., *C++ Scientific Programming*, Wiley, Nueva York, 2001.
- [Bra00] Braquelaire Jean-Pierre, *Méthodologie de la programmation en C*, Dunod, París, 2000.
- [Bro00] Bronson Gary J., *C++ para ingeniería y ciencias*, Int. Thomson, México, 2000.
- [BuF85] Burden Richard L. y Faires J. Douglas, *Numerical Analysis*, 3a. ed., Prindle-Weber-Schmidt, Boston, 1985.
- [Buz93] Buzzi-Ferraris Guido, *Scientific C++, Building Numerical Libraries the Object-Oriented Way*, Addison-Wesley, Wokingham, Inglaterra, 1993.
- [ChC99] Chapra Steven C. y Canale Raymond P., *Métodos Numéricos para Ingenieros*, 3 ed., McGraw-Hill, México, 1999.
- [Cap94] Capper Derek M., *Introducing C++ for Scientists, Engineers and Mathematicians*, Springer-Verlag, Londres, 1994.

- [DaB74] Dahlquist Germund y Björk Ake, *Numerical Methods*, Prentice Hall, Englewood Cliffs, 1974.
- [DeD99] Deitel H.M. y Deitel P.J., *C++, cómo programar*, Prentice Hall Hispanoamericana, México, 1999.
- [ElS90] Ellis Margaret A. y Stroustrup Bjarne, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, 1990.
- [EnU96] Engeln-Muellges Giesela y Uhlig Frank, *Numerical Algorithms with C*, Springer, Nueva York, 1996.
- [Flo95] Flowers B.H., *An Introduction to Numerical Methods in C++*, Clarendon Press, Oxford, 1995.
- [Fro70] Fröberg Karl-Erik, *An Introduction to Numerical Analysis*, 2a. ed. Addison-Wesley, Reading, 1970.
- [Gla93] Glasey Robert, *Numerical Computation Using C*, Academic Press, Boston, 1993.
- [GoV96] Golub Gene H. y Van Loan Charles H., *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, 1996.
- [IsK66] Isaacson Eugene y Keller Herbert B., *Analysis of Numerical Methods*, Wiley, Nueva York, 1966, Dover, Nueva York, 1994.
- [Kem87] Kempf James, *Numerical Software Tools in C*, Prentice Hall, Englewood Cliffs, 1987.
- [LaT87] Lascaux P. y Theodor R., *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, Tomos 1 y 2, Masson, París, 1987.
- [LIH01] Liberty Jesse y Horvath David B., *Aprendiendo C++ para Linux en 21 días*, Prentice Hall, Pearson, México, 2001.
- [Mor01] Mora Héctor, *Optimización no lineal y dinámica*, Departamento de Matemáticas, Universidad Nacional, Bogotá, 1997.
- [Nak93] Nakamura Shoichiro, *Applied Numerical Methods in C*, PTR Prentice Hall, Englewood Cliffs, 1993.
- [NoD88] Noble Ben y Daniel James W., *Applied Linear Algebra* 3rd ed., Prentice Hall, Englewood Cliffs, 1988.

- [OrG98] Ortega James M. y Grimshaw Andrew S., *An Introduction to C++ and Numerical Methods*, Oxford University Press, Oxford, 1998.
- [Pre93] Press Wiliam H. et al., *Numerical Recipes in C, The Art of Scientific Computing*, 2 ed., Cambridge University Press, Cambridge, 1993.
- [Pre02] Press Wiliam H. et al., *Numerical Recipes in C++, The Art of Scientific Computing*, 2 ed., Cambridge University Press, Cambridge, 2002.
- [ReD90] Reverchon Alain y Ducamp Marc, *Analyse numérique en C*, Armand Colin, París, 1990.
- [ReD93] Reverchon Alain y Ducamp Marc, *Mathematical Software Tools in C++*, Wiley, Chichester, 1993.
- [Sch91] Schatzman Michelle, *Analyse numérique, Cours et exercices pour la licence*, InterEditions, París, 1991.
- [Sch92] Schildt Herbert, *Turbo C/C++*, Manual de referencia, McGraw-Hill, Madrid, 1993.
- [StB93] Stoer J. y Bulirsch R., *Introduction to Numerical Analysis*, 2a. ed., Springer-Verlag, Nueva York, 1993.
- [Str02] Stroustrup Bjarne, *El lenguaje de programación C++*, ed. especial, Addison Wesley, Madrid, 2002.
- [Str86] Strang Gilbert, *Álgebra lineal y sus aplicaciones*, Addison-Wesley Iberoamericana, Wilmington, 1986.
- [Yan01] Yang Daoqi, *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*, Springer, Nueva York, 2001.
- [YoG72] Young David M. y Gregory Rober T., *A Survey of Numerical Mathematics*, vols. I y II, Addison-Wesley, Reading, 1974.

Apéndice A

Estilo en C

Para que la lectura de un programa en C sea más fácil para el mismo programador o para otros lectores, es conveniente tener ciertas reglas sobre la manera de escribir, de hacer sangrías, de separar las funciones, etc. Estas reglas no son absolutamente fijas y cada programador establece su propio estilo. Hay unos estilos que son más claros y dicientes que otros. Algunos son ventajosos en ciertos aspectos y desventajosos en otros. Otros son diferentes pero comparables. Algunas “reglas” son más universales y populares que otras.

Las siguientes normas sirven de modelo o referencia y fueron utilizadas en la elaboración de los ejemplos en C de este libro.

A.1 Generalidades

- El corchete { de inicio del cuerpo de una función debe ir en la primera columna y no debe haber nada más en la línea.

El corchete } de finalización de una función debe ir en la primera columna y no debe haber nada más en la línea. Si se desea escribir un comentario, éste se puede colocar en la línea anterior. Por ejemplo:

```
double factorial( ... )  
{
```

```
...
```

```

...
// fin de factorial
}

```

- El corchete de inicio de un bloque dentro de una estructura de control debe ir en la misma línea de la condición.
- En la línea de corchete correspondiente al final de un bloque en una estructura de control no debe haber nada más, salvo un comentario indicando qué tipo de estructura finaliza o cuál variable de control se usó, cuando la estructura de control tiene muchas líneas. Por ejemplo:

```

...
for( i = ... ){
    ...
    while( ... ){
        ...
        ...
        for( j = ... ){
            ...
            }
        ...
        ...
    } // fin while
    ...
    ...
} // fin i

```

- No escribir más allá de la columna 72 (excepcionalmente, hasta la 80).
- Las estructuras de control anidadas dentro de otras deben incrementar la sangría. Cada vez que hay incremento en la sangría, ésta debe estar dada siempre por un número fijo de columnas a lo largo de todo el programa. Este valor fijo puede ser 2, 3 o 4. Es preferible hacerlo con espacios y no con el tabulador.

- Las declaraciones de variables van al principio de las funciones¹.
- Antes de estas declaraciones puede haber solamente comentarios.
- Dejar una interlínea después de las declaraciones de variables.
- Utilizar una línea para cada una de las variables importantes, colocando un comentario explicativo. Por ejemplo,

```
double *factPrim;// apuntador para los factores primos
int maxFactPrim; // numero maximo de factores primos
int nFactPrim;   // numero de factores primos
```

- Las variables menos importantes pueden ser declaradas en una sola línea, por ejemplo:

```
int i, j, k;
double temp1, temp2;
```

- Separar de manera clara las funciones entre sí, por ejemplo con una línea como

```
//-----
```

- Dejar un espacio después de cada coma, por ejemplo,

```
int i, j, k;

x = maximo(a, b, c, d);
```

- En las asignaciones, dejar un espacio antes y después del signo = , pero la abreviación += (y las análogas) debe permanecer pegada. Por ejemplo:

```
a = b+3;
d += a;
```

¹Libros importantísimos de C++ como [Str02] (libro obligado de referencia), aconsejan justamente lo contrario, es decir, sugieren declarar las variables en el momento que se van a necesitar, por ejemplo, `for(int i = 0; i <n; i++)...`

- Los nombres de las variables, deben ser indicativos de su significado. Cuando hay varias palabras, éstas se separan con el signo _ o por medio de una mayúscula al empezar la nueva palabra. Por ejemplo, para el peso específico:

pe	no es muy diciente
peso_esp	puede ser
pesoEsp	puede ser

- Colocar comentarios adecuados para cada función. Deben indicar la acción de la función y el significado de cada parámetro.

```

void prodCompl( double a, double b, double x,
                 double y, double &u, double &v)
{
    // producto de dos complejos
    // u + iv = (a + ib)(x + iy)

    ...
}

```

- El orden para el programa puede ser:

comentarios iniciales
 include ...
 prototipos
 define
 typedef
 variables globales
 main
 definiciones de las funciones

A.2 Ejemplo

```
// Este programa calcula ....  
// ...  
  
#include <...>  
#include <...>  
  
#define N 20  
#define ... ...  
  
void funcion1( ..... );  
double funcion2( ... );  
  
int nMax = 10; // indica ....  
double abc; // es el ....  
  
//=====  
  
int main(void)  
{  
    int i, j, k;  
    double x, y;  
  
    ...  
    ...  
    for(...){  
        ...  
        ...  
        for(...){  
            ...  
            ...  
            if(...){  
                ...  
                ...  
                ...  
            }  
            else{  
                ...  
            }  
        }  
    }  
}
```

```
...
...
...
if(...){
    ...
    ...
    ...
}
}

}

//=====

void funcion1( ... )
{
    // comentarios
    // mas comentarios

    int m, n;
    float u; // indica ...

    ...
}

//=====

double funcion2( ... )
{
    // comentarios
    // mas comentarios

    int i, j;
    double x; // x indica ...
    double zz; // indica ...
```

```
    ...
    ...
}
```

A.3 Estructuras de control

```
if( ... ) {
    ...
    ...
    ...
}
else {
    ...
    ...
    ...
}

if( ... ) ... ;
else ... ;

if( ... ) ... ;
else {
    ...
    ...
    ...
}

if( ... ){
    ...
    ...
    ...
}
else ... ;

if( ... ){
    ...
    ...
}
```

```
    ...
}

...
.

for( ... ) {
    ...
    ...
    ...
}
for( ... ) ...;

while( ... ) {
    ...
    ...
    ...
}
while( ... ) ... ;

do {
    ...
    ...
    ...
} while ( ... );

switch( i ) {
    case 1:
        ...
        ...
        ...
        break;
    case 4:
        ...
}
```

```
...
...
break;
case 8: case 9:
...
...
...
break;
default:
...
...
...
}
}
```

Índice analítico

- ||, 45
- &, 31, 78, 79, 109
- &&, 45
- '\0', 98
- (double), 36
- (float), 36
- (int), 36
- *, 22, 78, 109
- **, 116
- *=, 27
- +, 22
- ++, 22, 23
- +=, 27
- , 22, 23
- , 22, 24
- =, 27
- /, 22
- /=, 27
- <, 45
- <<, 8
- <=, 45
- =, 21
- ==, 45
- >, 45
- >=, 45
- >>, 33
- ?, 171, 173
- %, 22, 23
- %c, 31
- %d, 31
- %e, 31
- %f, 31
- %i, 31
- %lf, 31
- %p, 31, 110
- %s, 31, 99
- %u, 31
- \n, 7
- a, 145
- abreviaciones, 26
- acos, 29
- Adams-Bashforth
 fórmula de, 363
- Adams-Moulton
 fórmula de, 367
- adición, 22
- algoritmo
 de Euclides, 54
- alloc.h, 85
- ámbito
 global, 82
 local, 82
- ANSI C, 70
- aproximación, 285
- aproximación por mínimos cuadrados, 306
- apuntador, 78
- apunadores, 109
 - a apunadores, 116
 - a estructuras, 177, 180

- a funciones, 165
- dobles, 116
- dobles y matrices, 135
- y arreglos bidimensionales, 116
- y arreglos unidimensionales, 111
- archivo
 - de texto, 145
 - texto, 145
- argc**, 172, 173
- argumento, 68
- argumentos de **main**, 171
- argv**, 172, 173
- arreglos, 87
 - a partir de 1, 139
 - aleatorios, 129
 - bidimensionales, 87, 94
 - y apuntadores, 116
 - como parámetros, 94, 97
 - como parámetros, 88
 - de estructuras, 177, 180
 - multidimensionales, 94
 - unidimensionales, 87
 - y apuntadores, 111
 - y matrices, 117
- ASCII, 5
- asignación dinámica de memoria, 131
 - para estructuras, 177, 180
- asin**, 29
- asm**, 17
- asociatividad, 25
 - de derecha a izquierda, 25
 - de izquierda a derecha, 25
- atan**, 29
- auto**, 17
- base, 287, 307
- biblioteca estándar, 85
- bloque, 43
- bloque de sentencias, 43
- Borland, 70
- break**, 17, 58
- bucle, 48
- burbuja, método, 194
- %c**, 31
- cadena de formato, 31
- cadenas, 98
- cambio de signo, 23
- caracteres, 19
- case**, 17, 57
- catch**, 17
- ceil**, 29
- char**, 17, 19, 20, 31, 158
- cin**, 33
- class**, 17
- código, 5
- comentario, 15
- compilador, 5
- complejo, 175
- complejos, 176
- complex.h**, 85
- condiciones de frontera
 - ecuaciones diferenciales con, 376
 - ecuaciones diferenciales lineales con, 379
- conio.h**, 70
- const**, 17, 159
- continue**, 17, 60
- control del paso, 353
- convergencia
 - cuadrática, 262
 - lineal, 262
- conversiones de tipo en expresiones mixtas, 35
- cos**, 29

- coseno**, 53
cosh, 29
cout, 8, 33
ctype.h, 85
cuadratura
 de Gauss, 326
 de Gauss-Legendre, véase cuadratura de Gauss
cuadro de una función, 69

%d, 31
decremento, 24
default, 17
#define, 164
delete, 17, 132
densidad, 254
derivación
 numérica, 332
derivadas parciales, 246
desviación estándar, 148
determinante, 229
diagonal estrictamente dominante por filas, 242
diferencias
 divididas de Newton, 293
 finitas, 302
diferencias finitas, 379
diferente, 45
distancia entre dos vectores, 183
división, 22
 entre enteros, 23
 sintética, 134
do, 17
do while, 54, 58, 60
doble precisión, 19
double, 17, 19, 20, 31, 35, 158
double(), 37
(double), 36

%e, 31
ecuación cuadrática, 43
ecuaciones
 diferenciales ordinarias, 337
ecuaciones diferenciales
 con condiciones de frontera, 376
 de orden superior, 373
 lineales con condiciones de frontera, 379
 sistemas de, 371
ecuaciones normales, 247
else, 17, 41
encadenamiento, 6
endl, 8
enlace, 6
entrada de datos, 30
enum, 17
EOF, 34, 148
errno.h, 85
error, 331
 de compilación, 5
 de ejecución, 6
 global, 317, 318, 320, 323, 339
 orden del, 359
local, 317, 320, 339
 método de Euler, 360
 método de Heun, 360
 método del punto medio, 360
 método RK4, 360
 método RK5, 360
 método RK6, 360
 orden del, 359
 método de Euler, 339
 orden del, 359
escritura

- de un vector, 89, 91, 117, 120
 - en un archivo, 150–152
- de una matriz, 94, 97, 117,
 - 120
 - en un archivo, 152, 154
 - en archivos, 143
- esquema de Hörner, 134
- estructura
 - devuelta por una función, 177, 180
- estructuras, 175
 - como parámetros, 177, 180
- estructuras de control, 41
- Euler
 - método de, 338, 348
 - orden del método de, 360
- exit**, 61
- exp**, 29
- extern**, 17, 173
- %f**, 31
- fabs**, 29
- factores primos de un entero, 92, 94
- factorial, 50, 67, 75
- factorización
 - de Cholesky, 229, 235, 237
 - LU, 215
 - PA=LU, 223
- fclose**, 143, 144
- feof**, 143
- feof**, 148–150
- FILE**, 144, 145
- float**, 17, 19, 20, 31, 35, 158
 - (**float**), 36
 - float.h**, 85
 - floor**, 29
 - fopen**, 143–145
 - for**, 17, 48, 58, 60
- formato, 31
- fórmula
 - de Adams-Bashforth, 363
 - de Simpson, 319
 - de Adams-Moulton, 367
 - del trapecio, 314
- formulas
- de Newton-Cotes, 314, 325
- fórmulas
 - de Newton-Cotes, 319
 - abiertas, 325
 - cerradas, 325
- Fortran, 1
- fprintf**, 143, 144
- free**, 131, 132
- friend**, 17
- fscanf**, 143, 144
- función
 - exponencial, 51
- funciones, 67
 - elementales, 183
 - en línea, 169
 - matemáticas, 27
 - recurrentes, 75
- funciones de la base, 287, 307
- Gauss, *véase* método de Gauss
- Gauss-Seidel, *véase* método de Gauss-Seidel
- getch**, 70
- getchar**, 71
- gets**, 147, 148
- goto**, 17, 61
- Heun
 - método de, 341, 348
 - orden del método de, 360
- %i**, 31

- identificador, 16
- if**, 17, 41
- igual, 45
- #include**, 7, 161
- incremento, 23
- inicialización de arreglos, 103
- inline**, 17, 170
- int**, 17, 19, 20, 31, 35, 158
- int()**, 37
- (int), 36
- integración numérica, 313
- intercambio, 78
 - de vectores, 183
- interpolación, 285–287
 - de Lagrange, 289
 - por diferencias divididas, 298
 - por diferencias finitas, 304
- iostream.h**, 8, 85
- Lagrange, véase interpolación de Lagrange
- lectura
 - de un vector, 89, 91
 - en un archivo, 150–152
 - de una matriz, 94, 97, 117, 120
 - en un archivo, 152, 154
 - en archivos, 143
- %lf**, 31
- limits.h**, 85
- log**, 29
- log10**, 29
- long**, 17, 20
- long double**, 21, 35, 158
- long int**, 21, 35, 158
- main**, 7, 171
- malloc**, 131, 132
- math.h**, 27, 85
- Matlab, 200
- matrices
 - ortogonales, 250
 - y apuntadores dobles, 135
 - y arreglos unidimensionales, 117
- matriz
 - banda, 253
 - de diagonal estrictamente dominante por filas, 242
 - de Givens, 250
 - de Householder, 250
 - definida positiva, 227, 229, 243
 - dispersa, 254
 - jacobiana, 280
 - positivamente definida, véase matriz definida positiva
- máximo común divisor, 54
- máximo de un vector, 183
- mayor, 45
- mayor o igual, 45
- menor, 45
- menor o igual, 45
- método
 - burbuja, 194
 - de Cholesky, 227, 237–239
 - de colocación, 286
 - de Euler, 338, 348
 - de Gauss, 206
 - de Gauss con pivoteo parcial, 217, 222, 223
 - de Gauss-Seidel, 239
 - de Heun, 341, 348
 - de la bisección, 71, 79, 267
 - de la secante, 263
 - de Newton, 257, 278
 - de Newton en \mathbb{R}^n , 279, 280

- de punto fijo, 272, 278
- de Regula Falsi, 268
- de Regula Falsi modificado, 270
- de Runge-Kutta (RK), 347
- de Runge-Kutta-Fehlberg, 354, 356
- del disparo (shooting), 376, 377
- del punto medio, 344, 348
- del trapecio, 341
- multipaso abierto, 363
- multipaso cerrado, 367
- multipaso explícito, 363
- multipaso implícito, 367
- orden del, 359
- predictor-corrector, 367
- RK, 347
- RK2, 351
 - deducción del, 350
- RK4, 348
- RK5, 354
- RK6, 354
- RKF, 354
- métodos
 - de Runge-Kutta, 347
 - indirectos, 239
 - iterativos, 239
 - multipaso explícitos, 362
 - multipaso implícitos, 366
 - RK, 347
- mínimos cuadrados, véase solución por...
- modificadores, 20
- modificadores de formato, 32
- molde funcional, 37
- moldes, 36
- multiplicación, 22
- \n, 7
- new, 17, 132
- no, 45
- norma
 - de un complejo, 177, 180
- notación de Matlab, 200
- notación de Scilab, 200
- NULL, 131, 132, 145
- número
 - complejo, 175
 - de cifras decimales, 32
 - de columnas, 32
 - de operaciones, 203, 211, 235
- números
 - enteros, 19
 - reales, 19
- o, 45
- operaciones elementales con vectores, 183
- operador
 - de asignación, 21
 - unario, 23
- operadores
 - lógicos, 45
 - relacionales, 45
- operadores aritméticos, 22
- operator, 17
- orden
 - del error, 359
 - verificación numérica, 360
 - del error global, 359
 - del error local, 359
 - del método, 359
 - de Euler, 360
 - de Heun, 360
 - del punto medio, 360
 - RK4, 360
 - RK5, 360

- RK6, 360
- orden de convergencia, 262, 264
- ordenamiento, 194
- overflow, 75
- %p**, 31, 110
- palabra clave, 16
- parámetro, 68
 - por defecto, 81
 - por referencia, 76–78, 88
 - por valor, 76, 77
- pausa, 70
- pivote, 217
- pivoteo
 - parcial, 217
 - total, 218
- polinomios
 - de Legendre, 331
 - polinomios de Lagrange, 290
- pow**, 29
- precedencia de los operadores aritméticos, 24
- precisión
 - doble, 19
 - sencilla, 19
- printf**, 7
- prioridad de los operadores aritméticos, 24
- private**, 17
- producto
 - de complejos, 177, 180
 - de matrices, 94, 97, 117, 120–123, 125, 128, 129
 - escalar, 126, 128, 183
- programa
 - ejecutable, 6
 - fuente, 5
- promedio, 89, 91, 113, 115, 148
- protected**, 17
- prototipo, 68
- public**, 17
- punteros, véase apuntadores
- punto flotante, 19
- punto medio
 - método del, 344, 348
 - orden del método de, 360
- r**, 145
- r+**, 145
- RAND_MAX**, 130
- rand()**, 130
- Raphson, véase método de Newton-Raphson
- register**, 17
- residuo entero, 23
- return**, 17
- RK, véase método de Runge-Kutta
- RK4, véase método RK4
- RKF, véase Runge-Kutta-Fehlberg
- Runge-Kutta
 - método de, 347
- Runge-Kutta-Fehlberg
 - método de, 354, 356
- %s**, 31, 99
- salida de resultados, 30
- scanf**, 30, 34
- Scilab, 200
- Seidel, véase método de Gauss-Seidel
- serie de Taylor, 51, 53
- seudosolución, 248
- short**, 17
- short int**, 35, 158
- signed**, 17
- sin**, 29
- sinh**, 29
- sistema

- diagonal, 202
- triangular inferior, 206
- triangular superior, 202
- sistemas
 - de ecuaciones diferenciales, 371
- sizeof**, 17, 132, 157, 158
- sobrecarga de funciones, 83
- solución
 - de ecuaciones, 255
 - de sistemas lineales, 199
 - de un sistema
 - diagonal, 202
 - triangular inferior, 206
 - triangular superior, 202
 - por mínimos cuadrados, 245
- sparse**, 254
- sqrt**, 27, 29
- static**, 17, 173
- stddef.h**, 85
- stdio.h**, 7
- stdio.h**, 85
- stdlib.h**, 85, 130
- strcat**, 101, 102
- strcpy**, 101
- string.h**, 85, 101
- strlen**, 101
- struct**, 17, 175
- sustracción, 22
- switch**, 17, 57
- tabla
 - de diferencias divididas, 296
 - de diferencias finitas, 302
- tan**, 29
- tanh**, 29
- tasa de convergencia, 262
- template**, 17
- this**, 17
- throw**, 17
- time.h**, 85
- tipos de datos, 19
- triangularización, 206, 209, 211
- try**, 17
- typedef**, 17, 159
- %u**, 31
- union**, 17
- unsigned**, 17, 20
- unsigned int**, 21
- valor
 - devuelto, 69
 - dominante, 70
 - propio, 228
- variables
 - globales, 82
 - locales, 82
- vectores
 - almacenados con salto, 184, 190
- virtual**, 17
- void**, 7, 17
- volatile**, 17
- w**, 145
- while**, 17, 51, 58, 60
- y, 45

*Introducción a C
y a métodos numéricos*

SE TERMINÓ DE IMPRIMIR
EN BOGOTÁ EL MES DE
ABRIL DE 2004 EN LAS
PRENSAS EDITORIALES DE
UNIBIBLOS, UNIVERSIDAD
NACIONAL DE COLOMBIA

Otros títulos de esta colección

Programación en Mathematica con aplicaciones a la Teoría de Nudos

Margarita María Toro Villegas

Cómo cuidar al paciente con soporte mecánico ventilatorio

María Antonia Jiménez de Morales

Pruebas de toxicidad acuática: fundamentos y métodos

María Consuelo Díaz Báez

María Cristina Bustos López

Adriana Janneth Espinosa Ramírez

El lenguaje en la educación. Una perspectiva fonoaudiológica

Rita Romero Flórez

Inferencia estadística

Jorge Humberto Mayorga Álvarez

Análisis clásico de estructuras

José Óscar Jaramillo Jiménez

Glosario ilustrado de dermatología y dermatopatología

José Gerzaín Rodríguez Toro

Introducción a C y a métodos numéricos

Héctor Manuel Mora Escobar

La adaptación neonatal inmediata. La reanimación neonatal

Santiago Currea Guerrero

Análisis de armónicos en sistemas de distribución

Estrella Esperanza Parra López

Lo malo y lo feo de los microbios

George Charles Volcy Etienne

Guías de laboratorio de Fisiología Vegetal

Víctor Julio Flórez Roncancio

Rafael Cruz

La enseñanza de los pueblos antiguos en la escuela

Dario Campos Rodríguez

Nelly Rodríguez Melo

Mítica koreabajü - español

Pedro Túlio Marín Silva

Este libro presenta los principales temas del lenguaje C y de métodos numéricos. También trata algunos tópicos muy sencillos y útiles de C++.

Los temas de C son ilustrados con ejemplos de problemas sencillos de matemáticas. El manejo de matrices está dirigido a su posterior utilización en la solución de ecuaciones lineales. Los métodos numéricos están presentados mediante la exposición de las principales ideas, la deducción intuitiva, resultados teóricos relativos al método (error, convergencia...), el algoritmo y la implementación en C.

En la página Internet del autor,
<http://www.matematicas.unal.edu.co/~hmora>,
están disponibles el código en C de varias
funciones para métodos numéricos y algunos
documentos adicionales.

ISBN 958-701-363-8



9 789587 013634