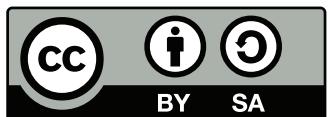


Universidad Complutense de Madrid

Laboratorio de Computación Científica

Juan Jiménez

14 de enero de 2021



El contenido de estos apuntes esté1 bajo licencia Creative Commons Atribution-ShareAlike 4.0
<http://creativecommons.org/licenses/by-sa/4.0/>
©Juan Jiménez

Índice general

1. Introducción al software científico	15
1.1. Introducción a los computadores	16
1.1.1. Niveles de descripción de un ordenador	16
1.1.2. El modelo de computador de Von Neumann	18
1.1.3. Representación binaria	20
1.2. Aplicaciones de Software Científico	23
2. Introducción a la programación en Matlab	25
2.1. El entorno de programación de Matlab	25
2.1.1. La ventana de comandos de Matlab	25
2.1.2. Variables.	26
2.1.3. Vectores y matrices.	33
2.1.4. Estructuras y células	38
2.2. Entrada y salida de Datos	41
2.2.1. Exportar e importar datos en Matlab	41
2.3. Operaciones aritméticas, relacionales y lógicas	47
2.3.1. Operaciones aritméticas	47
2.3.2. Precedencia de los operadores aritméticos	53
2.3.3. Operaciones Relacionales y lógicas.	55
2.4. Scripts y Funciones	67
2.4.1. El editor de textos de Matlab.	67
2.4.2. Scripts	68
2.4.3. Funciones	70
2.4.4. Funciones incluidas en Matlab.	76
2.4.5. Depuración.	76
2.5. Control de Flujo	83
2.5.1. Flujo condicional.	83
2.5.2. Bucles	88
2.5.3. Funciones recursivas.	93
2.5.4. Algoritmos y diagramas de flujo.	94
2.6. Representación Gráfica	97
2.6.1. El comando plot y las figuras en Matlab.	97
2.6.2. Gráficos en 2D	105
2.6.3. Gráficos en 3D.	113

3. Aritmética del Computador y Fuentes de error	127
3.1. Representación binaria y decimal	127
3.2. Representación de números en el ordenador	128
3.2.1. Números no enteros: Representación en punto fijo y en punto flotante	130
3.3. Errores en la representación numérica.	138
3.3.1. Error de redondeo unitario	138
3.3.2. Errores de desbordamiento	141
3.4. Errores derivados de las operaciones aritméticas	142
3.4.1. Acumulación de errores de redondeo	142
3.4.2. Anulación catastrófica	145
3.4.3. Errores de desbordamiento	146
4. Cálculo de raíces de una función	149
4.1. Raíces de una función	149
4.2. Metodos iterativos locales	151
4.2.1. Método de la bisección	151
4.2.2. Método de interpolación lineal o (<i>Regula falsi</i>)	152
4.2.3. Método de Newton-Raphson	157
4.2.4. Método de la secante	159
4.2.5. Método de las aproximaciones sucesivas o del punto fijo	161
4.3. Cálculo de raíces de funciones con Matlab.	172
4.3.1. La función de Matlab <code>fzero</code>	172
4.3.2. Cálculo de raíces de polinomios.	177
5. Aplicaciones del cálculo científico al álgebra lineal	181
5.1. Matrices y vectores	181
5.2. Operaciones matriciales	183
5.3. Operadores vectoriales	197
5.4. Tipos de matrices empleados frecuentemente	208
5.5. Factorización de matrices	209
5.5.1. Factorizacion LU	209
5.5.2. Factorización de Cholesky	220
5.5.3. Diagonalización	222
5.5.4. Factorización QR	228
5.5.5. Factorización SVD	235
6. Sistemas de ecuaciones lineales	239
6.1. Introducción	239
6.2. Condicionamiento	243
6.3. Métodos directos	246
6.3.1. Sistemas triangulares	246
6.3.2. Métodos basados en las factorizaciones	248
6.3.3. El método de eliminación de Gauss.	255
6.3.4. Gauss-Jordan y matrices en forma reducida escalonada	259
6.4. Métodos iterativos	262
6.4.1. El método de Jacobi.	263
6.4.2. El método de Gauss-Seidel.	270
6.4.3. Amortiguamiento.	273
6.4.4. Análisis de convergencia	276

7. Interpolación y ajuste de funciones	281
7.1. El polinomio de Taylor	282
7.2. Interpolación polinómica	285
7.2.1. La matriz de Vandermonde	285
7.2.2. El polinomio interpolador de Lagrange	286
7.3. Diferencias divididas	287
7.3.1. El polinomio de Newton-Gregory	290
7.4. Interpolación por intervalos	293
7.4.1. Interpolación mediante splines cúbicos	295
7.4.2. Funciones propias de Matlab para interpolación por intervalos	300
7.5. Ajuste polinómico por el método de mínimos cuadrados	301
7.5.1. Mínimos cuadrados en Matlab	305
7.5.2. Análisis de la bondad de un ajuste por mínimos cuadrados	306
7.6. Curvas de Bézier	308
8. Diferenciación e Integración numérica	315
8.1. Diferenciación numérica	315
8.1.1. Diferenciación numérica basada en el polinomio de interpolación	316
8.1.2. Diferenciación numérica basada en diferencias finitas	317
8.2. Integración numérica	320
8.2.1. La fórmula del trapecio	321
8.3. Las fórmulas de Simpson	323
8.4. Problemas de valor inicial en ecuaciones diferenciales	327
8.4.1. El método de Euler	328
8.4.2. Métodos de Runge-Kutta	332
9. Tratamiento estadístico de datos	335
9.1. Secuencias de números aleatorios	335
9.1.1. El generador de números aleatorios de Matlab	339
9.2. Probabilidad y distribuciones de probabilidad	341
9.2.1. Sucesos aleatorios discretos	342
9.2.2. Distribuciones de probabilidad continuas	344
9.3. El teorema del límite central	349
9.4. Incertidumbre en las medidas experimentales	352
9.4.1. Fuentes de incertidumbre	352
9.4.2. Intervalos de confianza	354
9.4.3. Propagación de la incertidumbre: Estimación de la incertidumbre de medidas indirectas	358
9.4.4. Ejemplo de estimación de la incertidumbre con Matlab	360
10. Introducción al cálculo simbólico	365
10.1. Cálculo simbólico en el entorno de Matlab	365
10.2. Variables y expresiones simbólicas	366
10.2.1. Variables simbólicas	366
10.2.2. Expresiones simbólicas	368
10.2.3. Simplificación de expresiones simbólicas	369
10.2.4. Sustitución de variables por valores numéricos	370
10.3. Cálculo infinitesimal	372
10.3.1. Derivación	372
10.3.2. Integración	375

ÍNDICE GENERAL

10.3.3. Series	376
10.3.4. Límites	377
10.4. Representación gráfica	379

Índice de figuras

1.1. Descripción por niveles de un computador	17
1.2. Modelo de Von Neumann	19
2.1. Entorno de desarrollo integrado de Matlab	26
2.2. El <i>Workspace</i> de Matlab	32
2.3. Aspecto de la herramienta de importación de datos	47
2.4. posición del botón <i>New Script</i> y del menú <i>New</i> en el IDE de Matlab (Señalados en rojo)	68
2.5. Vista del editor de textos de Matlab mostrando el contenido del fichero ejemplo1.m	69
2.6. Ejemplo de uso de memoria y ámbito de variables durante la ejecución de una función	73
2.7. Vista de el editor de texto de Matlab. Circulo rojo error en el código. Rodeado en azul advertencias de posible mejoras. Rodeado en verde mensaje de error en tiempo de ejecución	77
2.8. Vista de el editor de texto de Matlab. Circulo rojo error en el código. Rodeado en azul advertencias de posible mejoras. Rodeado en verde mensaje de error en tiempo de ejecución	78
2.9. Breakpoint activo	81
2.10. Parada de programa en breakpoint y herramientas de depuración	82
2.11. Esquema general de la estructura de flujo condicional if los términos escritos entre paréntesis son opcionales.	86
2.12. Esquema general de la estructura switch-case-otherwise	87
2.13. Esquema general de la estructura de un bucle for los términos escritos entre paréntesis son opcionales.	88
2.14. Esquema general de la estructura de un bucle while los términos escritos entre paréntesis son opcionales.	92
2.15. Símbolos empleados en diagramas de flujo	94
2.16. Diagrama de flujo para el problema de los números primos	96
2.17. Ventana gráfica de Matlab. representación de los punto de la tabla 2.6	98
2.18. gráfico de los puntos de la tabla 2.6 obtenida con el comando plot	99
2.19. Datos de la tabla 2.6 representados mediante distintos tipos de líneas y colores	101
2.20. gráficas de las funciones seno y coseno en el intervalo $(-\pi, \pi)$. Representadas en la misma figura, usando el comando hold on	103
2.21. Ejemplo de empleo del comando subplot	106
2.22. Ejemplo de empleo del comando fplot	107
2.23. Representación de la función $y = \log_{10}(x)$ empleando el comando semilogx	108
2.24. Representación de la función $y = 10^x$ empleando el comando semilogy	109
2.25. Representación de la función $r = \sqrt{\theta}$ empleando el comando polar	110

2.26. Comparación entre los comandos <code>stem</code> , <code>bar</code> y <code>stairs</code> representando la misma colección de datos	111
2.27. histogramas del número de automóviles por cada 1000 habitantes para 213 países	111
2.28. Ejemplo de uso de la función <code>plotyy</code>	112
2.29. Ejemplo de uso de la función <code>quiver</code>	113
2.30. Datos de la tabla 2.8 representados empleando el comando <code>errorbar</code>	115
2.31. Gráfico en 3D y rotaciones.	116
2.32. Retícula para representar superficies. Los puntos negros son los nodos definidos por las matrices X_m e Y_m	117
2.33. Superficie elemental obtenida elevando los cuatro puntos centrales de la figura 2.32.	118
2.34. Comparación entre <code>mesh</code> y <code>surf</code>	119
2.35. retícula con simetría circular	121
2.36. Cono representado sobre una retícula circular	122
2.37. retícula con simetría circular	123
2.38. Comparación entre los resultados de <code>contour</code> , <code>contour3</code> , <code>meshc</code> y <code>surfc</code> , para la obtención de las curvas de nivel de una superficie.	124
2.39. Curva trazada sobre una superficie	125
 3.1. Posición relativa de un número no máquina x y su redondeo a número máquina por truncamiento x_T y por exceso x_E Si redondeamos al más próximo de los dos, el error es siempre menor o igual a la mitad del intervalo $x_E - x_T$	139
3.2. Ilustración del cambio de precisión con la magnitud de los números representados.	141
3.3. Números representables y desbordamientos en el estándar IEEE 754 de precisión simple.	142
 4.1. Ejemplo de ecuación de Kepler para $a = 40$ y $b = 2$	150
4.2. Ilustración del teorema de Bolzano	151
4.3. Diagrama de flujo del método de la bisección	152
4.4. proceso de obtención de la raíz de una función por el método de la bisección	153
4.5. Obtención de la recta que une los extremos de un intervalo $[a, b]$ que contiene una raíz de la función	154
4.6. Diagrama de flujo del método de interpolación lineal	155
4.7. Proceso de obtención de la raíz de una función por el método de interpolación lineal	156
4.8. Recta tangente a la función $f(x)$ en el punto x_0	157
4.9. Diagrama de flujo del método de Newton-Raphson	158
4.10. Proceso de obtención de la raíz de una función por el método de Newton	160
4.11. Recta secante a la función $f(x)$ en los puntos x_0 y x_1	161
4.12. Diagrama de flujo del método de la secante	162
4.13. proceso de obtención de la raíz de una función por el método de la secante	163
4.14. Obtención gráfica del punto fijo de la función, $g(x) = -\sqrt{e^x}$	164
4.15. Diagrama de flujo del método del punto fijo. Nótese que la raíz obtenida corresponde a la función $f(x) = g(x) - x$	166
4.16. $g(x) = \pm\sqrt{e^x}$, Solo la rama negativa tiene un punto fijo.	167
4.17. proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = -\sqrt{e^x}$	168
4.18. primeras iteraciones de la obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \ln(x^2)$	169
4.19. proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \ln(x^2)$, el método oscila sin converger a la solución.	170

4.20. proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \frac{e^x}{x}$, el método diverge rápidamente.	171
4.21. Gráfica de la función $f(x) = e^x - x^2$, obtenida mediante <code>pinta_funcion</code>	174
 5.1. Representación gráfica de vectores en el plano	182
5.2. Representación gráfica de vectores en el espacio	183
5.3. interpretación geométrica de la norma de un vector	188
5.4. efecto del producto de un escalar por un vector	197
5.5. Representación gráfica de los vectores $a = (1, 2)$, $b = (-1, 1)$ y algunos vectores, combinación lineal de a y b	198
5.6. Representación gráfica de los vectores $a = (1, -2, 1)$, $b = (2, 0, -1)$, $c = (-1, 1, 1)$ y del vector $a - b + c$	199
5.7. Representación gráfica del vector a , en las base canónica \mathcal{C} y en la base \mathcal{B}	201
5.8. Transformaciones lineales del vector $a = [1; 2]$. D , dilatación/contracción en un factor 1,5/0,5. R_x , reflexión respecto al eje x. R_y , reflexión respecto al eje y. R_θ rotaciones respecto al origen para ángulos $\theta = \pi/6$ y $\theta = \pi/3$	205
5.9. Formas cuadráticas asociadas a las cuatro matrices diagonales: $ a_{11} = a_{22} = 1$, $a_{12} = a_{21} = 0$	207
5.10. Obtención de un vector ortogonal	230
 6.1. Sistema de ecuaciones con solución única	241
6.2. Sistema de ecuaciones sin solución	242
6.3. Sistema de ecuaciones con infinitas soluciones	243
6.4. Diagrama de flujo general de los métodos iterativos para resolver sistemas de ecuaciones. La función $M(x)$ es la que especifica en cada caso el método.	262
6.5. evolución de la tolerancia (módulo de la diferencia entre dos soluciones sucesivas) para un mismo sistema resuelto mediante el método de Gauss-Seidel y el método de Jacobi	273
6.6. Evolución de la tolerancia para un mismo sistema empleando el metodo de Jacobi (diverge) y el de Jacobi amortiguado (converge).	279
 7.1. Comparación entre resultados obtenidos para polinomios de Taylor del logaritmo natural. (grados 2, 3, 5, 10, 20)	283
7.2. Polynomios de Taylor para las funciones coseno y seno	284
7.3. Polinomio de interpolación de grado nueve obtenido a partir de un conjunto de diez datos	293
7.4. Interpolaciones de orden cero y lineal para los datos de la figura 7.3	294
7.5. Interpolación mediante spline cúbico de los datos de la figura 7.3	300
7.6. Polinomio de mínimos cuadrados de grado 0	303
7.7. Ejemplo de uso de la ventana gráfica de Matlab para realizar un ajuste por mínimos cuadrados	307
7.8. Comparación entre los residuos obtenidos para los ajustes de mínimos cuadrados de un conjunto de datos empleando polinomios de grados 1 a 4.	308
7.9. Curvas de Bézier trazadas entre los puntos $P_0 = (0, 0)$ y $P_n = (2, 1)$, variando el número y posición de los puntos de control.	310
7.10. Curvas de Bézier equivalentes, construidas a partir de una curva con tres puntos de control	312
7.11. Curva de Bézier y su derivada con respecto al parámetro del polinomio de Bernstein que la define: $t \in [0, 1]$	314
7.12. Interpolación de tres puntos mediante dos curvas de Bézier	314

8.1. Variación del error cometido al aproximar la derivada de una función empleando una fórmula de diferenciación de dos puntos.	318
8.2. Comparación entre las aproximaciones a la derivada de una función obtenidas mediante las diferencias de dos puntos adelantada y centrada	319
8.3. Interpretación gráfica de la fórmula del trapecio.	322
8.4. Interpretación gráfica de la fórmula extendida del trapecio.	323
8.5. Interpretación gráfica de la fórmula 1/3 de Simpson.	324
8.6. Interpretación gráfica de la fórmula 3/8 de Simpson.	325
8.7. Circuito RC	330
8.8. Comparación entre los resultados obtenidos mediante el método de Euler para dos pasos de integración $h = 0,05$ y $h = 0,001$ y la solución analítica para el voltaje V_o de un condensador durante su carga.	331
 9.1. Secuencia de 100 números pseudoaleatorios generada mediante el método <i>middle square</i>	337
9.2. Distribución de probabilidad y probabilidad acumulada de los resultados de lanzar una moneda al aire.	343
9.3. Distribución de probabilidad y probabilidad acumulada de los resultados de lanzar un dado al aire.	344
9.4. Distribución de probabilidad y probabilidad acumulada de los resultados de lanzar un dado trucado al aire.	345
9.5. Distribución de probabilidad uniforme para un intervalo $[a, b]$ y probabilidad acumulada correspondiente	346
9.6. Distribución de probabilidad exponencial	347
9.7. Distribución de probabilidad normal	348
9.8. Teorema del límite central: Comparación entre histogramas normalizados para un millón de medias y la distribución normal a que pertenecen	351
9.9. Modo correcto de expresar una medida experimental	352
9.10. Intervalo de confianza del 68,27 %	354
9.11. Función inversa de probabilidad normal acumulada	356
9.12. Intervalo de probabilidad P%	357
9.13. Comparación entre las distribuciones t de Student de 1, 5, 10, 20 y 30 grados de libertad y la distribución normal.	358
 10.1. Gráfica de la función $f(x) = e^{\sin(x)}$ obtenida a partir de su expresión simbólica con el comando <code>ezsurf</code> de Matlab	380
10.2. Gráfica de la curva paramétrica $x(t) = t \cos(t)$, $y(t) = t \sin(t)$ obtenida a partir de su expresión simbólica con el comando <code>ezsurf</code> de Matlab	381
10.3. Gráfica de la función $f(x) = \sin(\sqrt{x^2 + y^2})$ obtenida a partir de su expresión simbólica con el comando <code>ezsurf</code> de Matlab	382

Índice de cuadros

2.1.	Formatos numéricos más comunes en Matlab	33
2.2.	Operadores aritméticos definidos en Matlab	48
2.3.	Operadores relacionales definidos en Matlab	56
2.4.	Operadores lógicos elemento a elemento	63
2.5.	Algunas funciones matemáticas en Matlab de uso frecuente	76
2.6.	Datos de prueba	97
2.7.	tipos de línea y color del comando <code>plot</code>	100
2.8.	Resultados experimentales de la medida de la velocidad de un móvil	114
3.1.	Representación <i>en exceso a 127</i> , para un exponente de 8 bits.	134
3.2.	Comparación entre los estándares del IEEE para la representación en punto flotante. (b_s bit de signo, m_i bit de mantisa, e_i bit de exponente)	137
7.1.	$f(x) = \text{erf}(x)$	281
7.2.	Tabla de diferencia divididas para cuatro datos	289
7.3.	Tabla de diferencias para el polinomio de Newton -Gregory de cuatro datos	291
8.1.	Fórmulas de derivación basadas en diferencias finitas	320
9.1.	Mediciones de temperatura y Voltaje, sobre una resistencia de prueba de $100\ \Omega$. .	361
9.2.	Medidas y varianzas estimadas	363

Prefacio

Estos apuntes cubren de forma aproximada el contenido del *Laboratorio de computación científica* del primer curso del grado en física. La idea de esta asignatura es introducir al estudiante a las estructuras elementales de programación y al cálculo numérico, como herramientas imprescindibles para el trabajo de investigación.

Casi todos los métodos que se describen en estos apuntes fueron desarrollados hace siglos por los grandes: Newton, Gauss, Lagrange, etc. Métodos que no han perdido su utilidad y que, con el advenimiento de los computadores digitales, han ganado todavía más si cabe en atractivo e interés. Se cumple una vez más la famosa frase atribuida a Bernardo de Chartres:

“Somos como enanos a los hombros de gigantes. Podemos ver más, y más lejos que ellos, no por que nuestra vista sea más aguda, sino porque somos levantados sobre su gran altura.”

En cuanto a los contenidos, ejemplos, código, etc. Estos apuntes deben mucho a muchas personas. En primer lugar a Manuel Prieto y Segundo Esteban que elaboraron las presentaciones de la asignatura *Introducción al cálculo científico y programación* de la antigua licenciatura en físicas, de la que el laboratorio de computación científica es heredera.

En segundo lugar a mis compañeros de los departamentos de *Física de la Tierra, Astronomía y Astrofísica I* y *Arquitectura de computadores y Automática* que han impartido la asignatura durante estos años:

Rosa González Barras, Belén Rodríguez Fonseca, Maurizio Matessini, Pablo Zurita, Vicente Carlos Ruiz Martínez, Encarna Serrano, Carlos García Sánchez, Jose Antonio Martín, Victoria López López, Alberto del Barrio, Blanca Ayarzagüena, Javier Gómez Selles, Nacho Gómez Pérez, Marta Ávalos, Iñaqui Hidalgo, Daviz sánchez, Juan Rodriguez, María Ramirez, Álvaro de la Cámara (Espero no haberme olvidado de nadie).

Muchas gracias a todos por tantas horas de trabajo compartidas.

Por último, los errores y erratas que encuentres en estas notas, esos sí que son de mi exclusiva responsabilidad. Puedes —si quieres— ayudarme a corregirlos en futuras ediciones escribiendo a: juan.jimenez@fis.ucm.es

Juan Jiménez.

Capítulo 1

Introducción al software científico

En la actualidad, el ordenador se ha convertido en una herramienta imprescindible para el trabajo de cualquier investigador científico. Su uso ha permitido realizar tareas que sin su ayuda resultarían sencillamente imposibles de acometer. Entre otras, distinguiremos las tres siguientes:

- Adquisición de datos de dispositivos experimentales.
- Análisis y tratamiento de datos experimentales.
- Cálculo Científico.

La primera de éstas tareas queda fuera de los contenidos de esta asignatura. Su objetivo es emplear el ordenador para recoger datos automáticamente de los sensores empleados en un dispositivo experimental. El procedimiento habitual es emplear dispositivos electrónicos que traducen las lecturas de un sensor (un termómetro, un manómetro, un caudalímetro, una cámara etc.) a un voltaje. El voltaje es digitalizado, es decir, convertido a una secuencia de ceros y unos, y almacenando en un ordenador para su posterior análisis o/y directamente monitorizado, es decir, mostrado en la pantalla del ordenador. En muchos casos el ordenador es a su vez capaz de interactuar con el dispositivo experimental: iniciar o detener un experimento, regular las condiciones en que se realiza, disparar alarmas si se producen errores, etc.

De este modo, el investigador científico, queda dispensado de la tarea de adquirir por sí mismo los datos experimentales. Tarea que en algunos casos resultaría imposible, por ejemplo si necesita medir muchas variables a la vez o si debe medirlas a gran ritmo; y en la que, en general, es relativamente fácil cometer errores.

El análisis y tratamiento de datos experimentales, constituye una tarea fundamental dentro del trabajo de investigación científica. Los ordenadores permiten realizar dichas tareas, de una forma eficiente y segura con cantidades de datos que resultarían imposibles de manejar hace 50 años. Como veremos más adelante, una simple hoja de cálculo puede ahorrarnos una cuantas horas de cálculos tediosos. El análisis estadístico de un conjunto de datos experimentales, el cálculo –la estimación– de los errores experimentales cometidos, la posterior regresión de los datos obtenidos a una función matemática que permita establecer una ley o al menos una relación entre los datos obtenidos, formar parte del trabajo cotidiano del investigador, virtualmente en todos los campos de la ciencia.

Por último el cálculo. Cabría decir que constituye el núcleo del trabajo de investigación. El científico trata de explicar la realidad que le rodea, mediante el empleo de una descripción matemática. Dicha descripción suele tomar la forma de un modelo matemático más o menos complejo. La validez de un modelo está ligada a que sea capaz de reproducir los resultados experimentales

obtenidos del fenómeno que pretende explicar. Si el modelo es bueno será capaz de obtener mediante cálculo unos resultados similares a los obtenido mediante el experimento. De este modo, el modelo queda validado y es posible emplearlo para predecir cómo se comportará el sistema objeto de estudio en otras condiciones.

1.1. Introducción a los computadores

Más o menos todos estamos familiarizados con lo que es un computador, los encontramos a diario continuamente y, de hecho, hay muchos aspectos de nuestra vida actual que serían inimaginables sin los computadores. En términos muy generales, podemos definir un computador como una máquina que es capaz de recibir instrucciones y realizar operaciones (cálculos) a partir de las instrucciones recibidas. Precisamente es la capacidad de recibir instrucciones lo que hace del ordenador una herramienta versátil; según las instrucciones recibidas y de acuerdo también a sus posibilidades como máquina, el ordenador puede realizar tareas muy distintas, entre las que cabe destacar como más generales, las siguientes:

- Procesamiento de datos
- Almacenamiento de datos
- Transferencias de datos entre el computador y el exterior
- Control de las anteriores operaciones

El computador se diseña para realizar funciones generales que se especifican cuando se programa. La programación es la que concreta las tareas que efectivamente realiza un ordenador concreto.

1.1.1. Niveles de descripción de un ordenador

La figura 1.1 muestra un modelo general de un computador descrito por niveles. Cada nivel, supone y se apoya en el nivel anterior.

1. **Nivel Físico.** Constituye la base del *hardware* del computador. Está constituido por los componentes electrónicos básicos, diodos, transistores, resistencias, etc. En un computador moderno, no es posible separar o tan siquiera observar dichos componentes: Se han fabricado directamente sobre un cristal semiconductor, y forman parte de un dispositivo electrónico conocido con el nombre de circuito integrado.
2. **Circuito Digital.** Los componentes del nivel físico se agrupan formando circuitos digitales, (En nuestro caso circuitos digitales integrados). Los circuitos digitales trabajan solo con dos niveles de tensión (V_1, V_0) lo que permite emplearlos para establecer relaciones lógicas: V_1 =verdadero, V_2 =falso. Estas relaciones lógicas establecidas empleando los valores de la tensión de los circuitos digitales constituyen el soporte de todos los cálculos que el computador puede realizar.
3. **Organización Hardware del sistema.** Los circuitos digitales integrados se agrupan y organizan para formar el *Hardware* del ordenador. Los módulos básicos que constituyen el *Hardware* son la unidad central de procesos (CPU), La unidad de memoria y las unidades de entrada y salida de datos. Dichos componentes están conectados entre sí mediante un bus, que transfiere datos de una unidad a otra.

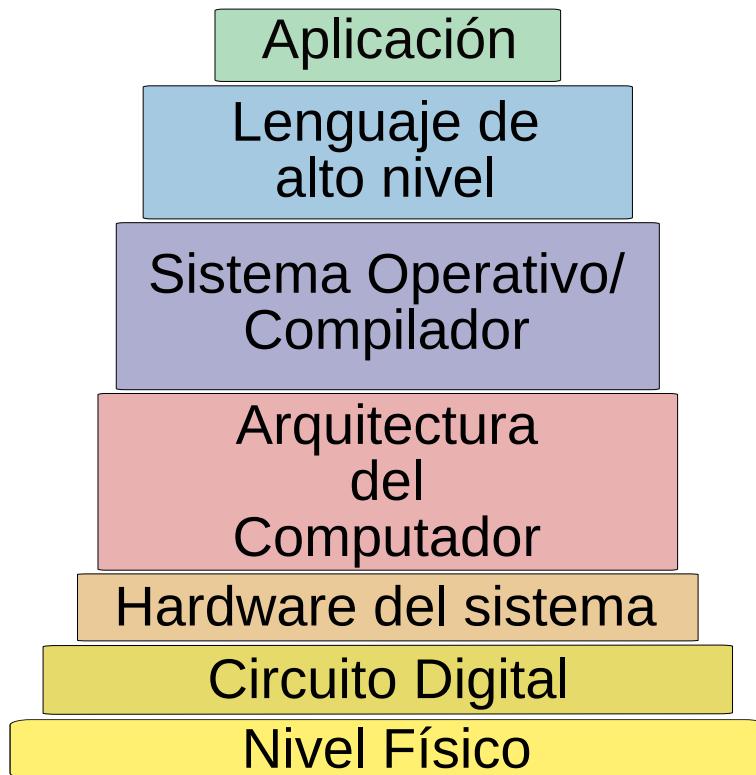


Figura 1.1: Descripción por niveles de un computador

4. **Arquitectura del computador.** La arquitectura define cómo trabaja el computador. Por tanto, está estrechamente relacionada con la organización hardware del sistema, pero opera a un nivel de abstracción superior. Establece cómo se accede a los registros de memoria, arbitra el uso de los buses que comunican unos componentes con otros, y regula el trabajo de la CPU.

Sobre la arquitectura se establece el lenguaje básico en el que trabaja el ordenador, conocido como lenguaje máquina. Es un lenguaje que emplea todavía niveles lógicos binarios (ceros o unos) y por tanto no demasiado apto para ser interpretado por los seres humanos. Este lenguaje permite al ordenador realizar operaciones básicas como copiar el contenido de un registro de memoria en otro, sumar el contenido de dos registros de memoria, etc.

El lenguaje máquina es adecuado para los computadores, pero no para los humanos, por eso, los fabricantes suministran junto con el computador un repertorio básico de instrucciones que su máquina puede entender y realizar en un lenguaje algo más asequible. Se trata del lenguaje ensamblador. Los comandos de éste lenguaje son fácilmente traducibles en una o varias instrucciones de lenguaje máquina. Aún así se trata de un lenguaje en el que programar directamente resulta una tarea tediosa y proclive a cometer errores.

5. **Compiladores y Sistemas Operativos** Los Compiladores constituyen un tipo de programas especiales que permiten convertir un conjunto de instrucciones, escritas en un lenguaje de alto nivel en lenguaje máquina. El programador escribe sus instrucciones en un fichero de texto normal, perfectamente legible para el ser humano, y el compilador convierte las instrucciones contenidas en dicho fichero en secuencias binarias comprensibles por la máquina.

Los computadores primitivos solo eran capaces de ejecutar un programa a la vez. A medida que se fueron fabricando ordenadores más sofisticados, surgió la idea de crear programas que se encargaran de las tareas básicas: gestionar el flujo de información, manejar periféricos, etc. Estos programas reciben el nombre de sistemas operativos. Los computadores modernos cargan al arrancar un sistema operativo que controla la ejecución del resto de las aplicaciones. Ejemplos de sistemas operativos son DOS (Disk Operating System), Unix y su versión para ordenadores personales Linux.

6. **Lenguajes de alto nivel.** Los lenguajes de alto nivel están pensados para facilitar la tarea del programador, desentendiéndose de los detalles de implementación del hardware del ordenador. Están compuestos por un conjunto de comandos y unas reglas sintácticas, que permiten describir las instrucciones para el computador en forma de texto.

De una manera muy general, se pueden dividir los lenguajes de alto nivel en lenguajes compilados y lenguajes interpretados. Los lenguajes compilados emplean un compilador para convertir los comandos del lenguaje de alto nivel en lenguaje máquina. Ejemplos de lenguajes compilados son C, C++ y Fortran. Los lenguajes interpretados a diferencia de los anteriores no se traducen a lenguaje máquina antes de ejecutarse. Si no que utilizan otro programa –el intérprete– que va leyendo los comandos del lenguaje y convirtiéndolos en instrucciones máquina a la vez que el programa se va ejecutando. Ejemplos de programas interpretados son Basic, Python y Java.

7. **Aplicaciones.** Se suele entender por aplicaciones programas orientados a tareas específicas, disponibles para un usuario final. Habitualmente se trata de programas escritos en un lenguaje de alto nivel y presentados en un formato fácilmente comprensible para quien los usa.

Existen multitud de aplicaciones, entre las más conocidas cabe incluir los navegadores para Internet, como Explorer, Mocilla o Google Chrome, los editores de texto, como Word, las hojas de cálculo como Excel o los clientes de correo como Outlook. En realidad, la lista de aplicaciones disponibles en el mercado sería interminable.

1.1.2. El modelo de computador de Von Neumann

Los computadores modernos siguen, en líneas generales, el modelo propuesto por Von Neumann. La figura 1.2 muestra un esquema de dicho modelo.

En el modelo de Von Neumann se pueden distinguir tres módulos básicos y una serie de elementos de interconexión. Los módulos básicos son:

- **La Unidad Central de Procesos.** CPU (*Central process unit*)), esta unidad constituye el núcleo en el que el ordenador realiza las operaciones.

Dentro de la CPU pueden a su vez distinguirse las siguientes partes

- La unidad de proceso ó ruta de datos: Está formada por La Unidad Aritmético Lógica (ALU), capaz de realizar las operaciones aritméticas y lógicas que indican las instrucciones del programa. En general las ALUs se construyen para realizar aritmética entre enteros, y realizar las operaciones lógicas básicas del álgebra de Boole (AND, OR, etc). Habitualmente, las operaciones para números no enteros, representados en *punto flotante* se suelen realizar empleando un procesador específico que se conoce con el nombre de Coprocesador matemático. La velocidad de procesamiento suele medirse en millones de operaciones por segundo (MIPS) o millones de operaciones en punto flotante por segundo (MFLOPS).

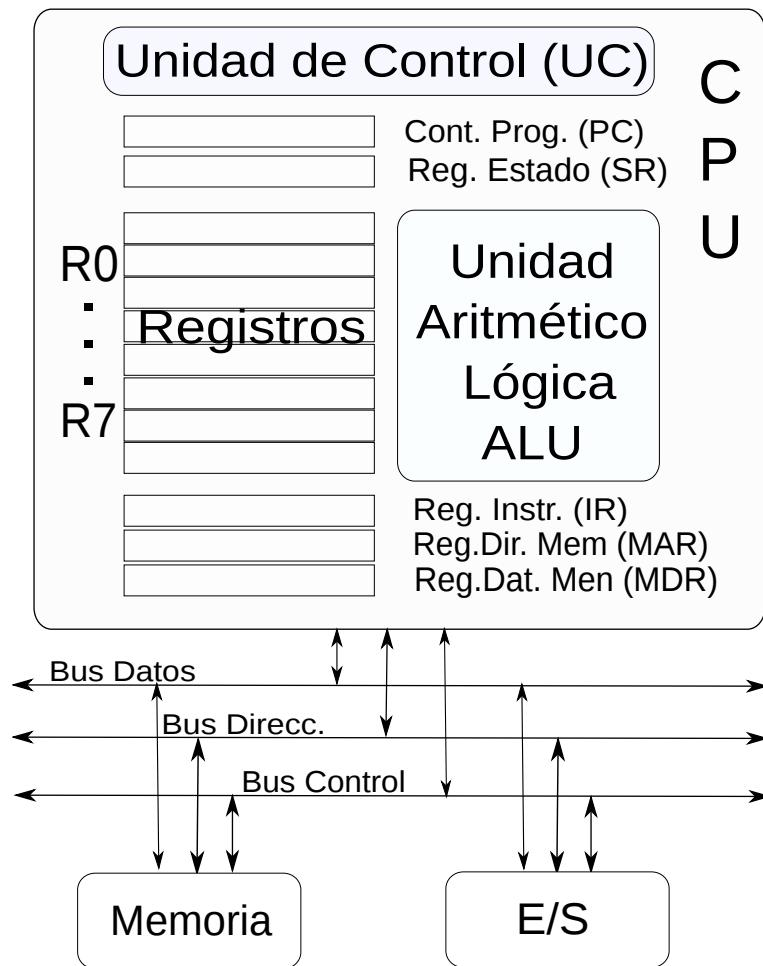


Figura 1.2: Modelo de Von Neumann

- El banco de registros: Conjunto de registros en los que se almacenan los datos con los que trabaja la ALU y los resultados obtenidos.
- La unidad de control (UC) o ruta de control: se encarga de buscar las instrucciones en la memoria principal y guardarlas en el registro de instrucciones, las decodifica, las ejecuta empleando la ALU, guarda los resultados en el registro de datos, y guarda las condiciones derivadas de la operación realizada en el registro de estado. El registro de datos de memoria, contiene los datos que se están leyendo de la memoria principal o van a escribirse en la misma. El registro de direcciones de memoria, guarda la dirección de la memoria principal a las que esta accediendo la ALU, para leer o escribir. El contador del programa, también conocido como puntero de instrucciones, es un registro que guarda la posición en la que se encuentra la CPU dentro de la secuencia de instrucciones de un programa.
- **La unidad de memoria.** Se trata de la memoria principal o primaria del computador. Está dividida en bloques de memoria que se identifican mediante una dirección. La CPU tiene acceso directo a dichos bloques de memoria.

La unidad elemental de información digital es el bit (0,1). La capacidad de almacenamiento de datos se mide en Bytes y en sus múltiplos, calculados como potencias de 2^1

$$\begin{aligned}1 \text{ Byte} &= 8 \text{ bits} \\1 \text{ Word} &= 16 \text{ bits} = 2B \\1 \text{ KiB} &= 2^{10} \text{ bits} = 1024 \text{ B} \\1 \text{ MiB} &= 2^{20} \text{ bits} = 1024 \text{ KB} \\1 \text{ GiB} &= 2^{30} \text{ bits} \\1 \text{ TiB} &= 2^{40} \text{ bits}\end{aligned}$$

- **Unidad de Entrada/Salida.** Transfiere información entre el computador y los dispositivos periféricos.

Los elementos de interconexión se conocen con el nombre de *Buses*. Se pueden distinguir tres: En bus de datos, por el que se transfieren datos entre la CPU y la memoria ó la unidad de entrada/salida. El bus de direcciones, para especificar una dirección de memoria o del registro de E/S. Y el bus de Control, por el que se envían señales de control, tales como la señal de reloj, la señal de control de lectura/escrituras entre otras.

1.1.3. Representación binaria

Veamos con algo más de detalle, cómo representa la información un computador. Como se explicó anteriormente, La electrónica que constituye la parte física del ordenador, trabaja con dos niveles de voltaje. Esto permite definir dos estados, –alto, bajo– que pueden representarse dos símbolos 0 y 1. Habitualmente, empleamos 10 símbolos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, es decir, empleamos una representación decimal. Cuando queremos representar números mayores que nueve, dado que hemos agotado el número de dígitos disponibles, lo que hacemos es combinarlos, agrupando cantidades de diez en diez. Así por ejemplo, el numero 16, representa seis unidades más un grupo de diez unidades y el número 462 representa dos unidades más seis grupos de diez unidades más cuatro grupos de 10 grupos de 10 unidades. Matemáticamente, esto es equivalentes a emplear sumas de dígitos por potencias de diez:

$$13024 = 1 \times 10^4 + 3 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$$

Si recorremos los dígitos que componen el número de izquierda derecha, cada uno de ellos representa una potencia de diez superior, porque cada uno representa la cantidad de grupos de 10 grupos, de grupos ... de diez grupos de unidades. Esto hace que potencialmente podamos representar cantidades tan grandes como queramos, empleando tan solo diez símbolos. Esta representación, a la que estamos habituados recibe el nombre de representación en base 10 . Pero no es la única posible.

Volvamos a la representación empleada por el computador. En este caso solo tenemos dos símbolos distintos el 0 y el 1. Si queremos emplear una representación análoga a la representación en base diez, deberemos agrupar ahora las cantidad en grupos de dos. Así los únicos números que admiten ser representados con un solo dígito son el uno y el cero. Para representar el número dos,

¹Los prefijos K(Kilo),M(Mega),G (Giga), etc., se reservan en el sistema internacional para indicar potencias de 10. Para su equivalente en potencias de 2 se deben emplear los términos *Ki* (Kibi), *Mi* (Mebi),*Gi* (Gibi),*Ti*,(Tebi). Por tanto, debería decirse Kibibyte, Mebibyte, etc. Sin embargo, esta notación no está muy extendida y se habla de KB (KiloBytes), MB (Megabytes), etc aunque se realice el cálculo en potencias de 2

necesitamos agrupar: tendremos 0 unidades y 1 grupo de dos, con lo que la representación del número dos en base dos será 10. Para representar el número tres, tendremos una unidad más un grupo de dos, por lo que la representación será 11, y así sucesivamente. Matemáticamente esto es equivalente emplear sumas de dígitos por potencias de 2:

$$10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

Esta representación recibe el nombre de representación binaria o en base 2. La expansión de un número representado en binario en potencias de 2, nos da un método directo de obtener su representación decimal. Así, para el ejemplo anterior, si calculamos las potencias de dos y sumamos los resultados obtenemos:

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 16 + 0 + 4 + 2 + 0 = 22$$

que es la representación en base 10 del número binario 10110.

Para números no enteros, la representación tanto en decimal como en binario, se extiende de modo natural empleando potencias negativas de 10 y de 2 respectivamente. Así,

$$835,41 = 8 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 + 4 \times 10^{-1} + 1 \times 10^{-2}$$

y para un número en binario,

$$101,01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

De nuevo, basta calcular el término de la derecha de la expresión anterior para obtener la representación decimal del número 101,01.

¿Cómo transformar la representación de un número de decimal a binario? De nuevo nos da la clave la representación en sumas de productos de dígitos por potencias de dos. Empecemos por el caso de un número entero. Supongamos un número D, representado en decimal. Queremos expandirlo en una suma de potencias de dos. Si dividimos el número por 2, podríamos representarlo cómo:

$$D = 2 \cdot C_1 + R_1$$

donde C_1 representa el cociente de la división y R_1 el resto. Como estamos dividiendo por dos, el resto solo puede valer cero o uno. Supongamos ahora que volvemos a dividir el cociente obtenido por dos,

$$C_1 = 2 \cdot C_2 + R_2$$

Si sustituimos el valor obtenido para C_1 en la ecuación inicial obtenemos,

$$D = 2 \cdot (2 \cdot C_2 + R_2) + R_1 = 2^2 \cdot C_2 + R_2 \cdot 2^1 + R_1 \cdot 2^0$$

Si volvemos a dividir el nuevo cociente obtenido C_2 por dos, y volvemos a sustituir,

$$C_2 = 2 \cdot C_3 + R_3$$

$$D = 2^2 \cdot (2 \cdot C_3 + R_3) + R_2 \cdot 2^1 + R_1 \cdot 2^0 = 2^3 \cdot C_3 + R_3 \cdot 2^2 + R_2 \cdot 2^1 + R_1 \cdot 2^0$$

Supongamos que tras repetir este proceso n veces, obtenemos un cociente $C_n = 1$. Lógicamente no tiene sentido seguir dividiendo ya que a partir de este punto, cualquier división posterior que hagamos nos dará cociente 0 y resto igual a C_n . Por tanto,

$$D = 1 \cdot 2^n + R_n \cdot 2^{n-1} + \dots + R_3 \cdot 2^2 + R_2 \cdot 2^1 + R_1 \cdot 2^0$$

La expresión obtenida, coincide precisamente con la expansión en potencias de dos del número binario $1R_n \dots R_3R_2R_1$.

Como ejemplo, podemos obtener la representación en binario del número 234, empleando el método descrito: vamos dividiendo el número y los cocientes sucesivos entre dos, hasta obtener un cociente igual a uno y a continuación, construimos la representación binaria del número colocando por orden, de derecha a izquierda, los restos obtenidos de las sucesivas divisiones y añadiendo un uno más a la izquierda de la cifra construida con los restos:

Dividendo	Cociente $\div 2$	Resto
234	117	0
117	58	1
58	29	0
29	14	1
14	7	0
7	3	1
3	1	1

Por tanto, la representación en binario de 234 es 11101010.

Supongamos ahora un número no entero, representado en decimal, de la forma $0, d$. Si lo multiplicamos por dos:

$$E_1, d_1 = 0, d \cdot 2 \quad (1.1)$$

Donde E_1 representa la parte entera y d_1 la parte decimal del número calculado. Podemos entonces representar $0, d$ como,

$$0, d = (E_1, d_1) \cdot 2^{-1} = E_1 \cdot 2^{-1} + 0, d_1 \cdot 2^{-1} \quad (1.2)$$

Si volvemos a multiplicar $0, d_1$ por dos,

$$E_2, d_2 = 0, d_1 \cdot 2 \quad (1.3)$$

$$0, d_1 = E_2 \cdot 2^{-1} + 0, d_2 \cdot 2^{-1} \quad (1.4)$$

y sustituyendo en 1.2

$$0, d = E_1 \cdot 2^{-1} + E_2 \cdot 2^{-2} + 0, d_2 \cdot 2^{-2} \quad (1.5)$$

¿Hasta cuando repetir el proceso? En principio hasta que obtengamos un valor cero para la parte decimal, $0, d_n = 0$. Pero esta condición puede no cumplirse nunca. Puede darse el caso –de hecho es lo más probable– de que un número que tiene una representación exacta en decimal, no la tenga en binario. El criterio para detener el proceso será entonces obtener un determinado número de decimales o bien seguir el proceso hasta que la parte decimal obtenida vuelva a repetirse. Puesto que los ordenadores tienen un tamaño de registro limitado, también está limitado el número de dígitos con el que pueden representar un número decimal. Por eso, lo habitual será truncar el número asumiendo el error que se comete al proceder así. De este modo, obtenemos la expansión del número original en potencias de dos,

$$0, d \cdot 2 = E_1 \cdot 2^{-1} + E_2 \cdot 2^{-2} + \dots + E_n \cdot 2^{-3} + \dots \quad (1.6)$$

Donde los valores $E_1 \dots E_n$ son precisamente los dígitos correspondientes a la representación del número en binario: $0.E_1E_2 \dots E_n$. (Es trivial comprobar que solo pueden valer 0 ó 1).

Veamos un ejemplo de cada caso, obteniendo la representación binaria del número 0,625, que tiene representación exacta, y la del número 0,626, que no la tiene. En este segundo caso, calcularemos una representación aproximada, tomando 8 decimales.

P decimal	$\times 2$	P entera	P decimal	$\times 2$	P entera
0,625	1,25	1	0,623	1,246	1
0,25	0,5	0	0,246	0,492	0
0,5	1,0	1	0,492	0,984	0
			0,984	1,968	1
			0,968	1,936	1
			0,936	1,872	1
			0,872	1,744	1
			0,744	1,488	1

Para construir la representación binaria del primero de los números, nos basta tomar las partes enteras obtenidas, por orden, escribir las de izquierda a derecha y añadir un 0 y la coma decimal a la izquierda. Por tanto la representación binaria de 0,625 es 0,101. Si expandimos su valor en potencias de dos, volvemos a recuperar el número original en su representación decimal.

En el segundo caso, la representación binaria, tomando nueve decimales de 0,623 es 0,10011111. Podemos calcular el error que cometemos al despreciar el resto de los decimales, volviendo a convertir el resultado obtenido a su representación en base diez,

$$0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0,62109375$$

El error cometido es, en este caso: Error = 0,623 - 0,62109375 = 0,00190625.

1.2. Aplicaciones de Software Científico

Dentro del mundo de las aplicaciones, merecen una mención aparte las dedicadas al cálculo científico, por su conexión con la asignatura.

Es posible emplear lenguajes de alto nivel para construir rutinas y programas que permitan resolver directamente un determinado problema de cálculo. En este sentido, el lenguaje FORTRAN se ha empleado durante años para ese fin, y todavía sigue empleándose en muchas disciplinas científicas y de la Ingeniería. Sin embargo, hay muchos aspectos no triviales del cálculo con un computador, que obligarían al científico que tuviera que programar sus propios programas a ser a la vez un experto en computadores. Por esta razón, se han ido desarrollando aplicaciones específicas para cálculo científico que permiten al investigador centrarse en la resolución de su problema y no en el desarrollo de la herramienta adecuada para resolverlo.

En algunos casos, se trata de aplicaciones a medida, relacionadas directamente con algún área científica concreta. En otros, consisten en paquetes de funciones específicos para realizar de forma eficiente determinados cálculos, como por ejemplo el paquete SPSS para cálculo estadístico.

Un grupo especialmente interesante lo forman algunos paquetes de software que podríamos situar a mitad de camino entre los lenguajes de alto nivel y las aplicaciones: Contienen extensas librerías de funciones, que pueden ser empleadas de una forma directa para realizar cálculos y además permiten realizar programas específicos empleando su propio lenguaje. Entre estos podemos destacar Mathematica, Maple, Matlab, Octave y Scilab y Python. El uso de estas herramientas

se ha extendido enormemente en la comunidad científica. Algunas como Matlab constituyen casi un estándar en determinadas áreas de conocimiento.

Capítulo 2

Introducción a la programación en Matlab

Este capítulo presenta una introducción general a la programación. Para su desarrollo, vamos a emplear una de las aplicaciones informáticas para cálculo científico que más aceptación ha tenido en los últimos años: el entorno de programación de Matlab. Matlab es el acrónimo de MATrix LABboratory. El nombre está relacionado con el empleo de matrices como elemento básico para el cálculo numérico.

Usaremos esta herramienta, entre otras motivos, porque nos ofrece la posibilidad de realizar programas similares a los que podríamos desarrollar con un lenguaje de programación de alto nivel, nos permite resolver problemas de cálculo científico directamente, empleando las herramientas que incluye y, además, nos permite combinar ambas cosas.

Este capítulo no pretende ser exhaustivo, –cosa que por otro lado resulta imposible en el caso de Matlab–, sino tan solo dar una breve introducción a su uso. Afortunadamente, Matlab cuenta con una muy buena documentación, accesible a través de la ‘ayuda’, eso sí, en inglés.

2.1. El entorno de programación de Matlab

Cuando iniciamos Matlab en el computador, se abre una ventana formada por uno o más paneles. Esta ventana, constituye lo que en programación se llama un entorno de desarrollo integrado o, abreviadamente, IDE (acrónimo tomado de su nombre en Inglés: *integrated development environment*). El IDE de Matlab, contiene todo los elementos necesarios para programar. La figura 2.1 muestra el aspecto del IDE de Matlab.

Según como se configure, el IDE de Matlab puede mostrar un número mayor o menor de paneles y una disposición de los mismos distinta a la mostrada en la figura. La mejor manera de aprender estos y otros detalles del IDE es usarlo. Aquí nos centraremos solo en algunos aspectos fundamentales.

2.1.1. La ventana de comandos de Matlab

De los paneles mostrados en la ventana de la figura 2.1, vamos a empezar examinando el situado en la parte inferior central. Se trata de la ventana de comandos (*command window*) de Matlab. La ventana muestra el simbolo `>>`, que recibe el nombre de *prompt* y, a continuación, una barra vertical | parpadeante. La ventana de comandos permite al usuario interactuar directamente con Matlab: Matlab puede recibir instrucciones directamente a través de la ventana de comandos,

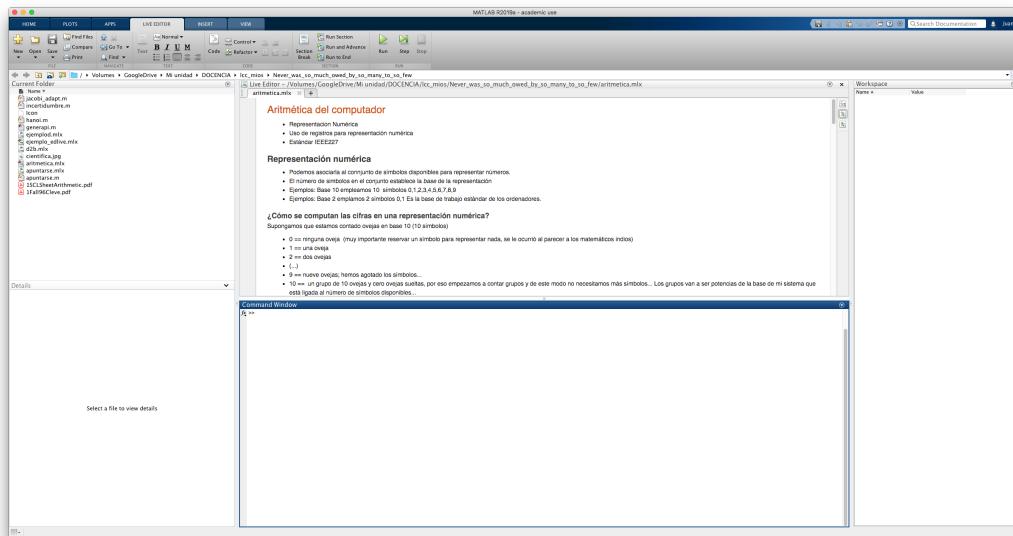


Figura 2.1: Entorno de desarrollo integrado de Matlab

ejecuta las instrucciones recibidas, y devuelve los resultados de nuevo en la ventana de comandos. Veamos un ejemplo muy sencillo: si escribimos en la ventana de comandos:

```
>> a=18 + 3
```

Matlab calcula la suma pedida, devuelve el resultado y, por último, vuelve a presentar el *prompt*, para indicarnos que está preparado para recibir otro comando.

a =

21

>>

De este modo, podemos emplear Matlab de un modo análogo a como emplearíamos una calculadora: realizamos una operación, obtenemos el resultado, realizamos otra operación, obtenemos el resultado y así sucesivamente.

2.1.2. Variables.

En el ejemplo que acabamos de ver, Matlab calcula el resultado pedido, y lo presenta en pantalla usando la expresión,

$$a = 21$$

¿Por qué hace falta escribir $a = 18 + 3$ en lugar de escribir directamente $18 + 3$? La razón tiene que ver con el modo de trabajo de Matlab, y de otros lenguajes de alto nivel. Esto nos lleva al concepto de variable.

Podemos ver una variable como una región de la memoria del computador, donde un programa guarda una determina información; números, letras, etc. Una característica fundamental de una

variable es su nombre, ya que permite identificarla. Como nombre para una variable se puede escoger cualquier combinación de letras y números, empezando siempre con una letra, en el caso de Matlab¹. Se puede además emplear el signo “_”. Matlab distingue entre mayúsculas y minúsculas, por lo que si elegimos como nombres de variable Pino, PINO y PiNo, Matlab las considerará como variables distintas.

En algunos lenguajes, es preciso indicar al ordenador qué tipo de información se guardará en una determinada variable, antes de poder emplearlas. Esto permite manejar la memoria del computador de una manera más eficiente, asignando zonas adecuadas a cada variable, en función del tamaño de la información que guardarán. A este proceso, se le conoce con el nombre de *declaración* de variables. En Matlab no es necesario declarar las variables antes de emplearlas.

El método más elemental de emplear una variable es asignarle la información para la que se creó. Para hacerlo, se emplea el símbolo de asignación =, que coincide con el signo = empleado en matemáticas. Como veremos más adelante la asignación en programación y la igualdad en matemáticas no representan exactamente lo mismo. La manera de asignar directamente información a una variable es escribir el nombre de la variable, a continuación el signo de asignación y, por último, la información asignada,

```
Nombre_variable = 4
```

Si escribimos en la ventana de comandos la expresión anterior y pulsamos el retorno de carro. Matlab devuelve el siguiente resultado:

```
>> Nombre_variable=4
```

```
Nombre_variable =
```

```
4
```

```
>>
```

Matlab ejecuta las instrucciones indicadas y nos confirma que ha creado en la memoria una variable **Nombre_variable** y que ha guardado en ella el número 4.

En Matlab podemos emplear el símbolo de asignación para construir variables que guarden distintos tipos de datos,

1. Enteros positivos y negativos

```
>> a=4
```

```
a =
```

```
4
```

```
>>b=-4
```

```
b =
```

```
-4
```

2. Números con parte entera y parte decimal separadas por un punto, positivos y negativos.

¹Como se verá más adelante, Matlab tiene un conjunto de nombres de instrucciones y comandos ya definidos. Se debe evitar emplear dichos nombres, ya que de hacerlo se pierde acceso al comando de Matlab que representan

```
>> a=13.4568
a =
```

13.4568

```
>> b=-13.4568
b =
```

-13.4568

3. Números expresados como potencias de 10 (la potencia de 10 se representa con la letra e seguida del valor del exponente).

```
>> f=3e10
f =
```

3.0000e+010

```
>> g=-3e10
g =
```

-3.0000e+010

```
>> h=3e-10
h =
```

3.0000e-010

```
>> t=-3e-10
t =
```

-3.0000e-010

4. Números complejos. Para indicar la parte imaginaria se puede emplear la letra i o la letra j.

```
>> s=2+3i
s =
```

2.0000 + 3.0000i

```
>> w=4-5j
w =
```

4.0000 - 5.0000i

5. Caracteres, letras o números; manejados estos últimos como símbolos. Se indica a Matlab que se trata de un carácter escribiéndolo entre comillas simples,

```
>> p='a'
```

```
p =
```

```
a
```

```
>> k='1'
k =
```

```
1
```

6. Cadenas de caracteres.

```
>> m='hola amigos'
```

```
m =
```

```
hola amigos
```

La forma que hemos visto de asignar un valor a una variable es la más sencilla pero no es la única. También podemos asignar un valor a una variable a partir de una expresión aritmética, como hemos visto antes. Ademas podemos asignar un valor a una variable copiando el contenido de otra variable:

```
>> a=18
```

```
a =
```

```
18
```

```
>> b=a
```

```
b =
```

```
18
```

```
>>
```

Por último, podemos asignar a una variable el valor de una función en un punto:

```
>> x=0
```

```
x =
```

```
0
```

```
>> y=cos(x)
```

```
y =
```

```
1
```

```
>>
```

La variable *y* contiene el valor de la función coseno en el punto *x=0*. Más adelante estudiaremos cómo manejar funciones en Matlab.

Si escribimos directamente en la línea de comandos de Matlab, un número, una expresión algebraica o una función, sin asignarlas a una variable, Matlab crea automáticamente una variable para guardar el resultado. Así por ejemplo:

```
>> 3 + 5
ans =
8
```

>>

Matlab guarda el resultado de la operación realizada: $3+5$, en la variable `ans`. Se trata del nombre de variable por defecto; es una abreviatura de la palabra inglesa *answer* (respuesta). En cualquier caso es recomendable asignar los resultados de las operaciones explícitamente a una variable. La razón para ello tiene que ver con lo que llamaremos reasignación de variables.

Imaginemos que creamos en Matlab una variable asignándole un valor:

```
>> a=34
a =
34
```

>>

Si a continuación, asignamos a esa misma variable el resultado de una operación,

```
>> a=12+5
a =
17
```

>>

El valor inicialmente asignado a la variable `a` se pierde. Sencillamente hemos *reasignado* a la variable un nuevo valor sobreescribiendo el anterior. Si en la línea de comandos escribimos operaciones sin asignar el resultado a una variable concreta, Matlab lo asignará a la variable `ans` pero esto significa que cada nueva operación reasigna su resultado a la variable `ans`, con lo que solo conservaremos al final el resultado de la última de las operaciones realizadas.

Es posible en Matlab crear una variable que no contenga nada. Para ello hay que emplear dos símbolos especiales: [y]. Así, si escribimos en la línea de comandos:

```
>> variable_vacia=[]
variable_vacia =
[]
```

>>

obtenemos una variable que no contiene nada. Más adelante veremos la utilidad de hacerlo.

Hasta ahora, siempre que hemos realizado una operación en la ventana de comandos, Matlab nos ha *respondido* escribiendo en pantalla el resultado de la misma. En muchas ocasiones, no nos interesa que Matlab nos muestre por pantalla el resultado de una operación; por ejemplo, porque se trata de un resultado intermedio, o porque es un resultado de gran tamaño y su visualización por pantalla no es útil y sin embargo sí que consume mucho tiempo. Podemos omitir la visualización por pantalla del resultado de una operación, si terminamos la operación, añadiendo al final un *punto y coma* (;),

```
>> A=3+5;
>> B=A+1
B =
```

9

En la primera operación hemos añadido (;) al final de la línea, Matlab no muestra el resultado. Sin embargo, sí que ha realizado la operación pedida y guardado el resultado en la variable A. Por eso es posible emplearla en la segunda operación para crear la variable B.

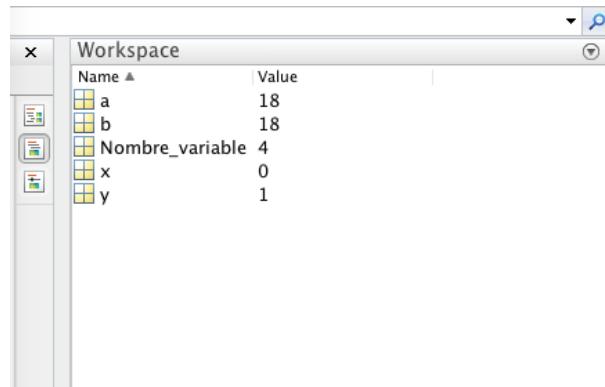
Recursión. Hemos indicado antes cómo el símbolo de asignación = en programación no coincide exactamente con la igualdad matemática. Un ejemplo claro de estas diferencias lo constituye la recursión. Esta se produce cuando la misma variable aparece a los dos lados del símbolo de asignación:

```
>> a=3
a =
3
>> a=a+1
a =
4
```

La expresión anterior no tiene sentido matemáticamente, ya que una variable no puede ser igual a sí misma más la unidad. Sin embargo, en programación, es una sentencia válida; el ordenador toma el valor almacenado en la variable *a*, le suma 1 y guarda el resultado en la variable *a*, sobreescribiendo el valor anterior.

La recursión se emplea muy a menudo en programación, entre otras aplicaciones, permite crear contadores, –variables que van incrementando o decrementando su valor progresivamente– y permite ahorrar espacio de memoria cuando se realizan operaciones que requieren cálculos intermedios.

El espacio de trabajo de Matlab *Workspace*. Matlab guarda en memoria las variables que creamos en la ventana de comandos y las asocia a lo que se conoce como el espacio de trabajo de Matlab. Dicho espacio de trabajo contiene una relación de las variables creadas de modo que podamos volver a utilizarlas en la ventana de comandos. Uno de los paneles que el IDE de Matlab puede mostrarnos es precisamente el *Workspace*. La figura 2.2 muestra dicho panel. En él se muestran los nombres de las variables contenidas en el espacio de trabajo, así como información relativa a su valor, tamaño en memoria etc.

Figura 2.2: El *Workspace* de Matlab

Además del panel que acabamos de describir, es posible listar el contenido de las variables presentes en el *Workspace* empleando dos comandos especiales de Matlab; se trata de los comandos `who` y `whos`. El primero de ellos nos devuelve en la ventana de comandos los nombres de las variables contenidas en el *Workspace*. El segundo nos devuelve los nombres de las variables junto con información adicional sobre su contenido, tamaño, etc.

```
>> who
```

```
Your variables are:
```

```
Nombre_variable  b          y
a              x
```

```
>> whos
```

Name	Size	Bytes	Class	Attributes
Nombre_variable	1x1	8	double	
a	1x1	8	double	
b	1x1	8	double	
x	1x1	8	double	
y	1x1	8	double	

```
>>
```

Para eliminar una variable del *Workspace*, se emplea el comando `clear`. Si escribimos en la ventana de comandos el comando `clear`, seguido del nombre de una variables, Matlab la elimina del *Workspace*. Si escribimos el comando `clear`, sin añadir nada más, Matlab eliminará TODAS las variables contenidas en el *Workspace*.

Formatos de visualización Hemos visto en los ejemplos anteriores cómo al realizar una operación en Matlab, se nos muestra el resultado en la ventana de comandos. Además podemos examinar el contenido de cualquier variable contenida en el *workspace* sin más que escribir su nombre en la ventana de comandos y pulsar la tecla *intro*.

Matlab permite elegir la forma en que los resultados se presenta por pantalla. Para ello se emplea el comando `format`. La siguiente tabla, resume los formatos más comúnmente empleados.

Comando	formato	Cuadro 2.1: Formatos numéricos más comunes en Matlab Comentario
format short	12,3457	coma fija. Cuatro decimales
format shortE	1,2346e + 01	coma flotante. Cuatro decimales
format long	12,345678901234500	coma fija. Quince decimales
format longE	1,234567890123450e + 01	coma flotante. Quince decimales

2.1.3. Vectores y matrices.

Una de las características más interesantes de Matlab, es la posibilidad de crear fácilmente matrices. Se pueden crear de muchas maneras, la más elemental de todas ellas, emplea el operador de asignación `=`, y los símbolos especiales `[]`, el punto y coma `;` y la coma `,`. Las matrices se crean introduciendo los valores de sus elementos por filas, separados por comas o espacios. Una vez introducidos todos los elementos de una fila, se añade un punto y coma, o se pulsa la tecla *intro*, y se añaden los elementos de la fila siguiente. El siguiente ejemplo muestra como crear una matriz de dos filas y tres columnas:

```
>> matriz23 =[ 1 3 4 ;3 5 -1]

matriz23 =
```

```
1      3      4
3      5     -1
```

o también:

```
>> matriz23 =[ 1 3 4
3 5 -1]
```

```
matriz23 =
```

```
1      3      4
3      5     -1
```

En el primer caso, se empleó el punto y coma para separar las filas y en el segundo se ha empleado la tecla *intro*. En ambos se emplea el símbolo `[` para indicar a Matlab que queremos empezar a construir una matriz, y el símbolo `]` para indicar a Matlab que hemos terminado de construirla. Una vez construida, Matlab nos devuelve en la ventana de comandos la Matriz completa. Matlab nos permite además emplear cada elemento de una matriz como si se tratase de una variable, es decir, se puede asignar a los elementos de una matriz un valor numérico, el resultado de una operación o un valor guardado en otra variable:

```
>> a=1
```

```
a =
```

```
1.00
```

```
>> b=2
```

```
b =
2.00
>> mtr=[ a a+b a-b; 1 0.5 cos(0)]
mtr =
1.00      3.00      -1.00
1.00      0.50      1.00
>>
```

Matlab considera las matrices como la forma básica de sus variables, así para Matlab un escalar es una matriz de una fila por una columna. Un vector *fila* de 3 elementos es una matriz de una fila por tres columnas y un vector *columna* de tres elementos es una matriz de tres filas y una columna.

Indexación. Al igual que se hace en álgebra, Matlab es capaz de referirse a un elemento cualquiera de una matriz empleando índices para determinar su posición (fila y columna) dentro de la matriz.

$$a = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

El criterio para referirse a un elemento concreto de una matriz, en Matlab es el mismo: se indica el nombre de la variable que contiene la matriz y a continuación, entre paréntesis y separados por una coma, el índice de su fila y después él de su columna:

```
>> a=[1 2 3; 4 5 6; 7 8 9]
a =
1.00      2.00      3.00
4.00      5.00      6.00
7.00      8.00      9.00
>> a(1,2)
ans =
2.00
>> a(2,1)
ans =
4.00
>>
```

Es interesante observar de nuevo cómo Matlab asigna por defecto el valor del elemento buscado a la variable `ans`. Como ya se ha dicho, es mejor asignar siempre una variable a los resultados, para asegurarnos de que no los perdemos al realizar nuevas operaciones:

```
>> a12=a(1,2)
```

```
a12 =
```

```
2.00
```

```
>> a21=a(2,1)
```

```
a21 =
```

```
4.00
```

```
>>
```

Ahora hemos creado dos variables nuevas que contienen los valores de los elementos a_{12} y a_{21} de la matriz a .

Matlab puede seleccionar dentro de una matriz no solo elementos aislados, sino también submatrices completas. Para ello, emplea un símbolo reservado, el símbolo *dos puntos* `:`. Este símbolo se emplea para recorrer valores desde un valor inicial hasta un valor final, con un incremento o paso fijo. La sintaxis es: `inicio:paso:fin`, por ejemplo podemos recorrer los números enteros de cero a 8 empleando un paso 2:

```
>> 0:2:8
```

```
ans =
```

0	2.00	4.00	6.00	8.00
---	------	------	------	------

```
>>
```

El resultado nos da la lista de los números 0, 2, 4, 6, 8. Además, si no indicamos el tamaño del paso, Matlab tomará por defecto un paso igual a uno. En este caso basta emplear un único símbolo *dos puntos* para separar el valor de inicio del valor final:

```
>> 1:5
```

```
ans =
```

1.00	2.00	3.00	4.00	5.00
------	------	------	------	------

```
>>
```

Podemos emplear el símbolo *dos puntos*, para obtener submatrices de una matriz dada. Así por ejemplo si construimos una matriz de cuatro filas por cinco columnas:

```
>> matriz=[1 2 4 5 6  
3 5 -6 0 2
```

```

4 5 8 9 0
3 3 -1 2 0]

matriz =

    1.00      2.00      4.00      5.00      6.00
    3.00      5.00     -6.00      0.00      2.00
    4.00      5.00      8.00      9.00      0.00
    3.00      3.00     -1.00      2.00      0.00

```

>>

Podemos obtener el vector formado por los tres últimos elementos de su segunda fila:

```

>> fil=matriz(2,3:5)

fil =

    -6.00      0      2.00

>>

```

o la submatriz de tres filas por tres columnas formada por los elementos que ocupan las filas 2 a 4 y las columnas 3 a 5:

```

>> subm=matriz(2:4,3:5)

subm =

    -6.00      0      2.00
     8.00     9.00      0
    -1.00     2.00      0

```

>>

o el vector columna formado por su segunda columna completa:

```

>> matriz(1:4,2)

ans =

    2.00
    5.00
    5.00
    3.00

>>

```

De hecho, si deseamos seleccionar todos los elementos en una fila o una columna, podemos emplear el símbolo : directamente sin indicar principio ni fin,

```
>> matriz(:,2)

ans =

    2.00
    5.00
    5.00
    3.00

>> matriz(3,:)
ans =

    4.00    5.00    8.00    9.00    0.00
```

A parte de la indexación típica del álgebra de los elementos de una matriz indicando su fila y columna, en Matlab es posible referirse a un elemento de una matriz empleando un único índice. En este caso, Matlab cuenta los elementos por columnas, de arriba abajo y de izquierda a derecha,

$$A = \begin{pmatrix} a_1 & a_4 & a_7 \\ a_2 & a_5 & a_8 \\ a_3 & a_6 & a_9 \end{pmatrix}$$

Así por ejemplo, en una matriz A de 3 filas y 4 columnas,

```
>> A=[3 0 -1 0; 2 1 5 7; 1 3 9 8]
A =
```

```
3      0      -1      0
2      1       5      7
1      3       9      8
```

las expresiones,

```
>> A(2,3)
```

```
ans =
5
```

y

```
>> A(8)
```

```
ans =
5
```

hacén referencia al mismo elemento de la matriz A .

Aunque el concepto de función no se explicará hasta la sección 2.4, vamos a hacer uso de un par de funciones sencillas relacionadas con el tamaño de una matriz. En primer lugar, tenemos la función `length`; esta función nos calcula el número total de elementos que contiene un vector, sea este fila o columna. Así, si tenemos un vector guardado en la variable `a`, para saber su longitud escribimos en matlab el nombre de la función seguida del nombre de la variable entre paréntesis `length(a)`,

```
>> a = [ 1 -2 0 6 8]
a =
1   -2    0    6    8
>> length(a)
ans =
5
```

>>

La segunda función es la función `size`, esta función permite obtener el número de filas y columnas de una matriz o vector. `size` nos da como resultado de aplicarlo a una matriz un vector cuyo primer elemento es el número de filas de la matriz, y cuyo segundo elemento el número de columnas,

```
>> A = [1 3 4 -5; 2 3 0 -2; -2 1 7 7]
A =
1   3   4   -5
2   3   0   -2
-2   1   7   7
>> size(A)
ans =
3   4
```

2.1.4. Estructuras y células

Se trata de dos tipos de variables especiales. Ambas comparten la propiedad de poder combinar dentro de sí variables de distintos tipos.

Estructuras. Una estructura es una variable que guarda la información dividida en campos. Por ejemplo, si escribimos en la ventana de matlab,

```
>> est.nombre='Ana'
est =
    nombre: 'Ana'
>> est.edad=25
est =
    nombre: 'Ana'
    edad: 25
```

obtenemos una estructura con dos campos, el primero de ellos es el campo `nombre`, y guarda dentro una cadena de caractéres '`Ana`', el segundo es el campo `edad` y guarda dentro el valor 25. La estructura que acabamos de definir es una sola variable llamada `est` y podemos aplicar cualquier comando de matlab cuyo resultado no dependa del contenido específico de la variable. Podemos copiarla en otra estructura,

```
>> est2=est
est2 =
    nombre: 'ana'
    edad: 25
```

podemos borrarla,

```
>> clear est
>> who
Your variables are:
est2
>>
```

No podemos realizar sobre ella, como un todo, operaciones aritméticas o relacionales, pero sí sobre sus campos,

```
>> x=est.edad+12
x =
    37
```

El número de campos de una estructura puede aumentarse añadiendo a su nombre, el nombre del nuevo campo separado por un punto y asignando un valor o una variable a dicho campo.

```
>> est.campo_nuevo=[1 2;3 4; 6 7]
est =
    nombre: 'ana'
    edad: 25
    campo_nuevo: [3x2 double]
    >> y=[1 2 3]
y =
    1      2      3
>> est.campo_nuevo2=y
est =
    nombre: 'ana'
    edad: 25
    campo_nuevo: [3x2 double]
    campo_nuevo2: [1 2 3]
```

Podemos tambien eliminar campos de una estructura mediante el comando `rmfield`,

```
>> est=rmfield(est,'edad')
est =
    nombre: 'ana'
    campo_nuevo: [3x2 double]
    campo_nuevo2: [1 2 3]
```

Por último, una estructura nos permite definir y utilizar varios niveles de campos. Para ello, basta ir definiendo los nombres de los campos de un nivel separados por un punto del nombre del nivel anterior,

```
>> multnivel.datos_personales.nombre='Ana'
multnivel =
    datos_personales: [1x1 struct]
>> multnivel.datos_personales.primer_apellido='Jiménez'
multnivel =
    datos_personales: [1x1 struct]
>> multnivel.datos_personales.segundo_apellido='Lasarte'
multnivel =
    datos_personales: [1x1 struct]
>> multnivel.domicilio.calle='Ponzano'
multnivel =
    datos_personales: [1x1 struct]
    domicilio: [1x1 struct]
>> multnivel.domicilio.numero=724
multnivel =
    datos_personales: [1x1 struct]
    domicilio: [1x1 struct]
>> multnivel.valor=[3 4 5; 3.5 2 3]
multnivel =
    datos_personales: [1x1 struct]
    domicilio: [1x1 struct]
    valor: [2x3 double]
```

La información se encuentra ahora estructurada en niveles. Así por ejemplo,

```
>> multnivel.domicilio
ans =
    calle: 'Ponzano'
    numero: 724
```

Me devuelve el contenido del campo `domicilio` que es a su vez una estructura formada por dos campos `calle` y `numero`. La información queda estructurada en niveles que pueden ramificarse tanto como se desee. Para obtener la información contenida al final de una rama, es preciso indicar todos los campos que se atraviesan hasta llegar a ella,

```
>> multnivel.datos_personales.segundo_apellido
ans =
Lasarte
```

Matlab tiene definidas funciones propias para conseguir un manejo eficiente de las estructuras. A parte de la función `rmfield` de la que hemos hablado anteriormente, cabe destacar la función `struct` que permite crear directamente una estructura. Su sintaxis es,

```
s= struct('field1', values1, 'field2', values2, ...)
```

donde `s` representa el nombre de la estructura, `field1, field2`, etc son los nombres correspondientes a cada campo, introducidos entre comillas, y `values1`, etc los valores contenidos en cada campo. Para un conocimiento más profundo del uso de las estructuras se aconseja consultar la ayuda de Matlab.

Células. Las células, *cells* en Matlab, son objetos que almacenan datos de diversos tipos en una misma variable. A diferencia de las estructuras, las células no expanden un árbol de campos sino que guardan cada dato en una “celda”. Para referirse a una celda concreta, empleamos un índice, de modo análogo a como hacemos con un vector. Para construir una célula, procedemos de modo análogo a como hacemos con un vector, pero en lugar de emplear corchetes como delimitadores, empleamos llaves. Así, por ejemplo,

```
>> a={[1 2 3; 4 5 6; 7 8 9], 'cadena', -45; 'pepe', [1 2 3], 'cadena2'}
```

```
a =
```

[3x3 double]	'cadena'	[-45]
'pepe'	[1x3 double]	'cadena2'

Hemos creado una célula *a*, cuyos elementos son, una matriz 3×3 , la cadena de caracteres *cadena*, el número -45 , otra cadena de caracteres *pepe*, el vector $[1, 2, 3]$ y una última cadena de caracteres *cadena2*.

Los elementos separados por espacios o por comas simples, pertenecen a la misma fila dentro de la célula. Los elementos pertenecientes a distintas filas están separados por un punto y coma. Podemos obtener el tamaño de la célula —su número de filas y columnas— empleando el comando *size*, igual que hicimos para el caso de vectores o matrices,

```
>> size(a)
```

```
ans =
```

2	3
---	---

Y podemos referirnos a una celda cualquiera de la célula, y obtener su contenido indicando entre llaves la fila y la columna a la que pertenece. Así por ejemplo, para obtener el vector $[1, 2, 3]$ de la célula *a* del ejemplo anterior hacemos,

```
>> vector=a{2,2}
```

```
vector =
```

1	2	3
---	---	---

Las celdas, de modo análogo a como sucedía con las estructuras, nos permiten agrupar datos de distinto tipo empleando una sola variable. Además, el hecho de que los datos estén ordenados por celdas en filas y columna, hace sencillo que se puede acceder a ellos.

2.2. Entrada y salida de Datos

Matlab posee una amplia colección de métodos para importar datos desde y exportar datos a un fichero. A continuación describimos algunos de los más usuales.

2.2.1. Exportar e importar datos en Matlab

Datos en formato propio de Matlab Matlab posee un formato de archivo propio para manejar sus datos. Los archivos de datos propios de Matlab, emplean la extensión *.mat*.

Si tenemos un conjunto de variables en el *workspace*, podemos guardarlas en un fichero empleando el comando *save*, seguido del nombre del fichero donde queremos guardarlos. No es preciso incluir la extensión *.mat*, Matlab la añade automáticamente:

```
>> save datos
```

Matlab creará en el directorio de trabajo un nuevo fichero `datos.mat` en el que quedarán guardadas todas las variables contenidas en el *workspace*. Los ficheros `.mat` generados por Matlab están escritos en binario. No se puede examinar su contenido empleando un editor de textos. Matlab almacena toda la información necesaria —nombre de las variables, tipo, etc— para poder volver a reconstruir las variables en el *workspace* tal y como estaban cuando se generó el archivo.

Es posible guardar tan solo algunas de las variables contenidas en el *workspace* en lugar de guardarlas todas. Para ello, basta añadir al comando `save`, detrás del nombre del archivo, el nombre de las variables que se desean guardar, separadas entre sí por un espacio

```
>>save datos1 A matriz_1 B
```

La instrucción anterior guarda en un archivo —llamado `datos1.mat`— las variables `A`, `matriz1` y `B`.

Los datos contenidos en cualquier fichero `.mat` generado con el comando `save` de Matlab, pueden volver a cargarse en el *workspace* empleando el comando `load`, seguido del nombre del fichero cuyos datos se desean cargar:

```
load datos.mat
```

carga en el *workspace* las variables contenidas en el fichero `datos.mat`, la extensión del fichero puede omitirse al emplear la función `load`. Si conocemos los nombres de las variables contenidas en un archivo `.mat`, podemos cargar en Matlab solo una o varias de las variables contenidas en el archivo, escribiendo sus nombres, separados por espacios, detrás del nombre del archivo:

```
>>load datos A G matriz1
```

cargará tan solo las variables `A`, `G` y `matriz1`, de entre las que contenga el fichero `datos.mat`

Datos en Formato ASCII e puede emplear también el comando `save` para exportar datos en formato ASCII. Pare ello es preciso añadir al comando modificadores,

```
>>save datos.txt A -ASCII
```

Este comando guardará la variable `A` en el fichero `datos.txt`. Matlab no añade ninguna extensión por defecto al nombre del fichero cuando empleamos el comando `save` con el modificador `-ASCII`. En este caso, hemos añadido explícitamente la extensión `.txt`, esto facilita que el archivo resultante se pueda examinar luego empleando un sencillo editor de texto o una hoja de cálculo.

Cuando se exportan datos desde Matlab en formato ASCII, Matlab guarda tan solo los valores numéricos contenidos en las variables, pero no los nombres de éstas. Por otro lado, guarda tan solo ocho dígitos, por lo que habitualmente se pierde precisión. Es posible guardar datos en formato ASCII, conservando toda la precisión, si añadimos al comando `save` el modificador `-DOUBLE`,

```
>>save nombre_Archivo matriz_1, matriz2, ... -ASCII -DOUBLE
```

Supongamos que en *workspace* tenemos guardada la siguiente matriz,

$$a = \begin{pmatrix} 1,300236890000000e + 000 & 3,456983000000000e + 000 & 4,321678000000000e + 006 \\ 4,000230000000000e + 003 & 1,245677000000000e + 001 & 1,231565670000000e + 002 \end{pmatrix}$$

Si ejecutamos en Matlab,

```
>>save datos.txt a -ASCII
```

El fichero resultante tendrá el aspecto siguiente,

```
1.3002369e+00 3.4569830e+00 4.3216780e+05
4.0002300e+03 1.2456770e+01 1.2315657e+02
```

es decir, los elementos de una misma fila de la matriz a se guardan en una fila separados por espacios, las filas de la matriz se separan empleando retornos de carro —cada una ocupa una línea nueva— y los valores cuyos dígitos significativos exceden de 8 se han truncado, redondeando el último dígito representado. Este último es el caso de los elementos a_{11} y a_{33} de la matriz del ejemplo.

Cuando se guardan varias variables o todo el *workspace* en un mismo fichero con formato ASCII, es preciso tener en cuenta que Matlab se limitará a guardar los contenidos de las variables, uno debajo de otro, en el orden en que las escribamos detrás del comando **save** (en el caso de que guardemos todas las variables del *workspace* las guardará una debajo de otra por orden alfabético), por lo que resulta difícil distinguir las variables originales. Así por ejemplo, si tenemos en el *workspace* las variables,

$$A = \begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 8 & 0 \end{pmatrix} a = \begin{pmatrix} 1 & 3 & 4 \\ 4 & 5,6 & 2 \\ 3 & 0 & 1 \end{pmatrix} c = (3 \quad 2)$$

La orden,

```
>>save datos.txt c a A -ASCII
```

produce el un archivo con el siguiente contenido,

```
3.0000000e+00 2.0000000e+00
1.0000000e+00 3.0000000e+00 4.0000000e+00
4.0000000e+00 5.6000000e+00 2.0000000e+00
3.0000000e+00 0.0000000e+00 1.0000000e+00
3.0000000e+00 5.0000000e+00
2.0000000e+00 1.0000000e+00
8.0000000e+00 0.0000000e+00
```

El comando **load**, presenta algunas limitaciones para cargar datos contenidos en un fichero ASCII. Solo funciona correctamente si el contenido del fichero puede cargarse en una única matriz, es decir, cada fila de datos en el fichero debe contener el mismo número de datos. Así, en el ejemplo anterior, los datos guardados en el fichero **datos.txt**, no pueden volver a cargarse en Matlab usando el comando **load**.

Para el caso de un fichero cuyos contenido pueda adaptarse a una matriz, el comando **load** carga todos los datos en una única matriz a la que asigna el nombre del fichero sin extensión.

Lectura y escritura de datos con formato Matlab puede también escribir y leer datos, empleando formatos y procedimientos similares a los de el lenguaje C. Para ello, es preciso emplear varios comandos²:

En primer lugar es preciso crear —o si ya existe abrir— el archivo en que se quiere guardar o del que se quieren leer los datos. En ambos casos se emplea para ello el comando **fopen**. La sintaxis de este comando es de la forma,

```
fid=fopen(nombre de fichero, permisos)
```

²Lo que se ofrece a continuación es solo un resumen del uso de los comandos y formatos más frecuentes. Para obtener una información completa, consultar la ayuda de Matlab.

El nombre del fichero debe ser una cadena de caracteres, es decir, debe ir escrito entre apóstrofos. Hay al menos ocho tipos de permisos distintos. Aquí describiremos tan solo tres de ellos, '**w**' —*write*— abre o crea un archivo con permiso de escritura. Si el archivo ya existía su contenido anterior se sobreescribe. Si se quiere añadir datos a un archivo sin perder su contenido se emplea el permiso '**a**' —*append*— los nuevos datos introducidos se escriben al final del fichero, a continuación de los ya existentes. '**r**' —*read*— permiso de lectura, es la opción por defecto, permite leer el contenido de un archivo. Cuando se trabaja en el sistema operativo *Windows*, es común emplear los permisos en la forma '**wt**' y '**rt**', de este modo, los archivos se manejan en el denominado formato 'texto'. La variable **fid**, es un identificador del fichero abierto y es también la forma de referirnos a él —en un programa, o en la línea de comandos— mientras permanece abierto.

Supongamos que hemos abierto —o creado— un archivo para escribir en él, datos contenidos en el *workspace*,

```
fichero1=fopen('mi_fichero','wt')
```

Para escribir en él emplearemos el comando **fprintf**. La sintaxis de este comando necesita un identificador de archivo —en nuestro caso sería **fichero1**—, un descriptor del formato con el que se desean guardar los datos y el nombre de la variable que se desea guardar.

```
control=fprintf(fichero1,formato,nombre_de_variable,...)
```

Los descriptores de formato se escriben entre apóstrofos. Empiezan siempre con el carácter %, seguido de un número con formato *e.d*. Donde *e* recibe el nombre de anchura de campo y *d* recibe el nombre de precisión. Por último se añade un carácter, conocido como carácter de conversión, que permite ajustar el formato numérico de los datos. Los más usuales son: **f** —notación de punto fijo—, **e** —notación exponencial— y **g** —la notación que resulte más compacta de las dos anteriores—. La anchura de campo representa en número mínimo de caracteres que se emplearán para representar el número. La precisión depende del carácter de conversión; para **f** y **e**, representa el número de dígitos a la derecha del punto decimal, para **g** el número de dígitos significativos.

Aquí hemos incluido solo los caracteres de conversión más usuales para el caso de datos numéricos. Cabe añadir que para el caso de una cadena de caracteres se emplea como carácter de conversión la letra **s**. Por ejemplo:

```
>>A='mi cadena de caracteres'
>>num=fopen(fid,'%s', A)
```

Escribe el texto contenido en el vector **A** en el archivo indicado por **fid**.

Matlab, almacena los datos consecutivamente uno detrás de otro. Si se trata de matrices, los va leyendo por columnas y una variable tras otra en el orden en que se hayan introducido al llamar a la función **fprintf**. Para separar entre sí los datos se puede añadir a los descriptores de formatos, espacios y también caracteres de escape como Retornos de carro \r, indicadores de salto de línea \n o tabuladores \t, entre otros.

Por ejemplo, supongamos que tenemos en Matlab las siguientes matrices

$$A = \begin{pmatrix} 3,25 & 5,22 \\ 23,1 & 130,5 \\ 8 & 0 \end{pmatrix} a = \begin{pmatrix} 1,2345 & 3,0879 & 4234,2 \\ 40 & 5000,6 & 223 \\ 3 & 0 & 1 \end{pmatrix} c = \begin{pmatrix} 30 & 2 \end{pmatrix}$$

Si empleamos,

```
>>num=fopen(fid,'%2.1f', A, a, c)
```

Los datos se guardarán en el archivo indicado por **fid** en la siguiente forma:

```
3.223.18.05.2130.50.01.240.03.03.15000.60.04234.2223.01.030.02.0
```

Guardados en este formato resulta bastante difícil reconocerlos. Si probamos,

```
>>num=fprintf(fid,'%2.1f ', A, a, c)
```

El resultado sería,

```
3.2 23.1 8.0 5.2 130.5 0.0 1.2 40.0 3.0 3.1 5000.6 0.0 4234.2 223.0 1.0 30.0 2.0
```

El formato es de punto fijo y los datos aparecen separados por un espacio —nótese que en el descriptor de formato se ha incluido un espacio entre la f y el apóstrofo—. Los datos de las tres matrices aparecen uno detrás de otro, y se han ido escribiendo en el archivo por columnas

Si cambiamos de nuevo el descriptor de formato,

```
>>num=fprintf(fid,'%2.3g\n', A, a c)
```

obtendremos,

```
3.25
23.1
8
5.22
130
0
1.23
40
3
3.09
5e+03
0
4.23e+03
223
1
30
```

Matlab elige el formato más compacto para cada dato, y guarda cada dato en una fila nueva, debido al término \n introducido al final del descriptor.

Por último indicar que es posible emplear varios descriptores consecutivos, en cuyo caso, Matlab los aplica a cada dato consecutivamente, cuando ha terminado con la lista de descriptores, comienza de nuevo por el principio.

Por ejemplo,

```
>>num=fprintf(fid,'%2.3g %2.3g %2.3g\n', a)
```

Guarda los datos contenidos en a como,

```
1.23 40 3.09
5e+03 4.23e+03 223
```

Es decir cada tres datos cambia a una línea nueva. Por supuesto, podríamos hacer que los datos se guardaran con un formato distinto en cada caso,

```
>>num=fprintf(fid,'%2.3g %3f %2.3f\n', a)
```

```
1.23 40.000000 3.088
5e+03 4234.200000 223.000
```

Por último indicar, que si se emplea el comando `fprintf` sin emplear un identificador de archivo o empleando como identificador el valor 1. Matlab escribe el resultado directamente en la ventana de comandos con el formato deseado.

Por ejemplo,

```
>>B = [8.8 7.7; 8800 7700];
>>fprintf(1, 'X is %.2f metros o %8.3f mm\n', 9.9, 9900, B)
```

Mostrará en la ventana de comandos las siguientes líneas:

```
X is 9.90 metros o 9900.000 mm
X is 8.80 metros o 8800.000 mm
X is 7.70 metros o 7700.000 mm
```

Para cargar archivos desde un fichero que contiene datos en binario, se emplea el comando `fscanf`. Su uso es similar al de `fprintf`, pero ahora los datos pasarán desde el fichero donde están guardados a una matriz.

```
>>A=fscanf(pid,'%f')
```

El comando `fscanf`, admite como parámetros el número máximo de datos que se leerán del fichero,

```
a=fscanf(fid,'%5.2f',M)
```

Lee como máximo M datos del fichero y los guarda en un vector `a`. Es posible dar a los datos formato de matriz, mediante dos parámetros (fila y columna),

```
a=fscanf(fid,'%5.2f',[M,N])
```

Ahora creará una matriz `a` de M filas por N columnas. Matlab irá cogiendo los datos del fichero, por columnas hasta llenar la matriz.

Es posible dar al segundo parámetro el valor `inf`. De este modo, Matlab creará una matriz de M filas y el número de columnas necesario para cargar todos los datos del fichero. (Si faltan elementos para completar la matriz resultante, Matlab rellena los huecos con ceros.) Por último, una vez que se han escrito o leído datos en el fichero es preciso cerrarlo correctamente empleando el comando `fclose`, la sintaxis de este comando solo precisa que se incluya el identificador del fichero que se quiere cerrar,

```
>>fclose(fid)
```

Herramienta de importación de datos Matlab posee una herramienta especial para importar datos. Con ella, es posible cargar en Matlab datos de muy diverso tipo, no sólo numéricos sino también imágenes, sonido, etc. Una de las ventajas de esta herramienta es que reconoce directamente —entre otros— los archivos creados por las hojas de cálculo.

Para abrir en Matlab la herramienta de importación, basta pulsar en la pestaña *home*, situada en la parte superior del IDE de Matlab, el botón *Import Data*. Matlab abre entonces una ventana que nos permite navegar por el árbol de directorio y seleccionar el archivo del que deseamos importar los datos. Una vez seleccionado, Matlab abre la ventana mostrada en la imagen 2.3. Se trata de un programa especial —*import wizard*— que sirve para guiar al usuario en el proceso de cargar las variables contenidas en el fichero en el *workspace* de Matlab.

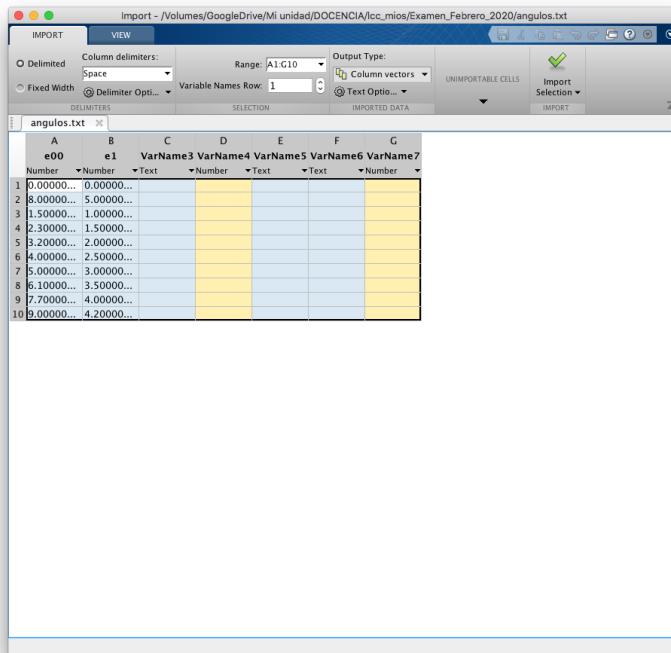


Figura 2.3: Aspecto de la herramienta de importación de datos

2.3. Operaciones aritméticas, relaciones y lógicas

2.3.1. Operaciones aritméticas

Una vez que sabemos como crear o importar variables en Matlab, vamos a ver como podemos realizar operaciones aritméticas elementales con ellas. La sintaxis es muy sencilla, y podemos sintetizarla de la siguiente manera:

$$\text{resultado} = \text{operando}_1 \text{operador}_1 \text{operando}_2 \text{operador}_2 \text{operando}_3 \cdots \text{operando}_{n-1} \text{operador}_n$$

Es decir basta concatenar los operadores con los operandos y definir una variable en la que guardar el resultado. Por ejemplo,

```
>>x=1; y=2; z=3; q=x+y+z
>>z=6
```

En este caso los operandos son las variables x , y , z , el operador, que se repite dos veces es el símbolo $+$ que representa la operación suma y q es la variable en la que se guarda el resultado, en este caso, de la suma de las tres variables anteriores.

Los operadores aritméticos disponibles en Matlab cubren las operaciones aritméticas habituales, pero hay que recordar que Matlab considera sus variables como matrices. Por lo tanto, las operaciones definidas Matlab las considera por defecto operaciones entre matrices. La tabla 2.2 contiene los operadores definidos en Matlab.

A continuación, veremos algunos ejemplos de manejo de operaciones básicas. Hemos visto ya el manejo de la suma. Si se trata de matrices en vez de números,

Cuadro 2.2: Operadores aritméticos definidos en Matlab

operación	símbolo	ejemplo	notas
-----------	---------	---------	-------

suma	+	$r=a+b$	suma matricial
diferencia	-		
producto	*	$r=a*b$	producto matricial
producto	.*	$r=a.*b$	producto por elementos
división	/	$d=a/b$	división: $a \cdot b^{-1}$
división	./	$d=a./b$	división por elementos
división	\	$d=a\b$	división por la izquierda: $a^{-1} \cdot b$
potenciación	^	$y=a ^ b$	potencia de una matriz
potenciación	.^	$y=a .^ b$	potencia elemento a elemento
trasposición	,	$y=a'$	matriz traspuesta

```
>> A=[1 2 3; 4 5 6; 7 8 9]
```

A =

```
1     2     3
4     5     6
7     8     9
```

```
>> B=[4 5 6; 2 0 3; -1 2 4]
```

B =

```
4     5     6
2     0     3
-1    2     4
```

```
>> C=A+B
```

C =

```
5     7     9
6     5     9
6    10    13
```

Por supuesto, hay que respetar las condiciones en que es posible realizar una operación aritmética entre matrices. La operación,

```
>> A=[1 2 3; 4 5 6; 7 8 9]
```

A =

```
1     2     3
4     5     6
7     8     9
```

```
>> B=[4 5 6; 2 0 3]
```

```
B =
4      5      6
2      0      3

>> C=A+B
Error using +
Matrix dimensions must agree.
```

da un error porque solo es posible sumar matrices del mismo tamaño.³

En el caso del producto, Matlab define dos operaciones distintas. La primera es el producto matricial normal,

```
>> A=[1 2 3; 4 5 6; 7 8 9]
A =
```

```
1      2      3
4      5      6
7      8      9
```

```
>> B=[3 4; -2 1; 6 7]
B =
```

```
3      4
-2     1
6      7
```

```
>> P=A*B
P =
```

```
17     27
38     63
59     99
```

El segundo producto, no es propiamente una operación aritmética definida sobre matrices. Se trata de un producto realizado elemento a elemento. Para ello los dos factores deben ser matrices del mismo tamaño. El resultado es una nueva matriz de igual tamaño que las iniciales en la que cada elemento es el producto de los elementos que ocupaban la misma posición en las matrices factores,

```
>> A = [1 2 3; 4 5 6]
```

```
A =
```

```
1      2      3
4      5      6
```

```
>> B = [2 3 4; 1 2 3]
```

³Ver las definiciones de las operaciones matriciales en el capítulo 5.2.

```
B =
2      3      4
1      2      3
```

```
>> C = A.*B
```

```
C =
2      6     12
4     10     18
```

En general, cualquier operador al que se antepone un punto `.*` `./` `.^` indica una operación realizada elemento a elemento.

La división no está definida para matrices. Sin embargo, en Matlab hay definidas tres divisiones. La primera, emplea el símbolo clásico de división, para simples números realiza la división normal. Para matrices, la operación es equivalente a multiplicar el primer operando por la matriz inversa del segundo. $A/B \equiv A \cdot B^{-1}$. Las siguientes tres operaciones son equivalentes en Matlab, aunque desde el punto de vista del cálculo empleado para obtener el resultado, son numéricamente distintas,

```
>> A=[1 2 3; 4 5 6; 7 8 9]
```

```
A =

```

```
1      2      3
4      5      6
7      8      9
```

```
>> B=[2 0 1; 3 2 -1; -2 0 2]
```

```
B =

```

```
2      0      1
3      2     -1
-2     0      2
```

```
>> C=A/B
```

```
C =

```

```
0.6667    1.0000    1.6667
1.6667    2.5000    3.4167
2.6667    4.0000    5.1667
```

```
>> C=A*inv(B)
```

```
C =

```

```
0.6667    1.0000    1.6667
1.6667    2.5000    3.4167
2.6667    4.0000    5.1667
```

```
>> C=A*B^-1
```

```
C =
0.6667    1.0000    1.6667
1.6667    2.5000    3.4167
2.6667    4.0000    5.1667
```

En el primer caso se ha empleado la división, en el segundo se ha multiplicado la matriz A por la inversa de B, calculándola mediante la función de Matlab `inv`, y en el tercer caso se ha multiplicado la matriz A por B elevado a -1 .

La división elemento a elemento, funciona de modo análogo a como lo hace la multiplicación elemento a elemento.

La división por la izquierda, representada mediante el símbolo \ es equivalente a multiplicar la inversa del primer operando por el segundo, $A \setminus B \equiv A^{-1} \cdot B$.

```
>> A=[1 2 3; 4 5 6; 3 -4 9]
A =
1      2      3
4      5      6
3     -4      9

>> B=[2 0 1; 3 2 -1; -2 0 2]
B =
2      0      1
3      2     -1
-2     0      2

>> C=A\B
C =
-0.9000    1.0000   -1.5500
0.8000        0     0.1000
0.4333   -0.3333     0.7833

>> C=A^-1*B
C =
-0.9000    1.0000   -1.5500
0.8000    0.0000     0.1000
0.4333   -0.3333     0.7833

>> C=inv(A)*B
C =
-0.9000    1.0000   -1.5500
0.8000    0.0000     0.1000
0.4333   -0.3333     0.7833
```

Uno de los usos típicos de la división por la izquierda, es la resolución de sistema de ecuaciones

lineales (ver tema 6). Por ejemplo, dado el sistema de ecuaciones,

$$\left. \begin{array}{l} 3x_1 + 2x_2 - 4x_3 = 3 \\ 2x_1 + x_2 + 3x_3 = -3 \\ x_1 + 3x_2 + 2x_3 = 7 \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} 3 & 2 & -4 \\ 2 & 1 & 3 \\ 1 & 3 & 2 \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_x = \underbrace{\begin{pmatrix} 3 \\ -3 \\ -7 \end{pmatrix}}_b$$

Podemos resolverlo en Matlab con una sencilla división por la izquierda,

```
>> A=[3 2 -4; 2 1 3; 1 3 2]
```

```
A =
```

```
3      2      -4
2      1       3
1      3       2
```

```
>> b=[3;-3;-7]
```

```
b =
```

```
3
-3
-7
```

```
>> x=A\b
```

```
x =
```

```
1
-2
-1
```

Aunque la potenciación en Matlab tiene más usos, solo consideraremos el caso de una matriz elevada a un número. El resultado será multiplicar la matriz por si misma tantas veces como indique el exponente, $A^3 = A \cdot A \cdot A$,

```
>> A=[3 0 -1; 2 1 0; 1 3 2]
```

```
A =
```

```
3      0      -1
2      1       0
1      3       2
```

```
>> A^3
```

```
ans =
```

```
13    -18    -18
24     -5    -12
54     18     -5
```

```
>> A*A*A
ans =
13   -18   -18
24    -5   -12
54    18    -5
```

La potenciación elemento a elemento eleva cada elemento de la matriz al valor que indique el exponente,

```
>> B = [1 2 3;3 4 5]
```

```
B =
1   2   3
3   4   5
>> B.^2
ans =
1   4   9
9  16  25
```

Trasponer una matriz es obtener una nueva matriz intercambiando las filas con las columnas de la matriz original, (ver capítulo 5.2). Así por ejemplo,

```
>> A=[3 0 -1 0; 2 1 0 0]
```

```
A =
3   0   -1   0
2   1   0   0
>> B=A'
B =
3   2
0   1
-1  0
0   0
```

Las matrices A y B son traspuestas entre sí.

2.3.2. Precedencia de los operadores aritméticos

Operadores!Precedencia Combinando operadores aritméticos, es posible elaborar expresiones complejas. Por ejemplo,

```
R=5*3-6/3+2^3+2-4
```

La pregunta que surge inmediatamente es en qué orden realiza Matlab las operaciones indicadas. Para evitar ambigüedades, Matlab —como todos los lenguajes de programación— establece un orden de precedencia, que permite saber exactamente en qué orden se realizan las operaciones. En Matlab el orden de precedencia es:

1. En primer lugar se calculan las potencias.
2. A continuación los productos y las divisiones, que tienen el mismo grado de precedencia.
3. Por último, se realizan las sumas y las restas.

Por tanto, en el ejemplo que acabamos de mostrar, Matlab calcularía primero,

$$2^3=8$$

a continuación el producto y la división

$$5*3=15$$

$$6/3=2$$

Por último sumaría todos los resultados intermedios, y guardaría el resultado en la variable R

$$15-2+8-4=17$$

$$R=17$$

Uso de paréntesis para alterar el orden de precedencia. Cuando necesitamos escribir una expresión complicada, en muchos casos es necesario alterar el orden de precedencia. Para hacerlo, se emplean paréntesis. Sus reglas de uso son básicamente dos:

- Las expresiones entre paréntesis tienen precedencia sobre cualquier otra operación.
- Cuando se emplean paréntesis anidados (unos dentro de otros) los resultados siempre se calculan del paréntesis más interno hacia fuera.

Por ejemplo,

$$>> y=2+4/2$$

$$y =$$

$$4$$

$$>> y=(2+4)/2$$

$$y =$$

$$3$$

En la primera operación, el orden de precedencia de los operadores hace que Matlab divida primero 4 entre 2 y a continuación le sume 2. En el segundo caso, el paréntesis tiene precedencia; Matlab suma primero 2 y 4 y a continuación divide el resultado entre 2.

El uso correcto de los paréntesis para alterar la precedencia de los operadores, permite expresar cualquier operación matemática que deseemos. Por ejemplo calcular la hipotenusa de un triángulo rectángulo a partir de valor de sus catetos,

$$h = (c_1^2 + c_2^2)^{\frac{1}{2}}$$

Que en Matlab podría expresarse como,

```
h=(c1^2+c2^2)^(1/2)
```

O la expresión general para obtener las raíces de una ecuación de segundo grado,

$$x = \frac{-b \pm (b^2 - 4 \cdot a \cdot c)^{\frac{1}{2}}}{2 \cdot a}$$

en este caso es preciso dividir el cálculo en dos expresiones, una para la raíz positiva,

```
x=(-b+(b^2-a*c)^(1/2))/(2*a)
```

y otra para la raíz negativa

```
x=(-b-(b^2-a*c)^(1/2))/(2*a)
```

Es necesario ser cuidadosos a la hora de construir expresiones que incluyen un cierto número de operaciones. Así, en el ejemplo que acabamos de ver, el paréntesis final `2*a` es necesario; si se omite, Matlab multiplicará por `a` el resultado de todo lo anterior, en lugar de dividirlo.

2.3.3. Operaciones Relacionales y lógicas.

Aunque son distintas, las operaciones relacionales y las lógicas estas estrechamente relacionadas entre sí. Al igual que en el caso de las operaciones aritméticas, en las operaciones relacionales y lógicas existen operandos –variables sobre las que se efectúa la operación– y operadores, que indican cuál es la operación que se efectúa sobre los operandos. La diferencia fundamental es que tanto en el caso de las operaciones relacionales como lógicas el resultado solo puede ser 1 (cierto) o 0 (falso).

Operadores relacionales. La tabla 2.3 muestra los operadores relacionales disponibles en el entorno de Matlab. Su resultado es siempre la verdad o falsedad de la relación indicada.

Los operadores relacionales pueden trabajar sobre matrices de igual tamaño, en ese caso la operación se realiza elemento a elemento y el resultado es una matriz de unos y ceros. Por ejemplo:

```
>> A=[3 0 -1; 2 1 0; 1 3 2]
```

```
A =
```

```
3     0     -1
2     1      0
1     3      2
```

```
>> B=[2 0 1; 3 2 -1; -2 0 2]
```

```
B =
```

```
2     0      1
3     2     -1
-2    0      2
```

```
>> C=A>B
```

```
C =
```

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

Cuadro 2.3: Operadores relacionales definidos en Matlab

operación	símbolo	ejemplo	notas
menor que	<	r=a<b	Compara matrices elemento a elemento o un escalar con todos los elementos de una matriz
mayor que	>	r=a>b	Compara matrices elemento a elemento o un escalar con todos los elementos de una matriz
mayor o igual que	>=	r=a>=b	Compara matrices elemento a elemento o un escalar con todos los elementos de una matriz
menor o igual que	<=	r=a<=b	Compara matrices elemento a elemento o un escalar con todos los elementos de una matriz
igual a	==	a==b	Compara matrices elemento a elemento o un escalar con todos los elementos de una matriz
Distinto de	a~=b	a~=b	Compara matrices elemento a elemento o un escalar con todos los elementos de una matriz
Especificadores			
todos	all	r=all(a)	Verdadero si todos los elementos de un vector son verdaderos. Para matrices el resultado se obtiene para cada columna
alguno	any	r=any(a)	Verdadero si algún(os) elemento(s) de un vector son verdadero(s). Para matrices el resultado se obtiene para cada columna
encontrar	find	r=find(a)	Devuelve como resultado los índices de los elementos verdaderos

La matriz C contiene el resultado lógico de comprobar uno a uno si los elementos de A son mayores que los elementos de B; como el elemento A(1,1) es mayor que B(1,1), la relación es cierta. Por tanto, el resultado C(1,1) es uno. La relación, –en este caso ser mayor que– se va comprobando elemento a elemento y su verdad o falsedad se consigna en el elemento correspondiente de la matriz resultado.

Por supuesto, si se comparan dos valores escalares el resultado es un también un escalar,

```
>> r=3<=7
r =
```

```
1
```

Los operadores relacionales admiten también que uno de sus operandos sea un escalar y el otro una matriz. En este caso, Matlab compara el escalar con todos los elementos de la matriz y guarda el resultado en una matriz del mismo tamaño,

```
>> A
A =
3     0    -1
2     1     0
1     3     2
>> menor_que_tres=A<3
```

```

menor_que_tres =
0      1      1
1      1      1
1      0      1

>> distinto_de_tres=A~=3

distinto_de_tres =
0      1      1
1      1      1
1      0      1

>> igual_a_tres=A==3

igual_a_tres =
1      0      0
0      0      0
0      1      0

```

Es importante señalar que el operador relacional que permite comparar si dos variables son iguales es == (doble igual), no confundirlo con el igual simple = empleado como sabemos como símbolo de asignación. En cuanto a la tilde,~, empleada en el operador ~=, se obtiene en Matlab pulsando la tecla 4 mientras se mantiene pulsada la tecla alt gr. la tilde en Matlab representa la negación lógica. Así por ejemplo si escribimos en Matlab,

```

>> ~1
ans =
0

>> ~0
ans =
1

```

Si negamos el uno (verdadero) nos da cero (falso) y viceversa.

Hablaremos por último de los especificadores, incluidos en la parte inferior de la tabla 2.3. No son operadores. Se trata de funciones definidas en Matlab.

La función any toma como variable de entrada una matriz. Como veremos en la sección siguiente, esto se indica colocando el nombre de la variable de entrada, a continuación del nombre de la función entre paréntesis. La salida de la función, entendiendo por tal el valor devuelto por la misma, se puede guardar en una variable mediante el símbolo de asignación,

```
salida=any(entrada)
```

Si introducimos como entrada un vector fila o un vector columna, la función any, devolverá un uno a la salida, si al menos uno de los elementos del vector de entrada es distinto de cero y devolverá un cero si todos los elementos del vector de entrada son ceros

```
>> s=[1 -2 0 2]
s =
1     -2      0      2

>> r=any(s)

r =
1

>> s=zeros(3,1)

s =
0
0
0

>> r=any(s)

r =
0
```

Si introducimos como variable de entrada una matriz, la función `any` buscará si hay algún valor distinto de cero por columnas de la matriz de entrada. La salida de la función `any` será entonces un vector fila con el mismo número de columnas que la matriz de entrada. Cada elemento del vector salida toma valor uno, si la columna correspondiente de la matriz de entrada tiene al menos un valor distinto de cero y toma valor cero si todos los elementos de dicha columna son cero. Por ejemplo, Si definimos en Matlab una matriz,

```
>> A=[3 0 -1 0; 2 1 0 0; 1 3 0 0]
A =
3     0     -1      0
2     1      0      0
1     3      0      0
```

y le aplicamos la función `any`,

```
>> r=any(A)
r =
1     1     1      0
```

El primer elemento del vector resultante es 1, puesto que todos los elementos de la primera columna de A son cero. El segundo también es uno, porque al menos dos de los elementos de la segunda columna de A son distintos de cero, lo mismo sucede con la tercera que tiene un elemento

distinto de cero. Solo el último elemento de la respuesta es cero ya que todos los elementos de la última columna de A son cero.

La función `all` funciona de modo análogo a `any`, pero en este caso, el vector resultante toma valor uno si todos los elementos de la columna correspondiente de la matriz de entrada son distintos de cero. Si aplicamos `all` a la misma matriz del ejemplo anterior,

```
>> r=all(A)
r =
1     0     0     0
```

Solo el primer elemento del vector salida `r` es ahora distinto de cero, ya que la matriz A tiene ceros en todas sus columnas menos en la primera.

Queda por señalar que ambas funciones pueden operar por filas en lugar de hacerlo por columnas. De hecho la función admite un segundo parámetro de entrada que se introduce detrás de la matriz de entrada y separado por una coma, si dicho parámetro vale 1 (o se omite, como hemos hecho en los ejemplos anteriores), la función operan por columnas. Si a dicho parámetro se le da valor 2, la función opera por filas,

```
>> r=any(A,2)
r =
1
1
1

>> r=all(A,2)
r =
0
0
0
```

En este caso las funciones nos devuelven vectores columnas que indican si en la fila correspondiente de la matriz de entrada hay algún elemento distinto de cero (caso del función `any`) o si todos los elementos son distintos de cero (caso de la función `all`).

La utilidad de estas dos funciones que acabamos de describir se ve más clara cuando las combinamos con el uso de los operadores relacionales. por ejemplo,

```
>> A=[3 0 -1 0; 2 1 0 0; 1 3 0 0]

A =
3     0     -1     0
2     1      0     0
1     3      0     0

>> C=A>2

C =
1     0     1     0
0     1     0     0
1     1     0     0
```

```

1      0      0      0
0      0      0      0
0      1      0      0

>> r1=any(C)

r1 =
1      1      0      0

>> r=any(r1)

r =
1

```

Mediante el uso del operador `>` y de la función `any`, hemos comprobado que en la matriz `A` hay al menos algún elemento distinto de cero. Por supuesto, esto podemos hacerlo en una sola sentencia, combinando operadores y funciones,

```
>> A=[3 0 -1 0; 2 1 0 0; 1 3 0 0]
```

```

A =
3      0      -1      0
2      1      0      0
1      3      0      0

```

```
>> r=any(any(A>2))
```

```

r =
1

```

Como un segundo ejemplo, vamos a comprobar si todos los elementos de la matriz `A` son menores que 4,

```
>> A=[3 0 -1 0; 2 1 0 0; 1 3 0 0]
```

```

A =
3      0      -1      0
2      1      0      0
1      3      0      0

```

```
>> r=all(all(A<4))
```

```

r =
1

```

Por último, insistir en que, si no se indica otra cosa, `any` and `all`, trabajan buscando los valores distintos de cero por columnas. Así por ejemplo la sentencia,

```
>> r=any(all(A>2))
r =
```

0

comprueba si en *alguna* de las columnas de A *todos* los elementos son menores que 2. y la sentencia,

```
>> r=all(any(A>-1))
r =
```

1

comprueba si en *todas* las columnas de A hay *algún* elemento mayor que -1.

La última de las funciones incluidas en la tabla 2.3 es la función `find`. Esta función admite como variable de entrada una matriz. Si se la llama con una sola variable de salida, devuelve un vector con los índices de los elementos de la matriz que son distintos de cero. Si se la llama con dos variables de salida en la primera devuelve un vector con el índice de las filas de los elementos de la matriz distintos de cero, y en la segunda variable devuelve un vector con los índices de las correspondientes columnas de los elementos de la matriz distintos de cero,

```
>> A
```

```
A =
```

1	0	1
0	1	1
1	1	0

```
>> indice=find(A)
indice =
```

1
3
5
6
7
8

Los elementos 1, 3 ,5, 6, etc, de la matriz A son distintos de cero (ver indexación con un único índice 2.1.3)

```
>> [fila,columna]=find(A)
```

```
fila =
```

1
3
2
3
1
2

```
columna =
```

```
1
1
2
2
3
3
```

```
>>
```

Los elementos (1,1), (3,1), (2,2), etc, de la matriz A son distintos de cero.

Podemos combinarlos con otros operadores relacionales para conocer qué elementos de una matriz cumplen una determinada condición. Por ejemplo:

```
>> A=[3 0 -1 0; 2 1 0 0; 1 3 0 0]
```

```
A =
```

```
3     0    -1     0
2     1     0     0
1     3     0     0
```

```
>> indice=find(A~=0)
```

```
indice =
```

```
1
2
3
5
6
7
```

Nos permite conocer los índices de los elementos de A que son distintos de cero. Como hemos dado una sola variable de salida, Matlab emplea un único índice para darnos la posición de los elementos distintos de cero (ver la sección indexación, pag. 37) para obtener los dos índices de cada elemento distinto de cero de la matriz A del ejemplo anterior basta llamar a la función con dos variables de salida,

```
>> [fila, columna] = find(A~=0)
```

```
fila =
```

```
1
2
3
2
3
1
```

```
columna =
1
1
1
2
2
3
```

Operadores Lógicos En Matlab se distinguen tres conjuntos de operadores lógicos según el tipo de variable sobre la que actúen. Aquí vamos a ver solo dos de ellos: los operadores lógicos elemento a elemento y los operadores lógicos para escalares.

La tabla 2.4 muestra los operadores lógicos elemento a elemento. Estos operadores lógicos esperan que sus operandos sean matrices de igual tamaño, aunque pueden actuar también sobre escalares.

El resultado, es una matriz del mismo tamaño que los operandos, compuesta por ceros y unos, que son el resultado de la operación lógica realizada entre los elementos de los operandos que ocupan la misma posición en sus respectivas matrices.

Cuadro 2.4: Operadores lógicos elemento a elemento

operación	símbolo	ejemplo	notas
and	&	r=a&b	Operación lógica <i>and</i> entre los elementos de a y b
or		r=a b	Operación lógica <i>or</i> entre los elementos de a y b
or exclusivo	xor()	r=xor(a,b)	Operación lógica <i>or exclusivo</i> entre los elementos de a y b
negación	~	r=~a	complemento de los elementos de a

En cuanto a su funcionamiento, son los operadores típicos del álgebra de Bool. Así el operador **&** sigue la tabla de verdad propia de la operación *and*, el resultado solo es verdadero (1) si sus operandos son verdaderos (1)⁴,

Tabla de verdad de la operación *and*

operando 1	operando 2	Resultado
1	1	1
1	0	0
0	1	0
0	0	0

Así por ejemplo,

```
>> A=[1 0 1; 0 1 1;1 1 0]
A =
```

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

⁴En realidad Matlab considerará verdadero cualquier operando distinto de 0

```
>> B=[1 1 1; 0 0 1; 1 0 1]
```

```
B =
```

1	1	1
0	0	1
1	0	1

```
>> R=A&B
```

```
R =
```

1	0	1
0	0	1
1	0	0

La matriz R contiene el resultado de realizar la operación **and** elemento a elemento entre las matrices A y B; solo aquellas posiciones que contienen a la vez un 1 en ambas matrices, obtienen un 1 como resulta en la matriz R.

La operación **or**, responde a la siguiente tabla de verdad,

Tabla de verdad de la operación **or**

operando 1	operando 2	Resultado
1	1	1
1	0	1
0	1	1
0	0	0

En este caso, el resultado es cierto si cualquiera de los dos operando es cierto. Si lo aplicamos a las matrices del ejemplo anterior,

```
>> r=A|B
```

```
r =
```

1	1	1
0	1	1
1	1	1

Solo es cero el elemento $r(2,1)$ de la matriz resultado, ya que solo los elementos $A(2,1)$ y $B(2,1)$ son a la vez cero.

La tabla de verdad de la operación **or exclusivo** es,

Tabla de verdad de la operación **xor**

operando 1	operando 2	Resultado
1	1	0
1	0	1
0	1	1
0	0	0

Es decir la salida solo es verdadera cuando una de las entradas es verdadera y la otra no. Esta operación solo existe en Matlab con formato de función. Usando de nuevo el ejemplo anterior,

```
>> r=xor(A,B)

r =

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

se puede comprobar que solo son uno los elementos de **r** para los cuales los correspondientes elementos de **A** y **B** no son uno o cero a la vez.

El operador negación actúa sobre un solo operando, negándolo es decir, transformando sus elementos con valor uno en ceros y sus elementos con valor cero en unos.

```
>> A
A =

1      0      1
0      1      1
1      1      0

>> r=^A
r =

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

Para entradas escalares se definen dos operadores lógicos, que se corresponden con los operadores *and* y *or* de la lógica de Bool. Por tanto siguen las tablas de verdad de dichas funciones que acabamos de ver. Su sintaxis es la misma que la de los operadores lógicos elemento a elemento, simplemente se escribe dos veces el símbolo del operador para indicar que se trata de una operación entre escalares. Así, por ejemplo para la operación **and**,

```
>> a=1
a =

1

>> b=0
b =

0

>> r=a&&b
r =
```

y para la operación **or**,

```
>> r=a||b
r =
1
```

Los operadores lógicos elemento a elemento también funcionan correctamente cuando actúan sobre escalares, pero son, en general menos eficientes que los operadores específicos que acabamos de ver.

Por último, indicar que los operadores lógicos pueden combinarse entre sí con operadores relacionales y con operadores aritméticos. El orden de precedencia es el siguiente:

1. Paréntesis ()
2. Operadores aritméticos en su orden de precedencia
3. Operadores relacionales, todos tienen el mismo orden de precedencia por lo que se evalúan de izquierda a derecha
4. **and** elemento a elemento
5. **or** elemento a elemento
6. **and** escalares
7. **or** escalares

Matlab aconseja el uso de paréntesis cuando se encadenan varias operaciones lógicas para asegurar su uso correcto y facilitar la lectura de las sentencias.

Por ejemplo,

```
>> A=[3 0 0; 2 1 0; 1 3 0]
A =
```

```
3     0     0
2     1     0
1     3     0
```

```
>> B=[2 0 1; 3 2 -1; -2 0 2]
B =
```

```
2     0     1
3     2    -1
-2     0     2
```

```
>> r=A>1&B>A
r =
```

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

```
>> r=(A>1)&(B>A)
r =
```

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

Ambas sentencias realizan la misma operación. Comprueban qué elementos de A son mayores que 1 y a la vez cumplen ser menores que el correspondiente elemento de B. Sin embargo, los paréntesis de la segunda sentencia facilitan entender el orden en que se realizan las operaciones.

2.4. Scripts y Funciones

Hasta ahora, hemos manejado siempre Matlab desde la línea de comandos. Es decir, hemos introducido las instrucciones en Matlab en la ventana de comandos. Este modo de emplear Matlab es poco eficiente, ya que exige volver a introducir todos los comandos de nuevo cada vez que queremos repetir un cálculo.

Matlab puede emplear ficheros de texto en los que introducimos un conjunto de comandos, guardarlos, y volver a emplearlos siempre que queramos. Esta es la forma habitual de trabajar no solo de Matlab, sino de otros muchos entornos de programación. Un fichero que contiene código de Matlab recibe el nombre genérico de *programa*. Un programa de Matlab no es más que un fichero de texto que contiene líneas formadas por comando válidos de Matlab. Lo habitual es que cada línea contenga un comando. El fichero se guarda con un nombre y la extensión .m. Por ejemplo: `miprograma.m`. El nombre del fichero, puede contener números y letras, pero el primer carácter debe ser siempre una letra. Los programas en Matlab pueden tomar dos formas básicas, *scripts* y *funciones*.

2.4.1. El editor de textos de Matlab.

Podemos emplear un editor de textos cualquiera, que genere texto en ASCII, como por ejemplo el 'block de notas', para escribir nuestros programas. Sin embargo, si trabajamos en el entorno de Matlab, lo ideal es emplear su propio editor de textos. Hay varias formas de abrir el editor de textos de Matlab; la más sencilla es pulsar el ícono de nuevo documento (*New Script*) o bien desplegando el menú *New* y seleccionando *script*, la posición de ambos se indica en la figura 2.4, dentro de la pestaña *home* del IDE de matlab.

La otra opción es emplear el comando `edit`. Este comando puede emplearse de dos maneras. Si se escribe en la ventana de comandos,

```
>>edit
```

Matlab abrirá el editor de textos y creará un documento nuevo sin nombre (*untitled*).

Si añadimos a continuación del comando `edit` el nombre de un fichero,

```
>>edit ejemplo1.m
```

Buscará un fichero con dicho nombre en los directorios de Matlab y en el directorio de trabajo. Si lo encuentra, abrirá el archivo encontrado. Si no lo encuentra, creará uno nuevo con dicho nombre. La figura 2.5 muestra el editor de textos de Matlab —Integrado en la parte central superior del IDE de Matlab— con el contenido del fichero `ejemplo1.m`. Además, de entre las pestañas situadas en la parte superior del IDE, se ha seleccionado la marcada como *EDITOR*. Es posible observar las barras de menú y de herramientas de las que dispone el editor de texto, para facilitar el trabajo de programación. Estas, entre otras facilidades —resaltar texto de palabras clave, cotar número de líneas, sangrar estructuras de programación, etc.— hacen que resulte especialmente atractivo emplear el editor de textos de Matlab en lugar de emplear otro editor de textos genérico.

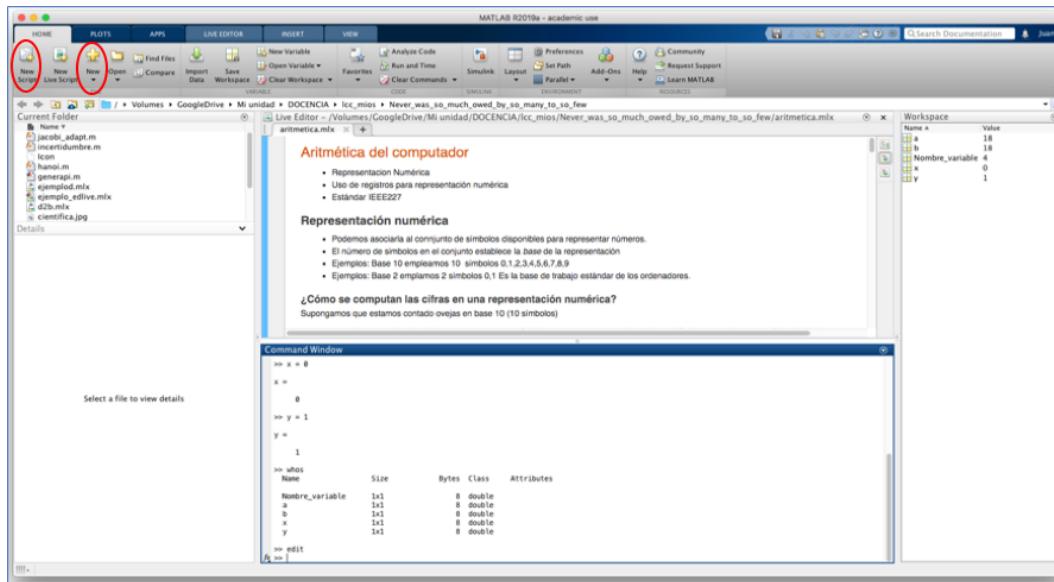


Figura 2.4: posición del botón *New Script* y del menú *New* en el IDE de Matlab (Señalados en rojo)

2.4.2. Scripts

Un script en Matlab es un simple fichero de texto que contiene un conjunto de sentencias de Matlab válidas. La manera de ejecutarlo, consiste en escribir el nombre del fichero en la línea de comandos de Matlab. Matlab va leyendo el contenido del fichero línea a línea, y va ejecutando los comandos que contiene cada línea exactamente igual que si se hubieran escrito directamente en la línea de comandos de Matlab. Veamos un ejemplo sencillo, que corresponde con el código contenido en la figura 2.5:

```
% Este programa toma dos matrices llamadas A y B del espacio de trabajo de
% Matlab, calcula su suma y la guarda en una variable llamada suma, calcula el
% producto y lo guarda en una variable llamada producto y, por última genera un
% mensaje en la ventana de comandos para indicar que ha terminado de ejecutarse.
suma=A+B
producto=A*B
disp('ejecución terminada')
```

Si escribimos el texto anterior en un archivo y lo guardamos con el nombre `ejemplo1.m`, podemos ejecutar su contenido en Matlab sin más que escribir en la línea de comandos:

```
>> ejemplo1
```

Las cuatro primeras líneas de código que empiezan con el símbolo `%` no se ejecutan. Cuando Matlab encuentra una línea que empieza con dicho símbolo interpreta que se trata de un comentario escrito por el programador, para explicar qué hace el programa o aclarar algún aspecto de su funcionamiento. Cuando el editor de Matlab detecta el símbolo `%` en una línea de código, resalta en color verde todo el texto de la línea a partir de dicho símbolo. De este modo, es inmediato ver que se trata de un comentario y que Matlab no tratará de ejecutarlo.

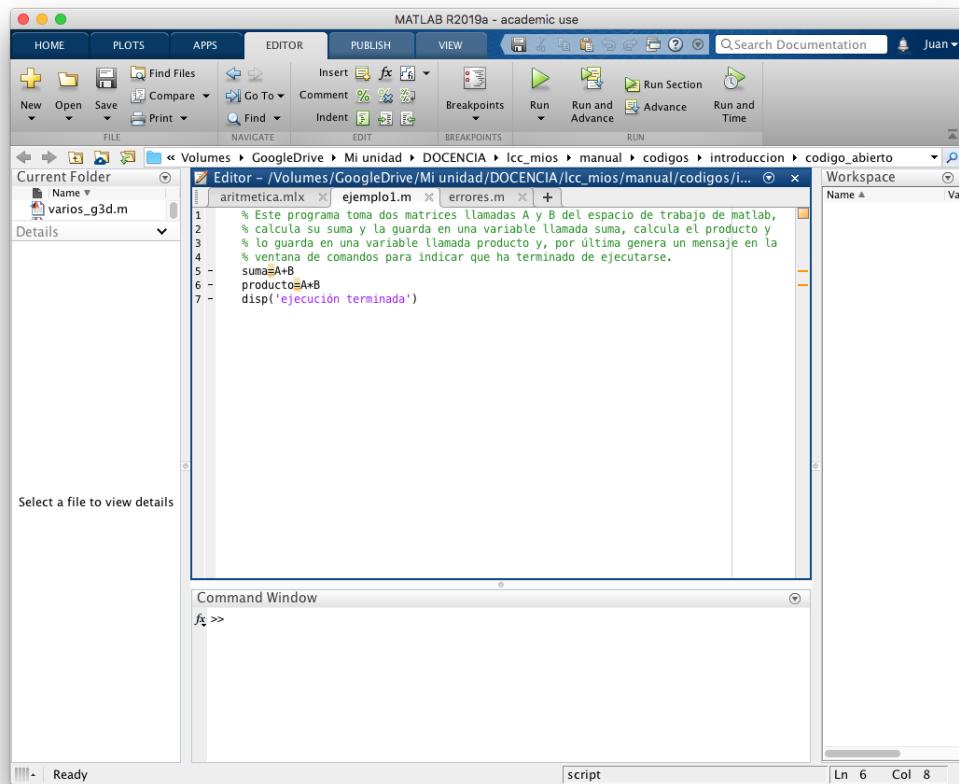


Figura 2.5: Vista del editor de textos de Matlab mostrando el contenido del fichero ejemplo1.m

Un aspecto importante de la programación en Matlab, y en cualquier otro lenguaje de programación, lo constituye el comentario del código de programa; facilita su uso por otros usuarios y permite al programador recordar qué fue lo que hizo cuando lo programó. La experiencia demuestra que, en poco tiempo, los programas no comentados se vuelven incompresibles incluso para quien los escribió.

La quinta línea del programa busca en el *workspace* las variables *A* y *B*. Si las variables no existen, es decir, si no han sido creadas previamente, el programa da un error, exactamente igual que si hubiéramos escrito directamente en la ventana de comandos de Matlab la sentencia *suma=A+B* sin haber definido antes *A* y *B*. Si las variables existen calcula la suma y guarda el resultado en el *workspace* en la variable *suma*. La siguiente línea de código realiza el cálculo del producto de las dos variables y guarda el resultado en la variable *producto*. La última línea del programa mostrará en la ventana de comandos la frase,

ejecución terminada

Para ello emplea la función de Matlab *disp* que escribe en la ventana de comandos cadenas de caracteres.

Un aspecto muy importante de los *scripts* es que hacen uso del *workspace* de Matlab tanto para buscar las variables que emplean como para guardar las variables resultantes de sus cálculo. Su

ejecución es idéntica a la que se realizaría si copiáramos línea a línea en la ventana de comandos y las fuéramos ejecutando una detrás de otra.

2.4.3. Funciones

Las funciones juegan un papel fundamental en cualquier lenguaje de programación. En el caso de Matlab, se escriben como ficheros de texto, de modo análogo a los *scripts*, lo que determina que Matlab los interprete como funciones es su primera línea de código. Nos referiremos a esta primera línea con el nombre de *cabecera de la función*. La cabecera de una función debe empezar siempre por la palabra clave **function**. Debe contener además, como mínimo, el nombre de la función. Por ejemplo,

```
function raices
```

Un detalle importante en Matlab es que el fichero de texto que contiene la función debe llamarse igual que ésta, para que Matlab pueda identificar la función correctamente. Es decir, en el caso del ejemplo anterior, el fichero que contenga la función debe llamarse *raíces.m*.

A parte del nombre de la función la cabecera puede incluir también nombres de las variables de entrada. Estos se escriben a continuación del nombre de la función, separadas por comas y encerradas entre paréntesis,

```
function raices(a,b,c)
```

En este ejemplo **a**, **b** y **c** son variables de entrada de la función **raices**. La razón por la que hay que incluir estas variables de entrada es que, a diferencia de los *scripts*, las funciones no pueden acceder directamente a los valores de las variables contenidos en el *workspace* de Matlab. Cuando se ejecuta una función es preciso dar valores a las variables que necesite utilizar, y que no se definen expresamente en el código de la función. Aclararemos esto más tarde con un ejemplo.

Por último la cabecera de una función puede incluir también una o varias variables de salida. Estas se escriben delante del nombre de la función separadas por comas y encerradas entre corchetes. Entre la(s) variable(s) de salida y el nombre de la función se incluye el símbolo de asignación **=**, para indicar que los resultados obtenidos por la función se han asignado (guardado en) a dichas variables.

```
function [x1,x2]=raices(a,b,c)
```

En este caso, las variables de salida serían **x1** y **x2**.

A continuación, vamos a completar el ejemplo para el que hemos construido la cabecera en los párrafos anteriores. Se trata de una función que obtiene las raíces de una ecuación de segundo grado, $ax^2 + bx + c = 0$ conocidos sus coeficientes **a**, **b**, **c**,

```
function [x1,x2]=raices(a,b,c)
%Esta función calcula las raíces de una ecuación de segundo grado ax^2+b^x+c=0
%las variables de entrada son los coeficientes a, b, c de la ecuación.
%las variables de salida x1 , x2 son las dos raíces de la ecuación

%calculo de la primera raiz
x1=(-b+(b^2-4*a*c)^(1/2))/(2*a)

%calculo de la segunda raiz
x2=(-b-(b^2-4*a*c)^(1/2))/(2*a)
```

Deberemos guardar estas líneas de código en un archivo con el nombre `raices.m`. Podemos ahora emplear la función (`raices`) para calcular las raíces de una ecuación de segundo grado. Supongamos que queremos obtener las raíces de la ecuación,

$$x^2 + x - 6$$

si escribimos en la línea de comandos,

```
>> [raiz1, raiz2]=raices(1,1,-6)
```

Matlab mostrará en la pantalla,

```
x1 =
```

```
2
```

```
x2 =
```

```
-3
```

```
raiz1 =
```

```
2
```

```
raiz2 =
```

```
-3
```

Analicemos con un poco de detalle lo que hemos hecho. Al escribir: `[raiz1,raiz2] = raices(1,1,-6)`, hemos *llamado* a la función `raices`, indicando que las variables de entrada deben tomar los valores `a=1,b=1, c=-6`, es decir, los valores de los coeficientes de la ecuación de segundo grado cuya solución queremos obtener. Además hemos pedido a Matlab que guarde los resultados en el *workspace* en las variable `raiz1` y `raiz2`. Cuando, tras llamar a la función, pulsamos el retorno de carro, Matlab empieza a ejecutar el código de la misma. Lo primero que hace es crear las variables `a`, `b` y `c` y asignarle los valores 1,1 y -6. Matlab crea estas variables pero no las guarda en el workspace sino en un espacio de memoria al que solo tiene acceso la función `raices`, desde la que se han creado. Esto constituye una característica muy importante de las funciones en Matlab. Cada vez que se llama a una función, se crea un espacio de memoria en la que se guardan las variable definidas en la función y a la que solo ésta tiene acceso. Además, una función no puede acceder ni modificar directamente ninguna variable que esté en el *workspace*.

Una vez que la función ha asignado valores a las variable de entrada, comienza a realizar los cálculos pedidos, en primer lugar calcula la raíz correspondiente al discriminante positivo,

$$+\sqrt{b^2 - 4ac}$$

y crea la variable `x1` para guardar el resultado. La variable `x1` solo existe en la memoria de la función `raices` y por tanto no existe ni es accesible desde el *workspace*. Como no hemos terminado la línea de programa en la que se calcula `x1` con un punto y coma, Matlab muestra en la ventana de comandos el resultado del cálculo realizado.

La raíz correspondiente al discriminante negativo se calcula de modo análogo. Una vez terminada la ejecución de las líneas del programa, Matlab vuelve a examinar la cabecera del programa y observa que debe dar como resultado, los valores contenidos en las variables `x1` y `x2`. Para ello, crea la variable `raiz1` en el *workspace* y copia en ella el contenido de la variable `x1`. Análogamente crea la variable `raiz2` y copia en ella el contenido de la variable `x2`. Con esto ha terminado la ejecución del programa. Matlab destruye las variables `x1` y `x2` que solo han existido durante la ejecución del programa, y nos muestra de nuevo el *prompt* en la ventana de comandos para indicarnos que está listo para ejecutar nuevas órdenes. Si analizamos el contenido del *workspace*, empleando el comando `who` de Matlab,

```
>> who
```

Your variables are:

```
raiz1  raiz2
```

observaremos que allí están las variables `raiz1` y `raiz2`, que contienen las raíces de la ecuación de segundo grado que queríamos resolver.

En el ejemplo anterior hemos asignado directamente valores a las variables de entrada de la función `raices`. En Matlab, una función puede también asignar valores a sus variables de entrada copiándolos de los de otras variables existentes en el *workspace*, supongamos que creamos en el *workspace* de Matlab, tres variables con los valores de los coeficientes de la ecuación de segundo grado del ejemplo anterior,

```
>>coef1=1, coef2=1 coef3=-6
```

Podríamos entonces llamar a nuestra función `raíces`, sustituyendo los valores de las variables de entrada por los nombres de éstas variables,

```
>> [raiz1, raiz2]= raices(coef1,coef2,coef3)
```

Matlab, copiará ahora los valores contenidos en `coef1`, `coef2` y `coef3` en las variables de entrada de la función `a`, `b`, `c`. Ni que decir tiene, que el resultado final de la ejecución será el mismo.

Ámbito de una variable La importancia del concepto de función está precisamente en el tratamiento que hace de las variables. Para entenderlo mejor, introduciremos el concepto de *ámbito* de una variable.

Como hemos visto, la forma usual de crear una variable en Matlab es mediante el símbolo de asignación. Cuando asignamos un valor o el resultado de una operación a una variable en la ventana de comandos de Matlab,

```
>> a=18
```

```
a =
```

Matlab reserva un espacio de memoria del computador para guardar la variable con el valor asignado. Esta variable forma parte del *workspace* de Matlab, que constituye su *ámbito* propio. La variable creada es solo visible para aquellos comandos y sentencias que,

1. Se ejecutan desde la ventana de comandos de Matlab.
2. Se ejecutan desde un script.

Cuando ejecutamos una función desde la ventana de comandos de Matlab, la función crea su propio espacio de memoria. Por así decir, es como un *workspace* particular de la función. Cualquier variable que cree la función se guardará en el espacio de memoria de la función, que constituirá su ámbito propio. La variable creada dentro de la función solo es visible para aquellos comandos y sentencias que se ejecutan dentro de la función.

Una vez que termina la ejecución de una función, el espacio de memoria que se creó al ejecutarse se destruye y con él cualquier variable que el programa haya creado durante su ejecución.

La única manera de pasar información contenida en una variable del *workspace* de Matlab a una función es copiándola a una variable de entrada de la función.

La única manera de pasar información contenida en una variable del espacio de memoria de una función, al *workspace* de Matlab, es copiándola a través de una variable de salida de la función.

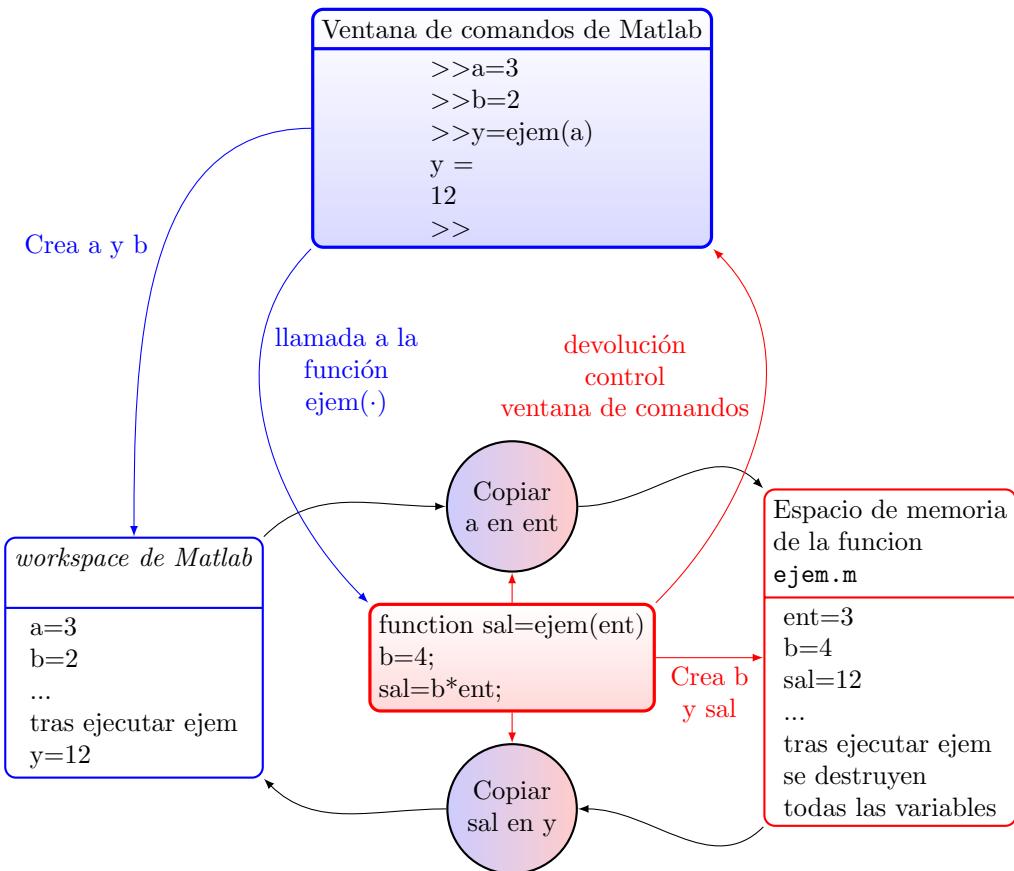


Figura 2.6: Ejemplo de uso de memoria y ámbito de variables durante la ejecución de una función

La figura 2.6 muestra esquemáticamente como se gestionan las variables en Matlab. En el cuadro superior se muestra la ejecución de varias sentencias en la ventana de trabajo de Matlab. Las dos primeras crean directamente dos variables **a** y **b** que se almacenan en el *workspace de Matlab*, representado por el cuadro azul de la derecha. A continuación se llama a la función ejem

(cuadro rojo central), asignándole como variable de entrada la variable **a** y como variable de salida la variable **y**. La ventana de comandos *cede* el control a la función que empieza a ejecutarse:

1. La función crea su propio espacio de memoria (cuadro rojo de la izquierda)
2. Copia el valor contenido en la variable **a** en la variable **ent**. Esta variable se almacena en el espacio de memoria de la función.
3. Crea la variable **b** asignándole el valor 4 y la guarda en el espacio de memoria de la función.
A partir de este momento *y* hasta el final de la ejecución de la función hay dos variables con el mismo nombre: **b=2** en el *workspace* de Matlab; **b=4** en el espacio de memoria de la función. Las dos variables coexisten pero no se pueden confundir porque pertenecen a ámbitos distintos.
4. Crea la variable **sal** asignándole el producto de **b=4** por **ent=3**. Almacena la variable **sal** en el espacio de memoria de la función.
5. La función a llegado al final de su código. vuelve a leer la cabecera y crea la variable **y** en el *workspace* de Matlab, copiando el contenido de la variable **sal**.
6. termina la ejecución destruyendo el espacio de memoria de la función y *devolviendo* el control a la ventana de comandos.

Llamar a una función desde otra función. En Matlab, como en cualquier lenguaje de alto nivel, es posible llamar una función desde dentro de otra. Incluso es posible que una función se llame a sí misma, aunque este caso lo veremos más adelante cuando hablemos de control de flujo.

Veamos un ejemplo sencillo de llamada de una función desde otra,

```
function salida=ejemplo2(entrada)
%esta función toma el valor de entrada lo eleva al cuadrado y pasa el
%resultado a una segunda función que calcula la raíz cuadrada...
%entrada y salida deberían ser iguales al final

x=entrada^2;

%%%%llamada a la segunda función%%%%%
salida=raiz(x);
```

La función **ejemplo2** llama a una segunda función **raiz** cuyo código,

```
function out=raiz(in)
out=in^(1/2);
```

Debe guardarse en uno de los tres lugares siguientes:

1. En el mismo fichero, **ejemplo2.m** en que se encuentra escrito el código de la función **ejemplo2**, justo debajo de dicha función.
2. En un fichero propio, **raiz.m** guardado en el directorio en que Matlab está trabajando.
3. En un fichero propio, **raiz.m** guardado en cualquier directorio de los incluidos en el *path* de Matlab.

Además al ejecutar la función `ejemplo2`, se buscará el código de la función raíz, precisamente en el orden que acabamos de indicar. Si añadimos directamente el código de la función `raiz` al fichero `ejemplo2.m`, el código quedaría,

```
function salida=ejemplo2(entrada)
%esta funcion toma el valor de entrada lo eleva al cuadrado y pasa el
%resultado aun segunda función que calcula la raiz cuadrada...
%entrada y salida deberían ser iguales al final

x=entrada^2;

%%%%llamada a la segunda función%%%%%
salida=raiz(x);

%%%%%codigo de la segunda funcion%%%%%
function out=raiz(in)
disp('version incluida en el archivo ejemplo2.m')
out=in^(1/2);
```

La ventaja de escribir el código de la segunda función en el mismo fichero de la primera es que el acceso es más rápido. Sin embargo, solo la función `ejemplo2` podrá llamarla. Es decir la función `raiz` no puede emplearse desde la ventana de comandos de Matlab ni desde ninguna otra función.

En general, si escribimos en un archivo .m de Matlab varias funciones,

```
function a=uno(b)
% aqui viene el codigo de la funcion uno
...

function c=dos(d)
% aqui viene el codigo de la funcion dos
...

function e=tres(f)
% aqui viene el codigo de la funcion dos
...
.
.
.
```

Todas las funciones incluidas en el fichero pueden llamarse entre unas a otras, pero solo la primera de ellas puede ser ejecutada desde la ventana de comandos de Matlab. Además el nombre del fichero debe coincidir con el nombre de la primera función contenida en él. En el ejemplo que acabamos de esbozar, el fichero debería llamarse `uno.m`.

Evidentemente si cada función está guardada en un fichero .m distinto, todas las funciones pueden en principio ser ejecutadas desde otra función o desde la ventana de comandos de Matlab.

Cada vez que se ejecuta una función, ésta crea su propio espacio de memoria. Las variables incluidas en la cabecera de la función como variables de entrada y salida se copian, tal y como hemos visto para el caso de una función simple, entre el espacio de memoria propio de la función y el espacio de memoria de la función que la ha llamado.

Cuadro 2.5: Algunas funciones matemáticas en Matlab de uso frecuente

tipo	nombre	variables	función matemática
Trigonométrica	cos	y=cos(x)	coseno de un ángulo en radianes
Trigonométrica	sin	y=sin(x)	seno de un ángulo en radianes
Trigonométricas	tan	y=tan(x)	tangente de un ángulo en radianes
Trigonométricas	csc	y=csc(x)	cosecante de un ángulo en radianes
Trigonométricas	sec	y=sec(x)	secante de un ángulo en radianes
Trigonométricas	cot	y=cot(x)	cotangente de un ángulo en radianes
Trigonométricas	...	y=a...(x)	inversa de una función trigonométrica en radianes
	asin	y=asin(x)	ejemplo, arcoseno en radianes
Exponencial	exp	y=exp(x)	e^x
Exponencial	log	y=log(x)	logaritmo natural
Exponencial	log10	log10(x)	logaritmo en base 10
Exponecial	sqrt	y=sqrt(x)	\sqrt{x}
Redondeo	ceil	y=ceil(x)	redondeo hacia $+\infty$
Redondeo	floor	y=floor(x)	redondeo hacia $-\infty$
Redondeo	round	y=round(x)	redondeo al entero más próximo
Redondeo	fix	y=fix(x)	redondeo hacia 0
Redondeo	rem	r=rem(x,y)	resto de la división entera de y entre x
Módulos	norm	y=norm(x)	módulo de un vector x
Módulos	abs	y=abs(x)	valor absoluto de x,(módulo de x si x es complejo)
Módulos	sign	y=sign(x)	función signo; 1 si $x > 0$, -1 si $x < 0$, 0 si $x = 0$

2.4.4. Funciones incluidas en Matlab.

Matlab incluye cientos de funciones. Estas funciones, están escritas con la misma filosofía que acabamos de describir aquí, es decir, admiten una o varias variables de entrada y devuelven sus resultados en una o varias funciones de salida. En algunos casos, se trata de ficheros de texto guardados con la extensión .m iguales a los que nosotros podemos crear ⁵. La manera de emplearlas desde la ventana de comandos de Matlab es idéntica a la descrita para las funciones creadas por el usuario.

En la tabla 2.5, se incluyen algunos ejemplos de las funciones matemáticas más corrientes. Son solo una pequeña muestra de las funciones disponibles. Para obtener una visión más completa de las funciones disponibles se aconseja emplear la ayuda de Matlab.

2.4.5. Depuración.

Siempre que escribimos un programa, tanto si se trata de un *script* como si es una función, es preciso comprobar su funcionamiento y, en muchos casos corregir los errores cometido. El proceso de corrección de código desde su versión original hasta la versión definitiva se conoce con el nombre de depuración de código. Podemos distinguir dos tipos fundamentales de errores:

1. Errores de sintaxis. Normalmente son errores de escritura. Hemos escrito mal el nombre de una función o un comando o bien no hemos escrito correctamente el código siguiendo las reglas del lenguaje. Matlab advierte directamente de estos errores, cuando se trata de ejecutar el

⁵En muchos casos las funciones incluidas en Matlab no están escritas en ficheros de texto accesibles para el usuario. Por razones de eficiencia, se trata de versiones de las funciones escritas por lo general en lenguaje C y compiladas.

código, escribiendo en la ventana de comandos un mensaje de error. Como ejemplo veamos los errores del siguiente script,

```
%script con errores,
y=[1 2 3; 4 5 6; 2 3] %a esta matriz le falta un elemento en la última fila

x=[1 2 3; 4 5 6]
z=y*x %las matrices no pueden multiplicarse entre si por que no coinciden
       %numero de columnas de la primera con numero de filas de la segunda
```

Si observamos el editor de textos, figura 2.7, puede observarse algunas de los caracteres del texto subrayados en rojo. Esto puede indicar la existencia de errores en esa línea de código, como en el caso del carácter que se ha rodeado en la figura de un círculo rojo.

En otros casos, –circulo azul de la figura– se trata de advertencias, el programa funciona pero puede hacerlo de forma más eficiente; en el caso de la figura simplemente nos sugiere que añadamos un punto y coma al final de las sentencias, para que Matlab no escriba el resultado de cada cálculo en la ventana de comandos. No se trata por tanto de errores sino de advertir al programador que con puntos y comas su programa se ejecutará más rápido.

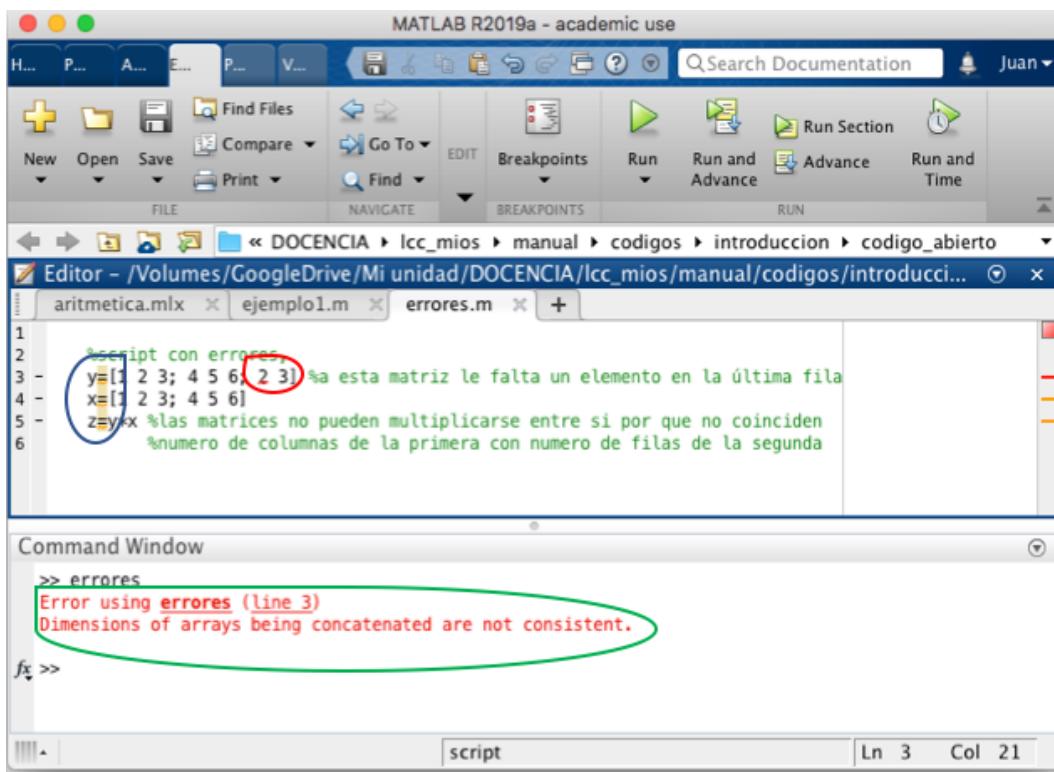


Figura 2.7: Vista de el editor de texto de Matlab. Circulo rojo error en el código. Rodeado en azul advertencias de posible mejoras. Rodeado en verde mensaje de error en tiempo de ejecución

Si pasamos el ratón por encima de los caractéres en rojo, Matlab nos ofrece información adicional sobre el error detectado o la advertencia de mejora. La figura ?? muestra la información asociada al error rodeado en rojo en la figura anterior (reffffig:ederror).

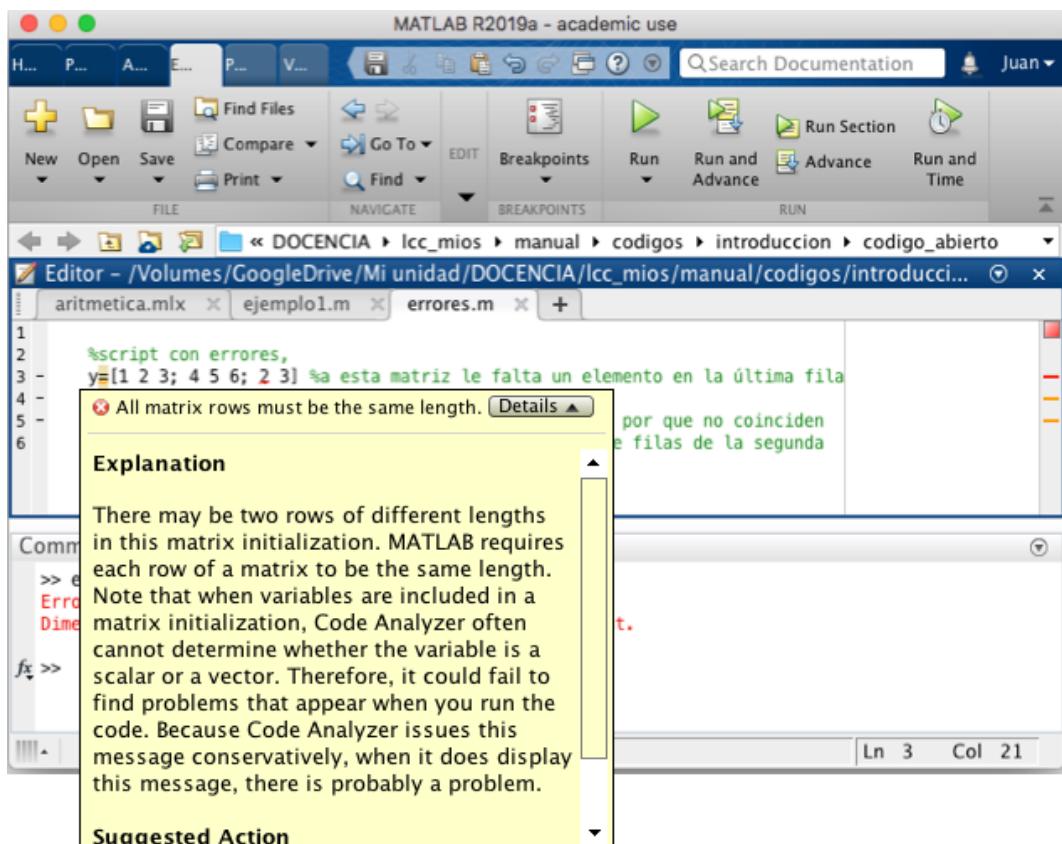


Figura 2.8: Vista de el editor de texto de Matlab. Circulo rojo error en el código. Rodeado en azul advertencias de posible mejoras. Rodeado en verde mensaje de error en tiempo de ejecución

Si ejecutamos el script, que hemos guardado con el nombre de `errores.m`,

```

>> errores
Error using errores (line 3)
Dimensions of arrays being concatenated are not consistent.

```

Matlab ha detectado el error en la construcción de la matriz `y`, nos indica mediante un mensaje el tipo de error cometido y la línea de código donde se ha producido y detiene la ejecución del *script*. Se trata del mensaje rodeado en verde en la figura 2.7.

Si corregimos el código, añadiendo a la matriz `y` el elemento que le falta,

```

%script con errores,
y=[1 2 3; 4 5 6; 2 3 8] %Ultima fila completada
x=[1 2 3; 4 5 6]
z=y*x %las matrices no pueden multiplicarse entre si por que no coinciden
%numero de columnas de la primera con numero de filas de la segunda

```

y volvemos a ejecutar el script.

```
>> errores

y =
1 2 3
4 5 6
2 3 8

x =
1 2 3
4 5 6

Error using *
Incorrect dimensions for matrix multiplication. Check that the number of
columns in the first matrix matches the number of rows in the second matrix.
To perform elementwise multiplication, use '.*'.

Error in errores (line 5)
z=y*x %las matrices no pueden multiplicarse entre si por que no coinciden
```

Matlab nos detecta el siguiente error cometido así como la línea en la que se comete. Es interesante notar que, aunque se trata de un error de sistaxis, el editor de textos no puede detectarlo y, por tanto, no nos muestra ningún carácter subrayado en rojo, asociado a él.

Si intercambiamos las posiciones entre las variables `x` e `y` en el producto, suponiendo que ésta es la causa del error cometido,

```
y=[1 2 3; 4 5 6; 2 3 8]

x=[1 2 3; 4 5 6]
z=x*y
```

El código se ejecuta con normalidad,

```
>> errores

y =
1 2 3
4 5 6
2 3 8

x =
1 2 3
4 5 6
```

```

z =
15    21    39
36    51    90

```

2. Errores de codificación. Este segundo tipo de errores son mucho más difíciles de detectar. El código se ejecuta sin problemas, pero los resultados no son los esperados. Ante esta situación, no queda más remedio que ir revisando el código, paso a paso para detectar donde está el error.

El siguiente código del script `trect.m` muestra un error de este tipo,

```
%este script toma los valores de los catetos de un triángulo rectangulo del
%workspace de matlab (variables a y b). calcula su hipotenusa, y a partir
%de estos datos, el seno el coseno y la tangente del angulo formado por la
%hipotenusa y el cateto mayor que será siempre a
```

```
%calculo hipotenusa,
h=sqrt(a^2+b^2)

%calculo del seno
s=a/h

%calculo del coseno
a=b/h %error estamos sobreescribiendo el valor del coseno en la variable
% que guardaba el valor del cateto

%calculo de la tangente
t=b/a
```

El programa funciona perfectamente, por ejemplo si hacemos `a=4` y `b=3`,

```
>> trect
```

```
h =
```

```
5
```

```
s =
```

```
0.8000
```

```
a =
```

```
0.6000

t =
5
```

Como hemos sobrescrito en `a` el valor del coseno, cuando tratamos de utilizar dicha variable como si fuera el cateto mayor para obtener la tangente, el resultado que obtenemos es erróneo.

El editor de texto de Matlab nos permite ejecutar un programa paso a paso, ver los valores que van tomando las variables en Matlab, etc, mediante el depurador que lleva incorporado. Para ello, se definen en el editor de Matlab *breakpoints*, esto es: líneas en las cuales Matlab detendrá la ejecución de un programa, entrará en modo de depuración y esperará instrucciones del usuario. La figura 2.9 muestra el código del ejemplo que acabamos de ver, en el que se ha definido un *breakpoint*, pulsando con el ratón sobre el guión que precede a la línea del programa en la que se desea parar la ejecución. Matlab indica que el *breakpoint* está activo, cambiando el guión por un círculo rojo. Alternativamente, también es posible establecer o remover *breakpoints* empleando el botón de la pestaña *EDITOR*, rodeado de rojo en la figura.

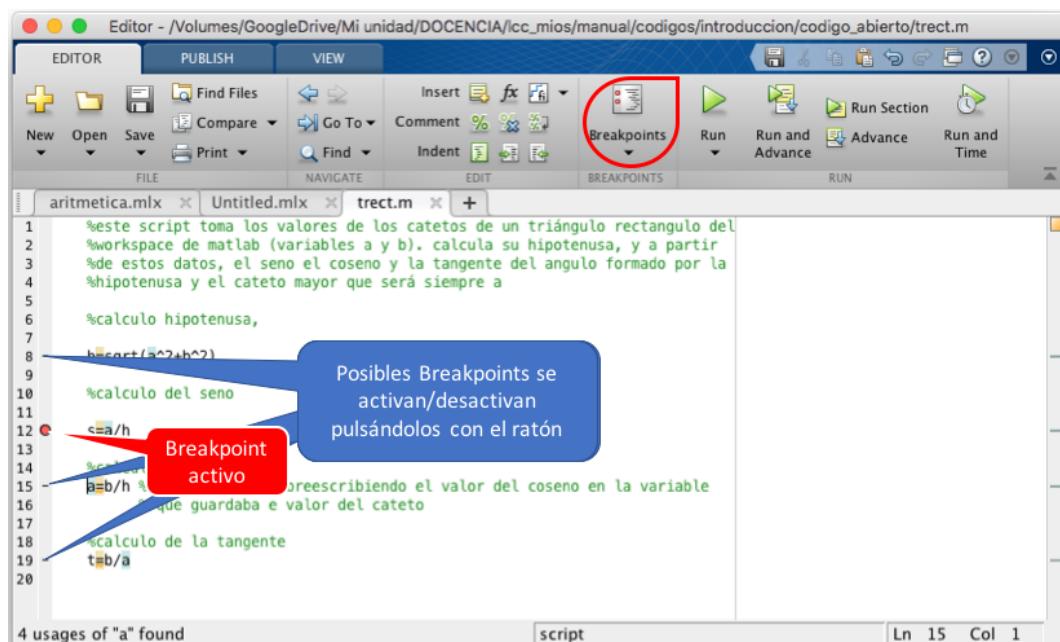


Figura 2.9: Breakpoint activo

Si una vez señalado el *breakpoint*, ejecutamos el *script*,

```
>> trect

h =
```

3.1477

```
12 s=a/h
K>>
```

Matlab nos indica que ha detenido la ejecución del programa en la línea marcada por el *breakpoint* (línea 12 en el ejemplo). además vuelve a mostrar el *prompt* en la ventana de comandos, pero esta vez precedido por la letra k, para indicarnos que ha entrado en modo de depuración.

A partir de aquí Matlab pone a nuestra disposición las herramientas de depuración, la figura 2.10 muestra la línea en que se ha parado la ejecución del programa, señalada con una flecha verde, y algunas de estas herramientas. Básicamente nos da la posibilidad de ejecutar el código paso a paso, de entrar e ir paso a paso en las funciones que llama nuestro programa, o de continuar la ejecución hasta que el final del programa o hasta el siguiente *breakpoint* activo. Para dominar los detalles del depurador se aconseja leer la ayuda de Matlab.

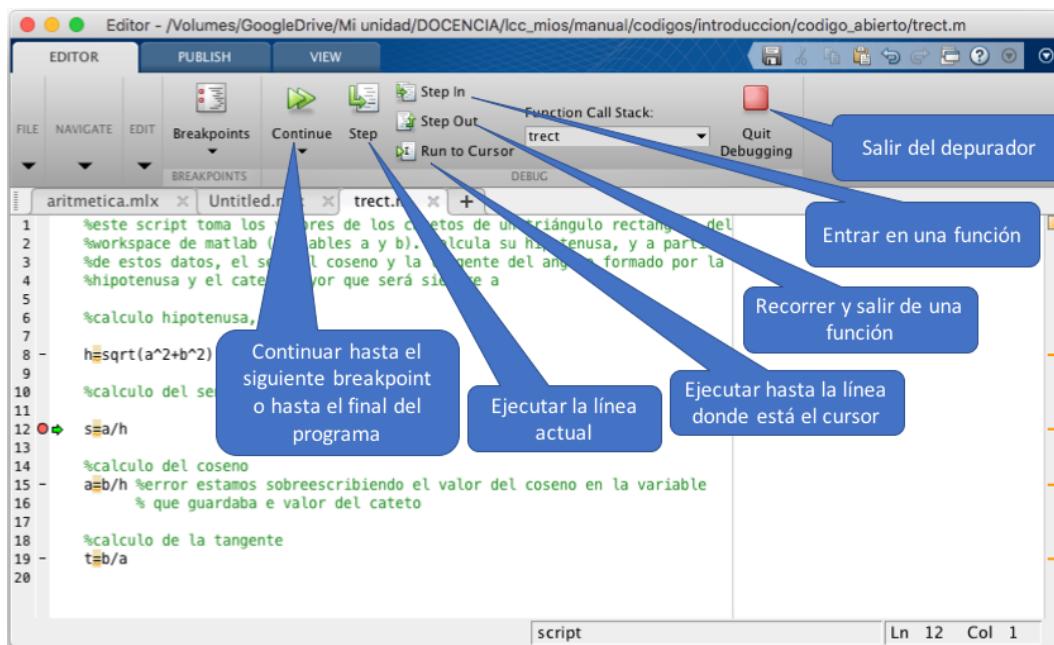


Figura 2.10: Parada de programa en breakpoint y herramientas de depuración

Si pulsamos el botón "ejecutar línea actual" Matlab ejecutará la línea de programa señalada con la flecha verde y se parará en la línea siguiente. En cada paso, podemos ver el valor que toman las variables, pidiendo su valor directamente en la ventana de comandos,

```
K>> a
```

```
a =
```

```
0.9531
```

o bien señalando (sin pulsar botones) con el ratón en el editor de texto la variable de la que se trate. En nuestro ejemplo del triángulo rectángulo es muy sencillo avanzar paso a paso en el programa con el depurador, y caer en la cuenta que, cuando se va a calcular la tangente, la variable `a` ya no contiene el valor del cateto.

3. Advertencias. Por último señalar la existencia de los *warnings* no se trata propiamente de errores, sino de simples advertencias de que algo puede funcionar de forma más eficiente, o puede no dar el resultado que esperábamos. En general, cuando se recibe un *warning* al ejecutar un programa, o cuando el editor de Matlab subraya en rojo algún carácter en el editor de textos, se debe corregir el programa para que desaparezcan, aunque propiamente no se trate de errores.

2.5. Control de Flujo

En la sección anterior, se introdujo el modo de escribir programas en Matlab mediante el uso de *scripts* y funciones. En todos los casos vistos, la ejecución del programa empezaba por la primera línea del programa, y continuaba, por orden, línea tras línea hasta alcanzar el final del programa. Se trata de programas en los que el *flujo* es lineal, porque los resultados de cada línea de programa se van obteniendo regularmente uno detrás de otro.

Hay ocasiones en las que, por diferentes razones que expondremos a continuación, puede interesar nos alterar el orden en que se ejecutan las sentencias de un programa, bien repitiendo una parte de los cálculos un determinado número de veces o bien ejecutando unas partes de código u otras en función de que se satisfagan unas determinadas condiciones.

El control del orden en que se ejecutan las sentencias de un programa es lo que se conoce con el nombre de *control de flujo*. Veremos dos tipos principales de control de flujo: El flujo condicional y los bucles.

2.5.1. Flujo condicional.

Empezaremos con un ejemplo sencillo de cómo y para qué condicionar el flujo de un programa. Supongamos que queremos construir un programa que reciba como variable de entrada un número cualquiera y nos muestra un mensaje por pantalla si el número es par.

Para ello, podríamos hacer uso de la función `rem` (ver tabla 2.5). Si el resto de la división entre dos del número suministrado a la función es cero, se trata de un número par; si no, es un número impar. Podríamos hacer uso de operadores relacionales, en particular de `==` para comprobar si el resto de la división entre dos es cero. Por último necesitaríamos algún mecanismo que permitiera al programa escribir un mensaje solo cuando el número introducido sea par.

if - elseif - else - end. El mecanismo que necesitamos nos lo suministra la estructura `if` de Matlab. Veamos en primer lugar el código del ejemplo del que venimos hablando,

```
function espar(x)
%Este programa recibe un numero entero como variable de entrada. y muestra
%por pantalla un mensaje indicando si el numero recibido es par o impar.

%Calculamos el resto de la division por dos
resto=rem(x,2);

%Empleamos una estructura if - else -end para decidir que mensaje mostrar
```

```

if resto==0
    %si el resto es cero el número es par
    disp('el número es par')
end

```

El programa toma un número como variable de entrada y calcula el resto de su división entre dos. A continuación entra en una parte de código especial, que se inicia con la palabra clave **if** y termina con la palabra clave **end**.

La palabra clave **if** va siempre seguida de una expresión que da un resultado lógico: verdadero (1) o falso (0). Esta expresión puede ser cualquier combinación válida de expresiones relacionales o lógicas. Esta expresión lógica, que sigue al **if** constituye una condición. El programa seguirá ejecutando las siguientes líneas solo si la condición se cumple; si no, se las saltará hasta llegar a la expresión **end**.

En nuestro ejemplo, se emplea una expresión relacional sencilla **resto==0**. Si se cumple, el programa ejecutará la siguiente línea de programa, escribiendo en la ventana de comandos la frase “el número es par” si no se cumple el programa se la salta, llega hasta el **end**, y no escribe nada por pantalla.

Acabamos de ver la estructura condicional **if** más sencilla posible. Podríamos complicarla un poco pidiendo que también nos saque un mensaje por pantalla cuando el número sea impar. Esto supone incluir en nuestro programa una disyuntiva; si es par el programa debe hacer una cosa y si no, debe hacer otra. Para incluir este tipo de disyuntivas en una estructura **if**, se emplea la palabra clave **else**. Veamos nuestro ejemplo modificado,

```

function espar(x)
%Este programa recibe un numero entero como variable de entrada. y muestra
%por pantalla un mensaje indicando si el numero recibido es par o impar.

%Calculamos el resto de la division por dos
resto=rem(x,2);

%Empleamos una estructura if - else -end para decidir que mensaje mostrar

if resto==0
    %si el resto es cero el número es par
    disp('el número es par')
else
    %si el resto no es cero el número es impar
    disp('el número es impar')
end

```

La palabra clave **else** marca ahora la disyuntiva, si el número es par, el programa ejecuta las líneas de código entre el **if** y el **else**, si el número no es par ejecutará las líneas entre el **else** y el **end**.

La estructura **if** admite todavía ampliar el número de posibilidades de elección mediante la palabra clave **elseif**. Al igual que con **if**, **elseif** va seguido de una expresión lógica que establece una condición, si se cumple se ejecutará el código de las líneas siguientes, si no se cumple, el programa saltará a la siguiente línea que contenga una palabra clave: otro **elseif**, un **else** o directamente el **end** que marca el final de la parte de código condicional. Para ver cómo funciona, vamos a modificar nuestro ejemplo anterior, para que, si el número introducido no es divisible por dos, compruebe si es divisible por tres,

```
function divis(x)
%Este programa recibe un numero entero como variable de entrada. y muestra
%por pantalla un mensaje indicando si el numero recibido es par. Si no es
%par, comprueba si es divisible por 3 y si lo es muestra un mensaje por
%pantalla indicandolo. Si no es par ni divisible por tres muestra un
%mensaje diciendo que no es par ni divisible por tres.
```

%Empleamos una estructura if-elseif-else-end para decidir que mensaje mostrar

```
if rem(x,2)==0
    %si el resto es cero el número es par
    disp('el número es par')
elseif rem(x,3)==0
    %El número es divisible por tres
    disp('el número es divisible por 3')
else
    %el numero no es par ni divisible por tres
    disp('el número no es par ni divisible por 3')
end
```

Si llamamos ahora a la función dando como valor de entrada un número par, Ejecutará el código situado debajo del `if` y antes del `elseif` y se saltará todo lo demás hasta llegar al `end`. Si el número introducido no es par, pero es divisible por tres, se saltará el código situado por debajo del `if`, ejecutará el código contenido debajo del `elseif` hasta el `else` y saltará el resto del código hasta llegar al `end`. Por último si el número no es par ni divisible por tres, solo ejecutará el código situado por debajo del `else`.

Un aspecto que debemos resaltar, es que el programa ejecutará el código correspondiente a la primera condición que se cumpla, y se saltará el resto hasta llegar al `end`. Así por ejemplo, si en nuestro ejemplo introducimos el número 6, el programa nos mostrará el mensaje “el número es par”, puesto que ésta es la primera condición que se cumple, pero nunca nos mostrará el mensaje “el número es divisible por 3”. Porque una vez comprobada y cumplida la primera condición (ser par) el programa salta directamente al `end` final de la estructura, sin comprobar nada más. La figura 2.11 muestra el esquema completo de una estructura `if`. Los términos entre paréntesis pueden estar o no presentes en una implementación concreta.

Estructuras if anidadas. En el ejemplo anterior, hemos visto cómo, si el número introducido en la función era par y además divisible por tres, el programa nunca nos informaría de esta segunda propiedad, debido al carácter excluyente de la estructura `if`. Una manera de resolver este problema, es mediante el uso de estructuras `if` anidadas. La idea es muy sencilla, se construye una estructura `if` para comprobar una determinada condición, si esta se cumple, dentro de su código se construye otra estructura `if` para comprobar una segunda condición, y así sucesivamente, todas las veces que sea necesario. Si modificamos nuestro ejemplo anterior, incluyendo un `if` anidado,

```
function divis23(x)
%Este programa recibe un numero entero como variable de entrada. y muestra
%por pantalla un mensaje indicando si el numero recibido es par.
%si el numero es par y divisible entre tres,
%si es divisible entre tres
```

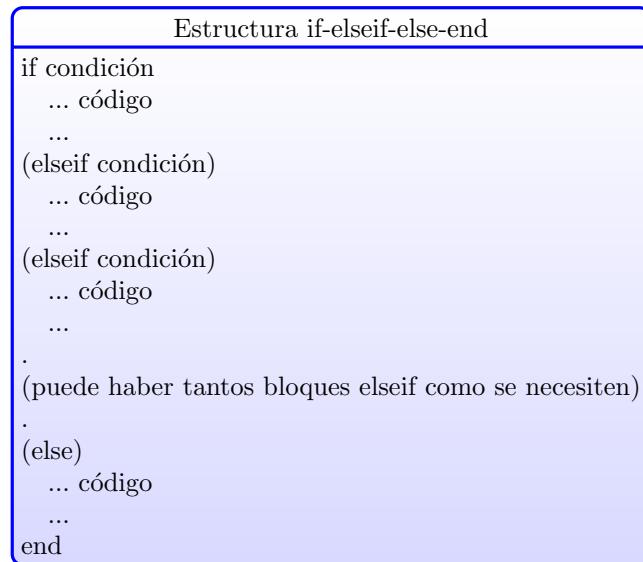


Figura 2.11: Esquema general de la estructura de flujo condicional **if** los términos escritos entre paréntesis son opcionales.

```
%Si no es par ni divisible por tres
```

```
%Empleamos una estructura if - else -end
%y un if anidado para decidir que mensaje mostrar

if rem(x,2)==0
    %si el resto es cero el número es par
    %comprobamos con un if anidado si además es divisible entre tres
    if rem(x,3)==0
        disp('el número es par y divisible entre tres')
    else
        disp('el número es par')
    end
elseif rem(x,3)==0
    %El número es divisible entre tres
    disp('el número es divisible entre 3')
else
    %el numero no es par ni divisible entre tres
    disp('el número no es par ni divisible entre tres')
end
```

Switch-case-otherwise. Se trata de otra estructura que permite también ejecutar una parte u otra de código de acuerdo con unas condiciones preestablecida. Estas condiciones se presentan en forma de *casos*, el programa comprueba al llegar a la estructura **switch** de qué caso se trata y ejecuta el código correspondiente. La figura 2.12 muestra la forma general que toma una estructura

```
switch.
```

La estructura `switch`, compara el valor contenido en la variable de entrada a la estructura con las expresiones contenidas en los casos, ejecutando el primer caso para el que coincidan. Si no encuentra ninguno, ejecuta entonces el código contenido debajo de la sentencia `otherwise`.

Veamos un ejemplo muy sencillo,

```
function signo(x)
%este programa emplea una estructura switch para informarnos del signo de
%un número.

%NOTA EL PROGRAMA NO COMPRUEBA QUE LA VARIABLE DE ENTRADA X SEA UN NUMERO,
%SI ES UN VECTOR O UNA MATRIZ EL RESULTADO NO TIENE SENTIDO

s=sign(x); %obtnemos el signo del número mediante la función sign

%construimos la estructura switch,

switch s
    case 1
        disp('el número es positivo')
    case -1
        disp('el número es negativo')
    otherwise
        disp('el número es cero')
end
```

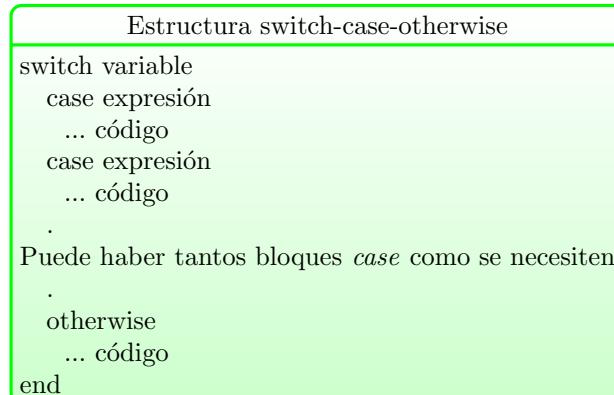


Figura 2.12: Esquema general de la estructura switch-case-otherwise

La función del ejemplo emplea a su vez la función de Matlab `sign` para obtener el signo (1, -1, 0) del número introducido. Guarda el resultado de esta operación en la variable `s`, que será precisamente la variable de entrada a la estructura `switch`. Los casos se limitan a chequear los posibles valores de la variable y enviar a la ventana de comandos el mensaje correspondiente.

2.5.2. Bucles

En ocasiones, es preciso repetir una operación un número determinado de veces o hasta que se cumple una cierta condición. Los lenguajes de alto nivel poseen estructuras específicas, para repetir la ejecución de un trozo de programa las veces que sea necesario. Cada repetición recibe el nombre de iteración. Estas estructuras reciben el nombre genérico de bucles. Vamos a ver dos tipos: los bucles **for** y los bucles **while**.

Bucles for. Un bucle **for** repite las sentencias contenidas en el bucle un determinado número de veces, es decir realiza un número fijo de iteraciones. La estructura general de un bucle **for** se muestra en la figura, [2.13](#)

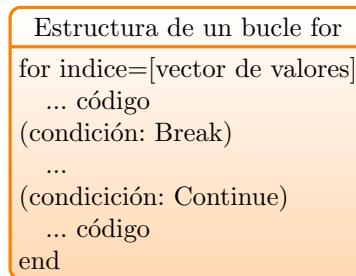


Figura 2.13: Esquema general de la estructura de un bucle **for** los términos escritos entre paréntesis son opcionales.

El bucle empieza con la palabra clave **for**, seguida de una variable a la que hemos dado el nombre genérico de índice. Esta variable irá tomando sucesivamente los valores de los elementos contenidos en el [vector de valores]. El código contenido en el bucle, desde la línea siguiente al **for** hasta el **end** se ejecutará tantas veces como valores tenga el vector de valores. Antes de hablar de las sentencias **break** y **continue**, veamos algunos ejemplos.

```

function y=demofor(x)
%este programa emplea un bucle for sencillo para ir mostrando uno a uno
%los elementos del vector de entrada x por pantalla. además los suma y
%guarda el resultado total en el vector y,

```

```
y=0; %iniciamos la suma a cero
```

```

for i=x
    disp(i)
    y=y+i;
end

```

Si ejecutamos el programa, usando como entrada el vector $d=[1 \ 9 \ 4 \ 18]$,

```
>> d=[1 9 4 18]
```

```
d =
```

```
1      9      4      18
```

```
>> suma=demofor(d)
```

```
1
```

```
9
```

```
4
```

```
18
```

```
suma =
```

```
32
```

Una vez que el programa llega al bucle **for**, iguala la variable **i** al primer valor contenido en el vector de entrada, lo muestra por pantalla y añade su valor a la variable de salida. cuando llega al **end** del bucle for comprueba que todavía quedan valores del vector de entrada por recorrer, así que vuelve al principio del **for**, igual la variable **i** al segundo valor del vector de entrada, suma dicho valor a la variable de salida, llega al **end** y así sucesivamente hasta que haya recorrido todos los valores del vector de entrada.

El uso más habitual de los bucles, es para recorrer elementos de un vector o de una matriz. por esta razón, lo más frecuente es que no se de explícitamente el vector cuyos elementos debe recorrer el índice del **for**, sino que se construya empleado el operador **:**,

```
for indice=principio:incremento:final
```

Veamos un ejemplo sencillo para obtener el vector suma de dos vectores,

```
function s=sumafor(x,y)
%este programa emplea un bucle for sencillo para sumar dos vectores
%primero comprueba si son del mismo tamaño. Si no lo son da un mensaje de
%aviso

l1=length(x);
l2=length(y);
if l1==l2
    %construimos un vector de ceros del mismo tamaño que x e y paraa
    %guardar el resultado de la suma,
    s=zeros(size(x));
    %si son iguales los suma elemento a elemento usando un bucle for
    for i=1:l1
        s(i)=x(i)+y(i);
    end
else
    disp('los vectores son de distinto tamaño')
end
```

En la figura 2.13, aparecen dos sentencias opcionales, **break** y **continue**.

La sentencia **break**, permite terminar el bucle **for** antes de que haya terminado de realizar todas las ejecuciones previstas. La sentencia **break** va siempre incluida dentro de una condición que, si se cumple interrumpe la ejecución del bucle. Por ejemplo, podemos emplear un bucle **for** y una sentencia **break**, para buscar la primera vez que un determinado número aparece en un vector,

```

function posicion=buscanum(x,n)
%este programa emplea un bucle for y un break para buscar en el vector x,
%la primera vez que aparece el número n

%obtenemos la longitud del vector
l1=length(x);
%iniciamos la variable posicion a un valor absurdo
posicion=-1;
for i=1:l1
    if x(i)==n
        posicion=i;
        break
    end
end
if posicion== -1
    disp('el numero pedido no se encuentra en el vector x')
end

```

El programa toma como variables de entrada un vector y el número que debe buscar dentro del vector. Construye un bucle **for**, con el mismo número de iteraciones que el tamaño del vector. El bucle va comparando los elementos del vector con el número pedido. Cuando encuentra un elemento igual, se ejecuta el código de la estructura **if** contenida en el bucle; la variable **posición** toma el valor del índice **i** del bucle y la sentencia **break** interrumpe la ejecución del bucle, saltando al final del mismo. Por último, se ha añadido una condición al terminar el bucle, para el caso de que se hayan recorrido todos los elementos del vector sin encontrar el número pedido.

La sentencia **continue**, se emplea para hacer que una determinada iteración interrumpa su ejecución y salte directamente al comienzo de la siguiente iteración. Debe ir, como en el caso de la sentencia **break**, incluida en una estructura condicional. Veamos un ejemplo para entender como funciona. Se trata de un programa que admite como entrada un vector de cualquier longitud y devuelve como salida otro vector que contiene solo los números pares contenidos en el vector de entrada,

```

function pares=buscapar(x)
%este programa emplea un bucle for y un continue para construir un vector
%de salida con los numeros pares contenidos en el vector de entrada.

%obtenemos la longitud del vector
l1=length(x);
%iniciamos el vector de salida a un vector vacío,
pares=[];
for i=1:l1
    if rem(x(i),2)^=0
        continue
    end
    pares=[pares x(i)];
end

```

El funcionamiento es muy sencillo, en cada iteración la sentencia **if** comprueba si el número es par. Si no lo es, entra en el código de la estructura **if**, ejecuta la sentencia **continue**, y se salta el resto del código del bucle, volviendo directamente a empezar la siguiente iteración.

bucles for anidados. Los bucles `for` pueden anidarse unos dentro de otros de modo análogo a como se hace con las estructuras `if`. Veamos un ejemplo de uso, muy común; calcular la suma de dos matrices,

```
function s=suma_mat(x,y)
%este programa emplea dos bucles for anidados para obtener la suma de dos
%matrices.

%obtenemos el tamaño de las matrices,
t1=size(x);
t2=size(y);
if t1(1)==t2(1)&& t1(2)==t2(2)
    %si las matrices tienen el mismo tamaño pueden sumarse...
    %construimos una matriz de dicho tamaño para guardar el resultado de
    %la suma
    s=zeros(size(x));
    %construimos un bucle que recorre las filas de ambas matrices
    for i=1:t1(1)
        %y dentro, anidamos un bucle que recorre las columnas
        for j=1:t1(2)
            %Empleando ambos índices para ir sumando los elementos de las
            %dos matrices,
            s(i,j)=x(i,j)+y(i,j);
        end
    end
else
    %si no tienen el mismo tamaño no se pueden sumar...
    disp('las matrices no son del mismo tamaño')
end
```

Es interesante observar en este ejemplo el funcionamiento de los dos bucles anidados. Cada interacción del bucle exterior, avanza una fila, en el recorrido de las matrices, cada iteración del bucle interior recorre todos los elementos de la fila indicada por el bucle exterior.

bucle while. Este bucle tiene la misma finalidad que un bucle `for`, repetir un trozo de código un determinado número de veces. Lo que cambia, es el mecanismo que determina cuantas iteraciones realizará el bucle. En el caso de un bucle `while` las iteraciones se repiten un número indefinido de veces mientras se cumpla una determinada condición impuesta al principio del bucle. La figura 2.14 muestra la estructura general de un bucle `while`.

Veamos un ejemplo sencillo.

```
function n=potencia(x,max)
%este programa emplea un bucle while para calcular la potencia a la que hay
%que elevar un número x para que el resultado sea mayor que otro
%determinado número max
%un número.

%NOTA EL PROGRAMA NO COMPRUEBA QUE LA VARIABLE DE ENTRADA X SEA UN NUMERO,
%Si ES UN VECTOR O UNA MATRIZ EL RESULTADO NO TIENE SENTIDO
```

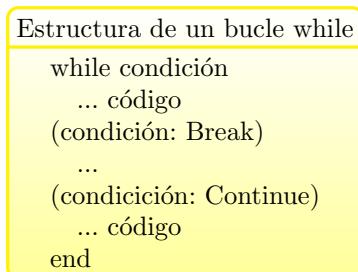


Figura 2.14: Esquema general de la estructura de un bucle `while` los términos escritos entre paréntesis son opcionales.

```

pot=1;
n=0;
while pot<max %mientras la potencia calculada sea menor que max
    n=n+1;
    pot=x^n;
end

```

El programa emplea un bucle `while` para calcular el exponente mínimo al que hay que elevar un número para que rebase una determinada cantidad. El bucle `while` se ejecuta mientras la potencia calculada sea menor que la variable `max`; una vez que la potencia rebasa dicho valor el bucle deja de ejecutarse. Un aspecto muy importante del bucle `while` es que al programarlo hay que asegurarse de que dentro del bucle existe la posibilidad de cambiar la condición de entrada. Si no, el programa no podrá terminar nunca el bucle⁶.

Las sentencias `break` y `continue` son idénticas a las descritas en el caso de los bucles `for`, por lo que no insistiremos más sobre el asunto.

Bucles while anidados. Del mismo modo que se anidan los bucles `for`, es posible anidar bucles `while`. Como un ejemplo, vamos a reproducir el programa desarrollado antes para sumar dos matrices, empleando ahora bucles `while`.

```

function s=suma_while(x,y)
%este programa emplea dos bucles while anidados para obtener la suma de dos
%matrices.

%obtenemos el tamaño de las matrices,
t1=size(x);
t2=size(y);
if t1(1)==t2(1)&& t1(2)==t2(2)
    %si las matrices tienen el mismo tamaño pueden sumarse...
    %construimos una matriz de dicho tamaño para guardar el resultado de
    %la suma
    s=zeros(size(x));
    %construimos un bucle que recorre las filas de ambas matrices
    i=1; %iniciamos un contador para las filas
    while i<=t1(1)

```

⁶Cuando se produce esta situación por un error en el diseño del programa, el bucle se puede parar pulsando a la vez las teclas `ctrl+c`

```
%y dentro, anidamos un bucle que recorre las columnas
j=1; %iniciamos un contador para las columnas
while j<=t1(2)
    %Empleando ambos índices para ir sumando los elementos de las
    %dos matrices,
    s(i,j)=x(i,j)+y(i,j);
    j=j+1; %vamos incrementando el indice de columnas
end
i=i+1; %vamos incrementando el indice de filas
end
else
    %si no tienen el mismo tamaño no se pueden sumar...
    disp('las matrices no son del mismo tamaño')
end
```

El ejemplo, es más complicado de programar y menos eficiente que si usáramos bucles `for`. La razón de incluirlo es puramente ilustrativa. En general, un bucle `while` debe utilizarse solo cuando el número de iteraciones que se precisa realizar no es fijo, sino que depende de alguna condición propia de los resultados que se van obteniendo a medida que se van realizando iteraciones.

2.5.3. Funciones recursivas.

Una función recursiva es una función que se llama a sí misma. Hemos esperado hasta aquí para hablar de ellas porque, de alguna manera, se comportan como un bucle y necesitan una condición de salida para dejar de llamarse a sí mismas y terminar su ejecución. En general, son delicadas de manejar y tienden a consumir mucha memoria ya que cada vez que la función se llama a sí misma necesita crear un nuevo espacio de memoria independiente. Veamos un ejemplo de función recurrente que permite obtener el término enésimo de la sucesión de Fibonacci

$$f_0 = 0, f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 5, f_6 = 8, \dots f_i = f_{i-1} + f_{i-2} \dots$$

La sucesión empieza con los términos 0 y 1 y a partir de ahí cada término es la suma de los dos anteriores. Podemos convertir directamente esta definición en código,

```
function s=fibonacci(n)
%obtiene el término enésimo de la sucesión de fibonacci empleando una
%función recursiva. que funcione no quiere decir que sea eficiente...
%n número del término que se desea obtener

if n<2
    %el valor del término de fibonacci es él mismo
    s=n;
else
    %si no es la suma de los dos anteriores...
    s=fibonacci(n-1)+fibonacci(n-2);
end
```

Si n es menor que dos, la función da como valor el término correspondiente (1 o 0). Si n es mayor que dos, vuelve a llamarse a sí misma con entrada $n-1$ y $n-2$, para calcular el valor enésimo de la sucesión a partir de la suma de los dos anteriores, la función se irá llamando a sí misma hasta llegar a $n < 2$. A partir de ahí irá devolviendo los valores obtenidos en cada llamada hasta obtener el enésimo término.

2.5.4. Algoritmos y diagramas de flujo.

Un algoritmo es, en la definición de la Real Academia:

Un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

La palabra algoritmo ha llegado hasta nosotros transcrita del nombre del matemático árabe *Al-Juarismi* (circa 780-850 dc). En programación los algoritmos son importantes, porque suponen un paso previo a la creación de un programa.

Habitualmente, partimos de un problema para el que tenemos un enunciado concreto. Por ejemplo: obtener los n primeros números primos.

El siguiente paso, sería pensar y definir un algoritmo que permita resolver nuestro problema, es importante caer en la cuenta de que un mismo problema puede resolverse, en muchos casos, por distintos caminos. Por tanto es posible diseñar distintos algoritmos para resolver un mismo problema. Un posible algoritmo para el problema de los números primos sería el siguiente,

- Considerar 2 como el primer números primo. (un número primo es aquel que solo es divisible por si mismo o por uno.)
- Recorrer todos los números impares desde 3 hasta que se complete el número n de números primos solicitados.
- Para cada número, probar a dividirlo por todos los primos obtenidos hasta ese momento. Si no es divisible por ninguno, el número es primo y se guarda, si es divisible por alguno de ellos, se interrumpe el proceso y se prueba con el siguiente.

En ocasiones, facilita la comprensión de un algoritmo representarlo gráficamente mediante un diagrama de flujo. Los diagramas de flujo emplean símbolos bien definidos para representar los distintos paso de un algoritmo y flechas para indicar la relación entre ellos; la relación en la que la información *fluye* de un paso del algoritmo a otro.

No hay una norma rígida para realizar un diagrama de flujo, el grado de detalle con que se describe el algoritmo va en función de las necesidades del programador, o de los destinatarios a quienes va dirigido el diagrama. La idea fundamental es que facilite la comprensión del algoritmo. Se utilizan diversos símbolos para indicar, procedimientos, condiciones, almacenamiento de resultados, etc. La figura 2.15 muestra los tres símbolos más empleados.

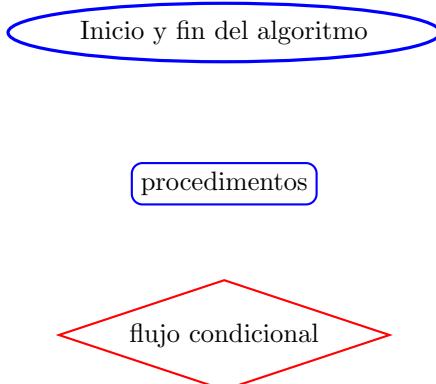


Figura 2.15: Símbolos empleados en diagramas de flujo

Para indicar el inicio y el fin de un algoritmo se emplea como símbolo una elipse. Para indicar un procedimiento concreto, como por ejemplo realizar un cálculo, asignar un valor a una variable, etc, se emplea como símbolo un rectángulo. Por último se emplea un rombo como símbolo, para representar una condición.

Los símbolos se relacionan mediante flechas que indican el sentido en que se ejecuta el algoritmo. los rombos suelen tener dos flechas de salida marcadas con las palabras “si” y “no”, para indicar por donde sigue el flujo de información dependiendo de si la condición representada se cumple o no.

Por último un bucle se representa habitualmente mediante una flecha que devuelve el flujo a un símbolo ya recorrido anteriormente.

La figura 2.16 muestra un posible diagrama de flujo para el problema de los números primos. Como puede observarse, contiene más información que la versión que hemos dado del algoritmo descrito con palabras.

Las líneas que marcan los flujos de información nos indican que será necesario implementar un bucle exterior hasta que se complete el número n de primos solicitados y un bucle interior que deberá comprobar si cada nuevo número impar que probamos, es divisible por los números primos encontrados hasta ese momento.

Hay una tercera condición que debe interrumpir la comprobación para el primer número primo que resulte ser divisor del número que se está comprobando.

Es fácil extraer del diagrama de flujo las estructuras de programación que necesitaremos para elaborar un código que nos permita resolver el problema planteado. Por ejemplo, parece lógico implementar el bucle exterior empleando un bucle `while`, implementar el bucle interior con un `for`, que de tantas iteraciones como primos se han encontrado hasta ese momento, Emplear un `break` para interrumpir la comprobación, etc.

Por supuesto es posible realizar un diagrama de flujo más detallado, en el que incluso se incluya explícitamente parte del código que se va a utilizar. Por ejemplo se podría indicar que se empleará la función `rem` para comprobar si un número es divisible entre otro. Sin embargo, hay que tener cuidado para evitar que un exceso de detalle dificulte entender la lógica del algoritmo contenida en el diagrama.

Por último el algoritmo se codifica dando lugar a un programa de ordenador que permite resolver el problema. Para ello, hay que identificar las instrucciones del algoritmo con estructuras de programación validas: bucles, condicionales, etc. Veamos un posible código para generar números primos, siguiendo el algoritmo descrito.

```
function p=primos(n)
%generador de números primos. Genera un vector con los primeros n números
%primos. (si n es muy grande tardará mucho en acabar.

%Creamos un vector de salida en el que incluimos el 2 como primer número
%primo...
p=zeros(1,n);
p(1)=2;

%iniciamos un contador a 1, puesto que ya tenemos el primer número primo
j=1;

%Creamos un segundo contador que recorra los números impares, a a partir
%del 3
s=3;
```

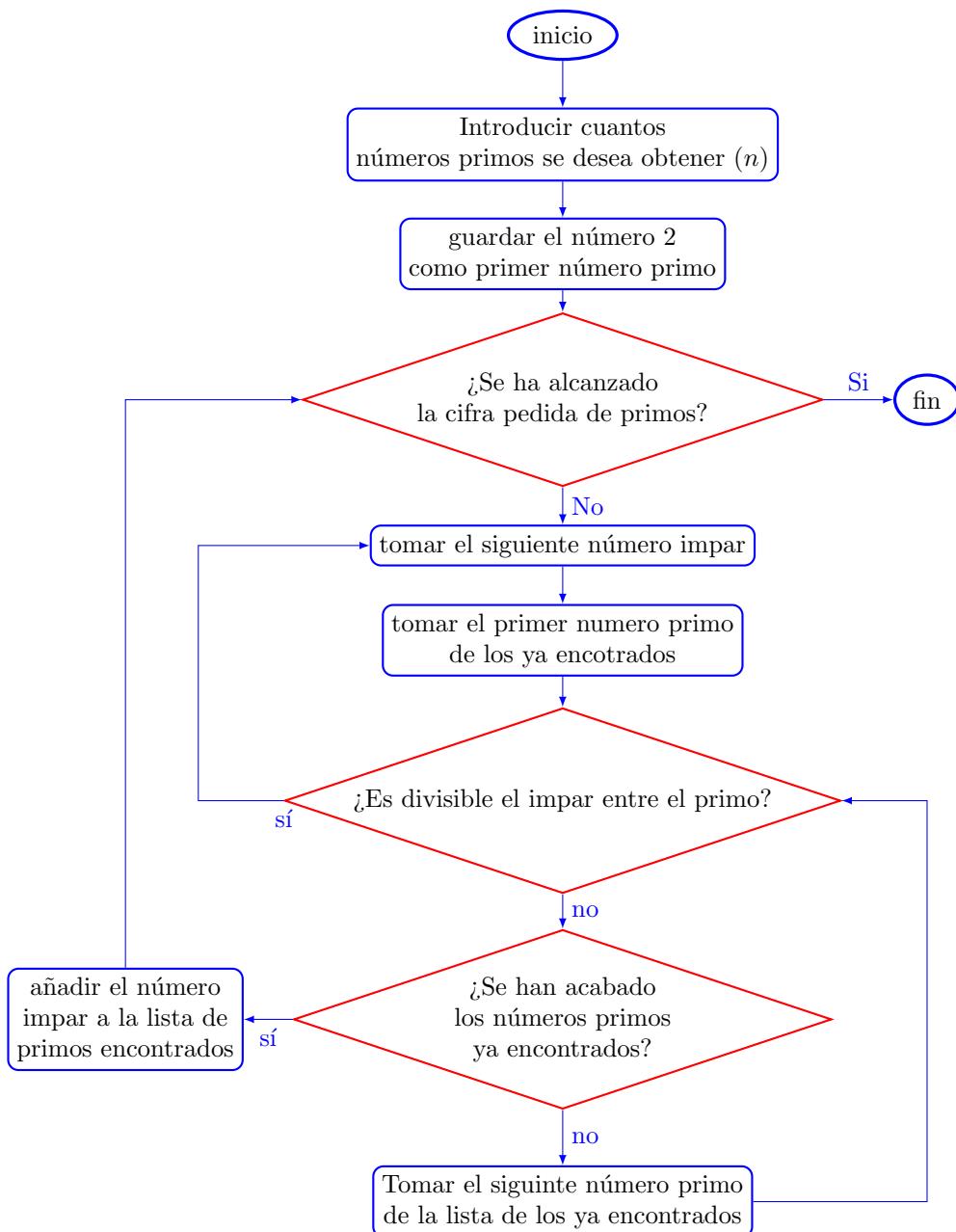


Figura 2.16: Diagrama de flujo para el problema de los números primos

```
%Creamos un bucle while, que realiza las iteraciones necesarias hasta tener
%los n números primos deseados.
while j<n
    %Creamos un bucle for que recorra todos los primos encontrados hasta la
    %fecha
    pr=1; %Activamos aviso de numero primo
    for i=2:j
        %Calculamos el resto de la división entera con cada primo de p
        if rem(s,p(i))==0
            %Si el numero es divisible por algun primo anterior
            pr=0; %Desactivamos el aviso: el número no es primo...
            break %... y cortamos la búsqueda
        end
    end

    if pr==1 %Si el número es primo
        p(j+1)=s; %Lo guardamos...
        %... e incrementamos el contador de números primos encontrados
        j=j+1;
    end
    %Por ultimo, generamos un nuevo candidato para ver si es primo...
    s=s+2;
end
```

2.6. Representación Gráfica

La posibilidades gráficas constituyen, junto a la facilidad para manejar matrices, uno de los aspectos más atractivos de Matlab como herramienta de cálculo científico. Matlab permite realizar gráficos en dos y tres dimensiones de muy diversos tipos.

2.6.1. El comando plot y las figuras en Matlab.

plot. El comando de dibujo más sencillo de Matlab es el comando **plot**. La filosofía de dibujo es muy sencilla se pasan como variables de entrada dos vectores, el primero de ellos con las coordenadas x y el segundo con las correspondientes coordenadas y de los puntos que se desea dibujar. Si no se indica nada, Matlab unirá los puntos mediante líneas rectas. Supongamos que deseamos representar gráficamente los puntos (x, y) de la siguiente tabla de datos,

Cuadro 2.6: Datos de prueba

x	y
0	0
2	3
-1	2
-2	-4

Para ello, lo primero que hacemos es construir dos vectores; uno con las coordenadas x de los puntos,

```
>> x=[0 2 -1 -2]
```

```
x =
```

```
0      2     -1     -2
```

y el otro con las coordenadas y de los puntos,

```
>> y=[0 3 2 -4]
y =
```

```
0      3     2     -4
```

Por último empleamos el comando `plot`, dando como variables de entrada los dos vectores construidos,

```
>> plot(x,y)
```

Matlab responde al comando abriendo una ventana gráfica, como la que muestra la figura 2.17, con la figura correspondiente a los puntos de la tabla, unidos mediante líneas rectas.

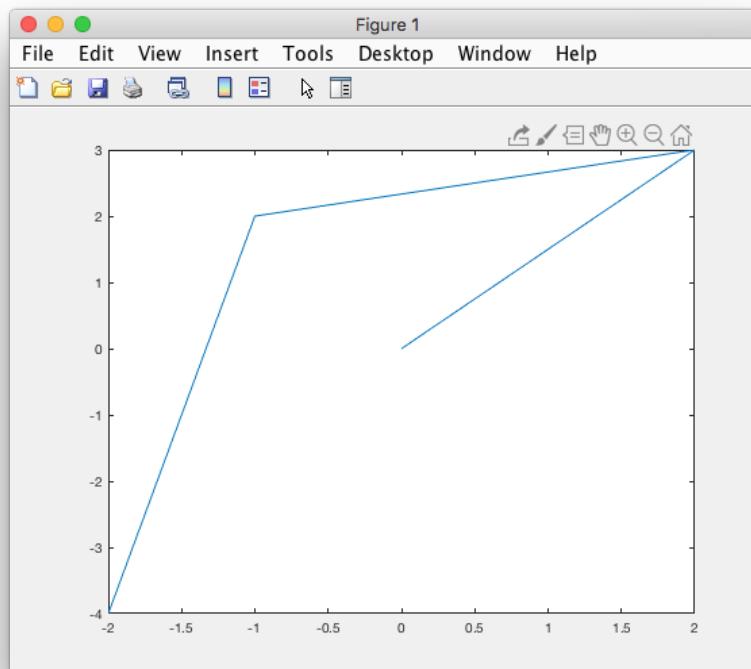


Figura 2.17: Ventana gráfica de Matlab. representación de los punto de la tabla 2.6

La ventana gráfica de Matlab, tiene en su parte superior una barra de herramientas y un menú desplegable con funciones específicas para la manipulación de los gráficos. Se aconseja leer la ayuda de Matlab sobre el uso de dichas herramientas.

Una de las opciones del menú desplegable, permite guardar la figura generada como un archivo gráfico. Además, es posible mediante otra de las opciones del menú copiar la figura y pegarla posteriormente en un editor de texto⁷. Si copiamos la figura 2.17 y la pegamos directamente en el texto obtendríamos un gráfico como el de la figura, 2.18. A partir de ahora importaremos de esta manera todas las figuras que construyamos con Matlab.

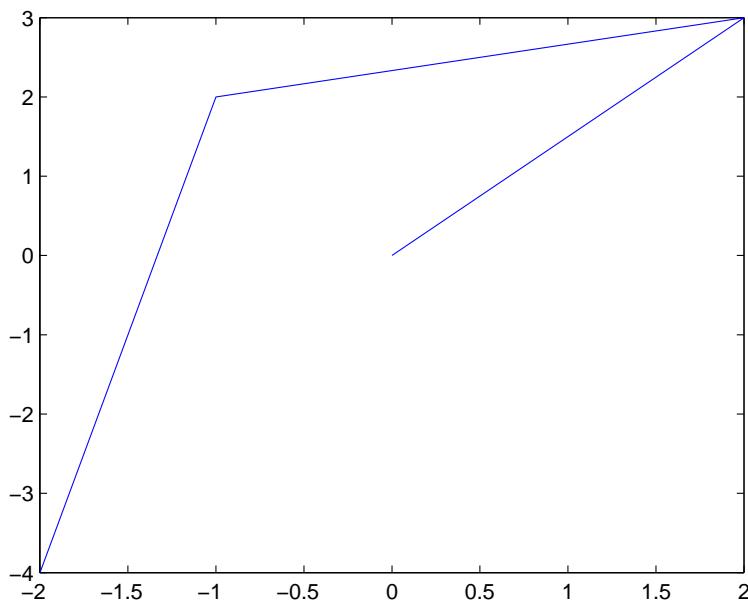


Figura 2.18: gráfico de los puntos de la tabla 2.6 obtenida con el comando `plot`

El comando `plot` admite un tercer parámetro de entrada. Se trata de símbolos, escritos entre comillas simples, que permiten definir:

- El tipo de línea que se empleará en el gráfico. por ejemplo `plot(x,y,'-.'`) une los puntos mediante una línea de puntos y guiones.
- El símbolo que se empleará para representar los puntos. Por ejemplo, `plot(x,y,'o')` dibuja un círculo en la posición de cada punto y no los une entre sí mediante líneas rectas.
- El color que se empleará para dibujar. Por ejemplo, `plot(x,y,'r')`. Dibuja la gráfica en color rojo.

Si no se define este tercer parámetro, `plot` dibujará los gráficos, por defecto, en color azul, uniendo los puntos con líneas continuas y no usará ningún símbolo para dibujar los puntos individuales.

La tabla 2.7 muestra los símbolos disponibles para dibujar con el comando `plot`.

Se puede combinar un símbolo de cada tipo en un mismo `plot`. Así por ejemplo si queremos representar los datos de la tabla 2.6 unidos mediante una línea de puntos,

```
plot(x,y,:')
```

⁷Al menos es posible hacerlo así si se trabaja en el sistema operativo Windows de Microsoft.

Cuadro 2.7: tipos de línea y color del comando <code>plot</code>					
Tipo de línea	Símbolo	Tipo de punto	Símbolo	Color	Símbolo
continua	-	punto	.	azul	b
puntos	:	círculo	o	verde	g
puntos y guiones	-.	equis	x	rojo	r
guiones	-	más	+	cyan	c
		asterisco	*	amarillo	y
		diamante	d	negro	k
		triangulo vértice abajo	v	blanco	w
		triangulo vértice arriba	^		
		triangulo vértice izquierda	<		
		triangulo vértice derecha	>		
		triangulo vértice arriba	^		
		cuadrado	s		
		pentágono	p		
		hexágono	h		

Si queremos que pinte solo los puntos sin unirlos con líneas y en color rojo,

```
plot(x,y,'.r')
```

Si queremos que pinte los puntos representados por triángulos con el vértice hacia arriba, unidos mediante una línea continua y en color negro,

```
plot(x,y,'^-k')
```

La figura 2.19 muestra los resultados de las combinaciones de símbolos que acabamos de describir.

Figuras. Cada vez que escribimos en la ventana de comandos de Matlab, un comando gráfico como por ejemplo `plot` Matlab comprueba si existe alguna figura (ventana de gráficos) abierta. Pueden darse entonces tres situaciones distintas.

1. No hay ninguna figura abierta. Matlab crea entonces una figura nueva y representa en ella el gráfico pedido.
2. Hay una figura abierta. Matlab empleará dicha figura para representar el gráfico pedido. Por defecto, Matlab borrará cualquier gráfico anterior que contuviese la figura.
3. Existe más de una figura abierta. Matlab empleará para dibujar la llamada figura activa, que corresponde con la figura que se haya utilizado o que se haya seleccionado por última vez con el ratón.

Es posible crear varias figuras distintas empleando directamente el comando `figure`. Cada vez que lo introduzcamos en la ventana de comandos, Matlab creará una figura nueva asignándole un número (figura 1, 2 ,3 etc.). Si empleamos el comando `figure`, seguido de un número entre paréntesis, `figure(25)`, Matlab creará una nueva figura asignándole dicho número y si ya existe la figura la convertirá en la figura activa. El siguiente script muestra un ejemplo del uso de `figure` y `plot` combinados.

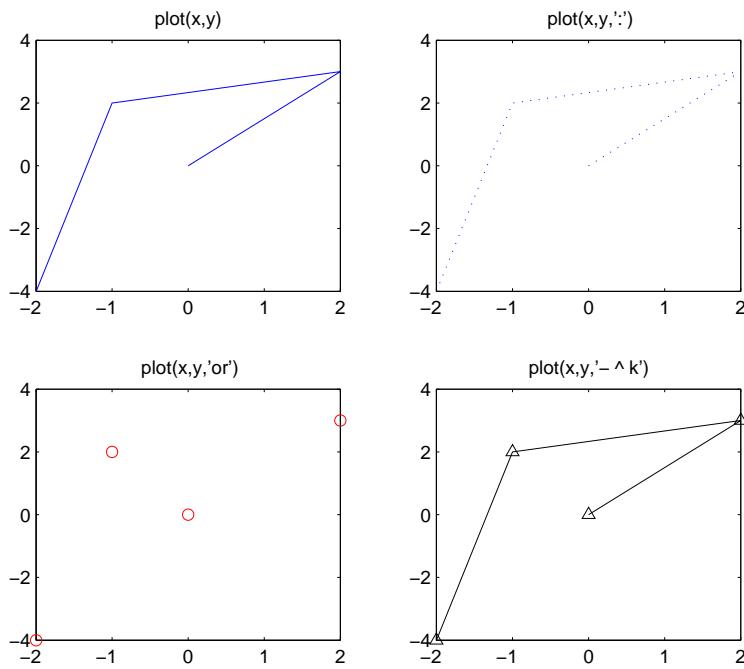


Figura 2.19: Datos de la tabla 2.6 representados mediante distintos tipos de líneas y colores

```
%este script (figuras.m) muestra el uso de los comandos figure y plot para pintar
%varias funciones se aconseja copiarlo y probarlo en Matlab para entender
%mejor como funciona.

%vamos a pintar un trozo de la función e^x, en concreto para el intervalo
%x=[0,1]

%Construimos un vector de 100 puntos equiespaciados en el intervalo [0,1]
x=linspace(0,1,100);

%calculamos el valor de la función e^x para los puntos construidos,
y1=exp(x);

%pintamos los puntos y frente a x,
plot(x,y1) %plot ha construido una figura en Matlab, la figura 1.

%Construimos una segunda figura en Matlab
figure %se ha construido la figura 2

%construimos una tercera figura
```

```

figure %se ha construido la figura 3

%calculamos los valores que tomará la función sin(2*pi*x) para los puntos
%x del intervalo [0,1] que ya tenemos
y2=sin(2*pi*x);

%hacemos activa la figura 2
figure(2)
%pintamos en esta figura los puntos de la función sin...
plot(x,y2)

%volvemos a hacer activa la figura 1
figure(1)
%pintamos ahora los puntos de la de la función y=e^x, pero invertidos x
%frente a y, La grafica anterior se borra y es sustituida por la nueva,
plot(y1,x)

%Creamos una nueva figura asignándole un numero al crearla,
figure(13)

%volvemos activar la figura 3
figure(3)

%volvemos a pintar, ahora en la figura 3, la función y=e^x,
plot(x,y1)

%volvemos a activar la figura 13 y pintamos en ella de nuevo la función
%sin..
figure(13)
plot(x,y2)

```

Como se ha señalado antes, cualquier comando gráfico que se ejecute borra por defecto el contenido anterior de la figura activa. Es posible cambiar este comportamiento, empleando para ello el comando `hold`. Si en la ventana de comandos escribimos `hold on`, a partir de ese momento la ventana activa mantendrá cualquier gráfico que contenga y añadirá a este los nuevos gráficos que se creen. Este comportamiento se mantiene hasta que vuelve a escribirse en la ventana de comandos la sentencia `hold off`. El siguiente script muestra un ejemplo del uso de este comando y La figura 2.20 el gráfico resultante.

```

%ejemplo de uso de hold on para representar dos funciones en el mismo
%gráfico

%vamos a representar las funciones seno y coseno en el intervalo [-pi, pi]

%Creamos un vector de 100 puntos en el intervalo,
x=[-pi:2*pi/99:pi];

```

```
%calculamos el valor de la función seno sobre los puntos x
seno=sin(x);

%calculamos el valor de la función coseno sobre los puntos x
coseno=cos(x);

%pintamos la función seno, con linea continua azul
plot(x,seno)

%le pedimos que mantenga el gráfico creado
hold on

%pintamos encima la función coseno en linea continua roja

plot(x,coseno,'r')
```

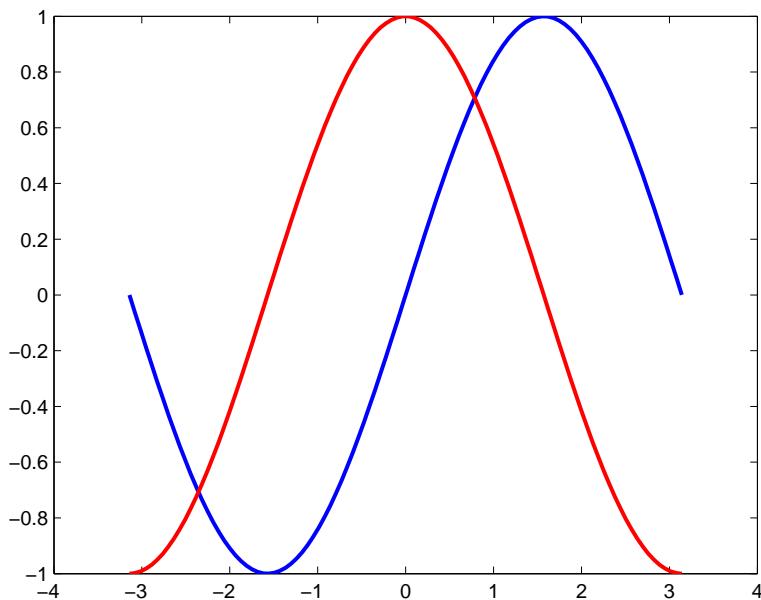


Figura 2.20: gráficas de las funciones seno y coseno en el intervalo $(-\pi, \pi)$. Representadas en la misma figura, usando el comando `hold on`.

Es posible también incluir varios gráficos separados en la misma figura. Para ello se emplea el comando `subplot(i,j,k)`. Este comando divide la figura en un total de $i \times j$ gráficos y activa el situado en la posición k , las posiciones se cuentan fila a fila de arriba a abajo. El siguiente script muestra el uso del comando `subplot`. La figura 2.21 muestra el resultado obtenido.

```
%Este script muestra el uso del comando subplot
```

```
%vamos a crear una figura con 2X3=6 gráficas, se disponen en la figura como
%si fueran los elementos de una matriz...
%usamos el comando subplot de modo que cree el primer eje de los 6
subplot(2,3,1)

%definimos un vector x de puntos equiespaciados en el intervalo (-1,1)
x=linspace(-1,1,20);

%calculamos los valores de polinomio 3x^2+2^x-1
y=3*x.^2+2*x-1;

%dibujamos la función en los ejes
plot(x,y)

%Añadimos rótulos a los ejes
xlabel('eje x')
ylabel('eje y')
%Añadimos un titulo al grafico
title('gráfico 1')

%Generamos los siguientes ejes (a la derecha del anterior)
subplot(2,3,2)
%dibujamos la misma función pero ahora en linea discontinua roja
plot(x,y,:r')

%Añadimos rótulos a los ejes
xlabel('eje x')
ylabel('eje y')
%Añadimos un titulo al grafico
title('gráfico 2')

%Generamos los siguientes ejes (a la derecha del anterior)
subplot(2,3,3)
%dibujamos la misma función pero ahora en linea de punto y raya negra
plot(x,y,'-.k')

%Añadimos rótulos a los ejes
xlabel('eje x')
ylabel('eje y')
%Añadimos un titulo al grafico
title('gráfico 3')

%Generamos los siguientes ejes (debajo de los primeros)
subplot(2,3,4)
%dibujamos la misma función pero ahora solo con circulos azules
plot(x,y,'o')

%Añadimos rótulos a los ejes
xlabel('eje x')
ylabel('eje y')
```

```
%Añadimos un titulo al grafico
title('gráfico 4')

%Generamos los siguientes ejes (a la derecha del anterior)
subplot(2,3,5)
%dibujamos la misma función pero ahora solo con cruce rojas
plot(x,y,'+r')

%Añadimos rótulos a los ejes
xlabel('eje x')
ylabel('eje y')
%Añadimos un titulo al grafico
title('gráfico 5')

%Generamos los últimos ejes (a la derecha del anterior)
subplot(2,3,6)
%dibujamos la misma función pero ahora en linea continua y asteriscos
%negros
plot(x,y,'-*k')

%Añadimos rótulos a los ejes
xlabel('eje x')
ylabel('eje y')
%Añadimos un titulo al grafico
title('gráfico 6')
```

En el ejemplo, se ha hecho usos de algunos comandos para gráficos que permiten introducir títulos. Estos son:

`title`, introduce un título a un gráfico, por ejemplo,

```
title('gráfico de temperaturas')
```

`xlabel`, añade un rótulo al eje x, por ejemplo,

```
xlabel('tiempo en segundos')
```

`ylabel` añade un rótulo al eje y , por ejemplo,

```
ylabel('distancia en metros')
```

2.6.2. Gráficos en 2D

Hasta ahora, hemos visto tan solo el comando `plot`, que nos ha servido para introducir las capacidades gráficas en Matlab. Como hemos visto, `plot` permite representar gráficamente colecciones de datos en dos dimensiones. Hay otros muchos comandos que permiten obtener representaciones *especializadas* de datos en dos dimensiones. A continuación veremos algunos de los más destacables.

fplot. Permite dibujar directamente una función en un intervalo de valores. El nombre de la función hay que introducirlo entre comillas simples y el intervalo como un vector de dos componentes. Por ejemplo,

```
>> fplot('exp(-x.^2).*cos(6*pi*x)',[-3 3])
```

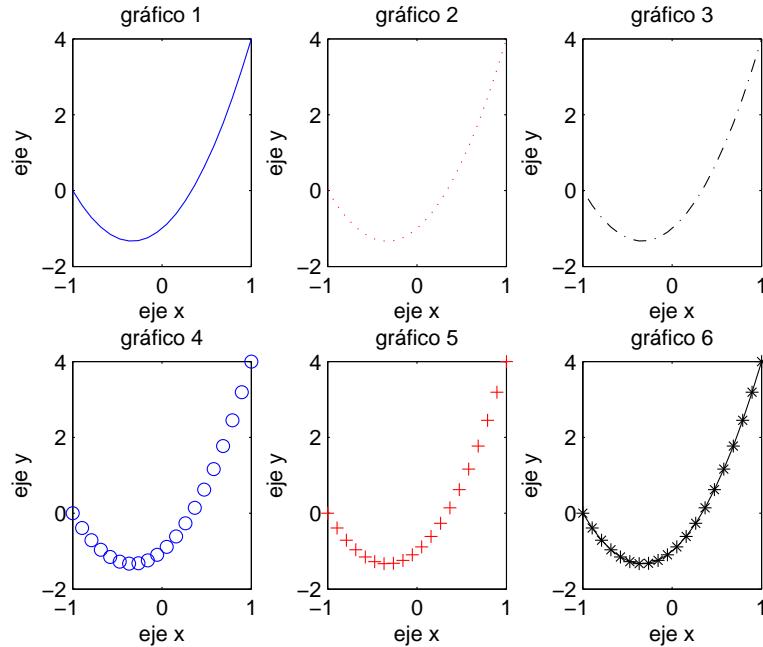


Figura 2.21: Ejemplo de empleo del comando subplot

dibuja la función,

$$f(x) = e^{-x^2} \cos(6\pi x)$$

en el intervalo $[-3, 3]$ (figura 2.22).

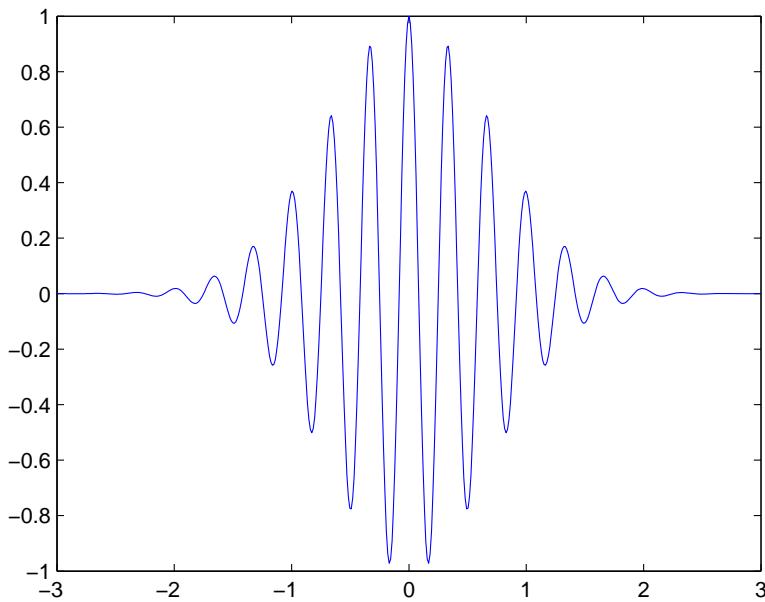
semilogx. El comando `semilogx` representa el eje de las x en escala logarítmica. Es decir, en lugar de representar frente a la variable x , se representa frente a $\log_{10}(x)$. Si dibujamos empleando este tipo de gráfico la función $y = \log_{10}(x)$ deberíamos obtener una líneas rectas de pendiente unidad. La figura 2.23 muestra el resultado, empleando para las *equis* el intervalo $(0, 1)$.

```
>> x=linspace(0,1,100);
>> y=log10(x);
>> semilogx(x,y)
>> grid on
```

Un par de observaciones sobre este ejemplo: En primer lugar las divisiones del eje x aparecen marcadas como potencias de 10. Como estamos representando empleando el logaritmo decimal de la variable x , las divisiones se corresponden con el exponente de la potencia de 10 de cada división, $\log_{10}(10^n) = n$.

En segundo lugar hemos empleado un nuevo comando gráfico; se trata del comando `grid`. Este comando añade una retícula al gráfico de modo que sea más fácil ver los valores que toman las variables en cada punto de la gráfica. `grid on` añade la retícula y `grid off` la retira.

semilogy. Análoga al anterior, simplemente que ahora es el eje y el que se representa en escala logarítmica. En este caso será si representamos la función $y = 10^x$ cuando obtengamos una línea recta,

Figura 2.22: Ejemplo de empleo del comando `fplot`

```
>> x=linspace(0,1,100);
>> y=10.^x;
>> semilogy(x,y)
>> grid on
```

loglog. Análoga a las anteriores, `loglog(x,y)` representa `y` frente a `x` empleando en ambos ejes una escala logarítmica.

polar. Representa funciones en coordenadas polares `polar(theta,r)`. La primera variable es un ángulo en radianes y la segunda el correspondiente radio. La figura 2.25 muestra la espiral,

$$r = 2 \cdot \sqrt{\theta}$$

Para el intervalo angular $[0, 8\pi]$.

```
>> theta=linspace(0,8*pi,100);
>> r=2*theta;
>> polar(theta,r)
>> r=sqrt(theta);
>> polar(theta,r)
```

stem, bar, stairs. En los tres casos, se obtienen representaciones *discretas* de un conjunto de datos. `stem` representa los datos mediante líneas verticales que parten del eje x y llegan hasta el

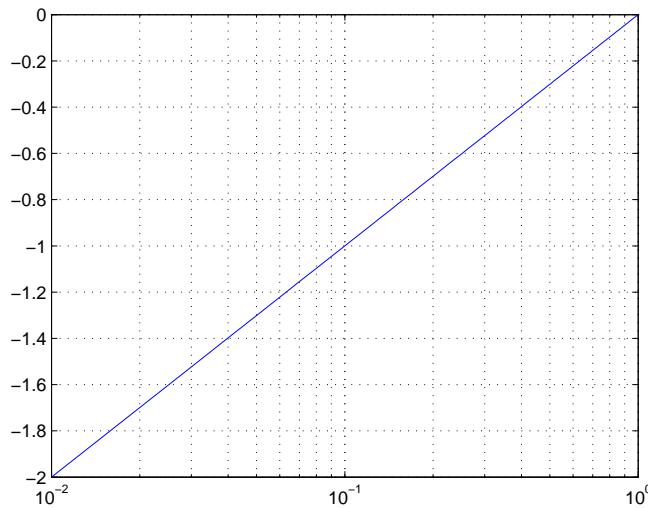


Figura 2.23: Representación de la función $y = \log_{10}(x)$ empleando el comando `semilogx`

valor correspondiente de y . Las líneas van rematadas por un círculo. `bar` Emplea barras solidas verticales y `stairs` realiza una representación en escalera. La figura 2.26 muestra el resultado de dibujar, empleando estos tres tipos de gráficos, los datos correspondientes al número de coches por cada 1000 habitantes en 2007 para cincuenta países distintos (los datos se guardaban en una matriz llamada `auto_50_2007`),

```
>> subplot(1,3,1)
>> stem(auto_50_2007)
>> subplot(1,3,2)
>> bar(auto_50_2007)
>> subplot(1,3,3)
>> stairs(auto_50_2007)
```

hist. Este comando permite dibujar el histograma de una colección de datos. El histograma es una forma de representar cuantas veces se repite un dato, o más exactamente cuantos datos de la colección caen dentro de un intervalo dado. La función `hist(x,n)` admite dos parámetros de entrada, un vector de datos x y un valor entero n que representa el número de intervalos en que se dividirá el rango de valores de x , para obtener el histograma. Si no se introduce esta segunda variable, Matlab por defecto divide el rango de los datos en 10 intervalos. Veamos un ejemplo de uso de `hist`, empleando los datos del ejemplo anterior relativos a número de coches por cada mil habitantes. Representaremos el histograma para un total de 213 países. Para tener una idea, del rango de los datos, calculamos el valor mínimo y máximo de los datos disponibles,

```
>> minimo=min(auto2007)
minimo =
```

0

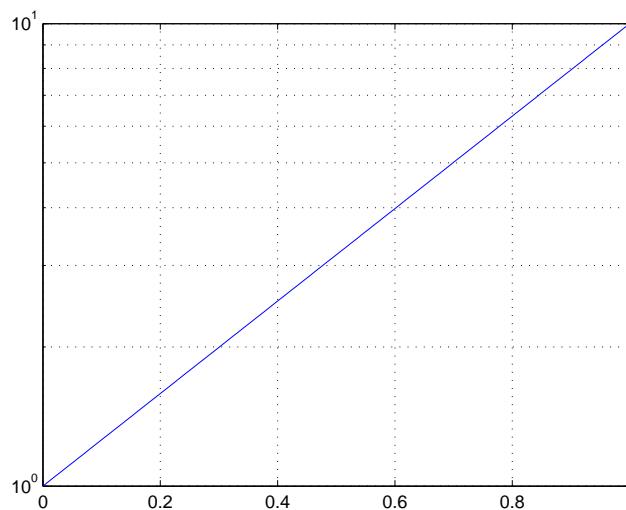


Figura 2.24: Representación de la función $y = 10^x$ empleando el comando `semilogy`

```
>> maximo=max(auto2007)

maximo =
874
```

El rango va de 0 a 879 automóviles por cada mil habitantes. La figura 2.27 muestra los histogramas obtenidos sin indicar el número de intervalos, —por lo que se tomarán 10—, tomando 5 intervalos y tomando 20.

```
>> subplot(1,3,1)
>> hist(auto2007)
>> subplot(1,3,2)
>> hist(auto2007,5)
>> subplot(1,3,3)
>> hist(auto2007,20)
>> subplot(1,3,1)
>> title('10 intervalos')
>> subplot(1,3,2)
>> title('5 intervalos')
>> subplot(1,3,3)
>> title('20 intervalos')
```

La interpretación del los histogramas depende lógicamente del número de intervalos. En el caso de 10 intervalos, estos dividen los datos en grupos de aproximadamente 100 coches. Si observamos el histograma resultante, podemos concluir que hay unos 120 países en los que hay entre 0 y 100 automóviles por cada 1000 habitante, unos 30 países en los que hay entre 100 y 200 automóviles por cada 1000 habitante, etc. Si miramos el siguiente histograma, en el que se han empleado tan solo 5 intervalos, los grupos son ahora de aproximadamente 200 coches. La primera barra de este

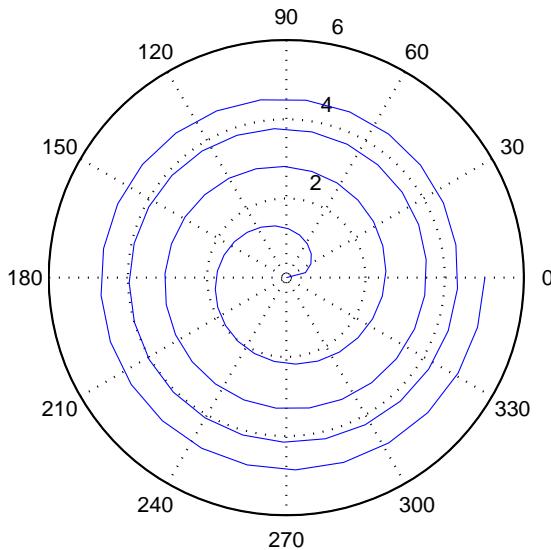


Figura 2.25: Representación de la función $r = \sqrt{\theta}$ empleando el comando `polar`

segundo histograma establece que hay unos 150 países en los que hay entre 0 y 200 coches por cada 1000 habitantes. Resultado que corresponde a la suma de los de dos primeros intervalos del histograma anterior. Para el tercer histograma los intervalos son ahora de 50 automóviles, lo que permite observar más en detalle que en los histogramas anteriores la distribución de vehículos: 110 países tienen menos de 50 automóviles por cada 1000 habitantes.

plotyy. Este comando permite representar dos gráficas en la misma figura de modo que comparten el mismo eje x y cada una tiene su propio eje y. La figura 2.28 muestra el resultado del siguiente ejemplo,

```
>> x1=linspace(0,10,100);
>> x2=linspace(0,12,50);
>> y1=x1.^2;
>> y2=x2.^{(2/3)};
>> plotyy(x1,y1,x2,y2)
>> grid on
```

quiver. Esta función de Matlab permite dibujar vectores en el plano. En realidad está pensada para dibujar campos vectoriales, con lo que hay que manejarla con cierto cuidado. En primera aproximación diremos que `quiver` necesita 5 variables de entrada: la coordenada x del origen del vector, la coordenada y del origen del vector, la componente x del vector y la componente y del vector, por último, un factor de escala al que daremos el valor 0 para que dibuje los vectores a su tamaño real, sin modificar su escala. Por ejemplo si queremos dibujar el vector $\vec{v} = (1, 2)$ situado en el origen de coordenadas,

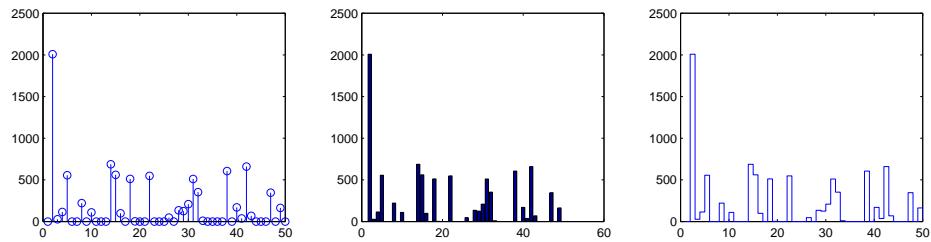


Figura 2.26: Comparación entre los comandos `stem`, `bar` y `stairs` representando la misma colección de datos.

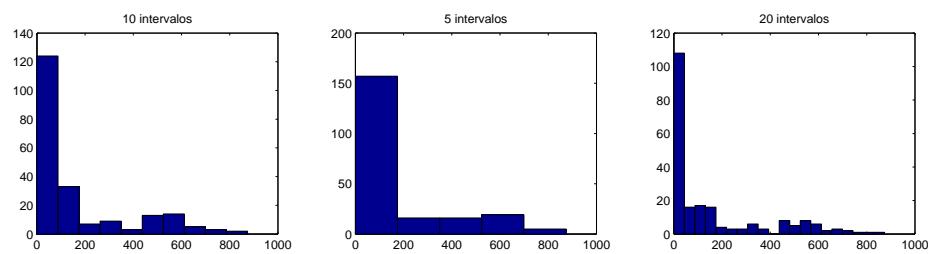


Figura 2.27: histogramas del número de automóviles por cada 1000 habitantes para 213 países

```
quiver(0,0,1,2,0)
```

Si queremos representar el mismo vector pero situado en el punto $(3, -1)$,

```
>>quiver(3,-1,1,2,0)
```

Podemos dibujar un conjunto de vectores con `quiver`, para ello empleamos como variables de entradas vectores, en lugar de escalares; un vector que contenga las posiciones x de los orígenes, un segundo vector que contenga la posiciones y de los orígenes, un tercer vector que contenga las componentes x de los vectores y un cuarto que contenga las componentes y , por último añadiríamos el parámetro de escala 0, que sigue siendo un escalar.

Por tanto, si queremos dibujar a la vez los dos vectores de los ejemplos anteriores,

```
>>quiver([0 3],[0 -1],[1 1],[2 2],0)
```

La figura 2.29 muestra los resultados del ejemplo que acabamos de ver,

errorbar. Permite añadir barras de error a un gráfico de puntos experimentales. Supongamos que tenemos la siguiente tabla (2.8) de resultados de la medida de la velocidad de un móvil frente al tiempo,

La primera columna representa el instante de tiempo en que se tomó la medida, la segunda columna representa el valor medido de la velocidad y la tercera columna la incertidumbre de cada medida de velocidad.

podemos emplear el comando (`errorbar`) para representar la velocidad frente al tiempo, añadiendo una barra de error en cada medida que represente la incertidumbre,

```
>> vel=[0 2.8 4.0 4.3 4.2 4.8 6.3 7.9 8.9 9.0 8.7]
vel =
```

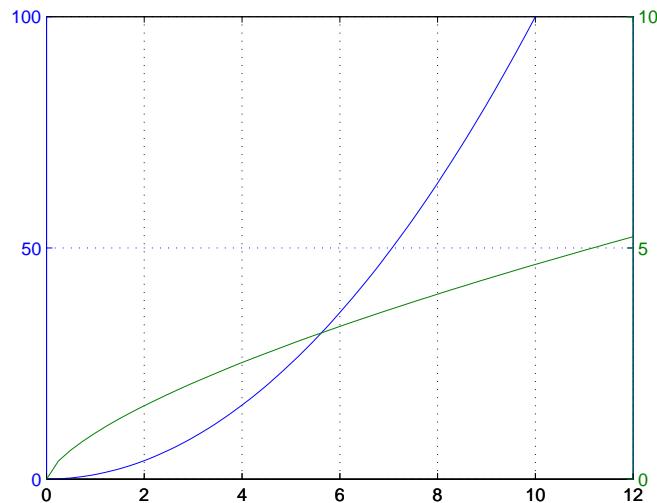


Figura 2.28: Ejemplo de uso de la función plotyy

```

Columns 1 through 7

    0      2.8000      4.0000      4.3000      4.2000      4.8000      6.3000

Columns 8 through 11

    7.9000      8.9000      9.0000      8.7000

>> inc=[0 0.8 0.8 0.9 0.9 1.0 1.0 1.0 1.1 1.1 1.1]
inc =

```

Columns 1 through 7						
0	0.8000	0.8000	0.9000	0.9000	1.0000	1.0000
Columns 8 through 11						
1.0000	1.1000	1.1000	1.1000			

```

>> t=0:10
t =

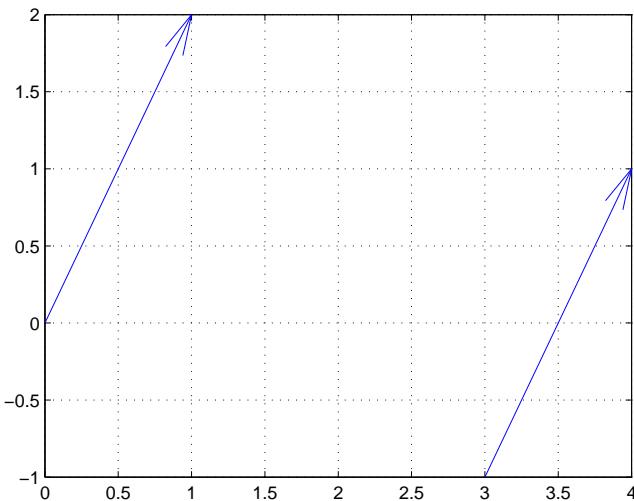
```

t	0	1	2	3	4	5	6	7	8	9	10
vel											

```

>> errorbar(t,vel,inc)
>> xlabel('tiempo s')
>> ylabel('velocidad m/s')

```

Figura 2.29: Ejemplo de uso de la función `quiver`

EL resultado se muestra en la figura 2.30. Es interesante hacer notar que la longitud total de cada barra de error es el doble del valor de la incertidumbre correspondiente al punto. Por ejemplo para $t = 1s, v + \Delta v = 2,8 \pm 0,8m/s$ la longitud de la barra de error es $2 \cdot \Delta v = 1,6m/s$). El comando `errorbar`, puede también emplearse para representar incertidumbres asimétricas. Para ello es preciso suministrar dos vectores uno `U` para dibujar la parte superior de la barra de error y otro `L` para dibujar la parte inferior; `errorbar(x,y,L,U)`.

2.6.3. Gráficos en 3D.

En tres dimensiones es posible representar dos tipos de gráficos: puntos y curvas, análogos a los representados en dos dimensiones y además superficies en el espacio.

plot3. Para dibujar líneas y puntos Matlab emplea los mismos comandos que hemos descrito para dos dimensiones, añadiendo al nombre de comando la terminación 3 para indicar que se trata de un gráfico en tres dimensiones. Así por ejemplo el comando `plot3` nos permite dibujar puntos y curvas en el espacio. El manejo es idéntico al de `plot`, simplemente que ahora es preciso añadir un vector que contenga los datos de la tercera coordenada, `z`.

Por ejemplo, podemos representar la curva,

$$\begin{aligned} y &= \sin(2\pi x) \\ z &= \cos(2\pi x) \end{aligned}$$

Para ello, seleccionamos un intervalo de valores para $x \in (0, 2)$, y calculamos los correspondientes valores de y y z ,

```
>> x=linspace(0,2,100);
>> y=sin(2*pi*x);
>> z=cos(2*pi*x);
```

Cuadro 2.8: Resultados experimentales de la medida de la velocidad de un móvil

tiempo <i>t</i> (s)	velocidad <i>v</i> (m/s)	incertidumbre $\pm\Delta v$ (m/s)
0	0	0
1	2.8	0.8
2	4.0	0.8
3	4.3	0.9
4	4.2	0.9
5	4.8	1.0
6	6.3	1.0
7	7.9	1.0
8	8.9	1.1
9	9.0	1.1
10	8.7	1.1

Podemos ahora representar la gráfica de nuestra función empleando el comando `plot3`,

```
>> plot3(x,y,z)
>> grid on
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
```

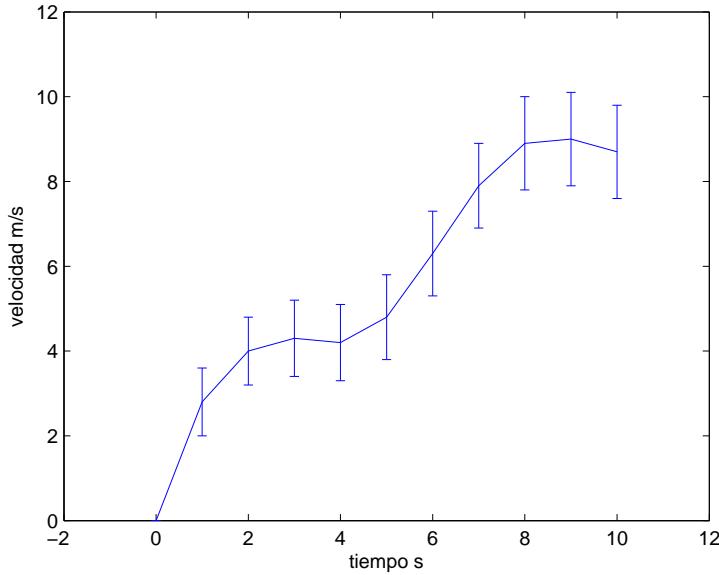
Hemos añadido los comandos `grid on` para obtener una trama en 3D que permita ver mejor el resultado. La figura 2.31(a) muestra la figura de Matlab obtenida, donde se ha señalado además un botón que permite rotar la figura, cambiando la vista. Para ellos, una vez pulsado el botón, basta con arrastrar el ratón sobre la figura manteniendo pulsado el botón izquierdo. La figura 2.31(b), muestra la misma gráfica en 3D, pero ahora vista de frente (como si nos situáramos en el eje x). La figura 2.31(c), nos muestra la grafica vista desde arriba (desde el eje z), por último, la figura 2.31(d) muestra una vista lateral de la gráfica (tomada desde el eje y).

Es posible rotar la figura para obtener una vista concreta mediante el comando `view(Az, El)`. Este comando admite dos parámetros; `Az`, representa el azimuth o ángulo de rotación horizontal, `El`, representa el ángulo de elevación. Ambos ángulos se introducen en grados. Así, por ejemplo las vistas representadas en las figuras anteriores, se pueden obtener como,

```
>> view(90,0)
>> view(0,90)
>> view(0,0)
```

bar3, stem3, hist3, quiver3. Existen versiones 3D de los comandos `bar`, `stem`, `hist` y `quiver`. Su funcionamiento es similar —aunque no siempre igual— al de la versión 2D que vimos en la sección anterior. En algunos casos necesitan tres variables de entrada, correspondientes a las componentes (*x,y,z*) de los datos que se quiere representar, y en otros necesitan que los datos de entrada se le suministren en forma de matriz. Para conocer en detalle su funcionamiento, lo ideal es acudir a la ayuda de Matlab.

Superficies. Para trazar superficies en el espacio, Matlab necesita en primer lugar que se defina una retícula en el plano (*x, y*) que sirve de base sobre la que calcular los puntos *z* sobre los que se alzará la superficie.

Figura 2.30: Datos de la tabla 2.8 representados empleando el comando `errorbar`

Para definir dicha retícula Matlab emplea dos matrices. una de ellas X_m contiene las coordenadas x de los nodos de la retícula y la otra Y_m las coordenadas y . Los elementos que ocupan la misma posición en ambas matrices, representan —juntos— un punto en el plano. Matlab emplea dichas matrices como matrices de *adyacencia*. Cada nodo, $(x_m(i, j), y_m(i, j))$, aparecerá en la gráfica conectado por una arista a cada uno de sus cuatro puntos vecinos, $(x_m(i - 1, j), y_m(i - 1, j))$, $(x_m(i, j - 1), y_m(i, j - 1))$, $(x_m(i + 1, j), y_m(i + 1, j))$, $(x_m(i, j + 1), y_m(i, j + 1))$. Supongamos que empleamos las siguientes matrices, X_m y Y_m para definir una retícula sobre la que dibujar una superficie,

$$X_m = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{pmatrix}, Y_m = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix} \xrightarrow[\text{nodos}]{\text{posiciones}} \begin{matrix} (0, 0) & - & (1, 0) & - & (2, 0) & - & (3, 0) \\ | & & | & & | & & | \\ (0, 1) & - & (1, 1) & - & (2, 1) & - & (3, 1) \\ | & & | & & | & & | \\ (0, 2) & - & (1, 2) & - & (2, 2) & - & (3, 2) \\ | & & | & & | & & | \\ (0, 3) & - & (1, 3) & - & (2, 3) & - & (3, 3) \end{matrix}$$

La retícula definida por Matlab, a partir de dichas matrices tendría el aspecto que se muestra en la figura 2.32.

Si nos fijamos en los ejes de la figura es fácil obtener las coordenadas de los nodos y comprobar como, están unidos entre sí por aristas los que ocupan posiciones adyacentes en las matrices X_m e Y_m .

Para construir una superficie sobre la retícula, lo único que hace falta es definir una altura (z), para cada punto de la retícula. Para ello, Matlab emplea una matriz, del mismo tamaño que X_m y Y_m . Así por ejemplo, si definimos,

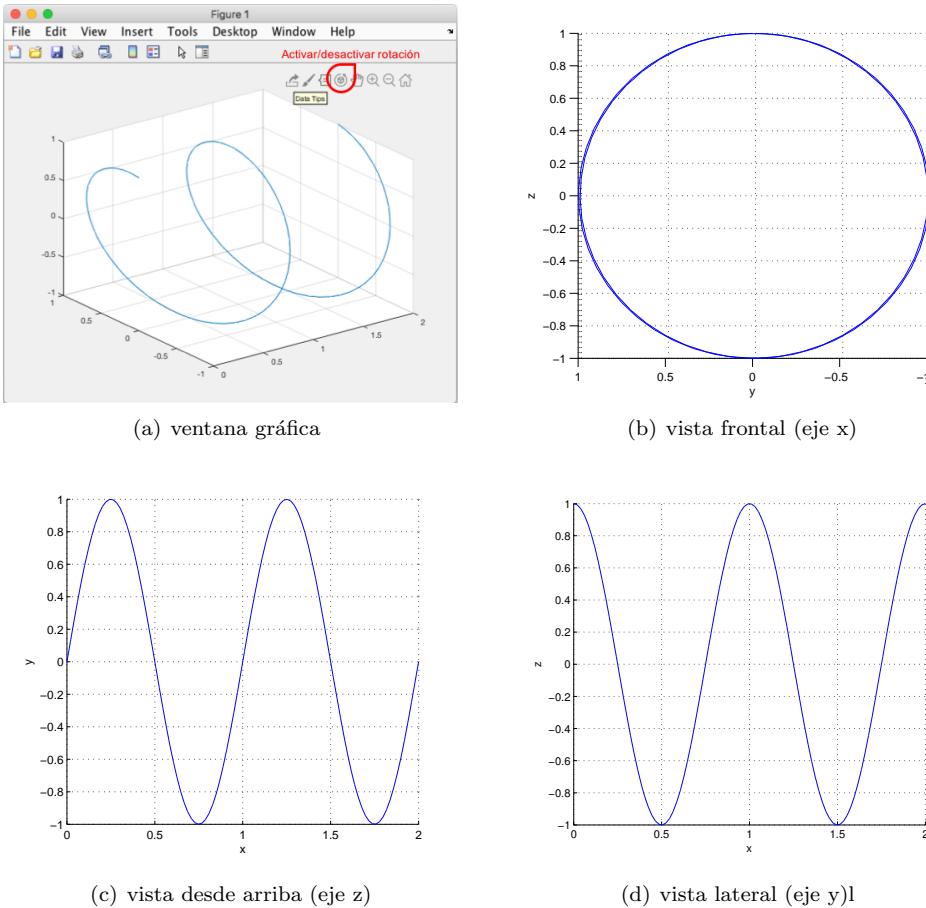


Figura 2.31: Gráfico en 3D y rotaciones.

$$Z_m = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 3 & 4 \\ 0 & 0 & 0 \end{pmatrix}$$

Cada elemento de la matriz Z_m representa la altura del nodo correspondiente a las posiciones marcadas por las matrices X_m e Y_m , tal y como se muestra en la figura 2.33.

La estructura de las matrices X_m e Y_m de los ejemplo anteriores, es la típica de las matrices de adyacencia de una retícula cuadrada; la matriz X_m tiene las filas repetidas y la matriz Y_m tiene repetidas las columnas. En el ejemplo las matrices son cuadradas y definen una retícula de 4×4 nodos. En general, podemos definir una retícula rectangular de $m \times n$ nodos. En este caso las matrices empleadas para definir la retícula tendrían dimensión $m \times n$.

Para dibujar en Matlab superficies podemos en primer lugar definir la retícula a partir de dos vectores de coordenadas empleando el comando `meshgrid`. En el ejemplo que acabamos de ver, hemos empleado una retícula que cubre el intervalo, $x \in [0, 3]$ e $y \in [0, 3]$. para definirlo creamos los vectores,

```
>> x=0:3
```

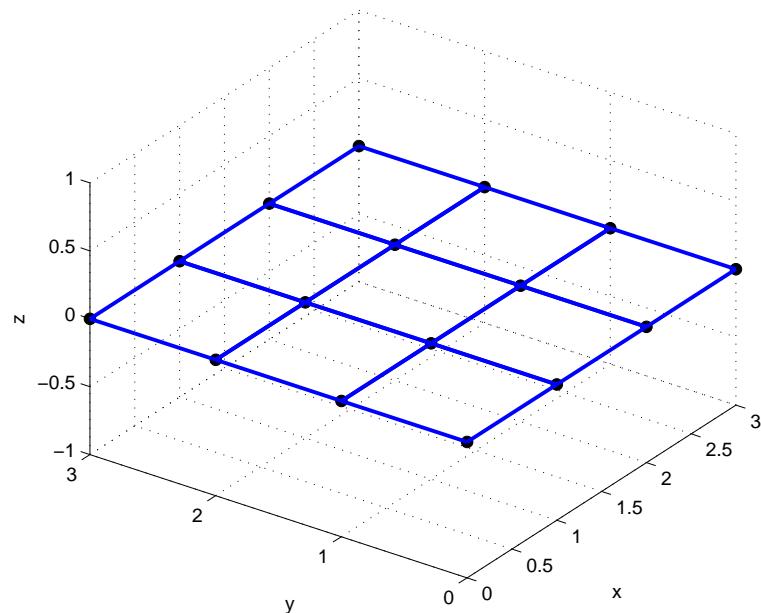


Figura 2.32: Retícula para representar superficies. Los puntos negros son los nodos definidos por las matrices X_m e Y_m .

```
x =
0   1   2   3
>> y=0:3
y =
0   1   2   3
```

A continuación empleamos el comando `meshgrid` para construir las dos matrices de adyacencia. Matlab se encargará de repetir las filas y columnas necesarias,

```
>> [Xm,Ym]=meshgrid(x,y)
Xm =
```

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

```
Ym =
0   0   0   0
```

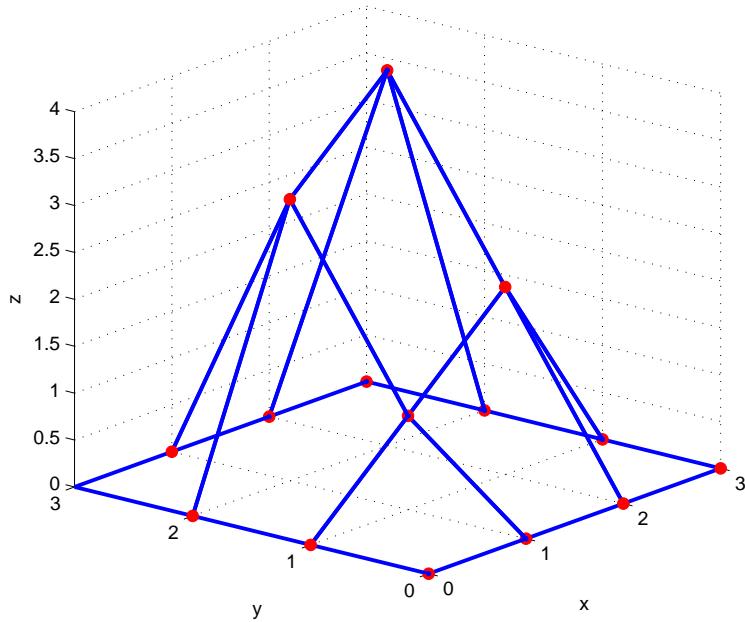


Figura 2.33: Superficie elemental obtenida elevando los cuatro puntos centrales de la figura 2.32.

1	1	1	1
2	2	2	2
3	3	3	3

Una vez construidas las matrices de adyacencia, solo necesitamos una matriz de valores para Z_m . Si definimos por ejemplo,

```
>> Zm=zeros(size(Xm))
```

$Zm =$

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Podríamos representar la retícula plana de la figura 2.32, empleando por ejemplo el comando `mesh(Xm, Ym, Zm)`.

mesh y surf. Una vez que hemos visto como construir una retícula rectangular sobre la que construir una superficie, veamos como dibujarla con un ejemplo. Supongamos que queremos dibujar la superficie,

$$z = x^3 + y^2$$

En la región del plano, $x \in [-1,5, 1,5]$, $y \in [-2, 2]$.

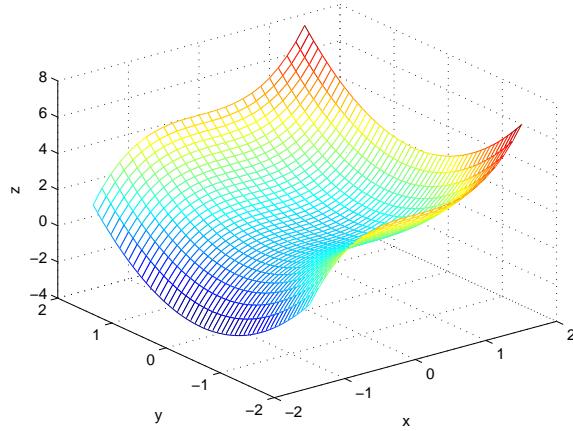
Igual que en el ejemplo inicial, lo primero que debemos hacer es construirnos una matrices de adyacencia que definan una retícula en la región de interés,

```
>> x=linspace(-1.5,1.5,25);
>> y=linspace(-2,2,50);
>> [Xm,Ym]=meshgrid(x,y);
```

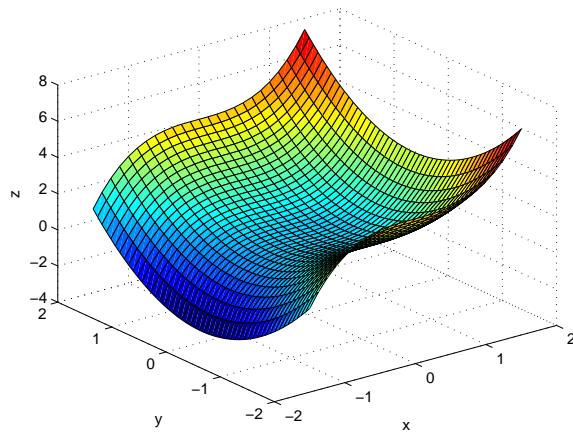
Es interesante notar que la región de interés no es cuadrada y que las matrices de adyacencia tampoco los son (50×25). Además los puntos no están espaciados igual en los dos ejes.

A continuación obtenemos la matriz de coordenadas z, aplicando la función a los puntos de la retícula,

```
>> Zm=Xm.^3+Ym.^2;
```



(a) Función $z = x^3 + y^2$ representada con **mesh**



(b) Función $z = x^3 + y^2$ representada con **surf**

Figura 2.34: Comparación entre **mesh** y **surf**

Para representar la superficie podemos emplear el comando `mesh`.

```
>> mesh(Xm,Ym,Zm)
```

Este comando admite como variables de entrada las dos matrices de adyacencia empleadas para definir la retícula y la matriz Z_m que contiene los valores calculados para la variable z , en todos los puntos de la retícula. `mesh` traza la superficie en forma reticular, es decir, nos dibuja una malla en el espacio. El color de la malla depende del valor que toma la coordenada z . Podemos también representar la superficie haciendo uso del comando `surf`, empleando las mismas variables de entrada que en el caso de `mesh`.

```
>> surf(Xm,Ym,Zm)
```

La diferencia está en que ahora la superficie muestra las caras definidas por la malla de colores, según el valor que toma la variable z . La figura 2.34(a) muestra el resultado de nuestro ejemplo empleando `mesh` y la figura 2.34(b) muestra el resultado empleando `surf`.

Para figuras que presentan simetría radial, puede ser más conveniente, definir las retículas en coordenadas polares. así por ejemplo,

```
>> r=0:2/20:2;
>> theta=0:2*pi/36:2*pi;
>> [rm,them]=meshgrid(r,theta);
```

Hemos cosntruido una retícula en las variables r y θ , si ahora definimos las matrices de adyacencia como las proyecciones sobre los ejes x e y ,

```
>> xm=rm.*cos(them);
>> ym=rm.*sin(them);
```

Obtenemos una retícula con simetría radial, centrada en el origen de coordenadas. Como la que se muestra en la figura, 2.35.

La retícula resulta muy adecuada para dibujar por ejemplo un cono (figura 2.36),

```
>> zm=2-sqrt(xm.^2+ym.^2);
>> mesh(xm,ym,zm)
```

A continuación, se incluye el código de un script con varios ejemplos más de diseño de retículas circulares y gráficos de superficies en 3D. Los resultados se muestran en la figura 2.37

```
%este script (varios_g3d.m) incluye el diseño de varias retículas
%circulares para trazar gráficos de superficies en el espacio.
```

```
%retícula circular para un cono
r=0:2/20:2;
theta=0:2*pi/36:2*pi;
[rm,them]=meshgrid(r,theta);

%calculamos las componentes en x e y partir de rm y thm,
xm=rm.*cos(them);
ym=rm.*sin(them);

%Definimos la componente z, a partir de la ecuación de un cono,
zm=2-sqrt(xm.^2+ym.^2);
```

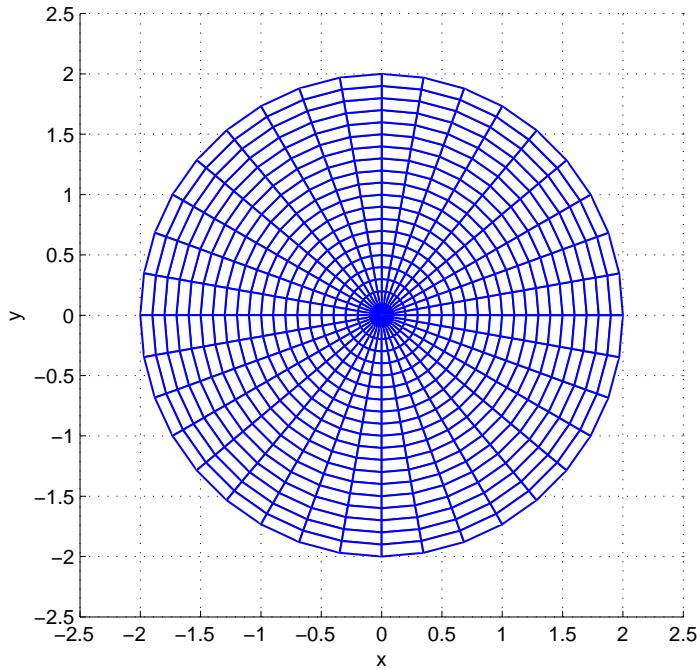


Figura 2.35: retícula con simetría circular

```
%lo dibujamos empleando el comando mesh
mesh(xm,ym,zm)

%Para dibujar una esfera, podríamos emplear la misma retícula, pero sale
%más proporcionada, si de emplear intervalos de r equiespaciados, los hacemos
%proporcionales al ángulo de elevación (0,pi/2). Así que recalculamos r.

r=2*cos(0:pi/2/18:pi/2); %tomamos 16 intervalos entre r=0 y r=2
%para theta usamos el mismo de antes, no hace falta volver a calcularlo.

%calculamos los puntos de la retícula en polares,
[rm,them]=meshgrid(r,theta);

%calculamos las componentes x e y a partir de rm y them
xm=rm.*cos(theim);
ym=rm.*sin(theim);

%definimos la componente z, como es la ecuación de una esfera, debemos
%calcularla en dos partes,
```

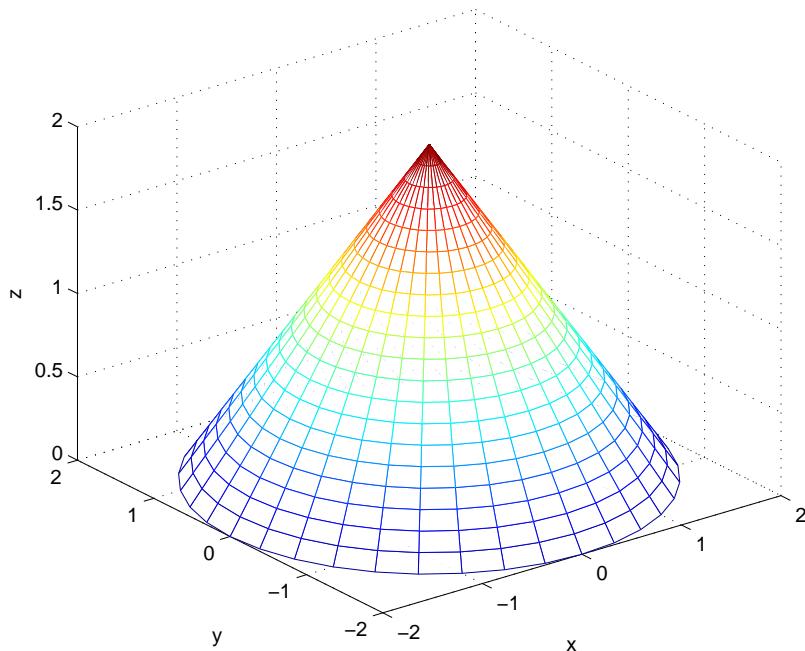


Figura 2.36: Cono representado sobre una retícula circular

```

zm_mas=sqrt(4-(xm.^2+ym.^2));
zm_menos=-sqrt(4-(xm.^2+ym.^2));

%vamos a emplear la misma ventana gráfica en la que hemos dibujado el
%cono, así que pedimos a Matlab que retenga lo ya dibujado,
hold on

%para que no salga un dibujo encima del otro, dibujamos la esfera
%desplazando su origen al punto (3,3,0)
%emplearemos ahora el comando surf para representarla

%primero representamos la mitad superior
surf(xm+3,ym+3,zm_mas)

%y despues la parte inferior
surf(xm+3,ym+3,zm_menos)

%por último vamos a obtener la gráfica de un cilindro vertical, Para ello,
%volvemos a alterar los valore de r. Costruimos un vector que repita
%siempre los mismos valores y le damos tanto elementos como divisiones
%'verticales queramos que tenga el cilindro, por ejemplo 20
r=2*ones(1,20);

```

```
%volvemos a crear la reticula
[rm,them]=meshgrid(r,theta);

%calculamos las componentes x e y a partir de rm y thm
xm=rm.*cos(theta);
ym=rm.*sin(theta);

% Para calcular los valores de z, debemos dividir la altura total que
% queremos dar al cilindro entre el número de divisiones en r que tiene la
% retícula original por ejemplo si la altura fuera 2
z=2/(length(r)-1)*(0:length(r)-1);
% Tenemos una columna de la matriz z, pero necesitamos repetirlas para
% cada valor de theta,
zm=ones(size(xm))*diag(z);

%podemos ahora dibujar en cilindro. Como hemos hecho con la esfera, lo
%vamos a desplazar a una posición en la que no se sobreponga a otra
%figura, por ejemplo al punto (+3 -3 -1).

%lo vamos a representar usando de nuevo en comando mesh

mesh(xm+3,ym-3,zm-1)
```

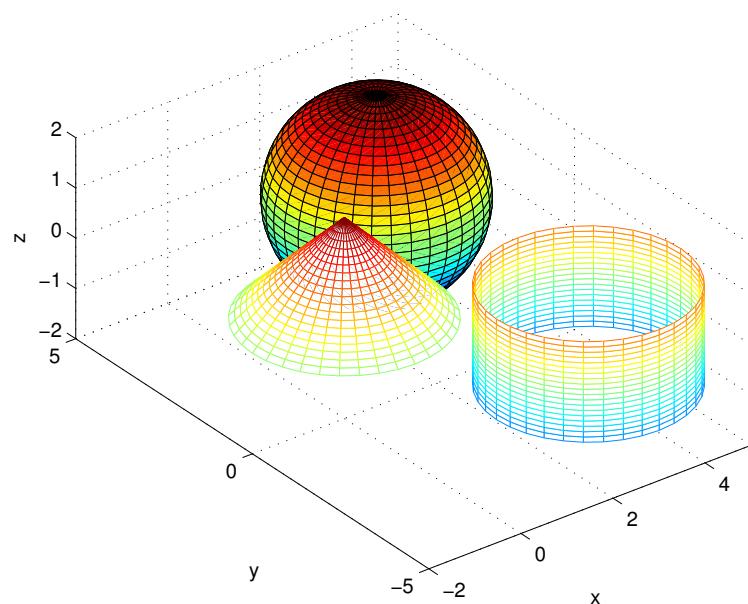


Figura 2.37: retícula con simetría circular

contour, contour3, meshc y surf. Este comando permite obtener y dibujar las curvas de nivel de una superficie. Su uso es idéntico al de los comandos anteriores. Es decir, también necesitan que se defina una retícula en el plano (x, y) , y se calculen los valores que tomará la variable z sobre los puntos de la retícula.

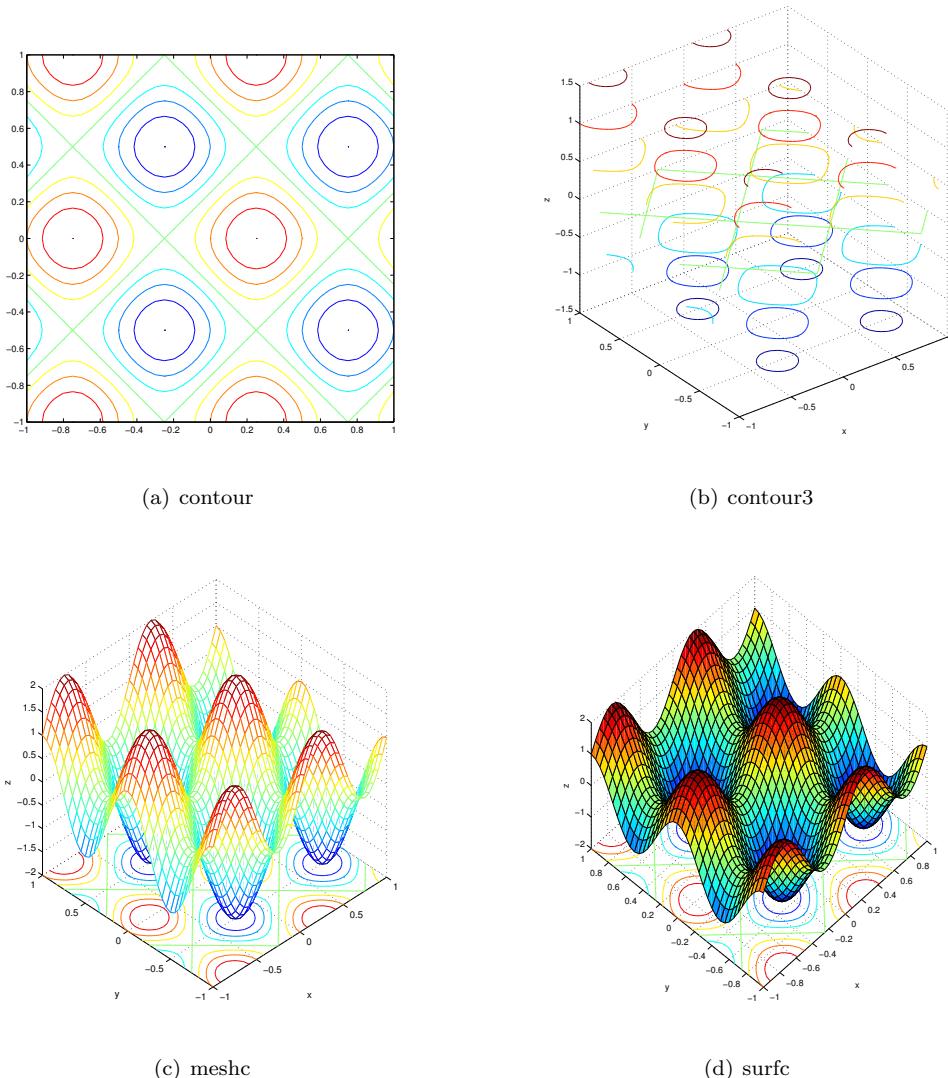


Figura 2.38: Comparación entre los resultados de `contour`, `contour3`, `meshc` y `surf`, para la obtención de las curvas de nivel de una superficie.

Veamos su funcionamiento con un último ejemplo. Obtenemos una retícula cuadrada, calculamos sobre ella los puntos de la superficie,

$$z = \sin(2\pi x) + \cos(2\pi y)$$

```
>> x=[-1:0.05:1];
```

```
>> y=x;
>> [xm,ym]=meshgrid(x,y);
>> zm=sin(2*pi*xm)+cos(2*pi*ym);
>> contour(xm,ym,zm)
>> contour3(xm,ym,zm)
>> meshc(xm,ym,zm)
>> surf3(xm,ym,zm)
```

La figura 2.38(a) muestra los resultados de aplicar el comando `contour`. La gráfica representa las curvas de nivel de la superficie dibujadas sobre el plano (x, y) . El comando `contour3` (figura 2.38(b)) representa de nuevo las curvas de nivel, pero sitúa cada una a su correspondiente altura z . Por último `meshc` y `surf3` representan la superficie y añaden en el plano (x, y) la representación de las curvas de nivel correspondiente a la superficie.

Para terminar la sección dedicada a los gráficos, vamos a combinar el comando `mesh` con el comando `plot3` para dibujar una curva sobre una superficie. Tomaremos como ejemplo la superficie,

$$z = \frac{\sin((\pi x)^2 + (\pi y^2))}{(\pi x)^2 + (\pi y^2)}$$

Sobre la que trazamos la curva,

$$y = \sin(\pi x),$$

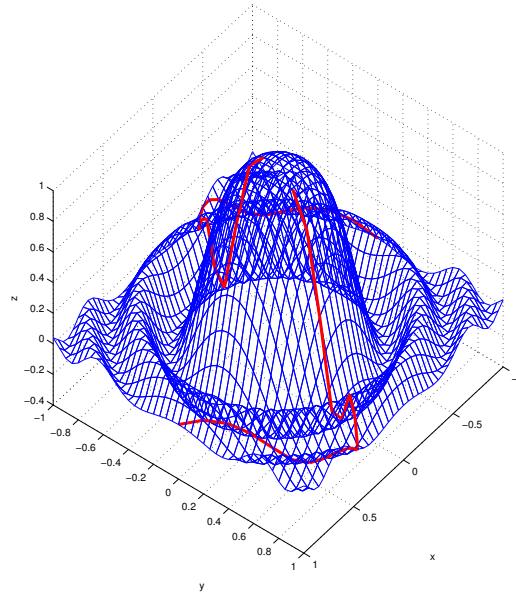


Figura 2.39: Curva trazada sobre una superficie

Es decir, obtenemos el valor de los puntos z de la superficie para los pares de puntos $[x, y = \sin(\pi x)]$, que definen la curva trazada sobre la superficie z ,

```
>> x=[-1:0.05:1];
>> y=x;
>> [xm,ym]=meshgrid(x,y);
>> zm=sin((pi*xm).^2+(pi*ym).^2)./((pi*xm).^2+(pi*ym).^2);
>> mesh(xm,ym,zm)
>> y=sin(pi*x);
>> z=sin((pi*x).^2+(pi*y).^2)./((pi*x).^2+(pi*y).^2);
>> hold on
>> plot3(x,y,z)
```

Es importante insistir en que a lo largo de esta sección nos hemos limitado a introducir algunas de las posibilidades gráficas de Matlab. Para obtener una visión completa de las mismas es imprescindible leer con detenimiento la ayuda de Matlab.

Capítulo 3

Aritmética del Computador y Fuentes de error

En el capítulo 1, introdujimos la representación binaria de números así como la conversión de binario a decimal y decimal a binario. A lo largo de este capítulo vamos a profundizar más en el modo en que el ordenador representa y opera con los números así como en una de sus consecuencias inmediatas: la imprecisión de los resultados.

3.1. Representación binaria y decimal

Los números reales pueden representarse empleando para ello una recta que se extiende entre $-\infty$ y $+\infty$. La mayoría de ellos no admiten una representación numérica exacta, puesto que poseen infinitos decimales.

Los números enteros, $1, -1, 2, -2, 3, -3, \dots$ admiten una representación numérica exacta. En cualquier caso, rara vez manejamos números enteros con una cantidad grande de dígitos, habitualmente, los aproximamos, expresándolos en notación científica, multiplicándolos por una potencia de 10. Tomemos un ejemplo de la química: la cantidad de átomos o moléculas que constituyen un mol de una sustancia se expresa por el número de Avogadro, $N_A = 6,02214179 \times 10^{23}$, dicho numero, —que debería ser un entero— se expresa empleando tan solo 9 de sus 23 cifras significativas (a veces tan solo se dan las tres primeras).

Cuando truncamos un número, es decir, despreciamos todos sus dígitos hacia la derecha a partir de uno dado, estamos aproximando dicho número por un número racional, es decir, por un número que puede expresarse como el cociente entre dos números enteros. $1/2, 3/4, \dots$

Algunos números racionales se reducen a su vez a números enteros, $6/3$, otros dan lugar a números cuya representación exige un número finito de dígitos:

$$11/2 = (5,5)_{10} = 5 \times 10^0 + 5 \times 10^{-1}$$

Si representamos el mismo número en base 2 obtenemos,

$$11/2 = (101,1)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}$$

Sin embargo, el que un número racional admita una representación finita, depende de la base en que se representa. Por ejemplo: $1/10 = (0,1)_{10}$ no admite representación finita en base 2.

$$1/10 = (0,0001100110011\dots)_2 = 1 \times 2^{-4} + 1 \times 2^{-5} + 0 \times 2^{-6} + 0 \times 2^{-7} + 1 \times 2^{-8} + \dots$$

En este último caso, se trata de una representación que, aunque no termina, es repetitiva o periódica; tras el primer cero decimal se repite indefinidamente la secuencia: 0011. Los números racionales admiten siempre una representación finita ó infinita periódica.

El número racional $1/3 = (0,333\cdots)_10 = (0,010101\cdots)_2$ solo admite representación infinita periódica tanto en base 10 como en base 2. Sin embargo, admite representación finita si lo representamos en base 3: $1/3 = (0,1)_3 = 0 \times 3^0 + 1 \times 3^{-1}$.

Habitualmente, en la vida ordinaria, representamos los números en base 10, sin embargo, como hemos visto en el capítulo 1 el ordenador emplea una representación binaria. Como acabamos de ver, una primera consecuencia de esta diferencia, es que al pasar números de una representación a otra estemos alterando la precisión con la que manejamos dichos números. Esta diferencia de representación supone ya una primera fuente de errores, que es preciso tener en cuenta cuando lo que se pretende es hacer cálculo científico con un ordenador. Como iremos viendo a lo largo de este capítulo, no será la única.

3.2. Representación de números en el ordenador

Enteros positivos La forma mas fácil de representar números sería empleando directamente los registros de memoria para guardar números en binario. Si suponemos un ordenador que tenga registros de 16 bits. Podríamos guardar en ellos números comprendidos entre el 0 y el $2^{16} - 1 = 65535$. Este último se representaría con un 1 en cada una de las posiciones del registro, en total 16 unos. Se trata del entero más grande que podríamos representar con un registro de 16 bits:

posición	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
valor	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Hasta ahora, nuestro sistema de representación solo admite números enteros. El menor número que podemos representar es el cero, y el mayor —dependerá del tamaño n de los registros del ordenador—, $2^n - 1$. Es evidente que este sistema de representación resulta insuficiente.

Enteros positivos y negativos Una primera mejora consistiría en ampliar el sistema de representación para que admita también números negativos. Una posibilidad sería reservar uno de los bits del array para representar el signo , y usar los restantes para representar el número. En nuestro ejemplo de un registro de 16 bits, el número más grande sería ahora $2^{15} - 1$ y el más pequeño sería $-2^{15} + 1$. El cero tendría una doble representación, una de ellas con signo más y la otra con signo menos.

Una representación alternativa, cuyo uso está muy extendido en las máquinas, es la representación conocida con el nombre de *Complemento a 2* . Supongamos un registro de n bits:

- Un entero x no negativo se almacena empleando directamente su representación binaria. el intervalo de enteros no negativos representables es: $0 \leq x \leq 2^{(n-1)} - 1$.
- Un entero negativo $-x$, donde $1 \leq x \leq 2^{(n-1)}$, se almacena como la representación binaria del entero positivo: $2^n - x$
- El número así construido, recibe el nombre de complemento a 2 de x.

En general, dado un número entero $N < 2^n$ su complemento a dos en una representación binaria de n dígitos se define como:

$$C_2^N = 2^n - N$$

Veamos un ejemplo de esta representación, para el caso de un registro de 4 bits:

La representación en complemento a 2 tiene las siguientes características:

Decimal	(4 bits)	$C_2 = 2^n - x$	n. representado
15	1111	$(16 - 1) = 15$	-1
14	1110	$(16 - 2) = 14$	-2
13	1101	$(16 - 3) = 13$	-3
12	1100	$(16 - 4) = 12$	-4
11	1011	$(16 - 5) = 11$	-5
10	1010	$(16 - 6) = 10$	-6
9	1001	$(16 - 7) = 9$	-7
8	1000	$(16 - 8) = 8$	-8
<hr/>			
7	0111		7
6	0110		6
5	0101		5
4	0100		4
3	0011		3
2	0010		2
1	0001		1
0	0000		0

- La representación de los números positivos coincide con su valor binario
- La representación del número 0 es única.
- El rango de valores representables en una representación de N bits es $-2^{N-1} \leq x \leq 2^{N-1} - 1$.
- Los números negativos se obtienen como el complemento a dos, cambiado de signo, del valor binario que los representa.

Una forma práctica de obtener el complemento a dos de un número binario es copiar el número original de derecha a izquierda hasta que aparezca el primer 1. Luego de copiar el primer 1, se complementan el resto de los bits del número (si es 0 se pone un 1 y si es 1 se pone un 0). Por ejemplo el complemento a dos del número 2, (0010 en una representación de 4 bits) sería 1110,

$$\begin{array}{r}
 0010 \xrightarrow{\text{copia bit derecha}} \dots 0 \\
 0010 \xrightarrow{\text{copia primer bit}=1} \dots 10 \\
 0010 \xrightarrow{\text{complemento bit } 0} \cdot 110 \\
 0010 \xrightarrow{\text{complemento bit } 0} 1110
 \end{array}$$

Una propiedad importante de la representación de complemento a dos, es que la diferencia de dos números puede realizarse directamente sumando al primero el complemento a dos del segundo. Para verlo podemos usar los dos números del ejemplo anterior: 0010 es la representación binaria del número 2, si empleamos un registro de cuatro bits. Su complemento a dos, 1110 representa al número -2. Si los sumamos ¹:

El resultado de la suma nos da un número cuyos primeros cuatro bits son 0. El bit distinto de cero, no puede ser almacenado en un registro de cuatro bits, se dice que el resultado de la operación *desborda* el tamaño del registro. Ese bit que no puede almacenarse se descarta al realizar

¹Sumar en binario es como sumar en decimal. Se procede de bit a bit, de derecha a izquierda y cuando se suman 1 y 1, el resultado es 10 (base 2), de modo que el bit resultante se coloca en 0 y se lleva el 1 hacia el siguiente bit de la izquierda.

	0	0	1	0
	1	1	1	0
1	0	0	0	0
	0	0	0	0

la operación de suma, con lo que el resultado sería cero, como corresponde a la suma de $2 + (-2)$. Esta es la motivación fundamental para emplear una representación en complemento a dos: No hace falta emplear circuitos especiales para calcular la diferencia entre dos números, ya que ésta se representa como la suma del minuendo con el complemento a dos del sustraendo.

3.2.1. Números no enteros: Representación en punto fijo y en punto flotante

La representación de números no enteros se emplea para representar de modo aproximado números reales. Como hemos dicho antes, los números racionales periódicos y los números irracionales no pueden representarse de forma exacta mediante un número finito de decimales. Por esta razón, su representación es siempre aproximada. Para los números racionales no periódicos, la representación será exacta o aproximada dependiendo del número que se trate, el tamaño de los registros del ordenador y el sistema empleado para la representación. Los dos sistemas de representación más conocidos son la representación en punto fijo y, de especial interés para el cálculo científico, la representación en punto flotante.

Representación en punto fijo. En la representación en punto fijo, el registro empleado para almacenar un número se divide en tres partes:

- 1 bit para almacenar el signo del número.
- Un campo de bits de tamaño fijo para representar la parte entera del número.
- Un campo para almacenar la parte decimal del número

Por ejemplo, si tenemos un registro de 16 bits podemos reservar 1 bit para el signo, 7 para la parte entera y 8 para la parte decimal. La representación en punto fijo del número binario $-10111,0011101101$ sería:

s	p. entera							p. decimal							
1	0	0	1	0	1	1	1	0	0	1	1	1	0	1	1

Es interesante notar cómo para representar la parte entera, nos sobran dos bits en el registro, ya que la parte entera solo tiene cinco cifras y tenemos 7 bits para representarla. Sin embargo, la parte decimal tiene 10 cifras; como solo tenemos 8 bits para representar la parte decimal, la representación trunca el número eliminando los dos últimos decimales.

Si asociamos cada bit con una potencia de dos, en orden creciente de derecha a izquierda, podemos convertir directamente el número representado de binario a decimal,

s	p. entera							p. decimal							
1	0	0	1	0	1	1	1	0	0	1	1	1	0	1	1
-	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

Por tanto el número representado sería,

$$-(0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot -8) = -23,23046875$$

De modo análogo podemos calcular cual sería el número más grande representable,

s	p. entera							p. decimal							
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} = 127,99609375$$

El número menor representable,

s	p. entera							p. decimal							
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
-	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$-(1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8}) = -127,99609375$$

El número entero más grande,

s	p. entera							p. decimal							
	0	1	1	1	1	1	1	0	0	0	0	0	0	0	
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + \\ + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 0 \cdot 2^{-8} = 127$$

El número decimal positivo más próximo a 1,

s	p. entera							p. decimal							
	0	0	0	0	0	0	0	1	1	1	1	1	1	1	
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + \\ + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0,99609375$$

O el número decimal positivo más próximo a 0,

s	p. entera							p. decimal							
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

$$0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + \\ + 0 \cdot 2^{-5} + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 1 \cdot 2^{-8} = 0,00390625$$

La representación en punto fijo fue la más usada en los primeros computadores. Sin embargo, es posible con el mismo tamaño de registro representar un espectro mucho más amplio de números empleando la representación en punto flotante.

Representación en punto flotante La representación de números en el ordenador en formato de punto flotante, es similar a la forma de representar los números en la llamada notación científica. En notación científica los números se representan como el producto de una cantidad por una potencia de diez: $234,25 \equiv 2,345 \times 10^2$, $0,000234 \equiv 2,34 \times 10^{-5}$, $-56,238 \equiv -5,6238 \times 10^1$. La idea es desplazar el punto decimal hasta que solo quede una cifra significativa en la parte entera del número y multiplicar el resultado por 10 elevado a la potencia adecuada para recuperar el número original.

Muchas calculadoras y programas de cálculo científico presentan los resultados por pantalla en formato científico. Habitualmente lo hacen con la siguiente notación,

$$-5,3572 \times 10^{-3} \xrightarrow{\text{calculadora}} -5,3572e-03$$

Es decir, en primer lugar se representa el signo del número si es negativo (si es positivo lo habitual es omitirlo). A continuación la parte significativa, 5,3572, que recibe el nombre de mantisa . Y por último se representa el valor del exponente de la potencia de 10, -3 , precedido por la letra e, —e de exponente—. En notación científica se asume que el exponente corresponde siempre a una potencia de diez. Es evidente que tenemos el número perfectamente descrito sin más que indicar los valores de su signo, mantisa y exponente,

numero	r. científica	r. calculadora	signo	mantisa	exponente
-327,43	$-3,2743 \times 10^2$	$-3,2743e2$	-	3,2743	2

La representación binaria en punto flotante sigue exactamente la misma representación que acabamos de describir para la notación científica. La única diferencia es que en lugar de trabajar con potencias de diez, se trabaja con potencias de dos, que son las que corresponden a la representación de números en binario. Así por ejemplo, el número binario $-1101,00101$ se representaría como $-1,10100101 \times 2^3$, y el número $0,001011101$ se representaría como $1,011101 \times 2^{-3}$.

El término punto flotante viene del hecho de que el punto decimal de la mantisa, no separa la parte entera del número de su parte decimal. Para poder separar la parte entera de la parte decimal del número es preciso emplear el valor del exponente. Así, para exponente 0, el punto decimal de la mantisa estaría en el sitio que corresponde al número representado, para exponente 1 habría que desplazarlo una posición a la izquierda, para exponente -1 habría que desplazarlo una posición a la derecha, para exponente 2, dos posiciones a la izquierda, para exponente -2 , dos posiciones a la derecha, etc.

¿Cómo representar números binarios en punto flotante empleando los registros de un computador? Una solución inmediata es dividir el registro en tres partes. Una para el signo, otra para la mantisa y otra para el exponente. Supongamos, como en el caso de la representación en punto fijo, que contamos con registros de 16 bits. Podemos dividir el registro en tres zonas, la primera, de un solo bit la empleamos para guardar el signo. La segunda de, por ejemplo 11 bits, la empleamos para guardar la mantisa del número y por último los cuatro bits restantes los empleamos para guardar el exponente en binario. Podríamos entonces representar el número $-1,10100101 \times 2^3$ como,

sig.	mantisa												exponente			
1	1	1	0	1	0	0	1	0	1	0	0	0	0	0	1	1
-	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^3	2^2	2^1	2^0	

Si bien la representación que acabamos de ver sirve para el número representado, es evidente que no podemos representar así un número con exponente negativo como por ejemplo $1,011101 \times 2^{-3}$. Además la división que hemos hecho del registro de 16 bits es arbitraria; podríamos haber elegido un reparto distinto de bits entre mantisa y exponente. De hecho, hasta la década de los noventa del siglo XX, cada ordenador adoptaba su propia representación de números en punto flotante. La consecuencia era que los mismos programas, ejecutados en máquinas distintas daban diferentes soluciones.

Ya en 1976 John Palmer, de la compañía INTEL, promueve la creación de un estándar, es decir, de una representación en punto flotante que fuera adoptada por todos los fabricantes de computadoras. El primer borrador del estándar fue elaborado por William Kahan (Intel), Jerome Coonen (Berkeley) y Harold Stone (Berkeley): documento K-C-S. Inspirado en el trabajo de Kahan en el IBM 7094 (1968). El proceso de estandarización fue bastante lento. La primera versión no fue operativa hasta 1985. El resultado de estos trabajos fue el estándar actual. Conocido como IEEE 754. IEEE son las siglas de Institute of Electrical and Electronics Engineers. Se trata de una asociación que nació en Estados Unidos en el siglo XIX y que goza actualmente de carácter internacional. El IEEE se ocupa, entre otras cosas, de establecer estándares industriales en los campos de la electricidad, la electrónica y la computación. Veamos en detalle el estándar IEEE 754.

El estándar IEEE 754. En primer lugar, el estándar se estableció pensando en dos tamaños de registro el primero de ellos emplea 32 bits para guardar los números en punto flotante y recibe el nombre de estándar de precisión simple. El segundo emplea 64 bits y se conoce como estándar de precisión doble.

Simple precisión. Empecemos por describir el estándar de precisión simple. En primer lugar, se reserva un bit para contener el signo. Si el bit vale 1, se trata de un número negativo, y si vale 0 de un número positivo. De los restantes 31 bits, 23 se emplean para representar la mantisa y 8 para representar el exponente.

Si analizamos como es la mantisa de un número binario en representación de punto flotante, nos damos cuenta de que la parte entera siempre debería valer 1. Por tanto, podemos guardar en la mantisa solo la parte del número que está a la derecha del punto decimal, y considerar que el 1 correspondiente a la parte entera está implícito. Por ejemplo, si tenemos el número binario $1,010111 \times 2^3$ solo guardaríamos la cifra 010111. El 1 de la parte entera sabemos que está ahí pero no lo representamos. A este tipo de mantisa la llamaremos normalizada, más adelante veremos por qué.

En cuanto al exponente, es preciso buscar una forma de representar exponentes positivos y negativos. El estándar establece para ello lo que se llama la representación en *exceso*. Con ocho bits podemos representar hasta $2^8 = 256$ números distintos. Éstos irían desde el $[00000000] = 0$ hasta el $[11111111] = 255$. Si partimos nuestra colección de números en dos, podríamos emplear la primera mitad para representar números negativos, reservando el mayor de ellos para representar el cero, y la segunda para representar números positivos. Para obtener el valor de un número basta restar al contenido del exponente el número reservado para representar el cero. Se dice entonces que la representación se realiza en *exceso* a dicho número.

En el caso del estándar de simple precisión la primera mitad de los números disponibles iría

Exceso a 127		
Bits de exponente	equivalente decimal	Exponente representado
00000000	0	$0 - 127 = -127$
00000001	1	$1 - 127 = -126$
00000010	2	$2 - 127 = -125$
:	:	:
01111110	126	$126 - 127 = -1$
01111111	127	$127 - 127 = 0$
10000000	128	$128 - 127 = 1$
:	:	:
11111110	254	$254 - 127 = 127$
11111111	255	$255 - 127 = 128$

Cuadro 3.1: Representación *en exceso a 127*, para un exponente de 8 bits.

del 0 al 127 y la segunda mitad del 128 al 255. La representación se realiza entonces en exceso a 127. La tabla 3.1 muestra unos cuantos ejemplos de cálculo de la representación *en exceso*.

Tenemos por tanto que el exponente de 8 bits del estándar de precisión simple permite representar todos los exponentes enteros desde el -127 al 128.

Ya tenemos casi completa la descripción del estándar. Solo faltan unos detalles –aunque muy importantes– relativos a la representación de números muy grandes y muy pequeños. Empecemos con los números muy grandes.

¿Cuál es el número más grande que podríamos representar en estándar de simple precisión? En principio cabría suponer que el correspondiente a escribir un 1 en todos los bits correspondientes a la mantisa (la mayor mantisa posible) y escribir también un 1 en todos los bits correspondientes al exponente (el mayor exponente posible). Sin embargo, el estándar está pensado para proteger las operaciones del computador de los errores de desbordamiento (ver sección 3.3 más adelante). Para ello reserva el valor más alto del exponente. Así cuando el exponente contenido en un registro es 128, dicho valor no se considera propiamente un número, de hecho es preciso mirar el contenido de la mantisa para saber qué es lo que representa el registro:

- Si los bits de la mantisa son todos cero, el registro contiene la representación del infinito ∞ . El estándar especifica el resultado de algunas operaciones en el infinito: $1/\infty = 0$, $\arctan(\infty) = \pi/2$.
- Si por el contrario, el contenido de la mantisa es distinto de cero, se considera el contenido del registro como no numérico (NaN, abreviatura en inglés de *Not a Number*). Este tipo de resultados, considerados no numéricos, surgen en algunas operaciones que carecen de sentido matemático: $0/0$, $\infty \times 0$, $\infty - \infty$.

Todo esto hace que el exponente mayor que realmente representa un número sea el 127. Como ejemplo podemos construir el número más grande que podemos representar distinto de infinito. Se trata del número con la mantisa más grande posible, una mantisa formada exclusivamente por unos, seguida por el exponente más grande posible. El exponente más grande en exceso es 127, que corresponde a un exponente en binario $127 + 127 = 254 \equiv [11111110]$. Por tanto el número se representaría como,

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	11111111111111111111111111111111	11111110

En base 10 el número representado sería, $(\mathbf{1} \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{127} \approx 3,40282346385289 \times 10^{38}$. Donde hemos representado en negrita el 1 implícito, no presente en la mantisa.

Veamos ahora los números más pequeños, de acuerdo con lo dicho, el número más próximo a cero que podríamos construir sería aquel que tuviera la mantisa más pequeña y el exponente más pequeño, es decir una mantisa formada solo por ceros y un exponente formado solo por ceros. Ese número sería $(1 \cdot 2^0) \cdot 2^{-127}$ (se debe recordar que la mantisa en la representación del estándar está normalizada; lleva un 1 entero implícito).

Es evidente que si no fuera por el 1 implícito de la mantisa sería posible representar números aún más pequeños. Por otro lado, si consideramos siempre la mantisa como normalizada, es imposible representar el número 0. Para permitir la representación de números más pequeños, incluido el cero, los desarrolladores del estándar decidieron añadir la siguiente regla: *Si los bits del exponente de un número son todos ceros, es decir, si el exponente representado es -127, se considera que la mantisa de ese número no lleva un 1 entero implícito. Además el exponente del número se toma como -126*. Los números así representados reciben el nombre de números desnormalizados. Veamos algunos ejemplos.

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	10100000000000000000000000000	00000000

El exponente del número representado en la tabla anterior es 0. Por tanto, en exceso a 127 el exponente sería $0 - 127 = -127$. Este exponente corresponde a un número desnormalizado por tanto el número expresado en base 10 sería,

$$(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdots) \times 2^{-126}$$

Nótese como el exponente ha sido reducido en una unidad (-126) a la hora de calcular la representación decimal del número.

El número más próximo a cero que podemos representar mediante números desnormalizados, será aquel que tenga la menor mantisa posible distinta de cero,

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	00000000000000000000000000000	00000000

que representado en base 10 sería el número: $(0 \cdot 2^{-1} + 0 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1,401298464324817 \times 10^{-45}$. Podemos calcular para comparar cual sería el número normalizado más próximo a cero, se trata del número que tiene una mantisa formada solo por ceros, multiplicada por el exponente más pequeño distinto de cero (un exponente igual a cero, implica automáticamente un número desnormalizado),

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	00000000000000000000000000000	00000001

Si lo representamos en formato decimal obtenemos: $(\mathbf{1} \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdots + 0 \cdot 2^{-23}) \times 2^{-126} = 2^{-126} \approx 1,175494350822288 \times 10^{-38}$

Como un último ejemplo, podemos calcular cual es el número desnormalizado más grande. Debería tener la mantisa más grande posible (todo unos) con un exponente formado exclusivamente por ceros,

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
0	11111111111111111111111111111	00000000

En formato decimal, el número sería: $(1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} \approx 1,175494210692441 \times 10^{-38}$. Los dos últimos números representados, son muy próximos entre sí. De hecho, hemos cal-

culado ya su diferencia, al obtener el número desnormalizado más próximo a cero representable, $(1 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdots + 0 \cdot 2^{-23}) \times 2^{-126} - (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = (0 \cdot 2^{-1} + 0 \cdots + 1 \cdot 2^{-23}) \times 2^{-126} = 2^{-149} \approx 1,401298464324817 \times 10^{-45}$.

Doble precisión La diferencia entre los estándares de simple y doble precisión está en el tamaño de los registros empleados para representar los números. En doble precisión se emplea un registro de 64 bits. Es decir, el tamaño de los registros es el doble que el empleado en simple precisión. El resto del estándar se adapta al tamaño de la representación; se emplea 1 bit para el signo del número, 52 bits para la mantisa y 11 bits para el exponente.

En este caso el exponente puede almacenar $2^{11} = 2048$ números (desde el 0 al 2047) como en el caso de estándar de simple precisión se dividen los números por la mitad, con lo que los exponentes se representan ahora en exceso a 1023, por tanto el exponente 0 representa el valor $0 - 1023 = -1023$ y el exponente 2047 representa el valor $2047 - 1023 = 1024$.

De nuevo, el valor mayor del exponente, 1024 se emplea para representar el infinito ∞ , si va acompañado de una mantisa formada exclusivamente por ceros. En caso de que acompañe a una mantisa no nula, el contenido del registro se considera que no representa un número NaN (error de desbordamiento). Podemos, como ejemplo, obtener para el estándar de doble precisión el número más grande representable,

sig.	←mantisa, 52 bits →	← exponente, 11 bits →
0	1110	111111111110

El número representado toma en base 10 el valor: $(1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdots + 1 \cdot 2^{-52}) \times 2^{1023} \approx 1,797693134862316 \times 10^{308}$

Por último, para exponente -1023 es decir, un exponente formado exclusivamente por ceros, el número representado se considera desnormalizado; se elimina el 1 entero implícito de la representación de la mantisa y se aumenta en una unidad el exponente, que pasa así a valer -1022. Como ejemplo, podemos calcular el número más próximo a cero representable en doble precisión,

sig.	←mantisa, 52 bits →	← exponente, 11 bits →
0	0001	000000000000

El número representado toma el base 10 el valor: $(0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdots + 1 \cdot 2^{-52}) \times 2^{-1022} \approx 4,940656458412465 \times 10^{-324}$

La tabla 3.2 resume y compara las características de los dos estándares vistos,
Para terminar veamos algunos ejemplos de representación de números en el estándar IEEE 754:

1. ¿Cuál es el valor decimal del siguiente número expresado en el estándar del IEEE 754 de simple precisión precisión?

sig.	←mantisa, 23 bits →	← exponente, 8 bits →
1	1100000000000000000000000	01111100

- El bit de signo es 1, por lo tanto el número es negativo.
- El exponente sería, $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 - 127 = 124 - 127 = -3$
- A la vista del exponente, la mantisa está normalizada, $1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1,75$
- Por tanto el número en representación decimal es: $(-1)^1 \times (1,75) \times 2^{-3} = -0,21875$

2. ¿Cuál es el valor decimal del siguiente número expresado en el estándar del IEEE 754 de simple precisión?

Precisión simple. Registro de 32 bits. Exponente exceso a 127

Mantisa (23 bits)	exponente (8 bits) (-exceso)	número representado
0	$0 - 127 = -127$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \dots + 0 \cdot 2^{-23}) \times 2^{0-127} = 0$
$\neq 0$	$0 - 127 \equiv -126$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \dots + m_{23} \cdot 2^{-23}) \times 2^{0-127} \equiv -126$
	$1 - 127 = -126$	
\forall	hasta $254 - 127 = 127$	$(-1)^{bs} \cdot (1 + m_1 \cdot 2^{-1} + \dots + m_{23} \cdot 2^{-23}) \times 2^{(es \cdot 2^8 + \dots + e_1 \cdot 2^0) - 127}$
0	$255 - 127 = 128$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \dots + 0 \cdot 2^{-23}) \times 2^{255-127} \equiv \infty$
$\neq 0$	$255 - 127 = 128$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \dots + m_{23} \cdot 2^{-23}) \times 2^{255-127} \equiv \text{NaN}$

Precisión doble. Registro de 64 bits. Exponente exceso a 1023

Mantisa (52 bits)	exponente (11 bits) (-exceso)	número representado
0	$0 - 1023 = -1023$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \dots + 0 \cdot 2^{-52}) \times 2^{0-1023} = 0$
$\neq 0$	$0 - 1023 \equiv -1022$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \dots + m_{52} \cdot 2^{-52}) \times 2^{0-1023} \equiv -1022$
	$1 - 1023 = -1022$	
\forall	hasta $2046 - 1023 = 1023$	$(-1)^{bs} \cdot (1 + m_1 \cdot 2^{-1} + \dots + m_{52} \cdot 2^{-52}) \times 2^{(e_{11} \cdot 2^{10} + \dots + e_1 \cdot 2^0) - 1023}$
0	$2047 - 1023 = 1024$	$(-1)^{bs} \cdot (0 \cdot 2^{-1} + \dots + 0 \cdot 2^{-52}) \times 2^{2047-1023} \equiv \infty$
$\neq 0$	$2047 - 1023 = 1024$	$(-1)^{bs} \cdot (m_1 \cdot 2^{-1} + \dots + m_{52} \cdot 2^{-52}) \times 2^{2047-1023} \equiv \text{NaN}$

Cuadro 3.2: Comparación entre los estándares del IEEE para la representación en punto flotante.
(bs bit de signo, m_i bit de mantisa, e_i bit de exponente)

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	0100000000000000000000000	10000001

- El bit de signo es 0, por lo tanto el número es positivo.
- El exponente sería, $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 - 127 = 129 - 127 = 2$
- A la vista del exponente, la mantisa está normalizada, $1 \cdot 2^0 + 1 \cdot 2^{-2} = 0,25$
- Por tanto el número en representación decimal es: $(-1)^0 \times (0,25) \times 2^2 = +5,0$

3. ¿Cuál es la representación en el estándar IEEE 754 de simple precisión del número: 347,625?

- Convertimos el número a binario, $347,625 = 101011011,101$
- Representamos el número en formato de coma flotante, $1,01011011101 \times 2^8$
- mantisa: 01011011101 (normalizada)
- exponente: $8 \xrightarrow{\text{exceso } 127} 127 + 8 = 135 \xrightarrow{\text{binario 8 bits}} 10000111$
- signo: 0 (positivo)

Con lo cual la representación de 347,625 es,

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	01011011101000000000000	10000111

4. ¿Cuál es la representación en el estándar IEEE 754 de simple precisión del número: $\frac{5}{3}$

- $\frac{5}{3} = 1,66666 \dots$
- Pasamos la parte entera a binario: $1 = 1 \cdot 2^0$
- Pasamos la parte decimal a binario:
 $0,666666 \dots \times 2 = 1,333333 \dots$
 $0,333333 \dots \times 2 = 0,666666 \dots$
A partir de aquí se repite el periodo $\widehat{10}$ indefinidamente: $1,666666 \xrightarrow{\text{binario}} 1,101010 \dots$
- Mantisa: el número en binario quedaría: $1,101010 \dots$, con la misma representación en punto flotante, $1,101010 \dots \times 2^0$. La mantisa será, 10101010101010101010101
- Exponente: $0 \xrightarrow{\text{exceso } 127} 127 + 0 = 127 \xrightarrow{\text{binario 8 bits}} 01111111$
- Signo: 0 (positivo)

Con lo cual la representación de $\frac{5}{3}$ es,

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	10101010101010101010101	01111111

3.3. Errores en la representación numérica.

Como se ha indicado anteriormente, cualquier representación numérica que empleemos con el ordenador, está sometida a errores derivados del tamaño finito de los registros empleados. Vamos a centrarnos en el estudio de los errores cometidos cuando representamos números empleando el formato de punto flotante del estándar del IEEE754.

En primer lugar, solo hay una cantidad finita de números que admiten una representación exacta, son los que se obtienen directamente de los valores –unos y ceros– contenidos en un registro al interpretarlos de acuerdo con las especificaciones del estándar. Estos números reciben el nombre de números máquina.

Un número real no será un número máquina si,

1. Una vez representado en formato de punto flotante su exponente está fuera del rango admitido para los exponentes: es demasiado grande o demasiado pequeño.
2. Una vez representado en formato de punto flotante su mantisa contiene más dígitos de los bits que puede almacenar la mantisa del formato.

Si el exponente se sale del rango admitido, se produce un error de desbordamiento. Si se trata de un valor demasiado grande, el error de desbordamiento se produce por exceso (*overflow*). El ordenador asignará al número el valor $\pm\infty$. Si el exponente es demasiado pequeño, entonces el desbordamiento se produce por defecto (*underflow*) y el ordenador asignará al número el valor cero.

Si el tamaño de la mantisa del número excede el de la mantisas representables, la mantisa se trunca al tamaño adecuado para que sea representable. Es decir, se sustituye el número por un número máquina cercano, este proceso se conoce como redondeo y el error cometido en la representación como error de redondeo.

3.3.1. Error de redondeo unitario

Supongamos que tenemos un número no máquina $x = (1.a_1a_2 \dots a_{23}a_{24} \dots) \times 2^{\exp}$.

Aproximación por truncamiento. Si queremos representarlo empleando el estándar de simple precisión, solo podremos representar 23 bits de los que componen su mantisa. Una solución es truncar el número, eliminando directamente todos los bits de la mantisa más allá del 23 $x \approx x_T = (1.a_1a_2 \cdots a_{23}) \times 2^{\text{exp}}$. Como hemos eliminado algunos bits, el número máquina x_T , por el que hemos aproximado nuestro número, es menor que él.

Aproximación por exceso. Otra opción sería aproximar el número máquina inmediatamente superior. Esto sería equivalente a eliminar todos los bits de la mantisa más allá del 23 y sumar un bit en la posición 23 de la mantisa $x \approx x_E = (1.a_1a_2 \cdots a_{23} + 2^{-23}) \times 2^{\text{exp}}$. En este caso, estamos aproximiando por un número máquina mayor que el número aproximado.

Redondeo Es evidente que cualquier número real que admita una representación aproximada en el formato del estándar estará comprendido entre dos números máquina: $x_T \leq x \leq x_E$.

En general, el criterio que se sigue es aproximar cada número real, por truncamiento o exceso, empleando en cada caso el número máquina más cercano. Siempre que redondeamos un número cometemos un error, que podemos definir como el valor absoluto de la diferencia entre el valor real del número y su aproximación. Este error recibe el nombre de error absoluto,

$$\text{Error absoluto} = |x - x_r|$$

Donde $x_r = x_T$ si se redondeó por truncamiento o $x_r = x_E$ si se redondeó por exceso.

El intervalo entre dos números máquina consecutivos puede obtenerse restando el menor del mayor,

$$x_E - x_T = (1.a_1a_2 \cdots a_{23} + 2^{-23}) \times 2^{\text{exp}} - (1.a_1a_2 \cdots a_{23}) \times 2^{\text{exp}} = 2^{-23} \times 2^{\text{exp}}$$

Si aproximamos ahora cualquier número real comprendido en el intervalo x_T y x_E por el más cercano de estos dos, el error absoluto que cometemos, sera siempre menor o como mucho igual que la mitad del intervalo,

$$|x - x_r| \leq \frac{1}{2}|x_E - x_T| = \frac{1}{2} \cdot 2^{-23} \cdot 2^{\text{exp}}$$

Este resultado se ilustra gráficamente en la figura 3.1

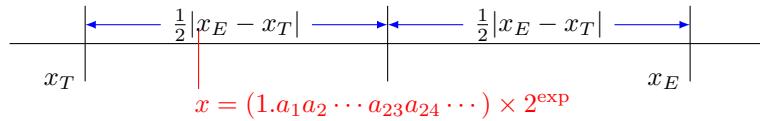


Figura 3.1: Posición relativa de un número no máquina x y su redondeo a número máquina por truncamiento x_T y por exceso x_E . Si redondeamos al más próximo de los dos, el error es siempre menor o igual a la mitad del intervalo $x_E - x_T$.

Examinando con un poco de detalle el resultado anterior, vemos que consta de tres términos. El término $\frac{1}{2}$ surge de aproximar un número real por su número máquina más cercano. El término 2^{exp} depende del tamaño del número. Para números grandes este factor será grande y para números pequeños será un factor pequeño. Por último queda el factor 2^{-23} ; este factor está directamente relacionado con la mantisa empleada en la representación. Efectivamente, si hubiéramos representado el número en el estándar de doble precisión, es fácil demostrar que el error absoluto cometido habría quedado acotado como,

$$|x - x_r| \leq \frac{1}{2} |x_E - x_T| = \frac{1}{2} \cdot 2^{-52} \cdot 2^{\text{exp}}$$

Es decir, el único factor que cambia en el error es precisamente el término relacionado con el tamaño de la mantisa. Este término recibe el nombre de precisión del computador o *epsilon del computador* (*eps*). Y vale siempre 2 elevado a menos ($-$) el número de bits de la mantisa. Por tanto, podemos generalizar la expresión para la cota del error absoluto como,

$$|x - x_r| \leq \frac{1}{2} |x_E - x_T| = \frac{1}{2} \cdot \text{eps} \cdot 2^{\text{exp}}$$

El significado del *epsilon* del computador queda aún más claro si definimos el error relativo..

$$\text{Error relativo} = \frac{|x - x_r|}{|x|} \leq \frac{1}{2} \frac{|x_E - x_T|}{|x|} = \frac{1}{2} \cdot \frac{\text{eps} \cdot 2^{\text{exp}}}{(1.a_1a_2 \dots a_{23}a_{24} \dots) \times 2^{\text{exp}}} \leq \frac{1}{2} \text{eps}$$

El error relativo pondera el valor del error cometido con la magnitud del número representado. Un ejemplo sencillo ayudará a entender mejor su significado. Imaginemos que tuviéramos un sistema de representación que solo nos permitiera representar números enteros. Si queremos representar los números $1,5$ y $1000000,5$ su representación sería 1 y 1000000 en ambos casos hemos cometido un error absoluto de $0,5$. Sin embargo si comparamos con los números representados, en el primer caso el error vale la mitad del número mientras que en el segundo no llega a una millonésima parte.

En el caso de la representación que estamos estudiando, el error relativo cometido para cualquier número representable es siempre más pequeño que la mitad del *epsilon* del computador,

$$x_r = x \cdot (1 + \delta); |\delta| \leq \frac{1}{2} \cdot \text{eps}$$

Un último comentario sobre el *epsilon* del computador, entendido como precisión. La diferencia entre dos números máquina consecutivos está estrechamente relacionada con el *epsilon*. Si tenemos un número máquina y queremos incrementarlo en la cantidad más pequeña posible, dicha cantidad es precisamente el *epsilon*, multiplicado por 2 elevado al exponente del número. La razón de que esto sea así está relacionada con el modo en que se suman dos números en la representación en punto flotante. Supongamos que tenemos un número cualquiera representado en el estándar de precisión simple,

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	1111000000000000000000000000000	10000000

El número representado tiene de exponente $2^7 - 127 = 1$. Supongamos ahora que quisieramos sumar a este número la cantidad 2^{-22} su representación en el estándar emplearía una mantisa de ceros (recordar el 1 implícito) y un exponente $-22 + 127 = 105$. Sería por tanto,

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	0000000000000000000000000000000	01101001

Para sumar dos números en notación científica es imprescindible que los dos estén representados con el mismo exponente, para entonces poder sumar directamente las mantisas. Disminuir el exponente del mayor de ellos hasta que coincida con el del menor no es posible, ya que eso supondría añadir dígitos a la parte entera de la mantisa, pero no hay bits disponibles para ello entre los asignados a la mantisa. Por tanto, la solución es aumentar el exponente del menor de los números, hasta que alcance el valor del exponente del mayor, y disminuir el valor de la mantisa desnormalizándola, es decir sin considerar el 1 implícito. Por tanto en nuestro ejemplo, debemos representar 2^{-22} empleando un exponente 1 , $2^{-22} \rightarrow 2^{-23} \cdot 2^1$,

sig.	\leftarrow mantisa (desnorm.), 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	000000000000000000000000000000001	100000000

La suma de ambos números se obtiene sumando directamente las mantisas,

sig.	\leftarrow mantisa, 23 bits \rightarrow	\leftarrow exponente, 8 bits \rightarrow
0	111100000000000000000000000000001	100000000

¿Qué pasa si tratamos de sumar un número más pequeño, por ejemplo 2^{-23} ? Al representarlo con exponente 1 para poder sumarlo el número tomaría la forma, $2^{-23} \rightarrow 2^{-24} \cdot 2^1$. Es fácil ver el problema, con una mantisa de 23 bits nos se puede representar el número 2^{-24} porque ya no hay hueco para él. La mantisa sería cero y –dado que se trata de una representación desnormalizada–, el número resultante sería cero. Por tanto, al sumarlo con el número inicial nos daría este mismo número.

Esto nos lleva a que la precisión del computador no es igual para todos los números representables, sino que depende de su magnitud. la precisión, tomada en función de la distancia entre dos números consecutivos, es: precisión = $\text{eps} \cdot 2^{exp}$ y su valor se duplica cada vez que aumentamos el exponente en una unidad. La figura 3.2 muestra esquemáticamente este fenómeno.

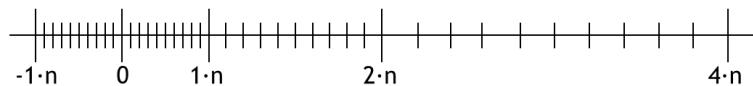


Figura 3.2: Ilustración del cambio de precisión con la magnitud de los números representados.

3.3.2. Errores de desbordamiento

En el apartado anterior hemos visto una de las limitaciones de la representación numérica de los ordenadores; el hecho de usar una mantisa finita hace que solo unos pocos números tengan representación exacta, el resto hay que aproximarlos cometiendo errores de redondeo. En este apartado nos centraremos en estudiar las limitaciones debidas al hecho de usar un exponente finito; solo podemos representar un rango limitado de valores. La figura 3.3 muestra esquemáticamente el rango de números representables. El número negativo más pequeño representable, viene definido, por un bit de signo 1, para indicar que se trata de un número negativo y la mantisa y el exponente más grandes que, dentro de las especificaciones del estándar, todavía representan un número finito. Cualquier número negativo menor que éste, produce un error de desbordamiento conocido en la literatura técnica con el nombre de *overflow* negativo. El número negativo más próximo a cero, que se puede representar será aquel que tenga bit de signo 1, la mantisa (desnormalizada) más pequeña posible y el exponente más pequeño posible. Cualquier número más próximo a cero que este, será representado por el ordenador como cero. Se ha producido en este caso un error de desbordamiento conocido como *underflow* negativo.

De modo análogo a como hemos definido el número negativo más pequeño representable, podemos definir el número positivo más grande representable. La única diferencia será que en este caso el bit de signo toma valor cero para indicar que es un número positivo. Cualquier número mayor que éste que queramos representar, provocará un error de desbordamiento (*overflow* positivo.) Por último, el número positivo más próximo a cero representable coincide con el correspondiente negativo, de nuevo salvo en el bit de signo, que ahora deberá ser cero. En el caso de tratar de representar un número más pequeño el ordenador no lo distinguirá de cero, produciéndose un desbordamiento denominado *underflow* positivo.

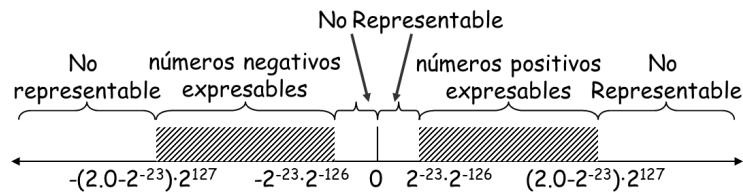


Figura 3.3: Números representables y desbordamientos en el estándar IEEE 754 de precisión simple.

3.4. Errores derivados de las operaciones aritméticas

Hasta ahora, nos hemos centrado siempre en el problema de la representación numérica del computador. En esta sección vamos a introducir un nuevo tipo de errores que tienen si cabe todavía más importancia desde el punto de vista del cálculo científico. Se trata de los errores derivados de las operaciones aritméticas.

Indirectamente ya se introdujo el problema en la sección anterior al hablar de la precisión del computador, y de la necesidad de igualar los exponentes de los sumandos al mayor de ellos antes de poder sumar en formato de punto flotante. Veamos en más detalle algunas consecuencias de la aritmética en punto flotante.

3.4.1. Acumulación de errores de redondeo

Para empezar, se ilustrará el proceso de la suma de dos números representados en base 10 en formato de punto flotante. Supongamos que queremos sumar los números 99,99 y 0,161. Supongamos además que seguimos una representación en punto flotante con las siguientes limitaciones la mantisa es de cuatro dígitos, el exponente es de dos dígitos.

Si sumamos los números, tal y como los hemos escrito más arriba, sin seguir formato de punto flotante, el resultado sería,

$$99,99 + 0,161 = 100,151$$

Supongamos que los representamos ahora en el formato de punto flotante descrito más arriba, $99,99 = 9,999 \times 10^1$, $0,161 = 1,610 \times 10^{-1}$. Vamos a descomponer el proceso de sumar estos dos números en cuatro pasos, que reflejan, esquemáticamente, el proceso que seguiría un computador.

- Alineamiento.** Consiste en representar el número más pequeño empleando el exponente del mayor. Para ellos se desplazan hacia la derecha los dígitos del número más pequeño tantas posiciones como indique la diferencia de los exponentes de los dos números y se cambiar el exponente del número mas pequeño por el del más grande.

$$1,610 \times 10^{-1} \rightarrow 0,016 \times 10^1$$

Como cabía esperar, el desplazamiento hacia la derecha de la mantisa produce la pérdida de los últimos dígitos del número. Tenemos aquí un primer error de redondeo en el proceso de alineamiento.

- Operación.** Una vez que los operandos están alineados, se puede realizar la operación. Si la operación es suma y los signos son iguales o si la operación es resta y los signos son diferentes, se suman las mantisa. En otro caso se restan.

Es importante comprobar tras realizar la operación si se ha producido desbordamiento de la mantisa; en el ejemplo propuesto:

$$\begin{array}{r} 9,999 \cdot 10^1 \\ + 0,016 \cdot 10^1 \\ \hline 10,015 \cdot 10^1 \end{array}$$

No es posible emplear directamente el resultado de la suma de las mantisas de los sumandos como mantisa de la suma ya que requeriría emplear un dígito más. El resultado obtenido desborda el tamaño de la mantisa.

3. **Normalización.** Si se ha producido desbordamiento de la mantisa, es preciso volver a normalizarla,

$$10,015 \times 10^1 = 1,0015 \times 10^2$$

4. **Redondeo.** El desplazamiento de la mantisa hace que no quepan todo los dígitos, por lo que es necesario redondearla (por truncamiento o por exceso),

$$1,0015 \times 10^2 = 1,002 \times 10^2$$

Por último, se comprueba si las operaciones realizadas han producido desbordamiento del exponente, en cuyo caso el resultado sería un número no representable: ∞ ó 0.

5. **Renormalización** Para números en representación binaria es preciso a veces volver a normalizar la mantisa después del redondeo. Supongamos, como en el ejemplo anterior, una mantisa de cuatro bits, —ahora con números representados en binario— y que tras realizar una operación suma el resultado que se obtiene es $11,111 \times 2^1$. La mantisa ha desbordado y hay que normalizarla $11,111 \times 2^1 \rightarrow 1,1111 \times 2^2$. Tenemos que redondear la mantisa y lo normal en este caso dado que el número estaría exactamente en la mitad del intervalo entre dos números representables, es hacerlo por exceso: $1,1111 \times 2^2 \rightarrow 10,000 \times 2^2$. la operación de redondeo por exceso ha vuelto a desbordar la mantisa y, por tanto, hay que volver a normalizarla: $10,000 \times 2^2 \rightarrow 1,000 \times 2^3$.

Analicemos el error cometido para la suma empleada en los ejemplos anteriores. En aritmética exacta, el número obtenido tras la suma es 100,152. Empleando la representación en punto flotante, con una mantisa de cuatro dígitos el resultado es $1,002 \times 10^2 = 100,2$. por tanto, el error absoluto cometido es,

$$|100,151 - 100,2| = 0,049$$

y el error relativo,

$$\left| \frac{100,151 - 100,2}{100,152} \right| \approx 0,000489$$

En general puede comprobarse que para cualquier operación aritmética básica \odot (suma, resta, multiplicación división) y dos números máquina x, y se cumple,

$$\text{flotante}(x \odot y) = (x \odot y) \cdot (1 + \delta) \quad |\delta| \leq \text{eps}$$

Este enunciado se conoce como el axioma fundamental de la aritmética en punto flotante: *El eps del computador es la cota superior del error relativo en cualquier operación aritmética básica realizada en punto flotante entre números máquina.*

El axioma fundamental de la aritmética en punto flotante establece una cota superior para el error. En la práctica, es frecuente que se encadenen un gran número de operaciones aritméticas elementales. Al encadenar operaciones, los errores cometidos en cada una de ellas se acumulan.

Supongamos por ejemplo que queremos realizar la operación $x \cdot (y + z)$ en aritmética flotante la operación podríamos describirla como,

$$\begin{aligned}\text{flotante}(x \cdot (y + z)) &= (x \cdot \text{flotante}(y + z)) \cdot (1 + \delta_1) \\ &= (x \cdot (y + z)) \cdot (1 + \delta_2) \cdot (1 + \delta_1) \\ &\approx (x \cdot (y + z)) \cdot (1 + 2\delta)\end{aligned}$$

Donde δ_1 representa el error relativo cometido en el producto y δ_2 el error relativo cometido en la suma. Ambos errores están acotados por el eps del ordenador ($\delta_1, \delta_2 \leq eps$). El valor δ se puede obtener como,

$$(1 + \delta_1) \cdot (1 + \delta_2) = 1 + \delta_1 \delta_2 + \delta_1 + \delta_2 \approx 1 + 2\delta, \delta = \max(\delta_1, \delta_2)$$

Podemos concluir que, en este caso, el error de redondeo relativo duplica al de una operación aritmética sencilla. En general, el error tenderá a multiplicarse con el número de operaciones aritméticas encadenadas.

Hay situaciones en las cuales los errores de redondeo que se producen durante una operación aritmética son considerables por ejemplo, cuando se suman cantidades grandes con cantidades pequeñas,

$$\text{flotante}(1,5 \cdot 10^{38} + 1,0 \cdot 10^0) = 1,5 \cdot 10^{38} + 0$$

En este caso, durante el proceso de alineamiento, es preciso desplazar la mantisa del número pequeño 38 posiciones decimales para poder sumarlo. Con cualquier mantisa que tenga menos de 38 dígitos el resultado es equivalente a convertir el segundo sumando en cero.

Otro ejemplo es la pérdida de la propiedad asociativa,

$$\left. \begin{array}{l} x = 1,5 \cdot 10^{38} \\ y = -1,5 \cdot 10^{38} \end{array} \right\} \Rightarrow (x + y) + 1 \neq x + (y + 1) \left\{ \begin{array}{l} (x + y) + 1 = 1 \\ x + (y + 1) = 0 \end{array} \right.$$

Los resultados pueden estar sometidos a errores muy grandes cuando la operación aritmética es la sustracción de cantidades muy parecidas. Si, por ejemplo, queremos realizar la operación $100,1 - 99,35 = 0,75$ y suponemos que estamos empleando una representación en punto flotante con una mantisa de cuatro dígitos y los números representados en base 10 ($100,1 = 1,001 \cdot 10^2$, $99,35 = 9,935 \cdot 10^1$)

1. Alineamiento

$$9,935 \cdot 10^1 \rightarrow 0,994 \cdot 10^2$$

2. Operación

$$\begin{array}{r} 1,001 \cdot 10^2 \\ - 0,994 \cdot 10^2 \\ \hline 0,007 \cdot 10^2 \end{array}$$

3. normalización

$$0,007 \cdot 10^2 \rightarrow 7,000 \cdot 10^{-1}$$

Los pasos 4 y 5 no son necesarios en este ejemplo. Si calculamos ahora el error absoluto de redondeo cometido,

$$|0,75 - 0,7| = 0,05$$

Y el error relativo,

$$\left| \frac{0,75 - 0,7}{0,75} \right| \approx 0,0666$$

Es decir, se comete un error de un 6,7 %. El problema en este caso surge porque en el proceso de alineamiento perdemos un dígito significativo.

En un sistema de representación en punto flotante en que los números se representan en base β y el tamaño de la mantisa es p , Si las sustracciones se realizan empleando p dígitos, el error de redondeo relativo puede llegar a ser $\beta - 1$.

Veamos un ejemplo para números en base 10 ($\beta = 10$). Supongamos que empleamos una mantisa de cuatro dígitos y que queremos realizar la operación $1,000 \cdot 10^0 - 9,999 \cdot 10^{-1}$. El resultado exacto sería, $1 - 0,9999 = 0,0001$. Sin embargo, en el proceso de alineamiento,

$$9,999 \cdot 10^{-1} \rightarrow 0,9999 \cdot 10^0$$

Si redondeamos el número por exceso, el resultado de la sustracción sería cero. Si lo redondeamos por defecto,

$$\begin{array}{r} 1,000 \cdot 10^0 \\ - 0,999 \cdot 10^0 \\ \hline 0,001 \cdot 10^0 \end{array}$$

y el error relativo cometido sería,

$$\left| \frac{1,000 \cdot 10^{-4} - 1,000 \cdot 10^{-3}}{1,000 \cdot 10^{-4}} \right| = \frac{10^{-4}(10 - 1)}{10^{-4}} = 10 - 1 \Rightarrow \beta - 1$$

Para paliar estos problemas, el estándar IEEE 754 establece que las operaciones aritméticas deben realizarse siempre empleando dos dígitos extra para guardar los resultados intermedios. Estos dos dígitos extra reciben el nombre de dígitos de protección o dígitos de guarda (*guard digits*). Si repetimos la sustracción, $100,1 - 99,35$, empleando los dos dígitos de guarda,

1. Alineamiento

$$9,935\textcolor{red}{00} \cdot 10^1 \rightarrow 0,993\textcolor{red}{50} \cdot 10^2$$

$$\begin{array}{r} 1,001\textcolor{red}{00} \cdot 10^2 \\ - 0,993\textcolor{red}{50} \cdot 10^2 \\ \hline 0,007\textcolor{red}{50} \cdot 10^2 \end{array}$$

3. normalización

$$0,007\textcolor{red}{50} \cdot 10^2 \rightarrow 7,500 \cdot 10^{-1}$$

En este caso, obtenemos el resultado exacto. En general, empleando dos bits de guarda para realizar las sustracciones el error de redondeo es siempre menor que el ϵ_{ps} del computador.

3.4.2. Anulación catastrófica

La anulación catastrófica se produce cuando en una operación aritmética, típicamente la sustracción, los dígitos más significativos de los operandos, que no están afectados por el redondeo, se cancelan entre sí. El resultado contendrá fundamentalmente dígitos que sí están afectados por errores de redondeo. Veamos un ejemplo,

Supongamos que queremos realizar la operación $b^2 - 4 \cdot a \cdot c$ empleando números en base 10, y una mantisa de 5 dígitos. Supongamos que los números empleados son: $b = 3,3357 \cdot 10^0$, $a = 1,2200 \cdot 10^0$, $c = 2,2800 \cdot 10^0$. El resultado exacto de la operación es,

$$b^2 - 4 \cdot a \cdot c = 4,944 \cdot 10^{-4} \quad (3.1)$$

Si realizamos las operaciones en la representación pedida,

$$\begin{aligned} b^2 &= 1,1126\textcolor{red}{89} \cdot 10^1 \\ 4 \cdot a \cdot c &= 1,1126\textcolor{red}{40} \cdot 10^1 \\ b^2 - 4 \cdot a \cdot c &= 5,0000 \cdot 10^{-4} \end{aligned}$$

El error relativo cometido es aproximadamente un 1 %. Como se han utilizado dos bits de guarda en las operaciones intermedias, la sustracción no comete en este caso error alguno. El error se debe a los redondeos de los resultados anteriores. Una vez realizada la sustracción, el resultado solo contiene los dígitos sometidos a redondeo ya que los anteriores se han anulado en la operación.

Como regla práctica se debe evitar al realizar operaciones aritméticas aquellas situaciones en las que se sustraen cantidades casi iguales. Veamos otro ejemplo, para ilustrar esta idea. Supongamos que queremos realizar la operación,

$$y = \sqrt{x^2 + 1} - 1$$

Para valores pequeños de x , esta operación implica una anulación con pérdida de dígitos significativos. Una posible solución es utilizar una forma alternativa de construir la ecuación. Si multiplicamos y dividimos por el conjugado,

$$y = (\sqrt{x^2 + 1} - 1) \cdot \left(\frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1} \right) = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

3.4.3. Errores de desbordamiento

Al hablar del rango finito de los números representables, se indicó cómo trata el estándar IEEE 754 los números que desbordan los límites de representación. Al realizar operaciones aritméticas, es posibles llegar a situaciones en las que el resultado sea un número no representable bien por ser demasiado grande en magnitud, (*Overflow* positivo o negativo) o por ser demasiado pequeño, (*underflow* positivo o negativo.) Un ejemplo que nos permite ilustrar este fenómeno es el del cálculo del módulo de un vector,

$$\|\vec{v}\| = \|(v_1, v_2 \dots v_n)\| = \sqrt{\sum_{i=1}^n v_i^2}$$

Supongamos que creamos el siguiente programa en Matlab para calcular el módulo de un vector

```
function m=norma(x)
%inicializamos a cero la variable que contendrá la norma del vector.
m=0;
n=length(x) %calculamos la longitud del vector.
%Creamos un bucle para ir sumando los cuadrados de las componentes
for i=1:n
m=m+x(i)^2;
end
```

```
%calculamos la raíz cuadrada del resultado del bucle
m=sqrt(m)
```

El programa anterior provocará un error de desbordamiento incluso para $n=1$, si introducimos un número cuyo cuadrado sea mayor que el mayor número representable. Por ejemplo si introducimos el número, $2^{1024/2} = 1,340780792994260e + 154$ en nuestro programa Matlab devolverá como solución *inf*. Es decir, el resultado produce un error de desbordamiento. El problema se produce en el bucle, al calcular el cuadrado del número,

$$\left(2^{1024/2}\right)^2 = 2^{1024} > (2 - 2^{-52}) \cdot 2^{1023}$$

Una vez producido el desbordamiento el resto de las operaciones quedan invalidadas. Como en casos anteriores, la solución a este tipo de problemas exige modificar la forma en que se realizan los cálculos. Un primer paso sería igualar el módulo al valor absoluto del número cuando $n=1$. De este modo, el modulo del número propuesto en el ejemplo anterior se podría calcular correctamente.

Todavía es posible mejorar el programa, y ampliar el rango de vectores para los que es posible calcular el módulo. Para ello es suficiente recurrir a un pequeño artificio: dividir los elementos del vector por el elemento de mayor tamaño en valor absoluto, calcular el módulo del vector resultante, y multiplicar el resultado de nuevo por el elemento de mayor tamaño,

$$\sqrt{\sum_{i=1}^n v_i^2} = |\max v_i| \cdot \sqrt{\sum_{i=1}^n \left(\frac{v_i}{|\max v_i|}\right)^2}$$

El siguiente código permite calcular el módulo de un vector usando este procedimiento²,

```
function m=norma(x)
n=length(x); %calculamos la longitud del vector.
%si n=1 nos limitamos a devolver el valor absoluto del número introducido
if n==1
x=abs(x);
else
%inicializamos con el primer elemento la variable que contendrá
% al mayor de los elementos del vector
mayor=abs(x(1));

%inicializamos a 1 la variable que contendrá la suma de
%los cuadrados de los elementos divididos por el valor de ello

nscalado=1

%creamos un bucle para ir sumando los cuadrados de las componentes.
%empezamos en el segundo elemento, puesto que el primero ya lo tenemos.
for i=2:n
%calculamos el valor absoluto del elemento i
modxi=abs(x(i))
%comparamos con modxi con el mayor elemento obtenido hasta
%aquí
```

²El programa no funcionará correctamente para vectores cuyos primeros elementos sean 0, (0,0,0…)

```

if mayor<modxi
%si modxi es mayor, será el elemento más grande encontrado
%hasta esta iteración
%cambiamos el valor de la suma
mscalado=1+mscalado*(modxi/mayor)^2;
% definimos mayor como el nuevo valor encontrado
mayor=modxi;
else
%si no es el más grande, nos limitamos a sumarlo al resto
mscalado=mscalado+(modxi/mayor)^2;

end
%una vez completado el bucle que calcula la suma de cuadrados,
%obtenemos la raíz cuadrada, y multiplicamos por el mayor.
m=mayor*sqrt(mscalado)

```

Si aplicamos este programa a obtener la norma del vector,

$$x = [2^{1024/2} \ 2^{1024/2}]$$

obtenemos como resultado,

$$m = 1,896150381621836 \cdot 10^{154}$$

En lugar de un error de desbordamiento (*inf*).

Capítulo 4

Cálculo de raíces de una función

4.1. Raíces de una función

Se entiende por raíces de una función real $f(x) : \mathbb{R} \rightarrow \mathbb{R}$, los valores $x = r$ que satisfacen, $f(r) = 0$.

El cálculo de las raíces de una función, tiene una gran importancia en la ciencia, donde un número significativo de problemas pueden reducirse a obtener la raíz o raíces de una ecuación.

La obtención de la raíz de una ecuación es inmediata en aquellos casos en que se conoce la forma analítica de su función inversa f^{-1} , ($f(x) = y \Rightarrow f^{-1}(y) = x$). En este caso, $r = f^{-1}(0)$. Por ejemplo,

$$\begin{aligned}f(x) &= x^2 - 4 \\f^{-1}(y) &= \pm\sqrt{y+4} \Rightarrow r = f^{-1}(0) = \pm 2\end{aligned}$$

Sin embargo, en muchos casos de interés las funciones no pueden invertirse. Un ejemplo, extraído de la física es la ecuación de Kepler para el cálculo de las órbitas planetarias,

$$x - a \sin(x) = b$$

Donde a y b son parámetros conocidos y se desea conocer el valor de x . La solución de la ecuación de Kepler es equivalente a obtener las raíces de la función $f(x) = x - a \sin(x) - b$. (La figura 4.1 muestra un ejemplo de dicha función.) En este caso, no se conoce la función inversa, y solo es posible conocer el valor de la raíz, aproximadamente, empleando métodos numéricos.

Métodos iterativos Todos los métodos que se describen en este capítulo, se basan en procedimientos iterativos. La idea es estimar un valor inicial para la raíz r_0 , y a partir de él ir refinando paso a paso la solución, de modo que el resultado se acerque cada vez más al valor real de la raíz. Cada nueva aproximación a la raíz se obtiene a partir de las aproximaciones anteriores.

$$\begin{aligned}r_0 &\rightarrow r_1 \rightarrow r_2 \rightarrow \cdots \rightarrow r_k \rightarrow \cdots \\|f(r_0)| &\geq |f(r_1)| \geq |f(r_2)| \geq \cdots \geq |f(r_k)| \geq \cdots\end{aligned}$$

El proceso que lleva de una solución aproximada a la siguiente se conoce con el nombre de *iteración*. Lo habitual es que en cada iteración se realicen las mismas operaciones matemáticas una y otra vez.

El proceso se detiene cuando la solución alcanzada se estima lo suficientemente próxima a la solución real como para darla por buena. Para ello, se suele establecer un valor (*tolerancia*) que

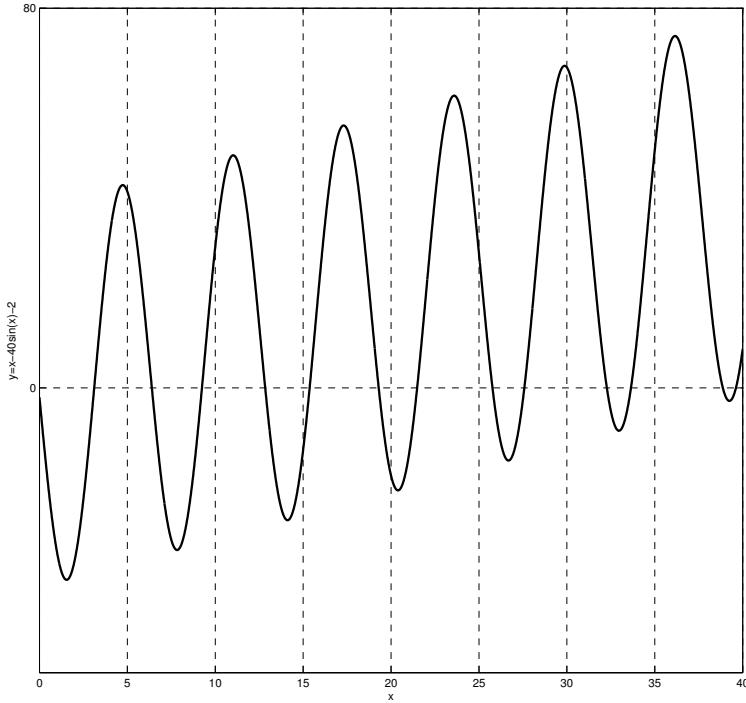


Figura 4.1: Ejemplo de ecuación de Kepler para $a = 40$ y $b = 2$

actúa como criterio de convergencia. De este modo, las iteraciones se repiten hasta que se llega a un valor r_n que cumple,

$$|f(r_n)| \leq (tol)$$

Se dice entonces que el algoritmo empleado para obtener la raíz ha convergido en n iteraciones. Por otro lado, es importante señalar que los algoritmos para el cálculo de raíces de una función no siempre convergen. Hay veces en que no es posible aproximarse cada vez más al valor de la raíz bien por la naturaleza de la función o bien por que el algoritmo no es adecuado para obtenerla.

Búsqueda local. Una función puede tener cualquier número de raíces, incluso infinitas, basta pensar por ejemplo en funciones trigonométricas como $\cos(x)$. Una característica importante de los métodos descritos en este capítulo es que solo son capaces de aproximar una raíz. La raíz de la función a la que el método converge depende de el valor inicial r_0 con el que se comienza la búsqueda iterativa¹. Por ello reciben el nombre de métodos locales. Si queremos encontrar varias (o todas) las raíces de una determinada función, es preciso emplear el método para cada una de las raíces por separado, cambiando cada vez el punto de partida.

¹En ocasiones, como veremos más adelante no se suministra al algoritmo un valor inicial, sino un intervalo en el que buscar la raíz

4.2. Metodos iterativos locales

4.2.1. Método de la bisección

Teorema de Bolzano.

Si una función $f(x)$, continua en el intervalo $[a, b]$, cambia de signo en los extremos del intervalo: $f(a) \cdot f(b) \leq 0$, debe tener una raíz en el intervalo $[a, b]$. (figura: 4.2)

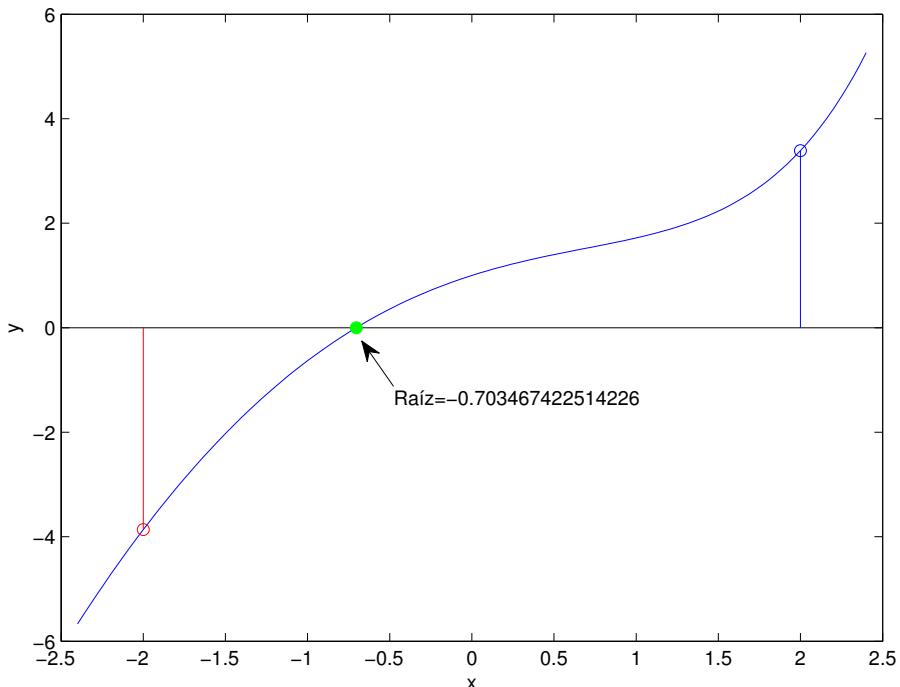


Figura 4.2: Ilustración del teorema de Bolzano

El conocido teorema de Bolzano, suministra el método más sencillo de aproximar la raíz de una función: Se parte de un intervalo inicial en el que se cumpla el teorema; y se va acotando sucesivamente el intervalo que contiene la raíz, reduciéndolo a la mitad en cada iteración, de forma que en cada nuevo intervalo se cumpla siempre el teorema de Bolzano.

En la figura 4.3 se muestra un diagrama de flujo correspondiente al método de la bisección. El punto de partida es un intervalo $[a, b]$ en el que se cumple el teorema de Bolzano, y que contiene por tanto al menos una raíz. Es interesante hacer notar que el teorema de Bolzano se cumple siempre que la función sea continua en el intervalo $[a, b]$ y existan un número impar de raíces. Por esto es importante realizar cuidadosamente la elección del intervalo $[a, b]$, si hay más de una raíz, el algoritmo puede no converger.

Una vez que se tiene el intervalo se calcula el punto medio c . A continuación se compara el valor que toma la función en c , es decir $f(c)$ con la tolerancia. Si el valor es menor que ésta, el algoritmo ha encontrado un valor aproximado de la raíz con la tolerancia requerida, con lo que c es la raíz y no hace falta seguir buscando. Si por el contrario, $f(c)$ está por encima de la tolerancia requerida,

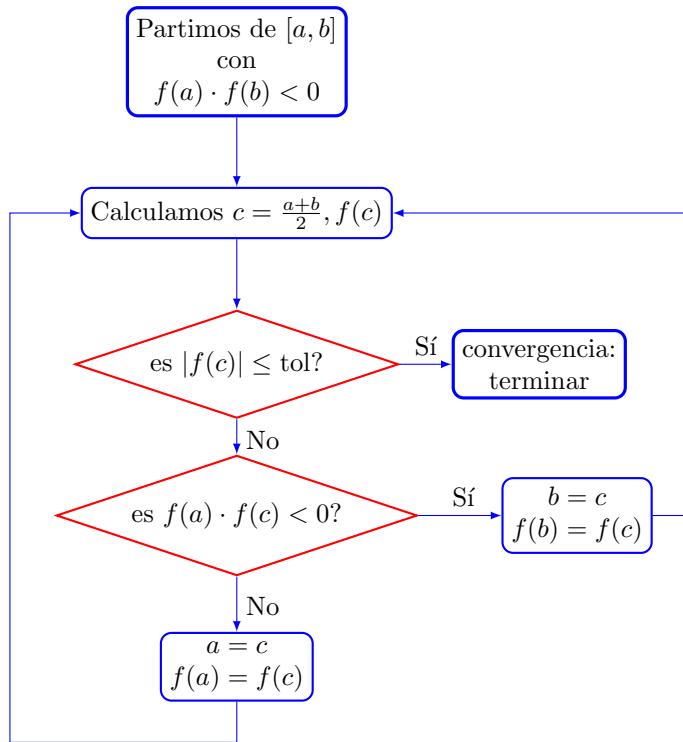


Figura 4.3: Diagrama de flujo del método de la bisección

comparamos su signo con el que toma la función en uno cualquiera de los extremos del intervalo, En el diagrama de flujo se ha elegido el extremo a , pero el algoritmo funcionaría igualmente si eligiéramos b . Si el signo de $f(c)$ coincide con el que toma la función en el extremo del intervalo elegido, c sustituye al extremo, (hacemos $a = c$ y $f(a) = f(c)$) si por el contrario el signo es distinto, hacemos que c sustituya al otro extremo del intervalo. (hacemos $b = c$ y $f(b) = f(c)$). Este proceso se repetirá hasta que se cumpla que $|f(c)| \leq tol$

El proceso se muestra gráficamente en la figura 4.4, para un caso particular. Se trata de obtener la raíz de la función mostrada en la figura 4.2, $f(x) = e^x - x^2$. Esta función tiene una única raíz: $r \approx -0,0735$. Para iniciar el algoritmo se ha elegido un intervalo $[a = -2, b = 2]$. La figura 4.4, muestra tres iteraciones sucesivas, y la solución final, que se obtiene al cabo de ocho iteraciones en este ejemplo, para el que se ha empleado una tolerancia $tol = 0,01$. En la secuencia de imágenes se puede observar también la evolución del intervalo de búsqueda, $[-2, 2] \rightarrow [-2, 0] \rightarrow [-1, 0] \rightarrow [-1, -0,5] \dots$; así como el cambio alternativo del límite derecho o izquierdo, para asegurar que la raíz queda siempre dentro de los sucesivos intervalos de búsqueda obtenidos.

4.2.2. Método de interpolación lineal o (*Regula falsi*)

Este método supone una mejora del anterior ya que, en general, converge más rápidamente. La idea es modificar el modo en que calculamos el punto c . En el caso del método de la bisección el criterio consistía en ir tomando en cada iteración el punto medio del intervalo que contiene la raíz. El método de interpolación lineal, elige como punto c el punto de corte con el eje x , de la recta que pasa por los puntos $(a, f(a))$ y $(b, f(b))$. Es decir la recta que corta a la función $f(x)$ en ambos límites del intervalo que contiene a la raíz buscada. La recta que pasa por ambos puntos

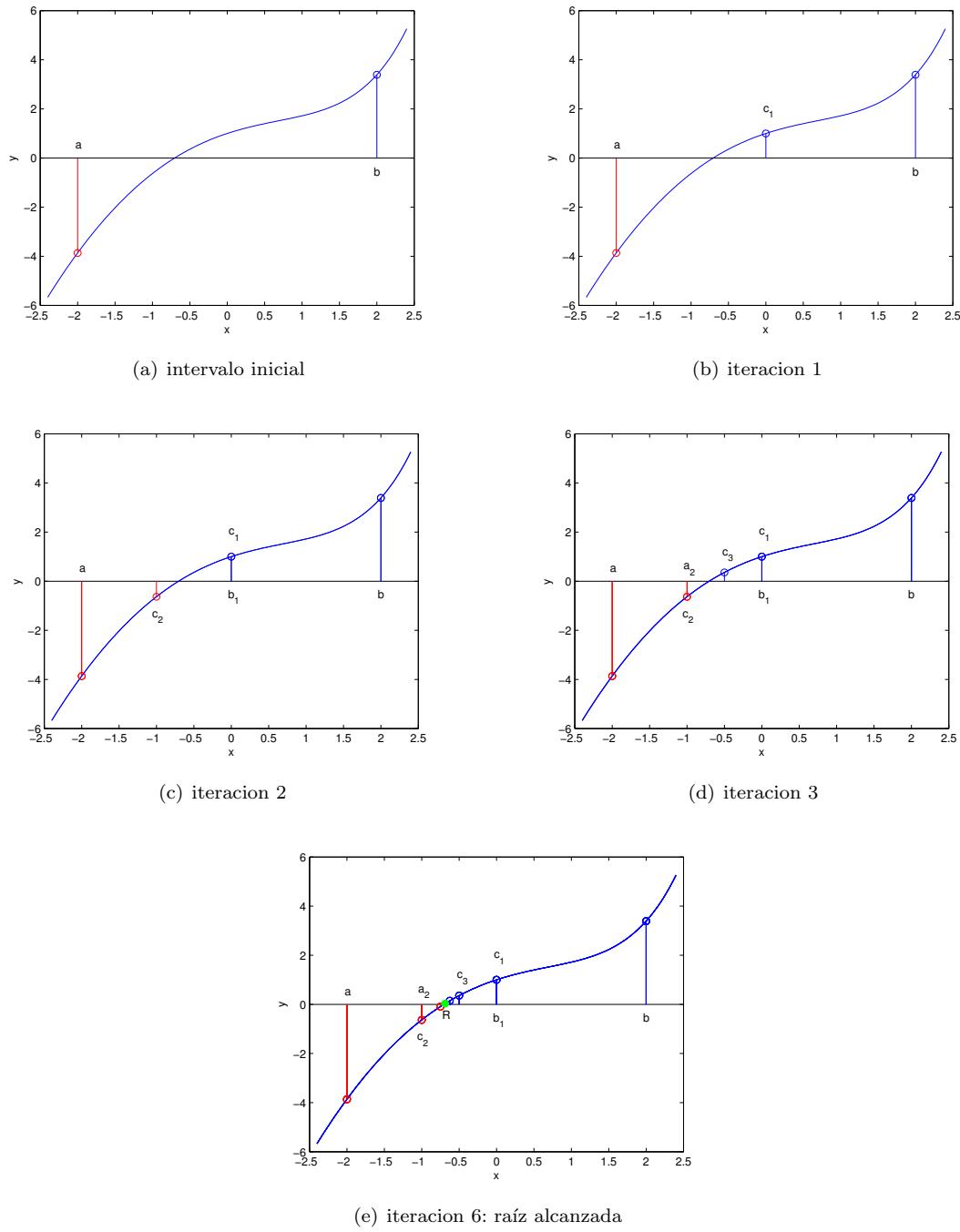


Figura 4.4: proceso de obtención de la raíz de una función por el método de la bisección

puede construirse a partir de ellos como,

$$y = \frac{f(a) - f(b)}{a - b} \cdot (x - b) + f(b)$$

el punto de corte con el eje x , que será el valor que tomaremos para c , se obtiene cuando $y = 0$,

$$0 = \frac{f(a) - f(b)}{a - b} \cdot (x - b) + f(b)$$

y despejando $c \equiv x$ en la ecuación anterior obtenemos,

$$c = b - \frac{f(b)}{f(b) - f(a)} \cdot (b - a)$$

La figura 4.5 muestra gráficamente la posición del punto c obtenido mediante el método de interpolación.

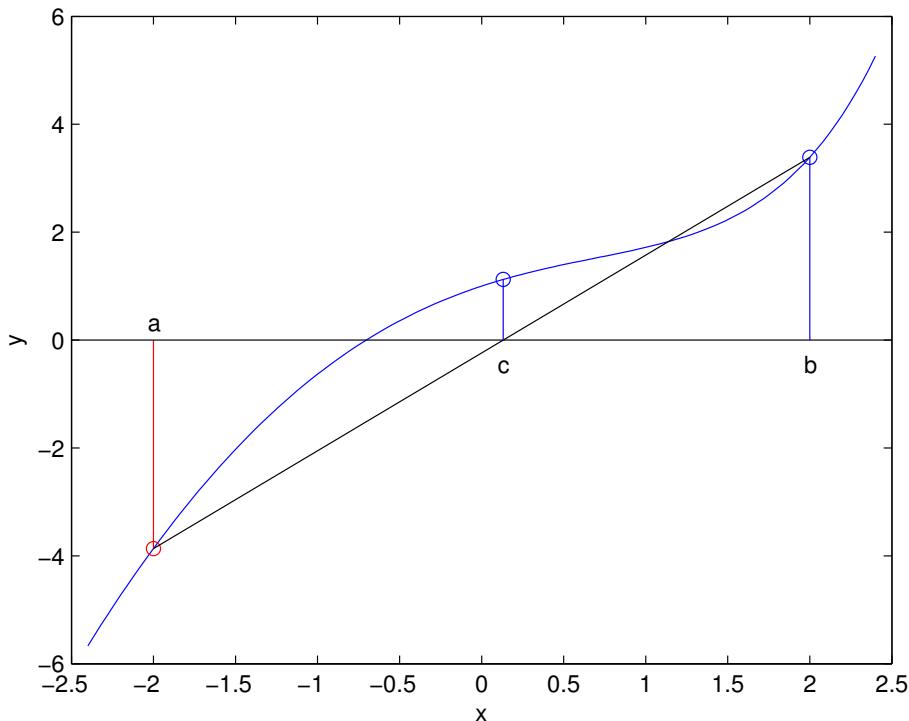


Figura 4.5: Obtención de la recta que une los extremos de un intervalo $[a, b]$ que contiene una raíz de la función

Por lo demás, el procedimiento es el mismo que en el caso del método de la biseción. Se empieza con un intervalo $[a, b]$ que contenga una raíz, se obtiene el punto c por el procedimiento descrito y se intercambia c con el extremo del intervalo cuya imagen $f(a)$ o $f(b)$ tenga el mismo signo que $f(c)$ el procedimiento se repite iterativamente hasta que $f(c)$ sea menor que el valor de tolerancia preestablecido.

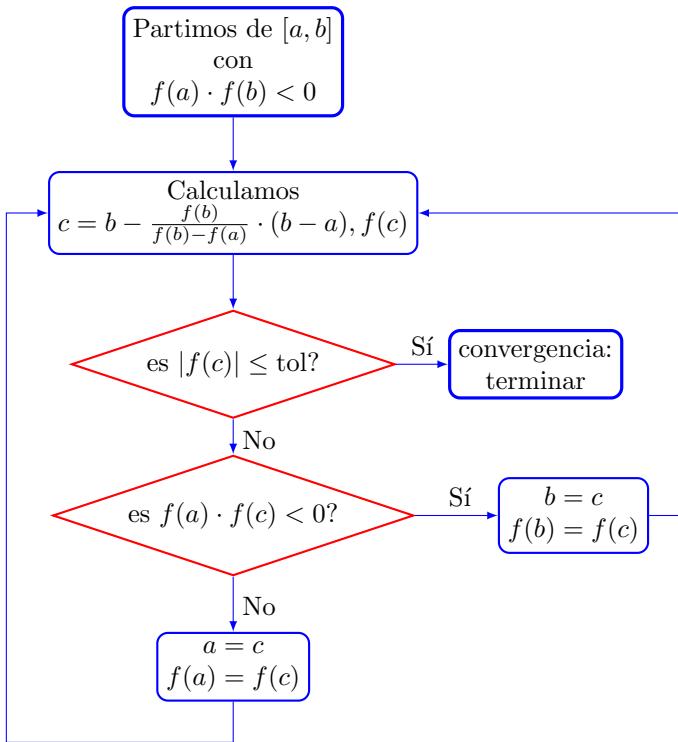


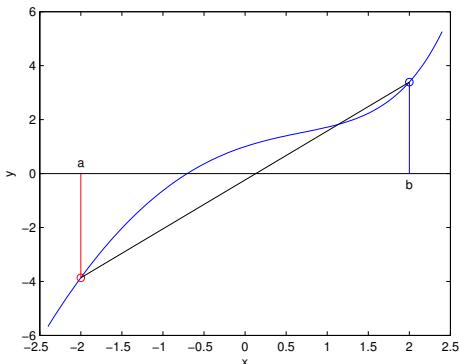
Figura 4.6: Diagrama de flujo del método de interpolación lineal

En la figura 4.6 se muestra el diagrama de flujo para el método de interpolación lineal. Como puede verse, es idéntico al de la bisección excepto en el paso en que se obtiene el valor de c , donde se ha sustituido el cálculo del punto medio del intervalo de búsqueda, por el cálculo del punto de corte con el eje de abscisas de la recta que une los extremos del intervalo.

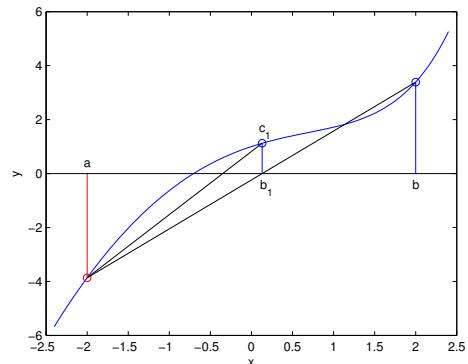
La figura 4.7 Muestra gráficamente el proceso iterativo seguido para obtener la raíz de una función en un intervalo mediante el método de interpolación lineal. Se ha empleado la misma función y el mismo intervalo inicial que en el caso de la bisección.

Es fácil ver, sin embargo, que los puntos intermedios que obtiene el algoritmo hasta converger a la raíz son distintos. De hecho, el algoritmo emplea ahora tan solo siete iteraciones para obtener la raíz, empleando el mismo valor para la tolerancia, 0.01, que se empleó en el método de la bisección.

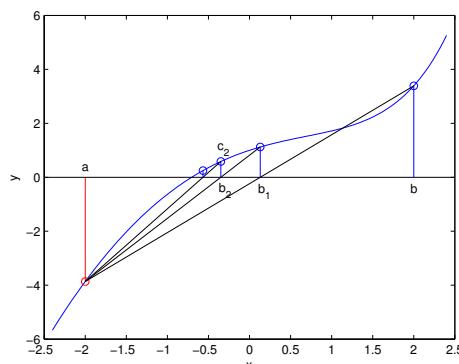
Una observación final, se ha dicho al principio que éste método supone una mejora al método anterior de la bisección. Esto no siempre es cierto. El método de la bisección tiene una tasa de convergencia constante, cada iteración divide el espacio de búsqueda por la mitad. Sin embargo la convergencia del método de interpolación lineal depende de la función $f(x)$ y de la posición relativa de los puntos iniciales $(a, f(a))$ y $(b, f(b))$ con respecto al la raíz. Por esto no es siempre cierto que converja más rápido que el método de la bisección. Por otro lado, el cálculo de los sucesivos valores del punto c , requiere más operaciones aritméticas en el método de interpolación, con lo que cada iteración resulta más lenta que en el caso de la bisección.



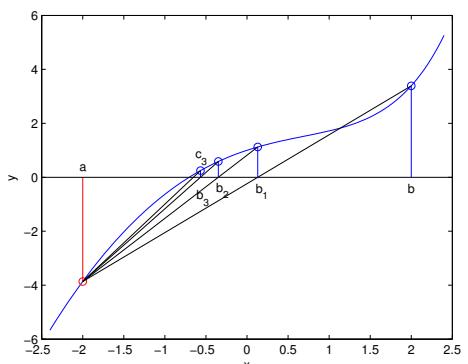
(a) intervalo inicial



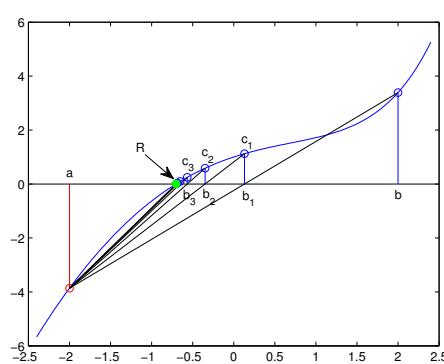
(b) iteracion 1



(c) iteracion 2



(d) iteracion 3



(e) iteracion 6: raíz alcanzada

Figura 4.7: Proceso de obtención de la raíz de una función por el método de interpolación lineal

4.2.3. Método de Newton-Raphson

El método de Newton se basa en la expansión de una función $f(x)$ en serie de Taylor en el entorno de un punto x_0 ,

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \cdots + \frac{1}{n!}f^{(n)}(x_0)(x - x_0)^n + \cdots$$

Pertenece a una familia de métodos ampliamente empleados en cálculo numérico. La idea en el caso del método de Newton es aproximar la función para la que se desea obtener la raíz, mediante el primer término de la serie de Taylor. Es decir aproximar localmente $f(x)$, en el entorno de x_0 por la recta,

$$f(x_0) + f'(x_0)(x - x_0)$$

Esta recta, es precisamente la recta tangente a la curva $f(x)$ en el punto x_0 (figura 4.8)

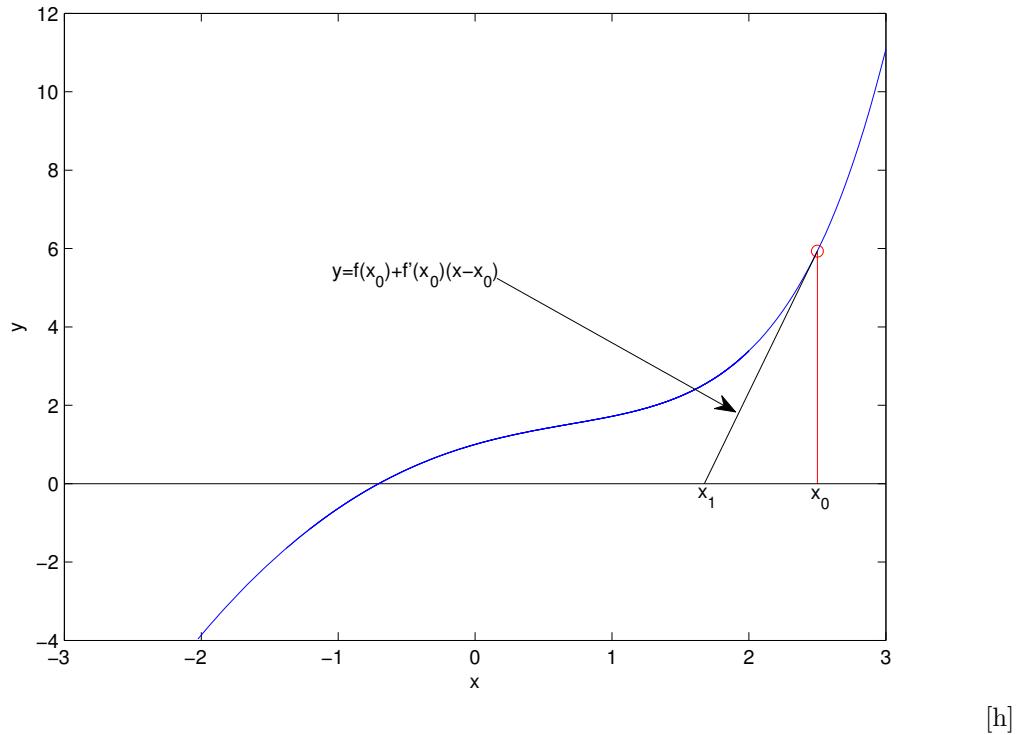


Figura 4.8: Recta tangente a la función $f(x)$ en el punto x_0

El método consiste en obtener el corte de esta recta tangente con el eje de abscisas,

$$0 = f(x_0) + f'(x_0)(x - x_0)$$

y despejando x,

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

A continuación se evalúa la función en el punto obtenido $x \rightarrow f(x)$. Como en los métodos anteriores, se compara el valor de $f(x)$ con una cierta tolerancia preestablecida. Si es menor, el valor de x se toma como raíz de la función. Si no, se vuelve aplicar el algoritmo, empleando ahora el valor de x que acabamos de obtener como punto de partida. Cada cálculo constituye una nueva iteración y los sucesivos valores obtenidos para x , convergen a la raíz,

$$x_0 \rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \rightarrow x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \rightarrow \dots \rightarrow x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \rightarrow \dots$$

La figura 4.9 muestra un diagrama de flujo correspondiente al método de Newton. Si se compara con los diagramas de flujo de los algoritmos anteriores, el algoritmo de Newton resulta algo más simple de implementar. Sin embargo es preciso evaluar en cada iteración el valor de la función y el de su derivada.

El cálculo de la derivada, es el punto débil de este algoritmo, ya que para valores x_0 próximos a un mínimo o máximo local obtendremos valores de la derivada próximos a cero, lo que puede causar un error de desbordamiento al calcular el punto de corte de la recta tangente con el eje de abscisas o hacer que el algoritmo converja a una raíz alejada del punto inicial de búsqueda.

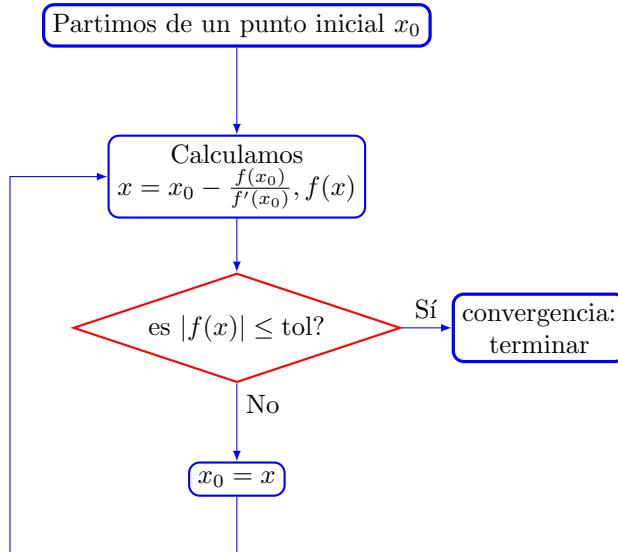


Figura 4.9: Diagrama de flujo del método de Newton-Raphson

La figura 4.10 muestra un ejemplo de obtención de la raíz de una función mediante el método de Newton. El método es más rápido que los dos anteriores, es decir, partiendo de una distancia comparable a la raíz, es el que converge en menos iteraciones.

En el ejemplo de la figura se ha obtenido la raíz para la misma función que en los ejemplos del método de la bisección e interpolación lineal. Se ha empezado sin embargo en un punto más alejado de la raíz, para que pueda observarse mejor en la figura la evolución del algoritmo. En cada uno de los gráficos que componen la figura pueden observarse los pasos del algoritmo: dado el punto x_i , se calcula la recta tangente a la función $f(x)$ en el punto y se obtiene un nuevo punto x_{i+1} , como el corte de dicha recta tangente con el eje de abscisas.

En este ejemplo el algoritmo converge en las cinco iteraciones que se muestran en la figura, para la misma tolerancia empleada en los métodos anteriores, $tol = 0,01$. El punto de inicio empleado

fue $x_0 = 2,5$, por tanto esta fuera del intervalo $[-2, 2]$ y más alejado de la raíz que en el caso de los métodos anteriores.

4.2.4. Método de la secante

El método de la secante podría considerarse una variante del método de newton en el que se sustituye la recta tangente al punto x_0 por la recta secante que une dos puntos obtenidos en iteraciones sucesivas. La idea es *aproximar* la derivada a la función f en el punto x_n por la pendiente de una recta secante, es decir de una recta que corta a la función en dos puntos,

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Las sucesivas aproximaciones a la raíz de la función se obtienen de modo similar a las del método de Newton, simplemente sustituyendo la derivada de la función por su valor aproximado,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \approx x_n - \frac{(x_n - x_{n-1}) \cdot f(x_n)}{f(x_n) - f(x_{n-1})}$$

Para iniciar el algoritmo, es preciso emplear en este caso dos puntos iniciales. La figura 4.11 muestra un ejemplo.

El método podría en este punto confundirse con el de interpolación, sin embargo tiene dos diferencias importantes: En primer lugar, la elección de los dos puntos iniciales x_0 e x_1 , no tienen por qué formar un intervalo que contenga a la raíz. Es decir, podrían estar ambos situados al mismo lado de la raíz. En segundo lugar, los puntos obtenidos se van sustituyendo por orden, de manera que la nueva recta secante se construye siempre a partir de los dos últimos puntos obtenidos, sin prestar atención a que el valor de la raíz esté contenido entre ellos. (No se comparan los signos de la función en los puntos para ver cual se sustituye, como en el caso del método de interpolación).

La figura 4.12 muestra un diagrama de flujo para el método de la secante. El diagrama es básicamente el mismo que el empleado para el método de Newton. Las dos diferencias fundamentales son, que ahora en lugar de evaluar la función y la derivada en cada iteración, se calcula el valor del punto de corte de la recta que pasa por los dos últimos puntos obtenidos (es decir, empleamos una recta secante, que corta a la curva en dos puntos, en lugar de emplear una recta tangente).

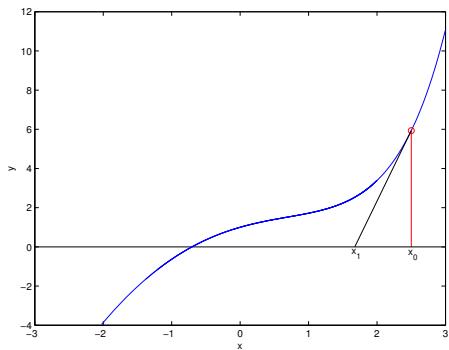
Además es preciso actualizar, en cada iteración, el valor de los dos últimos puntos obtenidos: el más antiguo se desecha, el punto recién obtenido sustituye al anterior y éste al obtenido dos iteraciones antes.

La figura 4.13 muestra un ejemplo de la obtención de una raíz por el método de la secante. Se ha empleado de nuevo la misma función que en los ejemplos anteriores, tomando como valores iniciales, $x_0 = -2,5$ y $x_1 = 0,5$. La tolerancia se ha fijado en $tol = 0,01$ también como en los anteriores algoritmos descritos. En este caso, el algoritmo encuentra la raíz en cinco iteraciones. Cada uno de los gráficos que compone la figura 4.13, muestra la obtención de un nuevo punto a partir de los dos anteriores.

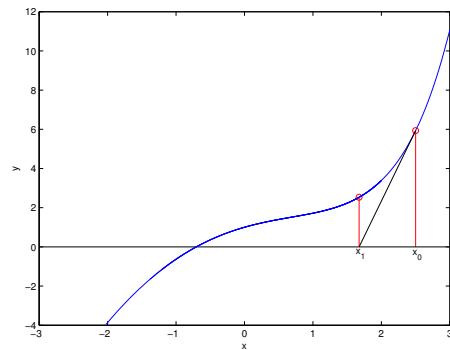
En la iteración 2, puede observarse como el nuevo punto se obtiene a partir de dos puntos que están ambos situados a la derecha de la raíz, es decir, no forman un intervalo que contenga a la raíz. Aquí se pone claramente de manifiesto la diferencia con el método de interpolación lineal. De hecho, com ya se ha dicho, el método de la secante puede iniciarse tomando los dos primeros puntos a uno de los lados de la raíz.

El método es, en principio, más eficiente que el de la bisección y el de interpolación lineal, y menos eficiente que el de Newton.

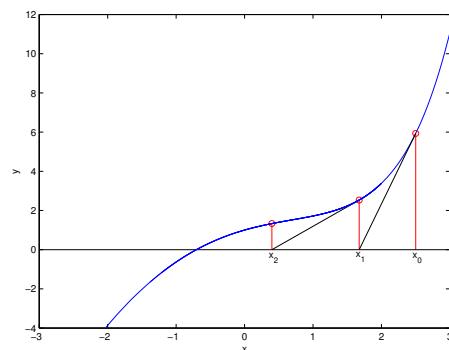
La ventaja de este método respecto al de Newton es que evita tener que calcular explícitamente la derivada de la función para la que se quiere calcular la raíz. El algunos casos, la obtención de la forma analítica de dicha derivada puede ser compleja.



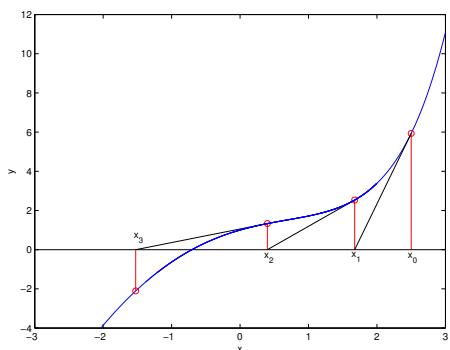
(a) intervalo inicial



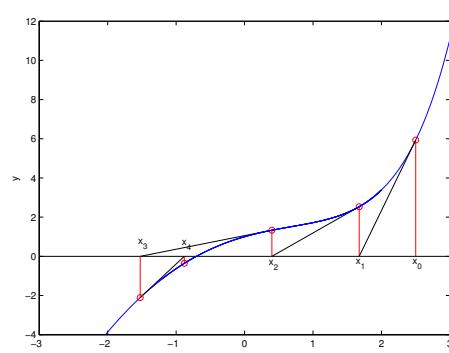
(b) iteracion 1



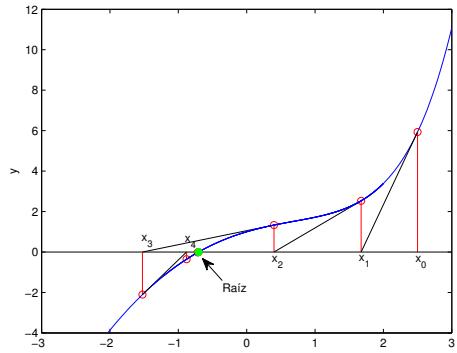
(c) iteracion 2



(d) iteracion 3

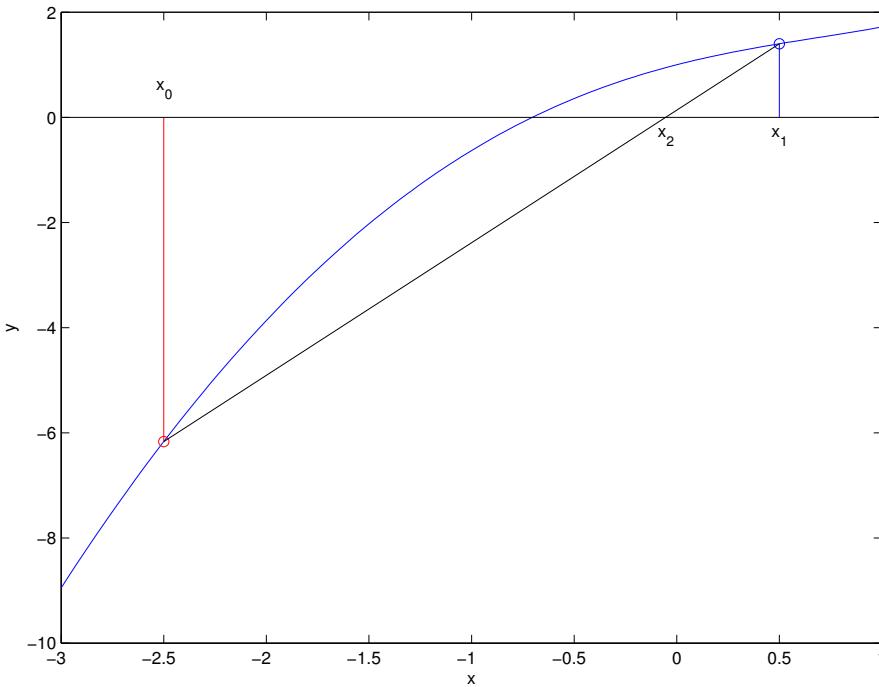


(e) iteracion 4



(f) iteracion 5: raíz de la función

Figura 4.10: Proceso de obtención de la raíz de una función por el método de Newton

Figura 4.11: Recta secante a la función $f(x)$ en los puntos x_0 y x_1

4.2.5. Método de las aproximaciones sucesivas o del punto fijo

El método del punto fijo es, como se verá a lo largo de esta sección, el más sencillo de programar de todos. Desafortunadamente, presenta el problema de que no podemos aplicarlo a todas las funciones. Hay casos en los que el método no converge, con lo que no es posible emplearlo para encontrar la raíz o raíces de una función.

Punto fijo de una función. Se dice que un punto x_f es un punto fijo de una función $g(x)$ si se cumple,

$$g(x_f) = x_f$$

Es decir, la imagen del punto fijo x_f es de nuevo el punto fijo. Así por ejemplo la función,

$$g(x) = -\sqrt{e^x}$$

Tiene un punto fijo en $x_f = -0,703467$, porque $g(-0,703467) = -0,703467$. La existencia de un punto fijo puede obtenerse gráficamente, representando en un mismo gráfico la función $y = g(x)$ y la recta $y = x$. Si existe un punto de corte entre ambas gráficas, se trata de un punto fijo. La figura 4.14, muestra gráficamente el punto fijo de la función $g(x) = -\sqrt{e^x}$ del ejemplo anterior.

Una función puede tener uno o más puntos fijos o no tener ninguno. Por ejemplo, la función $y = \sqrt{e^x}$ no tiene ningún punto fijo.

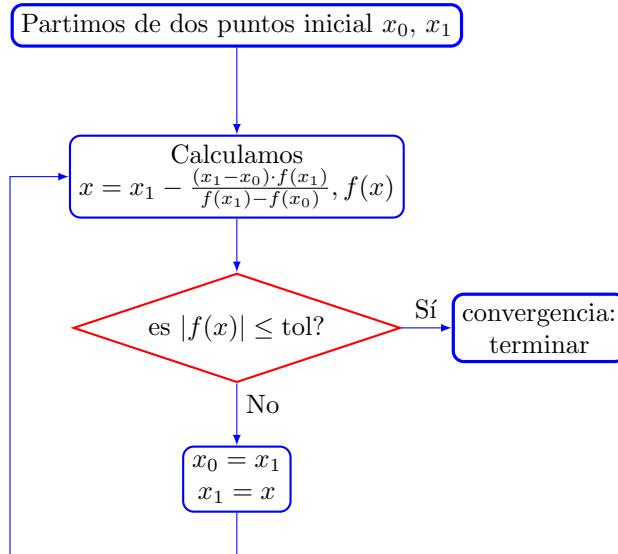


Figura 4.12: Diagrama de flujo del método de la secante

Punto fijo atractivo. Supongamos ahora que, a partir de la función $g(x)$ creamos la siguiente sucesión,

$$x_{n+1} = g(x_n)$$

Es decir, empezamos tomando un punto inicial x_0 y a partir de él vamos obteniendo los siguientes valores de la sucesión como,

$$x_0 \rightarrow x_1 = g(x_0) \rightarrow x_2 = g(x_1) = g(g(x_0)) \rightarrow \dots \rightarrow x_{n+1} = g(x_n) = g(g(\dots(g(x_0)))) \rightarrow \dots$$

Decimos que un punto fijo x_f de la función $g(x)$ es un punto fijo atractivo si la sucesión $x_{n+1} = g(x_n)$ converge al valor x_f , siempre que x_0 se tome suficientemente cercano a x_f . Cómo de cerca tienen que estar x_0 y x_f para que la serie converja, es una cuestión delicada. De entrada, es importante descartar que hay funciones que tienen puntos fijos no atractivos, por ejemplo, la función $g(x) = x^2$ tiene dos puntos fijos $x = 0$ y $x = 1$. El primero es el límite de la sucesión $x_{n+1} = g(x_n)$ para cualquier valor inicial x_0 contenido en el intervalo abierto $(-1, 1)$. El punto $x = 1$ resulta inalcanzable para cualquier sucesión excepto que el punto de inicio sea él mismo $x_0 = x_f = 1$.

Hay algunos casos en los que es posible, para determinadas funciones, saber cuando uno de sus puntos fijos es atractivo,

Teorema de existencia y unicidad del punto fijo. Dada una función $g(x)$, continua y diferenciable en un intervalo $[a, b]$, si se cumple que, $\forall x \in [a, b] \Rightarrow g(x) \in [a, b]$, entonces $g(x)$ tiene un punto fijo en el intervalo $[a, b]$.

Si además existe una constante positiva $k < 1$ y se cumple que la derivada $|g'(x)| \leq k$, $\forall x \in (a, b)$, entonces el punto fijo contenido en $[a, b]$ es único.

Para demostrar la primera parte del teorema, se puede emplear el teorema de Bolzano. Si se cumple que $g(a) = a$ o que $g(b) = b$, entonces a o b serían el punto fijo. Supongamos que no es así; entonces tiene que cumplirse que $g(a) > a$ y que $g(b) < b$. Si construimos una función, $f(x) = g(x) - x$ esta función, que es continua por construcción, cumple que $f(a) = g(a) - a > 0$ y

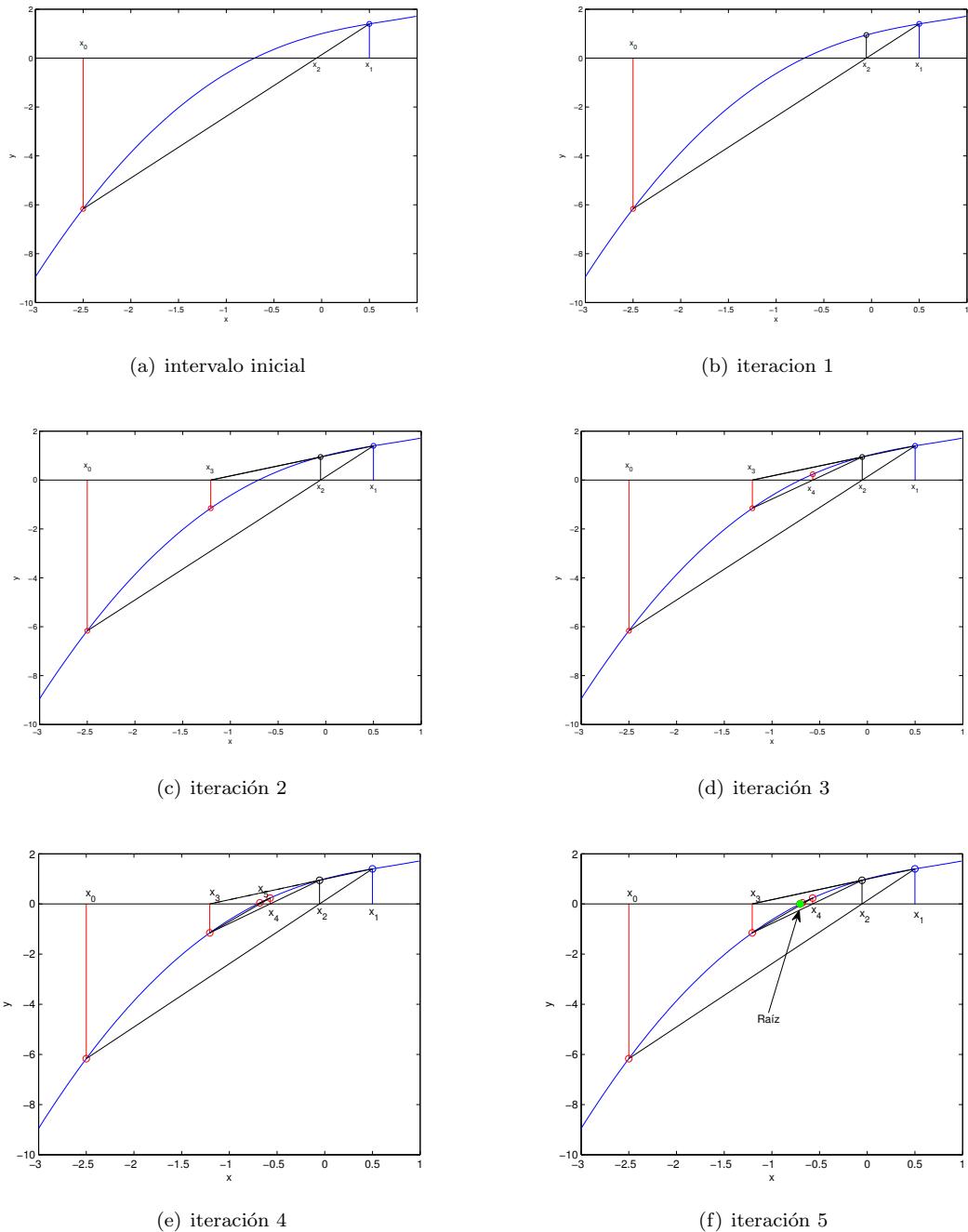


Figura 4.13: proceso de obtención de la raíz de una función por el método de la secante

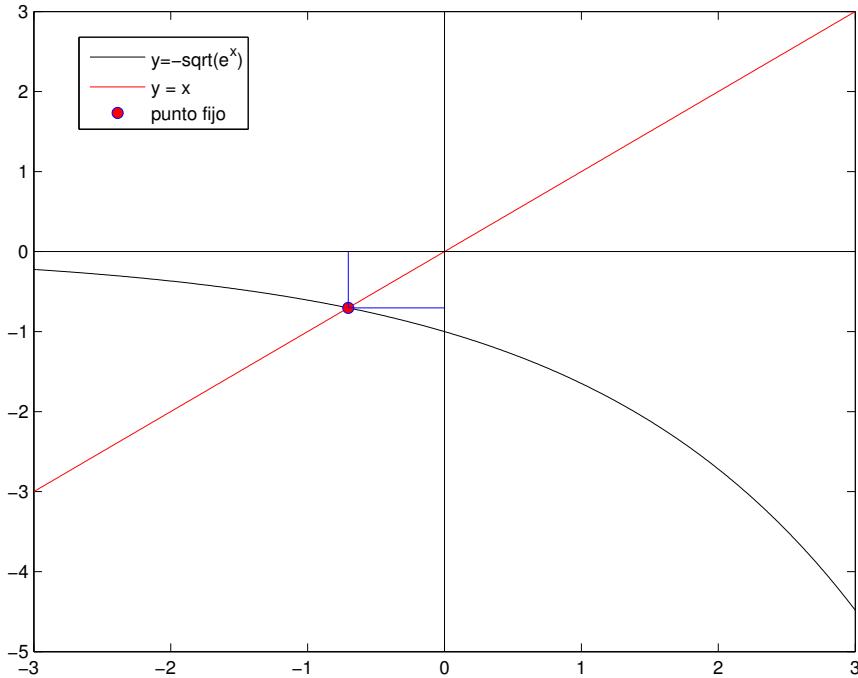


Figura 4.14: Obtención gráfica del punto fijo de la función, $g(x) = -\sqrt{e^x}$

$f(b) = g(b) - b < 0$. Pero entonces, debe existir un punto, $x_f \in [a, b]$ para el cual $f(x_f) = 0$ y, por tanto, $f(x_f) = g(x_f) - x_f = 0 \Rightarrow g(x_f) = x_f$. Es decir, x_f es un punto fijo de $g(x)$.

La segunda parte del teorema puede demostrarse empleando el teorema de valor medio. Si suponemos que existen dos puntos fijos distintos $x_{f1} \neq x_{f2}$ en el intervalo $[a, b]$, según el teorema del valor medio, existe un punto ξ comprendido entre x_{f1} y x_{f2} para el que se cumple,

$$\frac{g(x_{f1}) - g(x_{f2})}{x_{f1} - x_{f2}} = g'(\xi)$$

Por tanto,

$$|g(x_{f1}) - g(x_{f2})| = |x_{f1} - x_{f2}| \cdot |g'(\xi)| \leq |x_{f1} - x_{f2}| \cdot k < |x_{f1} - x_{f2}|$$

Pero como se trata de puntos fijos $|g(x_{f1}) - g(x_{f2})| = |x_{f1} - x_{f2}|$. con lo que llegaríamos al resultado contradictorio,

$$|x_{f1} - x_{f2}| = |g(x_{f1}) - g(x_{f2})| \leq |x_{f1} - x_{f2}| \cdot k < |x_{f1} - x_{f2}|$$

Salvo que, en contra de la hipótesis inicial, se cumpla que $x_{f1} = x_{f2}$. En cuyo caso, solo puede existir un único punto fijo en el intervalo $[a, b]$ bajo las condiciones impuestas por el teorema.

Teorema de punto fijo (atractivo). ² Dada una función $g(x)$, continua y diferenciable en un intervalo $[a, b]$, que cumple que, $\forall x \in [a, b] \Rightarrow g(x) \in [a, b]$ y que $|g'(x)| \leq k$, $\forall x \in (a, b)$, con $0 < k < 1$, entonces se cumple que, para cualquier punto inicial x_0 , contenido en el intervalo $[a, b]$, la sucesión $x_{n+1} = g(x_n)$ converge al único punto fijo del intervalo $[a, b]$.

La demostración puede obtenerse de nuevo a partir del teorema del valor medio. Si lo aplicamos al valor inicial x_0 y al punto fijo x_f , obtenemos,

$$|g(x_0) - g(x_f)| = |x_0 - x_f| \cdot |g'(\xi)| \leq |x_0 - x_f| \cdot k$$

Para la siguiente iteración tendremos,

$$|g(x_1) - g(x_f)| \leq |x_1 - x_f| \cdot k \leq |x_0 - x_f| \cdot k^2$$

puesto que, $x_1 = g(x_0)$ y $x_f = g(x_f)$, puesto que x_f es el punto fijo.

Por simple inducción tendremos que para el término enésimo de la sucesión,

$$|g(x_n) - g(x_f)| \leq |x_{n-1} - x_f| \cdot k \leq |x_{n-2} - x_f| \cdot k^2 \leq \cdots \leq |x_0 - x_f| \cdot k^n$$

Pero

$$\lim_{n \rightarrow \infty} k^n = 0 \Rightarrow \lim_{n \rightarrow \infty} |x_n - x_f| \leq \lim_{n \rightarrow \infty} |x_0 - x_f| k^n = 0$$

Es decir, la sucesión $x_{n+1} = g(x_n)$ converge al punto fijo x_f .

El método del punto fijo. Como ya hemos visto, obtener una raíz de una función $f(x)$, consiste en resolver la ecuación $f(x) = 0$. Supongamos que podemos descomponer la función $f(x)$ como la diferencia de dos términos, una función auxiliar, $g(x)$, y la propia variable x

$$f(x) = g(x) - x$$

Encontrar una raíz de $f(x)$ resulta entonces equivalente a buscar un punto fijo de $g(x)$.

$$f(x) = 0 \rightarrow g(x) - x = 0 \rightarrow g(x) = x$$

En general, a partir de una función dada $f(x)$, es posible encontrar distintas funciones $g(x)$ que cumplan que $f(x) = g(x) - x$. No podemos garantizar que cualquiera de las descomposiciones que hagamos nos genere una función $g(x)$ que tenga un punto fijo. Además, para funciones que tengan más de una raíz, puede suceder que distintas descomposiciones de la función converjan a distintas raíces. Si podemos encontrar una que cumpla las condiciones del teorema de punto fijo que acabamos de enunciar, en un entorno de una raíz de $f(x)$, podemos desarrollar un método que obtenga iterativamente los valores de la sucesión $x_{n+1} = g(x_n)$, a partir de un valor inicial x_0 . El resultado se aproximarán al punto fijo de $g(x)$, y por tanto a la raíz de $f(x)$ tanto como queramos. Bastará, como en los métodos anteriores, definir un valor (tolerancia), por debajo del cual consideraremos que el valor obtenido es suficientemente próximo a la raíz como para darlo por válido.

La figura 4.15 muestra un diagrama de flujo del método del punto fijo.

La idea es elegir cuidadosamente el punto inicial x_0 , para asegurar que se encuentra dentro del intervalo de convergencia del punto fijo. A continuación, calculamos el valor de $g(x_0)$, el resultado será un nuevo valor x . Comprobamos la diferencia entre el punto obtenido y el anterior y si es menor que una cierta tolerancia, consideramos que el método ha convergido, dejamos de iterar, y devolvemos el valor de x obtenido como resultado. Si no, copiamos x en x_0 y volvemos a empezar

²Hay varios teoremas de punto fijo definidos en distintos contextos matemáticos. Aquí se da una versión reducida a funciones $f(x) : \mathbb{R} \rightarrow \mathbb{R}$

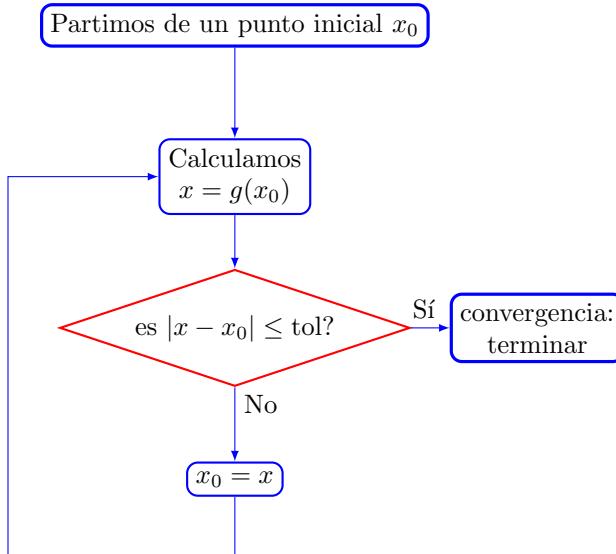


Figura 4.15: Diagrama de flujo del método del punto fijo. Nótese que la raíz obtenida corresponde a la función $f(x) = g(x) - x$

todo el proceso. Es interesante hacer notar que el algoritmo converge cuando la diferencia entre dos puntos consecutivos es menor que un cierto valor. De acuerdo con la *condición* de punto fijo $g(x_0) = x_0$, dicha distancia, sería equivalente a la que media entre $f(x_0) = g(x_0) - x_0$, la función para la que queremos obtener la raíz, y 0.

Veamos un ejemplo. Supongamos que queremos calcular por el método del punto fijo la raíz de la función $y = e^x - x^2$, que hemos empleado en los ejemplos de los métodos anteriores.

En primer lugar, debemos obtener a partir de ella una nueva función que cumpla que $f(x) = g(x) - x$. Podemos hacerlo de varias maneras despejando una ' x ', de la ecuación $e^x - x^2 = 0$. Para ilustrar los distintos casos de convergencia, despejaremos x de tres maneras distintas .

$$e^x - x^2 = 0 \Rightarrow \begin{cases} x = \pm\sqrt{e^x} \\ x = \ln(x^2) = 2 \cdot \ln(|x|) \\ x = \frac{e^x}{x} \end{cases}$$

En nuestro ejemplo hemos obtenido tres formas distintas de *despejar* la variable x . La cuestión que surge inmediatamente es, si todas las funciones obtenidas, tienen un punto fijo y, en caso de tenerlo, si es posible alcanzarlo iterativamente.

En el primer caso, $x = \pm\sqrt{e^x}$, obtenemos las dos ramas de la raíz cuadrada. Cada una de ellas constituye a los efectos de nuestro cálculo una función distinta. Si las dibujamos junto a la recta $y = x$ (figura 4.16), observamos que solo la rama negativa la corta. Luego será esta rama $g(x) = -\sqrt{e^x}$, la que podremos utilizar para obtener la raíz de la función original por el método del punto fijo. La rama positiva, al no cortar a la recta $y = x$ en ningún punto, es una función que carece de punto fijo.

No es difícil demostrar, que la función $g(x) = -\sqrt{e^x}$ cumple las condiciones del teorema de punto fijo descrito más arriba para el intervalo $(-\infty, 0]$. Luego el algoritmo del punto fijo debería converger para cualquier punto de inicio x_0 contenido en dicho intervalo. De hecho, para esta función, el algoritmo converge desde cualquier punto de inicio (Si empezamos en punto positivo,

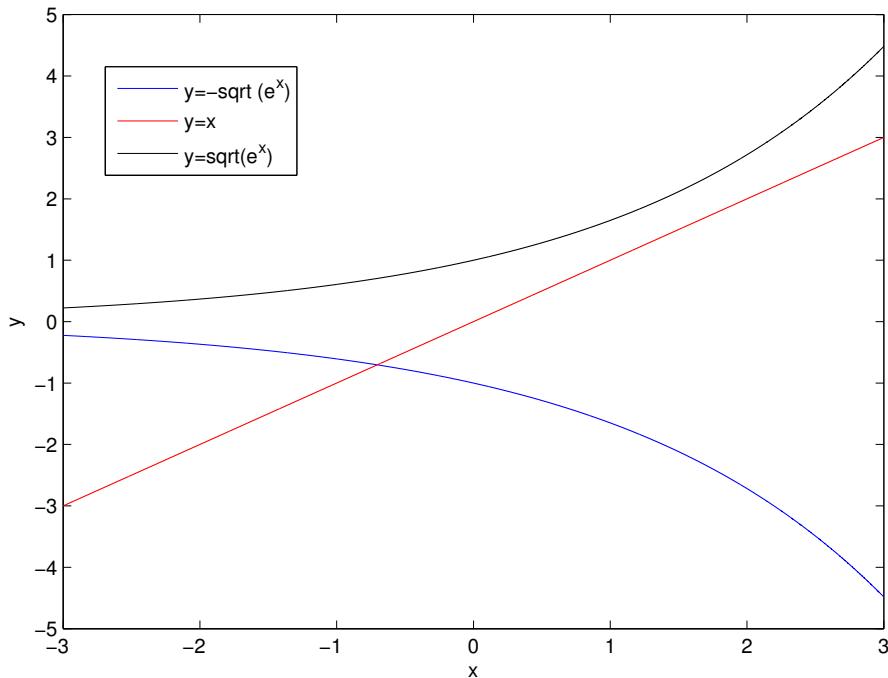


Figura 4.16: $g(x) = \pm\sqrt{e^x}$, Solo la rama negativa tiene un punto fijo.

el siguiente punto, x_1 será negativo, y por tanto estará dentro del intervalo de convergencia). Esta función es un ejemplo de que el teorema suministra una condición suficiente, pero no necesaria para que un punto fijo sea atractivo.

La figura 4.17 muestra un ejemplo del cálculo de la raíz de la función $f(x) = e^x - x^2$ empleando la función $g(x) = -\sqrt{e^x}$, para obtener el punto fijo. Se ha tomado como punto de partida $x_0 = 2,5$, un valor fuera del intervalo en el que se cumple el teorema. Como puede observarse en 4.17(a). A pesar de ello el algoritmo converge rápidamente, y tras 5 iteraciones, 4.17(f), ha alcanzado el punto fijo —y por tanto la raíz buscada—, con la tolerancia impuesta

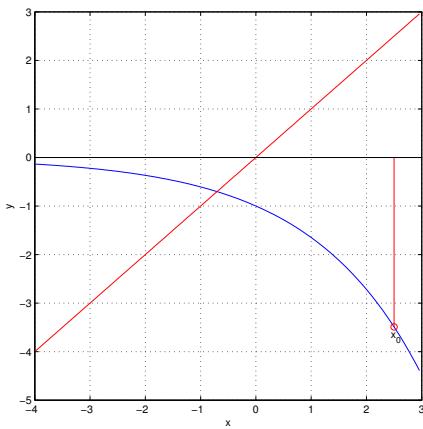
Si tratamos de emplear la función $g(x) = \ln(x^2)$ para obtener la raíz, observamos que la función no cumple el teorema para ningún intervalo que contenga la raíz.

La figura 4.18 muestra la función $g(x)$, la recta $y = x$ y la evolución del algoritmo tras cuatro evaluaciones. Es fácil deducir que el algoritmo saltará de la rama positiva a la negativa y de ésta volverá a saltar de nuevo a la positiva.

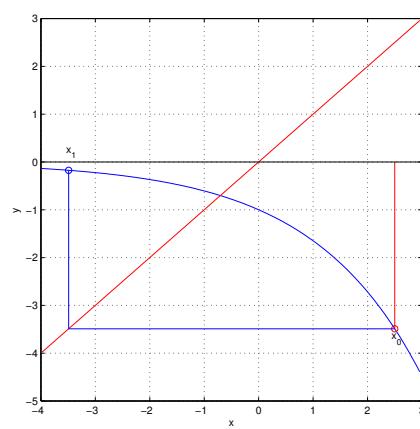
La función presenta una asíntota vertical en el 0. Si se empieza desde $x_0 = 0$, $x_0 = 1$ o $x_0 = -1$ el algoritmo no converge, puesto que la función diverge hacia $-\infty$. Para el resto de los valores, la función oscila entre una rama y otra. Si en alguna de las oscilaciones acierta a pasar suficientemente cerca del punto fijo, $x_n - x_{n-1} \leq tol$, el algoritmo habrá aproximado la raíz, aunque propiamente no se puede decir que converja.

La figura 4.19(a), muestra la evolución del algoritmo, tomando como punto inicial $x_0 = -0,2$. Tras 211 iteraciones el algoritmo 'atrapa la raíz'. En este caso la tolerancia se fijó en $tol = 0,01$.

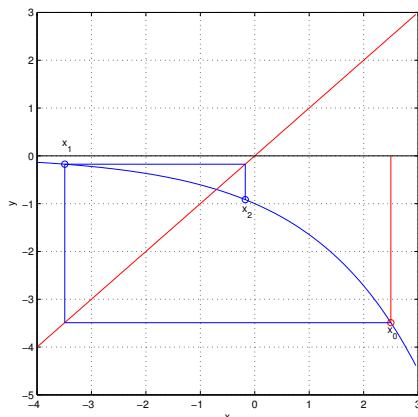
La gráfica 4.19(b) muestra una ampliación de 4.19(a) en la que pueden observarse en detalles los valores obtenidos para las dos últimas iteraciones. Las dos líneas horizontales de puntos marcan



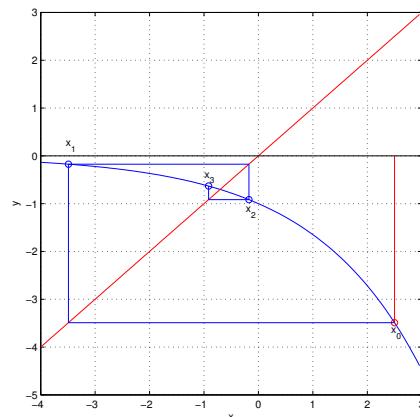
(a) valor inicial



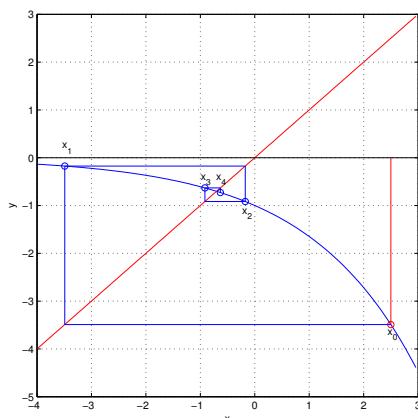
(b) iteracion 1



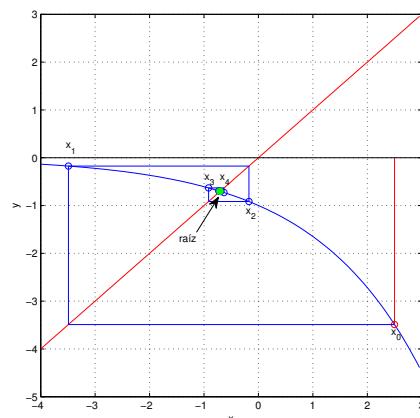
(c) iteración 2



(d) iteración 3



(e) iteración 4



(f) iteración 5

Figura 4.17: proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = -\sqrt{e^x}$

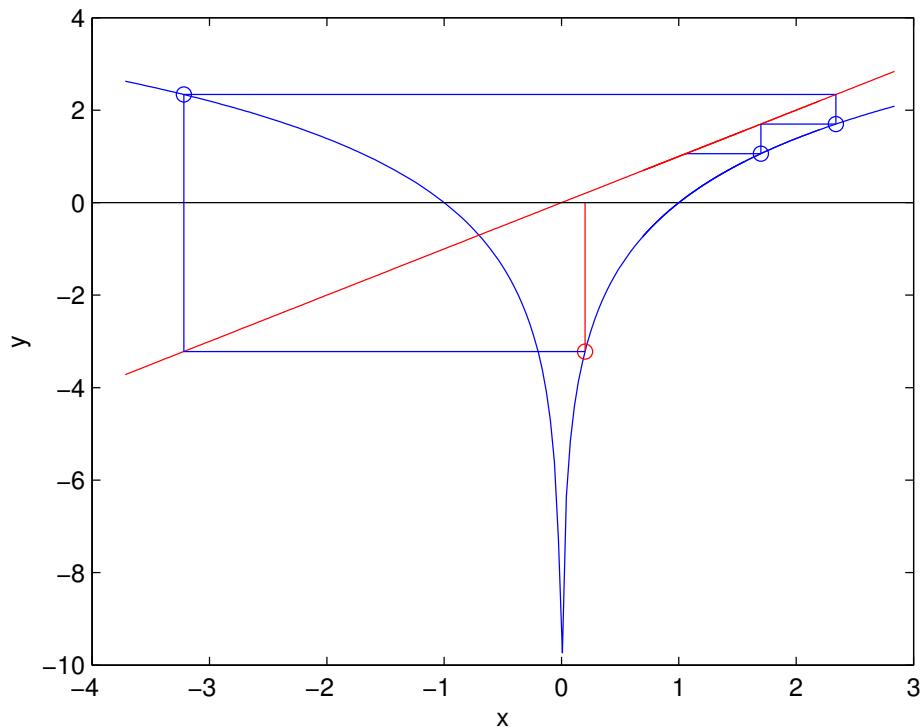


Figura 4.18: primeras iteraciones de la obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \ln(x^2)$.

los límites raíz $\pm tol$.

El algoritmo se detiene porque la diferencia entre los valores obtenidos en las dos últimas iteraciones caen dentro de la tolerancia. El valor obtenido en la penúltima iteración, que proviene de la rama positiva de la función $g(x)$ cae muy cerca del punto fijo. El último valor obtenido, se aleja de hecho del valor de la raíz, respecto al obtenido en la iteración anterior, pero no lo suficiente como para salirse de los límites de la banda marcada por la tolerancia. Como resultado, se cumple la condición de terminación y el algoritmo se detiene.

Si disminuimos el valor de la tolerancia, no podemos garantizar que el algoritmo converja. De hecho, si trazamos cuales habrían sido los valores siguientes que habría tomado la solución del algoritmo, caso de no haberse detenido, es fácil ver que se alejan cada vez más de la raíz. De nuevo habrá que esperar a que cambie de rama y vuelva a pasar otra vez cerca del punto fijo para que haya otra oportunidad de que el algoritmo *atrape* la solución.

La gráfica 4.19(c) muestra la evolución del error en función del número de iteración. Como puede observarse, el error oscila de forma caótica de una iteración a la siguiente. De hecho, el estudio de las sucesiones de la forma $x_{n+1} = g(x_n)$ constituyen uno de los puntos de partidas para la descripción y el análisis de los llamados sistemas caóticos.

Uno sencillo, pero muy interesante es el de la ecuación logística discreta, $x_{n+1} = R \cdot (1-x_n) \cdot x_n$. Esta ecuación muestra un comportamiento muy distinto, según cual sea el valor de R y el valor inicial x_0 con el que empecemos a iterar.

Por último, si empleamos la función $g(x) = \frac{e^x}{x}$, no se cumple el teorema de punto fijo en ningún

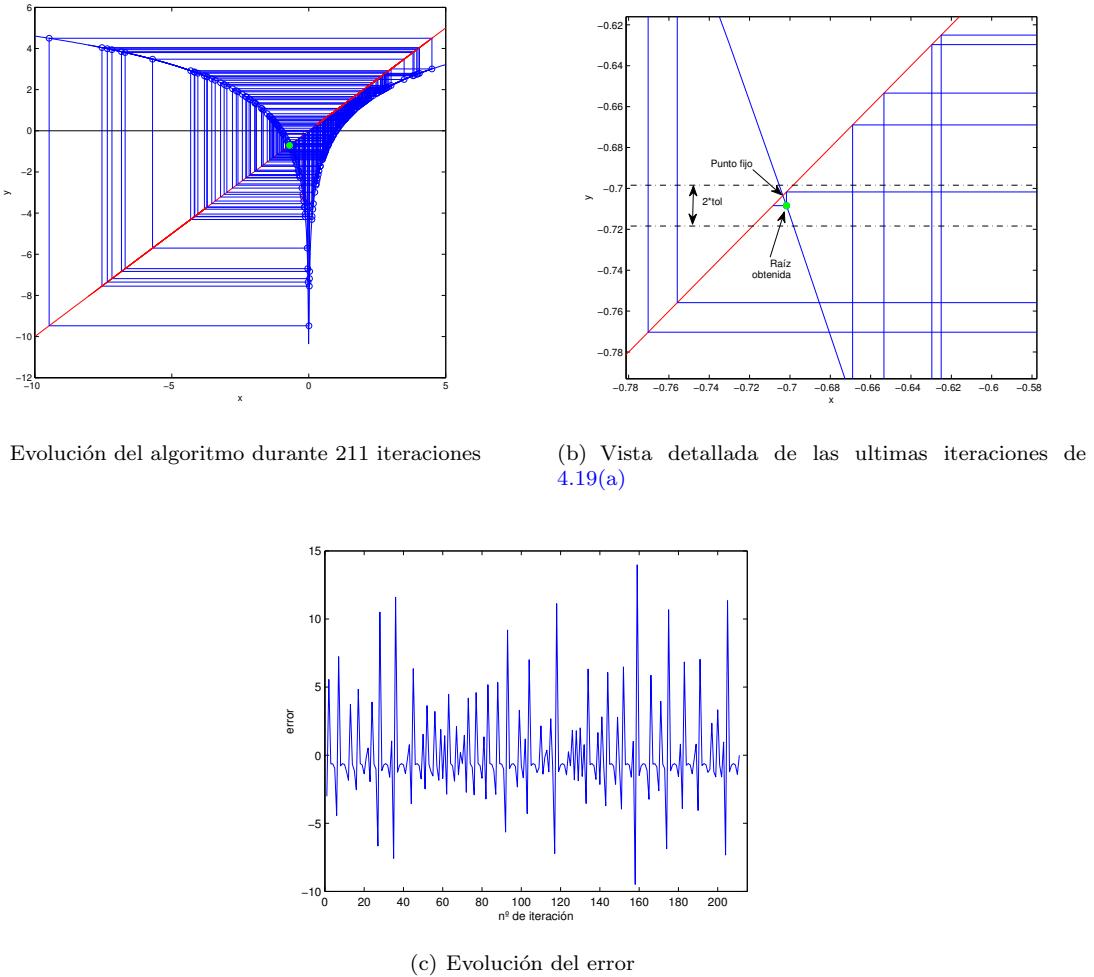


Figura 4.19: proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \ln(x^2)$, el método oscila sin converger a la solución.

punt. En este caso, el algoritmo diverge siempre. La figura 4.20 muestra la evolución del algoritmo del punto fijo para esta función. Se ha elegido un punto de inicio $x_0 = -0,745$, muy próximo al valor de la raíz, para poder observar la divergencia de las soluciones obtenidas con respecto al punto fijo. Como puede verse, el valor de x_n cada vez se aleja más de la raíz. LA solución oscila entre un valor que cada vez se approxima más a cero y otro que tiende hacia $-\infty$. Si se deja aumentar suficientemente el número de iteraciones, llegará un momento en que se producirá un error de desbordamiento.

A diferencia de lo que sucedía en la elección de $g(x) = \ln(x^2)$, en este caso, el algoritmo no oscila entre las dos ramas. Si empezamos en la rama de la derecha, eligiendo un valor positivo para x_0 , el algoritmo diverge llevando las soluciones hacia $+\infty$. Es un resultado esperable, ya que dicha rama no tiene ningún punto fijo.

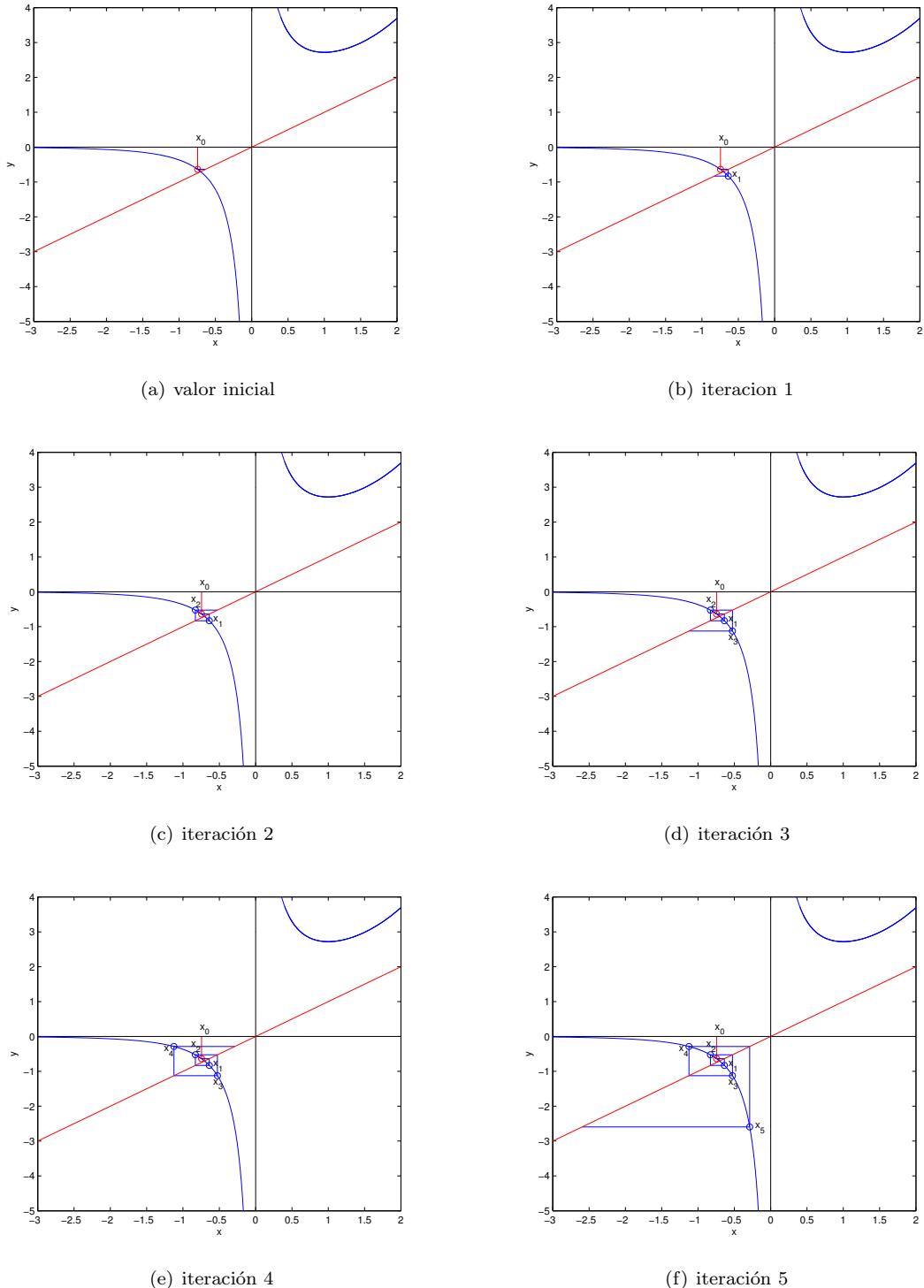


Figura 4.20: proceso de obtención de la raíz de la función $f(x) = e^x - x^2$ aplicando el método del punto fijo sobre la función $g(x) = \frac{e^x}{x}$, el método diverge rápidamente.

4.3. Cálculo de raíces de funciones con Matlab.

Matlab suministra funciones propias para calcular raíces de funciones. Las dividiremos en dos grupos. En primer lugar estudiaremos la función de Matlab `fzero` y después veremos un conjunto de funciones específicas para manejar polinomios.

4.3.1. La función de Matlab `fzero`.

La función `fzero` permite obtener la raíz de una función cualquiera real $f(x) : \mathbb{R} \rightarrow \mathbb{R}$. `fzero`, es una función especial, ya que opera sobre otras funciones, podemos considerarla como una *función de funciones*. Las funciones ordinarias actúan sobre variables, a lo largo de los capítulos anteriores hemos visto como asignar valores y *variables de entrada* a las funciones y también cómo guardar los resultados obtenidos de las funciones en *variables de salida*.

Matlab suministra varios mecanismos, para indicar a `fzero` —y en general a cualquier *función de funciones*— la función sobre la que queremos que actúe. Veremos a continuación algunos de los más comunes.

handle de una función. El primer mecanismo, es asociar a una función un nombre de variable especial. Hasta ahora, siempre hemos empleado las variables para guardar en ellas valores numéricos o caracteres. Sin embargo Matlab permite guardar también funciones en una variable. Estas variables, reciben el nombre de *handles*. Veamos un ejemplo. Si escribimos en la ventana de comandos,

```
>> sn =@sin
sn =
    @sin
```

Matlab asocia a la nueva variable `sn` la función seno (`sin`). Para indicar a Matlab que la nueva variable es el *handle* de una función es imprescindible emplear el símbolo `@`, después del símbolo de asignación `=`.

Si pedimos a Matlab que nos muestre qué variables tiene en el *workspace*,

```
>> whos
  Name      Size            Bytes  Class      Attributes
  sn       1x1              32  function_handle
```

Matlab nos indica que se ha creado una variable `sn`, cuya clase es `function_handle`. Esta variable tiene propiedades muy interesantes: por una parte, podemos manejarla como si se tratara de la función seno, asignando valores de entrada, y guardando el resultado en una variable de salida,

```
>> x=sn(pi/2)
x =
    1
```

Pero además podemos usarla como variable de entrada para otra función, tal y como se muestra en el siguiente código,

```

function pinta_funcion(fun,intervalo)
%Esta función dibuja la gráfica de una función cualquiera (fun) en un
%intervalo dado (intervalo). fun debe ser un handle de la función que se
%quiere pintar. intervalo debe ser un vector que contenga los extremos del
%intervalo que se desea pintar.

%Construimos cien puntos en el intervalo dado,
x=linspace(intervalo(1),intervalo(2),100);

%calculamos el valor de la función en los puntos del intervalo,
y=fun(x);

%dibujamos la gráfica
plot(x,y)

```

La función `pinta_funcion` nos dibujará la gráfica de cualquier función en el intervalo indicado. Para realizarlo bastará crear un *handle* de la función que se quiere dibujar y pasarlo a la función `pinta_fun` como una variable de entrada. Así por ejemplo, si escribimos en Matlab,

```

>>sn=@sin
>>pinta_funcion(sn,[-pi/2,pi/2])

```

Se obtendrá la gráfica de la función seno en el intervalo pedido.

Podemos asignar *handles* no solo a las funciones internas de Matlab sino a cualquier función que escribamos. Por ejemplo, en los métodos descritos más arriba para obtener raíces de funciones, usamos la función $f(x) = e^x - x^2$ como función de prueba. podemos crear un fichero que implemente esta función,

```

function y=prueba(x)
%esta es una funcioncilla de prueba para los algoritmos de obtención de
%raíces
y=exp(x)-x.^2;

```

Si guardamos el fichero, con el nombre `prueba.m` en el directorio de trabajo, podemos ahora asignar un *handle* a nuestra función,

```
mifuncion=@prueba realizar
```

Y a continuación, podemos emplear el programa `pinta_fun` para representar la función en un intervalo, por ejemplo $[-2, 2]$ que contenga la raíz,

```
pinta_funcion(mifuncion,[-2 2])
```

El resultado se muestra en la figura 4.21

la función `feval` de Matlab. Esta función suministra un método indirecto para calcular los resultados de una función cualquiera. Su sintaxis es la siguiente,

```

realizar
[y1, y2, ..., ym]=feval('fun', x1, x2, ..., xn)
```

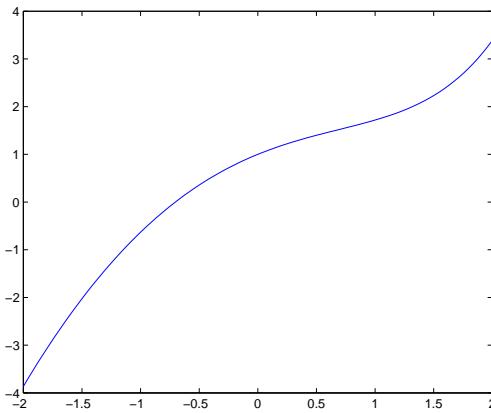


Figura 4.21: Gráfica de la función $f(x) = e^x - x^2$, obtenida mediante `pinta_funcion`.

donde `fun` representa el nombre de la función que se desea evaluar, `x1, x2, ..., xn`, son las variables de entrada empleadas por la función `fun`, `y1, y2, ..., ym` representan sus variables de salida. Es importante destacar que el nombre de la función que se desea evaluar hay que introducirlo entre comillas simples. Así por ejemplo si escribimos,

```
>> y=feval('sin',x)
y =
    1
>>
```

obtenemos el mismo resultado que empleando la función `sin` directamente para calcular el valor del seno de $\pi/2$,

```
>> x=pi/2;
>> y=sin(x)
y =
    1
>>
```

realizar

`feval` suministra un método alternativo al uso de `handles` para crear y manejar *funciones de funciones*. Para ver un ejemplo, el siguiente código muestra una versión alternativa del programa `pinta_funcion`, empleando la función `feval`,

```
function pinta_funcion2(fun,intervalo)
%Esta función dibuja la gráfica de una función cualquiera (fun) en un
%intervalo dado (intervalo). fun debe ser una cadena de caractéres que contengan exactamente el nombre
%que quiere pintar. intervalo debe ser un vector que contenga los extremos del
%intervalo que se desea pintar.

%Construimos cien puntos en el intervalo dado,
x=linspace(intervalo(1),intervalo(2),100);
```

```
%calculamos el valor de la funcion en los puntos del intervalo, EMPLEANDO LA FUNCION feval
y=feval(fun,x);

%dibujamos la gráfica
plot(x,y)
```

Para representar la función seno en el intervalo $-\frac{\pi}{2}, \frac{\pi}{2}$, empleando esta nueva función, introducimos en Matlab,

```
>> pinta_funcion2('sin',[-pi/2,pi/2])
```

realizar También podemos crear una variable alfa-numérica con el nombre de la función seno, y pasar directamente la variable creada,

```
>> funcion='sin'
>> pinta_funcion2(funcion,[-pi/2,pi/2])
```

Al igual que en el caso del uso de `handles` podemos emplear la función `feval` con funciones creadas por el usuario, por ejemplo podemos representar nuestra función `prueba`, introducida anteriormente,

```
>>pinta_funcion2('prueba', [-2 2])
```

El resultado sería de nuevo la figura 4.21. Una última propiedad importante de la función `feval` es que también admite que indiquemos la función a evaluar mediante un *handle*. Sí volvemos al último ejemplo, podríamos haber construido un *handle* para la función `prueba`,

```
>>mf=@prueba
>>pinta_funcion2(mf, [-2 2])
```

Obtendríamos una vez más el mismo resultado.

Funciones *inline*. Las funciones *inlin* realizar e suministran un tercer mecanismo en Matlab para manejar una función de modo que sirva de *variable* a otra función. Las funciones *inline* tienen una peculiaridad con respecto a las funciones que hemos visto hasta ahora; no se guardan en ficheros .m sino directamente en el *Workspace*. Las funciones *inline* solo existen mientras dura la sesión de Matlab en que se crearon, aunque es posible guardarlas en ficheros .mat y volver a cargarlas en Matlab, como se haría con cualquier otra variable.

Para crear una función *inline* se emplea el comando `inline`. En su forma más sencilla, el comando debe emplear como variable de entrada una expresión entre comillas simples que represente la expresión matemática de la función. Por ejemplo si queremos hacer una versión *inline* de la función `prueba`,

```
>> fun=inline('exp(x)-x.^2')
fun =
    Inline function:
    fun(x) = exp(x)-x^2
```

Para calcular el valor de la función en un punto, la función *inline* se maneja de modo análogo a cualquier otra función ordinaria.

```
>> y=fun(2)
y =
    3.3891
```

Como en el caso del uso de `handles`, la variable creada mediante una función *inline* realizar , hace referencia a una función y puede ser empleada como variable de entrada por otras funciones. Por ejemplo, podríamos emplear directamente nuestra primera versión del programa para pintar funciones, `pitan_funcion` para obtener la gráfica de nuestra función de prueba $f(x) = e^x - x^2$,

```
>>funcion=inline('exp(x)-x.^2')
>>pinta_funcion(funcion,[-2 2])
```

Una vez que hemos visto distintos métodos para manejar una función como variable de entrada de otra función, volvamos a la función `fzero`. En su forma más sencilla, `fzero` admite como variable de entrada, una función expresada mediante un `handle`, mediante su nombre escrito entrecomilladas o bien construida como función *inline*. Además es preciso introducir una segunda variable que puede ser un punto x_0 próximo a la raíz de la función o bien un vector $[ab]$ que defina un intervalo que contenga una raíz. La función `fzero`, devuelve como variable de salida el valor aproximado de la raíz. Si `fzero` no es capaz de encontrar la raíz de la función, devolverá NaN. Veamos un ejemplo con la función contenida en el fichero, `prueba.m`, descrito más arriba,

1. Empleando un *handle* y un punto próximo a la raíz,

```
>> hndl=@prueba
hndl =
@prueba
>> raiz=fzero(hndl,2)
raiz =
-0.703467422498392
```

2. Empleando un *handle* y un intervalo que contenga la raíz,

```
>> raiz=fzero(hndl,[-2 2])
raiz =
-0.703467422498392
```

3. Empleando el nombre de la función entre comillas y un punto cercano a la raíz,

```
>> raiz=fzero('prueba',2)
raiz =
-0.703467422498392
```

4. Empleando el nombre de la función entre comillas y un intervalo que contenga la raíz,

```
>> raiz=fzero('prueba',[-2 2])
raiz =
-0.703467422498392
```

5. Usando una función *inline* y un punto cercano a la raíz,

```
realizar
>> finl=inline('exp(x)-x.^2')
finl =
Inline function:
finl(x) = exp(x)-x.^2
```

```
>> raiz=fzero(fnl,2)
raiz =
-0.703467422498392
```

6. Usando una función *inline* y un intervalo que contenga la raíz,

```
>> raiz=fzero(fnl,[-2 2])
raiz =
-0.703467422498392
```

La función **fzero**, tiene muchas posibilidades de ajuste de la precisión, del método que emplea internamente para buscar la raíz, etc. Para obtener una visión mas completa de su uso, consultar la ayuda de Matlab.

4.3.2. Cálculo de raíces de polinomios.

Matlab tiene un conjunto de funciones especialmente pensadas para manejar polinomios. En primer lugar, en Matlab es habitual representar los polinomios mediante vectores cuyos elementos, son los coeficientes del polinomio ordenados de mayor a menor grado. Así por ejemplo, el polinomio. $y = 2x^3 + 3x^2 + 4x + 1$ se representa mediante el vector, $p1 = [2 3 4 1]$, el polinomio $y = 3x^4 + 2x^2 + 6x$ se representa mediante el vector, $p2 = [3 0 2 6 0]$ y, en general, el polinomio $y(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0$ se representa mediante el vector $p = [a_n a_{n-1} \dots a_2 a_1 a_0]$. Si al polinomio le falta algún o algunos términos, el elemento correspondiente toma el valor 0 en el vector que representa el polinomio.

Veamos a continuación un conjunto de funciones de Matlab, especialmente pensadas para manejar polinomios,

La función roots. Esta función calcula las raíces de un polinomio de grado n a partir de los coeficientes del polinomio, contenidos en un vector como los que acabamos de describir. La sintaxis es: **raices=roots([vector de coeficientes])**. veamos un ejemplo. Dado el polinomio $y(x) = x^3 - 6^2 + 11x - 6$ lo expresaríamos en Matlab como,

```
>> p=[1 -6 11 -6]
```

Y obtendríamos sus raíces como,

```
>> raices=roots(p)
raices =
3.0000
2.0000
1.0000
```

Matlab devuelve todas las raíces del polinomio en un único vector, tanto las reales como las complejas. Como por ejemplo en el caso del polinomio $y(x) = x^2 + 2x + 1$

```
>> p=[1 2 3]
p =
1      2      3
```

```
>> raices=roots(p)
raices =
-1.0000 + 1.4142i
-1.0000 - 1.4142i
```

la función poly. Esta función podría considerarse la opuesta a la anterior; dado un vector que contiene las raíces de un polinomio, nos devuelve los coeficientes del polinomio correspondiente, Por ejemplo si definimos el vector de raíces,

```
>>raices=[3 2 1]
```

podemos obtener los coeficientes del polinomio que posee esas raíces como,

```
>> raices=[1 2 3]
raices =
    1      2      3
>> coef=poly(raices)
coef =
    1     -6     11     -6
```

Es decir, las raíces pertenecen al polinomio, $y(x) = x^3 - 6x^2 + 11x - 6$.

la función polyval. Esta función calcula el valor de un polinomio en un punto. Para ello es preciso darle un vector con los coeficientes del polinomio —definido igual que en los casos anteriores— y un segundo vector con los puntos para los que se quiere calcular el valor del polinomio,

```
>> coef=[1 2 3 4]
coef =
    1      2      3      4
>> x=2
x =
    2
>>y= polyval(coef,x)
y =
    26
>> x=[1:10]
x =
    1      2      3      4      5      6      7      8      9      10
>> y=polyval(coef,x)
y =
    Columns 1 through 6
        10          26          58         112         194         310
    Columns 7 through 10
        466         668         922        1234
```

En este ejemplo se ha obtenido con `polyval` el valor del polinomio $y(x) = x^3 + 2x^2 + 2x + 4$ primero para el punto $x = 2$ y después para los puntos $x = [1 2 3 4 5 6 7 8 9 10]$.

La función conv. Calcula el producto de dos polinomios. Cada polinomio se representa mediante un vector de coeficientes y la función `conv` devuelve un vector con los coeficientes del polinomio producto. Por ejemplo, si multiplicamos el polinomio $y_1 = x + 2$ por el polinomio $y_2 = x - 1$ obtendremos como resultado, $p = y_1 \cdot y_2 = x^2 + x - 2$,

```
>> y1=[1 2]
y1 =
    1     2
>> y2=[1 -1]
y2 =
    1    -1
>> p=conv(y1,y2)
p =
    1     1    -2
```


Capítulo 5

Aplicaciones del cálculo científico al álgebra lineal

5.1. Matrices y vectores

En esta sección vamos a repasar algunos conceptos fundamentales de álgebra lineal y cómo pueden manejarse empleando Matlab. No daremos definiciones precisas ni tampoco demostraciones, ya que tanto unas como otras se verán en detalle en la asignatura de álgebra.

matrices. Desde un punto de vista funcional definiremos una matriz como una tabla bidimensional de números ordenados en filas y columnas,

$$A = \begin{pmatrix} 1 & \sqrt{(2)} & 3,5 & 0 \\ -2 & \pi & -4,6 & 4 \\ 7 & -19 & 2,8 & 0,6 \end{pmatrix}$$

Cada línea horizontal de números constituye una *fila* de la matriz y cada línea horizontal una *columna* de la misma.

A una matriz con m filas y n columnas se la denomina matriz de orden $m \times n$. m y n son las dimensiones de la matriz y se dan siempre en el mismo orden: primero el número de filas y después el de columnas. Así, la matriz A del ejemplo anterior es una matriz 3×4 . El orden de una matriz expresa el tamaño de la matriz.

Dos matrices son iguales si tienen el mismo orden, y los elementos que ocupan en ambas matrices los mismo lugares son iguales.

Una matriz es cuadrada, si tiene el mismo número de filas que de columnas. Es decir es de orden $n \times n$.

Mientras no se diga expresamente lo contrario, emplearemos letras mayúsculas A, B, \dots para representar matrices. La expresión $A_{m \times n}$ indica que la matriz A tiene dimensiones $m \times n$. Para denotar los elementos de una matriz, emplearemos la misma letra en minúsculas empleada para nombrar la matriz, indicando mediante subíndices, y siempre por este orden, la fila y la columna a la que pertenece el elemento. Así por ejemplo a_{ij} representa al elemento de la matriz A , que ocupa la fila i y la columna j .

$$A = \begin{pmatrix} 1 & \sqrt{(2)} & 3,5 & 0 \\ -2 & \pi & -4,6 & 4 \\ 7 & -19 & 2,8 & 0,6 \end{pmatrix} \rightarrow a_{23} = -4,6$$

vectores A una matriz compuesta por una sola fila, la denominaremos vector fila. A una matriz compuesta por una sola columna la denominaremos vector columna. Siempre que hablamos de un vector, sin especificar más, entenderemos que se trata de un vector columna.¹ Para representar vectores, emplearemos letras minúsculas. Para representar sus elementos añadiremos a la letra que representa al vector un subíndice indicando la fila a la que pertenece el elemento.

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_i \\ \vdots \\ a_n \end{pmatrix}$$

Podemos asociar los puntos del plano con los vectores de dimensión dos. Para ello, usamos una representación cartesiana, en la que los elementos del vector son los valores de las coordenadas (x, y) del punto del plano que representan. Cada vector se representa gráficamente mediante una flecha que parte del origen de coordenadas y termina en el punto (x, y) representado por el vector. La figura 5.1 representa los vectores,

$$a = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, b = \begin{pmatrix} 2 \\ -3 \end{pmatrix}, c = \begin{pmatrix} 0 \\ -2 \end{pmatrix}$$

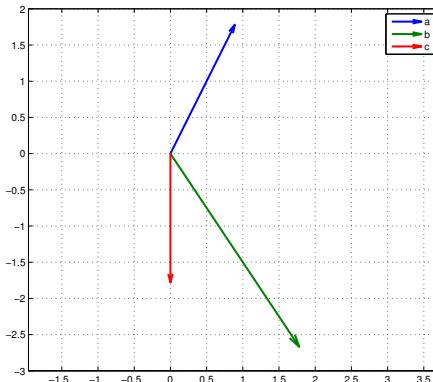


Figura 5.1: Representación gráfica de vectores en el plano

De modo análogo, podemos asociar vectores de dimensión tres con puntos en el espacio tridimensional. En este caso, los valores de los elementos del vector corresponden con la coordenadas (x, y, z) de los puntos en el espacio. La figura 5.2 muestra la representación gráfica en espacio tridimensional de los vectores,

$$a = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, b = \begin{pmatrix} 2 \\ -3 \\ -1 \end{pmatrix}, c = \begin{pmatrix} 0 \\ -2 \\ 1 \end{pmatrix}$$

¹Esta identificación de los vectores como vectores columna no es general. La introducimos porque simplifica las explicaciones posteriores.

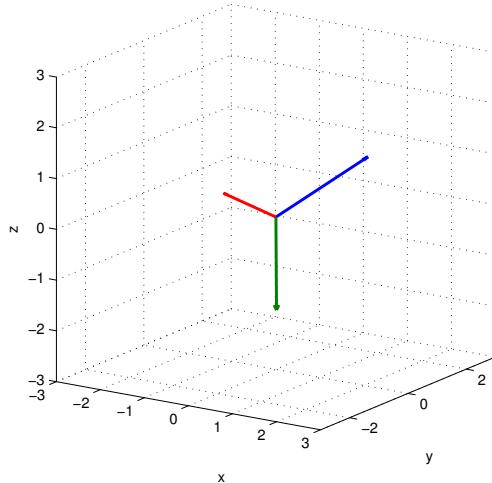


Figura 5.2: Representación gráfica de vectores en el espacio

Evidentemente para vectores de mayor dimensión, no es posible obtener una representación gráfica. Si embargo muchas de las propiedades geométricas, observables en los vectores bi y tridimensionales, pueden extrapolarse a vectores de cualquier dimensión.

5.2. Operaciones matriciales

A continuación definiremos las operaciones matemáticas más comunes, definidas sobre matrices.

suma. La suma de dos matrices, se define como la matriz resultante de sumar los elementos que ocupan en ambas la misma posición. Solo está definida para matrices del mismo orden,

$$C = A + B$$

$$c_{ij} = a_{ij} + b_{ij}$$

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

La suma de matrices cumple las siguientes propiedades,

1. Asociativa: $(A + B) + C = A + (B + C)$
2. Conmutativa: $A + B = B + A$
3. Elemento neutro: $O_{n \times m} + A_{n \times m} = A_{n \times m}$ El elemento neutro $O_{n \times m}$ de la suma de matrices de orden $n \times m$ es la matriz nula de dicho orden, —compuesta exclusivamente por ceros— .

En Matlab, podemos crear una matriz de cualquier orden, compuesta exclusivamente por ceros mediante el comando `zeros(m,n)`, donde m es el número de filas y n el de columnas de la matriz de ceros resultante,

```
>> O=zeros(2,3)
O =
    0     0     0
    0     0     0
>> A=[1 2 3; 4 3 6]
A =
    1     2     3
    4     3     6
>> B=A+O
B =
    1     2     3
    4     3     6
>>
```

4. Elemento opuesto: La opuesta a una matriz se obtiene cambiando de signo todos sus elementos, $A_{op} = -A$

```
>> A
A =
    1     2     3
    4     3     6
>> Aop=-A
Aop =
   -1    -2    -3
   -4    -3    -6
>> S=A+Aop
S =
    0     0     0
    0     0     0
```

En Matlab el signo + representa por defecto la suma de matrices, por lo que la suma de dos matrices puede obtenerse directamente como,

```
>> A=[1 2 3; 4 3 6]
A =
    1     2     3
    4     3     6
>> B=[1 2 3; 4 -3 2]
B =
    1     2     3
    4    -3     2
>> S=A+B
S =
    2     4     6
    8     0     8
```

Transposición Dada una matriz A , su transpuesta A^T se define como la matriz que se obtiene intercambiando sus filas con sus columnas,

$$A \rightarrow A^T$$

$$a_{ij} \rightarrow a_{ji}$$

$$A = \begin{pmatrix} 1 & -3 & 2 \\ 2 & 7 & -1 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 1 & 2 \\ -3 & 7 \\ 2 & -1 \end{pmatrix}$$

En Matlab, la operación de transposición se indica mediante un apóstrofo ('),

```
>> A=[1 2 3; 4 3 6]
A =
    1     2     3
    4     3     6
>> B=A'
B =
    1     4
    2     3
    3     6
```

Para vectores, la transposición convierte un vector fila en un vector columna y viceversa.

$$a \rightarrow a^T$$

$$a = \begin{pmatrix} 1 \\ -3 \\ 2 \end{pmatrix} \rightarrow a^T = (1 \quad -3 \quad 2)$$

Una matriz cuadrada se dice que es simétrica si coincide con su transpuesta,

$$A = A^T$$

$$a_{ij} = a_{ji}$$

$$A = A^T = \begin{pmatrix} 1 & 3 & -3 \\ 3 & 0 & -2 \\ -3 & -2 & 4 \end{pmatrix}$$

Una matriz cuadrada es antisimétrica cuando cumple que $A = -A^T$. Cualquier matriz cuadrada se puede descomponer en la suma de una matriz simétrica más otra antisimétrica.

La parte simétrica puede definirse como,

$$A_S = \frac{1}{2} (A + A^T)$$

y la parte antisimétrica como,

$$A_A = \frac{1}{2} (A - A^T)$$

Así, por ejemplo,

$$A = A_S + A_A \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 5 \\ 3 & 5 & 7 \\ 5 & 7 & 9 \end{pmatrix} + \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{pmatrix}$$

Por último, la transpuesta de la suma de matrices cumple,

$$(A + B)^T = A^T + B^T$$

Producto de una matriz por un escalar. El producto de una matriz A por un número b es una matriz del mismo orden que A , cuyos elementos se obtienen multiplicando los elementos de A por el número b ,

$$C = b \cdot A \rightarrow c_{ij} = b \cdot a_{ij}$$

$$3 \cdot \begin{pmatrix} 1 & -2 & 0 \\ 2 & 3 & -1 \end{pmatrix} = \begin{pmatrix} 3 & -6 & 0 \\ 6 & 9 & -3 \end{pmatrix}$$

En Matlab, el símbolo $*$ se emplea para representar el producto entre escalares (números), entre escalares y matrices y el producto entre matrices, como veremos en los siguientes párrafos.

Producto escalar de dos vectores. Dados vectores de la misma dimensión m se define su producto escalar como,

$$a \cdot b = \sum_{i=1}^n a_i b_i$$

$$\begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -2 \\ 0 \end{pmatrix} = 1 \cdot 1 + 3 \cdot (-2) + 4 \cdot 0 = -5$$

El resultado de producto escalar de dos vectores, es siempre un número; se multiplican los entre sí los elementos de los vectores que ocupan idénticas posiciones y se suman los productos resultantes.

Producto matricial El producto de una matriz de orden $n \times m$ por una matriz $m \times l$, es una nueva matriz de orden $n \times l$, cuyos elementos se obtiene de acuerdo con la siguiente expresión,

$$P = A \cdot B \rightarrow a_{ij} = \sum_{t=1}^m a_{it} b_{tj}$$

Por tanto, el elemento de la matriz producto que ocupa la fila i y la columna j , se obtiene multiplicando por orden los elementos de la fila i de la matriz A con los elementos correspondientes de la columna j de la matriz B , y sumando los productos resultantes.

Para que dos matrices puedan multiplicarse es imprescindible que el número de columnas de la primera matriz coincida con el número de filas de la segunda.

Podemos entender la mecánica del producto de matrices de una manera más fácil si consideramos la primera matriz como un grupo de vectores fila,

$$A_1 = (a_{11} \ a_{12} \ \cdots a_{1n})$$

$$A_2 = (a_{21} \ a_{22} \ \cdots a_{2n})$$

$$\vdots$$

$$A_m = (a_{m1} \ a_{m2} \ \cdots a_{mn})$$

$$\rightarrow A = \begin{pmatrix} a_{11} & a_{12} & \cdots a_{1n} \\ a_{21} & a_{22} & \cdots a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots a_{mn} \end{pmatrix}$$

y la segunda matriz como un grupo de vectores columna,

$$B_1 = \begin{pmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{pmatrix} B_2 = \begin{pmatrix} b_{12} \\ b_{22} \\ \vdots \\ b_{n2} \end{pmatrix} \cdots B_3 = \begin{pmatrix} b_{1m} \\ b_{2m} \\ \vdots \\ b_{nm} \end{pmatrix} \rightarrow B = \begin{pmatrix} b_{11} & b_{12} & \cdots b_{1n} \\ b_{21} & b_{22} & \cdots b_{2n} \\ \vdots & \vdots & \cdots \\ b_{m1} & b_{m2} & \cdots b_{mn} \end{pmatrix}$$

Podemos ahora considerar cada elemento p_{ij} de la matriz producto $P = A \cdot B$ como el producto escalar del vector fila A_i por el vector columna B_j , $p_{ij} = A_i \cdot B_j$. Es ahora relativamente fácil, deducir algunas de las propiedades del producto matricial,

1. Para que dos matrices puedan multiplicarse, es preciso que el número de columnas de la primera coincida con el número de filas de la segunda. Además la matriz producto tiene tantas filas como la primera matriz y tantas columnas como la segunda.
2. El producto matricial no es commutativo. En general $A \cdot B \neq B \cdot A$
3. $(A \cdot B)^T = B^T \cdot A^T$

Matriz identidad La matriz identidad de orden $n \times n$ se define como:

$$I_n = \begin{cases} i_{ll} = 1 \\ i_{kj} = 0, \quad k \neq j \end{cases}$$

Es decir, una matriz en la que todos los elementos que no pertenecen a la diagonal principal son 0 y los elementos de la diagonal principal son 1. Por ejemplo,

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

La matriz identidad I_n es el elemento neutro del producto de matrices cuadradas de orden $n \times n$,

$$A_{n \times n} \cdot I_n = I_n \cdot A_{n \times n}$$

Además,

$$\begin{aligned} A_{n \times m} \cdot I_m &= A_{n \times m} \\ I_n \cdot A_{n \times m} &= A_{n \times m} \end{aligned}$$

En Matlab se emplea el comando `eye(n)` para construir la matriz identidad de orden $n \times n$,

```
>> I4=eye(4)
I4 =
    1     0     0     0
    0     1     0     0
    0     0     1     0
    0     0     0     1
```

Una matriz cuadrada se dice que es ortogonal si cumple,

$$A^T \cdot A = I$$

Producto escalar de dos vectores y producto matricial Por conveniencia, representaremos el producto escalar de dos vectores como un producto matricial,

$$a \cdot b = a^T b = b^T a = b \cdot a$$

$$(1 \ 3 \ 4) \begin{pmatrix} 1 \\ -2 \\ 0 \end{pmatrix} = (1 \ -1 \ 0) \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix} = 1 \cdot 1 + 3 \cdot (-2) + 4 \cdot 0 = -5$$

Es decir, transponemos el primer vector del producto, convirtiéndolo en un vector fila.

Norma de un vector. La longitud euclídea, módulo, norma 2 o simplemente norma de un vector se define como,

$$\|x\|_2 = \|x\| = \sqrt{x \cdot x} = \sqrt{x^T x} = \sqrt{x_1^2 + x_2^2 + \cdots x_n^2} = \left(\sum_{i=1}^n x_i^2 \right)^{\frac{1}{2}}$$

Constituye la manera usual de medir la longitud de un vector. Tiene una interpretación geométrica inmediata a través del teorema de Pitágoras: nos da la longitud del segmento que representa al vector. La figura 5.3 muestra dicha interpretación, para un vector bidimensional.

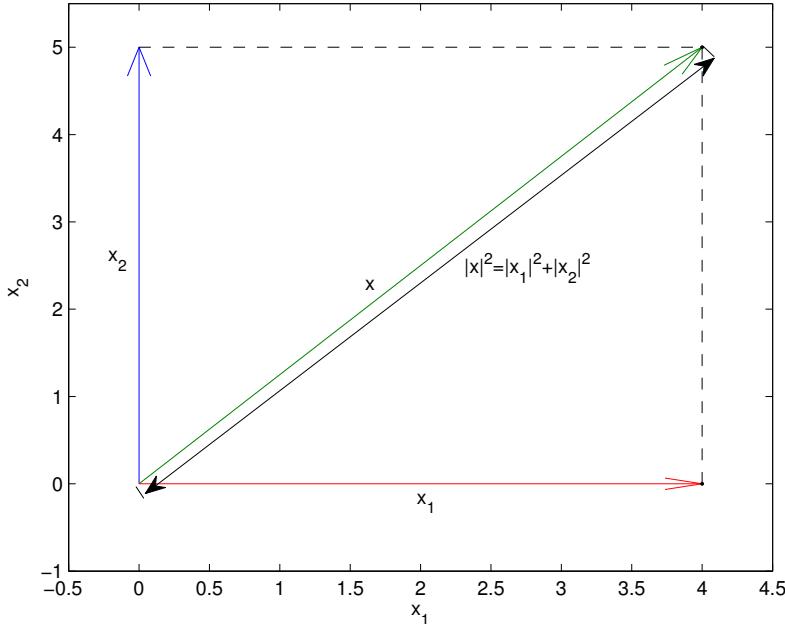


Figura 5.3: interpretación geométrica de la norma de un vector

A partir de la norma de un vector es posible obtener una expresión alternativa para el producto escalar de dos vectores,

$$a \cdot b = \|a\| \|b\| \cos \theta$$

Donde θ representa el ángulo formado por los dos vectores.

Aunque se trate de la manera más común de definir la norma de un vector, la norma 2 no es la única definición posible,

- Norma 1: Se define como la suma de los valores absolutos de los elementos de un vector,

$$\|x\|_1 = |x_1| + |x_2| + \cdots + |x_n|$$

- Norma p, Es una generalización de la norma 2,

$$\|x\|_p = \sqrt[p]{|x_1^p| + |x_2^p| + \cdots + |x_n^p|} = \left(\sum_{i=1}^n |x_i^p| \right)^{\frac{1}{p}}$$

- norma ∞ , se define como el mayor elemento del vector valor absoluto,

$$\|x\|_\infty = \max \{|x_i|\}$$

- Norma $-\infty$, el menor elemento del vector en valor absoluto,

$$\|x\|_{-\infty} = \min \{|x_i|\}$$

En Matlab la norma de un vector puede obtenerse mediante el comando `norm(v,p)` La primera variable de entrada debe ser un vector y la segunda el tipo de norma que se desea calcular. Si se omite la segunda variable de entrada, el comando devuelve la norma 2. A continuación se incluyen varios ejemplo de utilización,

```
>> x=[1;2;-3;0;-1]
x =
    1
    2
   -3
    0
   -1
>> norma_2=norm(x,2)
norma_2 =
    3.872983346207417e+00
>> norma=norm(x)
norma =
    3.872983346207417e+00
>> norma_1=norm(x,1)
norma_1 =
    7
>> norma_4=norm(x,4)
norma_4 =
    3.154342145529904e+00
>> norma_inf=norm(x,inf)
norma_inf =
    3
>> norma_minf=norm(x,-inf)
norma_minf =
    0
```

En general, una norma se define como una función de $\mathbb{R}^n \rightarrow \mathbb{R}$, que cumple,

$$\begin{aligned}\|x\| &\geq 0, \|x\| = 0 \Rightarrow x = 0 \\ \|x + y\| &\leq \|x\| + \|y\| \\ \|\alpha x\| &= |\alpha| \|x\|, \alpha \in \mathbb{R}\end{aligned}$$

Llamaremos vectores unitarios u , a aquellos para los que se cumple que $\|u\| = 1$.

Dos vectores a y b son ortogonales si cumplen que su producto escalar es nulo, $a^T b = 0 \Rightarrow a \perp b$.

Si además ambos vectores tienen módulo unidad, se dice entonces que los vectores son ortonormales. Desde el punto de vista de su representación geométrica, dos vectores ortogonales, forman entre sí un ángulo recto.

Traza de una matriz. La traza de una matriz cuadrada, se define como la suma de los elementos que ocupan su diagonal principal,

$$\begin{aligned}Tr(A) &= \sum_{i=1}^n a_{ii} \\ Tr \left(\begin{pmatrix} 1 & 4 & 4 \\ 2 & -2 & 2 \\ 0 & 3 & 6 \end{pmatrix} \right) &= 1 - 2 + 6 = 5\end{aligned}$$

La traza de la suma de dos matrices cuadradas A y B del mismo orden, coincide con la suma de las trazas de A y B ,

$$tr(A + B) = tr(A) + tr(B)$$

Dada una matriz A de dimensión $m \times n$ y una matriz B de dimensión $n \times m$, se cumple que,

$$tr(AB) = tr(BA)$$

En Matlab, puede obtenerse directamente el valor de la traza de una matriz, mediante el comando **trace**,

```
>> A=[1 3 4
      3 5 2
      2 -1 -2]
A =
      1      3      4
      3      5      2
      2     -1     -2
>> t=trace(A)
t =
      4
```

Determinante de una matriz. El determinante de una matriz A , se representa habitualmente como $|A|$ o, en ocasiones como $\det(A)$. Para poder definir el determinante de una matriz, necesitamos antes introducir una serie de conceptos previos. En primer lugar, si consideramos un escalar como una matriz de un solo elemento, el determinante sería precisamente el valor de ese único elemento,

$$A = (a_{11}) \rightarrow |A| = a_{11}$$

Se denomina menor complementario o simplemente menor, M_{ij} del elemento a_{ij} de una matriz A , a la matriz que resulta de eliminar de la matriz A la fila i y la columna j a las que pertenece el elemento a_{ij} . Por ejemplo,

$$A = \begin{pmatrix} 1 & 0 & -2 \\ 3 & -2 & 3 \\ 0 & 6 & 5 \end{pmatrix}, M_{23} = \begin{pmatrix} 1 & 0 \\ 0 & 6 \end{pmatrix}$$

$$M_{32} = \begin{pmatrix} 1 & -2 \\ 3 & 3 \end{pmatrix}, M_{33} = \begin{pmatrix} 1 & 0 \\ 3 & -2 \end{pmatrix} \dots$$

El cofactor C_{ij} de un elemento a_{ij} de la matriz A , se define a partir del determinante del menor complementario del elemento a_{ij} como,

$$C_{ij} = (-1)^{i+j} |M_{ij}|$$

Podemos ahora definir el determinante de una matriz A cuadrada de orden n , empleando la fórmula de Laplace,

$$|A| = \sum_{j=1}^n a_{ij} C_{ij}$$

o alternativamente,

$$|A| = \sum_{i=1}^n a_{ij} C_{ij}$$

En el primer caso, se dice que se ha desarrollado el determinante a lo largo de la fila i . En el segundo caso, se dice que se ha desarrollado el determinante a lo largo de la columna j .

La fórmula de Laplace, obtiene el determinante de una matriz de orden $n \times n$ a partir del cálculo de los determinantes de los menores complementarios de los elementos de una fila; n matrices de orden $(n-1) \times (n-1)$. A su vez, podríamos calcular el determinante de cada menor complementario, aplicando la fórmula de Laplace y así sucesivamente hasta llegar a matrices de orden 2×2 . Para una matriz 2×2 , si desarrollamos por la primera fila obtenemos su determinante como,

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$|A| = \sum_{j=1}^2 a_{1j} C_{1j} = a_{11} C_{11} + a_{12} C_{12}$$

$$= a_{11}(-1)^{1+1} |M_{11}| + a_{12}(-1)^{1+2} |M_{12}|$$

$$= -a_{11}a_{22} + a_{12}a_{21}$$

y si desarrollamos por la segunda columna,

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

$$|A| = \sum_{j=1}^2 a_{i2} C_{i2} = a_{12} C_{12} + a_{22} C_{22}$$

$$= a_{12}(-1)^{1+2} |M_{12}| + a_{22}(-1)^{2+2} |M_{22}|$$

$$= -a_{12}a_{21} + a_{22}a_{12}$$

Para una matriz de dimensión arbitraria $n \times n$, el determinante se obtiene aplicando recursivamente la fórmula de Laplace,

$$|A| = \sum_{j=1}^n a_{ij} C_{ij} = \sum_{j=1}^n a_{ij} (-1)^{i+j} \left| M_{ij}^{(n-1) \times (n-1)} \right|$$

$$\left| M_{ij}^{(n-1) \times (n-1)} \right| = \sum_{k=1}^{n-1} m_{lk} C_{lk} = \sum_{k=1}^{n-1} m_{lk} (-1)^{l+k} \left| M_{lk}^{(n-2) \times (n-2)} \right|$$

$$\vdots$$

$$\left| M_{st}^{1 \times 1} \right| = (-1)^{s+t} m_{st}$$

Así, por ejemplo, podemos calcular el determinante de la matriz,

$$A = \begin{pmatrix} 1 & 0 & -2 \\ 3 & -2 & 3 \\ 0 & 6 & 5 \end{pmatrix}$$

desarrollándolo por los elementos de la primera columna, como,

$$|A| = 1 \cdot (-1)^2 \cdot \begin{vmatrix} -2 & 3 \\ 6 & 5 \end{vmatrix} + 3 \cdot (-1)^3 \cdot \begin{vmatrix} 0 & -2 \\ 6 & 5 \end{vmatrix} + 0 \cdot (-1)^4 \cdot \begin{vmatrix} 0 & -2 \\ -2 & -3 \end{vmatrix}$$

$$= 1 \cdot (-1)^2 \cdot [(-2) \cdot 5 - 6 \cdot 3] + 3 \cdot (-1)^3 \cdot [0 \cdot 5 - 6 \cdot (-2)] + 0 \cdot (-1)^4 \cdot [0 \cdot 3 - (-2) \cdot (-2)] = -64$$

Podemos programar en Matlab una función recurrente que calcule el determinante de una matriz de rango $n \times n$. (El método no es especialmente eficiente pero ilustra el uso de funciones recursivas.

```
function d=determinante(M)
%este programa, calcula el determinante de una matriz empleando la formula
%de Laplace. La función es recursiva, (se llama a si misma sucesivamente
%para calcular los cofactores necesarios). Desarrolla siempre por los
%elementos de la primera columna. (Es un prodigo de ineficiencia numerica,
%pero permite manejar bucles y funciones recursivas, asi que supongo que
%puede ser útil para los que empiezan a programar).
%un posible ejercicio para ver lo malo que es el método, consiste ir
%aumentando la dimension de la matriz y comparar lo que lo tarde en
%calcular el determinante con lo que tarda la función de Matlab det...
```

```
%primero comprobamos que la matriz suministrada es cuadrada:
d=0;
[a,b]=size(M);
if a~=b
    disp('la matriz no es cuadrada, Campeón')
    d=[];
else
    for i=1:a
        if a==1
            d=M;
        else
            %Eliminamos la fila y columna que toque
            N=M([1:i-1 i+1:a],2:b);
            %Añadimos el calculo correspondiente al cofactor
            d=(-1)^(i+1)*M(i,1)*determinante(N)+d;
            %pause
        end
    end
end
```

En Matlab, el determinante de una matriz se calcula directamente empleando la función `det`. Así, para calcular el determinante de la matriz A del ejemplo anterior,

```
>> A=[1 0 -2; 3 -2 3; 0 6 5]
A =
1     0     -2
3     -2      3
0      6      5
>> da=det(A)
da =
-64
```

Entre las propiedades de los determinantes, destacaremos las siguientes,

1. El determinante del producto de un escalar a por una matriz A de dimensión $n \times n$ cumple,

$$|a \cdot A| = a^n \cdot |A|$$

2. El determinante de una matriz es igual al de su traspuesta,

$$|A| = |A^T|$$

3. El determinante del producto de dos matrices es igual al producto de los determinantes,

$$|A_{n \times n} \cdot B_{n \times n}| = |A_{n \times n}| \cdot |B_{n \times n}|$$

Una matriz es singular si su determinante es cero.

El rango de una matriz se define como el tamaño de la submatriz más grande dentro de A , cuyo determinante es distinto de cero. Así por ejemplo la matriz,

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \rightarrow |A| = 0$$

Es una matriz singular y su rango es dos,

$$\begin{vmatrix} 1 & 2 \\ 4 & 5 \end{vmatrix} = -3 \neq 0 \Rightarrow r(A) = 2$$

Para una matriz no singular, su rango coincide con su orden.

En Matlab se puede estimar el rango de una matriz mediante el comando `rank`,

```
>> A=[1 2 3
4 5 6
7 8 9]
A =
1     2     3
4     5     6
7     8     9
>> r=rank(A)
r =
2
```

Inversión. Dada una matriz cuadrada no singular A existe una única matriz A^{-1} que cumple,

$$A_{n \times n} \cdot A_{n \times n}^{-1} = I_{n \times n}$$

La matriz A^{-1} recibe el nombre de matriz inversa de A , y puede calcularse a partir de A como,

$$A^{-1} = \frac{1}{|A|} [\text{adj}(A)]^T$$

Donde $\text{adj}(A)$ es la matriz adjunta de A , que se obtiene sustituyendo cada elemento a_{ij} de A , por su cofactor C_{ij} . A continuación incluimos el código en Matlab de una función `inversa` que calcula la inversa de una matriz. La función `inversa` llama a su vez a la función `determinante` descrita más arriba. Lo ideal es crear un fichero `inversa.m` que incluya las dos funciones una detrás de la otra tal y como aparecen escritas a continuación. De este modo, si llamamos desde el `workspace` de Matlab a la función `inversa`, esta encuentra siempre el código de `determinante` ya que está contenido en el mismo fichero.

```
function B=inversa(A)
%este programa calcula la inversa de una matriz a partir de definición
%típica: A^(-1)=[adj(A)]'/det(A)
%Se ha incluido al final del programa una función (determinante) para
%calcular determinantes
%Todo el programa es MUY INEFICIENTE. El único interés de esto es enseñar que
%las estructuras básicas de programación funcionan, y como se manejan las
%llamadas a funciones en Matlab etc.
```

```
%Lo primero que hacemos es comprobar si la matriz es cuadrada
%primero comprobamos que la matriz suministrada es cuadrada:
[a,b]=size(A);
if a~=b
    disp('la matriz no es cuadrada, Campeón')
    B=[];
else
    %calculamos el determinante de A, si es cero hemos terminado
    dA=determinante(A)
    if dA==0
        %deberíamos condicionar en lugar de comparar con cero, los errores
        %de redondeo pueden matarnos.... Si el determinante es proximo a
        %cero
        disp('la matriz es singular, la pobre')
        B=[]
    else
        %Calculamos el cofactor de cada término de A mediante un doble bucle.
        for i=1:a
            for j=1:b
                %Construimos el menor correspondiente al elemento (i,j)
                m=A([1:i-1 i+1:a],[1:j-1 j+1:b])
                %calculamos el cofactor llamando a la función determinante
                %lo ponemos ya en la posición que corresponderia a la matriz
                %transpuesta de la adjunta.
                B(j,i)=(-1)^(i+j)*determinante(m)
            end
        end
        %Terminamos la operacion dividiendo por el determinante de A
        B=B/dA
    end
end
%%%%%
%Aquí incluimos la función determinante, así la función inversa, no tiene
%que ir a buscarla a ningún sitio ya que esta incluida en su mismo %fichero
%%%%%
function d=determinante(M)
%este programa, calcula el determinante de una matriz empleando la fórmula
%de Laplace. La función es recursiva, (se llama a si misma sucesivamente
%para calcular los cofactores necesarios). Desarrolla siempre por los
%elementos de la primera columna. (Es un prodigo de ineficiencia numérica,
%pero permite manejar bucles y funciones recursivas, así que supongo que
%puede ser útil para los que empiezan a programar).
%un posible ejercicio para ver lo malo que es el método, consiste ir
%aumentando la dimensión de la matriz y comparar lo que lo tarde en
%calcular el determinante con lo que tarda la función de Matlab det...
```

```
%primero comprobamos que la matriz suministrada es cuadrada:
d=0;
[a,b]=size(M);
if a~=b
    print('la matriz no es cuadrada, Campeón')
    d=[];
else
    for i=1:a
        if a==1
            d=M;
        else
            %Eliminamos la fila y columna que toque
            N=M([1:i-1 i+1:a],2:b);
            %Añadimos el calculo correspondiente al cofactor
            d=(-1)^(i+1)*M(i,1)*determinante(N)+d;
            %pause
        end
    end
end
end
```

Como siempre, Matlab incluye una función propia `inv` para calcular la inversa de una matriz,

```
A =
1     0     -2
3     -2      3
0      6      5
>> AI=inv(A)
AI =
0.4375    0.1875    0.0625
0.2344   -0.0781    0.1406
-0.2813    0.0938    0.0313
>> A*AI
ans =
1.0000      0      0
      0    1.0000      0
-0.0000    0.0000    1.0000
```

Alternativamente, podemos calcular la inversa, directamente como A^{-1} ,

```
>> AI=A^-1
AI =
0.4375    0.1875    0.0625
0.2344   -0.0781    0.1406
-0.2813    0.0938    0.0313
```

Algunas propiedades relacionadas con la inversión de matrices son,

1. Inversa del producto de dos matrices,

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

2. Determinante de la inversa,

$$|A^{-1}| = |A|^{-1}$$

3. Una matriz es ortogonal si su inversa coincide con su transpuesta,

$$A^{-1} = A^T$$

5.3. Operadores vectoriales

En esta sección vamos a estudiar el efecto de las operaciones matriciales, descritas en la sección anterior, sobre los vectores. Empecemos por considerar el producto por un escalar $\alpha \cdot a$. El efecto fundamental es modificar el módulo del vector,

$$\alpha \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} \alpha a_1 \\ \alpha a_2 \\ \alpha a_3 \end{pmatrix} \rightarrow \|\alpha \cdot a\| = \sqrt{\alpha^2 a_1^2 + \alpha^2 a_2^2 + \alpha^2 a_3^2} = |\alpha| \sqrt{a_1^2 + a_2^2 + a_3^2} = |\alpha| \cdot \|a\|$$

Gráficamente, si *alpha* es un número positivo y mayor que la unidad, el resultado del producto será un vector más largo que *a* con la misma dirección y sentido. Si por el contrario, *alpha* es menor que la unidad, el vector resultante será más corto que *a*. Por último si se trata de un número negativo, a los resultados anteriores se añade el cambio de sentido con respecto a *a*. La figura 5.4 muestra gráficamente un ejemplo del producto de un vector por un escalar.

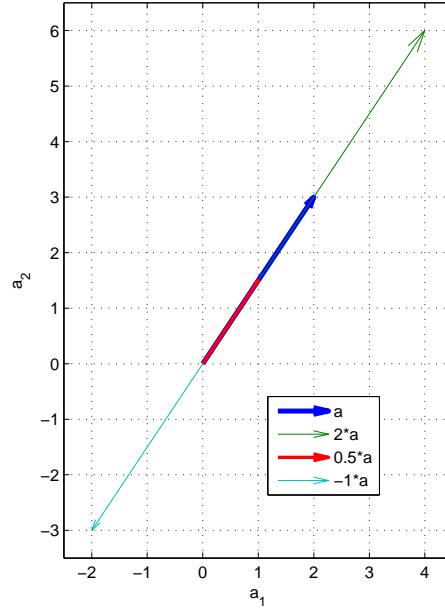


Figura 5.4: efecto del producto de un escalar por un vector

Combinación lineal. Combinando la suma de vectores, con el producto por un escalar, podemos generar nuevos vectores, a partir de otros, el proceso se conoce como combinación lineal,

$$c = \alpha \cdot a + \beta \cdot b + \cdots + \theta z$$

Así el vector c sería el resultado de una combinación lineal de los vectores a, b, \dots, z . Dado un conjunto de vectores, se dice que son linealmente independientes entre sí, si no es posible poner a unos como combinación lineal de otros,

$$\alpha \cdot a + \beta \cdot b + \cdots + \theta z = 0 \Rightarrow \alpha = \beta = \cdots = \theta = 0$$

Es posible expresar cualquier vector de dimensión n como una combinación lineal de n vectores linealmente independientes.

Supongamos $n = 2$, cualquier par de vectores que no estén alineados, pueden generar todos los vectores de dimensión 2 por ejemplo,

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \alpha \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \beta \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

La figura 5.5 muestra gráficamente estos dos vectores y algunos de los vectores resultantes de combinarlos linealmente.

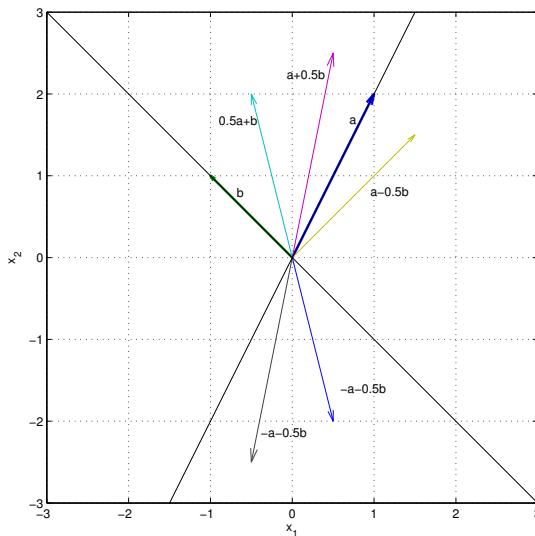


Figura 5.5: Representación gráfica de los vectores $a = (1, 2)$, $b = (-1, 1)$ y algunos vectores, combinación lineal de a y b .

Si tomamos como ejemplo $n = 3$, cualquier conjunto de vectores que no estén contenidos en el mismo plano, pueden generar cualquier otro vector de dimensión 3. Por ejemplo,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \alpha \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix} + \beta \begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix} + \gamma \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}$$

La figura 5.6 muestra gráficamente estos tres vectores y el vector resultante de su combinación lineal, con $\alpha = 1$, $\beta = -0,5$ y $\gamma = 1$. Es fácil ver a partir de la figura que cualquier otro vector de dimensión 3 que queramos construir puede obtenerse a partir de los vectores a , b y c .

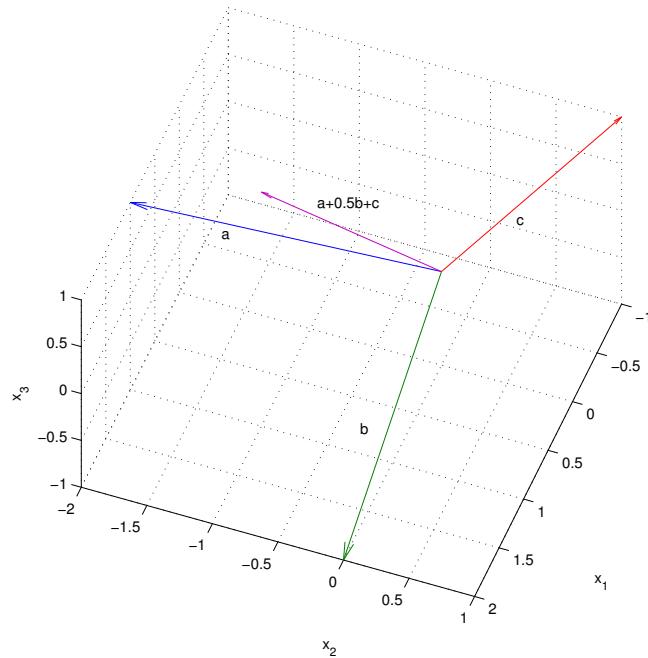


Figura 5.6: Representación gráfica de los vectores $a = (1, -2, 1)$, $b = (2, 0, -1)$, $c = (-1, 1, 1)$ y del vector $a - b + c$.

Espacio vectorial y bases del espacio vectorial. El conjunto de los vectores de dimensión n (matrices de orden $n \times 1$), junto con la suma vectorial y el producto por un escalar, constituye un *espacio vectorial* de dimensión n .

Como acabamos de ver, es posible obtener cualquier vector de dicho espacio vectorial a partir de n vectores linealmente independientes del mismo. Un conjunto de n vectores linealmente independientes de un espacio vectorial de dimensión n recibe el nombre de base del espacio vectorial. En principio es posible encontrar infinitas bases distintas para un espacio vectorial de dimensión n . Hay algunas particularmente interesantes,

Bases ortogonales. Una base ortogonal es aquella en que todos sus vectores son ortogonales entre sí, es decir cumple que su producto escalar es $b^i \cdot b^j = 0$, $i \neq j$. Donde b^i representa el i -ésimo vector de la base, $\mathcal{B} = \{b^1, b^2, \dots, b^n\}$.

Bases ortonormales. Una base ortonormal, es una base ortogonal en la que, además, los vectores de la base tienen módulo 1. Es decir, $b^i \cdot b^j = 0$, $i \neq j$ y $b^i \cdot b^j = 1$, $i = j$. Un caso

particularmente útil de base ortonormal es la base canónica, formada por los vectores,

$$\mathcal{C} = \left\{ c^1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, c^2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \dots, c^{n-1} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix}, c^n = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} \right\}$$

Podemos considerar las componentes de cualquier vector como los coeficientes de la combinación lineal de la base canónica que lo representa,

$$a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix} = a_1 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + a_2 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \dots + a_{n-1} \cdot \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{pmatrix} + a_n \cdot \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Por extensión, podemos generalizar este resultado a cualquier otra base, es decir podemos agrupar en un vector los coeficientes de la combinación lineal de los vectores de la base que lo generan. Por ejemplo, si construimos, para los vectores de dimensión 3 la base,

$$\mathcal{B} = \left\{ \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \right\}$$

Podemos entonces representar un vector en la base \mathcal{B} como,

$$\alpha \cdot \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} + \beta \cdot \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} + \gamma \cdot \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \rightarrow a^{\mathcal{B}} = \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$$

Donde estamos empleando el superíndice \mathcal{B} , para indicar que las componentes del vector a están definidas con respecto a la base \mathcal{B} .

Así por ejemplo el vector,

$$a^{\mathcal{B}} = \begin{pmatrix} 1,125 \\ 0,375 \\ 0,75 \end{pmatrix}$$

Tendría en la base canónica las componentes,

$$a^{\mathcal{B}} = \begin{pmatrix} 1,125 \\ 0,375 \\ 0,75 \end{pmatrix} \rightarrow a = 1,125 \cdot \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} + 0,375 \cdot \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} + 0,75 \cdot \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1,5 \\ 1,5 \\ 1,5 \end{pmatrix}$$

La figura 5.7, muestra gráficamente la relación entre los vectores de la base canónica \mathcal{C} , los vectores de la base \mathcal{B} , y el vector a , cuyas componentes se han representado en ambas bases.

Podemos aprovechar el producto de matrices para obtener las componentes en la base canónica \mathcal{C} de un vector representado en una base cualquiera \mathcal{B} . Si agrupamos los vectores de la base \mathcal{B} , en una matriz B ,

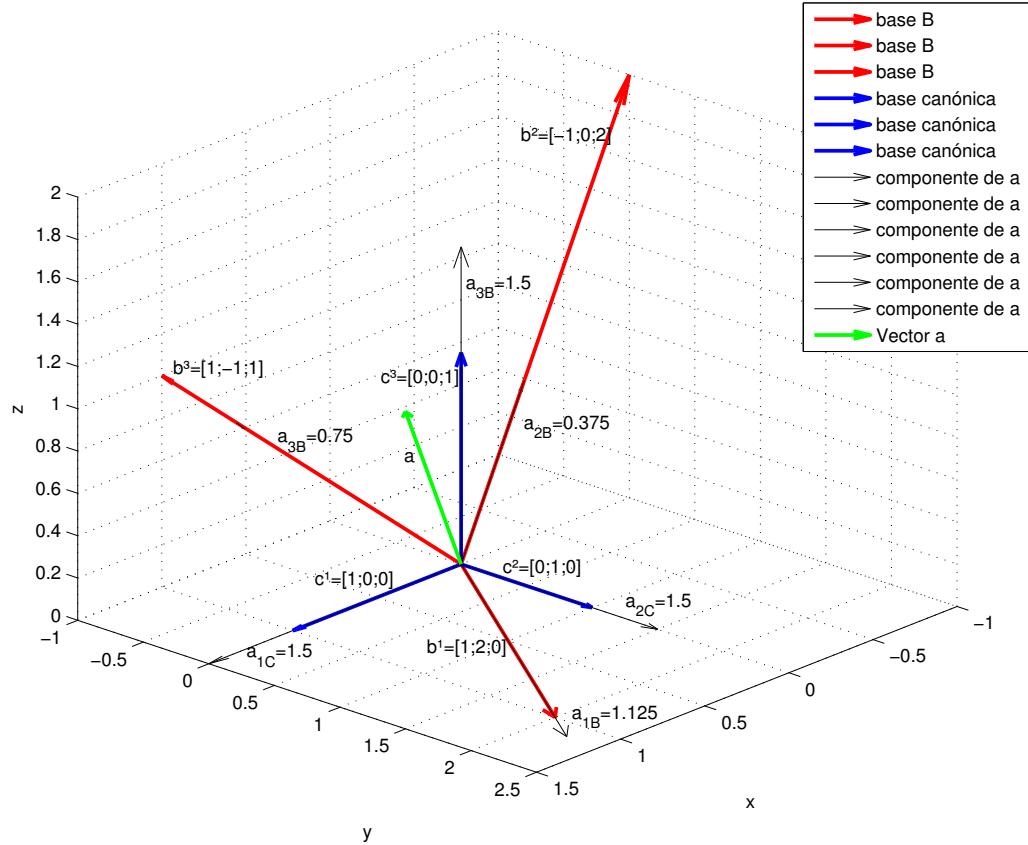


Figura 5.7: Representación gráfica del vector a , en las base canónica \mathcal{C} y en la base \mathcal{B}

$$\mathcal{B} = \left\{ b^1 = \begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ \vdots \\ b_{n1} \end{pmatrix}, b^2 = \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ \vdots \\ b_{n2} \end{pmatrix}, \dots, b^n = \begin{pmatrix} b_{1n} \\ b_{2n} \\ \vdots \\ b_{(n-1)n} \\ b_{nn} \end{pmatrix} \right\} \rightarrow B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ b_{31} & b_{32} & \cdots & \vdots \\ \vdots & \vdots & \cdots & b_{(n-1)n} \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

Supongamos que tenemos un vector a cuyas componentes en la base \mathcal{B} son,

$$a^{\mathcal{B}} = \begin{pmatrix} a_1^{\mathcal{B}} \\ a_2^{\mathcal{B}} \\ \vdots \\ a_n^{\mathcal{B}} \end{pmatrix}$$

Para obtener las componentes en la base canónica, basta entonces multiplicar la matriz B , por el vector $a^{\mathcal{B}}$. Así en el ejemplo que acabamos de ver,

$$a = B \cdot a^B \rightarrow a = \begin{pmatrix} 1 & -1 & 1 \\ 2 & 0 & -1 \\ 0 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1,125 \\ 0,375 \\ 0,75 \end{pmatrix} = \begin{pmatrix} 1,5 \\ 1,5 \\ 1,5 \end{pmatrix}$$

Por último, una podemos combinar el producto de matrices y la matriz inversa, para obtener las componentes de un vector en una base cualquiera a partir de sus componentes en otra base. Supongamos que tenemos dos bases \mathcal{B}_1 y \mathcal{B}_2 y un vector a . Podemos obtener las componentes de a en la base canónica, a partir de las componentes en la base \mathcal{B}_1 como, $a = B_1 \cdot a^{\mathcal{B}_1}$ y a partir de sus componentes en la base \mathcal{B}_2 como $a = B_2 \cdot a^{\mathcal{B}_2}$. Haciendo uso de la matriz inversa,

$$\begin{aligned} a = B_1 \cdot a^{\mathcal{B}_1} &\Rightarrow a^{\mathcal{B}_1} = B_1^{-1} \cdot a \\ a = B_2 \cdot a^{\mathcal{B}_2} &\Rightarrow a^{\mathcal{B}_2} = B_2^{-1} \cdot a \end{aligned}$$

Y sustituyendo obtenemos,

$$\begin{aligned} a^{\mathcal{B}_1} &= B_1^{-1} \cdot B_2 \cdot a^{\mathcal{B}_2} \\ a^{\mathcal{B}_2} &= B_2^{-1} \cdot B_1 \cdot a^{\mathcal{B}_1} \end{aligned}$$

El siguiente código permite cambiar de base un vector y representa gráficamente tanto el vector como las bases antigua y nueva.

```
function aB2=cambia_vb(aB1,B1,B2)
%este programa cambia de base un vector de dimensión 3 y lo representa en
%relación con las bases antigua y nueva.
%variables de entrada:
%aB1, componentes del vector en la base 1
%B1 base representada como una matriz, (cada columna contiene un vector de
%la base) En la que está representado el vector aB1
%B2, base representada como una matriz, (cada columna contiene un vector de
%la base) en la que se quiere representar el vector aB1
%Si solo se incluye un vector y una base, el programa asume que la segunda
%base es la canonica B2=[1 0 0;0 1 0; 0 0 1]
%variables de salida:
%aB2, Componentes del vector aB1 en la nueva base B2.
%la función hace uso de una función auxiliar (pintavec) incluida al final
%del fichero para dibujar los vectores.

if nargin==2
    %asumimos que queremos cambiar el vector de B1 a la base canonica
    %¿Es B1 una base sensata?
    if det(B1)<=eps
        error('los vectores de la base no son l. independientes')
    end
    aB2=B1*aB1;
    B2=eye(3); %creamos la base para luego pintarla...
elseif nargin==3
    %cambio de base B1 a B2
    if (det(B1)<=eps)||(det(B2)<=eps)
```

```

        error('los vectores de al menos una de las bases no son l. independientes')
    end
    %invertimos la base nueva y multiplicamos por la antigua y por el
    %vector para obtener las componentes del vector en la base nueva.
    aB2=inv(B2)*B1*aB1;
else
    error('el numero de variables de entrada es menor de dos o mayor de tres')
end

%Dibujo de los vectores con la función pintavec.....

pintavec(B1,'r') %vectores de la base original
xlabel('x')
ylabel('y')
zlabel('z')
grid on
pintavec(B2,'b') %vectores de la base nueva
for i=1:3
    pintavec(aB1(i)*B1(:,i),'k') %componentes de aB1
    pintavec(aB2(i)*B2(:,i),'k') %componentes de aB2
end
aBc=B1*aB1; %representacion del vector en la base canónica
pintavec(aBc,'g') %vector representado

function pintavec(a,par)
%función auxiliar para pintar vectores... con origen en el origen de
%coordenadas (0,0,0).
%la variable a puede ser un vector o una matriz. y par, es una cadena que contiene los
%típicos parámetros(color, tipo de línea'etc'). El programa considera que
%los vectores están siempre definidos como vectores columnas...
d=size(a,2); %miramos cuantas columnas tiene a, cada columna representará un
%vector distinto
if nargin==2
    for i=1:d
        quiver3(0,0,0,a(1,i),a(2,i),a(3,i),0,par)
        hold on
    end
else
    for i=1:d
        quiver3(0,0,0,a(1,i),a(2,i),a(3,i),0)
    end
end

```

Operadores lineales. A partir de lo visto en las secciones anteriores, sabemos que el producto de una matriz de A de orden $n \times n$ multiplicada por un vector b de dimensión n da como resultado un nuevo vector $c = A \cdot b$ de dimensión n . Podemos considerar cada matriz $n \times n$ como un *operador*

lineal, que transforma unos vectores en otros. Decimos que se trata de un operador lineal porque las componentes del vector resultante, están relacionadas linealmente con las del vector original, por ejemplo para $n = 3$,

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow \begin{array}{l} y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{array}$$

Entre los operadores lineales, es posible destacar aquellos que producen transformaciones geométricas sencillas. Veamos algunos ejemplos para vectores bidimensionales,

1. Dilatación: aumenta el módulo de un vector en un factor $\alpha > 1$. Contracción: disminuye el módulo de un vector en un factor $0 < \alpha < 1$. En ambos casos, se conserva la dirección y el sentido del vector original.

$$R = \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \rightarrow R \cdot a = \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} \alpha \cdot a_1 \\ \alpha \cdot a_2 \end{pmatrix}$$

2. Reflexión de un vector respecto al eje x, conservando su módulo,

$$R_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \rightarrow R_x \cdot a = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ -a_2 \end{pmatrix}$$

3. Reflexión de un vector respecto al eje y, conservando su módulo,

$$R_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow R_y \cdot a = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} -a_1 \\ a_2 \end{pmatrix}$$

4. Reflexión respecto al origen: Invierte el sentido de un vector, conservando su módulo y dirección,

$$R = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \rightarrow R \cdot a = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} -a_1 \\ -a_2 \end{pmatrix}$$

Sería equivalente a aplicar una reflexión respecto al eje x y luego respecto al eje y o viceversa,
 $R = R_x \cdot R_y = R_y \cdot R_x$.

5. Rotación en torno al origen un ángulo θ ,

$$R_\theta = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \rightarrow R_\theta \cdot a = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \cos(\theta) - a_2 \sin(\theta) \\ a_1 \sin(\theta) + a_2 \cos(\theta) \end{pmatrix}$$

La figura 5.8 muestra los vectores resultantes de aplicar las transformaciones lineales que acabamos de describir al vector, $a = (\frac{1}{2})$,

Norma de una matriz. La norma de una matriz se puede definir a partir del efecto que produce al actuar, como un operador lineal, sobre un vector. En este caso, se les llama normas *inducidas*. Para una matriz A de orden $m \times n$, $y_{(m)} = A_{(m \times n)}x_{(n)}$, La norma inducida de A se define en función de las normas de los vectores x de su dominio y de las normas de los vectores y de su rango como,

$$\|A\| = \max_{x \neq 0} \frac{\|y\|}{\|x\|} = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

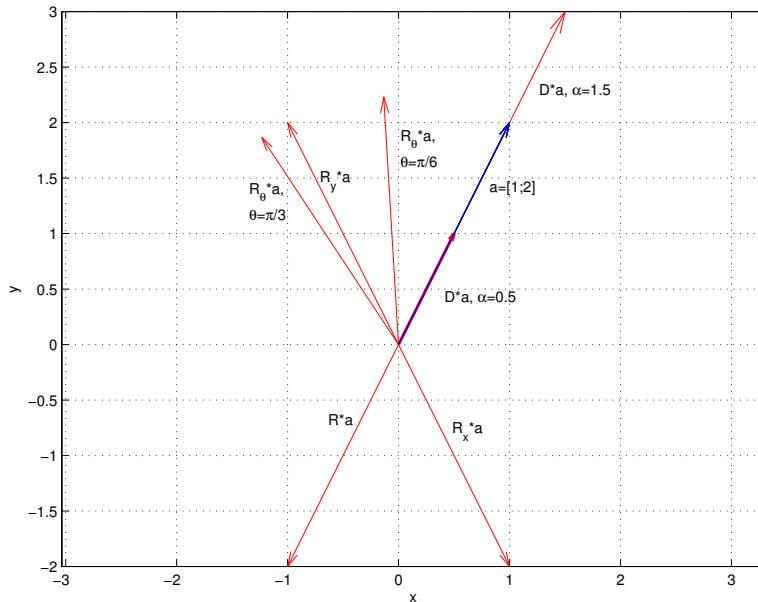


Figura 5.8: Transformaciones lineales del vector $a = [1; 2]$. D , dilatación/contracción en un factor 1,5/0,5. R_x , reflexión respecto al eje x. R_y , reflexión respecto al eje y. R_θ rotaciones respecto al origen para ángulos $\theta = \pi/6$ y $\theta = \pi/3$

Se puede interpretar como el factor máximo con que el que la matriz A puede *alargar* un vector cualquiera. Es posible definir la norma inducida en función de los vectores unitarios del dominio,

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{x \neq 0} \left\| A \frac{x}{\|x\|} \right\| = \max_{\|x\|=1} \|Ax\|$$

Junto a la norma inducida que acabamos de ver, se definen las siguientes normas,

1. Norma 1: Se suman los elementos de cada columna de la matriz, y se toma como norma el valor máximo de dichas sumas,

$$\|A_{m,n}\|_1 = \max_j \sum_{i=1}^m a_{ij}$$

2. Norma ∞ : Se suman los elementos de cada fila y se toma como norma ∞ el valor máximo de dichas sumas.

$$\|A_{m,n}\|_\infty = \max_i \sum_{j=1}^m a_{ij}$$

3. Norma 2: Se define como el mayor de los valores singulares de una matriz. (Ver sección 5.5.5).

$$\|A_{m,n}\|_2 = \sigma_1$$

4. Norma de Frobenius. Se define como la raíz cuadrada de la suma de los cuadrados de todos los elementos de la matriz,

$$\|A_{m,n}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^m a_{ij}^2}$$

Que también puede expresarse de forma mas directa como,

$$\|A_{m,n}\|_F = \sqrt{tr(A^T \cdot A)}$$

En Matlab, es posible calcular las distintas normas de una matriz, de modo análogo a como se calculan para el caso de vectores, mediante el comando `norm(A,p)`. Donde `A`, es ahora un a matriz y `p` especifica el tipo de norma que se quiere calcular. En el caso de una matriz, el parámetro `p` solo puede tomar los valores, 1 (norma 1), 2 (norma 2), `inf` (norma ∞), y `'fro'` (norma de frobenius). El siguiente ejemplo muestra el cálculo de las normas 1, 2, ∞ y de Frobenius de la misma matriz.

```
>> A=[1 3 4 5
2 5 6 -3
1 0 4 3]
A =
    1     3     4     5
    2     5     6    -3
    1     0     4     3
>> n1=norm(A,1)
n1 =
    14
>> n2=norm(A,2)
n2 =
    1.022217214669622e+01
>> ninf=norm(A,inf)
ninf =
    16
>> nfro=norm(A,'fro')
nfro =
    1.228820572744451e+01
```

Formas cuadráticas. Se define como forma cuadrática a la siguiente operación entre una matriz cuadrada A de orden $n \times n$ y un vector x de dimensión n ,

$$\alpha = x^T \cdot A \cdot x, \quad \alpha \in \mathbb{R}$$

El resultado es un escalar. Así por ejemplo,

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 0 & 2 \\ 3 & 2 & -2 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \rightarrow (1 \ 2 \ 3) \cdot \begin{pmatrix} 1 & 2 & -1 \\ 2 & 0 & 2 \\ 3 & 2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = 21$$

Para dimensión $n = 2$,

$$\alpha = (x_1 \ x_2) \cdot \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow x_3 \equiv \alpha = a_{11}x_1^2 + (a_{12} + a_{21})x_1x_2 + a_{22}x_2^2$$

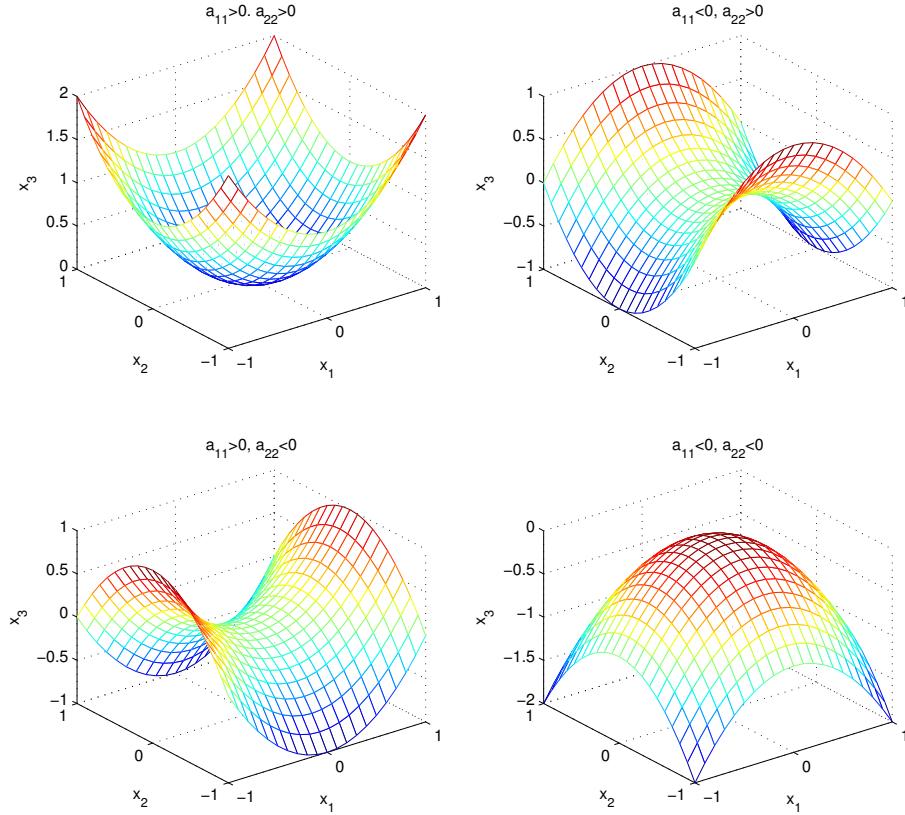


Figura 5.9: Formas cuadráticas asociadas a las cuatro matrices diagonales: $|a_{11}| = |a_{22}| = 1$, $a_{12} = a_{21} = 0$

Lo que obtenemos, dependiendo de los signos de a_{11} y a_{12} , es la ecuación de un paraboloido o un hiperboloide. En la figura 5.9 Se muestra un ejemplo,

Veamos brevemente, algunas propiedades relacionadas con las formas cuadráticas,

1. Una matriz A de orden $n \times n$ se dice que es definida positiva si da lugar a una forma cuadrática que es siempre mayor que cero para cualquier vector no nulo,

$$x^T \cdot A \cdot x > 0, \forall x \neq 0$$

2. Una matriz *simétrica* es definida positiva si todos sus *valores propios* (ver sección 5.5.3) son positivos.
3. Una matriz no simétrica A es definida positiva si su parte simétrica $A_s = (A + A^T)/2$ lo es.

$$x \cdot A_s \cdot x > 0, \forall x \neq 0 \Rightarrow x \cdot A \cdot x > 0, \forall x \neq 0$$

5.4. Tipos de matrices empleados frecuentemente

Definimos a continuación algunos tipos de matrices frecuentemente empleados en álgebra, algunos ya han sido introducidos en secciones anteriores. Los reunimos todos aquí para facilitar su consulta

1. Matriz ortogonal: Una matriz $A_{n \times n}$ es ortogonal cuando su inversa coincide con su traspuesta.

$$A^T = A^{-1}$$

ejemplo,

$$A = \begin{pmatrix} 1/3 & 2/3 & 2/3 \\ 2/3 & -2/3 & 1/3 \\ 2/3 & 1/3 & -2/3 \end{pmatrix} \rightarrow A \cdot A^T = A^T \cdot A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2. Matriz simétrica: Una matriz $A_{n \times n}$ es simétrica cuando es igual que su traspuesta,

$$A = A^T \rightarrow a_{ij} = a_{ji}$$

ejemplo,

$$A = \begin{pmatrix} 1 & -2 & 3 \\ -2 & 4 & 0 \\ 3 & 0 & -5 \end{pmatrix}$$

3. Matriz Diagonal: Una matriz A es diagonal si solo son distintos de ceros los elementos de su diagonal principal,

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \rightarrow a_{ij} = 0, \forall i \neq j$$

4. Matriz triangular superior: Una matriz es triangular superior cuando todos los elementos situados por debajo de la diagonal son cero. Es estrictamente diagonal superior si además los elementos de la diagonal también son cero,

$$\begin{aligned} TRS &\rightarrow a_{ij} = 0, \forall i \geq j \\ ETRS &\rightarrow a_{ij} = 0, \forall i > j \end{aligned}$$

ejemplos,

$$\begin{aligned} TRS &= \begin{pmatrix} 1 & 3 & 7 \\ 0 & 2 & -1 \\ 0 & 0 & 4 \end{pmatrix} \\ ETRS &= \begin{pmatrix} 0 & 3 & 7 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

5. Matriz triangular inferior: Una matriz es triangular inferior si todos los elementos pro encima de su diagonal son cero. Es estrictamente triangular inferior si además los elementos de su diagonal son también cero,

$$\begin{aligned} TRI &\rightarrow a_{ij} = 0, \forall i \leq j \\ ETRI &\rightarrow a_{ij} = 0, \forall i < j \end{aligned}$$

ejemplos,

$$\begin{aligned} TRI &= \begin{pmatrix} 1 & 0 & 0 \\ 3 & 2 & 0 \\ 7 & -1 & 4 \end{pmatrix} \\ ETRI &= \begin{pmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 7 & -1 & 0 \end{pmatrix} \end{aligned}$$

6. Matriz definida Positiva. Una Matriz $A_{n \times n}$ es definida positiva si dado un vector x no nulo cumple,

$$x^T \cdot A \cdot x > 0, \forall x \neq 0$$

si,

$$x^T \cdot A \cdot x \geq 0, \forall x \neq 0$$

entonces la matriz A es semidefinida positiva.

7. Una matriz es Diagonal dominante si cada uno de los elementos de la diagonal en valor absoluto es mayor que la suma de los valores absolutos de los elementos de fila a la que pertenece.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \forall i$$

ejemplo,

$$A = \begin{pmatrix} 10 & 2 & 3 \\ 2 & -5 & 1 \\ 4 & -2 & 8 \end{pmatrix} \rightarrow \begin{cases} 10 > 2 + 3 \\ 5 > 2 + 1 \\ 8 > 4 + 2 \end{cases}$$

5.5. Factorización de matrices

La factorización de matrices, consiste en la descomposición de una matriz en el producto de dos o más matrices. Las matrices resultantes de la factorización se eligen de modo que simplifiquen, o hagan más robustas numéricamente determinadas operaciones matriciales: Cálculos de determinantes, inversas, etc. A continuación se describen las más comunes.

5.5.1. Factorización LU

Consiste en factorizar una matriz como el producto de una matriz triangular inferior L por una matriz triangular superior U , $A = L \cdot U$. Por ejemplo,

$$\begin{pmatrix} 3 & 4 & 2 \\ 2 & 0 & 1 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1 & 3/4 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 4 & 2 \\ 0 & -8/3 & -1/3 \\ 0 & 0 & -3/4 \end{pmatrix}$$

Una aplicación inmediata, es el cálculo del determinante. Puesto que el determinante de una matriz triangular, es directamente el producto de los elementos de la diagonal.

En el ejemplo anterior,

$$|A| = 6 \equiv |L| \cdot |U| = 1 \cdot 1 \cdot 1 \cdot 3 \cdot \left(-\frac{8}{3}\right) \cdot \left(-\frac{3}{4}\right) = 6$$

Uno de los métodos más conocidos para calcular la factorización LU de una matriz, se basa en el método conocido como eliminación gaussiana. La idea es convertir en ceros los elementos situados por debajo de la diagonal de la matriz. Para ello, se sustituyen progresivamente las filas de la matriz, exceptuando la primera, por combinaciones formadas con la fila que se sustituye y la fila anterior. veamos en qué consiste con un ejemplo. Supongamos que tenemos la siguiente matriz de orden 4×4 ,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Si sustituimos la segunda fila por el resultado de restarle la primera multiplicada por 2 y dividida por 3 obtendríamos la matriz,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix} \rightarrow [2 \ 0 \ 1 \ -2] - \frac{2}{3}[3 \ 4 \ 2 \ 5] \rightarrow U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

De modo análogo, si sustituimos ahora la tercera fila por el resultado de restarle la primera multiplicada por 3 y dividida 3,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix} \rightarrow [3 \ 2 \ 1 \ 8] - \frac{3}{3}[3 \ 4 \ 2 \ 5] \rightarrow U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & -2 & -1 & -3 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Por último si sustituimos la última fila por el resultado de restarle la primera multiplicada por 5 y dividida por 3,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & -2 & -1 & -3 \\ 5 & 2 & 3 & 2 \end{pmatrix} \rightarrow [5 \ 2 \ 3 \ 2] - \frac{5}{3}[3 \ 4 \ 2 \ 5] \rightarrow U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & -2 & -1 & -3 \\ 0 & -4,66 & -0,33 & -6,33 \end{pmatrix}$$

El resultado que hemos obtenido, tras realizar esta transformación, es una nueva matriz U en la que todos los elementos de su primera columna, por debajo de la diagonal, son ceros.

Podemos proceder de modo análogo para *eliminar* ahora los elementos de la segunda columna situados por debajo de la diagonal. Para ellos sustituimos la tercera fila por la diferencia entre ella y las segunda fila multiplicada por -2 y dividida por $-2,6$.

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & -2 & -1 & -3 \\ 0 & -4,66 & -0,33 & -6,33 \end{pmatrix} \rightarrow [0 \ -2 \ -1 \ -3] - \frac{-2}{-2,6} [0 \ -2,6 \ -0,33 \ -5,33] \rightarrow$$

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & 0 & -0,75 & 7 \\ 0 & -4,66 & -0,33 & -6,33 \end{pmatrix}$$

Y sustituyendo la última fila por la diferencia entre ella y la segunda multiplicada por $-4,66$ y dividida por $-2,6$,

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & -2 & -1 & -3 \\ 0 & -4,66 & -0,33 & -6,33 \end{pmatrix} \rightarrow [0 \ -4,66 \ -0,33 \ -6,33] - \frac{-4,66}{-2,6} [0 \ -2,6 \ -0,33 \ -5,33] \rightarrow$$

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & 0 & -0,75 & 7 \\ 0 & 0 & 0,25 & 3 \end{pmatrix}$$

De este modo, los elementos de la segunda columna situados debajo de la diagonal, han sido sustituidos por ceros. Un último paso, nos llevará hasta una matriz triangular superior; sustituimos la última fila por la diferencia entre ella y la tercera fila multiplicada por $0,25$ y dividida por $-0,75$,

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & 0 & -0,75 & 7 \\ 0 & 0 & 0,25 & 3 \end{pmatrix} \rightarrow [0 \ 0 \ 0,25 \ 3] - \frac{0,25}{-0,75} [0 \ 0 \ -0,75 \ 7] \rightarrow$$

$$U_3 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & 0 & -0,75 & 7 \\ 0 & 0 & 0 & 5,33 \end{pmatrix} = U$$

Podemos ahora, a partir del ejemplo, deducir un procedimiento general. Para *eliminar* — convertir en 0— el elemento a_{ij} situado por debajo de la diagonal principal, $i > j$:

- Dividimos los elementos de la fila j por el elemento de dicha fila que a su vez pertenece a la diagonal, a_{jj}

$$[0/a_{jj} \quad 0/a_{jj} \quad \cdots \quad a_{jj}/a_{jj} \quad a_{jj+1}/a_{jj} \quad \cdots]$$

- Multiplicamos el resultado de la operación anterior por el elemento a_{ij} ,

$$[a_{ij} \cdot 0/a_{jj} \quad a_{ij} \cdot 0/a_{jj} \quad \cdots \quad a_{ij} \cdot a_{jj}/a_{jj} \quad a_{ij} \cdot a_{jj+1}/a_{jj} \quad \cdots]$$

- Finalmente, sustituimos la fila i de la matriz de partida por la diferencia entre ella y el resultado de la operación anterior.

$$[0 \quad 0 \quad \cdots \quad a_{ij} \quad a_{ij+1} \quad \cdots] - [a_{ij} \cdot 0/a_{jj} \quad a_{ij} \cdot 0/a_{jj} \quad \cdots \quad a_{ij} \cdot a_{jj}/a_{jj} \quad a_{ij} \cdot a_{jj+1}/a_{jj} \quad \cdots]$$

Este procedimiento se aplica iterativamente empezando en por el elemento a_{21} de la matriz y desplazando el cálculo hacia abajo, hasta llegar a la última fila y hacia la derecha hasta llegar en cada fila al elemento anterior a la diagonal.

El siguiente código aplica el procedimiento descrito a una matriz de cualquier orden,

```
function U=eligauss(A)
%Esta función obtiene una matriz triangular superior, a partir de una
%matriz dada, aplicando el método de eliminación gaussiana.
%No realiza pivoteo de filas, así que si algún elemento de la diagonal de A
%quedá cero o próximo a cero al ir eliminado dará problemas...
%Obtenemos el número de filas de la matriz..
nf=size(A,1);
U=A
%
for j=1:nf-1 %recorro todas las columnas menos la última
    for i=j+1:nf %Recorro las filas desde debajo de la diagonal hasta la última
        %en Matlab tengo la suerte de poder manejar cada fila de un solo
        %golpe.
        U(i,:)=U(i,:)-U(j,:)*U(i,j)/U(j,j)
    end
end
```

Hasta ahora, hemos descrito un procedimiento para transformar una matriz cualquiera en una matriz triangular superior. Nuestro objetivo era obtener la descomposición de un matriz en el producto de dos, una triangular inferior y otra triangular superior. En primer lugar, podemos asociar el procedimiento descrito de eliminación gaussiana al producto de matrices. Volviendo al ejemplo anterior, si construimos la matriz λ_1

$$\lambda_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -2/3 & 1 & 0 & 0 \\ -3/3 & 0 & 1 & 0 \\ -5/3 & 0 & 0 & 1 \end{pmatrix}$$

, El producto $\lambda_1 \cdot A$ da como resultado la matriz,

$$U_1 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,66 & -0,33 & -5,33 \\ 0 & -2 & -1 & -3 \\ 0 & -4,66 & -0,33 & -6,33 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -2/3 & 1 & 0 & 0 \\ -3/3 & 0 & 1 & 0 \\ -5/3 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

De modo análogo, $U_2 = \lambda_2 \cdot U_1$

$$\lambda_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -2/2,66 & 1 & 0 \\ 0 & -4,66/2,66 & 0 & 1 \end{pmatrix}$$

$$U_2 = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,6 & -0,33 & -5,33 \\ 0 & 0 & -0,75 & 7 \\ 0 & 0 & 0,25 & 3 \end{pmatrix}$$

Por último, $U = \lambda_3 \cdot U_2$

$$\lambda_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0,25/0,75 & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 0 & -2,66 & -0,33 & -5,33 \\ 0 & 0 & -0,75 & 7 \\ 0 & 0 & 0 & 5,33 \end{pmatrix}$$

De nuevo, podemos generalizar el procedimiento empleado; cada matriz λ_j *elimina* todos los elementos de la columna n de una matriz A , situados por debajo de la diagonal. La matriz λ_j toma la forma general,

$$\lambda_j = \begin{pmatrix} 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & -a_{j+1,j}/a_{jj} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & -a_{j+2,j}/a_{jj} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & -a_{n,j}/a_{jj} & 0 & \cdots & 1 \end{pmatrix}$$

Solo los elementos de la diagonal, que toman el valor 1, y los elementos de la columna j situados por debajo de la diagonal son distintos de cero.

A partir de la definición de las matrices λ_j , podemos obtener una relación entre la matriz triangular superior U obtenida al final del proceso de eliminación, y la matriz inicial A de orden $n \times n$ para ello, en cada paso multiplicamos tanto por λ_j como por su inversa λ_j^{-1} ,

$$\begin{aligned} A &= \lambda_1^{-1} \cdot \overbrace{\lambda_1 \cdot A}^{U_1} \\ A &= \lambda_1^{-1} \cdot \lambda_2^{-1} \cdot \overbrace{\lambda_2 \cdot \lambda_1 A}^{U_2} \\ A &= \lambda_1^{-1} \cdot \lambda_2^{-1} \cdot \overbrace{\lambda_3^{-1} \cdot \lambda_3 \cdot \lambda_2 \cdot \lambda_1 \cdot A}^{U_3} \\ &\vdots \\ A &= \lambda_1^{-1} \cdot \lambda_2^{-1} \cdot \lambda_3^{-1} \cdots \lambda_n^{-1} \cdot \overbrace{\lambda_n \cdots \lambda_3 \cdot \lambda_2 \cdot \lambda_1 \cdot A}^U \end{aligned}$$

Las matrices λ_j^{-1} tienen dos propiedades que las hacen particularmente fáciles de manejar: la primera es que cumplen que su inversa λ_j^{-1} puede obtenerse a partir de λ_j , sin más que cambiar

de signo los elementos distintos de cero que no pertenecen a la diagonal (Compruébalo),

$$\lambda_j^{-1} = \begin{pmatrix} 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & a_{j+1j}/a_{jj} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & a_{j+2j}/a_{jj} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{nj}/a_{jj} & 0 & \cdots & 1 \end{pmatrix}$$

La segunda propiedad es que el producto $L = \lambda_1^{-1} \cdot \lambda_2^{-1} \cdot \lambda_3^{-1} \cdots \lambda_n^{-1}$, se puede obtener progresivamente, a la vez que se construye U , sin más que ir juntando en una única matriz L las columnas de λ_1^{-1} , λ_2^{-1} , etc. que contienen elementos no nulos, en nuestro ejemplo,

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2/3 & 1 & 0 & 0 \\ 3/3 & 2/2,66 & 1 & 0 \\ 5/3 & 4,66/2,66 & -0,25/0,75 & 1 \end{pmatrix}$$

y, en general,

$$L = \begin{pmatrix} 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{j-11}/a_{11} & \cdots & 1 & 0 & 0 & \cdots & 0 \\ a_{j1}/a_{11} & \cdots & a_{jj-1}/a_{j-1j-1} & 1 & 0 & \cdots & 0 \\ a_{j+11}/a_{11} & \cdots & a_{j+1j-1}/a_{j-1j-1} & a_{j+1j}/a_{jj} & 0 & \cdots & 0 \\ a_{j+11}/a_{11} & \cdots & a_{j+2j-1}/a_{j-1j-1} & a_{j+2j}/a_{jj} & 1 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{n1}/a_{11} & \cdots & a_{nj-1}/a_{j-1j-1} & a_{nj}/a_{jj} & a_{nj+1}/a_{j+1j+1} & \cdots & 1 \end{pmatrix}$$

Por construcción, la matriz L es una matriz inferior, Con lo que quedaría completo el cálculo de la factorización LU,

$$A = \overbrace{\lambda_1^{-1} \cdot \lambda_2^{-1} \cdot \lambda_3^{-1} \cdots \lambda_n^{-1}}^L \cdot \overbrace{\lambda_n \cdots \lambda_3 \cdot \lambda_2 \cdot \lambda_1 \cdot A}^U = L \cdot U$$

La factorización que acabamos de describir, puede presentar problemas numéricos dependiendo del cómo sea la matriz que se desea factorizar. El primer problema se produce cuando el elemento de la diagonal de u_{jj} por el que toca dividir para eliminar los elementos de la columna j , situados por debajo de la diagonal, es 0. En ese caso el ordenador dará un error de desbordamiento y no se podrá seguir factorizando. El segundo problema surge cuando los elementos de la matriz son dispares en magnitud; las operaciones matemáticas realizadas durante el proceso de factorización pueden dar lugar a errores de redondeo importantes que hagan incorrecto el resultado de la factorización. Veamos un ejemplo un tanto extremo,

$$\begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}}^L \cdot \overbrace{\begin{pmatrix} 1 & 0 \\ -10^{20} & 1 \end{pmatrix}}^U \cdot \overbrace{\begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix}}^L = \overbrace{\begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}}^L \cdot \overbrace{\begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix}}^U$$

Como el eps del ordenador es del orden de 10^{-16} , 1 es despreciado frente 10^{20} . Es decir, $(1 - 10^{20}) \approx 10^{20}$, con lo cual el ordenador tendrá una versión aproximada de U

$$U \approx \hat{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix}$$

Pero si ahora volvemos a multiplicar $L \cdot U$ para recuperar A ,

$$L \cdot \hat{U} = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix} \cdot \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 0 \end{pmatrix} \neq A$$

Una manera de paliar los efectos de redondeo, es reordenar las filas de la matriz de modo que el elemento a_{jj} por el que se va a dividir los elementos de la fila j en el proceso de eliminación sea lo mayor posible. Este procedimiento se conoce con el nombre de pivoteo de filas. Para el ejemplo que acabamos de examinar, supongamos que cambiamos de orden las dos filas de la matriz,

$$\begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 \\ 10^{-20} & 1 \end{pmatrix}$$

Si recalculamos la factorización LU, para la nueva matriz con la filas intercambiadas,

$$\begin{pmatrix} 1 & 1 \\ 10^{-20} & 1 \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 0 \\ 10^{-20} & 1 \end{pmatrix}}^L \cdot \overbrace{\begin{pmatrix} 1 & 0 \\ -10^{-20} & 1 \end{pmatrix}}^U \cdot \overbrace{\begin{pmatrix} 1 & 1 \\ 10^{-20} & 1 \end{pmatrix}}^L = \overbrace{\begin{pmatrix} 1 & 0 \\ 10^{-20} & 1 \end{pmatrix}}^L \cdot \overbrace{\begin{pmatrix} 1 & 1 \\ 0 & 1 - 10^{-20} \end{pmatrix}}^U$$

De nuevo, por errores de redondeo el ordenador tendrá una versión aproximada de U .

$$U \approx \hat{U} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Sin embargo si recalculamos el producto $L \cdot \hat{U}$ y volvemos a reordenar las filas del resultado,

$$L \cdot \hat{U} = \begin{pmatrix} 1 & 0 \\ 10^{-20} & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 10^{-20} & 1 + 10^{-20} \end{pmatrix} \rightarrow \begin{pmatrix} 10^{-20} & 1 + 10^{-20} \\ 1 & 1 \end{pmatrix} \approx A$$

La permutación de las filas de una matriz A de orden $n \times m$, se puede definir a partir del producto con las matrices de permutación de orden $n \times n$. Éstas se obtienen permutando directamente las filas de la matriz identidad $I_{n \times n}$. Si una matriz de permutación multiplica a otra matriz por la izquierda, permuta el orden de sus filas. Si la multiplica por la derecha, permuta el orden de sus columnas. Así por ejemplo, para matrices de orden $3 \times n$,

$$I_{n \times n} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \rightarrow P_{1 \leftrightarrow 3} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Si multiplicamos $P_{1 \leftrightarrow 3}$ con cualquier otra matriz A de orden $3 \times n$, El resultado es equivalente a intercambiar en la matriz A la fila 1 con la 3. Por ejemplo

$$P_{1 \leftrightarrow 3} \cdot A = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 5 & 3 \\ 4 & 2 & 3 & 0 \\ 3 & 6 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 6 & 2 & 1 \\ 4 & 2 & 3 & 0 \\ 1 & 2 & 5 & 3 \end{pmatrix} = A_{1 \leftrightarrow 3}$$

Volvamos al cálculo de la factorización LU, pero ahora empleando pivoteo de filas. Supongamos una matriz A de orden $n \times n$

El primer paso, es buscar el elemento mayor en valor absoluto de la primera columna en intercambia la primera fila de la matriz de A , con la fila que contiene dicho elemento. Si utilizamos la matriz de permutación adecuada, esto puede expresarse como,

$$A \rightarrow P_1 \cdot A$$

A continuación eliminamos los elementos de la primera fila situados por debajo de la diagonal,

$$A \rightarrow P_1 \cdot A \rightarrow \lambda_1 \cdot P_1 \cdot A$$

Volvemos a buscar el mayor elemento en valor absoluto para la segunda columna (Solo desde la diagonal hasta el ultimo elemento de la columna).

$$A \rightarrow P_1 \cdot A \rightarrow \lambda_1 \cdot P_1 \cdot A \rightarrow P_2 \cdot \lambda_1 \cdot P_1 \cdot A$$

Eliminamos los elementos de la segunda fila situados por debajo de la diagonal,

$$A \rightarrow P_1 \cdot A \rightarrow \lambda_1 \cdot P_1 \cdot A \rightarrow P_2 \cdot \lambda_1 \cdot P_1 \cdot A \rightarrow \lambda_2 \cdot P_2 \cdot \lambda_1 \cdot P_1 \cdot A$$

Si seguimos todo el proceso hasta eliminar los elementos situados por debajo de la diagonal en la fila $n - 1$, obtendremos la expresión de la matriz triangular superior U resultante,

$$\lambda_{n-1} P_{n-1} \lambda_{n-2} P_{n-2} \cdots \lambda_2 P_2 \lambda_1 P_1 A = U$$

Aunque hemos obtenido U , necesitamos una manera de obtener L , ahora no es inmediato como en el caso de la factorización sin pivoteo, no basta con invertir las matrices λ_i ya que tenemos por medio todas las matrices de permutación utilizadas. Para obtenerla realizaremos la siguiente transformación,

$$\lambda_{n-1} P_{n-1} \lambda_{n-2} P_{n-2} \cdots \lambda_2 P_2 \lambda_1 P_1 A = \lambda'_{n-1} \lambda'_{n-2} \cdots \lambda'_2 \lambda'_1 P_{n-1} P_{n-2} \cdots P_2 P_1 A$$

Donde las matrices λ'_i pueden obtenerse a partir de las matrices λ_i y de las matrices de permutación de la manera siguiente,

$$\begin{aligned} \lambda'_{n-1} &= \lambda_{n-1} \\ \lambda'_{n-2} &= P_{n-1} \lambda_{n-2} P_{n-1}^{-1} \\ \lambda'_{n-3} &= P_{n-1} P_{n-2} \lambda_{n-3} P_{n-2}^{-1} P_{n-1}^{-1} \\ &\vdots \\ \lambda'_k &= P_{n-1} P_{n-2} \cdots P_{k+1} \lambda_k P_{k+1}^{-1} \cdots P_{n-2}^1 P_{n-1}^{-1} \\ &\vdots \\ \lambda'_2 &= P_{n-1} P_{n-2} \cdots P_3 \lambda_2 P_3^{-1} \cdots P_{n-2}^1 P_{n-1}^{-1} \\ \lambda'_1 &= P_{n-1} P_{n-2} \cdots P_3 P_2 \lambda_1 P_2^{-1} P_3^{-1} \cdots P_{n-2}^1 P_{n-1}^{-1} \end{aligned}$$

Matemáticamente el cálculo de las matrices λ'_i requiere calcular el producto de un gran número de matrices. Sin embargo dicho cálculo es equivalente a permutar los elementos de λ_i situados por debajo de la diagonal. Así Por ejemplo para,

$$\lambda_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 3 & 0 & 1 \end{pmatrix}, P_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\lambda'_2 = P_3 \cdot \lambda_2 P_3^{-1} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 3 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{pmatrix}$$

Por último podemos representar el producto de todas las matrices de permutación como una sola matriz, $P_{n-1}P_{n-2}\cdots P_2P_1 = P$. (El producto de matrices de permutación entre sí, da como resultado una nueva matriz de permutación.

De esta manera, la factorización LU con pivoteo de filas podríamos representarla como $P \cdot A = L \cdot U$,

$$\overbrace{P_{n-1}P_{n-2}\cdots P_2P_1}^P \cdot A = \overbrace{\lambda'^{-1}_1 \lambda'^{-2}_2 \cdots \lambda'^{-1}_{n-2} \lambda'^{-1}_{n-1}}^L \cdot \overbrace{\lambda_{n-1} P_{n-1} \lambda_{n-2} P_{n-2} \cdots \lambda_2 P_2 \lambda_1 P_1}^U A$$

Como ejemplo, vamos a volver a calcular la factorización LU de la matriz,

$$A = \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix}$$

Pero empleando ahora pivoteo de filas.

En primer lugar buscamos el elemento de la primera columna mayor en valor absoluto. En esta caso es el último elemento de la columna por lo que intercambiamos la primera fila con la cuarta,

$$P_1 \cdot A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 3 & 4 & 2 & 5 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 5 & 2 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 2 & 3 & 2 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 3 & 4 & 2 & 5 \end{pmatrix}$$

Eliminamos ahora los elementos de la primera columna situados por debajo de la diagonal,

$$\overbrace{\lambda_1 \cdot P_1 \cdot A}^{U_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -2/5 & 1 & 0 & 0 \\ -3/5 & 0 & 1 & 0 \\ -3/5 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 & 2 & 3 & 2 \\ 2 & 0 & 1 & -2 \\ 3 & 2 & 1 & 8 \\ 3 & 4 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & -0,8 & -0,2 & -2,8 \\ 0 & 0,8 & -0,8 & -6,8 \\ 0 & 2,8 & 0,2 & 3,8 \end{pmatrix}$$

Para eliminar los elementos de la segunda columna, repetimos los mismos pasos. El elemento mayor de la segunda columna de U_1 es ahora 2,8, por tanto intercambiamos entre sí la segunda y la cuarta fila,

$$P_2 \cdot \overbrace{\lambda_1 \cdot P_1 \cdot A}^{U_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & -0,8 & -0,2 & -2,8 \\ 0 & 0,8 & -0,8 & -6,8 \\ 0 & 2,8 & 0,2 & 3,8 \end{pmatrix} = \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & 2,8 & 0,2 & 3,8 \\ 0 & 0,8 & -0,8 & -6,8 \\ 0 & -0,8 & -0,2 & -2,8 \end{pmatrix}$$

A continuación, eliminamos los elementos de la segunda columna situados por debajo de la diagonal,

$$\overbrace{\lambda_2 \cdot P_2 \cdot \lambda_1 \cdot P_1 \cdot A}^{U_2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -0,8/2,8 & 1 & 0 \\ 0 & 0,8/2,8 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & 2,8 & 0,2 & 3,8 \\ 0 & 0,8 & -0,8 & -6,8 \\ 0 & -0,8 & -0,2 & -2,8 \end{pmatrix} = \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & 2,8 & 0,2 & 3,8 \\ 0 & 0 & -0,85 & -5,71 \\ 0 & 0 & -0,14 & -1,71 \end{pmatrix}$$

Para eliminar el elemento que queda debajo de la diagonal en la tercera columna, procedemos igual que para las columnas anteriores. Como en este caso, el elemento situado en la diagonal es mayor en valor absoluto no es preciso permutar. (Desde un punto de vista formal, podríamos decir que en este caso la matriz de permutaciones aplicada es la matriz identidad).

$$\begin{aligned} \overbrace{\lambda_3 \cdot \lambda_2 \cdot P_2 \cdot \lambda_1 \cdot P_1 \cdot A}^U &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -0,14/0,85 & 1 \end{pmatrix} \cdot \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & 2,8 & 0,2 & 3,8 \\ 0 & 0 & -0,85 & -5,71 \\ 0 & 0 & -0,14 & -1,71 \end{pmatrix} \\ &= \begin{pmatrix} 5 & 2 & 3 & 2 \\ 0 & 2,8 & 0,2 & 3,8 \\ 0 & 0 & -0,85 & -5,71 \\ 0 & 0 & 0 & -2,66 \end{pmatrix} = U \end{aligned}$$

Como hemos visto, la matriz λ'_i las obtenemos a partir de λ_i , permutando los elementos situados por debajo de la diagonal, distintos de cero. En nuestro ejemplo solo hay matriz λ_1 afectada por una única permutación P_2 ,

$$\lambda'_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3/5 & 1 & 0 & 0 \\ 3/5 & 0 & 1 & 0 \\ 2/5 & 0 & 0 & 1 \end{pmatrix}$$

Para el resto, $\lambda'_2 = \lambda_2$ y $\lambda'_3 = \lambda_3$. Para obtener la matriz L cambiando los signos a los elementos situados por debajo de la diagonal de λ'_1 , λ_2 y λ_3 , y los agrupamos en una única matriz,

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3/5 & 1 & 0 & 0 \\ 3/5 & 0,8/2,8 & 1 & 0 \\ 2/5 & -0,8/2,8 & 0,14/0,85 & 1 \end{pmatrix}$$

Por último debemos multiplicar las matrices de permutaciones para agruparlas en una sola,

$$P = P_2 \cdot P_1 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Es trivial comprobar que las matrices obtenidas cumplen ² $P \cdot A = L \cdot U$.

El siguiente código calcula la factorización LU de una matriz de orden arbitrario con pivoteo de filas.

²Los valores numéricos empleados en el texto, están redondeados, por tanto dan una solución incorrecta. Si se reproducen las operaciones descritas en Matlab el resultado es mucho más preciso.

```

function [L,U,P]=lufact(A)
%este programa calcula una factorizacionLU, de la matrix A, empleando
%el método de eliminacion gaussiana con pivoteo de filas...
%L es una matriz triangular inferior, U es una matriz triangular superior y
%P es una matriz de permutaciones tal que P*A=L*U

%vamos a aprovechar la potencia de Matlab para evitar algunos buques en
%los calculos..

%primero definimos la matriz de permutaciones como la matriz identidad de
%orden igual al número de filas de la matriz A, (si no hay pivoteo de filas
%la matriz de permutaciones es precisamente la identidad)

t=size(A,1); %obtenemos el numero de filas de la matriz A
P=eye(t); %Construimos la matriz identidad
%Ademas, inicializamos las matrices L y U
L=P;
U=A;

%iniciamos un bucle para ir recorriendo las columnas (solo tenemos que
%recorrer tantas columnas como fila - 1 tenga la matriz A
for j=1:t-1
    LA=zeros(t); %Matriz auxiliar (Lambda) de cada iteración.
    %Buscamos el elemento mas grande de la columna j
    maxcol=abs(U(j,j));
    index=j;
    for i=j:t
        if abs(U(i,j))>maxcol
            maxcol=abs(U(i,j));
            index=i;
        end
    end
    Aux=U(j,:);
    Aux2=P(j,:);
    Aux3=L(j,1:j-1);

    %Reordenamos U (Toda la fila)
    U(j,:)=U(index,:);
    U(index,:)=Aux;

    %Reordenamos L (Solo los elementos de la columnas anteriores...situados

```

```
%por debajo de la diagonal).
L(j,1:j-1)=L(index,1:j-1);
L(index,1:j-1)=Aux3;

%modificamos la matriz de permutaciones
P(j,:)=P(index,:);
P(index,:)=Aux2;

%Calculamos una matriz auxiliar con los factores de los elementos
%a eliminar.
LA(j+1:t,j)=U(j+1:t,j)/U(j,j); %estos elementos son directamente los que
%se añaden a la matriz L
%Modificamos L y U
L(j+1:t,j)=U(j+1:t,j)/U(j,j);
U=(eye(t)-LA)*U; %la expresión eye(t)-LA nos permite cambiar de signo los
%elementos situados por debajo de la diagonal principal...
end
```

Desde el punto de vista de la implementación, el código anterior simplifica el cálculo de la factorización LU,

1. no se calculan la inversa de λ_i . En realidad lo que se hace es ir construyendo iterativamente la matriz L: en cada iteración, primero se permutan los elementos de las columnas de la matriz L construida en la iteración anterior, y se añaden los elementos de la columna correspondiente.
2. la matriz de permutación se va calculando también iterativamente, intercambiando las filas de la matriz de permutación obtenida en la iteración anterior.

Matlab posee un comando propio para calcular la factorización LU de una Matriz, $[L, U, P] = lu(A)$. Es importante pedir siempre que devuelva la matriz P, en otro caso, la matriz L devuelta por Matlab no tiene por qué ser triangular inferior.

El comando `lu` de Matlab, permite calcular la factorización LU, por otros métodos distintos a la eliminación gaussiana con pivoteo de filas. La explicación de estos otros métodos queda fuera del alcance de estos apuntes. Simplemente indicar que la factorización LU de una matriz no es única — es posible encontrar distintos pares L y U que factorizan a la misma matriz A.

5.5.2. Factorización de Cholesky

Dada una matriz cuadrada, simétrica y definida positiva, es siempre posible factorizarla como $A = L \cdot L^T$. Donde L es una matriz triangular inferior,

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & \cdots & L_{nn} \end{pmatrix} \cdot \begin{pmatrix} L_{11} & L_{21} & \cdots & L_{n1} \\ 0 & L_{22} & \cdots & L_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & L_{nn} \end{pmatrix}$$

Para obtener la factorización de Cholesky podemos emplear directamente la definición. Así si calculamos $L \cdot L^T$, obtendríamos $a_{11} = L_{11} \cdot L_{11} = L_{11}^2$, $a_{22} = L_{21} \cdot L_{21} + L_{22} \cdot L_{22} = L_{21}^2 + L_{22}^2$ etc. Es fácil comprobar que los elementos de la diagonal a_{ii} de la matriz A cumplen,

$$a_{ii} = \sum_{k=1}^j L_{ki}^2 = L_{ii} + \sum_{k=1}^{j-1} L_{ki}^2$$

A partir de esta expresión podemos despejar L_{ii} ,

$$L_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{j-1} L_{ki}^2}$$

Del mismo modo, para los elementos que no pertenecen a la diagonal,

$$a_{ij} = \sum_{k=1}^j L_{ik} \cdot L_{kj} = L_{ij} \cdot L_{jj} + \sum_{k=1}^{j-1} L_{ik} \cdot L_{kj}$$

A partir de esta expresión podemos despejar L_{ii} ,

$$L_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} L_{ik} \cdot L_{kj}}{L_{jj}}$$

Los elementos de la matriz L, se pueden calcular de las expresiones anteriores iterativamente por columnas: para obtener L_{11} basta emplear a_{11} , para obtener los restantes elementos de la primera columna, basta conocer L_{11} . Conocida la primera columna de L, se puede calcular L_{22} y así sucesivamente hasta completar toda la matriz. El siguiente código calcula la factorización de Cholesky de una matriz, siempre que cumpla las condiciones requeridas,

```
function L=cholesky(A)
%hacemos que devuelva la matriz triangular inferior.
%Comprobamos que es cuadrada, simetrica, y definida positiva.
[a,b]=size(A);
if a-b~=0
    disp('La matriz no es cuadrada')
else
    %ha resultado cuadrada comprobamos si es simetrica
    c=A-A';
    if any(c)
        disp('LA matriz no es simetrica')
    else
        %ha resultado simetrica, comprobamos si es definida positiva
        auto=eig(A);
        if any(auto<=0)
            disp('la matriz no es definida positiva')
        else
            %una vez que la matriz cumple las condiciones necesarias,
            %factorizamos por cholesky

            for k=1:a
                L(k,k)=A(k,k);
                for i=1:k-1
                    L(k,k)=L(k,k)-L(k,i)^2;
                end
                L(k,k)=sqrt(L(k,k));
                for j=k+1:a
                    L(j,k)=A(j,k);
                    for i=1:k-1

```

```

        L(j,k)=L(j,k)-L(j,i)*L(k,i);
    end
    L(j,k)=L(j,k)/L(k,k);
end
end
end
end
end

```

Matlab tiene una función interna `chol`, que permite obtener la factorización de Cholesky, Por defecto devuelve una matriz triangular superior U , de modo que la factorización calculada cumple $A = U^T \cdot U$. (En realidad se trata tan solo de una forma alternativa de definir la factorización de Cholesky), Así por ejemplo,

```

>> A
A =
    6     3     4
    3    14     3
    4     3     5
>> U=chol(A)
U =
    2.4495    1.2247    1.6330
         0    3.5355    0.2828
         0         0    1.5011
>> R=U'*U
R =
    6.0000    3.0000    4.0000
    3.0000   14.0000    3.0000
    4.0000    3.0000    5.0000

```

Si se quiere obtener directamente la matriz L , hay que indicarlo expresamente: `L=chol(A, 'lower')`. Para obtener la factorización Matlab emplea SOLO la parte triangular superior o la triangular inferior de la matriz A . Supone que A es simétrica pero NO lo comprueba. Si la matriz no es definida positiva, la función `chol` da un mensaje de error.

5.5.3. Diagonalización

Autovectores y autovalores. Dada una matriz A de orden $n \times n$, se define como autovector, o vector propio de dicha matriz al vector x que cumple,

$$A \cdot x = \lambda \cdot x, \quad x \neq 0, \quad \lambda \in \mathbb{C}$$

Es decir, el efecto de multiplicar la matriz A por el vector x es equivalente a multiplicar el vector x por un número λ que, en general, será un número complejo.

λ recibe el nombre de autovalor, o valor propio de la matriz A . Así por ejemplo,

$$A = \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix}$$

tiene un autovalor $\lambda = 3$ para el vector propio, $x = [0, -3, 3]^T$,

$$\begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ -3 \\ 3 \end{pmatrix} = 3 \cdot \begin{pmatrix} 0 \\ -3 \\ 3 \end{pmatrix} = \begin{pmatrix} 0 \\ -9 \\ 9 \end{pmatrix}$$

El vector propio asociado a un valor propio no es único, si x es un vector propio de una matriz A , asociado a un valor propio λ , Es trivial comprobar que cualquier vector de la forma $\alpha \cdot x$, $\alpha \in \mathbf{C}$ también es u vector propio asociado a λ ,

$$A \cdot x = \lambda x \Rightarrow A \cdot \alpha x = \lambda \alpha x \quad (5.1)$$

En realidad, cada vector propio expande un subespacio $E_\lambda S$ de \mathbf{R}^n . Aplicar la matriz A , a un vector de $E_\lambda S$ es equivalente a multiplicar el vector por λ . Cada subespacio asociado a un autovalor recibe el nombre de autosubespacio o subespacio propio.

El conjunto de todos los autovalores de una matriz A recibe el nombre de espectro de A y se representa como $\Lambda(A)$. Una descomposición en autovalores de una matriz A se define como.

$$A = X \cdot D \cdot X^{-1}$$

Donde D es una matriz diagonal formada por los autovalores de A .

$$\begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

Esta descomposición no siempre existe. En general, si dos matrices A y B cumplen,

$$A = X \cdot B \cdot X^{-1}$$

se dice de ellas que son similares. A la transformación que lleva a convertir B en A se le llama una transformación de semejanza. Una matriz es diagonalizable cuando admite una transformación de semejanza con una matriz diagonal de autovalores.

Volviendo al ejemplo anterior podemos factorizar la matriz A como,

$$A = \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -3 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} -2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 1/3 \end{pmatrix}$$

Por tanto en este ejemplo, los autovalores de A serían $\lambda_1 = -2$, $\lambda_2 = 2$ y $\lambda_3 = 3$. Para estudiar la composición de la matriz X Reescribimos la expresión de la relación de semejanza de la siguiente manera,

$$A = X \cdot D \cdot X^{-1} \rightarrow A \cdot X = X \cdot D$$

Si consideramos ahora cada columna de la matriz X como un vector,

$$A \cdot X = X \cdot D \rightarrow A \cdot (x_1 | x_2 | \cdots | x_n) = (x_1 | x_2 | \cdots | x_n) \cdot \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

Es fácil comprobar que si consideramos el producto de la matriz A , por cada una de las columnas de la matriz X se cumple,

$$\begin{aligned} A \cdot x_1 &= \lambda_1 \cdot x_1 \\ A \cdot x_2 &= \lambda_2 \cdot x_2 \\ &\vdots \\ A \cdot x_n &= \lambda_n \cdot x_n \end{aligned}$$

Lo que nos lleva a que cada columna de la matriz X tiene que ser un autovector de A puesto que cumple,

$$A \cdot x_i = \lambda_i \cdot x_i$$

Polinomio característico. El polinomio característico de una matriz A de dimensión $n \times n$, se define como,

$$P_A(z) = \det(zI - A)$$

Donde I es a matriz identidad de dimensión $n \times n$. Así por ejemplo para una matriz de dimensión 2×2 ,

$$\begin{aligned} P_A(z) &= \det \left(z \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \right) = \begin{vmatrix} z - a_{11} & -a_{12} \\ -a_{21} & z - a_{22} \end{vmatrix} = \\ &= (z - a_{11}) \cdot (z - a_{22}) - a_{12} \cdot a_{21} = z^2 - (a_{11} + a_{22}) \cdot z + a_{11} \cdot a_{22} - a_{12} \cdot a_{21} \end{aligned}$$

Los autovalores λ de una matriz A son las raíces del polinomio característico de la matriz A ,

$$p_A(\lambda) = 0$$

Las raíces de un polinomio pueden ser simples o múltiples, es decir una misma raíz puede repetirse varias veces. Por tanto, los autovalores de una matriz, pueden también repetirse. Se llama multiplicidad algebraica de un autovalor al número de veces que se repite.

Además las raíces de un polinomio pueden ser reales o complejas. Para una matriz cuyos elementos son todos reales, si tiene autovalores complejos estos se dan siempre como pares de números complejos conjugados. Es decir si $\lambda = a + bi$ es un autovalor entonces $\lambda^* = a - bi$ también lo es.

Veamos algunos ejemplos:

La matriz,

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

Tiene como polinomio característico,

$$P_A(z) = \begin{vmatrix} z - 3 & -2 \\ 2 & z + 2 \end{vmatrix} = z^2 - z - 2$$

Igualando el polinomio característico a cero y obteniendo las raíces de la ecuación de segundo grado resultante, obtenemos los autovalores de la matriz A ,

$$\lambda^2 - \lambda - 2 = 0 \rightarrow \begin{cases} \lambda_1 = 2 \\ \lambda_2 = -1 \end{cases}$$

Un vector propio asociado a $\lambda_1 = 2$ sería $x_1 = [2, -1]^T$,

$$\begin{pmatrix} 3 & 2 \\ -2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ -1 \end{pmatrix} = 2 \cdot \begin{pmatrix} 2 \\ -1 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

y un vector propio asociado a $\lambda_2 = -1$ sería $x_2 = [1, -2]^T$,

$$\begin{pmatrix} 3 & 2 \\ -2 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -2 \end{pmatrix} = -1 \cdot \begin{pmatrix} 1 \\ -2 \end{pmatrix} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

La matriz,

$$B = \begin{pmatrix} 4 & -1 \\ 1 & 2 \end{pmatrix}$$

Tiene como polinomio característico,

$$P_B(z) = \begin{vmatrix} z-4 & -1 \\ 1 & z-2 \end{vmatrix} = z^2 - 6z + 9$$

procediendo igual que en el caso anterior, obtenemos los autovalores de la matriz B ,

$$\lambda^2 - 6\lambda + 9 = 0 \rightarrow \begin{cases} \lambda_1 = 3 \\ \lambda_2 = 3 \end{cases}$$

En este caso, hemos obtenido para el polinomio característico una raíz doble, con lo que obtenemos un único autovalor $\lambda = 3$ de multiplicidad algebraica 2.

La matriz,

$$C = \begin{pmatrix} 2 & -1 \\ 1 & 2 \end{pmatrix}$$

tiene como polinomio característico,

$$P_C(z) = \begin{vmatrix} z-2 & 1 \\ -1 & z-2 \end{vmatrix} = z^2 - 4z + 5$$

Con lo que sus autovalores resultan ser dos números complejos conjugados,

$$\lambda^2 - 4\lambda + 5 = 0 \rightarrow \begin{cases} \lambda_1 = 2 + i \\ \lambda_2 = 2 - i \end{cases}$$

Para que una matriz A de orden $n \times n$ sea diagonalizable es preciso que cada autovalor tenga asociado un número de autovectores linealmente independientes, igual a su multiplicidad algebraica. La matriz B del ejemplo mostrado más arriba tiene tan solo un autovector, $x = [1, 1]^T$ asociado a su único autovalor de multiplicidad 2. No es posible encontrar otro linealmente independiente; por lo tanto B no es diagonalizable.

La matriz,

$$G = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Tiene un autovalor $\lambda_1 = 3$, de multiplicidad 2, y un autovalor $\lambda_2 = 1$ de multiplicidad 1. Para el autovalor de multiplicidad 2 es posible encontrar dos autovectores linealmente independientes, por ejemplo: $x_1 = [1, 0, 0]^T$ y $x_2 = [0, 1, 0]^T$. Para el segundo autovalor un posible autovector sería, $x_3 = [-2, 0, 1]^T$. Es posible por tanto diagonalizar la matriz G ,

$$G = \begin{pmatrix} 3 & 0 & 1 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} = X \cdot D \cdot X^{-1} = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Propiedades asociadas a los autovalores. Damos a continuación, sin demostración, algunas propiedades de los autovalores de una matriz,

1. La suma de los autovalores de una matriz, coincide con su traza,

$$\text{tr}(A) = \sum_{i=1}^n \lambda_i$$

2. El producto de los autovalores de una matriz coincide con su determinante,

$$|A| = \prod_{i=1}^n \lambda_i$$

3. Cuando los autovectores de una matriz A son ortogonales entre sí, entonces la matriz A es diagonalizable ortogonalmente,

$$A = Q \cdot D \cdot Q^{-1}; \quad Q^{-1} = Q^T \Rightarrow A = Q \cdot A \cdot Q^T$$

Cualquier matriz simétrica posee autovalores reales y es diagonalizable ortogonalmente. En general, una matriz es diagonalizable ortogonalmente si es normal: $A \cdot A^T = A^T \cdot A$

4. El mayor de los autovalores en valor absoluto, de una matriz A de orden n . recibe el nombre de *radiopectral* de dicha matriz,

$$\rho(A) = \max_{i=1}^n |\lambda_i|$$

No vamos a ver ningún algoritmo concreto para diagonalizar matrices. Quizá el mas utilizado, por su robustez numérica es el basado en la factorización de Schur: cualquier matriz A puede factorizarse como,

$$A = Q \cdot T \cdot Q^T$$

Donde Q es una matriz ortogonal y T es una matriz triangular superior. Los elementos de la diagonal de T son precisamente los autovalores de A .

Matlab incluye la función `eig` para calcular los autovectores y autovalores de una matriz. esa función admite como variable de entrada una matriz cuadrada de dimensión arbitraria y devuelve como variable de salida un vector columna con los autovalores de la matriz de entrada,

```
>> A=[1 2 3;2 3 -2;3 0 1]
```

```
A =
```

$$\begin{matrix} 1 & 2 & 3 \\ 2 & 3 & -2 \\ 3 & 0 & 1 \end{matrix}$$

```
>> Lambda=eig(A)
```

```
Lambda =
```

```
-2.7016
4.0000
3.7016
```

También puede llamarse a la función `eig` con dos variables de salida. En este caso la primera variable de salida es una matriz en la que cada columna representa un autovector de la matriz de entrada normalizado —de módulo 1—. La segunda variable de salida es una matriz diagonal cuyos elementos son los autovalores de la matriz de entrada.

```
>> A=[1 2 3;2 3 -2;3 0 1]
```

```
A =
```

```
1      2      3
2      3     -2
3      0      1
```

```
>> [X,D]=eig(A)
```

```
X =
```

```
-0.6967    0.7071    0.6548
0.4425    0.0000   -0.2062
0.5647    0.7071    0.7271
```

```
D =
```

```
-2.7016      0      0
0      4.0000      0
0          0    3.7016
```

Por último, Matlab también incluye una función para calcular la factorización de Schur. Si lo aplicamos a la misma matriz `A`, para la que acabamos de calcular los autovalores,

```
>> [Q,T]=schur(A)
```

```
Q =
```

```
-0.6967    0.6449   -0.3142
0.4425    0.0415   -0.8958
0.5647    0.7632    0.3142
```

```
T =
```

```
-2.7016   -0.6285   -1.2897
0        4.0000   -1.3934
0          0      3.7016
```

```
>> Q*Q'
ans =
1.0000    0.0000    0.0000
0.0000    1.0000    0.0000
0.0000    0.0000    1.0000
```

Se observa como los elementos de la diagonal de T son los autovalores de A como la matriz Q es ortogonal.

5.5.4. Factorización QR

La idea es factorizar una matriz A en el producto de dos matrices; una matriz ortogonal Q y una matriz triangular superior R .

$$A = Q \cdot R, \leftarrow Q \cdot Q^T = I$$

Supongamos que consideramos cada columna de A y cada columna de Q como un vector,

$$A = (a_1 | a_2 | \cdots | a_n), \quad Q = (q_1 | q_2 | \cdots | q_n)$$

Podemos ahora considerar el producto $Q \cdot R$ como combinaciones lineales de las columnas de Q , que dan como resultado las columnas de A ,

$$\begin{aligned} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} &= \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ q_{n1} & q_{n2} & \cdots & q_{nn} \end{pmatrix} \cdot \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \Rightarrow \\ a_1 = q_1 \cdot r_{11} \rightarrow \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix} &= \begin{pmatrix} q_{11} \\ q_{21} \\ \vdots \\ q_{n1} \end{pmatrix} \cdot r_{11}, \\ a_2 = q_1 \cdot r_{12} + q_2 \cdot r_{22} \rightarrow \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{pmatrix} &= \begin{pmatrix} q_{11} \\ q_{21} \\ \vdots \\ q_{n1} \end{pmatrix} \cdot r_{12} + \begin{pmatrix} q_{12} \\ q_{22} \\ \vdots \\ q_{n2} \end{pmatrix} \cdot r_{22} \\ &\vdots \\ a_n = q_1 \cdot r_{1n} + q_2 \cdot r_{2n} + \cdots + r_{nn} \rightarrow \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{nn} \end{pmatrix} &= \begin{pmatrix} q_{11} \\ q_{21} \\ \vdots \\ q_{n1} \end{pmatrix} \cdot r_{1n} + \begin{pmatrix} q_{12} \\ q_{22} \\ \vdots \\ q_{n2} \end{pmatrix} \cdot r_{2n} + \cdots + \begin{pmatrix} q_{1n} \\ q_{2n} \\ \vdots \\ q_{nn} \end{pmatrix} \cdot r_{nn} \end{aligned}$$

Para que la matriz Q sea ortogonal, basta hacer que sus columnas, consideradas cada una como un vector, sean ortonormales entre sí. Podemos obtener la matriz Q a partir de la matriz A , aplicando a sus columnas algún método de ortogonalización de vectores. En particular, emplearemos el método conocido como ortogonalización de Gram-Schmidt.

Ortogonalización de Grand-Schmidt Este método permite transformar un conjunto de vectores linealmente independientes arbitrarios en un conjunto de vectores ortonormales. Lo describiremos a la vez que estudiamos su uso para obtener la factorización QR de una matriz A .

En primer lugar, consideremos el vector a_1 , formado por los elementos de la primera columna de la matriz A . Si lo dividimos por su módulo, obtendremos un vector unitario. Este vector, será la primera columna de la matriz Q ,

$$q_1 = \frac{a_1}{\|a_1\|}$$

De la expresión anterior es fácil deducir quien sería el elemento r_{11} de la matriz R ,

$$a_1 = q_1 \cdot \|a_1\| \Rightarrow r_{11} = \|a_1\|$$

Supongamos por ejemplo que queremos factorizar la siguiente matriz,

$$A = \begin{pmatrix} 2 & 3 & 1 \\ 2 & 0 & 2 \\ 1 & 4 & 3 \end{pmatrix}$$

La primera columna de la matriz Q sería,

$$q_1 = \frac{a_1}{\|a_1\|} = \frac{1}{3} \cdot \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2/3 \\ 2/3 \\ 1/3 \end{pmatrix}$$

y el elemento r_{11} de la matriz R ,

$$r_{11} = \|a_1\| = 3$$

Para obtener la segunda columna Q , Calculamos en primer lugar la proyección de la segunda columna de A , a_2 sobre el vector q_1 . En general, la proyección de un vector a , sobre un vector b se calcula obteniendo el producto escalar de los dos vectores y dividiendo el resultado por el módulo del vector b . En nuestro caso el vector q_1 , sobre el que se calcula la proyección, es unitario, por lo que calcular la proyección de a_2 sobre q_1 se reduce a calcular el producto escalar de los dos vectores,

$$\phi_{a_2/q_1} = q_1^T \cdot a_2$$

Siguiendo con nuestro ejemplo,

$$\phi_{a_2/q_1} = q_1^T \cdot a_2 = (2/3 \quad 2/3 \quad 1/3) \cdot \begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix} = \frac{10}{3}$$

Podemos interpretar la proyección del vector a_2 sobre el vector q_1 como la *componente* del vector a_2 en la dirección de q_1 . Por eso, si restamos al vector a_2 su proyección sobre q_1 multiplicada por q_1 el resultado será un nuevo vector q_2 ortogonal a q_1 ,

$$v_2 = a_2 - \phi_{a_2/q_1} \cdot q_1 = a_2 - (q_1^T \cdot a_2) \cdot q_1$$

La figura 5.10 muestra gráficamente el proceso para dos vectores en el plano.

Como q_2 tiene que ser un vector unitario para que la matriz Q sea ortogonal, dividimos el vector obtenido, q'_2 , por su módulo,

$$q_2 = \frac{V_2}{\|v_2\|} = \frac{a_2 - (q_1^T \cdot a_2) \cdot q_1}{\|a_2 - (q_1^T \cdot a_2) \cdot q_1\|}$$

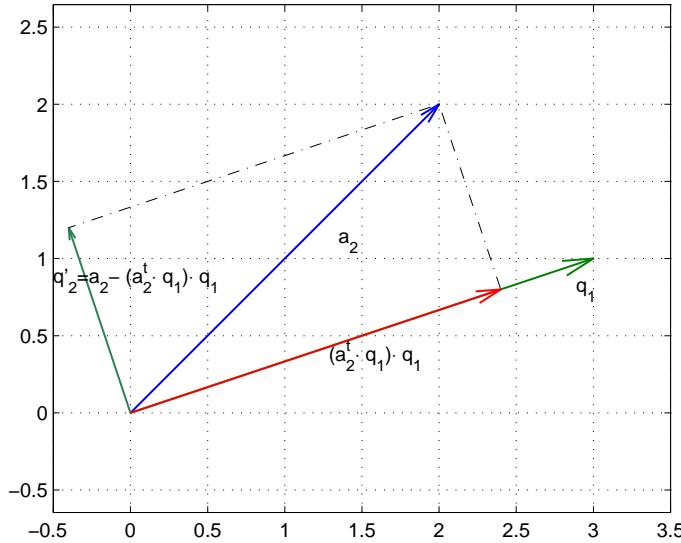


Figura 5.10: Obtención de un vector ortogonal

Por último despejando a_2 podemos identificar los valores de r_{12} y r_{22} , tal y como se describieron anteriormente,

$$\begin{aligned} a_2 &= (q_1^T \cdot a_2) \cdot q_1 + \|a_2 - (q_1^T \cdot a_2) \cdot q_1\| \cdot q_2 \\ r_{12} &= q_1^T \cdot a_2 \\ r_{22} &= \|a_2 - (q_1^T \cdot a_2) \cdot q_1\| \end{aligned}$$

Volviendo a la matriz A de nuestro ejemplo la segunda columna de la matriz Q y los valores de r_{12} y r_{22} quedarían,

$$\begin{aligned} q_2 &= \frac{v_2}{\|v_2\|} = \frac{a_2 - (q_1^T \cdot a_2) \cdot q_1}{\|a_2 - (q_1^T \cdot a_2) \cdot q_1\|} = \left[\begin{pmatrix} 3 \\ 0 \\ 4 \end{pmatrix} - \frac{10}{3} \cdot \begin{pmatrix} 2/3 \\ 2/3 \\ 1/3 \end{pmatrix} \right] \cdot \frac{3}{5\sqrt{5}} = \frac{1}{5\sqrt{5}} \begin{pmatrix} 7/3 \\ -20/3 \\ 26/3 \end{pmatrix} \\ r_{12} &= \frac{10}{3} \\ r_{22} &= \frac{5\sqrt{5}}{3} \end{aligned}$$

Se puede comprobar que los vectores q_1 y q_2 son ortogonales,

$$q_1^T \cdot q_2 = (2/3 \quad 2/3 \quad 27/3) \cdot 5\sqrt{5} \begin{pmatrix} 7/3 \\ -20/3 \\ 26/3 \end{pmatrix} = 5\sqrt{5} \cdot \left(\frac{14}{3} - \frac{40}{3} + \frac{26}{3} \right) = 0$$

Para obtener la tercera columna de la matriz Q , q_3 procederíamos de modo análogo: calcularíamos la proyección de la tercera columna de A , a_3 sobre los vectores formados por la los dos

primeras columnas de Q , restamos de a_3 las dos proyecciones y normalizamos el vector resultante dividiéndolo por su módulo,

$$q_3 = \frac{v_3}{\|v_3\|} = \frac{a_3 - (q_1^T \cdot a_3) \cdot q_1 - (q_2^T \cdot a_3) \cdot q_2}{\|a_3 - (q_1^T \cdot a_3) \cdot q_1 - (q_2^T \cdot a_3) \cdot q_2\|}$$

y de modo análogo al caso de la segunda columna,

$$\begin{aligned} r_{13} &= q_1^T \cdot a_3 \\ r_{23} &= q_2^T \cdot a_3 \\ r_{33} &= \|a_3 - (q_1^T \cdot a_3) \cdot q_1 - (q_2^T \cdot a_3) \cdot q_2\| \end{aligned}$$

Es fácil observar como vamos obteniendo las sucesivas columnas de la matriz Q , iterativamente. Podemos generalizar los resultados anteriores para cualquier columna arbitraria de la matriz Q ,

$$q_i = \frac{a_i - \sum_{j=1}^i (q_j^T \cdot a_i) \cdot q_j}{\|a_i - \sum_{j=1}^i (q_j^T \cdot a_i) \cdot q_j\|}$$

y para los valores de $r_{1i}, r_{2i}, \dots, r_{ii}$,

$$\begin{aligned} r_{ji} &= q_j^T \cdot a_i, \quad j < i \\ r_{ii} &= \|a_i - \sum_{j=1}^i (q_j^T \cdot a_i) \cdot q_j\| \end{aligned}$$

El siguiente código de Matlab, calcula la factorización QR de una matriz, empleando el método descrito,

```
function [Q,R]=QRF1(A)
%%%%%
%Factorización QR obtenida directamente por ortogonalización de grand-schmidt
%Ojo, el algoritmo es inestable... Con lo que la bondad de las soluciones
%dependera de la matriz que se quiera factorizar.
%%%%%
%En primer lugar obtenemos las dimensiones de la matriz
[m,n]=size(A);

%fatorizamos columna a columna
for j=1:n
    %Construimos un vector auxiliar v, nos servira para ir obteniendo las
    %columnas de la matriz Q.
    for i=1:m %Solo llegamos hasta m factorización incompleta si m>n
        v(i)=A(i,j)
    end
    for i=1:j-1
        %obtenemos los elementos de la matriz R, correspondientes a la
        %columna j, solo podemos construir hasta una fila antes de la
        %diagonal i=j-1. cada fila es el producto escalar de la columna i
        %de la matriz Q for la columna j de la Matriz A.
        R(i,j)=0
    end
end
```

```

for k=1:m
R(i,j)=R(i,j)+Q(k,i)*A(k,j)
end

%obtenemos las componentes del vector auxiliar que nos permitira
%construir la columna j de la matriz Q
for k=1:m
    v(k)=v(k)-R(i,j)*Q(k,i)
end
end
%Obtenemos el valor del elemento de la diagonal R(j,j) de la matriz R
R(j,j)=0
for k=1:m
    R(j,j)=R(j,j)+v(k)^2
end
R(j,j)=sqrt(R(j,j))

%Y por ultimo, divimos los elementos del vector v por R(j,j), para
%obtener la columna j de la matriz Q
for k=1:m
    Q(k,j)=v(k)/R(j,j)
end
end

```

En general, el algoritmo que acabamos de describir para obtener la ortogonalización de Grand-Schmidt, es numéricamente inestable. La estabilidad puede mejorarse, si vamos modificando progresivamente la matriz A , a medida que calculamos las columnas de Q .

Cada vez que obtenemos una nueva columna de Q , modificamos las columnas de A de modo que sean ortogonales a la columna de Q obtenida. Para ello, lo más sencillo es crear una matriz auxiliar V que hacemos, inicialmente, igual a A , $V^{(0)} = A$

para obtener la primera columna de Q , procedemos igual que antes, normalizando la primera columna de $V^{(0)}$

$$q_1 = \frac{v_1^{(0)}}{\|v_1^{(0)}\|}$$

$$r_{11} = \|v_1^{(0)}\|$$

A continuación calculamos la proyección de todas las demás columnas de la matriz $V^{(0)}$ con respecto a q_1 , esto es equivalente a calcular los restantes elementos de la primera fila de la matriz R : $r_{21}, r_{31}, \dots, r_{n1}$,

$$r_{1j} = q_1^T \cdot v_j^{(0)}$$

Una vez calculadas las proyecciones, modificamos todas las columnas de $V^{(0)}$, excepto la primera restando a cada una su proyección con respecto a q_1 .

$$v_j^{(1)} = v_j^{(0)} - r_{1j} \cdot v_j^{(0)}, \quad j = 2, 3, \dots, n$$

La nueva matriz $V^{(1)}$ cumple que todas sus columnas a partir de la segunda son ortogonales a q_1 . Para obtener q_2 es suficiente dividir $v_2^{(1)}$ por su módulo,

$$q_2 = \frac{v_2^{(1)}}{\|v_2^{(1)}\|}$$

$$r_{22} = \|v_2^{(1)}\|$$

Podemos ahora calcular el resto de los elementos de la segunda fila de la matriz R de modo análogo a como hemos calculado los de la primera,

$$r_{2j} = q_2^T \cdot v_j^{(1)}$$

y, de nuevo actualizariamos todos las columnas de V^1 , a partir de la tercera, para que fueran ortogonales a q_2 ,

$$v_j^{(2)} = v_j^{(1)} - r_{2j} \cdot v_j^{(1)}, \quad j = 3, 4, \dots, n$$

Las columnas de $V^{(2)}$ serían ahora ortogonales a q_1 y q_2 . Si seguimos el mismo procedimiento n veces, calculando cada vez una nueva columna de Q y una fila de R , obtenemos finalmente la factorización QR de la matriz inicial A . En general, para obtener la columna q_i y los elementos de la fila i de la matriz R tendríamos,

$$q_i = \frac{v_i^{(i-1)}}{\|v_i^{(i-1)}\|}$$

$$r_{ii} = \|v_i^{(i-1)}\|$$

$$r_{ij} = q_i^T \cdot v_j^{(i-1)}$$

y la actualización correspondiente de la matriz V sería,

$$v_j^{(i)} = v_j^{(i-1)} - r_{ij} \cdot v_j^{(i-1)}, \quad j = i+1, i+2, \dots, n$$

El resultado es el mismo que el que se obtiene por el primer método descrito. La ventaja es que numéricamente es más estable.

El siguiente código de Matlab, calcula la factorización QR de una matriz de orden $n \times n$, empleando el método descrito,

```
function [Q,R]=QRF2(A)
%%%%%
%Calculo de factorizacion QR de la matriz A, mediante la ortogonalizacion de
%grand.schmidt modificada. Este algoritmo si que es estable...
%Obtenemos las dimensiones de A
%%%%%
[m,n]=size(A);

%Creamos una matriz auxiliar v sobre la que vamos a realizar la
%factorizacion. Se podria realizar directamente sobre A Machacando sus
%cOLUMNAS segun la factorizacion progres... Seria lo correcto para ahorrar
%espacio de almacenamiento. Pero en fin, quizas asi queda mas claro aunque
%sea menos eficiente.
```

```

v=A;
%Como siempre, vamos factorizando por columnas de la matriz A

for i=1:m %la matriz Q tiene que ser mXm, aunque el numero de columnas de A sea n)
    %calculamos cada R(i,i) como el modulo del vector auxiliar v(1:m,i)
    R(i,i)=0;
    for k=1:m
        R(i,i)=R(i,i)+v(k,i)^2;
    end
    R(i,i)=sqrt(R(i,i));
    %calculamos el la columna i de la matriz Q, normalizando la columna i
    %de la matriz v
    for k=1:m
        Q(k,i)=v(k,i)/R(i,i);
    end
    %Modificamos todos los R(i,j) con ij>i, en cuanto tenemos la columna j
    %de la matriz Q, nos basta calcular el producto escalar con las
    %columnas de A (En nuestro caso de v porque están copiadas, de las
    %filas siguientes
    for j=i+1:n
        R(i,j)=0;
        for k=1:m
            R(i,j)=R(i,j)+Q(k,i)*v(k,j);
        end
        %i por último modificamos todas las columnas de la matriz v desde
        %i+1 hasta el final de la matriz. Aquí es donde cambia el algoritmo
        %ya que estamos modificando la matriz A, y las sucesivas matrices V
        %cada vez que obtenemos una nueva fila de valores de R
        for k=1:m
            v(k,j)=v(k,j)-R(i,j)*Q(k,i);
        end
    end
end

```

Para terminar, indicar que Matlab tiene su propia función, $[Q, R] = qr(A)$, para calcular la factorización QR de una matriz. Para calcularla, emplea el método de ortogonalización de Householder. Este método es aún más robusto que la ortogonalización de Gram-Schmidt modificada. Pero no lo veremos en este curso. Damos a continuación un ejemplo de uso de la función `qr`,

```
>> A=[2 3 1;2 0 2;1 4 3]
```

```
A =
```

2	3	1
2	0	2
1	4	3

```
>> [q,r]=qr(A)
```

```
q =
```

```

-0.6667    0.2087   -0.7155
-0.6667   -0.5963    0.4472
-0.3333    0.7752    0.5367

r =

```

```

-3.0000   -3.3333   -3.0000
      0     3.7268    1.3416
      0         0    1.7889

```

```
>> q*r
```

```
ans =
```

```

2.0000    3.0000    1.0000
2.0000    0.0000    2.0000
1.0000    4.0000    3.0000

```

5.5.5. Factorización SVD

Dada una matriz cualquiera A de orden $m \times n$ es posible factorizarla en el producto de tres matrices,

$$A = U \cdot S \cdot V^T$$

Donde U es una matriz de orden $m \times m$ ortogonal, V es una matriz de orden $n \times n$ ortogonal y S es una matriz diagonal de orden $m \times n$. Además los elementos de S son positivos o cero y están ordenados en orden no creciente,

$$S = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_i \end{pmatrix}; \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_i; i = \min(m, n)$$

Los elementos de la diagonal de la matriz S , $(\sigma_1, \sigma_2, \dots, \sigma_i)$, reciben el nombre de *valores singulares* de la matriz A . De ahí el nombre que recibe esta factorización; SVD son las siglas en inglés de *Singular Value Decomposition*.

No vamos a describir ningún algoritmo para obtener la factorización SVD de una matriz. En Matlab existe la función `[U,S,V]=svd(A)` que permite obtener directamente la factorización SVD de una matriz A de dimensión arbitraria. A continuación se incluyen unos ejemplos de uso para matrices no cuadradas,

```
>> A=[1 3 4;2 3 2;2 4 5;3 2 3]
```

```
A =
```

```

1      3      4
2      3      2
2      4      5

```

```

3      2      3

>> [U,S,V]=svd(A)

U =

```

-0.4877	0.5175	0.1164	-0.6934
-0.3860	-0.3612	-0.8375	-0.1387
-0.6517	0.3024	0.0552	0.6934
-0.4340	-0.7144	0.5311	-0.1387

```

S =

```

10.2545	0	0	
0	1.9011	0	
0	0	1.1097	
0	0	0	

```

V =

```

-0.3769	-0.9170	0.1307	
-0.5945	0.1313	-0.7933	
-0.7103	0.3767	0.5946	

Como la matriz A tiene más filas que columnas, la matriz S resultante termina con una fila de ceros.

```

>> B=A'

B =

```

1	2	2	3
3	3	4	2
4	2	5	3

```

>> [U,S,V]=svd(B)

U =

```

-0.3769	-0.9170	0.1307	
-0.5945	0.1313	-0.7933	
-0.7103	0.3767	0.5946	

```

S =

```

10.2545	0	0	0
0	1.9011	0	0

$$0 \quad 0 \quad 1.1097 \quad 0$$

$V =$

$$\begin{array}{cccc} -0.4877 & 0.5175 & 0.1164 & -0.6934 \\ -0.3860 & -0.3612 & -0.8375 & -0.1387 \\ -0.6517 & 0.3024 & 0.0552 & 0.6934 \\ -0.4340 & -0.7144 & 0.5311 & -0.1387 \end{array}$$

Como la matriz B , transpuesta de la matriz A del ejemplo anterior, tiene más columnas que filas, la matriz S termina con una columna de ceros.

A continuación enunciamos sin demostración algunas propiedades de la factorización SVD.

1. El rango de una matriz A coincide con el número de sus valores singulares distintos de cero.
2. La norma-2 inducida de una matriz A coincide con su valor singular mayor σ_1 .
3. La norma de Frobenius de una matriz A cumple:

$$\|A\|_F = \sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2}$$

4. Los valores singulares de una matriz A distintos de cero son iguales a la raíz cuadrada positiva de los autovalores distintos de cero de las matrices $A \cdot A^T$ ó $A^T \cdot A$. (los autovalores distintos de cero de estas dos matrices son iguales),

$$\sigma_i^2 = \lambda_i(A \cdot A^T) = \lambda_i(A^T \cdot A)$$

5. El valor absoluto del determinante de una matriz cuadrada $A, n \times n$, coincide con el producto de sus valores singulares,

$$|\det(A)| = \prod_{i=1}^n \sigma_i$$

6. El número de condición de una matriz cuadrada $A n \cdot n$, que se define como el producto de la norma-2 inducida de A por la norma-2 inducida de la inversa de A , puede expresarse como el cociente entre el el valor singular mayor de A y su valor singular más pequeño,

$$k(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \sigma_1 \cdot \frac{1}{\sigma_n} = \frac{\sigma_1}{\sigma_n}$$

El número de condición de una matriz, es una propiedad importante que permite estimar cómo de estables serán los cálculos realizados empleando dicha matriz, en particular aquellos que involucran directa o indirectamente el cálculo de su inversa.

Capítulo 6

Sistemas de ecuaciones lineales

6.1. Introducción

Una ecuación lineal es aquella que establece una relación *lineal* entre dos o más variables, por ejemplo,

$$3x_1 - 2x_2 = 12$$

Se dice que es una relación lineal, porque las variables están relacionadas entre sí tan solo mediante sumas y productos por coeficientes constantes. En particular, el ejemplo anterior puede representarse geométricamente mediante una línea recta.

El número de variables relacionadas en una ecuación lineal determina la dimensión de la ecuación. La del ejemplo anterior es una ecuación bidimensional, puesto que hay dos variables. El número puede ser arbitrariamente grande en general,

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

será una ecuación n-dimensional.

Como ya hemos señalado más arriba, una ecuación bidimensional admite una línea recta como representación geométrica, una ecuación tridimensional admitirá un plano y para dimensiones mayores que tres cada ecuación representará un hiperplano de dimensión n. Por supuesto, para dimensiones mayores que tres, no es posible obtener una representación gráfica de la ecuación.

Las ecuaciones lineales juegan un papel muy importante en la física y, en general en la ciencia y la tecnología. La razón es que constituyen la aproximación matemática más sencilla a la relación entre magnitudes físicas. Por ejemplo cuando decimos que la fuerza aplicada a un resorte y la elongación que sufre están relacionadas por la ley de Hooke, $F = Kx$ estamos estableciendo una relación lineal entre las magnitudes fuerza y elongación. ¿Se cumple siempre dicha relación? Desde luego que no. Pero es razonablemente cierta para elongaciones pequeñas y, apoyados en ese sencillo modelo *lineal* de la realidad, se puede aprender mucha física.

Un sistema de ecuaciones lineales está constituido por varias ecuaciones lineales, que expresan relaciones lineales distintas sobre las mismas variables. Por ejemplo,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

Se llaman soluciones del sistema de ecuaciones a los valores de las variables que satisfacen simultáneamente a todas las ecuaciones que componen el sistema. Desde el punto de vista de la

obtención de las soluciones a las variables se les suele denominar incógnitas, es decir valores no conocidos que deseamos obtener o calcular.

Un sistema de ecuaciones puede tener infinitas soluciones, puede tener una única solución o puede no tener solución. En lo que sigue, nos centraremos en sistemas de ecuaciones que tienen una única solución.

Una primera condición para que un sistema de ecuaciones tengan una única solución es que el número de incógnitas presentes en el sistema coincida con el número de ecuaciones.

De modo general podemos decir que vamos a estudiar métodos numéricos para resolver con un computador sistemas de n ecuaciones con n incógnitas,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

Una de las grandes ventajas de los sistemas de ecuaciones lineales es que puede expresarse en forma de producto matricial,

$$\left. \begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

La matriz A recibe el nombre de matriz de coeficientes del sistema de ecuaciones, el vector x es el vector de incógnitas y el vector b es el vector de términos independientes. Para resolver un sistema de ecuaciones podríamos aplicar lo aprendido en el capítulo anterior sobre álgebra de matrices,

$$A \cdot x = b \Rightarrow x = A^{-1} \cdot b$$

Es decir, bastaría invertir la matriz de coeficientes y multiplicarla por la izquierda por el vector de coeficientes para obtener el vector de términos independientes. De aquí podemos deducir una segunda condición para que un sistema de ecuaciones tenga una solución única; Su matriz de coeficientes tiene que tener inversa. Veamos algunos ejemplos sencillos.

Tomaremos en primer lugar un sistema de dos ecuaciones con dos incógnitas,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 3x_1 - 2x_2 &= -1 \end{aligned}$$

Si expresamos el sistema en forma de producto de matrices obtenemos,

$$\begin{pmatrix} 4 & 1 \\ 3 & -2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 6 \\ -1 \end{pmatrix}$$

e invirtiendo la matriz de coeficientes y multiplicándola por el vector de términos independientes se llega al vector de soluciones del sistema,

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 & 1 \\ 3 & -2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 6 \\ -1 \end{pmatrix} = \begin{pmatrix} 2/11 & 1/11 \\ 3/11 & -4/11 \end{pmatrix} \begin{pmatrix} 6 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

En el ejemplo que acabamos de ver, cada ecuación corresponde a una recta en el plano, en la figura 6.1 se han representado dichas rectas gráficamente. El punto en que se cortan es precisamente la solución del sistema.

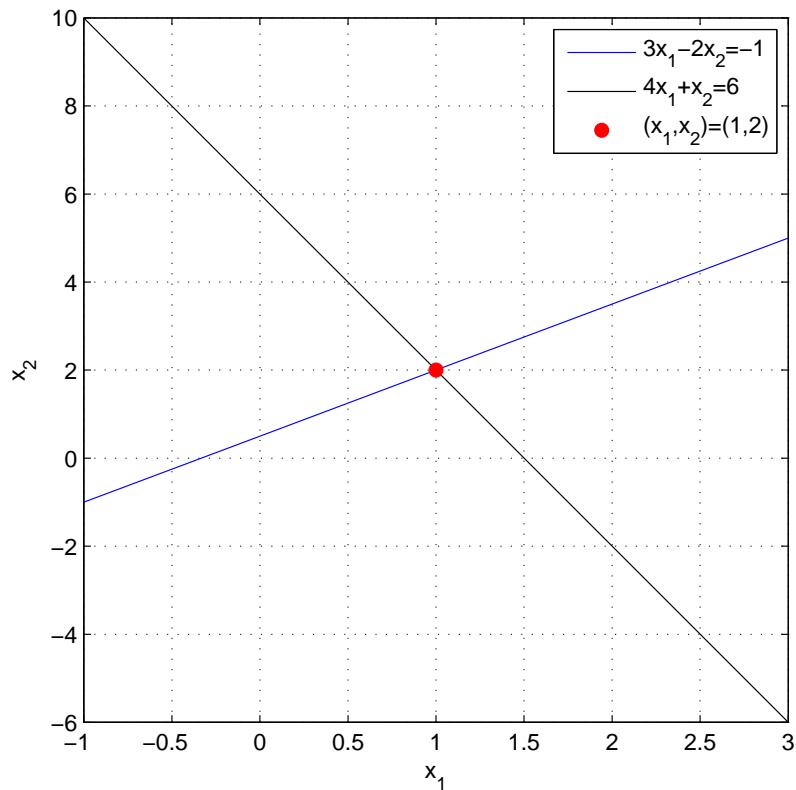


Figura 6.1: Sistema de ecuaciones con solución única

Supongamos ahora el siguiente sistema, también de dos ecuaciones con dos incógnitas,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + \frac{1}{2}x_2 &= -1 \end{aligned}$$

El sistema no tiene solución. Su matriz de coeficientes tiene determinante cero, por lo que no es invertible,

$$|A| = \begin{vmatrix} 4 & 1 \\ 2 & 1/2 \end{vmatrix} = 0 \Rightarrow \nexists A^{-1}$$

Si representamos gráficamente las dos ecuaciones de este sistema (figura 6.2) es fácil entender lo que pasa, las rectas son paralelas, no existe ningún punto (x_1, x_2) que pertenezca a las dos rectas, y por tanto el sistema carece de solución.

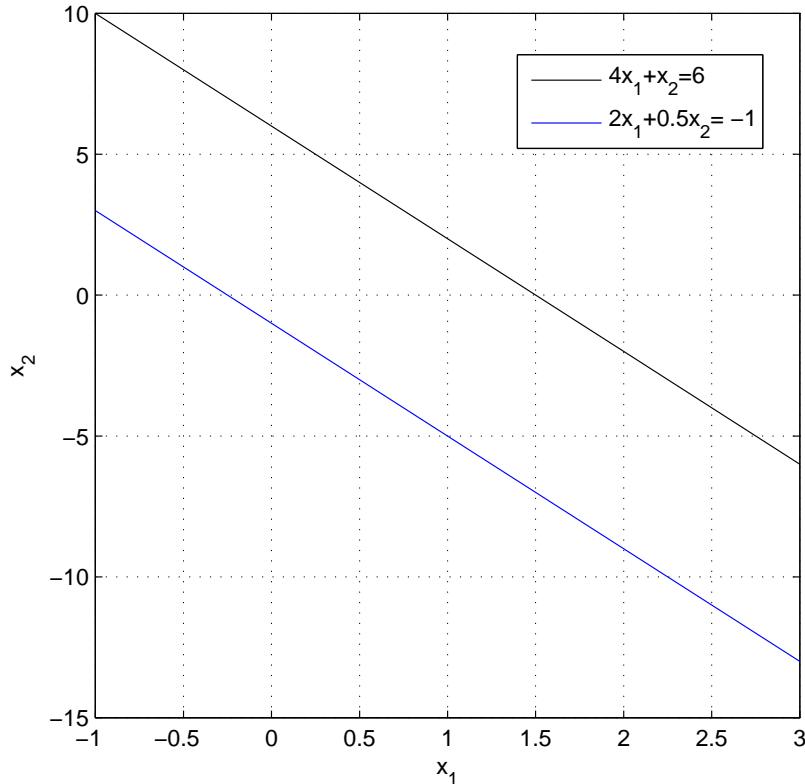


Figura 6.2: Sistema de ecuaciones sin solución

Dos rectas paralelas lo son, porque tienen la misma pendiente. Esto se refleja en la matriz de coeficientes, en que las filas son proporcionales; si multiplicamos la segunda fila por dos, obtenemos la primera.

Por último, el sistema,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + \frac{1}{2}x_2 &= 3 \end{aligned}$$

posee infinitas soluciones. la razón es que la segunda ecuación es igual que la primera multiplicada por dos: es decir, representa exactamente la misma relación lineal entre las variables x_1 y x_2 , por tanto, todos los puntos de la recta son solución del sistema. De nuevo, la matriz de coeficientes del sistema no tiene inversa ya que su determinante es cero.

Para sistemas de ecuaciones de dimensión mayor, se cumple también que que el sistema no tiene solución única si el determinante de su matriz de coeficiente es cero. En todos los demás casos, es posible obtener la solución del sistema invirtiendo la matriz de coeficientes y multiplicando el resultado por el vector de términos independientes.

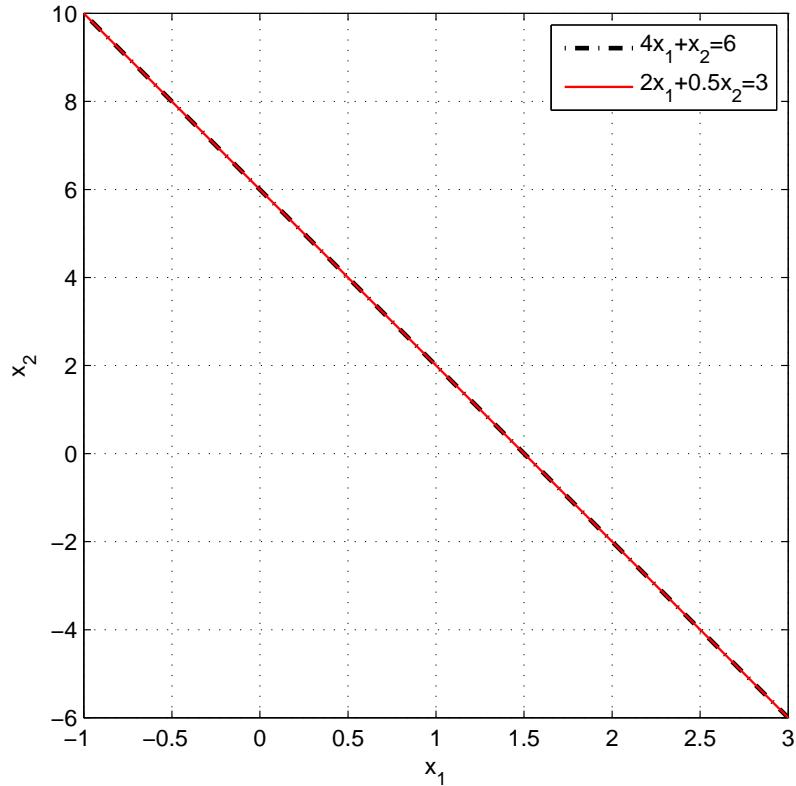


Figura 6.3: Sistema de ecuaciones con infinitas soluciones

En cuanto un sistema de ecuaciones tiene una dimensión suficientemente grande, invertir su matriz de coeficientes se torna un problema costoso o sencillamente inabordable.

Desde un punto de vista numérico, la inversión de una matriz, presenta frecuentemente problemas debido al error de redondeo en las operaciones. Por esto, casi nunca se resuelven los sistemas de ecuaciones invirtiendo su matriz de coeficientes. A lo largo de este capítulo estudiaremos dos tipos de métodos de resolución de sistemas de ecuaciones. El primero de ellos recibe el nombre genérico de métodos directos, el segundo tipo lo constituyen los llamados métodos iterativos.

6.2. Condicionamiento

En la introducción hemos visto que para que un sistema de ecuaciones tenga solución, es preciso que su matriz de coeficientes sea invertible. Sin embargo cuando tratamos de resolver un sistema de ecuaciones numéricamente, empleando un ordenador, debemos antes examinar cuidadosamente la matriz de coeficientes del sistema. Veamos un ejemplo: el sistema,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0,4x_2 &= -1 \end{aligned}$$

Tiene como soluciones,

$$x = \begin{pmatrix} -8,5 \\ 40 \end{pmatrix}$$

Si alteramos ligeramente uno de sus coeficientes,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0,49x_2 &= -1 \end{aligned}$$

Las soluciones se alteran bastante; se vuelven aproximadamente 10 veces más grande,

$$x = \begin{pmatrix} -98,5 \\ 400 \end{pmatrix}$$

y si volvemos a alterar el mismo coeficiente,

$$\begin{aligned} 4x_1 + x_2 &= 6 \\ 2x_1 + 0,499x_2 &= -1 \end{aligned}$$

La solución es aproximadamente 100 veces más grande,

$$x = \begin{pmatrix} -998,5 \\ 4000 \end{pmatrix}$$

La razón para estos cambios es fácil de comprender intuitivamente; a medida que aproximamos el coeficiente a 0,5, estamos haciendo que las dos ecuaciones lineales sean cada vez mas paralelas, pequeñas variaciones en la pendiente, modifican mucho la posición del punto de corte.

Cuando pequeñas variaciones en la matriz de coeficientes generan grandes variaciones en las soluciones del sistema, se dice que el sistema está mal condicionado, en otras palabras: que no es un sistema bueno para ser resuelto numéricamente. Las soluciones obtenidas para un sistema mal condicionado, hay que tomarlas siempre con bastante escepticismo.

Para estimar el buen o mal condicionamiento de un sistema, se emplea el número de condición, que definimos en el capítulo anterior, 5.5.5, al hablar de la factorización SVD. El número de condición de una matriz es el cociente entre sus valores singulares mayor y menor. Cuanto más próximo a 1 sea el número de condición, mejor condicionada estará la matriz y cuanto mayor sea el número de condición peor condicionada estará.

Matlab tiene un comando específico para obtener el número de condición de una matriz, sin tener que calcular la factorización SVD, el comando `nc=cond(A)`. Si lo aplicamos a la matriz de coeficientes del último ejemplo mostrado,

```
>> A=[4 1; 2, 0.499]
```

```
A =
```

```
4.0000    1.0000
```

```

2.0000    0.4990

>> nc=cond(A)

nc =
5.3123e+003

```

El número está bastante alejado de uno, lo que, en principio, indica un mal condicionamiento del sistema.

Incidentalmente, podemos calcular la factorización svd de la matriz de coeficientes y dividir el valor singular mayor entre el menor para comprobar que el resultado es el mismo que nos da la función `cond`,

```

>> [U,S,V]=svd(A)

U =
-0.8944   -0.4472
-0.4472    0.8944

S =
4.6097      0
0      0.0009

V =
-0.9702    0.2424
-0.2424   -0.9702

>> nc=S(1,1)/S(2,2)

nc =
5.3123e+003

```

Matlab emplea también la función `rnc=rcond(A)` para estimar el condicionamiento de una matriz. No describiremos el método, simplemente diremos que en lugar de obtener un número de condición para la matriz, se utiliza un valor recíproco. De este modo, cuanto más se aproxima a uno el resultado de `rcond`, mejor condicionada está la matriz, y cuanto más se aproxime a cero peor. Para nuestro ejemplo anterior,

```

>> rnc=rcond(A)

rnc =
1.3333e-004

```

Próximo a cero, lo que indica un mal condicionamiento de A . Nótese que el resultado de `rcond`, no es el inverso del valor de `cond`.

6.3. Métodos directos

6.3.1. Sistemas triangulares

Vamos a empezar el estudio de los métodos directos por los algoritmos de resolución de los sistemas más simples posibles, aquellos cuya matriz de coeficientes es una matriz diagonal, triangular superior o triangular inferior.

Sistemas diagonales. Un sistema diagonal es aquel cuya matriz de coeficientes es una matriz diagonal.

$$\left. \begin{array}{l} a_{11}x_1 = b_1 \\ a_{22}x_2 = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

Su resolución es trivial, basta dividir cada término independiente por el elemento correspondiente de la diagonal de la matriz de coeficientes,

$$x_i = \frac{b_i}{a_{ii}}$$

Para obtener la solución basta crear en Matlab un sencillo bucle `for`,

```
n=size(A,1);
x=zeros(n,1);
for i=1:n
    x(i)=b(i)/A(i,i);
end
```

Sistemas triangulares inferiores: método de sustituciones progresivas. Un sistema triangular inferior de n ecuaciones con n incógnitas tendrá la forma general,

$$\left. \begin{array}{l} a_{11}x_1 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

El procedimiento para resolverlo a mano es muy sencillo, despejamos la primera la primera incógnita de la primera ecuación,

$$x_1 = \frac{b_1}{a_{11}}$$

A continuación sustituimos este resultado en la segunda ecuación, y despejamos x_2 ,

$$x_2 = \frac{b_2 - a_{21}x_1}{a_{22}}$$

De cada ecuación vamos obteniendo una componente del vector solución, sustituyendo las soluciones obtenidas en las ecuaciones anteriores, así cuando llegamos a la ecuación i ,

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}}$$

Si repetimos este mismo proceso hasta llegar a la última ecuación del sistema, n , habremos obtenido la solución completa.

El siguiente código calcula la solución de un sistema triangular inferior mediante sustituciones progresivas,

```
function x = progresivas(A,b)
%Esta Función permite obtener la solucion de un sistema triangular inferior
%empleando sustituciones progresivas. La variables de entrada son la matriz
%de coeficientes A y el vector de terminos independientes b. la solucion se
%devuelve como un vector columna en la variable x

%%%%%%%%%%%%%%%
%Obtenemos el tamaño de la matriz de coeficientes y comprobamos que es
%cuadrada,
[f,c]=size(A);
if f~=c
    error('la matriz de coeficientes no es cuadrada')
end
%construimos un vector solucion, inicialmente formado por ceros,
x=zeros(f,1);
%construimos un bucle for para ir calculando progresivamente las soluciones
%del sistema
for i=1:f
    %primero igualamos la solución al termino independiente de la ecuación
    %que toque...
    x(i)=b(i)
    %y luego creamos un bucle para ir restando todas las soluciones
    %anteriores...
    for j=1:i-1
        x(i)=x(i)-A(i,j)*x(j)
    end
    %para terminar dividimos por el elemento de la diagonal de la matriz de
    %coeficientes...
    x(i)=x(i)/A(i,i)
end
```

Sistemas triangulares superiores: método de sustituciones regresivas. En este caso, el sistema general de n ecuaciones con n incógnitas tendrá la forma general,

$$\left. \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \right\} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

El método de resolución es idéntico al de un sistema triangular inferior, simplemente que ahora, empezamos a resolver por la última ecuación,

$$x_n = \frac{b_n}{a_{nn}}$$

Y seguimos sustituyendo hacia arriba,

$$x_{n-1} = \frac{b_{n-1} - a_{(n-1)n}x_n}{a_{(n-1)(n-1)}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^n a_{ij}x_j}{a_{ii}}$$

El código para implementar este método es similar al de las sustituciones progresivas. Se deja como ejercicio el construir una función en Matlab que calcule la solución de un sistema triangular superior por el método de las sustituciones regresivas.

6.3.2. Métodos basados en las factorizaciones

Puesto que sabemos como resolver sistemas triangulares, una manera de resolver sistemas más complejos sería encontrar métodos para reducirlos a sistemas triangulares. De este modo evitamos invertir la matriz de coeficientes y los posibles problemas de estabilidad numérica derivados de dicha operación. En el capítulo anterior 5.5, vimos varios métodos de factorizar una matriz, que permiten una aplicación directa a la resolución de sistemas.

Factorización LU. En el capítulo anterior 5.5.1 vimos como factorizar una matriz en el producto de dos, una triangular inferior L y una triangular superior U . La factorización podía incluir pivoteo de filas, para alcanzar una solución numéricamente estable. En este caso la factorización LU tomaba la forma,

$$P \cdot A = L \cdot U$$

Donde P representa una matriz de permutaciones.

Supongamos que queremos resolver un sistema de n ecuaciones lineales con n incógnitas que representamos genéricamente en forma matricial, como siempre,

$$A \cdot x = b$$

Si calculamos la fatorización LU de su matriz de coeficientes,

$$A \rightarrow P \cdot A = L \cdot U$$

Podemos transformar nuestro sistema de ecuaciones en uno equivalente aplicando la matriz de permutaciones, por la izquierda, a ambos lados del igual. El efecto es equivalente a que cambiáramos el orden en que se presentan las ecuaciones del sistema,

$$A \cdot x = b \rightarrow P \cdot A \cdot x = P \cdot b$$

Si sustituimos ahora $P \cdot A$ por su factorización LU,

$$P \cdot A \cdot x = P \cdot b \rightarrow L \cdot U \cdot x = P \cdot b$$

El nuevo sistema puede resolverse en dos pasos empleando sustituciones regresivas y sustituciones progresivas. Para ello, asociamos el producto $U \cdot x$, a un vector de incógnitas auxiliar al que llamaremos z ,

$$U \cdot x = z$$

Si sustituimos nuestro vector auxiliar z en la expresión matricial de nuestro sistema de ecuaciones,

$$L \cdot \overbrace{U \cdot x}^z = P \cdot b \rightarrow L \cdot z = P \cdot b$$

El sistema resultante es triangular inferior, por lo que podemos resolverlo por sustituciones progresivas, y obtener de este modo los valores de z . Podemos finalmente obtener la solución del sistema a través de la definición de z ; $U \cdot x = z$, se trata de un sistema triangular superior, que podemos resolver mediante sustituciones regresivas.

Veamos un ejemplo. Supongamos que queremos resolver el sistema de ecuaciones lineales,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

En primer lugar deberíamos comprobar que la matriz de coeficiente esta bien condicionada,

```
>> A=[1 3 2;2 -1 1;1 4 3]
A =
```

```
1      3      2
2      -1     1
1      4      3
>> nc=cond(A)
nc =
```

24.3827

No es un valor grande, con lo que podemos considerar que A está bien condicionada. Calculamos la factorización LU de la matriz de coeficientes, para ello podemos emplear el programa lufact.m, incluido en el capítulo anterior 5.5.1, o bien la función lu de Matlab,

```
>> [L U P]=lufact(A)
```

```
L =
```

```
1.0000      0      0
0.5000    1.0000      0
0.5000    0.7778    1.0000
```

```
U =
```

2.0000	-1.0000	1.0000
0	4.5000	2.5000
0	0	-0.4444

```
P =
```

0	1	0
0	0	1
1	0	0

A continuación debemos aplicar la matriz de permutaciones, al vector de términos independientes del sistema, para poder construir el sistema equivalente $L \cdot U = P \cdot b$,

```
>> b=[13;3;18]
```

```
b =
```

13
3
18

```
>> bp=P*b
```

```
bp =
```

3
18
13

Empleamos la matriz L obtenida y el producto $bp = P \cdot b$ que acabamos de calcular, para obtener, por sustituciones progresivas, el vector auxiliar z descrito más arriba. Empleamos para ello la función **progresivas**, cuyo código incluimos en la sección anterior,

```
>> z=progresivas(L,bp)
z =
```

3.0000
16.5000
-1.3333

Finalmente, podemos obtener la solución del sistema sin más que aplicar el método de las sustituciones regresiva a la matriz U y al vector auxiliar z que acabamos de obtener,¹

```
>> x=regresivas(U,z)
x =
```

¹La función **regresivas** no se ha suministrado ni existe en Matlab. Su construcción se ha dejado como ejercicio en la sección anterior.

```
1.0000
2.0000
3.0000
```

Para comprobar que la solución es correcta basta multiplicar la matriz de coeficientes del sistema original por el resultado obtenido para x y comprobar que obtenemos como resultado el vector de términos independientes.

```
>> A*x
```

```
ans =
```

```
13
3
18
```

Factorización de Cholesky. Como vimos en el capítulo anterior 5.5.2, la factorización de Cholesky permite descomponer una matriz A en el producto de una matriz triangular inferior, por su traspuesta.

$$A = L \cdot L^T$$

Para ello, es preciso que A sea simétrica y definida positiva (ver 5.4).

Por tanto, en el caso particular de un sistema cuya matriz de coeficientes fuera simétrica y definida positiva, podríamos descomponerla empleando la factorización de Cholesky y resolver el sistema de modo análogo a como hicimos con la factorización LU, sustituimos A por el producto $L \cdot L^T$,

$$A \cdot x = b \rightarrow L \cdot L^T \cdot x = b$$

Definimos el vector auxiliar z ,

$$L^T \cdot x = z$$

Resolvemos por sustituciones progresivas el sistema,

$$L \cdot z = b$$

y por último obtenemos x resolviendo por sustituciones regresivas el sistema,

$$L^T \cdot x = z$$

La siguiente secuencia de comandos de Matlab muestra la resolución del sistema,

$$\begin{pmatrix} 2 & 5 & 1 \\ 5 & 14 & 2 \\ 1 & 2 & 6 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 15 \\ 39 \\ 23 \end{pmatrix}$$

```
>> A=[2 5 1;5 14 2;1 2 6]
```

```
A =
```

```
2      5      1
5     14      2
1      2      6
```

```

>> b=[15;39;23]
b =
    15
    39
    23

>> L=cholesky(A)
L =
    1.4142      0      0
    3.5355    1.2247      0
    0.7071   -0.4082    2.3094

>> z=progresivas(L,b)
z =
    10.6066
    1.2247
    6.9282

>> x=regresivas(L',z)
x =
    1.0000
    2.0000
    3.0000

>> A*x
ans =
    15
    39
    23

```

En este ejemplo se ha empleado la función `cholesky`, cuyo código se incluyó en el capítulo anterior 5.5.2, para factorizar la matriz de coeficientes del sistema. La factorización se podría haber llevado a cabo igualmente, empleando la función de Matlab `chol`²

Factorización QR Como vimos en el capítulo anterior 5.5.4, la factorización QR, descompone una matriz en el producto de una matriz ortogonal (ver 5.4) Q por una matriz triangular superior R . Si obtenemos la factorización QR de la matriz de coeficientes de un sistema,

$$A \cdot x = b \rightarrow Q \cdot R \cdot x = b$$

Podemos resolver ahora el sistema en dos pasos. En primer lugar, como Q es ortogonal, $Q^{-1} = Q^T$, podemos multiplicar por Q^T a ambos lados de la igualdad,

$$Q \cdot R \cdot x = b \rightarrow Q^T \cdot Q \cdot R \cdot x = Q^T \cdot b \rightarrow R \cdot x = Q^T \cdot b$$

Pero el sistema resultante, es un sistema triangular superior, por lo que podemos resolverlo por sustituciones regresivas. Tomando el mismo ejemplo que resolvimos antes por factorización LU,

²Matlab, por defecto, devuelve una matriz triangular superior: `U = chol(A)`. La factorización es la misma que la descrita en este apartado. Simplemente $L^T = U$ y $L = U^T$. Por tanto, si se emplea directamente el comando `chol` de matlab: $A \cdot x = b \rightarrow U^T \cdot U \cdot x = b$

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

podemos ahora resolverlo mediante factorización QR. Para ello aplicamos la función `QRF2` incluida en el capítulo anterior [5.5.4](#), o directamente la función de Matlab `qr`, a la matriz de coeficientes del sistema,

```
>> A=[1 3 2;2 -1 1;1 4 3]
A =
    1     3     2
    2    -1     1
    1     4     3

>> [Q,R]=QRF2(A)
Q =
    0.4082    0.4637   -0.7863
    0.8165   -0.5707    0.0874
    0.4082    0.6777    0.6116

R =
    2.4495    2.0412    2.8577
        0    4.6726    2.3898
        0         0    0.3495
```

A continuación multiplicamos Q^T por el vector de términos independientes,

```
>> b=[13;3;18]
b =
    13
    3
    18

>> z=Q'*b
z =
    15.1052
    16.5147
    1.0484
```

Por último resolvemos el sistema $R \cdot x = Q^T \cdot b$ mediante sustituciones regresivas y comprobamos el resultado,

```
>> x=regresivas(R,z)

x =
    1.0000
    2.0000
    3.0000

>> A*x
```

```
ans =
13.0000
3.0000
18.0000
```

Factorización SVD. La factorización svd 5.5.5, descompone una matriz cualquiera en el producto de tres matrices,

$$A = U \cdot S \cdot V^T$$

Donde U y V son matrices ortogonales y S es una matriz diagonal. Si calculamos la factorización svd de la matriz de coeficiente de un sistema,

$$A \cdot x = b \rightarrow U \cdot S \cdot V^T \cdot x = b$$

Como en el caso de la factorización QR, podemos aprovechar la ortogonalidad de las matrices U y V para simplificar el sistema,

$$U \cdot S \cdot V^T \cdot x = b \rightarrow U^T \cdot U \cdot S \cdot V^T \cdot x = U^T \cdot b \rightarrow S \cdot V^T \cdot x = U^T \cdot b$$

Como en casos anteriores, podemos crear un vector auxiliar z ,

$$V^T \cdot x = z$$

de modo que resolvemos primero el sistema,

$$S \cdot z = U^T \cdot b$$

Como la matriz S es diagonal, se trata de un sistema diagonal que, como hemos visto, es trivial de resolver.

Una vez conocido Z podemos obtener la solución del sistema original, haciendo ahora uso de la ortogonalidad de la matriz V ,

$$V^T \cdot x = z \rightarrow V \cdot V^T \cdot x = V \cdot z \rightarrow x = V \cdot z$$

Volvamos a nuestro ejemplo,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

En primer lugar factorizamos la matriz de coeficientes empleando el comando `svd` de Matlab,

```
>> A=[1 3 2;2 -1 1;1 4 3]
A =
```

$$\begin{matrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{matrix}$$

```
>> [U,S,V]=svd(A)
U =
```

```
-0.5908 -0.0053 -0.8068
-0.0411 -0.9985 0.0366
-0.8058 0.0548 0.5897
```

S =

```
6.3232 0 0
0 2.4393 0
0 0 0.2593
```

V =

```
-0.2339 -0.7983 -0.5549
-0.7835 0.4927 -0.3786
-0.5757 -0.3463 0.7408
```

Calculamos a continuación el valor del vector auxiliar z ,

```
>> ba=U'*b
ba =
-22.3076
-2.0777
0.2360

>> z=S^-1*ba
z =
-3.5279
-0.8518
0.9101
```

por último, calculamos x y comprobamos la solución obtenida,

```
>> x=V*z
x =
1.0000
2.0000
3.0000

>> A*x
ans =
13.0000
3.0000
18.0000
```

6.3.3. El método de eliminación de Gauss.

El método de eliminación gaussiana es el mismo descrito en el capítulo anterior 5.5.1 para obtener la matriz triangular superior U en la factorización LU de una matriz. Como ya se explicó

en detalle, el método consiste en *hacer cero* todos los elementos situados por debajo de la diagonal de una matriz. Para ello se sustituyen progresivamente las filas de la matriz, exceptuando la primera, por combinaciones adecuadas de dicha fila con las anteriores.

Toda la discusión incluida en el capítulo anterior sobre la eliminación de Gauss, es válida también para la resolución de sistemas, por lo que no volveremos a repetirla. Nos centraremos solo en su aplicación al problema de resolver un sistema de ecuaciones lineales.

La idea fundamental, es sacar partido de las siguientes propiedades de todo sistema de ecuaciones lineales;

1. Un sistema de ecuaciones lineales no cambia aunque se altere el orden de sus ecuaciones.
2. Un sistema de ecuaciones lineales no cambia aunque se multiplique cualquiera de sus ecuaciones por una constante distinta de cero.
3. Un sistema de ecuaciones no cambia si se sustituye cualquiera de sus ecuaciones por una combinación lineal de ella con otra ecuación.

Si usamos la representación matricial de un sistema de ecuaciones, Cualquiera de los cambios descritos en las propiedades anteriores afecta tanto a la matriz de coeficientes como al vector de términos independientes, por ejemplo dado el sistema,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix}$$

Si cambio de orden la segunda ecuación con la primera obtengo el siguiente sistema equivalente,

$$\begin{pmatrix} 2 & -1 & 1 \\ 1 & 3 & 2 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 13 \\ 18 \end{pmatrix}$$

Es decir, se intercambia la primera fila de la matriz de coeficientes con la segunda, y el primer elemento del vector de términos independientes con el segundo. El vector de incógnitas permanece inalterado.

Si ahora sustituimos la segunda fila, por la diferencia entre ella y la primera multiplicada por 0,5, obtenemos de nuevo un sistema equivalente,

$$\begin{pmatrix} 2 & -1 & 1 \\ 0 & 3,5 & 1,5 \\ 1 & 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 11,5 \\ 18 \end{pmatrix}$$

Acabamos de dar los dos primeros pasos en el proceso de eliminación de Gauss para convertir la matriz de coeficientes del sistema en una matriz triangular superior: hemos 'pivotead' las dos primeras filas y después hemos transformado en cero el primer elemento de la segunda fila, combinándola con la primera. Hemos aplicado también esta misma combinación al segundo elemento del vector de términos independientes, para que el sistema obtenido sea equivalente al original.

Para poder trabajar de una forma cómoda con el método de eliminación de Gauss, se suele construir una matriz, –conocida con el nombre de matriz ampliada–, añadiendo a la matriz de coeficientes, el vector de términos independientes como una columna más,

$$A, b \rightarrow AM = (A|b)$$

En nuestro ejemplo,

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & -1 & 1 \\ 1 & 4 & 3 \end{pmatrix}, \begin{pmatrix} 13 \\ 3 \\ 18 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 3 & 2 & 13 \\ 2 & -1 & 1 & 3 \\ 1 & 4 & 3 & 18 \end{pmatrix}$$

Podemos aplicar directamente a la matriz ampliada AM el programa de eliminación de Gauss, `eligauss`, que incluimos en el capítulo anterior 5.5.1, en la sección dedicada a la factorización LU. Si aplicamos `eligauss` a la matriz ampliada del sistema del ejemplo,

```
>> A=[1 3 2;2 -1 1;1 4 3]
A =
1     3     2
2    -1     1
1     4     3

>> b=[13;3;18]
b =
13
3
18

>> AM=[A b]
AM =
1     3     2     13
2    -1     1     3
1     4     3     18

>> GA=eligauss(AM)
GA =
1.0000    3.0000    2.0000   13.0000
0    -7.0000   -3.0000  -23.0000
0        0    0.5714    1.7143
```

El programa ha obtenido como resultado una nueva matriz en la que los elementos situados por debajo de la diagonal son ahora cero. Podemos reconstruir, a partir del resultado obtenido un sistema equivalente actual Separando la última columna del resultado,

$$\begin{pmatrix} 1 & 3 & 2 & 13 \\ 0 & -7 & -3 & -23 \\ 0 & 0 & 0,5714 & 1,7143 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 3 & 2 \\ 0 & -7 & -3 \\ 0 & 0 & 0,571 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ -23 \\ 1,7143 \end{pmatrix}$$

El sistema resultante de la eliminación de Gauss es triangular superior, con lo que podemos resolverlo directamente mediante sustituciones regresivas,

```
>> G=GA(:,1:3)
G =
1.0000    3.0000    2.0000
```

```

0   -7.0000   -3.0000
0       0    0.5714
>> bp=GA(:,4)
bp =
13.0000
-23.0000
1.7143
>> x=regresivas(G,bp)
x =
1.0000
2.0000
3.0000

```

El programa `eligauss`, presenta el problema de que no incluye el pivoteo de filas. Como se explicó en el capítulo anterior este es necesario para evitar que un cero o un valor relativamente pequeño en la diagonal, impida completar el proceso de eliminación o haga los cálculos inestables. A continuación, incluimos una versión modificada de `eligauss` que incluye el pivoteo de filas.

```

function U=eligauss(A)
%Esta función obtiene una matriz triangular superior, a partir de una
%matriz dada, aplicando el método de eliminación gaussiana.
%realiza pivoteo de filas, así que lo primero que hace es comprobar si el
%el elemento de la diagonal, de la fila que se va a emplear para eliminar
%el mayor que los que tiene por debajo en su columna, si no es así,
% intercambia la fila con la que tenga el elemento mayor en dicha columna.

%Obtenemos el número de filas de la matriz..
nf=size(A,1);
U=A;
%
for j=1:nf-1 %recorro todas las columnas menos la última
%%%%%%%%%%%%%pivoteo de filas%%%%%%%%%%%%%
    %Buscamos el elemento mayor de la columna j de la diagonal para abajo
    maxcol=abs(U(j,j));
    index=j;
    for l=j:nf
        if abs(U(l,j))>maxcol
            maxcol=abs(U(l,j));
            index=l;
        end
    end
    %
    %si el mayor no era el elemento de la diagonal U(j,j), intercambiamos la
    %fila l con la fila j
    if index~=j
        aux=U(j,:);
        U(j,:)=U(index,:);
        U(index,:)=aux;
    end
%%%%%%%%%%%fin del pivoteo de filas%%%%%%%%%%%%%

```

```

for i=j+1:nf %Recorro las filas desde debajo de la diagonal hasta la
    %última en Matlab tengo la suerte de poder manejar cada fila de un
    %solo golpe.
    U(i,:)=U(i,:)-U(j,:)*U(i,j)/U(j,j);
end

```

Si aplicamos esta función a nuestro ejemplo de siempre,

```

>> GA=eligausspp(AM)
GA =

```

```

1     4     3     18
0    -1    -1    -5
0     0     4    12

```

y separando en la matriz ampliada la matriz de coeficientes y el vector de términos independientes podemos resolver el sistema por sustituciones regresivas,

```
>> G=GA(:,1:3)
```

```
G =
```

```

1     4     3
0    -1    -1
0     0     4

```

```
>> bp=GA(:,4)
```

```
bp =
18
-5
12
```

```
>> x=regresivas(G,bp)
```

```
x =
1
2
3
```

6.3.4. Gauss-Jordan y matrices en forma reducida escalonada

El método de eliminación de Gauss, permite obtener a partir de una matriz arbitraria, una matriz triangular superior. Una vez obtenida ésta, podríamos seguir transformando la matriz de modo que hiciéramos cero todos los elementos situados por encima de su diagonal. Para ello bastaría aplicar el mismo método de eliminación de Gauss, la diferencia es que ahora eliminaríamos – haríamos ceros– los elementos situados por encima de la diagonal principal de la matriz, empezando por la última columna y moviéndonos de abajo a arriba y de derecha a izquierda. Este proceso se conoce con el nombre de eliminación de Gauss-Jordan. Por ejemplo supongamos que partimos de la matriz que acabamos de obtener por eliminación de Gauss en ejemplo anterior,

$$GA = \begin{pmatrix} 1 & 4 & 3 & 18 \\ 0 & -1 & -1 & -5 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

Empezaríamos por hacer cero el elemento ga_{23} que es el que está situado encima de último elemento de la diagonal principal, para ello restaríamos a la segunda columna la tercera multiplicada por -1 y dividida por 4 ,

$$\begin{pmatrix} 1 & 4 & 3 & 18 \\ 0 & -1 & 0 & -1 + 4/4 \\ 0 & 0 & 4 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 3 & 18 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

A continuación, eliminaríamos el elemento situado en la misma columna, una fila por encima. Para ello restamos a la primera la tercera multiplicada por 3 y dividida por 4 ,

$$\begin{pmatrix} 1 & 4 & 0 & 9 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 0 & 9 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

Como hemos llegado a la primera columna, pasamos a eliminar los elementos de la siguiente columna de la izquierda. En este ejemplo solo tenemos un elemento por encima de la diagonal. Para eliminarlo restamos de la primera fila la segunda multiplicada por 4 y dividida por -1 ,

$$\begin{pmatrix} 1 & 4 - 4 \cdot (-1)/(-1) & 0 & 9 - (-2) \cdot 4/(-1) \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix}$$

Si ahora separamos en la matriz ampliada resultante, la matriz de coeficientes y el vector de términos independientes, obtenemos un sistema diagonal, cuya solución es trivial,

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & -2 \\ 0 & 0 & 4 & 12 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 12 \end{pmatrix} \Rightarrow x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

El siguiente programa `gauss-jordan`, añade las líneas de código necesarias a `eligausspp` para realizar la eliminación de Gauss-Jordan completa de una matriz.

```
function U=gauss_jordan(A)
%Esta función realiza la eliminación de GAUSS-JORDAN, que permite obtener,
%a partir de una matriz dada, una matriz cuyos elementos por encima y
%debajo de la diagonal son todos cero.

%Obtenemos el número de filas de la matriz..
nf=size(A,1);
U=A;
%
%primera parte: reducción a una matriz triangular pro eliminación
%progresiva
for j=1:nf-1 %recorro todas la columnas menos la última
%%%%%%%%%%%%%pivoteo de filas%%%%%%%%%%%%%
    %Buscamos el elemento mayor de la columna j de la diagonal para abajo
    maxcol=abs(U(j,j));
    index=j;
    for l=j:nf
        if abs(U(l,j))>maxcol
            maxcol=abs(U(l,j));
            index=l;
        end
    end
    %Intercambio de filas
    aux=U(index,:);
    U(index,:)=U(j,:);
    U(j,:)=aux;
    %
    %Resta de la fila j a las demás
    for l=j+1:nf
        U(l,:)=U(l,:)-U(j,:)*U(l,j)/U(j,j);
    end
    %
    %División entre el elemento de la diagonal
    U(j,:)=U(j,:)/U(j,j);
end
```

```

        end
    end
    %si el mayor no era el elemento de la diagonal U(j,j), intecambiamos la
    %fila l con la fila j
    if index~=j
        aux=U(j,:);
        U(j,:)=U(index,:);
        U(index,:)=aux;
    end
%%%%%fin del pivoteo de filas%%%%%
    for i=j+1:nf %Re corro las filas desde debajo de la diagonal hasta la
        %última en Matlab tengo la suerte de poder manejar cada fila de un
        %solo golpe.
        U(i,:)=U(i,:)-U(j,:)*U(i,j)/U(j,j);
    end
end

%segunda parte, obtención de una matriz diagonal mediante eliminación
%regresiva, Recorremos ahora las columnas en orden inverso empezando por el
%final... (Eliminación de Gauss Jordan)

for j=nf:-1:2
    for i=j-1:-1:1 %Re corro las filas desde encima de la diagonal hasta la
        %primera,
        U(i,:)=U(i,:)-U(j,:)*U(i,j)/U(j,j);
    end
end

```

Si tras aplicar la eliminación de Gauss-Jordan a la matriz ampliada de un sistema, dividimos cada fila por el elemento que ocupa la diagonal principal, obtendríamos en la última columna las soluciones del sistema. En nuestro ejemplo,

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{pmatrix}$$

La matriz resultante se dice que esta en *forma escalonada reducida por filas*. Se deja como un ejercicio, añadir el código necesario al programa anterior para que dé como resultado la *forma escalonada reducida por filas* de la matriz ampliada de un sistema. Como siempre, Matlab tiene su propia función para obtener formas escalonadas reducidas por filas. Se trata de la función **rref** (el nombre es la abreviatura en inglés de Row Reduced Echelon Form).

```

>> A
A =

```

1	3	4	5	39
2	-1	2	3	18
4	-3	2	1	8
2	3	4	-2	12

```
>> EF=rref(A)
EF =
1 0 0 0 1
0 1 0 0 2
0 0 1 0 3
0 0 0 1 4
```

6.4. Métodos iterativos

Los métodos iterativos se basan en una aproximación distinta al problema de la resolución de sistemas de ecuaciones lineales. La idea en todos ellos es buscar un método que, a partir de un valor inicial para la solución del sistema, vaya refinándolo progresivamente, acercándose cada vez más a la solución real. La figura 6.4, muestra un diagrama de flujo general para todos los métodos iterativos de resolución de sistemas.

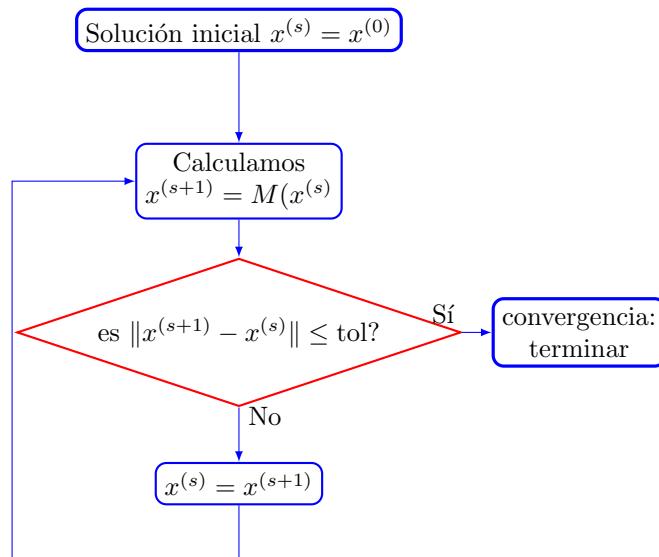


Figura 6.4: Diagrama de flujo general de los métodos iterativos para resolver sistemas de ecuaciones. La función $M(x)$ es la que especifica en cada caso el método.

Siguiendo el diagrama de flujo, el primer paso, es proponer un vector con soluciones del sistema. Si se conocen valores próximos a las soluciones reales se eligen dichos valores como solución inicial. Si, como es habitual, no se tiene idea alguna de cuales son las soluciones, lo más habitual es empezar con el vector (0) ,

$$x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

A partir de la primera solución, se calcula una segunda, siguiendo la especificaciones concretas del método que se esté usando. En el diagrama de flujo se ha representado de modo genérico el

método mediante la función $M(\cdot)$. Dedicaremos buena parte de esta sección a estudiar algunos de los métodos más usuales.

Una vez que se tienen dos soluciones se comparan. Para ello, se ha utilizado el módulo del vector diferencia de las dos soluciones. Este módulo nos da una medida de cuanto se parecen entre sí las dos soluciones. Si la diferencia es suficientemente pequeña, –menor que un cierto valor de tolerancia tol – damos la solución por buena y el algoritmo termina. En caso contrario, copiamos la última solución obtenida en la penúltima y repetimos todo el proceso. El bucle se repite hasta que se cumpla la condición de que el módulo de la diferencia de dos soluciones sucesivas sea menor que la tolerancia establecida.

Una pregunta que surge de modo inmediato del esquema general que acabamos de introducir es si el proceso descrito converge; y, caso de hacerlo, si converge a la solución correcta del sistema.

La respuesta es que los sistemas iterativos no siempre convergen, es decir, no está garantizado que tras un cierto número de iteraciones la diferencia entre dos soluciones sucesivas sea menor que un valor de tolerancia arbitrario. La convergencia, como veremos más adelante, dependerá tanto del sistema que se quiere resolver como del método iterativo empleado. Por otro lado, lo que sí se cumple siempre es que, si el método converge, las sucesivas soluciones obtenidas se van aproximando a la solución real del sistema.

6.4.1. El método de Jacobi.

Obtención del algoritmo. Empezaremos por introducir el método iterativo de Jacobi, por ser el más intuitivo de todos. Para introducirlo, emplearemos un ejemplo sencillo. Supongamos que queremos resolver el siguiente sistema de ecuaciones,

$$\begin{aligned} 3x_1 + 3x_2 &= 6 \\ 3x_1 + 4x_2 &= 7 \end{aligned}$$

Supongamos que supiéramos de antemano el valor de x_2 , para obtener x_1 , bastaría entonces despejar x_1 , por ejemplo de la primera ecuación, y sustituir el valor conocido de x_2 ,

$$x_1 = \frac{6 - 3x_2}{3}$$

De modo análogo, si conociéramos previamente x_1 , podríamos despejar x_2 , ahora por ejemplo de la segunda ecuación, y sustituir el valor conocido de x_1 ,

$$x_2 = \frac{7 - 3x_1}{4}$$

El método de Jacobi lo que hace es *suponer* conocido el valor de x_2 , $x_2^{(0)} = 0$, y con él obtener un valor de $x_1^{(1)}$,

$$x_1^{(1)} = \frac{6 - 3x_2^{(0)}}{3} = \frac{6 - 3 \cdot 0}{3} = 2$$

a continuación *supone* conocido el valor de x_1 , $x_1^{(0)} = 0$ y con él obtiene un nuevo valor para x_2 ,

$$x_2^{(1)} = \frac{7 - 3x_1^{(0)}}{4} = \frac{7 - 3 \cdot 0}{4} = 1,75$$

En el siguiente paso, tomamos los valores obtenidos, $x_1^{(1)}$ y $x_2^{(1)}$ como punto de partida para calcular unos nuevos valores,

$$\begin{aligned}x_1^{(2)} &= \frac{5 - 3x_2^{(1)}}{3} = \frac{6 - 3 \cdot 1,75}{3} = 0,25 \\x_2^{(2)} &= \frac{7 - 3x_1^{(1)}}{4} = \frac{7 - 3 \cdot 1,67}{4} = 0,25\end{aligned}$$

y en general,

$$\begin{aligned}x_1^{(s+1)} &= \frac{6 - 3x_2^{(s)}}{3} \\x_2^{(s+1)} &= \frac{7 - 3x_1^{(s)}}{4}\end{aligned}$$

Si repetimos el mismo cálculo diez veces obtenemos,

$$\begin{aligned}x_1^{(10)} &= 0,7627 \\x_2^{(10)} &= 0,7627\end{aligned}$$

Si lo repetimos veinte veces,

$$\begin{aligned}x_1^{(20)} &= 0,9437 \\x_2^{(20)} &= 0,9437\end{aligned}$$

La solución exacta del sistema es $x_1 = 1$, $x_2 = 1$. Según aumentamos el número de iteraciones, vemos cómo las soluciones obtenidas se aproximan cada vez más a la solución real.

A partir del ejemplo, podemos generalizar el método de Jacobi para un sistema cualquiera de n ecuaciones con n incógnitas,

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\\dots \\a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n\end{aligned}$$

La solución para la incógnita x_i en la iteración, $s + 1$ a partir de la solución obtenida en la iteración s toma la forma,

$$x_i^{(s+1)} = \frac{b_i - \sum_{j \neq i} a_{ij}x_j^{(s)}}{a_{ii}}$$

A continuación, incluimos el código correspondiente al cálculo de una iteración del algoritmo de Jacobi. Este código es el que corresponde, para el método de jacobi, a la función $M(\cdot)$ del diagrama de flujo de la figura 6.4.

```

xs=xs1;
%volvemos a inicializar el vector de soluciones al valor de los
%terminos independientes
xs1=b;
for i=1:n %bucle para recorrer todas las ecuaciones
    for j=1:i-1 %restamos la contribucion de todas las incognitas
        xs1(i)=xs1(i)-A(i,j)*xs(j);           %por encima de x(i)
    end
    for j=i+1:n %restamos la contribución de todas las incognitas
        %por debajo de x(i)
        xs1(i)=xs1(i)-A(i,j)*xs(j);
    end
    %dividimos por el elemento de la diagonal,
    xs1(i)=xs1(i)/A(i,i);
end

```

Expresión matricial para el método de Jacobi. El método de Jacobi que acabamos de exponer puede expresarse también en forma matricial. En Matlab, el empleo en forma matricial, tiene la ventaja de ahorrar bucles en el cálculo de la nueva solución a partir de la anterior. Si expresamos un sistema general de orden n en forma matricial,

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_A \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

Podríamos separar en tres sumandos la expresión de la izquierda del sistema: una matriz diagonal D , una matriz estrictamente diagonal superior U , y una matriz estrictamente triangular inferior L ,

$$\left[\underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_D + \underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b$$

Si pasamos los términos correspondientes a las dos matrices triangulares al lado derecho de la igualdad,

$$\begin{aligned}
 & \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_D \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x = \\
 &= \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b - \left[\underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_x
 \end{aligned}$$

Si examinamos las filas de las matrices resultantes a uno y otro lado del igual, es fácil ver que cada una de ellas coincide con la expresión correspondiente a una iteración del método de Jacobi, sin más que cambiar los valores de las incógnitas a la izquierda del igual por $x^{(s+1)}$ y la derecha por $x^{(s)}$,

$$\begin{aligned}
 & \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_D \cdot \underbrace{\begin{pmatrix} x_1^{(s+1)} \\ x_2^{(s+1)} \\ \vdots \\ x_n^{(s+1)} \end{pmatrix}}_{x^{s+1}} = \\
 &= \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_b - \left[\underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_U + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_L \right] \cdot \underbrace{\begin{pmatrix} x_1^s \\ x_2^s \\ \vdots \\ x_n^s \end{pmatrix}}_{x^s}
 \end{aligned}$$

y multiplicar a ambos lados por la inversa de la matriz D^3 ,

³ D es una matriz diagonal su inversa se calcula trivialmente sin más cambiar cada elemento de la diagonal por su inverso.

$$\underbrace{\begin{pmatrix} x_1^{(s+1)} \\ x_2^{(s+1)} \\ \vdots \\ x_n^{(s+1)} \end{pmatrix}}_{x^{(s+1)}} = \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_{D^{-1}}^{-1} \cdot \left[\underbrace{\begin{pmatrix} b \\ b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_{b} - \left[\underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix}}_{U} + \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_{L} \right] \cdot \begin{pmatrix} x_1^s \\ x_2^s \\ \vdots \\ x_n^s \end{pmatrix} \right]$$

El resultado final es una operación matricial,

$$x^{(s+1)} = D^{-1} \cdot b - D^{-1} \cdot (L + U) \cdot x^s$$

que equivale al código para una iteración y, por tanto a la función $M(\cdot)$ del método de Jacobi.

Si analizamos la ecuación anterior, observamos que el término, $D^{-1} \cdot b$ es fijo. Es decir, permanece igual en todas las iteraciones que necesitemos realizar para que la solución converja. El segundo término, tiene también una parte fija, $-D^{-1}(L + U)$. Se trata de una matriz de orden n , igual al del sistema que tratamos de resolver. Esta matriz, recibe el nombre de matriz del método, se suele representar por la letra H y como veremos después esta directamente relaciona con la convergencia del método.

Si queremos implementar en Matlab el método de Jacobi en forma matricial, lo más eficiente es que calculemos en primer lugar las matrices $D, L, U, f = D^{-1} \cdot b$ y H . Una vez calculadas, empleamos el resultado para aproximar la solución iterativamente. Veamos con un ejemplo como construir las matrices en Matlab. Supongamos que tenemos el siguiente sistema,

$$\begin{pmatrix} 4 & 2 & -1 \\ 3 & -5 & 1 \\ 1 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -4 \\ 17 \end{pmatrix}$$

En primer lugar, calculamos la matriz D , a partir de la matriz de coeficientes del sistema, empleando el comando de Matlab `diag`. Aplicando `diag` a una matriz extraemos en un vector los elementos de su diagonal,

```
A =
4      2      -1
3     -5       1
1     -1       6
>> d=diag(A)
d =
4
-5
6
```

Aplicando `diag` a un vector construimos una matriz con los elementos del vector colocados sobre la diagonal de la matriz. El resto de los elementos son cero.

```
>> D=diag(d)
D =
```

4	0	0
0	-5	0
0	0	6

Si aplicamos dos veces `diag` sobre la matriz de coeficientes de un sistema, obtenemos directamente la matriz D ,

```
>> D=diag(diag(A))
D =
```

4	0	0
0	-5	0
0	0	6

Para calcular las matrices L y U , podemos emplear los comandos de Matlab, `triu` y `tril` que extraen una matriz triangular superior y una matriz triangular inferior, respectivamente, a partir de una matriz dada.

```
>> TU=triu(A)
TU =
```

4	2	-1
0	-5	1
0	0	6

```
>> TL=tril(A)
TL =
```

4	0	0
3	-5	0
1	-1	6

Las matrices L y U son estrictamente triangulares superior e inferior. Podemos obtenerlas restando a las matrices que acabamos de obtener la matriz D ,

```
>> U=triu(A)-D
U =
```

0	2	-1
0	0	1
0	0	0

```
>> L=tril(A)-D
L =
```

0	0	0
---	---	---

```
3      0      0
1     -1      0
```

También podemos obtenerlas directamente, empleando solo la matriz de coeficientes,

```
>> U=A-tril(A)
```

```
U =
```

```
0      2      -1
0      0       1
0      0       0
```

```
>> L=A-triu(A)
```

```
L =
```

```
0      0      0
3      0      0
1     -1      0
```

Construimos a continuación el vector f ,

```
>> f=D^-1*b
```

```
f =
```

```
1.2500
0.8000
2.8333
```

Construimos la matriz del sistema,

```
>> H=-D^-1*(L+U)
```

```
H =
```

```
0     -0.5000    0.2500
0.6000        0    0.2000
-0.1667    0.1667        0
```

A partir de las matrices construidas, el código para calcular una iteración por el método de Jacobi sería simplemente,

```
xs1=f+H*xs
```

En el siguiente fragmento de código se reúnen todas las operaciones descritas,

```
...
...
%obtenemos el tamaño del sistema,
```

```
n=size(A,1);
%Creamos un vector de soluciones inicial,
```

```

xs=zeros(n,1);
%Creamos las matrices del método
D=diag(diag(A));
U=A-tril(A);
L=A-triu(A);
%Alternativamente para jacobi podemos crear una sola matriz equivalente a
%L+U, LpU=A-D
f=D^-1*b;
H=-D^-1*(L+U);

%calculamos la primera iteración,
xs1=f+H*xs;
%calculamos la diferencia entre las dos soluciones,
tolf=norm(xs1-xs);
%ponemos a 1 el contador de iteraciones.
it=1;

%a partir de aquí vendría el código necesario para calcular las
%sucesivas iteraciones hasta que la solución converja
...
...

```

6.4.2. El método de Gauss-Seidel.

Obtención del algoritmo. Este método aplica una mejora, sencilla y lógica al método de Jacobi que acabamos de estudiar. Si volvemos a escribir la expresión general para calcular una iteración de una de las componentes de la solución de un sistema por el método de Jacobi,

$$x_i^{(s+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(s)}}{a_{ii}}$$

Observamos que para obtener el término $x_i^{(s+1)}$ empleamos todos los términos $x_j^{(s)}$, $j \neq i$ de la iteración anterior. Sin embargo, es fácil darse cuenta que, como el algoritmo va calculando las componentes de cada iteración por orden, cuando vamos a calcular $x_i^{(s+1)}$, disponemos ya del resultado de todas las componentes anteriores de la solución para la iteración $s + 1$. Es decir, sabemos ya cuál es el resultado de $x_l^{(s+1)}$, $l < i$. Si el algoritmo converge, esta soluciones serán mejores que las obtenidas en la iteración anterior. Si las usamos para calcular $x_i^{(s+1)}$ el resultado que obtendremos, será más próximo a la solución exacta y por tanto el algoritmo converge más deprisa.

La idea, por tanto sería:

Para calcular $x_1^{(s+1)}$ procedemos igual que en el método de Jacobi.

$$x_1^{(s+1)} = \frac{b_1 - \sum_{j=2}^n a_{1j} x_j^{(s)}}{a_{11}}$$

Para calcular $x_2^{(s+1)}$ usamos la solución que acabamos de obtener $x_1^{(s+1)}$ y todas las restantes soluciones de las iteraciones anteriores,

$$x_2^{(s+1)} = \frac{b_2 - a_{21}x_1^{s+1} - \sum_{j=3}^n a_{2j}x_j^{(s)}}{a_{22}}$$

Para calcular $x_3^{(s+1)}$ usamos las dos soluciones que acabamos de obtener $x_1^{(s+1)}$, $x_2^{(s+1)}$ y todas las restantes soluciones de las iteraciones anteriores,

$$x_3^{(s+1)} = \frac{b_i - a_{31}x_1^{s+1} - a_{32}x_2^{s+1} - \sum_{j=4}^n a_{3j}x_j^{(s)}}{a_{33}}$$

Y para una componente i cualquiera de la solución, obtenemos la expresión general del método de Gauss-Seidel,

$$x_i^{(s+1)} = \frac{b_i - \sum_{j < i} a_{ij}x_j^{(s+1)} - \sum_{j > i} a_{ij}x_j^{(s)}}{a_{ii}}$$

El siguiente código de Matlab implementa una iteración del método de Gauss-Seidel y puede considerarse como la función $M(\cdot)$, incluida en el diagrama de flujo de la figura 6.4, para dicho método.

```
xs=xs1;
%volvemos a inicializar el vector de soluciones al valor de los
%terminos independientes
xs1=b;
for i=1:n %bucle para recorrer todas las ecuaciones
    for j=1:i-1 %restamos la contribucion de todas las incognitas
        xs1(i)=xs1(i)-A(i,j)*xs1(j); %por encima de x(i)
    end
    for j=i+1:n %restamos la contribución de todas las incognitas
        %por debajo de x(i)
        xs1(i)=xs1(i)-A(i,j)*xs(j);
    end
    %dividimos por el elemento de la diagonal,
    xs1(i)=xs1(i)/A(i,i);
end
%calculamos la diferencia entre las dos soluciones,
tolf=norm(xs1-xs);
%incrementamos el contador de iteraciones
it=it+1;
```

Es interesante destacar, que el único cambio en todo el código respecto al método de Jacobi es la sustitución de la variable $xs(j)$ por la variable $xs1(j)$ al final del primer bucle for anidado.

Forma matricial del método de Gauss-Seidel. De modo análogo a cómo hicimos para el método de Jacobi, es posible obtener una solución matricial para el método de Gauss-Seidel.

Supongamos que realizamos la misma descomposición en sumandos de la matriz de coeficientes que empleamos para el método de Jacobi,

$$A \cdot x = b \rightarrow (D + L + U) \cdot x = b$$

En este caso, en cálculo de cada componente de la solución en una iteración intervienen las componentes de la iteración anterior que están por debajo de la que queremos calcular, por tanto solo deberemos pasar a la derecha de la igualdad, la matriz U , Que contiene los coeficientes que multiplican a dichas componentes,

$$(D + L + U) \cdot x = b \rightarrow (D + L) \cdot x = b - U \cdot x$$

Sustituyendo x a cada lado de la igualdad por $x^{(s+1)}$ y $x^{(s)}$ y multiplicando por la inversa de $D + U$ a ambos lados, obtenemos la expresión en forma matricial del cálculo de una iteración por el método de Gauss-Seidel,

$$x^{(s+1)} = (D + L)^{-1} \cdot b - (D + L)^{-1} \cdot U \cdot x^{(s)}$$

Igual que hemos hecho en con el método de Jacobi, podemos identificar las partes fijas que no cambian al iterar: $f = (D+L)^{-1} \cdot b$ y la matriz del método, que en este caso es, $H = -(D+L)^{-1} \cdot U$.

El siguiente fragmento de código muestra la construcción de las matrices necesarias para implementar en Matlab el método de Gauss-Seidel,

```

...
%
%obtenemos el tamaño del sistema,
n=size(A,1);
%Creamos un vector de soluciones inicial,
xs=zeros(n,1);
%Calculamos las matrices necesarias

D=diag(diag(A));
U=A-tril(A);
L=A-triu(A);

f=(D+L)^(-1)*b;
H=-(D+L)^(-1)*U

%Calculamos la primera iteración
xs1=f+H*xs;
%Calculamos la diferencia entre las dos soluciones,
tolf=norm(xs1-xs);
%Ponemos a 1 el contador de iteraciones.
it=1;

%A partir de aquí vendría el código necesario para calcular las
%sucesivas iteraciones hasta que la solución converja.

```

En general, el método de Gauss-Seidel es capaz de obtener soluciones, para un sistema dado y un cierto valor de tolerancia, empleando menos iteraciones que el método de Jacobi. Por ejemplo, si aplicamos ambos métodos a la resolución del sistema,

$$\begin{pmatrix} 4 & 2 & -1 \\ 3 & -5 & 1 \\ 1 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -4 \\ 17 \end{pmatrix}$$

Empleando en ambos casos una tolerancia $tol = 0,00001$, el método de Jacobi necesita 23 iteraciones para lograr una solución, mientras que el de Gauss-Seidel solo necesita 13. La figura 6.5, muestra la evolución de la tolerancia $tol = \|x^{(s+1)} - x^{(s)}\|$, en función del número de iteración. En ambos casos el valor inicial de la solución se tomó como el vector $(0, 0, 0)^T$.

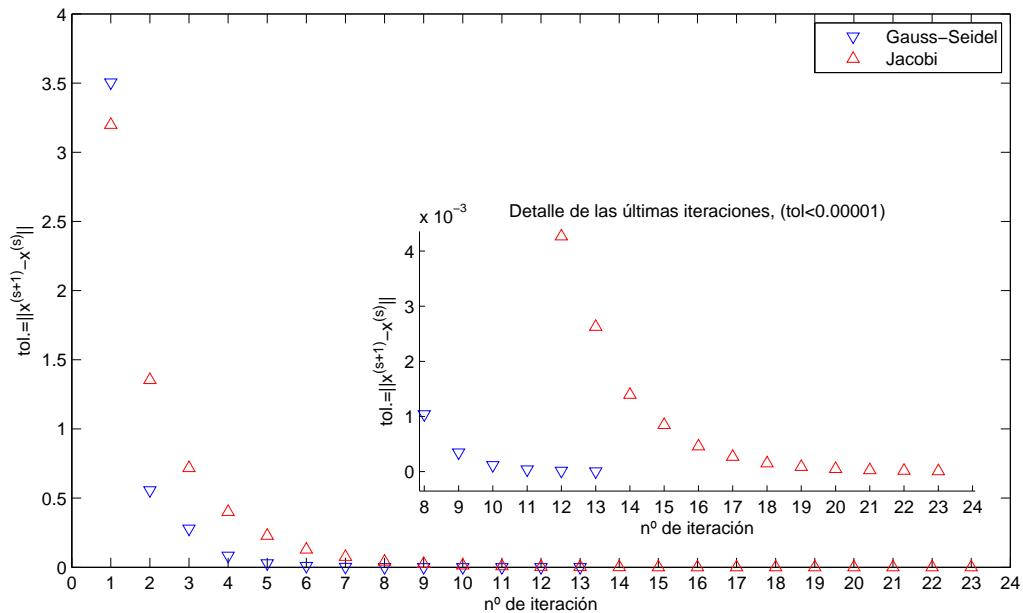


Figura 6.5: evolución de la tolerancia (módulo de la diferencia entre dos soluciones sucesivas) para un mismo sistema resuelto mediante el método de Gauss-Seidel y el método de Jacobi

6.4.3. Amortiguamiento.

El amortiguamiento consiste en modificar un método iterativo, de modo que en cada iteración, se da como solución la media ponderada de los resultados de dos iteraciones sucesivas,

$$x^{(s+1)} = \omega \cdot x^* + (1 - \omega) \cdot x^{(s)}$$

Donde $x^{(*)}$ representa el valor que se obtendría aplicando una iteración del método a $x^{(s)}$, es decir sería el valor de $x^{(s+1)}$ si no se aplica amortiguamiento.

El parámetro ω recibe el nombre de factor de relajamiento. Si $0 < \omega < 1$ se trata de un método de subrelajación. Su uso permite resolver sistemas que no convergen si se usa el mismo método sin relajación. Si $w > 1$ el método se llama de sobrarelajación, permite acelerar la convergencia respecto al mismo método sin relajación. Por último, si hacemos $\omega = 1$, recuperamos el método original sin relajación.

El método de Jacobi amortiguado. Se obtiene aplicando el método de relajación que acabamos de describir, al método de Jacobi. La expresión general de una iteración del método de Jacobi amortiguando sería,

$$x_i^{(s+1)} = \omega \cdot \underbrace{\frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(s)}}{a_{ii}}}_{x^{(*)}} + (1 - \omega) \cdot x_i^s$$

Para implementarlo en Matlab, bastaría añadir al código del método de Jacobi una línea incluyendo el promedio entre las dos soluciones sucesivas,

```

while norm(xs1-xs)>tol
    xs=xs1;
    %volvemos a inicializar el vector de soluciones al valor de los
    %terminos independientes
    xs1=b;
    for i=1:n %bucle para recorrer todas las ecuaciones
        for j=1:i-1 %restamos la contribucion de todas las incognitas
            xs1(i)=xs1(i)-A(i,j)*xs(j); %por encima de x(i)
        end
        for j=i+1:n %restamos la contribución de todas las incognitas
            %por debajo de x(i)
            xs1(i)=xs1(i)-A(i,j)*xs(j);
        end
        %dividimos por el elemento de la diagonal,
        xs1(i)=xs1(i)/A(i,i);
    end
    %promediamos la solución obtenida con la anterior (amortiguamiento)
    xs1=w*xs1+(1-w)*xs
end

```

En forma matricial, la expresión general del método de Jacobi amortiguado sería,

$$x^{(s+1)} = \underbrace{\omega \cdot \left(D^{-1} \cdot b - D^{-1} \cdot (L + U) \cdot x^{(s)} \right)}_{x^{(*)}} + (1 - w)x^{(s)}$$

Si reorganizamos esta expresión,

$$x^{(s+1)} = \omega \cdot D^{-1} \cdot b + [(1 - w) \cdot I - w \cdot D^{-1} \cdot (L + U)] \cdot x^{(s)}$$

Podemos identificar fácilmente el término fijo, $f = \omega \cdot D^{-1} \cdot b$ y la matriz del método $H = ((1 - w) \cdot I - w \cdot D^{-1} \cdot (L + U))$.

Para implementar el código del método de Jacobi amortiguado en Matlab, debemos calcular la matriz identidad del tamaño de sistema y modificar las expresiones de f y H ,

```

...
...
%obtenemos el tamaño del sistema,
n=size(A,1);
%Creamos un vector de soluciones inicial,
xs=zeros(n,1);
%Creamos las matrices del método
D=diag(diag(A));
U=A-tril(A);
L=A-triu(A);
I=eye(n);
%Alternativamente para jacobi podemos crear una sola matriz equivalente a
%L+U, LpU=A-D
f=w*D^-1*b;
H=-w*D^-1*(L+U)+(1-w)*I;

```

```
%calculamos la primera iteración,
xs1=f+H*xs;
%calculamos la diferencia entre las dos soluciones,
tolf=norm(xs1-xs);
%ponemos a 1 el contador de iteraciones.
it=1;

%a partir de aquí vendría el código necesario para calcular las
%sucesivas iteraciones hasta que la solución converja
...
...
```

El método SOR. El método SOR –*Succesive OverRelaxation*– se obtiene aplicando amortiguamiento al método de Gauss-Seidel. Aplicando el mismo razonamiento que el caso de Jacobi amortiguado, la expresión general para una iteración del método SOR es,

$$x_i^{(s+1)} = \omega \cdot \overbrace{\frac{b_i - \sum_{j < i} a_{ij}x_j^{(s+1)} - \sum_{j > i} a_{ij}x_j^{(s)}}{a_{ii}} + (1 - \omega) \cdot x_i^{(s)}}^{x^{(*)}}$$

Al igual que en el caso de Jacobi amortiguado, para implementar en Matlab el método SOR es suficiente añadir una línea que calcule el promedio de dos soluciones sucesivas,

```
xs=xs1;
%volvemos a inicializar el vector de soluciones al valor de los
%terminos independientes
xs1=b;
for i=1:n %bucle para recorrer todas las ecuaciones
    for j=1:i-1 %restamos la contribucion de todas las incognitas
        xs1(i)=xs1(i)-A(i,j)*xs1(j); %por encima de x(i)
    end
    for j=i+1:n %restamos la contribución de todas las incognitas
        %por debajo de x(i)
        xs1(i)=xs1(i)-A(i,j)*xs(j);
    end
    %dividimos por el elemento de la diagonal,
    xs1(i)=xs1(i)/A(i,i);
end
%amortiguamos la solución
xs1=w*xs1+(1-w)*xs;
%calculamos la diferencia entre las dos soluciones,
tolf=norm(xs1-xs);
%incrementamos el contador de iteraciones
it=it+1;
```

En forma matricial la expresión para una iteración del método SOR sería,

$$x^{(s+1)} = \omega \cdot \overbrace{\left((D + L)^{-1} \cdot b - (D + L)^{-1} \cdot U \cdot x^{(s)} \right)}^{x^{(*)}} + (1 - \omega) \cdot x^{(s)}$$

Y tras reordenar,

$$x^{(s+1)} = \omega \cdot ((D + L)^{-1} \cdot b) + [(1 - \omega) \cdot I - \omega \cdot (D + L)^{-1} \cdot U] \cdot x^{(s)}$$

De nuevo, podemos identificar, el término fijo, $f = \omega \cdot ((D + L)^{-1} \cdot b)$ y la matriz del método, $H = ((1 - \omega) \cdot I - \omega \cdot (D + L)^{-1} \cdot U)$.

El siguiente fragmento de código muestra la obtención de f y H para el método SOR,

```
%obtenemos el tamaño del sistema,
```

```
n=size(A,1);
%creamos un vector de soluciones inicial,
xs=zeros(n,1);
%calculamos las matrices necesarias

D=diag(diag(A));
U=A-tril(A);
L=A-triu(A);
I=eye(n);

f=w*(D+L)^(-1)*b;
H=(1-w)*I-w*(D+L)^(-1)*U;
```

```
%calculamos la primera iteración
x0=zeros(n,1);
x1=f+H*x0;
```

```
%calculamos la diferencia entre las dos soluciones,
tolf=norm(x-x0);
%ponemos a 1 el contador de iteraciones.
it=1;
```

6.4.4. Análisis de convergencia

En la introducción a los métodos iterativos dejamos abierta la cuestión de su convergencia. Vamos a analizar en más detalle en qué condiciones podemos asegurar que un método iterativo, aplicado a un sistema de ecuaciones lineales concreto, converge.

En primer lugar, tenemos que definir qué entendemos por convergencia. Cuando un método iterativo converge, lo hace en forma asintótica. Es decir, haría falta un número infinito de iteraciones para alcanzar la solución exacta.

$$x^{(0)} \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \dots \rightarrow x^{(\infty)} = x$$

Lógicamente, es inviable realizar un número infinito de iteraciones. Por esta razón, las soluciones de los métodos iterativos son siempre aproximadas; realizamos un número finito de iteraciones hasta cumplir una determinada condición de convergencia. Como no conocemos la solución exacta, imponemos dicha condición entre dos iteraciones sucesivas,

$$\|x^{(s+1)} - x^{(s)}\| \leq C \Rightarrow \|x^{(s+1)} - x\| = |e^{(s+1)}|$$

Donde $e^{(s+1)}$ representaría el error *real* de convergencia cometido en la iteración $s + 1$.

Tomando como punto de partida la expresión general del cálculo de una iteración en forma matricial,

$$x^{(s+1)} = f + H \cdot x^{(s)}$$

Podemos expresar el error de convergencia como,

$$e^{(s+1)} = x^{(s+1)} - x = f + H \cdot x^{(s)} - x$$

Pero la solución exacta, si pudiera alcanzarse, cumpliría,

$$x = f + H \cdot x$$

Y si sustituimos en la expresión del error de convergencia,

$$e^{(s+1)} = f + H \cdot x^{(s)} - f - H \cdot x$$

Llegamos finalmente a la siguiente expresión, que relaciona los errores de convergencia de dos iteraciones sucesivas,

$$e^{(s+1)} = H \cdot e^{(s)}$$

Para que el error disminuya de iteración en iteración y el método converja, es necesario que la matriz del método H tenga norma-2 menor que la unidad.

Supongamos que un sistema de dimensión n su matriz del método H , tiene un conjunto de n autovectores linealmente independientes, w_1, w_2, \dots, w_n , cada uno asociado a su correspondiente autovalor, $\lambda_1, \lambda_2, \dots, \lambda_n$. El error de convergencia, es también un vector de dimensión n , por tanto podemos expresarlo como una combinación lineal de los n autovectores linealmente independientes de la matriz H . Supongamos que lo hacemos para el error de convergencia $e^{(0)}$ correspondiente al valor inicial de la solución $x^{(0)}$,

$$e^{(0)} = \alpha_1 \cdot w_1 + \alpha_2 \cdot w_2 + \dots + \alpha_n \cdot w_n$$

Si empleamos la ecuación deducida antes para la relación del error entre dos iteraciones sucesivas y recordando que aplicar una matriz a un autovector, es equivalente a multiplicarlo por el autovalor correspondientes: $H \cdot w_i = \lambda_i \cdot w_i$, obtenemos para el error de convergencia en la iteración s ,

$$\begin{aligned} e^{(1)} &= H \cdot e^{(0)} = \alpha_1 \cdot \lambda_1 \cdot w_1 + \alpha_2 \cdot \lambda_2 \cdot w_2 + \dots + \alpha_n \cdot \lambda_n \cdot w_n \\ e^{(2)} &= H \cdot e^{(1)} = H^2 \cdot e^{(0)} = \alpha_1 \cdot \lambda_1^2 \cdot w_1 + \alpha_2 \cdot \lambda_2^2 \cdot w_2 + \dots + \alpha_n \cdot \lambda_n^2 \cdot w_n \\ &\vdots \\ e^{(s)} &= H \cdot e^{(s-1)} = H^s \cdot e^{(0)} = \alpha_1 \cdot \lambda_1^s \cdot w_1 + \alpha_2 \cdot \lambda_2^s \cdot w_2 + \dots + \alpha_n \cdot \lambda_n^s \cdot w_n \end{aligned}$$

Para que el error tienda a cero, $e^{(s)} \rightarrow 0$ al aumentar s , para cualquier combinación inicial de valores α_i , esto es para cualquier aproximación inicial $x^{(0)}$, es necesario que todos los autovalores de la matriz del método cumplan,

$$|\lambda_i| < 1$$

Por tanto, el sistema converge si el radio espectral de la matriz del método es menor que la unidad.⁴

$$\rho(H) < 1 \Rightarrow \lim_{s \rightarrow \infty} e^{(s)} = 0$$

⁴El radio espectral de una matriz es el mayor de sus autovalores en valor absoluto. Ver capítulo 5.5.3.

Velocidad de convergencia. Para un número de iteraciones suficientemente grande, el radio espectral de la matriz del método nos da la velocidad de convergencia. Esto es debido a que el resto de los términos del error, asociados a otros autovalores más pequeños tienden a cero más deprisa. Por tanto podemos hacer la siguiente aproximación, donde estamos suponiendo que el autovalor λ_n es el radio espectral,

$$e^{(s)} \approx c_n \rho(h)^s w_n = c_n \lambda_n^s w_n$$

Podemos ahora calcular cuantas iteraciones nos costará reducir un error inicial por debajo de un determinado valor. Esto dependerá de la solución inicial y de radio espectral de la matriz del método,

$$e^{(s)} \approx \rho(h)^s e^{(0)} \Rightarrow \frac{e^{(s)}}{e^{(0)}} \approx \rho(h)^s$$

así por ejemplo si queremos reducir el error inicial en m dígitos,

$$\frac{e^{(s)}}{e^{(0)}} \approx \rho(h)^s \leq 10^{-m} \Rightarrow s \geq \frac{-m}{\log_{10}(\rho(H))}$$

La matriz del método juega por tanto un papel fundamental tanto en la convergencia del método, como en la velocidad (número de iteraciones) con la que el método converge. Los métodos amortiguados, permiten modificar la matriz de convergencia, gracias al factor de amortiguamiento ω , haciendo que sistemas para los que no es posible encontrar una solución mediante un método iterativo converjan.

Como ejemplo, el sistema,

$$\begin{pmatrix} 1 & 2 & -1 \\ 2 & -5 & 1 \\ 1 & -1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -5 \\ 8 \end{pmatrix}$$

No converge si tratamos de resolverlo por el método de Jacobi. Sin embargo si es posible obtener su solución empleando el método de Jacobi Amortiguado. La figura 6.6 muestra la evolución de la tolerancia para dicho sistema empleando ambos métodos.

Si calculamos el radio espectral de la matriz del método, para el método de Jacobi tendríamos,

```
>> A=[1 2 -1 ; 2 -5 1;1 -1 3]

A =
1     2     -1
2    -5      1
1    -1      3

>> H=diag(diag(A))^-1*(A-diag(diag(A)))

H =
0     2.0000   -1.0000
-0.4000         0   -0.2000
0.3333   -0.3333         0

>> l=eig(H)
l =

```

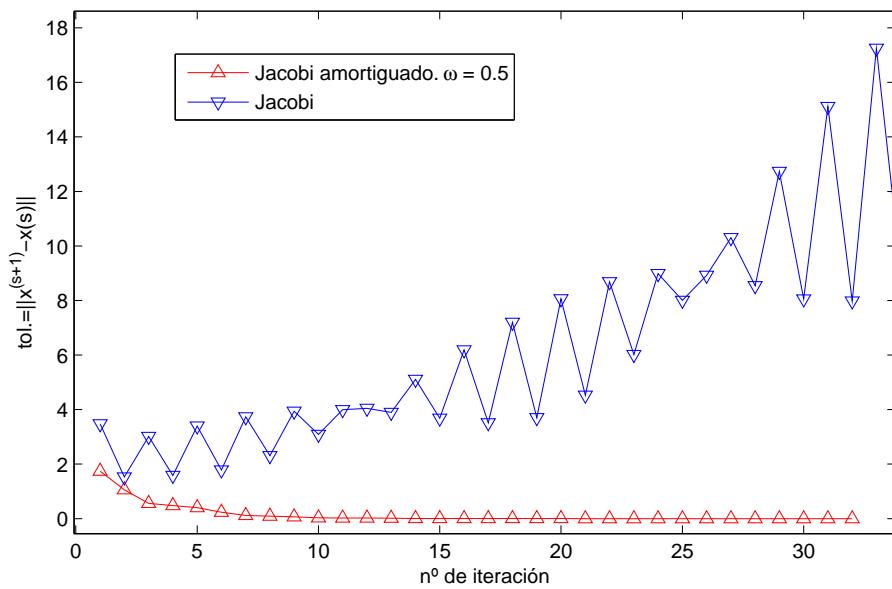


Figura 6.6: Evolución de la tolerancia para un mismo sistema empleando el método de Jacobi (diverge) y el de Jacobi amortiguado (converge).

```
0.1187 + 1.0531i
0.1187 - 1.0531i
-0.2374
```

```
>> radio_espectral=max(abs(1))
radio_espectral =
1.0597
```

El radio espectral es mayor que la unidad y el método no converge.
SI repetimos el cálculo para el método de Jacobi amortiguado, con $\omega = 0,5$

```
>> H=(1-0.5)*eye(3)-0.5*diag(diag(A))^-1*(A-diag(diag(A)))
```

```
H =
0.5000    -1.0000    0.5000
0.2000     0.5000    0.1000
-0.1667    0.1667    0.5000
```

```
>> l=eig(H)
```

```
l =
0.4406 + 0.5265i
```

```
0.4406 - 0.5265i  
0.6187
```

```
>> radio_espectral=max(abs(1))  
  
radio_espectral =  
  
0.6866
```

El radio espectral es ahora menor que la unidad y el método converge.
Por último indicar que cualquiera de los métodos iterativos descrito converge para un sistema que cumpla que su matriz de coeficientes es estrictamente diagonal dominante.

Capítulo 7

Interpolación y ajuste de funciones

En este capítulo vamos a estudiar distintos métodos de aproximación polinómica. En términos generales el problema consiste en sustituir una función $f(x)$ por un polinomio,

$$f(x) \approx p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 + \cdots + a_n \cdot x^n$$

Para obtener la aproximación podemos partir de la ecuación que define $f(x)$, por ejemplo la función error,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

O bien, puede suceder que solo conozcamos algunos valores de la función, por ejemplo a través de una tabla de datos,

Cuadro 7.1: $f(x) = \operatorname{erf}(x)$

x	$f(x)$
0,0	0,0000
0,1	0,1125
0,2	0,2227
0,3	0,3286
0,4	0,4284
0,5	0,5205

La aproximación de una función por un polinomio, tiene ventajas e inconvenientes.

Probablemente la principal ventaja, desde el punto de vista del cómputo, es que un polinomio es fácil de evaluar mediante un ordenador ya que solo involucra operaciones aritméticas sencillas. Además, los polinomios son fáciles de derivar e integrar, dando lugar a otros polinomios.

En cuanto a los inconvenientes hay que citar el crecimiento hacia infinito o menos infinito de cualquier polinomio para valores de la variable independiente alejados del origen. Esto puede dar lugar en algunos casos a errores de redondeo difíciles de manejar, haciendo muy difícil la aproximación para funciones no crecientes.

Vamos a estudiar tres métodos distintos; en primer lugar veremos la aproximación mediante el polinomio de Taylor, útil para aproximar una función en las inmediaciones de un punto. A continuación, veremos la interpolación polinómica y, por último, estudiaremos el ajuste polinómico por mínimos cuadrados.

El uso de uno u otro de estos métodos esta asociado a la información disponible sobre la función que se desea aproximar y al uso que se pretenda hacer de la aproximación realizada.

7.1. El polinomio de Taylor.

Supongamos una función infinitamente derivable en un entorno de un punto x_0 . Su expansión en serie de Taylor se define como,

$$f(x) = f(x_0) + f'(x_0) \cdot (x - x_0) + \frac{1}{2} f''(x_0) \cdot (x - x_0)^2 + \cdots + \frac{1}{n!} f^{(n)}(x_0) \cdot (x - x_0)^n + \frac{1}{(n+1)!} f^{(n+1)}(z) \cdot (x - x_0)^{n+1}$$

Donde z es un punto sin determinar situado entre x y x_0 . Si eliminamos el último término, la función puede aproximarse por un polinomio de grado n

$$f(x) \approx f(x_0) + f'(x_0) \cdot (x - x_0) + \frac{1}{2} f''(x_0) \cdot (x - x_0)^2 + \cdots + \frac{1}{n!} f^{(n)}(x_0) \cdot (x - x_0)^n$$

El error cometido al aproximar una función por un polinomio de Taylor de grado n , viene dado por el término,

$$e(x) = |f(x) - p(x)| = \left| \frac{1}{(n+1)!} f^{(n+1)}(z) \cdot (x - x_0)^{n+1} \right|$$

Es fácil deducir de la ecuación que el error disminuye con el grado del polinomio empleado y aumenta con la distancia entre x y x_0 . Además cuanto más suave es la función (derivadas pequeñas) mejor es la aproximación.

Por ejemplo para la función exponencial, el polinomio de Taylor de orden n desarrollado en torno al punto $x_0 = 0$ es,

$$e^x \approx 1 + x + \frac{1}{2}x^2 + \cdots + \frac{1}{n!}x^n = \sum_{i=0}^n \frac{1}{i!}x^i$$

y el del logaritmo natural, desarrollado en torno al punto $x_0 = 1$,

$$\log(x) \approx (x - 1) - \frac{1}{2}(x - 1)^2 + \cdots + \frac{(-1)^{n+1}}{n}(x - 1)^n = \sum_{i=1}^n \frac{(-1)^{i+1}}{i}(x - 1)^i$$

La existencia de un término general para los desarrollos de Taylor de muchas funciones elementales lo hace particularmente atractivo para aproximar funciones mediante un ordenador. Así por ejemplo, la siguiente función de Matlab, aproxima el valor del logaritmo natural en un punto, empleando un polinomio de Taylor del grado que se deseé,

```
function y=taylorln(x,n)
%Esta función aproxima el valor del logaritmo natural de un numero
%empleando para ello un polinomio de Taylor de grado n desarrollado en
%torno a x=1. Las variables de entrada son: x, valor para el que se desea
%calcular el logaritmo. n Grado del polinomio que se empleará en el
%cálculo. La variable de salida y es el logaritmo de x. (nota si x
%es un vector, calculará el logaritmo para todos los puntos del vector)

%inicializamos la variable de salida y
y=0;

%construimos un bucle para ir añadiendo términos al desarrollo
for i=1:n
    y=y+(-1)^(i+1)*(x-1).^i/i;
end
```

La aproximación funciona razonablemente bien para puntos comprendidos en el intervalo $0 < x < 2$. La figura 7.1 muestra los resultados obtenidos en dicho intervalo para polinomios de Taylor del logaritmo natural de grados 2, 5, 10 20. La linea continua azul representa el valor del logaritmo obtenido con la función de Matlab `log`.

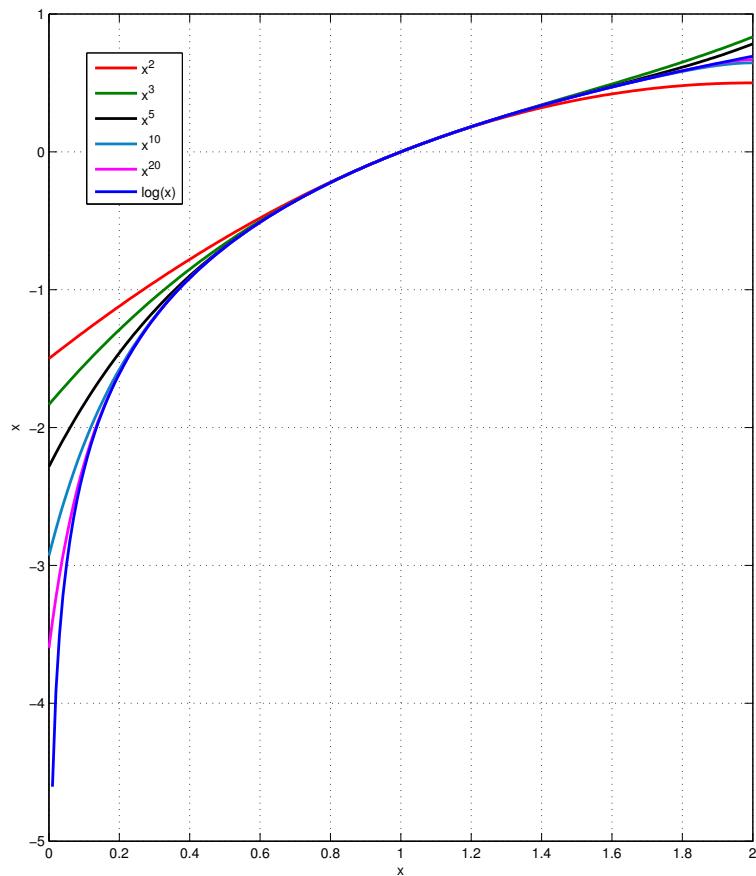


Figura 7.1: Comparación entre resultados obtenidos para polinomios de Taylor del logaritmo natural. (grados 2, 3, 5, 10, 20)

Las funciones $\sin(x)$ y $\cos(x)$, son también simples de aproximar mediante polinomios de Taylor. Si desarrollamos en torno a $x_0 = 0$, la serie del coseno solo tendrá potencias pares mientras que la del seno solo tendrá potencias impares,

$$\cos(x) \approx \sum_{i=0}^n \frac{(-1)^i}{(2i)!} x^{2i}$$

$$\sin(x) \approx \sum_{i=0}^n \frac{(-1)^i}{(2i+1)!} x^{2i+1}$$

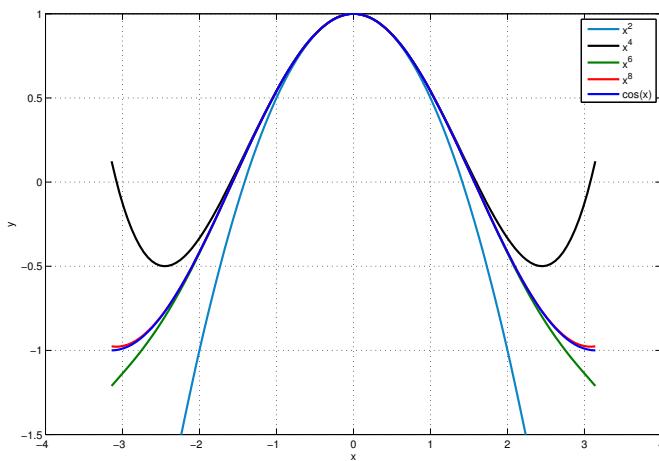
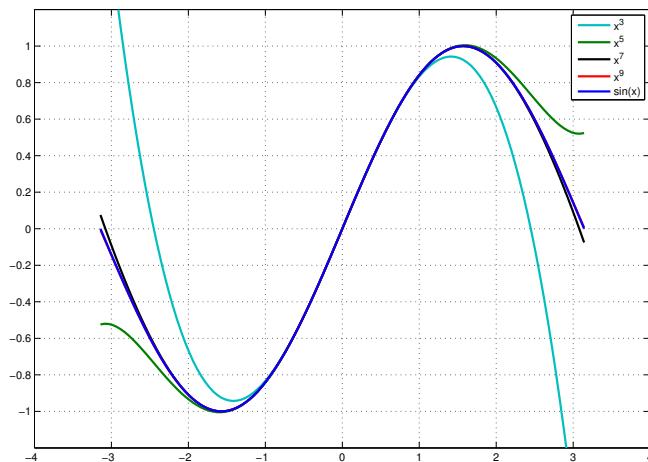
(a) $\cos(x)$, polinomios 2, 4, 6 y 8 grados(b) $\sin(x)$, polinomios 3, 5, 7 y 9 grados

Figura 7.2: Polinomios de Taylor para las funciones coseno y seno

En las figuras 7.2(a) y 7.2(b) Se muestran las aproximaciones mediante polinomios de Taylor de las funciones coseno y seno. Para el coseno se han empleado polinomios hasta grado 8 y para el seno

hasta grado 9. En ambos casos se dan los resultados correspondientes a un periodo $(-\pi, \pi)$. Si se comparan los resultados con las funciones \cos y \sin , suministradas por Matlab, puede observarse que la aproximación es bastante buena para los polinomios de mayor grado empleados en cada caso.

7.2. Interpolación polinómica.

Se entiende por interpolación el proceso por el cual, dado un conjunto de pares de puntos $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ se obtiene una función $f(x)$, tal que, $y_i = f(x_i)$, para cada par de puntos (x_i, y_i) del conjunto. Si, en particular, la función empleada es un polinomio $f(x) \equiv p(x)$, entonces se trata de interpolación polinómica.

Teorema de unicidad. Dado un conjunto $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ de $n+1$ pares de puntos, tales que todos los valores x_i de dicho conjuntos son diferentes entre sí, solo existe un polinomio $p(x)$ de grado n , tal que $y_i = p(x_i)$ para todos los pares de puntos del conjunto.

Si tratamos de interpolar los puntos con un polinomio de grado menor que n , es posible que no encontramos ninguno que pase por todos los puntos. Si, por el contrario empleamos un polinomio de grado mayor que n , nos encontramos con que no es único. Por último si el polinomio empleado es de grado n , entonces será siempre el mismo con independencia del método que empleemos para construirlo.

7.2.1. La matriz de Vandermonde

Supongamos que tenemos un conjunto de pares de puntos \mathcal{A} ,

x	$f(x)$
x_0	y_0
x_1	y_1
x_2	y_2
\vdots	\vdots
x_n	y_n

Para que un polinomio de orden n ,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$

pase por todos los pares de \mathcal{A} debe cumplir,

$$y_i = a_0 + a_1 x_i + a_2 x_i^2 + \dots + a_n x_i^n, \quad \forall (x_i, y_i) \in \mathcal{A}$$

Es decir, obtendríamos un sistema de n ecuaciones lineales, una para cada par de valores, en la que las incógnitas son precisamente los n coeficientes a_i del polinomio.

Por ejemplo para los puntos,

x	$f(x)$
1	2
2	1
3	-2

Obtendríamos,

$$\begin{aligned} a_0 + a_1 \cdot 1 + a_2 \cdot 1^2 &= 2 \\ a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 &= 1 \\ a_0 + a_1 \cdot 3 + a_2 \cdot 3^2 &= -2 \end{aligned}$$

que podríamos expresar en forma matricial como,

$$\begin{pmatrix} 1 & 1 & 1^2 \\ 1 & 2 & 2^2 \\ 1 & 3 & 3^2 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}$$

Y en general, para n pares de datos,

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}$$

La matriz de coeficientes del sistema resultante recibe el nombre de matriz de Vandermonde. Está formada por las n primeras potencias de cada uno de los valores de la variable independiente, colocados por filas. Es evidente que cuanto mayor es el número de datos, mayor tenderá a ser la diferencia de tamaño entre los elementos de cada fila. Por ello, en la mayoría de los casos, resulta ser una matriz mal condicionada para resolver el sistema numéricamente. En la práctica, para obtener el polinomio interpolador, se emplean otros métodos alternativos,

7.2.2. El polinomio interpolador de Lagrange.

A partir de los valores x_0, x_1, \dots, x_n , se construye el siguiente conjunto de $n+1$ polinomios de grado n

$$l_j(x) = \prod_{\substack{k=0 \\ k \neq j}}^n \frac{x - x_k}{x_j - x_k} = \frac{(x - x_0)(x - x_1) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)}{(x_j - x_0)(x_j - x_1) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)}$$

Los polinomios así definidos cumplen una interesante propiedad en relación con los valores x_0, x_1, \dots, x_n , empleados para construirlos,

$$l_j(x_i) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

A partir de estos polinomios podemos construir ahora el siguiente polinomio de interpolación empleando las imágenes y_0, y_1, \dots, y_n correspondientes a los valores x_0, x_1, \dots, x_n ,

$$p(x) = \sum_{j=0}^n l_j(x) \cdot y_j$$

Efectivamente, es fácil comprobar que, tal y como se ha construido, este polinomio pasa por los pares de puntos (x_i, y_i) , puesto que $p(x_i) = y_i$.

El siguiente código de Matlab calcula el valor en un punto x cualquiera del polinomio de interpolación de Lagrange construido a partir un conjunto de puntos $\mathcal{A} \equiv \{(x_i, y_i)\}$.

```

function y1=Lagrange(x,y,x1)
%este programa obtiene el valor interpolado y1 correspondiente al valor x1
%empleando el polinomio interpolador de Lagrange de grado n, obtenido a
%partir de los vectores x e y (de longitud n)

%obtenemos el tamaño del conjunto de datos,
n=length(x);
%inicializamos la variable de salida
y1=0;
%construimos el valor a partir de los polinomios de Lagrange,
for j=1:n
    %inicializamos el polinomio de Lagrange correspondiente al dato i
    lj=1;
    %y lo calculamos...
    for i=1:j-1
        lj=lj*(x1-x(i))/(x(j)-x(i));
    end
    for i=j+1:n
        lj=lj*(x1-x(i))/(x(j)-x(i));
    end

    %sumamos la contribución del polinomio de Lagrange lj
    y1=y1+lj*y(j);
end

```

7.3. Diferencias divididas.

Tanto el método de la matriz de Vandermonde como el de los polinomios de Lagrange, presentan el inconveniente de que si se añade un dato más (x_{n+1}, y_{n+1}) a la colección de datos ya existentes, es preciso recalcular el polinomio de interpolación desde el principio.

El método de las diferencias divididas, permite obtener el polinomio de interpolación en un número menor de operaciones que en el caso del polinomio de Lagrange y además, el cálculo se hace escalonadamente, aprovechando todos los resultados anteriores cuando se añade al polinomio la contribución de un nuevo dato.

El polinomio de orden n de diferencias divididas se construye de la siguiente manera,

$$p_n(x) = a_0 + (x - x_0) \cdot a_1 + (x - x_0) \cdot (x - x_1) \cdot a_2 + \cdots + (x - x_0) \cdot (x - x_1) \cdots (x - x_{n-2}) \cdot (x - x_{n-1}) \cdot a_n$$

Donde, como siempre, $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$, representan los datos para los que se quiere calcular el polinomio interpolador de grado n . Si sustituimos los datos en el polinomio, llegamos a un sistema de ecuaciones, triangular inferior, en el que las incógnitas son los coeficientes del

polinomio.

$$\begin{aligned}
 a_0 &= y_0 \\
 a_0 + (x_1 - x_0)a_1 &= y_1 \\
 a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2 &= y_2 \\
 \vdots & \\
 a_0 + (x_n - x_0)a_1 + \cdots + (x_n - x_0)(x_n - x_1) \cdots (x_n - x_{n-2})(x_n - x_{n-1})a_n &= y_n
 \end{aligned}$$

Este sistema se resuelve explícitamente empleando un esquema de diferencias divididas. La diferencia dividida de primer orden entre dos puntos (x_0, y_0) y (x_1, y_1) se define como,

$$f[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0}$$

Para tres puntos se define la diferencia dividida de segundo orden como, (x_0, y_0) , (x_1, y_1) y (x_2, y_2)

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

y, en general definiremos la diferencia dividida de orden i para $i+1$ puntos como,

$$f[x_0, x_1, \dots, x_i] = \frac{f[x_1, x_2, \dots, x_i] - f[x_0, x_1, \dots, x_{i-1}]}{x_i - x_0}$$

Si despejamos por sustitución progresiva los coeficientes del polinomio de interpolación del sistema triangular inferior obtenido, cada coeficiente puede asociarse a una diferencia dividida,

$$\begin{aligned}
 a_0 &= f[x_0] = y_0 \\
 a_1 &= f[x_0, x_1] \\
 &\vdots \\
 a_i &= f[x_0, x_1, \dots, x_i] \\
 &\vdots \\
 a_n &= f[x_0, x_1, \dots, x_n]
 \end{aligned}$$

Por tanto, podemos obtener directamente los coeficientes del polinomio calculando las diferencias divididas. Veamos un ejemplo empleando el siguiente conjunto de cuatro datos,

x	0	1	3	4
y	1	-1	2	3

Habitualmente, se construye a partir de los datos una tabla, como la 7.2, de diferencias divididas. Las primera columna contiene los valores de la variable x , la siguiente los valores de las diferencias divididas de orden cero (valores de y). A partir de la segunda, las siguientes columnas contienen las diferencias divididas de los elementos de la columna anterior, calculados entre los elementos que ocupan filas consecutivas. La tabla va perdiendo cada vez una fila, hasta llegar a la diferencia dividida de orden n de todos los datos iniciales.

Los coeficientes del polinomio de diferencias divididas se corresponden con los elementos de la primera fila de la tabla. Por lo que en nuestro ejemplo el polinomio resultante sería,

Cuadro 7.2: Tabla de diferencia divididas para cuatro datos

x_i	y_i	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$
$x_0 = 0$	$y_0 = 1$	$f[x_0, x_1] = -2$	$f[x_0, x_1, x_2] = 7/6$	$f[x_0, x_1, x_2, x_3] = -1/3$
$x_1 = 1$	$y_1 = -1$	$f[x_1, x_2] = 3/2$	$f[x_1, x_2, x_3] = -1/6$	
$x_2 = 3$	$y_2 = 2$	$f[x_2, x_3] = 1$		
$x_3 = 4$	$y_3 = 3$			

$$p_3(x) = 1 - 2x + \frac{7}{6}x(x-1) - \frac{1}{3}x(x-1)(x-3)$$

Es importante hacer notar que el polinomio de interpolación obtenido por diferencias divididas siempre aparece representado como suma de productos de binomios $(x - x_0)(x - x_1) \cdots$ y los coeficientes obtenidos corresponden a esta representación y no a la representación habitual de un polinomio como suma de potencias de la variable x .

El siguiente código permite calcular los coeficientes del polinomio de diferencias divididas a partir de un conjunto de n datos.

```

function a=difdiv(x,y)
%este polinomio permite obtener los coeficientes del polinomio de
%diferencias divididas que interpola los datos contenidos en los vectores x
%e y. Da como resultado un vector fila a con los coeficientes

%miramos cuantos datos tenemos
n=length(x);

%inicializamos el vector de coeficientes con las diferencias de orden 0, es
%decir los valores de y,
a=y;

%y ahora montamos un bucle, si tenemos n datos debemos calcular n
%diferencias, como ya tenemos la primera, iniciamos el bucle en 2,
for j=2:n
    %en cada iteración calculamos las diferencias de un orden superior,
    %como solo nos vale la primera diferencia de cada orden empezamos el
    %bucle interior en el valor del exterior j
    for i=j:n
        a(i)=(a(i)-y(i-1))/(x(i)-x(i-j+1));
    end
    %volvemos a copiar en y las diferencias obtenidas para emplearlas en la
    %siguiente iteracion
    y=a;
end

```

Como el polinomio de diferencias divididas toma una forma especial, es preciso tenerlo en cuenta a la hora de calcular su valor en un punto x determinado. Es siguiente código permite evaluar un polinomio de diferencias divididas en un punto dado, conocidos sus coeficientes y los valores x_1, \dots, x_n a partir de los cuales se obtuvo el polinomio,

```

function y=evdif(a,x,x1)
%esta función obtiene el valor de un polinomio de diferencias divididas a
%partir de los coeficientes (a) del polinomio, los puntos (x) sobre los que
%se ha calculado el polinomio y el punto o vector de puntos (x1) para el
%que se quiere calcular el valor que toma el polinomio.

%obtenemos el tamaño del vector de coeficientes del polinomio,
n=length(a);

%Construimos un bucle para calcular el valor del polinomio,
y=a(1);
for k=1:n-1
    %calculamos el valor del producto de los binomios que multiplican al
    %coeficiente i
    binprod=1;
    for j=1:k
        binprod=binprod.*(x1-x(j));
    end
    y=y+a(k+1)*binprod;
end

```

Por último se podrían reunir las dos funciones anteriores en una única función que permitiera obtener directamente el valor que toma el polinomio de diferencias divididas en un punto x , partiendo de los datos interpolados. Una solución sencilla, es crear una función que llame a las dos anteriores,

```

function y=intdifdiv(x,xp,yp)
%Esta función calcula el valor del polinomio de diferencias divididas que
%interpola los puntos (xp,yp) el el punto, o %los puntos contenidos en x.
%Empleando las funciones, difdiv, para calcular los coeficientes del
%polinomio y evdif para evaluarlo

%llamamos a difdiv
a=difdiv(xp,yp);

%y a continuación llamamos a evdif
y=evdif(a,xp,x);

```

7.3.1. El polinomio de Newton-Gregory

Supone una simplificación al cálculo del polinomio de diferencias divididas para el caso particular en que los datos se encuentran equiespaciados y dispuestos en orden creciente con respecto a los valores de la coordenada x .

En este caso, calcular los valores de las diferencias es mucho mas sencillo. Si pensamos en las diferencias de primer orden, los denominadores de todas ellas son iguales, puesto que los datos están equiespaciados,

$$\Delta x \equiv x_i - x_{i-1} = h$$

En cuanto a los numeradores, se calcularían de modo análogo al de las diferencias divididas normales,

$$\Delta y_0 = y_1 - y_0, \Delta y_1 = y_2 - y_1, \dots, \Delta y_i = y_{i+1} - y_i, \dots, \Delta y_{n-1} = y_n - y_{n-1}$$

Las diferencias de orden superior para los numeradores se pueden obtener de modo recursivo, a partir de las de orden uno, puesto que los denominadores de todas ellas h , son iguales.

$$\Delta^2 y_0 = \Delta(\Delta y_0) = (y_2 - y_1) - (y_1 - y_0) = (y_2 - 2y_1 + y_0)$$

En este caso, el denominador de la diferencia sería $x_2 - x_0 = 2h$, y la diferencia tomaría la forma,

$$f[x_0, x_1, x_2] = \frac{\Delta^2 y_0}{2h^2}$$

En general, para la diferencias de orden n tendríamos,

$$\Delta^n y_0 = y_n - \binom{n}{1} \cdot y_{n-1} + \binom{n}{2} \cdot y_{n-2} - \dots + (-1)^n \cdot y_0$$

Donde se ha hecho uso de la expresión binomial,

$$\binom{k}{l} = \frac{k!}{l! \cdot (k-l)!}$$

Para obtener la diferencia dividida de orden n , bastaría ahora dividir por $n! \cdot h^n$.

$$f[x_0, x_1, \dots, x_n] = \frac{\Delta^n y_0}{n! \cdot h^n}$$

A partir de las diferencias, podemos representar el polinomio de diferencias divididas resultante como,

$$p_n(x) = y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_1) \cdot (x - x_0)}{2 \cdot h^2} \Delta^2 y_0 + \dots + \frac{(x - x_{n-1}) \cdots (x - x_1) \cdot (x - x_0)}{n! \cdot h^n} \Delta^n y_0$$

Este polinomio se conoce como el polinomio de Newton-Gregory, y podría considerarse como una aproximación numérica al polinomio de Taylor de orden n de la posible función asociada a los datos empleados.

En este caso, podríamos construir la tabla para obtener los coeficientes de diferencias, calculando en cada columna simplemente las diferencias de los elementos de la columna anterior. Por ejemplo,

Cuadro 7.3: Tabla de diferencias para el polinomio de Newton -Gregory de cuatro datos

x_i	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$
$x_0 = 0$	$y_0 = 1$	-2	5	-7
$x_1 = 1$	$y_1 = -1$	3	-2	
$x_2 = 2$	$y_2 = 2$	1		
$x_3 = 3$	$y_3 = 3$			

Una vez calculadas las diferencias, basta dividir por $n! \cdot h^n$ los elementos de la primera fila de la tabla,

$$a_0 = 1, a_1 = \frac{-2}{1}, a_2 = \frac{5}{2 \cdot 1^2}, a_3 = \frac{-7}{6 \cdot 1^3}$$

El siguiente código muestra un ejemplo de implementación en Matlab del polinomio de Newton-Gregory

```

function [a,y1]=newgre(x,y,x1)
%este polinomio permite obtener los coeficientes del polinomio de
%newton-gregory que interpola los datos contenidos en los vectores x
%y e y. Da como resultado un vector fila a con los coeficientes, si se le da
%ademas un punto o vector de puntos calcula los valores que toma el
%polinomio en dichos puntos.

%miramos cuantos datos tenemos
n=length(x);

%inicializamos el vector de coeficientes con las diferencias de orden 0, es
%decir los valores de y,
a=y;
h=x(2)-x(1);

%y ahora montamos un bucle, si tenemos n datos debemos calcular n
%diferencias, como ya tenemos la primera, iniciamos el bucle en 2,
for j=2:n
    %en cada iteración calculamos las diferencias de un orden superior,
    %como solo nos vale la primera diferencia de cada orden empezamos el
    %bucle interior en el valor del exterior j
    for i=j:n
        %ahora basta dividir en todos los casos por la distancia h
        %multiplicada por el orden de la diferencia
        a(i)=(a(i)-y(i-1))/((j-1)*h);
    end
    %volvemos acopiar en y las diferencias obtenidas para emplearlas en la
    %siguiente iteracion
    y=a;
end

y1=[];

if nargin==3
    %Construimos un bucle para calcular el valor del polinomio,
    y1=a(1);
    for k=1:n-1
        %calculamos el valor del producto de los binomios que multiplican al
        %coeficiente i
        binprod=1;
        for j=1:k
            binprod=binprod.*((x1-x(j)));
        end
        y1=y1+a(k+1)*binprod;
    end
end

```

```
end
```

7.4. Interpolación por intervalos.

Hasta ahora, hemos visto cómo interpolar un conjunto de $n + 1$ datos mediante un polinomio de grado n . En muchos casos, especialmente cuando el número de datos es suficientemente alto, los resultados de dicha interpolación pueden no ser satisfactorios. La razón es que el grado del polinomio de interpolación crece linealmente con el número de puntos a interpolar, así por ejemplo para interpolar 11 datos necesitamos un polinomio de grado 10. Desde un punto de vista numérico, este tipo de polinomios pueden dar grandes errores debido al redondeo. Por otro lado, y dependiendo de la disposición de los datos para los que se realiza la interpolación, puede resultar que el polinomio obtenido tome una forma demasiado complicada para los valores comprendidos entre los datos interpolados..

La figura 7.3 muestra el polinomio de interpolación de grado nueve para un conjunto de 10 datos. Es fácil darse cuenta, simplemente observando los datos, que no hay ninguna razón que justifique las curvas que traza el polinomio entre los puntos 1 y 2 o los puntos 9 y 10, por ejemplo.

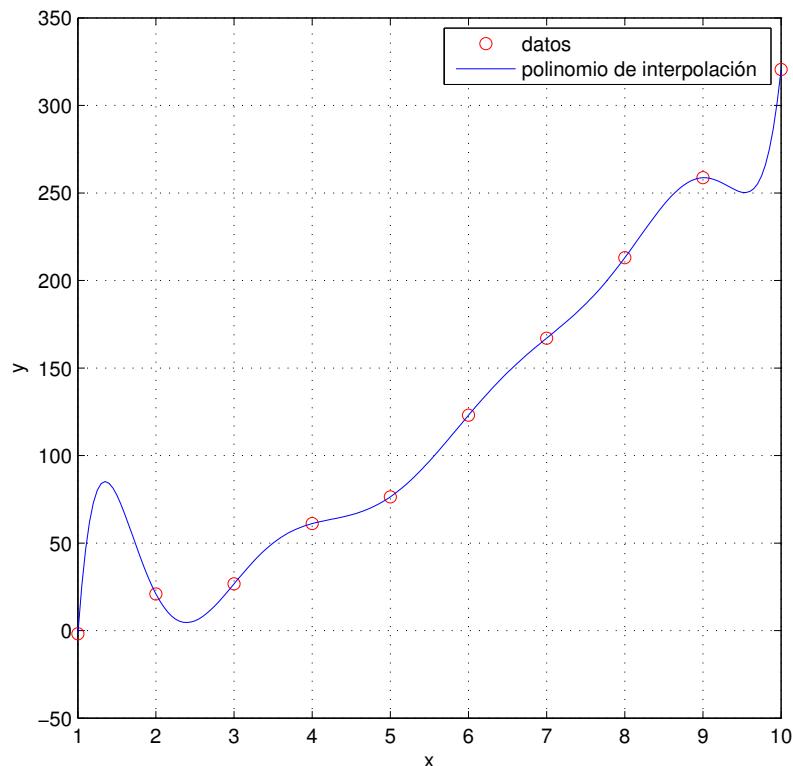


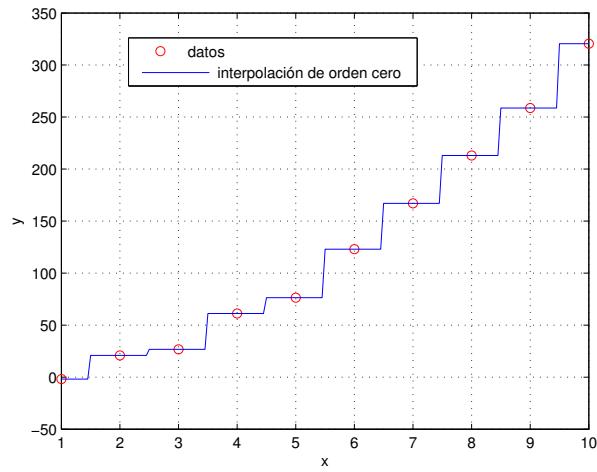
Figura 7.3: Polinomio de interpolación de grado nueve obtenido a partir de un conjunto de diez datos

En muchos casos es preferible no emplear todos los datos disponibles para obtener un único

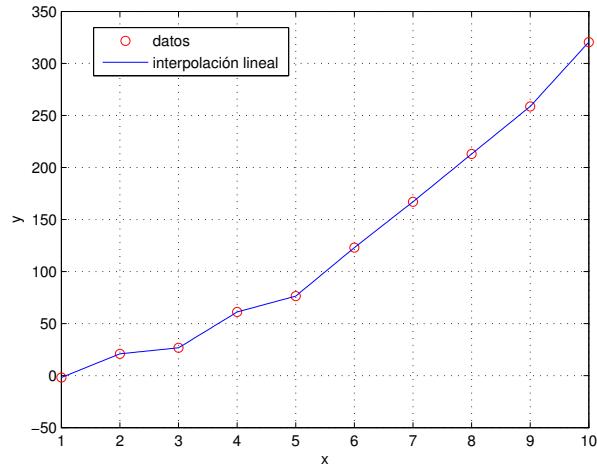
polinomio de interpolación. En su lugar, lo que se hace es dividir el conjunto de datos en varios grupos —normalmente se agrupan formando intervalos de datos consecutivos— y obtener varios polinomios de menor grado, de modo que cada uno interpole los datos de un grupo distinto.

El grado de los polinomios empleados deberá estar, en principio, relacionado con los datos contenidos en cada tramo.

interpolación de orden cero si hacemos que cada intervalo contenga un solo dato, obtendríamos polinomios de interpolación de grado cero, $a_{0i} = y_i$. El resultado, es un conjunto de escalones cuya valor varía de un intervalo a otro de acuerdo con el dato representativo contenido en cada tramo. La figura 7.4(a) muestra el resultado de la interpolación de orden cero para los mismos diez datos de la figura 7.3.



(a) Interpolación de orden cero



(b) Interpolación lineal

Figura 7.4: Interpolaciones de orden cero y lineal para los datos de la figura 7.3

interpolación lineal. En este caso, se dividen los datos en grupos de dos. Cada par de datos consecutivos se interpola calculando la recta que pasa por ellos. La interpolación lineal se emplea en muchas aplicaciones debido a su sencillez de cálculo. La figura 7.4(b), muestra el resultado de aproximar linealmente los mismos datos contenidos en los ejemplos anteriores.

Siguiendo el mismo procedimiento, aumentando el número de datos contenidos en cada intervalo, podríamos definir una interpolación cuadrática, con polinomios de segundo grado, tomando intervalos que contengan tres puntos, una interpolación cúbica, para intervalos de cuatro puntos etc.

7.4.1. Interpolación mediante splines cúbicos

Hemos descrito antes cómo el polinomio interpolador de orden n para un conjunto de $n+1$ datos puede presentar el inconveniente de complicar excesivamente la forma de la curva obtenida entre los puntos interpolados. La interpolación a tramos que acabamos de describir, simplifica la forma de la curva entre los puntos pero presenta el problemas de la continuidad en las uniones entre tramos sucesivos. Sería deseable encontrar métodos de interpolación que fueran capaces de solucionar ambos problemas simultáneamente. Una buena aproximación a dicha solución la proporcionan los *splines*.

Una función *spline* está formado por un conjunto de polinomios, cada uno definido en un intervalo, que se unen entre sí obedeciendo a ciertas condiciones de continuidad.

Supongamos que tenemos una tabla de datos cualquiera,

x	x_0	x_1	\cdots	x_n
y	y_0	y_1	\cdots	y_n

Para construir una función *spline* S de orden m , que interpole los datos de la tabla, se definen intervalos tomando como extremos dos puntos consecutivos de la tabla y un polinomio de grado m para cada uno de los intervalos,

$$S = \begin{cases} S_0(x), & x \in [x_0, x_1] \\ S_1(x), & x \in [x_1, x_2] \\ \vdots \\ S_i(x), & x \in [x_i, x_{i+1}] \\ \vdots \\ S_{n-1}(x), & x \in [x_{n-1}, x_n] \end{cases}$$

Para que S sea una función Spline de orden m debe cumplir que sea continua y tenga $m-1$ derivadas continuas en el intervalo $[x_0, x_n]$ en que se desean interpolar los datos.

Para asegurar la continuidad, los polinomios que forman S deben cumplir las siguientes condiciones en sus extremos;

$$\begin{aligned} S_i(x_{i+1}) &= S_{i+1}(x_{i+1}), \quad (1 \leq i \leq n-1) \\ S'_i(x_{i+1}) &= S'_{i+1}(x_{i+1}), \quad (1 \leq i \leq n-1) \\ S''_i(x_{i+1}) &= S''_{i+1}(x_{i+1}), \quad (1 \leq i \leq n-1) \\ &\vdots \\ S_i^{m-1}(x_{i+1}) &= S_{i+1}^{m-1}(x_{i+1}), \quad (1 \leq i \leq n-1) \end{aligned}$$

Es decir, dos polinomios consecutivos del spline y sus $m - 1$ primeras derivadas, deben tomar los mismos valores en el extremo común.

Una consecuencia inmediata de las condiciones de continuidad exigidas a los splines es que sus derivadas sucesivas, S' , S'' , \dots son a su vez funciones spline de orden $m - 1$, $m - 2$, \dots . Por otro lado, las condiciones de continuidad suministran $(n - 1) \cdot m$ ecuaciones que, unidas a las $n + 1$ condiciones de interpolación —cada polinomio debe pasar por los datos que constituyen los extremos de su intervalo de definición—, suministran un total de $n \cdot (m + 1) - (m - 1)$ ecuaciones. Este número es insuficiente para determinar los $(m + 1) \cdot n$ parámetros correspondientes a los n polinomios de grado m empleados en la interpolación. Las $m - 1$ ecuaciones que faltan se obtienen imponiendo a los splines condiciones adicionales.

Splines cúbicos. Los splines más empleados son los formados por polinomios de tercer grado. En total, tendremos que determinar $(m + 1) \cdot n = 4 \cdot n$ coeficientes para obtener todos los polinomios que componen el spline. Las condiciones de continuidad más la de interpolación suministran en total $3 \cdot (n - 1) + n + 1 = 4 \cdot n - 2$ ecuaciones. Necesitamos imponer al spline dos condiciones más. Algunas típicas son,

1. Splines naturales $S''(x_0) = S''(x_n) = 0$
2. Splines con valor conocido en la primera derivada de los extremos $S'(x_0) = y'_0$, $S'(x_n) = y'_n$
3. Splines periódicos,

$$\begin{cases} S(x_0) = S(x_n) \\ S'(x_0) = S'(x_n) \\ S''(x_0) = S''(x_n) \end{cases}$$

Intentar construir un sistema de ecuaciones para obtener a la vez todos los coeficientes de todos los polinomios es una tarea excesivamente compleja porque hay demasiados parámetros. Para abordar el problema partimos del hecho de que $S''(x)$ es también un spline de orden 1 para los puntos interpolados. Si los definimos como,

$$S''_i(x) = -M_i \frac{x - x_{i+1}}{h_i} + M_{i+1} \frac{x - x_i}{h_i}, \quad i = 0, \dots, n - 1$$

donde $h_i = x_{i+1} - x_i$ representa el ancho de cada intervalo y donde cada valor $M_i = S''(x_i)$ será una de las incógnitas que deberemos resolver.

Si integramos dos veces la expresión anterior,

$$\begin{aligned} S'_i(x) &= -M_i \frac{(x - x_{i+1})^2}{2 \cdot h_i} + M_{i+1} \frac{(x - x_i)^2}{2 \cdot h_i} + A_i, \quad i = 0, \dots, n - 1 \\ S_i(x) &= -M_i \frac{(x - x_{i+1})^3}{6 \cdot h_i} + M_{i+1} \frac{(x - x_i)^3}{6 \cdot h_i} + A_i(x - x_i) + B_i, \quad i = 0, \dots, n - 1 \end{aligned}$$

Empezamos por imponer las condiciones de interpolación: el polinomio S_i debe pasar por el punto (x_i, y_i) ,

$$S_i(x_i) = -M_i \frac{(x_i - x_{i+1})^3}{6 \cdot h_i} + B_i = y_i \Rightarrow B_i = y_i - \frac{M_i \cdot h_i^2}{6}, \quad i = 0, \dots, n - 1$$

A continuación imponemos continuidad del spline en los nodos comunes: El polinomio S_{i-1} también debe pasar por el punto (x_i, y_i) ,

$$S_{i-1}(x_i) = M_i \frac{(x_i - x_{i-1})^3}{6 \cdot h_i} + A_{i-1}(x_i - x_{i-1}) + y_{i-1} - \overbrace{\frac{M_{i-1} \cdot h_{i-1}^2}{6}}^{B_{i-1}} = y_i \Rightarrow \\ \Rightarrow A_{i-1} = \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{M_i - M_{i-1}}{6} \cdot h_{i-1}, \quad i = 1, \dots, n$$

Y por tanto,

$$A_i = \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} \cdot h_i, \quad i = 0, \dots, n-1$$

En tercer lugar imponemos la condición de que las derivadas también sean continuas en los nodos comunes,

$$S'_i(x_i) = -M_i \frac{(x_i - x_{i+1})^2}{2 \cdot h_i} + M_{i+1} \frac{(x_i - x_i)^2}{2 \cdot h_i} + \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} \cdot h_i, \quad i = 0, \dots, n-1 \\ S'_{i-1}(x_i) = -M_{i-1} \frac{(x_i - x_i)^2}{2 \cdot h_{i-1}} + M_i \frac{(x_i - x_{i-1})^2}{2 \cdot h_{i-1}} + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{M_i - M_{i-1}}{6} \cdot h_{i-1}, \quad i = 1, \dots, n \\ S'_i(x_i) = S'_{i-1}(x_i), \quad i = 1, \dots, n-1 \Rightarrow \\ \Rightarrow -M_i \frac{h_i}{2} + \frac{y_{i+1} - y_i}{h_i} - \frac{M_{i+1} - M_i}{6} \cdot h_i = M_i \frac{h_{i-1}}{2} + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{M_i - M_{i-1}}{6} \cdot h_{i-1}$$

Si agrupamos a un lado los valores M_{i-1}, M_i, M_{i+1} ,

$$h_{i-1} \cdot M_{i-1} + 2 \cdot (h_{i-1} + h_i) \cdot M_i + h_i \cdot M_{i+1} = 6 \cdot \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right) \\ i = 1, \dots, n-1$$

En total tenemos $M_0, \dots, M_n, n+1$ incógnitas y la expresión anterior, solo nos suministra $n-1$ ecuaciones. Necesitamos dos ecuaciones más. Si imponemos la condición de splines naturales, Para el extremo de la izquierda del primer polinomio y para el extremo de la derecha del último,

$$M_0 = S''(x_0) = 0 \\ M_n = S''(x_n) = 0$$

Con estas condiciones y la expresión obtenida para el resto de los M_i , podemos construir un sistema de ecuaciones tridiagonal

$$\begin{pmatrix} 2(h_0 + h_1) & h_1 & 0 & 0 & \cdots & 0 & 0 \\ h_1 & 2(h_1 + h_2) & h_2 & 0 & \cdots & 0 & 0 \\ 0 & h_2 & 2(h_2 + h_3) & h_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 2(h_{n-3} + h_{n-2}) & h_{n-2} \\ 0 & 0 & 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2} + h_{n-1}) \end{pmatrix} \cdot \begin{pmatrix} M_1 \\ M_2 \\ M_3 \\ \vdots \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

Donde hemos hecho,

$$b_i = 6 \cdot \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right)$$

Tenemos un sistema de ecuaciones en el que la matriz de coeficientes es tridiagonal y además diagonal dominante, por lo que podríamos emplear cualquiera de los métodos vistos en capítulo 6. Una vez resuelto el sistema y obtenidos los valores de M_i , obtenemos los valores de A_i y B_i a Partir de las ecuaciones obtenidas más arriba.

Por último, la forma habitual de definir el polinomio de grado 3 S_i , empleado para interpolar los valores del intervalo $[x_i, x_{i+1}]$, mediante splines cúbicos se define como,

$$S_i(x) = \alpha_i + \beta_i(x - x_i) + \gamma_i(x - x_i)^2 + \delta_i(x - x_i)^3, \quad x \in [x_i, x_{i+1}], \quad (i = 0, 1, \dots, n - 1)$$

Donde,

$$\begin{aligned}\alpha_i &= y_i \\ \beta_i &= \frac{y_{i+1} - y_i}{h_i} - \frac{M_i \cdot h_i}{3} - \frac{M_{i+1} \cdot h_i}{6} \\ \gamma_i &= \frac{M_i}{2} \\ \delta_i &= \frac{M_{i+1} - M_i}{6 \cdot h_i}\end{aligned}$$

La siguiente función permite obtener los coeficientes y el resultado de interpolar un conjunto de puntos mediante splines cúbicos,

```
function [c,yi]=spcubic(x,y,xi)
%uso [c,yi]=spcubic(x,y,xi)
%Esta funci?n interpola los puntos contenidos en los vectores columna x e y
%empleando parae ello splines c?bicos naturales. devuelve los coeficientes
%de los polinomios en una matriz de dimension (n-1)*4. Cada fila contiene
%un splin desde So a Sn-1 los coeficiente est?n guardados el la fila en
%orden depotencia creciente: ejemplo c(i,1)+c(i,2)*x+c(i,3)*x^2+c(i,4)*x^3
%ademas devuelve los valores interpolados yi correspondientes a puntos xi
%contenidos en el intervalo definido por los valores de x

%obtenemos la longitud de los datos...
l=length(x);
%obtencion de los coeficientes M
%Construimos el vector de diferencias h y un vector de diferencias
%Dy=y(i+1)-y(i) Que nos ser? muy ?til a la hora de calcular el vector de
%t?rminos independientes del sistema

for i=1:l-1
    h(i)=x(i+1)-x(i);
    Dy(i)=y(i+1)-y(i);
end

%Construimos la matriz del sistema. (Lo ideal seria definirla como una
```

```
%matriz sparse pero en fin no sabeis, porque sois todavia?os, etc...)

CSP(1-2,1-2)=2*(h(1-2)+h(1-1));
b(1-2,1)=6*(Dy(1-1)/h(1-1)-Dy(1-2)/h(1-2));
for i=1:l-3
    CSP(i,i)=2*(h(i)+h(i+1));
    CSP(i,i+1)=h(i+1);
    CSP(i+1,i)=h(i+1);
    b(i,1)=6*(Dy(i+1)/h(i+1)-Dy(i)/h(i));
end

%calculamos los coeficientes M,
M=CSP\b;

%A?adimos el primer y el ?ltimo valor como ceros (Splines naturales)
M=[0;M;0];

%calulamos los coeficientes A,
for i=1:l-1
    A(i)=Dy(i)/h(i)-(M(i+1)-M(i))*h(i)/6;
end
%Calculamos los coeficientes B,
for i=1:l-1
    B(i)=y(i)-M(i)*h(i)^2/6;
end
%Podemos ahora calcular el valor que toma el polinomio para los puntos
%que se desea interpolar

%miramos cuantos puntos tenemos
l2=length(xi);
for i=1:l2
    %miramos en que intervalo esta el punto xi(i)
    j=1;
    while xi(i)>x(j)
        j=j+1;
    end
    if j>l-1
        j=l-1; %aunque estamos extrapolando
    elseif j<2
        j=2; %estamos calculando el primer punto o estamos tambien
        %extrapolando
    end

    yi(i)=-M(j-1)*(xi(i)-x(j))^3/(6*h(j-1))+...
        M(j)*(xi(i)-x(j-1))^3/(6*h(j-1))+A(j-1)*(xi(i)-x(j-1))+B(j-1);
end

%calcumos los coeficientes c del spline en forma 'normal'
%la primera columna, son los valores de i,
c=zeros(l-1,4);
```

```

for i=1:l-1
    c(i,1)=y(i);
    c(i,2)=Dy(i)/h(i)-M(i)*h(i)/3-M(i+1)*h(i)/6;
    c(i,3)=M(i)/2;
    c(i,4)=(M(i+1)-M(i))/(6*h(i));
end

```

La figura, 7.5 muestra el resultado de interpolar mediante un spline cúbico, los datos contenidos en la figura 7.3. Es fácil observar cómo ahora los polinomios de interpolación dan como resultado una curva suave en los datos interpolados y en la que además las curvas son también suaves, sin presentar variaciones extrañas, para los puntos contenidos en cada intervalo entre dos datos.

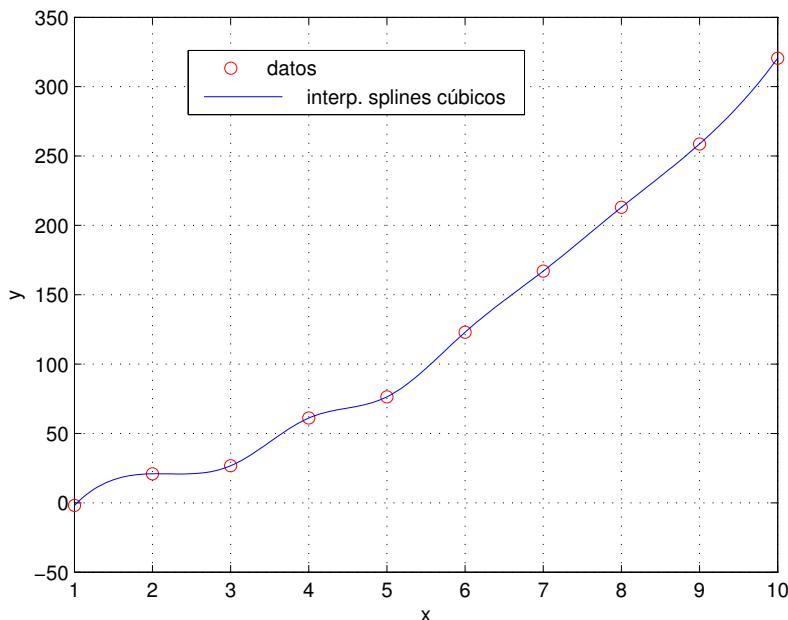


Figura 7.5: Interpolación mediante spline cúbico de los datos de la figura 7.3

7.4.2. Funciones propias de Matlab para interpolación por intervalos

Para realizar una interpolación por intervalos mediante cualquiera de los procedimientos descritos, Matlab incorpora la función propia `interp1`. Esta función admite como variables de entrada dos vectores con los valores de las coordenadas x e y de los datos que se desea interpolar y un tercer vector $x1$ con los puntos para los que se desea calcular el valor de la interpolación. Además, admite como variable de entrada una cadena de caracteres que indica el método con el que se quiere realizar la interpolación. Dicha variable puede tomar los valores:

1. '`nearest`'. Interpola el intervalo empleando el valor y_i correspondiente al valor x_i más cercano al punto que se quiere interpolar. El resultado es una interpolación a escalones.
2. '`linear`' realiza una interpolación lineal entre los puntos del conjunto de datos que se desea interpolar.

3. 'spline'. Interpola empleando splines cúbicos naturales.
4. 'cubic' o 'pchip' Emplea polinomios de Hermite cúbicos. Es un método similar al de los splines que no describiremos en estos apuntes.

La función devuelve como salida los valores interpolados correspondientes a los puntos de $x1$. El siguiente código muestra el modo de usar el comando `interp1`. Para probarlo se han creado dos vectores x e y que contienen el conjunto de datos que se empleará para calcular la interpolación. Además, se ha creado otro vector $x1$ que contiene los puntos para los que se quiere calcular el resultado de la interpolación.

```
>> x=[1:2:16]
x =
    1      3      5      7      9     11     13     15
>> y=[1 3 4 2 -1 4 -5 3]
y =
    1      3      4      2     -1      4     -5      3
>> x1=[3.5 7.5]
x1 =
    3.5000    7.5000
>> y1=interp1(x,y,x1,'spline')
y1 =
    3.4110    0.7575
>> x=[1:2:16]
x =
    1      3      5      7      9     11     13     15
>> y=[1 3 4 2 -1 4 -5 3]
y =
    1      3      4      2     -1      4     -5      3
>> x1=[3.5 7.5]
x1 =
    3.5000    7.5000
>> y1=interp1(x,y,x1,'nearest')
y1 =
    3      2
>> y1=interp1(x,y,x1,'linear')
y1 =
    3.2500    1.2500
>> y1=interp1(x,y,x1,'cubic')
y1 =
    3.3438    1.1938
```

7.5. Ajuste polinómico por el método de mínimos cuadrados

Los métodos de interpolación que hemos descrito en las secciones anteriores pretenden encontrar un polinomio o una función definida a partir de polinomios que pase por un conjunto de datos. En el caso del ajuste por mínimos cuadrados, lo que se pretende es buscar el polinomio, de un grado dado, que mejor se aproxime a un conjunto de datos.

Supongamos que tenemos un conjunto de m datos,

x	x_1	x_2	\cdots	x_m
y	y_1	y_2	\cdots	y_m

Queremos construir un polinomio $p(x)$ de grado $n < m - 1$, de modo que los valores que toma el polinomio para los datos $p(x_i)$ sean lo más cercanos posibles a los correspondientes valores y_i .

En primer lugar, necesitamos clarificar qué entendemos por *lo mas cercano posible*. Una posibilidad, es medir la diferencia, $y_i - p(x_i)$ para cada par de datos del conjunto. Sin embargo, es más frecuente emplear el cuadrado de dicha diferencia, $(y_i - p(x_i))^2$. Esta cantidad tiene, entre otras, la ventaja de que su valor es siempre positivo con independencia de que la diferencia sea positiva o negativa. Además, representa el cuadrado de la distancia entre $p(x_i)$ e y_i . Podemos tomar la suma de dichas distancias al cuadrado, obtenidas por el polinomio para todos los pares de puntos,

$$\sum_{i=1}^m (y_i - p(x_i))^2$$

como una medida de la distancia del polinomio a los datos. De este modo, el polinomio *lo mas cercano posible* a los datos sería aquel que minimice la suma de diferencias al cuadrado que acabamos de definir. De ahí el nombre del método.

En muchos casos, los datos a los que se pretende ajustar un polinomio por mínimos cuadrados son datos experimentales. En función del entorno experimental y del método con que se han adquirido los datos, puede resultar que algunos resulten más fiables que otros. En este caso, sería deseable hacer que el polinomio se aproxime más a los datos más fiables. Una forma de hacerlo es añadir unos *pesos*, ω_i , a las diferencias al cuadrado en función de la confianza que nos merece cada dato,

$$\sum_{i=1}^m \omega_i (y_i - p(x_i))^2$$

Los datos fiables se multiplican por valores de ω grandes y los poco fiables por valores pequeños.

Para ver cómo obtener los coeficientes de un polinomio de mínimos cuadrados, empezaremos con el caso más sencillo; un polinomio de grado 0. En este caso, el polinomio es una constante, definida por su término independiente $p(x) = a_0$. El objetivo a minimizar sería entonces,

$$g(a_0) = \sum_{i=1}^m \omega_i (y_i - a_0)^2$$

El valor mínimo de esta función debe cumplir que su derivada primera $g'(a_0) = 0$ y que su derivada segunda $g''(a_0) \geq 0$,

$$g'(a_0) = -2 \sum_{i=1}^m \omega_i (y_i - a_0) = 0 \Rightarrow a_0 = \frac{\sum_{i=1}^m \omega_i \cdot y_i}{\sum_{i=1}^m \omega_i}$$

$$g''(a_0) = 2 \sum_{i=1}^m \omega_i \Rightarrow g''(a_0) \geq 0$$

El resultado obtenido para el valor de a_0 es una media, ponderada con los pesos w_i de los datos. Si hacemos $w_i = 1 \forall w_i$ obtendríamos exactamente la media de los datos. Este resultado resulta bastante razonable. Aproximar un conjunto de valores por un polinomio de grado cero, es tanto como suponer que la variable y permanece constante para cualquier valor de x . Las diferencias

observadas deberían deberse entonces a errores aleatorios experimentales, y la mejor estima del valor de y será precisamente el valor medio de los valores disponibles. La figura 7.6 muestra el resultado de calcular el polinomio de mínimos cuadrados de grado cero para un conjunto de datos.

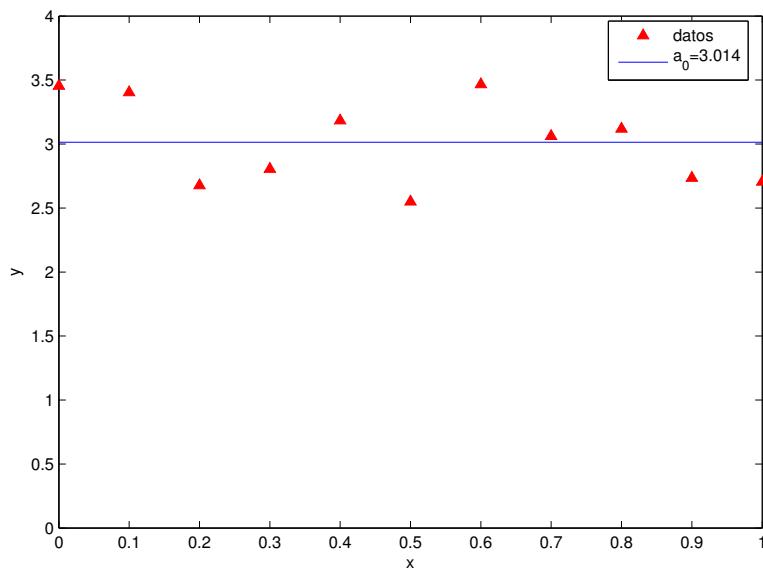


Figura 7.6: Polinomio de mínimos cuadrados de grado 0

El siguiente paso en dificultad sería tratar de aproximar un conjunto de datos por un polinomio de grado 1, es decir, por una linea recta, $p(x) = a_0 + a_1x$. En este caso, la suma de diferencias al cuadrado toma la forma,

$$g(a_0, a_1) = \sum_{i=1}^m \omega_i (y_i - a_0 - a_1 x_i)^2$$

En este caso, tenemos dos coeficientes sobre los que calcular el mínimo. Éste se obtiene cuando las derivadas parciales de $g(a_0, a_1)$ respecto a ambos coeficientes son iguales a cero.

$$\begin{aligned} \frac{\partial g}{\partial a_0} &= -2 \sum_{i=1}^m \omega_i (y_i - a_0 - a_1 x_i) = 0 \\ \frac{\partial g}{\partial a_1} &= -2 \sum_{i=1}^m \omega_i x_i (y_i - a_0 - a_1 x_i) = 0 \end{aligned}$$

Si reordenamos las ecuaciones anteriores,

$$\begin{aligned} \left(\sum_{i=1}^m \omega_i \right) a_0 + \left(\sum_{i=1}^m \omega_i x_i \right) a_1 &= \sum_{i=1}^m \omega_i y_i \\ \left(\sum_{i=1}^m \omega_i x_i \right) a_0 + \left(\sum_{i=1}^m \omega_i x_i^2 \right) a_1 &= \sum_{i=1}^m \omega_i x_i y_i \end{aligned}$$

Obtenemos un sistema de dos ecuaciones lineales cuyas incógnitas son precisamente los coeficientes de la recta de mínimos cuadrados.

Podemos ahora generalizar el resultado para un polinomio de grado n , $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. La función g toma la forma,

$$g(a_0, a_1, \dots, a_n) = \sum_{i=1}^m \omega_i (a_0 + a_1x_i + \cdots + a_nx_i^n - y_i)^2$$

De nuevo, para obtener los coeficientes del polinomio igualamos las derivadas parciales a cero,

$$\frac{\partial g(a_0, a_1, \dots, a_n)}{\partial a_j} = 0 \Rightarrow \sum_{i=1}^m \omega_i x_i^j (a_0 + a_1x_i + \cdots + a_nx_i^n - y_i) = 0, \quad j = 0, 1, \dots, n$$

Si reordenamos las expresiones anteriores, llegamos a un sistema de $n+1$ ecuaciones lineales, cuyas incógnitas son los coeficientes del polinomio de mínimos cuadrados,

$$\begin{pmatrix} s_0 & s_1 & \cdots & s_n \\ s_1 & s_2 & \cdots & s_{n+1} \\ \vdots & \vdots & \ddots & \\ s_n & s_{n+1} & \cdots & s_{2n} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}$$

Donde hemos definido s_j y c_j como,

$$s_j = \sum_{i=1}^m \omega_i x_i^j$$

$$c_j = \sum_{i=1}^m \omega_i x_i^j y_i$$

El siguiente código permite obtener el polinomio de mínimos cuadrados que aproxima un conjunto de n datos,

```
function a=mc(x,y,n,w)
%uso: a=mc(x,y,n,w). Esta función permite obtener los coeficientes del
%polinomio de mínimos cuadrados que ajusta un conjunto de datos. Las
%variables de entrada son: x, vector con la componente x de los datos a
%ajustar. y, vector con la componente y de los datos a ajustar. n grado del
%polinomio de mínimos cuadrados. w vector de pesos asociados a los datos. si no
%se suministra se toman todos los pesos como 1. Salidas: vector columna con
%los coeficientes del polinomio=> a(1)+a(2)*x+a(3)*x^2+a(n+1)*x^n

%comprobamos en primer lugar que tenemos datos suficientes,
m=length(x);
if m<n+1
    error('no hay datos suficientes para calcular el polinomio pedido')
end
%Si no se ha suministrado vector de pesos construimos uno formado por unos,
if nargin < 4
```

```
w=ones(m,1);
end
%Montamos un bucle para crear los elementos s de la matriz de coeficientes,
for j=1:2*n+1
    s(j)=0;
for i=1:m
    s(j)=s(j)+w(i)*x(i)^(j-1);
end
end

%y un segundo bucle para crear los términos independientes...
for j=1:n+1
    c(j,1)=0;
    for i=1:m
        c(j,1)=c(j,1)+w(i)*x(i)^(j-1)*y(i)
    end
end

% a partir de los valores de s, construimos la matriz del sistema,
for i=1:n+1
    for j=1:n+1
        A(i,j)=s(i+j-1)
    end
end

%solo nos queda resolver el sistema... Empleamos la division por la
%izquierda de matlab

a=A\c;
```

Una última observación importante es que si intentamos calcular el polinomio de mínimos cuadrados de grado $m - 1$ que aproxima un conjunto de m datos, lo que obtendremos será el polinomio de interpolación. En general, cuanto mayor sea el grado del polinomio más posibilidades hay de que la matriz de sistema empleado para obtener los coeficientes del polinomio esté mal condicionada.

7.5.1. Mínimos cuadrados en Matlab.

Matlab suministra dos maneras distintas de ajustar un polinomio a unos datos por mínimos cuadrados. La primera es mediante el uso del comando `polyfit`. Este comando admite como entradas un vector de coordenadas x y otro de coordenadas y , de los datos que se quieren aproximar, y una tercera variable con el grado del polinomio. Como resultado devuelve un vector con los coeficientes del polinomio de mínimos cuadrados. Ordenados en orden de potencias decrecientes $a(1)*x^n+a(2)*x^{n-1}+\dots+a(n+1)$. La siguiente secuencia de comandos crea un par de vectores y muestra como manejar el comando,

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

```
>> y=[-1 0 3 4 5.2 6 10 12 13 15.5];\index{Raíces de un polinomio}
>> a=polyfit(x,y,3)
a =
    0.0001    0.0411    1.3774   -2.4133
```

A continuación, podemos emplear el comando `polyval`, para obtener en valor del polinomio de mínimos cuadrados obtenido en cualquier punto. En particular, si lo aplicamos a los datos x ,

```
>> yhat=polyval(a,x)
yhat =
    Columns 1 through 7
    -0.9947    0.5067    2.0913    3.7596    5.5121    7.3491    9.2713
    Columns 8 through 10
    11.2790   13.3727   15.5529
```

Por último, podemos calcular el error cometido por el polinomio $e_i = |p(x_i) - y_i|$

```
>> error=abs(yhat-y)
error =
    Columns 1 through 7
    0.0053    0.5067    0.9087    0.2404    0.3121    1.3491    0.7287
    Columns 8 through 10
    0.7210    0.3727    0.0529
```

Además del comando `polyfit` Matlab permite ajustar un polinomio por mínimos cuadrados a un conjunto de datos a través de la ventana gráfica de Matlab. Para ello, es suficiente representar los datos con el comando `plot(x,y)`. Una vez que Matlab muestra la ventana gráfica con los datos representados, se selecciona en el menú desplegable `tools` la opción `Basic Fitting`. Matlab abre una segunda ventana que permite seleccionar el polinomio de mínimos cuadrados que se desea ajustar a los datos, así como otras opciones que permiten obtener los coeficientes del polinomio y analizar la bondad del ajuste a partir de los residuos (ver sección siguiente). La figuras 7.7, muestra un ejemplo de uso de la ventana gráfica de Matlab para obtener un ajuste por mínimos cuadrados

7.5.2. Análisis de la bondad de un ajuste por mínimos cuadrados.

Supongamos que tenemos un conjunto de datos obtenidos como resultado de un experimento. En muchos casos la finalidad de un ajuste por mínimos cuadrados, es encontrar una ley que nos permita relacionar los datos de la variable independiente con la variable dependiente. Por ejemplo si aplicamos distintas fuerzas a muelle y medimos la elongación sufrida por el muelle, esperamos obtener, en primera aproximación una relación lineal: $\Delta x \propto F$. (Ley de Hooke).

Sin embargo, los resultados de un experimento no se ajustan nunca exactamente a una ley debido a errores aleatorios que no es posible corregir.

Cuando realizamos un ajuste por mínimos cuadrados, podemos emplear cualquier polinomio desde grado 0 hasta grado $m-1$. Desde el punto de vista del error cometido con respecto a los datos disponibles el mejor polinomio sería precisamente el de grado $m-1$ que da error cero para todos los datos, por tratarse del polinomio de interpolación. Sin embargo, si los datos son experimentales estamos incluyendo los errores experimentales en el ajuste.

Por ello, para datos experimentales y suponiendo que los datos solo contienen errores aleatorios, el mejor ajuste lo dará el polinomio de menor grado para el cual las diferencias entre los datos y el polinomio $y_i - p(x_i)$ se distribuyan aleatoriamente. Estas diferencias reciben habitualmente el nombre de residuos.

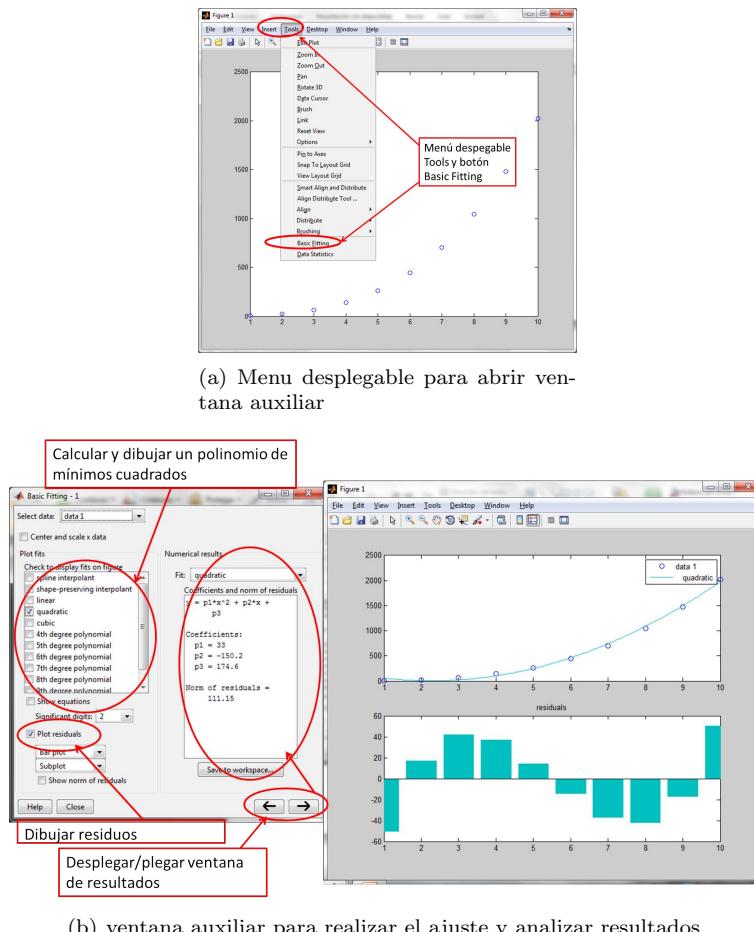
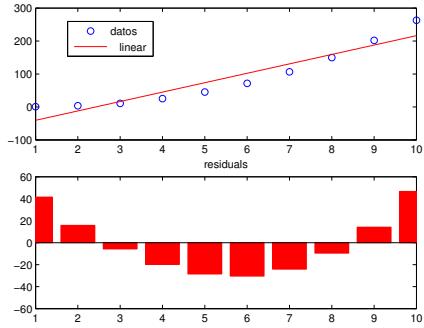


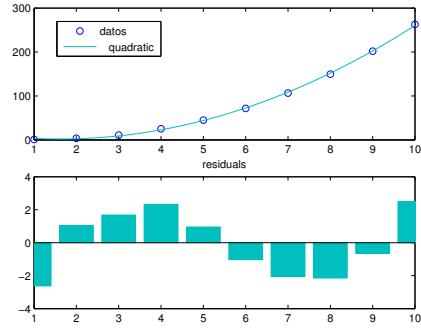
Figura 7.7: Ejemplo de uso de la ventana gráfica de Matlab para realizar un ajuste por mínimos cuadrados

Entre las herramientas que suministra la ventanas gráficas de Matlab para el ajuste por mínimos cuadrados hay una que calcula y representa los residuos. La figura 7.8 muestra un ejemplo de ajuste por mínimos cuadrados empleando cada vez un polinomio de mayor grado.

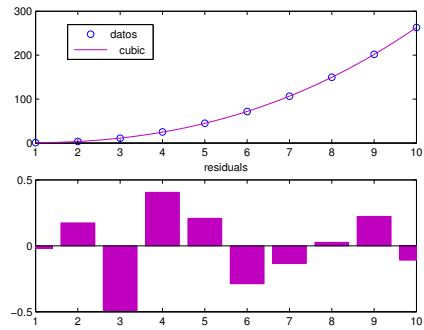
En la figura 7.8(a) se observa claramente que el ajusto no es bueno, la recta de mínimos cuadrados no es capaz de adaptarse a la forma que presentan los datos. Los residuos muestran claramente esta tendencia: no están distribuidos de forma aleatoria. En la figura 7.8(b), la parábola aproxima mucho mejor el conjunto de datos, a simple vista parece un buen ajuste. Sin embargo, los residuos presenta una forma funcional clara que recuerda la forma de un polinomio de tercer grado. En la figura 7.8(c), los residuos están distribuidos de forma aleatoria. Si comparamos estos resultados con los de la figura 7.8(d) vemos que en este último caso los residuos son más pequeños, pero conservan esencialmente la misma distribución aleatoria que en la figura anterior. La aproximación de los datos empleando un polinomio de cuarto grado no añade información sobre la forma de la función que siguen los datos, y ha empezado a incluir en el ajuste los errores de los datos.



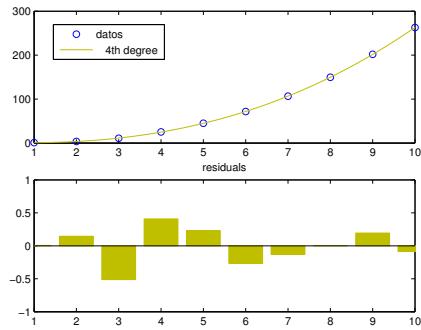
(a) Recta de mínimos cuadrados y residuos obtenidos



(b) Parábola de mínimos cuadrados y residuos obtenidos



(c) polinomio de tercer grado de mínimos cuadrados y residuos obtenidos



(d) polinomio de cuarto grado de mínimos cuadrados y residuos obtenidos

Figura 7.8: Comparación entre los residuos obtenidos para los ajustes de mínimos cuadrados de un conjunto de datos empleando polinomios de grados 1 a 4.

7.6. Curvas de Bézier

Las curvas de Bézier permiten unir puntos mediante curvas de forma arbitraria. Una curva de Bézier viene definida por un conjunto de $n + 1$ puntos, $\{\vec{p}_0, \vec{p}_2, \dots, \vec{p}_n\}$, que reciben el nombre de puntos de control. El orden en que se presentan los puntos de control es importante para la definición de la curva de Bézier correspondiente. Ésta pasa siempre por los puntos \vec{p}_0 y \vec{p}_n . El resto de los puntos de control permiten dar forma a la curva, que tiene siempre la propiedad de ser tangente en el punto \vec{p}_0 a la recta que une \vec{p}_0 con \vec{p}_1 y tangente en el punto \vec{p}_n a la recta que une \vec{p}_{n-1} con \vec{p}_n .

Para definir la curva, se asocia a cada punto de control \vec{p}_i una función conocida con el nombre de función de fusión. En el caso de las curvas de Bézier, las funciones de fusión empleadas son los polinomios de Bernstein,

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i, \quad i = 0, 1, \dots, n$$

El grado de los polinomios de Bernstein empleados está asociado al número de puntos de control; para un conjunto de $n + 1$ puntos, los polinomios empleados son de grado n . La variable t es un

parámetro que varía entre 0 y 1.

La ecuación de la curva de Bézier que pasa por un conjunto de $n + 1$ puntos de control, $\{\vec{p}_0, \vec{p}_2, \dots, \vec{p}_n\}$, se define a partir de los polinomios de Bernstein como,

$$\vec{p}(t) = \sum_{i=0}^n B_i^n(t) \cdot \vec{p}_i = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i \cdot \vec{p}_i$$

La expresión anterior da como resultado \vec{p}_0 si hacemos $t = 0$ y \vec{p}_n si hacemos $t = 1$. para los valores comprendidos entre $t = 0$ y $t = 1$, los puntos $\vec{p}(t)$ trazarán una curva entre \vec{p}_0 y \vec{p}_n .

Veamos un ejemplo sencillo. Supongamos que queremos unir los puntos $\vec{p}_0 = (0, 1)$ y $\vec{p}_n = (3, 1)$ mediante curvas de Bézier. Si no añadimos ningún punto más de control, $\{\vec{p}_0 = (0, 1), \vec{p}_1 = (3, 1)\}$, el polinomio de Bézier que obtendremos será una recta que unirá los dos puntos,

$$\vec{p}(t) = \binom{1}{0} (1-t)^1 t^0 \cdot (0, 1) + \binom{1}{1} (1-t)^0 t^1 \cdot (3, 1) = (1-t) \cdot (0, 1) + t \cdot (3, 1)$$

Si sepáramos las componentes x e y del vector $\vec{p}(t)$,

$$x = 3t$$

$$y = 1$$

Se trata de la ecuación del segmento horizontal que une los puntos $\vec{p}_0 = (0, 1)$ y $\vec{p}_n = (3, 1)$

Si añadimos un punto más de control, por ejemplo: $\vec{p}_1 = (1, 2) \rightarrow \{\vec{p}_0 = (0, 1), \vec{p}_1 = (1, 2), \vec{p}_2 = (3, 1)\}$, la curva resultante será ahora un segmento de un polinomio de segundo grado en la variable t ,

$$\begin{aligned} \vec{p}(t) &= \binom{2}{0} (1-t)^2 t^0 \cdot (0, 1) + \binom{2}{1} (1-t) t \cdot (1, 2) + \binom{2}{2} (1-t)^0 t^2 \cdot (3, 1) = \\ &= (1-t)^2 \cdot (0, 1) + 2(1-t)t \cdot (1, 2) + t^2 \cdot (3, 1) \end{aligned}$$

Según vamos aumentando el número de los puntos de control, iremos empleando polinomios de mayor grado. El segmento de polinomio empleado en cada caso para unir los puntos de control inicial y final variará dependiendo de los puntos de control intermedios empleados.

La figura 7.9 muestra un ejemplo en el que se han construido varias curvas de Bézier sobre los mismos puntos de paso inicial y final. Es fácil observar cómo la forma de la curva depende del número y la posición de los puntos de control. Si unimos éstos por orden mediante segmentos rectos, obtenemos un polígono que recibe el nombre de polígono de control. En la figura 7.9 se han representado los polígonos de control mediante líneas de puntos.

El siguiente código permite calcular y dibujar una curva de Bézier a partir de sus puntos de control.

```
function ber = bezier(p,res)
%estas funcion pinta la curva de Bezier correspondiente a los puntos de
%control p.
%p debe ser una matriz de dimension n*2 donde n es el número de puntos de
%control empleados La primera columna contiene las coordenadas x
%y la segunda las coordenadas y.
%El codigo sigue directamente la definicion de los polinomios de Bernstein
%por tanto, NO es un codigo optimo. Calcula varias veces las mismas
%cantidades.
%Los coeficientes binomiales de los polinomios de Berstein se
```

```
%pueden calcular directamente con el comando de matlab nchoosek. Sin
%embargo, en el programa se han calculado a partir de la definicion:
% $n!/(n-k)!k!$ 
%la variable de entrada res nos da el paso que empleará el parámetro t
%para calcular y dibujar el polinomio
```

```
t = 0:res:1;
ber = zeros(2,length(t));
%calculamos un vector de coeficientes para los polinomios.
```

```
%coeficientes para la coordenada x.
```

```
n = size(p,1)-1;
num = factorial(n);
for i = 0:n
    f=num/factorial(i)/factorial(n-i);
    ber(1,:) = ber(1,:) + f*p(i+1,1)*(1-t).^(n-i).*t.^i;
    ber(2,:) = ber(2,:) + f*p(i+1,2)*(1-t).^(n-i).*t.^i;
end
plot(p(:,1),p(:,2),'or')
hold on
plot(p(:,1),p(:,2),'r')
plot(ber(1,:),ber(2,:))
```

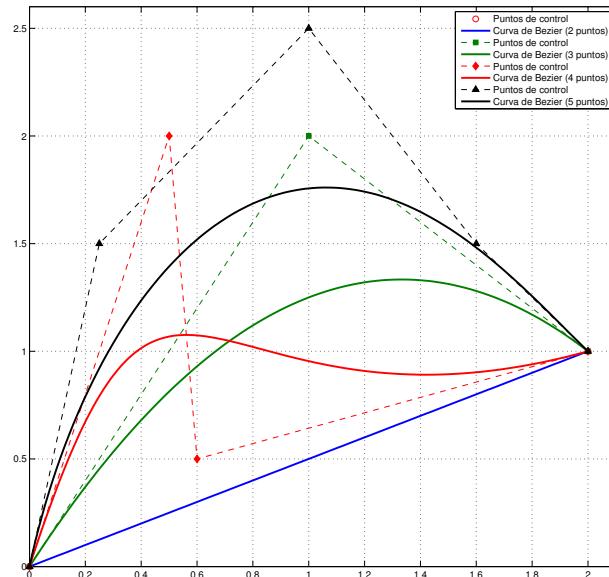


Figura 7.9: Curvas de Bézier trazadas entre los puntos $P_0 = (0,0)$ y $P_n = (2,1)$, variando el número y posición de los puntos de control.

Curvas equivalentes de grado superior. Dada una curva de Bézier, representada por un polinomio de Bernstein de grado n , es posible encontrar polinomios de grado superior que representan la misma curva de Bézier. Lógicamente esto supone un aumento del número de puntos de control.

Supongamos que tenemos una curva de Bézier con cuatro puntos de control,

$$\vec{p}(t) = \vec{p}_0 B_0^3(t) + \vec{p}_1 B_1^3(t) + \vec{p}_2 B_2^3(t) + \vec{p}_3 B_3^3(t)$$

Si multiplicamos este polinomio por $t + (1 - t) \equiv 1$ El polinomio no varía y por tanto representa la misma curva de Bézier. Sin embargo, tendríamos ahora un polinomio un grado superior al inicial. Si después de la multiplicación agrupamos términos de la forma $(1 - t)^{n-1} t^i$, Podríamos obtener el valor de los nuevos puntos de control,

$$\begin{aligned}\vec{p}_0^+ &= \vec{p}_0 \\ \vec{p}_1^+ &= \frac{1}{4} \vec{p}_0 + \frac{3}{4} \vec{p}_1 \\ \vec{p}_2^+ &= \frac{2}{4} \vec{p}_2 + \frac{2}{4} \vec{p}_3 \\ \vec{p}_3^+ &= \frac{3}{4} \vec{p}_2 + \frac{1}{4} \vec{p}_3 \\ \vec{p}_4^+ &= \vec{p}_3\end{aligned}$$

y, en general, los puntos de control de la curva de Bézier de grado $n + 1$ equivalente a una dada de grado n puede obtenerse como,

$$\vec{p}_i^+ = \alpha_i \vec{p}_{i-1} + (1 - \alpha_i) \vec{p}_i, \quad \alpha_i = \frac{i}{n+1}$$

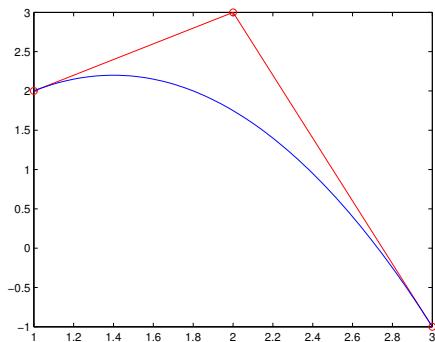
Mediante la ecuación anterior, es posible obtener iterativamente los puntos de control de la curva de Bézier equivalente a una dada de cualquier grado. Una propiedad interesante es que según aumentamos el número de puntos de control empleados, estos y el polígono de control correspondiente, van convergiendo a la curva de Bézier.

La figura 7.10 muestra un ejemplo para una curva de Bézier construida a partir de tres puntos de control. Es fácil ver cómo a pesar de aumentar el número de puntos de control, la curva resultante es siempre la misma. También es fácil ver en la figura (7.10(d)) la convergencia de los polígonos de control hacia la curva.

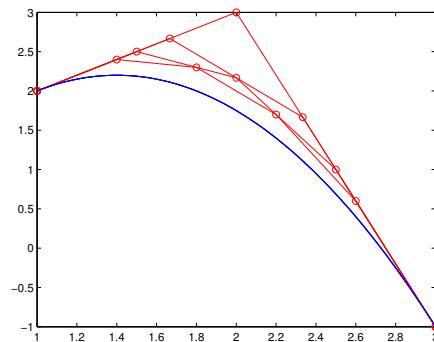
El siguiente código permite calcular la curva equivalente de Bézier a una dada, para cualquier número de puntos de control que se desee.

```
function pp = bzeq(p,n)
%Esta función calcula los puntos de control de una curva de Bezier
%equivalente de grado superior. p es matriz de tamaño mx2 que contiene los
%puntos de control originales. n es el número de puntos de paso de la nueva
%curva de Bezier equivalente. n tiene que ser mayor que m.
%pp contiene los n puntos de control de la curva de Bézier resultante.

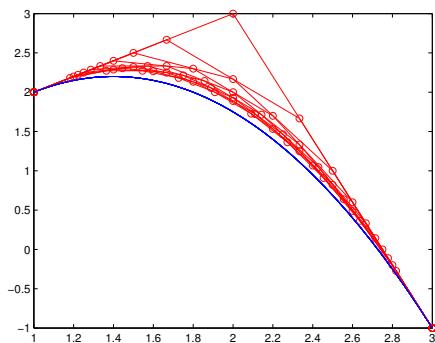
%Construimos una matriz con una fila mas para guardar un nuevo elemento
c = size(p,1);
%La siguiente linea sirve solo para ir pintando los resultados y ver que
%efectivamente todos los polinomios dan la misma curva de Bezier...
```



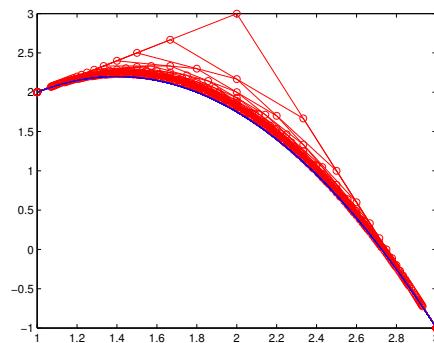
(a) Curva original (3 puntos)



(b) Curvas equivalentes de 4 a 6 puntos



(c) Curvas equivalentes de 4 a 12 puntos



(d) Curvas equivalentes de 4 a 30 puntos

Figura 7.10: Curvas de Bézier equivalentes, construidas a partir de una curva con tres puntos de control

```
% (si se tiene la funcion bezier).
bezier(p,0.01);
if c == n
    pp = p;
    return
else
    p = [p;zeros(1,2)];
    pp = p;
    for i = 2 : c+1
        pp(i,:) = p(i-1,:)*(i-1)/(c) + (1 -(i-1)/(c))*p(i,:);
    end
    %llamamos a la funcion recursivamente hasta tener n puntos de control
    %el grado del polinomio sera n-1...
    pp = bzeq(pp,n);
end
```

Derivadas. Las derivadas de una curva de Bézier con respecto al parámetro t son particularmente fáciles de obtener a partir de los puntos de control. Si tenemos una curva de Bézier de grado n , definida mediante puntos de control \vec{p}_i su derivada primera con respecto a t será una curva de Bézier de grado $n - 1$. Cuyos puntos de control \vec{d}_i puede obtenerse como:

$$\vec{d}_i = n (\vec{p}_{i+1} - \vec{p}_i)$$

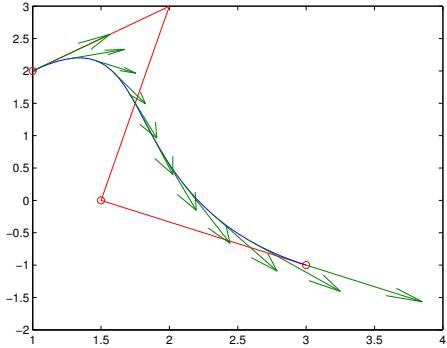
La nueva curva de Bézier obtenida de este modo, es una hodógrafa; representa el extremo de un vector tangente en cada punto a la curva de Bézier original y guarda una relación directa con la velocidad a la que se recorrería dicha curva.

La figura 7.11(a), muestra una curva de Bézier sobre la que se ha trazado el vector derivada para algunos puntos. La figura 7.11(b) muestra la hodógrafa correspondiente y de nuevo los mismos vectores derivada de la figura 7.11(a).

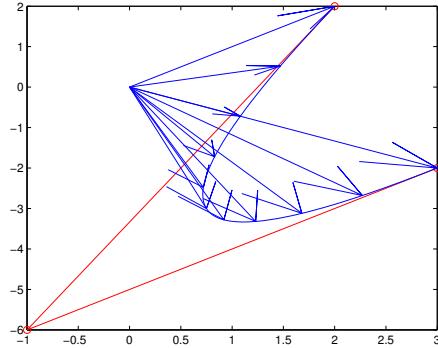
El siguiente código muestra como calcular los puntos de control de la derivada de una curva de Bézier.

```
function d =dbez(p)
%esta función obtiene los puntos de control de la derivada con respecto al
%parametro de una curva de bezier cuyos puntos de control están contenidos
% la matriz p. p de be ser una matriz de nx2 donde n es el número de puntos
% de control.
grado = size(p,1) -1
d = zeros(grado,2)
for i = 1:grado
    d(i,:) = (grado-1)*(p(i+1,:)-p(i,:))
end
%Código añadido para dibujar y analizar los resultados
%pintamos la hodografa
v = bezier(d,0.01)
%le añadimos algunos vectores para que se vea que tocan la hodografa
hold on
compass(v(1,1:10:size(v,2)),v(2,1:10:size(v,2)))
figure
%las pintamos además como vectores tangentes a la curva original
x = bezier(p,0.01)
hold on
quiver(x(1,1:10:size(v,2)),x(2,1:10:size(v,2)),v(1,1:10:size(v,2))...
,v(2,1:10:size(v,2)))
```

Interpolación con curvas de Bézier Podemos emplear curvas de Bézier para interpolar un conjunto de puntos $\{\vec{p}_0, \dots, \vec{p}_m\}$. Si empleamos un curva para interpolar cada par de puntos, $\vec{p}_i, \vec{p}_{i+1}, \quad i = 1, \dots, m - 1$ tenemos asegurada la continuidad en los puntos interpolados puesto que las curvas tienen que pasar por ellos. Como en el caso de la interpolación mediante splines, podemos imponer continuidad en las derivadas para conseguir una curva de interpolación suave. En el caso de las curvas de Bézier esto es particularmente simple. Si llamamos B a la curva de Bézier de grado n construida entre los puntos \vec{p}_{i-1}, \vec{p}_i con puntos de control, $\vec{p}_{i-1}, b_1, \vec{b}_2, \dots, \vec{b}_{n-1}, \vec{p}_i$. Y C a la curva de Bézier de grado s construida entre los puntos \vec{p}_i, \vec{p}_{i+1} con puntos de control, $\vec{p}_i, \vec{c}_1, \vec{c}_2, \dots, \vec{c}_{s-1}, \vec{p}_{i+1}$. Para asegurar la continuidad en la primera derivada en el punto \vec{p}_i basta imponer,



(a) Curva de Bézier (4 puntos)



(b) Derivada (hodógrafa)

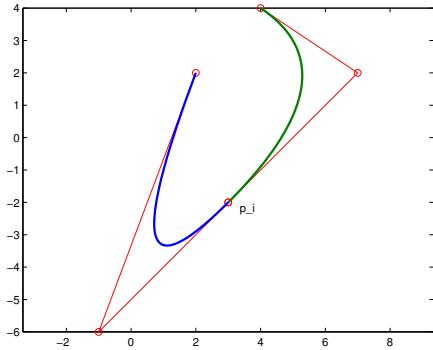
Figura 7.11: Curva de Bézier y su derivada con respecto al parámetro del polinomio de Bernstein que la define: $t \in [0, 1]$

$$n \cdot (\vec{p}_i - \vec{b}_{n-1}) = s \cdot (\vec{c}_1 - \vec{p}_i)$$

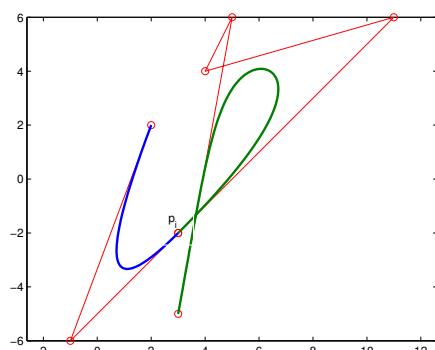
Esta condición impone una relación entre el penúltimo punto de control de la curva B y el segundo punto de control de la curva C . Pero deja completa libertad sobre el resto de los puntos de control elegidos para construir las curvas.

Podemos, por ejemplo, elegir libremente todos los puntos de control de la curva B y obtener a partir de ella el punto \vec{c}_1 ,

$$\vec{c}_1 = \frac{n+s}{s} \vec{p}_i - \frac{n}{s} \vec{b}_{n-1}$$



(a) Interpolación mediante curvas de Bézier de 3 puntos



(b) Interpolación mediante curvas de Bézier de 3 y 4 puntos

Figura 7.12: Interpolación de tres puntos mediante dos curvas de Bézier

La figura 7.12 muestra un ejemplo de interpolación en la que se ha aplicado la condición de continuidad en la derivada que acabamos de describir

Capítulo 8

Diferenciación e Integración numérica

La diferenciación, y sobre todo la integración son operaciones habituales en cálculo numérico. En muchos casos obtener la expresión analítica de la derivada o la integral de una función puede ser muy complicado o incluso imposible. Además en ocasiones no disponemos de una expresión analítica para la función que necesitamos integrar o derivar, sino tan solo de un conjunto de valores numéricos de la misma. Este es el caso, por ejemplo, cuando estamos trabajando con datos experimentales. Si solo disponemos de valores numéricos, entonces solo podemos calcular la integral o la derivada numéricamente.

Los sistemas físicos se describen generalmente mediante ecuaciones diferenciales. La mayoría de las ecuaciones diferenciales no poseen una solución analítica, siendo posible únicamente obtener soluciones numéricas.

En términos generales la diferenciación numérica consiste en aproximar el valor que toma la derivada de una función en un punto. De modo análogo, la integración numérica aproxima el valor que toma la integral de una función en un intervalo.

8.1. Diferenciación numérica.

Como punto de partida, supondremos que tenemos un conjunto de puntos $\{x_i, y_i\}$,

x	x_0	x_1	\cdots	x_n
y	y_0	y_1	\cdots	y_n

Que pertenecen a una función $y = f(x)$ que podemos o no conocer analíticamente. El objetivo de la diferenciación numérica es estimar el valor de la derivada $f'(x)$ de la función, en alguno de los puntos x_i en los que el valor de $f(x)$ es conocido.

En general existen dos formas de aproximar la derivada:

1. Derivando el polinomio de interpolación. De este modo, obtenemos un nuevo polinomio que approxima la derivada.

$$f(x) \approx P_n(x) \Rightarrow f'(x) \approx P'_n(x)$$

2. Estimando la derivada como una fórmula de diferencias finitas obtenida a partir de la aproximación del polinomio de Taylor.

Si partimos de la definición de derivada,

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h} \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

Podemos asociar esta aproximación con el polinomio de Taylor de primer orden de la función $f(x)$,

$$f(x) \approx f(x_0) + f'(x_0) \cdot (x - x_0) \Rightarrow f'(x_0) \approx \frac{f(x) - f(x_0)}{x - x_0}$$

Si hacemos $x - x_0 = h$, ambas expresiones coinciden.

En general, los algoritmos de diferenciación numérica son inestables. Los errores iniciales que puedan contener los datos debido a factores experimentales o al redondeo del ordenador, aumentan en el proceso de diferenciación. Por eso no se pueden calcular derivadas de orden alto y, los resultados obtenidos de la diferenciación numérica deben tomarse siempre extremando la precaución.

8.1.1. Diferenciación numérica basada en el polinomio de interpolación.

El método consiste en derivar el polinomio $P_n(x)$ de interpolación obtenido por alguno de los métodos estudiados en el capítulo 7 y evaluar el polinomio derivada $P'_n(x)$ en el punto deseado.

Un ejemplo particularmente sencillo, para la expresión del polinomio derivada se obtiene en el caso de datos equidistantes interpolados mediante el polinomio de Newton-Gregory,

$$p_n(x) = y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_1) \cdot (x - x_0)}{2 \cdot h^2} \Delta^2 y_0 + \cdots + \frac{(x - x_{n-1}) \cdots (x - x_1) \cdot (x - x_0)}{n! \cdot h^n} \Delta^n y_0$$

Si lo derivamos, obtenemos un nuevo polinomio,

$$\begin{aligned} p'_n(x) &= \frac{\Delta y_0}{h} + \frac{\Delta^2 y_0}{2 \cdot h^2} [(x - x_1) + (x - x_0)] + \\ &+ \frac{\Delta^3 y_0}{3! \cdot h^3} [(x - x_1)(x - x_2) + (x - x_0)(x - x_1) + (x - x_0)(x - x_2)] + \cdots + \\ &+ \frac{\Delta^n y_0}{n! \cdot h^n} \sum_{k=0}^{n-1} \frac{(x - x_0)(x - x_1) \cdots (x - x_{n-1})}{x - x_k} \end{aligned}$$

Este polinomio es especialmente simple de evaluar en el punto x_0 ,

$$\begin{aligned} p'_n(x_0) &= \frac{\Delta y_0}{h} + \frac{\Delta^2 y_0}{2 \cdot h^2} \overbrace{(x_0 - x_1)}^{-h} + \cdots + \frac{\Delta^n y_0}{n! \cdot h^n} [\overbrace{(x_0 - x_1)}^{-h} \overbrace{(x_0 - x_2)}^{-2h} \cdots \overbrace{(x_0 - x_{n-1})}^{-(n-1)h}] \\ p'_n(x_0) &= \frac{1}{h} \left(\Delta y_0 - \frac{\Delta^2 y_0}{2} + \frac{\Delta^3 y_0}{3} + \cdots + \frac{\Delta^n y_0}{n} (-1)^{n-1} \right) \end{aligned}$$

Es interesante remarcar como en la expresión anterior, el valor de la derivada se va haciendo más preciso a medida que vamos añadiendo diferencias de orden superior. Si solo conocíramos dos datos, (x_0, y_0) y (x_1, y_1) . Solo podríamos calcular la diferencia dividida de primer orden. En esta caso nuestro cálculo aproximado de la derivada de x_0 sería,

$$p'_1(x_0) = \frac{1}{h} \Delta y_0$$

Si conocemos tres datos, podríamos calcular $\Delta^2 y_0$ y añadir un segundo término a nuestra estimación de la derivada,

$$p'_2(x_0) = \frac{1}{h} \left(\Delta y_0 - \frac{\Delta^2 y_0}{2} \right)$$

y así sucesivamente, mejorando cada vez más la precisión.

Veamos como ejemplo el cálculo la derivada en el punto $x_0 = 0,0$, a partir de la siguiente tabla de datos,

x_i	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$	$\Delta^4 y_i$
0,0	0,000	0,203	0,017	0,024	0,020
0,2	0,203	0,220	0,041	0,044	
0,4	0,423	0,261	0,085		
0,6	0,684	0,346			
0,8	1,030				

- Empleando los dos primeros puntos,

$$y'(0,0) = p_1^1(0,0) = \frac{1}{0,2} \cdot 0,203 = 1,015$$

- Empleando los tres primeros puntos,

$$y'(0,0) = p_2^1(0,0) = \frac{1}{0,2} \left(0,203 - \frac{0,017}{2} \right) = 0,9725$$

- Empleando los cuatro primeros puntos,

$$y'(0,0) = p_3^1(0,0) = \frac{1}{0,2} \left(0,203 - \frac{0,017}{2} + \frac{0,024}{3} \right) = 1,0125$$

- Empleando los cinco puntos disponibles

$$y'(0,0) = p_4^1(0,0) = \frac{1}{0,2} \left(0,203 - \frac{0,017}{2} + \frac{0,024}{3} - \frac{0,020}{4} \right) = 0,9875$$

8.1.2. Diferenciación numérica basada en diferencias finitas

Como se explicó en la introducción, la idea es emplear el desarrollo de Taylor para aproximar la derivada de una función en punto. Si empezamos con el ejemplo más sencillo, podemos aproximar la derivada suprimiendo de su definición el *paso al límite*,

$$f'(x_k) = \lim_{h \rightarrow 0} \frac{f(x_k + h) - f(x_k)}{h} \approx \frac{f(x_k + h) - f(x_k)}{h}$$

La expresión obtenida, se conoce con el nombre de formula de diferenciación adelantada de dos puntos. El error cometido debido a la elección de un valor de h finito, se conoce con el nombre de error de truncamiento. Es evidente que desde un punto de vista puramente matemático, la aproximación es mejor cuanto menor es h . Sin embargo, desde un punto de vista numérico esto no

es así. A medida que hacemos más pequeño el valor de h , aumenta el error de redondeo debido a la aritmética finita del computador. Por tanto, el error cometido es la suma de ambos errores,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \underbrace{C \cdot h}_{\text{e. de truncamiento}} + \underbrace{D \cdot \frac{1}{h}}_{\text{e. de redondeo}}, \quad C \gg D$$

El valor óptimo de h es aquel que hace mínima la suma del error de redondeo y el error de truncamiento. La figura 8.1 muestra de modo esquemático como domina un error u otro según hacemos crecer o decrecer el valor de h en torno a su valor óptimo.

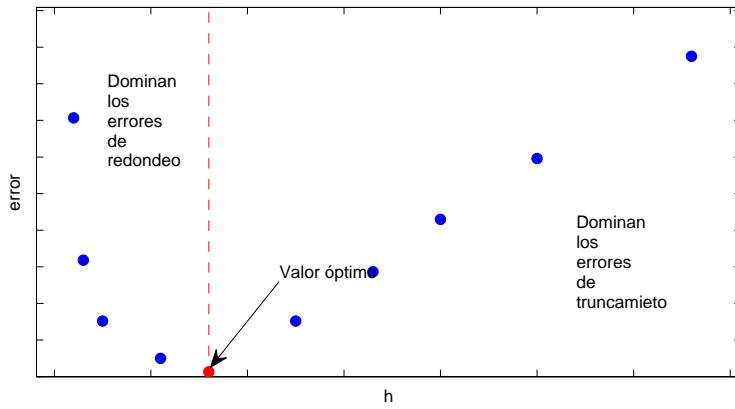


Figura 8.1: Variación del error cometido al aproximar la derivada de una función empleando una fórmula de diferenciación de dos puntos.

Como vimos en la introducción a esta sección, partiendo de el desarrollo de Taylor de una función es posible obtener fórmulas de diferenciación numérica y poder estimar el error cometido. Así por ejemplo, a partir del polinomio de Taylor de primer orden,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(z) \Rightarrow f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(z), \quad x < z < x+h$$

El error que se comete debido a la aproximación, es proporcional al tamaño del intervalo empleado h . La constante de proporcionalidad depende de la derivada segunda de la función, $f''(z)$ en algún punto indeterminado ($z \in x, x+h$). Para indicar esta relación lineal entre el error cometido y el valor de h , se dice que el error es del *orden* de h y se representa como $O(h)$.

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

Podemos mejorar la aproximación, calculando el valor de polinomio de Taylor de tercer orden para dos puntos equidistantes situados a la izquierda y la derecha del punto x , restando dichas expresiones y despejando la derivada primera del resultado,

$$\left. \begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(z) \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f'''(z) \end{aligned} \right\} \Rightarrow f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(z)$$

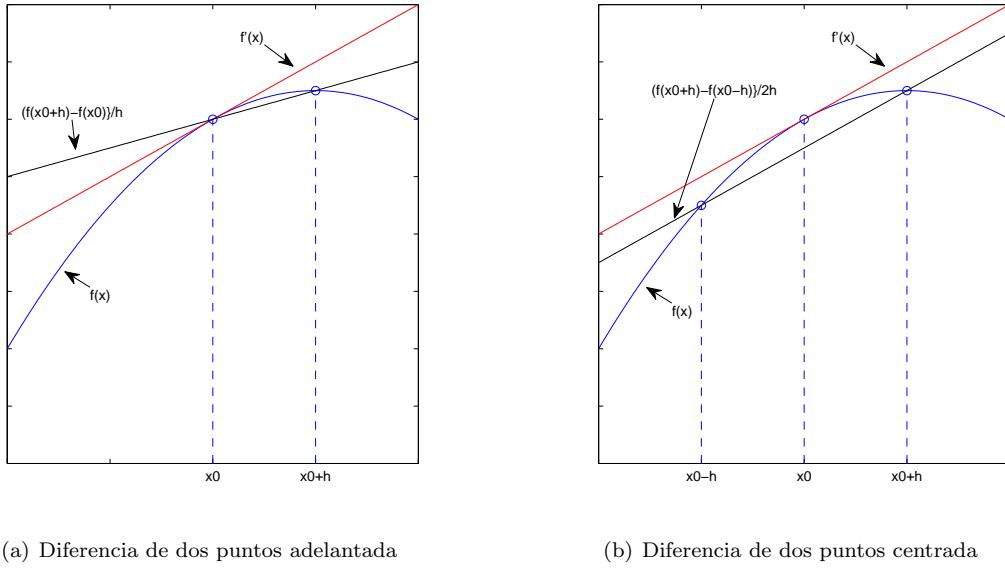


Figura 8.2: Comparación entre las aproximaciones a la derivada de una función obtenidas mediante las diferencias de dos puntos adelantada y centrada

En este caso, el error es proporcional al cuadrado de h , por tanto,

$$f'(x_0) = \frac{f(x_1) - f(x_{-1})}{2h} + O(h^2)$$

Donde hemos hecho $x \equiv x_0$, $x + h \equiv x_1$ y $x - h \equiv x_{-1}$. Esta aproximación recibe el nombre de diferencia de dos puntos centrada. La figura 8.2(a) muestra una comparación entre la derivada real de una función y su aproximación mediante una diferencia adelantada de dos puntos. La figura 8.2(b) muestra la misma comparación empleando esta vez la aproximación de dos punto centrada. En este ejemplo es fácil ver como la aproximación centrada da un mejor resultado. No hay que olvidar que la bondad del resultado, para un valor de h dado, depende también del valor de las derivadas de orden superior de la función, por lo que no es posible asegurar que el resultado de la diferencia centrada sea siempre mejor.

Empleando el desarrollo de Taylor y tres puntos podemos aproximar la derivada por la diferencia de tres puntos adelantada,

$$\left. \begin{aligned} & 4 \cdot \left(f(x_1) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{3!}f'''(z) \right) \\ & f(x_2) = f(x_0) + 2hf'(x_0) + 2h^2f''(x_0) + \frac{4h^3}{3}f'''(z) \end{aligned} \right\} \Rightarrow$$

$$\Rightarrow f'(x_0) = \frac{-f(x_2) + 4f(x_1) - 3f(x_0)}{2h} - \frac{h^2}{3}f'''(z)$$

En este caso el error es también de orden h^2 pero vale el doble que para la diferencia de dos puntos centrada.

A partir del desarrollo de Taylor y mediante el uso del número de puntos adecuado, es posible obtener aproximaciones a la derivada primera y a las sucesivas derivadas de una función, procediendo de modo análogo a como acaba de mostrarse para el caso de las diferencias de dos y tres puntos. La tabla 8.1 muestra algunas de las fórmulas de derivación mediante diferencias finitas más empleadas. Para simplificar la notación, en todos los casos se ha tomado $y_i = f(x_i)$, $y_i^{(j)} = f^{(j)}(x_i)$.

Fórmulas primera derivada	Fórmulas segunda derivada
$y'_0 = \frac{y_1 - y_0}{h} + O(h)$	$y''_0 = \frac{y_2 - 2y_1 + y_0}{h^2} + O(h)$
$y'_0 = \frac{y_1 - y_{-1}}{2h} + O(h^2)$	$y''_0 = \frac{y_1 - 2y_0 + y_{-1}}{h^2}$
$y'_0 = \frac{-y_2 + 4y_1 - 3y_0}{2h} + O(h^2)$	$y''_0 = \frac{-y_3 + y_2 - 5y_1 + 2y_0}{h^2} + O(h^2)$
$y'_0 = \frac{-y_2 + 8y_1 - 8y_{-1} + y_{-2}}{12h} + O(h^4)$	$y''_0 = \frac{-y_2 + 16y_1 - 30y_0 + 16y_{-1} - y_{-2}}{12h^2} + O(h^4)$
Fórmulas tercera derivada	Fórmulas cuarta derivada
$y'''_0 = \frac{y_3 - 3y_2 + 3y_1 - y_0}{h^3} + O(h)$	$y^{iv}_0 = \frac{y_4 - 4y_3 + 6y_2 - 4y_1 + y_0}{h^4} + O(h)$
$y'''_0 = \frac{y_2 - 2y_1 + 2y_{-1} - y_{-2}}{2h^3} + O(h^2)$	$y^{iv}_0 = \frac{y_2 - 4y_1 + 6y_0 - 4y_{-1} + y_{-2}}{h^4} + O(h^2)$

Cuadro 8.1: Fórmulas de derivación basadas en diferencias finitas

8.2. Integración numérica.

Dada una función arbitraria $f(x)$ es en muchos casos posible obtener de modo analítico su primitiva $F(x)$ de modo que $f(x) = F'(x)$. En estos casos, la integral definida de $f(x)$ en un intervalo $[a, b]$ puede obtenerse directamente a partir de su primitiva,

$$I(f) = \int_a^b f(x)dx = F(x)|_a^b = F(b) - F(a)$$

Hay sin embargo muchos casos en los cuales se desconoce la función $F(x)$ y otros en los que ni siquiera se conoce la expresión de la función $f(x)$, como por ejemplo, cuando solo se dispone de una tabla de valores $\{x_i, y_i = f(x_i)\}$ para representar la función. En estos casos se puede aproximar la integral definida de la función $f(x)$ en un intervalo $[a, b]$, a partir de los puntos disponibles, mediante lo que se conoce con el nombre de una fórmula de cuadratura,

$$I(f) = \int_a^b f(x)dx \approx \sum_{i=0}^n A_i f(x_i)$$

Una técnica habitual de obtener los coeficientes A_i , es hacerlo de modo implícito a partir de la integración de los polinomios de interpolación,

$$I(f) = \int_a^b f(x)dx \approx \int_a^b P_n(x)dx$$

Para ello, se identifican los extremos del intervalo de integración con el primer y el último de los datos disponibles, $[a, b] \equiv [x_0, x_n]$.

Así por ejemplo, a partir de los polinomios de Lagrange, definidos en la sección 7.2.2,

$$p(x) = \sum_{j=0}^n l_j(x) \cdot y_j$$

Podemos obtener los coeficientes A_i como,

$$I(f) = \int_a^b f(x) dx \approx \int_a^b P_n(x) dx = \int_{x_0}^{x_n} \left(\sum_{j=0}^n l_j(x) \cdot y_j \right) dx \Rightarrow A_j = \int_{x_0}^{x_n} l_j(x) dx$$

La familia de métodos de integración, conocidas como fórmulas de Newton-Cotes, puede obtenerse a partir del polinomio de interpolación de Newton-Gregory descrito en la sección 7.3.1. Supongamos que tenemos la función a integrar definida a partir de un conjunto de puntos equiespaciados a lo largo del intervalo de integración $\{(x_i, y_i)\}_{0, \dots, n}$. Podemos aproximar la integral $I(y)$ como,

$$\int_{x_0}^{x_n} y dx \approx \int_{x_0}^{x_n} \left(y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_0)(x - x_1)}{2!h^2} \Delta^2 y_0 + \dots + \frac{(x - x_0) \cdots (x - x_{n-1})}{n!h^n} \Delta^n y_0 \right)$$

Las fórmulas de Newton-Cotes se asocian con el grado del polinomio de interpolación empleado en su obtención:

- Para $n = 1$, se obtiene la regla del trapezio,

$$I(y) = \int_{x_0}^{x_1} y dx \approx \frac{h}{2}(y_0 + y_1)$$

- Para $n = 2$, se obtiene la regla de Simpson

$$I(y) = \int_{x_0}^{x_2} y dx \approx \frac{h}{3}(y_0 + 4y_1 + y_2)$$

- Para $n = 3$, se obtiene la regla de 3/8 de Simpson

$$I(y) = \int_{x_0}^{x_3} y dx \approx \frac{3h}{8}(y_0 + 3y_1 + 3y_2 + y_3)$$

No se suelen emplear polinomios de interpolación de mayor grado debido a los errores de redondeo y a las oscilaciones locales que dichos polinomios presentan.

8.2.1. La fórmula del trapecio.

La fórmula del trapecio emplea tan solo dos puntos para obtener la integral de la función en el intervalo definido por ellos.

$$I(y) = \int_{x_0}^{x_1} y dx \approx \int_{x_0}^{x_1} \left(y_0 + \frac{x - x_0}{h} \Delta y_0 \right) dx = y_0 x + \frac{\Delta y_0 (x - x_0)^2}{2} \Big|_{x_0}^{x_1} = \frac{h}{2}(y_0 + y_1)$$

La figura 8.3 muestra gráficamente el resultado de aproximar la integral definida de una función $y = f(x)$ mediante la fórmula del trapecio. Gráficamente la integral coincide con el área del *trapézio* formado por los puntos $(x_0, 0)$, (x_0, y_0) , (x_1, y_1) y $(x_1, 0)$. De ahí su nombre y la expresión matemática obtenida,

$$I(y) = \frac{h}{2}(y_0 + y_1)$$

que coincide con el área del trapecio mostrado en la figura.

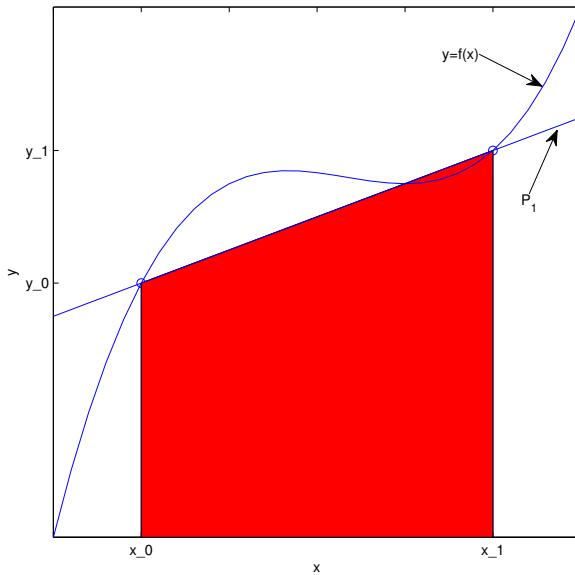


Figura 8.3: Interpretación gráfica de la fórmula del trapecio.

Formula extendida (o compuesta) del trapecio. La figura 8.3 permite observar la diferencia entre el área calculada y el área comprendida entre la curva real y el eje x . Como se ha aproximado la curva en el intervalo de integración por un líneal recta (polinomio de grado 1 p_1), El error será tanto mayor cuando mayor sea el intervalo de integración y/o la variación de la función en dicho intervalo. Una solución a este problema, si se conoce la expresión analítica de la función que se desea integrar o se conocen suficientes puntos es subdividir el intervalo de integración en intervalos más pequeños y aplicar a cada uno de ellos la fórmula de trapecio,

$$I(y) = \int_{x_0}^{x_n} y dx \approx \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} y(x) dx = \sum_{i=0}^{n-1} \frac{h}{2}(y_i + y_{i+1}) = \frac{h}{2} (y_0 + 2y_1 + 2y_2 + \dots + 2y_{n-1} + y_n)$$

La figura 8.4, muestra el resultado de aplicar la fórmula extendida del trapecio a la misma función de la figura 8.3. En este caso, se ha dividido el intervalo de integración en cuatro subintervalos. Es inmediato observar a partir de la figura, que la aproximación mejorar progresivamente si se aumenta el número de subintervalos y se reduce el tamaño de los mismos.

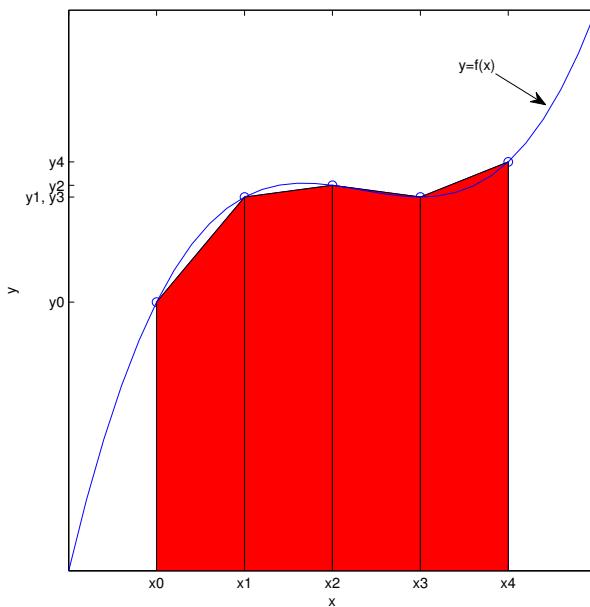


Figura 8.4: Interpretación gráfica de la fórmula extendida del trapecio.

8.3. Las fórmulas de Simpson.

Se conocen con el nombre de fórmulas integrales de Simpson, a las aproximaciones a la integral definida obtenida a partir de los polinomios interpoladores de Newton-Gregory de grado dos (Simpson 1/3) y de grado tres (Simpson 3/8) .

En el primer caso, es preciso conocer tres valores equiespaciados de la función en el intervalo de integración y en el segundo es preciso conocer cuatro puntos.

Fórmula de Simpson 1/3. La fórmula de Simpson, o Simpson 1/3, emplea un polinomio de interpolación de Newton-Gregory de grado dos para obtener la aproximación a la integral,

$$I(y) \approx \int_{x_0}^{x_2} P_2(x) dx = \int_{x_0}^{x_2} \left(y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_0) \cdot (x - x_1)}{2h^2} \Delta^2 y_0 \right) dx = \frac{h}{3} (y_0 + 4y_1 + y_2)$$

La figura 8.5, muestra gráficamente el resultado de aplicar el método de Simpson a la misma función de los ejemplos anteriores. De nuevo, la bondad de la aproximación depende de lo que varíe la función en el intervalo. La diferencia fundamental con el método del trapecio es que ahora el área calculada esta limitada por el segmento de parábola definido por el polinomio de interpolación empleado.

Fórmula de Simpson 3/8. En este caso, se emplea un polinomio de Newton-Gregory de grado 3 para obtener la aproximación a la integral,

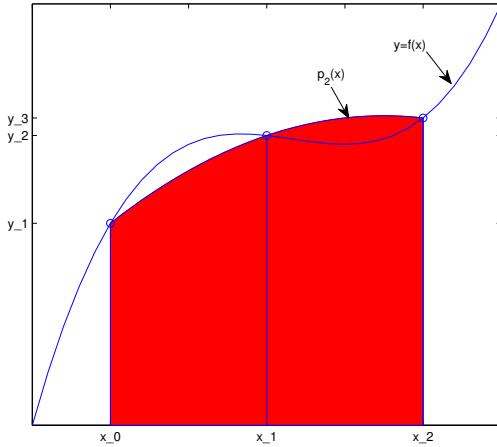


Figura 8.5: Interpretación gráfica de la fórmula 1/3 de Simpson.

$$\begin{aligned}
 I(y) &\approx \int_{x_0}^{x_3} P_2(x) dx = \\
 &= \int_{x_0}^{x_3} \left(y_0 + \frac{x - x_0}{h} \Delta y_0 + \frac{(x - x_0) \cdot (x - x_1)}{2h^2} \Delta^2 y_0 + \frac{(x - x_0) \cdot (x - x_1) \cdot (x - x_2)}{3!h^3} \Delta^3 y_0 \right) dx \\
 &= \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + y_3)
 \end{aligned}$$

La figura 8.6 muestra el resultado de aplicar la fórmula de Simpson 3/8 a la misma función de los ejemplos anteriores. En este caso, la integral sería exacta porque la función de ejemplo elegida es un polinomio de tercer grado y coincide exactamente con el polinomio de interpolación construido para obtener la integral.

Al igual que en el caso del método del trapecio, lo normal no es aplicar los métodos de Simpson a todo el intervalo de integración, sino dividirlo en subintervalos más pequeños y aplicar el método sobre dichos subintervalos. El resultado se conoce como métodos extendidos de Simpson. Al igual que sucede con la fórmula del trapecio, los métodos extendidos de Simpson mejoran la aproximación obtenida para la integral tanto más cuanto más pequeño es el tamaño de los subintervalos empleados.

Así, la fórmula extendida de Simpson 1/3 toma la forma,

$$\begin{aligned}
 I(y) &\approx \sum_{i=0}^{n-2} \int_{x_i}^{x_{i+2}} P_2(x) dx = \sum_{i=0}^{\frac{n}{2}-2} \frac{h}{3} (y_{2i} + 4y_{2i+1} + y_{2i+2}) \\
 &= \frac{h}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)
 \end{aligned}$$

Donde se ha dividido el intervalo de integración en n subintervalos, la fórmula de Simpson se ha calculado para cada dos subintervalos y se han sumado los resultados.

Por último para la fórmula extendida de Simpson 3/8 se puede emplear la expresión,

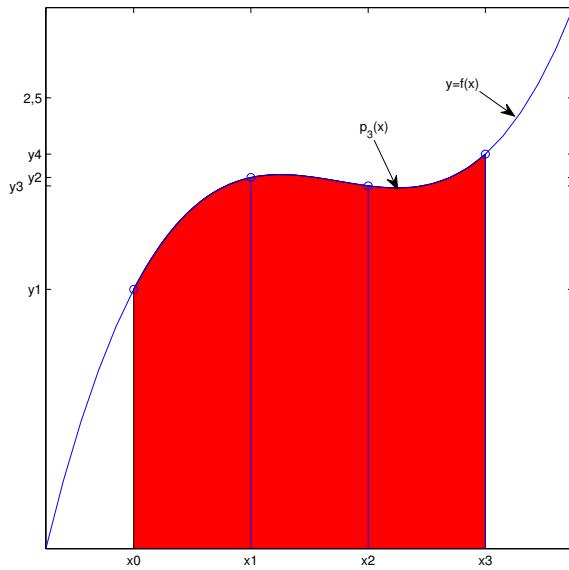


Figura 8.6: Interpretación gráfica de la fórmula 3/8 de Simpson.

$$\begin{aligned}
 I(y) &\approx \sum_{i=0}^{n-3} \int_{x_i}^{x_{i+3}} P_3(x) dx = \sum_{i=0}^{\frac{n}{2}-3} \frac{3h}{8} (y_{3i} + 3y_{3i+1} + 3y_{3i+2} + y_{3i+3}) \\
 &= \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + 2y_3 + 3y_4 + 3y_5 + 2y_6 + \dots + 2y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n)
 \end{aligned}$$

En este caso también se divide el intervalo en n subintervalos pero ahora se ha aplicado la regla de Simpson 3/8 a cada tres subintervalos.

A continuación se incluye un código que permite aproximar la integral definida de una función en un intervalo por cualquiera de los tres métodos descritos: Trapecio, Simpson o Simpson 3/8,

```

function int=integra(fun,met,inter,dib)
%implementa los métodos del trapecio, etc...
%Uso: int=integra(fun,inter,dib)
%fun, nombre de la función que se desea intergrar, debe ir entre comillas
%met, método que se desea emplear para calcular la integral. los nombres
%validos para los métodos son: 'trapecio', 'simpson' y 'simpson38'
%inter, es un vector de dos elementos que contiene lo extremos de intervalo
%de integracion [a,b]
%dib, si se da un valor a esta variable de entrada, dibuja la función en el
%intervalo de integración y la integral obtenida. Si se omite no dibuja
%nada.
%int, valor de la integral obtenida.
%Este programa no divide el intervalo de integración en subintervalos...

```

```
%para calcular la integral empleando subintervalos usar la función trocea,
```

```
if strcmp(met,'trapecio')
    f0=feval(fun,inter(1));
    f1=feval(fun,inter(2));
    h=inter(2)-inter(1);
    int=h*(f0+f1)/2;
    if nargin==4
        x=inter(1):(inter(2)-inter(1))/100:inter(2);
        yfuncion=feval(fun,x);
        ypolinomio=f0+(x-inter(1))*(f1-f0)/h;
        plot(x,yfuncion,x,ypolinomio,'r')
        hold on
        fill([x(1) x x(length(x))],[0 ypolinomio 0],'r')
    end
elseif strcmp(met,'simpson')
    pmedio=(inter(2)+inter(1))/2;
    f0=feval(fun,inter(1));
    f1=feval(fun,pmedio);
    f2=feval(fun,inter(2));
    h=(inter(2)-inter(1))./2;
    int=h*(f0+4*f1+f2)/3;
    if nargin==4
        x=inter(1):(inter(2)-inter(1))/100:inter(2);
        yfuncion=feval(fun,x);
        ypolinomio=f0+(x-inter(1)).*(f1-f0)/h+(x-inter(1)).*(x-pmedio).*(f2...
        -2*f1+f0)./(2*h^2);
        plot(x,yfuncion,x,ypolinomio,'r')
        hold on
        fill([x(1) x x(length(x))],[0 ypolinomio 0],'r')
    end
elseif strcmp(met,'simpson38')
    inter=inter(1):(inter(2)-inter(1))/3:inter(2);

    f=feval(fun,inter);
    h=(inter(4)-inter(1))/3;
    int=3*h*(f(1)+3*f(2)+3*f(3)+f(4))/8;
    if nargin==4
        x=inter(1):(inter(4)-inter(1))/100:inter(4);
        yfuncion=feval(fun,x);
        ypolinomio=f(1)+(x-inter(1)).*(f(2)-f(1))/h+(x-inter(1)).*(x...
        -inter(2)).*(f(3)-2*f(2)+f(1))./(2*h^2)+(x-inter(1)).*(x...
        -inter(2)).*(x-inter(3)).*(f(4)-3*f(3)+3*f(2)-f(1))./(6*h^3);
        plot(x,yfuncion,x,ypolinomio,'r')
        hold on
        fill([x(1) x x(length(x))],[0 ypolinomio 0],'r')
    end
else
    int='metodo desconocido, los metodos conocidos son ''trapecio'', ...
    ''simpson'' y ''simpson38'''
```

```
end
```

Este programa aplica directamente el método deseado sobre el intervalo de integración. Para obtener los métodos extendidos, podemos emplear un segundo programa que divida el intervalo de integración inicial en el número de subintervalos que deseemos, aplique el programa anterior a cada subintervalo, y, por último, sume todo los resultados para obtener el valor de la integral en el intervalo deseado. El siguiente programa, muestra un ejemplo de como hacerlo,

```
function total=trocea(fun,met,inter,div,dib)
%esta funcion lo unico que hace es trocear un intervalo en el numero de
%tramos indicados por div, y llamar a la funcion integra para que integre
%en cada intervalo.
%USO: total=trocea(fun,met,inter,div,dib)
% las variables de entrada son la mismas que las de integra, salvo div que
% representa el numero de subintervalos en que se desea dividir el
% intervalo de integración.
tramos=inter(1):(inter(2)-inter(1))/div:inter(2);
total=0;
if nargin==5
    hold on
end
for i=1:size(tramos,2)-1
    int=integra(fun,met,[tramos(i) tramos(i+1)],dib);
    total=total+int;
end
hold off
```

t

Por último indicar que Matlab posee un comando propio para calcular la integral definida de una función, el comando `quad`. Este comando admite como variables de entrada el nombre de una función, y dos valores que representan los límites de integración. Como variable de salida devuelve el valor de la integral definida en el intervalo introducido. Por ejemplo,

```
>> i=quad('sin',0,pi)

i =

2.0000
```

8.4. Problemas de valor inicial en ecuaciones diferenciales

Las leyes de la física están escritas en forma de ecuaciones diferenciales.

Una ecuación diferencial establece una relación matemática entre una variable y sus derivadas respecto a otra u otras variables de las que depende. El ejemplo más sencillo lo encontramos en las ecuaciones de la dinámica en una sola dimensión que relacionan la derivada segunda de la posición de un cuerpo con respecto al tiempo, con la fuerza que actúa sobre el mismo.

$$m \cdot \frac{d^2x}{dt^2} = F$$

Si la fuerza es constante, o conocemos explícitamente como varía con el tiempo, podemos integrar la ecuación anterior para obtener la derivada primera de la posición con respecto al tiempo —la velocidad— de una forma directa,

$$m \cdot \frac{d^2x}{dt^2} = F \rightarrow v(t) = \frac{dx}{dt} = \int \frac{F(t)}{m} dt + v(0)$$

Donde suponemos conocido el valor $v(0)$ de la velocidad del cuerpo en el instante inicial.

Si volvemos a integrar ahora la expresión obtenida para la velocidad, obtendríamos la posición en función del tiempo,

$$v(t) = \frac{dx}{dt} = \int \frac{F(t)}{m} dt + v(0) \rightarrow x(tt) = \int \left(\int \frac{F(t)}{m} dt + v(0) \right) dt + x(0)$$

Donde suponemos conocida la posición inicial $x(0)$.

Quizá el sistema físico idealizado más conocido y estudiado es el oscilador armónico. En este caso, el sistema está sometido a una fuerza que depende de la posición y, si existe disipación, a una fuerza que depende de la velocidad,

$$m \frac{d^2x}{dt^2} = -kx - \mu \frac{dx}{dt} + F(t)$$

En este caso, la expresión obtenida constituye una ecuación diferencial ordinaria y ya no es tan sencillo obtener una expresión analítica para $x(t)$. Para obtener dicha expresión analítica, es preciso emplear métodos de resolución de ecuaciones diferenciales.

El problema del oscilador armónico, pertenece a una familia de problemas conocida como problemas de valores iniciales. En general, un problema de valores iniciales de primer orden consiste en obtener la función $x(t)$, que satisface la ecuación,

$$x'(t) \equiv \frac{dx}{dt} = f(x(t), t), \quad x(t_0)$$

Donde $x(t_0)$ representa un valor inicial conocido de la función $x(t)$.

En muchos casos, las ecuaciones diferenciales que describen los fenómenos físicos no admiten una solución analítica, es decir no permiten obtener una función para $x(t)$. En estos caso, es posible obtener soluciones numéricas empleando un computador. El problema de valores iniciales se reduce entonces a encontrar un aproximación discretizada de la función $x(t)$.

El desarrollo de técnicas de integración numérica de ecuaciones diferenciales constituye uno de los campos de trabajo más importantes de los métodos de computación científica. Aquí nos limitaremos a ver los más sencillos.

Esencialmente, los métodos que vamos a describir se basan en discretizar el dominio donde se quiere conocer el valor de la función $x(t)$. Así por ejemplo si se quiere conocer el valor que toma la función en el intervalo $t \in [a, b]$ Se divide el intervalo en n subintervalos cada uno de tamaño h_i . Los métodos que vamos a estudiar nos proporcionan una aproximación de la función $x(t)$, x_0, x_1, \dots, x_n en los $n + 1$ puntos t_0, t_1, \dots, t_n , donde $t_0 = a$, $t_n = b$, y $t_{i+1} - t_i = h_i$. El valor de h_i recibe el nombre de paso de integración. Además se supone conocido el valor que toma la función $x(t)$ en el extremo inicial a , $x(a) = x_a$.

8.4.1. El método de Euler.

El método de Euler, puede obtenerse a partir del desarrollo de Taylor de la función x , entorno al valor conocido (a, x_a) . La idea es empezar en el valor conocido e ir obteniendo iterativamente el resto de los valores x_1, \dots hasta llegar al extremos b del intervalo en que queremos conocer el

valor de la función x . En general podemos expresar la relación entre dos valores sucesivos a partir del desarrollo de Taylor como,

$$x(t_{i+1}) = x(t_i) + (t_{i+1} - t_i)x'(t_i) + \frac{(t_{i+1} - t_i)^2}{2}x''(t_i) + \cdots + \frac{(t_{i+1} - t_i)^n}{n!}x^{(n)}(t_i) + \cdots$$

Como se trata de un problema de condiciones iniciales, conocemos la derivada primera de la función $x(t)$, explícitamente, $x'(t) = f(x(t), t)$. Por tanto podemos sustituir las derivadas de x por la función f y sus derivadas,

$$x(t_{i+1}) = x(t_i) + (t_{i+1} - t_i)f(t_i, x_i) + \frac{(t_{i+1} - t_i)^2}{2}f'(t_i, x_i) + \cdots + \frac{(t_{i+1} - t_i)^n}{n!}f^{(n-1)}(t_i, x_i) + \cdots$$

Donde $x_i \equiv x(t_i)$. Si truncamos el polinomio de Taylor, quedándonos solo con el término de primer grado, y hacemos que el paso de integración sea fijo, $h_i \equiv h = \text{cte}$ obtenemos el método de Euler,

$$x_{i+1} = x_i + h \cdot f(t_i, x_i)$$

A partir de un valor inicial, x_0 es posible obtener valores sucesivos mediante un algoritmo iterativo simple,

$$\begin{aligned} x_0 &= x(a) \\ x_1 &= x_0 + h f(a, x_0) \\ x_2 &= x_1 + h f(a + h, x_1) \\ &\vdots \\ x_{i+1} &= x_i + h f(a + ih, x_i) \end{aligned}$$

El siguiente código implementa el método de Euler para resolver un problema de condiciones iniciales de primer orden a partir de una función $f(t)$ y un valor inicial x_0 .

```
%inicial o valor inicial de la variable independiente (t).
%b, instante de tiempo final o valor final de la variable independiente
%h paso de integración
%podemos por elemental prudencia, comprobar que b>a
%lo elegante no es pasar el paso como he hecho aqui, sino calcularlo en
%funcion de cuantos puntos de la solucion queremos en el intervalo...
if a>=b
    y='el intervalo debe ser creciente, para evitarse lios';
    return
end
y=ya;
paso=1;
t(paso)=a;
while t(paso)<=b
    t(paso+1)=t(paso)+h;
    y(paso+1)=y(paso)+h*feval(fun,t(paso),y(paso));
    paso=paso+1;
end
%dibuja la solucion,
plot(t,y)
```

Un ejemplo sencillo de problema de condiciones iniciales de primer orden, nos los suministra la ecuación diferencial de la carga y descarga de un condensador eléctrico. La figura 8.7, muestra un circuito eléctrico elemental formado por una resistencia R en serie con un condensador C .

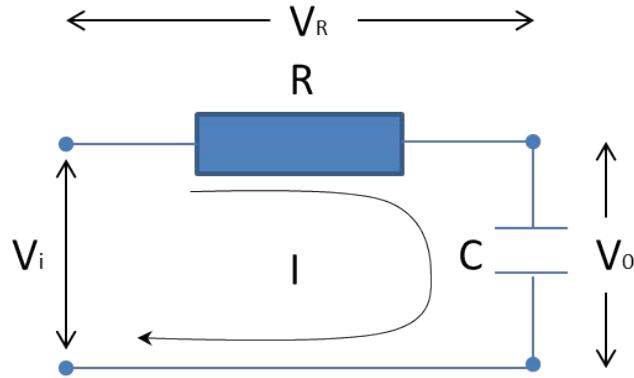


Figura 8.7: Circuito RC

La intensidad eléctrica que atraviesa un condensador depende de su capacidad, y de la variación con respecto al tiempo del voltaje entre sus extremos,

$$I = c \frac{dV_o}{dt}$$

La intensidad que atraviesa la resistencia se puede obtener a partir de la ley de Ohm,

$$V_R = I \cdot R$$

Por otro lado, la intensidad que recorre el circuito es común para la resistencia y el condensador. El voltaje suministrado tiene que ser la suma de las caídas de voltaje en la resistencia y en el condensador, $V_i = V_o + V_R$. Sustituyendo y despejando,

$$V_i = V_o + V_R \rightarrow V_i = V_o + I \cdot R \rightarrow V_i = V_o + R \cdot C \frac{dV_o}{dt}$$

Si reordenamos el resultado,

$$\frac{dV_o}{dt} = \frac{V_i - V_o}{R \cdot C}$$

Obtenemos una ecuación diferencial para el valor del voltaje en los extremos del condensador que puede tratarse como un problema de valor inicial. Para este problema, la función $f(t, x)$ toma la forma,

$$f(t, x) \equiv \frac{V_i - V_o}{R \cdot C}$$

además necesitamos conocer un valor inicial $V_o(0)$ para el voltaje en el condensador. Si suponemos que el condensador se encuentra inicialmente descargado, entonces $V_o(0) = 0$. Para este problema se conoce la solución analítica. El voltaje del condensador en función del tiempo viene dado por la función,

$$V_o(t) = V_i \left(1 - e^{-t/R \cdot C} \right)$$

Podemos resolver el problema de la carga del condensador, empleando el código de la función de Euler incluido más arriba. Lo único que necesitamos es definir una función de Matlab para representar la función $f(x, t)$ de nuestro problema de valor inicial, t

```
function s=condensador(~,Vo)
%t tiempo en este ejemplo la función no depende del tiempo por lo que esta
%variable realmente no se usa... pero como el programa Euler se la pasa,
%empleamos el simbolo ~ para que no de errores...
%Vo Valor del voltaje en cada iteración

%hay que definir los parametros fijos del circuito...
C=0.0001; %Capacidad del condensador en Faradios
R=1000; %resistencia en Ohmios
%el voltaje de entrada podría Vi podria ser una función del tiempo... Aquí
%vamos a considerar que es constante...
Vi=10; %Voltaje suministrado al circuito en Voltios
%Aqui cosntruimos la funcion que representa el circuito RC,
s=(Vi-Vo)/R/C;
```

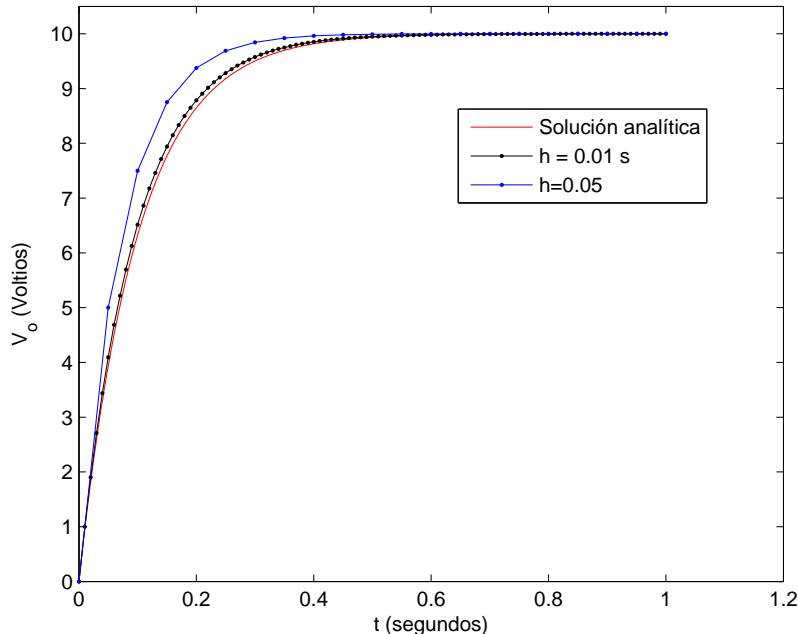


Figura 8.8: Comparacion entre los resultados obtenidos mediante el método de Euler para dos pasos de integración $h = 0,05$ y $h = 0,001$ y la solución analítica para el voltaje V_o de un condensador durante su carga.

La figura 8.8 compara gráficamente los resultados obtenido si empleamos la función descrita más arriba, `euler('condensador',0,0,1,h)`, para obtener el resultado de aplicar al circuito un

voltaje de entrada constante $V_i = 10V$ durante un segundo, empleando dos pasos de integración distintos $h = 0,05s$ y $h = 0,01s$. Además, se ha añadido a la gráfica el resultado analítico.

En primer lugar, es interesante observar como el voltaje V_0 crece hasta alcanzar el valor $V_i = 10V$ del voltaje suministrado al circuito. El tiempo que tarda el condensador en alcanzar dicho voltaje y quedar completamente cargado depende de su capacidad y de la resistencia presente en el circuito.

Como era de esperar, al hacer menor el paso de integración la solución numérica se aproxima más a la solución analítica. Sin embargo, como pasaba en el caso de los métodos de diferenciación de funciones, hay un valor de h óptimo. Si disminuimos el paso de integración por debajo de ese valor, los errores de redondeo empiezan a dominar haciendo que la solución empeore.

8.4.2. Métodos de Runge-Kutta

Si intentamos emplear polinomios de Taylor de grado superior, al empleado en el método de Euler, nos encontramos con la dificultad de obtener las derivadas sucesivas, con respecto al tiempo, de la función f . Así, si quisieramos emplear el polinomio de Taylor de segundo grado, para la función x , tendríamos,

$$x(t_{i+1}) = x(t_i) + h \cdot f(t_i, x_i) + \frac{h^2}{2} f'(t_i, y_i)$$

Pero,

$$f'(t, x) = \frac{\partial f(t, x)}{\partial t} + \frac{\partial f(t, x)}{\partial x} \cdot f(t, x)$$

Sacando factor común a h , obtenemos,

$$x(t_{i+1}) = x(t_i) + h \left(f(t_i, x_i) + \frac{h}{2} \left(\frac{\partial f(t_i, x_i)}{\partial t} + \frac{\partial f(t_i, x_i)}{\partial x} \cdot f(t_i, x_i) \right) \right)$$

A partir de este resultado, es fácil comprender que resulte complicado obtener métodos de resolución basados en el desarrollo de Taylor. De hecho, se vuelven cada vez más complicados según vamos empleando polinomios de Taylor de mayor grado.

Desde un punto de vista práctico, lo que se hace es buscar aproximaciones a los términos sucesivos del desarrollo de Taylor, para evitar tener que calcularlos explícitamente. Estas aproximaciones se basan a su vez, en el desarrollo de Taylor para funciones de dos variables.

Los métodos de integración resultantes, se conocen con el nombre genérico de métodos de Runge-Kutta. Veamos cómo se haría para el caso que acabamos de mostrar del polinomio de Taylor de segundo grado.

En primer lugar obtenemos el desarrollo del polinomio de Taylor de primer grado, en dos variables de la función $f(t, x)$, en un entorno del punto t_i, x_i ,

$$f(t, x) = f(t_i, x_i) + \left((t - t_i) \frac{\partial f(t_i, x_i)}{\partial t} + (x - x_i) \frac{\partial f(t_i, x_i)}{\partial x} \right)$$

Si ahora comparamos este resultado con la ecuación anterior, podríamos tratar de identificar entre sí los términos que acompañan las derivadas parciales,

$$\begin{aligned} t - t_i &= \frac{h}{2} \rightarrow t = t_i + \frac{h}{2} \\ x - x_i &= \frac{h}{2} \cdot f(t_i, x_i) \rightarrow x = x_i + \frac{h}{2} \cdot f(t_i, x_i) \end{aligned}$$

Es decir,

$$f(t_i, x_i) + \frac{h}{2} \left(\frac{\partial f(t_i, x_i)}{\partial t} + \frac{\partial f(t_i, x_i)}{\partial x} \cdot f(t_i, x_i) \right) = f(t_i + \frac{h}{2}, x_i + \frac{h}{2} \cdot f(t_i, x_i))$$

Si ahora sustituimos este resultado en nuestra expresión del polinomio de Taylor de segundo grado de la función $x(t)$,

$$x(t_{i+1}) = x(t_i) + h \cdot f(t_i + \frac{h}{2}, x_i + \frac{h}{2} f(t_i, x_i))$$

Donde $x_i \equiv x(t_i)$. Esta aproximación da lugar al primero y más sencillo de los métodos de Runge-Kutta, conocido como método del punto medio. El nombre es debido a que la función f se evalúa en un punto a mitad de camino entre t_i y $t_{i+1} = t_i + h$. El cálculo de la solución de un problema de valor inicial mediante este método, se puede expresar de un modo análogo al del método de Euler,

$$\begin{aligned} x_0 &= x(a) \\ x_1 &= x_0 + h \cdot f(a + \frac{h}{2}, x_0 + \frac{h}{2} f(a, x_0)) \\ &\vdots \\ x_{i+1} &= x_i + h \cdot f(t_i + \frac{h}{2}, x_i + \frac{h}{2} f(t_i, x_i)) \end{aligned}$$

La siguiente función de Matlab implementa el método del punto medio,

```
function y=pmedio(fun,ya,a,b,h)
%metodo de rungekutta de segundo orden mas conocido como del punto medio
%uso: y=pmedio(fun,ya,a,b,h)
% variables de entrada: fun, nombre entre comillas de la función que define
% el problema de valor inicial que se desea resolver. ya, valor inicial. a,
% instante de tiempo inicial o valor inicial de la variable independiente.
% b valor de tiempo final o valor final de la variable independiente. h,
% paso de integracion
%podemos por elemental prudencia, comprobar que b>a
%lo elegante no es pasar el paso como he hecho aqui, sino calcularlo en
%funcion de cuantos puntos de la solucion queremos en el intervalo...
if a>=b
    y='el intervalo debe ser creciente, para evitarse lios'
    return
end
y=ya;
paso=1;
t(paso)=a;
while t(paso)<=b
    t(paso+1)=t(paso)+h;
    y(paso+1)=y(paso)+h*feval(fun,t(paso)+h/2,y(paso)+h*feval(fun,t(paso),y(paso))/2);
    paso=paso+1;
end
%pintamos la solucion,
plot(t,y)
```

El resto de los métodos de Runge-Kutta, los dejaremos para cuando seáis más mayores... Pero que conste que es de lo más interesante de los métodos numéricos.

Capítulo 9

Tratamiento estadístico de datos

9.1. Secuencias de números aleatorios

Se entiende por secuencia de número aleatorios, aquella en la que no es posible encontrar relación alguna entre un número de la secuencia y los que le preceden.

No es posible general secuencias de números aleatorios con un computador. La razón es que un computador es una máquina completamente determinista; las salidas que puede producir cualquier programa son completamente predecibles. Solo los procesos físicos pueden ser realmente —intrínsecamente— aleatorios.

Sin embargo, es posible con un ordenador generar secuencias de números que simulan ser aleatorios. Estas secuencias reciben el nombre de secuencias de números *pseudoaleatorios*.

El primer algoritmo para obtener números pseudoaleatorios, lo desarrolló John Von Newman en 1946. Se conoce con el nombre de *Middle Square*. La idea consiste en elegir un número inicial, formado por n cifras, que recibe el nombre de semilla. A continuación se eleva el número al cuadrado. Por último, se extraen las n cifras centrales del número así generado, que constituye el segundo número en la secuencia de números aleatorios. Este número se vuelve a elevar al cuadrado y se extraen de él las n cifras centrales que constituirán el tercer número aleatorio. Este proceso se repite tantas veces como números aleatorios se deseen generar.

Veamos un ejemplo con una semilla formada por $n = 8$ dígitos,

$$\begin{aligned} \text{semilla} &= 87289689 \rightarrow 87289689^2 = 7619\ 48980571\ 6721 \\ &\quad 48980571 \rightarrow 48980571^2 = 2399\ 09633548\ 6041 \\ &\quad 09633548 \rightarrow 09633548^2 = 0092\ 80524706\ 8304 \\ &\quad 80524706 \rightarrow 80524706^2 = 6484\ 22827638\ 6436 \\ &\quad 22827638 \\ &\quad \vdots \end{aligned}$$

El siguiente código de Matlab, permite obtener un vector de números pseudoaleatorios mediante el método *middle square*

```
function secuencia=middlesquare(semilla, longitud)
%Este programa obtiene una secuencia de números psduoaleatorios empleado el
%método 'middle square'.
%uso: secuencia=middlesquare(semilla, longitud)
```

```
%Variables entrada: semilla, numero inicial que será empleado como semilla.
%longitud, cantidad de números aleatorios que debe generar la secuencia.
%variable de salida: secuencia vector de numeros aleatorios resultante

%Creamos un vector de ceros para guardar los resultados
% convertimos el número a entero de 64 bits, así todas las operaciones son
% enteras...
%ojo si usamos una semilla de mas de nueve dígitos, el cuadrado puede
%desbordar el entero más grande representable.

semilla=uint64(semilla)
%calculamos la longitud de la semilla
c=semilla;
n=0;
while c>0
    c=idivide(c,10);
    %la linea de código anterior, puede no funcionar en algunas versiones
    %antiguas de matlab. Alternativamente, podría implementarse como,
    %c=(c-rem(c,10))/10;
    n=n+1;
end

%Creamos un bucle para ir generando los números pseudoaleatorio a partir de
%la semilla
%elevamos al cuadrado la semilla

for i=1:longitud
    l=semilla.^2
    %primero extraemos las n/2 cifras menores
    semilla=idivide(l,10^fix(n/2));
    %la linea de código anterior, puede no funcionar en algunas versiones
    %antiguas de matlab. Alternativamente, podría implementarse como,
    %semilla=(l-rem(l,10^fix(n/2)))/10^fix(n/2);

    % y a continuación extraemos las n cifras siguientes, que serán el
    % siguiente número aleatorio
    semilla = rem(semilla,10^n)
    % guardamos el número generado en el vector de salida
    secuencia(i,1)=semilla;
end
```

Las secuencias generadas mediante el método *middle square* no son aleatorias, cada número viene completamente determinado por el número anterior. Sin embargo, dichas secuencias parecen aleatorias. La figura 9.1 muestra una secuencia de 100 valores pseudoaleatorios generados mediante *middle square*. Aparentemente, cada valor parece no guardar ninguna relación con los anteriores.

El método descrito, presenta sin embargo serios inconvenientes: Si la semilla contiene ceros o unos a la derecha, estos tienden a repetirse indefinidamente. Además todos los métodos inducen ciclos que hacen que los números generados comiencen a repetirse periódicamente. Por ejemplo, si empezamos con la semilla de cuatro dígitos 3708, La secuencia cae, tras generar los primeros cuatro números aleatorios, en un ciclo en el que los cuatro siguientes números generados se repiten

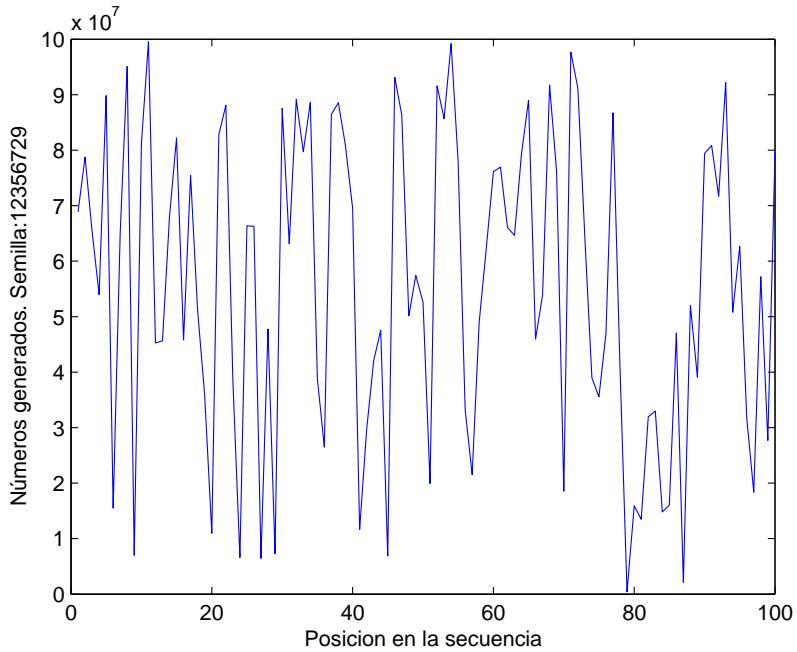


Figura 9.1: Secuencia de 100 números pseudoaleatorios generada mediante el método *middle square*.
0.4

indefinidamente.

$$\begin{aligned}
 \text{semilla} &= 3708 \rightarrow 3708^2 = 13\ 7492\ 64 \\
 7492^2 &= 56\ 1300\ 64 \\
 1300^2 &= 01\ 6900\ 00 \\
 6900^2 &= 47\ 6100\ 00 \\
 6100^2 &= 37\ 2100\ 00 \\
 2100^2 &= 04\ 4100\ 00 \\
 4100^2 &= 16\ 8100\ 00 \\
 8100^2 &= 65\ 6100\ 00 \\
 6100^2 &= 37\ 2100\ 00 \\
 2100^2 &= 04\ 4100\ 00 \\
 4100^2 &= 16\ 8100\ 00 \\
 &\vdots
 \end{aligned}$$

Además, en la secuencia generada puede observarse cómo los ceros situados a las derecha una vez que aparecen en la secuencia, se conservan siempre.

En los algoritmos de generación de números aleatorios, la idea es obtener números en el intervalo $(0, 1)$. Para ello, se generan números entre 0 y un número natural m y luego se dividen los números generados, X_n entre m ,

$$U_n = \frac{X_n}{m}$$

Usualmente, se elige para m un valor tan grande como sea posible. En 1948 Lehmer propuso un algoritmo conocido con el nombre de *linear congruential*. El algoritmo emplea cuatro elementos numéricos,

- I_0 , Semilla. Análoga a la del método *middle square*.
- $m > I_0$, Módulo. Se elige tan grande como sea posible. Cuanto mayor es el módulo, mayor es el periodo del ciclo inducido.
- a , Multiplicador. Debe elegirse de modo que $0 \leq a < m$
- c , Incremento. Debe elegirse de modo que $0 \leq c < m$

El algoritmo de obtención de la secuencia de números pseudoaleatorios se define en este caso como,

$$I_{n+1} = \text{rem}(a \cdot I_n + c, m)$$

Es decir, cada número se obtiene como el resto de la división entera del multiplicador por el número aleatorio anterior más el resto, dividido entre el módulo. Por tanto, se trata de un número comprendido entre 0 y $m - 1$.

El siguiente código permite calcular una secuencia de números aleatorios por el método *linear congruential*

```
function secuencia=linealcr(i0,a,c,m,l)
%Esta función emplea el metodo linear congruential para obtener una
%secuencia de numeros aleatorios de longitud arbitraria
%Uso: secuencia=linealcr(i0,a,c,m,l)
%Variables de entrada: i0=semilla, a=multiplicador, c=incremento, m=modulo.
%l= longitud de la secuencia obtenida.
%variables de salida: secuencia vector columna con los numeros aleatorios
%generados.
%nota: para generarlos con el metodo multiplicative congruential tomar c=0
if (m<=a)|| (m<=c) || (m<=i0)
    error('el modulo debe ser mayor que los otros tres parametros')
end

%inicializamos el vector secuencia
secuencia=zeros(l,1);
%guardamos la semilla como el primer numero de la secuencia
secuencia(1)=i0;
%Creamos un bucle para ir calculando los sucesivos numeros de la secuencia

for j=2:l
    secuencia(j)=rem(a*secuencia(j-1)+c,m);
end
%para obtener numeros en el intervalo (0,1), dividimos el resultado por el
%modulo

secuencia=secuencia/m
```

Una variante del algoritmo descrito, frecuentemente empleada, se obtiene haciendo el incremento igual a cero, $c = 0$. Dicha variante se conoce con el nombre de *multiplicative congruential*, su expresión general sería,

$$I_{n+1} = \text{rem}(a \cdot I_n)$$

Dado que el resto de la división entera de un número cualquiera entre el módulo m solo puede tomar valores enteros comprendidos entre 0 y $m - 1$, es fácil deducir que, en el mejor de los casos, podríamos obtener un ciclo de longitud m , antes de que secuencia de números aleatorios comenzara a repetirse. De ahí la importancia de elegir m lo más grande posible. Por otra parte, los valores de I_0 e c , también influyen en la longitud de los ciclos inducidos.

Si ahora dividimos los numero aleatorios obtenidos por el modulo,

$$x_n = \frac{I_n}{m} \Rightarrow x_n \in [0, 1]$$

Los números así obtenidos pertenecen al intervalo $[0, 1]$ y están regularmente distribuidos.

En la década de los 60 del siglo pasado, La compañía IBM, desarrolló una función, conocida como *Randu*, que utilizaba el algoritmo *multiplicative congruential*. *Randu* tenía definido un multiplicador $c = 2^{16} + 3$ y un módulo $m = 2^{31}$. Se han desarrollado diferentes variante de *Randu*, mejorando progresivamente su rendimiento. Una de las mejoras introducidas fue aumentar el número de semillas empleadas,

$$I_{n+1} = \text{rem}(a \cdot I_n + b \cdot I_{n-1} + \dots)$$

Este método permite obtener periodos mayores.

9.1.1. El generador de números aleatorios de Matlab.

Se trata de un generador mucho mas evolucionado que los ejemplo descritos hasta ahora. La implementación concreta depende de la versión de Matlab empleada. Aquí solo nos referiremos a la conocida como '*twister*'. Se basa en el algoritmo *Mersenne Twister*. Genera números aleatorios, uniformemente distribuidos, en el intervalo $[2^{-53}, 1 - 2^{-53}]$. El periodo de la secuencia generada es $P = (2^{19937} - 1)/2$. Es el algoritmo que emplea Matlab, por defecto, desde al versión 2007 hasta la actualidad.

Para generar números aleatorios se emplea el comando `rand`, que es capaz de generar una matriz de tamaño arbitrario de números aleatorios. Si se invoca el comando `rand`, sin indicar ninguna variable de entrada,

```
>> y=rand
y =
    8.147236863931789e-01
```

Matlab nos devuelve un único número aleatorio. Si introducimos un único valor entero como variable de entrada,

```
>> x=rand(3)
x =
    9.057919370756192e-01    6.323592462254095e-01    5.468815192049838e-01
    1.269868162935061e-01    9.754040499940952e-02    9.575068354342976e-01
    9.133758561390194e-01    2.784982188670484e-01    9.648885351992765e-01
```

Obtenemos una matriz cuadra de números aleatorios cuyas dimensiones coinciden con el número entero introducido.

Si empleamos dos enteros como variables de entrada, `rand(f, c)`, obtenemos una matriz de números aleatorios de dimensión $f \times c$.

```
>> x=rand(3,2)
x =
    7.922073295595544e-01    3.571167857418955e-02
    9.594924263929030e-01    8.491293058687771e-01
    6.557406991565868e-01    9.339932477575505e-01
```

Cada vez que se arranca Matlab, el generador de números aleatorios es reiniciado. Es decir, cada vez que arrancamos Matlab, volvemos a obtener la misma secuencia de números aleatorios. Podemos hacer que el generador se reinicie siempre que queramos, empleado para ello una semilla constituida por un número entero que debe estar comprendido entre 0 y $2^{32} - 1$. Para cada semilla obtendremos una secuencia de números distinta. Para reiniciar el generador, empleamos la función `rng` y el valor de la semilla. Por ejemplo,

```
>> rng(10)
```

Si generamos ahora una secuencia de 3 números aleatorios,

```
>> s1=rand(1,3)
s1 =
    0.7713    0.0208    0.6336
```

Si reiniciamos de nuevo el algoritmo, con la misma semilla de antes y volvemos a generar una secuencia de 3 números aleatorios,

```
>> rnd(10)
>> s1=rand(1,3)
s1 =
    0.7713    0.0208    0.6336
```

Volvemos a obtener los mismo números aleatorios. Esto sucederá siempre que reiniciemos el generador empleado la misma semilla.

Por ultimo, podemos emplear `rng` para conocios el estado del generador de números aleatorios en cualquier momento, durante una sesión con Matlab. Para ello, usamos `rng` Sin ninguna variable de entrada,

```
estado=rnd
```

El resultado es:

```
estado =
  struct with fields:

    Type: 'twister'
    Seed: 10
    State: [625x1 uint32]
```

Nos devuelve el generador utilizado para obtener los números aleatorios, la semilla, en este caso `twister`, La semilla empleada para inicializar el generador y un vector de 625 enteros que son las semillas que el generador empleará para generar el siguiente número aleatorio.

Si queremos cambiar el generador de números aleatorios empleado por matlab, empleamos de nuevo la función `rnd`, indicando un valor para la semilla y el nombre del generador entre comillas¹,

¹Para obtener una relación completa de generadores, consultar la ayuda de matlab

```

>> rng(0,'philox')
>> estado = rng
estado =
    struct with fields:

        Type: 'philox'
        Seed: 0
        State: [7x1 uint32]

```

Supongamos que después de obtener el estado del generador, seguimos generando números aleatorios. Podemos hacer que el generador vuelva al estado que tenía empleando la estructura obtenida para reiniciar el generador. A partir de ahí el generador volverá a repetir de nuevo la misma secuencia de números aleatorios. La siguiente secuencia de instrucciones de Matlab ilustra este procedimiento,

```

%ejemplo de vuelta del generador de numeros aleatorios a un estado
%anterior

%generamos una secuencia de 5 números,
s1=rand(1,5)
%guardamos el estado actual del generador
estado=rng;

%generamos una nueva secuencia de 5 números
s2=rand(1,5)

%volvemos a llevar el generador al estado anterior,
rng(estado)

%volvemos a generar una secuencia de 5 números que será igual que s2
s2bis=rand(1,5)

```

Si ejecutamos la secuencia obtendremos,

```

s1 =
    0.3655    0.6975    0.1789    0.4549    0.4396

s2 =
    0.5174    0.7001    0.4564    0.7388    0.4850

s2bis =
    0.3655    0.6975    0.1789    0.4549    0.4396

```

Por último podemos reiniciar el generador empleando,

```
>> rng('shuffle')
```

En este caso, matlab suministra una semilla obtenida a partir de la hora actual.

9.2. Probabilidad y distribuciones de probabilidad

La probabilidad es una propiedad asociada a los sucesos aleatorios. Decimos que un suceso es aleatorio cuando no depende de una causa determinista que nos permite predecir cuándo va a

suceder.

9.2.1. Sucesos aleatorios discretos

Por ejemplo, obtener *cara* o *cruz* al lanzar una moneda al aire es un ejemplo sencillo de proceso aleatorio, ya que no es posible conocer de antemano cuál será el resultado del lanzamiento. La probabilidad da una medida de las posibilidades de que un determinado suceso aleatorio tenga lugar.

Volviendo al ejemplo de la moneda lanzada al aire, las posibilidades son dos:

1. Que salga cara.
2. Que salga cruz.

Ambos sucesos son igualmente probable tras un lanzamiento, uno de los dos debe darse necesariamente y no hay, en principio, ningún otro suceso posible. Podemos entonces asociar valores numéricos a la probabilidad con la que se dan las distintas posibilidades:

- La probabilidad de que salga cara o cruz —cualquiera de las dos— debe tener asociado el valor máximo ya que *necesariamente* ha de salir cara o cruz. Habitualmente se toma como valor máximo de un suceso aleatorio el valor 1, que estaría asociado con el suceso necesario o cierto,

$$P(\text{cara o cruz}) = 1$$

- La probabilidad de que no salga ni cara ni cruz. Puesto que hemos establecido que ha de salir necesariamente cara o cruz, estamos ante un suceso imposible. A los sucesos imposibles se les asigna probabilidad 0,

$$P(\emptyset) = 0$$

- La probabilidad de que salga cara. Como en una moneda cara y cruz son igualmente probables, la probabilidad de sacar cara, sera la *mitad* de la probabilidad de sacar cara o cruz,

$$P(\text{cara}) = \frac{1}{2} = 0,5$$

- La probabilidad de sacar cruz. De modo análogo al caso anterior será la mitad de la probabilidad de sacar cara o cruz,

$$P(\text{cruz}) = \frac{1}{2} = 0,5$$

Lanzar una moneda o un dado, elegir una carta al azar, rellenar una quiniela, son ejemplos de fenómenos aleatorios discretos. Reciben este nombre, porque los sucesos posibles asociados al fenómeno son numerables, es decir, se pueden contar. Así por ejemplo en el lanzamiento de una moneda solo hay dos sucesos posibles (*cara* o *cruz*), en el de un dado hay seis (cada una de sus caras), en el de la elección de una carta hay cuarenta (si se trata de una baraja española) etc. El conjunto de sucesos posibles recibe el nombre de variable aleatoria.

Una distribución de probabilidad discreta, es una función que asocia cada caso posible de un fenómeno aleatorio con su probabilidad. Si vamos sumando sucesivamente las probabilidades de todos los fenómenos posibles, lo que obtenemos es una nueva función que recibe el nombre de probabilidad acumulada (Más adelante volveremos sobre esta idea). La figura 9.2 muestra la distribución de probabilidad y la probabilidad acumulada asociadas al lanzamiento de una moneda al aire.

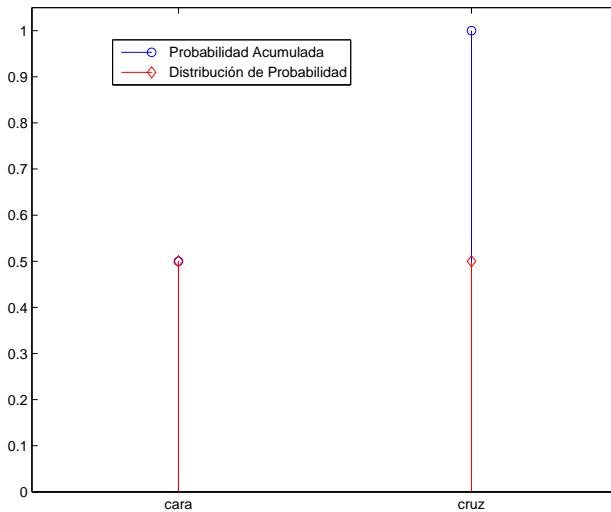


Figura 9.2: Distribución de probabilidad y probabilidad acumulada de los resultados de lanzar una moneda al aire.

Si todos los sucesos posibles dentro de un fenómeno aleatorio discreto tienen la misma probabilidad de suceder, es posible determinar directamente la probabilidad de cada uno de ellos sin más que dividir los casos favorables entre los casos posibles,

$$P(a) = \frac{\text{casos en que se da } a}{\text{numero total de casos posibles}}$$

Así por ejemplo la probabilidad de sacar un 6 cuando se lanza un dado será,

$$P(a) = \frac{\text{casos en que se da } a = 1}{\text{numero total de casos posibles} = 6} = \frac{1}{6}$$

Una propiedad que deben cumplir los sucesos aleatorios es que la suma de las probabilidades de todos los sucesos posibles debe ser igual a la unidad,

$$\sum_{i=1}^n P(i) = 1$$

Es decir, *necesariamente* debe darse alguno de los casos posibles.

Esto nos lleva al carácter aditivo o acumulativo de la probabilidad. Si elegimos un conjunto de casos posibles de un fenómeno aleatorio cualquiera, la probabilidad de que se de alguno de ellos será la suma de las probabilidades de los casos individuales. Por ejemplo la probabilidad al lanzar un dado de obtener un 2, un 3 ó un 6 será,

$$P(2, 3, 6) = P(2) + P(3) + P(6) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{3}{6} = \frac{1}{2}$$

lo que coincide con lo que intuitivamente cabía esperar, que la probabilidad de sacar uno cualquiera de los tres números fuera 0,5. La figura 9.3 muestra la distribución de probabilidad y la probabilidad acumulada para el lanzamiento de un dado. La probabilidad acumulada guarda

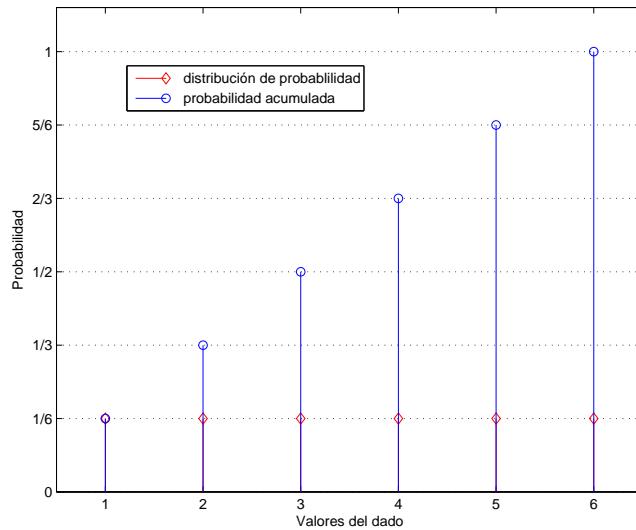


Figura 9.3: Distribución de probabilidad y probabilidad acumulada de los resultados de lanzar un dado al aire.

relación con el carácter aditivo de la probabilidad que acabamos de describir. Si consideramos el conjunto de valores posibles en orden: $1 < 2 < 3 < 4 < 5 < 6$, la probabilidad acumulada para cada valor es la probabilidad de sacar un número al lanzar el dado menor o igual que dicho valor. Así por ejemplo la probabilidad de sacar un número menor o igual que tres sería,

$$P(n \leq 3) = P(1) + P(2) + P(3) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{3}{6} = \frac{1}{2}$$

Hasta ahora, todas las distribuciones discretas de probabilidad que hemos visto —moneda, dado— asignan la misma probabilidad a todos los sucesos posibles. En general esto no tiene porqué ser así. Supongamos un dado trucado en el que la probabilidad de sacar un 6 valga 0,5, es decir en promedio tiende a salir un 6 la mitad de las veces que se lanza el dado. Supongamos además que todos los demás números salen con la misma probabilidad,

$$\begin{aligned} P(1) &= P(2) = P(3) = P(4) = P(5) = \frac{1}{10} \\ P(6) &= \frac{1}{2} \\ \sum_i P(i) &= 1 \end{aligned}$$

La figura 9.4 muestra la distribución de probabilidad y la probabilidad asociada al lanzamiento del dado trucado del ejemplo.

9.2.2. Distribuciones de probabilidad continuas

Hasta ahora, hemos considerado fenómenos en los que los casos posibles —la variable aleatoria— son finitos y numerables. Supongamos ahora un fenómeno aleatorio en que su variable aleatoria,

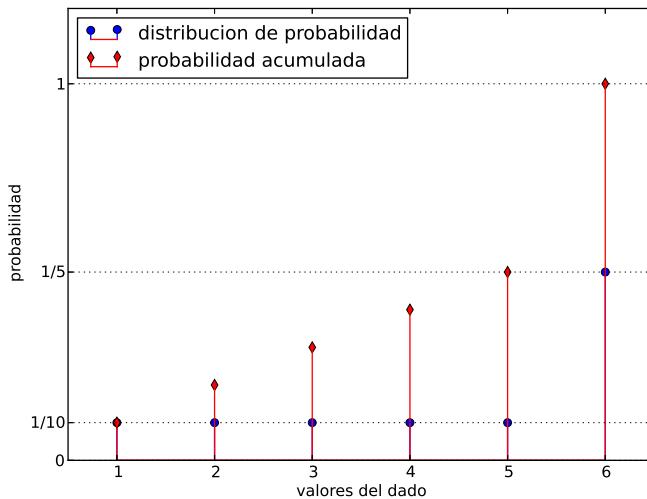


Figura 9.4: Distribución de probabilidad y probabilidad acumulada de los resultados de lanzar un dado trucado al aire.

x , es continua. Esto es: puede tomar todos los valores reales comprendidos en cierto intervalo, $x \in [a, b]$. Supongamos que todos los números contenidos en dicho intervalo pueden ser elegidos con igual probabilidad. Podríamos representar la distribución de probabilidad como una línea horizontal continua $f(x) = c$ que cubriera todo el intervalo $[a, b]$. Para obtener el valor constante c de dicha distribución de probabilidad bastaría *sumar* la probabilidad asociada por la distribución a cada valor del intervalo, igualar el resultado a uno y despejar c .

En realidad, no podemos sumar los infinitos valores que contiene el intervalo. De hecho lo que hacemos es sustituir la suma por una integral,

$$\int_a^b c \cdot dx = 1 \Rightarrow c = \frac{1}{b - a}$$

La figura 9.5 muestra la distribución de probabilidad resultante

Es importante darse cuenta de la diferencia con el caso discreto. El intervalo $[a, b]$ contiene infinitos números. Si tratáramos de asignar un valor a la probabilidad de obtener un número dividiendo casos favorables entre casos posibles, como hacíamos en el caso discreto, obtendríamos un valor 0 para todos los números.

La distribución de probabilidad $f(x) \geq 0$ representa en el caso continuo una *densidad* de probabilidad. Así, dado un número cualquiera $x_0 \in [a, b]$, el valor que toma la distribución de probabilidad $f(x_0)$ representa la probabilidad de obtener un número al azar en un intervalo dx en torno a x_0 ,

$$P\left(x_0 - \frac{dx}{2} \leq x \leq x_0 + \frac{dx}{2}\right) = f(x_0)$$

La probabilidad, vendrá siempre asociada a un intervalo, y se obtendrá integrando la distribución de probabilidad en dicho intervalo. Así por ejemplo,

$$P(x_1 \leq x \leq x_2) = \int_{x_1}^{x_2} f(x) dx$$

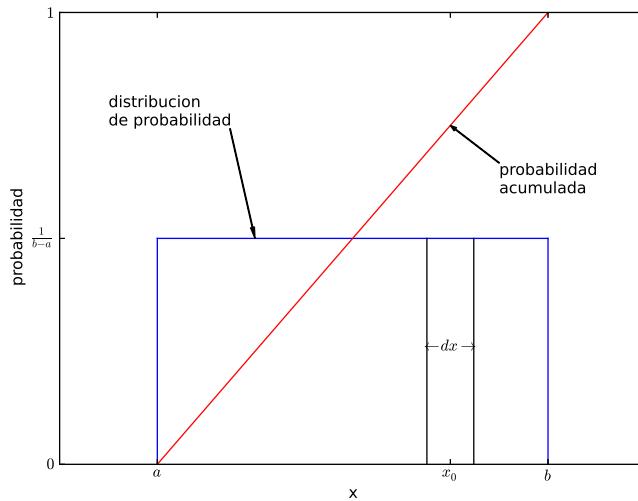


Figura 9.5: Distribución de probabilidad uniforme para un intervalo $[a, b]$ y probabilidad acumulada correspondiente

representa la probabilidad de obtener un número al azar en intervalo $[x_1, x_2]$. La probabilidad acumulada, se obtiene también mediante integración,

$$P(x) = \int_a^x f(x) dx$$

Para el caso de la distribución de probabilidad constante descrita más arriba obtendríamos una línea recta que corta el eje de abcisas en a y el valor de ordenada 1 en b ,

$$P(x) = \int_a^x f(x) dx = \int_a^x \frac{1}{b-a} dx = \frac{x-a}{b-a}, \quad x \in [a, b]$$

La figura 9.5 muestra la probabilidad acumulada para el ejemplo de la distribución uniforme.

Si integramos la densidad de probabilidad sobre todo el intervalo de definición de la variable aleatoria, el resultado debe ser la unidad, puesto que dicha integral representa la probabilidad de obtener un número cualquiera entre todos los disponibles,

$$\int_a^b f(x) dx = 1, \quad a \leq x \leq b$$

En probabilidad se emplean muchos tipos de distribuciones para representar el modo en que se dan los sucesos aleatorios en la naturaleza. A continuación presentamos dos ejemplos muy conocidos:

Distribución exponencial La distribución exponencial está definida para sucesos que pueden tomar cualquier valor real positivo ó 0, $x \in [0, \infty)$. se representa mediante una función exponencial decreciente,

$$f(x) = \frac{1}{\mu} e^{-\frac{x}{\mu}}$$

La figura 9.6 muestra un ejemplo de distribución exponencial. Es interesante observar como su probabilidad acumulada tiende a 1 a medida que x tiende a ∞ .

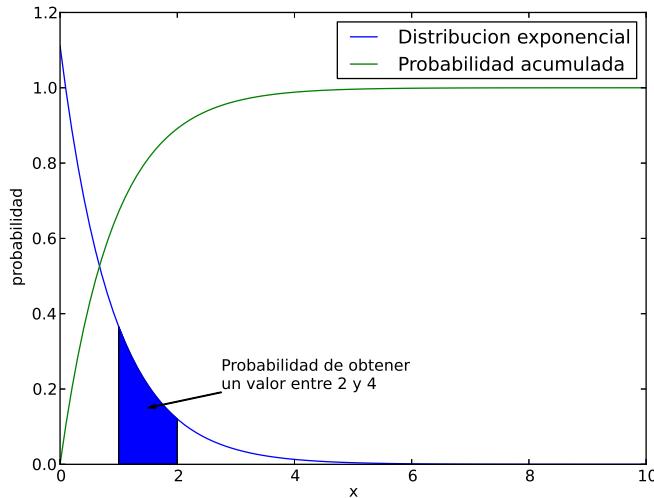


Figura 9.6: Distribución de probabilidad exponencial

$$P(0 \leq x < \infty) = \int_0^{\infty} \frac{1}{\mu} e^{-\frac{x}{\mu}} dx = 1$$

Distribución Normal Aparece con gran frecuencia en la naturaleza. En particular, como veremos más adelante, está relacionada con la incertidumbre inevitable en las medidas experimentales. Esta definida para sucesos aleatorios que pueden tomar cualquier valor real, $-\infty < x < \infty$. Se representa mediante la función de Gauss,

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

La figura 9.7 muestra un ejemplo de distribución normal.

Parámetros característicos de una distribución Los parámetros característicos permiten definir propiedades importantes de las distribuciones de probabilidad. Nos limitaremos a definir los dos más importantes:

- **Media ó valor esperado.** El valor esperado de una distribución se obtiene integrando el producto de cada uno de los valores aleatorios sobre los que está definida la distribución, por su densidad de probabilidad,

$$\mu = \int_a^b xf(x)dx, \quad a \leq x \leq b$$

Así, para una distribución uniforme definida en un intervalo $[a, b]$,

$$\mu = \int_a^b x \frac{1}{b-a} dx = \frac{a+b}{2}$$

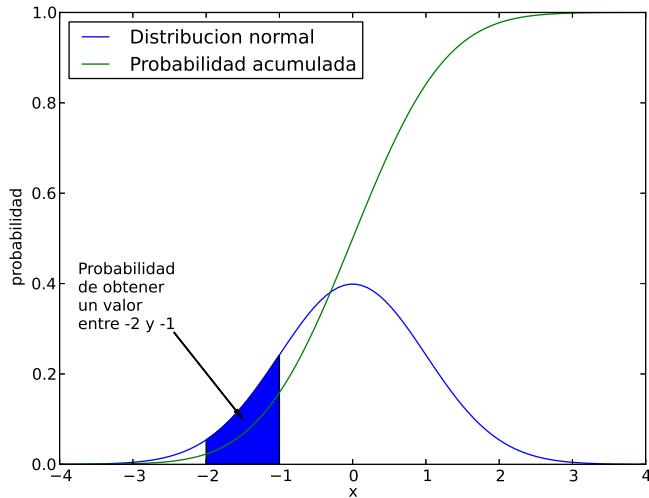


Figura 9.7: Distribución de probabilidad normal

para una distribución exponencial,

$$\mu = \int_0^\infty x \frac{1}{\mu} e^{-\frac{x}{\mu}} dx = \mu$$

y para la distribución normal,

$$\mu = \int_{-\infty}^\infty x \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \mu$$

Es interesante observar como en el caso de las distribuciones exponencial y normal la media es un parámetro que forma parte de la definición de la función de distribución.

- **Varianza** La varianza da una medida de la dispersión de los valores de la distribución en torno a la media. Se define como,

$$\sigma^2 = \int_a^b (x - \mu)^2 f(x) dx, \quad a \leq x \leq b$$

Para una distribución uniforme definida en el intervalo $[a, b]$, tomará el valor,

$$\sigma^2 = \int_a^b (x - \mu)^2 \frac{1}{b-a} dx = \frac{(b-a)^2}{12}$$

para una distribución exponencial,

$$\sigma^2 = \int_0^\infty (x - \mu)^2 \frac{1}{\mu} e^{-\frac{x}{\mu}} dx = \mu^2$$

y para la distribución normal,

$$\sigma^2 = \int_{-\infty}^\infty x \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \sigma^2$$

- **Desviación estándar.** Es la raíz cuadrada de la varianza: $\sigma = \sqrt{\sigma^2}$

9.3. El teorema del límite central

El teorema del límite central juega un papel muy importante a la hora de analizar y evaluar resultados experimentales. Nos limitaremos a enunciarlo, pero no daremos una demostración.

Supongamos que tenemos un fenómeno aleatorio continuo que viene caracterizado por una variable aleatoria x . Además el fenómeno aleatorio viene descrito por una distribución de probabilidad arbitraria $f(x)$ de media μ y varianza σ^2 .

Supongamos que realizamos un experimento consistente en obtener n valores de x al azar. Los valores así obtenidos, deberán seguir la distribución de probabilidad $f(x)$. Para caracterizar nuestro experimento, calcularemos la media de los n valores obtenidos,

$$\bar{x}_n = \frac{1}{n} \sum_{i=1}^n x_i$$

Si repetimos el experimento muchas veces, obtendremos al final una colección de medias; tantas, como veces hemos repetido el experimento, $\{\bar{x}_{n1}, \bar{x}_{n2}, \dots, \bar{x}_{nj}, \dots\}$. El teorema del límite central, establece que las medias así obtenidas constituyen a su vez valores de una variable aleatoria que sigue una distribución normal, cuya media coincide con la media μ de la distribución original $f(x)$ y cuya varianza coincide con el valor de la varianza de la distribución original σ^2 dividida por el número n de valores de x obtenidos al azar en cada uno de los experimentos. (En todos los experimentos se obtiene siempre el mismo número de valores de la variable aleatoria x).

$$x, f(x), \mu, \sigma^2 \Rightarrow \bar{x}_n, f(\bar{x}_n) = \frac{1}{\sqrt{2\pi\sigma^2/n}} e^{-\frac{(x-\mu)^2}{2\sigma^2/n}}$$

Veamos un ejemplo para ilustrar el teorema. Hemos visto en la sección 9.1.1 que el comando `rand` de Matlab genera números aleatorios uniformemente distribuidos en el intervalo² $(0, 1)$. Podemos considerar que el ordenador genera números aleatorios que siguen aproximadamente una distribución continua en dicho intervalo. Podemos alterar dicho intervalo multiplicando el resultado de `rand` por un número cualquiera. Así, por ejemplo,

```
>> x=5*rand(100)
```

genera un vector de 100 números aleatorios en el intervalo $(0, 5)$. Además podemos desplazar el centro del intervalo sumando o restando al resultado anterior un segundo número. Así,

```
>> x =5*rand(100) -2
```

genera un vector de 100 números aleatorios en el intervalo $(-2, 3)$.

Para comprobar el teorema del límite central, podemos construir un bucle que genere un vector de n números aleatorios en un determinado intervalo, calcule su media y guarde el resultado en un vector de medias. El siguiente código de Matlab, realiza dicho cálculo primero generando un vector de 10 números aleatorios ($n = 10$) y después generando un vector de 100, ($n = 100$). El programa genera en cada caso un millón de vectores distintos.

```
%Este código genera un millón de vectores, formados por números
%aleatorios distribuidos uniformemente en el intervalo (-1,1).
%Calcula el valor medio de los valores de cada vector y guarda
%el resultado en un nuevo vector.
```

²En realidad es el intervalo $[2^{-52}, 1 - 2^{-52}]$ podemos considerarlo equivalente al intervalo $(0, 1)$ debido a la precisión finita de los números máquina

```

media10 = zeros(1,1000000); %vector para guardar las medias
for i=1:1000000
    media10(i) = mean(2*rand(10,1)-1);
end

%se repite el mismo calculo pero ahora con vectores de 100
%elementos
media100 = zeros(1,1000000)
for i=1:1000000
    medias100(i) = mean (2*rand(100,1)-1);
end

```

S

De acuerdo con el teorema del límite central, los dos vectores de medias obtenidos, `media10` y `media100`, deben seguir distribuciones normales tales que,

1. La media de la distribución normal debe coincidir con la media de la distribución a la que pertenecían los datos originales. Como hemos tomado datos distribuidos uniformemente en el intervalo $[-1, 1]$, La media de dicha distribución es,

$$\mu = \frac{(b+a)}{2} = \frac{1+(-1)}{2} = 0$$

2. La varianza de las distribuciones normales a las que pertenecen las medias obtenidas será en cada caso el resultado de dividir la varianza de la distribución uniforme original entre el número de datos empleados para calcular dichas medias,

$$\sigma_{10}^2 = \frac{\sigma^2}{10} = \frac{\frac{(b-a)^2}{12}}{10} = \frac{\frac{(1-(-1))^2}{12}}{10} = \frac{1}{30}$$

$$\sigma_{100}^2 = \frac{\sigma^2}{100} = \frac{\frac{(b-a)^2}{12}}{100} = \frac{\frac{(1-(-1))^2}{12}}{100} = \frac{1}{300}$$

S

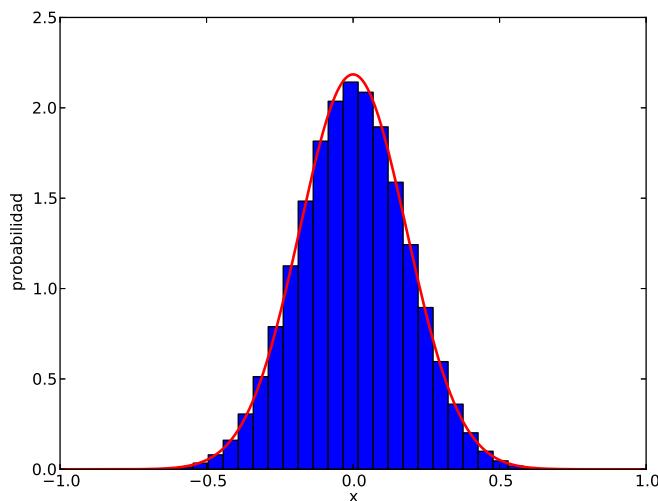
Por tanto, las distribuciones normales a las que pertenecen las medias calculadas son,

$$f(\bar{x}_{10}) = \frac{1}{\sqrt{2\pi\frac{1}{30}}} e^{-\frac{(\bar{x}-\mu)^2}{2\frac{1}{30}}}, \quad f(\bar{x}_{100}) = \frac{1}{\sqrt{2\pi\frac{1}{300}}} e^{-\frac{(\bar{x}-\mu)^2}{2\frac{1}{300}}}$$

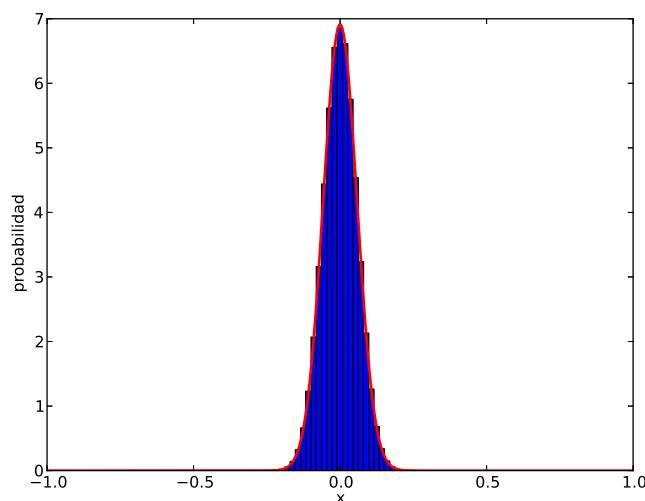
Las figuras 9.8(a) y 9.8(b), muestran un histograma normalizado de las medias obtenidas con el código que acabamos de describir ³. Sobre dicho histograma se ha trazado mediante una línea roja, la función de Gauss que representa en cada caso la distribución normal a que pertenecen las medias. Como puede observarse en ambos casos el histograma normalizado y la distribución coinciden bastante bien.

De las figuras, es fácil deducir que cuantos más datos empleemos en el cálculo de las medias, más centrados son los resultados obtenidos en torno a la media de la distribución original. Por otro lado, esto es una consecuencia lógica del hecho de que la varianza de la distribución normal que siguen las medias, se obtenga dividiendo la varianza de la distribución original, por el número de datos n ; cuantos mayor es n más pequeña resulta la varianza de la distribución normal resultante.

³El histograma normalizado se obtiene dividiendo el número de puntos que pertenecen a cada barra del histograma entre el número total de puntos y el ancho de la barra. De esta manera, si sumamos los valores representados en cada barra multiplicados por su ancho, el resultado es la unidad. Solo normalizando el histograma es posible compararlo con la distribución a que pertenecen los datos, que cumple por definición tener área unidad.



(a) Distribución para medias de 10 valores



(b) Distribución para medias de 100 valores

Figura 9.8: Teorema del límite central: Comparación entre histogramas normalizados para un millón de medias y la distribución normal a que pertenecen

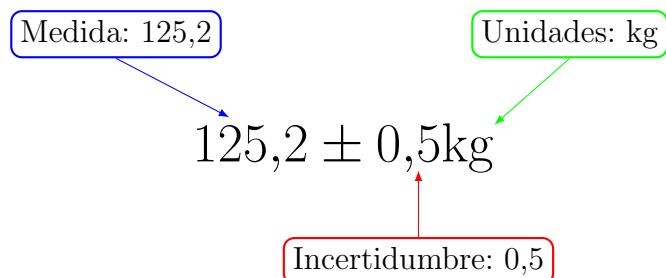


Figura 9.9: Modo correcto de expresar una medida experimental

9.4. Incertidumbre en las medidas experimentales

Siempre que realizamos una medida de una cantidad física dicha medida estará afectada por un cierto error. Dado que no conocemos cual es el valor verdadero que toma la magnitud que estamos midiendo, no podemos tampoco conocer el error que cometemos al medirla. Decimos entonces que toda medida está afectada por un cierto grado de incertidumbre.

Lo que sí es posible hacer, es acotar el grado de incertidumbre de una medida, es decir estimar unos límites con respecto a la medida realizada dentro de los cuales debe estar contenido el verdadero valor que toma la magnitud medida.

Cualquier medida experimental debe incluir junto al resultado de la medida el valor de su incertidumbre y, por supuesto, las unidades empleadas en la medida. La figura 9.9 muestra un modo correcto de expresar una medida experimental, aunque no es el único. Como puede observarse, la medida y su incertidumbre se separan en la representación mediante el símbolo \pm . La incertidumbre se representa como un entorno alrededor del valor medido, dentro del cual estaría incluido el valor real de la medida⁴.

9.4.1. Fuentes de incertidumbre.

Antes de entrar en el estudio de las fuentes de incertidumbre, es importante destacar que el estudio de la medición constituye por sí mismo una ciencia, conocida con el nombre de metrología. Lo que vamos a describir a continuación tanto referido a las fuentes de incertidumbre como al modo de estimarla, es incompleto y representa tan solo una primera aproximación al problema. La incertidumbre de una medida tiene fundamentalmente dos causas:

Incertidumbre sistemática de precisión. La primera es la precisión limitada de los aparatos de medida. Cualquier aparato de medida tiene una precisión que viene determinada por la unidad o fracción de unidad más pequeña que es capaz de resolver. Por ejemplo, una regla normal, es capaz de resolver (distinguir) dos longitudes que se diferencien en $0,5\text{mm}$, un reloj digital sencillo es capaz de resolver tiempos con una diferencia de 1s . etc. La incertidumbre debida a la precisión finita de los aparatos de medida recibe el nombre de *Incertidumbre sistemática de precisión*. Se llama así porque depende exclusivamente de las características del aparato de medida que, en principio, son siempre las mismas.

La incertidumbre sistemática de precisión suele expresarse añadiendo a la medida la mitad de la división mínima de la escala del aparato de medida empleado. Así, por ejemplo, si medimos

⁴Como se verá más adelante se debe indicar también el *grado de confianza* con el que esperamos que la medida caiga dentro del intervalo indicado por la incertidumbre

123mm con una regla graduada en milímetros, podríamos expresar la medida como $123 \pm 0,5\text{mm}$ o también como $12,3 \pm 0,05\text{cm}$.

La incertidumbre sistemática de precisión representa una distribución uniforme de media el valor de la medida realizada y anchura la división mínima de la escala. Es decir, se considera que el valor real de la medida está dentro de dicho intervalo intervalo con una probabilidad del 100 %. Volviendo al ejemplo anterior de la regla, el valor real de nuestra medida estaría comprendido en el intervalo [122,5, 123,5], y podría tomar cualquier valor de dicho intervalo con igual probabilidad.

Incertidumbre estadística. La segunda fuente de incertidumbre se debe a factores ambientales, que modifican de una vez para otra las condiciones en que se realiza una medida experimental. Estos factores hacen que las medidas realizadas sean sucesos aleatorios. En principio podría describirse mediante una distribución de probabilidad $f(x)$, pero en la práctica desconocemos de qué distribución se trata y ni siquiera sabemos cuál es su valor esperado μ o su varianza σ .

Afortunadamente, el teorema del límite central, que describimos en la sección 9.3 proporciona un sistema de estimar la incertidumbre estadística.

Supongamos que repetimos n veces la misma medida experimental. Por ejemplo: tenemos un recipiente con agua, lo calentamos y tomamos la temperatura del agua cuando ésta rompe a hervir. Repetimos n veces el experimento y obtenemos un conjunto $\{x_1, x_2, \dots, x_n\}$ de medidas de la temperatura de ebullición del agua. Cabe esperar que, debido a la incertidumbre estadística, los valores obtenidos sean distintos y, a la vez próximos entre sí⁵.

Sabemos que dichos valores deben seguir una cierta distribución de probabilidad. Podríamos estimar su valor esperado μ mediante el cálculo de la media aritmética de las medidas tomadas.

$$\mu \approx \bar{x}_n = \frac{1}{n} \sum_{i=1}^n x_i$$

La media así calculada se toma como el valor resultante de la medida realizada.

De acuerdo con el teorema del límite central, \bar{x}_n es una variable aleatoria que debe seguir una distribución normal de media μ y de varianza σ^2/n . Como tampoco conocemos el valor de la varianza σ^2 de la distribución de probabilidad que siguen los datos medidos, la estimamos a partir de los propios datos medidos, de acuerdo con la siguiente ecuación,

$$\sigma^2 \approx s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

Dividiendo s^2 por el número de muestras empleado se puede aproximar la varianza de la distribución normal que debe seguir \bar{x}_n ,

$$\sigma_{xn}^2 \approx s_{xn}^2 = \frac{s^2}{n}$$

La incertidumbre, se puede asociar con la raíz cuadrada la varianza que acabamos de estimar,

$$s_{xn} = \sqrt{s_{xn}^2} = \sqrt{\frac{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}{n}}$$

esta cantidad recibe el nombre de desviación estándar.

⁵De hecho, si aparecen algunos valores claramente alejados del resto, lo habitual es considerar que están afectados por algún error desconocido y desecharlos. Dichos valores suelen recibir el nombre de valores o datos aberrantes.

9.4.2. Intervalos de confianza.

Como hemos visto, a partir del teorema central del límite podemos asociar la media de los valores obtenidos al repetir una medida y su desviación estándar con una distribución normal. Podemos ahora, en un segundo paso, asociar la incertidumbre de la medida realizada con la probabilidad representada por dicha distribución normal.

Si el valor obtenido tras realizar n repeticiones de una medida es \bar{x}_n y su desviación típica es s_{xn} ¿Cuál es la probabilidad de que el valor medido esté realmente comprendido en el intervalo $[\bar{x}_n - s_{xn}, \bar{x}_n + s_{xn}]$?

Como vimos en el apartado 9.2.2, dicha probabilidad puede calcularse integrando la distribución normal obtenida entre $\bar{x}_n - s_{xn}$ y $\bar{x}_n + s_{xn}$,

$$P(\bar{x}_n - s_{xn} \leq x \leq \bar{x}_n + s_{xn}) = \frac{1}{\sqrt{2\pi}\sigma_{xn}^2} \int_{\bar{x}_n - s_{xn}}^{\bar{x}_n + s_{xn}} e^{-\frac{(x-\bar{x}_n)^2}{2s_{xn}^2}}$$

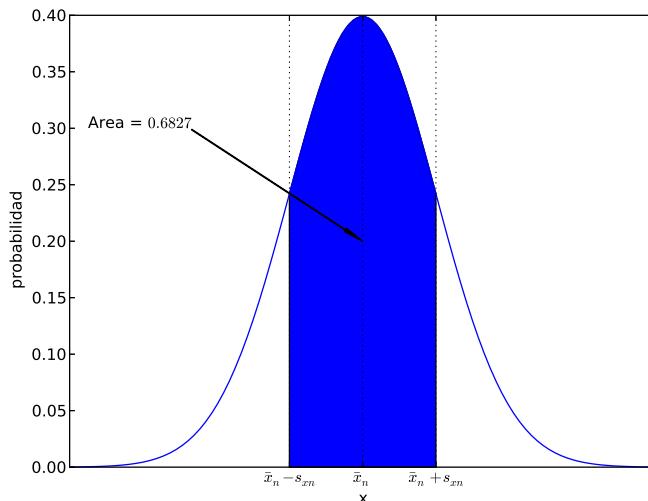


Figura 9.10: Intervalo de confianza del 68,27%

El valor de dicha integral es 0,6827. Si expresamos este valor como un tanto por ciento, concluimos que la probabilidad de que el valor medido esté en el intervalo $[\bar{x}_n - s_{xn}, \bar{x}_n + s_{xn}]$ es del 68,27 %. Dicho de otra manera: el intervalo $[\bar{x}_n - s_{xn}, \bar{x}_n + s_{xn}]$ corresponde con el *intervalo de confianza* del 68,27 %. La figura 9.10 muestra el área bajo la distribución normal, correspondiente a dicha probabilidad.

Volviendo al problema de la medida, la manera correcta de expresarla sería en nuestro caso: $\bar{x} \pm s_{xn}$ (unidades). Indicando que la incertidumbre corresponde con el intervalo de confianza del 68,27 %.

Observando la figura 9.10, es fácil plantearse la siguiente cuestión: ¿Qué valor de la incertidumbre contendría el valor de real de la medida con una probabilidad de, por ejemplo, el 95 %? O, dicho de otra manera, ¿Cuánto tengo yo que *alargar* el intervalo de integración para que el área contenida bajo la distribución normal valga 0,95?

Para estimarlo se emplea la inversa de la función de probabilidad acumulada. Como se explicó en el apartado 9.2.2, la función de probabilidad acumulada se obtiene por integración de la distribución

de probabilidad correspondiente.

Dicha función representa la probabilidad de obtener un resultado entre el límite inferior para el que está definida la distribución de probabilidad y el valor x para el que se calcula la función. La inversa de la función de probabilidad acumulada nos indica, dado un valor de la probabilidad comprendido entre 0 y 1, a qué valor x corresponde dicha probabilidad acumulada.

Dado que la distribución normal no tiene primitiva, solo puede integrarse numéricamente. Por tanto, su inversa, solo puede obtenerse también numéricamente. Matlab, suministra la función `norminv(p)`, que permite obtener directamente los valores de la inversa de la función de probabilidad normal acumulada.

Además, la función `norminv(p)` calcula la función de probabilidad inversa acumulada para una distribución de probabilidad normal de media 0 y varianza 1. En realidad esto no supone ningún problema ya que para obtener el valor correspondiente a cualquier otra distribución normal, basta multiplicar el valor x obtenido con `norminv(p)` por el valor de la desviación estándar y sumarle el valor de la media de la distribución deseada.

Antes de seguir adelante, veamos algunos ejemplos de uso de esta función. Por ejemplo si introducimos en la función el valor 0,1,

```
>> p = 0.1;
>> x = norminv(p)
>> x = -1.2815515655446004
```

Es decir la probabilidad de obtener un número en el intervalo $(-\infty, -1.2815515655446004]$ vale 0,1 o, lo que es equivalente, un 10%.

Otro ejemplo especialmente significativo: Si introducimos el valor 0,5,

```
>> p = 0.5;
>> x = norminv(p)
>> x = 0.0
```

Como cabría esperar de la simetría de la distribución normal, hay un 0,5 —50%— de probabilidades de obtener un número entre $-\infty$ y 0. La figura 9.11 muestra la función inversa de probabilidad acumulada y en particular el punto correspondiente al valor 0,5.

Supongamos ahora que queremos repetir el cálculo para una distribución normal de media $\mu = 6$ y desviación típica $\sigma = 2$. Para el valor 0,1 obtendríamos,

```
>> p = 0.1;
>> x = norminv(p)
>> x = -1.2815515655446004
>> mu = 6
>> sigma = 2
>> x2 = sigma*x+mu
>> x2 = -1.5631031310892007
```

Por tanto, ahora tenemos un 10% de probabilidades de obtener un número entre $-\infty$ y -1.5631031310892007

Retomando el hilo de nuestro problema, lo que a nosotros realmente nos interesa, es encontrar intervalos de confianza, es decir conocer el intervalo en torno al valor medio \bar{x}_n que corresponde a una determinada probabilidad. La función inversa de probabilidad acumulada nos da el intervalo entre $-\infty$ y x correspondiente a una determinada probabilidad. Si combinamos los resultados de dicha función con las propiedades de simetría y área unidad de la función normal podemos obtener el intervalo centrado en torno a la media correspondiente a una determinada probabilidad.

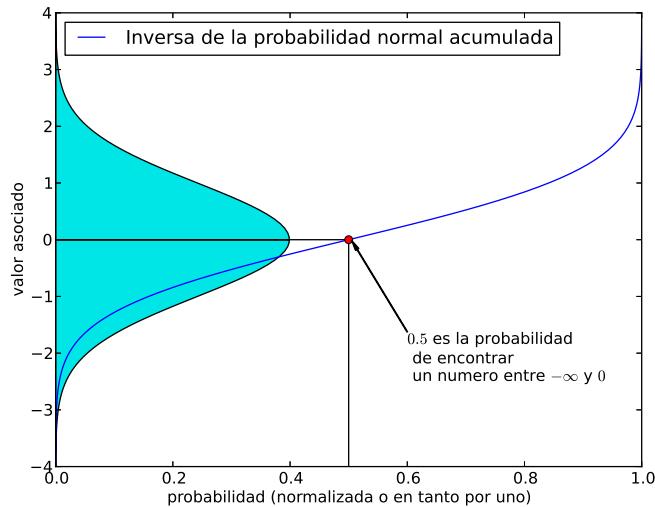


Figura 9.11: Función inversa de probabilidad normal acumulada
s

Así el intervalo $[-x, x]$ correspondiente a una probabilidad del $P\%$, abarca un área bajo la curva $P/100$. Por tanto el área que queda en las colas —a derecha e izquierda del intervalo— es $1 - x/100$. Como la distribución normal es simétrica, dicha área debe dividirse en dos, una correspondiente a la cola $[-\infty, -x]$ y la otra correspondiente a la cola $[x, \infty]$.

La figura 9.12 muestra gráficamente la relación de áreas que acabamos de describir; las dos colas aparecen pintadas en azul.

Para obtener el límite inferior, $-x$, del intervalo de probabilidad $P\%$ basta emplear la función `norminv`, introduciendo como variable de entrada el área de la cola, `x=norminv((1-P/100)/2)`. Así por ejemplo, para obtener el límite inferior del intervalo de confianza correspondiente a una probabilidad del 90 %,

```
>> p=(1-0.90)/2
p =
    0.0500

>> x=norminv(p)
x =
   -1.6449
```

Por tanto el intervalo de confianza correspondiente a una probabilidad del 90 % para una distribución normal de media cero y desviación estándar uno es, $I_{90\%} = [-1,6449, 1,6449]$

Supongamos ahora que hemos realizado un conjunto de n medidas de una determinada magnitud física, por ejemplo temperatura en $^{\circ}\text{C}$, y obtenemos su media $\bar{x}_n = 6,5 ^{\circ}\text{C}$ y su desviación estándar $s_{xn} = 2$. Si queremos dar el resultado de la medida con una incertidumbre correspondiente a un intervalo de confianza del 95 %, calculamos el intervalo de confianza empleando la función inversa de probabilidad acumulada normal, y después, multiplicamos el resultado por s_{xn} para ajustarlo a la distribución normal que sigue la media de nuestros datos,

```
>> xm = 6; %media de las medidas
```

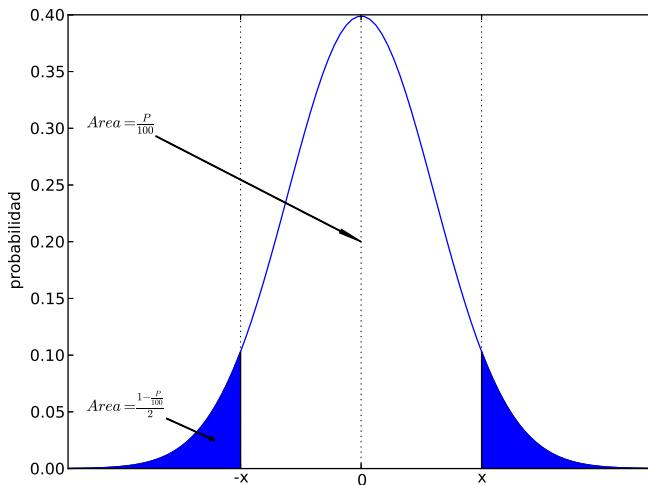


Figura 9.12: Intervalo de probabilidad P %

```

>> sigma = 2; %desviacion de las medidas
>> p = (1-0.95)/2 % probabilidad acumulada hasta limite inferior del intervalo
p =
    0.0250
>> i95 = norminv(p) % limite inferior del intervalo
i95 =
   -1.9600
>> incert = i95*sigma %limite inferior de la incertidumbre
incert =
   -3.9199

```

Por tanto, en este ejemplo, debemos expresar la medida como $6,5 \pm 3,92$ °C indicando que el intervalo de confianza es del 95 %.

La distribución T de Student. Habitualmente, cuando el número de medidas que se han tomado para estimar el valor de \bar{x}_n es pequeño, los intervalos de confianza se calculan empleando la distribución t de Student de $n-1$ grados de libertad, donde n representa el número de medidas tomadas.

Para un número pequeño de muestras, la distribución t de Student da una aproximación mejor que la distribución normal. Para un número de muestras grande, ambas distribuciones son prácticamente iguales. La figura 9.13 muestra una comparación entre la distribución normal y la distribución t de Student, calculada para distintos grados de libertad. De la figura se puede concluir en primer lugar que la desviación estándar de la distribución t de Student es en todos los casos mayor que la de la distribución normal, con lo cual los intervalos de confianza para un valor dado de la probabilidad son siempre mayores. En segundo lugar se observa también que para 30 o más medidas — $n \geq 30$ — la diferencia entre las distribuciones t de Student y normal es prácticamente despreciable.

El procedimiento de cálculo es el mismo que si empleamos la distribución normal, basta sustituir

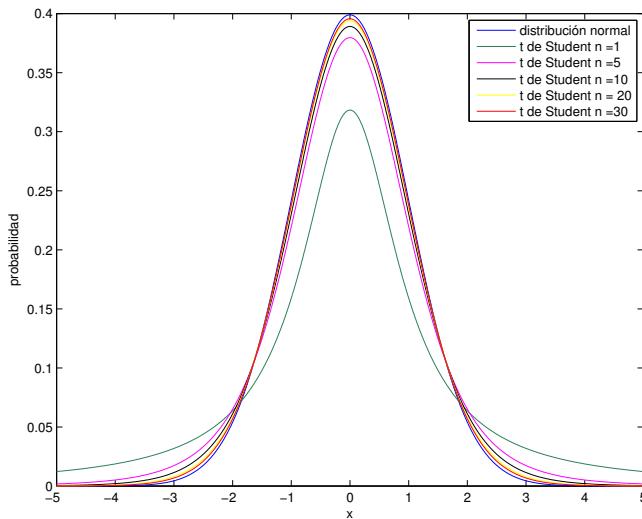


Figura 9.13: Comparación entre las distribuciones t de Student de 1, 5, 10, 20 y 30 grados de libertad y la distribución normal.

la inversa de la distribución normal acumulada por la inversa de la distribución t de Student acumulada. En Matlab, esta función está definida como: $x = \text{tinv}(p, n)$.

Así, para el ejemplo anterior de las medidas de temperatura de media $\bar{x}_n = 6,5^{\circ}\text{C}$ y desviación estándar $s_{xn} = 2$ si suponemos que el número de medidas tomadas es $n = 5$. El intervalo de confianza correspondiente a un 95 % lo calcularíamos mediante la inversa de la distribución de probabilidad t de Student de $n - 1$ grados de libertad acumulada como,

```
>> xm = 6; %media de las cinco medidas
>> sigma = 2; %desviacion de las cinco medidas
>> p = (1-0.95)/2 % probabilidad acumulada hasta limite inferior del intervalo
p =
    0.0250
>> i95 = tinv(p,4) % limite inferior del intervalo 4 = 5-1 grados de libertad para cinco medidas
i95 =
   -2.7764
>> incert = i95*sigma %limite inferior de la incertidumbre
incert =
   -5.5529
```

Por tanto deberíamos expresar el valor de la medida realizada como, $6,5 \pm 5,55^{\circ}\text{C}$. Si comparamos este resultado con el obtenido empleando la distribución normal, vemos que el intervalo de confianza y, por tanto, la incertidumbre han aumentado.

9.4.3. Propagación de la incertidumbre: Estimación de la incertidumbre de medidas indirectas.

Supongamos que deseamos obtener la incertidumbre de una determinada magnitud física y que no hemos medido directamente, sino que se determina a partir de otras cantidades x_1, x_2, \dots, x_n

empleando una relación funcional f ,

$$y = f(x_1, x_2, \dots, x_n)$$

La función f suele recibir el nombre de ecuación de medida, y no tiene por qué representar simplemente una ley física; de hecho, debería incluir entre sus entradas cualquier cantidad que pueda contribuir a modificar significativamente la incertidumbre de y . Por ejemplo: diferentes observadores, laboratorios, experimentos, horas a las que se han realizado las observaciones, etc.

La incertidumbre asociada al valor de y , a la que llamaremos $u_c(y)$, se expresa en función de la desviación estándar de y ,

$$u_c(y) = \sqrt{s_c^2(y)}$$

donde $s_c^2(y)$ se define como la varianza *combinada* de y , que se estima en primera aproximación a partir de la expansión en serie de Taylor⁶ de la función f

$$s_c^2(y) = \sum_{i=1}^n \left(\frac{\partial f}{\partial x_i} \right)^2 s^2(x_i) + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j} c_v(x_i, x_j)$$

Donde $s^2(x_i)$ representa la varianza asociada a la entrada x_i y $c_v(x_i, x_j)$ la covarianza entre las entradas x_i y x_j .

La covarianza entre dos variables z y t de las que se tienen n muestras, puede estimarse a partir de las muestras como,

$$c_v(z, t) = \frac{1}{n-1} \sum_{i=1}^n (z_i - \bar{z})(t_i - \bar{t})$$

De modo análogo a como hicimos en el caso de la varianza de la media de un conjunto de medidas, podemos relacionar la covarianza entre las medias de dos variables, con la covarianza entre n medidas de dichas variables como,

$$c_v(\bar{z}, \bar{t}) = \frac{c_v(z, t)}{n}$$

En muchos casos de interés no existe correlación entre las variables de entrada. En estos casos, los valores de la covarianza será nulos y podemos simplificar la ecuación empleada en el cálculo de $s_c^2(y)$,

$$s_c^2(y) = \sum_{i=1}^n \left(\frac{\partial f}{\partial x_i} \right)^2 s^2(x_i)$$

Un ejemplo simple lo podemos obtener de la ecuación para la potencia disipada en una resistencia eléctrica. En este caso, el valor de la potencia disipada depende del voltaje, V , de la resistencia R_o medida a una temperatura de referencia t_0 , de la temperatura real a que se encuentra la resistencia t y, en primera aproximación, de un coeficiente que b , que establece una relación lineal entre temperatura y resistencia.

$$P = f(V, R_o, b, t) = \frac{V^2}{R_o (1 + b(t - t_0))}$$

⁶En primera aproximación empleamos un desarrollo de Taylor de primer orden. En realidad, lo adecuado o no de este método depende de las características de la función f . No discutiremos aquí este problema.

Supongamos que b es un coeficiente numérico no sujeto a incertidumbre y que las variables V , R_0 , t y t_0 no están correlacionadas entre sí. Podemos entonces expresar la varianza en la potencia disipada, a partir de las varianzas de las variables de entrada como,

$$s_c^2(P) = \left(\frac{\partial f}{\partial V} \right)^2 s^2(V) + \left(\frac{\partial f}{\partial R_0} \right)^2 s^2(R_0) + \left(\frac{\partial f}{\partial t} \right)^2 s^2(t) + \left(\frac{\partial f}{\partial t_0} \right)^2 s^2(t_0)$$

Por tanto,

$$\begin{aligned} s_c^2(P) &= \left(\frac{2V}{R_0(1+b(t-t_0))} \right)^2 s^2(V) + \left(\frac{-V^2(1+b(t-t_0))}{R_0^2(1+b(t-t_0))^2} \right)^2 s^2(R_0) + \\ &\quad + \left(\frac{-V^2R_0b}{R_0^2(1+b(t-t_0))^2} \right)^2 s^2(t) + \left(\frac{V^2R_0b}{R_0^2(1+b(t-t_0))^2} \right)^2 s^2(t_0) \end{aligned}$$

y la incertidumbre en la potencia disipada será la raíz cuadrada de la expresión que acabamos de obtener: $u_c(P) = \sqrt{s_c^2(P)}$

Establecer los intervalos de confianza, para una incertidumbre propagada no es tan sencillo como en el caso de una incertidumbre obtenida directamente de una medida experimental. En primera aproximación, podemos suponer que y sigue una distribución normal de media el valor obtenido, a partir de la función f y desviación $u_c(p)$ y estimar los intervalos de confianza del mismo modo que se describió anteriormente para una medida directa. Pero esto no es siempre correcto. No podemos establecer en realidad ninguna ley general que relacione las distribuciones de probabilidad de las incertidumbres de las variables de entrada con la distribución de probabilidad de la incertidumbre de la variable de salida.

9.4.4. Ejemplo de estimación de la incertidumbre con Matlab.

Para aclarar los métodos descritos en los apartados anteriores, vamos obtener la incertidumbre de la potencia disipada en una resistencia, siguiendo la ecuación descrita en el apartado anterior. La tabla 9.1 muestra los datos obtenidos tras realizar 20 medidas experimentales de la tensión y la temperatura en una resistencia.

Para obtener la incertidumbre en el valor de la potencia disipada por la resistencia debemos en primer lugar calcular la varianza de las medidas directas suministradas en la tabla.

- Varianza del valor nominal de la resistencia R_0 : En este caso, la tabla nos suministra como dato el intervalo de confianza correspondiente al 98 %. Podemos emplear la función de Matlab `norminv(x)` para calcular la desviación estándar de R_0 . Como hemos visto, el límite inferior del intervalo de confianza del 98 %, corresponde con una probabilidad acumulada de $(1 - 0,98)/2$,

```
>> i98 = norminv((1-0.98)/2)
i98 =
-2.3263
```

Dicha cantidad, multiplicada por la desviación estándar de R_0 , corresponde con el límite inferior del intervalo de confianza suministrado en la tabla para R_0 . Por tanto,

```
>> sR0 = -5/i98 %desviación estandar de R0
sR0 =
2.1493
```

Cuadro 9.1: Mediciones de temperatura y Voltaje, sobre una resistencia de prueba de 100Ω

Valor nominal de la resistencia	$R_0 = 100 \pm 5\Omega$ (intervalo confianza 98 %)
Temperatura de referencia	$t_0 = 50 \pm 0,1^\circ C$ (división menor del sensor $0,2^\circ C$)
Valor del parámetro b	$b = 0,4\Omega/\text{ }^\circ C$ (incertidumbre despreciable)
Temperatura $^\circ C$	Votaje mV
55,7	761
62,8	897
58,1	879
60,9	767
66,3	758
59,4	763
61,8	762
62,5	761
59,4	790
61,4	852
55,7	816
65,3	752
56,9	848
57,8	920
62,9	743
59,9	858
62,6	761
59,1	788
57,9	775
53,8	791

```
>> s2R0 = sR0^2 %varianza de R0
s2R0 =
4.6195
```

luego $s^2(R_0) = 4,6195$

- Varianza de la temperatura de referencia t_0 : En este caso se ha expresado la incertidumbre en función de la mitad de la división más pequeña de la escala del aparato de medida. Podemos considerar por tanto que la incertidumbre estaría representada por una distribución uniforme definida en el intervalo $[-0,1, 0,1]$. En esta caso, podemos asociar la varianza de la medida directamente con dicha distribución uniforme,

```
>> s2t0=(1-(-1))^2/12
s2t0 =
0.3333
```

Así, $s^2(t_0) = 0,3333$

- Varianzas de las medidas de temperatura y voltaje: En primer lugar promediamos los valores obtenidos de voltaje y temperatura, para calcular el valor de la medida. Matlab tiene definida la función `mean(x)` que permite obtener directamente el valor medio de los elementos de `x`. Podemos por tanto definir en Matlab dos vectores, unos para los datos de temperatura y otro para los datos de voltaje y calcular directamente las medias,

```

>> t=[55.7 62.8 58.1 60.9 66.3 59.4 61.8 62.5 59.4 61.4 55.7 65.3 56.9 57.8 62.9...
59.9 62.6 59.1 57.9 53.8]

t =
    Columns 1 through 6
    55.7000    62.8000    58.1000    60.9000    66.3000    59.4000
    Columns 7 through 12
    61.8000    62.5000    59.4000    61.4000    55.7000    65.3000
    Columns 13 through 18
    56.9000    57.8000    62.9000    59.9000    62.6000    59.1000
    Columns 19 through 20
    57.9000    53.8000

>> v=[761 897 879 767 758 763 762 761 790 852 816 752 848 920 743 858 761...
788 775 791]

v =
    Columns 1 through 10
    761    897    879    767    758    763    762    761    790    852
    Columns 11 through 20
    816    752    848    920    743    858    761    788    775    791

>> tm =mean(t)
tm =
    60.0100

>> vm =mean(v)
vm =
    802.1000

```

Una vez estimadas las medias, que emplearemos como valores de las medidas de la temperatura y el voltaje, podemos ahora estimar la varianza o la desviación estándar de las medias haciendo uso de las función de Matlab `std(x)`, para la desviación estandar, y de `var(x)`, para la varianza. Además, debemos aplicar el teorema central del límite, dividiendo por el número de datos disponible.

```

>> s2t=var(t)/length(t) %varianza de la temperatura media
s2t =
    0.5328
>> s2t=var(v)/length(v) %varianza del voltaje medio

s2t =
    145.5837
>> st=std(v)/sqrt(length(v)) %desviacion del voltaje medio
st =
    12.0658

>> st=std(t)/sqrt(length(t)) %desviacion de la temperatura media

```

```
st =
0.7300
```

Lógicamente, los resultados muestran que las desviaciones estándar son las raíces cuadradas de las varianzas. Tenemos por tanto, $\bar{t} = 60,01$, $\bar{V} = 802,01$, $s^2(t) = 0,5382$, $s^2(V) = 145,5837$.

En este momento, hemos reunido ya toda la información necesaria para estimar la incertidumbre de la potencia consumida en la resistencia de nuestro ejemplo. La tabla 9.2 contiene los datos calculados,

Variable (unidades)	Valor	Varianza
$R_0(\Omega)$	100	4,6195
b ($\Omega/{}^\circ\text{C}$)	0,4	–
t_0 (${}^\circ\text{C}$)	50	0,3333
\bar{t} (${}^\circ\text{C}$)	60,01	0,5328
$V(\text{mV})$	802,01	145,5837

Podemos ahora calcular la potencia,

$$P = \frac{V^2}{R_0(1 + b(t - t_0))} = \frac{0,80201^2}{100(1 + 0,4(60,01 - 50))} = 0,001285W = 1,285\text{mW}$$

Donde hemos introducido el voltaje en voltios, para obtener la potencia directamente en vatios. A continuación, propagamos la incertidumbre de las variables de entrada a la de salida,

$$\begin{aligned} s_c^2(P) &= \left(\frac{2V}{R_0(1 + b(t - t_0))} \right)^2 s^2(V) + \left(\frac{-V^2(1 + b(t - t_0))}{R_0^2(1 + b(t - t_0))^2} \right)^2 s^2(R_0) + \\ &\quad + \left(\frac{-V^2 R_0 b}{R_0^2(1 + b(t - t_0))^2} \right)^2 s^2(t) + \left(\frac{V^2 R_0 b}{R_0^2(1 + b(t - t_0))^2} \right)^2 s^2(t_0) = \\ &= \left(\frac{2 \cdot 0,80201}{100(1 + 0,4(60,01 - 50))} \right)^2 \cdot 1,455836 \cdot 10^{-4} + \left(\frac{-0,80201^2(1 + 0,4(60,01 - 50))}{100^2(1 + 0,4(60,01 - 50))^2} \right)^2 \cdot 4,619 + \\ &\quad + \left(\frac{-0,80201^2 \cdot 100 \cdot 0,4}{100^2(1 + 0,4(60,01 - 50))^2} \right)^2 \cdot 0,5328 + \left(\frac{0,80201^2 \cdot 100 \cdot 0,4}{100^2(1 + 0,4(60,01 - 50))^2} \right)^2 0,3333 = \\ &= 1,4959 \cdot 10^{-9} + 7,6327 \cdot 10^{-10} + 5,6252 \cdot 10^{-9} + 3,5189 \cdot 10^{-9} = 1,1403 \cdot 10^{-8} \end{aligned}$$

En este caso, hemos multiplicado la varianza del voltaje $s^2(V)$ por 10^{-6} , para obtener $s_c^2(P)$ en W^2 . Podemos expresar ahora la incertidumbre de la potencia consumida en función de la desviación estándar como,

$$u_c(P) = \sqrt{s_c^2(P)} = 1,0678 \cdot 10^{-4}\text{W}$$

Con lo que finalmente expresaríamos la medida indirecta de la potencia consumida como, $P = 1,285 \pm 0,10678 \text{ mW}$. Si suponemos que la distribución de probabilidad asociada a la incertidumbre de la potencia es normal, la incertidumbre así expresada (mediante la desviación estándar) representaría un intervalo de confianza del 68,27%.

Capítulo 10

Introducción al cálculo simbólico

En este último capítulo vamos a introducir un método de cálculo que difiere notablemente de lo visto en los capítulos anteriores: se trata del cálculo simbólico. Hasta ahora, hemos empleado siempre el ordenador para hacer cálculo numérico. Las variables se definían asignándoles un valor numérico y después eran empleadas en operaciones algebraicas o se les aplicaban funciones matemáticas para obtener a partir de ellas resultados numéricos.

Sin embargo, en matemáticas y en física, es práctica habitual realizar operaciones con variables y funciones sin asignarles un resultado numérico. Un ejemplo típico es el de la derivación de una función,

$$f(x) = \sqrt{1 - \ln x} \rightarrow f'(x) = -\frac{1}{2x\sqrt{1 - \ln x}}$$

A partir de una expresión *simbólica* de la función obtenemos su derivada expresada también de modo *simbólico* como otra función. También con un computador es posible hacer este tipo de operaciones, a las que se da el nombre de cálculo simbólico para distinguirlas del cálculo numérico.

10.1. Cálculo simbólico en el entorno de Matlab

Como se ha mostrado en capítulos anteriores, la funcionalidad que aportan los comandos básicos de Matlab puede ser ampliada, por parte del usuario, mediante la construcción de funciones y *scripts*. Matlab por su parte, tiene un amplio conjunto de funciones para aplicaciones específicas agrupadas en las llamadas *toolboxes*. Así por ejemplo, hay *toolboxes* específicas para tratamiento de datos, procesamiento de señal, ecuaciones diferenciales en derivadas parciales y cálculo simbólico entre otras.

Las funciones de una *toolbox* no se distinguen, en cuanto a su uso, de las funciones básicas de Matlab y su sintaxis es la misma. Lo único necesario es que la *toolbox* en cuestión esté instalada en Matlab. Para saber qué *toolboxes* de Matlab están instaladas en un ordenador concreto basta introducir en la ventana de comandos de Matlab el comando *ver*.

```
>> ver
-----
MATLAB Version: 8.5.0.197613 (R2015a)
MATLAB License Number: 161052
Operating System: Microsoft Windows 10 Home Version 10.0 (Build 10586)
Java Version: Java 1.7.0_60-b19 with Oracle Corporation
```

Java HotSpot(TM) 64-Bit Server VM mixed mode

MATLAB	Version 8.5	(R2015a)
Simulink	Version 8.5	(R2015a)
Bioinformatics Toolbox	Version 4.5.1	(R2015a)
Control System Toolbox	Version 9.9	(R2015a)
Curve Fitting Toolbox	Version 3.5.1	(R2015a)
Database Toolbox	Version 5.2.1	(R2015a)
Datafeed Toolbox	Version 5.1	(R2015a)
Fixed-Point Designer	Version 5.0	(R2015a)
Fuzzy Logic Toolbox	Version 2.2.21	(R2015a)
Global Optimization Toolbox	Version 3.3.1	(R2015a)
Image Acquisition Toolbox	Version 4.9	(R2015a)
Image Processing Toolbox	Version 9.2	(R2015a)
MATLAB Coder	Version 2.8	(R2015a)
MATLAB Compiler	Version 6.0	(R2015a)
MATLAB Compiler SDK	Version 6.0	(R2015a)
MATLAB Distributed Computing Server	Version 6.6	(R2015a)
MATLAB Report Generator	Version 4.1	(R2015a)
Mapping Toolbox	Version 4.1	(R2015a)
Model Predictive Control Toolbox	Version 5.0.1	(R2015a)
Model-Based Calibration Toolbox	Version 4.8.1	(R2015a)
Neural Network Toolbox	Version 8.3	(R2015a)
OPC Toolbox	Version 3.3.3	(R2015a)
Optimization Toolbox	Version 7.2	(R2015a)
Parallel Computing Toolbox	Version 6.6	(R2015a)
Partial Differential Equation Toolbox	Version 2.0	(R2015a)
Robotics Toolbox	Version 8	December
Robust Control Toolbox	Version 5.3	(R2015a)
Signal Processing Toolbox	Version 7.0	(R2015a)
Statistics and Machine Learning Toolbox	Version 10.0	(R2015a)
Symbolic Math Toolbox	Version 6.2	(R2015a)
System Identification Toolbox	Version 9.2	(R2015a)
Wavelet Toolbox	Version 4.14.1	(R2015a)

Este comando muestra la versión de Matlab así como las *toolboxes* instaladas. Para poder realizar cálculo simbólico necesitamos tener **Symbolic Math Toolbox**, marcada en rojo en la relación de Toolboxes que se acaba de mostrar.

10.2. Variables y expresiones simbólicas

10.2.1. Variables simbólicas

Una variable simbólica, a diferencia de las variables ordinarias de Matlab, no contiene un valor; simplemente es un objeto que puede manipularse empleando reglas algebraicas ordinarias. Para crear una variable simbólica, se emplea el comando `syms`,

```
>> syms x
>> x
```

```
x =
x
```

Cuando pedimos a Matlab que nos muestre el contenido de la variable `x` nos indica que contiene el símbolo `x`. Es importante remarcar que aquí no se trata de un carácter sino de un símbolo con el que se pueden realizar operaciones algebraicas. Así, por ejemplo,

```
>> syms a b c
>> c = a - b + a + b
c =
2*a
```

Las tres variables `a`, `b` y `c` admiten operaciones aritméticas, cuyo resultado es simbólico, `2*a`, en lugar de numérico.

Una forma alternativa de crear variables simbólicas es empleando la función `sym`.

```
>> a = sym('b')
a =
b

>> sqrt(a)
ans =
b^(1/2)
```

En este caso el nombre de la variable y el símbolo no tienen porqué coincidir y las variables hay que crearlas de una en una. La función `sym`, nos permite también definir números 'simbólicos', que Matlab maneja de modo análogo a cualquier otro símbolo,

```
>> dos = sym('2')
dos =
2

>> sqrt(dos)
ans =
2^(1/2)
```

También es posible crear vectores y matrices simbólicas a partir de otras variables simbólicas,

```
>> syms a b c d

>> A = [a b c; d a b; c c b]
A =
[ a, b, c]
[ d, a, b]
[ c, c, b]
```

o bien, lo que suele ser más frecuente, indicando la dimensión de la matriz que deseamos crear,

```
>> A = sym('a',[3,3])
A =
[ a1_1, a1_2, a1_3]
[ a2_1, a2_2, a2_3]
[ a3_1, a3_2, a3_3]
```

Es posible elegir el formato de los elementos simbólicos empleando comandos parecidos a los que emplea la función `fprintf` (ver sección 2.2.1),

```
>> A = sym('a%d%d',[3,3])
A =
[ a11, a12, a13]
[ a21, a22, a23]
[ a31, a32, a33]
```

Las operaciones que realicemos con matrices simbólicas darán también resultados simbólicos,

```
>> b = sym('b%d',[3,1])
b =
b1
b2
b3

>> c = A*b
c =
a11*b1 + a12*b2 + a13*b3
a21*b1 + a22*b2 + a23*b3
a31*b1 + a32*b2 + a33*b3
```

Por último, el comando `sym` permite también convertir una expresión numérica en su equivalente simbólica,

```
>> a = [0.1 0.3 0.22
2.4 3 3.3
2 1 -1/3]
a =
0.1000    0.3000    0.2200
2.4000    3.0000    3.3000
2.0000    1.0000   -0.3333

>> A = sym(a)
A =
[ 1/10, 3/10, 11/50]
[ 12/5,     3, 33/10]
[    2,     1, -1/3]
```

10.2.2. Expresiones simbólicas

Es posible combinar una o más variables simbólicas con los operadores matemáticos de Matlab para obtener expresiones simbólicas,

```
>> syms a b c x
>> y = a*x^2+b*x+c
y =
a*x^2 + b*x + c

>> rp = (-b+(b^2-4*a*c)^(1/2))/(2*a)
rp =
```

```

-(b - (b^2 - 4*a*c)^(1/2))/(2*a)

>> rm = (-b-(b^2-4*a*c)^(1/2))/(2*a)
rm =
-(b + (b^2 - 4*a*c)^(1/2))/(2*a)

```

En este ejemplo, se han creado primero cuatro variables simbólicas y, a partir de ellas, se han generado tres expresiones simbólicas, la primera es un polinomio de grado dos y las dos siguientes las expresiones analíticas de sus raíces.

También es posible crear funciones simbólicas a partir de variables simbólicas, expresiones simbólicas o funciones matemáticas. Hay varias maneras de generarlas, pero quizás la más sencilla, es usar variables simbólicas previamente definidas,

```

>> syms x
f(x) = sin(x)
f(x) =
sin(x)

```

Las variables de la función se indican entre paréntesis a continuación del nombre de la función y antes del símbolo '='. Si hay varias (posibles) variables y no se indica expresamente, Matlab tomará como variable de la función aquella que alfabéticamente esté más cerca de la letra *x*.

Es posible definir funciones de más de una variable,

```

>> syms x y
>> g(x,y)=sin(sqrt(x^2+y^2))
g(x, y) =
sin((x^2 + y^2)^(1/2))

```

O también, combinar funciones simbólicas mediante operadores aritméticos o composición de funciones,

```

>> h(x,y) = f(x) - g(x,y)
h(x, y) =
sin(x) - sin((x^2 + y^2)^(1/2))

>> t(x) = g(x,y) ^f(x)
t(x) =
sin((x^2 + y^2)^(1/2))^sin(x)

>> r(x,y) = f(g(x,y))
r(x, y) =
sin(sin((x^2 + y^2)^(1/2)))

```

10.2.3. Simplificación de expresiones simbólicas

Matlab permite, dada una expresión simbólica, obtener una equivalente más sencilla. En general, este proceso de simplificación no es trivial ni unívoco, ya que una expresión puede admitir varias formas simplificadas. Es también posible que la expresión simbólica que deseamos simplificar, no sea simplificable.

Es interesante, sobretodo cuando se trata de expresiones obtenidas a partir de cálculos simbólicos, tratar de obtener una expresión lo más simplificada posible. La razón es que resulta mucho más fácil de entender y manipular. Matlab suministra el comando **simplify** para simplificar expresiones simbólicas. Así por ejemplo, si construimos la función simbólica,

```
>> f(x) = sin(x)^2 + cos(x)^2
f(x) =
cos(x)^2 + sin(x)^2
```

y le aplicamos el comando **simplify**,

```
>> simplify(f)
ans(x) =
1
```

Obtendremos un 1, como cabría esperar. Este ejemplo es particularmente simple y, aunque sirve para ilustrar el funcionamiento del comando, no siempre es posible conseguir soluciones tan satisfactorias.

Veamos otro ejemplo que muestra la ambigüedad del comando **simplify**,

```
>> syms x
>> p1 = 3*x^2+2*x-1
p1 =
3*x^2 + 2*x - 1
>> p1 = x^2-2*x+1
p1 =
x^2 - 2*x + 1
>> simplify(p1)
ans =
(x - 1)^2
```

Matlab simplifica el polinomio de grado dos suministrado convirtiéndolo en un binomio. La pregunta que cabe hacerse es en qué medida podemos considerar el binomio una expresión simplificada del polinomio del que procede. Así por ejemplo, si estamos sumando polinomios, la expresión desarrollada resulta más *simple* que la expresión binomial.

10.2.4. Sustitución de variables por valores numéricos

Es posible, una vez definida una expresión simbólica, sustituir sus variables por valores numéricos mediante el uso del comando **subs**,

```
>> p = sin(x)^2 - 2*x
p =
sin(x)^2 - 2*x
>> v = subs(p,2)
v =
sin(2)^2 - 4
```

La variable x ha sido sustituida, en la expresión simbólica p , por el número 2. La expresión resultante sigue siendo todavía simbólica. Si lo que deseamos es obtener el valor numérico de la función p en el punto 2, tenemos que convertir la expresión simbólica resultante en su correspondiente valor numérico. Para ello se emplea el comando `double`,

```
>> double(v)
ans =
-3.1732
```

Este comando toma su nombre del tipo de formato con que el ordenador va a representar el resultado; un número real guardado en doble precisión¹

Otro comando de interés para pasar de simbólico a numérico es el comando `vpa` (*variable precision arithmetic*).

```
>> vpa(v)
ans =
-3.1731781895681940426804159084511
```

El número de dígitos obtenidos al hacer la conversión es, por defecto, 32. Dicho número puede ajustarse al valor que se desee introduciéndolo en `vpa` como un segundo parámetro,

```
>> vpa(v,5)
ans =
-3.1732
```

Si una expresión engloba más de una variable simbólica es posible sustituir solo alguna o algunas de ellas, para ello basta indicar el nombre de la variable que se desea sustituir seguida de su valor,

```
>> syms x y z

>> f = x^2*y + x * y^2 + z*x*y
f =
x^2*y + x*y^2 + z*x*y

>> fx = subs(f,x,2)
fx =
4*y + 2*y*z + 2*y^2

>> fy = subs(f,y,3)
fy =
9*x + 3*x*z + 3*x^2

>> fxz = subs(f,2,4)
fxz =
x^4*y + x*y^4 + z*x*y
```

Además, es posible sustituir una variable simbólica por otra,

```
>> fxyx = subs(f,z,x)
fxyx =
```

¹En realidad Matlab convierte los resultados a números reales o complejos de acuerdo con la expresión simbólica de la que se han obtenido.

```

2*x^2*y + x*y^2

>> f
f =
x^2*y + x*y^2 + z*x*y

>> fxxx = subs(fxyx,y,x)
fxxx =
3*x^3

```

10.3. Cálculo infinitesimal

Es posible realizar operaciones típicas del cálculo infinitesimal como derivación e integración, empleando métodos de cálculo simbólico. A diferencia del cálculo numérico, el resultado será ahora una expresión.

10.3.1. Derivación

Para obtener la derivada de una función simbólica, se emplea el comando **diff**,

```

>> syms x a
>> f = a*exp(-x^2)
f =
a*exp(-x^2)
>> df = diff(f)
df =
-2*a*x*exp(-x^2)

```

Matlab devuelve como resultado la derivada con respecto a la variable **x** de la función simbólica introducida. (Quizá sea útil recordar que Matlab toma como variable de la función la variable simbólica mas próxima alfabéticamente a la letra **x**.)

Si deseamos derivar una función respecto a una variable simbólica específica, es preciso indicarlo expresamente, introduciendo dicha variable en la llamada al comando **diff** a continuación de la función que vamos a derivar y separada de ésta por una coma. Así lo indicamos si queremos derivar la función del ejemplo anterior respecto a la variable simbólica **a**,

```

>> dfa = diff(f,a)
dfa =
exp(-x^2)

```

Es posible emplear el comando **diff** para obtener derivadas de orden superior. Para ello se introduce en la llamada el orden de la derivada a continuación del nombre de la función o, si es el caso, a continuación de la variable simbólica con respecto a la que se quiere derivar,

```

>> df3 = diff(f,3)
df3 =
12*a*x*exp(-x^2) - 8*a*x^3*exp(-x^2)

>> dfa2 = diff(f,a,2)
dfa2 =
0

```

En el primer caso hemos obtenido la derivada tercera con respecto a x de la función f de los ejemplos anteriores y en el segundo la derivada segunda de dicha función con respecto a la variable a .

Es fácil comprobar que el resultado sería el mismo si derivamos la función original y las expresiones de las sucesivas derivadas obtenidas hasta alcanzar la derivada del orden deseado,

```
>> df = diff(f)
df =
-2*a*x*exp(-x^2)
>> df2 = diff(df)
df2 =
4*a*x^2*exp(-x^2) - 2*a*exp(-x^2)

>> df3 = diff(df2)
df3 =
12*a*x*exp(-x^2) - 8*a*x^3*exp(-x^2)
```

En análisis matemático, cuando una función posee más de una variable independiente, la derivación respecto a una de sus variables recibe el nombre de derivada parcial. Así por ejemplo la función, $f(x, y) = \sin(\sqrt{x^2 + y^2})$ es una función de dos variables x e y . Su derivada (parcial) con respecto a x y su derivada parcial con respecto a y , se representan como,

$$\frac{\partial f}{\partial x} = x \cdot \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}}$$

$$\frac{\partial f}{\partial y} = y \cdot \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}}$$

Podemos obtener dichas derivadas parciales, indicando al comando **diff** respecto a qué variable debe derivar, tal y como se ha mostrado anteriormente, siendo en nuestro ejemplo con respecto a x e y ,

```
>> syms x y
>> fxy = sin(sqrt(x^2+y^2))
fxy =
sin((x^2 + y^2)^(1/2))

>> sfx = diff(fxy,x)
sfx =
(x*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(1/2)

>> sfy = diff(fxy,y)
sfy =
(y*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(1/2)
```

Las derivadas parciales de orden superior pueden calcularse respecto a la misma o a distintas variables; por ejemplo para las derivadas segundas,

$$\begin{aligned}\frac{\partial^2 f}{\partial x^2} &= \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} - x^2 \cdot \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{(x^2 + y^2)^3}} - x^2 \cdot \frac{\sin(\sqrt{x^2 + y^2})}{x^2 + y^2} \\ \frac{\partial^2 f}{\partial y^2} &= \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}} - y^2 \cdot \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{(x^2 + y^2)^3}} - y^2 \cdot \frac{\sin(\sqrt{x^2 + y^2})}{x^2 + y^2} \\ \frac{\partial^2 f}{\partial x \partial y} &= \frac{\partial^2 f}{\partial y \partial x} = -x \cdot y \cdot \frac{\cos(\sqrt{x^2 + y^2})}{\sqrt{(x^2 + y^2)^3}} - x \cdot y \cdot \frac{\sin(\sqrt{x^2 + y^2})}{x^2 + y^2}\end{aligned}$$

Las derivadas calculadas respecto a variables distintas se llaman derivadas cruzadas, y no influye en ellas el orden en que se realice la derivación.

Para calcular las derivadas parciales de orden superior de una función, es suficiente indicar las variables con respecto a las cuales se desea derivar o, en el caso de que se calculen derivadas respecto a la misma variable, se puede también indicar el orden. En los siguientes ejemplos de código se muestra como obtener las derivadas segundas de la función que hemos venido utilizando a lo largo de esta sección empleando ambos métodos,

```
>> sfx2 = diff(fxy,x,x)
sfx2 =
cos((x^2 + y^2)^(1/2))/(x^2 + y^2)^(1/2)
-(x^2*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(3/2)
-(x^2*sin((x^2 + y^2)^(1/2)))/(x^2 + y^2)

>> sfx2 = diff(fxy,x,2)
sfx2 =
cos((x^2 + y^2)^(1/2))/(x^2 + y^2)^(1/2)
-(x^2*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(3/2)
-(x^2*sin((x^2 + y^2)^(1/2)))/(x^2 + y^2)

>> sfy2 = diff(fxy,y,y)
sfy2 =
cos((x^2 + y^2)^(1/2))/(x^2 + y^2)^(1/2)
-(y^2*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(3/2)
-(y^2*sin((x^2 + y^2)^(1/2)))/(x^2 + y^2)

>> sfy2 = diff(fxy,y,2)
sfy2 =
cos((x^2 + y^2)^(1/2))/(x^2 + y^2)^(1/2)
-(y^2*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(3/2)
-(y^2*sin((x^2 + y^2)^(1/2)))/(x^2 + y^2)

>> sfxy = diff(fxy,x,y)
sfxy =
-(x*y*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(3/2)
-(x*y*sin((x^2 + y^2)^(1/2)))/(x^2 + y^2)

>> sfyx = diff(fxy,y,x)
sfyx =
-(x*y*cos((x^2 + y^2)^(1/2)))/(x^2 + y^2)^(3/2)
-(x*y*sin((x^2 + y^2)^(1/2)))/(x^2 + y^2)
```

Es posible emplear el comando de Matlab `pretty` para obtener una representación de una expresión simbólica más fácil de visualizar. Si la aplicamos al último de los resultados obtenidos en el ejemplo anterior obtenemos,

```
>> pretty(sfyx)
      2      2                  2      2
x y cos(sqrt(x + y ))  x y sin(sqrt(x + y ))
-----
      2      2 3/2                  2      2
(x + y )                x + y
```

Es importante hacer notar que este comando no altera en modo alguno el valor de la expresión simbólica, simplemente la muestra por pantalla en un formato distinto.

Es posible emplear Matlab para realizar cálculos de gradientes, jacobianos, rotacionales, etc. No se incluye su descripción en este manual, los interesados pueden consultar la ayuda de Matlab.

10.3.2. Integración

La integración indefinida en cálculo simbólico permite obtener la primitiva de una función dada,

$$F(x) = \int f(x)dx \Leftrightarrow f(x) = \frac{dF(x)}{dx}$$

Se trata de una operación más compleja que la derivación. De hecho, hay funciones para las que no es posible obtener una primitiva en forma simbólica, bien porque no existe, o bien porque los algoritmos empleados por Matlab no son capaces de obtenerla.

El comando empleado para obtener la integral de una función es `int`. Así por ejemplo, para obtener la primitiva de la función $f(x) = \sqrt{1 - x^2}$ haremos,

```
>> syms x
>> f = sqrt(1-x^2)
f =
(1 - x^2)^(1/2)

>> F = int(f)
F =
asin(x)/2 + (x*(1 - x^2)^(1/2))/2
```

Podemos comprobar el resultado obtenido aplicando la función `diff` vista en el apartado anterior,

```
>> diff(F)
ans =
1/(2*(1 - x^2)^(1/2)) - x^2/(2*(1 - x^2)^(1/2)) + (1 - x^2)^(1/2)/2

>> simplify(ans)
ans =
(1 - x^2)^(1/2)
```

Donde se ha hecho uso del comando `simplify` para recuperar la expresión original.

Como en otras funciones simbólicas, `int` calcula la integral con respecto a la variable simbólica más próxima a `x`, salvo que se indique expresamente respecto a qué variable simbólica se desea integrar. Así por ejemplo,

```
>> syms x a

>> f = a * x/(a^2 +x^2)
f =
(a*x)/(a^2 + x^2)

>> Fx = int(f)
Fx =
(a*log(a^2 + x^2))/2

>> Fx = int(f,x)
Fx =
(a*log(a^2 + x^2))/2

>> Fa = int(f,a)
Fa =
(x*log(a^2 + x^2))/2
```

en el primer y el segundo caso la integración se realiza con respecto a la variable x . En el tercer caso, la integración se realiza respecto a la variable a .

Cuando no es posible encontrar una primitiva a una función, Matlab devuelve como resultado la expresión `int(integrand)`, para indicar que no ha sido posible realizar la operación,

```
>> f = sin(sinh(x))
f =
sin(sinh(x))

>> F = int(f)
F =
int(sin(sinh(x)), x)
```

La función `int`, puede emplearse también para calcular integrales definidas, si se le suministran los límites de integración,

```
>> f = 10 * sin(x)
f =
10*sin(x)

>> F0pi = int(f,0,pi)
F0pi =
20
```

10.3.3. Series

Suma de series Matlab permite calcular varias operaciones sobre sucesiones de las que se conoce el término general. En este capítulo, solo nos referiremos a la suma de series. Para ello se aplica el comando `symsum` indicando los límites de la serie que se quiere sumar,

```
>> syms k

>> symsum(1/k^2,1,inf)
```

```

pi^2/6

>> symsum((a)^k,0,inf)
ans =
piecewise([1 <= a, Inf], [abs(a) < 1, -1/(a - 1)])

```

El segundo ejemplo muestra el límite de una serie geométrica, donde si $a > 1$ la serie no converge. Como siempre, Matlab toma por defecto como variable (índice) la variable simbólica más próxima a x .

Serie de Taylor Es posible calcular la expresión de la serie de Taylor para una función simbólica. Para ello se emplea el comando **taylor**. Por defecto se obtienen tan solo los elementos de la serie hasta orden 5, aunque es posible obtener órdenes superiores. A continuación se dan algunos ejemplos de uso,

```

>> syms x
>> f = exp(x)

f =

exp(x)

>> T = taylor(f)

T =

x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1

>> T2 = taylor(f,x,1)
T2 =
exp(1) + exp(1)*(x - 1) + (exp(1)*(x - 1)^2)/2 + (exp(1)*(x - 1)^3)/6 +
(exp(1)*(x - 1)^4)/24 + (exp(1)*(x - 1)^5)/120

```

Los ejemplos anteriores nos dan cinco términos del desarrollo de Taylor de la función exponencial. Primero en torno a $x = 0$, ya que no le hemos indicado un punto en torno al cual calcularlo. Después en torno a $x = 1$, valor definido en la llamada al comando **taylor** después del nombre de la función simbólica para la que queremos calcular el desarrollo.

Para calcular desarrollos de Taylor de otro orden distinto al quinto, debemos indicarlo expresamente, empleando en la llamada al comando **taylor** la palabra clave '**Order**', seguida del orden que deseamos para la expansión,

```

>> T2 = taylor(f,x,'Order',7)
T2 =
x^6/720 + x^5/120 + x^4/24 + x^3/6 + x^2/2 + x + 1

```

10.3.4. Límites

Para el cálculo de límites de una expresión simbólica se emplea el comando **limit**. Es necesario incluir la función simbólica y el valor de la variable independiente para el que se desea calcular el límite. Dicho valor puede ser también simbólico,

```
>> syms x a
>> f =x^2/(3*x^2+log(x))
f =
x^2/(log(x) + 3*x^2)
>> limit(f,0)
ans =
0
>> limit(f,a)
ans =
a^2/(log(a) + 3*a^2)
```

Si la expresión para la que se desea calcular el límite tiene más de una variable simbólica, Matlab calculará el límite empleando como variable la más cercana a x , excepto si se le indica expresamente con respecto a qué variable debe calcularlo. El siguiente ejemplo muestra como emplear el comando `limit` y la definición de derivada para calcular la derivada de una función,

```
>> syms h
>> f = log(x)

f =
log(x)

>> limit (log(x+h)/h ,0)
ans =
log(h)/h

>> limit (log(x+h)/h ,h,0)
ans =
limit(log(h + x)/h, h == 0)

>> limit ((log(x+h)-log(x))/h ,h,0)
ans =
1/x
```

Es posible calcular límites en el *infinito*

```
>> limit (1/x,x,inf)
ans =
0
```

Por último, es posible calcular el límite por la izquierda y el límite por la derecha cuando estos no coinciden. Para ello se añade al final del comando la palabra '`right`' (derecha) o '`left`' (izquierda)

```
>> limit (1/x,x,0,'right')
ans =
Inf

>> limit (1/x,x,0,'left')
ans =
-Inf

>> limit (1/x,x,0)
ans =
NaN
```

Es interesante notar que si no se especifica la dirección (left/right) al calcular este límite, el resultado es `NaN`; puesto que el límite, al no ser bilateral, no está bien definido.

10.4. Representación gráfica

Matlab dispone de un conjunto de funciones que permiten representar funciones simbólicas. Su funcionamiento es similar al de las funciones generales de Matlab para representación gráfica. A continuación se presenta una revisión de las más comunes. Para obtener la gráfica de una función de una sola variable se emplea el comando `ezplot`. Este comando admite como variables de entrada una función simbólica y un vector con los límites $[x_{min} \ x_{max}]$ entre los que se desea dibujar la función. Si no se suministran los límites, Matlab toma por defecto el intervalo $[-2\pi \ 2\pi]$ para realizar la representación. Así, por ejemplo para representar la función $f(x) = e^{\sin(x)}$,

```
>> syms x

>> f = exp(sin(x))
f =
exp(sin(x))

>> ezplot(f)

>> figure

>> ezplot(f, [-10 10])
```

El resultado se muestra en la figuras 10.1(a), 10.1(b).

La función `ezplot` puede emplearse también para representar curvas paramétricas, $x(t)$, $y(t)$. Por ejemplo, $x(t) = t \cos(t)$, $y(t) = t \sin(t)$ representa una espiral en el plano x , y . Para obtener su gráfica se debe pasar a `zplot` las funciones simbólicas $x(t)$ e $y(t)$,

```
>> syms t x y
>> x = t*cos(t)
x =
t*cos(t)
```

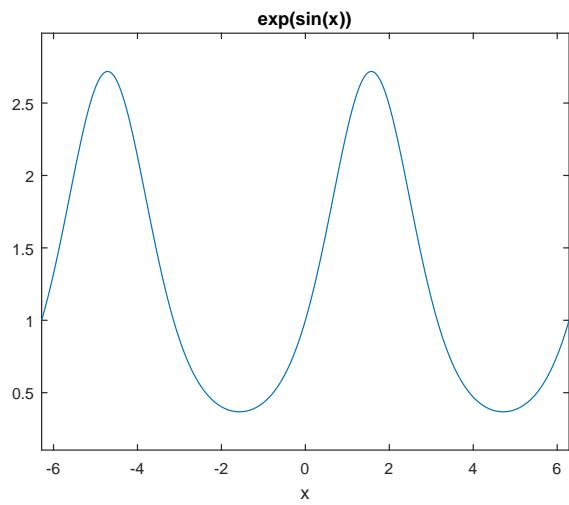
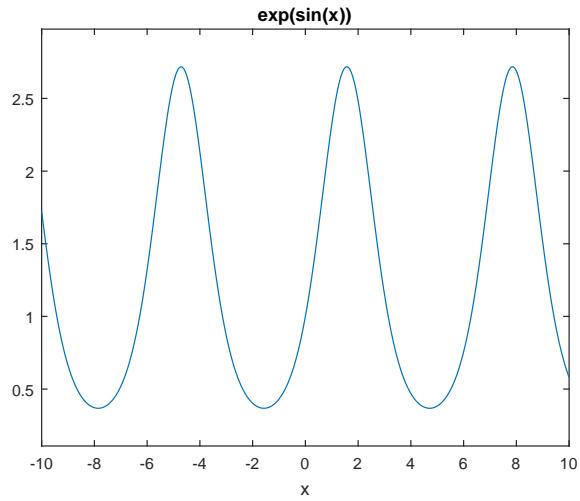
(a) Intervalo por defecto: $-2\pi \leq x \leq 2\pi$ (b) Intervalo suministrado: $-10 \leq x \leq 10$

Figura 10.1: Gráfica de la función $f(x) = e^{\sin(x)}$ obtenida a partir de su expresión simbólica con el comando **ezsurf** de Matlab

```
>> y = t*sin(t)
y =
t*sin(t)
>> ezplot(x,y)
```

El resultado se muestra en la figura 10.2. Si no se indica un intervalo concreto para el parámetro t , Matlab toma por defecto el intervalo $[0 \ 2\pi]$. El intervalo puede especificarse mediante un vector $[t_{min} \ t_{max}]$ de modo análogo al caso de la representación de funciones de una sola variable. Una aplicación directa de la representación de curvas paramétricas es la obtención de la trayectoria de un móvil conocida su posición en función del tiempo. Por ejemplo, la obtención de la parábola trazada por un cuerpo que se mueve bajo la fuerza de la gravedad.

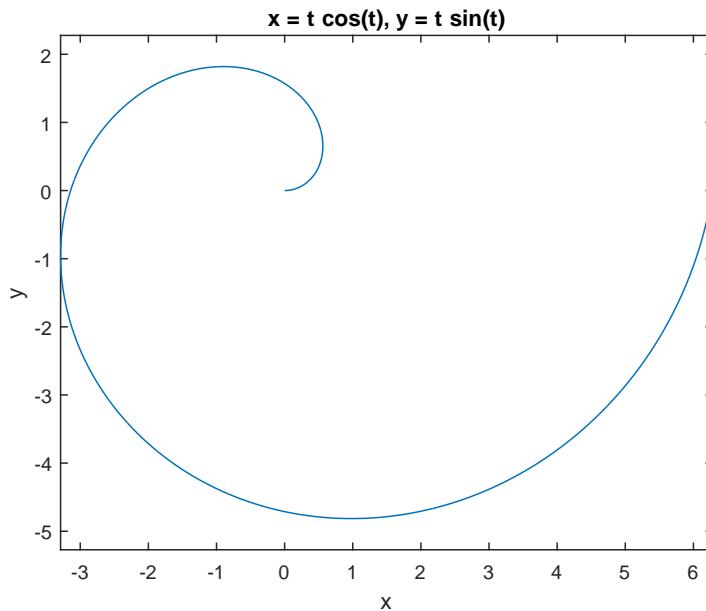


Figura 10.2: Gráfica de la curva paramétrica $x(t) = t \cos(t)$, $y(t) = t \sin(t)$ obtenida a partir de su expresión simbólica con el comando **ezsurf** de Matlab

Por último es también posible representar superficies empleando funciones simbólicas de dos variables. Para ello se emplea el comando **ezsurf**. En este caso, es preciso suministrarle al menos la función de dos variables que se desea representar. Si no se especifica rango, Matlab por defecto representará la función en el intervalo $[-2\pi \ 2\pi]$, tanto para la abscisa como para la ordenada. Además, **ezsurf** admite como un segundo parámetro un vector de cuatro elementos $[x_{min} \ x_{max} \ y_{min} \ y_{max}]$ que indica el área sobre la que se quiere representar la función. Es posible también añadir un segundo parámetro al comando que especifica el tamaño de la retícula empleada para representar la superficie. Si no se indica este segundo parámetro, Matlab traza la superficie empleando una retícula de 60×60 ,

```
>> f = sin(sqrt(x^2+y^2))
```

```
f =
sin((t^2*vo^2 + (h0 - (a*t^2)/2)^2)^(1/2))

>> syms t x y
>> f = sin(sqrt(x^2+y^2))
f =
sin((x^2 + y^2)^(1/2))

>> ezsurf(f)
```

La figura 10.3 muestra los resultados de este ejemplo.

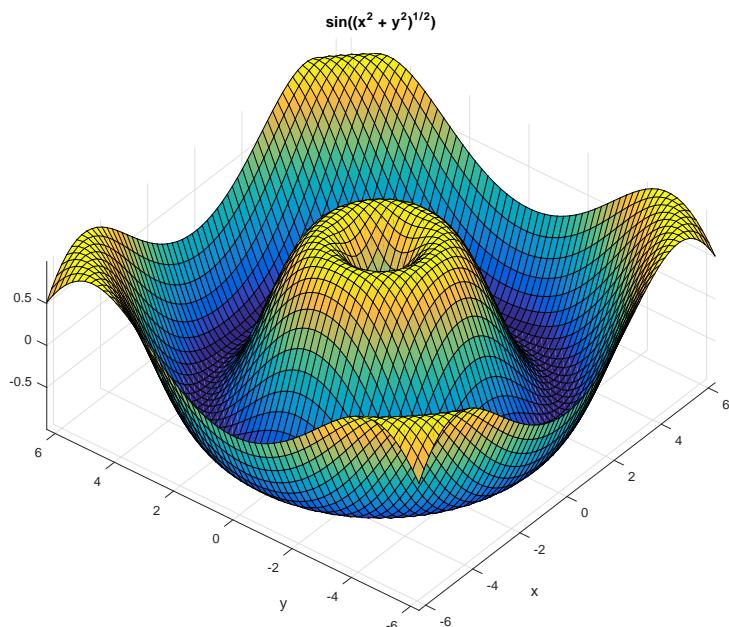


Figura 10.3: Gráfica de la función $f(x) = \sin(\sqrt{x^2 + y^2})$ obtenida a partir de su expresión simbólica con el comando `ezsurf` de Matlab

Hasta aquí la introducción al cálculo simbólico. Para obtener una visión completa de sus posibilidades se aconseja consultar la ayuda de Matlab