

Física Computacional

Tomo 1

Teoría, Problemas y Prácticas

Grado de Física de la Universidad de Zaragoza

Alfonso Tarancón Lafita
Departamento de Física Teórica
Instituto de Biocomputación y Física de Sistemas Complejos (BIFI)
Universidad de Zaragoza
España

21 de enero de 2024

Índice de contenido

1. Presentación de la Asignatura	11
1.1. Entorno de Programación	11
1.2. Bibliografía Principal y Compendio de Exámenes	12
1.3. Ayuda en el canal de Youtube	12
1.4. Bibliografía Adicional	12
1.5. Aclaraciones sobre como enfocar el estudio	13
1.6. Competencias específicas adicionales	13
1.7. Acercamiento a la investigación	14
1.8. Buenas prácticas para la programación	14
1.8.1. Organización del trabajo personal del alumno	16
1.9. Estructura de los apuntes	16
1.9.1. Tomo I	16
1.9.2. Tomo II	17
1.10. Evaluación	17
1.10.1. Examen Teórico	17
1.10.2. Trabajo en equipo	18
1.10.3. Nota Práctica	18
Opción A: Examen práctico	18
Opción B: Trabajos prácticos individuales	18
1.10.4. Calificación Final	18
1.10.5. Convocatorias de Junio y Julio	18
2. Ecuaciones Diferenciales	21
2.1. El método de Euler	21
2.1.1. Segmento de Código en C	23
2.1.2. Limitaciones del método	24
2.1.3. Conservación de la Energía	25
2.1.4. Reversibilidad temporal	25
2.1.5. Precisión y Errores acumulados	25
2.2. Método de Verlet	26
2.2.1. Segmento de Código en C	28
2.3. Ejercicios	29
2.3.1. El oscilador armónico	29
2.3.2. Sistema de Partículas en interacción gravitatoria	29
2.4. Problemas	30
2.5. Código en C	31
2.5.1. Conceptos de Programación: El preprocesador	31
2.5.2. El Oscilador Armónico	31
2.5.3. Sistema de partículas en interacción gravitatoria	33

3. Derivadas Parciales	37
3.1. La Ecuación del Calor	37
3.2. Sistema Unidimensional	37
3.2.1. Solución estacionaria	37
3.3. Discretización de la ecuación del calor	39
3.3.1. Nota	42
3.4. Nota sobre Punteros, vectores y funciones en C	42
3.5. Visualización del algoritmo	42
3.6. Detalles de implementación	42
3.7. Ejercicios	44
3.7.1. Ecuación del Calor en $d = 1$	44
3.8. Problemas	45
3.9. Código en C	46
3.9.1. Conceptos de Programación: Debug, Punteros y Funciones	46
3.9.2. Ecuación del calor en $d = 1$	46
3.10. Apéndice: Punteros y vectores en C	48
3.10.1. Algunos conceptos básicos de Hardware	48
3.10.2. ¿Qué son los punteros ?	50
3.10.3. Variables puntero	51
3.10.4. Los operadores de punteros: & y *	51
3.10.5. Expresiones de punteros	51
3.10.6. Punteros y arrays	52
3.10.7. Arrays de punteros	53
3.10.8. Indirección múltiple	53
3.10.9. Funciones de asignación dinámica de C	54
3.11. Apéndice: Funciones en C	54
3.11.1. Forma general de una función	54
3.11.2. Reglas de ámbito de las funciones	55
3.11.3. Argumentos de funciones	55
3.11.4. Llamada por valor	55
3.11.5. Llamada por referencia	56
3.11.6. Llamada a funciones con arrays	56
3.11.7. Punteros a funciones	58
4. Números aleatorios	59
4.1. Función densidad de probabilidad	59
4.1.1. Distribuciones Uniformes	60
4.1.2. Distribuciones no uniformes	62
4.1.3. Secuencias de números	63
4.2. Generación de distribuciones uniformes en C	63
4.2.1. Un ejemplo sencillo de generador uniforme	63
4.2.2. Generación de un número plano en un intervalo dado	64
4.2.3. Secuencias de números aleatorios en C	65
4.3. Construcción de Histogramas	65
4.4. Aparición de correlaciones en los números aleatorios	66
4.5. Sobre la reproducibilidad de secuencias random	67
4.6. Ejercicios	68
4.6.1. Cálculo de Histogramas	68
4.6.2. Distribución uniforme en un intervalo a base de pequeños cambios	68
4.6.3. Distribución Uniforme sobre un Círculo	68
4.7. Problemas	70
4.8. Código en C	71
4.8.1. Conceptos de Programación: Archivos, Comentarios, Input, Macros, Preprocesado	71
4.8.2. Cálculo de Histogramas	71
4.8.3. Distribución uniforme en un intervalo a base de pequeños cambios	74

4.9.	Apéndice: Representación de datos	76
4.9.1.	Código ASCII	76
4.9.2.	Números Enteros	76
4.9.3.	Números en Coma Flotante	78
5.	Distribuciones arbitrarias	79
5.1.	Método de la altura	79
5.2.	Método de la función de distribución	80
5.3.	Método de Metropolis	81
5.4.	Generadores uniformes de alta calidad	82
5.5.	Generación de puntos uniformes sobre curvas	84
5.5.1.	Método de la longitud de arco	84
5.5.2.	Método de la pendiente	84
5.6.	Generación de puntos uniformes sobre una superficie	85
5.7.	Ejercicios	87
5.7.1.	Evolución en el algoritmo de Metropolis	87
5.7.2.	Método de la Altura	87
5.7.3.	Generando puntos uniformemente sobre una superficie	87
5.7.4.	Puntos homogéneos sobre una superficie esférica	88
5.7.5.	Puntos Uniformes sobre un disco	90
5.8.	Problemas	91
5.9.	Código en C	94
5.9.1.	Conceptos de Programación: Datos, truncamiento, manejo de bits, macros avanzados	94
5.9.2.	Generación de distribuciones con el Método de Metropolis	94
5.9.3.	Puntos sobre una Superficie	95
5.9.4.	Puntos sobre una esfera	98
5.9.5.	Volumen de hiperesferas	99
5.9.6.	Sorteo Uniforme	100
5.10.	Apéndice: Teoría para generar puntos sobre una Superficie Parabólica	102
6.	Análisis estadístico y cálculo de errores	105
6.1.	Estadística básica	105
6.1.1.	Valores medios	106
6.1.2.	Cálculo de valores medios con datos numéricos	107
6.2.	Densidades de probabilidad Continuas	107
6.2.1.	Distribución Uniforme	107
6.2.2.	Distribución gaussiana	108
6.3.	Dispersión para la suma de variables aleatorias	110
6.3.1.	Varianza de la suma de dos distribuciones idénticas	111
6.3.2.	Ejemplos: Números y Juegos de Azar	111
6.3.3.	Calculo explícito para $x, y \in [0, 1]$	111
6.4.	Estimadores estadísticos	113
6.5.	Estimación de los Errores en una serie de datos	114
6.5.1.	Funciones de usuario útiles en C	115
6.6.	Ejercicios	116
6.6.1.	Funciones de usuario para la media y la varianza	116
6.6.2.	Medias y errores en una distribución plana	116
6.6.3.	Medias y errores en una distribución gaussiana	116
6.6.4.	Errores en Integración numérica	116
6.6.5.	Simulación de un Reactor de Fusión	116
6.7.	Código en C	118
6.7.1.	Conceptos de Programación: Los argumentos de main	118
6.7.2.	Errores en Integración numérica	118
6.8.	Apéndice: Errores en Integración Numérica	120
6.8.1.	Cálculo Elemental de Integrales	120

6.8.2. Ejemplo de Cálculo de Integrales	121
6.8.3. Estimación de Errores	121
6.9. Apéndice: Cálculo de integrales gaussianas	123
7. Análisis estadístico avanzado	125
7.1. El Teorema del Límite Central	125
7.1.1. Distribución de las Medias	125
7.2. Distribuciones no gaussianas o con autocorrelaciones	126
7.3. Análisis de errores con bloques	126
7.4. Errores para operadores compuestos	128
7.4.1. Propagación de errores	128
7.4.2. Análisis por bloques	129
7.5. Ejercicios	131
7.5.1. Media de distribuciones	131
7.5.2. Teorema del Límite Central	131
7.5.3. Análisis de Errores por Bloques	131
7.6. Problemas	132
7.7. Código en C	133
7.7.1. Análisis de Errores usando Bloques	133
7.8. Apéndice: Generador de distribuciones gaussianas	137
8. Movimiento Browniano	139
8.1. Imagen intuitiva del problema	140
8.2. Solución analítica	140
8.2.1. Distribución de probabilidad en función de r	142
8.3. Ejercicios	143
8.3.1. Simulación del movimiento Browniano	143
8.3.2. Difusión de Partículas	143
8.4. Problemas	145
8.5. Código en C	146
8.5.1. Conceptos de Programación: Código estructurado, uso de funciones predefinidas	146
8.5.2. Movimiento browniano en $d = 2$	146
8.5.3. Código para el Trabajo de Examen sobre Difusión	148
9. El Modelo de Ising	153
9.1. Introducción	153
9.2. El modelo de Ising	153
9.2.1. Definición de la Red y de los Spines	153
9.2.2. Definición de la Energía	155
9.2.3. Definición de la Magnetización	156
9.3. Construcción de la Red y condiciones de contorno	156
9.3.1. Construcción de los Direcccionamientos	159
9.4. Segmento de Código para el cálculo de la Energía	160
9.5. Configuración Inicial	161
9.6. Ejercicios	163
9.6.1. Generación de la Red y Cálculo de E y m	163
9.6.2. Manejo del Input/Output	163
9.7. Problemas	164
10. Principios básicos de la Mecánica Estadística	165
10.1. Función de Partición	166
10.1.1. Entropía	167
10.1.2. Un ejemplo	168
10.2. Simetrías	169
10.3. Valores esperados	169

10.3.1. Energía y Calor Específico	169
10.3.2. Magnetización y Susceptibilidad	170
10.4. Sistemas finitos	171
10.5. Evolución del Sistema con la Temperatura	171
10.5.1. Alta Temperatura	171
10.5.2. Baja Temperatura	174
10.5.3. Temperaturas Intermedias	176
10.6. Histogramas y Evolución temporal	176
10.6.1. Histograma de la Energía	176
10.6.2. Histograma de la magnetización	178
10.6.3. Evolución de la Energía y la Magnetización	178
10.7. Límite Termodinámico	179
10.7.1. Ruptura de Simetría, no analiticidad y magnetización espontánea	179
10.7.2. La magnetización en un retículo finito	181
10.7.3. Resumen de los valores medios en los casos asintóticos	182
10.7.4. Clasificación de las transiciones de fase	183
10.8. Ejercicios	184
10.8.1. Multiples configuraciones	184
10.8.2. Programa de Análisis	184
10.9. Problemas	185
10.10 Apéndice: Cálculo de observables en el límite $\beta = 0$	186
10.10.1. Cálculo de $\langle m^2 \rangle$	186
11. Simulación de Monte Carlo	189
11.1. Algoritmo de Metropolis	189
11.1.1. Cálculo del cociente de probabilidades	190
11.1.2. Cálculo del nuevo spin	191
11.1.3. Comprobación de signos	192
11.1.4. Medida y Ergodicidad	192
11.1.5. Definición de aceptancia	192
11.1.6. Optimización	193
11.1.7. Comprobación	194
11.1.8. Complejidad computacional del problema	195
11.2. Proceso de Markov e Importance Sampling	195
11.2.1. Termalización	196
11.3. Ciclo de Histeresis	196
11.3.1. Escritura de datos	197
11.3.2. Elección de parámetros	198
11.3.3. Simulación en el entorno del punto crítico	199
11.3.4. Simulación de diferentes tamaños	199
11.4. Ejercicios	200
11.4.1. Trabajo en Grupo	200
11.5. Problemas	201
11.6. Apéndice: Resultados para el Problema	201
12. Simulación avanzada del Modelo de Ising	207
12.1. Generación de Datos	207
12.1.1. Lectura de datos iniciales	208
12.1.2. Rutina de medidas y Escritura de resultados	209
12.1.3. Estructura global del Programa	210
12.2. Análisis de Resultados	210
12.2.1. Cálculo de errores	211
12.3. Análisis de Tamaño finito	212
12.4. Test de Consistencia: Ecuaciones de Schwinger-Dyson	212
12.5. Método de la Densidad Espectral	213
12.6. Ejercicios	217

12.6.1. Programa de Análisis	217
12.7. Problemas	218
13. Simulated Annealing	219
13.1. Planteamiento del Problema	219
13.2. Método del Simulated Annealing: Caso General	221
13.2.1. Construcción del Modelo	221
13.2.2. Proceso de Optimización	221
13.3. Resolución de $A\vec{x} = \vec{b}$	222
13.3.1. Construcción del modelo	223
13.3.2. Proceso de Optimización	223
13.4. El problema del viajante	224
13.5. Construcción del modelo	224
13.6. Proceso de Optimización	226
13.7. Generación de Cambios Tentativos	227
13.8. Ejercicios	229
13.8.1. Resolución de $A\vec{x} = \vec{b}$	229
13.8.2. Problema del Viajante	229
13.9. Problemas	231
13.9.3. Problema 3	231
13.9.4. Problema 4	231
13.10. Código en C	232
13.10.1. Simulated Annealing para $Ax = b$	232
14. Redes Complejas	235
14.1. Introducción	235
14.2. Definiciones	236
14.3. Componentes, distancias y diámetro	238
14.4. Comunidades	238
14.5. Medidas de Centralidad: Betweeness y Page Rank	239
14.5.1. Betweeness	239
14.5.2. Page Rank	240
14.6. Algoritmos de Posicionamiento	241
14.6.1. Campo de Fuerzas	242
14.6.2. Parámetros del Potencial	243
14.6.3. Solución de Equilibrio	244
14.7. Ejercicios	246
14.7.1. Algoritmo de Posicionamiento	246
14.8. Problemas	247
14.9. Código en C	248
15. Neural Networks	259
15.1. Cerebro y Neuronas	260
15.2. El perceptrón	262
15.3. Reconocimiento de Patrones	264
15.4. Procesos de Aprendizaje Supervisado	267
15.5. Attractores	269
15.6. Ejercicios	272
15.7. Código en C	273
16. Ejemplo de Examen	285
16.1. Examen Teórico	285
16.2. Examen Práctico	285
16.3. Ejemplo de Convocatoria de Examen	286
16.4. Ejemplo Examen Práctico	287
16.5. Ejemplo Código Práctica	290

ÍNDICE DE CONTENIDO

9

16.6. Ejemplo Teórico	292
---------------------------------	-----

Capítulo 1

Presentación de la Asignatura

Esta asignatura de Física Computacional tiene como objetivo introducir a los alumnos en el cálculo numérico con ordenador para resolver diferentes tipos de problemas comunes en la Física y en otras disciplinas.

La asignatura se imparte durante 14 semanas efectivas. Se imparten 4 horas semanales (6 créditos), de las cuales 2 horas corresponden a clases Teóricas y de Problemas y 2 a Clases con ordenador.

En las clases teóricas se explicará el fundamento físico o matemático del problema, los algoritmos a utilizar y los aspectos del Código del lenguaje C aún no conocidos y necesarios para el caso. Se comentará brevemente la organización del código en funciones, macros, etc. y los aspectos especialmente complicados del mismo.

En la parte de problemas se implementará lo explicado anteriormente escribiendo las partes centrales del código (el núcleo) con *papel y lápiz*, previo a su escritura en el ordenador, al trabajo individual del alumno y a la clase de prácticas.

En estos apuntes figura el código de varios de los ejercicios y prácticas que se realizarán durante el curso. El alumno no debe consultar este código hasta después de haber escrito una versión propia. Sólo ante dificultades importantes debe consultarla previamente. Una vez escrito su código, compilado y ejecutado, deberá consultar el código de los apuntes para corregir errores comunes, aprender a escribir programas bien estructurados, con nuevos métodos e ideas.

En el programa no se incluyen Temas de Matrices o de Análisis Espectral, ya incluidos en otras asignaturas del Plan de Estudios.

1.1. Entorno de Programación

Las prácticas se realizan en un aula con 20 ordenadores, y usaremos el entorno de compilación abierto CodeBlocks (<http://www.codeblocks.org/>) y para realizar gráficas el programa de software libre **gnuplot**. Aquellos estudiantes que lo deseen podrán usar otros entornos. Recomendamos muy especialmente el uso de Linux, con el compilador **gcc**, el editor **emacs** y el programa de dibujo **gnuplot**, todos ellos de software libre. En Windows, es conveniente que el alumno utilice compilación directa con **gcc**, lo que facilita el uso especialmente en el caso de introducir parámetros en la linea de comandos; además el uso directo de **gcc** simplifica todo el entorno de trabajo y permite usar cualquier editor que el alumno conozca.

Sobre Linux, **gcc**, **emacs** y **gnuplot** existen en la red cantidades ingentes de manuales, introducciones y ejemplos, fácilmente accesibles. Pueden consultarse las páginas :

- <https://www.linuxparty.es/TutorialLinux/>
- <http://www.gnu.org/software/emacs/#Manuals>
- <http://iie.fing.edu.uy/~vagonbar/gcc-make/gcc.htm>
- <http://gcc.gnu.org/onlinedocs/>
- <http://www.gnuplot.info/>

1.2. Bibliografía Principal y Compendio de Exámenes

Estos Apuntes pueden descargarse en formato PDF desde:

<https://drive.google.com/open?id=1p7Gfwjao2VtWzPEUGIlcmHZM9pPawu88>.

En esa URL se puede acceder tambien al Tomo II de los apuntes así como a un compendio de exámenes de la asignatura de años anteriores.

1.3. Ayuda en el canal de Youtube

Hemos creado un canal de Youtube donde pueden verse tutoriales sobre algunos temas. El canal se accede en esta link: <https://www.youtube.com/channel/UCwTuw5lwdIG3RS7Kw8hUEXw>

Concretamente conviene al arrancar el curso visualizar en detalle el tutorial sobre el uso de CodeBlocks.

1.4. Bibliografía Adicional

Si bien en Internet se encuentran infinidad de páginas sobre cálculo numérico, física computacional y programacion, tambien la consulta de libros aporta conocimientos importantes, muy bien organizados y filtrados ayudando enormemente al estudio.

- **Numerical Recipes in C (The Art of Scientific Computing)**, Editorial Cambridge y https://es.wikipedia.org/wiki/Numerical_Recipes

Probablemente el mejor libro en cálculo numérico en el mundo; sin duda el más reconocido. Contiene prácticamente todos los algoritmos numéricos más usados en Ciencia, con una explicación clara, concisa y profunda de todos ellos, junto al codigo correspondiente en C. Es un libro de consulta frecuente que vale la pena tener si se utiliza el cálculo numérico. Existen versiones on line gratuitas, o en pdf e incluso pueden descargarse los programas.

- **Scientific Programming: C-Language, Algorithms and Models in Science** Luciano Maria Barone , Enzo Marinari, Giovanni Organin, Federico Ricci Ter-senghi. (“Sapienza” Università di Roma, Italy). Editorial Worldscientific

Escrito por Físicos de reconocido prestigio internacional en simulaciones avanzadas y en investigación de nuevos algoritmos, contiene un buen número de temas, todos ellos de interés, con ejemplos y simulaciones de utilidad general.

- **Computational Physics: Simulations of Classical and Quantum Systems**, Philipp O.J. Scherer, Editorial Springer.

Publicación muy precisa, tratando los temas con gran profundidad.

- **The C programming laguage**, B.W. Kernighan and Dennis M. Ritchie, Editorial Prentice Hall

El libro de referencia por excelencia en C. Dennis M. Ritchie fue el diseñador del lenguaje, y esta publicación fue la que difundió su uso por todo el mundo. Incluso fijó en la práctica el standard ANSI C, y sigue siendo un libro de referencia para el lenguaje.

- **A book on C**, Al Kelley, Ira Pohl. Editorial Addison Wesley

Contiene desde los aspectgos básicos de C, a aspectos avanzados de programación y algoritmos generales. Incluye una sucinta pero excelente introducción a la Programacion Orientada a Objetos y a lenguajes interpretados, con los conceptos básicos de C++ y Java.

- **Networks, An Introduction**, M.E.J. Newman

Más que una introducción un compendio bastante completo de la Teoría de Redes Complejas. Amplio, detallado y comprensible.

- **Complex Networks: Structure and dynamics**, S. Bocaletti *et al.* Physics Reports, 424 (2006) 175-308.

Un excelente report en Redes Complejas, sucinto y destinado a personas interesadas en la investigación en este campo.

- **Modelin Brain Function: The world of attractor neural networks**, Daniel J. Amit, Cambridge University Press

Uno de los primeros libros de texto en Redes Neuronales, escrito por uno de los Físicos Teóricos que contribuyó a dar a este campo consistencia teórica.

- **Introduction to the theory of Neural Computation**, J. Hertz, A. Krogh y R.G. Palmer, Westview Press Una visión global de las Redes Neuronales con diferentes aplicaciones a procesos de reconocimiento de imágenes, procesos cognitivos, optimización, y redes multicapa.

1.5. Aclaraciones sobre como enfocar el estudio

Los alumnos deberán realizar el trabajo de escritura del código de forma individual (salvo en el Capítulo 11) y previamente a la clase práctica. En la misma, se tratará de resolver los problemas comunes, dar indicaciones a aquellos que vayan más retrasados, y proponer ampliaciones y mejoras. Para obtener un buen rendimiento en la asignatura conviene aclarar algunos aspectos, y ciertas ideas erróneas detectadas al impartir este tipo de asignaturas.

Actualmente muchos alumnos llegan a la Facultad con el convencimiento de que poseen amplios conocimientos de informática. Ciertamente los alumnos están familiarizados con los ordenadores y con aplicaciones en general de ofimática (tipo Word, Excel, ...) y por supuesto con el manejo de Internet. Esto por un lado es una ventaja pues esta parte básica ya la conocen, pero por otra supone un gran problema.

- La formación en muchos casos autodidacta hace mantener a los alumnos una cierta posición de autosuficiencia muy negativa para lograr que adquieran nuevos hábitos y conocimientos.
- El ordenador es visto como una caja negra, sobre la cual jamás se plantean preguntas sobre cuestiones básicas, y el plantearlas no parece en absoluto interesante
- Se ve el ordenador como una ventana a Internet, donde a base de buscar, sin necesidad de entender ni aprender, se puede encontrar respuesta a todo: incluso código en C o Fortran para resolver problemas. Puede llegar a pensarse que la forma de resolver un problema es “Busquemos en Internet alguien que ya lo haya hecho antes”.
- El mundo del ordenador (o *virtual*) y el mundo real se ven como *Universos* diferentes. El alumno no asimila que con programas desarrollados por él mismo, el ordenador puede simular problemas que se corresponden con el comportamiento real de sistemas físicos.

En estos años, donde han ido emergiendo todos estos problemas, hemos incorporado acciones metodológicas, reflejados en estos Apuntes, que corrigen en lo posible estos defectos. Pero la aportación esencial procede del alumno, que debe afrontar la asignatura dejándose guiar por los profesores y trabajando diariamente.

1.6. Competencias específicas adicionales

Además de los objetivos generales ya explícitos en el Plan de Estudios, en esta asignatura pretendemos que los alumnos adquieran nuevas capacidades,

- **Capacidad Crítica:** La construcción de programas informáticos que simulan sistemas físicos permite extraer resultados que pueden ser comparados con soluciones analíticas, o que pueden ser testados de diferentes maneras (por ejemplo comprobando magnitudes

conservadas). La confrontación entre lo obtenido en el ordenador y los resultados reales es un criterio básico de fiabilidad de sus resultados. Típicamente cuando el alumno piensa que el problema está resuelto correctamente, estas comprobaciones muestran diferentes errores. El alumno debe acostumbrarse a que también las simulaciones en ordenador, al igual que las teorías físicas, deben pasar el tamiz de la comprobación experimental (numérica en nuestro caso) para establecer su validez.

- **Capacidad de Resolver Problemas:** En la asignatura no se trata sólo de dar conocimientos de informática, programación, cálculo numérico, etc. sino principalmente de que, dados estos conocimientos básicos, el alumno sea capaz de enfrentarse a problemas nuevos, plantearlos correctamente y elegir la metodología óptima para resolverlos, con más o menos aproximaciones y simplificaciones, pero dando siempre una respuesta. Esto es especialmente relevante para el contacto con problemas computacionales que se presentarán en la Investigación o en las Empresas.
- **Capacidad de Trabajo en Equipo:** El desarrollo moderno de programas hace obligatorio que cada programador desarrolle una parte de un código que luego debe ser ensamblado con otros. Esto implica la necesidad de una disciplina en el trabajo personal, pensando en el trabajo del resto, y la puesta en común final de todas las partes para que el programa funcione correctamente. En este proceso el trabajo en equipo puede ser evaluado y visualizado por los alumnos, mostrando explícitamente si se ha desarrollado de forma correcta.

1.7. Acercamiento a la investigación

Durante el curso se desarrollarán actividades para que los alumnos lleguen a *ver* cómo la computación ayuda a resolver problemas científicos y tecnológicos, suponiendo una herramienta vital en la investigación.

Para ello se realizará una visita a un centro de investigación, donde los alumnos podrán interaccionar con técnicos y grupos de investigación que usen el ordenador como herramienta para sus investigaciones. En concreto se visitará el Instituto de Biocomputación y Física de Sistemas complejos (BIFI). El BIFI es un Instituto de Investigación Universitario donde la Computación, la Física y la Bioquímica se unen para estudiar diferentes sistemas como las Redes Sociales, los Vidrios de Spin, las Proteínas, algunos fármacos o el desarrollo de ordenadores específicos. Se realizará un recorrido por las instalaciones más relacionadas con la materia de la asignatura, y se recibirá información sobre algunas de las investigaciones en curso; en concreto

- *Centro de Supercomputación:* Instalaciones generales, Clusters, Ordenadores de Memoria Compartida, Tecnologías GRID y Cloud, Ordenadores dedicados y Computación voluntaria. Nodo Cesaraugusta, de la Red Española de Supercomputación. Diferentes investigadores explicarán las aplicaciones típicas que ejecutan y los resultados científicos que se esperan obtener.
- *Laboratorios:* Instalaciones de Microscopía, Rayos X
- *Redes Complejas* Definición, investigaciones actuales y ejemplos de uso.
- *Desarrollo de aplicaciones* Webs avanzadas, Ciencia Ciudadana: experimento, web, aplicaciones móviles.
- *Simulaciones Masivas* Simulaciones de dinámica molecular en superordenadores.

En la página <https://www.bifi.es> puede verse información adicional.

1.8. Buenas prácticas para la programación

La programación es una tarea en general atractiva por lo que una vez iniciada la escritura del código es difícil sustraerse a una actividad que absorbe toda nuestra atención y muchas veces

deja poco tiempo para la reflexión sobre lo que se está haciendo. El proceso de aprendizaje es vital para crear buenas prácticas, y evitar vicios y defectos que luego son difíciles o imposibles de corregir.

Los temas y ejercicios de estos apuntes han sido elegidos con una complejidad creciente, con ejercicios de ampliación, cuestiones y preguntas para favorecer un buen aprendizaje. Es importante seguir los consejos dados en los apuntes y los dados en clase para adquirir estas buenas prácticas.

Lo habitual es que tras leer u oír el problema, nos pongamos delante del ordenador a escribir código según nos va viniendo a la cabeza. Esto es una práctica pésima, que debe ser evitada. Daremos aquí algunos consejos preliminares, aprovechando el desarrollo cronológico, para escribir código:

1. **Planteamiento del problema:** Debe estar especificado con precisión, contemplando todas las posibilidades. Debemos reflexionar sobre todos los posibles problemas, dificultades, y si tenemos previstos todos los casos. En este paso usamos principalmente nuestro cerebro. Por supuesto no usamos el ordenador.
2. **Fijar el algoritmo:** Debemos decidir qué tipo de algoritmo usamos para resolver el problema. En general un problema puede ser atacado con muchos métodos. Dependiendo del problema y del contexto (¿tenemos mucho tiempo?, ¿el ordenador donde se resolverá es grande o pequeño?...) deberemos usar unos u otros. En general buscaremos el más eficiente y sólido. Aquí, de nuevo la herramienta principal es el cerebro
3. **Esbozar el código:** Debemos escribir con lápiz y papel un esquema del código, de sus partes, flujo de datos, Input/Output, estructura de `main()` y de funciones. Aquí todavía no hemos encendido el ordenador.
4. **Escritura del Código:** Ahora escribimos el código en el ordenador. Hay que incluir comentarios. Hay que indentar el programa. Debemos escribir todas las tareas posibles en funciones, estructurando correctamente. Los nombres que se da a las variables deben ser significativos, en correspondencia con lo que significan en el programa para ayudar a una mejor comprensión. Por ejemplo, si usamos una variable que contiene el módulo del momento angular, vale la pena teclear un poco más y llamarla `double Modulo_momento_angular`, pues se tiene tendencia a algo del tipo `mma` o `mod` o incluso `k`; esto hace, pasado el momento justo de la escritura, que recordar lo que hace sea una tarea larga y tediosa.
5. **Fase de Compilación:** El compilador da mensajes de error precisos. Hay que leerlos con atención, comenzando por el primero de ellos, y corregir de forma sistemática. El compilador también informa de *Warnings* o Avisos, que indican que hay algo no del todo correcto, si bien no está seguro de que sea un error. Es habitual no hacer caso de ellos, pero es un hábito muy negativo. Los warnings son errores en muchos casos, e incluso aunque no lo sean, mantenerlos genera código de peor calidad, que puede dar errores en otras máquinas y que entorpece el proceso de tratamiento y corrección de errores (fase de depuración o *debugging*) .
6. **Fase de Debugging:** Una vez que el código se compila correctamente, debemos comprobar que al ejecutarse hace lo que queremos de forma correcta. Siempre debemos buscar formas para comprobarlo. Un fallo común es probar con unos datos fijos, y cuando el resultado es correcto, se supone correcto el programa. Esto es un error. Hay que probar con un gran número de datos diferentes, de los cuales sepamos los resultados, para dar el programa por correcto, lo que es conocido como *Validación*.
7. **Almacenamiento de los códigos finales:** Una vez estamos seguros de que todo es correcto, debemos guardar las versiones finales del código en un directorio específico para ese problema, incluyendo solamente los programas correctos, eliminando versiones intermedias, ficheros obsoletos o no útiles, etc. Conviene crear un fichero con un nombre del tipo `README` donde se explique como funciona el programa, como se compila y ejecuta, qué datos son necesarios en la linea de comandos, archivos de entrada y salida y todo lo necesario

para que otra persona, o nosotros pasado un largo tiempo, podamos compilarlo, ejecutarlo y comprender los datos de entrada y salida.

1.8.1. Organización del trabajo personal del alumno

Esta asignatura no es conceptualmente compleja. Resulta además atractiva especialmente en la fase de programación con el ordenador. Sin embargo requiere de nuevas ideas, que a diferencia de las Matemáticas o la Física, no se han estudiado con suficiente tiempo en el Bachillerato. También requiere hábitos de trabajo nuevos, que no son fáciles de adquirir por los vicios previos y por trabajar de forma aislada. La complejidad de la programación, de la Física involucrada y de los algoritmos es media. Pero un desfase en cualquiera de estas partes hace incomprensible la asignatura, y que el trabajo diario en clase y en los ordenadores sea absolutamente sin provecho. En esta asignatura no cabe dejarlo todo para el final. Los conceptos requieren de tiempo en el cerebro para ser asimilados, y es imposible hacerlo en poco tiempo.

Para un aprovechamiento óptimo del tiempo, de las clases y del trabajo individual, recomendamos los puntos siguientes,

1. Clases de Teoría y Problemas: Es necesario un conocimiento de todo lo anterior y preferiblemente un repaso rápido anterior a la clase del tema a tratar. Dado que los Apuntes suministrados contienen exactamente el contenido del curso, no es necesario tomar apuntes, es más, lo desaconsejamos absolutamente. Basta con tener los Apuntes delante para sobre ellos tomar alguna nota, comentario, o respuestas a preguntas que el alumno considere relevantes.
2. Estudio y trabajo particular: El alumno debe estudiar los conceptos nuevos, repasar lo que sea necesario. Debe escribir el código propuesto para las prácticas y realizar un esfuerzo para lograr compilarlo y que funcione correctamente al menos en sus partes básicas. Caso de no lograrlo puede consultar los Apuntes, y preparar las preguntas para la clase de prácticas. Es muy conveniente que se resuelvan los problemas indicados en el capítulo.
3. Clase de Prácticas: El objetivo es acabar el programa propuesto, incorporar todas las funcionalidades o añadir nuevas. Caso de no tener problemas es el momento de consultarlas con el Profesor.

1.9. Estructura de los apuntes

Los Apuntes constan de dos partes. La Primera, este Tomo, es el contenido propiamente dicho de la Asignatura de Física Computacional. La Segunda, el Tomo II (descargable también en <https://drive.google.com/open?id=1p7Gfwjao2VtWzPEUGIlcmHZM9pPawu88>) son contenidos de apoyo que en parte ya son conocidos por el Alumno o sirven de referencia y guía para el trabajo. Este segundo Tomo no es imprescindible para el alumno, si tiene los conocimientos previos básicos, o prefiere consultar otras referencias del lenguaje C y *gnuplot*.

1.9.1. Tomo I

Cada capítulo (excepto este de presentación) se corresponde con una *Clase de Teoría*, una *Clase de Problemas* y una *Sesión de Prácticas*, a excepción del Capítulo 11 que pueden corresponder hasta con dos clases dependiendo del desarrollo del curso. Existe cierta holgura en el tiempo para repasar los temas que en el proceso de aprendizaje los alumnos en general consideren más difíciles.

En cada capítulo pueden distinguirse hasta 5 partes

1. **Desarrollo teórico:** Se explican los conceptos físicos y los algoritmos numéricos para resolverlos. Corresponde a la hora de Teoría.
2. **Ejercicios:** Corresponde al trabajo a realizar durante la sesión de prácticas con el ordenador. Se presentará, junto con segmentos de código, en la clase de Problemas.

3. **Problemas:** El alumno debe resolverlos para fijar ideas y comprobar que ha adquirido los conocimientos exigidos.
4. **Código en C:** Se incluyen listados de programas completos que corresponden a algunos de los ejercicios o problemas. Todos compilan y funcionan correctamente, si bien en general no contienen todo lo explicado ni están debidamente estructurados. Esto es así para que el alumno pueda reescribirlo debidamente estructurado (organizado, comentado, dividido en funciones, etc...), debiendo completarlo incluyendo todos los detalles discutidos en clase y no implementados en los ejemplos. No obstante insistimos en que este código sólo debe ser consultado tras la escritura por parte del alumno de su versión propia; sólo en caso de dificultades graves debería consultarse antes de la escritura del código.
5. **Apéndices:** Ampliaciones de aspectos puntuales o de referencia.

Durante el curso no se explicarán en clase los capítulos 7 ni 12 marcados como *Avanzados*, que contienen conceptos algo más complejos que el resto, y no son imprescindibles. Los alumnos que lo deseen podrán trabajarlos personalmente, con la ayuda de los profesores en las tutorías o en la clase de prácticas en lo referente a los conceptos y el código.

1.9.2. Tomo II

Está contenida en un Volumen diferente. Contiene los siguientes apartados,

1. **Nociones básicas de un Ordenador Personal** Presentación del hardware básico de un PC, sus características y cuestiones elementales de electrónica digital y operaciones con puertas lógicas.
2. **Infraestructuras de Cálculo y Aplicaciones Científicas** Contiene una descripción de los diferentes tipos de ordenadores utilizados en cálculo científico, las diferentes arquitecturas para cada tipo de problema.
3. **Introducción al gnuplot:** Programa para dibujo de datos y gráficas en $d = 2$ y $d = 3$ en Windows o Linux. Se usará a lo largo del curso para visualizar resultados.
4. **Introducción al Lenguaje C:** Manual básico de C. Los contenidos ya son conocidos por los alumnos, pues han sido impartidos en asignaturas previas.
5. **El Lenguaje C: Nivel Medio:** Manual avanzado de C, con conceptos nuevos para los alumnos, parte de los cuales se usarán en el curso; conocimientos necesarios para desarrollar programas habituales en el estudio del Grado.
6. **Programas Paralelos: la Ecuación del Calor** Contiene un ejemplo de cómo puede parallelizarse el algoritmo usado en la Ecuación del Calor (Capítulo 3 del Tomo I). No se incluye código sino sólo las ideas generales, la dependencia del hardware, eficiencia, comunicaciones, etc.

1.10. Evaluación

1.10.1. Examen Teórico

Examen sobre los conocimientos adquiridos durante el curso. Consta de 5 preguntas sobre teoría, algoritmos, código, etc. Pueden verse ejemplos concretos al final de este Tomo I, en el Capítulo 16

Puntuación Máxima: 6 puntos.

1.10.2. Trabajo en equipo

Para fomentar el trabajo en equipo, y las buenas prácticas de programación, el ejercicio del Capítulo 11 se resolverá por grupos de 6 alumnos (Este número depende del total de alumnos matriculados). El grupo presentará por escrito el código completo y los resultados, gráficas y conclusiones, concretados en la sección **Ejercicio** de dicho capítulo. Cada grupo realizará una presentación de su trabajo, usando los recursos disponibles en clase (pizarra y cañón) en una clase de Teoría, durante 10 minutos. Los alumnos dispondrán de al menos una clase de Teoría, Problemas y Prácticas para discutir las dificultades y detalles del trabajo. **Puntuación Máxima: 2 puntos**

1.10.3. Nota Práctica

Existen dos opciones, entre las que puede elegir cada alumno libremente.

Opción A: Examen práctico

Previo a la fecha de examen (aproximadamente una semana antes), se propondrá un problema de competencia similar a los estudiados durante el curso que el alumno deberá resolver. Este código será presentado el día de la prueba práctica. El código presentado y la solución del problema propuesto se puntuará con hasta 1 punto. Tras esta presentación, se propondrá una modificación del problema *in situ*; la correcta solución de lo propuesto se puntuará hasta con 1 punto. En el Tomo I, en el Capítulo 16 puede verse una convocatoria de examen con el texto de la prueba práctica. **Puntuación Máxima: 2 puntos.**

Opción B: Trabajos prácticos individuales

En esta opción cada alumno deberá realizar dos trabajos. Para cada ejercicio, se proporcionará el enunciado a lo largo del curso con, al menos, 4 semanas de antelación. El estudiante deberá resolverlo siguiendo las pautas indicadas en clase. Después, deberá realizar una pequeña prueba durante las clases prácticas en las que se pedirá hacer una modificación a su código. La participación en la primera de estas pruebas implicará que el estudiante ha escogido esta opción de evaluación práctica. **Puntuación Máxima: 2 puntos.**

1.10.4. Calificación Final

En el Examen Teórico es necesario sacar al menos una puntuación de 3 puntos para poder aprobar la asignatura. Por lo tanto, los puntos del trabajo en equipo y de la prueba práctica sólo se sumarán en caso de superar los 3 puntos en dicha prueba teórica.

1.10.5. Convocatorias de Junio y Julio

En la convocatoria de Julio, el examen teórico será obligatorio para todos los que se presenten. La nota del trabajo en equipo se mantiene.

Aquellos alumnos que lo deseen, podrán mantener la nota práctica de Junio (obtenida en el examen práctico o en el trabajo individual) si lo desean. Se podrá optar por presentarse a la prueba práctica, que será siempre en la modalidad de Examen práctico y en este caso la puntuación será la obtenida en esta convocatoria de Julio. Es decir *se guarda la nota de Junio* de la parte práctica, siendo opcional presentarse de nuevo, pero en este caso, la nota de esta parte será la obtenida en Julio, perdiendo la nota anterior.

Agradecimientos

Estos apuntes han sido preparados gracias a la confianza que me ha inspirado el contar en todo momento con la ayuda, enseñanza y apoyo constante de Luis Antonio Fernández, del Departamento de Física Teórica I de la Universidad Complutense de Madrid. Sin su continuo trabajo,

yo habría sido incapaz de prepararlos. Quede pues manifiesto mi más profundo agradecimiento y admiración a Luis Antonio, para el que no existen secretos en estos campos, salvo lo puramente irresoluble.

Javier Moreno Gordo y Francisco Bauzá Minguez han creado el Canal de Youtube que es de una gran ayuda para la comprensión de los temas tratados. También han preparado los excelentes códigos de las secciones [6.6.5](#) y [8.3.1](#). En general, ambos han contribuido significativamente a la mejora de estos Apuntes.

Capítulo 2

Ecuaciones Diferenciales Ordinarias

Dado un sistema de partículas sometidas a una fuerza conocida, las Leyes de Newton nos permiten escribir las ecuaciones diferenciales que obedece su movimiento. Sin embargo, pocos son los casos donde podemos resolver estas ecuaciones diferenciales de forma analítica. En general debemos recurrir a su resolución numérica. Comenzaremos estudiando una partícula sometida a una fuerza en una dimensión, en concreto el oscilador armónico. Escribiremos las ecuaciones diferenciales que obedece el sistema, y las discretizaremos para su resolución en el ordenador.

El campo de las ecuaciones diferenciales es uno de los más amplios del análisis numérico. De los numerosos métodos disponibles utilizaremos en primer lugar el más simple de todos ellos, pero que da una idea intuitiva clara: el método de Euler. Para obtener una solución más precisa, en concreto que conserva la Energía y genera trayectorias reversibles en el tiempo, usaremos el método de Verlet. Si bien no los estudiaremos aquí, existen diferentes métodos muy efectivos para resolver el movimiento de partículas. Cuando se quiere hacer con precisión, un método especialmente efectivo es el Runge-Kutta de orden 4, que puede ser consultado fácilmente en la bibliografía.

Siempre que sea posible, es necesario realizar pruebas exhaustivas para verificar que nuestros programas sean correctos. En algunos casos (como el oscilador armónico) disponemos de una solución analítica exacta, con la cual podemos comparar. Cuando no se dispone de solución exacta, pueden existir cantidades conservadas, como la Energía, Momento Angular, etc. que deberemos verificar que efectivamente son constantes (salvo redondeos numéricos). Como control adicional o en caso de no poder usar estas cantidades, podemos simular el sistema con parámetros especiales, o partiendo de condiciones iniciales dadas, de modo que la evolución sea conocida.

Tras introducir las nociones básicas de cada método, discutiremos varios ejemplos.

2.1. El método de Euler

Consideremos una partícula sometida a una fuerza conocida en función de la posición y del tiempo $F(x, t)$. Esta partícula se moverá de acuerdo con la Ley de Newton, $F = ma$. Si conocemos $F(x, t)$, podemos encontrar el movimiento de la partícula resolviendo la ecuación

$$\frac{d^2x(t)}{dt^2} = \frac{1}{m}F(x, t) \quad (2.1)$$

Sabemos resolver analíticamente esto sólo en casos sencillos. En general no es posible, y lo haremos numéricamente. En los ejemplos consideraremos sólo fuerzas que dependen de x y no del tiempo, por lo que para simplificar la notación escribiremos la fuerza como $F(x)$.

Resolver la ecuación anterior puede parecer complicado; podemos reducirlo a un problema más sencillo.

Empecemos estudiando el caso más simple de una Ecuación con derivadas primas

$$\frac{dx(t)}{dt} = g(x) \quad (2.2)$$

Discretizamos el eje temporal: $t = hn$ donde h es un número pequeño y n un entero. La derivada, para valores pequeños de h , será aproximadamente:

$$\frac{dx(t)}{dt} \Big|_{t=hn} \approx \frac{x(hn+h) - x(hn)}{h} \quad (2.3)$$

Entonces, si conocemos $x(0)$, podemos calcular de forma iterativa $x(hn)$

$$\begin{aligned} \frac{dx(t)}{dt} \Big|_{t=hn} &\approx \frac{x(hn+h) - x(hn)}{h} = g(x(hn)), \\ x(hn+h) &= x(hn) + hg(x(hn)). \end{aligned} \quad (2.4)$$

Es decir, dado el valor de la función $x(t)$ en $t = 0$, y conocida la función $g(x)$ en $g(x(0))$, podemos calcular el valor de la función en $x(h)$; veáse la figura 2.1. Iterando este proceso podemos alcanzar cualquier valor de $t = hn$, con lo que tendremos resuelto el problema, al menos formalmente.

Una notación habitual es

$$t_n = hn, t_{n+1} = h(n+1), \dots \quad (2.5)$$

La variable t_n corresponde al tiempo físicamente, h es el paso temporal y n un índice entero sin dimensiones. Por extensión, para variables que dependen de $x(t)$, usaremos indistintamente alguna de las notaciones siguientes,

$$g(x(t)) \equiv g(x(t_n)) \equiv g(x_n) \equiv g_n \dots \quad (2.6)$$

En nuestros algoritmos numéricos para resolver el problema, usaremos n como el índice de los bucles, o índice de vectores.

Pasemos ahora al problema con derivadas segundas, como corresponde a la Ecuación de Newton. Una opción simple es convertirlo en dos ecuaciones con derivadas primeras, introduciendo la velocidad,

$$\frac{d^2x(t)}{dt^2} = \frac{1}{m}F(x) \Rightarrow \begin{cases} v(t) = \frac{dx(t)}{dt}, \\ \frac{dv(t)}{dt} = \frac{1}{m}F(x). \end{cases} \quad (2.7)$$

Ahora debemos resolver dos ecuaciones de primer orden acopladas. Para ello haremos

$$\begin{aligned} \frac{dv(t)}{dt} &\approx \frac{v(hn+h) - v(hn)}{h} = \frac{1}{m}F(x) \\ v(hn+h) &= v(hn) + h\frac{1}{m}F(x) \\ \frac{dx(t)}{dt} &\approx \frac{x(hn+h) - x(hn)}{h} = v(hn) \\ x(hn+h) &= x(hn) + hv(hn) \end{aligned} \quad (2.8)$$

Ahora, dados $x(0)$ y $v(0)$, y conocida $F(0)$, podemos calcular la nueva velocidad y la nueva posición de acuerdo a las expresiones anteriores: $x(1)$ y $v(1)$. De nuevo iterando el proceso, partiendo de

$$x(0), v(0)$$

resolvemos el problema:

En notación más compacta,

$$\begin{aligned} &\text{Inicial : } x_0, v_0 \\ &n \rightarrow x_n \rightarrow F_n \rightarrow v_{n+1} \\ &x_n, v_n \rightarrow x_{n+1} \end{aligned} \quad (2.9)$$

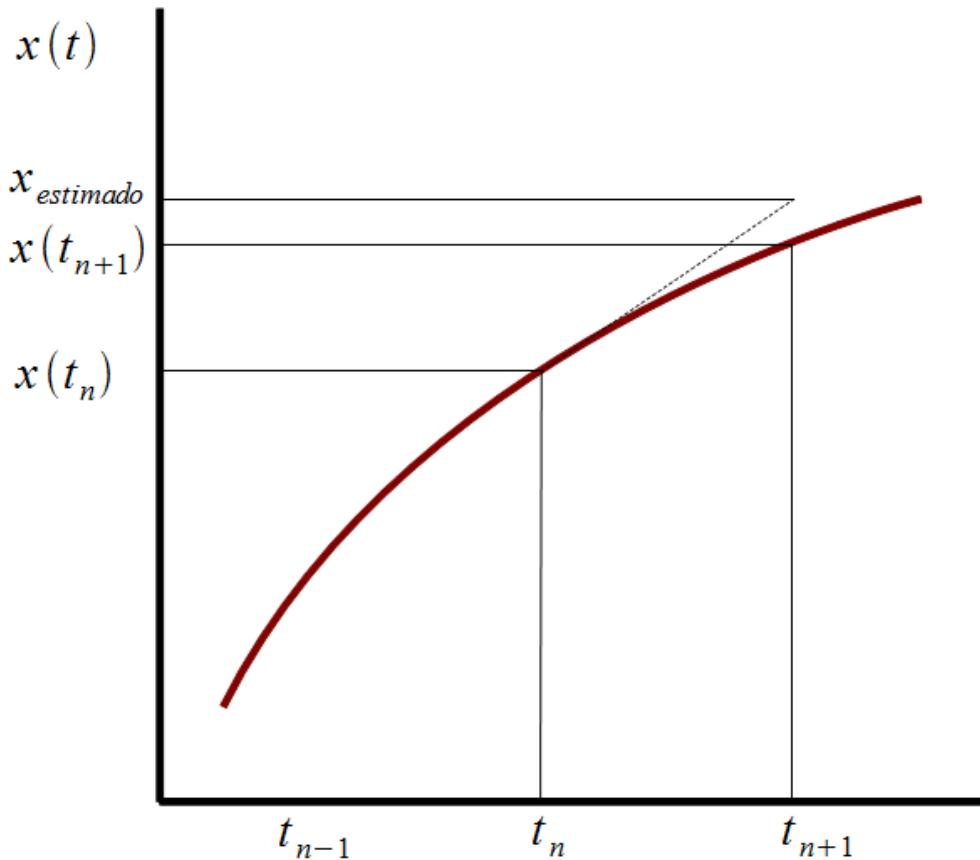


Figura 2.1: Estimación de $x(t_{n+1})$ a primer orden en el desarrollo de Taylor

Podemos esquematizar los pasos a seguir de forma gráfica, tal como puede verse en la figura 2.2. Su significado es el siguiente:

- En el momento t_n , conocemos la velocidad y la posición y por tanto la Fuerza, es decir

$$v(t_n), x(t_n), F(x(t_n))$$

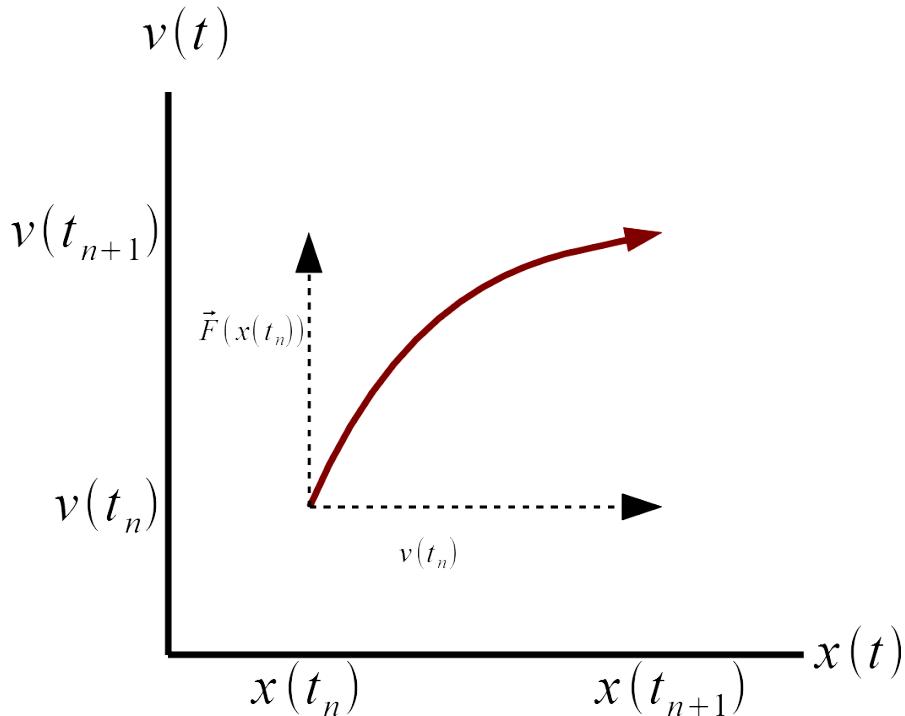
- Dada la velocidad en t_n , podemos conocer $x(t_{n+1})$. Esto está indicado en la gráfica 2.2 por la flecha horizontal.
- Conocida la Fuerza en t_n , sabemos la aceleración.
- Dada la aceleración y la velocidad en t_n , podemos conocer la velocidad en t_{n+1} . Esto está indicado por la flecha vertical.

Es decir: usando solamente datos en t_n , calculamos los valores en t_{n+1} . En forma más operativa, el algoritmo para pasar de un tiempo t_n al siguiente, t_{n+1} es:

$$\begin{aligned} v_{n+1} &= v_n + h \frac{1}{m} F_n \\ x_{n+1} &= x_n + h v_n \end{aligned} \tag{2.10}$$

2.1.1. Segmento de Código en C

A continuación damos un segmento de código en C para aplicar el algoritmo de Euler. Si la fuerza deriva de un potencial conocido $V(x)$, podemos calcular la energía mecánica, que debe

Figura 2.2: Paso de t_n a t_{n+1} en Euler

conservarse, con propósitos de control.

```

x=posicion_inicial //x_0
v=velocidad_inicial //v_0
for(i=0;i<I_MAX;i++)//Hacemos I_MAX pasos, en tiempo: h*I_MAX
{
    F=Fuerza(x); //Funcion conocida y programada correctamente.
    x+=h*v;
    v+=F*h/M;

    //Escribir datos en pantalla o archivo...
}
/* Calculo de la Energia Final */
Energia_p=V(x); // Energia Potencial supuesta conocida y programada
Energia_c=0.5*M*v*v;
Energia=Energia_c+Energia_p;

```

2.1.2. Limitaciones del método

Como hemos visto el método consiste en ir iterando el movimiento de la partícula, conocida la fuerza en cada punto. Es evidente que un pequeño error en un paso determinado supone una pequeña desviación en el siguiente punto. Estas desviaciones se irán acumulando y dependiendo del tipo de problema puede ocurrir que finalmente el resultado no se parezca en nada al resultado correcto. Este error procede de varios factores.

- **Intervalo mayor que cero entre iteraciones.** Las expresiones anteriores suponen que h es muy pequeño, pero puede ser que comparado con las cantidades que aparecen en el problema no lo sean tanto. El desarrollo de Taylor a primer orden deja de ser una buena aproximación y la trayectoria acumula errores.
- **Errores de precisión numérica.** En cada paso de la iteración el cálculo de la posición siguiente requiere muchas operaciones numéricas, en general de números pequeños ($h, \delta x, \dots$)

y otros normales (F, x, \dots) con lo cual debido a que la precisión del ordenador es finita, el error cometido puede ser grande, sobre todo si vamos a tiempos largos, pues el error es acumulativo.

- **Orden de variación de las cantidades.** En la opción elegida anteriormente, dada la posición, calculamos la velocidad en dicho punto, y a continuación la nueva posición con la vieja velocidad. Otro ordenamiento nos daría otra evolución.

2.1.3. Conservación de la Energía

Relacionado con lo anterior, es fácil comprobar numéricamente que con el algoritmo implementado, el sistema no conserva su energía mecánica en sistemas conservativos. Esto implica que el método es incorrecto, pero además esto es especialmente peligroso pues no nos permite controlar, como hemos comentado antes, la marcha del programa.

2.1.4. Reversibilidad temporal

En un sistema de Partículas, si una trayectoria es solución del movimiento, si invertimos el signo del tiempo, la trayectoria inversa también es solución del movimiento. Dicho de forma precisa, las leyes de Newton son invariantes bajo cambio del signo del tiempo.

Es imposible distinguir mirando el movimiento de una partícula si el tiempo va hacia delante o hacia atrás. Sin embargo sabemos que en un sistema de partículas tiende al desorden (aumentar la Entropía) y esto permite distinguir el sentido del tiempo: Dos gases separados inicialmente se mezclan irreversiblemente. Esto nos permite distinguir el sentido del tiempo: si la Entropía aumenta el sentido del tiempo es el actual; si la Entropía disminuye en el sistema completo, el tiempo tiene cambiado el signo. Esto supone una aparente contradicción con las leyes de Newton; la discusión de este punto sobrepasa los límites de este curso y sería necesario conocer las leyes básicas de la Mecánica Estadística.

En cualquier caso las leyes de Newton son siméticas bajo el cambio de signo del tiempo, pero el algoritmo de Euler no lo es. Cuando nos movemos de un punto al siguiente, usamos la velocidad y la fuerza en el punto actual. Si ahora vamos hacia atrás, usaremos para recorrer el mismo movimiento, la fuerza y velocidad del punto que antes era el final, por tanto no volveremos al mismo lugar ni llevaremos la misma velocidad, como puede verse en la figura 2.3.

Si partimos de t_n y vamos hacia adelante en el tiempo llegamos a x_{positivo} ; cuando vamos hacia atrás en el tiempo, partimos de t_{n+1} y llegamos a x_{negativo} , que serán diferentes en general dado que las pendientes en el punto de origen son diferentes. En el caso especial de que las pendientes sean iguales, si que coincidirán; pero en este caso trivial, la aceleración es nula, y el algoritmo de Euler es exacto.

2.1.5. Precisión y Errores acumulados

El algoritmo de Euler corresponde a resolver el sistema quedándonos hasta primer orden del desarrollo de Taylor como puede verse en la figura 2.1. Despreciamos el segundo orden del desarrollo en h , por lo que el error cometido en cada paso es $O(h^2)$. Cuando resolvemos el sistema durante un intervalo de tiempo t , realizamos t/h pasos, cada uno con un error de orden h^2 , por lo que en total el error sera de orden h . Decimos pues que Euler es un algoritmo de orden h .

Esto implica que para mejorar resultados, es necesario usar valores muy bajos de h ; pero valores muy bajos implican errores de precisión y redondeo importantes, que se acumulan en el tiempo. Para atacarlo en general se trata de usar más términos del desarrollo de Taylor.

El problema de la reversibilidad temporal y conservación de la energía es de más fácil solución como veremos en el siguiente apartado. En realidad la reversibilidad temporal la podemos mantener de forma exacta. La conservación de la energía sólo de forma aproximada. En todos los casos el error de redondeo no queda eliminado.

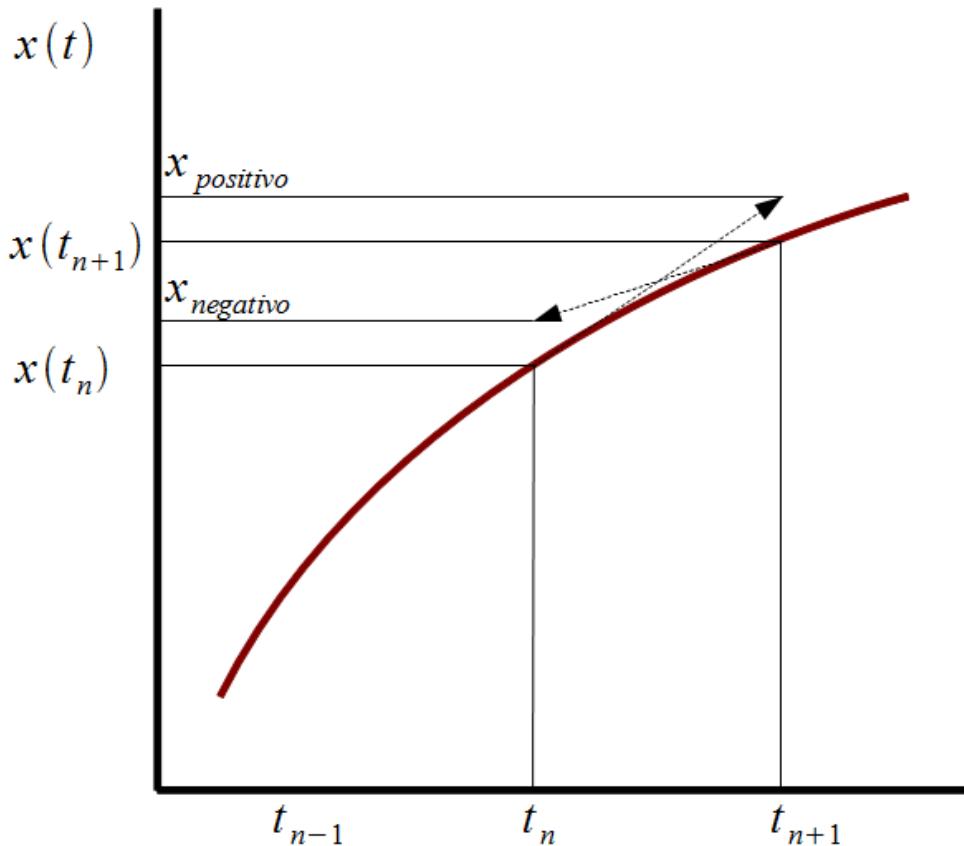


Figura 2.3: Error al cambiar el signo del tiempo

2.2. Método de Verlet

El método de Euler tiene varios problemas como ya hemos comentado.

En primer lugar aumentaremos el orden del desarrollo de Taylor, lo que aumentará la precisión.

Para evitar la no reversibilidad temporal trataremos de encontrar una forma de cambiar la posición y velocidades de modo que mirando hacia atrás o hacia delante en el tiempo no veamos diferencia.

Hay diferentes métodos para incorporar estas mejoras. Aquí implementaremos uno de los más simples y efectivos, el algoritmo de Verlet.

La idea central es que para movernos de un punto al siguiente lo hagamos en dos pasos, en lugar de en uno sólo como con Euler.

En el momento t_n , conocemos la velocidad, posición y por tanto la Fuerza:

$$v(t_n), x(t_n), F(x(t_n))$$

Hasta aquí estamos como con Euler. Ahora procedemos del siguiente modo:

1. Conocida la Fuerza en t_n , sabemos la aceleración. Entonces dada la velocidad en t_n , podemos conocer $v(t_{n+\frac{1}{2}})$. Es decir, $v(t_{n+\frac{1}{2}}) = v(t_n) + \frac{F(x(t_n)) h}{m} \frac{1}{2}$
2. Una vez conocida la velocidad en $t_{n+(1/2)}$, podemos usar esta velocidad en el “punto medio” para movernos de $x(t_n)$ a $x(t_{n+1})$. $x(t_{n+1}) = x(t_n) + v(t_{n+\frac{1}{2}}) h$
3. Finalmente usamos la fuerza en $x(t_{n+1})$, punto ya conocido, para cambiar la velocidad desde la conocida en $t_{n+\frac{1}{2}}$ hasta $v(t_n)$. $v(t_{n+1}) = v(t_{n+\frac{1}{2}}) + \frac{F(x(t_{n+1})) h}{m} \frac{1}{2}$

De modo que el algoritmo completo queda finalmente:

$$\begin{aligned} v(t_{n+\frac{1}{2}}) &= v(t_n) + \frac{F(x(t_n)) h}{m} \frac{1}{2}, \\ x(t_{n+1}) &= x(t_n) + v(t_{n+\frac{1}{2}}) h, \\ v(t_{n+1}) &= v(t_{n+\frac{1}{2}}) + \frac{F(x(t_{n+1})) h}{m} \frac{1}{2}. \end{aligned} \quad (2.11)$$

Un cálculo elemental permite reescribirlo como

$$\begin{aligned} v(t_{n+\frac{1}{2}}) &= v(t_n) + \frac{F(x(t_n)) h}{m} \frac{1}{2}, \\ x(t_{n+1}) &= x(t_n) + v(t_n) h + \frac{F(x(t_n)) h^2}{m} \frac{1}{2}, \\ v(t_{n+1}) &= v(t_n) + \frac{F(x(t_n)) + F(x(t_{n+1}))}{2m} h. \end{aligned} \quad (2.12)$$

La diferencia esencial respecto a Euler es que ahora para calcular la nueva velocidad usamos la Fuerza en los puntos inicial y final de la partícula. Si ahora vamos hacia atrás en el tiempo usamos los mismos valores de la fuerza y la velocidad, por supuesto en sentido contrario y volveremos exactamente al punto anterior. Esto preserva la simetría respecto al orden y por tanto reversibilidad temporal, y conserva la energía mecánica. La precisión de Verlet es orden h^2 , con lo que podemos mejorar la precisión sin ir a valores demasiado bajos de h .

En la gráfica 2.4 puede visualizarse mejor el algoritmo.

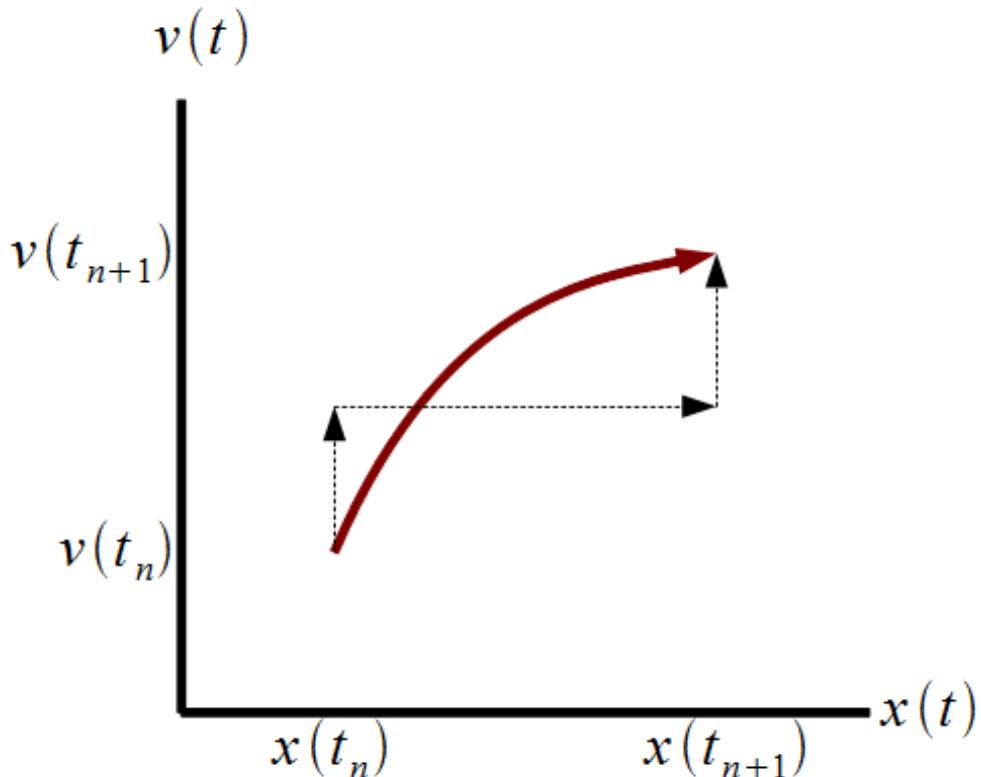


Figura 2.4: Paso de t_n a t_{n+1} en Verlet

2.2.1. Segmento de Código en C

A continuación damos un segmento de código en C para aplicar el algoritmo de Verlet, partiendo de una posición y velocidad inicial y realizando I_MAX pasos temporales de amplitud h .

```
x= posicion_inicial
v= velocidad_inicial
for(i=0;i<I_MAX;i++)
{
    F=Fuerza(x);
    v_temp=v+0.5*F*h/M;
    x+=v_temp*h;
    F=Fuerza(x); //Ya hemos cambiado x, esta es la nueva fuerza
    v=v_temp+0.5*F*h/M;
}
```

Conviene que el alumno compare con atención este código con el del algoritmo de Euler de la sección [2.1.1](#).

2.3. Ejercicios

2.3.1. El oscilador armónico

Aplicaremos los algoritmos anteriores a un sistema de una sola partícula, bien conocido, como es el oscilador armónico en $d = 1$.

Se proporciona el código para ambos algoritmos. El programa resuelve el modelo usando o bien el Algoritmo de Euler o bien Verlet, dependiendo de un `#define`.

Tanto éste como el resto de código de los apuntes, requiere un esfuerzo por parte del alumno para su total comprensión. El alumno debe intentar escribir el código antes de leer el proporcionado aquí. Sólo ante dificultades importantes o al final del trabajo, debe consultarse.

Consideraremos un oscilador armónico en una dimensión. La solución de este sistema puede ser obtenida analíticamente, y es bien conocida, por lo cual debe ser usada para controlar que el programa funciona correctamente.

Supongamos que el muelle esta anclado en la posición s , su longitud natural es l_0 y la constante de recuperación K , las ecuaciones del movimiento y la Energía cinética y potencial del sistema (importante para comprobar que todo vaya bien) son:

$$\begin{aligned} F(x(n)) &= -K(x(n) - l_0 - s), \\ E_c &= \frac{1}{2}mv(n)^2, \\ E_p &= \frac{1}{2}K(x(n) - l_0 - s)^2. \end{aligned} \tag{2.13}$$

Conocida la Fuerza en función de la posición, y dadas la posición y velocidad iniciales podemos aplicar el método de Euler o de Verlet para conocer la evolución del sistema.

El programa debe escribir en un fichero las posiciones, velocidades y energías en cada paso temporal. Dado que disponemos de la solución exacta, debemos comprobar que dicha solución coincide con la obtenida numéricamente, dentro de la precisión del algoritmo.

2.3.2. Sistema de Partículas en interacción gravitatoria

Consideraremos el caso de N partículas sometidas a interacción gravitatoria en tres dimensiones. El problema es una generalización del caso anterior. Escribir el código usando Euler y Verlet.

Para la simulación correcta debe tenerse en cuenta que cuando en un paso de tiempo se mueve una partícula, el resto de partículas ven la posición vieja; la nueva sólo la ven en el intervalo temporal siguiente.

Es importante comprobar que el programa es correcto usando sistemas sencillos. En todos los casos la energía mecánica debe conservarse. También debemos analizar la diferencia entre los dos algoritmos.

2.4. Problemas

2.4.1. Problema 1

Dibujar con `gnuplot` para el oscilador armónico la posición, velocidad, Energía Cinética, Energía Potencial y Energía Mecánica, en función del tiempo de simulación. Resolver el sistema analíticamente, y dibujar con `gnuplot` la solución analítica y numérica y comprobar que coinciden dentro de la precisión. Comprobar que la energía mecánica se conserva de forma aproximada en Euler y de forma exacta en Verlet, dentro de la precisión numérica.

2.4.2. Problema 2

Sea una superficie esférica de radio R y centrada en el origen. Sean dos puntos sobre la esfera \vec{r}_1, \vec{r}_2 . Calcular d : la distancia (más corta) moviéndonos siempre sobre la superficie esférica entre estos puntos. Escribir completamente una función que recibe \vec{r}_1, \vec{r}_2, R , y devuelve d .

Este problema corresponde a una pregunta de examen. Una posible solución correcta es

```
void Esfera(double* r1, double* r2, double R, double* d)
{
    double Prod_esc, Norm, theta;
    Prod_esc = r1[0] * r2[0] + r1[1] * r2[1] + r1[2] * r2[2];
    Norm = R*R;
    theta = acos(Prod_esc / Norm); //en el intervalo [0,PI]... que es correcto
    //para encontrar la distancia mínima, sin dar la vuelta por el otro lado...
    *d = R * theta;

}
```

2.4.3. Problema 3 (Ampliación)

Modificar el código del Ejercicio 1 para que tras N pasos de evolución, el sistema evolucione hacia atrás en el tiempo. Estudiar como de diferentes son las trayectorias. Usar Euler y Verlet.

2.4.4. Problema 4 (Ampliación)

Simular el sistema Tierra-Luna partiendo del código del ejercicio 2. El Sol puede ser considerado fijo. Hay que prestar especial atención a las unidades físicas y a las condiciones iniciales. Debe lograrse que se generen órbitas estables similares a las reales.

Escribir en un archivo las coordenadas con las trayectorias y dibujarlas con `gnuplot`. Puede escribirse también la energía de la Tierra y la Luna, y el momento angular del sistema (respecto al Sol) que debe conservarse.

2.5. Código en C

2.5.1. Conceptos de Programación: El preprocesador

Previa a la compilación propiamente dicha, se realiza una lectura del archivo de modo que todas aquellas instrucciones precedidas de una almohadilla (#), son tratadas de forma apropiada. Tras este *preprocesado* es cuando se pasa a compilar el programa.

Algunos ejemplos,

```
#define L 100
#include "a.txt"
```

La primera directiva susbstituye a partir de ese momento todas las variales del código L por 100. La segunda inserta en esa misma línea el contenido del archivo a.txt que debe estar en el mismo directorio desde el que compilamos. Hay muchas instrucciones disponibles para hacer los programas mas eficientes y fáciles de programar.

2.5.2. El Oscilador Armónico

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

#define X_M 640
#define Y_M 480

double M,soporte[2],K,l;
double h;

void Evoluciona_dt(double *, double *);
void Escribe_resultados(double,double,double);

main()
{
    double x,v;
    double tiempo;
    int i,j;

    int mesfr=1600;

/* Fijamos los puntos de soporte */

    soporte[0]=X_M/4; soporte[1]=Y_M/2;

    M=10.0; // masa del oscilador
    K=20.0; // Constante del muelle
    l=X_M/4;// longitud natural
    v=50.0; // velocidad inicial
    x=soporte[0]+X_M/4; //Posicion inicial
    h=0.0001; //Paso temporal de la ecuacion diferencial discreta

    tiempo=0;

    printf("# t          T          V          E_t          x          v\n");

    for(i=0;i<1000;i++)
    {
        for(j=0;j<mesfr;j++)
            Evoluciona_dt(&x,&v);

        tiempo+=mesfr*h;

        Escribe_resultados(tiempo,x,v);
    }
}

void Evoluciona_dt(double *x,double *v)
```

```
{
//#define EULER
#define VERLET

    double F;

#endif EULER
    F=-K*((*x)-soporte[0]-1);
    (*x)+=h*(*v);
    (*v)+=F*h/M;
#endif

#ifndef VERLET
    double v_temp;
    F=-K*((*x)-soporte[0]-1);
    v_temp=(*v)+0.5*F*h/M;
    *x+=v_temp*h;
    F=-K*((*x)-soporte[0]-1); //Ya hemos cambiado x, esta es la nueva fuerza
    (*v)=v_temp+0.5*F*h/M;
#endif
}

void Escribe_resultados(double time,double pos,double speed)
{
    double Energia,Energia_p,Energia_c;

    Energia_p=0.5*K*(pos-soporte[0]-1)*(pos-soporte[0]-1);
    Energia_c=0.5*M*speed*speed;
    Energia=Energia_c+Energia_p;

    printf("%f %-10.3lf %-10.3lf %-10.3lf %-10.3f  %-10.3f\n",
           time,Energia_c,Energia_p,Energia,pos,speed);
}
```

2.5.3. Sistema de partículas en interacción gravitatoria

```

#include <math.h>
#include <stdio.h>
//#define DEBUG

#define N_par 10

double h;
double G_Un;
double epsilon;
double M[N_par];

void Evoluciona_dt(double *x, double *y,double *z,double *v_x,double *v_y,double *v_z);
void Calcula_Fuerza(double *x,double *y,double *z,double *Fx,double *Fy,double *Fz);
void Escribe_resultados(double time,double *x,double *y, double *z,
double *v_x,double *v_y,double *v_z);

main()
{
    double x[N_par],y[N_par],z[N_par],v_x[N_par],v_y[N_par],v_z[N_par];
    double p_cdm_x,p_cdm_y,p_cdm_z,s_m,restar_x,restar_y,restar_z;
    double tiempo;
    int i,j;

    int mesfr=1000;
    G_Un=1000.0;

    p_cdm_x=p_cdm_y=p_cdm_z=s_m=0.0;

    for(i=0;i<N_par;i++)
    {
        M[i]=1.0+i;
        x[i]=(N_par*i + 10);
        y[i]=68*sqrt((double)i) + 13;
        z[i]=N_par*sin(6.28*i/N_par)+18;
        v_x[i]=(float)i/(float)(10*N_par);
        v_y[i]=2*v_x[i];
        v_z[i]=v_x[i];
        p_cdm_x+=M[i]*v_x[i];
        p_cdm_y+=M[i]*v_y[i];
        p_cdm_z+=M[i]*v_z[i];
        s_m+=M[i];
    }

    restar_x=p_cdm_x/s_m;
    restar_y=p_cdm_y/s_m;
    restar_z=p_cdm_z/s_m;

    for(i=0;i<N_par;i++) //Nos ponemos en el sistema de referencia centro de masas
    {
        v_x[i]-=restar_x;
        v_y[i]-=restar_y;
        v_z[i]-=restar_z;
    }

#ifndef DEBUG
    for(i=0;i<N_par;i++)
        printf("%f %f %f %f %f %f\n",x[i],y[i],z[i],v_x[i],v_y[i],v_z[i]);
#endif

    h=0.0001; //Paso temporal de la ecuacion diferencial discreta
    epsilon=1.0; //Para evitar la singularidad en el cero
    tiempo=0;

    printf("#      t          T          V          E_t\n");

    for(i=0;i<200;i++)
    {
        for(j=0;j<mesfr;j++)

```

```

    Evoluciona_dt(x,y,z,v_x,v_y,v_z);

    tiempo+=mesfr*h;
    Escribe_resultados(tiempo,x,y,z,v_x,v_y,v_z);
}
}

void Evoluciona_dt(double *x, double *y,double *z,double *v_x,double *v_y,double *v_z)
{
    int i;
    double Fx[N_par],Fy[N_par],Fz[N_par];
    double v_temp_x[N_par],v_temp_y[N_par],v_temp_z[N_par];

#define EULER
//#define VERLET

    Calcula_Fuerza(x,y,z,Fx,Fy,Fz); //Siempre sobre coordenadas Antiguas

#ifdef DEBUG
    printf("En evoluciona, h=%f\n",h);

    for(i=0;i<N_par;i++)
        printf("Evol: F[%d]=%f %f %f ,v=%f %f %f\n",
               i,Fx[i],Fy[i],Fz[i],v_x[i],v_y[i],v_z[i]);
#endif

#ifdef EULER

    for(i=0;i<N_par;i++)
    {
        x[i]+=h*v_x[i];
        y[i]+=h*v_y[i];
        z[i]+=h*v_z[i];
    }

    for(i=0;i<N_par;i++)
    {
        v_x[i] += (Fx[i]*h/M[i]);
        v_y[i] += (Fy[i]*h/M[i]);
        v_z[i] += (Fz[i]*h/M[i]);
    }
#endif

#ifdef VERLET

    for(i=0;i<N_par;i++)
    {
        v_temp_x[i]=v_x[i]+0.5*Fx[i]*h/M[i];
        x[i]+=h*v_temp_x[i];
        v_temp_y[i]=v_y[i]+0.5*Fy[i]*h/M[i];
        y[i]+=h*v_temp_y[i];
        v_temp_z[i]=v_z[i]+0.5*Fz[i]*h/M[i];
        z[i]+=h*v_temp_z[i];
    }

    Calcula_Fuerza(x,y,z,Fx,Fy,Fz); //Ya hemos cambiado r, esta es la nueva fuerza
    for(i=0;i<N_par;i++)
    {
        v_x[i]=v_temp_x[i]+0.5*Fx[i]*h/M[i];
        v_y[i]=v_temp_y[i]+0.5*Fy[i]*h/M[i];
        v_z[i]=v_temp_z[i]+0.5*Fz[i]*h/M[i];
    }
#endif
}

void Calcula_Fuerza(double *x,double *y,double *z,
double *Fx,double *Fy,double *Fz)
{

```

```

double r2,distance;
int i,j;

for(i=0;i<N_par;i++)
{
    Fx[i]=Fy[i]=Fz[i]=0.0;
    for(j=0;j<N_par;j++)
    {
        if(i==j)continue;

        r2=epsilon+
(x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j])+(z[i]-z[j])*(z[i]-z[j]);
        distance=sqrt(r2);
        r2=r2*distance;

        Fx[i]-=(G_Un*M[i]*M[j]*(x[i]-x[j])/r2);
        Fy[i]-=(G_Un*M[i]*M[j]*(y[i]-y[j])/r2);
        Fz[i]-=(G_Un*M[i]*M[j]*(z[i]-z[j])/r2);
    }

#ifdef DEBUG
    printf("F[%d]=%f %f %f\n",i,Fx[i],Fy[i],Fz[i]);
#endif
}
}

void Escribe_resultados(double time,double *x,double *y,
double *z,double *v_x,double *v_y,double *v_z)
{
    double r2,distancia;
    double Energia_c,Energia_p,Energia;
    int i,j;

    Energia_p=Energia_c=0.0;

    for(i=0;i<N_par;i++)
    {
        for(j=0;j<N_par;j++)
        {
            if(j==i)continue;

            r2=epsilon+
(x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j])+(z[i]-z[j])*(z[i]-z[j]);
            distancia=sqrt(r2);
            Energia_p-=(G_Un*M[i]*M[j]/distancia/2.0);
            /* Cuento todo dos veces !!! por eso /2*/
        }
        Energia_c+=0.5*M[i]*(v_x[i]*v_x[i]+v_y[i]*v_y[i]+v_z[i]*v_z[i]);

#ifdef DEBUG
    printf("i=%3d,r=(-10.31f,-10.31f,-10.31f),v=(-10.31f,-10.31f,
    -10.31f),Ec=-10.31f,Ep=-10.31f\n",
           i,x[i],y[i],z[i],v_x[i],v_y[i],v_z[i],Energia_c,Energia_p);
#endif
    }

    Energia=Energia_c+Energia_p;
    printf("%f %f %f %lf %f %f %f %f %f %f\n",
           time, Energia_c,Energia_p, Energia,x[0],y[0],z[0],x[1],y[1],z[1]);
    //getchar();
}

```

Comparando Euler y Verlet usando gnuplot

Para visualizar la diferencia entre los algoritmos de Euler y Verlet, es conveniente ejecutar el mismo sistema con las misma condiciones iniciales, una vez con cada uno de ellos, guardando el output de pantalla en dos ficheros diferentes.

Si usamos el código anterior, basta, cuando ejecutemos Euler, ejecutarlo en la linea de comandos

```
programa.exe > g_euler.dat
```

y para Verlet,

```
programa.exe > g_verlet.dat
```

Tendremos así dos ficheros que podremos visualizar con gnuplot. Con el siguiente fichero de comandos, puede visualizarse la diferencia en la conservación de la energía, y la diferencia en las trayectorias de la misma partícula (hemos elegido la partícula 0, pero puede ser cualquier otra) según el algoritmo,

```
plot "g_euler.dat" u 1:4 t "Energy Euler"
replot "g_verlet.dat" u 1:4 t "Energy Verlet"

pause -1

splot "g_euler.dat" u 5:6:7 t "Path Euler"
replot "g_verlet.dat" u 5:6:7 t "Path Verlet"
```

Una vez escritas estas instrucciones en un fichero (por ejemplo `compara.plt`), para cargarlo una vez dentro de gnuplot, basta escribir el comando `load "compara.plt"`.

Capítulo 3

Ecuaciones en Derivadas Parciales: Procesos de Difusión

La difusión es un proceso de gran importancia en Física, y uno de los fenómenos de no equilibrio más sencillos. Describe el transporte de calor o de materia. El fenómeno es bien conocido y originalmente estudiado por Newton. Tiene importantes connotaciones tecnológicas y es sencillo desde el punto de vista conceptual y numérico.

3.1. La Ecuación del Calor

Dado un cierto material, cada uno de sus puntos está a una temperatura dada en un momento inicial. Si suponemos conocidas todas sus características, la pregunta es ¿Cómo cambia la temperatura del material con el tiempo?

Para materiales homogéneos, la física del problema es enormemente simple, y obedece a la siguiente regla:

La temperatura de cada punto varía con el tiempo proporcionalmente al Laplaciano de la temperatura en dicho punto.

En términos matemáticos, la ecuación que rige la evolución de la temperatura T con el tiempo t en cada punto \vec{r} es:

$$\frac{\partial T(\vec{r}, t)}{\partial t} = k \left(\frac{\partial^2 T(\vec{r}, t)}{\partial x^2} + \frac{\partial^2 T(\vec{r}, t)}{\partial y^2} + \frac{\partial^2 T(\vec{r}, t)}{\partial z^2} \right) \quad (3.1)$$

Dada una situación inicial, es decir, dada la temperatura en todos los puntos del material en un instante fijo, el calor fluye de las partes calientes a las frías, es decir, las partes rodeadas de puntos más calientes, se calientan, las partes rodeadas de punto más fríos, se enfrian.

Veamos algunas propiedades elementales, para comprender mejor el problema físico, comprensión (siempre) necesaria antes de comenzar a escribir código.

3.2. Sistema Unidimensional

Consideraremos a partir de ahora un sistema unidimensional, en concreto una varilla extremadamente fina y de longitud finita. La solución del sistema es simple desde el punto de vista técnico, mientras que conceptualmente contiene todos los ingredientes necesarios para luego estudiar sistemas en dos o tres dimensiones.

3.2.1. Solución estacionaria

Para resolver la ecuación diferencial, necesitamos conocer el estado del sistema, su temperatura en cada punto, en un momento dado del tiempo. Consideremos el caso donde los bordes de la

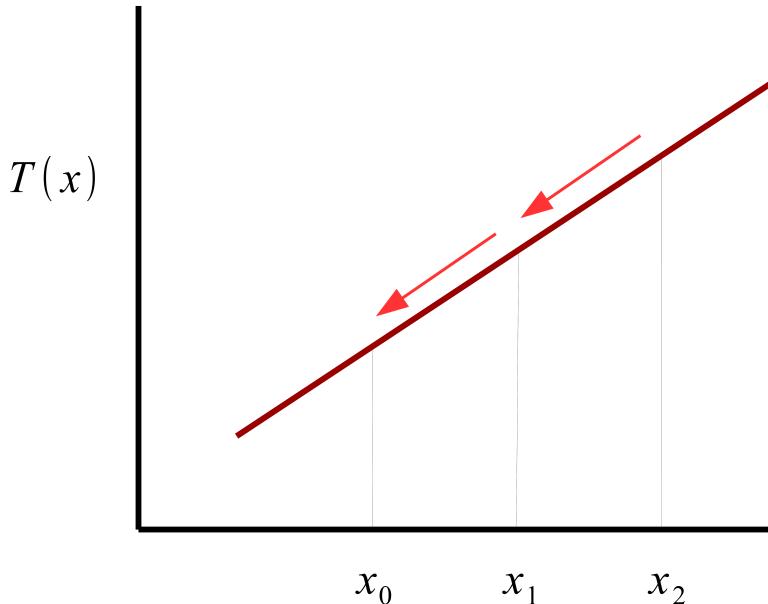


Figura 3.1: Situación de equilibrio en una barra unidimensional. En el punto x_1 existe flujo de calor según lo indicado por las flechas. El hecho de que la curva sea cóncava ($\partial^2 T / \partial x^2 > 0$) implica que la pendiente *por la derecha* es un poco mayor que por la izquierda, entonces recibe más calor (por la derecha) del que emite (por la izquierda) siendo pues el efecto neto el de acumulación de calor, y por tanto de subida de Temperatura.

varilla se mantienen a una temperatura fija durante todo el tiempo (será necesaria por supuesto una fuente externa en cada extremo). En este caso, sabemos que este sistema físico, independiente de la temperatura inicial del interior de la varilla, tiende a una solución estacionaria. Por ejemplo, supongamos una barra de longitud R , con un extremo a la temperatura T_0 y el otro a la temperatura T_1 . En este caso la ecuación anterior toma la forma

$$\frac{\partial T(\vec{r}, t)}{\partial t} = k \frac{\partial^2 T(\vec{r}, t)}{\partial x^2} \quad (3.2)$$

Si dejamos pasar mucho tiempo, la temperatura será constante en cada punto: no variará con el tiempo, con lo cual tendremos para la solución estacionaria ($t \rightarrow \infty$),

$$\frac{\partial T(x, t)}{\partial t} = 0 = k \frac{\partial^2 T(x, t)}{\partial x^2} \quad (3.3)$$

Entonces la solución es una temperatura con derivada espacial segunda nula. La solución en este caso, teniendo en cuenta que tenemos los extremos a temperaturas fijas, es

$$T(x, t) = T(x_0) + \frac{x}{R} (T(x_1) - T(x_0)) \quad (3.4)$$

es decir, la temperatura va cambiando linealmente desde un extremo a otro, y esa es una solución que se mantiene en el tiempo. Esto se puede comprobar inmediatamente: la derivada segunda de una recta es cero.

Esto se corresponde con lo esquematizado en la Figura 3.1. Es conveniente llamar la atención sobre el hecho de que estamos en un problema unidimensional, y que la figura bidimensional *no* corresponde con la forma del material. El material es una linea unidimensional, en concreto correspondiendo al eje x , y el Eje y representa la Temperatura en el punto x correspondiente. La variación lineal de la Temperatura corresponde con la situación de equilibrio, es decir a la situación Física de la barra conectada a dos temperaturas fijas (en este caso la menor es la del lado izquierdo), y tras un paso de tiempo de suficiente para que la temperatura de cada punto

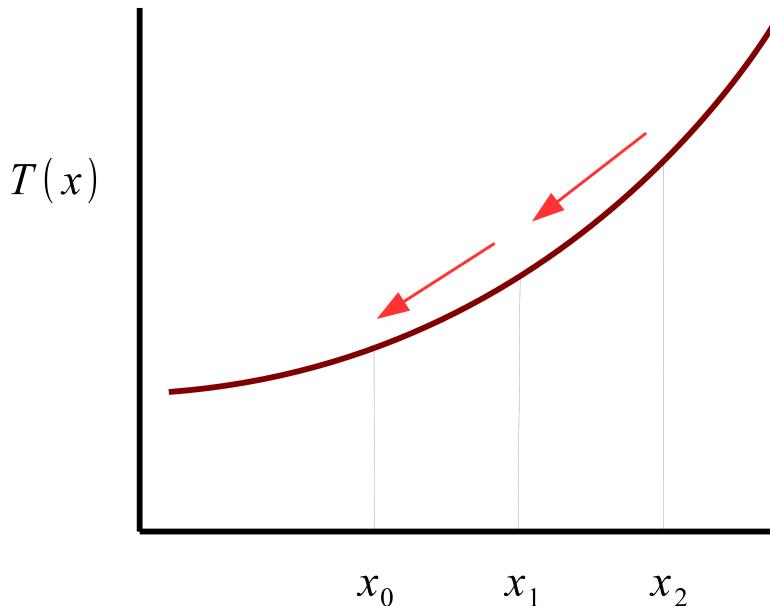


Figura 3.2: Situación fuera del equilibrio en una barra unidimensional. La temperatura en el punto x_1 varía, pues el flujo de calor entrante y saliente es diferente en general.

ya no varie en el tiempo. Efectivamente la derivada segunda de la recta es cero, y por tanto la temperatura no varia. Sin embargo esto no significa que no haya flujo de calor o energía a lo largo de la barra. De hecho el flujo de calor de un punto es proporcional al gradiente de temperatura en dicho punto (cambiado de signo), en este caso si nos fijamos en el punto x_1 , este punto recibe una cierta cantidad de calor proveniente del punto x_2 y a su vez emite calor (en la misma cantidad) hacia el punto x_0 . Existe pues flujo continuo de calor a lo largo de la barra, aunque la temperatura no cambia a lo largo de la misma.

Esta es la situación (de forma muy simplificada) a través de las paredes de las viviendas, por ejemplo. El punto x_2 es la cara interna de la pared (en invierno...) y x_0 la externa. Tras llegar al equilibrio ambas caras se mantienen a temperatura constante. La cantidad de calor que fluye de x_2 a x_0 depende del valor de k . De ahí la importancia de que k sea lo más pequeño posible (uso de materiales aislantes en las paredes).

Cuando no estamos en Equilibrio, la situación es diferente. Si en un momento dado de tiempo, el perfil de Temperatura a lo largo de la barra es el de la figura 3.2, la derivada segunda es no nula, la temperatura cambiará con el tiempo en el punto x_1 debido a que recibirá una energía diferente a la que emite.

Para conocer la solución completa y cómo el sistema llega al equilibrio, o cuando las condiciones de contorno son más complejas, debemos recurrir al cálculo numérico.

3.3. Discretización de la ecuación del calor

En el ordenador habitualmente se pasa a trabajar en lugar de con el espacio y tiempos continuos, con espacio y tiempo discretos. Es decir, el material es sustituido por un conjunto de puntos distribuidos en el espacio: en lugar de tener un continuo tenemos una red más o menos homogénea. Por ejemplo, en el caso de nuestra varilla, la sustituimos por un conjunto de L puntos ($n_0, n_1, n_2, \dots, n_{L-1}$). Si la longitud de la varilla era R metros, ahora estos puntos estarán separados por una distancia $a = R/(L - 1)$. Cuanto menor sea a , mejor es la aproximación de sustituir el continuo por el discreto.

También discretizamos el tiempo, de modo que ahora sera (t_0, t_1, t_2, \dots) , y el intervalo entre cada valor sera $\delta = t_{i+1} - t_i$



Figura 3.3: Discretización del Espacio para resolver la Ecuación del Calor

En esta forma discreta, la cantidad a estudiar, la Temperatura, es una función que depende de n_i , y que toma valores en general diferentes para cada valor del tiempo t_k , es decir

$$T(x, t) \rightarrow T(n_i, t_k) \quad (3.5)$$

Debemos ver ahora qué forma toma la ecuación diferencial en términos de las variables discretas. La forma de discretizar no es única. Aquí utilizaremos una forma sencilla que tiene la particularidad de preservar algunas simetrías.

Recordemos que la derivada de una función $f(x)$, se puede aproximar por

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h} \quad (3.6)$$

siempre que h sea suficientemente pequeño.

La derivada parcial con respecto al tiempo será

$$\frac{\partial T(x, t)}{\partial t} \approx \frac{T(x, t+\delta) - T(x, t)}{\delta} \quad (3.7)$$

y tomando tiempos sólo en los puntos discretos, y recordando que $t_i + \delta = t_{i+1}$, tendremos

$$\frac{\partial T(x, t)}{\partial t} \equiv \frac{T(n_i, t_{k+1}) - T(n_i, t_k)}{\delta} \quad (3.8)$$

Discreticemos ahora la derivada segunda espacial en la ecuación 3.1. Para ello basta recordar que la derivada segunda es la derivada de una derivada. Para la primera derivada espacial tendríamos,

$$\frac{\partial T(x, t)}{\partial x} \equiv \frac{T(n_{i+1}, t_k) - T(n_i, t_k)}{a} \quad (3.9)$$

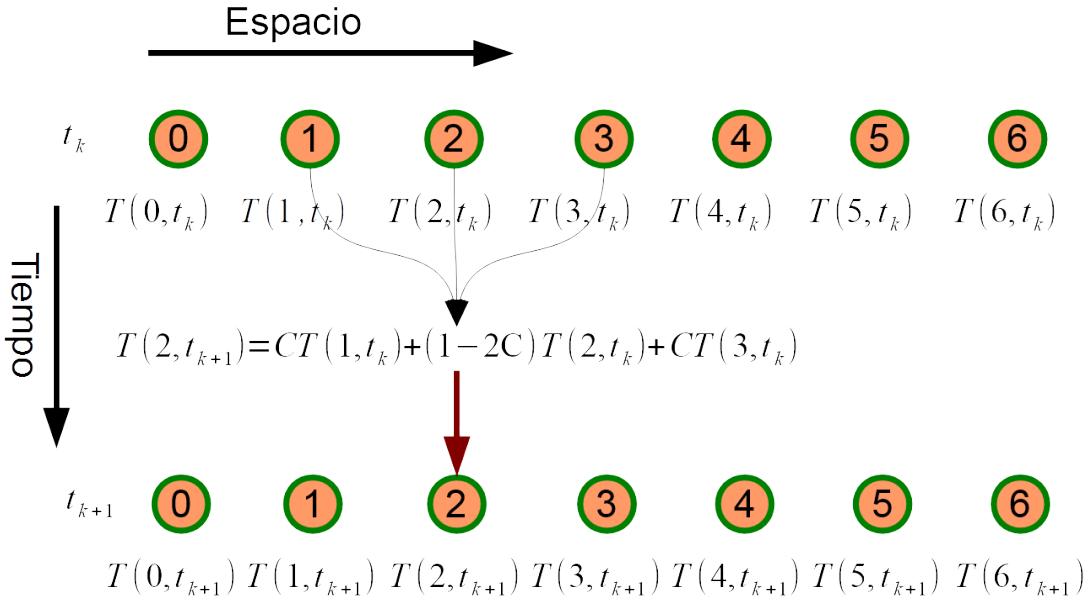
Ahora debemos derivar de nuevo la expresión anterior.

Para obtener una expresión simétrica es necesario introducir la derivada hacia adelante y la derivada hacia atrás; cuando hemos definido anteriormente la derivada de una función en la expresión 3.6, podríamos haberlo hecho también así

$$\frac{df(x)}{dx} \approx \frac{f(x) - f(x-h)}{h} \quad (3.10)$$

lo que podríamos interpretar como la *derivada hacia atrás*. Para definir una derivada simétrica, podríamos definir la media de ambas,

$$\frac{df(x)}{dx}|_{sim} \approx \frac{f(x+h) - f(x-h)}{2h} \quad (3.11)$$

Figura 3.4: Cálculo de la Temperatura $T(n_j, t_{k+1})$ usando solamente datos en t_k

Para obtener una derivada segunda simétrica, podemos hacer la diferencia de las dos derivadas, hacia delante y hacia atrás. Es decir construimos la derivada segunda como

$$\frac{\partial^2 f(x)}{\partial x^2} \approx \frac{1}{h} \left(\frac{f(x+h) - f(x)}{h} - \frac{f(x) - f(x-h)}{h} \right) = \frac{1}{h^2} (f(x+h) - 2f(x) + f(x-h))$$

En nuestro caso, prestando atención a que tenemos dos variables, pero sólo consideramos la variable x (el tiempo es fijo ahora), y que el espaciado es a , tendremos:

$$\frac{\partial^2 T(x, t)}{\partial x^2} \equiv \frac{T(n_{i+1}, t_k) + T(n_{i-1}, t_k) - 2T(n_i, t_k)}{a^2} \quad (3.12)$$

Con todo ello la ecuación del calor discretizada toma la forma

$$\frac{T(n_i, t_{k+1}) - T(n_i, t_k)}{\delta} = k \frac{T(n_{i+1}, t_k) + T(n_{i-1}, t_k) - 2T(n_i, t_k)}{a^2} \quad (3.13)$$

o en otra forma

$$T(n_i, t_{k+1}) = T(n_i, t_k) + \frac{k\delta}{a^2} (T(n_{i+1}, t_k) + T(n_{i-1}, t_k) - 2T(n_i, t_k)) \quad (3.14)$$

y agrupando los términos

$$T(n_i, t_{k+1}) = T(n_i, t_k) - 2 \frac{k\delta}{a^2} T(n_i, t_k) + \frac{k\delta}{a^2} (T(n_{i+1}, t_k) + T(n_{i-1}, t_k)) \quad (3.15)$$

o mejor aún

$$T(n_i, t_{k+1}) = (1 - 2 \frac{k\delta}{a^2}) T(n_i, t_k) + \frac{k\delta}{a^2} (T(n_{i+1}, t_k) + T(n_{i-1}, t_k)) \quad (3.16)$$

y finalmente

$$T(n_i, t_{k+1}) = (1 - 2C) T(n_i, t_k) + C (T(n_{i+1}, t_k) + T(n_{i-1}, t_k))$$

$$C = \frac{k\delta}{a^2} \quad (3.17)$$

Vemos que escrito así, la Temperatura en el momento de tiempo $k + 1$ sólo aparece en el miembro izquierdo. En esta forma ya tenemos la solución y el algoritmo numérico para resolver el problema:

Dado el valor de la temperatura en todos los puntos de la varilla en un momento dado de tiempo $T(n_i, t_k) \forall n_i$, mediante la ecuación anterior podemos calcular el valor de la temperatura en todos los puntos en el instante de tiempo siguiente $T(n_i, t_{k+1}) \forall n_i$. Iterando el proceso, tenemos la solución para cualquier instante de tiempo.

Ahora las condiciones iniciales son el conjunto de valores de la Temperatura para todos los puntos en un tiempo dado, es decir el conjunto $T(n_k, t_0) \forall k$. En la figura 3.4 puede verse una visualización del algoritmo.

3.3.1. Nota

El algoritmo propuesto aquí para resolver este problema es muy poco eficiente, en el sentido de que es necesario un gran tiempo de CPU (o número de iteraciones) para resolverlo. Existen algoritmos mucho más eficaces (leap-frog, Runge-Kutta, predictor-corrector o incluso algoritmos de path integral) pero son más complejos numérica y conceptualmente.

3.4. Nota sobre Punteros, vectores y funciones en C

C es un lenguaje de medio nivel, lo que significa que contiene pocas instrucciones pero puede manipular prácticamente todos los componentes, hardware y software de un ordenador. En la base de ello están los accesos a la memoria RAM, puertos, buses, etc. que se hace principalmente a través de punteros. Su uso presenta dificultades para quien no lo haya usado previamente. Los alumnos no familiarizados con estos conceptos deberían estudiar en este punto el Apéndice 3.10 o repasar el Tomo II.

3.5. Visualización del algoritmo

Recordemos la ecuación 3.17,

$$T(n_i, t_{k+1}) = C_0 T(n_i, t_k) + C(T(n_{i+1}, t_k) + T(n_{i-1}, t_k)), \quad C_0 = 1 - 2C \quad (3.18)$$

En cada momento debemos tener dos copias del sistema: el sistema en el momento actual, y a partir de esos datos calculamos la temperatura en el momento siguiente. Remarcar que para calcular los nuevos datos usamos sólo los viejos: no usamos los que se van calculando a cada paso. Sólo cuando acabamos de cambiar todos los puntos, el sistema nuevo pasa a ser el viejo para la siguiente iteración.

Para que el algoritmo sea convergente, el paso temporal debe ser suficientemente pequeño; en concreto es necesario que la constante C_0 sea positiva, o lo que es lo mismo $0 < C \ll \frac{1}{2}$. La razón es que estamos haciendo un desarrollo en serie, como es evidente de 3.7 o 3.14, siendo $C = k\delta/a^2$ el parámetro de dicho desarrollo. Sólo si esta cantidad es pequeña quedarnos con los primeros términos tiene sentido. Dentro de la región de convergencia, valores de C muy pequeños nos darán resultados más precisos, pero tiempos de convergencia (iteraciones) muy altos. Es interesante buscar los valores óptimos para C , que dan buenos resultados y en tiempos de ordenador bajos.

3.6. Detalles de implementación

Durante el proceso trabajamos con dos copias del sistema. Una mantiene las variables en el momento inicial, la otra en el momento final. Cuando calculamos las variables en el momento final, para el siguiente paso, éstas pasan a ser las variables iniciales. Puede visualizarse como se manejan los punteros en la figura 3.5.

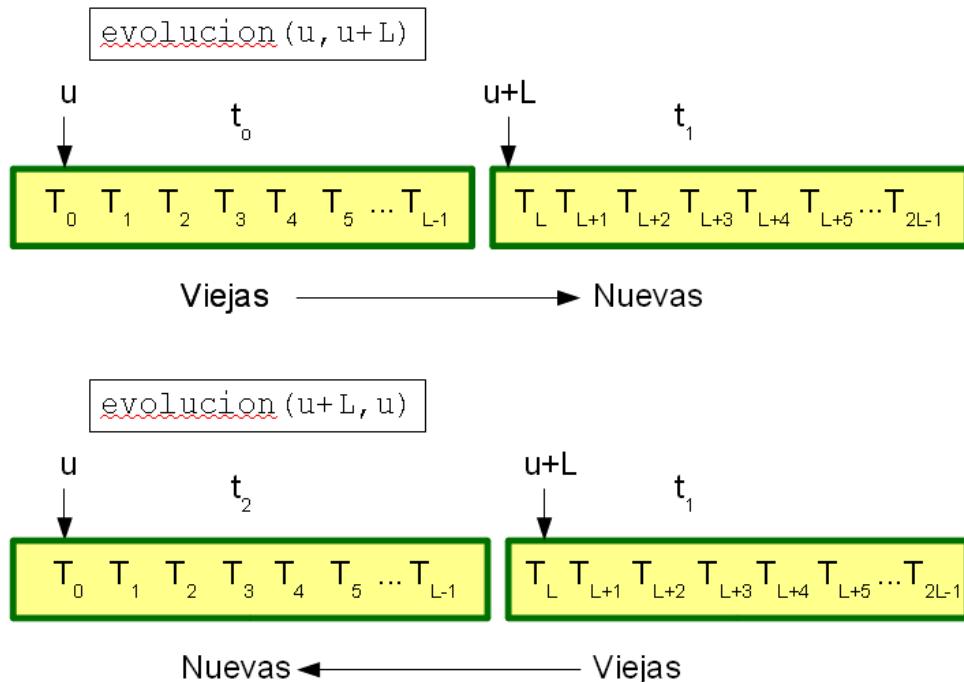


Figura 3.5: Representación gráfica del manejo de punteros en la función `evolucion()`

Las constantes se calculan al principio una sola vez. Los puntos inicial y final no se cambian pues los suponemos a temperatura fija.

Veamos el segmento de código correspondiente al núcleo del algoritmo,

```
#define L 100000 // tamaño de la malla

double u[2*L]; // u apunta a la primera copia
                // u+L apunta a la segunda copia

int Numero_Iteraciones;

Numero_Iteraciones=20000;

for (i=0;i<Numero_Iteraciones;i+=2) //En cada iteracion hacemos dos pasos
{
    evolucion(u,u+L);
    evolucion(u+L,u);
}
```

y la función `evolucion()` será:

```
double evolucion(double *uin, double *uout) // uin:sistema en t
                                              // uout: sistema en t+1
{
    int i;
    for (i=1;i<L-1;i++) // No tocamos los bordes
        uout[i]=C0*uin[i]+C*(uin[i-1]+uin[i+1]); //Ecuacion del Calor
}
```

3.7. Ejercicios

3.7.1. Ecuación del Calor en $d = 1$

Resolver el problema para una varilla con $T = 10$ en un extremo y $T = 100$, en el otro, ambas fijas. Considerar una longitud $L = 1$ metro. Dibujar la evolución de la Temperatura con el paso del tiempo en tres dimensiones: el eje X como el punto de la varilla, el eje Y como el tiempo, y en el Z la Temperatura correspondiente. Comprobar que la solución para tiempos grandes corresponde con la solución exacta.

3.8. Problemas

3.8.1. Problema 1

- ¿Cuál es el significado físico de la constante K ? ¿Qué unidades tiene?
- ¿Por qué la solución estacionaria no depende de K ? ¿Qué depende de K ?
- ¿Cuáles son las unidades de C ? ¿Cómo depende la llegada al equilibrio de las constantes K, δ, a ?

3.8.2. (Ampliación) Problema 2

Considerar un sistema bidimensional rectangular $R \times P$. Reescribir las ecuaciones diferenciales para este caso y escribir el código para simular el sistema.

3.9. Código en C

3.9.1. Conceptos de Programación: Debug, Punteros y Funciones

Conviene en el código de este capítulo fijarse especialmente en el uso de las instrucciones al preprocesador que nos permite en la fase de escritura de código, imprimir resultados parciales en la pantalla para verificar que todo es correcto, y que cambiando una definición pueden eliminarse de la compilación.

Aquí, como se comenta en el texto, se usan los punteros de forma relativamente compleja, y es necesario reflexionar sobre ello para comprender el detalle; igualmente usamos diferentes funciones con argumentos a los que debemos prestar atención.

3.9.2. Ecuación del calor en $d = 1$

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

#ifndef DEBUG

#define L 100      // Numero de puntos de la malla

void evolucion(double *uin, double *uout);
double C,C0;

main()
{
    int i;
    double u[2*L];   // *u apunta a la primera copia
                      // *u+L apunta a la segunda copia
    double L_fisica;
    double delta_t,k_difusion,a;
    int N_iter;

    delta_t=0.00001; // Paso temporal
    k_difusion=0.02; // Constante de difusion
    L_fisica=1.0;    // Tamaño fisico de la barra

    a=L_fisica/(L-1);
    C=k_difusion*delta_t/(a*a);
    C0=1-2*C;

    N_iter=1000000; //Numero de iteraciones del algoritmo

#ifndef DEBUG
    printf("a=%f,C=%f,C0=%f\n",a,C,C0);
#endif

    if(C0<=0)
    {
        printf("ERROR: C0 debe ser positivo y vale %f\n",C0);
        exit(1);
    }

    //Condiciones iniciales
    u[0]=u[L]=0; // Inicializo las dos copias 0J0
    u[L-1]=u[2*L-1]=1000;
    for(i=1;i<L-1;i++)
        u[i]=1;

    for (i=0;i<N_iter;i+=2)
    {
        evolucion(u,u+L);
        evolucion(u+L,u);
    }
}
```

```
for(i=0;i<L;i++)
    printf("%d %lf\n",i,u[i]);
}

void evolucion(double *uin, double *uout)
{
    int i;

    for (i=1;i<L-1;i++)
    {
        uout[i]=C0*uin[i]+C*(uin[i-1]+uin[i+1]);

#ifndef DEBUG
    printf("i=%d, u[%d]=%f u[%d]=%f, u[%d]=%f, uout[%d]=%f\n",
           i,i,uin[i],i-1,uin[i-1],i+1,uin[i+1],i,uout[i]);
#endif
    }
}
```

3.10. Apéndice: Punteros y vectores en C

Este Apéndice es una versión resumida del Capítulo correspondiente en el Tomo II, al que deberán referirse los alumnos que deseen una explicación más amplia de estos conceptos. Incluimos un rápido repaso de algunas ideas fundamentales.

3.10.1. Algunos conceptos básicos de Hardware

Veremos aquí las partes constitutivas de un ordenador personal, y una idea elemental de cómo funcionan cada una de las mismas.

Un ordenador personal (PC) consta esencialmente de:

- Unidad Central de Proceso (CPU)
- Memoria Central
- Unidades de Entrada (Discos Duros (HD), Teclado, Ratón...)
- Unidades de Salida (Discos Duros, Monitor, Impresoras...)

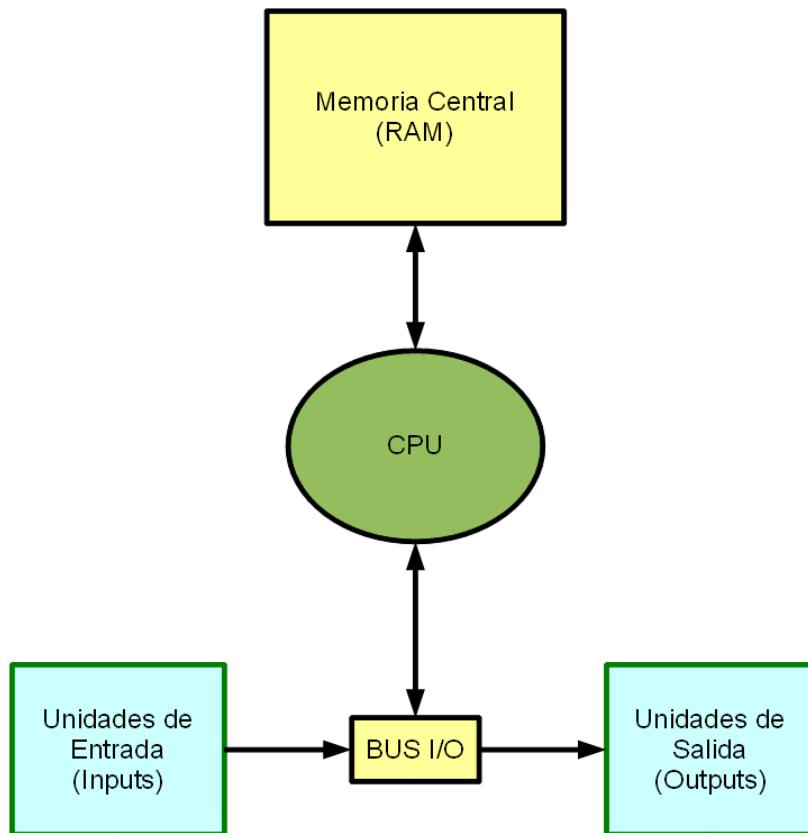


Figura 3.6: Esquema de un Ordenador

La CPU (Unidad Central de Proceso) es la parte inteligente del ordenador; prácticamente todo lo que sucede al ordenador, todos los cálculos, información, pasa por la CPU. La CPU o microprocesador es capaz de realizar varios miles de millones de operaciones elementales por segundo. La CPU sin embargo tiene una escasa capacidad de almacenamiento de información, y por ello los datos relevantes en cada momento están guardados en la Memoria Central, de rápido

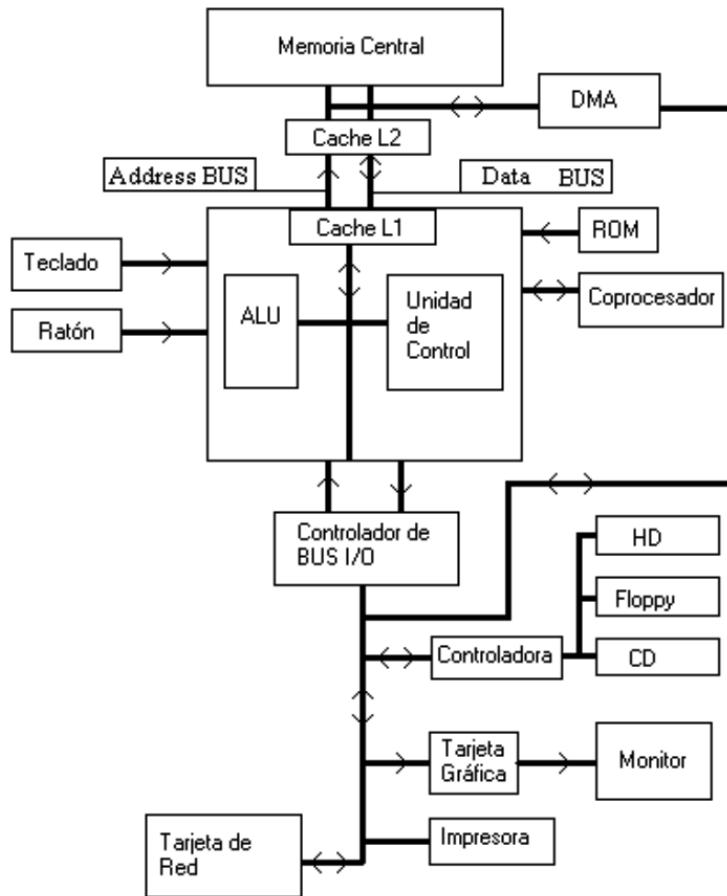


Figura 3.7: Diagrama de un Ordenador Personal

acceso. El ordenador se comunica con el exterior a través de las unidades de entrada y de salida, conectadas a la CPU y al resto del ordenador a través de un BUS de Input/Output. De este modo introducimos información en el ordenador a través del teclado, ratón..., y el ordenador nos devuelve información en el monitor, discos, impresoras, etc.

Esto es una visión muy simplificada, que podemos ahora detallar algo más, como puede verse en la figura 3.7. Hay que aclarar que para obtener una idea rápida, ha sido necesario omitir algunos aspectos (como por ejemplo la circuitería de Interrupciones) y simplificar otros (como por ejemplo el BUS de I/O) o incluso separar partes que a veces están integradas.

Pasemos ahora a describir cada una de las partes y sus funciones principales.

Memoria Central

Aquí es dónde el ordenador almacena los datos, programas y en general todo lo necesario para funcionar de forma rápida. Esta memoria se borra cada vez que el ordenador se apaga. Para entender su funcionamiento lo mejor es describir como está construida esta memoria físicamente.

La unidad básica es el chip de memoria DRAM (Dynamic Random Acces Memory) o solamente RAM. Un chip típico de Memoria contiene n patas en el *BUS de direcciones*, m patas en el *BUS de datos* y varias líneas de control, además de alimentación eléctrica.

El interior es un circuito integrado, y en lo que aquí nos interesa, basta saber que contiene 2^n bloques de m bits cada uno.

Cuando el procesador necesita un dato almacenado en la Memoria (Read) pone a 5 Voltios (1 lógico) la pata Read, y pone en el ADDRESS BUS la dirección en binario del dato solicitado.

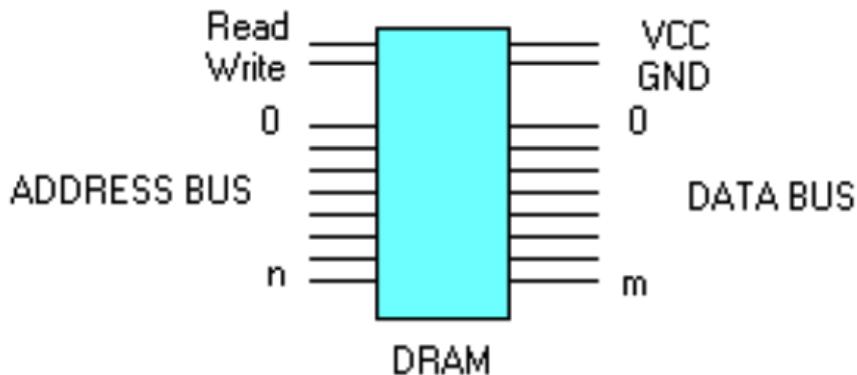


Figura 3.8: Esquema de un Chip de RAMI

El chip entonces, tras un cierto tiempo retraso, del orden de decenas de nanosegundos, (1 nanosegundo = 1 ns = 10^{-9} sg.) pone en el DATA BUS el dato almacenado en esa dirección. Para escribir un dato (Write), el procesador pone alta la señal de Write, y pone en el ADDRESS BUS la dirección donde quiere escribir y en el DATA BUS el dato que quiere escribir. El chip entonces almacena en esa dirección el dato presentado.

Si el ADDRESS BUS tiene n bits, las direcciones posibles son, como puede comprobarse fácilmente 2^n ; dado que en cada dirección hay m bits, la capacidad total de almacenamiento de este chip será ($m \times 2^n$).

3.10.2. ¿Qué son los punteros ?

Un puntero es una variable que contiene una dirección de memoria. En esa dirección se encuentra almacenada una variable determinada. Si una variable contiene la dirección de otra variable, se dice que la primera apunta a la segunda.

Pongamos un ejemplo introductorio al tema. La memoria de un PC esta numerada consecutivamente y cada posición en la memoria tiene asignada una dirección, de modo que cuando tenemos una variable `var`, en realidad el compilador traduce estos nombres a direcciones en la memoria, y cuando nos referimos a `var` lo que hace el programa es leer o escribir en la dirección que el compilador ha asignado a `var`.

En términos de la figura 3.8 la variable es lo que viaja en el Bus de Datos, y el puntero es lo que viaja en el Bus de Direcciones.

Muchas veces en C queremos saber la dirección en que esta almacenada una variable. Obtener dicha dirección es sencillo, basta usar el operador `&`. En concreto `&var` nos devuelve la posición de memoria donde está guardada la variable en cuestión. Hay que tener cuidado con lo que devuelve `&var`, pues devuelve un puntero (no un `int` o un `float`).

Entonces podemos hacer

```
int var;
int *p;
var=8;
p=&var;
```

La declaración `int *p` significa que `p` es un puntero a un entero: es decir el valor de `p` es la dirección de la memoria donde está guardado un `int`.

Para acceder a una variable, hasta ahora escribíamos el nombre de la variable, por ejemplo

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

accede a las variables **a**, **b**, **c** y hace la suma. Pero ¿cómo podemos acceder a una variable si lo que sabemos es su dirección en la memoria? Es decir, supongamos conocido el valor del puntero que apunta a una variable, ¿cómo accedo a esa variable?. En C es sencillo: si **p** es el valor del puntero, la variable almacenada en esa dirección se obtiene con el operador *****, por ejemplo ***p** nos devuelve lo que hay almacenado en **p**.

En el programa anterior si hago después

```
a=*p;
```

la variable **a** valdrá **var** (es decir 8).

Los punteros son importantes porque:

- Proporcionan los medios por los cuales las funciones pueden modificar sus argumentos.
- Se usan para soportar las rutinas de asignación dinámica de C.
- Su uso puede mejorar la eficiencia de ciertas rutinas, p. ej. en el indexamiento de arrays.

Por otro lado, son muy peligrosos porque, si apuntan a donde no deben, escribimos en memoria encima de alguna otra cosa y se producen errores que pueden ser gravísimos y difícil detectar de dónde provienen.

3.10.3. Variables puntero

Si una variable va a contener un puntero, debe declararse como tal:

```
tipo_varialbe *nombre;
```

tipo variable puede ser por ejemplo **int** o **double** e indica el tipo de variables a las que puede apuntar el puntero.

3.10.4. Los operadores de punteros: & y *

& devuelve la dirección de memoria de su operando.

***** devuelve el valor de la variable localizada en la dirección de memoria que dice su operando.

P. ej. **m=&cuenta;** pone en **m** la dirección de memoria donde está la variable **cuenta**. **q=*m;** pone en **q** el valor de **cuenta** (ya que lo que hay guardado en la dirección **m** es **cuenta**). En este ejemplo, deberíamos haber declarado **q** y **cuenta** como variables "normales" pero la **m** como un puntero (al tipo de **q** y **cuenta**) ya que va a contener una dirección; por ejemplo

```
. int q,cuenta,*m;
```

3.10.5. Expresiones de punteros

En general, las expresiones que involucran punteros se ajustan a las mismas reglas que cualquier otra expresión de C. Pero veamos algunos casos particulares:

- **Asignaciones de punteros.** P. ej. **int *p1,*p2; p1=p2;** es perfectamente válido.
- **Aritmética de punteros.** Sólo pueden usarse dos operaciones aritméticas con punteros: la suma y la resta de un puntero y un entero (no se pueden sumar punteros). Por ejemplo la expresión **p = p + 1 ;** donde **p** es un puntero a tipo **tip** hace que **p** apunte adonde apuntaba antes más el número de bytes que tenga el tipo **tip**. Es decir, si p. ej. ***p** es **int** de 2 bytes y **p** apuntaba a la dirección 2000, **p ++;** hace que apunte a la dirección 2002, y **p+2** apuntara a la dirección 2004. Para un **double**, **p++** apuntará 8 bytes más adelante. Otra forma de verlo es recordar que si **p** apunta a la dirección base de un cierto vector, por ejemplo a **v[0]**, entonces, **p+10** apunta a **v[10]**, independientemente del tipo del vector **v**.
- **Comparación de punteros.** Se pueden comparar dos punteros en una expresión relacional. P. ej. **if (p < q) printf ("p apunta a una direc. menor que q");**

3.10.6. Punteros y arrays

Los punteros y los arrays están relacionados: *El nombre de un array sin índice es un puntero al primer elemento del array.*

Es decir, si tenemos

```
char p[10];
```

son idénticos `p` y `&p[0]`, que valen la dirección donde está almacenado el primer elemento.

Por otro lado, cualquier variable puntero puede indexarse como si se tratase de un array.

P. ej. si hacemos

```
int *p, i[10];
p=i;
```

son válidas (y equivalentes) las dos expresiones siguientes

```
p[5]=100;
*(p+5)=100;
```

Esto puede generalizarse a más dimensiones. P. ej. si declaramos

```
int a[10][10];
```

es equivalentes `a` y `&a[0][0]` y el elemento (0,4) de `a` puede referirse como

`a[0][4]`

o como

`*(a+4)`,

y `a[1][2]` sería

`*(a+12)`.

Recordar que los índices comienzan en el cero y que corren primero los índices más a la derecha. Entonces el orden de almacenamiento del array `a` es

`[0][0], [0][1], [0][2], ..., [0][9], [1][0], [1][1], [1][2], ...`

Un programa que aclara este importante punto:

```
#include <stdio.h>

int a[3][5];
int i,j;
int *p;
main()
{
    for(i=0;i<3;i++)
        for(j=0;j<5;j++)
            a[i][j]=i*100+j;

    p=&a[0][0];
    for(i=0;i<15;i++)
        printf("\n p[%d]=%d",i,*p++);
}
```

Y la salida es:

```
p[ 0]= 0
p[ 1]= 1
p[ 2]= 2
p[ 3]= 3
p[ 4]= 4
p[ 5]= 100
p[ 6]= 101
```

```
p[ 7]= 102
p[ 8]= 103
p[ 9]= 104
p[10]= 200
p[11]= 201
p[12]= 202
p[13]= 203
p[14]= 204
```

Es decir, si hacemos:

```
int a[size1][size2];
x=a[j][k] ;
y=*( a + j*size2 + k );
```

las variables `x`,`y` tomarán el mismo valor. El alumno debe escribir y ejecutar el programa para una mejor comprensión de los conceptos.

Resaltamos, que la aritmética de punteros es frecuentemente más rápida que la indexación de arrays.

Se puede generar un puntero al primer elemento de un array simplemente especificando el nombre del array sin índice. P. ej.

```
int *p;
int ejemplo[10];
p=ejemplo;
```

asigna a `p` la dirección del primer elemento del vector `ejemplo[]`.

Puede obtenerse la dirección del primer elemento de un array como `ejemplo` o `&ejemplo[0]`.

3.10.7. Arrays de punteros

Los punteros pueden estructurarse en arrays como cualquier otro tipo de datos. P. ej. para declarar un array de 10 punteros a enteros `int *x[10];` Para asignar la dirección de una variable entera llamada `var` al tercer puntero del array hacemos:

```
x[2]=&var;
```

Para encontrar el valor de `var`, hacemos `*x[2]` .

3.10.8. Indirección múltiple

Se puede hacer que un puntero apunte a otro puntero que apunte a un valor de destino (podría hacerse más larga la cadena, es decir, puntero a puntero a puntero ..., pero no suele ser necesario). Los punteros a punteros hay que declararlos como tales, es decir:

```
tipo **nombre;
P. ej.
int x, *p, **q;
x=10;
p=&x;
q=&p;
printf ("%d", **q);
```

escribiría el valor de `x` (Es decir 10).

3.10.9. Funciones de asignación dinámica de C

La asignación dinámica es la forma en que un programa puede obtener más memoria mientras se está ejecutando. Como ya hemos dicho, el compilador asigna la memoria necesaria a un programa para almacenar el código y los datos. Esta cantidad de memoria es fija en general durante la ejecución. Sin embargo es posible durante la ejecución obtener más memoria para el programa para almacenar variables, datos, etc. También es posible liberarla.

Es muy útil para los que crean los programas de utilidades de los ordenadores (como procesadores de texto, bases de datos,...) porque, como no todos los ordenadores ni en todas las situaciones se tiene la misma RAM disponible, mediante la asignación dinámica se puede sacar el máximo rendimiento. Para asignar más memoria se usa la función `malloc()` y para devolverla la función `free()`.

`malloc(int)` reserva en la memoria el número bytes solicitado.

Por ejemplo, para reservar espacio para un vector de enteros de 100 elementos, haremos

```
int *ip
ip=(int)malloc(100*sizeof(int));
```

el argumento es siempre en bytes; para usarlo como puntero a enteros hay que convertirlo como se hace en el ejemplo. Ahora `ip[0]` (o `*ip`) contiene el primer elemento del vector, `ip[1]` (o `*(ip+1)`) el segundo, etc...

Importante: Notar que NO hemos declarado `int ip[100]`. Si lo hubieramos declarado así, el compilador ya habría reservado la memoria, y no tendríamos que usar `malloc()`.

Una vez usada la memoria, hay que liberarla, realizando una llamada a la función `free(ip);`

Si se hacen muchas llamadas a `malloc()` sin liberar el espacio, puede llenarse la memoria, y eventualmente bloquear el sistema.

3.11. Apéndice: Funciones en C

Las funciones son los bloques constructores de C y el lugar donde se da toda la actividad del programa.

3.11.1. Forma general de una función

La forma general de una función (su declaración, su definición, la forma de decir lo que hace) es

```
espec_de_tipo nombre_de_la_función(lista de parámetros )
{
    cuerpo de la función
}
```

El `espec_de_tipo` dice el tipo de valor que devuelve la sentencia `return` de la función. Si no se especifica ningún tipo, el compilador asume que se devuelve un `int`. Si no devuelve nada, entonces hay que declararla `void`. La lista de parámetros es la lista de nombres de variable separados por comas con sus tipos asociados que reciben los valores de los argumentos cuando se llama a la función. Una función puede no tener parámetros, pero los paréntesis hay que ponerlos e indicar en la lista de parámetros `void`, por ejemplo `int a(void)`. Notar que al definir la función, no hay que acabar la linea con `;`.

Para cada uno de los parámetros de la declaración debe decirse de qué tipo es (aunque haya varios del mismo tipo), es decir, p. ej.

```
float fun( int x, int y, int z, float w){ cuerpo}
```

En el cuerpo se dice todo lo que hace la función y puede haber llamadas a otras funciones, incluso a sí misma (recursividad).

Una vez declarada la función podemos llamarla desde otras funciones. (Notar que el programa principal o `main()` tambien es una función)

Cuando una función1 llama a una función2, el programa ejecuta lo que se dice en la función2 y, cuando termina, el control (la ejecución) vuelve a la función1 al mismo punto donde se había quedado (es decir, al punto dónde se había producido la llamada). Para llamar a una función lo único que hay que hacer es escribir su nombre y, entre paréntesis y separados por comas, tantos argumentos como parámetros tenga la función (los argumentos han de ser del mismo tipo que los parámetros respectivos).

P. ej. para llamar a la función definida arriba haremos :

```
int var;
float z;
var=80;

z=fun( 6, var, 357, 4.58);
```

3.11.2. Reglas de ámbito de las funciones

Son las reglas que controlan si un fragmento de código conoce o tiene acceso a otro fragmento de código o de datos. En C cada función es un bloque discreto de código. El código que comprende el cuerpo de una función está oculto al resto del programa y, a no ser que se usen datos o variables globales, no puede ser afectado por otras partes del programa ni afectarlas (salvo con una llamada a la función, claro).

En C, todas las funciones están al mismo nivel de ámbito, es decir, no se puede definir una función dentro de otra.

3.11.3. Argumentos de funciones

Como ya hemos visto, si una función va a usar argumentos, debe declarar variables que acepten los valores de los argumentos. Estas variables se llaman parámetros formales de la función. Al hacer una llamada a la función los argumentos deben ser del mismo tipo que los parámetros declarados.

Los valores de los argumentos pueden modificarse en la función que los usa, pero **NO** se modifican en ningun caso en la la función desde donde se realiza la llamada.

Sí que hay una forma de que una función pueda modificar los valores de una variable: usar como argumento, no la variable, sino un puntero a ella. Así, el argumento no cambia, ya que es el puntero, es decir la dirección de memoria, pero sí que puedes cambiar lo que esté allí guardado, que es la variable.

El pasar como argumento una variable "normal" (no un puntero) se denomina llamada por *valor*, mientras que si se pasa un puntero se dice llamada por *referencia*.

3.11.4. Llamada por valor

La llamada por valor copia el valor del argumento en el parámetro formal de la función al entrar en ella en otra variable que ya es diferente y en la función se trabaja con esa nueva variable y, por tanto, el valor del argumento no se modifica.

Ejemplo:

```
# include <stdio.h>
int cuadrado(int x);
void main(void)
{
    int t=10;
    printf ("%d %d", cuadrado(t), t);
}
int cuadrado(int x)
```

```
{
x=x*x;
return x;
}
```

Esto imprime en pantalla 100 10 .

Comentarios al ejemplo:

- La segunda línea no es la definición de la función `cuadrado()`, es lo que se llama un prototipo y que comentaremos luego. La definición de dicha función son las dos últimas líneas (donde está el cuerpo de la misma).
- El valor del argumento `t` no se ha modificado (sigue valiendo 10) aunque según la definición de la función parece que debería cambiar (se pone `x=x*x;`), pero lo que se modifica es la copia que se ha hecho de `t`, es decir, `x`. Notar que el resultado sería el mismo aunque en la función usáramos la variable `t` en lugar de `x`.

3.11.5. Llamada por referencia

Se pasa como argumento la dirección de una variable `y`, por tanto, el valor de la variable sí que puede ser modificado por la función (escribiendo en esa dirección).

Para usar esa forma, hay que declarar los parámetros de la función como punteros. Y al hacer la llamada a la función lo que hay que poner como argumento es una dirección.

P. ej. una función que intercambia los valores de dos variables:

```
void inter(int *x,int *y);      /Esto es el prototipo */
void main(void)
{
    int x,y;
    x=10;    y=20;
    inter(&x, &y);
    printf("\n x=%d, y=%d",x,y);
}
void inter(int *x, int *y)
{
    int temp;
    temp = *x;
    *x=*y;
    *y=temp;
}
```

La salida en la pantalla será:

`x=20, y=10`

3.11.6. Llamada a funciones con arrays

En C no se puede pasar un array completo como argumento a una función. Pero se puede pasar un puntero a un array, especificando el nombre del array sin índice.

Ejemplo:

```
int i[10];
funcion( i );
```

Si una función recibe un array unidimensional, se puede **declarar** el parámetro formal de tres formas: como un puntero, como un array delimitado o como un array no delimitado.

Ejemplo:

```
funcion( int *x ) {.....}
funcion( int x[10] ) {.....}
funcion( int x[] ) {.....}
```

pero esto es en la declaración, no en el uso (que, como ya hemos visto, hay que hacerlo con un puntero). Eso sólo avisa al compilador de que a esa función se le pasará un puntero, pero nada más; igual hubiera sido poner

```
funcion( int x[1000] ){...}
```

pues en la función no se reserva espacio para el array. Es responsabilidad del programador no salirse del rango correcto.

La llamada a una función con arrays es un caso particular de la llamada por referencia. Ya vimos que una función que parece que tiene como parámetro (y por tanto como argumento) un array, lo que tiene en realidad es un puntero (a su primer elemento). Por tanto, el array puede ser modificado por la función.

Ejemplo

Programa que escribe 10 números, guardados en el vector `t[]`, en la pantalla.

```
#include <stdio.h>
void mostrar(int num[10]);      /*prototipo*/
void main(void)
{
    int t[10],i;
    for ( i=0; i<10; i++)  t[i]=i;
    mostrar[t];      /*esta línea no está dentro del bucle*/
}
void mostrar(int num[10])
{
    int i;
    for ( i=0; i<10; i++)  printf("%d  ", num[i]);
}
```

En la definición de la función `mostrar(int num[10])` el compilador ya sabe que lo que se pasa es un puntero. Igualmente se podría haber definido así:

```
void mostrar(int num[])
{
    int i;
    for ( i=0; i<10; i++)  printf("%d  ", num[i]);
}
```

y también así, que es la recomendada:

```
void mostrar(int *num)
{
    int i;
    for ( i=0; i<10; i++)  printf("%d  ", num[i]);
}
```

Si el argumento no es un array sino sólo un elemento de un array, se trata como cualquier otra variable simple.

3.11.7. Punteros a funciones

Aunque una función no es una variable, tiene una posición física en memoria y, por tanto, pueden definirse punteros a funciones.

Se declara p. ej.

```
int (*p)();
```

y para poner en p la dirección de una función f() se pone p = f ; sin los paréntesis.

Esto puede usarse para pasar distintas funciones como argumentos de otra función, pasando sus punteros correspondientes (si la función que se quiere pasar es siempre la misma puede pasarse como argumento simplemente sin poner el paréntesis).

En el Tomo II se encuentran ejemplos explicados con todos estos conceptos.

Capítulo 4

Números aleatorios

Los números aleatorios (o *random* en inglés) juegan un papel central en los algoritmos numéricos y las simulaciones en ordenador.

El azar está presente en la Naturaleza de muy diversas formas, en la Física, la Biología, la Ingeniería... incluso en los sistemas sociológicos, económicos, etc.

Si bien las leyes de Newton son deterministas, cualquier sistema mecánico con un elevado número de grados de libertad es tan complejo que una descripción exacta es imposible, dando lugar esta incertidumbre a comportamientos impredecibles en la práctica, que son similares a comportamientos con aspectos random. Por fortuna en sistemas con un elevado número de grados de libertad, no es necesaria una descripción en detalle, basta con estudiar propiedades generales, promedio.

La termodinámica y la mecánica estadística están basadas en esta descripción promedio de la naturaleza. Lo mismo muchos otros problemas aparentemente lejanos a estos tratamientos, como el estudio del tráfico rodado o de personas, la propagación de enfermedades o de un rumor en la sociedad o en Internet. En estos casos no nos importa tanto los detalles concretos, individuales (por otra parte impredecibles) sino comportamientos globales. Para estudiar todos estos fenómenos los números aleatorios juegan un papel de gran importancia.

Incluso existen fenómenos intrínsecamente aleatorios como la lotería, o los resultados obtenidos tras un proceso de medida en los fenómenos dónde es necesaria la Mecánica Cuántica para su correcta descripción.

También el estudio de problemas en absoluto aleatorios, necesitan de números de este tipo para ser resueltos de forma eficiente. Por ejemplo para evaluar ciertas integrales multidimensionales, uno de los métodos más eficientes es el método de Monte Carlo, llamado así por el uso de números al azar, azar omnipresente en el famoso casino del mismo nombre.

Este capítulo tiene un gran número de detalles que si bien no son especialmente complicados, conviene prestarles atención y seguir los apuntes de forma ordenada para asimilar todos los puntos. Además conviene recordar que existe en paralelo una asignatura de estadística donde se profundiza en los aspectos más teóricos de los conceptos introducidos aquí.

4.1. Función densidad de probabilidad

Una función real de variable real $p(x)$ se dice una *densidad de probabilidad*, si cumple

$$p(x) \geq 0 \quad \forall x; \quad \int_{-\infty}^{+\infty} p(x)dx = 1 \quad (4.1)$$

Si x toma valores discretos, tendremos

$$p(x_k) \geq 0 \quad \forall k; \quad \sum_k p(x_k) = 1 \quad (4.2)$$

Estudiemos dos ejemplos.

Primero, nos situamos en una carretera con un aparato que nos permite medir con gran precisión la velocidad de los vehículos que pasan. Iremos obteniendo diferentes velocidades v (el módulo de la velocidad). Esta variable es continua, pues pueden aparecer vehículos con velocidades arbitrarias en un cierto intervalo (aproximadamente $[0, 220] \text{ Km/h}$ por ejemplo). Si realizamos muchas medidas, se irán adaptando a la distribución real que nos da la probabilidad de encontrar cada velocidad $p(v)$. Encontraremos muchos vehículos con velocidades en torno a 120 km/h, muy pocos con velocidades inferiores a 50 o superiores a 180.

En el segundo, consideremos el lanzamiento de un dado con seis caras, sin trucar. En este caso las variables pueden ser $x = \{1, 2, 3, 4, 5, 6\}$, y ningún otro, es decir, tenemos una variable discreta. Es evidente que en este caso $p(x) = 1/6 \quad \forall x$

4.1.1. Distribuciones Uniformes

En la base de la generación de números random en el ordenador, está la densidad de probabilidad *plana* o *uniforme* en un intervalo: la probabilidad de que aparezca un número es igual para cualquier valor en el intervalo. Por ejemplo, la lotería tiene una densidad de probabilidad plana: cualquier número del sorteo (en contra de lo que algunos piensan) tiene la misma probabilidad de aparecer.

Distribución Uniforme Continua

El valor de x es continuo; consideremos como ejemplo el intervalo $[0, 2]$; la distribución plana tendrá la misma probabilidad en cada punto, por tanto,

$$p(x) = C \quad (4.3)$$

El valor de C viene fijado por el hecho de que la distribución debe estar normalizada,

$$\int_0^2 C dx = 1 \Rightarrow 2C = 1 \Rightarrow C = \frac{1}{2} \quad (4.4)$$

es decir

$$p(x) = \frac{1}{2} \quad (4.5)$$

Remarcamos aquí que $p(x)$, aunque de valor constante, es una función de x como cualquier otra, aunque su valor no depende de x .

Nota: En el caso de una distribución continua, no tiene mucho sentido decir

¿Cuál es la probabilidad de que que salga el número 1.30?

Estrictamente la respuesta a esa pregunta es 0: la probabilidad de encontrar un número real *exacto* es cero. Para ilustrar esto consideremos el siguiente Ejemplo:

Sea un dado de 10 caras. Construimos un número Real así: las tiradas pares van a la parte entera, y las tiradas impares a la parte decimal. Nos preguntamos ahora sobre la probabilidad de que salga un cierto número real exacto, para lo que fijamos infinitos dígitos consecutivos. Evidentemente la probabilidad de que salga ese número concreto es 0.

Lo que tiene sentido es preguntarnos

¿Cuál es la probabilidad de que salga un número entre 1.30 y 1.31?

La respuesta a esta pregunta es

$$p(x) \in [1.30, 1.31] \approx p(1.30) \times 0.01 \quad (4.6)$$

o de forma general, la probabilidad de que x esté entre x y $x + dx$ viene dada por

$$p(x)dx$$

Para un intervalo finito $[a, b]$, tendremos,

$$p(x \in [a, b]) = \int_a^b p(x)dx. \quad (4.7)$$

En la figura 4.1 se ilustra este concepto.

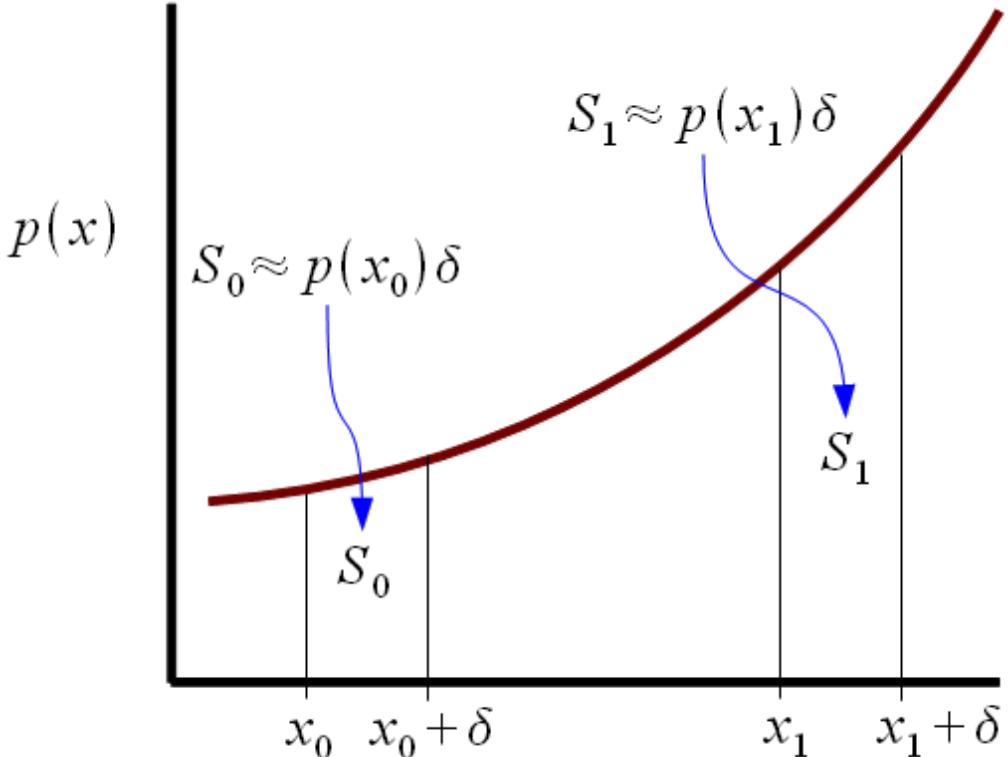


Figura 4.1: En una distribución continua la probabilidad de que aparezca un valor en el entorno de x es proporcional a $p(x)$.

Distribución Uniforme Discreta

La variable x sólo toma una serie de valores discretos $\{x_k\}$. Supongamos como ejemplo que x toma valores enteros en el intervalo $[0, 99]$; la distribución uniforme significa que la probabilidad de cualquier número en el intervalo será idéntica, por tanto constante,

$$p(x_k) = C \quad (4.8)$$

La normalización nos permite fijar el valor de C ,

$$\sum_0^{99} C = 1 \Rightarrow 100C = 1 \Rightarrow C = \frac{1}{100} \quad (4.9)$$

y por tanto

$$p(x_k) = \frac{1}{100} \quad \forall k \quad (4.10)$$

Ahora la probabilidad de que aparezca un número es directamente la probabilidad de ese valor. En este caso discreto podemos decir $p(x = 3) = \frac{1}{100}$, o que $p(x \in \{1, 5, 54\}) = \frac{3}{100}$. Si consideramos un intervalo, tendremos en este caso $p(x \in [3, 8]) = \frac{6}{100}$

4.1.2. Distribuciones no uniformes

La distribución uniforme es básica en muchas aplicaciones directamente, pero sobre todo es importante pues con ella podemos generar otros tipos de distribuciones no uniformes, como veremos en el Capítulo siguiente. Las distribuciones random no uniformes aparecen en amplias ramas de la programación.

Distribuciones no uniformes continuas

La variable aleatoria es continua. Un ejemplo usual son distribuciones polinómicas en un intervalo acotado,

$$p(x) \propto ax^2 + bx, \quad x \in [\alpha, \beta] \quad (4.11)$$

También son habituales las distribuciones de Poisson,

$$p(x; k) \propto \frac{1}{k!} x^k e^{-x}, \quad k \in \mathbb{N}, \quad x \in [0, \infty]. \quad (4.12)$$

Un caso con algunas particularidades es el de la distribución de Cauchy o Lorentziana,

$$p(x) \propto \frac{1}{x^2 + a^2}, \quad x \in [-\infty, \infty] \quad (4.13)$$

Distribuciones no uniformes discretas

Similar al caso anterior pero la variable aleatoria sólo toma valores discretos. Un caso simple es la probabilidad de que al elegir una persona española al azar haya nacido en una cierta provincia (numeradas éstas convenientemente). También podemos considerar el caso de la distribución polinómica anterior (4.11) pero donde ahora el valor de la variable x queda limitado a números enteros positivos en el intervalo $[0, 100]$.

Nota: Es importante remarcar que no hay que confundir que una densidad de probabilidad sea uniforme con el hecho de que sea una densidad de probabilidad. En las distribuciones no uniformes, pueden aparecer también cualquier valor en su rango de forma aleatoria e impredecible, pero lo que ocurre es que unos números aparecen con mayor o menor probabilidad que otros.

Consideremos como ejemplo una densidad de probabilidad en el intervalo $[0, 2]$ proporcional a

$$p(x) \propto x^2. \quad (4.14)$$

Calculemos la constante de proporcionalidad C para que la función esté normalizada,

$$p(x) = Cx^2 \Rightarrow \int_0^2 Cx^2 dx = 1 \Rightarrow C \frac{x^3}{3} \Big|_0^2 = 1 \Rightarrow C = \frac{3}{8} \quad (4.15)$$

y por tanto

$$p(x) = \frac{3}{8}x^2. \quad (4.16)$$

En este caso, si generamos números de acuerdo a esta distribución, nos aparecerá cualquier número (de forma impredecible) en el intervalo $[0, 2]$, pero valores en el entorno de $x = 3/4$ aparecerán con un probabilidad 9 veces mayor que los valores en el entorno de $x = 1/4$. Es importante fijarse en que a pesar de que la distribución no es plana, los resultados de cada suceso son independientes de los anteriores, e impredecibles. No confundir el concepto de *aleatoriedad* con *distribución plana*.

4.1.3. Secuencias de números

Dada una serie de números

$$C = \{x_0, x_1, x_2, x_3, \dots, x_{N-1}\} \quad (4.17)$$

decimos con frecuencia que

Los números del conjunto están distribuidos de acuerdo a la densidad de probabilidad $p(x)$.

¿Qué significa esta afirmación?

Consideremos números enteros en el intervalo $[0,9]$ por ejemplo. Tomemos

$$p(x) \propto x^2, x \in [0, 9], x \in \mathbb{Z} \quad (4.18)$$

Entonces la pregunta anterior, se responde del siguiente modo: La cantidad de veces que aparece un número en la secuencia será proporcional a su valor al cuadrado. Por ejemplo, el número 8 aparecerá 4 veces más que el 4; el número 6 aparecerá cuatro veces más que el 3, etc. Es decir, en una secuencia dada, si el número 4 aparece 1000 veces, el 16 aparecerá unas 16000, y el 3 aparecerá aproximadamente $1000 \frac{9}{16}$.

Nota: Usamos índices de $\{0, 1, \dots, N - 1\}$ para seguir la notación de C. Es por supuesto equivalente a usar $\{1, 2, \dots, N\}$ con la sucesión $\{x_1, x_2, \dots, x_N\}$

4.2. Generación de distribuciones uniformes en C

En el lenguaje C existe una función de librería que genera una distribución plana. La función tiene el prototipo `int rand(void)`, y devuelve un número entero aleatorio con *distribución plana* en el intervalo $[0, \text{RAND_MAX}]$,

Tanto la función `rand()` como el macro `RAND_MAX` están definidos en la librería `stdlib.h`, que hay que incluirla por tanto en el encabezamiento del programa. El valor de `RAND_MAX` no es necesario conocerlo a priori, basta con saber que la función `rand()` devuelve un número en el rango fijado anteriormente; de hecho este valor depende del tipo de procesador que se está utilizando, del tipo de compilador u otros factores. Si queremos averiguar su valor en un caso concreto, basta con imprimir su valor en pantalla,

```
printf("Valor de RAND_MAX = %d, en Hexadecimal = %X \n", RAND_MAX, RAND_MAX);
```

Cada vez que invocamos a la función `rand()`, ésta nos devuelve un número al azar diferente. Un segmento de código típico para generar números aleatorios sería

```
for(i=0;i<100;i++)
{
    alea=rand();
    printf("Nuevo numero aleatorio=%d\n", alea);
}
```

Los números generados en el ordenador no son estrictamente aleatorios, sino *pseudoaleatorios*, en el sentido de que si sabemos el algoritmo y todos los datos iniciales, podemos predecir con exactitud toda la secuencia posterior. Sin embargo desde el punto de vista matemático son números aleatorios que cumplen todas las propiedades formales.

4.2.1. Un ejemplo sencillo de generador uniforme

El siguiente programa imprime en la pantalla 10 números aleatorios enteros (tipo `unsigned int`). La función `my_ran` usa un método de congruencias.

```
#include <stdio.h>
unsigned int my_ran(void);
int main()
{
    int i;
    for(i=0;i<10;i++)
        printf("%d\n",my_ran());
}
unsigned int my_ran(void)
{
    //Numeros "magicos".No cambiar
    static unsigned int FACTOR = 67397, SUM = 7364893;
    static unsigned intINI=123456789;

   INI = (INI*FACTOR + SUM);
    return INI;
}
```

Observese que la secuencia depende del valor del primer numero `INI`. Este programa, donde el valor inicial de `INI` es fijo, devuelve siempre la misma secuencia. Para cambiar la secuencia basta cambiar al inicio el valor de `INI`.

4.2.2. Generación de un número plano en un intervalo dado

La función `rand()` genera un número entero en el intervalo `[0,RAND_MAX]`. A partir de esto podemos generar números en otros intervalos, tanto `char`, `int`, `float` o `double`.

Si queremos generar un `float` en el intervalo `[0,1)`, el siguiente código lo permite

```
x=rand()/(double)RAND_MAX+1
```

Notas:

- Es obligatorio el uso del cast (`double`) para que sea correcto, como puede inferirse fácilmente del comportamiento de la división en C entre tipos diferentes. Sin usar el *cast*, *x* sería siempre 0, lo que es un error frecuente.
- Es simple comprobar que la variable *x* calculada en la expresión anterior, se sitúa en el intervalo `[0,1)`.
- Es forzoso usar `double` pues si usaramos `float` el redondeo haría que apareciera el 1 exactamente alguna vez, lo que no es el standard de los generadores en C.
- Si queremos generar un `float` en el intervalo `[a, b)`, podemos usar el código

```
x=a+(b-a)*(rand()/(double)RAND_MAX+1)
```

- Podemos generar un entero en el intervalo `[0, N)` del siguiente modo

```
x=(int)(N*(rand()/(double)RAND_MAX+1))
```

Es muy importante remarcar que con esta definición, el número entero generado está en el intervalo `[0, N - 1]`. Esto es especialmente relevante cuando se usa la función `rand()` para generar índices de vectores, pues en C dado un vector de *N* componentes, su índice va de 0 a *N* - 1. Una código típico donde se usan números aleatorios como índice para vectores es el siguiente,

```
float V[100],valor;
int k;

k=100*(rand()/(double)RAND_MAX+1));
valor=V[k];
```

Dados los rangos de las variables, con el código anterior, nunca nos saldremos de memoria, pues k estará efectivamente entre 0 y $N - 1$.

4.2.3. Secuencias de números aleatorios en C

La función `rand()` parte de un número inicial, y a continuación genera una secuencia que depende enteramente del primer número (semilla).

Si ejecutamos el programa varias veces, la secuencia será siempre idéntica.

Si queremos cambiar la secuencia, debemos cambiar la semilla inicial; para ello existe la función `void srand(int semilla)` que tras invocarla, establece como semilla el dato pasado como argumento, cambiando por tanto la secuencia. Dada una semilla, la secuencia es siempre la misma.

El mecanismo correcto para generar una secuencia de N números aleatorios sería

```
semilla=12345;
srand(semilla);
for(i=0;i<N;i++)
{
    x=(rand()/(double)RAND_MAX+1));
    printf("Nuevo numero aleatorio=%f\n", x);
}
```

Un error común es escribir lo siguiente

```
semilla=12345;
for(i=0;i<N;i++)
{
    srand(semilla);
    x=(rand()/(double)RAND_MAX+1));
    printf("Nuevo numero aleatorio=%f\n", x); //x seria siempre el mismo
}
```

En este caso el número aleatorio x sería siempre el mismo, lo que evidentemente es incorrecto. Volveremos sobre esto en la sección [4.5](#).

4.3. Construcción de Histogramas

Como hemos dicho, es importante comprobar de diferentes maneras que los programas proporcionan los resultados correctos. En el caso de la generación de densidades de probabilidad, debemos comprobar que efectivamente los números generados responden a la distribución pedida.

La herramienta imprescindible para ello es construir un histograma. Un histograma es una representación de la frecuencia de aparición de cada una de las variables aleatorias, agrupadas en intervalos finitos. Una vez normalizado, un histograma es la versión discreta de la densidad de probabilidad $p(x)$. En general si la variable aleatoria es un número real (un `float` en el ordenador) no tiene sentido hablar de la probabilidad de que salga *exactamente* un número. Recordad [4.6](#) y [4.7](#).

Si que tiene sentido calcular cuántas veces nos aparece un número en cierto intervalo. Si $x \in [a, b]$, y dividimos el intervalo en N subintervalos, podemos preguntarnos cuantas veces el número generado cae en cada subintervalo. Esta función, es decir el número de veces que el número generado cae en cada subintervalo, es lo que llamamos histograma, debidamente normalizado para que el área encerrada sea 1. Si la variable x es un entero, no es estrictamente necesario agrupar en intervalos, pero si el rango de la variable aleatoria es muy grande, conviene hacerlo para tener una estadística de sucesos suficiente en cada intervalo.

De este modo, una condición necesaria para que nuestra generación de números aleatorios reproduzca una cierta función densidad de probabilidad $p(x)$, es que el Histograma generado con varios millones de números tenga la forma funcional correcta, idéntica a $p(x)$.

Supongamos una cierta distribución, con x en un rango finito, por ejemplo

$$p(x), x \in [-1, 2] \quad (4.19)$$

Operativamente, el histograma se construye de la siguiente manera.

- En primer lugar debemos decidir de cuantos intervalos hacemos el histograma. Si usamos pocos intervalos tendrá poca precisión para comparar con $p(x)$. Si usamos demasiados, entonces debemos utilizar una gran cantidad de números aleatorios para que todos los intervalos reciban suficientes puntos. Hay que buscar un compromiso. Supongamos que elegimos 100 intervalos. El histograma será para nosotros un vector con 100 componentes, `float H[100]`, que inicializaremos a 0. El vector $H[k]$ contendrá al final del cálculo la probabilidad de que x caiga dentro del intervalo k .
- El tamaño δ de cada intervalo es el de la región donde $p(x)$ es no nula, dividido entre el número de intervalos, en este caso $\delta = 3/100$
- Generamos un número aleatorio $x \in [-1, 2]$ de acuerdo a la densidad $p(x)$.
- Ahora debemos calcular en qué intervalo está el número anterior. Si ocurre que $x \in [-1, -1 + \delta]$ el intervalo será el primero (índice cero). En este caso para contabilizar el suceso haremos `H[0]++`. Si ocurre que $x \in [-1 + \delta, -1 + 2\delta]$, el intervalo será el segundo (índice 1) y haremos `H[1]++`. Y así sucesivamente. Es muy simple (una función lineal que el alumno debería calcular ahora como ejercicio) obtener, dado x , el intervalo correspondiente k , y a continuación hacer `H[k]++`
- Finalmente debemos normalizar H para que el área encerrada valga 1.

Ahora debemos dibujar simultáneamente la función $p(x)$ de partida y el histograma construido numéricamente, que deberían coincidir. Esta coincidencia será sólo aproximada, debido a que el número de puntos lanzados es finito. Si vamos aumentando el número de lanzamientos, la distribución numérica se irá pareciendo cada vez más a la teórica.

En el ejemplo anterior hemos supuesto conocido el intervalo de variación de x y que los puntos tienen una probabilidad significativa en todo el intervalo. Sin embargo en general la situación es algo más complicada. Puede ser que el intervalo de variación de x sea $[-\infty, \infty]$, o que sea desconocido. Puede ocurrir tambien que aunque el intervalo sea conocido, la distribución este muy picada en torno a un valor determinado, y si realizamos los intervalos en todo el rango, el histograma obtenido será carente de precisión. En estos casos (la mayoría), es conveniente generar la secuencia de números y guardarlos en un vector, calculando el valor máximo y mínimo de x . Estos dos valores nos sirven como extremos de los intervalos para calcular el histograma, utilizando ahora los números guardados en el vector, para calcular un histograma significativo.

4.4. Aparición de correlaciones en los números aleatorios

El problema de muchos generadores aleatorios es que aunque la distribución que generan sea la correcta, las correlaciones entre números sucesivos no son cero.

¿Qué significa esto? En una secuencia de números aleatorios, cada uno de ellos debe ser absolutamente independiente de todos los anteriores. Pensemos en la lotería o en el lanzamiento de un dado. Si en un lanzamiento obtenemos el número 4, el dado a la vez siguiente no se acuerda de esto en absoluto, y todos los números tendrán la misma probabilidad. Sin embargo muchos generadores, en concreto `rand()`, no cumplen esto. Existen correlaciones entre los diferentes números, de modo que si una vez sale uno determinado, existe a continuación una probabilidad no homogénea de que salgan unos u otros.

4.5. Sobre la reproducibilidad de secuencias random

En los ordenadores las secuencias de números aleatorios se basan en algoritmos que son deterministas; para comprender esto, escribimos a continuación una función similar a `rand()`, que usa un algoritmo basado en el método de congruencias.

```
int cong_ran(int alea)
{
    int FACTOR=67397,SUM=7364893; //Numeros magicos: su valor es importante
    alea=(alea*FACTOR+SUM);
    return alea;
}
```

Esta función recibe una variable entera, y a partir de dicha variable construye otra que es la que devuelve al programa de llamada. La cantidad devuelta tiene una distribución uniforme en su rango.

La forma típica de usar esta función es la siguiente,

```
aleatorio=3821897; //semilla inicial
for(i=0;i<100;i++)
{
    aleatorio=cong_ran(aleatorio);
    printf("Nuevo numero aleatorio=%d\n",aleatorio);
}
```

La variable `aleatorio` irá tomando valores random; su secuencia depende del primer valor con que llamemos a la función `cong_ran()`. Estamos utilizando aritmética entera, que es igual en todos los ordenadores, dependiendo sólo en realidad del número de bits usados para representar los enteros, pues las cantidades enteras se truncan siempre. Por tanto en cualquier ordenador que ejecutemos este código, con un valor inicial fijo, tendremos siempre la misma secuencia,

Con la función standard `rand()` de C, esto no está garantizado, pues puede depender del procesador, sistema operativo o compilador.

Según el problema que estemos tratando se nos presentan dos opciones extremas,

1. Queremos que la secuencia sea siempre idéntica, independientemente del ordenador y del momento de la ejecución. De este modo los resultados obtenidos, serán siempre los mismos aunque usemos números random (recordamos que en realidad trabajamos con números *pseudorandom*).
2. Queremos que la secuencia sea siempre diferente

Si estamos en el caso 1, no debemos usar las funciones de librería de C; debemos usar funciones con un código conocido; una buena solución es usar el generador de Parisi-Rapuano, y para inicializar las variables, usar el método de congruencias anterior, donde la semilla inicial se la damos en el código. En el Problema 5.8.3 del *Sorteo Uniforme*, se utiliza este método.

En el caso 2, podemos usar cualquier tipo de generador, pero para inicializarlo debemos usar números distintos cada vez para garantizar que si lo utilizamos en un mismo entorno nos dé resultados diferentes. Una forma común de hacer esto, es utilizar una función que nos devuelva el tiempo del sistema, y usar ese valor como semilla.

4.6. Ejercicios

El alumno debe prestar especial atención a los dos primeros ejercicios, pues ambos son de gran importancia para el aprendizaje y para los capítulos posteriores, usando las funciones construidas aquí de forma frecuente. El tercer ejercicio es para comprobar si se han asimilado correctamente los nuevos conceptos de este capítulo.

4.6.1. Cálculo de Histogramas

Generar un número entero aleatorio plano en el intervalo $[a, b]$. Construir el histograma de frecuencias usando K intervalos iguales. Lanzar N números y calcular el histograma debidamente normalizado. Dibujarlo usando `gnuplot`. Ver como evoluciona el histograma al aumentar N . Elegir valores concretos para a, b, K, N . Probar diferentes conjuntos de ellos.

El cálculo del histograma debe ser realizado de forma autónoma por una función `Histogram` a la que se deben pasar todos los argumentos necesarios, y debe devolver el histograma debidamente normalizado. Esta función será utilizada en otras partes del curso. Mostramos un código completo para esto (la función de llamada y la función que calcula el histograma, para que el alumno lo estudie antes de escribir código, y luego lo escriba autónomamente de forma modular y estructurada)

4.6.2. Distribución uniforme en un intervalo a base de pequeños cambios

Supongamos que necesitamos una función densidad uniforme en el intervalo $[a, b]$. Hemos visto como generarla a partir de un número uniforme como `rand()`. Esto nos da un número que cae en un punto absolutamente al azar en dicho intervalo. Existen problemas donde necesitamos generar una distribución uniforme, pero partiendo de un punto dado, y realizando movimientos pequeños en el tiempo. Para lograr esto basta con calcular una cierta cantidad δ en el intervalo $[-\epsilon, \epsilon]$, con $\epsilon \ll [b - a]$, e ir sumando δ a x para cada paso. El problema está cuando atravesamos la frontera, es decir cuando $x + \delta \notin [a, b]$ ¿qué hacemos entonces? Hay varias opciones, pero no todas son correctas. El alumno debe proponer y programar una solución al problema. Para verificar si hemos elegido la correcta, bastará generar varios millones de números y construir un histograma para ver si efectivamente la distribución obtenida finalmente es uniforme.

Al final del capítulo damos el código completo que realiza esto, con tres soluciones al problema de la frontera. Se debe averiguar cuáles son correctas.

4.6.3. Distribución Uniforme sobre un Círculo

Llueve sobre un disco de radio R . Calcular analíticamente la probabilidad de que caiga una gota de agua sobre un punto de coordenadas (x, y) .

Nota: Se pide la solución analítica (teórica). Para la solución numérica se requieren los conocimientos del Capítulo 5.

Dividimos el problema en dos partes: primero generaremos un valor para r , la distancia al origen, y finalmente generaremos los dos puntos (x, y) .

Si llueve uniformemente sobre el círculo, la probabilidad de que una gota caiga a distancia r del centro, será proporcional a la longitud de la circunferencia de radio r , como debería ser evidente. Es decir

$$p(r) \propto r, r \in [0, R]$$

Ahora debemos normalizar la probabilidad, para lo cual debemos calcular A de modo que

$$\int_0^R Ardr = 1$$

Integrando

$$\int_0^R Ardr = AR^2/2 \Rightarrow A = 2/R^2; p(r) = \frac{2}{R^2}r$$

Como nos piden un punto (x, y) , tras generar r de acuerdo a la distribución anterior, debemos generar un angulo random plano en el intervalo $[0, 2\pi]$.

4.7. Problemas

4.7.1. Problema 1

Escribir el código para generar un entero en el intervalo $[M, N]$ con probabilidad uniforme.

4.7.2. Problema 2

Considerar las distribuciones indicadas en las ecuaciones 4.11, 4.12 y 4.13. Comprobar si están normalizadas, y en caso negativo normalizarlas apropiadamente.

4.7.3. Problema 3 (Ampliación)

Un generador muy usado en los primeros ordenadores IBM, en los programas escritos en FORTRAN, era el generador RANDU, que usaba un método de congruencias, en concreto

```
x= (65539*x) & 0x7fffffff
```

Con RANDU, generar miles de tríos de números, usarlos como coordenadas (X, Y, Z) y dibujarlos con `gnuplot`. Observar el dibujo tridimensional desde varios puntos de vista a la búsqueda de correlaciones.

4.7.4. Problema 4

El código de la función que calcula el Histograma en la sección 4.8.2 no soporta datos donde todos los números sean idénticos. Ciertamente este caso tiene ciertas peculiaridades teóricas y prácticas. Dar una prescripción para definir el Histograma en este caso, e implementarlo después modificando el código de 4.8.2 apropiadamente.

4.8. Código en C

4.8.1. Conceptos de Programación: Archivos, Comentarios, Input, Macros, Preprocesado

Usaremos ahora la creación de archivos para guardar datos de salida. También recordaremos la importancia de los comentarios en el código. Para hacer llegar información al programa, utilizaremos el teclado para leer valores de variables. Utilizaremos una directiva `define` para construir una función *online*, que luego es llamada desde el código, y sustituida en el preprocesado. En concreto

```
#define Random_C (rand()/(double)RAND_MAX+1)) // genera un numero
//random en el intervalo [0,1)
```

De este modo escribiendo en el código `x=Random_C;`, x toma un valor random en el intervalo $[0, 1]$.

4.8.2. Cálculo de Histogramas

```
#include <stdio.h>
#include <stdlib.h>
#ifndef DEBUG

void Lee_Iteraciones(int *NDat);
void Histogram(double *data,double *Hist, int N_data,int N_Intervalos,
              double *d,double *m, double *M);
double frandom(double min,double max);
FILE *fout;
main()
{
#define N_Inter 100
#define N_Datos_Max 1000000
    double H[N_Inter],v[N_Datos_Max];

    int Niter,i;
    double a,b,minimo,maximo,delta;

    a=-3;b=4; //Fijamos el rango de variacion del generador plano
    Lee_Iteraciones(&Niter); //Leemos el numero de pasos

    if(Niter>N_Datos_Max) //Control
    {
        printf("La cantidad de numeros generados no puede ser superior a %d\n",N_Datos_Max);
        exit(1);
    }

    for(i=0;i<Niter;i++) //Generamos los datos
    v[i]=frandom(a,b);

    Histogram(v,H,Niter,N_Inter,&delta,&minimo,&maximo); //Calculamos el histograma

#endif DEBUG
    for(i=0;i<N_Inter;i++) //Escribimos en pantalla los datos
    printf("%d %f %f\n",i,i*delta+minimo,H[i]);
#endif

    fout=fopen("hist.dat","wt");
    for(i=0;i<N_Inter;i++) //Escribimos en un archivo los datos
    fprintf(fout,"%d %f %f\n",i,i*delta+minimo,H[i]);
    fclose(fout);
}

double frandom(double min,double max)
{
/* Genera un numero random plano double en el intervalo [min,max] que
   son los dos inputs de la funcion. */
    return min+(max-min)*(rand()/(double)RAND_MAX+1));
}
```

```

}

void Histogram(double *data,double *Hist, int N_data,int N_Intervalos,
              double *d,double *m, double *M)
{
/* Genera un histograma. Calcula, en funcion de los datos el minimo y maximo
   de los mismos para ajustar mejor los intervalos.

   *data-> input Datos sobre los que se genera el histograma
   *Hist-> output Histograma calculado
   N_data-> input Numero de datos
   N_Intervalos-> input Numero de intervalos del histograma
   *d -> output Medida de cada intervalo del histograma
   *m -> output Valor minimo de los datos
   *M -> output Valor maximo de los datos
*/
int i,Indice;
double del,min,max,Norm;

for(i=0;i<N_Intervalos;i++)//Inicializo
Hist[i]=0;

min=10000000;
max=-10000000;

for(i=0;i<N_data;i++) //Calculo el minimo y el maximo
{
if(data[i]<min)min=data[i];
if(data[i]>max)max=data[i];
}

del=(max-min)/N_Intervalos;
if(del==0)
{
printf("Error: No se pueden calcular los intervalos; Max=%lf,Min=%lf\n",max,min);
exit(1);
}

for(i=0;i<N_data;i++) //***** Calculo el histograma: Nucleo del Programa*****
{
Indice=(data[i]-min)/del;
if(Indice==N_Intervalos)
    Indice=N_Intervalos-1;
Hist[Indice]++;
#endif
}
// ***** Fin del nucleo del programa *****

*d=del;
*m=min;
*M=max;

/* Ahora normalizo */
// Recordar:  $1=A*\sum(h_i * \delta_i)=A*\delta*\sum(h_i)=A*\delta*N \Rightarrow A=1/(\delta*N)$ 

Norm=1.0/(N_data*del);
for(i=0;i<N_Intervalos;i++)
Hist[i]*=Norm;
}

void Lee_Iteraciones(int *NDat)
{

/* Lee de la consola el numero de datos a generar */

printf("Introducir la cantidad de numeros a generar:\n");
scanf("%d",NDat);
}

```

```
if(*NDat<=0)
{
printf("El numero de datos a generar debe ser mayor que cero; leido=%d\n",*NDat);
exit(1);
}
```

4.8.3. Distribución uniforme en un intervalo a base de pequeños cambios

```

// Generacion de un numero uniformemente distribuido
// en el intervalo [a,b];

#include <stdio.h>
#include <stdlib.h> // aqui esta definido RAND_MAX
#define N 200 // Rango entre a y b, y ademas numero de
           // subintervalos para calcular las frecuencias
#define M 100000000 // Numero total de sucesos generados
#define Random_C (rand()/(double)RAND_MAX+1)) // genera un numero
//random en el intervalo [0,1)

//#define RECHAZO // Si el numero se sale del intervalo, no lo aceptamos
//#define REFLEJO // Si el numero se sale del intervalo, lo reflejamos
#define TORO //Condiciones de contorno periodicas

double a,b;
double n;
int i,frec[N];
float delta,step;
FILE *fout; //Puntero a un archivo
main()
{
    a=0;
    b=N;
    step=N/10; // relacionado con la variacion maxima por intento.
    if(step<1)step=1;
    n=N-1; // Empezamos aqui.

    for(i=0;i<N;i++)
        frec[i]=0; //Inicializamos las frecuencias.

    for(i=0;i<M;i++) // bucle en sucesos
    {
        // el cambio en cada intento es en el intervalo [-step/2,step/2)
        delta=step*(0.5-Random_C);
        //OJO delta debe ser float, si no el redondeo crea sesgo.
#ifdef RECHAZO
        // Si el numero generado se sale del intervalo, no
        // cambiamos n

        if(((n+delta)<b)&&(n+delta)>a)
{
            n+=delta;
            frec[(int)n]++;
//Calculamos el histograma. Intervalos a parte entera
}
#endif

#ifdef REFLEJO
        // Reflejamos especularmente el numero obtenido respecto
        // del extremo del intervalo sobrepasado.
        n+=delta;
        if(n>b)n=2*b-n;
        if(n<a)n=2*a-n;
        frec[(int)n]++;
//Calculamos el histograma. Intervalos a parte entera
#endif

#ifdef TORO
        // Si se sale por la derecha, vuelvo por la izquierda
//la misma distancia. Y viceversa.

        n+=delta;
        if(n>b)n=a+n-b;
        if(n<a)n=b-a+n;
        frec[(int)n]++;
//Calculamos el histograma. Intervalos a parte entera
#endif
    }
}

```

```
    }
    fout=fopen("frec.dat","wt"); //nombre del archivo
    for(i=0;i<N;i++)
        if(frec[i]!=0)
            fprintf(fout,"%d %d\n",i,frec[i]);
    fclose(fout);
}
```

4.9. Apéndice: Representación de datos

Los ordenadores trabajan en binario. Por tanto toda la información que procesan debe tener este formato. Sin embargo un mismo número en binario es interpretado de forma diferente por el ordenador, según sea asignado a un tipo determinado de dato.

4.9.1. Código ASCII

Es un código universal que hace corresponder los caracteres alfanumericos a un código binario.

- Las letras a-z se corresponden con los números 97 al 122 (alfabeto Inglés)
- Las mayúsculas A-Z del 65 al 90.
- Los número 0-9 con el 48 al 57.
- La "ñ"se corresponde con el 164.

Una forma directa de introducir caracteres ASCII por el teclado es presionar la tecla ALT y con el pad numérico escribir el código correspondiente. Así, en un teclado inglés sin la letra "ñ" podemos escribir este carácter, manteniendo presionada la tecla ALT a la vez que escribimos 164 en el pad numérico (El carácter aparece al soltar la tecla ALT). Una lista con los caracteres ASCII puede encontrarse en Internet.

4.9.2. Números Enteros

Los números enteros son procesados en base binaria. Los positivos exactamente así. Los negativos sin embargo se almacenan de forma ligeramente diferente para facilitar su tratamiento interno en la CPU, usando “el complemento a 2”.

Supongamos que trabajamos con palabras de 8 bits, y numeramos los bits de 0 (el menos significativo, más a la derecha) a 7 (el más significativo, más a la izquierda).

Los números con el bit 7 igual a 0 son positivos y representan el número en binario directamente. Así el 00101011 representa el 43 en decimal. De este modo podemos representar desde el 0 al $01111111 = 127 = 7FH = 2^7 - 1$.

Para representar números negativos, se usa el complemento a 2. Si el bit más significativo está a 1, el número asociado se obtiene sumando a -128 el número binario positivo codificado por los 7 bits menos significativos. El número más negativo representable es por tanto $10000000 = -128 + 0 = -128$. Por ejemplo el número $10101011 = -128 + 43 = -85$. De este modo podemos representar desde el $10000000 = -128 = -2^7$ al $11111111 = -1$.

Así pues en general con n bits podemos representar desde el -2^{n-1} al $2^{n-1} - 1$ (en total 2^n números).

Con esta representación la suma y resta de enteros se simplifica muchísimo. En primer lugar hay que notar que cuando sumamos palabras de n bits, si el resultado tiene más de n bits significativos, el ordenador se quedara sólo con los n primeros tirando los más significativos (truncamiento), pues no dispone de más bits, dando en todo caso un aviso de desbordamiento (overflow de enteros).

Veamos esto con un ejemplo : Supongamos que trabajamos con 4 bits, con lo cual podemos codificar del -8 al 7. Queremos realizar con esta aritmética la operación 5 - 2. Primero recordemos como se representan estos números en binario; para el 5

$$5 = 0101$$

y para el -2, dado que estamos usando la representacion complemento a 2 tendremos

$$-2 = -8 + 6 = 1110.$$

Sumamos ahora bit a bit:

$$0101 + 1110 = 10011$$

pero esto nos da un overflow, es decir nos aparecen 5 bits significativos; el sistema sólo admite 4 bits, de modo que eliminamos el bit más significativo y nos queda

$$0011 = 3$$

que es el resultado justo. Puede comprobarse con más ejemplos que el procedimiento funciona correctamente. Eliminando el bit de overflow obtenemos resultados correctos, recordando siempre que los valores mayores que 7 o menores que -8 no pueden codificarse.

Para aclarar esta representación, es conveniente resaltar que cuando al máximo número representable le sumamos 1, el resultado es el número más pequeño; es decir si *nos pasamos* por encima, volvemos a la parte más baja. Del mismo modo si restamos 1 al más bajo, el resultado es el mayor. Podemos visualizar esta representación pensando que los números están sobre un círculo, de modo correlativo, con el más bajo y el mas alto uno al lado del otro.

Esta representación en C se dice que es *con signo* y se declaran las variables como

- char Utiliza 8 bits
- int Utiliza 32 bits
- long long int Utiliza 64 bits

No obstante esto depende del tipo de procesador, compilador y sistema operativo, y conviene verificarlo con las especificaciones del entorno de trabajo. Puede usarse representaciones donde sólo se usan variables positivas, pudiéndose representar entonces con n bits el intervalo $[0, 2^n - 1]$. En este caso, cuando se produce un overflow el resultado es equivalente al resto de la división por 2^n .

Incluimos un código simple que puede ayudar a comprender este tipo de comportamientos; sólo se incluyen algunos casos posibles; el alumno puede modificarlo para ver toda la casuística.

```
#include <stdio.h>
main()
{
    char a;
    int k,m;

    a=34+118-64+101+23;

    printf("suma= %X,%d,a=%d\n",34+118-64+101+23,34+118-64+101+23,a);

    a=((a&0xFF)*3)&0xFF;

    printf("pro= %X,%d,a=%d,%X\n", (34+118-64+101+23)*3,(34+118-64+101+23)*3,a,a);
    printf("a=%d %c\n",a,a);

    k=0xF;
    m=0x11;
    printf("%d %d %d\n",k & m,k | m,k+m);

}
```

En el primer cálculo de a , al sumar a la derecha, lo hace como enteros (int32, el default) y tendremos $a = 212$. En binario $a = 11010100$, que en la notación de complemento a 2 representa el número $a = -128 + 84 = -44$.

Veamos ahora el producto de a por 3. Podemos hacerlo como $a + a + a$. En la primera suma tendremos $a + a = 11010100 + 11010100 = 110101000$. El primer bit (el más significativo, a la izquierda) se pierde, pues sólo trabajamos con 8 bits, por tanto tendremos $a + a = 10101000$ que representa el número $a + a = -128 + 40 = -88$ (Notar que efectivamente $-44 - 44 = -88$ al no

haber desbordamientos). Ahora sumamos de nuevo a , y obtenemos $10101000 + a = 10101000 + 11010100 = 111011100$. Eliminado el bit de desbordamiento, tenemos que $3a = 11011100 = -128 + 92 = -36$

Recomendamos al alumno que repase y rehaga completamente los cálculos anteriores para familiarizarse con las operaciones en binario y las representaciones en el ordenador.

4.9.3. Números en Coma Flotante

La representación de números enteros tiene enormes ventajas para su uso en el ordenador: no presenta problemas de redondeo y se maneja muy rápidamente. Sin embargo presenta dos inconvenientes. Por un lado hay operaciones que dan resultados que no son números enteros (una división en general, una función trigonométrica, una exponencial), y por otro el rango de variación de los enteros a veces es pequeño para lo que necesitamos en el problema.

Para resolver estos problemas se usa la representación en coma flotante, donde el número se guarda en formato (*mantisa, exponente*), usando un número determinado de bits para cada campo. El número de bits usual es 32 (`float`, `int`) y 64 `double` si bien ya es posible usar 128.

El primer problema de los números en coma flotante es la precisión. Hay que controlar que las operaciones que se hacen tienen sentido, y no estamos obteniendo solamente resultados de redondeo. Por ejemplo, restar dos números muy parecidos, que se diferencian en algo del orden de la precisión de un número en coma flotante, puede dar un error relativo gigantesco.

Otra dificultad es que las funciones con números en coma flotante necesitan en general un número de operaciones elementales elevadas, por ejemplo para calcular el seno, la exponencial, etc. lo que hace más lentos los programas.

Capítulo 5

Distribuciones arbitrarias partiendo de un generador plano

Con gran frecuencia se nos plantea el problema de generar una distribución de sucesos de acuerdo a una densidad de probabilidad dada; este problema aparece en ramas de la Física, las Matemáticas, la Química o la Ingeniería entre otras. Hemos visto métodos para generar densidades planas. Veremos en este Capítulo cómo a partir de ellas podemos generar densidades de probabilidad arbitrarias.

5.1. Método de la altura

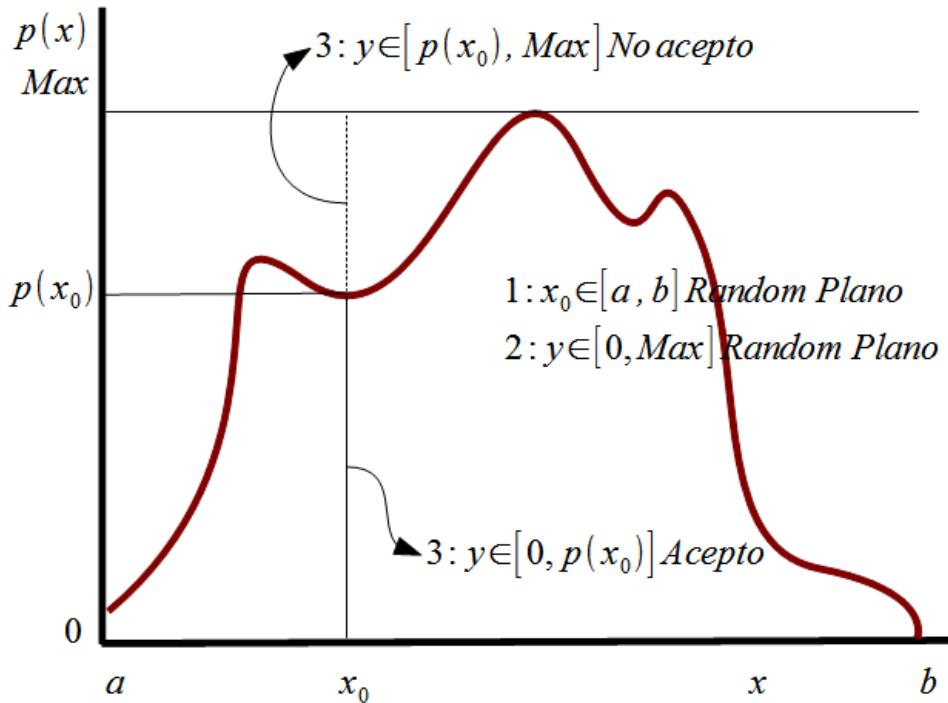


Figura 5.1: Elección de un nuevo punto de la distribución por el método de la altura

Sea una función densidad de probabilidad conocida y acotada. Supongamos que los valores

posibles de x están en el intervalo $[a, b]$. Llamemos M al máximo de $p(x)$

Para generar números de acuerdo a la función $p(x)$ procedemos del siguiente modo

1. Generamos x_0 , un valor tentativo de x en el intervalo $[a, b]$ de forma plana
2. Generamos y en el intervalo $[0, M]$ uniformemente
3. Si $y \leq p(x_0)$, aceptamos x_0
4. Si $y > p(x_0)$, rechazamos x_0

La figura 5.1 puede ayudar a comprender el algoritmo: dado un punto x al azar y con función densidad uniforme, este punto será elegido con una probabilidad proporcional a la altura de la función en ese punto, es decir, proporcional a $p(x)$.

Para que este método sea aplicable, es necesario conocer exactamente $p(x)$ y que sea acotada.

Nota: Veamos en detalle qué entendemos cuando decimos *aceptamos* o *rechazamos* un número en el proceso. Supongamos que los números de acuerdo con la distribución $p(x)$ los vamos escribiendo en una lista, que luego alguien puede utilizar. Pues bien, cuando aceptamos el número, dicho número es escrito en la lista; cuando lo rechazamos, no lo escribimos. En el caso de que estemos trabajando con una función que devuelve números al programa de llamada, si encontramos un número que aceptamos, lo devolvemos directamente; si nos aparece un número que no aceptamos, repetimos el proceso hasta que nos aparece un número que si aceptamos, y entonces lo devolvemos.

Pregunta:

¿Existen densidades de probabilidad que sean no acotadas?

5.2. Método de la función de distribución

Dada $p(x)$, definimos la *función de distribución* asociada a la función densidad de probabilidad $p(x)$, como

$$\tau(x) = \int_{-\infty}^x p(y)dy \quad (5.1)$$

La función de distribución es positiva y creciente. Si conocemos la función de distribución

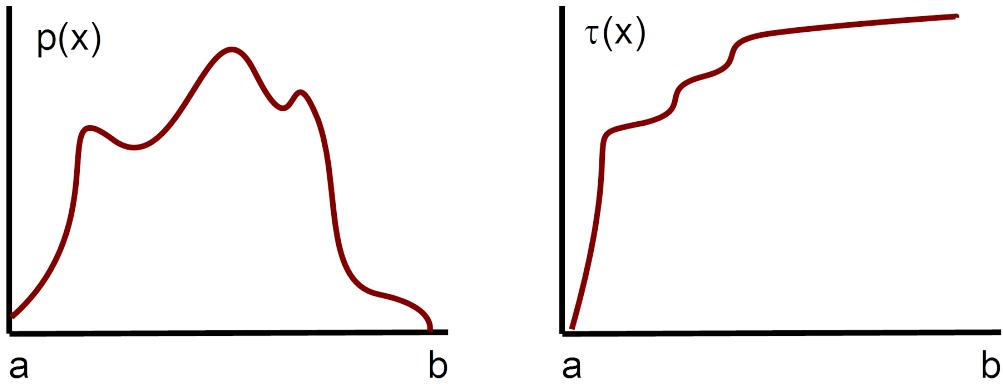


Figura 5.2: Densidad de probabilidad y función de distribución asociada

$\tau(x)$, podemos generar una función densidad de probabilidad $p(x)$, del siguiente modo

1. Generamos ω plano en el intervalo $[0, 1]$

2. Invertimos la función de distribución y buscamos el valor de x_0 tal que $\tau(x_0) = \omega$, como está indicado en la figura 5.3; este x_0 es el valor buscado

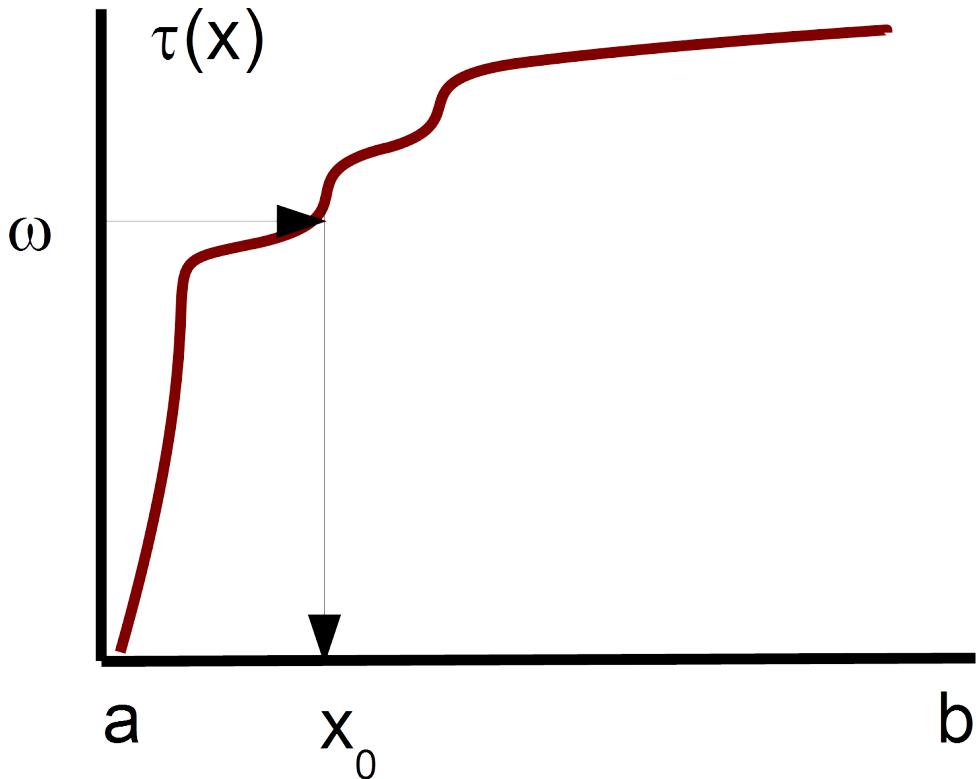


Figura 5.3: Cálculo de un punto de la densidad de probabilidad invirtiendo la función de distribución

Para que este método funcione debemos conocer $p(x)$, saber calcular su integral indefinida $\tau(x)$ y saber invertir dicha función de distribución. La demostración de que con este método obtenemos la densidad correcta es sencillo; para ello basta fijarse en que la probabilidad de obtener un valor x es proporcional a la derivada de la función $\tau(x)$ en el punto x . Pero la derivada de $\tau(x)$ es precisamente $p(x)$ como se desprende trivialmente de 5.1.

Remarcar que aquí siempre se acepta el número obtenido.

5.3. Método de Metropolis

Es el método menos intuitivo, pero el más potente, pues no es necesario el conocimiento completo de $p(x)$, sino solamente el cociente de probabilidad entre dos puntos; es decir basta con conocer la función densidad de probabilidad salvo la constante de normalización. Esta situación es frecuente.

Consideremos $p(x)$ conocida salvo la constante de normalización, es decir conocemos con exactitud cantidades como $p(x)/p(y)$. Supongamos que x varía en el intervalo $[a, b]$.

El algoritmo es el siguiente

1. Generamos $x \in [a, b]$ de forma uniforme.
2. Se calcula $p(x)$
3. Se genera otro valor de x, x_n , en el intervalo $[a, b]$ de forma plana

4. Calculamos $p(x_n)$
5. Calculamos $C = p(x_n)/p(x)$
6. Generamos ω en el intervalo $[0, 1]$ uniforme.
7. Si $C > \omega$, elegimos x_n , en caso contrario elegimos (seguimos) con x
8. Volvemos al punto 3 si queremos generar otro número, donde tomaremos como valor de x el elegido en el punto anterior.

Notar que si $p(x_n)$ es mayor que $p(x)$, aceptaremos siempre el cambio, pues el cociente será mayor que 1 y ω está en el intervalo $[0, 1]$. Es decir si la probabilidad del punto nuevo es mayor, vamos al punto nuevo. Si la probabilidad es menor, podemos aceptar el cambio también, dependiendo del cociente de probabilidades y del número ω generado.

En este algoritmo siempre necesitamos un dato *viejo* y uno *nuevo* para poder comparar entre ambos. Siempre por tanto debemos tener un dato para comenzar.

Nota Importante En el algoritmo de Metropolis, vemos que al final el dato puede ser diferente del inicial o incluso el mismo (si el cambio no es aceptado). En el caso de no aceptar el cambio, el número SI que es aceptado en cualquier caso; es decir si vamos poniendo los números en un archivo, el número obtenido lo escribimos siempre, y por tanto eventualmente algunos datos aparecerán repetidos, cuando el cambio tentativo no es aceptado; insistimos, aunque no se acepte el cambio, el dato es válido y es obligatorio aceptarlo para que la distribución sea la correcta.

Ejercicio de comprensión: Para entender este importante concepto, el alumno puede hacer el siguiente ejercicio: Consideremos como espacio muestral el conjunto discreto $x = \{0, 1\}$. Definimos $p(0) = 0.9, p(1) = 0.1$. Escribir un programa que genere una secuencia de números con esta distribución usando el algoritmo de Metropolis. Comprobar que se reproduce la probabilidad de partida, dibujando el histograma. A continuación, modificar el programa para que haga las cosas de forma *incorrecta*: es decir, si el valor propuesto no es aceptado, no repetimos el anterior en la lista aceptada. Dibujar el nuevo histograma y comprobar que no es correcto.

5.4. Generadores uniformes de alta calidad

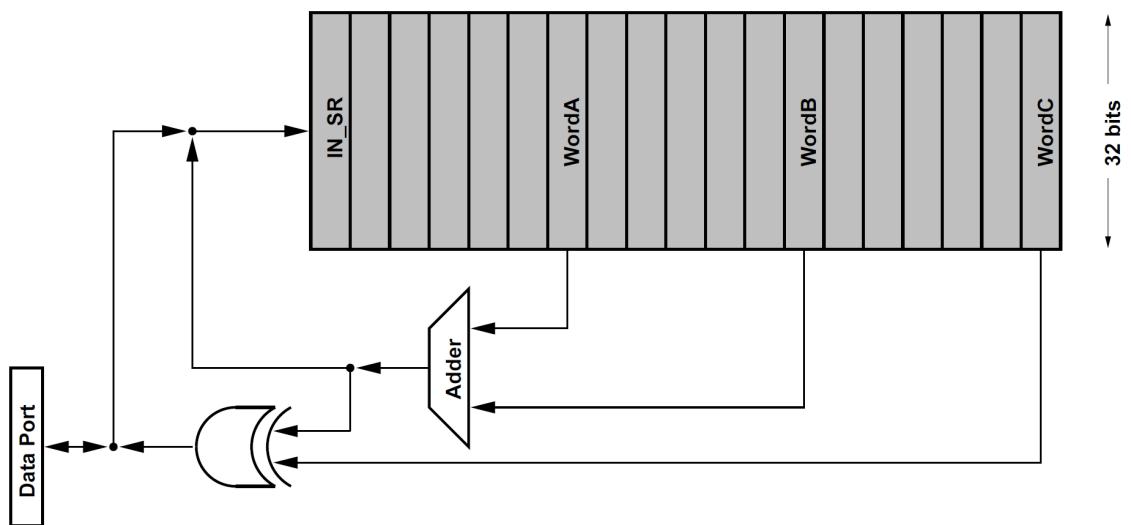


Figura 5.4: Esquema lógico del generador de Parisi-Rapuano, optimizado para programar en silicio.

El problema de las correlaciones puede mejorarse usando generadores más sofisticados. Existen en la literatura una ingente cantidad de generadores, tests y documentación sobre el tema. Aquí propondremos un generador de gran calidad y que consume pocos recursos de computación. Es habitual su uso en Simulaciones de Monte Carlo. Fue propuesto por Giorgio Parisi y Federico Rapuano y se encuadra en los generadores tipo shift-register. Para comprender esta sección es necesario conocer como se representan los datos en un ordenador internamente. En la sección 4.9 se da un breve resumen.

En primer lugar definimos un vector de 256 componentes de un entero sin signo, vector que llamaremos *la rueda*.

```
unsigned int irr[256];
```

A continuación lo inicializamos con un generador cualquiera, por ejemplo `rand()`, asegurándonos que rellenamos los 32 bits de cada componente del vector. La calidad de este generador inicial no es importante; una posible opción para inicializar la rueda es,

```
for(i=0;i<256;i++)
    irr[i]=(rand()<<16)+rand();
```

Definimos 4 indices, cada uno de los cuales sólo toma valores entre [0,255], es decir son de tipo `unsigned char`, en concreto

```
unsigned char ind_ran,ig1,ig2,ig3;
```

Inicializamos también estos índices

```
ind_ran=ig1=ig2=ig3=0;
```

Ahora, para generar un número aleatorio, a la vez que modificamos un número de la rueda, el código es:

```
ig1=ind_ran-24; //Modificamos los indices cada vez que generamos un numero
ig2=ind_ran-55;
ig3=ind_ran-61;
irr[ind_ran]=irr[ig1]+irr[ig2]; //Cambiamos la propia rueda
ir1=(irr[ind_ran]^irr[ig3]); //Numero random generado
ind_ran++; // Cambiamos la posicion base para el siguiente numero random
```

La variable `ir1` contiene el número aleatorio plano entre 0 y $2^{32} - 1$.

Si queremos generar un número entre [0,1), deberemos dividir por 2^{32} . Sin embargo este paso se delicado. Por problemas de precisión y redondeo, si dividimos directamente, dependiendo incluso del tipo de procesador, podríamos a veces encontrar un 1 exacto. Esto no es lo estándar en C. Para evitar este problema, es necesario ajustar el numero de modo que nunca pueda aparecer el 1.

Una elección posible es

```
#define NormRANu (2.3283063671E-10F)
```

y entonces

```
float r; // en el inicio del fichero
...
r=ir1*NormRANu;
```

la variable `r` contendrá un número correcto.

Este algoritmo es usado en muchos programas científicos, especialmente para sistemas con cálculo entero. Es también fácilmente programable directamente sobre silicio, es decir a base de puertas lógicas. Damos un esquema de una implementación optimizada, que puede ayudar a comprender el funcionamiento del algoritmo en la figura 5.4.

En el Ejercicio 5.7.2, se incluye un ejemplo con una implementación concreta de todo esto y la definición de una función que puede usarse en otros programas a lo largo del curso. El el problema 5.8.3 y en el código correspondiente 5.9.6 se usa este generador a través de su definición en un *macro*, muy compacto para incluir en otros códigos.

5.5. Generación de puntos uniformes sobre curvas

Suponemos conocido por el alumno los conceptos básicos de Teoría de Curvas y Superficies; si necesitan un repaso de los mismos, el libro *Geometría Diferencial* de Martin Lipschutz en la Colección Schaum contiene una excelente introducción.

Dada una curva $y = f(x)$ o en parámetricas $x = \alpha_1(t), y = \alpha_2(t)$, que supondremos suficientemente suave, queremos generar puntos uniformemente sobre la misma. Estudiaremos dos métodos diferentes.

5.5.1. Método de la longitud de arco

Supongamos la curva en parámetricas

$$\begin{aligned} x(t) &= \alpha_1(t) \\ y(t) &= \alpha_2(t) \end{aligned} \tag{5.2}$$

o en notación más compacta

$$\vec{r}(t) = \vec{\alpha}(t)$$

Introducimos la longitud de arco, s que es la longitud de la curva medida desde un punto arbitrario, por ejemplo t_0

$$s = \int_{t_0}^t |\vec{\alpha}'(t)| dt$$

Hecho esto, la respuesta a nuestra pregunta es trivial: para generar puntos uniformemente sobre una curva, basta generar puntos uniformes sobre la longitud de arco s . Tras generar puntos sobre s , ahora debemos invertir la ecuación parámetrica para obtener x, y . Ver el Problema 5.8.5.

5.5.2. Método de la pendiente

En el método anterior necesitamos realizar un integral, en general difícil, para obtener s , y luego invertir las ecuaciones parámetricas. Podemos usar otro método más sencillo para generar puntos uniformemente. Consideremos la curva $y = f(x), x \in (a, b)$. La idea es generar puntos sobre el intervalo (a, b) , pero no generarlos uniformemente sino con una distribución tal que al proyectarlos sobre la curva, su distribución sea uniforme. En la figura 5.5 mostramos la relación entre el ángulo α , y la proyección de b sobre el eje x , en este ejemplo δ_x . Recordando la sección 5.2, concluimos que los valores de x deben ser generados con una distribución de probabilidad

$$p(x) \propto \frac{1}{|\cos(\alpha)|}$$

donde α es el ángulo que forma la pendiente de la curva, es decir

$$\tan(\alpha) = |f'(x)|$$

con lo cual

$$p(x) \propto \sqrt{1 + f'(x)^2}$$

El cálculo de la derivada y su módulo es trivial. El problema aquí es la constante de proporcionalidad. Podriamos calcularla, y usar luego el método de la altura o de la función de distribución, pero llegariamos a algo similar al método de la longitud de arco. Pero si usamos el Algoritmo de Metropolis, no necesitamos cálculos adicionales y es sumamente simple. Si existen regiones con derivadas infinitas, el método puede dejar de funcionar. En esa misma línea, hay que remarcar que si la función $f(x)$ tiene zonas con derivadas muy diferentes, el algoritmo de Metropolis puede tener problemas de convergencia. Ver el Problema 5.8.6.

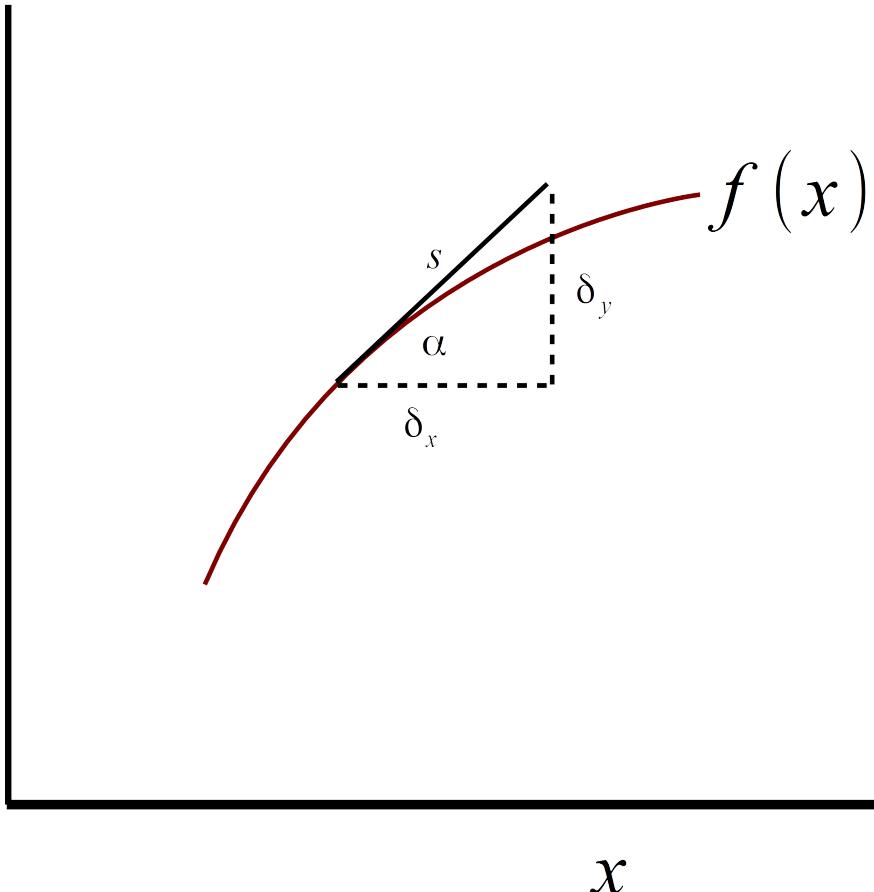


Figura 5.5: Gráfica para visualizar la proyección de una curva sobre el eje x en un punto. Dado un segmento δ_x queremos saber cuánto más grande es el *levantamiento* (s) sobre la curva de dicho segmento. Es decir, la relación entre δ_x y s . En este caso vemos que $s/\delta_x = 1/\cos(\alpha)$ con $\alpha = f'(x)$.

5.6. Generación de puntos uniformes sobre una superficie

Pensemos en una superficie dada en paramétricas,

$$\begin{aligned} x &= f_1(u, v) \\ y &= f_2(u, v) \\ z &= f_3(u, v) \end{aligned} \tag{5.3}$$

o en forma más compacta

$$\vec{r} = \vec{f}(u, v)$$

Geometricamente esto representa un *mapping* de una región del plano Euclídeo $(u, v) \subset \mathbb{R}^2$ en una superficie sobre el espacio \mathbb{R}^3 . La idea ahora es similar al método de la pendiente para las curvas. En ese caso, podemos pensar que la derivada de la curva nos dice **cuánto se estira** un segmento del eje x al llevarlo (o proyectarlo) sobre la curva $f(x)$. Así, cuando la derivada sea alta, el segmento del eje x (podemos pensar en el intervalo $[x, x + dx]$) se estira mucho y debemos generar muchos puntos, para que al proyectar se repartan con una densidad correcta. Para una superficie, la situación es similar: dada una región (pequeña, podemos pensar en el rectángulo de vértices $(u, v) \rightarrow (u + du, v + dv)$), esta región al llevarla sobre la superficie se deforma, ampliándose o encogiéndose. ¿Qué cantidad nos dice cuánto varía el área de una región en el plano (u, v) al llevarla sobre la superficie?

El estudio de las superficies nos dice que esta cantidad es precisamente el módulo del vector normal a la superficie con la parametrización considerada. Para construirlos, definimos en primer lugar

$$\vec{r}_u = \frac{\partial \vec{r}}{\partial u}, \vec{r}_v = \frac{\partial \vec{r}}{\partial v}$$

y entonces el vector normal viene dado por

$$\vec{N}(u, v) = \vec{r}_u \times \vec{r}_v$$

Pues bien la cantidad que buscamos y que nos da la relación entre el área en la región (u, v) y el área en la superficie $\vec{r}(u, v)$ viene dado precisamente por el módulo de dicho vector Normal. Escrito en forma diferencial

$$dS = |\vec{N}(u, v)| dudv$$

donde dS es un elemento de superficie.

El procedimiento pues para generar puntos uniformemente sobre la Superficie S es claro: generamos puntos sobre el plano (u, v) , pero no uniformemente sino proporcionalmente a $|\vec{N}(u, v)|$.

Debe realizarse ejercicio 5.7.3 para aclarar estos conceptos, dándose la solución y el código correspondiente como ayuda para el estudio.

5.7. Ejercicios

5.7.1. Evolución en el algoritmo de Metropolis

Este ejercicio tiene una gran importancia conceptual para comprender el algoritmo de Metrópolis y poder acometer más adelante aplicaciones más complejas. Debe ser pues estudiado con suma atención

Considerar la función

$$f(x) = Ae^{-20x^2}, \quad x \in [-10, 10] \quad (5.4)$$

Usando en Algoritmo de Metropolis generar una función densidad de probabilidad de acuerdo a esta función.

En este caso si calculamos directamente la exponencial para $f(x)$, en la mayoría de los casos el ordenador será incapaz de procesar números tan grandes; la forma de evitar esto es calcular sólo el cociente, o lo equivalente en términos de exponentiales (restar los exponentes), de modo que podamos calcular los cocientes y no encontremos *overflows*.

Dado pues lo explosivo del comportamiento de la función, además debemos movernos en x utilizando el método de los pequeños cambios, es decir

$$x \rightarrow x + \delta, \quad \delta \ll 1 \quad (5.5)$$

En caso contrario, incluso sólo la variación del exponente, haría explotar el cálculo de la exponencial.

Simular partiendo de un valor inicial de x igual a -9.8 . Resolver numéricamente, calcular el histograma y comprobar que coincide con lo exigido. Notar que cuando dibujemos el histograma, dado lo pequeño de la función para $|x| > 1$, si usamos escala lineal en el eje y todo parecerá correcto. Sin embargo para ver diferencias incluso cuando la función es muy pequeña debemos dibujar en escala logarítmica en dicho eje. Hacerlo y comprobar si efectivamente hay diferencias en la región no central del histograma. ¿En cuántos órdenes de magnitud se diferencia el histograma numérico de la función exacta?

Observar especialmente el comportamiento inicial del sistema, dibujando en el eje X el *tiempo* de Metropolis, es decir, el índice de la iteración en el ordenador, y en el eje Y el valor de x obtenido. Comentar lo que ocurre.

Se puede tomar como base el código de la sección 5.9.2.

5.7.2. Método de la Altura

Escribir el código para la generación de un número aleatorio con función densidad de probabilidad $f(x)$, donde $f(x)$ está definida en una función de C. El punto x se sitúa en el intervalo $[x_{\min}, x_{\max}]$. Usar el método de la altura.

El punto x tentativo debe ser generado por el método usado anteriormente de pequeños cambios.

Como generador plano debe usarse el generador de Parisi-Rapuano, implementado en funciones de modo que puedan ser usadas en otros programas.

El programa debe calcular el histograma de los números generados para comprobar si todo es correcto. Debemos utilizar la función `Histogram` construida en el Ejercicio 4.6.1. Probar con diferentes funciones $f(x)$.

5.7.3. Generando puntos uniformemente sobre una superficie

Definimos la superficie de ecuación

$$z = a(x^2 + y^2)$$

Considerar la región de la superficie con $0 < \sqrt{x^2 + y^2} < R$. Generar puntos uniformemente distribuidos sobre dicha región. Calcular sobre ellos $\langle r^n \rangle$ con n Natural. Hacer una estimación analítica aproximada de este valor en función de a, R, n y comprobar que coincide con el numérico para diferentes valores de estos parámetros.

Veáse el Apéndice 5.10 para los cálculos teóricos y la sección 5.9.3 para una implementación de lo anterior en C.

5.7.4. Puntos homogéneos sobre una superficie esférica

Dada la superficie de una esfera, queremos generar puntos al azar sobre su superficie de forma homogénea.

Existen varias formas de hacer esto. Veremos dos de ellas a continuación.

Método de la Función de Distribución

En primer lugar consideraremos la parametrización de la superficie y su medida de integración para extraer la función densidad de probabilidad sobre la superficie esférica, que en este caso dependerá de 2 variables. Elegimos las coordenadas polares de la figura 5.6.

Consideraremos el ángulo $\phi \in [0, 2\pi]$ y el ángulo $\theta \in [0, \pi]$,

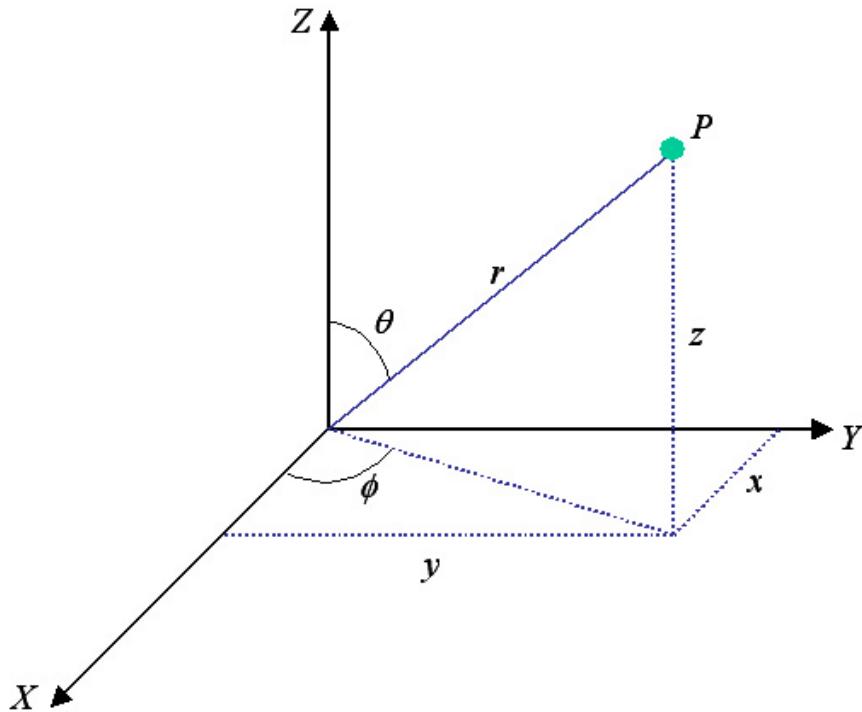


Figura 5.6: Coordenadas esféricas y cartesianas

El elemento de superficie viene dado por

$$R \frac{1}{2\pi} d\phi \frac{1}{2} \sin \theta d\theta \quad (5.6)$$

Consideramos R fijo igual a 1. La distribución de probabilidad ya normalizada es

$$p(\phi, \theta) = \frac{1}{2\pi} \frac{1}{2} \sin \theta, \phi \in [0, 2\pi], \theta \in [0, \pi] \quad (5.7)$$

Podemos usar el método de la Función de Distribución. Para ello debemos integrar $p(\varphi, \theta)$ y calcular la función de Distribución,

$$\tau(\hat{\phi}, \hat{\theta}) = \int_0^{\hat{\phi}} d\phi \int_0^{\hat{\theta}} d\theta p(\phi, \theta) = \int_0^{\hat{\phi}} d\phi \int_0^{\hat{\theta}} d\theta \frac{1}{2\pi} \frac{1}{2} \sin \theta \quad (5.8)$$

La integración es trivial,

$$\tau(\hat{\phi}, \hat{\theta}) = \frac{\hat{\phi}}{2\pi} \frac{1}{2} (1 - \cos \hat{\theta}) \quad (5.9)$$

Dado que factorizan, podemos generar de forma independiente cada uno de los ángulos.

$$\tau_0(\hat{\phi}) = \frac{\hat{\phi}}{2\pi}, \quad \tau_1(\hat{\theta}) = \frac{1}{2}(1 - \cos \hat{\theta}) \quad (5.10)$$

Además son invertibles trivialmente, por tanto,

- Generación de $\hat{\phi}$: Generamos un random plano $\omega \in [0, 1]$, entonces $\hat{\phi} = 2\pi\omega$
- Generación de $\hat{\theta}$: Generamos un random plano $z \in [0, 1]$, entonces $\hat{\theta} = \arccos(1 - 2z)$

Idea intuitiva: La medida de integración sirve para que el área de la superficie paramétrica (θ, ϕ) , que es un cuadrado, se convierta gracias a la medida, en el área correcta sobre la esfera. Así, cuando θ está cerca del ecuador ($\theta \approx \frac{\pi}{2}$) los círculos en ϕ tienen una longitud $2\pi R$. Sin embargo cuando θ se aleja del ecuador, los círculos con θ fijo, tienen un radio solamente de $2\pi R \sin \theta$. Este factor adicional es lo que nos da la diferencia de áreas entre el plano (θ, ϕ) y la superficie esférica. Si generáramos puntos homogéneos sobre el plano (θ, ϕ) , al llevarlos sobre la superficie daría una densidad mucho mayor en torno a los polos que al ecuador.

Método Geométrico

Un segundo método consiste en considerar la superficie esférica, por ejemplo de radio 1, inscrita en un cubo de lado 2, ambos centrados en el origen.

Lanzar puntos al azar en el interior del cubo es fácil; basta generar 3 números random independientes en el intervalo $[-1, 1]$, para las coordenadas (x, y, z) . Ingenuamente podríamos pensar que bastaría a continuación proyectar ese punto sobre la superficie esférica, normalizándolo simplemente, y ya tendríamos un punto válido.

Este punto efectivamente está sobre la superficie esférica, pero NO está distribuido uniformemente. Los puntos de la superficie esférica próximos a las esquinas del cubo recibirían más puntos (un factor $\sqrt{3}$ más), y por tanto la distribución sería más densa en la zona proyectada correspondiente.

Una forma de evitar este problema, es considerar sólo aquellos puntos que caen dentro de la esfera; si caen fuera, continuamos hasta que otro caiga dentro. Si ahora normalizamos estos puntos, sí que estarán distribuidos uniformemente sobre la superficie, cómo es fácil de ver fijándose en que ahora proyectamos sólo los interiores a la esfera, que tienen simetría esférica.

Comprobación

Hemos insistido bastante en la necesidad de comprobar que los resultados obtenidos con los programas son los correctos. Conviene hacer primero pruebas sencillas que detecten los problemas más graves, que serán los que se presenten en las primeras fases de desarrollo. Pero luego debemos comprobar exhaustivamente los resultados. Proponemos a continuación dos posibles formas de comprobación para este ejercicio.

La forma más rápida es dibujar con `gnuplot` en tres dimensiones los puntos generados, y girando el dibujo con el ratón se puede comprobar si aproximadamente la densidad de puntos es uniforme. Para ello es suficiente escribir un archivo con todos los puntos, donde cada fila del

mismo son las coordenadas x, y, z de cada punto. Por supuesto debemos controlar que todos los puntos están sobre una superficie esférica.

Una forma exhaustiva es la siguiente: consideremos la proyección sobre el plano $z = 0$ de los puntos (x, y, z) generados. A continuación calculamos el histograma en función de r , es decir $p(r)$, donde r es la distancia al origen de los puntos proyectados. Notar que $p(r)$ no es uniforme. Es posible (se deja como ejercicio) calcular exactamente esta función $p(r)$.

Lanzando suficientes puntos, el histograma numérico, debe superponerse de forma casi exacta con $p(r)$. Podemos repetir esto para otros planos. Veáse el problema 1. Una ampliación de estas ideas está propuesta en el problema 2.

5.7.5. Puntos Uniformes sobre un disco

Generar numéricamente gotas de lluvia sobre un disco de radio R de acuerdo con la solución analítica para $p(r, \theta)$ obtenida en 4.6.3. Dibujar los puntos con `gnuplot` para una verificación visual (parcial) de la solución obtenida.

5.8. Problemas

5.8.1. Problema 1

Tras resolver el ejercicio 2 de este Capítulo, dibujar con `gnuplot` los puntos generados sobre la superficie de la esfera.

Comprobar que una generación homogénea de los ángulo $[\theta, \varphi]$ no proporciona una distribución homogénea sobre la esfera.

Comprobar también que si cuando usamos el método de inscribir la esfera en un cubo, no descartáramos los que caen fuera de la esfera, el resultado sería incorrecto.

5.8.2. Problema 2

Calcular el volumen de una esfera de radio R en dimensión d . Para ello, inscribir la esfera en un cubo de lado $2R$, y lanzar puntos uniformemente distribuidos en su interior. El cociente de puntos que caen dentro de la esfera sobre los totales, dan la relación entre el volumen de la esfera y del cubo, este último ya conocido.

El programa debe leer de la linea de comandos el Radio de la esfera y la dimensión del espacio.

Ayuda:

El volumen exacto de una esfera de radio R en d dimensiones es

$$V_d(R) = \frac{\pi^{d/2} R^d}{\Gamma((d/2) + 1)} \quad (5.11)$$

Por ejemplo,

$$V_2(R) = \pi R^2 \quad (5.12)$$

$$V_3(R) = \frac{4}{3}\pi R^3 \quad (5.13)$$

$$V_4(R) = \frac{1}{2}\pi^2 R^4 \quad (5.14)$$

5.8.3. (Ampliación) Problema 3

En un sorteo se reparten números del 3000 al 4010. Disponemos de bolas numeradas del 0 al 9 para realizar el sorteo. Proponer un mecanismo de sorteo de modo que todos los números tengan la misma probabilidad de ser elegidos. Comprobar que la hipótesis es correcta, realizando una simulación en el ordenador de varios millones de sorteos, y viendo que el histograma es efectivamente plano.

5.8.4. (Ampliación) Problema 4

Generamos puntos uniformes en el interior de una esfera de radio 1. Notar que ahora hablamos de *esfera*, no de *superficie esférica*. El método más simple es generar un vector de 3 componentes cada una de ellas en el intervalo $[-1, 1]$ y tirar aquellos con norma mayor de 1. Se pide calcular la siguientes distribuciones de probabilidad analíticamente,

1. Sea $p(r)$ la distribución de probabilidad en función de la coordenada radial r .
2. Es evidente que $(x, y, 0)$ es el resultado de proyectar un punto (x, y, z) sobre el plano $z = 0$. Calcular la distribución $q(x, y)$ de dicha proyección.
3. Sea $s(r)$ la distribución radial de la distribución anterior $q(x, y)$

Para cada una de las distribuciones anteriores, $p(r)$, $q(x, y)$ y $s(r)$, escribir el código para calcularlas numéricamente. Escribir el histograma normalizado para cada una de ellas.

Dibujar lo obtenido numéricamente y lo calculado analíticamente y comprobar que coinciden.

5.8.5. Problema 5

Generar puntos uniformemente sobre la parábola $y = 4x^2$ en el intervalo $x \in [-1, 8]$ usando el método de la longitud de arco. Dibujar un histograma usando el parámetro longitud de arco como eje coordenado para comprobar que la distribución obtenida es uniforme. ¿Cómo es la distribución (Histograma) considerando la variable x como coordenada en el Histograma?

5.8.6. Problema 6

Considerar la curva $y = x \sin(x)$. Generar puntos en el intervalo $x \in [0, 2\pi]$ usando el método de la pendiente. Pensar alguna forma de comprobar que la solución es la correcta, y verificarlo.

5.8.7. Problema 7

Proponer una distribución razonable para la distribución de estaturas en España y escribir una fórmula analítica para ella. Generar posteriormente un conjunto de personas de acuerdo a dicha distribución de probabilidad. Comprobar finalmente que los resultados están de acuerdo con la distribución propuesta. Para encontrar una distribución aproximadamente correcta, buscar en Internet datos al respecto.

5.8.8. Problema 8

Proponer una distribución razonable para la distribución de ingresos anuales de las personas en España y escribir una fórmula analítica para ella. Generar posteriormente un conjunto de personas de acuerdo a dicha distribución de probabilidad. Comprobar finalmente que los resultados están de acuerdo con la distribución propuesta. Para encontrar una distribución aproximadamente correcta, buscar en Internet datos al respecto. Hacemos notar que la distribución de este Ejercicio es completamente diferente cualitativa y cuantitativamente de la del Ejercicio anterior.

5.8.9. Problema 9

Sea un numero natural N mayor de 2 y otro numero Natural Q . Escribir completamente una función que recibe N y Q y devuelve un número entero en el intervalo $[Q, Q + N - 1]$ distribuido proporcionalmente a \sqrt{x} . Utilizar el método de la función de distribución.

Este ejercicio corresponde a un problema de examen, y mostramos a continuación una solución correcta.

```
void Prod_Raiz(int N, int Q, int* p)
{
#define P_MAX 1000
    int i, j;
    double Norm, sum, omega;
    double tau[P_MAX];

    Norm = 0;
    for (i = Q; i < N + Q; i++)
        Norm += sqrt((double)i);

    for (i = Q; i < N + Q; i++)
    {
        sum = 0;
        for (j = Q; j <= i; j++)
            sum += sqrt((double)j);

        tau[i] = sum / Norm;
    }
}
```

```
    }
omega = fran;
i = Q;
while (omega > tau[i] && i < (N + Q))
    i++;

*p = i;
}
```

5.9. Código en C

5.9.1. Conceptos de Programación: Datos, truncamiento, manejo de bits, macros avanzados

Usaremos el generador de números random de Parisi-Rapuano, donde es importante comprender el truncamiento de variables, su representación y rangos, así como las operaciones entre bits como AND, XOR. También el uso de máscaras para enteros. Usaremos también directivas `define` para definir funciones complejas, en concreto el generador random de Parisi-Rapuano, lo que permite portarlo y usarlo en otros programas fácilmente.

5.9.2. Generación de distribuciones con el Método de Metropolis

```
/*
Genero una distribucion de probabilidad dada por
la funcion f(x) Usando el metodo de Metropolis.
Se usa el generador de Parisi-Rapuano
Alfonso Tarancón
*/
#include <stdio.h>
#include <stdlib.h>

#define NormRANu (2.3283063671E-10F)

#define Frec_Max 100
unsigned int irr[256];
unsigned int ir1;
unsigned char ind_ran,ig1,ig2,ig3;
int frec[Frec_Max];

extern float f(float y);
extern float Random(void);
extern void ini_ran(int SEMILLA);

main()
{
    float x_min,x_max;
    float x,x_old;
    int i,NITER;
    float eps,cociente;
    float delta;

    ini_ran(123456789); //Inicializo el generador random
    eps=0.5; //Valor para el movimiento en x

    x_min=0; //Intervalo de p(x)
    x_max=5;

    delta=(float)(x_max-x_min)/Frec_Max;

    NITER=10000000;

    x=0.5; //valor inicial para x, necesario para Metropolis

    for(i=0;i<Frec_Max;i++)
        frec[i]=0;

    for(i=0;i<NITER;i++)
    {
        x_old=x+eps*(0.5-Random()); //Me muevo en x poco a poco
        if(x_old<x_min)x_old=x_max-x_min+x_old; //Toro
        if(x_old>x_max)x_old=x_min-x_max+x_old; //Toro
    }
}
```

```

cociente=f(x_old)/f(x);

    if(Random()<cociente)
{
    x=x_old;
    //printf("acepto\n");
}
    //printf("x=%f\n",x);

    freq[(int)((x-x_min)/delta)]++;

}

for(i=0;i<Frec_Max;i++)
    printf("%f %f\n",x_min+delta*i,Frec_Max*(float)freq[i]/((x_max-x_min)*NITER));

}

float f(float y)
{
    return y*y;
}

float Random(void)
{
    float r;

    ig1=ind_ran-24;
    ig2=ind_ran-55;
    ig3=ind_ran-61;
    irr[ind_ran]=irr[ig1]+irr[ig2];
    ir1=(irr[ind_ran]^irr[ig3]);
    ind_ran++;
    r=ir1*NormRANu;
    //printf("r=%f\n",r);
    return r;
}

void ini_ran(int SEMILLA)
{
    int INI,FACTOR,SUM,i;

    srand(SEMILLA);

    INI=SEMILLA;
    FACTOR=67397;
    SUM=7364893;

    for(i=0;i<256;i++)
    {
        INI=(INI*FACTOR+SUM);
        irr[i]=INI;
    }
    ind_ran=ig1=ig2=ig3=0;
}

```

5.9.3. Generación de puntos homogéneos sobre una Superficie

```

// Genera puntos distribuidos uniformemente
//sobre la superficie z=a(x*x+y*y)
//Alfonso Tarancón
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#define Random_C ((float)rand()/RAND_MAX) // genera un numero random [0,1)

void Histogram(double *data,double *Hist, int N_data,int N_Intervalos,
               double *d,double *m, double *M);

FILE *out;
#define M_MAX 1000000
double puntos[M_MAX];
main()
{

#define Inter_Max 10000
double H[Inter_Max];
double delta,minimo,maximo;
double a,R,rn_mean,theta,TWOPi,r,omega,p_r,p_R_Max;
int M,n,i,N_Inter;

printf("Escribe el valor de a que su usa para z=a(x+x*y*y): ");
scanf("%lf",&a);

printf("Escribe el radio donde se inscribe el calculo: ");
scanf("%lf",&R);

printf("Escribe la potencia de r para valor medio (int): ");
scanf("%d",&n);

printf("Escribe el numero de puntos a generar: ");
scanf("%d",&M);

printf("a=%lf,R=%lf, M=%d\n",a,R,M);

rn_mean=0;;
TWOPi=4.0*asin(1.0);
p_R_Max=R*sqrt(1.0+4.0*a*a*R*R);
printf("P_R_Max=%lf\n",p_R_Max);
out=fopen("evol_V.dat","wt");
for(i=0;i<M;i++) // bucle en sucesos
{
    //Genero theta plano
    theta=TWOPi*Random_C;

    //Genero el radio entre (0,R) con probabilidad
    //proporcional a r. Metodo de la altura.

    do
    {
        r=Random_C*R;
        p_r=r*sqrt(1.0+4.0*a*a*r*r);

        omega=Random_C*p_R_Max;
    }while(omega>p_r);

    puntos[i]=r;
    fprintf(out,"%lf %lf\n",theta,r);

    rn_mean+=pow(r,n);
    //<x>
}

fclose(out);
printf("Valor medio de r%d =%lf\n",n,rn_mean/M);

//Solo vale para R>>1
printf("Valor teorico = %lf", (3.0/(n+3))*pow(R,n));
//Ejemplod de numeros

```

```

//a=0.01, R=1 n=1 Valor numerico= 0.666554, Valor teorico = 0.75 (Regular)
//a=0.1, R=1 n=2 Valor numerico= 0.501439, Valor teorico = 0.6 (Regular)
//a=3, R=1 n=3 Valor numerico= 2.246423, Valor teorico = 2.25

//a=0.01, R=10 n=1 Valor numerico= 06.678760, Valor teorico = 7.5
//a=0.1, R=10 n=2 Valor numerico= 55.893709, Valor teorico = 60.0
//a=3, R=10 n=3 Valor numerico= 499.684694, Valor teorico = 500.0

//a=0.01, R=100 n=1 Valor numerico= 71.471693, Valor teorico = 75
//a=0.1, R=100 n=2 Valor numerico= 5988.658, Valor teorico = 6000 (Bien)
//a=3, R=100 n=3 Valor numerico= 499774.411, Valor teorico = 500000 (Bien)

N_Inter=50;
Histogram(puntos,H,M,N_Inter,&delta,&minimo,&maximo); //Calculamos el histograma

#ifndef DEBUG
    for(i=0;i<N_Inter;i++) //Escribimos en pantalla los datos
    printf("%d %f %f\n",i,i*delta+minimo,H[i]);
#endif

    out=fopen("hist.dat","wt");
    for(i=0;i<N_Inter;i++) //Escribimos en un archivo los datos
    fprintf(out,"%d %f %f\n",i,i*delta+minimo,H[i]);
    fclose(out);

}

void Histogram(double *data,double *Hist, int N_data,
               int N_Intervalos,double *d,double *m, double *M)
{
/* Genera un histograma. Calcula, en funcion de los datos el minimo y maximo
   de los mismos para ajustar mejor los intervalos.

   *data-> input Datos sobre los que se genera el histograma
   *Hist-> output Histograma calculado
   N_data-> input Numero de datos
   N_Intervalos-> input Numero de intervalos del histograma
   *d -> output Medida de cada intervalo del histograma
   *m -> output Valor minimo de los datos
   *M -> output Valor maximo de los datos
   */
   int i,Indice;
   double del,min,max,Norm;

   for(i=0;i<N_Intervalos;i++)//Inicializo
   Hist[i]=0;

   min=10000000;
   max=-10000000;

   for(i=0;i<N_data;i++) //Calculo el minimo y el maximo
   {
if(data[i]<min)min=data[i];
if(data[i]>max)max=data[i];
}

   del=(max-min)/N_Intervalos;
   if(del==0)
   {
printf("Error: No se pueden calcular los intervalos; Max=%lf,Min=%lf\n",max,min);
exit(1);
}

   for(i=0;i<N_data;i++) //***** Calculo el histograma *****
   {
Indice=(data[i]-min)/del;
Hist[Indice]++;
}
}

```

```

#ifndef DEBUG
printf("x=%f, H[%d]=%f\n", Indice*del+min, Indice, Hist[Indice]);
#endif
}
*d=del;
*m=min;
*M=max;

/* Ahora normalizo */

Norm=1.0/(N_data*del);
for(i=0;i<N_Intervalos;i++)
Hist[i]*=Norm;
}

```

5.9.4. Puntos sobre una superficie esférica (Método Geométrico)

```

// Generacion de un numero uniformemente distribuido sobre una esfera

// Metodo de inscribir la esfera en un cubo

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define Random_C (rand() / ((double)RAND_MAX + 1))

double norm,v[3];
int M,i,in;
FILE *fout;

main()
{
    M=10000;
    fout=fopen("esfera.dat","wt");

    for(in=0;in<M;in++) // bucle en sucesos
    {
        norm=2.0;
        while(norm>1.0)
        {
            norm=0;
            for(i=0;i<3;i++)
            {
                v[i]=2*Random_C-1.0;
                norm+=v[i]*v[i];
            }
        }

        for(i=0;i<3;i++)
            v[i]/=sqrt(norm);

        fprintf(fout,"%lf %lf %lf\n",v[0],v[1],v[2]);
    }
    fclose(fout); //Los datos pueden visualizarse con splot en gnuplot
}

```

5.9.5. Cálculo por Monte Carlo del volumen de una esfera en d dimensiones

```

// Calculo del volumen de una esfera en dimension D
// Metodo de Monte Carlo: Lanzar puntos en un cubo
// donde la esfera esta inscrita.
// El cociente de puntos dentro de la esfera
// con el total, da el volumen
// Alfonso Tarancón

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

#define Random_C (rand()/(double)RAND_MAX+1)

int i,is,dentro,Total,dim;
double sum,x,R,V_esf;
FILE *out;
double V_temp;

main()
{
    int M;
    int mesfr=100;

    printf("Escribe la dimension\n");
    scanf("%d",&dim);

    printf("Escribe EL RADIO\n");
    scanf("%lf",&R);

    printf("Escribe el numero de puntos a generar\n");
    scanf("%d",&M);

    printf("dim=%d,R=%f, M=%d\n",dim,R,M);
    dentro=0;
    Total=M;

    out=fopen("evol_V.dat","wt");

    for(i=0;i<M;i++) // bucle en sucesos
    {
        sum=0;
        for(is=0;is<dim;is++)
        {
            x=R*(0.5-Random_C)*2;
            sum+=x*x;
        }
        if(sum<=(R*R))dentro++;

        if((i&mesfr)==0 && i>0)
        {
            V_temp=((double)dentro/i)*pow(2*R,(double)dim);
            fprintf(out,"%d %f\n",i,V_temp);
        }
    }
    fclose(out);
    V_esf=((double)dentro/Total)*pow(2*R,(double)dim);

    printf("dentro=%d, Volumen=%f, Test=%f\n",dentro,V_esf,pow(2*R,(double)dim));
}

```

5.9.6. Sorteo Uniforme

```

/*
Genero un sorteo uniforme en el intervalo [x_min,x_max+delta-1]
Construyo el histograma con intervalos igual a un numero
Alfonso Tarancón
*/

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

#define Frec_Max 5000

//Generador de Parisi Rapuano como Macro

#define FNORM      (2.3283063671E-10F)
#define RANDNEW    ( (irr[ip++]=irr[ip1++]+irr[ip2++]) ^ irr[ip3++] )
#define RANDOM     (FNORM*RANDNEW)

unsigned char ip=128,ip1=128-24,ip2=128-55,ip3=128-61;
unsigned int irr[256];

int freq[Frec_Max];

extern void ini_ran(int SEMILLA);
int Acuerdo,millares,ganador;

FILE *fout;

main()
{
    float x_min,x_max;
    float x_old,x_delta;
    int i,NITER;

    ini_ran(123456789);

    //El sorteo es entre [3000,3010]

    x_min=3000;
    x_delta=11; //1 de mas
    x_max=4000+x_delta;

    NITER=1000000;

    for(i=0;i<Frec_Max;i++)
        freq[i]=0;

    for(i=0;i<NITER;i++)
    {
        Acuerdo=0;
        while(Acuerdo==0)
        {
            x_old=1000*FRANDOM;
            millares=3+2*FRANDOM; //3 o 4 (mil)

            if(x_old<=x_delta)Acuerdo=1;

            if(x_old>x_delta)
            {
                if(millares==4)
                    Acuerdo=0; //redundante pero mas claro
                else
                    Acuerdo=1;
            }
        }
        ganador=x_old+millares*1000;
        freq[ganador]++;
    }
}

```

```
}

for(i=0;i<Freq_Max;i++)
    printf("%d %d\n",i,freq[i]);

fout=fopen("distribucion.dat","wt");

for(i=0;i<Freq_Max;i++)
    fprintf(fout,"%d %d\n",i,freq[i]);

fclose(fout);
}

void ini_ran(int SEMILLA)
{
    intINI,FACTOR,SUM,i;

    srand(SEMILLA);

   INI=SEMILLA;
    FACTOR=67397;
    SUM=7364893;

    for(i=0;i<256;i++)
    {
        INI=(INI*FACTOR+SUM);
        irr[i]=INI;

        //printf("irr[%d]=%d\n",i,irr[i]);
    }
}
```

5.10. Apéndice: Teoría para generar puntos sobre una Superficie Parabólica

Representamos la superficie como

$$\vec{x}(r, \theta) = (r \cos \theta, r \sin \theta, ar^2)$$

y el vector normal toma la forma

$$\vec{N}(r, \theta) = (2ar^2 \cos \theta, 2ar^2 \sin \theta, -r \sin^2 \theta - r \cos^2 \theta)$$

y su norma

$$|\vec{N}(r, \theta)| = r \sqrt{1 + 4a^2r^2}$$

Por tanto la distribución de puntos sobre la superficie es

$$p(r, \theta) dr d\theta = d\theta r \sqrt{1 + 4a^2r^2} dr$$

o equivalentemente

$$p(r, \theta) \propto r \sqrt{1 + 4a^2r^2}$$

Nota: Podríamos hacer todo el cálculo partiendo de otra representación de la superficie, por ejemplo,

$$\vec{x}(x, y, z) = (x, y, a(x^2 + y^2))$$

repitiendo todos los pasos de este ejercicio. El alumno puede hacerlo como ejercicio, comprobando que se llega a los mismos resultados.

Necesitaríamos ahora calcular la constante de proporcionalidad de modo que $p(r, \theta)$ cumpla que su integral en r, θ sea 1, lo cual es complicado. Pero en este caso podemos realizar los cálculos sin conocer esta constante. Efectivamente, llamemos A la constante de normalización; para calcular los valores medios, por ejemplo $\langle r^n \rangle$, debemos calcular

$$\langle r^n \rangle = \frac{A \int_0^{2\pi} d\theta \int_0^R r \sqrt{1 + 4a^2r^2} r^n dr}{A \int_0^{2\pi} d\theta \int_0^R r \sqrt{1 + 4a^2r^2} dr} \quad (5.15)$$

y vemos que A cancela.

En cualquier caso estas integrales son difíciles de calcular. Podemos comprobar que todo es correcto usando ciertas aproximaciones que simplifican estas integrales.

Remarcamos que si bien estas aproximaciones sirven para obtener resultados analíticos razonables, el cálculo numérico debe ser realizado con el valor exacto de la distribución, sin ninguna aproximación.

Supongamos que $R \gg 1$, en cuyo caso en la integral domina la región de gran R , y podemos hacer la aproximación

$$r \sqrt{1 + 4a^2r^2} \approx 2ar^2$$

y por tanto para la integral 5.15 tendremos

$$\langle r^n \rangle = \frac{\int_0^{2\pi} d\theta \int_0^R r^{2+n} dr}{\int_0^{2\pi} d\theta \int_0^R r^2 dr} \quad (5.16)$$

Ahora el cálculo es trivial, obteniendo

$$\langle r^n \rangle = \frac{3}{n+3} R^n \quad (5.17)$$

Podemos también aproximar para el caso en que $a \ll 1$ y R no muy grande. En este caso podemos hacer la aproximación $\sqrt{1 + 4a^2r^2} \approx 1$, con lo cual la integral 5.15 toma la forma

$$\langle r^n \rangle = \frac{\int_0^{2\pi} d\theta \int_0^R r^{1+n} dr}{\int_0^{2\pi} d\theta \int_0^R r dr} = \frac{2}{2+n} R^n \quad (5.18)$$

Con todo lo anterior el núcleo de cálculo del programa para el cálculo numérico exacto tomará la forma

```

rn_mean=0;
for(i=0;i<M;i++) // bucle en sucesos
{
    //Genero theta plano
    theta=TWOPi*Random_C;

    //Genero el radio entre (0,R) con probabilidad
    //proporcional a r. Método de la altura.

    do
    {
        r=Random_C*R;
        p_r=r*sqrt(1.0+4.0*a*a*r*r);

        omega=Random_C*p_R_Max;
    }while(omega>p_r);

    puntos[i]=r;

    rn_mean+=pow(r,n);
}

printf("Valor medio de r%d =%lf\n",n,rn_mean/M);

```

Damos a continuación tres casos particulares para que se pueda comprobar si el trabajo es correcto

```

a=0.1, R=1      n=2 Valor numerico exacto= 0.5014, Valor teorico aproximado = 0.6
a=0.1, R=10     n=2 Valor numerico exacto= 55.894, Valor teorico aproximado = 60.0
a=0.1, R=100    n=2 Valor numerico exacto= 5988.6, Valor teorico aproximado = 6000

```


Capítulo 6

Análisis estadístico y cálculo de errores

Cuando se realiza un experimento en la Naturaleza, los resultados obtenidos tienen un cierto error. El conocimiento de este error es tan importante como el dato en sí mismo, pues nos dice cómo de fiable es el dato obtenido. Dada una cantidad a , de la que su valor estimado es x y dicha estimación tiene un error δ , esto se escribe como

$$a = x \pm \delta$$

El error significa por convenio que el valor de a se sitúa en el intervalo $[x - \delta, x + \delta]$ con una probabilidad del 68 %. Pero el valor correcto puede estar incluso más lejos, con menor probabilidad. Estos conceptos no son complicados, pero sí a veces desconocidos para el alumno, por lo que dedicaremos un capítulo a introducir algunas ideas básicas de estadística y cálculo de errores. Por fuerza, daremos un especial énfasis a los conceptos, no tanto a las demostraciones, que pueden encontrarse en numerosos libros de texto y fuentes de Internet.

En capítulos anteriores, hemos definido diferentes distribuciones. En ese capítulo introduciremos cantidades básicas que nos dan información sobre dichas distribuciones, como la media o la varianza.

En los experimentos reales o en simulaciones del ordenador, obtenemos resultados y series de datos de los que queremos extraer información. Estos datos responderán a alguna densidad de probabilidad que en general desconocemos.

Es decir, ahora disponemos de una secuencia de datos $\{x_0, x_1, \dots\}$ de los que no sabemos su densidad de probabilidad, pero de los que queremos inferir toda la información posible, como los valores de la media o la varianza, o su histograma (que es equivalente, como hemos visto, a su densidad de probabilidad).

En este capítulo definiremos las cantidades básicas para las densidades de probabilidad, y a continuación discutiremos cómo calcular estas cantidades, y como estimar la fiabilidad de las mismas, cuando lo que tenemos es una secuencia de datos obtenidos de experimentos reales o en ordenador.

También estudiaremos los errores que se producen en el cálculo numérico debido a las limitaciones de los algoritmos y de los ordenadores (precisión, tiempo finito...).

6.1. Estadística básica

Dada una densidad de probabilidad $p(x)$ con x una variable continua, definimos la media

$$\langle x \rangle = \int xp(x) \quad (6.1)$$

y la varianza como

$$V = \langle x^2 \rangle - \langle x \rangle^2 = \int x^2 p(x) dx - \left(\int x p(x) dx \right)^2 \quad (6.2)$$

Notar que V es también $\langle (x - \langle x \rangle)^2 \rangle$:

$$\begin{aligned} \langle (x - \langle x \rangle)^2 \rangle &= \int (x - \langle x \rangle)^2 p(x) dx = \int (x^2 + \langle x \rangle^2 - 2x\langle x \rangle) p(x) dx = \\ &\int x^2 dx + \langle x \rangle^2 \int p(x) dx - 2\langle x \rangle \int x p(x) dx = \langle x^2 \rangle - \langle x \rangle^2 \end{aligned} \quad (6.3)$$

De este modo V es una medida de lo que se separa en media la distribución del valor medio, para lo cual debemos considerar la separación al cuadrado, pues como es fácil de demostrar $\langle (x - \langle x \rangle) \rangle$ es siempre nulo.

En el caso de que tengamos una variable discreta $\{x_k\}$, con probabilidades $p(x_k)$, la media será

$$\langle x \rangle = \sum_k x_k p(x_k) \quad (6.4)$$

y la varianza

$$V = \langle x^2 \rangle - \langle x \rangle^2 = \sum_k x_k^2 p(x_k) - \left(\sum_k x_k p(x_k) \right)^2 \quad (6.5)$$

Tambien en este caso ocurre que $V = \langle (x - \langle x \rangle)^2 \rangle$

Definimos la Desviación Standard σ , como

$$\sigma = \sqrt{V} \quad (6.6)$$

Nota: Para estudiar como es en media la separación de los datos de la media, uno podría pensar que lo correcto sería calcular $\langle x - \langle x \rangle \rangle$. Sin embargo como se comprueba trivialmente, esta cantidad es siempre cero. Para calcular algo similar, debemos eliminar el problema de sumar cantidades positivas y negativas que se anulen: para ello calculamos precisamente $\langle (x - \langle x \rangle)^2 \rangle$, y luego tomamos la raiz cuadrada, obteniendo finalmente un número (positivo) que nos indica cuanto se separan en media los datos de la media.

6.1.1. Valores medios

Dada una densidad de probabilidad $p(x)$ y una función de x , $g(x)$, al ir generando valores para x de acuerdo con $p(x)$, la función $g(x)$ irá tomando los valores correspondientes. El valor medio de $g(x)$ vendrá dado por

$$\langle g(x) \rangle = \int g(x) p(x) dx \quad (6.7)$$

Un caso simple es cuando $g(x) = x$. Aquí estamos hablando en realidad del valor medio de x , o de la media de x . Para encontrar su valor bastará realizar la integral

$$\langle x \rangle = \int x p(x) dx \quad (6.8)$$

El caso de que $g(x) = \text{constante} = a$ es también simple, obteniendo para cualquier $p(x)$,

$$\langle a \rangle = \int a p(x) dx = a \int p(x) dx = a \times 1 = a \quad (6.9)$$

Son útiles en muchos casos los *momentos* de la distribución, definidos como

$$\langle x^n \rangle = \int x^n p(x) dx, \quad n \in \mathbb{N}. \quad (6.10)$$

Nota: No para todas las densidades de probabilidad están bien definidos los momentos; éstos pueden diverger en algunos casos, para ciertos valores de n . Considerar los ejemplos de la sección 4.1.2 y estudiar si los diferentes momentos son finitos para cada una de ellas. Como corolario, construir alguna densidad de probabilidad que tenga varianza infinita.

6.1.2. Cálculo de valores medios con datos numéricos

Estudiemos el caso práctico de calcular valores medios para una cierta distribución $p(x), x \in [a, b]$ continua (el caso discreto es similar). Tenemos dos opciones. En primer lugar podemos elegir un conjunto elevado de puntos en el intervalo $[a, b]$, por ejemplo N valores equidistribuidos, y ahora simplemente calculamos

$$\langle x \rangle = \frac{1}{N} \sum_1^N x_i p(x_i) \quad \langle x^2 \rangle = \frac{1}{N} \sum_1^N x_i^2 p(x_i) \dots$$

Esta forma es en general correcta, aunque presenta el problema de que si la distribución está muy picada en torno a algún valor, podemos cometer grandes errores en la evaluación. También es prohibitiva si trabajamos en R^n con n alto.

La segunda opción consiste en generar los puntos $\{x_i\}$ no de forma uniforme en el intervalo, sino de manera proporcional a su probabilidad. Tras obtener N puntos generados de acuerdo a la probabilidad $p(x)$ podemos calcular los valores medios mediante

$$\langle x \rangle = \frac{1}{N} \sum_1^N x_i \langle x^2 \rangle = \frac{1}{N} \sum_1^N x_i^2 \dots$$

6.2. Densidades de probabilidad Continuas

Estudiemos a continuación algunas densidades de probabilidad continuas muy usuales.

6.2.1. Distribución Uniforme

Sea el intervalo $[a, b]$. La distribución uniforme en dicho intervalo, debidamente normalizada sabemos que es

$$p(x) = \frac{1}{b-a} \quad (6.11)$$

Media

$$\langle x \rangle = \int x p(x) dx = \int_a^b \frac{1}{b-a} x dx = \frac{b+a}{2} \quad (6.12)$$

Varianza

$$V = \langle x^2 \rangle - \langle x \rangle^2 = \int x^2 p(x) dx - \left(\int x p(x) dx \right)^2 \quad (6.13)$$

El segundo término es la media al cuadrado, ya conocido. Ahora debemos calcular el primero,

$$\langle x^2 \rangle = \int \frac{1}{b-a} x^2 dx = \frac{1}{b-a} \frac{1}{3} (b^3 - a^3) \quad (6.14)$$

obteniendo finalmente

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2 = \frac{b^3 - a^3}{3(b-a)} - \frac{(b+a)^2}{4} = \frac{(b-a)^2}{12} \quad (6.15)$$

Nota: Para la densidad de probabilidad uniforme y continua, con $x \in [0, 1]$, de uso tan frecuente en programación, de acuerdo con 6.15, tendremos $\sigma^2 = 1/12$.

6.2.2. Distribución gaussiana

Para seguir el desarrollo de esta sección, es necesario conocer las integrales definidas donde aparecen gaussianas y potencias de x . En el Apéndice 6.9 se dispone de un resumen.

Consideremos la densidad de probabilidad,

$$p(x) = Ae^{-\frac{(x-a)^2}{2b}}, x \in [-\infty, \infty] \quad (6.16)$$

Es bien conocido que esta función tiene integral finita; recordamos el resultado

$$\int_{-\infty}^{+\infty} e^{-\alpha(x-a)^2} = \sqrt{\frac{\pi}{\alpha}} \quad (6.17)$$

Notar que al ser la integral invariante (tanto el integrando como el intervalo de integración) por el cambio x por $x + c$, la integral no depende de a .

En nuestro caso,

$$\int_{-\infty}^{+\infty} e^{-\frac{(x-a)^2}{2b}} = \sqrt{2b\pi} \quad (6.18)$$

Por tanto, para que $p(x)$ esté normalizada, A deberá tomar el valor

$$A = \frac{1}{\sqrt{2b\pi}} \quad (6.19)$$

con lo que finalmente

$$p(x) = \frac{1}{\sqrt{2b\pi}} e^{-\frac{(x-a)^2}{2b}} \quad (6.20)$$

Media

Sabemos que para cualquier densidad de probabilidad el valor esperado de una constante es la constante misma.

$$\langle C \rangle = C \quad (6.21)$$

Entonces,

$$\begin{aligned} \langle x \rangle &= \int xp(x)dx = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{(x-a)^2}{2b}} x dx \\ x - a &= t, dx = dt \\ \langle x \rangle &= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{t^2}{2b}} (t + a) dt = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{t^2}{2b}} t dt + \langle a \rangle \end{aligned} \quad (6.22)$$

Recordemos que

$$\int_{-\infty}^{\infty} e^{-\alpha t^2} t dt = 0 \quad (6.23)$$

y dada 6.9 tendremos que

$$\langle x \rangle = 0 + a = a \quad (6.24)$$

y finalmente obtenemos

$$\langle x \rangle = a \quad (6.25)$$

Varianza

$$V = \langle x^2 \rangle - \langle x \rangle^2 = \int x^2 p(x) dx - (\int x p(x) dx)^2 \quad (6.26)$$

Ya conocemos

$$\langle x \rangle^2 = a^2 \quad (6.27)$$

Ahora necesitamos calcular $\langle x^2 \rangle$

$$\begin{aligned} \langle x^2 \rangle &= \int x^2 p(x) dx = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{(x-a)^2}{2b}} x^2 dx = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{t^2}{2b}} (t+a)^2 dt \\ \langle x^2 \rangle &= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{t^2}{2b}} (t^2 + a^2 + 2ta) dt \end{aligned} \quad (6.28)$$

por tanto

$$\langle x^2 \rangle = a^2 + \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{t^2}{2b}} t^2 dt \quad (6.29)$$

Haciendo un cambio de variable, obtenemos,

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-\frac{t^2}{2b}} t^2 dt = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2b\pi}} e^{-u^2} \sqrt{2b} u^2 2b du = \int_{-\infty}^{\infty} 2 \frac{b}{\sqrt{\pi}} e^{-u^2} u^2 du \quad (6.30)$$

Recordando el Apéndice 6.9, obtenemos

$$\int_{-\infty}^{\infty} 2 \frac{b}{\sqrt{\pi}} e^{-u^2} u^2 du = 2 \frac{b}{\sqrt{\pi}} \sqrt{\pi} \frac{1}{2} = b \quad (6.31)$$

con lo cual

$$\langle x^2 \rangle = a^2 + b \quad (6.32)$$

obteniendo finalmente

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2 = b \quad (6.33)$$

Dado que a coincide con la media y b con la varianza, es habitual escribir la densidad de probabilidad gaussiana como

$$p(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\langle x \rangle)^2}{2\sigma^2}} \quad (6.34)$$

Si queremos visualizar gráficamente el significado de σ , vemos de la expresión anterior que cuando x se separa de la media una σ , el valor de $p(x)$ pasa de su valor máximo a un valor \sqrt{e} menor.

El caso de la densidad de probabilidad gaussiana, un resultado que se desprende del Apéndice 6.9 es que todos sus momentos son finitos.

6.3. Dispersión para la suma de variables aleatorias

Demostraremos en esta sección que

Dadas dos variables descorrelacionadas, la varianza de la suma es la suma de las varianzas.

Recordemos primero el concepto de independencia estadística. Dados dos sucesos A y B , se dice que son estadísticamente independientes si la probabilidad de que sucedan ambos coincide con el producto de las probabilidad de que sucedan cada uno de ellos. Vamos a concretar este concepto en el caso de que exista una densidad de probabilidad. Consideremos dos cantidades x e y y una función de dos variables $p(x, y)$ que satisface:

$$p(x, y) \geq 0; \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} p(x, y) dx dy = 1 \quad (6.35)$$

$p(x, y)$ diremos que es la densidad de probabilidad conjunta de x e y . Las funciones

$$p_1(x) = \int_{-\infty}^{\infty} p(x, y) dy \quad (6.36)$$

$$p_2(y) = \int_{-\infty}^{\infty} p(x, y) dx \quad (6.37)$$

son las densidades de probabilidad de x (cualquiera que sea el valor de y) y de y (cualquiera que sea el valor de x). Entonces, de la independencia estadística entre x e y se sigue que:

$$p(x, y) = p_1(x) p_2(y) \quad (6.38)$$

Para funciones de dos variables, definimos el valor medio como

$$\langle f(x, y) \rangle = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) p(x, y) dx dy \quad (6.39)$$

Si x e y son estadísticamente independientes ($p(x, y) = p(x)p(y)$), podemos escribir :

$$\langle xy \rangle = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x y p(x, y) dx dy = \int_{-\infty}^{\infty} x p_1(x) dx \int_{-\infty}^{\infty} y p_2(y) dy = \langle x \rangle \langle y \rangle \quad (6.40)$$

La independencia también permite encontrar una relación muy importante para las varianzas: la varianza de la suma es la suma de las varianzas¹

$$\begin{aligned} \text{Var}(x + y) &= \langle (x + y)^2 \rangle - \langle x + y \rangle^2 = \langle x^2 + y^2 + 2xy \rangle - (\langle x \rangle + \langle y \rangle)^2 \\ &= \langle x^2 \rangle - \langle x \rangle^2 + \langle y^2 \rangle - \langle y \rangle^2 = \text{Var}(x) + \text{Var}(y) \end{aligned} \quad (6.41)$$

o en términos de las dispersiones (raíz cuadrada de la varianza):

$$\sigma_{x+y} = \sqrt{\sigma_x^2 + \sigma_y^2} \quad (6.42)$$

y para el caso particular en que $x \equiv y$, tendremos

$$\sigma_{2x} = \sqrt{2}\sigma_x$$

¹Si no hay independencia estadística entre x e y , la relación sería: $\text{Var}(x + y) = \text{Var}(x) + \text{Var}(y) + 2\text{Covar}(x, y)$, donde $\text{Covar}(x, y) \equiv \langle xy \rangle - \langle x \rangle \langle y \rangle$.

6.3.1. Varianza de la suma de dos distribuciones idénticas

Un caso de especial interés es el de cantidades independientes e idénticamente distribuidas (con la misma densidad de probabilidad). La varianza de la semisuma es

$$\text{Var}\left(\frac{x+y}{2}\right) = \frac{\text{Var}(x+y)}{4} = \frac{\text{Var}(x) + \text{Var}(y)}{4} = \frac{\text{Var}(x)}{2} \quad (6.43)$$

o en términos de las dispersiones

$$\sigma_{\frac{x+y}{2}} = \frac{1}{2} \sqrt{\sigma_x^2 + \sigma_y^2} \quad (6.44)$$

que para $x \equiv y$ toma la forma

$$\sigma_{\frac{x+y}{2}} = \frac{1}{\sqrt{2}} \sigma_x$$

La generalización al promedio, \bar{x} (Ec. (6.49)), de N cantidades independientes e idénticamente distribuidas $\{x_0, \dots, x_{N-1}\}$ es inmediato:

$$\sigma_{\bar{x}} = \sigma_{x_i} / \sqrt{N} \quad (6.45)$$

6.3.2. Ejemplos: Números y Juegos de Azar

Consideremos el caso de un número plano en el intervalo $[0, 1]$. Si consideramos una nueva serie, cuyos números son los obtenidos haciendo la media de 500 consecutivos de los anteriores, la nueva sucesión será (casi) de la forma $\{0.5, 0.5, 0.5, \dots\}$.

Sea ahora un juego de azar en el que lanzamos una moneda al aire (sin trucar) y apostamos a cara y cruz, ganando o perdiendo lo apostado. Si nos preguntamos cuánto perdemos o ganamos al cabo de N partidas, es decir, cuánto nos separamos de la media (que es 0) podemos afirmar que la cantidad se comportará como \sqrt{N} (Recordad 6.42). Y si lo que nos preguntamos es cuánto ganaremos o perderemos por partida al cabo de N partidas, la respuesta es $\sqrt{N}/N = 1/\sqrt{N}$, como se indicaba en 6.45. En resumen: si no hay trampas, jugando muchas partidas, lo que perderemos o ganaremos por partida tiende a Cero, pero eso no es de mucho consuelo, pues en términos de dinero absoluto, lo que ganaremos o perderemos, tiende a infinito...

6.3.3. Calculo explícito para $x, y \in [0, 1]$

Consideremos dos funciones de distribución planas en el intervalo $[0, 1]$. Sabemos que toman la forma

$$p(x) = 1, p(y) = 1$$

Queremos calcular ahora de forma analítica cual es la distribución para la variable $z = x + y$ (o también $y = z - x$). Dicho de otro modo, nos preguntamos:

¿Cuál es la probabilidad de obtener un valor de z dado generando los valores de x e y ?

Un valor dado de z aparecerá cuando hayamos obtenido un cierto valor de x y luego el valor de y haya sido precisamente $z - x$. Ambos sucesos son independientes, por tanto la probabilidad de que salgan ambos números es el producto de las probabilidades. Es decir

$$p(z) \equiv p(x)p(z-x)$$

El valor de z anterior puede obtenerse a partir de muchos valores de x , y para obtener la probabilidad total debemos sumar (integrar) a todos ellos.

$$p(z) \equiv \int p(x)p(z-x)dx \quad (6.46)$$

Nos queda ahora fijar el límite de integración; la cuestión es que z varía en el intervalo $[0, 2]$, mientras que x, y en el $[0, 1]$. Dado un valor de $z - x \notin [0, 1]$, la expresión 6.46 carece de sentido, pues no está definido $p(x)$ para $x > 1$ o $x < 0$. Por tanto debemos considerar dos casos por separado; Si $z < 1$ la expresión 6.46 es válida. Pensemos que z vale 0.8 por ejemplo. En este caso,

x podrá tomar valores desde 0 (pues entonces y podrá tomar el valor 0.8) hasta 0.8 (entonces y valdrá 0). Por tanto,

$$p(z) = \int_0^z p(x)p(z-x)dx = \int_0^z 1dx = z(z < 1) \quad (6.47)$$

dado que en el integrando, x no puede ser mayor que z ya que $p(z-x)$ debe estar definida. Veáse la figura 6.1

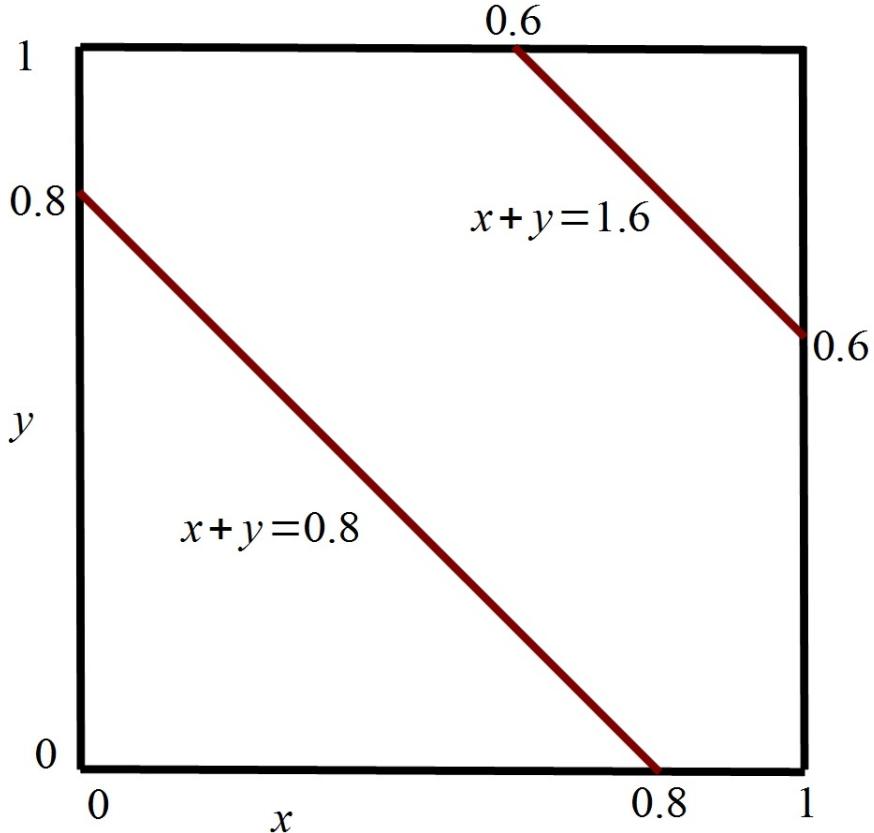


Figura 6.1: Rango de integración en la ecuación 6.46.

Cuando $z > 1$, el rango de variacion de x ahora será entre z y 1. Supongamos que z vale 1.6. Entonces x podrá tomar valores desde 0.6 (en cuyo caso y valdrá 1) hasta 1 (ahora y valdrá 0.6). Entonces,

$$p(z) = \int_{z-1}^1 p(x)p(z-x)dx = 2 - z(z < 1) \quad (6.48)$$

Resumiendo, como puede verse en la figura 6.2

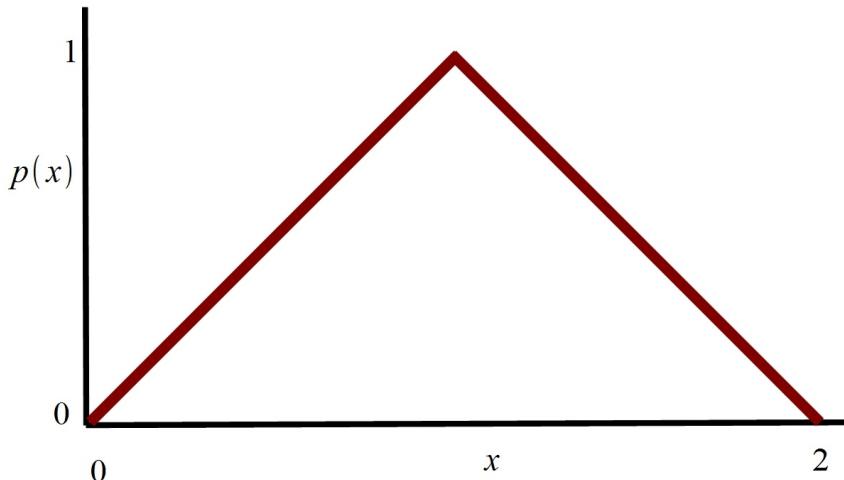
$$p(z) = z(z \in [0, 1]), 2 - z(z \in [1, 0])$$

Notemos que $p(z)$ está correctamente normalizada. Es sencillo calcular la media y la dispersión,

$$\langle z \rangle = 1$$

y

$$\langle z^2 \rangle = \int_0^1 zz^2 dz + \int_1^2 (2-z)z^2 dz = \frac{1}{4} + \frac{11}{12} = \frac{14}{12}$$

Figura 6.2: Distribución de probabilidad de $x + y$.

Para la varianza obtenemos

$$V(x + y) = \langle (x + y)^2 \rangle - \langle (x + y) \rangle^2 = \frac{14}{12} - 1 = \frac{2}{12}$$

como corresponde efectivamente con lo demostrado en esta Sección.

En este caso es trivial de las fórmulas anteriores comprobar que se cumple 6.44, en concreto que

$$\sigma_x = \sigma_y = \frac{1}{\sqrt{12}}; \sigma_{(x+y)/2} = \frac{1}{\sqrt{2}} \frac{1}{\sqrt{12}}$$

Ejercicio: Repetir los cálculos anteriores para el caso de la tirada de dos dados (numerados del 1 al 6).

Ejercicio: Dados x, y dos números aleatorios planos entre $(0, 1)$, consideramos el número $z = (x + y) \% 1.0$, entendiendo la operación $a \% 1$ como que si a es mayor que 1 nos devuelve la parte decimal, y si a es menor que 1 devuelve a . ¿Es plana o no la distribución de z ? ¿Cómo se relaciona esto con lo explicado en esta sección?

6.4. Estimadores estadísticos

En los capítulos siguientes, obtendremos series de números de los cuales queremos extraer información, como su valor esperado, y conocer las limitaciones de los mismos a través del cálculo del error.

Se trata de afrontar la situación inversa a la contemplada hasta ahora: disponemos de una serie de números procedentes de un experimento (real o numérico) y queremos extraer de ellos información sobre la función de distribución $p(x)$. En Estadística, a estos números se les llama valores de *variables aleatorias* para un experimento dado.

Dado este conjunto de números, queremos construir con ellos funciones que nos den una estimación correcta de las cantidades como la media o la varianza definidas anteriormente. Llamarímos *estimadores* a estas funciones.

Si estos estimadores son tales que su valor medio coincide con el valor esperado de la cantidad de partida, diremos que el operador es *no sesgado*; lo llamamos *sesgado* en caso contrario.

Consideremos el caso más simple: tenemos N números reales, estadísticamente independientes, y distribuidos todos ellos según la misma densidad de probabilidad $p(x)$. Sea $\{x_0, x_1, \dots, x_{N-1}\}$ esta sucesión de números.

Definimos el **promedio**

$$\bar{x} = \frac{1}{N} \sum_{j=0}^{N-1} x_j \quad (6.49)$$

que es un **estimador no sesgado** de $\langle x \rangle$ (media de la densidad $p(x)$). Efectivamente, si cada x_i está distribuida según $p(x_i)$, es inmediato comprobar que

$$\langle \bar{x} \rangle = \frac{1}{N} \int_{-\infty}^{\infty} p(x_0) dx_0 \cdots \int_{-\infty}^{\infty} p(x_{N-1}) dx_{N-1} (x_0 + \cdots + x_{N-1}) = \frac{1}{N} (\langle x \rangle + \cdots + \langle x \rangle) = \langle x \rangle$$

También podemos definir un **estimador no sesgado** para la **varianza** de la distribución $p(x)$

$$\bar{V} = \frac{N}{N-1} \frac{1}{N} \sum_{j=0}^{N-1} (x_j - \bar{x})^2 = \frac{N}{N-1} \left(\frac{1}{N} \sum_{j=0}^{N-1} x_j^2 - \bar{x}^2 \right) \quad (6.50)$$

En efecto

$$\begin{aligned} \langle \bar{V} \rangle &= \frac{N}{N-1} \left(\frac{1}{N} \sum_j \langle x_j^2 \rangle - \frac{1}{N^2} \sum_{jk} \langle x_j x_k \rangle \right) = \\ &= \frac{N}{N-1} \left(\langle x^2 \rangle - \frac{1}{N^2} (N \langle x^2 \rangle + N(N-1) \langle x \rangle^2) \right) = V \end{aligned} \quad (6.51)$$

Pero lo más frecuente es interesarse por la propia varianza del estimador \bar{x} (que será el cuadrado de su error estadístico, como veremos más adelante). Recordando 6.3 sabemos que la varianza de una suma de cantidades estadísticamente independientes es la suma de las varianzas. Es decir,

$$\begin{aligned} \text{Var}(\bar{x}) &= \text{Var} \left(\frac{1}{N} \sum_{j=0}^{N-1} x_j \right) = \frac{1}{N^2} \text{Var} \left(\sum_{j=0}^{N-1} x_j \right) = \\ &= \frac{1}{N^2} N \text{Var}(x) = \frac{1}{N} \text{Var}(x) = \frac{V}{N} \end{aligned} \quad (6.52)$$

de modo que \bar{V}/N es un estimador no sesgado de la varianza de \bar{x} . Si llamamos $\bar{\sigma}$ al error estadístico en \bar{x} llegamos finalmente al estimador

$$\bar{\sigma} = \sqrt{\frac{1}{N-1} \left(\frac{1}{N} \sum_{j=0}^{N-1} x_j^2 - \left(\frac{1}{N} \sum_{j=0}^{N-1} x_j \right)^2 \right)} \quad (6.53)$$

En la práctica, dispondremos de una secuencia de números generados en el ordenador, y escribiremos unas sencillas funciones en C que nos permitirán calcular todos los estimadores anteriores.

6.5. Estimación de los Errores en una serie de datos

Cuando realizamos diferentes medidas de una misma cantidad, con la suma de esos números dividido por el número de sumandos (promedio), tenemos una estimación de la media; pero la

pregunta importante es *¿cuál es la desviación que uno espera entre esa estimación y la media exacta?*

Pongámoslo en términos más precisos. Sea una serie de medidas $\{x_i\}$. Podemos realizar un histograma de las mismas, y supongamos que ese histograma nos da una distribución gaussiana. Dicha distribución tiene una cierta varianza. En general, si vamos aumentando el número de datos, la media y la varianza se irán haciendo casi independientes del número de medidas consideradas, con pequeñas fluctuaciones.

Sabemos que cuando tenemos una distribución gaussiana, la probabilidad de que un punto este a una distancia menor que σ de la media es del 68%. Esa es precisamente la definición del error: cuando decimos que una cantidad vale $a \pm \epsilon$ lo que queremos decir es que el valor exacto de a está en el intervalo $[a - \epsilon, a + \epsilon]$ con una probabilidad del 68%.

La pregunta ahora es: si tomamos un sólo número de la serie, entonces ese número concreto se separará de dicha media a una σ , con un probabilidad del 68%; pero ¿qué ocurre si en vez de tomar un sólo número como estimación tomamos el promedio de N de ellos?

La respuesta, obtenida del análisis de la sección 6.4 es la siguiente:

Dada una secuencia de números $\{x_i\}$ con media a y dispersión σ , si consideramos la suma de N números de la secuencia dividido por N como estimador de la media, la dispersión de la secuencia de dichas medias vale σ/\sqrt{N} . Por tanto el error para el estimador vale $\epsilon = \sigma/\sqrt{N}$

6.5.1. Funciones de usuario útiles en C

Dado que con frecuencia deberemos calcular la media y la varianza de una secuencia de números, conviene escribir una función que realice estas tareas y que pueda ser utilizada cuando se necesite, en concreto en diferentes programas de este curso.

Para el cálculo, la función debe conocer todos los puntos de la secuencia y el número de ellos. Por tanto debemos pasar un puntero a un vector que contenga los datos, y un entero con el número de datos a procesar. La función debe devolver la media y la varianza. Ver el Ejercicio 6.6.1.

La escritura de funciones de uso general permite reducir enormemente el trabajo y disminuir la generación de errores, permitiendo además el intercambio de código entre programadores.

6.6. Ejercicios

6.6.1. Funciones de usuario para la media y la varianza

Escribir una función en C, con los siguientes argumentos

- Un puntero a un vector que contenga la secuencia de números (Input)
- El número N de puntos (Input)
- La media (Output)
- La varianza (Outuput)

Ayuda: El prototipo podría ser de la forma,

```
void med_var(float *serie, int Numero, float *Media, float *Varianza)
```

Las funciones definidas deben poder ser utilizadas en el resto del curso, de forma que sus prototipos deben ser definidos con atención.

6.6.2. Medias y errores en una distribución plana

Generar N números (**floats**) uniformemente distribuidos en el intervalo $[-2, 5]$. Escribir un programa que calcule la media y la dispersión utilizando las funciones del ejercicio 6.6.1. Repetir el cálculo para varios valores de N y dibujar con **gnuplot** como evoluciona la media y el error en función de N .

6.6.3. Medias y errores en una distribución gaussiana

Generar N números (**floats**) con densidad de probabilidad

$$p(x) \propto e^{-5x^2}.$$

Escribir un programa que calcule la media y la dispersión utilizando las mismas funciones que en el ejercicio anterior. Repetir el cálculo para varios valores de N y dibujar con **gnuplot** como evoluciona la media y el error en función de N .

6.6.4. Errores en Integración numérica

Consideremos la función $f(x) = \sin(x)$ en el intervalo $[0, \pi]$ por defecto. Ahora calculamos el valor de la integral con un número de intervalos N dado, y escribimos el resultado en un fichero. A continuación aumentamos N y volvemos a escribir el resultado, etc. de este modo podemos ver como se diferencia el resultado obtenido del exacto (conocido) al variar N (o lo que es lo mismo, h). Utilizar valores de N de 2 hasta 1000 y dibujar con **gnuplot** el error cometido en función de N .

6.6.5. Simulación de un Reactor de Fusión

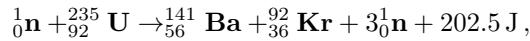
(Este Ejercicio se corresponde con un Trabajo de Examen de la Asignatura)

Un conjunto de N átomos de Urano-235 se encuentran estáticos y distribuidos aleatoriamente y uniformemente en una caja de volumen $L_x \times L_y \times L_z$. Se introduce en dicho volumen un número M de neutrones con posiciones aleatorias y uniformemente distribuidas en la caja, y velocidades con direcciones aleatorias uniformemente distribuidas sobre la esfera unidad y con módulo 1 m/s.

Cuando un neutrón se encuentra a una distancia $d < d_\varepsilon$ de un átomo de Urano-235, el neutrón será absorbido. Nótese que, para un tiempo t , puede ocurrir que dos o más neutrones se

encuentren a distancia $d < d_\varepsilon$ de un átomo dado o que un solo neutrón se encuentre a distancia $d < d_\varepsilon$ de dos o más átomos. En esta situación, escoger qué neutrón es absorbido o qué átomo absorbe el neutrón es irrelevante y el programador podrá escoger hacerlo de la forma que le resulte más conveniente, por ejemplo, realizando la absorción del primer neutrón que encuentre que cumpla la condición de absorción ($d < d_\varepsilon$).

Con una probabilidad del 82 %, el átomo de Urano-235 se transformará en un átomo estable de Urano-236 mientras que, con una probabilidad del 18 % el átomo fisionará siguiendo la reacción



considérese que tanto el átomo de Kriptón como el de Bario se mantienen estáticos en la posición en la que se encontraba el átomo de Urano-235 antes de la fisión y no interaccionan más con los neutrones.

La energía cinética de cada neutrón resultante de la fisión es de 1.6 J y su masa de 2 Kg, la posición inicial de dichos neutrones será la del átomo de Urano en el que se ha producido la fisión y saldrán en direcciones aleatorias uniformemente distribuidas sobre la esfera unidad.

Escribir un programa que simule el sistema anteriormente descrito recibiendo como input el número de átomos N , el número de neutrones iniciales M , las dimensiones de la caja $L_x \times L_y \times L_z$, la distancia de absorción d_ε y el tiempo de simulación t .

El alumno deberá usar el algoritmo de Euler para simular las trayectorias de los neutrones, que se comportarán como partículas libres (esto es, no están sometidas a fuerzas externas). Los neutrones que salgan de la caja simulada serán eliminados de la simulación.

Estudiar, para un tiempo dado t : el número de neutrones M_{final} , el número de átomos fisionados N_{fis} , el número de átomos de Urano-235 restantes N_{235} , el número de átomos de Urano-236 N_{236} y la energía obtenida por las reacciones producidas E .

Puesto que el anterior problema depende de condiciones iniciales aleatorias, el resultado tendrá una componente estocástica, se pide al alumno repetir la anterior simulación un número N_{iter} veces y obtener la media y su desviación estándar de los observables anteriormente mencionados (M_{final} , N_{fis} , N_{235} , N_{236} y E). N_{iter} debe elegirse tal que la desviación estandar sea del orden del 1 % del valor medio.

A continuación, se dan los ordenes de magnitud de los parámetros iniciales para que el alumno haga pruebas previas al examen.

- $N \sim 10^3$
- $M \sim 10^3$
- $L_x \sim L_y \sim L_z \sim 1\text{ m}$
- $d_\varepsilon \sim 0.1\text{ m}$
- $t \sim 1\text{ s}$

Todas las magnitudes se encuentran en el Sistema Internacional de Unidades y no tienen como objetivo representar de forma realista el problema, sino simplificar los cálculos para evitar conversiones de unidades.

6.7. Código en C

6.7.1. Conceptos de Programación: Los argumentos de main

Es común y útil poder pasar argumentos a un programa escribiéndolo detrás del nombre, cuando es ejecutado. Para pasar estos argumentos usamos los argumentos de `main` que son `argc` y `argv`. La primera variable, `argc` toma el valor del número de argumentos de la línea de comandos, contando el propio nombre del programa ejecutable. La segunda variable, `argv`, es en realidad un *puntero doble* a caracteres (`char`), que contiene todas las palabras de la linea de comandos.

Por ejemplo si tenemos un programa ejecutable llamado `prog` y escribimos

```
prog hola 476
entonces argc valdrá 3, argv[0]=""prog", argv[1]=""hola", argv[2]=""476".
```

Es importante remarcar que obtenemos cadenas de caracteres; aunque esta cadena sea de números, NO es un número, y para convertirlo a un número debemos usar la función standar de C `sscanf()`.

6.7.2. Errores en Integración numérica

Ver el Apéndice 6.8 para los detalles teóricos.

```
#include <stdio.h>
#include <math.h>
#define f(x) (sin(x))
FILE *salida;
double integral, delta,punto_x,x_i,x_f;
int i,iter;

main(int argc,char **argv)
{
salida=fopen("error.dat","wt");
x_i=0.0;x_f=2.0*asin(1.0);
switch(argc)
{
case 2:
    sscanf(argv[1],"%d",&iter);
    break;
case 4:
    sscanf(argv[1],"%d",&iter);
    sscanf(argv[2],"%lf",&x_i);
    sscanf(argv[3],"%lf",&x_f);
    break;
default:
    iter=10;
}
for(iter=2;iter<100;iter++)
{
delta=(x_f-x_i)/iter;
for(i=0,integral=0,punto_x=x_i+delta/2;i<iter;i++)
{
integral+=f(punto_x);
punto_x+=delta;
}
integral*=delta;
fprintf(salida,"\n %d %lf ",iter,fabs(2.0-integral));
}
```

```
fclose(salida);
printf("\n resultado = %lf",integral);getchar();
}
```

Visualización de resultados con gnuplot

Podemos ahora dibujar el fichero así obtenido con el `gnuplot`, para ver si efectivamente el error va como h^2 . El código del `gnuplot` es:

```
set logscale x
set logscale y
set xlabel "x"
set ylabel "x*x"
h(x)=a/(x**n)
fit h(x) 'error.dat' via a,n
a=0.927517
n=2.06908
plot 'error.dat' t "Numerico",h(x) t "Error"
```

y la gráfica obtenida puede verse en la Figura 6.3.

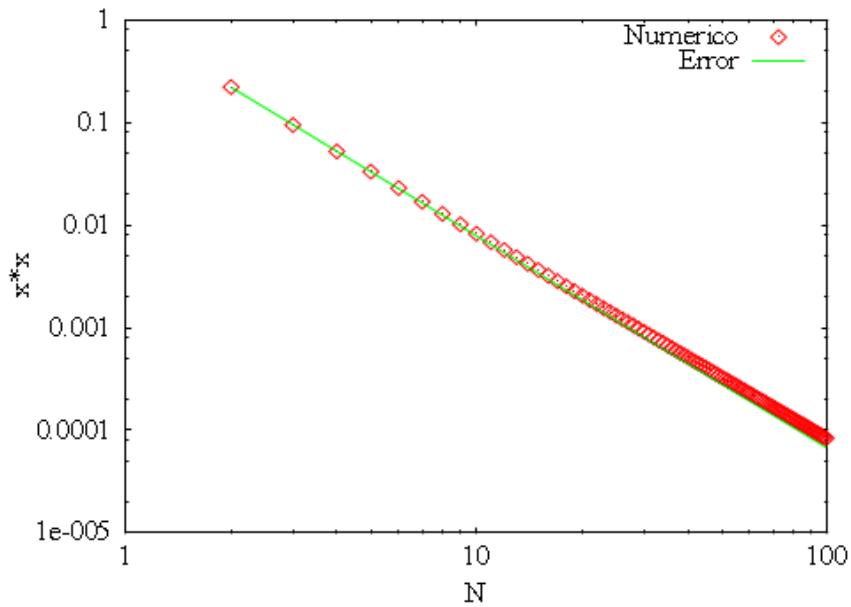


Figura 6.3: Error al calcular la integral de $f(x) = \sin(x)$ en función del número de subintervalos considerados. La gráfica está realizada con `gnuplot` según las instrucciones dadas en el texto.

6.8. Apéndice: Errores en Integración Numérica

Este es un tópico relativamente diferente de lo anterior.

Cuando realizamos cálculos en el ordenador, no es posible en general obtener resultados exactos. La precisión de los números es finita, pues están representados con un número finito de bits. Recordar que un el conjunto de los números reales es isomorfo al de las N-tuplas de infinitos bits (ceros y unos). Por tanto cuando usamos un número finito de bits estamos trabajando con un subconjunto reducido, no podemos representar todos los números y las operaciones entre ellos dan resultados con un cierto error.

Además los propios algoritmos no son exactos: aproximar una derivada por una diferencia finita conlleva un cierto error.

Discutiremos a continuación un ejemplo que ayuda a comprender todo esto.

6.8.1. Cálculo Elemental de Integrales

La resolución numérica de integrales representa una de las ramas más útiles y amplia del cálculo numérico. Esto es debido a los numerosos problemas en los que es necesario calcular integrales, y a que la inmensa mayoría de ellas no pueden ser obtenidas analíticamente. Aquí consideraremos el caso más sencillo. La función a integrar es una función continua en un intervalo cerrado y acotado.

El procedimiento es sencillo y en realidad es una aplicación directa de la definición de Integral de Riemman. Dividimos el intervalo de integración en N subintervalos, y calculamos en cada uno de ellos el valor de la función en el punto medio del intervalo. Podemos calcular entonces el área del rectángulo asociado en cada subintervalo, y obtenemos así una estimación de la integral.

Para obtener un valor fiable, debemos aumentar el numero de intervalos los suficiente para alcanzar un valor estable, dentro de la precisión de la máquina. Este proceso de paso al límite es a veces complicado y es necesario tener una estimación del error cometido. discutiremos esto más adelante.

En la figura 6.4 puede verse como funciona el algoritmo mencionado para estimar el valor de la integral para una función $f(x) = x^2$, en el intervalo $[0, 2]$.

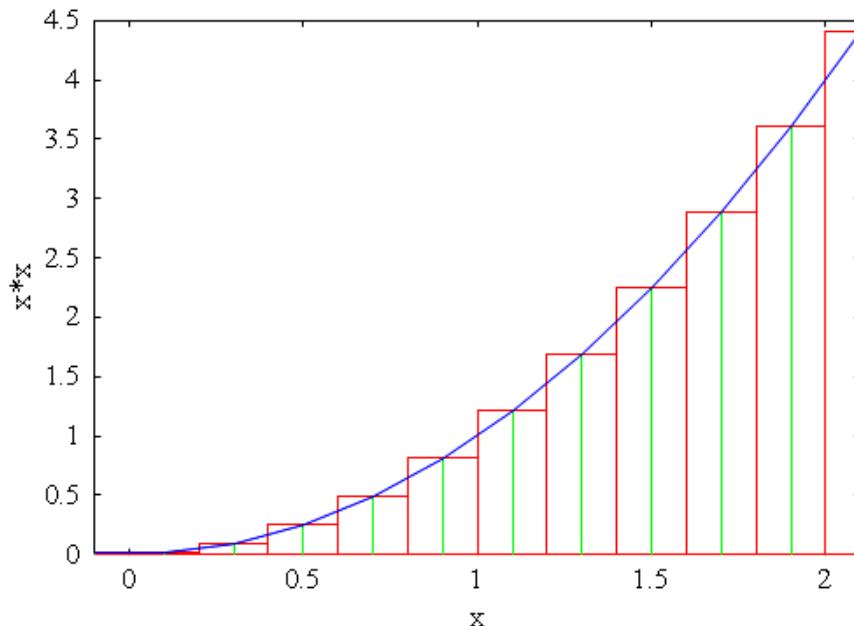


Figura 6.4: Grafica generada con gnuplot donde pueden verse los intervalos y los valores numéricos usados para calcular la aproximación a la Integral exacta.

La figura 6.4 puede generarse con gnuplot con las siguientes instrucciones:

```
set samples 12
f(x)=x*x
set xlabel "x"
set ylabel "x*x"
plot [-0.1:2.1] f(x) t "" with boxes,f(x) t "" with impulses, f(x) t ""
```

6.8.2. Ejemplo de Cálculo de Integrales

A continuación puede verse un listado de un programa en C para calcular la integral de esta función. Por defecto calcula en el intervalo [0,2] con 10 subintervalos, si bien estos valores pueden cambiarse en la línea de comandos. Para ejecutar el programa debe escribirse

nombre [iter][a b]

por ejemplo

nombre 10 3 4

ejecutará el programa con 10 divisiones e integrará la función con los valores de x entre 3 y 4.

Los corchetes es una notación habitual para indicar que los argumentos son opcionales. Si se omite alguno de ellos, el programa elegirá unos valores por defecto, indicados en el código.

Es importante prestar atención a como se introducen los datos en la linea de comandos.

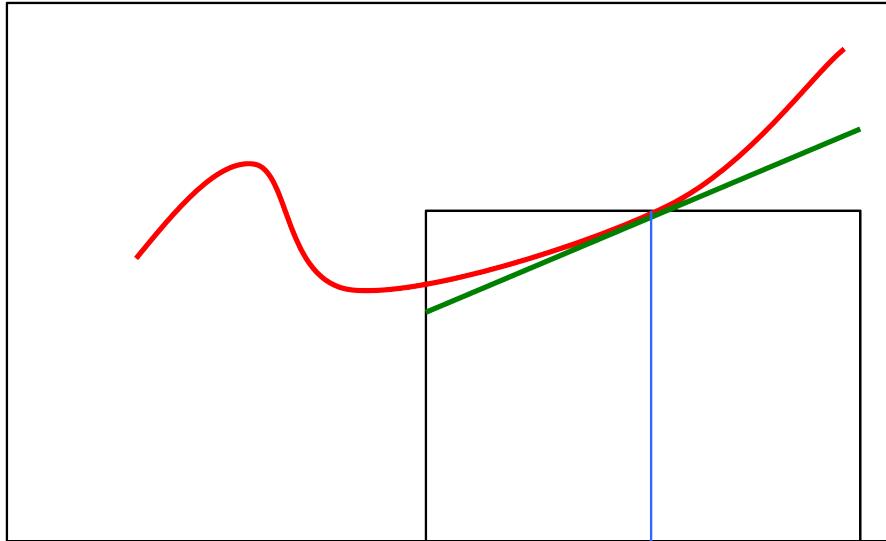
```
main(int argc,char **argv)
{
    salida=fopen("error.dat","wt");
    x_i=0.0;x_f=2.0*asin(1.0);
    switch(argc)
    {
        case 2:
            sscanf(argv[1],"%d",&iter);
            break;
        case 4:
            sscanf(argv[1],"%d",&iter);
            sscanf(argv[2],"%lf",&x_i);
            sscanf(argv[3],"%lf",&x_f);
            break;
        default:
            iter=10;
    }
}
```

6.8.3. Estimación de Errores

El problema se plantea del siguiente modo: dados N subintervalos, cada uno de longitud h , el valor numérico que obtengamos será una aproximación al valor exacto de la integral. Lo que podemos hacer es ir aumentando el valor de N (disminuir h) y ver si el resultado se va estabilizando. Para una función bien comportada, como la del ejemplo anterior, es posible saber como evoluciona el error al variar h .

Para ello realizamos un desarrollo de Taylor de la función en torno al valor central de cada subintervalo, y dado que nos estamos quedando con el valor de la función en el centro, sabemos que el error proviene de lo que se separa la función de ese valor constante usado. Pero esa diferencia se puede estimar con dicho desarrollo de Taylor. En realidad lo que se puede estimar es como depende esa diferencia de h .

Para concretar hagamos el desarrollo de una función genérica y estudiemos como depende el error de h . En cada subintervalo estamos sustituyendo el valor de la función por una recta con pendiente la derivada en el punto central, tal como puede verse en la figura 6.5.



$$a+ih \quad a+(i+1/2)h \quad a+(i+1)h$$

Figura 6.5: Estimación del error cometido en cada subintervalo al estimar el valor de la Integral.

En esta gráfica, la línea horizontal corresponde al valor constante (primer término del desarrollo de Taylor), la línea recta inclinada a los dos primeros términos del desarrollo de Taylor, y la curva al valor de $f(x)$.

El desarrollo de Taylor pues tomará la forma siguiente para cada subintervalo:

$$\begin{aligned}
 f(x) &= f^0(x_0) + f^1(x_0)(x - x_0) + \frac{1}{2!}f^2(x_0)(x - x_0)^2 + \dots \\
 \int_{-h/2}^{h/2} f(x)dx &= \int_{-h/2}^{h/2} (f^0(0) + f^1(0)h + \frac{1}{2!}f^2(0)h^2 + \dots)dx = \\
 &\quad f^0(0)h + 0 + \frac{1}{6}f^3(0)h^3 + \dots
 \end{aligned} \tag{6.54}$$

Ahora sumamos todos los subintervalos, recordando que $h = \frac{b-a}{N}$,

$$\begin{aligned} & \int_a^b f(x)dx = \\ & \sum_{i=0}^{N-1} \int_{a+ih}^{a+(i+1)h} (f^0(a + (i + 1/2)h) + f^1(a + (i + 1/2)h)h + \frac{1}{2!}f^2(a + (i + 1/2)h)h^2 + \dots)dx = \\ & (b-a) \sum_{i=0}^{N-1} (f^0(a + (i + 1/2)h) + \frac{h^3}{6} \sum_{i=0}^{N-1} f^2(a + (i + 1/2)h)) + \dots \approx \\ & (b-a) \sum_{i=0}^{N-1} f^0(a + (i + 1/2)h) + \frac{h^3(b-a)}{6h} \langle f^2(a + (i + 1/2)h) \rangle = \\ & (b-a) \sum_{i=0}^{N-1} f^0(a + (i + 1/2)h) + \theta(h^2) \end{aligned} \tag{6.55}$$

Vemos pues que dentro de cada subintervalo el error cometido disminuye como h^3 , es decir si disminuimos h a la mitad, el error disminuye un factor 8. El valor final de la integral lo obtenemos sumando los valores de todos los subintervalos, con lo cual, el error final va como h^2 .

6.9. Apéndice: Cálculo de integrales gaussianas

Queremos calcular integrales del tipo (con $p \in \mathbb{N}$),

$$\int_{-\infty}^{\infty} e^{-\alpha x^2} x^p dx, p \in \mathbb{N} \tag{6.56}$$

Si p es impar, la integral es cero, pues el integrando es *impar* en este caso: al cambiar x por $-x$, su valor cambia de signo, y como el intervalo de integración es simétrico, el resultado de la integral en x positivo cancela el resultado con x negativo.

Consideremos ahora el caso con $p = 0$,

$$\begin{aligned} I(\alpha) &= \int_{-\infty}^{\infty} e^{-\alpha x^2} dx \\ I^2(\alpha) &= \int_{-\infty}^{\infty} dx \int_{-\infty}^{\infty} dy e^{-\alpha(x^2+y^2)} = \int_0^{\infty} r dr \int_0^{2\pi} d\theta e^{-\alpha r^2} = 2\pi \int_0^{\infty} \frac{1}{2} dt e^{-\alpha t} = \frac{\pi}{\alpha} \\ I(\alpha) &= \sqrt{\frac{\pi}{\alpha}} \end{aligned} \tag{6.57}$$

El resto de potencias pares podemos calcularlas fácilmente derivando $I(\alpha)$ con respecto a α las veces necesarias, por ejemplo

$$\begin{aligned} \int_{-\infty}^{\infty} e^{-\alpha x^2} x^2 dx &= \frac{-d}{\alpha} I(\alpha) = \frac{1}{2} \sqrt{\pi} \alpha^{-3/2} \\ \int_{-\infty}^{\infty} e^{-\alpha x^2} x^4 dx &= \frac{d^2}{\alpha^2} I(\alpha) = \frac{1}{2} \frac{3}{2} \sqrt{\pi} \alpha^{-5/2} \end{aligned} \tag{6.58}$$

Capítulo 7

Análisis estadístico avanzado

7.1. El Teorema del Límite Central

Existen varios enunciados equivalentes de este importante Teorema. Lo enunciaremos en la siguiente forma,

Sea una densidad de probabilidad arbitraria con media β y varianza finita σ^2 . El conjunto de números formado por las medias de N medidas, para N grande, tiende a una distribución gaussiana con media β y dispersión $\frac{\sigma}{\sqrt{N}}$.

7.1.1. Distribución de las Medias

Conviene resaltar dos aspectos del Teorema del Límite Central (TLC),

- Sean una o varias sucesiones $\{x_i^0\}$, $\{x_i^1\}$, $\dots \{x_i^{N-1}\}$; construimos una nueva sucesión ξ con la media de N números, uno de cada sucesión, es decir

$$\xi_i = \frac{1}{N} \sum_{k=0}^{N-1} x_i^k \quad (7.1)$$

Si todas las densidades individuales tienen varianza finita, entonces la nueva variable aleatoria ξ tendrá una distribución gaussiana, aunque cada una de las originales no la tenga. Las originales pueden ser distribuciones planas, polinómicas, etc. La construida con la media, tendrá distribución gaussiana. Este punto requiere una demostración complicada. Aquí pondremos varios ejercicios para comprobar numéricamente que esto es así efectivamente, y visualizaremos gráficamente el resultado.

- Supongamos que todas las sucesiones $\{x_i^k\}$ tienen la misma dispersión σ . Consideremos de nuevo la sucesión ξ construida en 7.1. La anchura de la distribución para ξ se hace cada vez más pequeña. En concreto, si llamamos γ a la dispersión de ξ , tendremos, $\gamma = \sigma/\sqrt{N}$. Este resultado sencillo es de suma importancia en muchos aspectos de la Estadística y la Física. La demostración es la correspondiente a la sección 6.3.

Conviene resaltar un caso particular del segundo ítem anterior, de uso frecuente. Supongamos que tenemos una secuencia de números con densidad plana, media cero y dispersión σ . Si consideramos la media de N como nueva variable, ésta tendrá media 0 y dispersión σ/\sqrt{N} . Dicho de otro modo: si sumamos N números de la secuencia original, y hacemos la media, el número obtenido se acercará a 0 (que es su media) como $\frac{1}{\sqrt{N}}$.

7.2. Distribuciones no gaussianas o con autocorrelaciones

Supongamos que tenemos la secuencia x_i de P números con dispersion finita σ . Construimos ahora una nueva variable aleatoria ξ formada como la media de N elementos consecutivos de x_i . Tendremos ahora P/N variables ξ . ¿Cuánto vale la dispersión de ξ ?

Más concretamente: definimos

$$\xi_k = \frac{1}{N} \sum_{i=(k-1)N}^{kN-1} x_i \quad (7.2)$$

es decir, ξ_k es la media de un bloque de N datos consecutivos de la serie $\{x_i\}$. Remarcamos que ahora trabajamos con una sola serie; esto es diferente del caso de la expresión 7.1.

Supuesta conocida la dispersión de x , digamos σ , nos preguntamos cuánto vale la dispersión de ξ , que llamaremos γ .

La primera cuestión importante se refiere a la presencia o no de autocorrelaciones en la variable original $\{x_i\}$. Si bien en los datos experimentales la condición 6.40 se cumple habitualmente, en las simulaciones en ordenador (o experimentos numéricos), debido a las limitaciones de los algoritmos, a los números aleatorios o a otros factores, no se cumple en general.

Una serie de datos está correlacionada si es posible inferir en parte al menos, los siguientes números de los anteriores.

Por ejemplo un encuestador *tramposo* al que encarguen 100 encuestas puede hacer 20 y luego cada una ellas copiarla 5 veces. En realidad tendríamos 20 datos, y si usamos 100, obtendremos la media correcta, pero el error real será mucho más alto que el estimado.

Consideremos una sucesión de números de acuerdo a una cierta distribución $p(x)$, descorrelacionados. Esto significa que el sistema no tiene memoria, no es posible inferir nada sobre el siguiente número aunque conozcamos toda la serie anterior.

Definimos entonces la *función de autocorrelación* de la serie x como

$$C(d) = \frac{1}{P-d} \sum_{i=0}^{P-1-d} x_i x_{i+d} - \left(\frac{1}{P} \sum_{i=0}^{P-1} x_i \right)^2 \equiv \langle x_i x_{i+d} \rangle - \langle x \rangle^2 \quad (7.3)$$

Notar que $C(0)$ es precisamente la varianza. En el caso de que la secuencia sea aleatoria, no existirán correlaciones entre datos a distancia mayor de 0, es decir

$$C(d) = 0 \quad \forall d > 0 \quad (7.4)$$

Si existe correlación entre los datos, su media no cumple las condiciones del TLC y si estimamos el error usando

$$\epsilon(\bar{x}) = \frac{\sigma(x)}{\sqrt{P}}$$

lo estaremos subestimando, pues deberíamos usar sólo los datos independientes (no correlacionados) que serán menos que P .

Por otra parte, muchas veces tenemos que trabajar con datos que no corresponden a distribuciones gaussianas. En este caso la estimación del error que hemos realizado en función de la dispersión no es correcta, pues para una distribución arbitraria, no se cumple que los datos están a una σ con una probabilidad del 68 %.

7.3. Análisis de errores con bloques

Cuando tenemos datos con correlaciones o una distribución no gaussiana, debemos establecer una forma para estimar el error que cometemos al estimar la media con una serie de P datos.

Los dos problemas pueden ser evitados aplicando el Teorema del Límite Central. Si nos dan una secuencia de números y queremos calcular su error, lo que debemos hacer es *agrupar* los

datos en bloques de N números y construir una sucesión nueva cuyos elementos son la media de esos N datos de acuerdo a 7.2.

Esto nos permite solucionar ambos problemas simultáneamente. En efecto:

- Sabemos que para N grande la distribución que originalmente puede no ser gaussiana, se convierte en gaussiana. Y cuando lo sea, ya se cumplirá que el 68 % de los datos estarán a una distancia de una σ y la definición de error será correcta.
- Además si los datos están correlacionados, al hacer bloques, a partir de un cierto tamaño ya no lo estarán. En el ejemplo del encuestador trámposo, cuando hagamos bloques con un número de datos originales superior a 5, las medias por bloque se irán convirtiendo en independientes.

Es decir, cuando el tamaño de nuestro bloque sea pequeño, la distribución puede ser no gaussiana y los datos estarán correlacionados. Por tanto estimando el error de acuerdo a la formula standard usando todos los datos individuales,

$$\epsilon = \frac{1}{\sqrt{P}}\sigma \quad (7.5)$$

obtendremos resultados erróneos, pues los P datos no son todos independientes, subestimando el error. Haciendo crecer el tamaño del bloque N , a partir de un tamaño dado, las medidas serán independientes y además la distribución de las medias por bloques será gaussiana.

El error a partir se ese momento se mantendrá constante al aumentar el tamaño del bloque, lo que será una indicación de que todo funciona correctamente.

Resumiendo pues, cuando se cumplen las dos condiciones, de que las medidas no tienen autocorrelaciones y que la distribución de las variables de bloque es gaussiana, el error es independiente del tamaño del bloque.

La demostración de la independencia del error es sumamente sencilla.

Sean P datos sin autocorrelaciones y con distribución gaussiana. El error es en este caso $\epsilon_x = \sigma_x/\sqrt{P}$.

Supongamos que agrupamos los P datos en bloques de N elementos en la variable ξ de acuerdo a 7.2. Si ahora consideramos la dispersión de la nueva variable de bloque ξ , de acuerdo con el TLC tendremos que $\sigma_\xi = \sigma_x/\sqrt{N}$. Tenemos P/N bloques, o variables ξ , por lo cual el error en esta variable valdrá, $\epsilon_\xi = \sigma_\xi/\sqrt{(P/N)}$, y comparando ahora los errores para x y para ξ ,

$$\epsilon_\xi = \frac{1}{\sqrt{P/N}}\sigma_\xi = \frac{1}{\sqrt{P/N}}\frac{\sigma_x}{\sqrt{N}} = \frac{\sigma_x}{\sqrt{P}} \quad (7.6)$$

obteniendo pues el mismo valor independientemente del tamaño del bloque.

Repasemos ahora el proceso a seguir para dar una estimación correcta de los errores.

- Sea una secuencia de P números $X = \{x_0, x_1, \dots, x_{P-1}\}$
- Estimamos su media previamente a la división en bloques; sea su valor \bar{x} .
- Ahora hacemos bloques de N medidas. Si P no es divisible por N , podemos eliminar unos pocos datos de X . Para cada bloque calculamos la media y_i (i índice de bloque), obteniendo la nueva sucesión: $Y = \{y_0, y_1, \dots, y_{(P/N)-1}\}$
- Calculamos entonces $\sigma_Y = \langle y^2 \rangle - \langle y \rangle^2 \Rightarrow \epsilon = \frac{1}{\sqrt{P/N}}\sigma_Y$

Insistimos en que el error estimado, ϵ tomará un valor constante (independiente de N) cuando el tamaño del bloque sea suficientemente grande.

La correlación entre datos es común en los experimentos numéricos, pero rara vez aparece en los datos obtenidos de medidas experimentales, por lo que en general no es necesario este proceso para datos extraídos en el laboratorio.

7.4. Errores para operadores compuestos

Supongamos que queremos obtener el error de cantidades algo más complejas que las medias: funciones de las medias. Algunos ejemplos serían

$$\begin{aligned} A &= \langle x \rangle^2 \\ B &= \frac{\langle x \rangle}{\langle y \rangle} \\ C &= \langle x \rangle - \langle y \rangle \\ D &= \langle f(x) \rangle \end{aligned} \tag{7.7}$$

Existen varias formas para calcular el error de las cantidades compuestas. Discutimos a continuación las dos más interesantes para nosotros.

7.4.1. Propagación de errores

Supondremos conocido el error (la dispersión) en cada uno de los valores medios, y buscaremos la forma de estimar el error en función de esos valores medios. Consideremos el caso A en primer lugar. Dada una serie de datos $\{x_i\}$, $i = 0, N - 1$ podemos calcular el promedio de los N números como un estimador de la media exacta. Como siempre, a este estimador lo llamaremos \bar{x} para distinguirlo de la media exacta $\langle x \rangle$. El promedio \bar{x} tendrá una desviación respecto de la media exacta, de modo que si realizamos la estimación con secuencias diferentes de N números, tendríamos valores diferentes de \bar{x} . Por tanto podemos escribir que

$$\bar{x} = \langle x \rangle + \eta \equiv a + \eta \tag{7.8}$$

donde para simplificar usamos $\langle x \rangle = a$, constante. La variable aleatoria η cumple

$$\langle \eta \rangle = 0, \quad \langle \eta^2 \rangle = \sigma_{\bar{x}}^2 \tag{7.9}$$

Estimemos ahora la media de $\langle x \rangle^2$. Consideremos la cantidad $\bar{x}^2 = (a + \eta)^2$,¹

$$\bar{x}^2 = a^2 + \eta^2 + 2a\eta \tag{7.10}$$

si suponemos el error pequeño, esto implica que $|\eta| \ll a$ y podemos despreciar el término η^2 ; el último término es cero y recuperamos que $\bar{x}^2 = a^2$. Calculemos ahora el error en \bar{x}^2 , más concretamente la dispersión en esa cantidad,

$$\begin{aligned} \bar{x} &= a + \eta \\ \bar{x}^2 - a^2 &= \eta^2 + 2a\eta \\ (\bar{x}^2 - a^2)^2 &= \eta^4 + 4a^2\eta^2 + 2a\eta^3 \end{aligned} \tag{7.11}$$

el término dominante es el que contiene η^2 , y tomando valores medios, tendremos

$$(\bar{x}^2 - a^2)^2 = 4a^2\sigma_{\bar{x}}^2 \tag{7.12}$$

Por tanto concluimos que

$$\sigma(\bar{x}^2) = 2\bar{x}\sigma(\bar{x}) \tag{7.13}$$

Este mismo razonamiento puede aplicarse al resto de los casos; consideremos el caso B .

$$\begin{aligned} \bar{x} &= a + \eta \\ \bar{y} &= b + \xi \\ \frac{\bar{x}}{\bar{y}} &= \frac{a + \eta}{b + \xi} \approx \frac{a}{b} \frac{1 + \frac{\eta}{a}}{1 + \frac{\xi}{b}} \approx \frac{a}{b} \left(1 + \frac{\eta}{a}\right) \left(1 - \frac{\xi}{b}\right) \approx \frac{a}{b} \left(1 + \frac{\eta}{a} - \frac{\xi}{b}\right) \end{aligned} \tag{7.14}$$

¹ Nótese que \bar{x}^2 es un estimador sesgado de $\langle x \rangle^2$, pues $\langle \bar{x}^2 \rangle = \langle x \rangle^2 + \sigma_{\bar{x}}^2$

donde hemos despreciado todas las potencias de orden mayor que 1 en los errores.

Calculamos ahora la dispersion de $\frac{\bar{x}}{\bar{y}}$,

$$\left(\frac{a+\eta}{b+\xi}\right)^2 - \left(\frac{a}{b}\right)^2 \approx \frac{a^2}{b^2} \left(\frac{\eta^2}{a^2} + \frac{\xi^2}{b^2} - \frac{2\eta\xi}{ab} \right) \quad (7.15)$$

es decir

$$\sigma^2 \left(\frac{\bar{x}}{\bar{y}} \right) = \frac{a^2}{b^2} \left(\frac{\sigma_x^2}{a^2} + \frac{\sigma_y^2}{b^2} - 2 \frac{\sigma_{\bar{x},\bar{y}}}{ab} \right) \quad (7.16)$$

Donde $\sigma_{\bar{x},\bar{y}}$ es la covarianza entre \bar{x} e \bar{y} y que puede estimarse con

$$\sigma_{\bar{x},\bar{y}} = \frac{N}{N-1} \left(\frac{1}{N} \sum_{i=0}^{N-1} x_i y_i - \left(\frac{1}{N} \sum_{i=0}^{N-1} x_i \right) \left(\frac{1}{N} \sum_{i=0}^{N-1} y_i \right) \right) \quad (7.17)$$

En el caso de que las variables x e y no tengan correlación, este término es estrictamente cero.

Para el caso C , tendremos,

$$\begin{aligned} \bar{x} &= a + \eta \\ \bar{y} &= b + \xi \\ \bar{x} - \bar{y} &= a - b + \eta - \xi \\ (a - b + \eta - \xi)^2 - (a - b)^2 &= \eta^2 + 2a\eta + \xi^2 + 2b\xi - 2a\xi - 2b\eta - 2\eta\xi \end{aligned} \quad (7.18)$$

tomando valores medios, y recordando que $\langle \eta \rangle = \langle \xi \rangle = 0$,

$$\sigma^2(\bar{x} - \bar{y}) = \sigma_x^2 + \sigma_y^2 - 2\sigma_{\bar{x},\bar{y}} \quad (7.19)$$

El el caso de una función arbitraria $f(x)$ (caso D), el procedimiento es similar: desarrollamos la función en serie de Taylor, y nos quedamos con los términos más significativos.

Una ventaja de este procedimiento, es que conociendo la media y el error de una cantidad, es posible (si el operador es simple y no hay correlaciones) calcular el error del operador compuesto sin volver a realizar ningún análisis, o sin disponer de todos los datos de la secuencia. Sin embargo este procedimiento es complejo incluso para operadores relativamente simples y se complica aún más cuando existen correlaciones. Incluso cuando los errores no son muy pequeños, parte del análisis anterior puede perder su validez. Por todo ello, salvo en casos muy puntuales, lo habitual es usar otros métodos numéricos más sencillos y precisos, que pueden ser usados siempre que dispongamos de los datos de la sucesión para realizar los cálculos.

7.4.2. Análisis por bloques

El método de propagación de errores permite una estimación simple de los errores en cantidades compuestas. Cuando las medidas tienen distribución gaussiana, son estadísticamente independientes, los errores pequeños y las cantidades no tienen correlación, da resultados correctos. Sin embargo cuando los datos tienen correlaciones, aunque tengamos una buena estimación de los errores individuales, el error final puede estar mal calculado. También existen ciertas cantidades tales que si bien cada una de sus partes tiene un gran error, al construir cierta función de las misma, la cantidad compuesta tiene una fluctuación mucho menor. Por ejemplo supongamos que queremos calcular el error en

$$\alpha = \frac{\langle x^4 \rangle}{\langle x^2 \rangle^2} \quad (7.20)$$

Si calculamos el error con propagación de errores, consideramos las variaciones en torno a la media del numerador completamente independientes de las del denominador. Sin embargo es evidente que cuando se produzcan fluctuaciones de x por encima de la media, tanto el numerador como el

denominador se situarán por encima, de modo que el cociente variará poco, y por tanto el error correcto en el cociente sera mucho menor que el estimado suponiendo variaciones independientes.

Una forma simple de calcular los errores correctamente evitando todos los problemas inherentes al método de la propagación de errores está inspirado en el siguiente hecho.

Dada una cantidad compleja, realizamos K experimentos diferentes, medimos en cada uno de ellos las cantidades pertinentes, y calcularemos el error como la dispersión obtenida a partir de esas K medidas independientes.

Por ejemplo, supongamos que queremos calcular el error en el operador definido en 7.20. Entonces una persona mide el valor medio de numerador, el del denominador, y nos envia el valor obtenido para el cociente. Tendremos así un dato ξ_0 . Otra persona, con datos diferentes, mide lo mismo y nos envia ξ_1 , y así sucesivamente. Finalmente disponemos de la serie de datos ξ_i . Con dicha serie, podemos extraer el error en los datos. Remarcamos que la media obtenida con los datos ξ_i no coincidirá con la media correcta, que debe ser calculada con todos los datos para calcular el valor medio del numerador, todos los datos para calcular el valor del denominador, y luego el cociente. Los resultados son diferentes, pues evidentemente la media del cociente no es el cociente de las medias. De modo que este procedimiento es válido para estimar el error pero no la media, que debe ser calculada disponiendo, en este ejemplo, de $\langle x^4 \rangle$ y $\langle x^2 \rangle^2$ calculados con todos los datos individuales.

En vez de que K individuos hagan experimentos diferentes, nosotros agruparemos nuestros P datos en bloques de N medidas, y cada uno de estos bloques calcularemos la media de las cantidades complejas. Tendremos finalmente pues $K = P/N$ datos, con los cuales, calculando su dispersión podremos estimar el error en la cantidad dada. Insistimos en que esto es válido para calcular el error, no la estimación de la media.

Es decir el proceso es similar al indicado en 6.5, pero ahora construyendo en cada bloque el operador necesario. Por ejemplo para calcular el error de

$$\alpha = \langle x^2 \rangle - \langle x \rangle^2 \quad (7.21)$$

calcularemos en cada bloque

$$\xi_i = \frac{1}{N} \sum_{k=iN}^{iN+N-1} x_k, \quad \rho_i = \frac{1}{N} \sum_{k=iN}^{iN+N-1} x_k^2, \quad \alpha_i = \rho_i - \xi_i^2 \quad (7.22)$$

Disponemos ahora de la serie de K datos α_i , de los cuales podremos calcular su dispersión, y de ahí el error en 7.21. Remarcamos de nuevo que la media de las α_i no coincide con la cantidad α calculada con todos los datos. De modo que el cálculo con los bloques sólo sirve para estimar el error, no la media, que debe ser estimada usando todos los datos de una sola vez.

Veáse el Ejercicio 7.5.3 para una mejor comprensión de lo explicado en este capítulo. Repasar además el código correspondiente para ver como estructurar y organizar el código.

7.5. Ejercicios

7.5.1. Media de distribuciones

Generar una secuencia x (`floats`) uniformemente distribuidos en el intervalo $[0, 1]$. Considerar la nueva variable aleatoria ξ obtenida tras hacer la media agrupando datos de N en N ; considerar varios valores de N . Dibujar el histograma (debidamente normalizado) de la distribución de ξ para los diferentes valores de N y comprobar que la distribución de las medias tiende a una gaussiana, a pesar de ser uniforme la distribución de la variable original x .

7.5.2. Teorema del Límite Central

Usando las sucesiones del Ejercicio anterior, comprobar que efectivamente la media se comporta como dice el Teorema del Límite Central. Para ello (recordando 6.15) calcular analíticamente la dispersión σ de los datos originales, y la dispersión γ de la medias de N datos (ξ). Calcular la expresión en base al TLC para la distribución de ξ para cada N y dibujarla sobre los diferentes histogramas para ver que coinciden al crecer N .

7.5.3. Análisis de Errores por Bloques

Escribir un programa que genere diferentes secuencias de números $\{x_n\}$, algunas de ellas con correlaciones; realizar un análisis por bloques de esos datos. Calcular, las siguientes cantidades con su dispersion y error:

$$\langle x \rangle, \langle x^2 \rangle, \langle x^2 \rangle - \langle x \rangle^2$$

Calcular y dibujar la dispersión y los errores para diferentes tamaños de bloques, y comprobar que se comportan de acuerdo a lo explicado en este capítulo.

7.6. Problemas

7.6.1. Problema 1

Lanzamientos de moneda: Una persona lanza repetidamente una moneda y apunta cuantas veces le sale cara o cruz. Supongamos que tenemos muchas personas haciendo lo mismo. Lo que le pasa a cada una de ellas es impredecible, pero sí podemos decir algo sobre lo que sucede en media.

Si consideramos 10 personas y en cada lanzamiento calculamos la media (asignando +1 a la cara y -1 a la cruz por ejemplo), aunque la distribución de cara y cruz no es gaussiana, la distribución de la media si debería serlo. Calculad para este caso el valor medio y la distribución de probabilidad. Estudiar qué ocurre cuando agrupemos los lanzamientos en números más grandes.

7.6.2. Problema 2

El jugador de azar: El dinero total perdido o ganado tras N partidas (sin trampas ni sesgos) va en media como \sqrt{N} , pero la media por partida va a cero como $1/\sqrt{N}$. Comprobarlo con un programa, donde se simula una partida en la que se gana o pierde al azar.

7.6.3. (Ampliación) Problema 3

Sea una distribución de probabilidad uniforme $p(x), x \in [-1, 1]$. Consideremos la sucesión formada por

$$y_n = x_n + \operatorname{sen}\left(\frac{2\pi}{L}n\right) \quad (7.23)$$

Considerar diferentes valores de L enteros. Calcular numéricamente y analíticamente la autocorrelación $C(d)$ de la serie $\{y_n\}$ y dibujar comparar los resultados. Discutir lo obtenido y relacionar la correlación con el valor de L .

7.6.4. Problema 4

En el ejercicio anterior generar un número elevado de datos (del orden de millones), y realizar un cálculo del error usando bloques. Ver como evoluciona la estimación del error con el tamaño del bloque y relacionar esto con L .

7.7. Código en C

7.7.1. Análisis de Errores usando Bloques

Este código contiene un programa completo que genera una simple secuencia random, con o sin autocorrelaciones (según un `#define`), y realiza un análisis de errores usando bloques, escribiendo los resultados en función del tamaño del bloque para visualizarlo con `gnuplot`. Es importante fijarse en el uso de funciones para simplificar el problema, que puede volverse complejo caso de no estructurarlo correctamente.

```
/*
    Calculo de errores con bloques
    Se usa el generador de Parisi-Rapuano
*/

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

#define NormRANu (2.3283063671E-10F)

#define Tr_Max 100000
#define Size_B1_Max (Tr_Max/100)

//Así se genera una secuencia con autocorrelaciones
//#define Correlacion
//Secuencia distribuida uniformemente sin autocorrelaciones
#define Plana

unsigned int irr[256];
unsigned int ir1;
unsigned char ind_ran,ig1,ig2,ig3;

typedef double precision;

precision d[Tr_Max],d_new[Tr_Max],dsigma[Tr_Max];
precision d_blo[Tr_Max],d_blo_new[Tr_Max],d_final[Tr_Max];
void analisis(precision *x,int n,precision *mean,precision *sigma,
              precision *err);
void Construye_Blocks(precision *x,precision *xb,int n_b,int b_s);

FILE *fout;

precision Random(void);
void ini_ran(int SEMILLA);

main()
{
    int i,j,n_b;
    int block_size,n_blocks,n_blocks_min;
    precision A,B,a;
    precision med,med2,sig,error;
    int N;
```

```

A=-1; //Genero N numeros planos en [A,B]
B=5;
N=100000;

ini_ran(123456789); //Inicializo el generador random

//Genero la secuencia de numeros random planos

//Como ejemplo consideramos que repito de 100 en 100 ...

#ifndef Correlacion
//asi,mientras el size del bloque sea<100, no sera asintotico
for(i=0;i<N/100;i++)
{
    a=A+(B-A)*Random();
    for(j=0;j<100;j++)
        d[i*100+j]=a;
}
#endif
#ifndef Plana
/* Calculo sin autocorrelaciones

for(i=0;i<N;i++)
    d[i]=A+(B-A)*Random();

*/
#endif
//Calculo la media, dispersion y error naive.

analisis(d,N,&med,&sig,&error);
printf("Origen:#datos=%d, media=%lf, sigma=%lf, error=%lf\n",
      N,med,sig,error);

//Ahora hago un analisis por bloques

fout=fopen("analisis.dat","wt");

for(block_size=2;block_size< Size_Bl_Max; block_size++)
{
    n_blocks =(int)(N/block_size);

    Construye_Blocks(d,d_blo,n_blocks,block_size);

    analisis(d_blo,n_blocks,&med,&sig,&error);
    fprintf(fout,"%d %d %lf %lf %lf\n",n_blocks,block_size,med,sig,error);
}
fclose(fout);

//Ahora analizamos una cantidad derivada, por ejemplo
//la secuencia de cuadrados

```

```

//primero construimos la cantidad

for(i=0;i<N;i++)
    d_new[i]=d[i]*d[i];

//Ahora calculamos su media

analisis(d_new,N,&med,&sig,&error);
printf("Cuadrados:#datos=%d, media=%lf, sigma=%lf, error=%lf\n",
       N,med,sig,error);

//Ahora calculamos su error por bloques

fout=fopen("analisis2.dat","wt");

for(block_size=2;block_size< Size_Bl_Max; block_size++)
{
    n_blocks =(int)(N/block_size);

    Construye_Blocks(d_new,d_blo,n_blocks,block_size);

    analisis(d_blo,n_blocks,&med,&sig,&error);
    fprintf(fout,"%d %d %lf %lf %lf\n",n_blocks,block_size,med,sig,error);
}
fclose(fout);

//Ahora repetimos con una cantidad compuesta, por ejemplo <x*x>-<x>*<x>
//Aprovechamos parte de lo anterior para calcular este valor

analisis(d,N,&med,&sig,&error);
analisis(d_new,N,&med2,&sig,&error);

printf("<x*x>-<x>*<x>: %lf",med2-med*med);

//Calculamos los errores para la cantidad anterior usando bloques
fout=fopen("analisis3.dat","wt");

for(block_size=2;block_size< Size_Bl_Max; block_size++)
{
    n_blocks =(int)(N/block_size);

    Construye_Blocks(d,d_blo,n_blocks,block_size);
    Construye_Blocks(d_new,d_blo_new,n_blocks,block_size);
    //Construyo la nueva cantidad para cada bloque
    for(n_b=0;n_b<n_blocks;n_b++)
        d_final[n_b]=d_blo_new[n_b]-d_blo[n_b]*d_blo[n_b];

    analisis(d_final,n_blocks,&med,&sig,&error);
    fprintf(fout,"%d %d %lf %lf %lf\n",n_blocks,block_size,med,sig,error);
}
fclose(fout);

}

```

```

precision Random(void)
{
    precision r;

    ig1=ind_ran-24;
    ig2=ind_ran-55;
    ig3=ind_ran-61;
    irr[ind_ran]=irr[ig1]+irr[ig2];
    ir1=(irr[ind_ran]^irr[ig3]);
    ind_ran++;
    r=ir1*NormRANu;
    //printf("r=%f\n",r);
    return r;
}

void ini_ran(int SEMILLA)
{
    intINI,FACTOR,SUM,i;

    srand(SEMILLA);

   INI=SEMILLA;
    FACTOR=67397;
    SUM=7364893;

    for(i=0;i<256;i++)
    {
        INI=(INI*FACTOR+SUM);
        irr[i]=INI;
    }
    ind_ran=ig1=ig2=ig3=0;
}

void analisis(precision *x,int n,precision *mean, precision *sigma,
              precision *err)
{
    int i;
    precision xs,xs2;

    xs=xs2=0;
    for(i=0;i<n;i++)
    {
        xs+=x[i];
        xs2+=x[i]*x[i];
    }

    xs/=n;
    xs2/=n;

    *mean=xs;
    *sigma=sqrt((n/(n-1.0))*(xs2-xs*xs));
}

```

```

*err=(*sigma)/sqrt((precision)n);

}

void Construye_Blocks(precision *x,precision *xb,int n_b,int b_s)
{
    int i,j;

    for(i=0;i<n_b;i++)
    {
        xb[i]=0;
        for(j=0;j<b_s;j++)
        {
            xb[i]+=x[i*b_s+j];
        }
        xb[i]/=b_s;
    }

}

```

7.8. Apéndice: Generador de distribuciones gaussianas

La distribución gaussiana tiene gran utilidad, y por ello daremos aquí un algoritmo para generar números con esta distribución partiendo de una distribución plana.

Queremos generar números con una distribución gaussiana con media 0 y varianza 1.
Lo más simple sería calcular la función densidad,

$$\tau(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} e^{-(t-\bar{t})^2} dt \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2} dx \quad (7.24)$$

El problema es que esta integral indefinida no puede calcularse de forma sencilla, no tiene una forma analítica en términos de funciones simples. Por tanto invertirla no es posible.

Existe una forma de resolver el problema, que es similar al truco usado para calcular la integral gaussiana en el Apéndice 6.9. Escribimos

$$\begin{aligned} I(x) &= \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \\ J(x, y) &= I(x)I(y) = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}} \end{aligned} \quad (7.25)$$

y pasando a coordenadas polares

$$J(x, y)dx dy = \frac{1}{2\pi} e^{-\frac{\rho^2}{2}} \rho d\rho d\varphi \quad (7.26)$$

con $\rho \in [0, \infty]$, $\varphi \in [0, 2\pi]$. La distribución de probabilidad para las nuevas variables es ahora

$$p(\rho, \varphi) = \frac{1}{2\pi} e^{-\frac{\rho^2}{2}} \rho \quad (7.27)$$

Es decir en φ tenemos una distribución plana, que factoriza con la distribución de ρ ,

$$p_0(\rho) = \rho e^{-\frac{\rho^2}{2}}, \quad p_1(\varphi) = \frac{1}{2\pi}$$

Notar que ambas están normalizadas.

Ahora ya podemos calcular la función densidad de probabilidad para ρ y φ recordando la sección 5.2.

En el caso de φ es trivial (recordar la figura 5.2),

$$p(\varphi) = \frac{1}{2\pi}; \tau(\varphi) = \frac{1}{2\pi} \int_0^\varphi dx = \frac{\varphi}{2\pi} \equiv \omega \quad (7.28)$$

de modo que para generar φ , generamos $\omega \in [0, 1]$ plano; entonces $\varphi = \omega 2\pi$

Procedemos ahora a generar ρ . La función densidad es ahora

$$\tau(\rho) = \int_0^\rho \tilde{\rho} e^{-\frac{\tilde{\rho}^2}{2}} d\tilde{\rho} \quad (7.29)$$

Esta integral puede evaluarse fácilmente

$$\int_0^\rho \tilde{\rho} e^{-\frac{\tilde{\rho}^2}{2}} d\tilde{\rho} = \frac{1}{2} \int_0^{t=\rho^2} e^{-\frac{t}{2}} dt = -(e^{-\frac{t}{2}})_0^{t=\rho^2} = -e^{-\frac{\rho^2}{2}} + 1 \quad (7.30)$$

Invirtiendo, tenemos

$$\begin{aligned} e^{-\frac{\rho^2}{2}} &= 1 - \tau(\rho) \\ -\frac{\rho^2}{2} &= \ln(1 - \tau(\rho)) \\ \rho &= \sqrt{-2 \ln(1 - \tau(\rho))} \end{aligned} \quad (7.31)$$

Recordando la construcción de ρ y de φ , tendremos que finalmente nuestro número con una distribución gaussiana será

$$\gamma = \rho \cos \varphi \quad (7.32)$$

Resumiendo entonces, para obtener un número random de acuerdo a una distribución gaussiana de media 0 y varianza 1, el procedimiento es

$$\begin{aligned} \text{Generamos } \omega &\in [0, 1] (\text{Uniforme}) \\ \text{Generamos } \nu &\in [0, 1] (\text{Uniforme}) \end{aligned} \quad (7.33)$$

$$\text{Calculamos } \rho = \sqrt{-2 \ln(1 - \nu)}; \gamma = \rho \cos(2\pi\omega)$$

el valor de γ es el número buscado, con distribución gaussiana normal.

Capítulo 8

Movimiento browniano en el plano

El movimiento browniano es un problema simple pero que ha jugado un importante papel en el desarrollo de la Física. Con lo expuesto hasta ahora podemos resolverlo completamente, tanto usando herramientas analíticas como numéricas. Servirá además para repasar y afianzar los conceptos adquiridos.

Existen diferentes variaciones del problema, con detalles diferentes pero propiedades similares. Aquí, el planteamiento del problema será el siguiente:

- Sea una partícula que se mueve con dos grados de libertad (plano).
- La partícula en $t = 0$ está en el origen de coordenadas.
- Cada intervalo de tiempo δt (constante) la partícula se mueve una distancia $\delta h \in [-\alpha, \alpha]$ en cada uno de los ejes.
- La distribución de δh es uniforme en el intervalo.

La partícula irá describiendo una trayectoria aleatoria en el plano. Una trayectoria concreta es impredecible, y cada vez que lancemos una partícula, su recorrido será diferente. Sin embargo sí que es posible calcular valores medios sobre miles de trayectorias, y hacer estadística sobre el sistema.

En concreto las preguntas que nos haremos serán:

1. **Distancia media:** Calcular en media la distancia al origen en función del tiempo. $\langle d(t) \rangle$
2. **Distribución de la distancia media:** A un tiempo fijo calcular la función densidad de probabilidad de la distancia al origen, $d(t)$, que llamaremos $P(t, d)$. Es decir, consideremos un valor del tiempo t : cada vez que una trayectoria llegue a este tiempo, estará a una determinada distancia del origen. Lo que queremos saber es: ¿Cuántas trayectorias están entre distancia 10 y 11? y entre distancia 23 y 29?, es decir *cuántas partículas están a una distancia dada del origen en un momento dado del tiempo*. En realidad en un intervalo de distancias, por supuesto (por ejemplo entre 20 y 30). El tiempo ya está discretizado en pasos δt . Estudiaremos como depende $P(t, d)$ del tiempo.
3. **Solución analítica:** Por último queremos resolver el problema analíticamente y encontrar la solución exacta para $\langle d(t) \rangle$ y $P(d, t)$ y comparar con la solución numérica obtenida.

Dejaremos en este caso como ejercicio al lector la escritura del código para resolver los puntos 1 y 2 (Ejercicio 8.3.1), y resolveremos en detalle el punto tercero.

8.1. Imagen intuitiva del problema

Cada trayectoria individual es impredecible. Sea el conjunto de M trayectorias C_i , con $i = 0, M - 1$. El tiempo para cada trayectoria varía con t_k con $k = 0, 1 \dots$

Situémonos en un tiempo fijo, digamos $t_N = N * \delta t$.

Cada trayectoria habrá recorrido un camino diferente y en ese momento estará a una distancia $d_i^N \equiv d_i(t_N)$ del origen: esta notación significa que una trayectoria concreta, por ejemplo la etiquetada como i , en el momento de tiempo t_N está a una distancia d_i^N del origen.

La distancia exacta es impredecible; pero no obstante podemos extraer información importante del sistema.

La idea que desarrollaremos analíticamente en la siguiente sección, consiste en calcular en primer lugar la densidad de probabilidad para la posición de la partícula en función de N , para a continuación calcular los valores medios integrando sobre todas las trayectorias, que es equivalente a calcular valores medios sobre la densidad anterior.

Todos los cálculos analíticos deberán ser comparados con los cálculos numéricos. En la simulación numérica del modelo, debemos generar miles o millones de trayectorias, y calcular la distancia media en función del tiempo. Es muy conveniente calcular el histograma a unos cuantos tiempos elegidos, para ver como evoluciona al ir avanzando dicho tiempo.

8.2. Solución analítica

La partícula se mueve en cada instante de tiempo una distancia

$$\begin{aligned} \delta x &\in [-\alpha, \alpha] \\ \delta y &\in [-\alpha, \alpha] \end{aligned} \tag{8.1}$$

con una distribución uniforme, siendo ambas variables independientes. Podemos por tanto estudiar el problema como dos movimientos independientes, uno en el eje X y otro en el eje Y . Notar que para que en nuestras simulaciones esta independencia de las variables sea cierta, necesitaremos usar un generador aleatorio que no tenga correlaciones a ninguna distancia.

Comencemos con el eje X ; el eje Y será idéntico.

Queremos estudiar la posición del móvil tras N pasos elementales; llamemos δ_i al incremento de x en el momento de tiempo i . Cada cambio es iterativo, es decir pasamos de x_0 a $x_1 = x_0 + \delta h_0$, a continuación a $x_2 = x_0 + \delta h_0 + \delta h_1$, etc. De este modo la posición final (recordemos que el valor inicial de x es x_0 que podemos tomar como 0), es

$$x_N = \sum_{i=0}^{N-1} \delta h_i$$

Entonces x_N es el resultado de una suma de variables todas ellas distribuidas de acuerdo a una densidad de probabilidad plana, dada por

$$p_\alpha(\delta x) = \frac{1}{2\alpha}. \tag{8.2}$$

Podemos calcular ahora la media y dispersión de δx , recordando la sección 6.2.1, obteniendo,

$$\begin{aligned} \langle \delta x \rangle &= \int_{-\alpha}^{\alpha} y P(y) dy = \frac{1}{2\alpha} \int_{-\alpha}^{\alpha} y dy = 0 \\ \langle (\delta x)^2 \rangle &= \int_{-\alpha}^{\alpha} y^2 P(y) dy = \frac{1}{2\alpha} \int_{-\alpha}^{\alpha} y^2 dy = \frac{1}{2\alpha} 2 \frac{\alpha^3}{3} \end{aligned} \tag{8.3}$$

y finalmente

$$\sigma_0 = \frac{\alpha}{\sqrt{3}} \tag{8.4}$$

Una vez conocida la media y dispersión de cada uno de los sumandos, podemos deducir la distribución de la *suma*, es decir de x_N , que nos dará la densidad de probabilidad para la coordenada x tras N pasos; basta recordar el TLC, que nos dice que aunque las densidades de cada δ_i sean planas, su suma será una gaussiana, con la misma media y con dispersión σ_N dada por

$$\sigma_N = \sqrt{N}\sigma_0$$

Concluimos finalmente (recordad 6.34) que tras N pasos (con N grande) la densidad de probabilidad para x es

$$P(x) = \frac{1}{\sqrt{N}\sigma_0\sqrt{2\pi}} e^{-\frac{x^2}{2N\sigma_0^2}} \quad (8.5)$$

Es decir, una gaussiana centrada en el origen, y con anchura que crece con \sqrt{N} . Esto corresponde con la idea intuitiva de que las trayectorias son simétricas alrededor del origen (la media es cero) y a la idea ya discutida abundantemente, de que la dispersión de N sumandos va como \sqrt{N} .

Podemos ahora considerar el movimiento en el eje y . Ambos movimientos son absolutamente independientes, por lo cual la densidad de probabilidad $p(x, y)$ será el producto de las densidades de probabilidad, y por tanto,

$$P_N(x, y) = \frac{1}{N\sigma_0^2 2\pi} e^{-\frac{x^2+y^2}{2N\sigma_0^2}} \quad (8.6)$$

o equivalentemente

$$P_N(x, y) = \frac{3}{N\alpha^2 2\pi} e^{-3\frac{x^2+y^2}{2N\alpha^2}} \quad (8.7)$$

A la vista de lo anterior, no es complicado extender el resultado para dimensión arbitraria. También se simple estudiar el caso suponiendo distribuciones no planas para $\delta x, \delta y$. El alumno puede hacerlo y llegar a los resultados finales sin mucha dificultad.

Una vez conocida exactamente la distribución para la posición x tras N pasos temporales, podemos calcular los valores medios que deseemos; dichos valores medios corresponden físicamente a promediar sobre miles (eventualmente infinitas) trayectorias. Concretamente nos hacemos la siguiente pregunta,

¿Cuál es el valor medio de la distancia al origen tras N pasos?

Notar que ahora la distancia es una cantidad definida positiva que ya no tiene media cero, como lo tenían los valores medios de las coordenadas.

La distancia es la Euclídea usual, es decir para un punto que tras N pasos tiene coordenadas (x, y) ,

$$d_N = \sqrt{x^2 + y^2} \quad (8.8)$$

y la media tras N pasos,

$$\langle d_N \rangle = \int_{-\infty}^{\infty} dx \int_{-\infty}^{\infty} dy \sqrt{x^2 + y^2} P_N(x, y) = \quad (8.9)$$

y pasando a coordenadas polares,

$$\begin{aligned} \langle d_N \rangle &= \int_0^{\infty} r^2 dr \int_0^{2\pi} d\theta \frac{1}{\frac{\alpha^2}{3} N 2\pi} \exp\left(-\frac{r^2}{\frac{2}{3} N \alpha^2}\right) = \frac{6\pi}{2N\pi\alpha^2} \int_0^{\infty} r^2 \exp\left(\frac{-r^2}{\frac{2}{3} N \alpha^2}\right) dr = \\ &= \frac{3}{2N\pi\alpha^2} 2\pi \frac{\sqrt{\pi}}{4} \left(\frac{2N\alpha^2}{3}\right)^{3/2} = \frac{3\sqrt{\pi}}{4N\alpha^2} \frac{2\sqrt{2}}{3\sqrt{3}} N^{3/2} \alpha^3 = \frac{\sqrt{\pi}\alpha}{\sqrt{6}} \sqrt{N} \end{aligned} \quad (8.10)$$

Reescribiendo este importante resultado,

$$\langle d_N \rangle = \frac{\sqrt{\pi}\alpha}{\sqrt{6}}\sqrt{N} \quad (8.11)$$

La distancia al promediar a muchas trayectorias es pues una función que crece con \sqrt{N} . Según pasa el tiempo, en media las partículas se alejan del origen como la raíz del número de pasos. O dicho en modo diferente, más coloquial, la partícula se aleja del origen con la raíz del tiempo.

8.2.1. Distribución de probabilidad en función de r

En las expresiones anteriores, el valor medio de la distancia tras N pasos, es en realidad el valor medio de r , y recordando 8.10 y podemos escribir

$$\begin{aligned} A &= \frac{3}{N\alpha^2} \\ \langle d_N \rangle &= A \int_0^\infty r dr e^{-(\frac{Ar^2}{2})} \end{aligned} \quad (8.12)$$

y por tanto la distribución de probabilidad de r tras N pasos viene dada por

$$p(r, N) = Ar e^{-Ar^2/2} \quad (8.13)$$

Conviene reflexionar sobre la forma de la función $p(r, N)$. El máximo lo tiene en $r = \frac{1}{\sqrt{A}} = \frac{\alpha}{\sqrt{3}}\sqrt{N}$. El valor de la función en el máximo es $h = \sqrt{A} \exp -1/2 \propto \frac{3}{\alpha\sqrt{N}}$. Por otra parte $p(r, N)$ vale 0 tanto en $r = 0$ como en $r = \infty$ y es definida positiva. Tenemos pues una especie de *campana* tal que el valor de x donde alcanza el máximo crece con la raíz de N , y cuya altura en ese máximo disminuye como $1/\sqrt{N}$. Como el área es fija (pues está normalizada a 1), esto significa que al crecer N la campana se hace cada vez más ancha y baja, a la vez que su centro se aleja del origen. Comprobar todo esto con *gnuplot*, e interpretar físicamente el comportamiento señalado.

8.3. Ejercicios

8.3.1. Simulación del movimiento Browniano

Escribir un código que genere trayectorias para un movimiento browniano de acuerdo a lo definido en el texto. Generar N de estas trayectorias, y calcular la distancia media y la distribución de las distancias en función del tiempo. Dibujar estas cantidades. Comprobar que los resultados coinciden con el cálculo analítico.

8.3.2. Difusión de Partículas

Este Ejercicio se corresponde con un Trabajo de Examen. Se muestra también las preguntas de Comprobación y Modificación del Examen. Puede verse el código correspondiente en [8.5.3](#)

Una persona estornuda y emite N (grande) gotas microscópicas de agua, todas iguales, que se difunden de acuerdo a un movimiento Browniano: a cada paso de tiempo δt (pequeño) cada gota puede moverse independientemente de las demás, una distancia d distribuida uniformemente, $d \in [0, \rho]$, y de forma isótropa en las tres dimensiones (es decir, la partícula a cada paso se mueve en el interior de una esfera de radio ρ). El estornudo se produce en el **centro** de una habitación cúbica de lados $L \times L \times L$, donde las partículas que llegan a cualquiera de las seis paredes, rebotan reflejándose especularmente.

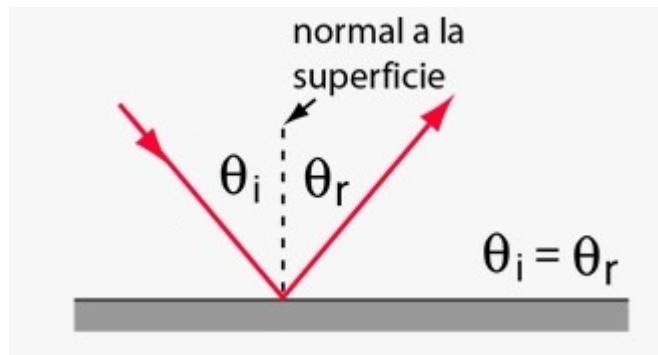


Figura 8.1: Esquema de una trayectoria de reflexión especular.

Calcular el número de partículas en cualquier punto de la habitación transcurrido un cierto tiempo $T = n \cdot \delta t$, $n \in \mathbb{N}$, $n > 0$.

Para ello, dividir la habitación en pequeñas cajas cúbicas de lado $L/10$ (es decir, habrá un total de 1000 cajas cúbicas). El programa recibirá como *input* unas coordenadas de medición y deberá identificar en qué caja se encuentran dichas coordenadas y medir las partículas en esa caja.

Además, el programa debe calcular el número de choques de las partículas en cualquiera de las seis paredes de la habitación.

Nota: Sitúese el origen de coordenadas en una esquina cualquiera de la habitación.

Como guía para realizar pruebas, el alumno deberá considerar los siguientes órdenes de magnitud en las cantidades relevantes del problema.

- $N \sim 500000$ partículas
- $\rho \sim 0.2$ m
- $L \sim 10$ m
- $T \sim 15$ s
- $\delta t \sim 0.01$ s

Comprobación

Considérese una habitación de geometría (5, 4, 2) donde generamos 10^6 partículas. Utilizar $\rho = 0.1$ y $\delta t = 0.01$.

Calcúlese el número de partículas al cabo de 12 segundos en los siguientes puntos:

- Punto 1: coordenadas (2.3, 2.8, 1.5).
- Punto 2: coordenadas (0, 0, 0).
- Punto 3: coordenadas (1.3, 3.2, 1.0).

Además, calcúlese el número total de colisiones de las partículas contra las paredes de la habitación.

Modificación

Repítase el anterior programa con los datos indicados, pero suponiendo que las partículas al chocar contra cualquiera de las seis paredes son absorbidas con una probabilidad $x = 0.008$.

Calcúlese el número de partículas al cabo del tiempo indicado en la comprobación, en los tres puntos previos. Además, calcúlese el número total de colisiones de las partículas contra cualquiera de las paredes de la habitación y el número total de partículas absorbidas.

8.4. Problemas

8.4.1. Problema 1

Considerar un movimiento browniano donde en cada paso el movimiento no es continuo sino sólo uno de los dos valores $+a$ o $-a$. Realizar el cálculo teórico y simularlo en el ordenador, usando a del orden de 1, comprobando que se han obtenido los mismos resultados.

8.4.2. Problema 2

Calcular $p(r, N)$ de la expresión 8.13 en el caso del Ejercicio 8.3. Comprobar que coincide el resultado numérico y analítico usando gnuplot para dibujar ambos resultados superpuestos.

8.5. Código en C

8.5.1. Conceptos de Programación: Código estructurado, uso de funciones predefinidas

Este código se da como apoyo al alumno. Esta sin estructurar, y el alumno debe escribir un nuevo código, extrayendo algunas ideas, pero donde todo esté estructurado, delegado en funciones, etc. En concreto, la parte del Histograma debe hacerse usando la función ya programada con anterioridad.

8.5.2. Movimiento browniano en $d = 2$

```
/*
    Movimiento browniano en el plano
    Se usa el generador de Parisi-Rapuano
*/

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

#define NormRANu (2.3283063671E-10F)
#define Frec_Max 50
#define Tr_Max 100000

unsigned int irr[256];
unsigned int ir1;
unsigned char ind_ran,ig1,ig2,ig3;

float freq_d[Frec_Max];
float d[Tr_Max];
float d_final[Tr_Max];

FILE *fout;

extern float Random(void);
extern void ini_ran(int SEMILLA);

main()
{
    float epsilon;
    float x,y;
    float corte,inc;
    int NITER,N_tr;
    int i,i_tr;
    float min,max,delta,temp_d,Norm;
    int inter;

    ini_ran(123456789); //Inicializo el generador random
    epsilon=8.0; // Cada paso es uniforme entre [-epsilon,epsilon]
    corte=100.0; //Caja para calcular histogramas

    NITER=500;
    N_tr=50000;

    for(i=0;i<Frec_Max;i++) //Histograma de distancia
        freq_d[i]=0;

    for(i=0;i<Tr_Max;i++) //Distancia media
        d[i]=d_final[i]=0;

    for(i_tr=0;i_tr<N_tr;i_tr++)
    {
        x=y=0.0; //Para cada trayectoria vuelvo al origen.
        for(i=0;i<NITER;i++)
        {
            inc=2*epsilon*(0.5-Random());
            x+=inc;
        }
    }
}
```

```

inc=2*epsilon*(0.5-Random());
y+=inc;
temp_d=sqrt(x*x+y*y);
d[i]+=temp_d; //Distancia media al origen en tiempo i
}

//Guardo datos para Calcular el histograma en el ultimo punto de la trayectoria

d_final[i_tr]+=temp_d;
}

//Calculo el histograma

min=100000; //Ajusto los rangos
max=-100000;

for(i_tr=0;i_tr<Tr_Max;i_tr++)
{
    if(d_final[i_tr]<min)min=d_final[i_tr];
    if(d_final[i_tr]>max)max=d_final[i_tr];
}
printf("Rango histograma=%f %f\n",min,max);

delta=(max-min)/Frec_Max;
for(i=0;i<N_tr;i++)
{
    inter=(int)(d_final[i]-min)/delta;
    printf("indice hist=%d\n",inter);
    freq_d[inter]++;
}
Norm=1.0/(delta*N_tr); //Para normalizar freq_d
for(i=0;i<Frec_Max;i++)

printf("%f %f\n",min+i*delta,freq_d[i]*Norm);
fout=fopen("rw_d.dat","wt");

for(i=0;i<NITER;i++)
    fprintf(fout,"%d %f\n",i,d[i]/N_tr);

fclose(fout);

fout=fopen("freq_d.dat","wt");

for(i=0;i<Frec_Max;i++)
    fprintf(fout,"%f %f\n",min+i*delta,freq_d[i]*Norm);

fclose(fout);
}

float Random(void)
{
    float r;

ig1=ind_ran-24;
ig2=ind_ran-55;
ig3=ind_ran-61;
irr[ind_ran]=irr[ig1]+irr[ig2];
ir1=(irr[ind_ran]^irr[ig3]);

ind_ran++;

r=ir1*NormRANu;

//printf("r=%f\n",r);

return r;
}

void ini_ran(int SEMILLA)
{

```

```

intINI,FACTOR,SUM,i;
srand(SEMILLA);
INI=SEMILLA;
FACTOR=67397;
SUM=7364893;

for(i=0;i<256;i++)
{
    INI=(INI*FACTOR+SUM);
    irr[i]=INI;
}

ind_ran=ig1=ig2=ig3=0;
}

```

Resultado exacto versus numérico con gnuplot

Con los datos del programa anterior, con el siguiente fichero de comandos podemos dibujar en **gnuplot** los resultados obtenidos numéricamente y los obtenidos en el texto analíticamente. Cambiando valores de los parámetros, el resultado debe coincidir en todos los casos.

```

p "rw_d.dat" t "Distancia media"
epsilon=8.0
pi=2.0*asin(1.0)
C=sqrt(pi)*epsilon/sqrt(6.0)

rep C*sqrt(x)

pause -1

p "frec_d.dat" u 1:2 t "Histograma de distancias"

Niter=500
A=3/(Niter*epsilon*epsilon)
B=A/2.0

p_d(x)=A*x*exp(-B*x*x)

rep p_d(x)

```

8.5.3. Código para el Trabajo de Examen sobre Difusión

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdarg.h>
4 #include <math.h>
5 #include <string.h>
6 #include <time.h>
7
8 //Parisi-Rapuano
9 unsigned char ip=128,ip1=128-24,ip2=128-55,ip3=128-61;
10 unsigned irr[256];
11 //#define RAND_MAX 0x7fffffff
12 #define NormRAN (1.0/( (float) RAND_MAX+1.0))
13 #define RAN() ( (float) rand() * NormRAN )
14 #define FNORM (2.3283063671E-10F)
15 #define RANDNEW ((irr[ip++]=irr[ip1++]+irr[ip2++]) ^ irr[ip3++])
16 #define FRANDOM (FNORM*RANDNEW)
17
18 /* Global variables */
19 int N; //Numero de particulas
20 double d; //distancia maxima a la que se puede mover una particula
21 double A,B,C; //Tamagno de la habitacion

```

```

23 double Lx,Ly,Lz; //Tamagno de las cajas que subdividen la habitacion
24 double xp,yp,zp; //Coordenadas del punto que quiero medir
25 int xcoord,ycoord,zcoord; //Coordenadas de la caja en la que voy a medir
26 int npasos; //numero de pasos en la simulacion
27 int choques=0; //Cuenta del numero de choques en las paredes
28 int part_caja=0; //Particulas finales en la caja que mido
29 double densidad; //Densidad en el punto de interes
30 double const PI=acos(-1.);

31
32 typedef struct{
33     double x,y,z; //posicion de la particula
34 }particula;
35
36 /* Funciones */
37 void print_and_exit(char *, ...);
38 void ini_rng();
39 void vector_esfera(double *, double *, double *, double);
40 void read_input(char *file_name);

41
42
43
44 int main(int argc, char **argv){
45     //particula *parts;
46     int i,j;
47     double xn,yn,zn;
48     double dist;
49     char file_name[1024];

50
51
52     //Leo input
53     read_input(file_name);
54
55     //Reservamos memoria
56     //parts=(particula *)malloc(N*sizeof(particula));
57     particula part;
58
59     //Inicializamos el generador de numeros aleatorios
60     srand(time(NULL));
61     ini_rng();

62
63     //Calculo las coordenadas de la caja donde voy a medir
64     xcoord=(int)(xp/Lx);
65     ycoord=(int)(yp/Ly);
66     zcoord=(int)(zp/Lz);

67
68
69     //printf("N=%d ; npasos=%d \n",N,npasos);
70     //Inicializamos simulacion
71     for(i=0;i<N;i++){
72         //Condiciones iniciales
73         part.x=A/2.;
74         part.y=B/2.;
75         part.z=C/2.;
76         if((N%1000)==0)
77             printf("i=%d\n",i);

78
79         //printf("particula %d: \n",i);
80         for(j=0;j<npasos;j++){
81             //Doy un paso
82             dist=d*FRANDOM;
83             vector_esfera(&xn,&yn,&zn,dist);
84             part.x+=xn;
85             part.y+=yn;
86             part.z+=zn;

87             //printf("x,y,z = %.14g ,%.14g ,%.14g \n",part.x,part.y,part.z);
88             //Compruebo paredes
89             if(part.x<0){
90                 part.x*=-1;
91                 choques++;
92             }

```

```

93     } else if(part.x>A){
94     part.x=2*A-part.x;
95     choques++;
96     }
97
98     if(part.y<0){
99     part.y*=-1;
100    choques++;
101    } else if(part.y>B){
102    part.y=2*B-part.y;
103    choques++;
104    }
105
106    if(part.z<0){
107    part.z*=-1;
108    choques++;
109    } else if(part.z>C){
110    part.z=2*C-part.z;
111    choques++;
112    }
113
114 //Miro si estoy en la caja que me interesa
115 if((int)(part.x/Lx)==xcoord && (int)(part.y/Ly)==ycoord && (int)(part.z/Lz)==
zcoord){
116     part_caja++;
117 }
118 }
119
120 densidad=(double)part_caja/(Lx*Ly*Lz);
121 /* printf("particulas en la caja: %d\n",part_caja); */
122 /* printf("densidad=% .14g\n",densidad); */
123 /* printf("numero de choques: %d \n",choques); */
124 printf("%d %.14g %d \n",part_caja,densidad,choques);
125
126 return 0;
127 }
128
129
130
131 void read_input( char *file_name){
132 FILE *Fin;
133 int dummy; //para que no me salgan warnings por no hacer nada con el fscanf
134
135 if(NULL==(Fin=fopen("input_test.txt","rt"))){
136     print_and_exit("Error al abrir el archivo %s \n",file_name);
137 }
138
139 dummy=fscanf(Fin,"%d", &N);
140 dummy=fscanf(Fin,"%lf", &d);
141 dummy=fscanf(Fin,"%lf", &A);
142 dummy=fscanf(Fin,"%lf", &B);
143 dummy=fscanf(Fin,"%lf", &C);
144 dummy=fscanf(Fin,"%lf", &Lx);
145 dummy=fscanf(Fin,"%lf", &Ly);
146 dummy=fscanf(Fin,"%lf", &Lz);
147 dummy=fscanf(Fin,"%lf", &xp);
148 dummy=fscanf(Fin,"%lf", &yp);
149 dummy=fscanf(Fin,"%lf", &zp);
150 dummy=fscanf(Fin,"%d", &npasos);
151
152 printf(" N=%d,d=%f,npasos=%d",N,d,npasos);
153
154 dummy++;
155
156 fclose(Fin);
157
158 //Pinto lo leido
159 /* printf("----- INPUT ----- \n"); */
160 /* printf("N = %d \n",N); */

```

```

162 /* printf("d = %.14g \n",d); */
163 /* printf("A = %lf \n",A); */
164 /* printf("B = %lf \n",B); */
165 /* printf("C = %lf \n",C); */
166 /* printf("Lx = %lf \n",Lx); */
167 /* printf("Ly = %lf \n",Ly); */
168 /* printf("Lz = %lf \n",Lz); */
169 /* printf("xp = %lf \n",xp); */
170 /* printf("yp = %lf \n",yp); */
171 /* printf("zp = %lf \n",zp); */
172 /* printf("npasos = %d \n",npasos); */
173 /* printf("-----\n"); */
174 }
175
176 void vector_esfera(double *x, double *y, double *z, double dist){
177     double theta,phi;
178
179     theta=2*PI*FRANDOM;
180     phi=acos(1-2*FRANDOM);
181
182     *x=dist*sin(phi)*cos(theta);
183     *y=dist*sin(phi)*sin(theta);
184     *z=dist*cos(phi);
185 }
186
187
188 void ini_rng(){
189     int i;
190     for (i=0; i<256; i++){
191         irr[i]=rand()+rand();
192     }
193 }
194
195 void print_and_exit(char *format, ...){
196     va_list list;
197
198     va_start(list,format);
199     vprintf(format,list);
200     va_end(list);
201     exit(1);
202 }
203 }
```


Capítulo 9

El Modelo de Ising

9.1. Introducción

En este capítulo estudiaremos el modelo de Ising, que presenta un complejidad media y requiere prácticamente todos los conocimientos adquiridos durante el curso. Además es un sistema físico sumamente interesante que permite observar resultados no triviales.

9.2. El modelo de Ising

El modelo de Ising tiene su origen en el estudio de los materiales magnéticos; su formulación es extremadamente simple y está en la base de la Mecánica Estadística, de la Teoría Cuántica de Campos y de los más modernos desarrollos en Sistemas Complejos, Optimización, incluso el estudio del plegamiento de proteínas o el modelado de las relaciones en Redes Sociales. Inicialmente su estudio se basó en cálculos analíticos, logrando Onsager la solución exacta en $d = 2$. En $d=3$ no se conoce la solución exacta, y se ha podido avanzar gracias a complejas aproximaciones analíticas, al estudio del Grupo de Renormalización, y finalmente con el uso de simulaciones en ordenador.

El modelo de Ising pretende, dada su simplicidad, reproducir las propiedades esenciales, cuantitativas, de diferentes materiales magnéticos. Dichos materiales, cuando baja la temperatura por debajo de una dada, adquieren magnetización espontánea, sufriendo una *transición de fase*; al volver a calentarlos, la magnetización desparece. El origen del fenómeno está en el comportamiento ferromagnético de los átomos que componen el material.

El modelo de Ising reúne las características mínimas para reproducir lo esencial de este fenómeno; ciertamente no los detalles, como por ejemplo el valor exacto en grados de la Temperatura de la transición, pero sí las propiedades esenciales (Universales), como el tipo de transición de fase, el comportamiento crítico del sistema, la aparición de magnetización espontánea, etc.

9.2.1. Definición de la Red y de los Spines

Consideremos un sistema bidimensional. Formamos un retículo cuadrado, y llamaremos *nodo* a cada uno de los puntos de la malla, y *link* a cada uno de las líneas que une estos puntos, como se indica en la figura 9.1. Esto forma parte de la geometría del sistema. Ahora definimos las variables dinámicas sobre estos puntos. A cada nodo le asignaremos una variable que *vivirá* allí, y que solamente podrá tomar los valores 1 o -1; a esta variable la llamaremos spin, $s \in \{-1, 1\}$. Veáse la fig 9.1.

Pongamos pues en cada nodo un valor al azar 1 o -1. Este conjunto de valores en todo el retículo es lo que llamaremos una *Configuración*, etiquetada con un índice para distinguirla de las demás: C_α .

Supongamos que tenemos una red de lado $L \times L$, con $V = L^2$ nodos por tanto. Dar una configuración consiste en dar todos y cada uno de los L^2 valores de los spines. Si los numeramos de 0 a $V - 1$, tendremos pues que una configuración C_α viene dada por

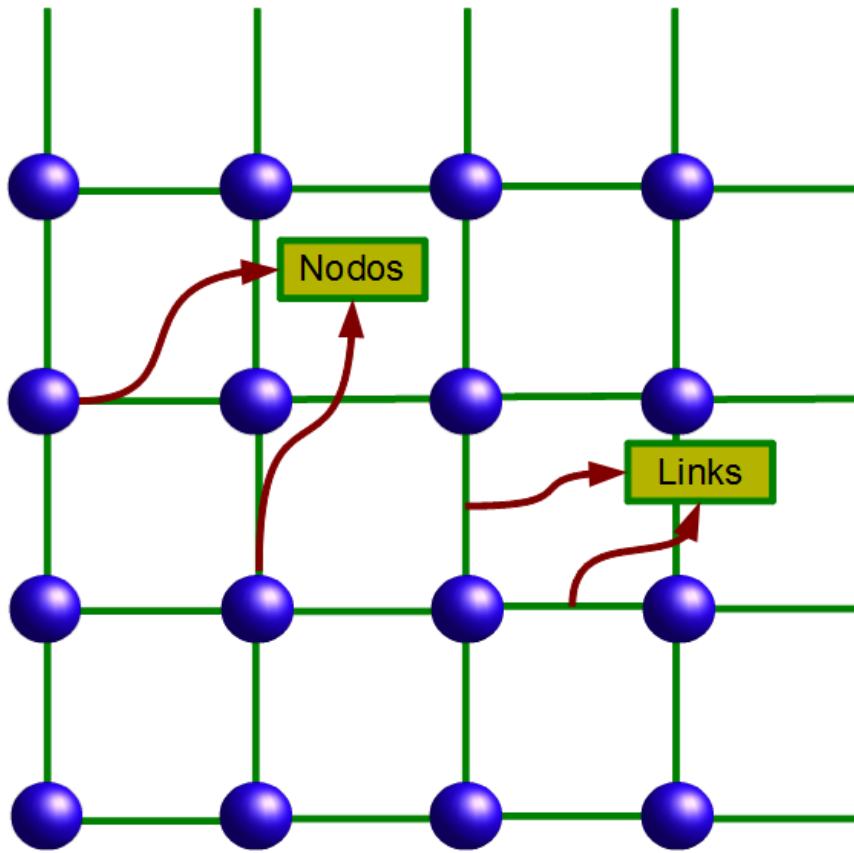


Figura 9.1: Definición de la Red y las variables en el modelo de Ising en $d = 2$

$$C_\alpha = \{s_0, s_1, \dots, s_{V-1}\} \quad (9.1)$$

En la figura 9.2 podemos ver una configuración típica en una red 5×4 .

El conjunto que contiene todas las configuraciones posibles, es lo que llamaremos *Espacio de Configuraciones* y lo denotaremos por C ; de este modo tenemos que $C_\alpha \in C$

En el caso del modelo de Ising, donde hemos discretizado el espacio y las variables que viven en los nodos son también discretas, el espacio de configuraciones es numerable y finito (para L finito). Es fácil calcular cuántas configuraciones diferentes contiene. Para ello basta fijarse que al ser cada nodo independiente de los demás, en cada uno de ellos podemos tener dos valores del spin: $s_n = \{+1, -1\}$. Si tuviéramos una red con un sólo punto, el número de configuraciones sería 2. Si tuviéramos dos puntos, el número sería 2^2 , si tuviéramos 3 puntos, 2^3 , etc. En el caso de V puntos tendremos que el espacio de configuración C contiene 2^V puntos o configuraciones.

Si bien para retículos muy pequeños, el número de configuraciones es relativamente bajo, hay que notar que para valores de L del orden de los que pueden simularse en los ordenadores actuales el espacio de configuraciones contiene un número astronómico de configuraciones. Un retículo fácilmente asequible es $L = 32$, en el cual tenemos 2^{1024} configuraciones, aproximadamente igual a 10^{300} . Este número es increíblemente mayor que el número de protones que cabrían en un Universo macizo de materia nuclear, que sería del orden de 10^{120} ; o del número estimado de protones del Universo actualmente, en torno a 10^{80} . Con los algoritmos que estudiaremos a continuación, y con otros habituales en la literatura, se pueden simular redes de hasta $L = 1000$.

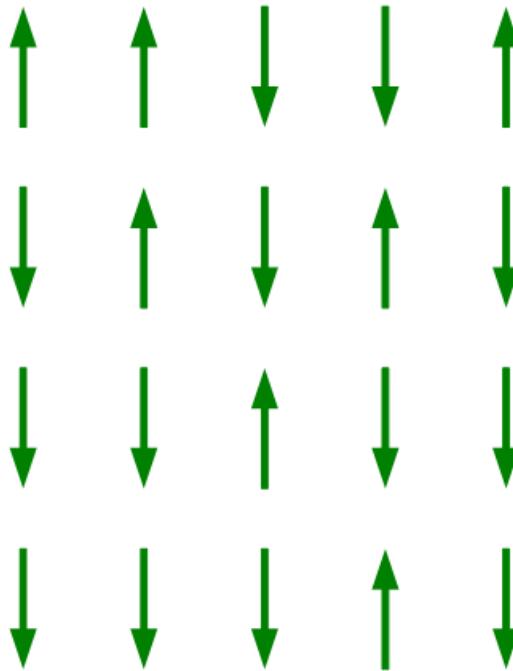


Figura 9.2: Una configuración genérica en el modelo de Ising en $d = 2$

9.2.2. Definición de la Energía

En los sistemas ferromagnéticos reales que queremos describir, los spines interactúan entre sí y tienden a alinearse. Sin embargo si la agitación térmica es suficientemente alta no logran orientarse todos y el sistema no está magnetizado. Si la agitación térmica disminuye (la temperatura desciende) la fuerza entre los spines logra que todos se orienten, apareciendo la magnetización.

La *agitación térmica* está íntimamente relacionada con la Energía del sistema. Por tanto, para describir este comportamiento, comenzamos asociando a cada configuración una *Energía* construida del siguiente modo.

- Llamamos n a los puntos de la red, y definimos los vectores unitarios en las direcciones X e Y , que llamaremos indistintamente \hat{i}, \hat{j} o $\hat{0}, \hat{1}$, genéricamente $\hat{\mu}$.
- Nos situamos en el nodo n , con su spin asociado, s_n
- Dicho spin tiene dos vecinos en la dirección positiva de los dos ejes, $s_{n+\hat{i}}, s_{n+\hat{j}}$.
- Multiplicamos s_n por cada uno de ellos y sumamos: $s_n s_{n+\hat{i}} + s_n s_{n+\hat{j}}$
- Repetimos esto para cada punto de la red y sumamos todos los términos.
- Obtenemos finalmente la Energía

$$E = - \sum_{n=0}^{V-1} (s_n s_{n+\hat{i}} + s_n s_{n+\hat{j}}) = - \sum_{n=0}^{V-1} \sum_{\hat{\mu}=\hat{0}, \hat{1}} s_n s_{n+\hat{\mu}}$$

donde n recorre el volumen y μ las dos dimensiones (x, y) , es decir los dos vectores \hat{i}, \hat{j} .

- E es una cantidad extensiva. Definimos la cantidad intensiva como

$$e = \frac{-1}{Vd} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}$$

con d la dimensión, en este caso 2. De este modo e varía en el intervalo $[-1, 1]$.

Gráficamente, con los nodos los puntos azules y los links las líneas verdes, uno de los términos del sumatorio puede verse indicado con flechas en la figura 9.3. Estrictamente, E o e son *funcionales*, es decir aplicaciones que van del espacio de configuración C en \mathbb{R} .

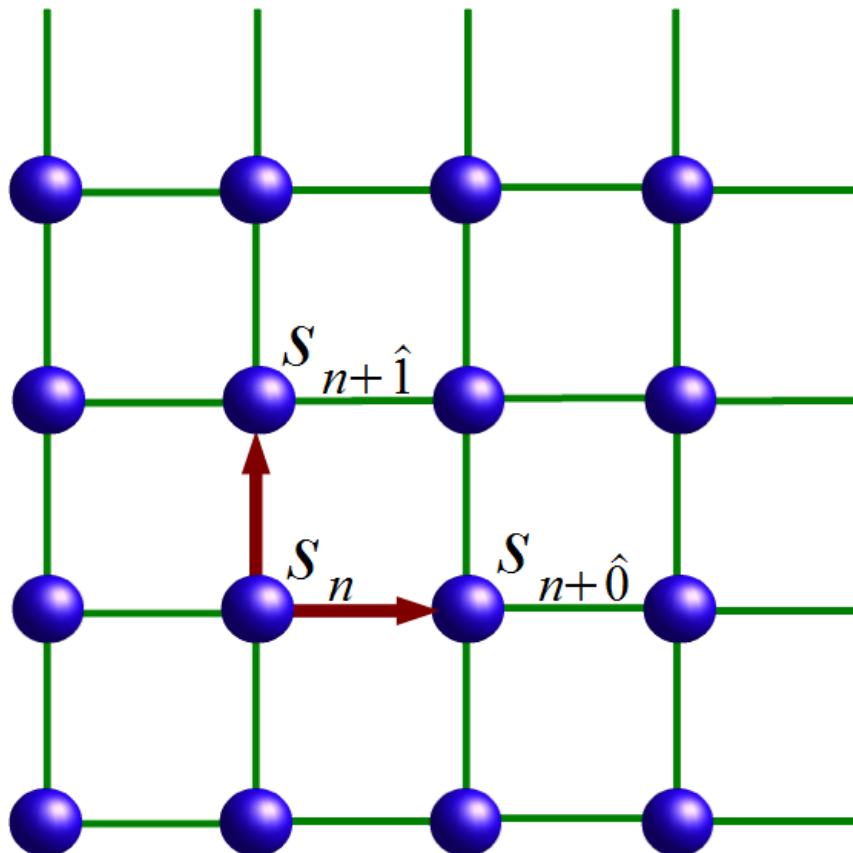


Figura 9.3: Primeros vecinos hacia adelante en la red cuadrada del modelo de Ising

9.2.3. Definición de la Magnetización

Desde el punto de vista físico, la propiedad más importante que define los materiales magnéticos es su *Magnetización*. Ésta se presenta cuando la mayoría de los spins del sistema, o *imanes elementales* están orientados en la misma dirección, produciendo un campo magnético macroscópico, observable fácilmente. En nuestro modelo la magnetización corresponde simplemente con la suma de los spins de toda la red. Si queremos una cantidad intensiva, la definición correcta es

$$m = \frac{1}{V} \sum_{n=0}^{V-1} s_n \quad (9.2)$$

Esta cantidad así definida varía en el intervalo $[-1, +1]$.

9.3. Construcción de la Red y condiciones de contorno

Existen diferentes maneras para numerar los puntos de la red bidimensional.

La que primero tiende a implementarse es considerar dos coordeandas, y construir el spin con dos indices, el primero indicando la coordenada x y el segundo la coordenada y . Tendríamos así, que el spin sería $s(x, y)$, e implementado en C temdríamos un vector $s[L][L]$.

Para encontrar el spin a la derecha de uno dado haríamos $s[x + 1][y]$, para encontrar el de la izquierda $s[x - 1][y]$ etc.

Nosotros aquí no usaremos el anterior método, sino que optaremos por uno de los más eficientes, aunque tal vez no el más intuitivo; este modo de numeración sin embargo es el más simple para implementar todas las partes del programa y el más fácilmente generalizable cuando queremos cambiar el tamaño o incluso pasar a dimensiones más altas.

Tomaremos el eje X en la dirección horizontal hacia la derecha y el eje Y en la dirección vertical hacia arriba. Esta elección sirve sólo para visualizar y dibujar la red; desde el punto de vista del programa esta elección carece de contenido. Para numerar los puntos usaremos un solo índice que correrá de 0 a $V - 1$. El primer punto será el 0, el siguiente en el eje X , el 1, el ultimo de su fila el $L - 1$. El primero de la segunda fila será el L , el siguiente el $L + 1$ y así sucesivamente. El último punto será el $V - 1$. Gráficamente, considerando una red cuadrada con $L = 4$, tendremos todos los puntos numerados como indica la figura 9.4.

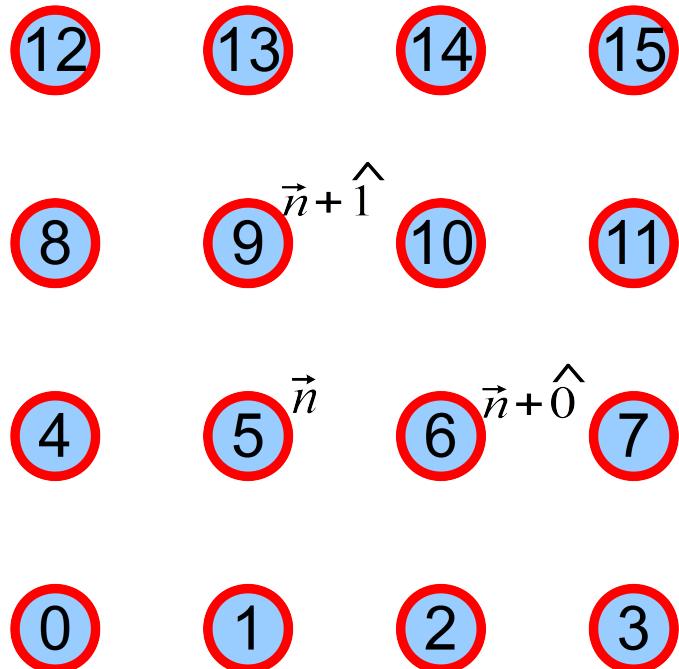


Figura 9.4: Numeración de la red y primeros vecinos hacia adelante

De este modo si queremos recorrer toda la red, el código es sumamente simple. Por ejemplo, para calcular la magnetización bastaría,

```
for(magnet=0,n=0;n<V;n++)
    magnet+=s[n];
```

Este es un observable que sólo depende de un punto; la Energía depende de un punto y sus vecinos. Entonces debemos saber para cada punto cual está a su derecha y cual está arriba. Mirando la gráfica 9.4 vemos que dado n , para pasar a $n + \hat{1}$ (movernos en la dirección x positiva) debemos simplemente sumar 1. Por ejemplo dado el punto 2, que en término de sus coordenadas cartesianas es $(x, y) = (2, 0)$, para pasar al siguiente en la dirección $\hat{0}$, le sumamos 1, es decir el 3, que corresponde a $(x, y) = (3, 0)$.

De forma similar si queremos pasar de un punto n con coordenadas (x, y) al punto vecino en la dirección $\hat{1}$, o dirección y , le sumamos L , pasando al de coordenadas $(x, y + 1)$.

Esto es cierto en todos los puntos salvo para los que están en la frontera por la derecha, como por ejemplo el punto 7, que corresponde a coordenadas $(x, y) = (3, 1)$ o a los que están en la frontera por arriba, como el punto 13 que corresponde al $(1, 3)$

Debemos decidir ahora qué hacemos en estos bordes. Comencemos con los puntos en la frontera de la derecha, *i.e.* con la coordenada x igual a $L - 1$. Podríamos decir que a la derecha de dicho punto no hay nada, y en este caso, al calcular la energía para este punto, en vez de dos términos tendríamos sólo uno (el de interacción del punto 7 con el punto 11). Esto es llamado *condiciones de contorno libres* (Free Boundary Conditions, fbc) y corresponde físicamente a una muestra real de material que acaba ahí. Sin embargo desde el punto de vista teórico, de simulación, y de búsqueda del límite termodinámico no es la mejor opción.

Un sistema termodinámico supuesto bidimensional tiene del orden de $V = 10^{16}$ puntos, o $L = 10^8$. Esto está fuera del alcance de cualquier ordenador. En un sistema en $d = 3$, tendríamos del orden de $V = 10^{24}$ puntos (Número de Avogadro).

En el ordenador podemos simular tamaños para este modelo con L del orden de 10^2 . Si ponemos condiciones de contorno libres, rompemos la invariancia translacional del modelo: los puntos cerca de la pared se comportan diferentes de los más alejados. Eso hace que observables calculados en regiones diferentes tengan comportamientos diferentes. Si L fuera extremadamente grande, lo que ocurre en la frontera sería despreciable en el interior; si L es pequeño, su influencia se deja notar en todo el sistema, siendo este hecho muy diferente de lo que ocurre en el límite termodinámico. El acercamiento del sistema a dicho límite termodinámico es mucho menos suave. Por otro lado los cálculos teóricos se vuelven enormemente complejos con estas condiciones de contorno libres.

Habitualmente se usan unas condiciones de contorno que preserven la invariancia por traslaciones del modelo, de modo que todos los puntos son equivalentes. En concreto usaremos Condiciones de Contorno Periódicas (*Periodic Boundary Conditions*).

Éstas consisten en decir que cuando un punto de la frontera derecha mira a la derecha, ve al primer punto de su fila. Lo mismo en el eje Y : cuando un punto de la frontera superior mira hacia arriba, ve al primer punto de la misma columna. De forma similar cuando miramos hacia atrás desde un punto de la primera fila o columna, nos encontramos con los últimos de la fila o columna correspondiente. Veáse la figura 9.5

Estas condiciones de contorno hacen que desde el punto de vista geométrico, conectemos el lado de la derecha con el de la izquierda y el de arriba con el de abajo. Es como si sobre una hoja de papel primero pegáramos un lado con el opuesto: obtendríamos un cilindro. Si a continuación pegamos las dos circunferencias de los extremos del cilindro, tendremos la topología completa; para ello el papel debería ser muy flexible. La figura geométrica que aparece es un toro, similar a la superficie de una cámara de neumático o de un donuts. Una posible visualización del toro como una superficie inmersa en el espacio tridimensional, puede verse en la figura 9.6, donde los nodos están en los cruces de las líneas, que representan los links.

Notar que aquí para visualizar el toro, hemos pasado de dos a tres dimensiones, si bien el sistema es intrínsecamente bidimensional.

Esto se extiende a dimensiones mayores: en $d = 3$ con condiciones de contorno periódicas, tendremos un hipertoro, pero ahora para visualizarlo, nosotros deberíamos vivir en $d = 4$, cosa que por desgracia no es posible...

Si hubieramos usado los direccionamientos con dos índices, $s[x][y]$, tendríamos que para movernos a la derecha, e implementar las condiciones de contorno periódicas, deberíamos hacer $s[(x + 1) \% L][y]$, para movernos hacia arriba $s[x][(y + 1) \% L]$, etc. Estas operaciones son relativamente costosas, tanto por direccionar dos índices como por realizar la operación de módulo, y dado que se hacen en la parte más interna del programa, es más eficiente el método con un sólo índice.

Existen otras posibles condiciones de contorno, como las antisimétricas, similares a la periódicas, pero tales que al atravesar la frontera el spin que se ve es el cambiado de signo del otro lado.

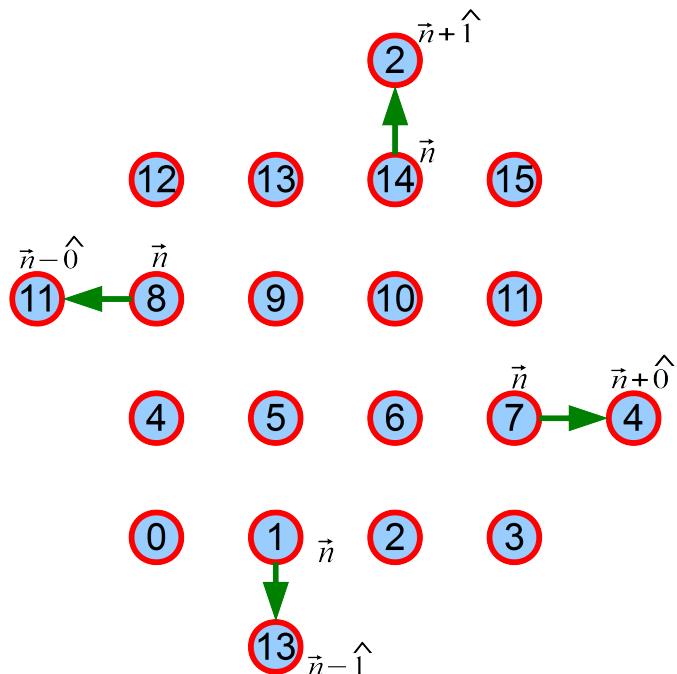


Figura 9.5: Vecinos hacia adelante y hacia atrás en la frontera de la red con condiciones de contorno periódicas

9.3.1. Construcción de los Direccionamientos

Dado que hemos decidido que los spines estén numerados con un solo índice, debemos ahora establecer un algoritmo tal que dado un punto nos permita movernos a la derecha o arriba, lo que es necesario para calcular la energía. En la figura 9.5 (dibujada con $L = 4$, pero hagámoslo general) vemos que

- Dirección x positiva:

si la coordenada x es menor que $L - 1$, $n + \hat{0} = n + 1$

si la coordenada x es igual a $L - 1$, $n + \hat{0} = n - (L - 1)$

- #### ■ Dirección *y* positiva:

si la coordenada y es menor que $L - 1$, $n + \hat{1} = n + L$

si la coordenada y es igual a $L - 1$, $n + \hat{1} = n - L(L - 1)$

Podemos implementar esto en el programa del siguiente modo (comencemos con la dirección x): Definimos un vector que dependa de la coordenada cartesiana x del punto, y que nos indique lo que debemos sumar al punto n para ir hacia adelante. En concreto

```
for(i=0;i<(L-1);i++)  
    xp[i]=1;  
xp[L-1]=-(L-1);
```

De este modo cuando queramos saber el spin hacia adelante en el eje X desde un punto n , con coordenada en dicho eje igual a x , bastará hacer

$s[n+xp[x]]$

De forma similar para el eje Y ,

Podrás construir los vectores usando el siguiente segmento de código:

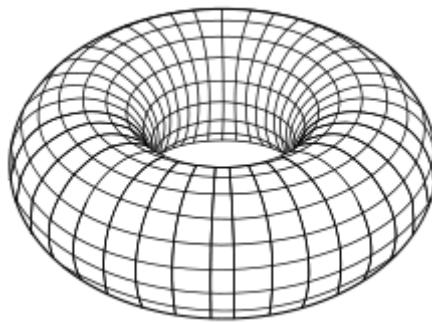


Figura 9.6: Visualización en $d = 3$ de la red en $d = 2$ con condiciones de contorno periódicas, como un toro

```
for(i=0;i<(L-1);i++)
    yp[i]=L;
yp[L-1]=-L*(L-1);
```

Para ver que efectivamente funciona, consideremos por ejemplo el punto 7 de la figura 9.5; sus coordenadas son $(x, y) = (3, 1)$. Notar que $7 = y * L + x$. El punto a la derecha del 7 es el 4. Efectivamente pues $x_p[3] = -(L - 1) = -3$, y entonces $7_{derecha} = 7 + x_p[3] = 4$; el punto de arriba del 7 será $7 + y_p[1] = 7 + L = 11$, como vemos en la figura.

Ahora para movernos hacia adelante en el eje Y ,

```
s[n+yp[y]]
```

9.4. Segmento de Código para el cálculo de la Energía

Ahora ya sabemos cómo dado un punto numerado como n y del que conocemos sus coordenadas x e y , encontrar sus vecinos hacia adelante. Dada una configuración, ya podemos calcular su Energía. Para ello es importante escribir el bucle para recorrer toda la Red, como un bucle doble en X e Y , de este modo conocemos en todo momento las coordenadas cartesianas de n . Para el cálculo de n basta irlo incrementando de uno en uno.

```
n=0; ener=0;
for(j=0;j<L;j++)
    for(i=0;i<L;i++)
    {
        ener+=s[n]*(s[n+xp[i]]+s[n+yp[j]]);
        n++;
    }
```

Nótese el orden de los bucles, el más interno, índice i , es el eje X y el más externo, índice j , el Y : debe ser así para respetar el orden de numeración de la red. Si se cambiara, sería incorrecto.

Veremos que para completar el algoritmo, necesitaremos movernos también hacia atrás, es decir, saber cual es el punto hacia la izquierda o hacia abajo de uno dado; procedemos de forma similar, recordando la numeración de los puntos y la figura 9.5,

- Dirección x **negativa**:

- si la coordenada x es igual a 0, $n - \hat{0} = n + (L - 1)$
- si la coordenada x es mayor que 0, $n - \hat{0} = n - 1$

- Dirección y **negativa**:

- si la coordenada y es igual a 0, $n - \hat{1} = n + L(L - 1)$
- si la coordenada y es mayor que 0, $n - \hat{1} = n - L$

Podemos implementar esto en el programa del siguiente modo; comenzemos con el eje x :

Definimos un vector que dependa de la coordenada x del punto, y que nos indique lo que debemos sumar al punto n para ir hacia atrás.

En concreto

```
for(i=1;i<L;i++)
    xm[i]=-1;
xm[0]=(L-1);
```

De este modo cuando queramos saber el spin hacia atrás en el eje X desde un punto n , con coordenada x , bastará hacer

```
s[n+xm[x]]
```

De forma similar para el eje Y ,

```
for(i=1;i<L;i++)
    ym[i]=-L;
ym[0]=L*(L-1);
```

Ahora para movernos hacia atrás en el eje Y ,

```
s[n+ym[y]]
```

La construcción de los vectores xp , yp , xm , ym debe hacerse en una función específica y una sola vez al inicio del programa, pues sus valores son fijos.

9.5. Configuración Inicial

Dado que el algoritmo de Metropolis que usaremos para generar la secuencia de configuraciones, se basa en calcular el cociente de probabilidades de dos puntos o configuraciones, necesitaremos crear una configuración de partida; esto es así pues el algoritmo comparte aspectos básicos con el ejercicio 5.7.1 en que generábamos distribuciones de probabilidad a base de pequeños cambios, siendo obligatorio empezar con un punto. Recordamos que aquí *punto* se refiere a un elemento del espacio de configuraciones, es decir a una *Configuración completa*: los V spines de cada punto de la red. Hay muchas alternativas para la elección de la primera configuración, pero aquí las reduciremos a tres.

- **Configuración random:** cada spin se elige al azar. Es una configuración con alta probabilidad a Temperatura infinita. Sea RAN() un generador de una variable continua con distribución plana en $[0,1)$.

```
for (i=0;i<V;i++)
    if(RAN()>0.5)
        s[i]=1;
    else
        s[i]=-1;
```

- **Configuración congelada:** todos los spins al mismo valor. Es la configuración dominante a Temperaturas muy bajas.

```
for (i=0;i<V;i++)
    s[i]=1; // Tambien podria ser -1
```

- **Partir de un backup:** Empezamos con una configuración calculada anteriormente y escrita en un fichero. Para leer el archivo, debemos saber el formato exacto de escritura. Supongamos que la hemos escrito en un fichero de texto

```
char s[V];
Fconfig=fopen("conf_ising.dat","wt");
for(i=0;i<V;i++)
    fprintf(Fconfig,"%c ",s[i]);
fclose(Fconfig);
```

Entonces para leerla el código sería,

```
char s[V];
Fconfig=fopen("config_ising.dat","rt");
for(i=0;i<V;i++)
fscanf(Fconfig,"%c ",&s[i]);
fclose(Fconfig);
```

Nota: Para archivos relativamente grandes, o que deben accederse velozmente, es recomendable utilizar ficheros binarios, lo que corresponde al siguiente código:

```
char s[V];
Fconfig=fopen("conf_ising.dat","wb");
fwrite(s,sizeof(s[0]),V,Fconfig);
fclose(Fconfig);
```

Entonces para leerla el código sería,

```
char s[V];
Fconfig=fopen("config_ising.dat","rb");
fread(s,sizeof(s[0]),V,Fconfig);
fclose(Fconfig);
```

9.6. Ejercicios

9.6.1. Generación de la Red y Cálculo de E y m

Escribir como primera parte del programa, uno que genere una configuración y a continuación calcule su energía y magnetización. Utilizar diferentes configuraciones. Utilizar algunas tales que pueda calcularse e y m exactamente, para comprobar que el programa lo hace correctamente. Usar diferentes tamaño de retículos. Deben definirse en primer lugar los direccionamientos para moverse en la Red y calcular los vecinos a un punto dado.

9.6.2. Manejo del Input/Output

El tipo de configuración a generar puede darse desde un *flag* en un archivo. Por ejemplo si el flag es 0, generamos una configuración random, si es 1 una configuración constante con todos los spines iguales a 1, y así para diferentes tipos de configuraciones. Añadir al programa anterior una función que lea dicho parámetro de un archivo. Escribir los resultados de la Energía y la magnetización en un archivo. Escribir la configuración en otro archivo. Uno de los valores del flag debe indicar que la configuración debe ser leída de un archivo (creado anteriormente). Escribir por tanto la función que lee dicho archivo y asigna esos valores como configuración inicial; es conveniente repasar la sección 9.5. Usar archivos binarios para optimizar la velocidad de lectura y escritura y el espacio de almacenamiento.

9.7. Problemas

9.7.1. Problema 1

Generar configuraciones iniciales Ferromagnéticas, Antiferromagneticas y Random. Calcular su Energia y Magnetización y comprobar que concuerdan con los cálculos analíticos.

Capítulo 10

Principios básicos de la Mecánica Estadística

Cuando tenemos un material, un gas o cualquier otro sistema a una temperatura dada y en equilibrio, queremos decir que el sistema en contacto con un baño térmico externo está en un estado que macroscópicamente no evoluciona. Es estable en el tiempo.

Sin embargo desde el punto de vista microscópico el sistema está pasando por miles de estados diferentes, en cada uno de los cuales las propiedades de las moléculas que lo componen son diferentes.

Pensemos en un Gas dentro de una habitación. Macroscópicamente tiene una Temperatura, Presión y Volumen dados, que describen el sistema. Estas cantidades, si el sistema está en equilibrio se mantienen constantes.

Si mirásemos con mucha más resolución veríamos que el gas está compuesto de moléculas, que chocan e interactúan entre sí y con las paredes, intercambiando energía, momento angular o excitaciones internas. Si hicieramos una foto del sistema, donde además de la posición quedaría reflejada la velocidad, modos internos, y todos los datos relevantes, esta foto instantánea correspondería a lo que antes hemos definido como una *configuración*. Una foto inmediatamente después nos daría una configuración completamente diferente. Un *macroestado*, es decir un estado de un sistema visto como un todo y mirando sus propiedades globales, es algo sólo aparentemente estable en el tiempo, está hecho de infinidad de *microestados* que cambian a lo largo del tiempo, cada uno de los cuales está caracterizado por propiedades microscópicas diferentes.

La pregunta básica es:

Dado un sistema del que conocemos su dinámica a escala microscópica, es decir sus configuraciones, y para cada una de ellas sabemos calcular su energía, ¿Cómo se comporta a escala macroscópica a una cierta Temperatura?

Podemos formularla de otra manera.

Dado un estado macroscópico a una cierta Temperatura, ¿cuales son los estados microscópicos que aparecen y con qué probabilidad aparece cada uno de ellos?

La respuesta a estas importantes preguntas las dió Boltzman, postulando

Dado un sistema con configuraciones C_α , y cada una de ella con Energía $E(C_\alpha)$, el estado macroscópico de equilibrio a una temperatura T es aquel donde aparecen todas las configuraciones posibles, cada una de ellas con probabilidad

$$p(C_\alpha) \propto e^{-\frac{1}{kT} E(C_\alpha)} \quad (10.1)$$

Aquí k es la Constante de Boltzman. Dado que el exponente no debe tener dimensiones, k tiene dimensiones de Energía/Temperatura, $[k] = J/\text{°K}$. Su valor es $k = 1.3805 \times 10^{-23} J K^{-1}$.

Cuando queremos reproducir en el ordenador los resultados de sistemas naturales, debemos prestar una gran atención a las dimensiones de las cantidades con las que trabajamos, de forma que todas ellas estén expresadas en las dimensiones correctas. Sin embargo, una vez introducidos los datos en el ordenador, trabajamos con números y obtenemos resultados numéricos a los que debemos volver a dar carácter dimensional de forma correcta. De hecho para simplificar los cálculos podemos trabajar suponiendo que las cantidades no tienen dimensiones, como es el caso en nuestro simple modelo de Ising, donde los spines, las energías, la temperatura o la constante de Boltzman, podemos suponerlos adimensionales. Procederemos de este modo en lo que sigue.

10.1. Función de Partición

Establecida la Ley de Boltzman, definimos la función de partición, Z , que nos dará todas las propiedades del sistema

$$Z = \int_C d\mu e^{-\frac{1}{kT} E(C_\alpha)} \quad (10.2)$$

La medida de integración $d\mu$ es la medida sobre el espacio de configuración.

Notar que Z es en realidad la constante de proporcionalidad para la normalización de la densidad de probabilidad.

En el caso del modelo de Ising, donde el espacio de configuración C es discreto, tendremos

$$Z = \sum_C e^{-\frac{1}{kT} E(C_\alpha)} = \sum_C e^{\frac{1}{kT} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \quad (10.3)$$

Usaremos la notación habitual,

$$\frac{1}{kT} = \beta \quad (10.4)$$

y por tanto

$$Z = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} = \sum_C e^{-\beta E(C_\alpha)} \quad (10.5)$$

La función de partición está construida con la distribución de probabilidad a partir de la cual calculamos todas las medias, dispersiones, etc. en el sistema, de forma equivalente a lo desarrollado en el capítulo 4 ; es decir, nuestro espacio de probabilidad es el espacio de configuraciones, y cada una de ellas aparece con una probabilidad

$$p(C_\alpha) = \frac{1}{Z} e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \quad (10.6)$$

Los valores medios de funciones de los spines, con esta distribución de probabilidad se calculan de acuerdo a la expresión 6.7.

En general, Z no es conocida, por lo que es más correcto escribir

$$p(C_\alpha) \propto e^{\beta \sum_{n,\hat{\mu}}^C s_n s_{n+\hat{\mu}}} \quad (10.7)$$

siendo esta cantidad simple de calcular para cada configuración.

En concreto, para una función cualquiera de los spines $\theta(\{s_n\})$, su valor medio viene dado por,

$$\langle \theta(\{s_n\}) \rangle = \frac{1}{Z} \sum_C \theta(\{s_n\}) e^{-\beta E(C)} \quad (10.8)$$

Casos particulares para la función θ son la Energía o la Magnetización.

10.1.1. Entropía

Hemos dicho que la probabilidad de que aparezca una configuración es proporcional al factor de Boltzman $e^{-\beta E(C_\alpha)}$. Pero hay que remarcar un importante matiz.

Dada una configuración su probabilidad depende de su Energía. Pero en general, en todos los sistemas hay muchas configuraciones diferentes que tienen la misma energía; llamemos $N(E)$ al número de configuraciones que tienen energía E .

Por ejemplo en el caso del modelo de Ising es trivial cambiar unos pocos spines que compensen sus cambios y la energía quede igual. O basta simplemente con cambiar el signo de todos los spines.

Entonces es incorrecto decir

La probabilidad de que aparezca una configuración con Energía E es proporcional a

$$e^{-\beta E(C_\alpha)}$$

Lo correcto es decir:

La probabilidad de que aparezca una configuración con Energía E es proporcional a

$$N(E)e^{-\beta E(C_\alpha)}$$

Este número $N(E)$ da cuenta de la entropía del Sistema.

Demostración:

Podemos hacer un cambio de variable usando la delta de Kronecker, definida del siguiente modo:

$$\delta(k) = \begin{cases} 1 & \text{Si } k = 0, \\ 0 & \text{Si } k \neq 0. \end{cases} \quad (10.9)$$

Considerando un valor entero $N \geq 0$, es simple demostrar la siguiente igualdad,

$$\sum_{n=0}^{\infty} \delta(n - N) = 1 \quad (10.10)$$

Introduciendo esto en la función de partición, obtenemos

$$Z = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \sum_E \delta(E - (-\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}})) \quad (10.11)$$

Debemos remarcar ahora que en un retículo finito, la función de partición es una suma finita de términos, en concreto 2^V sumandos, cada uno de los cuales es una función suave de β , o de forma más precisa, Z es una suma finita de términos analíticos, por lo cual la propia Z será analítica. Como tenemos un número finito de términos todos ellos bien comportados, podemos intercambiar el orden de los sumatorios sin modificar el resultado. Además al intercambiar este orden, podemos sustituir el sumatorio de spines $\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}$ por E ,

$$Z = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \sum_E \delta(E - (-\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}})) = \sum_E e^{-\beta E} \sum_C \delta(E - (-\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}})) \quad (10.12)$$

Ahora, dado que el sumatorio en C sólo afecta a la δ , que es la única función que depende de C , podemos efectuar dicho sumatorio, para cada valor de E . Ese sumatorio recorre todas las configuraciones del sistema, y cada vez que la configuración tiene energía E , suma 1. Así pues, al final, y dado un valor de E , este sumatorio vale el número de configuraciones del Espacio de configuraciones con Energía E , es decir,

$$\sum_C \delta(E - (-\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}})) = N(E) \quad (10.13)$$

Configuración	Energía
0	43
1	58
2	61
3	43
4	84
5	58
6	43
7	95
8	95
9	28

Cuadro 10.1: Ejemplo de un espacio de Configuración: a la izquierda un número identificativo de la configuración; a la derecha su Energía.

Entonces para la función de partición, tenemos finalmente

$$Z = \sum_E e^{-\beta E} N(E) \quad (10.14)$$

Notar que ahora el sumatorio es en Energías diferentes, no en Configuraciones; recorremos todas las energías posibles, no todas las configuraciones. A cambio, para cada energía, multiplicamos por el número de veces que aparece esa energía. Escrito del siguiente modo, definimos la entropía S (salvo constantes)

$$Z = \sum_E e^{-\beta E + \ln N(E)} = \sum_E e^{-\beta E + S} \quad (10.15)$$

de modo que con esta definición, la entropía es el logaritmo del número de estados con una energía dada. Si hay un único estado, la Entropía es cero, si existe una alta degeneración y hay muchas configuraciones con la misma energía, la entropía será alta.

En términos de probabilidad de encontrar una cierta energía, ahora tendremos

$$p(E) \propto N(E) e^{-\beta E} \quad (10.16)$$

Si conociésemos $N(E)$, la expresión anterior nos permitiría calcular $Z(\beta)$ pues la dependencia en β es trivial, y por tanto podríamos resolver exactamente el modelo. Sin embargo, en general, el cálculo de $N(E)$ es sumamente complejo. No obstante algunos desarrollos analíticos permiten estimaciones del número de estados y por consiguiente obtener resultados aproximados (Veáse la sección 12.5).

10.1.2. Un ejemplo

Para aclarar las ideas anteriores, consideremos un cierto espacio de configuración con 10 configuraciones, donde numeramos las mismas y calculamos para cada una de ellas su Energía, obteniendo los resultados que se muestran en la Tabla 10.1).

Ahora contamos, para cada Energía, cuantas configuraciones hay con ese valor; obtenemos así los resultados de la Tabla 10.2.

Si nos fijamos en las configuraciones individuales, con índice i , tendremos que por ejemplo,

$$p(i=0) = e^{-\beta 43}; p(i=1) = e^{-\beta 58}; \dots$$

Si nos fijamos en la Energía, tendremos

$$p(E=43) = 3e^{-\beta 43}; p(E=58) = 2e^{-\beta 58}; \dots$$

E	N(E)
28	1
43	3
58	2
61	1
84	1
95	2

Cuadro 10.2: Número de estados de Energía E en el espacio de configuración de la Tabla 10.1

10.2. Simetrías

Dentro de las 2^V configuraciones, dada una configuración cualquiera, existe otra obtenida de la anterior cambiando el signo de todos los spines. El Espacio de Configuraciones es simétrico; o lo que es lo mismo, en el sumatorio de Z recorremos una configuración y su opuesta.

Por otra parte, igual de importante, el integrando, en concreto la Energía es invariante bajo este cambio: en efecto, haciendo un cambio global de todos los spines,

$$\{s_n\} \rightarrow \{-s_n\} \forall n \quad (10.17)$$

la Energía se convierte en

$$E = - \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \rightarrow - \sum_{n,\hat{\mu}} (-s_n)(-s_{n+\hat{\mu}}) = E \quad (10.18)$$

Z es pues invariante bajo el cambio global de signo. Esto es llamado una *simetría global* bajo transformaciones de signo, o simetría \mathbb{Z}_2 global.

10.3. Valores esperados

Dado un operador $\theta(\{s_n\})$, hemos definido en 10.8 como calcular sus valores medios, o valores esperados.

Definiremos a continuación los observables más básicos, que nos permitirán caracterizar las propiedades de nuestro modelo y como cambian con la Temperatura.

10.3.1. Energía y Calor Específico

La cantidad elemental es el valor medio de la *energía intensiva*:

$$\langle e \rangle = \frac{1}{Z} \sum_C \left(\frac{-1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \right) e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \quad (10.19)$$

que un sencillo cálculo muestra que puede ser escrita como

$$\langle e \rangle = \frac{-1}{2V} \frac{\partial \ln(Z)}{\partial \beta} \quad (10.20)$$

Asociada a la Energía, tenemos el *Calor Específico* (salvo constantes), definido como

$$C_v = \frac{-\partial \langle e \rangle}{\partial \beta} = \frac{1}{2V} \frac{\partial^2 \ln(Z)}{\partial \beta^2} \quad (10.21)$$

Veamos a continuación como C_v puede escribirse en función de observables sencillos. Sustituyendo $\langle e \rangle$ por su valor en 10.19,

$$C_v = -\frac{\partial}{\partial \beta} \left(\frac{1}{Z} \sum_C \left(\frac{-1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \right) e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \right) \quad (10.22)$$

La dependencia en β está en Z y en el interior del sumatorio. Aplicamos la regla elemental para la derivada de un producto de funciones, y obtenemos

$$C_v = \frac{1}{2V} \left(\left(\frac{\partial}{\partial \beta} \frac{1}{Z} \right) \left(\sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E} \right) + \frac{1}{Z} \frac{\partial}{\partial \beta} \left(\sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E} \right) \right) \quad (10.23)$$

En el primer término de la ecuación, calculamos

$$\frac{\partial}{\partial \beta} \frac{1}{Z} = -\frac{1}{Z^2} \frac{\partial Z}{\partial \beta} = -\frac{1}{Z^2} \sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E}. \quad (10.24)$$

Para el segundo término, tendremos

$$\frac{\partial}{\partial \beta} \left(\sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E} \right) = \sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) \frac{\partial}{\partial \beta} e^{\beta E} = \sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E} \quad (10.25)$$

Introduciendo estos cálculos parciales en la expresión 10.23, tenemos

$$C_v = \frac{1}{2V} \left(-\frac{1}{Z^2} \left(\sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E} \right)^2 + \frac{1}{Z} \sum_C (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) (\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}) e^{\beta E} \right) \quad (10.26)$$

Podemos reescribir la expresión anterior en función de valores medios sencillos, obteniendo

$$C_v = 2V(\langle e^2 \rangle - \langle e \rangle^2) \quad (10.27)$$

Notar las constantes de normalización: en el cálculo de e incluimos un factor $\frac{1}{2V}$ para hacerla intensiva, pues es la suma de V números. e ya es una cantidad intensiva, y por tanto también lo será su derivada con respecto a β .

Es al derivar con respecto a β cuando aparecen las constantes indicadas, que provienen de que lo que multiplica a β en Z es extensivo. C_v continua siendo una cantidad intensiva, aunque aparezca V en 10.27.

10.3.2. Magnetización y Susceptibilidad

La energía es un observable invariante bajo la simetría global de cambio de signo. Por lo tanto no es un buen indicador para monitorizar si el sistema mantiene o no dicha simetría. Introduciremos ahora un observable que no es invariante, y por tanto nos dará información adicional sobre las propiedades dinámicas del sistema. El observable más simple con esta característica es la magnetización. Definimos el observable

$$m = \frac{1}{V} \sum_n s_n \quad (10.28)$$

y su valor medio

$$\langle m \rangle = \frac{1}{Z} \sum_C \left(\frac{1}{V} \sum_n s_n \right) e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \quad (10.29)$$

Esta cantidad puede ser expresada como una derivada si añadimos un término de interacción con un campo magnético externo; este término tiene el doble interés de que es necesario en muchos desarrollos teóricos, y que además se corresponde con la situación real cuando sometemos el material a un campo magnético externo de valor h (salvo constantes). La función de partición viene dada por

$$Z = \sum_C e^{\beta \sum_{n,\mu} s_n s_{n+\mu} + h \sum_n s_n} \quad (10.30)$$

entonces, tendremos que

$$\langle m \rangle = \frac{1}{V} \left. \frac{\partial \ln(Z)}{\partial h} \right|_{h=0} \quad (10.31)$$

De forma similar al calor específico, definimos ahora la *susceptibilidad magnética*,

$$\chi = \left. \frac{\partial \langle m \rangle}{\partial h} \right|_{h=0} = \frac{1}{V} \left. \frac{\partial^2 \ln(Z)}{\partial h^2} \right|_{h=0} \quad (10.32)$$

y ahora de nuevo puede demostrarse la igualdad

$$\chi = \frac{1}{V} \left. \frac{\partial^2 \ln(Z)}{\partial h^2} \right|_{h=0} = V(\langle m^2 \rangle - \langle m \rangle^2) \quad (10.33)$$

10.4. Sistemas finitos

En un retículo finito de lado L , todas las funciones con las que trabajamos son analíticas, los sumatorios finitos para Z , y por tanto todos los observables definidos anteriormente serán funciones suaves de β .

En concreto es importante remarcar que la simetría global hace que el valor medio de la magnetización, o de cualquier potencia impar de m , sea nulo estrictamente, pues en el sumatorio, por cada configuración que interviene contribuyendo con un valor, habrá otra igual de probable, con todos los spines cambiados, que contribuirá con signo opuesto. Por tanto,

$$\langle m \rangle = \langle m^3 \rangle = \dots = 0 \quad (10.34)$$

y en general para cualquier función de los espines, $g(s)$, que sea impar bajo la simetría de cambio de signo global, es decir, si dada una configuración C_α , llamamos $C_{-\alpha}$ a la configuración obtenida de la anterior cambiando el signo de todos los espines, una función impar es aquella que cumple,

$$g(s)_{s \in C_{-\alpha}} = -g(s)_{s \in C_\alpha} \quad (10.35)$$

como además tenemos que

$$p(C_\alpha) = p(C_{-\alpha}) \quad (10.36)$$

entonces para cualquier L finito,

$$\langle g(s) \rangle = 0 \quad (10.37)$$

Dado que en el sistema termodinámico (en el límite $L \rightarrow \infty$) para $T \rightarrow 0$ existe magnetización, parecería que con este modelo no podremos describir en L finitos el comportamiento correcto del sistema. Sin embargo veremos más adelante como es posible introducir observables que sí nos indiquen en una red finita lo que ocurre en una red infinita, marcando las regiones donde existe o no magnetización.

10.5. Evolución del Sistema con la Temperatura

10.5.1. Alta Temperatura

Cuando la Temperatura del sistema es muy alta, β es muy baja. Supongamos el caso en que sea cero. Entonces

$$p(C_\alpha) \propto e^{\beta \sum_{n,\mu}^{C_\alpha} s_n s_{n+\mu}} \propto 1 \quad (10.38)$$

y todas las configuraciones son igualmente probables, y un sistema macroscópico recorre todas ellas uniformemente.

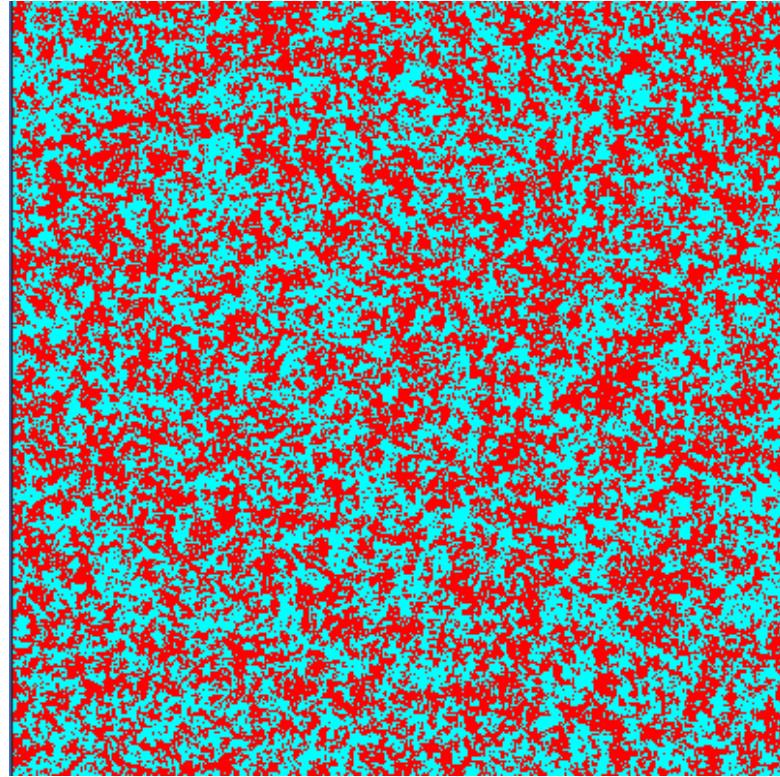


Figura 10.1: Configuración típica a altas temperaturas. Los dos colores representan los dos posibles valores del spin.

En términos de la Energía

$$p(E) \propto N(E)e^{-\beta E} \propto N(E) \quad (10.39)$$

Dado que existen muchas más configuraciones desordenadas (Veáse la figura 10.1) que ordenadas, o dicho de otra manera, las Energías con entropía alta tienen también energía en la zona central, es decir con $E \approx 0$ según nuestra definición, casi todas las configuraciones que encontraremos con probabilidad significativa serán de energía nula. Recordar que $E \approx 0$ es precisamente la zona central de la Energía, que puede variar en el intervalo $[-2V, 2V]$.

Remarquemos primero que en este límite Z puede ser calculado trivialmente

$$Z = \sum_C 1 = 2^V \quad (10.40)$$

Para la magnetización, dado que todos los spines están al azar, tendremos que

$$\langle m \rangle = 0 \quad (10.41)$$

y el valor medio de la energía intensiva será

$$\langle e \rangle = -\frac{1}{Z} \sum_C \left(\frac{1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \right) = 0 \quad (10.42)$$

pues la suma es de números al azar.

Podemos calcular también el calor específico, usando el Teorema del Límite Central. Para casi todas las configuraciones, la suma

$$\frac{1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \quad (10.43)$$

es una suma de $2V$ términos, cada uno de los cuales es 1 o -1 con igual probabilidad. El TLC nos dice que

$$\langle e^2 \rangle = \langle \left(\frac{1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \right)^2 \rangle \propto \left(\frac{1}{2V} \sqrt{V} \right)^2 \propto \left(\frac{1}{\sqrt{V}} \right)^2 \quad (10.44)$$

Por tanto

$$C_v = 2V(\langle e^2 \rangle - \langle e \rangle^2) \propto 1 \quad (10.45)$$

Esto significa que en $T \rightarrow \infty$, C_v es una constante; a pesar de que tiene un factor V multiplicativo, la cantidad no crece con V . Lo mismo ocurre con χ .

En esta situación de $T = \infty$, $\beta = 0$, el modelo es trivial, así como Z , como hemos visto anteriormente. Es posible incluso calcular C_v de forma exacta; consideremos una red de volumen V finito. Ya sabemos que $\langle e \rangle$ es cero dado que sumamos a todas las configuraciones con idéntico peso, es decir con tantas energías positivas como negativas. El término $\langle e^2 \rangle$ es definido positivo, y por tanto no nulo. Calculemos su valor de forma exacta,

$$\langle e^2 \rangle = \frac{1}{Z} \sum_C \left(\frac{1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \right)^2 \quad (10.46)$$

Calculemos para una configuración genérica el valor del interior del sumatorio,

$$e_C^2 = \left(\frac{1}{2V} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \right)^2 = \frac{1}{4V^2} \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} \sum_{m,\hat{\nu}} s_m s_{m+\hat{\nu}} = \frac{1}{4V^2} \sum_{n,\hat{\mu}} \sum_{m,\hat{\nu}} s_n s_{n+\hat{\mu}} s_m s_{m+\hat{\nu}} \quad (10.47)$$

Este sumatorio podemos ahora agruparlo en dos partes: por un lado los términos con $n = m, \hat{\mu} = \hat{\nu}$ (es decir, puntos iguales) y por otro el resto, es decir:

$$e_C^2 = \frac{1}{4V^2} \sum_{n,\hat{\mu}} \sum_{m,\hat{\nu}} s_n s_{n+\hat{\mu}} s_m s_{m+\hat{\nu}} = \frac{1}{4V^2} \left(\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} s_n s_{n+\hat{\mu}} + \sum_{n,m(n \neq m),\hat{\mu},\hat{\nu}} s_n s_{n+\hat{\mu}} s_m s_{m+\hat{\nu}} \right) \quad (10.48)$$

Dado que $s_n s_{n+\hat{\mu}} s_n s_{n+\hat{\mu}} = 1$ siempre, tendremos

$$e_C^2 = \frac{1}{4V^2} \left(2V + \sum_{n,\hat{\mu}} \sum_{m \neq n, \hat{\nu} \neq \hat{\mu}} s_n s_{n+\hat{\mu}} s_m s_{m+\hat{\nu}} \right) \quad (10.49)$$

Esto es cierto para cualquier configuración. Al promediar ahora sobre todas ellas, el segundo término será cero pues todos los spins y energías positivas o negativas tienen la misma probabilidad, por tanto

$$\langle e^2 \rangle = \frac{1}{Z} \sum_C \left(\frac{1}{4V^2} \left(2V + \sum_{n,\hat{\mu}} \sum_{m \neq n, \hat{\nu} \neq \hat{\mu}} s_n s_{n+\hat{\mu}} s_m s_{m+\hat{\nu}} \right) \right) = \frac{1}{Z} \sum_C \left(\frac{1}{4V^2} (2V + 0) \right) \quad (10.50)$$

tenemos pues

$$\langle e^2 \rangle = \frac{1}{Z} \sum_C \left(\frac{1}{2V} \right) \quad (10.51)$$

al sumar sobre todas las configuraciones tendremos

$$\sum_C \left(\frac{1}{2V} \right) = 2^V \frac{1}{2V} \quad (10.52)$$

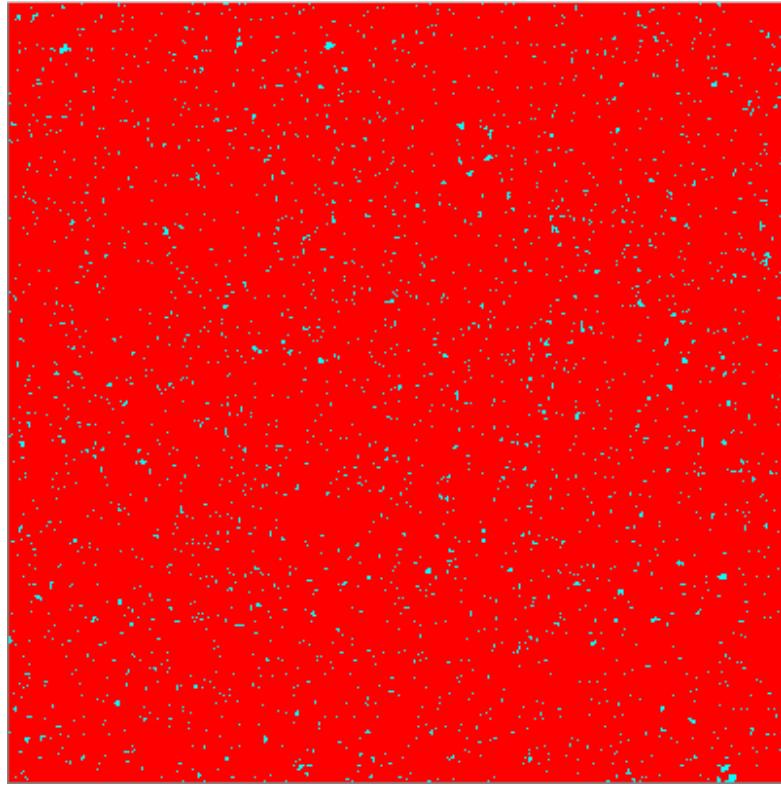


Figura 10.2: Configuración típica a bajas temperaturas

y recordando el valor de Z en 10.40, finalmente

$$\langle e^2 \rangle = \frac{1}{2V} \quad (10.53)$$

que coincide con 10.44.

Este tratamiento es la parte más simple de lo conocido como *desarrollo de alta temperatura*, que consiste en desarrollar Z en potencias de β ,

$$Z = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} = \sum_C \left(1 + \beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} + \frac{\beta^2}{2} \sum_{n,\hat{\mu}} \sum_{m,\hat{\nu}} s_n s_{n+\hat{\mu}} s_m s_{m+\hat{\nu}} + \dots \right) \quad (10.54)$$

El término constante, corresponde al desarrollo anterior con $\beta = 0$; considerando términos lineales, cuadráticos, etc. en β se puede ir aproximando el cálculo a los resultados correctos para $\beta \approx 0$.

10.5.2. Baja Temperatura

Pasemos ahora a considerar el caso donde la temperatura es muy baja, es decir β muy alta. Veamos que en este caso las configuración con energía mínima, máxima en valor absoluto, tienen una probabilidad muy alta, siendo las demás despreciables.

La configuración de energía menor se consigue con todos los spines apuntando en la misma dirección, pues en este caso para el exponente de Z tendremos,

$$C_0 = \{\sigma_n = 1 \forall n\} \Rightarrow E = - \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}} = -2V \quad (10.55)$$

La configuración con un spin cambiado (por ejemplo el punto $n = 0$), tendrá una energía

$$C_1 = \{\sigma_0 = -1, \sigma_n = 1 \forall n \neq 0\} \Rightarrow E = - \sum_{n,\hat{n}} s_n s_{n+\hat{n}} = -(2V - 8) \quad (10.56)$$

debido a que 4 links (dos hacia delante y dos hacia atrás) que contaban como +1 cada una en el sumatorio, ahora cuentan como -1. Es importante resaltar que el número de configuraciones del tipo C_0 es solamente 1, mientras que del tipo C_1 tendremos tantas como puntos a elegir para voltear un spin, es decir, V configuraciones. Este factor da cuenta de la Entropía diferente, que debe ser tenida en cuenta. No sabemos la probabilidad absoluta de cada una de las anteriores configuraciones, pero si que podemos calcular la probabilidad relativa entre ellas; aplicando la expresión 10.16,

$$\frac{p(C_1)}{p(C_0)} = \frac{Ve^{\beta(2V-8)}}{e^{\beta 2V}} = Ve^{-\beta 8} \quad (10.57)$$

Para valores grandes de β , la probabilidad de la configuración C_1 es absolutamente despreciable; por tanto en esta primera aproximación, podemos suponer que la única configuración presente para los promedios estadísticos es C_0 . Notar que el factor multiplicativo V en la expresión anterior proviene de la Entropía de la configuración con un spin volteado, Entropía que tiene un valor de $\ln V$.

Por tanto para calcular valores medios, simplemente consideramos el valor del observable en la configuración C_0 .

Para la energía intensiva tendremos

$$\langle e \rangle \approx -\left\langle \frac{1}{2V} \sum_{n,\hat{n}} s_n s_{n+\hat{n}} \right\rangle = -\left(\frac{1}{2V} 2V\right) = -1 \quad (10.58)$$

Para C_v , al ser las configuraciones dominantes constantes, no habrá fluctuaciones y tendremos que

$$\langle e^2 \rangle = \langle e \rangle^2 \quad (10.59)$$

con lo cual para $T \rightarrow 0$, tenemos que $C_v \rightarrow 0$.

La configuración dominante es pues aquella con todos los spines alineados, supongamos que en la dirección positiva, para la cual la magnetización vale

$$\langle m \rangle = \frac{1}{Z} \sum_C \left(\frac{1}{V} \sum_n s_n \right) e^{\beta \sum_{n,\hat{n}} s_n s_{n+\hat{n}}} = \frac{1}{V} \sum_n^{C_\alpha=\{1\}} s_n = 1 \quad (10.60)$$

y al no haber fluctuaciones de spines, $\chi \rightarrow 0$.

En un retículo finito, como hemos dicho antes, el razonamiento que conduce a 10.60 es incorrecto: en el sumatorio a configuraciones, aparecen todas ellas, y en concreto, por cada una existe otra justo opuesta, con todos los spines cambiados, de modo que esto implica que si consideramos la Configuración C_0 , con todos los spines hacia arriba, la configuración opuesta, C_{-0} con todos los spines hacia abajo, está presente y con la misma probabilidad, es decir $p(C_0) = p(C_{-0})$ contribuyendo ambas con signo opuesto a la magnetización, con lo cual $\langle m \rangle = 0$. Pero en un retículo infinito la situación, como veremos más adelante, es bien diferente y en ese caso, la expresión 10.60 es correcta.

En lenguaje matemático, cuando L es finito Z es analítica, tanto la suma sobre configuraciones como la suma para calcular la energía E de cada configuración (en el exponente) están bien definidas, la simetría global es siempre exacta y por tanto $\langle m \rangle = 0$. Cuando L es infinito, las sumas (en configuraciones y en spines) se convierten en series, divergiendo incluso el valor de E en la mayoría de las configuraciones. Entonces ya no podemos afirmar nada acerca de la analiticidad de Z o argumentar que se conservará la simetría.

En lenguaje mas fisico, en una red finita si el sistema al bajar la temperatura elige el vacío (mínimo local de la Energía) con todos los spines +1, y las configuraciones próximas a este

mínimo local, por ejemplo las que tienen unos pocos spines iguales a -1 , tienen una energía mucho mayor; de modo que el mínimo local elegido está rodeado de puntos muy energéticos, que debería atravesar para poder llegar al otro mínimo local, el que tiene todos los spines -1 ; este otro mínimo local es simétrico del anterior, es decir, ambos tienen igual energía E , pero la barrera de potencial a atravesar para llegar de uno a otro crece exponencialmente con L .

Para L finito, con tiempo suficiente el sistema puede evolucionar de uno a otro, y por tanto se recupera la simetría y $\langle m \rangle = 0$. Pero con $L = \infty$, la barrera de potencial se vuelve infinita, y para T baja no puede ser atravesada por el sistema, que se queda anclado en uno de ellos. Si el vacío elegido es $s_n = 1$ tendremos $\langle m \rangle = 1$. Si el vacío es $s_n = -1$ tendremos $\langle m \rangle = -1$

Sin embargo existen modelos donde incluso en el límite termodinámico, el sistema en su dinámica, aún a temperaturas muy bajas, podría salir de la configuración semi-congelada y viajar hasta justo la opuesta, con lo que la suma anterior daría cero; esta escapatoria se produce por motivos dinámicos: debido a las características del modelo, el sistema puede viajar lentamente entre todas las configuraciones y seguir recorriendo todo el espacio fase, sin encontrar obstrucciones. De hecho esto es lo que ocurre cuando consideramos otros modelos como:

- El modelo de Ising en dimensión $d = 1$.
- El modelo XY en $d = 2$: similar al modelo de Ising, pero las variables son vectores de dos componentes de módulo 1 .
- El modelo de Ising en $d=2$ con acoplamientos random (Modelo *Spin Glass*).

Pero en Ising en $d = 2$ el sistema no es capaz de salir del mínimo para recorrer otros puntos del espacio fase. Esto es lo que se llama *ruptura espontánea de simetría*, e implica una no analiticidad en los observables asociados, como $\langle m \rangle$ en este caso. La no analiticidad sólo puede darse en el caso de Volumen infinito.

La región de T alta o β baja diremos que es la *fase desordenada* o de alta temperatura. La de T baja o β alta, la *fase ordenada*, fase de simetría rota o baja temperatura.

10.5.3. Temperaturas Intermedias

Tanto a Alta como a Baja Temperatura el modelo de Ising es relativamente trivial, en el sentido de que con cálculos simples, aproximados, puede describirse bien el modelo. Sin embargo existe una región intermedia, donde estos análisis no dan resultados correctos, y es precisamente ahí donde se presenta toda la riqueza y no trivialidad del modelo.

Resolver esta región ha sido un gran desafío para la Física y las Matemáticas. La introducción a raíz de este simple modelo, de conceptos como el Grupo de Renormalización y la Universalidad por L.P. Kadanoff y K.G. Wilson , entre otros permitió grandes avances tanto en la Mecánica Estadística como en la Teoría Cuántica de Campos y en el estudio de las Partículas Elementales.

Las configuraciones típicas son complejas, presentando clusters o dominios de spines de diferentes tamaños (invariancia de Escala). En la figura 10.3 puede verse una configuración con alta probabilidad en la región de Temperaturas intermedias.

10.6. Histogramas y Evolución temporal

10.6.1. Histograma de la Energía

Para cualquier valor de la temperatura una cuestión importante es estudiar la distribución de energía, es decir su histograma, que dependerá del tamaño de la red y de la temperatura. Este histograma representa precisamente la distribución de probabilidad $p(E)$ dada en 10.16.

La forma típica de los histogramas de Energía es similar a una Gaussiana. En la figura 10.4 puede verse un histograma para la Energía en la zona de temperaturas intermedias, calculada en una red con $L = 128$.

La desviación estándar de $\langle e \rangle$ vale (dejemos el punto critico aparte)

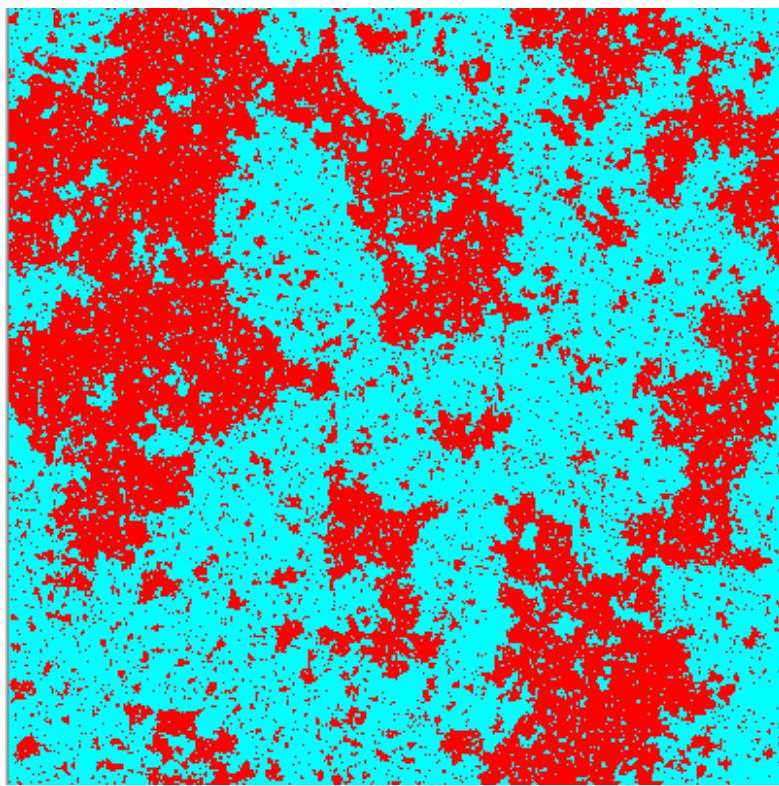


Figura 10.3: Configuración típica en temperaturas intermedias.

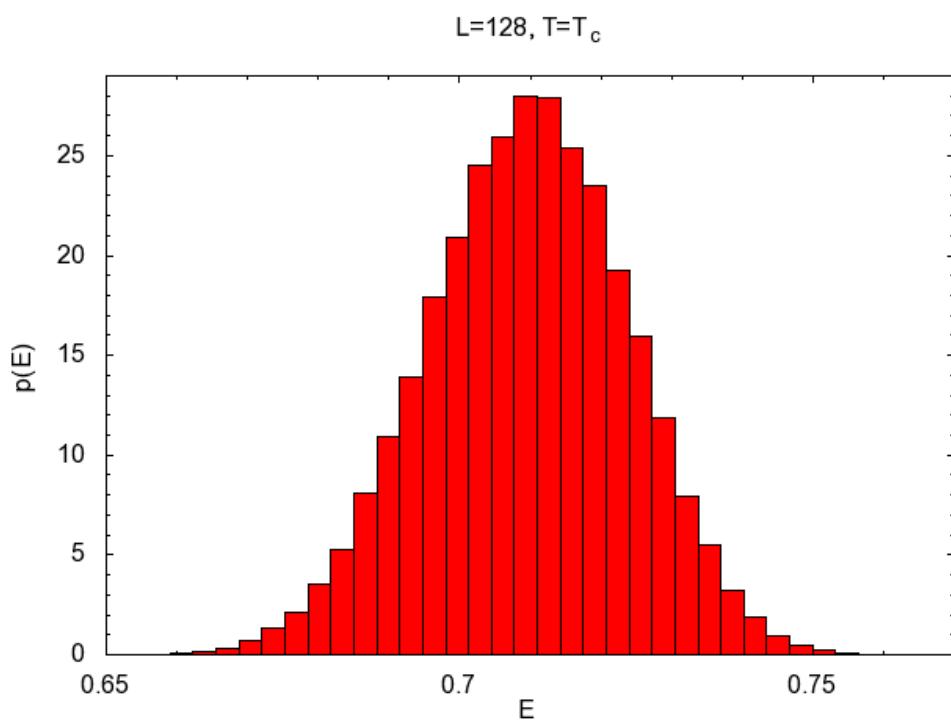


Figura 10.4: Histograma de la energía en β_c en un retículo con $L = 128$.

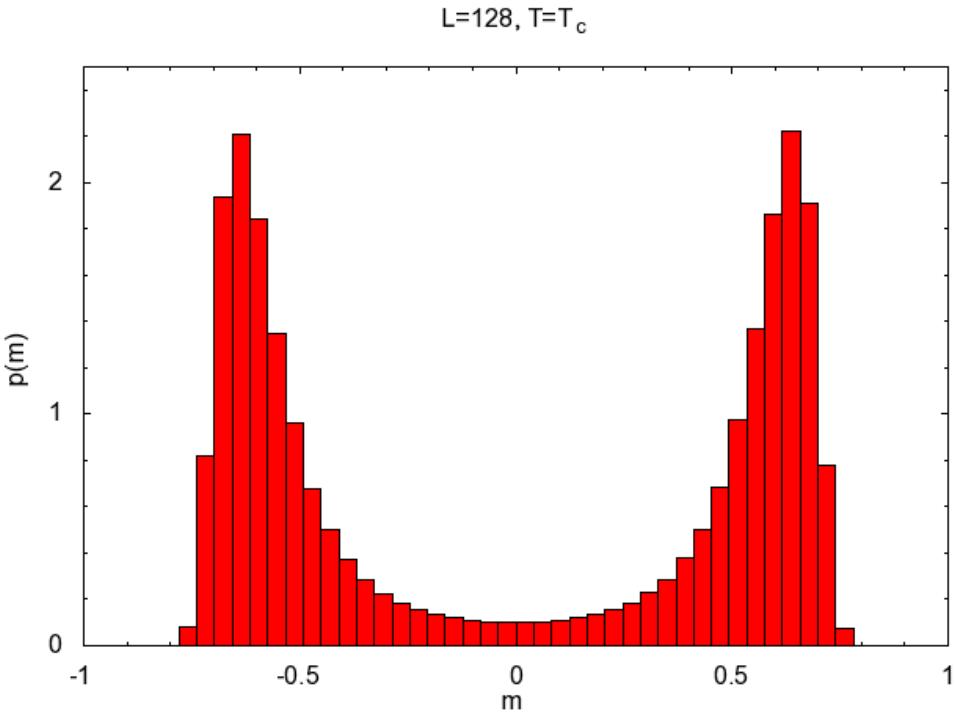


Figura 10.5: Histograma de la magnetización en β_c en un retículo con $L = 128$.

$$\sigma = \sqrt{\langle e^2 \rangle - \langle e \rangle^2} \propto \frac{1}{\sqrt{V}} \quad (10.61)$$

es decir, en retículos grandes, para casi todas las configuraciones con probabilidad significativa, su energía está muy cerca de la media, y en el límite $L \rightarrow \infty$, la distribución gaussiana tiende a una delta. Es lo observado en la Naturaleza, donde a pesar de que un sistema macroscópico está hecho de muchos estados diferentes, no se aprecian fluctuaciones en las propiedades termodinámicas.

10.6.2. Histograma de la magnetización

El histograma de la magnetización nos da información valiosa sobre el comportamiento del sistema. En la fase desordenada, la magnetización media es cero, y su distribución será una gaussiana centrada en el cero. Por el contrario en la fase rota, tendremos que las configuraciones están aproximadamente la mitad del tiempo en configuraciones tales que en ellas $m \approx -1$ y la otra mitad del tiempo en configuraciones con $m \approx 1$, con unas pocas configuraciones en regiones intermedias, con magnetización también intermedia, es decir $m \approx 0$. Un histograma muestra una estructura de doble pico simétrico en esta situación, como puede verse en la figura 10.5.

10.6.3. Evolución de la Energía y la Magnetización

Es significativo también observar la evolución de las medidas individuales de la magnetización y de la Energía.

Comencemos con la magnetización.

Si nos situamos en la fase desordenada, veremos valores pequeños con oscilaciones frecuentes en torno al 0.

Cuando estamos en la fase rota, veremos como el sistema permanece durante un largo número de iteraciones en el entorno de valores próximos a 1 para saltar de forma relativamente rápida

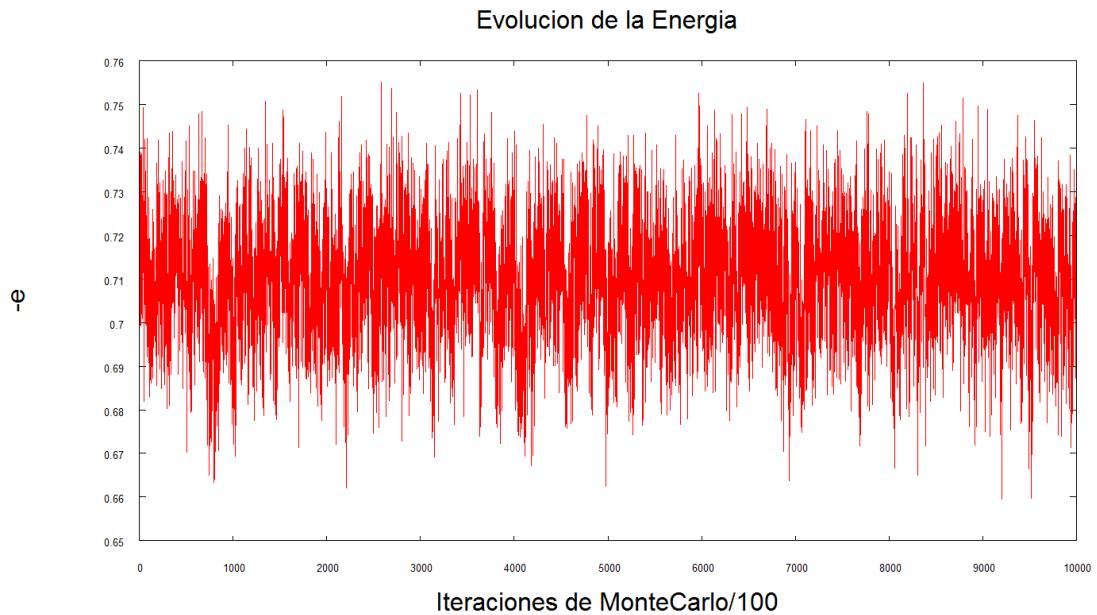


Figura 10.6: Evolución de la energía intensiva (cambiada de signo) en β_c en un retículo con $L = 128$. Se dibuja la energía cada 100 iteraciones.

a valores próximos a 1. Llamaremos a este proceso *flip-flop*, pues indica una cambio entre las dos fases simétricas del modelo, una con casi todos los spins *up*, y otra con casi todos *down*. La energía será similar en ambas fases. Sólo en la zona donde se produce el flip-flop, la energía sera ligeramente más alta, al atravesar el sistema la zona con mayores inhomogeneidades, mayor desorden y por tanto valores más altos de E .

En la figura 10.6 podemos ver la evolución de la Energía con los parámetros anteriores. En la figura 10.7 puede verse la evolución de la magnetización. Vemos que la magnetización pasa la mayoría del tiempo en estados con magnetización alta en valor absoluto, correspondiendo a las proximidades de los dos mínimos locales simétricos, produciéndose saltos entre ellos, para lo cual tiene que atravesar una barrera de potencial, lo que implica que tiene energías mayores.

10.7. Límite Termodinámico

La función de partición es una función analítica de β y h para cualquier retículo finito. Esto implica que todo será continuo, y en concreto nunca aparecerá magnetización. No describe pues correctamente el modelo físico inicial.

Sin embargo, si consideramos el límite $L \rightarrow \infty$, las sumas se convierten en series y la función de partición o alguna de sus derivadas puede dejar de ser analítica.

Esto es exactamente lo que ocurre en el modelo de Ising en $d = 2$ o $d = 3$, no así en $d = 1$, por ejemplo.

En los ordenadores sólo podremos simular sistemas con L finito, y por tanto todo será siempre analítico. Deberemos definir observables y técnicas precisas para poder averiguar qué ocurre en el límite $L \rightarrow \infty$. Estas técnicas se denominan de *Escalado de tamaño finito*, o *Finite Size Scaling*; nos permiten a partir de datos en retículos finitos, extrapolar a tamaño infinito y ver si allí aparecen no analiticidades.

10.7.1. Ruptura de Simetría, no analiticidad y magnetización espontánea

Ya hemos visto que para un L finito tendremos siempre que $\langle m \rangle = 0$.

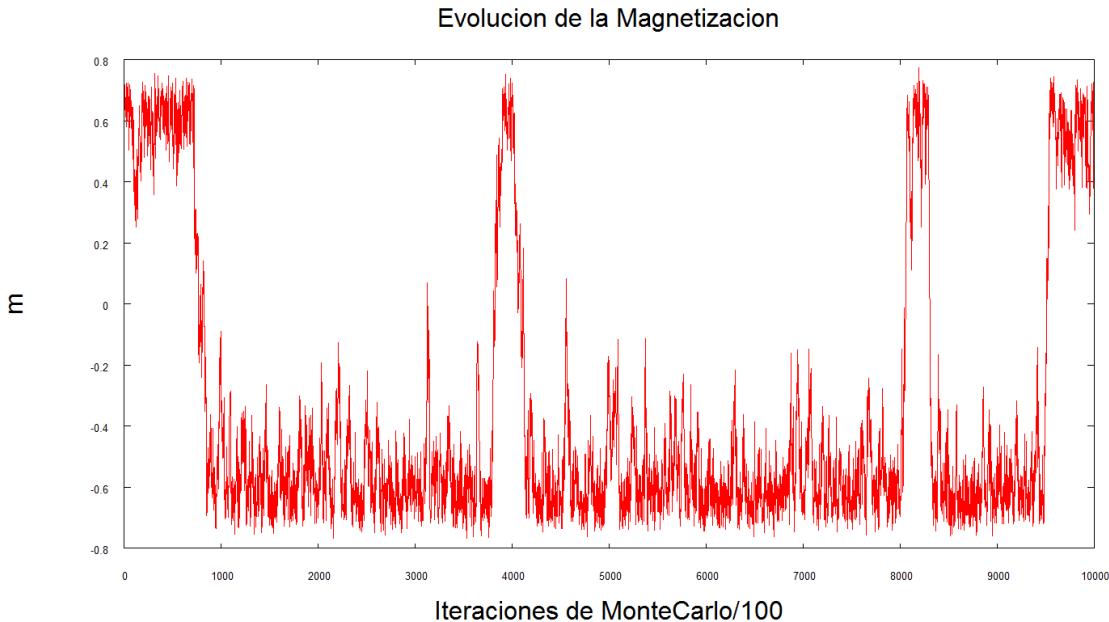


Figura 10.7: Evolución de la magnetización en β_c en un retículo con $L = 128$. Se dibuja la magnetización cada 100 iteraciones.

Por tanto para cualquier valor de L finito, si dibujamos $\langle m \rangle$ en función de T , obtendríamos cero en todo el rango

Cuando vamos al límite termodinámico ($L \rightarrow \infty$), ocurre que al ir bajando la Temperatura, al sistema le cuesta más tiempo pasar de una configuración a la opuesta. Llega un momento que cuando el sistema cae en configuraciones donde casi todos los spines están por ejemplo hacia arriba, ya no puede moverse a configuraciones hacia abajo. En ese momento tendremos que $\langle m \rangle \neq 0$. Al dibujar $\langle m \rangle$ frente a T aparecen dos regiones: para Temperaturas altas, la magnetización es cero. Esto ocurre en una región amplia. Si $\langle m \rangle$ fuera analítica, debería ser cero para todo valor de la Temperatura¹. Pero para valores por debajo de una cierta temperatura, la Temperatura crítica, T_c , es diferente de cero. Por tanto la magnetización no es una función analítica.

El punto que separa ambas fases, donde se presenta la no analiticidad, diremos que es el punto crítico, β_c o T_c .

El modelo de Ising en $d = 2$ puede ser resuelto exactamente. Para el valor de la Temperatura crítica, se obtiene

$$\beta_c = \frac{1}{2} \ln(1 + \sqrt{2}) = 0.440686793509771\dots \quad (10.62)$$

La magnetización para $\beta < \beta_c$ es cero, y para $\beta > \beta_c$, vale

$$\langle m \rangle = \left(1 - \frac{1}{\sinh^4(2\beta)} \right)^{1/8} \quad (10.63)$$

Esta curva para $\langle m \rangle$ tiene una forma omnipresente en Física. Conviene dedicar un poco de tiempo a dibujarla explicitamente.

Por ejemplo usando `gnuplot`, deberemos introducir los siguientes comandos

```
b_critico=0.5*log(1+sqrt(2.0))
f(x)=x<b_critico?0:(1.0-1.0/sinh(2*x)**4)**(1.0/8)
p [0:1] [-0.1:] f(x)
```

¹Dada una función analítica, si es constante en un intervalo $[a, b]$, todas sus derivadas serán nulas en un punto del interior, y por tanto si hacemos el desarrollo de Taylor, la función seguirá valiendo lo mismo en cualquier valor fuera del intervalo

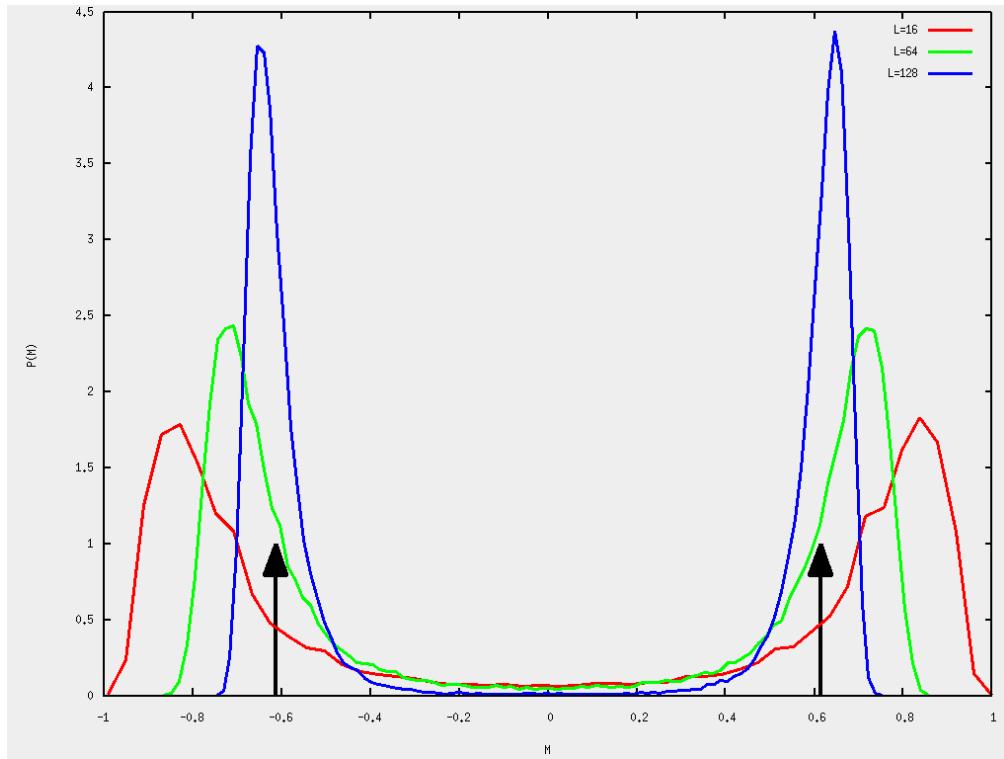


Figura 10.8: Histograma de la magnetización, $P(m)$ por debajo de la Temperatura crítica (fase de simetría rota), para diferentes tamaños del retículo. Las flechas indican el valor exacto en el límite termodinámico.

10.7.2. La magnetización en un retículo finito

En un retículo finito, si recorremos todo el espacio de configuraciones, tendremos $\langle m \rangle = 0$. Como esto ocurre para todo L , este no es un buen observable para estudiar el límite termodinámico. La opción habitual es considerar el operador

$$\langle |m| \rangle = \left\langle \frac{1}{V} \left| \sum_V s_n \right| \right\rangle \quad (10.64)$$

Esta cantidad es invariante bajo la simetría global Z_2 , y por tanto ya no es obligatoriamente cero, de hecho es una cantidad definida positiva. Sin embargo podemos estimar su comportamiento en ambas fases del modelo.

En la fase de alta temperatura, los spines están distribuidos aleatoriamente. Sabemos que si sumamos V números cada uno de los cuales es $+1$ o -1 al azar, su suma se separará de la media (que en ese caso es cero) como la raíz del número de sumandos, es decir tendremos que

$$\left| \sum_V s_n \right| \propto \sqrt{V} \quad (10.65)$$

y por tanto

$$\langle |m| \rangle \propto \frac{1}{V} \sqrt{V} \propto \frac{1}{\sqrt{V}} \quad (10.66)$$

de modo que en esta fase, si bien es una cantidad positiva, tiende a cero al aumentar el tamaño de la red, y en el límite termodinámico recuperamos el resultado como si no estuviera el módulo.

Nota: la cantidad definida es $|\sum_V s_n|$ que no tiene nada que ver con $\sum_V |s_n|$, cuyo valor sería siempre V . También hay que prestar atención a que $\langle |m| \rangle$ no tiene nada que ver con $|\langle m \rangle|$.

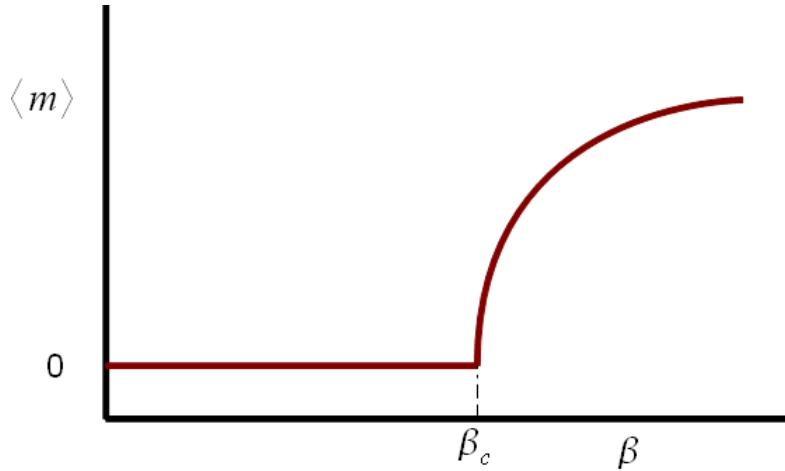


Figura 10.9: Magnetización en función de β mostrando el punto crítico

En la fase rota, el origen de que $\langle m \rangle$ sea cero en redes finitas es que el sistema está durante mucho tiempo en una región con los spines hacia arriba, y de pronto flipa todos ellos y cae en la región con todos hacia abajo (recordar la figura 10.8 que es similar en un retículo finito, incluso para $T < T_c$). Al promediar sale cero. Sin embargo, al considerar el módulo, la contribución es idéntica para ambas situaciones, y la contribución neta es como si no hiciera el flip-flop entre vacíos. El módulo por tanto reproduce el comportamiento del sistema infinito, donde el flip-flop está suprimido por una barrera infinita de energía.

Así pues, en nuestros cálculos, para estimar la magnetización usaremos siempre el observable

$$\langle |m| \rangle = \left\langle \frac{1}{V} \left| \sum_V s_n \right| \right\rangle \quad (10.67)$$

y para la Susceptibilidad

$$\chi = V(\langle m^2 \rangle - \langle |m| \rangle^2) \quad (10.68)$$

El observable $\langle |m| \rangle$ tiene el problema de que es definido positivo y por tanto no es cero ni siquiera en la fase desordenada. Estrictamente hablando entonces, este observable no es cero para $T > T_c$ y diferente de cero para $T > T_c$, no siendo un indicador perfecto para localizar la transición de fase, es decir no tendrá un comportamiento similar al indicado en la figura 10.9. Sin embargo, recordando 10.66, para $T > T_c$, $\langle |m| \rangle$ tendrá un valor muy bajo, aproximándose a cero, si calculamos en redes con L creciente. Veremos que en la fase desordenada $\langle |m| \rangle$ disminuye con el volumen como $\frac{1}{\sqrt{V}}$: esto es un ejemplo de las técnicas de *finite size scaling* para extrapolar el comportamiento a $L = \infty$.

En la región de $T < T_c$, esta cantidad es netamente diferente de cero, y de hecho cuando el retículo es grande y ya no encontramos flips-flops, reproduce el mismo valor que antes de introducir el módulo.

Por tanto, este operador, en la fase ordenada es similar a $\langle m \rangle$ y en la fase desordenada tiene una valor pequeño que se va acercando a cero cuando el Volumen de nuestro sistema crece, siendo finalmente un indicador apropiado para localizar la transición de fase.

10.7.3. Resumen de los valores medios en los casos asintóticos

Resumimos en la Tabla 10.3 el comportamiento de diferentes observables en los casos de Temperatura nula o infinita. Salvo el valor de $\chi_{|m|}$, todos se infieren directamente de las secciones anteriores. Para $\chi_{|m|}$ es necesario un cálculo adicional basado en lo estudiado hasta ahora, y que

Observable	$\beta = 0$	$\beta = \infty$
$\langle e \rangle$	0	-1
$\langle e^2 \rangle$	$1/(2V)$	1
$\langle C_v \rangle$	1	0
$\langle m \rangle$	0	{1,-1}
$\langle m^2 \rangle$	$1/V$	1
$\langle \chi \rangle$	1	0
$\langle m \rangle$	$\sqrt{\frac{2}{\pi V}}$	1
$\langle \chi_{ m } \rangle$	$1 - (2/\pi)$	0

Cuadro 10.3: Valores medios en los casos límite de Temperatura.

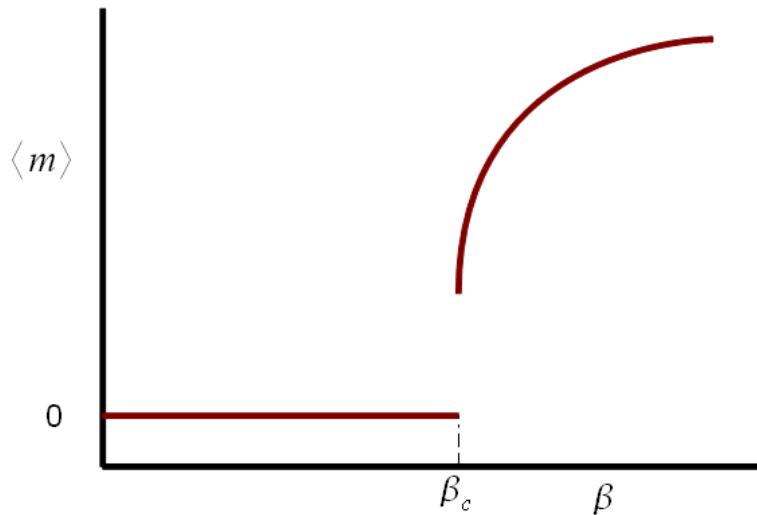
el alumno debería hacer como Ejercicio. En el Apéndice 10.10 se realizan los cálculos para la magnetización y magnitudes derivadas en el caso $\beta = 0$, que el alumno sólo debería repasar tras intentar resolver por sí sólo toda la Tabla 10.3.

10.7.4. Clasificación de las transiciones de fase

En este estudio del modelo de Ising en $d = 2$, no aparece una discontinuidad en Z ni en $\langle m \rangle$, ni en $\langle e \rangle$, sino en C_v y χ , es decir en la derivada con respecto a h de $\langle m \rangle$ o en la derivada con respecto a β de $\langle e \rangle$ (recordar las expresiones 10.21 y 10.32). Es decir se produce una discontinuidad en *alguna derivada segunda* de la función de partición Z . Entonces diremos que esta transición es de *segundo orden*.

Si la discontinuidad se produjera en al menos una derivada primera, por ejemplo en $\langle m \rangle$ o en $\langle e \rangle$, como ocurre en la figura 10.10, diríamos que la transición es de primer orden.

Discontinuidades en derivadas más altas dan lugar a transiciones de tercer orden, etc.

Figura 10.10: Magnetización en función de β en una transición de primer orden

10.8. Ejercicios

10.8.1. Multiples configuraciones

Modificar el programa del Ejercicio 9.6.1 incluyendo un bucle para generar N configuraciones, donde N debe ser otro parámetro a leer del fichero de Input, donde antes estaba sólo el flag de la configuración de partida. Calcular C_v promediando sobre las N configuraciones al azar, lo que corresponde a estudiar el sistema a Temperatura infinita (recordar 10.38). Comprobar que los resultados coinciden con los obtenidos en el texto en la sección 10.5.1. Dibujar el Histograma de la energía y la magnetización (intensivas) y comprobar que su anchura obedece la ecuación 10.27.

10.8.2. Programa de Análisis

Modificar el programa anterior, haciendo que los resultados de las medidas individuales (Energía y Magnetización) se escriban en un archivo. Posteriormente, escribir el código de un programa que llamaremos por ejemplo `ana.c`, que leera dicho archivo y realizará los análisis estadísticos, errores, histogramas, etc. con dichos datos. En el programa original de simulación ya no será necesario que se realicen todos estos cálculos, únicamente algo básico que puede irse mostrando por pantalla para verificar que todo va bien.

10.9. Problemas

10.9.1. Problema 1

Calcular exactamente (es decir, incluyendo las constantes) χ a Temperatura infinita.

10.9.2. Problema 2

Dibujar con `gnuplot` la solución exacta para la magnetización del modelo de Ising en $d = 2$.

10.9.3. Problema 3

A partir la ecuación 10.63 que nos da analíticamente el valor de la magnetización, podemos calcular el valor de β_c . En efecto, esta ecuación no tiene solución cuando el argumento es negativo, indicando precisamente esto el valor del punto crítico. Encontrar dicho valor y comprobar que coincide con 10.62.

10.10. Apéndice: Cálculo de observables en el límite $\beta = 0$

10.10.1. Cálculo de $\langle m^2 \rangle$

Método 1

La definición nos dice que

$$\langle m^2 \rangle = \left\langle \left(\frac{1}{V} \sum_i s_i \right)^2 \right\rangle = \frac{1}{V^2} \left\langle \sum_{i,j} s_i s_j \right\rangle$$

Podemos separar el sumatorio en dos grupos: por un lado los términos con $i = j$, y por otro el resto. En los términos con $i = j$ tenemos $s_i s_j = s_i^2 = 1$, teniendo V de estos términos, y por tanto

$$\langle m^2 \rangle = \frac{1}{V^2} \left(V + \left\langle \sum_{i,j, i \neq j} s_i s_j \right\rangle \right)$$

Dado que los spines son aleatorios, lo será asimismo el producto $s_i s_j$. Esta suma tiene tantos términos positivos como negativos, con lo que será cero. De la expresión anterior entonces sólo nos queda

$$\langle m^2 \rangle = \frac{1}{V^2} V = \frac{1}{V}$$

Método 2

A Temperatura infinita, cada spin toma el valor $\{+1, -1\}$ de forma aleatoria, pues la probabilidad de cualquier Energía es la misma. El hecho de que los spines sean por tanto completamente aleatorios, hace que a su vez cualquier configuración tenga la misma probabilidad. Consideremos una Red suficientemente grande. Al calcular valores medios podemos considerar que cada spin tiene el 50% de estar hacia arriba y el 50% de estar hacia abajo; del mismo modo una suma de spines del tipo

$$m = \frac{1}{V} \sum_i s_i$$

puede ser calculada facilmente usando el Teorema del Límite Central (TLC).

El problema se plantea pues en estos términos: Sea una variable s con valores $\{-1, +1\}$ con igual probabilidad. Considerar una nueva variable como la suma de V de estos términos. Calcular el valor medio y la varianza de esta cantidad.

Para la variable individual de spin s tendremos que

$$p(s = +1) = 1/2, p(s = -1) = -1/2$$

entonces

$$\langle s \rangle = \frac{1}{2}(1 + (-1)) = 0; \langle s^2 \rangle = \frac{1}{2}(1 + 1) = 1$$

Por tanto tendremos que

$$\sigma_s = \sqrt{\langle s^2 \rangle - \langle s \rangle^2} = 1$$

Definimos

$$\xi_\alpha = \frac{1}{V} \sum_i s_i$$

Dado que a Temperatura infinita las variables de spin son completamente independientes (la correlación es cero), podemos aplicar el TLC y ξ_α será una variable con media igual a $\langle s \rangle$ (es decir media 0) y con una distribución gaussiana con

$$\sigma_\xi = \frac{1}{\sqrt{V}} \sigma_s = \frac{1}{\sqrt{V}}$$

es decir

$$p(\xi) = Ae^{-\frac{\xi^2}{2\sigma_\xi^2}} = Ae^{-\frac{\xi^2 V}{2}}$$

Conocida la distribución de probabilidad de ξ , podemos calcular valores medios, en concreto la cantidad que nos interesa, a saber

$$\langle m^2 \rangle \equiv \langle \xi^2 \rangle = \frac{A \int_{-\infty}^{\infty} \xi^2 e^{-\frac{\xi^2 V}{2}} d\xi}{A \int_{-\infty}^{\infty} e^{-\frac{\xi^2 V}{2}} d\xi}$$

Esta integral se evalúa fácilmente recordando como se calculan integrales gaussianas,

$$\langle m^2 \rangle = \frac{(1/2)\sqrt{\pi}(V/2)^{-3/2}}{\sqrt{\pi/(V/2)}} = \dots = \frac{1}{V}$$

que era el resultado buscado.

Cálculo de $\langle |m| \rangle$

Definimos

$$|m| = \frac{1}{V} \left| \sum_i s_i \right|$$

y recordando el apartado anterior tenemos que para una configuración dada

$$|m| = |\xi_\alpha|$$

Dado que conocemos la distribución de ξ , podemos calcular fácilmente la expresión anterior

$$\langle |m| \rangle \equiv \langle |\xi| \rangle = \frac{A \int_{-\infty}^{\infty} |\xi| e^{-\frac{|\xi|^2 V}{2}} d\xi}{A \int_{-\infty}^{\infty} e^{-\frac{\xi^2 V}{2}} d\xi}$$

que puede ser simplificado a

$$\langle |m| \rangle = \frac{A \int_0^{\infty} \xi e^{-\frac{|\xi|^2 V}{2}} d\xi}{A \int_0^{\infty} e^{-\frac{\xi^2 V}{2}} d\xi} = 2\sqrt{\frac{V/2}{\pi}} \int_0^{\infty} x e^{-(1/2)Vx^2} dx$$

Esta integral puede realizarse mediante el cambio de variable $x^2 = t$, obteniendo

$$\langle |m| \rangle = \sqrt{\frac{V}{2\pi}} \frac{2}{V} = \sqrt{\frac{2}{\pi V}}$$

Cálculo de $\chi_{|m|}$

Recordamos que la susceptibilidad asociada a $|m|$ está definida como

$$\chi_{|m|} = V(\langle m^2 \rangle - \langle |m| \rangle^2)$$

Para calcular su valor basta recordar lo obtenido en los apartados anteriores, que nos da la expresión

$$\chi_{|m|} = V\left(\frac{1}{V} - \frac{2}{\pi V}\right) = 1 - \frac{2}{\pi}$$

Los resultados analíticos anteriores son especialmente relevantes pues nos permitirán comprobar los que obtengamos en las simulaciones que introduciremos al capítulo siguiente.

Capítulo 11

Simulación de Monte Carlo del Modelo de Ising

El objetivo es construir un programa que simule el modelo de Ising, midiendo los observables oportunos para detectar las regiones de alta y baja temperatura, y el punto crítico donde se produce la transición de fase. Debemos prestar especial atención al hecho de que simulamos en un retículo finito, por lo que no veremos señales netas de no analiticidades, sino comportamientos siempre continuos, pero más o menos abruptos, dependiendo de L .

Es necesario controlar que las partes mas consumidoras de CPU (los bucles internos del algoritmo de Metropolis principalmente, y en parte las rutinas de medidas) estén optimizadas al máximo.

Lo que pretendemos es generar un conjunto de *puntos* C_α pertenecientes al espacio de configuraciones C , y que dichos puntos estén distribuidos de acuerdo a la ley de probabilidad de Boltzman

$$p(C_\alpha) = \frac{1}{Z} e^{-\beta E(C_\alpha)}$$

pero dado que Z no es conocida sólo sabemos que

$$p(C_\alpha) \propto e^{-\beta E(C_\alpha)}$$

Por tanto recordando la sección 5.3, deberemos usar el algoritmo de Metropolis para generar la secuencia de configuraciones con la distribución exigida. Conviene en este punto que el alumno repase dicha sección y el ejercicio 5.7.1.

Dado que el programa en su versión final es relativamente complejo, es importante escribirlo tomando en consideración todas las recomendaciones dadas hasta ahora, sobre estructuración, uso de funciones, comentarios, etc. Es habitual la tendencia a escribir primero un programa *sencillo, todo seguido, sin funciones ni estructura*, pensando que luego ya se escribirá correctamente. Esto es una práctica pésima de programación, que conlleva programas ilegibles, mal estructurados y multiplicar por dos el tiempo para su reescritura.

Se debe proceder escribiendo el programa con funciones para todas las tareas posibles, construyendo el programa incrementalmente en funcionalidades, y comprobando con frecuencia que cada avance es correcto.

11.1. Algoritmo de Metropolis

Es la parte central del programa. Ya hemos hablado anteriormente de este algoritmo que resumimos ahora

Partiendo de una configuración, se realizan cambios tentativos, aceptando o no el cambio de acuerdo con el cociente de probabilidad entre ambas configuraciones (la vieja y la nueva), repitiendo el proceso tantas veces como sea necesario.

O de forma esquemática

$$\begin{aligned}
 C_{old} &\rightarrow p(C_{old}) \\
 \text{Cambio en } C_{old} &\rightarrow C_{new} \rightarrow p(C_{new}) \\
 \omega \in [0, 1] &(\text{random}) \\
 \text{if } \omega < \frac{p(C_{new})}{p(C_{old})} &\text{ cambia : } C_{old} = C_{new} \\
 &\text{else} \\
 &\text{Sigo con } C_{old} \text{ y repito el proceso.}
 \end{aligned} \tag{11.1}$$

En el caso de nuestro modelo, aunque sumamente simple, es más complejo que los problemas anteriores, y todavía tenemos algunas opciones a decidir,

- **Número de cambios simultáneos:** Para pasar de C_{old} a C_{new} podríamos cambiar un spin o varios cada vez. En general es más eficiente cambios individuales, que tienen mayor aceptancia y acumulan el efecto mejor que muchos simultáneos. Una vez decidido cambiar sólo un spin cada vez, el proceso en este modelo no deja más libertad: el único cambio posible es pasar del valor actual s_n a un valor con el signo cambiado, $s_n^{tentativo} = -s_n$.
- **Orden para recorrer los puntos:** podemos hacerlo secuencial (un punto tras otro) o random. La elección random es más correcta desde el punto de vista del algoritmo, pero el update secuencial es más efectivo, y en el modelo de Ising no da ningún problema (Salvo para el caso $\beta = 0$; ver la sección 11.1.4).

Llamaremos al proceso de cambiar de signo un spin, *flipar* el spin, o realizar un *update*.

11.1.1. Cálculo del cociente de probabilidades

Dada una configuración C_α con Energía $E(C_\alpha)$, su probabilidad es

$$p(C_\alpha) \propto e^{-\beta E(C_\alpha)} \tag{11.2}$$

si ahora cambiamos un spin, la configuración pasa a C_γ , con probabilidad

$$p(C_\gamma) \propto e^{-\beta E(C_\gamma)} \tag{11.3}$$

No conocemos la constante de probabilidad, pero no es necesaria pues sólo necesitamos el cociente de probabilidades,

$$R = \frac{p(C_\gamma)}{p(C_\alpha)} = e^{-\beta(E(C_\gamma) - E(C_\alpha))} \tag{11.4}$$

Parecería que necesitamos conocer la energía total de ambos sistemas y luego restarlo. Sin embargo, el modelo de Ising tiene una interacción local: cada variable interacciona con unas pocas de su entorno; por tanto cuando cambiamos un spin, solamente cambia la interacción con los dos vecinos hacia adelante y los dos vecinos hacia atrás. Y para calcular la diferencia solamente debemos mirar a estos cuatro términos, que son los que están indicados en la figura 11.1.

Este punto es especialmente importante. Es un error común calcular toda la energía del sistema con el spin original, luego calcular toda la energía con el spin flipado, y luego restar. Esto es correcto, pero el proceso tiene una complejidad V : es necesario recorrer todos los puntos de la red para calcular la energía total. Sin embargo, lo que hemos dicho es que basta con calcular la energía de un punto; esto es de complejidad 1. Dicho de otro modo: el cálculo del cambio de energía en una red con $L = 100$, sería 10000 veces más largo si nos empeñásemos en calcular toda la energía en lugar de sólo la del punto afectado por el cambio.

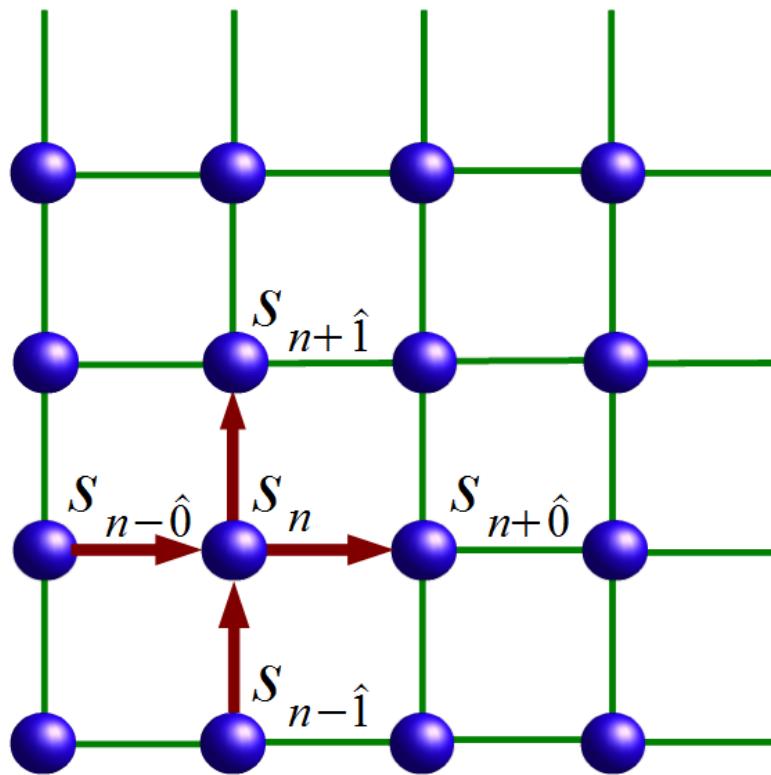


Figura 11.1: Puntos de la red necesarios para calcular el cambio de energía cuando modificamos el spin s_n

11.1.2. Cálculo del nuevo spin

Resumamos todo lo discutido hasta ahora, a saber

1. Numeración de la Red
2. Geometría y condiciones de contorno
3. Offsets para ir hacia adelante y hacia atrás
4. Uso de coordenadas cartesianas para los offsets
5. Orden de recorrido de ejes X e Y
6. Cálculo sólo de los puntos afectados por el flip del spin
7. Cálculo de la probabilidad relativa
8. Criterio de Aceptar el cambio según el algortimo de Metropolis

Con todo ello el núcleo del algoritmo de Metropolis queda del siguiente modo,

```

n=0;
for (y=0;y<L;y++)
    for (x=0;x<L;x++)
    {
        s_inicial=s[n];
        s_final=-s[n];
        E_i= -s_inicial*(s[n+xp[x]]+s[n+yp[y]]+s[n+xm[x]]+s[n+ym[y]]);
        E_f=-s_final * (s[n+xp[x]]+s[n+yp[y]]+s[n+xm[x]]+s[n+ym[y]]);
    }
}

```

```

Prob=exp(-beta*(E_f - E_i)
RAN=rand_entre_0_1();
if(RAN<Prob)
s[n]=-s[n]; //Aceptamos el cambio

n++;
}

```

Llamaremos *sweep* a este proceso, es decir a recorrer una vez toda la red y realizar un *update* en cada uno de los puntos (que puede ser aceptado o no).

11.1.3. Comprobación de signos

Es bastante frecuente confundirse con los signos en la expresión anterior. Para ver que son correctos, y a modo de ejercicio, comprobemos con un caso límite sencillo que es correcto.

Consideremos β suficientemente grande (por ejemplo del orden de 10). Sea un spin 1 rodeado también de spines con valor 1. En este caso, la física del modelo nos indica que *NO* deberá cambiar el spin al aplicar el algoritmo. En este caso, en el código anterior tendremos: $E_i = -4$, $E_f = 4$, $E_f - E_i = 8$, $Prob = e^{-8\beta} = e^{-80} \approx 0$. Generamos ahora un número aleatorio *RAN* entre 0 y 1 plano. Aceptaremos el cambio si el *RAN* < e^{-80} , es decir, prácticamente nunca, lo que es correcto.

Si el signo en la exponencial estuviera cambiado, aceptaríamos con seguridad, generando un sistema antiferromagnético, que no corresponde con la situación física que estamos estudiando (podríamos decir que correspondería con un sistema a Temperatura negativa, donde el estado de equilibrio para Temperaturas elevadas (y negativas) es la configuración antiferromagnética).

11.1.4. Medida y Ergodicidad

La variable elemental del modelo de Ising es extremadamente simple: sólo puede tomar dos valores. La posibilidad pues de cambio tentativo es única: cambiar el signo. Si la variable elemental del sistema fuera más compleja (por ejemplo, un número real, un ángulo, un vector con varias componentes...) tendríamos muchas opciones para cada cambio.

A la hora de generar la nueva configuración tenemos aún más opciones, por ejemplo el orden de cambio al elegir los puntos sobre la red, o si cambiamos un punto cada vez o varios, etc.

Se debe extremar el cuidado a la hora de elegir entre todas estas opciones. La elección debe ser tal que se cumplan las siguientes condiciones

1. En ausencia del término $e^{-\beta E}$, todas las configuraciones deberían tener el mismo peso. Es lo mismo que decir que para el valor de $\beta = 0$, todas las configuraciones son equiprobables. Esto implica que las configuraciones generadas aceptando siempre el cambio, o lo que es lo mismo, la secuencia de configuraciones creada sucesivamente con los cambios tentativos sin el factor de energía, deben serlo todas con igual probabilidad. Es decir nuestros cambios tentativos deben respetar la medida del espacio de configuración.
2. Los cambios tentativos deben ser tales que partiendo de cualquier punto del espacio de configuraciones podamos llegar, no importa en cuantos pasos, a cualquier otra configuración. (Si el espacio de configuraciones es continuo, exigimos que el recorrido sea *denso* en el espacio de configuraciones). Esto es lo que llamamos que los cambios generen trayectorias ergódicas.

En el caso que nos ocupa (Ising con Metropolis, update secuencial) es fácil ver que lo anterior falla en el caso $\beta = 0$. El algoritmo es correcto, pero la trayectoria no es ergódica. El alumno puede pensar como evitar este caso patológico.

11.1.5. Definición de aceptancia

Los cambios en la configuración se producen cuando aceptamos el cambio de un spin, en concreto cuando entramos en el *if* del bucle de la subsección 11.1.2. El número de veces que

aceptamos el cambio es por supuesto el número de spines que cambiamos. Este número es proporcional al cambio real en las configuraciones al ejecutar el algoritmo. Si este número es muy bajo, realmente no estamos haciendo nada, no nos estamos moviendo en el espacio de configuraciones. Es importante controlar el número de cambios aceptados para verificar que el programa se desarrolla correctamente.

Definimos la **aceptancia** como el porcentaje de aciertos, es decir como el cociente entre el número de cambios aceptados y el número total de intentos. Para calcular esta cantidad modificamos el código anterior,

```
#define DEBUG
n=0; N_tot=N_acep=0; //variables enteras
for (y=0;y<L;y++)
    for (x=0;x<L;x++)
    {
        s_inicial=s[n];
        s_final=-s[n];
        E_i= -s_inicial*(s[n+xp[x]]+s[n+yp[y]]+s[n+xm[x]]+s[n+ym[y]]);
        E_f=-s_final * (s[n+xp[x]]+s[n+yp[y]]+s[n+xm[x]]+s[n+ym[y]]);

        Prob=exp(-beta*(E_f - E_i))
#ifdef DEBUG
        N_tot++;
#endif

        if(RAN<Prob)
        {
            s[n]=-s[n]; //Aceptamos el cambio
#ifdef DEBUG
            N_acep++;
#endif
        }
        n++;
    }
}
```

Obtendremos la aceptancia como $(\text{float})N_{\text{acep}}/N_{\text{tot}}$. Conviene calcular esta cantidad solamente en el proceso de debugging; una vez que todo es correcto puede eliminarse para acelerar los cálculos. De hecho en el código anterior, existe un **define** para eliminar esa parte del código de la compilación tras la fase de debugging.

11.1.6. Optimización

El núcleo de cálculo anterior, correspondiente al algoritmo de Metropolis, es la parte donde se consume toda la potencia de cálculo. Optimizarlo supone un rendimiento mucho mayor de la simulación. Por tanto es conveniente dedicar un poco de atención para tratar de mejorar el cálculo.

Dado que los spines valen 1 o -1, la variación de la energía es siempre

$$E_f - E_i = 2 * s[n] * (s[n+xp[x]] + s[n+yp[y]] + s[n+xm[x]] + s[n+ym[y]])$$

siendo más eficiente el cálculo anterior, que calcular por separado cada una de sus partes y restarlo. Además el término $DE = E_f - E_i$ sólo puede tomar unos pocos valores. Si llamamos **sum** a la suma de los 4 spines vecinos,

$$\text{sum}=s[n+xp[x]]+s[n+yp[y]]+s[n+xm[x]]+s[n+ym[y]]$$

tendremos las posibilidades que pueden verse en la Tabla 11.1.

Vemos que la variación de Energia **DE** sólo puede tomar los valores

$$E_f - E_i = \{-8, -4, 0, 4, 8\} \quad (11.5)$$

sum	s[n]	DE
4	1	8
4	-1	-8
2	1	4
2	-1	-4
0	1	0
0	-1	0
-2	1	-4
-2	-1	4
-4	1	-8
-4	-1	8

Cuadro 11.1: Valores de la variación de la Energía al cambiar un spin, en función del valor de la suma de los spines vecinos. Implementando esta Tabla apropiadamente nos ahorramos un cálculo que aunque sea sencillo se repite millones de veces, aumentando por tanto la eficiencia del programa

Dado que el cálculo de la exponencial es costosa desde el punto de vista computacional y puesto que como acabamos de ver el exponente solo puede tomar 5 valores, podemos definir el siguiente vector

$$\begin{aligned}
 \text{prob}[0] &= e^{-\beta(-8)} \\
 \text{prob}[1] &= e^{-\beta(-4)} \\
 \text{prob}[2] &= e^{-\beta(0)} \\
 \text{prob}[3] &= e^{-\beta(4)} \\
 \text{prob}[4] &= e^{-\beta(8)}
 \end{aligned} \tag{11.6}$$

De este modo en lugar de calcular una exponencial, leeremos de la memoria este vector. Puesto que este cálculo se produce en la parte más interna del programa, se realiza V veces por cada iteración de Monte Carlo. Disminuir el tiempo aquí significa una ganancia sustancial en la eficiencia del programa.

Notar que el vector de la expresión 11.6, no es una probabilidad, que no la conocemos, sino un cociente de probabilidades, que conocemos exactamente.

Ahora dado el valor de `DE=E_f-E_i`, fácilmente calculable en el programa, el índice del vector `prob[]` viene dado por `Ind=(DE/4) +2`

Con todo ello el código optimizado queda de la forma

```

n=0;
for (y=0;y<L;y++)
    for (x=0;x<L;x++)
    {
        Ind=s[n]*(s[n+xp[x]]+s[n+yp[y]]+s[n+xm[x]]+s[n+ym[y]])/2+2;
        if(RAN<prob[Ind])
            s[n]=-s[n];
        n++;
    }
}

```

Al inicio del programa debemos construir los vectores `prob[]`, calculándolos una sola vez.

11.1.7. Comprobación

Podemos comprobar que el código anterior funciona correctamente en algún caso sencillo. Sea un spin 1 rodeado de 1, con β grande. Tendremos $Ind = 4/2 + 2 = 4$, y por tanto $Prob[4] = e^{-8\beta}$ y, correctamente, no aceptaremos el cambio.

11.1.8. Complejidad computacional del problema

Recordemos que cada intento de cambio de un spin en un punto, lo llamamos Update.

El bucle anterior, donde recorremos secuencialmente todos los puntos del retículo cambiando tentativamente los spines es lo que hemos llamado un paso de Monte Carlo o un *sweep*. Su complejidad es de orden V ; es decir si un update elemental nos cuesta un tiempo de CPU t_0 , entonces un sweep nos costará un tiempo $t_1 = Vt_0$. Por tanto, realizar un Sweep en un retículo con $L = 16$ nos costaría 4 veces más de tiempo de CPU que hacerlo en una red con $L = 8$.

Decimos que la complejidad del algoritmo va como L^2 . Además para obtener resultados con igual calidad al aumentar el tamaño, es necesario realizar más sweeps por lo que el tiempo de CPU necesario crece más rápido que L^2 .

También dicho tiempo de CPU depende, para L fijo, del valor de β en que estemos simulando. En la región crítica son necesarios muchos más sweeps para obtener resultados fiables, que en los valores con β muy baja o muy alta.

Inicialización del generador Random

El generador random se usa tanto para generar la configuración inicial como para el proceso de Metropolis. Depende de como escribamos el código la secuencia podría ser siempre la misma. Conviene en este punto repasar la sección 5.4. Que el sistema sea determinista es conveniente en la fase de debug pues los resultados de todo el programa deberían ser idénticos en ejecuciones diferentes para ayudarnos a detectar errores: una secuencia de números aleatorios diferentes podría generar errores en lugares diferentes, confundiendo al programador.

Una vez pasada la fase de debug, generar siempre la misma secuencia es pésimo desde el punto de vista estadístico. Conviene pues en el programa contemplar estas dos opciones: utilizar una secuencia fija partiendo de una semilla fija, o utilizar una secuencia diferente cada vez, partiendo de una semilla aleatoria. Esto puede indicarse en el fichero de Input; por ejemplo si la semilla escrita en el fichero de Input es un 0, podemos leer el tiempo del sistema y usar dicho tiempo como argumento para llamar a la función `srand()` con . Cada vez que ejecutemos el programa el tiempo será diferente y la secuencia cambiará. Si escribimos en el fichero de input un número distinto de cero, entonces usaremos ese número como semilla, por ejemplo para llamar a la función `srand()`, con lo cual la secuencia será idéntica si no cambiamos el número en el fichero de input.

11.2. Proceso de Markov e Importance Sampling

Cuando generábamos una distribución con pequeños cambios, el objetivo era construir una secuencia de números, que reproduzca la distribución de probabilidad pedida. En esta secuencia aquellos valores que tengan más probabilidad aparecerán con más frecuencia.

Nuestro algoritmo nos permite ir generando nuevos sucesos (en este caso, nuevas configuraciones) a partir de la anterior. Cada paso de Metropolis generamos una nueva configuración, que sólo depende justo de la anterior, no de las más antiguas. Esto es lo que llamamos *proceso de Markov*. Una vez que tenemos un número elevado de puntos generados correctamente, podemos calcular medias, errores, etc.

Pero puede ocurrir lo que ya hemos comentado en el caso en que queríamos generar una distribución muy estrecha (por ejemplo una gaussiana estrecha) con este método (Ver el ejercicio 5.7.1). Si comenzamos en un punto muy alejado del centro de la gaussiana (*i.e.* de la región de mayor probabilidad), al irnos moviendo lentamente, los puntos iniciales estarán muy alejados del centro; estos puntos aparecerán con una cierta frecuencia, muy por encima de lo que les correspondería. Por ejemplo si lanzamos 1000 puntos, y necesitamos 500 para llegar al centro, los primeros 500 tendrán una probabilidad por encima de la correcta. Este proceso hasta que llegamos a la zona central es lo que llamaremos *proceso de termalización*, y en el caso de las simulaciones de Mecánica Estadística debe ser controlado exhaustivamente.

11.2.1. Termalización

La Mecánica Estadística nos dice que cualquier configuración tiene una probabilidad no nula de aparecer a una cierta Temperatura. Sin embargo, para retículos de tamaño significativo, casi todas las configuraciones están situadas en torno a una energía central, y las demás tienen una probabilidad despreciable.

Dado que comenzamos de una configuración creada por nosotros arbitrariamente, estará casi con seguridad absoluta muy lejos de las configuraciones más probables. Estamos pues en un caso similar al comentado anteriormente de una distribución gaussiana muy estrecha (veáse el problema 5.7.1). La dificultad ahora es que no sabemos generar una configuración con probabilidad significativa; debemos realizar un número tal de sweeps que el sistema llegue a configuraciones relevantes, llegue a termalizar.

Para saber si el sistema está termalizado lo habitual es dibujar la evolución de diferentes observables y cuando estos llegan a un valor estable en torno al cual fluctúan sin derivas constantes, podemos decir que ha termalizado. No obstante esto tiene sus peligros.

No todos los observables termalizan simultáneamente. Algunos como la Energía lo hacen rápidamente. Otros lo hacen mucho más lentamente. Estrictamente, el sistema está termalizado cuando todos los observables han alcanzado valores estables.

Por tanto una fracción de la estadística deberá ser descartada antes de realizar los análisis pertinentes para encontrar los valores medios y errores en los mismos.

11.3. Ciclo de Histeresis

El ciclo de histéresis consiste en, partiendo de un sistema a una cierta temperatura por encima de su punto crítico, irlo enfriando en pasos lentos, hasta llegar bien debajo de la temperatura crítica, para luego volver a repetir el proceso a la inversa para acabar en la temperatura de partida. Si dibujamos para cada temperatura ciertas cantidades, como por ejemplo la Energía o la Magnetización, los valores al atravesar la zona crítica sufrirán fuertes cambios, dándonos una primera indicación de en torno a qué valores de la temperatura se sitúa el punto crítico.

Si la transición es muy *fuerte*, o en términos más precisos, si es de *primer orden*, los valores de estas cantidades a la ida y a la vuelta no coinciden en la zona alrededor del punto crítico. Es el fenómeno del agua sobrecalentada, por ejemplo: el agua a una temperatura ligeramente superior a 100 grados puede ser todavía líquida, si bien inestable, como puede observarse con un sistema tan simple como un vaso de agua calentado en un microondas; este fenómeno produce de hecho una fuerte explosión en el agua que se vaporiza instantáneamente por cualquier vibración o movimiento, significando un grave peligro si se coge con la mano un recipiente con agua sobrecalentada.

En nuestro caso, el ciclo de histéresis no se presenta si se hace estadística suficiente, pues la transición es de segundo orden. Sin embargo, haciendo poca estadística, la no termalización puede simular algo parecido a un ciclo de histéresis.

De todos modos, realizar este ciclo es la mejor aproximación para tener una primera estimación de la región donde se sitúa la temperatura crítica.

Para obtener una visión global del comportamiento del sistema, realizaremos un cálculo del siguiente modo

1. Fijamos β a un valor inicial pequeño.
2. Realizamos N_{Ter} pasos de termalización, sin realizar medidas en estas configuraciones. Esperamos que durante este tiempo el sistema se aproxime al equilibrio.
3. Realizamos N_{med} ciclos, cada uno de ellos con N_{Met} ciclos de Metropolis, midiendo al finalizar cada uno de estos últimos bloques. Dispondremos pues de N_{med} medidas.
4. Con las medidas anteriores, calculamos los valores medios, así como los errores.

5. Escribimos los resultados en un archivo.
6. Incrementamos el valor de β a $\beta + \delta_\beta$ y repetimos todos los pasos anteriores, partiendo de la configuración anterior, que estará termalizada para la β anterior. Si el cambio en β no es muy grande, el tiempo para termalizar al nuevo valor, será pequeño.
7. Una vez alcanzado el valor de β que consideremos máximo, comenzamos a decremetar β en pasos iguales, es decir en cada paso hacemos $\beta = \beta - \delta_\beta$.
8. Realizamos ahora los mismos pasos que antes, hasta llegar al valor inicial de β

Un posible segmento de código que hace esto, tendría la forma

```
main()
{
    lee_input(); //Lee beta_inicial, beta_final,delta_beta,N_Ter,N_med,N_Met
    Inicializa_Random();
    Genera_configuracion_Inicial(); //Puede generarla o leerla

    N_pasos=(beta_final-beta_inicial)/delta_beta;//atencion: N_pasos es entero
    beta=beta_inicial;

    for(sentido=0;sentido<2;sentido++)
    {
        for(N_betas=0;N_betas<N_pasos;N_beta++)
        {

            Calcula_prob(beta); //Calculamos la tabla de probabilidad para la nueva beta

            for(M_Met=0;N_Met<N_Ter;N_Met++) //Proceso de termallizacion
                Metropolis();

            Inicia_Vector_Medidas(N_med); //Pon a cero los acumuladores
            for(N_m=0;N_m<N_med;N_m++) //bucle en medidas
            {
                for(N_M=0;N_M<N_Met;N_M++) //Iteraciones de Monte Carlo
                    Metropolis();

                Medidas(); //Mido
                Construir_Vector_Medidas(N_m); //Guarda resultados en un vector
            }

            Calcula_Valores_Medios(); //Calcula las medias y errores para ese valor de beta
            Escribe_Valores_Medios(); // Escribe beta, medias y errores en tipo append
            Escribe_Informacion_en_Pantalla(); // VA todo bien??
            beta+=delta_beta; //Incremento el valor de beta
        }
        delta_beta=-delta_beta
    }
}
```

El alumno debe repasar con suma atención el código anterior que contiene importantes detalles y conceptos imprescindibles para una comprensión global del problema.

Si bien en el segmento de código anterior no se escriben los datos del proceso de termalización, en los primeros runs conviene escribir todos los resultados (por ejemplo poniendo $N_{Ter} = 0$) para observar el proceso de termalización y calcular cuantas iteraciones debemos realizar para que el sistema termalice; una vez averiguado este número, podemos volver a poner N_{Ter} a este valor, para así obtener medidas con configuraciones todas ellas termalizadas.

11.3.1. Escritura de datos

la función `Calcula_Valores_Medios()`, debe calcular las medias con sus errores para ese valor de β . Es importante utilizar las funciones desarrolladas en el Ejercicio 6.6.1 para calcular medias y dispersiones. Con esos datos rellenamos un vector del tipo `double results[7][N_pasos]`, donde guardamos para cada valor del índice `N_betas` los datos en el siguiente orden,

1. double results[0][N_betas] $\equiv \beta$
2. double results[1][N_betas] $\equiv \langle e \rangle$
3. double results[2][N_betas] $\equiv \epsilon_e$
4. double results[3][N_betas] $\equiv \langle |m| \rangle$
5. double results[4][N_betas] $\equiv \epsilon_m$
6. double results[5][N_betas] $\equiv \langle e^2 \rangle$
7. double results[6][N_betas] $\equiv \langle m^2 \rangle$

Notar que el calor específico vendrá dado por

$$C_v = 2V(\langle e^2 \rangle - \langle e \rangle^2) \equiv V(\text{results}[5][N_betas] - \text{results}[1][N_betas] * \text{results}[1][N_betas]) \quad (11.7)$$

que puede ser calculado directamente en `gnuplot`. Una expresión equivalente para la susceptibilidad,

$$\chi = V(\langle m^2 \rangle - \langle |m| \rangle^2) \equiv V(\text{results}[6][N_betas] - \text{results}[3][N_betas] * \text{results}[3][N_betas]) \quad (11.8)$$

La rutina `Escribe_Valores_Medios()` debe escribir estos datos en un archivo en formato texto, generando una nueva línea para cada valor de β . Para no tener que esperar hasta el final de la ejecución y poder ver los resultados con `gnuplot` en tiempo real, es necesario abrir el archivo con formato `append`, del siguiente modo

```
main()
{
    ...
    fout=fopen("datos.dat","at");
    Escribe_Valores_Medios(); // Escribe beta, medias y errores en tipo append
    fclose(fout);
    ...
}
```

Si dejáramos el código así, dos ejecuciones sucesivas del programa escribirían en el mismo archivo, lo que sería incorrecto. Es necesario borrar el archivo antes de cada ejecución. Puede hacerse de modo automático, añadiendo el siguiente código al inicio del programa.

```
FILE *fout;
...
fout=fopen("datos.dat","wt");
fclose(fout);
```

de este modo, si el archivo `datos.dat` no existe, se crea vacío, y si existe se borra y se crea otro vacío, que es lo que pretendíamos.

11.3.2. Elección de parámetros

El objetivo es describir el comportamiento global del sistema. Para ello debemos elegir un retículo de tamaño medio, por ejemplo entre $L = 8$ y $L = 12$.

En este retículo debemos realizar un barrido por ejemplo en el rango

$$\beta \in [0, 2] \quad (11.9)$$

Podemos realizar una simulación de Monte Carlo para valores de β en pasos pequeños; la elección depende del tiempo de CPU que queramos consumir, del tamaño de la red, la precisión, etc. Una buena elección para empezar puede ser

$$\delta_\beta = 0.05 \quad (11.10)$$

Debemos fijar todos los parámetros necesarios en el fichero de datos iniciales, que se lee en la función `lee_input`; un fichero razonable podría ser el siguiente

```

0 (beta inicial)
2.0 (beta final)
0.05 (delta beta)
200 (Pasos de termalizacion)
1000 (pasos para medir)
5 (pasos de monte carlo entre medidas)

```

En este caso, en cada valor de β realizaremos $200 + 5 \times 1000$ sweeps y tendremos $(\beta_f - \beta_i)/\delta_\beta$ valores de β en cada dirección. En el ejemplo, nuestro fichero de salida tendrá $2 \times \frac{2-0}{0.05}$ líneas de datos.

Para cada temperatura o β , nos basta calcular los valores medios directamente; como hemos comentado los resultados finales de cada temperatura debemos escribirlos en un archivo de texto. Podremos dibujar con `gnuplot` el comportamiento global del sistema, viendo los valores de los observable con sus errores. Localizando donde se producen cambios bruscos en la energía y la magnetización, y también picos en el calor específico y la susceptibilidad, identificaremos aproximadamente donde se encuentra el punto crítico.

11.3.3. Simulación en el entorno del punto crítico

Una vez identificado aproximadamente el punto crítico, debemos repetir el cálculo anterior, pero en un entorno reducido, con menos valores de β , y con mucha más estadística en cada punto. El ciclo de histéresis ahora basta hacerlo sólo en un sentido, pues no debería haber diferencias si el sistema es debidamente termalizado, usando la mitad del tiempo. De este modo podremos determinar con mayor precisión el punto crítico del modelo.

11.3.4. Simulación de diferentes tamaños

Para comprender mejor el comportamiento del sistema, y una vez realizado el estudio completo en el primer tamaño conviene repetir los cálculos en un retículo mayor, por ejemplo $L = 32$. Es importante controlar que los valores elegidos para termallizar son correctos.

Una vez obtenidos los resultados finales, es muy instructivo superponer los observables en los dos tamaños para observar las semejanzas y diferencias. Deberemos fijarnos en concreto, en los dos hechos siguientes,

1. El modulo de la magnetización se acerca a cero en la fase desordenada con $\frac{1}{\sqrt{V}}$, y es casi constante en la fase rota
2. Los resultados de los observables Energía, Calor específico y Susceptibilidad, apenas cambian fuera del entorno del punto crítico.
3. En la zona crítica (en el entorno de T_c), la Susceptibilidad y el Calor específico crecen al aumentar el tamaño de la red, indicando la divergencia que aparece en el límite termodinámico.

El resultado final de todos estos cálculos debe ser la comprensión de la transición de fase del modelo, la evolución de los observables, la aparición de señales de discontinuidades y la localización del punto crítico, β_c .

11.4. Ejercicios

11.4.1. Simulación del Modelo de Ising: Trabajo de Grupo

Este ejercicio deberá ser realizado por grupos de 5 alumnos. El ejercicio completo ocupará dos clases de prácticas. Durante la clase de prácticas los alumnos pueden trabajar en grupo o individualmente según lo consideren oportuno. En la siguiente clase teórica, los alumnos realizarán una presentación en clase utilizando el cañón, durante 10 minutos resumiendo el trabajo, especialmente los resultados obtenidos. Se puntuará conjuntamente el trabajo y la presentación, con un máximo de 1 punto para la nota final.

En el programa realizado en los ejercicios del capítulo anterior, introducir el Algoritmo de Metropolis. Considerar retículos de lado $L = 8, 12, 16, 24$. Simular varios valores de β en torno a β_c . Los datos para obtener las medias, deben ser datos termalizados, es decir, deben descartarse los resultados previos a la termalización. Calcular para cada β y L , los valores de C_v o χ con sus errores.

Estimar el valor crítico aparente de β en cada L , es decir $\beta_c(L)$

Realizar todas las gráficas necesarias para mostrar los resultados, y dar toda la información necesaria para demostrar que los resultados presentados han sido calculados correctamente.

Incluyendo el programa de análisis, la estructura y el flujo de datos completo queda como puede verse en la Figura 11.2.

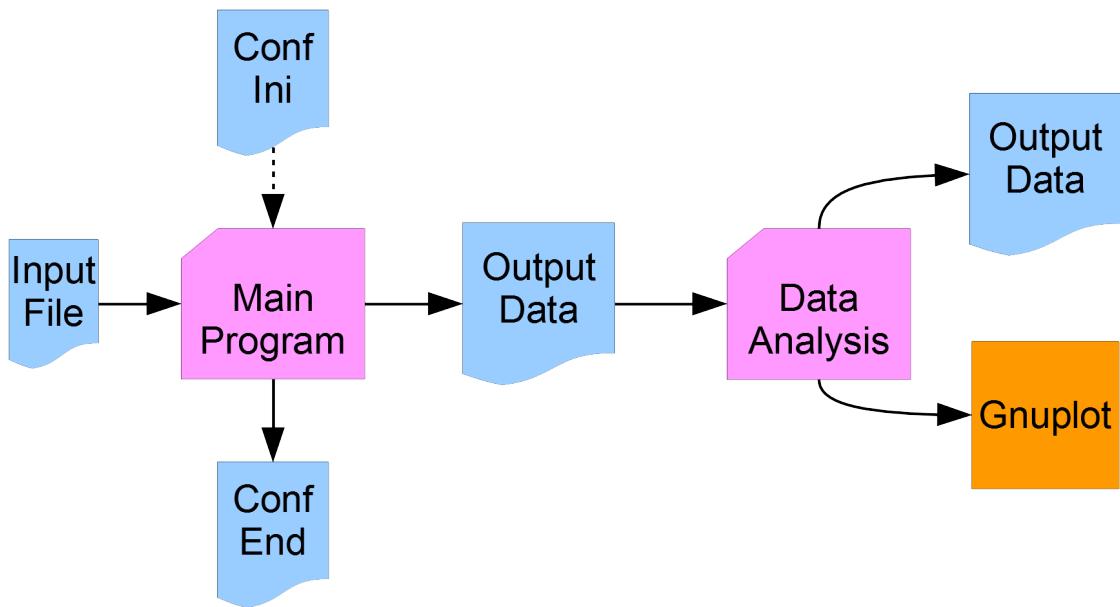


Figura 11.2: Estructura y flujo de Información en el Programa de Simulación de MonteCarlo del Modelo de Ising.

11.5. Problemas

11.5.1. Problema 1

A partir de un archivo donde se escriban los valores de las cantidades relevantes cada K iteraciones de Monte Carlo, dibujar la evolución del sistema y estimar el tiempo de termalización para diferentes valores de β y para diferentes valores de L .

11.5.2. Problema 2

Realizar un fit de $\langle |m| \rangle$ para $T \leq T_c$ de la forma $\langle |m| \rangle = A(T_c - T)^\beta$. Este valor de β es un *exponente crítico*; comparar el valor obtenido con el que se da en la literatura científica y que puede extraerse mediante un desarrollo en serie de la expresión 10.63 en torno a β_c . ¿Qué ocurre con $\langle m \rangle$?

11.5.3. Problema 3

Considerar el modelo de Ising en una red de tamaño $L_x = 64$, $L_y = 32$. En esta red, fijamos a $+1$, por medios externos (p.ej. con un fuerte campo magnético), los espines en las líneas $x = 0$ y $x = 63$, de manera que estarán siempre fijos y sobre ellos no aplicamos el algoritmo de Metropolis. En el resto de los espines ($x \in [1, 62]$) se aplica la dinámica de Ising habitual.

Los espines con $x = 1$, ven a su izquierda el spin fijo (con valor $+1$), al igual que los espines con $x = 62$, que ven congelado el spin a su derecha (también con valor $+1$). Podemos definir ésto como condiciones de contorno fijas en el eje x . En el eje y mantenemos condiciones periódicas.

Definimos la Magnetización en función de la variable x , es decir,

$$M(x) = \frac{1}{L_y} \sum_{y=0}^{L_y-1} s[x, y].$$

Estudiar el modelo para $\beta = 0.4$, 0.4406868 , 0.49 , calculando el comportamiento de $M(x)$ en estos casos. Recordar que para todas las cantidades calculadas debemos estimar el error, en concreto para $M(x)$ en cada x .

Hacer un fit de $M(x)$ a la función $a \cosh(m(x - \frac{L_x-1}{2}))$ y calcular a y m para $\beta = 0.4$. Para $\beta = 0.4406868$ y para $\beta = 0.49$ hacer un fit con `gnuplot` a la función $C + a \cosh(m(x - \frac{L_x-1}{2}))$ y calcular C , a y m . Para realizar el fit, debemos tener en cuenta los errores en $M(x)$, que serán funciones de x .

Nota En el Apéndice 11.6 pueden verse gráficas resultantes de la simulación y el resultado del fit para C, a, m obtenidos con `gnuplot`

11.6. Apéndice: Resultados para el Problema

Mostramos los resultados de la simulación del Modelo de Ising en $d = 2$ con $L_X = 64$, $L_Y = 32$ correspondientes al problema 11.5.3. En cada gráfica los puntos están dibujados con sus errores; la línea verde que une los puntos es una simple unión para guiar el ojo. La línea azul es el resultado del fit a la función coseno hiperbólico especificada en el problema mencionado. La estadística ha sido de 100000 medidas, cada una separada por 500 iteraciones, es decir un total de 5^7 iteraciones para cada β .

Veamos ahora como pueden dibujarse los gráficos anteriores con `gnuplot`. Supongamos que el formato de datos tiene la siguiente forma:

```
0 1.000000 0.000000
1 0.784119 0.000053
2 0.646808 0.000112
3 0.541207 0.000171
...

```

```

60 0.541430 0.000172
61 0.647063 0.000113
62 0.784241 0.000053
63 1.000000 0.000000

```

donde el primer número es la coordenada x de la linea, el segundo es la media de la magnetización de la línea, y el último el error.

El código a cargar en gnuplot con el comando `load` es el siguiente:

```

cd 'D:\test\ising'
L=64.0
f(x)=a*cosh(m*((L-1.0)/2.0-x))
set yrang [0.005:]
plot "ising2d_M_64_Op4.dat" u 1:2:3 with errorbars title "L(64,32) B=0.4, N=10^5x500"
replot "ising2d_M_64_Op4.dat" u 1:2:3 w lines title ""
a=0.01
m=0.1
fit [0.1:(L-2)+0.1] f(x) "ising2d_M_64_Op4.dat" u 1:2 via a,m
set label sprintf("a=%g",a) at L*4.0/6.0,0.5
set label sprintf("m=%g",m) at L*4.0/6.0,0.4
set logscale y
rep f(x)title "fit to cosh"

pause -1

set term png enhanced linewidth 4 font arial 64 size 4096,2896
set output "L64_b_Op4.png"
rep
reset
set output
set terminal windows

f(x)=C+a*cosh(m*((L-1.0)/2.0-x))
C=0.1
set yrang [0.5:]
plot "ising2d_M_64_Op4406868.dat" u 1:2:3 with errorbars t "L(64,32)B=0.4406868, N=10^5x500"
replot "ising2d_M_64_Op4406868.dat" u 1:2:3 w lines title ""
a=0.01
m=0.1
fit [0.1:(L-2)+0.1] f(x) "ising2d_M_64_Op4406868.dat" u 1:2 via C,a,m
set label sprintf("a=%g",a) at L*4.0/5.0,0.92
set label sprintf("m=%g",m) at L*4.0/5.0,0.87
set label sprintf("C=%g",m) at L*4.0/5.0,0.82
set logscale y
rep f(x)title "fit to cosh"

pause -1

set term png enhanced linewidth 4 font arial 64 size 4096,2896
set output "L64_b_Op4406868.png"
rep
reset
set output
set terminal windows

set yrang [0.88:]
f(x)=C+a*cosh(m*((L-1.0)/2.0-x))

```

```

C=0.88
plot "ising2d_M_64_0p49.dat" u 1:2:3 with errorbars t "L(64,32)B=0.49, Niter=10^5x500"
replot "ising2d_M_64_0p49.dat" u 1:2:3 w lines title ""
a=0.01
m=0.1
fit [0.1:(L-2)+0.1] f(x) "ising2d_M_64_0p49.dat" u 1:2 via C,a,m
set label sprintf("a=%g",a) at L*4.0/5.0,0.98
set label sprintf("m=%g",m) at L*4.0/5.0,0.96
set label sprintf("C=%g",C) at L*4.0/5.0,0.92

set logscale y
set ytics 0.85,0.05,1.0
rep f(x)title "fit to C+cosh"

pause -1

set term png enhanced linewidth 4 font arial 64 size 4096,2896
set output "L64_b_0p49.png"
rep
reset
set output
set terminal windows

set yrangle [0.2:]
L=24
f(x)=a*cosh(m*((L-1.0)/2.0-x))
a=0.01
m=0.2
plot "ising2d_M_24_0p4.dat" u 1:2:3 with errorbars title "L(24,12) B=0.40, Niter=10^5x500"
replot "ising2d_M_24_0p4.dat" u 1:2:3 w lines title ""
a=0.01
m=0.2
fit [0.1:(L-2)+0.1] f(x) "ising2d_M_24_0p4.dat" u 1:2 via a,m
set label sprintf("a=%g",a) at L*4.0/5.0,0.8
set label sprintf("m=%g",m) at L*4.0/5.0,0.7

set logscale y
rep f(x)title "fit to cosh"
pause -1

set term png enhanced linewidth 4 font arial 64 size 4096,2896
set output "L24_b_0p4.png"
rep
reset
set output
set terminal windows

```

Tras realizar los cálculos y mostrar los resultados, se crean tres fichero .png para poder ser usados como imágenes (De hecho son las imágenes de este apéndice).

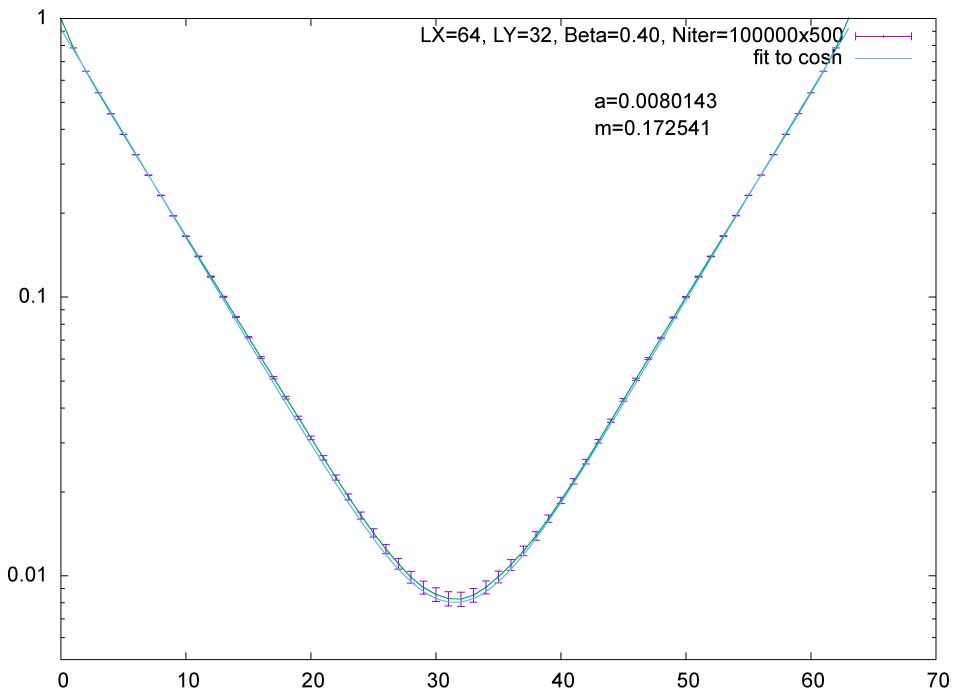
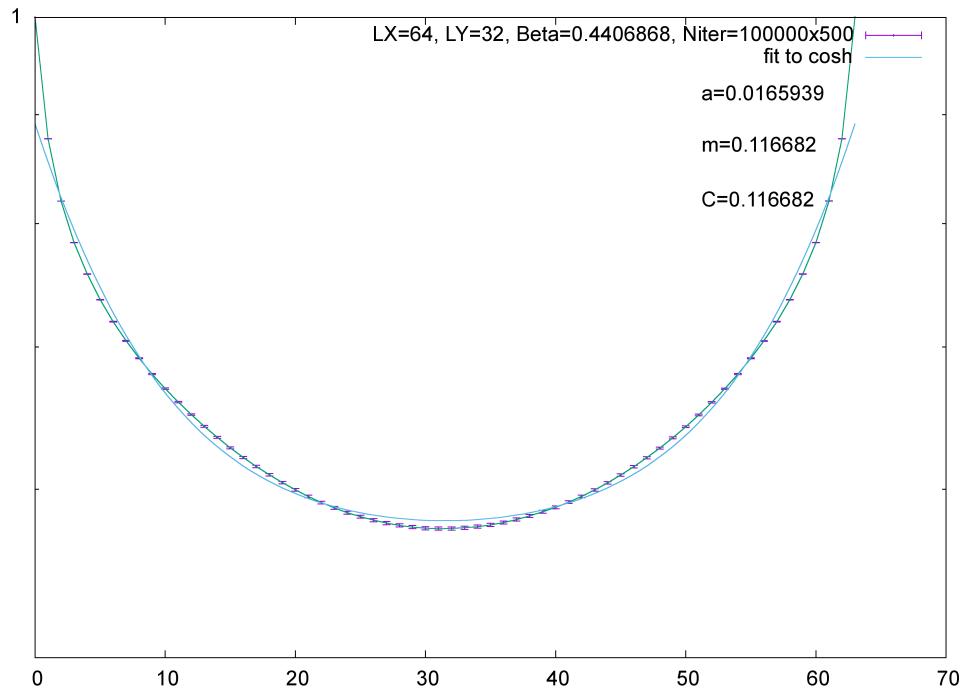
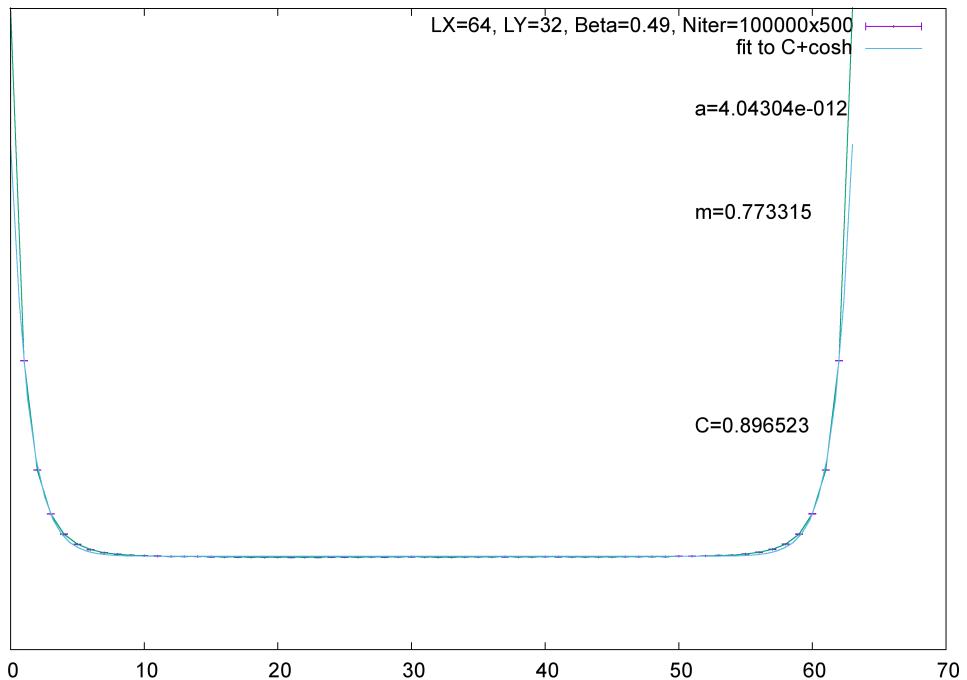


Figura 11.3: Magnetización en el Modelo de Ising en función de x a $\beta = 0.4$ ($\beta < \beta_c$) para $L = 64$, donde se han fijado los spines para $x = 0$ y $x = 63$.

Figura 11.4: Como Figura 11.3 pero a $\beta = 0.4406868$. ($\beta \approx \beta_c$)Figura 11.5: Como Figura 11.3 pero a $\beta = 0.49$. ($\beta > \beta_c$)

Capítulo 12

Simulación avanzada del Modelo de Ising

En el capítulo anterior hemos realizado una simulación que presenta algunas carencias. En primer lugar simulamos varios valores de β , sin controlar exhaustivamente si los datos están debidamente termalizados. Además en el análisis de errores no controlamos si existen o no correlaciones. Por último el error de los operadores compuestos no ha sido estimado

Para poder realizar un estudio más detallado del modelo, introduciremos varias modificaciones

- Simularemos en cada ejecución un único valor de β .
- Durante la simulación escribiremos todas las medidas de los observables elementales (en nuestro caso son e y m) en uno o más archivos que guardaremos en el disco para su posterior análisis.
- Escribiremos un nuevo programa, cuya función es leer los archivos de datos generados por el programa de simulación, y estudiar la termalización, calcular los valores medios y los errores de todas las cantidades que necesitemos.

Pasaremos a continuación a profundizar y detallar todos esos pasos.

12.1. Generación de Datos

Existen numerosas dificultades en el proceso de Monte Carlo para el modelo de Ising, que destacamos a continuación, así como las medidas para solucionarlos,

- **Autocorrelaciones** Con Metropolis cada configuración se va generando a partir de la anterior. Cuando la temperatura está próxima al valor crítico, las configuraciones tienden a guardar *memoria* durante mucho tiempo de las anteriores; es decir existen correlaciones a muy largas distancias en el tiempo de Monte Carlo. Estamos pues ante una serie de datos correlacionados que deben ser tratados correctamente. Dada esta correlación, si midiéramos en todas las configuraciones, serían medidas casi idénticas, que no aportarían nada nuevo a los datos. Por ello lo usual es realizar un bucle interno donde se realizan un número dado de sweeps consecutivos, seguido de una medida. Todo este proceso se debe realizar muchas veces.
- **Archivos de Medidas** Las medidas que vamos realizando, serán motivo posterior de diferentes análisis, que no pueden ser previstos en general, pues no sabemos si están termalizados, si necesitaremos más estadística, si será necesario algún observable no previsto, etc. Esto es imposible si al análisis de los datos se realiza a la vez que se van simulando las configuraciones, como en la versión simple anterior. Por todo ello es necesario ir escribiendo los datos en archivos cada cierto tiempo, datos que luego serán leídos y analizados por un programa específico. En cada archivo pueden guardarse no una sino muchas medias

consecutivas. Lo natural es escribir un archivo con los resultados cada varios minutos de CPU, así como la configuración actual para poder continuar caso necesario o si se produce algún error por desconexión, fallo de hardware, etc. Cada archivo debe tener un nombre diferente para poder guardar la información de modo más seguro.

- **Termalización** El proceso de llegada al equilibrio es complejo y dependiente de muchos factores. Si disponemos de un buen número de archivos con medidas sucesivas, el proceso de termalización se realiza fácilmente descartando para el análisis un número apropiado de ficheros.
- **Configuraciones** Como a priori es difícil conocer con precisión estos números, es conveniente guardar la última configuración, sobreescritiendo la anterior, para que en el caso de que necesitemos más estadística, podamos continuar de la última, aprovechando el proceso anterior de termalización y los datos termalizados si los hubiere. Esto previene perdidas graves si hay algún error de hardware en el sistema.
- **Sweeps y Medidas** Dentro del bucle de medidas, realizamos varias de ellas. Consideremos como ejemplo la Energía E . Dado que escribimos tras muchas medidas, debemos guardar los resultados temporalmente en la RAM para no perderlos; lo más simple es construir un vector de Energías, por ejemplo $E_med[i]$, donde el índice i es el de medida. Cuando escribamos los resultados, escribiremos todas las medidas simultáneamente, en concreto, de acuerdo al ejemplo posterior, cada vez escribiremos tantas como $N_medidas$. Lo mismo para el resto de observables. Esas medidas las escribimos en un archivo con un nombre dado, nombre que debe ser diferente en el ciclo siguiente; para ello debemos generar un nombre diferente de archivo cada vez que vayamos a escribir uno nuevo.

Los valores exactos de estos parámetros dependen del tamaño de la red, del valor de la temperatura y del error con que queramos obtener los resultados.

La estructura recomendada para el bucle principal del programa, en `main()`, sería aproximadamente

```
for(N_a=0;N_a<N_archivos;N_a++) //Bucle en archivos diferentes
{
    for(N_med=0;N_med<N_medidas;N_med=0) //Bucle en Medidas para cada archivo
    {
        for(N_Met=0;N_Met<N_Metropolis;N_Met++) //Bucle en iteraciones MC sin Medir
            Metropolis();

        Medidas(); //Calculo la energia y magnetizacion
        Construir_Vector_Medidas(N_med); //Guardo las medidas en un vector
    }
    Escribe_Medidas(N_a); //Escribe el vector con las N_medidas del bloque: nuevo archivo
    Escribe_Conf(N_a); // Escribe la ultima configuracion: sobreescribe la anterior
    Escribe_Informacion_en_Pantalla(); // VA todo bien??
}
```

Los prototipos de las funciones, su valor y argumentos son meramente indicativos.

12.1.1. Lectura de datos iniciales

De lo visto hasta ahora, debemos dar al programa varios valores para realizar la simulación. Concretamente,

1. Directorio de datos de Input-Output: El directorio donde está el fichero de configuración si se va a leer, y donde se escribirán los ficheros de resultados.
2. Valor de β
3. Valor Random para semilla inicial. Si es cero, la leemos del reloj del sistema, lo que generará cada vez una secuencia diferente. Si no es cero, usamos el número como semilla, con lo cual podremos reproducir resultados si queremos, por ejemplo con propósitos de debugging.

4. Tipo de configuración de partida: un flag que nos indique si queremos partir de una configuración random, congelada o leída de un archivo.
5. Número de archivos a generar: `N_archivos`
6. Número de medidas por archivo: `N_medidas`
7. Número de pasos de Metropolis entre medidas: `N_Metropolis`

Todos estos datos deben estar en un archivo tipo texto que será leído por el programa desde una función específica. Esta función debe controlar que se lean todos los números correctamente y avisar si hay algún error. El nombre del archivo puede ser fijo, o bien puede ser dado en la línea de comandos.

Con esta estructura el número total de pasos de Monte Carlo será de

$$N_{\text{archivos}} * N_{\text{medidas}} * N_{\text{Metropolis}}$$

esa cantidad debe ser controlada para tener una estimación del tiempo de CPU necesario para ejecutar el programa. Este tiempo será proporcional a

$$t_{CPU} \propto N_{\text{archivos}} * N_{\text{medidas}} * N_{\text{Metropolis}} * V \quad (12.1)$$

12.1.2. Rutina de medidas y Escritura de resultados

Las cantidades que debemos medir son la Energía y la Magnetización. A partir de ellas podemos calcular el Calor específico y la Susceptibilidad, entre otros observables.

Debemos guardar las medidas individuales para tener toda la información. Del mismo modo debemos escribir

$$m = \frac{1}{V} \sum_V s_n \quad (12.2)$$

sin tomar el módulo, que puede ser hecho durante el proceso de análisis; de este modo no perdemos información física sobre el problema.

A partir de la energía y la magnetización podemos calcular también otros observables si son necesarios, como $\langle E^4 \rangle$, $\langle Em \rangle$, etc. así como los errores en todas estas cantidades.

En cada bloque de medidas, realizamos `N_Medidas`. Tendremos pues $2 * N_{\text{Medidas}}$ números para escribir, que deberemos guardar en un vector de tamaño `N_Medidas` para la Energía y en otro similar para la magnetización.

Si lo escribimos en un fichero binario usaremos menos espacio y será más rápido de acceder. Dado que para obtener resultados fiables podemos necesitar millones de datos, es conveniente usar archivos binarios para almacenamiento masivo. Esto permite además mantener todos los bits significativos. Si lo escribimos en un fichero texto, debemos escribir suficientes cifras para no perder precisión. Cada fichero de datos debe tener un nombre diferente; para ello un código apropiado es concatenar el nombre con el índice del bucle de medidas.

Damos a continuación una función que hace todo lo anterior, escribiendo los datos en un archivo binario,

```
void escribe_medidas(int i)
{
    // i es el indice de archivo (el valor de N_med)
    // dir es el directorio leido en el input donde se guardan los datos
    // el archivo creado tiene el nombre (supongamos que i vale 12, y
    // que dir vale C:\Ising\L12\b1\

    // Archivo: C:\Ising\L12\b1\OUT012.DAT

    int idat;
    char name[256]; //suficiente a no ser que el path sea super largo

    //Construyo una string con el nuevo nombre para el archivo
```

```

sprintf(name,"%s%s%03d.DAT",dir,"OUT",i);
Foutput=fopen(name,"wb");

// Suponemos que Energia y Magnetización son tipo double

fwrite(Energia,8*N_Medidas,1,Foutput);
fwrite(Magnetizacion,sizeof(double)*N_Medidas,1,Foutput);

fclose(Foutput);
}

```

Si queremos un archivo tipo texto, cambiamos las siguientes líneas, dejando el resto idéntico

```

Foutput=fopen(name,"wt");

// Suponemos que Energia y Magnetización son tipo double

for(idat=0;idat<N_Medidas;idat++)
    fprintf(Foutput,"%20.17f %20.17f\n",Energia[idat],Magnetizacion[idat]);

```

También es muy conveniente poder ver durante la simulación como van los números, si los resultados son los esperados, si el tiempo de CPU es el que hemos estimado antes, etc. Para ello es escribimos en pantalla algunos resultados parciales. Para facilitar la visualización gráfica, estos mismos datos, en formato apropiado, los podemos escribir en un archivo tipo texto para dibujar con *gnuplot*. Esta puede ser una opción en la fase de debugging, y eliminarla después, para no saturar el sistema de Input/Output.

12.1.3. Estructura global del Programa

Veamos una posible estructura del `main()`. No hemos escrito el prototipo de las funciones, que debe ser considerado por el alumno. La idea es que debemos minimizar el número de variables globales, pasando como argumento los datos necesarios. En cualquier caso, las variables que son comunes absolutamente para todas las funciones, pueden ser declaradas globales.

```

main()
{
    lee_input();
    Inicializa_Random();
    Genera_Configuracion_Inicial(); //Puede generarla o leerla

    for(N_a=0;N_a<N_archivos;N_a++)
    {
        for(N_med=0;N_med<N_medidas;N_med=0)
        {
            for(N_Met=0;N_Met<N_Metropolis;N_Met++)
                Metropolis();

            Medidas();
            Construir_Vector_Medidas(N_med);
        }

        Escribe_Medidas(N_a); //Escribe las N_medidas del bloque
        Escribe_Conf(N_a); // Escribe la ultima configuracion
        Escribe_Information_en_Pantalla(); // VA todo bien??
    }
}

```

12.2. Análisis de Resultados

Una vez realizada una simulación para un cierto retículo y una cierta temperatura, tendremos una serie de archivos con los resultados en un directorio conocido.

Debemos ahora escribir un programa que lea estos datos y los procese.

En primer lugar debemos controlar el tiempo de termalización. Para ello debemos dibujar la evolución de los observables y ver a partir de qué tiempo de Monte Carlo fluctúan en torno a una

constante. Una forma precisa es analizar tirando ficheros iniciales hasta que los valores medios se mantengan estables.

Una vez que sepamos a partir de qué archivo el sistema está termalizado, procederemos al análisis estadístico de los mismos. Los resultados deben ser al menos,

- **Evolución** Una fichero para ver la evolución en el tiempo de Monte Carlo del sistema, tanto de la energía como de la magnetización (no del módulo). Escribir medidas individuales, no medias de medidas, pues se pierde información.
- **Histogramas** Un histograma de la energía y de la magnetización
- **Valores Medios** Valor de la energía, el Calor específico, la magnetización (ahora con módulo) y la Susceptibilidad magnética (restando también la magnetización con el módulo); cada uno de estos observables, con su error.
- **Fichero final** Un archivo con los resultados finales, con sus errores y el valor de β en un archivo texto.

Esto debemos hacerlo para diferentes valores de β en el entorno de la región crítica. Finalmente debemos reunir todos los valores medios obtenidos para cada simulación en un fichero, para poder visualizar el sistema a lo largo del eje β , mostrando la transición de fase como un punto de cambio cualitativo de las propiedades del sistema.

12.2.1. Cálculo de errores

Los datos que vamos obteniendo tienen dos problemas

- Termalización
- Autocorrelaciones

Supondremos que hemos determinado el tiempo de termalización. Ahora nos queda el problema de la autocorrelación, es decir, que las medidas sucesivas no son estadísticamente independientes y por tanto la estimación del error calculado como $\epsilon = \frac{\sigma}{\sqrt{N}}$ con N todas las medidas, es incorrecta.

Ya hemos comentado como evitar esto: debemos agrupar los datos en bloques, calcular las medidas en cada bloque, y cuando el error calculado sea aproximadamente independiente del tamaño del bloque, la estimación del error será la correcta.

Esto es fácil en el caso de observables simples como la Energía o la Magnetización.

Pero ¿qué ocurre en el caso de observables compuestos como el Calor específico o la Susceptibilidad?

En este caso hay varias opciones

- Propagación de errores a partir de sus partes
- Cálculos por bloques y estimación a partir de esos datos

Aquí debemos utilizar el segundo método, mucho más fiable en este problema. Recordemos que el método del cálculo por bloques está basado en la idea de que cada bloque puede considerarse como un experimento independiente, siempre que el número de medidas en el bloque sea suficientemente grande: en ese caso cada bloque es independiente estadísticamente, y las medias de los bloques tendrán una distribución gaussiana, por lo que la estimación del error será la correcta. Conviene repasar la sección 6.5 donde se presenta en detalle este procedimiento.

Recordamos la forma de implementarlo en este caso, tomando como ejemplo el C_v .

- En cada bloque de datos calculamos $\langle e^2 \rangle_i$ y $\langle e \rangle_i$, donde el índice i , indica el bloque de medidas donde ha sido calculada la media. En este bloque calculamos $C_v^i = 2V(\langle e^2 \rangle_i - \langle e \rangle_i^2)$
- Finalmente dispondremos de tantos valores de C_v^i como bloques de medidas. Con todas estas medidas podemos calcular su error, pues es ya una medida individual.

Insistimos en que el valor correcto de C_v es el obtenido con el valor correcto de $\langle e^2 \rangle$ y de $\langle e \rangle$ obtenidos con TODAS las medidas. C_v NO se obtiene como la media de los C_v^i como puede comprobarse inmediatamente. No obstante, dado que las cantidades $\{C_v^i\}$ tendrán una distribución gaussiana, su media no separará mucho del valor correcto; pero operacionalmente son cantidades bien diferentes.

Conviene dibujar el valor de C_v^i en función de i para comprobar que todos los datos fluctúan en torno a un valor central, sin derivas ni puntos fuera de control. También es interesante dibujar el error en función de tamaño del bloque usado, para ver a partir de qué tamaño el error es estable.

12.3. Análisis de Tamaño finito

Una vez hecho el cálculo en el retículo $L = 16$, podemos rehacer el análisis en torno al punto crítico con retículos mayores, por ejemplo $L = 24$ o $L = 32$. Ahora el tiempo de simulación será más largo por varias razones

1. Cada iteración de Monte Carlo tiene complejidad V , y por tanto el tiempo crecerá con este factor. Por ejemplo respecto de $L = 16$, el retículo $L = 32$ necesitará un factor 4 en tiempo de CPU para cada iteración
2. Ahora el proceso de termalización será más largo, serán necesarias mas iteraciones, lo que aumentará el tiempo
3. El tiempo de autocorrelación aumenta con L , de modo que para tener errores pequeños los tamaños de los bloques deberán ser mayores, lo que implica que es necesaria una mayor estadística.

Una vez obtenidos los resultados deberemos observar que el calor específico y la susceptibilidad presentan un pico en el punto crítico, que crecen notablemente al aumentar el tamaño de la red, de modo que obtenemos una divergencia en el límite termodinámico.

12.4. Test de Consistencia: Ecuaciones de Schwinger-Dyson

Como hemos comentado a lo largo de este texto, es necesario tras escribir cualquier código realizar test intensivos de consistencia; es decir, realizar cálculos de los que sepamos el resultado exacto para comprobar que nuestro código es correcto absolutamente. En el caso del modelo de Ising existe una forma de realizar un test de estas características, que se enmarca dentro de lo que en Teoría Cuántica de Campos se conocen como las Ecuaciones de Schwinger-Dyson.

La teoría para el caso más general escapa del alcance de este texto, pero una de sus aplicaciones para este simple modelo nos da una forma asequible y su demostración puede realizarse en base a conceptos conocidos.

Consideremos una integral en dos dimensiones de la forma

$$I_a = \int_{-\infty}^{\infty} dx dy e^{-x^2 - y^2 - xy}$$

Realizando el cambio de variable $x \rightarrow -x$, con jacobiano igual a -1 y teniendo en cuenta que el intervalo de integración cambia de sentido, obtenemos finalmente

$$I_b = \int_{-\infty}^{\infty} dx dy e^{-x^2 - y^2 + xy} = I_a$$

Podemos escribir la igualdad trivial

$$1 = \frac{I_b}{I_a} = \frac{\int_{-\infty}^{\infty} dx dy e^{-x^2 - y^2 + xy}}{\int_{-\infty}^{\infty} dx dy e^{-x^2 - y^2 - xy}}$$

Utilizando que $xy = -xy + 2xy$ en la exponencial del numerador,

$$1 = \frac{\int_{-\infty}^{\infty} dx dy e^{-x^2-y^2-xy} e^{2xy}}{\int_{-\infty}^{\infty} dx dy e^{-x^2-y^2-xy}}$$

Si entendemos I_a como la función Z , tendremos

$$1 = \frac{\int_{-\infty}^{\infty} dx dy e^{-x^2-y^2-xy} e^{2xy}}{Z} = \langle e^{2xy} \rangle$$

Consideremos ahora el modelo de Ising, y tomemos como variable x un cierto spin concreto, digamos s_q ; escribimos la función de partición separando del sumatorio el término que contiene s_q

$$Z = \sum_C e^{\beta(\sum_{n \neq q, \hat{\mu}} s_n s_{n+\hat{\mu}} + s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{q+\hat{\mu}})}$$

Ahora consideramos el cambio de variable $s_q \rightarrow -s_q$. Dado que el espacio de configuración no cambia (el cambio de variable hace recorrerlo de forma diferente, pero se recorre idénticamente), tendremos

$$\sum_C e^{\beta(\sum_{n \neq q, \hat{\mu}} s_n s_{n+\hat{\mu}} + s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{q+\hat{\mu}})} = \sum_C e^{\beta(\sum_{n \neq q, \hat{\mu}} s_n s_{n+\hat{\mu}} - s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{q+\hat{\mu}})}$$

y por tanto

$$1 = \frac{\sum_C e^{\beta(\sum_{n \neq q, \hat{\mu}} s_n s_{n+\hat{\mu}} + s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{q+\hat{\mu}} - 2 * s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{n+\hat{\mu}})}}{\sum_C e^{\beta(\sum_{n \neq q, \hat{\mu}} s_n s_{n+\hat{\mu}} + s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{q+\hat{\mu}})}}$$

y en términos de valores esperados tendremos,

$$1 = \langle e^{2\beta s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{n+\hat{\mu}}} \rangle$$

La igualdad anterior es cierta para cualquier spin s_q . Para ganar en estadística, y por tanto en precisión, podemos calcular esta expresión para todos los espines y promediar, es decir que tendremos también

$$1 = \frac{1}{V} \langle \sum_q e^{2\beta s_q \sum_{+\hat{\mu}, -\hat{\mu}} s_{q+\hat{\mu}}} \rangle$$

Esta es la cantidad que debemos implementar en nuestro código de medidas, y analizar debidamente para ver si su valor es compatible con 1 en el error.

Dado que la cantidad a medir es una exponencial, presenta fuertes fluctuaciones, y no es fácil de obtener valores con precisión; sin embargo en la fase de depuración del programa, es necesario hacer algún run con alta estadística para obtener errores pequeños que nos muestren que el programa está libre de fallos.

12.5. Método de la Densidad Espectral

Para localizar con precisión el valor donde se sitúa la transición de fase en un retículo finito, hemos mirado el máximo del calor específico o de la susceptibilidad, por ejemplo. En redes grandes encontrar este valor de β es difícil, dado que al ser los picos muy estrechos debemos simular en muchos puntos diferentes para localizarlo con precisión. Esto aumenta el tiempo de CPU necesario para las simulaciones y aumenta la incertidumbre en el cálculo.

Es posible sin embargo evitar este problema, usando un método que permite, dada una simulación en un valor de β , extraer los resultados a su entorno próximo; este método se conoce como de la Densidad Espectral.

Partimos de la expresión 10.14 donde tenemos la Función de Partición sumando sobre valores de la Energía, y la cuestión importante es que la dependencia en β de Z es simple,

$$Z(\beta) = \sum_E e^{-\beta E} N(E) \quad (12.3)$$

Podemos decir que para ese valor de β conocemos la distribución de probabilidad $p(E)$, recordando 10.16

$$p_\beta(E) \propto N(E) e^{-\beta E} \quad (12.4)$$

$p_\beta(E)$ no es más que el histograma que hemos calculado ya varias veces en el curso. La cuestión ahora es ¿Podemos calcular $p(E)$ para otro valor de β diferente al usado en la simulación partiendo de los datos anteriores? La respuesta es sencilla; ya que la dependencia en β es explícita, podemos escribir

$$p_{\beta+\delta}(E) \propto N(E) e^{-\beta E} e^{-\delta E} \quad (12.5)$$

Dado que el histograma calculado para β lo suponemos calculado por una simulación de Metropolis en ese valor, tendremos que

$$p_{\beta+\delta}(E) \propto p_\beta(E) e^{-\delta E}$$

Las constantes de proporcionalidad se fijan siempre de modo que $p(E)$ esté normalizada, y en este caso

$$p_{\beta+\delta}(E) = \frac{p_\beta(E) e^{-\delta E}}{\sum_E p_\beta(E) e^{-\delta E}}$$

Notar que en el exponente la energía que aparece es la extensiva.

Conocida la nueva distribución de la energía podemos calcular trivialmente valores medios que la contengan, en concreto,

$$\langle E \rangle_{\beta+\delta} = \frac{\sum_E E p_\beta(E) e^{-\delta E}}{\sum_E p_\beta(E) e^{-\delta E}}$$

En el proceso de simulación disponemos de una secuencia de $E_k, k \in [0, N]$ números para la Energía, distribuidos ya de acuerdo con $p_\beta(E)$. Por tanto, si pasamos a sumar sobre los valores calculados, estamos ya considerando el factor $p(E)$, y por tanto tendremos

$$\langle E \rangle_{\beta+\delta} = \frac{\sum_k E_k e^{-\delta E_k}}{\sum_k e^{-\delta E_k}}$$

Hemos obtenido pues la forma de calcular el valor de la Energía en un valor diferente al de la simulación.

Esto podemos hacerlo para otros observables. Consideraremos por ejemplo la magnetización.

Reescribamos las expresiones de la sección 10.1.1 con la dependencia explícita en m ; llamaremos M a la magnetización *extensiva*.

$$Z = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} = \sum_C e^{\beta \sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}}} \sum_E \delta(E - (-\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}})) \sum_M \delta(M - (\sum_n s_n)) \quad (12.6)$$

Podemos intercambiar el orden de los sumatorios,

$$Z = \sum_E \sum_M e^{-\beta E} \sum_C \delta(E - (-\sum_{n,\hat{\mu}} s_n s_{n+\hat{\mu}})) \sum_M \delta(M - (\sum_n s_n)) \quad (12.7)$$

y sumando sobre las configuraciones,

$$Z = \sum_E \sum_M e^{-\beta E} N(E, M) \quad (12.8)$$

En términos de probabilidad de encontrar una cierta energía y una cierta magnetización, ahora tendremos

$$p(E, M) \propto N(E, M) e^{-\beta E} \quad (12.9)$$

Si ahora disponemos de un fichero donde tenemos escritos la Energía y la Magnetización de cada configuración medida, etiquetadas con un índice k , entonces podemos calcular el valor de $\langle m \rangle$ en el nuevo valor de β calculando simplemente

$$\langle m \rangle_{\beta+\delta} = \frac{\sum_k m_k e^{-\delta E_k}}{\sum_k e^{-\delta E_k}}$$

dado que el factor $p(E, M)$ está contenido en la secuencia de números obtenida en el archivo debidamente termalizado.

Evitando los Overflows de la exponencial

Si realizamos el cálculo anterior usando la ecuación 12.5 tendremos un problema grave pues el ordenador no puede trabajar con números tan grandes. Efectivamente, para cada término del sumatorio debemos calcular

$$e^{-\delta E}$$

donde ya hemos remarcado que E es la energía extensiva. Esta cantidad es pues del orden del volumen del sistema. Para una red de tamaño común, como por ejemplo una red con $L = 16$, en el exponente tendremos números del orden de 256, y la exponencial de tal número es demasiado grande para el formato `float` o incluso `double`. Para evitar este problema, notemos que podemos escribir la expresión 12.5 como

$$\langle E \rangle_{\beta+\delta} = \frac{A \sum_k E_k e^{-\delta E_k}}{\sum_k A e^{-\delta E_k}}$$

con A constante y arbitraria. Tomemos $A = e^{\delta \langle E \rangle_\beta}$, y obtendremos

$$\langle E \rangle_{\beta+\delta} = \frac{\sum_k E_k e^{-\delta(E_k - \langle E \rangle_\beta)}}{\sum_k e^{-\delta(E_k - \langle E \rangle_\beta)}}$$

Ahora la cantidad $(E_k - \langle E \rangle_\beta)$ es mucho más pequeña que antes, en concreto del orden de la fluctuación de la Energía, que como sabemos es proporcional a la raíz del volumen, en lugar de al volumen como anteriormente.

Limitaciones del método

Formalmente el método anterior permite resolver el sistema para cualquier δ ; por tanto una vez realizada una simulación para un valor de β , podríamos extrapolar a todo valor de la temperatura y resolver completamente el problema. Sin embargo la situación es más compleja. Si conocieramos *exactamente* $p(E)$ para un cierto valor de β , ciertamente usando 12.5 podríamos calcular la Energía para cualquier otro valor; pero no conocemos $p(E)$ de forma exacta: el histograma lo conocemos a través de una simulación en el ordenador, y por tanto con un cierto error. El nuevo histograma para $\beta+\delta$ se obtiene del histograma para β , multiplicándolo por una exponencial $e^{-\delta E}$ (y normalizando después). Sobre la forma, en general gaussiana de $p(E)$, al multiplicarla obtenemos una gaussiana desplazada: valores que estaban muy lejos del centro de la gaussiana original se van convirtiendo en centrales; pero estos valores periféricos tenían un error, que quedan multiplicados por la exponencial; llega un momento que estos errores son tan grandes que los errores inducidos en las cantidades medias son gigantescos, haciendo que el cálculo no tenga sentido.

Esto se traduce en que el valor de δ debe ser pequeño; su valor máximo estará limitado por los errores en los valores medios.

Búsqueda de picos

Una vez desarrollado el método, vayamos a la parte numérica. En primer lugar debemos hacer una simulación en un valor de β lo más próximo al punto crítico, para que el valor de δ sea luego pequeño. Debemos escribir todos los resultados en un archivo, con las medidas individuales de la Energía y a su lado la de las cantidades básicas que queramos medir, como la magnetización.

En el programa de análisis leeremos los datos, y calcularemos el valor medio de la Energía extensiva para poder restar en el exponente. A continuación elegiremos un valor de δ y calcularemos los valores medios deseado. Luego otro valor de δ y así sucesivamente hasta recorrer un rango en torno al punto crítico estimado, hasta localizar con suficiente precisión donde está el punto crítico real para ese tamaño de red, localizando los máximos del Calor Específico, la Susceptibilidad u otros observables.

Los errores deben ser calculados correctamente, pues nos dirán si los valores de δ utilizados tienen o no sentido. Para ello el método más sencillo es utilizar el método de los bloques: dados unos datos totales, los dividimos en bloques, calculamos todas las cantidades para cada bloque, y con la dispersión de los resultados, podremos calcular el error.

12.6. Ejercicios

12.6.1. Programa de Análisis

Escribir el programa de análisis, que lea los archivos creados por el programa de simulación, y calcule los valores y los errores de los observables e, m, C_v, χ .

Dibujar los valores medios en cada bloque para ver su evolución con dicho tamaño. Comprobar la termalización dibujando también los valores medios de los bloques. Comprobar la termalización especialmente para C_v y χ .

En la línea de comandos se le debe pasar un índice con el fichero inicial y final para realizar el análisis, de forma que dejando sin analizar los primeros eliminamos los datos no termalizados. En la línea de comandos debemos pasar también del tamaño del bloque para realizar el análisis de errores. Debemos producir un archivo con todos los valores medios y sus errores. Además, en la fase de debugging, conviene escribir en otro fichero las medidas individuales de e y m para verificar visualmente la termalización. Podemos escribir sólo 1 de cada K medidas para no generar ficheros excesivamente grandes. El valor de K se debe dar en linea de comandos sólo durante la fase de debugging.

12.7. Problemas

12.7.1. Problema 1

Incorporar en el programa de análisis el cálculo de los histogramas de E y m , calculando también el error en cada punto.

12.7.2. Problema 2

El calor específico es exactamente

$$C_v = \frac{\partial \langle e \rangle}{\partial \beta} \quad (12.10)$$

Entonces en una gráfica de $\langle e \rangle$ frente a β , si calculamos su pendiente, debe coincidir con C_v . Comprobar que es así en las simulaciones.

12.7.3. Problema 3

Buscar en Internet valores del modelo, como β_c o $\langle E(\beta) \rangle$, etc. y comprobar que el programa los reproduce correctamente.

12.7.4. Problema 4

Modificar el programa para $d = 3$. Localizar el punto crítico y comparar con los datos de la literatura.

Capítulo 13

Simulated Annealing

La formulación de la Mecánica Estadística tiene su objetivo principal en el estudio de sistemas Físicos reales. Sin embargo, es suficientemente potente para ser aplicada en otros campos con objetivos muy diferentes. Introduciremos aquí el Método de *Simulated Annealing*, que permite la búsqueda de mínimos en sistemas altamente complejos.

Cuando tenemos un sistema con un elevado número de grados de libertad, con interacción entre todos ellos en forma compleja, y nos piden que encontremos el mínimo para cierta función de estas variables, la mayoría de los métodos de minimización presentan problemas de convergencia o de implementación. Es el caso en muchos problemas de optimización, donde se busca la solución óptima para un problema donde intervienen muchas variables.

Comenzaremos con un problema sencillo, para ilustrar el algoritmo. En la segunda parte del capítulo utilizaremos el mismo método en un problema bien diferente, con interés tanto en estudiós teóricos como aplicados.

13.1. Planteamiento del Problema

El problema es el siguiente:

Dada la matriz A hermítica, y el vector \vec{b} , encontrar el vector \vec{x} que cumpla $A\vec{x} = \vec{b}$.

Es un problema presente en muchas áreas de las matemáticas y la física, mucho más simple computacionalmente que resolver la inversa de A . Podría ser atacado con métodos convencionales como el de Paso Descendente o el de Gradiente Conjugado, que no consideraremos aquí.

El algoritmo de Simulated Annealing está basado en la Mecánica Estadística. La idea es buscar un función tal que tenga su mínimo en la solución correcta del problema planteado. Llamaremos Energía a esta función, cuya búsqueda en cada caso requiere, además del conocimiento del problema, un poco de experiencia e incluso de imaginación.

El problema a resolver, puede ser convertido en uno donde la solución se encuentra en la configuración dominante a Temperatura 0 de un sistema creado ad hoc. La configuración dominante a Temperatura 0 es la de mínima Energía, que nos da la solución al problema.

Si la E definida tuviera un *paisaje* suave y un único mínimo, bastaría comenzar con cualquier solución, ir cambiando ligeramente la misma y aceptar los cambios si la energía disminuye. Pensemos en una energía con una forma similar a una parábola, donde empezando en cualquier punto, si vamos cambiando siempre hacia Energías más bajas, llegaremos al mínimo sin ningún impedimento.

Sin embargo en los casos no triviales, con cierta complejidad, el *paisaje* de mínimos contiene miles o millones de ellos, y casi con toda probabilidad al ir disminuyendo la Energía caeremos en el mínimo relativo más próximo y no en uno absoluto que es lo que nos interesa. Para evitar este *atasco* en un mínimo local, lo que hacemos es permitir *fluctuaciones térmicas* al sistema, que recordando el Algoritmo de Metropolis, consiste en que también podemos admitir cambios que suban la energía. A temperaturas altas, aceptaremos estos cambios casi siempre, de modo

que iremos viajando casi libremente por todo el paisaje del modelo; a medida que bajemos la temperatura iremos buscando los mínimos del sistema, que probablemente ahora, serán los próximos al mínimo absoluto.

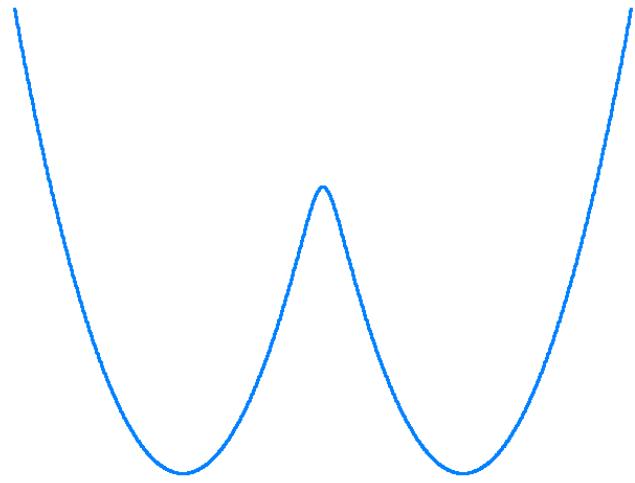


Figura 13.1: Energía libre en función de m en el modelo de Ising, como ejemplo de estructura simple

Muchos sistemas tienen un vacío sin estructura. Por ejemplo en el modelo de Ising, dada una configuración con energía E y magnetización m , la simetría del modelo nos dice que para esa misma energía, existe una configuración con magnetización $-m$. Esquemáticamente podemos dibujar la energía en función de m , obteniendo el resultado que puede verse en la figura 13.1. En

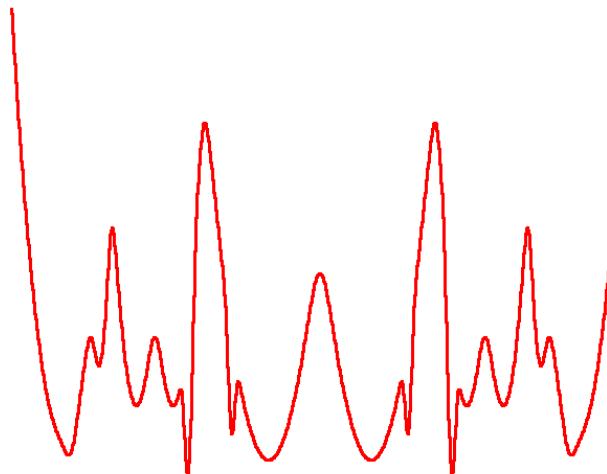


Figura 13.2: Paisaje de Energía libre en un sistema con estructura compleja

modelos más complejos, este paisaje de energías puede ser mucho más intrincado, por ejemplo del tipo mostrado en la figura 13.2

A la vista de una gráfica de este tipo, puede intuirse que recorrer los diferentes mínimos puede ser realmente difícil.

13.2. Método del Simulated Annealing: Caso General

Introducimos aquí los detalles más generales del método, usando el ejemplo en cuestión para facilitar el aprendizaje.

Recordamos que el problema que nos hemos planteado es:

Encontrar un vector \vec{x} , que dados A y \vec{b} cumpla que $A\vec{x} = \vec{b}$.

13.2.1. Construcción del Modelo

¿Cómo podemos convertir esto en un problema de Mecánica Estadística?

Los pasos son similares para todos los problemas a los que queremos aplicar este método y esencialmente son

1. **Espacio de configuración C :** Es el espacio de todas las variables dinámicas, que estarán indexadas con algún conjunto de etiquetas α discretas, continuas, numerables o no. Llamemos $\phi(\alpha)$ a las variables dinámicas; entonces $\phi(\alpha) \in C$.
2. **Configuración inicial:** Elegimos un punto del espacio de configuración para comenzar: si tenemos algún criterio para situarnos cerca el mínimo, es conveniente usarlo para acelerar la solución. En caso contrario elegimos un punto al azar.
3. **Definición de la Energía:** Construir E , función de todas las variables del problema (funcional), que presente un mínimo en la solución. A ser posible esta Energía debe tener un único mínimo, precisamente en la solución.
4. **Función de Partición:** Construimos un modelo de Mecánica Estadística, definiendo Z , integrando a todas las variables dinámicas, a una cierta Temperatura:

$$Z = \int d\phi(n) e^{-\beta E\{\phi(n)\}}; \quad \beta = 1/T$$

5. **Cambios tentativos:** Debemos definir como recorrer el espacio fase, es decir como ir recorriendo las diferentes configuraciones en el modelo para realizar la simulación de Monte Carlo. Supongamos que disponemos de un parámetro ϵ que nos sirve para graduar la intensidad de los cambios. Este punto es clave para el correcto funcionamiento del proceso. Nuestro método para realizar el movimiento a través de C debe cumplir varios requisitos

- **Ergodicidad:** El proceso de cambios aplicado directamente, sin usar para nada el algoritmo de Simulated Annealing, ha de permitir recorrer todo el espacio fase (Debe ser ergódico). En caso contrario, si dejáramos regiones sin recorrer, podríamos dejar fuera la solución correcta.
- **Cambios parametrizables:** El método debe permitir cambios parametrizables: debe ser posible realizar cambios controlados de modo que modifiquen mucho o poco la Energía según convenga. Esto debe ser así para poder recorrer a diferente velocidad el espacio fase: más rápido en las regiones muy lejanas de los mínimos, más despacio en las regiones próximas.

13.2.2. Proceso de Optimización

Una vez escrito correctamente el código, realizamos el proceso de optimización:

1. **Valor inicial de β :** Debemos dar un valor inicial a β . Si llamamos $\delta E(\epsilon)$ a la variación de la Energía típica al hacer un cambio tentativo usando un cierto valor del parámetro que controla la intensidad de los cambios (ϵ), β debe ser elegida de modo que al inicio de la simulación $\beta\delta E(\epsilon) \approx 1$. Para ajustar β podemos realizar unos pocos cambios tentativos y calcular el valor absoluto de la variación de energía media en ellos, es decir $\langle |E_{\text{new}} - E_{\text{old}}| \rangle$, y tomar $\beta_{\text{initial}} = \frac{1}{\langle |E_{\text{new}} - E_{\text{old}}| \rangle}$. Estrictamente sería mejor considerar el Calor Específico parcial y calcular β_{initial} a partir de él.

2. Protocolo de enfriamiento:

- Simulamos el Sistema a una cierta Temperatura hasta que el sistema termaliza
- Bajamos la Temperatura
- Iteramos los dos pasos anteriores hasta que la Energía permanezca prácticamente estable en torno a el valor más bajo.

3. Control de la aceptancia: Recordad la definición del capítulo anterior: la aceptancia es el porcentaje de cambios aceptados frente a intentos totales. Es necesario controlar la aceptancia para cada Temperatura. Una aceptancia muy baja indica que el sistema apenas se está moviendo, y por tanto estamos explorando muy lentamente el espacio de configuración. Esto es admisible sólo cuando estamos muy próximos al mínimo. En caso contrario, al principio de la simulación, si la aceptancia es baja debemos modificar los parámetros de cambio para permitir cambios tentativos mayores en el espacio de configuración. Un compromiso razonable es que al inicio la aceptancia debe situarse en torno al 50 %, y que al acercarnos al mínimo este número debe ir bajando para no alejarnos de él. Si la aceptancia es alta, quiere decir que la Energía cambia poco, o sea que estamos moviéndonos poco en el espacio de configuraciones; para disminuir la aceptancia podemos:

- Aumentar el valor de ϵ , en cuyo caso los cambios tentativos serán mayores, nos alejaremos más de la configuración inicial y por tanto, en general, los cambios en la Energía serán mayores.
- Aumentar β , es decir, disminuir la Temperatura: esta disminución hará que solo aceptemos cambios con una variación muy pequeña de la Energía, lo que disminuirá la aceptancia.

4. Estadística y precisión: Podemos utilizar varios criterios para decidir cuando finalizamos la simulación. El más simple es fijar un determinado número de iteraciones de Monte Carlo, lo que fija el tiempo de CPU. Otros métodos más sofisticados son simular hasta que la solución no cambie prácticamente, o cuando la aceptancia sea muy baja.

5. Mínimos locales: Tratamos de asegurarnos de que no hay otras soluciones: otros mínimos con menor valor para la Energía, o incluso soluciones degeneradas con el mismo valor. Si partimos de una configuración fija y realizamos diferentes simulaciones con una secuencia random diferente, en general la solución final será diferente. Sin embargo, es muy probable que aunque haya muchas soluciones diferentes, si partimos del mismo punto el mínimo encontrado sea el mismo, pues en un sistema con muchos mínimos no es fácil explorarlos todos, y caeremos en un mínimo próximo al vector de partida, aunque sea un mínimo local y no absoluto. Por ello es conveniente realizar diversos intentos partiendo de puntos lo más alejados posibles en el espacio de configuración para explorar mejor el sistema.

6. Verificación: Comprobar explícitamente que el resultado obtenido es una solución correcta del problema. Esto es un requisito importante, que a veces se da por evidente sin control alguno, con resultados desastrosos.

Si todo ha funcionado apropiadamente, al llegar a Temperaturas muy bajas (0 eventualmente) el sistema busca la *configuración con mínima energía*, que es la que nos da la solución al problema.

Hay que remarcar que en sistemas complejos, es difícil encontrar una función Energía de la que sepamos a priori cuál es su valor en el mínimo absoluto. Y sin saber esto no podemos garantizar que la solución encontrada sea la correcta; será un mínimo local pero tal vez no global.

13.3. Resolución de $A\vec{x} = \vec{b}$

En este caso particular usado como ejemplo, el procedimiento esbozado anteriormente se concreta del siguiente modo.

13.3.1. Construcción del modelo

1. **Espacio de configuración:** Cada valor de $\phi(\alpha) \equiv \vec{x} \in \mathbb{R}^n$ es una posible solución, por tanto $C = \mathbb{R}^n$.
2. **Configuración inicial:** Elegimos un vector inicial al azar, con sus componentes en el intervalo $[-1, 1]$ por ejemplo.
3. **Definición de la Energía:** Buscamos una energía que nos indique las soluciones correctas. Una opción simple es $E = |A\vec{x} - \vec{b}|^2$. Cumple que E tiene un mínimo en la solución y sólo en la solución y además sabemos que en el mínimo y sólo en él $E = 0$. Puede haber varias soluciones, si la matriz es singular, pero todas ellas serán válidas. Notar que podríamos elegir también otras energías como por ejemplo $E = |A\vec{x} - \vec{b}|$ o incluso funciones más complejas como $E = \sin(|A\vec{x} - \vec{b}|^2)$. Elegimos aquí la más simple y que es además analítica.
Nota: En el caso de elegir $E = |A\vec{x} - \vec{b}|$, ¿sería E una función analítica?
4. **Función de Partición:** Definimos $Z = \int_{\mathbb{R}^n} d\vec{x} e^{-\beta E\{\vec{x}\}}$
5. **Cambios tentativos:** Usaremos varios métodos para el cambio tentativo del vector \vec{x} . Cumplimos con los requisitos de ergodicidad y control sobre el cambio en la energía.
 - El cambio básico consiste en generar un número aleatorio plano $\omega \in [-\epsilon, \epsilon]$ a una componente del vector, es decir $x_i^{new} = x_i^{old} + \omega$. Notar que si eligieramos $\omega \in [0, \epsilon]$ el proceso no sería ergódico.
 - Se van cambiando más o menos componentes del vector, elegidos a veces random, a veces secuencialmente
 - El valor de ϵ se va modificando con la evolución del programa.

13.3.2. Proceso de Optimización

1. **Valor inicial de β :** En este caso concreto, el valor depende del tamaño de la Matriz, de su forma, y también del valor inicial del parámetro ϵ utilizado para los cambios tentativos del vector \vec{x} . Debemos controlar ambos números para que inicialmente se cumpla la condición $\beta\delta E(\epsilon) \approx 1$. Como hemos dicho en el caso general, sería mejor considerar el Calor Específico parcial y calcular β_{inicial} a partir de él. Dejamos esto como modificación para el alumno que lo desee. En cualquier caso este ajuste para la β de partida se hace sólo una vez al inicio de la ejecución, realizando unos cuantos cambios en la configuración de partida.
2. **Protocolo de Enfriamiento:** Realizamos dos bucles. Fijamos la Temperatura y ϵ a un valor, y realizamos un Simulación de Monte Carlo de modo que el sistema se aproxime al equilibrio. Este es el bucle interno. A continuación cambiamos la Temperatura y ϵ y repetimos la Simulación de Monte Carlo. Este es el bucle externo. En este caso el cambio de T y ϵ en el bucle externo se hace así $\beta = \beta + \delta_\beta$, $\epsilon = \epsilon + \delta_\epsilon$
3. **Control de la aceptancia:** Para que el algoritmo funcione correctamente, la aceptancia debe ser alta (del orden del 50% al principio y más baja al final, cuando estemos en el entorno del mínimo). La aceptancia, para un problema fijo, depende de los valores de β y ϵ , que deberán irse cambiando a lo largo de la simulación. Estos cambios pueden realizarse de diversos modos y con diferentes parámetros. Pueden modificarse aditivamente, multiplicativamente o con otras operaciones. Supongamos que los cambios son aditivos, definiendo entonces sus incrementos como $\delta_\beta, \delta_\epsilon$. Los cambios en β, ϵ pueden ser fijados de antemano o dinámicos dependiendo de los resultados de la simulación. El valor de δ_ϵ puede elegirse ajustando la aceptación a valores en torno al 50% inicialmente. El valor de δ_β lo podemos elegir en función del tiempo de CPU de que dispongamos; si tenemos mucho tiempo, δ_β puede ser baja y recorreremos el espacio fase de forma lenta, explorando bien

todos los mínimos. Si δ_β lo hacemos grande tendremos pronto una solución pero puede estar lejos de la correcta.

4. **Estadística y precisión:** Debemos contorlar el número de iteraciones total del algoritmo, que puede ser fijo o prolongarse hasta una precisión dada. En este ejemplo fijar la precisión es trivial, pues sabemos que la solución está en $E = 0$; por tanto podemos detener el programa cuando la energía alcance un valor por debajo de uno prefijado.
5. **Mínimos locales:** Podemos partir de vectores construidos de muy diferente manera para tratar de encontrar otros mínimos, repitiendo todo el proceso para cada valor inicial. En este problema la Energía nos indica si el mínimo es Absoluto; pero puede haber muchos mínimos absolutos si la matriz es singular.
6. **Verificación:** Comprobamos que la solución es correcta, en este caso es simple: comprobamos si $A\vec{x} = \vec{b}$. Para ello basta escribir en la pantalla la componente $(A\vec{x})_i$ y a su lado \vec{b}_i ; esto es exhaustivo; una cantidad que integra toda la información de la validez de la solución es en este caso la Energía, que debe ser casi cero si hemos encontrado la solución correcta.

13.4. El problema del viajante

Tras habermos familiarizado con el método con un problema sencillo, nos podemos enfrentar a uno un poco más complejo. El problema se plantea en los siguientes términos,

Un viajante debe pasar por N lugares en el plano, una y sólo una vez. Nos preguntamos cuál es la trayectoria más corta que cumple este requisito.

Este sistema ha atraído gran interés desde el punto de vista teórico y computacional, pues su complejidad crece exponencialmente con N , el número de puntos. Es lo que se conoce como un problema NP completo.

13.5. Construcción del modelo

Para resolver el problema no debemos tratar de pensar el problema globalmente y resolverlo *de golpe*. Debemos aplicar estrictamente los pasos indicados en las secciones anteriores; la solución de cada uno de ellos, nos permitirá ir adquiriendo una visión global del sistema y plantear un algoritmo y un código adaptado al problema. Por tanto ataquemos cada uno de los pasos.

La formulación más concreta es:

Consideremos N puntos, $\{\vec{r}_0, \vec{r}_1, \dots, \vec{r}_{N-1}\}$, situados sobre el plano. Definimos la distancia Euclídea usual. Tomamos un punto al azar y trazamos una trayectoria que recorra todos los puntos, una y sólo una vez, volviendo al punto de partida. Calculamos finalmente la distancia total de la trayectoria. Debemos encontrar la trayectoria que hace mínima dicha distancia.

Aplicamos la misma estrategia anterior. Ahora, estos pasos se concretan así:

1. **Espacio de configuración:** El espacio es el de todas las trayectorias. Cada punto de dicho espacio es una trayectoria. La cuestión es como formalizamos esto matemáticamente. Para ello debemos pensar qué información debemos dar para individualizar una trayectoria concreta.

Lo más simple sería decir: “Primero empezamos en el punto 8, viajamos al punto 23, luego al 3, luego al 5, ... luego al 83 y finalmente volvemos al 8”.

Una forma más compacta de dar esta información es escribir los puntos recorridos como un vector: $\{8, 23, 3, 5, \dots, 83, 8\}$. En realidad el último punto no es necesario ponerlo, pues debe ser siempre el primero, por tanto podemos escribir $\{8, 23, 3, 5, \dots, 83\}$. Este vector es una trayectoria, es un punto del espacio de configuraciones.

El punto clave es : este vector es una permutación del vector $\{0, 1, 2, \dots, N - 2, N - 1\}$; por tanto

cada punto del espacio de configuración es una permutación de N elementos.

El espacio de configuraciones es el todas las permutaciones de un vector de N componentes que llamaremos $P(N)$, y tiene por tanto $N!$ elementos, cada uno de los cuales lo llamaremos τ , es decir $\tau \in P(N)$. En realidad sin perder generalidad podemos empezar y terminar siempre en un punto fijo dado, con lo cual tenemos $N - 1$ puntos a tratar.

- Consideremos un ejemplo para aclarar el papel de los índices. Podemos escribir una trayectoria o bien indicando los índices, o también indicando los puntos de la trayectoria en el plano. Esto último es más conveniente para calcular luego las distancias. Escribimos la trayectoria de la forma

$$\{\vec{r}_{k(0)}, \vec{r}_{k(1)}, \vec{r}_{k(2)}, \dots, \vec{r}_{k(N-1)}\}; k(N) \equiv k(0) \quad (13.1)$$

Pongamos un ejemplo. Supongamos que los puntos son recorridos en el orden

$$\{3, 5, 8, 0, \dots\} \quad (13.2)$$

Esto correspondería con

$$k(0) = 3, k(1) = 5, k(2) = 8, k(3) = 0, \dots \quad (13.3)$$

2. **Configuración inicial:** Elegimos un estado inicial; tenemos varias opciones

- Una permutación al azar
- La permutación cero: $\{0, 1, 2, \dots, N - 2, N - 1\}$
- Empezamos en el 0, y elegimos el siguiente aquel que esté más próximo en el plano; a continuación, de entre los que quedan sin elegir, el más próximo al elegido y así sucesivamente. **Nota:** ¿Cuál es la complejidad de este proceso?. O dicho de otro modo, ¿cómo depende de N el número de pasos de el mejor algoritmo para hacer esto?

Conviene aclarar que con una visión simplificada, uno podría pensar que el camino más corto se alcanza siempre usando el algoritmo anterior: ir siempre al más próximo. De hecho es lo que hacemos habitualmente en nuestras vidas. Sin embargo en general este algoritmo no da la mejor solución. Basta mirar la Figura 13.3. En ella los puntos están separados en la dirección horizontal por dos unidades y en la vertical por una unidad. La parte superior es el camino más corto (distancia recorrida: 14 unidades). La parte inferior es el resultado de movernos cada vez al más próximo, que como es evidente, no es el mejor camino (distancia recorrida: 16 unidades).

3. **Definición de la Energía:** Definimos ahora la distancia, es decir la Energía :

$$E \equiv d = \sum_{j=0}^{N-1} \|\vec{r}_{k(j)} - \vec{r}_{k(j+1)}\|; k(N) \equiv k(0)$$

En este caso, en trayectorias no triviales, no tenemos ningún criterio claro para saber si hemos encontrado la solución óptima; por tanto cuando encontramos un mínimo existirá la posibilidad de que exista otro mínimo con Energía menor. Recordar que

$$\|\vec{r}_a - \vec{r}_b\| = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

4. **Función de Partición:** Definimos

$$Z = \sum_{\tau \in P(N)} e^{-\beta E\{\tau\}} = \sum_{\tau \in P(N)} e^{-\beta \sum_{j=0}^{N-1} \|\vec{r}_{k(j)} - \vec{r}_{k(j+1)}\|}$$

Nota: Caben otras muchas definiciones de Energía, pero que en general no responderían a la pregunta original de recorrer la mínima distancia Física. Por ejemplo, tratando de realizar

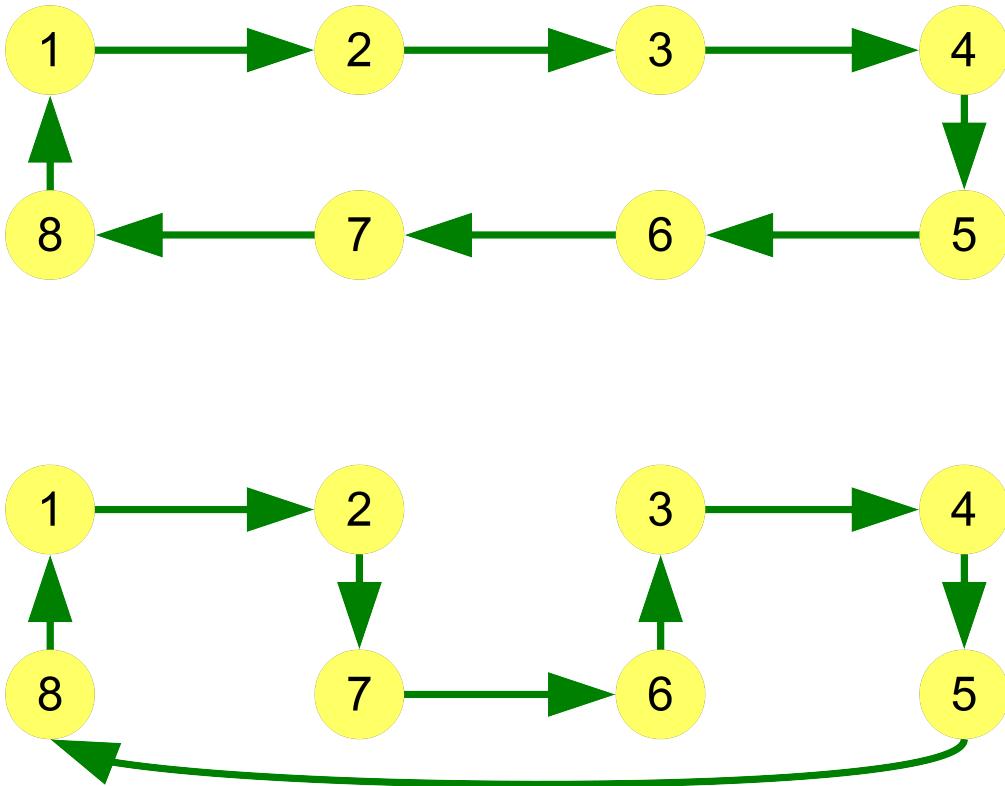


Figura 13.3: Los 8 puntos se sitúan según su posición en el plano. En la parte superior mostramos el camino óptimo. En la parte inferior el camino de usar el algoritmo de ir siempre al punto más cercano. Empezamos en el punto 8 en ambos casos.

menos cálculos para acelerar la ejecución, se podría pensar en eliminar la raíz cuadrada y definir,

$$E = \sum_{j=0}^{N-1} \|\vec{r}_{k(j)} - \vec{r}_{k(j+1)}\|^2; k(N) \equiv k(0)$$

Comprobar con algún ejemplo que en este caso, trayectorias sobre el plano con mayor distancia recorrida, darían un menor valor para esta Energía.

5. **Cambios tentativos:** Realizar un cambio ahora es pasar de una permutación a otra $\tau \rightarrow \tau^{tentativa}$, en definitiva cambiar la trayectoria. Debemos realizar cambios que permitan ajustes locales, y también deshacer *nudos* en las trayectorias. Si el cambio es completamente al azar, los cambios en la energía serán gigantescos y tal vez al inicio, con la temperatura muy alta podamos aceptar algún cambio, pero no serán cambios que nos acerquen al mínimo. Debemos pues realizar cambios que sean relativamente pequeños. Dado que esto es complejo, volveremos sobre ello más adelante.

13.6. Proceso de Optimización

1. **Valor inicial de β :** Depende del número de puntos y de la amplitud de los cambios tentativos. Debemos aplicar las ideas generales, controlando la aceptancia inicial para que se sitúe en torno al 50 %. Esto nos permitirá fijar el valor inicial de β .
2. **Protocolo de enfriamiento:** En este caso el único parámetro que tenemos que controlar es β . De nuevo realizamos dos bucles. Fijamos la Temperatura a un valor, y realizamos un Simulación de Monte Carlo de modo que el sistema se aproxime al equilibrio. Este es

el bucle interno. A continuación cambiamos la Temperatura y repetimos la Simulación de Monte Carlo. Este es el bucle externo. El cambio de T podemos hacerlo como $\beta = \beta + \delta_\beta$.

3. **Control de la aceptancia:** En el transcurso de la simulación debemos controlar la aceptancia para verificar que el proceso es eficiente. El término δ_β indicado anteriormente puede ser calculado de forma dinámica de modo que la aceptancia vaya bajando poco a poco, pues en este problema, al final, una vez alcanzada la solución que suponemos óptima, la aceptancia debe ser prácticamente nula.
4. **Estadística y precisión:** Ahora no disponemos de un criterio para saber si la solución es la óptima. Podemos por tanto fijar el tiempo de CPU o algún otro criterio como que el sistema ya no evolucione.
5. **Mínimos locales:** Podemos partir de permutaciones iniciales diferentes y repetir todo el proceso de optimización, para tratar de encontrar otros mínimos. Esto es doblemente importante en tanto que en este caso no sabremos si la solución encontrada es la mejor. Partiendo de permutaciones diferentes exploraremos regiones del espacio de configuración diferentes.
6. **Verificación:** Deberemos realizar una visualización gráfica de la solución para cerciorarnos de que la solución es razonable. Con `gnuplot` es sumamente sencillo dibujar las trayectorias.

13.7. Generación de Cambios Tentativos

Supongamos que tenemos 15 puntos. La permutación identidad es pues

$$P_0(15) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}. \quad (13.4)$$

Aunque los puntos no sabemos en que posición están, a efectos lógicos, podemos dibujarlos como en la Figura 13.4.

El primer cambio, que llamaremos Tipo A, se obtiene cortando dos links y uniendo los puntos de forma cruzada; gráficamente, partiendo de la figura 13.4, el resultado puede verse en la figura 13.5.

Esto corresponde a elegir un segmento dentro de la permutación e invertir sus elementos, es decir:

$$\begin{aligned} P_0(15) &= \left\{ 0, 1, 2, 3, 4, \overbrace{\mathbf{5, 6, 7, 8}}^{\text{segmento}}, 9, 10, 11, 12, 13, 14 \right\} \\ P_A(15) &= \left\{ 0, 1, 2, 3, 4, \overbrace{\mathbf{8, 7, 6, 5}}^{\text{segmento}}, 9, 10, 11, 12, 13, 14 \right\} \end{aligned} \quad (13.5)$$

El segundo cambio, que llamaremos Tipo B, se obtiene tomando un segmento e insertándolo en otro lugar, para lo cual es necesario cortar tres links, como puede verse en la figura 13.6.

En términos de la permutación, podemos verlo así,

$$\begin{aligned} P_0(15) &= \left\{ 0, 1, 2, 3, 4, 5, 6, 7, \overbrace{\mathbf{8, 9, 10, 11, 12}}^{\text{segmento}}, 13, 14 \right\} \\ P_B(15) &= \left\{ 0, 1, 2, 3, \overbrace{\mathbf{8, 9, 10, 11, 12}}^{\text{segmento}}, 4, 5, 6, 7, 13, 14 \right\} \end{aligned} \quad (13.6)$$

Podemos hacer también una combinación de ambos cambios de forma consecutiva. Esto es conveniente pues podría ser que hubiera “nudos” en el sistema que fueran difíciles de deshacer con cambios Tipo A o Tipo B, si hubiera barreras grandes de Energía, pero que con un cambio simultáneo, la barrera de Energía fuera mucho menor. El cambio en concreto sería invertir un segmento y luego moverlo a otro lugar.

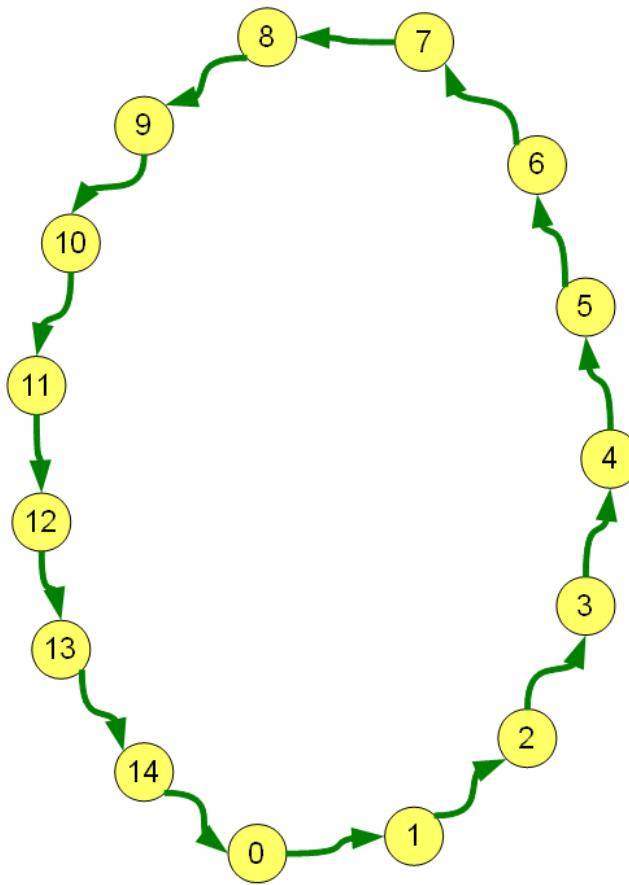


Figura 13.4: Una posible configuración inicial del recorrido del viajante. La posición en el plano aquí es *lógica*, no física.

En todos los casos, para realizar el cambio debemos en primer lugar elegir el segmento a cambiar. Para ello debemos elegir por ejemplo el punto inicial y final. Si es tipo A debemos permutarlos. Si es tipo B, debemos elegir al azar otro punto donde insertar el segmento.

Todas estas elecciones deben hacerse con atención para que sean correctas. Conviene en la primera fase de escritura del programa, generar cambios y escribir en la pantalla la permutación inicial y final y comprobar con atención que todo es correcto.

Nota: No se da el código para este problema, que debe ser escrito íntegramente por el alumno como se indica en el Ejercicio 13.8.2, como un prueba de autoevaluación de conocimientos adquiridos durante todo el curso.

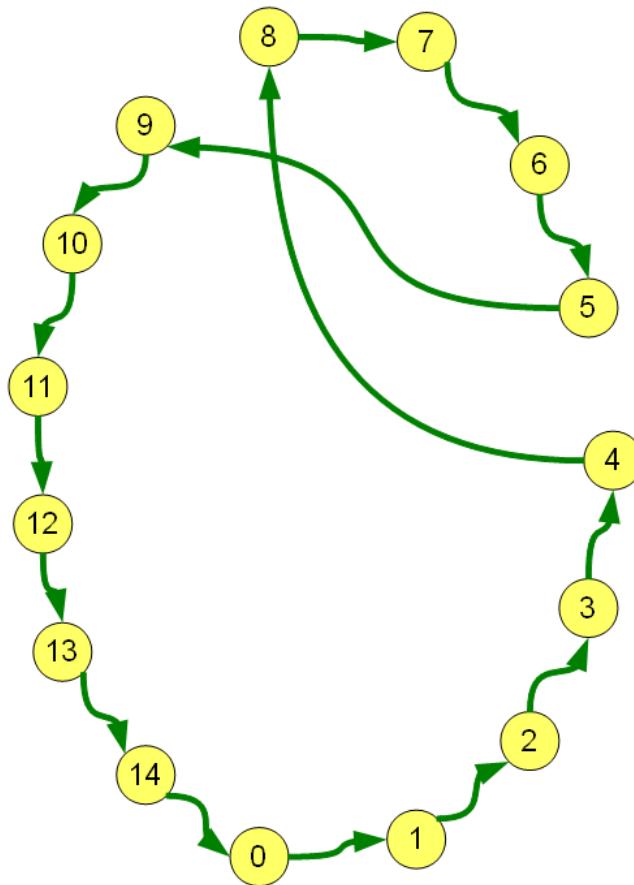


Figura 13.5: Ejemplo de cambio tentativo permutando un segmento del viaje

13.8. Ejercicios

13.8.1. Resolución de $A\vec{x} = \vec{b}$

Escribir el código para resolver la ecuación $A\vec{x} = \vec{b}$, debidamente estructurado en funciones elementales. Comenzar con matrices de tamaño $N = 2$ y de las que sepamos la solución, para comprobar que todo funciona correctamente. Pasar a matrices del orden de $N = 100$. Realizar diferentes test de consistencia, utilizando matrices para las que sepais el resultado, como por ejemplo matrices ortogonales, matrices diagonales, etc. En todos los casos, comprobar que la solución es correcta escribiendo al final de la ejecución los vectores Ax y b para ver si son aproximadamente iguales. Finalmente el algoritmo debería funcionar con $N \approx 1000$.

El código que mostramos es básico para orientar al alumno, y no contiene los detalles discutidos en el texto. El alumno debe implementar la teoría presentada y escribir un código compacto, estructurado, con comentarios, etc. Es muy recomendable que el alumno comience la escritura del código partiendo del desarrollado anteriormente para el ciclo de Histéresis del Modelo de Ising, muy similar en estructura al de este ejercicio.

13.8.2. Problema del Viajante

Escribir el código para resolver el problema del viajante con el algoritmo del Simulated Annealing. Es posible aprovechar gran parte del código del Ejercicio 13.8.1 anterior.

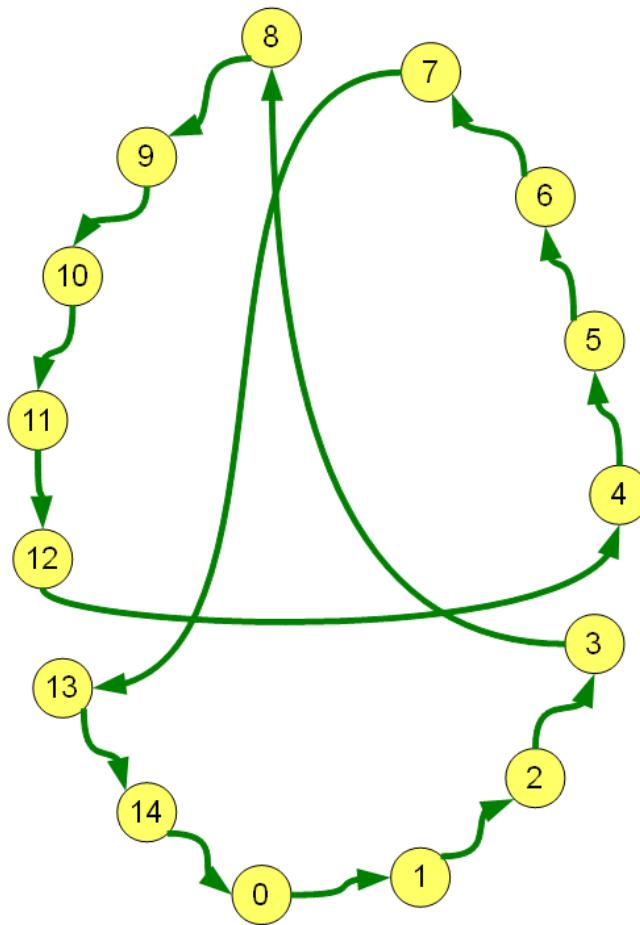


Figura 13.6: Ejemplo de cambio tentativo, intercambiando dos segmentos del viaje

En la fase de debugging conviene trabajar con configuraciones de puntos de las que sepamos la solución. Por ejemplo si construimos un polígono regular de 8 lados, el camino más corto es precisamente el octógono. Podemos repetir esto para N puntos situándolos sobre una circunferencia, donde de nuevo la solución es evidente. Ejecutar el programa partiendo de una trayectoria random para ver si se alcanza la solución óptima. Otra forma simple de comprobar si la solución es la correcta es dibujar con `gnuplot` la trayectoria final. Tras estas pruebas se deben utilizar caminos con un número de puntos $N \approx 1000$ para verificar que todo está ajustado correctamente. Finalmente dibujar la trayectoria para ver si por inspección la solución es correcta, dado que no tenemos un criterio absoluto para saber si la solución es la óptima.

13.9. Problemas

13.9.1. Problema 1

Para el caso $Ax = b$ dibujar la evolución de la Energía para estudiar la convergencia para diferentes matrices A y vectores b .

13.9.2. Problema 2

Modificar el algoritmo para usar matrices complejas en la resolución de $Ax = b$.

13.9.3. Problema 3

En el Problema del Viajante escribir en un fichero la evolución de las cantidades relevantes, como la Energía y la temperatura y dibujarlas posteriormente con `gnuplot`. Dibujar también la trayectoria final.

13.9.4. Problema 4

En el Problema del Viajante ejecutar el programa con diferentes valores de N . Modificar parámetros, tipos de cambios, valores de β , para lograr una mayor eficiencia para grandes valores de N (del orden de $10^3 - 10^4$)

13.10. Código en C

13.10.1. Simulated Annealing para $Ax = b$

```
/*
    Resolvemos por Metropolis-Monte Carlo
    la ecuacion Ax=b
    No es muy eficiente, pero da una aproximacion
    razonable casi siempre
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 100
#define Random_C (rand()/(double)RAND_MAX+1)

void Gradiente(double *p,double *q, double *r,int n);
double Norma(double *p,int n);
void A_por_vector(double *p,double *q,int n);
void construye_matriz(void);
void inicia_vector(double *p,int n,int flag);
void genera_cambio(double *p,double *q,int n,int k);
void copia(double *p,double *q,int n);

FILE *fout;
double A[N][N];
double epsilon;

int main()
{
    double x[N],b[N],v[N],g[N],x_n[N];
    double Resto,Resto_n,beta,Ener,delta_beta,delta_epsilon;
    int i;
    int tot,acept,iter,inter,ITER,INTER;

    construye_matriz();
    inicia_vector(b,N,1);
    inicia_vector(x,N,0);
    copia(x,x_n,N);

    beta=1.0*sqrt((double)N);
    epsilon=0.1/sqrt((double)N);

    ITER=300;
    INTER=400;
    delta_epsilon=-epsilon/(ITER+1);
    delta_beta=beta/ITER;

    fout=fopen("evol.dat","wt");

    for(iter=0;iter<ITER;iter++)
    {
        tot=acept=0;
        beta+=delta_beta;
        epsilon+=delta_epsilon;

        for(inter=0;inter<INTER;inter++)
        {
            A_por_vector(x,v,N);      // v=Ax
            Gradiente(b,v,g,N);      // g=-b+Ax
            Resto=Norma(g,N);        // Resto={\text{Resto}}\text{g}^{\text{2}}
            genera_cambio(x,x_n,N,iter);

            A_por_vector(x_n,v,N);   // v=Ax_n
            Gradiente(b,v,g,N);      // g=-b+Ax_n
            Resto_n=Norma(g,N);       // Resto=|g|^{\text{2}}
        }
    }
}
```

```

Ener=beta*(Resto_n-Resto);

#ifndef DEBUG
    printf("beta=%f,(%d,%d) Resto=%f,Resto_n=%f,Ener=%f\n",
           beta,tot,acept,Resto,Resto_n,Ener);
#endif

tot++;

if(exp(-Ener)>ran) // acepta el cambio
{
    copia(x_n,x,N);
    Resto=Resto_n;
    acept++;
}
fprintf(fout,"%f %d %d %f\n",beta,tot,acept,Resto); //Dibujar con gnuplot
printf("b=%f e=%f Resto=%f\n",beta,epsilon,Resto);
fclose(fout);

A_por_vector(x,v,N); //v=Ax

for(i=0;i<N;i++)
    printf("b=%f ,Ax=%f\n",b[i],v[i]);

return 0;
}

void escribe(double *p,int n)
{
    int i;

    for(i=0;i<n;i++)
        printf("v[%d]=%f\n",i,p[i]);
}

void genera_cambio(double *p,double *q,int n,int k)
{
    int i,j,n_pas;

    if((k%4)==0)
        n_pas=n/10;
    else
        n_pas=n;

    if(n_pas<=0)n_pas=1;

    for(i=0;i<n_pas;i++)
    {
        if((k%4)==0)
            j=rand()%n;
        else
            j=i;
        q[j]=p[j]+epsilon*(0.5-ran);

        // printf("Cambiado x[%d]=%f,x_new[%d]=%f\n",j,p[j],j,q[j]);
    }
}

void copia(double *p,double *q,int n)
{
    int i;

    for(i=0;i<n;i++)
        q[i]=p[i];
}

void Gradiente(double *p,double *q, double *r,int n)

```

```

{
    int i;

    for(i=0;i<n;i++)
        r[i]=-p[i]+q[i];
}

double Norma(double *p,int n)
{
    int i;
    double sum;

    sum=0;
    for(i=0;i<n;i++)
        sum+=(p[i]*p[i]);

    return sum;
}

void A_por_vector(double *p,double *q,int n)
{
    int i,j;

    for(i=0;i<n;i++)
    {
        q[i]=0;
        for(j=0;j<n;j++)
            q[i]+=A[i][j]*p[j];
    }
}

void construye_matriz(void)
{
    int i,j;

    for(i=0;i<N;i++)
        for(j=i;j<N;j++)
    {
        A[i][j]=i+j+ran;
        A[j][i]=A[i][j]; // es redundante en la diagonal, pero no hace daño
    }
}

void inicia_vector(double *p,int n,int flag)
{
    int i;

    if(flag==1)
        for(i=0;i<n;i++)
            p[i]=i;
    else
        for(i=0;i<n;i++)
            p[i]=(double)ran;
}

```

Capítulo 14

Redes Complejas

14.1. Introducción

Hacia la tercera década del Siglo XX se desarrollaron los Sociogramas, donde se construían gráficas con nodos representando a personas, y líneas que unían a personas que tenían algún tipo de relación (amistad, dependencia, liderazgo, ...). Estos estudios se situaban en el marco de la sociología principalmente, y se aplicó a las relaciones entre estudiantes, grupos de trabajo y en general grupos de unas decenas de personas accesible a las tecnologías disponibles. La disciplina avanzó pero de forma tímida en las siguientes décadas. En 1967 se produjo un salto cualitativo con el experimento de Stanley Milgram: unas cuantas personas debían enviar una carta a una persona concreta, pero desconocida para ellos, de la que sólo sabían su nombre, su ocupación y su posición aproximada. Debían enviar la carta a algún conocido suyo que pensaran que podría tener relación con la persona buscada. La pregunta del experimento era ¿por cuántas personas pasará la carta antes de llegar a su destino? Otra manera de formular la pregunta es saber cuántos amigos has de pasar para llegar a cualquier persona del mundo. La respuesta fue sorprendente: solamente hacían falta entre 5 y 6 intermediarios. Este experimento se ha reproducido de diferentes maneras, y la conclusión es básicamente la misma. La investigación tuvo importantes conclusiones sobre cómo se veía el mundo de las relaciones entre personas, los modelos válidos para describir las relaciones humanas, las matemáticas subyacentes, y a partir de ello emergió la idea de *Aldea Global*, tan conocida en la actualidad.

Los Físicos por su parte venían desarrollando modelos para el estudio de Fenómenos Colectivos; los ejemplos clásicos son el modelo de Ising, las transiciones de fase, excitaciones topológicas, etc. Más allá de la Mecánica Estadística se empezaron a usar las ideas de fenómenos colectivos para el estudio de ruptura de Fallas y Terremotos, propagación de enfermedades, pilas de arena y un largo etc.

La eclosión de las Redes Sociales tipo *Facebook* o *Twitter* ha permitido establecer redes amplias y sobre todo fácilmente analizables con herramientas matemáticas; de nuevo se ha constatado esta distancia en torno a 5 pasos para llegar de uno a cualquier otro en el mundo.

Fue la aparición y uso masivo de las Redes Sociales lo que impulsó definitivamente estos estudios de Redes Complejas, al tener una aplicación sociológica importante, poder realizar estudios con millones de participantes, testear modelos y aportar también ideas y herramientas en campos puramente comerciales como el marketing, comportamientos comerciales, ventas, formación de comunidades, fidelización de clientes, La Física en torno a estas disciplinas salió de estudios más básicos para introducirse en terrenos más aplicados, configurando una pujante comunidad científica internacional en Redes Complejas.

En el libro de M.E.J. Newman puede encontrarse un excelente texto. Un resumen más compacto y profundo en el reporte escrito por Yamir Moreno *et. al.*

14.2. Definiciones

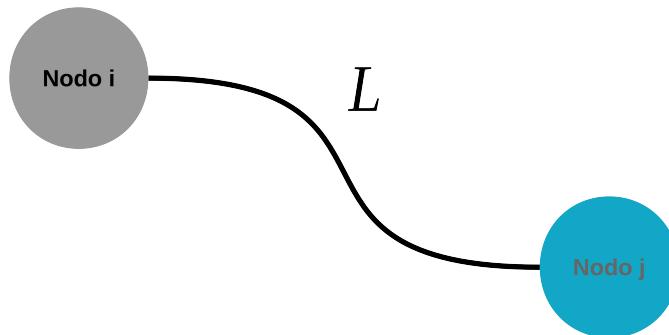


Figura 14.1: Red no dirigida formada por dos Nodos y una Link

En la base de la Redes están los *Nodos* y las *Links*.

Entendemos por *Nodos* los elementos que constituyen la Red, y las *Links* las relaciones que se establecen entre nodos.

Pongamos un ejemplo sencillo. Consideremos como Nodos un conjunto de alumnos de una Universidad. Establezcamos una relación entre ellos, diciendo que dos alumnos están relacionados (tienen una Link entre ellos) si han tenido algún profesor en común. Esta Red tiene la propiedad de que la relación entre ambos nodos es simétrica; dicho de otro modo, la Link *no tiene direccionalidad, o flecha*. Llamaremos a las Redes donde ocurre esto para todas sus Links, Redes No Dirigidas. Gráficamente podemos ver una red no dirigida de dos nodos en la figura 14.1.

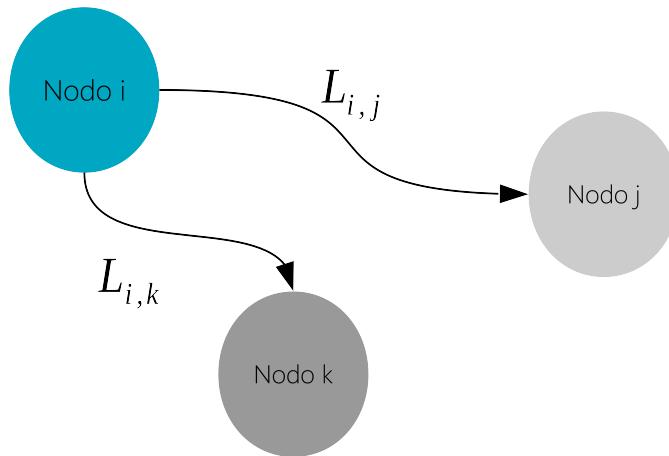


Figura 14.2: Red dirigida formada por tres Nodos y dos Links entre ellos

Las relaciones entre nodos no necesariamente son todas iguales. Unas relaciones pueden ser más intensas que otras. Por ejemplo en la Red de Alumnos anterior, es natural pensar que la relación entre alumnos será más fuerte si han compartido más profesores. Podríamos por tanto asignar a cada link entre dos alumnos un valor numérico correspondiente al número de profesores comunes. Hablaremos entonces de *Links Pesados* y *Redes Pesadas*.

Consideremos ahora una red donde los nodos son personas y diremos que hay una link entre ellas si una ha contagiado la gripe a otra. En esta Red, la Link si que tiene *flecha*, va de un nodo a otro. Hablaremos entonces de *Redes Dirigidas*. En la figura 14.2 vemos un ejemplo donde el Nodo i ha contagiado la gripe a los nodos j y k .

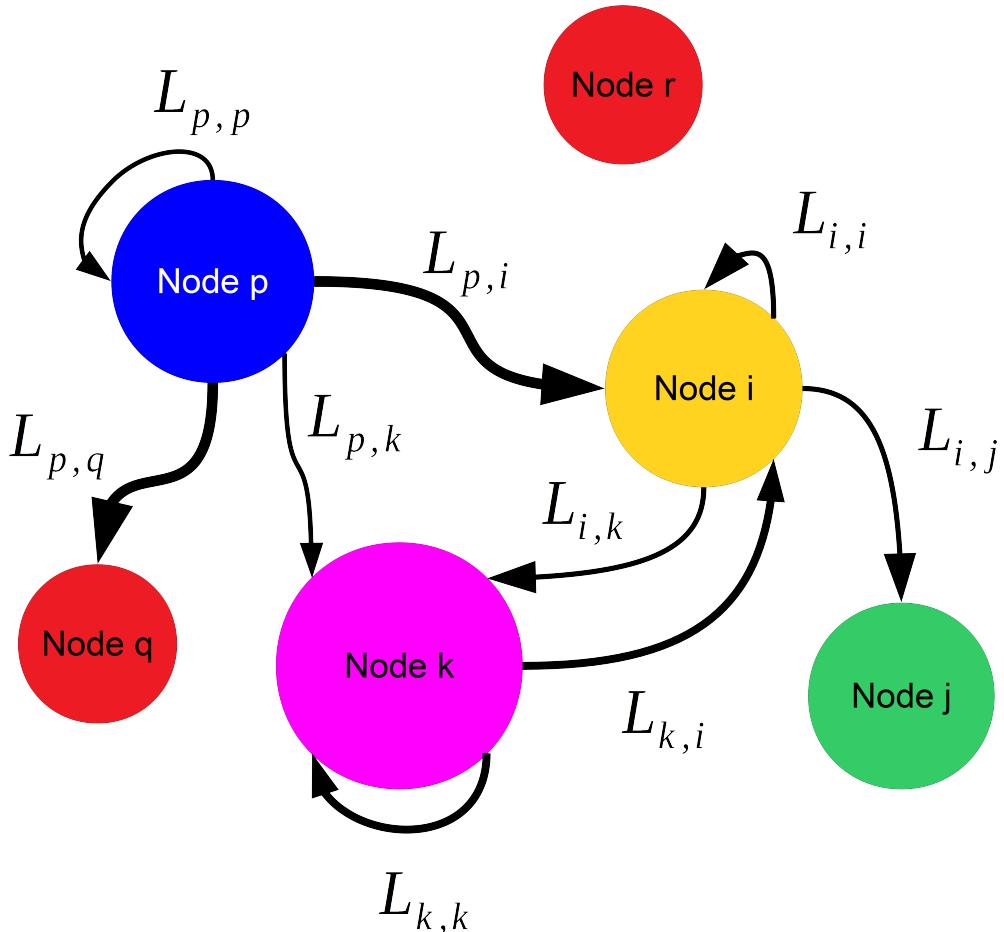


Figura 14.3: Una Red dirigida y pesada. Notar las autolinks y que no todos los nodos tienen que estar relacionados.

En las redes dirigidas, las relaciones son tales que la relación de A con B es diferente de la relación de B con A , pudiendo hablar de relaciones en un sentido o en otro. En este caso podemos tener relaciones entre dos nodos en ambas direcciones. Consideremos la Red cuyos nodos son países, y una Link dirigida y pesada que representa las exportaciones (en Euros) de un país a otro. Dado que un país puede exportar a otro, y el segundo exportar también al primero, nos aparecerán dos links dirigidas yendo de uno a otro.

Además, en general, un nodo puede estar relacionado consigo mismo. Con todos ellos podemos considerar que la red más general tendría una forma similar a la de la Figura 14.3

De entre las propiedades básicas de una Red, nos queda por definir el *Grado* de un Nodo; consideremos una red no dirigida y no pesada. Definimos el grado de cada nodo como el número de links del nodo. Nodos aislados, por ejemplo, tienen grado cero. Si la Red es pesada, llamamos grado a la suma del peso de todos los links del nodo. Si la Red es dirigida, podemos definir el *Grado In* y el *Grado Out*: El Grado In es la suma de los Links que llegan al nodo, y el Grado Out, la suma de los Links que salen del nodo.

Podemos resumir ahora todo lo anterior,

- **Nodo:** Elementos constitutivos de la Red.
- **Link:** Relaciones entre nodos; pueden tener peso (Redes Pesadas) o dirección (Redes dirigidas).

- **Red no dirigida:** Red donde sus links son no dirigidas.
- **Red dirigida:** Red donde sus links tienen definida una dirección.
- **Peso de Link:** Valor numérico de la Relacion entre nodos.
- **Grado de Nodo:** Según el tipo de red, suma de las links de un nodo (No pesadas, no dirigidas), suma de sus pesos (Pesada, no dirigida), suma de pesos de links entrantes (In) o salientes (out).
- **Autolinks o Autoloops:** Links que unen un nodo consigo mismo.

14.3. Componentes, distancias y diámetro

Definimos una Componente como el conjunto de Nodos tales que puede llegarse de uno cualquiera a todos los demás a través de links; tambien lo llamaremos Cluster. Llamamos *Cluster Gigante* a la componente con mayor número de nodos. Redes con alta conectividad tienen un porcentaje elevado de sus nodos en el cluster gigante.

Consideremos una Red no dirigida y no pesada. Dados dos nodos, definimos la *distancia* entre ambos como el mínimo número de links que debemos recorrer para llegar de uno a otro. Llamaremos geodésica, por motivos obvios, a cada una de las trayectorias de longitud mínima entre dos nodos. La definición tiene sentido sólo dentro del mismo cluster; es decir, la distancia no está definida para dos nodos entre los cuales no hay ninguna trayectoria posible (están en diferentes componentes).

Dado una Red, llamaremos *diámetro* a la mayor de las distancias entre nodos; y *distancia media* a la media de las distancias.

14.4. Comunidades

Queremos identificar los grupos de nodos que tienen relaciones más estrechas entre sí que con nodos de otras comunidades, de modo que podemos identificar subgrupos afines dentro del conjunto global. Una comunidad es pues un subconjunto de nodos dentro de una red mayor, tal que entre los nodos de ese suconjunto existen relaciones más estrechas que entre ellos y los de otras comunidades. Los algoritmos para la detección de comunidades es un área de investigación importante, y no entraremos aquí en los detalles.

En la figura 14.4 mostramos una red de investigadores relacionados cuando han firmado un artículo en común; es decir los nodos son investigadores y los links indican coautoría. Es una red no dirigida (consideraremos a todos los autores iguales en un artículo) y pesada (el valor de la link es proporcional al número de artículos en común de los dos nodos). Para hacer la comunidades más patentes, los nodos de una misma comunidad, se han coloreado con el mismo color. Vemos que hay comunidades claramente identificadas, pero otras son más complejas; de hecho con algoritmos diferentes pueden identificarse distintas comunidades.

En el origen de la identificación de comunidades está el caso del *Club de Kárate*, una red real de deportistas en un Club de Kárate de Nueva York, donde tras una discusión interna se dividió en dos. Dadas las relaciones de amistad previas, y utilizando un algoritmo de detección de comunidades, se pudo reproducir exactamente la división real en los dos clubes.

Es importante no confundir *Componente* con *Comunidad*. Componente es un conjunto de nodos que están conectados todos con todos directa o indirectamente; es decir, en una componente podemos ir de un nodo a cualquier otro *viajando* sobre links.

Una Comunidad es un un conjunto de nodos especialmente relacionados respecto a los de su entorno, de modo que estas relaciones más intensas permiten definirlos como un subconjunto diferenciado, con una relación especial.

Una Comunidad es un subconjunto de una Componente. Dentro de una Componente puede haber varias Comunidades. Una Comunidad no puede estar en varias componentes.

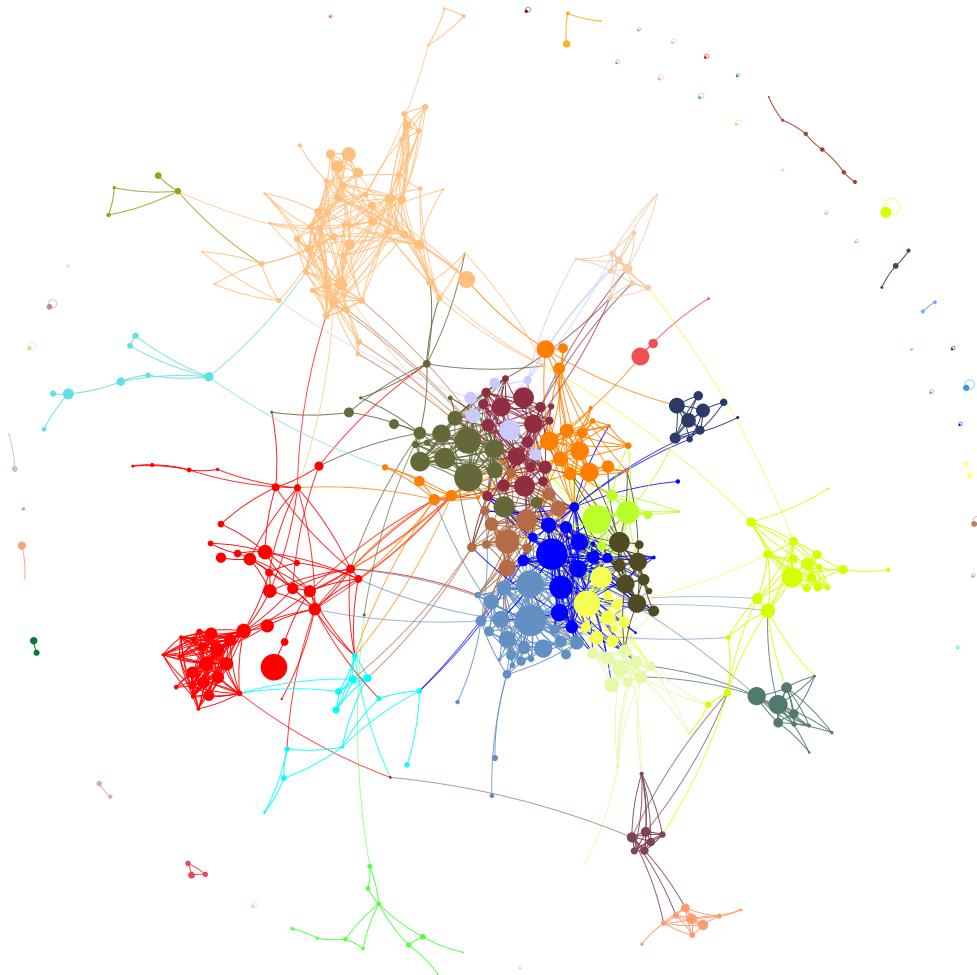


Figura 14.4: Red de coautoría de artículos. El área es proporcional al grado del nodo. El color indica la comunidad a la que pertenece el nodo.

14.5. Medidas de Centralidad: Betweenness y Page Rank

Pensemos en Redes no pesadas y no dirigidas, si bien los conceptos son fácilmente ampliables al resto de tipo de redes. En una Red, el grado nos da información de la conectividad de un nodo. En una red donde los nodos son personas y las links indican amistad (supongamos que siempre mutua), el grado es el número de amigos que tiene una persona, que es una propiedad importante para saber la influencia de una persona en la Red. Sin embargo hay otra propiedades de los nodos que nos dan información relevante del papel que juega una persona en la Red.

14.5.1. Betweenness

La primera propiedad que buscamos se refiere a la importancia de un nodo para que la información viaje por la Red. Llamaremos *Betweeness* a esta propiedad.

Para calcularla, comenzamos en un nodo y viajamos desde él a todos los demás a través de una geodésica; para cada camino contamos cuántas veces pasamos por cada nodo. Si para ir de

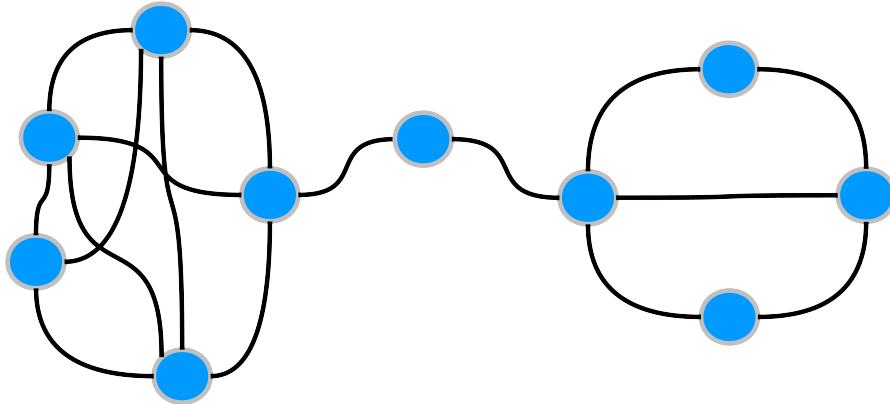


Figura 14.5: En esta Red la betweenness mayor corresponde al nodo central, con grado 2. De hecho este nodo es el de menor grado, pero el de mayor Centralidad de la Red, indicando que a pesar de sus escasas relaciones juega un papel fundamental en el flujo de información y en la cohesión de la Red.

un nodo a otro hay varias geódesicas (trayectorias de igual longitud mínima) las contamos todas ellas. Repetimos el proceso partiendo de todos los nodos, y vamos sumando el número de veces que pasamos por cada uno de los nodos. Al final tenemos un número para cada nodo: el número de veces que hemos pasado por él, cuando hemos ido de cada punto a todos los demás a través de los caminos más cortos. Este número, es lo que llamamos betweenness de cada nodo.

Esta propiedad es cualitativamente distinta del Grado, y nos da una medida de como de importante es un nodo al construir caminos entre nodos, es decir, como es de importante para que la información fluya en la red, o qué nodos son los importantes para cohesionar la red. En la figura 14.5 mostramos una red donde el nodo de mayor betweenness, el situado en el centro del dibujo, tiene un grado muy pequeño. Vemos que en este dibujo uno puede intuir al menos dos comunidades principales: los nodos de la derecha y de la izquierda. El nodo de mayor centralidad aquí, juega el papel de unión entre comunidades diferentes.

14.5.2. Page Rank

Queremos saber ahora cómo de importante es un nodo en función de la importancia de los nodos que están relacionados con él. Consideremos por ejemplo la Red de todas la páginas web del mundo, y conectamos dos nodos cuando una página apunta (tiene un *Link*) a la otra. Esta es una red dirigida, pero el razonamiento es general. Una página aquí tendrá mucha betweenness *grosso modo* si la apuntan muchas otras páginas o hace de puente entre comunidades. Pero no es lo mismo que le apunten muchas páginas de *amiguetes* a que le apunten páginas como la del periódico *New York Times*. Diremos que un nodo tiene mayor *Page Rank* cuanto mayor sea la relevancia de las páginas que le apuntan. Esto nos da un conjunto de ecuaciones que puede ser resulta en casos simples; para redes grandes, se resuelve mediante un proceso iterativo sencillo. Si llamamos $P_A, P_B \dots$ al Page Rank de los nodos $A, B \dots$, en la Figura 14.6, las ecuaciones a resolver serían las siguientes,

$$\begin{aligned}
 P_A &= \frac{P_B}{2} \\
 P_B &= P_C \\
 P_D &= P_A + \frac{P_B}{2} + \frac{P_E}{2} + \frac{P_F}{2} \\
 P_E &= \frac{P_F}{2} \\
 P_F &= \frac{P_D}{2} + \frac{P_E}{2}
 \end{aligned} \tag{14.1}$$

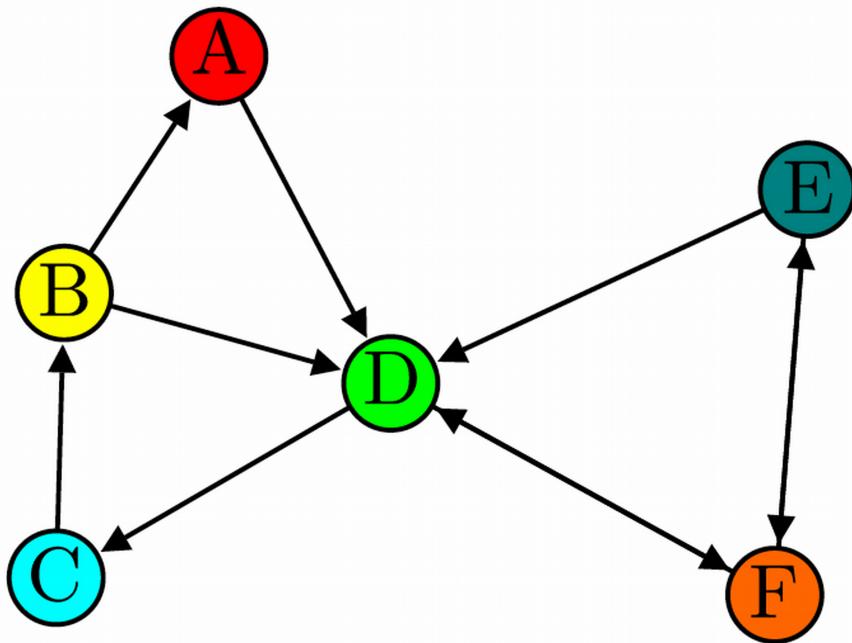


Figura 14.6: Red dirigida usada en el texto para el cálculo del page rank. Una link con dos flechas indica en realidad dos links, uno en cada dirección.

En esta ecuación si el conjunto $P_i, i \in \{A \dots F\}$ es una solución, si multiplicamos dicha solución por una constante, obtenemos una nueva. Podemos elegir la constante de modo que la suma de las P_i valga 1. Dicho de otra forma: cuando calculamos el page rank, no nos interesa el valor absoluto de cada nodo sino sus valores relativos. Obtenemos en este caso $A = 0.08, B = 0.15, C = 0.15, D = 0.31, E = 0.10, F = 0.21$.

Esta cantidad es precisamente la que usa *Google* para ordenar el output que nos muestra tras una búsqueda: páginas con mayor Page Rank son mostradas primero (Evidentemente, salvo las páginas que son mostradas *fueras de orden* por haber pagado a Google para estar mejor situadas).

14.6. Algoritmos de Posicionamiento

Una red es conocida a partir del listado de sus nodos y sus relaciones. En concreto podemos determinar una red indicando en un fichero de texto en cada fila, el nodo inicial, el nodo final y el valor de la link que los une; por ejemplo

```

0 1 10
3 7 1
4 2 13
4 7 5
...
  
```

donde indicamos que el nodo 0 y el nodo 1 están conectados por una Link de valor 10, el 3 y el 7 con una link de valor 1, etc.

Dada esta información podemos proceder a calcular el grado de cada nodo, centralidades, distancias, etc. Sin embargo esta información *cuantitativa* es difícil de visualizar de forma integrada con el mero fichero de texto. Para ayudar a comprender cómo es la red y hacer explícitas muchas de sus propiedades, es conveniente realizar una representación visual

de la misma, de modo que la geometría, tamaños, colores, etc. nos de información integrada de forma global y atractiva.

Los colores pueden ser elegidos por alguna propiedad de los nodos, bien su comunidad, como hemos hecho en la Figura 14.4, u otras diferentes como por ejemplo el Departamento de adscripción de los investigadores o su edad.

El tamaño puede ser elegido en base a propiedades como el grado o la centralidad. En el caso del mapa de Artículos, el grado nos indica el número de artículos firmado por una persona, y si asignamos el área del nodo al grado, en el mapa se harán más visibles los nodos con mayor productividad científica.

La geometría, o posición de cada nodo, es algo más complejo de asignar. Se trata de dar una posición a cada nodo que nos indique cual es su entorno en la Red, situarlo en un lugar donde estén también aquellos nodos con los que más links tiene, o próximos a los de su comunidad, etc. Existen un buen número de algoritmos para hacer esto, de maneras muy diferentes.

Nosotros aquí estudiaremos uno de los algoritmos más habituales y que mejores resultados visuales proporciona: el algoritmo de Fruchterman-Reingold. Para facilitar su visualización implementaremos el algoritmo en dos dimensiones.

14.6.1. Campo de Fuerzas

Consideremos un red no dirigida y pesada. Debemos definir una fuerza entre nodos, de modo que se atraigan los nodos que tienen links entre sí, y que la fuerza sea mayor cuanto mayor sea el valor del link. De esta manera, nodos con links fuertes estarán más cerca que nodos sin links o con links débiles. Una forma simple de obtener esto es crear una fuerza atractiva entre nodos con links. Pero sólo con eso produciríamos un colapso de todos los nodos conectados en un punto.

Para evitar el colapso de los nodos, debemos aplicar también un *core* repulsivo entre todos ellos, tengan o no link entre ellos. Este core repulsivo debe tener un alcance finito para evitar que los nodos se alejen indefinidamente.

Hay muchos campos de fuerzas con estas propiedades; en este algoritmo consideramos la siguiente fuerza entre dos nodos i, j :

$$\vec{F}_{i,j} = \left(\frac{K^2}{r^2} - \left(\frac{K^2}{A} + \frac{W(i,j)}{K} \right) r \right) \vec{r}_{i,j} \quad (14.2)$$

donde $r = |\vec{r}_{i,j}|$, A, K son constantes que definiremos apropiadamente, y $W(i,j)$ es el peso de la Link entre los nodos i, j (que podemos asignar a 1 para redes no pesadas). Para los nodos no conectados por una link, $W(i,j)$ vale 0.

La Fuerza 14.2 deriva de un Potencial de la forma

$$V(r) = -K^2 \ln r + \left(\frac{K^2}{3A} + \frac{W(i,j)}{3K} \right) r^3 \quad (14.3)$$

Es sencillo comprobar que efectivamente el potencial 14.3 nos da la Fuerza 14.2.

$$\vec{F}(\vec{r}) = -\vec{\nabla}V(r) = K^2 \frac{\partial \ln r}{\partial r} \vec{\nabla}r - \left(\frac{K^2}{3A} + \frac{W(i,j)}{3K} \right) \frac{\partial r^3}{\partial r} \vec{\nabla}r \quad (14.4)$$

Las derivadas son simples, y basta recordar que $\vec{\nabla}r = \frac{\vec{r}}{r}$ para recuperar el valor correcto de la Fuerza.

Podemos ver la forma de la fuerza y del potencial para el caso de dos partículas con el siguiente programa para cargar en `gnuplot`, donde hemos asignado un valor a las constantes A y K que luego justificaremos:

```
N=2.0
A=N**3
W_m=1
```

```

K=sqrt(N)*W_m**(1.0/3.0)
PrevF=(K*K/A)+W_m/K
PrevV=PrevF/3
F(x)=(K*K/x**2 -x*PrevF)*x
V(x)=-K*K*log(x)+x**3*PrevV

set xrange[0.3:3]
set yrang[-7:7]
p F(x)
rep V(x)
rep 0

```

obteniendo el resultado de la Figura 14.7

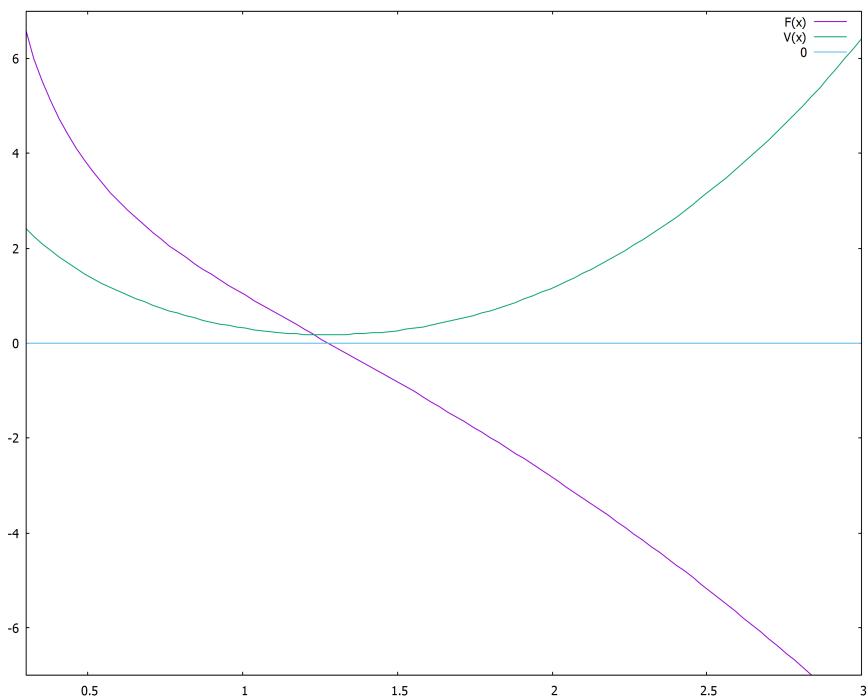


Figura 14.7: Fuerza y potencial en función de r de la interacción de Fruchterman-Reingold para dos nodos conectados. Se dibuja el cero para identificar más fácilmente el mínimo del potencial, lugar donde se anula la fuerza, o punto de equilibrio de los dos nodos para este campo de fuerzas.

14.6.2. Parámetros del Potencial

Nos falta fijar el valor de A y K . Para ello, calculemos el mínimo del potencial, el lugar donde la Fuerza se hace cero. La condición de anulación de la Fuerza nos da el punto de equilibrio r_0 :

$$r_0^3 = \frac{K^2}{\frac{K^2}{A} + \frac{W(i,j)}{K}}$$

Razonemos primero en el caso de que no existan Links, es decir $W(i,j) = 0, \forall i, j$ y que tengamos solamente 2 partículas. En este caso, tendremos $r_0 = A^{1/3}$. Vemos que la presencia de un valor positivo del peso de la Link, hace disminuir r_0 .

Cuando tenemos más partículas, dada la simetría bajo rotaciones del potencial, las partículas se situarán sobre el plano ocupando una región circular. Como la atracción es relativamente de

largo alcance, las alejadas se atraerán entre sí, hasta llegar a un equilibrio entre las que están más próximas de r_0 , que se repelerán, y las que estén más alejadas, que se atraerán. Tendremos pues que la distancia de equilibrio será menor que r_0 , situándose las partículas, en posiciones sobre una red subyacente triangular-hexagonal.

Supongamos que tenemos unos pocos puntos unidos con links iguales. Hagamos la hipótesis muy grosera de que la Fuerza sólo es significativa entre los vecinos más cercanos. Esta no es una aproximación muy buena, pero es suficiente para redes con pocos nodos, y conectividad baja. Fijémonos en dos de esos nodos, y la fuerza entre ellas. Tendremos

$$r_0^3 = \frac{K^2}{\frac{K^2}{A} + \frac{\langle W \rangle}{K}}$$

Si elegimos para A y K los valores, en función del número de nodos, N , $A = N^3$ y $K = \sqrt{N}\langle W \rangle^{1/3}$, entonces tendremos para la distancia de equilibrio

$$r_0^3 = \frac{N\langle W \rangle^{2/3}}{\frac{N\langle W \rangle^{2/3}}{N^3} + \frac{\langle W \rangle}{\sqrt{N}\langle W \rangle^{1/3}}}$$

En el caso en que no haya links, es decir sea un red sin conexiones, tendremos

$$r_0^3 = A = N^3$$

Hemos elegido en este caso que la distancia de equilibrio sea del orden de N .

Si suponemos que $\langle W \rangle \neq 0$, podemos estimar el comportamiento de r_0 en para valores grandes de N , approximando 14.6.2,

$$r_0^3 \approx \frac{N}{\frac{1}{N^2} + \frac{1}{\sqrt{N}}} \approx N^{3/2}$$

significativamente más próximos, como corresponde a que los nodos con interacción queremos que se sitúen más próximos en el grafo.

14.6.3. Solución de Equilibrio

Dada una posición y velocidad inicial para cada nodo, el campo de Fuerzas 14.2 haría que el sistema evolucionara temporalmente según las leyes de la Mecánica. Sin embargo lo que queremos es llegar a una *foto* final estática del sistema, lo más próximo a una situación donde las fuerzas sean nulas o muy pequeñas, de modo que el sistema no evolucione y tenga las propiedades requeridas de una geometría que represente de forma fidedigna la Red subyacente; llámenos posición de equilibrio a esta situación.

Esperamos que en la situación de equilibrio los nodos se sitúen en posiciones tales que estén rodeados de aquellos nodos con los que tengan mayor relación. Incluso conseguiremos de este modo que los nodos de la misma comunidad se agrupen en posiciones próximas, si bien en general habrá nodos fuera de la región.

La cuestión es pues llegar al equilibrio. Nuestras redes contendrán habitualmente cientos, miles, incluso decenas de miles de nodos. Todos ellos situados en el Potencial anterior, evolucionarán de acuerdo a las fuerzas que sufren. Podríamos usar un algoritmo como Euler o Leap Frog u otros para hacer evolucionar el sistema, partiendo de una configuración inicial que podemos generar de diferentes modos.

Pero una evolución de este tipo, partiendo de una cierta configuración inicial, sería un sistema en evolución constante, se conservaría la energía y los nodos no llegarían nunca a detenerse. Por otro lado cuando los nodos son numerosos y las links también, el sistema puede evolucionar muy lentamente, incluso quedarse atrapado en mínimos locales por la imposibilidad de que las partículas salgan de zonas donde están atrapadas para llegar a zonas más afines.

Para evitar el problema de quedarnos atrapados en mínimos locales, ya sabemos una forma eficiente de evitar esto cuando tenemos muchos grados de libertad: usar un algoritmo tipo Simulated Annealing. Aquí usaremos una modificación del algoritmo, dada la sencillez de la interacción.

Veamos ahora como evitar el problema de la evolución temporal del sistema. Si recordamos el algoritmo de Euler (Capítulo 2.1) si queremos hacer evolucionar el sistema, para cada partícula i debemos calcular la fuerza total que actúa sobre ella, y tras conocerla para todas las partículas, mover cada una en un paso elemental temporal, correspondiente a un tiempo infinitesimal h . Las partículas irán así evolucionando en el tiempo, pero dependiendo de la energía inicial, y dado que esta se conserva, no llegaremos a la situación de equilibrio, sino a una situación de las partículas en torno a esa posición, como oscilaciones en torno a la misma, de modo que tengan la misma energía inicial. Y no es esto lo que queremos, pues queremos llegar al equilibrio estático, es decir a una situación de energía minima, con energía cinética nula.

Una forma de conseguir esto es añadir un término disipativo al potencial: por ejemplo una fuerza de rozamiento proporcional a (menos) la velocidad. De este modo el sistema con el tiempo va perdiendo energía mecánica, y llega al estado deseado: el sistema iría evolucionando poco a poco (h es pequeño), atraido por el mínimo en donde dejará de evolucionar por la pérdida de energía, pero no necesariamente llegando al mínimo absoluto, si hay mínimos locales de los que no puede salir. Lo más efectivo en este caso es considerar que el término disipativo es tal que frena a la partícula al final de cada movimiento, es decir, en la práctica consideramos que la velocidad de partícula es cero al iniciar y al finalizar el movimiento.

En resumen nuestro algoritmo será usar una mezcla de Simulated Annealing una fuerza disipativa, del siguiente modo.

- **Valor de h :** Usaremos un algoritmo de Euler con $h = 1$. De este modo para una fuerza dada, el desplazamiento que sufre cada partícula será muy grande, lo que producirá grandes movimientos que nos permitirá salir de los mínimos locales. Puede ser interpretado como una simulación a alta Temperatura.
- **Término disipativo:** Dado el desplazamiento anterior, para cada iteración fijamos un valor máximo, Δ , tal que si el desplazamiento es mayor que Δ , hacemos que el módulo de dicho desplazamiento valga Δ
- **Llegada al equilibrio:** Inicialmente, en las primera iteraciones, con Δ grande, el sistema explora casi todo el espacio fase y esperamos que se acerque al mínimo global. Vamos haciendo que el valor de Δ vaya disminuyendo al avanzar en las iteraciones, de modo que el sistema se vaya acomodando en su mínimo local. Llegamos finalmente a un valor casi nulo en las últimas iteraciones, de modo que el sistema ya no se mueva apenas, buscando el equilibrio. Esto es equivalente a un término disipativo creciente según va pasando el tiempo de simulación. Una buena elección para el protocolo de cambio de Δ , es hacerla cambiar con las iteraciones del siguiente modo:

$$\Delta = M_\Delta ((i/N_{iter})^{3/2}, i = N_{iter}, N_{iter} - 1, \dots 0)$$

con $M_\Delta = N^2$, N el número de nodos o partículas y donde i es el índice de la iteración, que son en total N_{iter} . De este modo inicialmente permitimos movimientos de módulo hasta N^2 , que mueven suficientemente las partículas

Al inicio del proceso, las partículas estarán en general muy alejadas del equilibrio, y por tanto sufrirán fuerzas grandes, de modo que al usar $h = 1$, se moverán muchísimo, yendo a parar a una posición también muy alejada del equilibrio (y recordemos que llegan con velocidad cero), de modo que en el siguiente paso sufrirán un gran desplazamiento en sentido contrario. Veremos pues un movimiento altamente oscilante, tipo diente de sierra, para cada coordenada. La anchura de este diente de sierra es del orden de Δ . Pero cuando Δ vaya disminuyendo, las oscilaciones irán disminuyendo en amplitud, lo que provocará una evolución de los dientes de sierra con una envolvente que va a cero, y llevará a cada partícula (nodo) a una situación próxima al equilibrio, tras explorar todo (o casi todo) el espacio.

14.7. Ejercicios

14.7.1. Algoritmo de Posicionamiento

Escribir un programa debidamente estructurado, que implemente el algoritmo de posicionamiento para Redes explicado en el texto, para el caso de Redes pesadas, que sirva para redes dirigidas o no dirigidas, excluyendo autolinks. El código debe leer de un fichero el número de partículas a simular y el número de iteraciones. También se debe indicar si el fichero de la red (conteniendo los nodos y links) se lee (en cuyo caso se debe dar el nombre del archivo) o se genera automáticamente. Lo mismo para un fichero con las posiciones iniciales de los nodos. El programa debe producir un fichero con las posiciones finales de las partículas, y también con las links, de modo que desde gnuplot se pueda visualizar el resultado de nodos y links.

14.8. Problemas

14.8.1. Problema 1

Construir redes simples donde se identifiquen fácilmente las comunidades, y comprobar que el algoritmo de posicionamiento agrupa nodos en comunidades iguales, en posiciones próximas.

14.9. Código en C

Utilizamos en este código *estructuras*, un instrumento importante en C para manejar datos relativamente complejos. En el Tomo II puede estudiarse lo necesario.

Describiremos a continuación algunas de las características del código en C propuesto, que no deja de ser una mera ayuda en la que el alumno pueda apoyarse para la escritura de un programa propio; dicho de otro modo: el alumno puede, y debería, modificar el código, cambiar algunos de los detalles de Input/Output, de la estructura de las funciones, etc. Por supuesto en la explicación subsiguiente nos referimos al código concreto dado.

14.9.1. Fichero de Input

Al ejecutar el programa, en la línea de comandos podemos pasar el nombre de un archivo que contiene la información básica para empezar. Si no damos ningún nombre, por defecto busca el archivo `parameters.dat` en el directorio de ejecución. El fichero es de la siguiente forma

```
60 N_par
1000 niter
1 dos_bolas.dat
0 posicion.dat
```

Cada línea tiene un valor numérico y un texto.

- **60 N_par :** La primera línea indica el número de Nodos de la Red. Este número es relevante sólo si indicamos (luego) al programa que genere la Red. El texto siguiente sólo sirve para recordar qué es el número anterior.
- **1000 niter:** Número de iteraciones del algoritmo de posicionamiento.
- **1 dos_bolas.dat:** Si el primer campo es 1, el texto siguiente debe ser el nombre de un archivo que será leido para construir la Red. El formato del archivo deben ser N líneas y en cada una debe aparecer $Nodo_i Nodo_j W(i,j)$ en formato

```
fscanf(fin,"%d %d %lf\n",&A,&B,&graph.peso[iLink])
```

Una vez leídos todos los nodos, el programa supone que existen nodos desde el 0 hasta el máximo nodo, aunque no aparezcan los nodos intermedios en el fichero, que simplemente no tendrán links. Así por ejemplo, el archivo de entrada

```
3 4 8.0
4 7 4.0
2 3 5.0
```

se generará una red con nodos $\{0, 1, 2, 3, 4, 5, 6, 7\}$, donde los nodos $\{0, 1, 5, 6\}$ no estarán conectados. El número de nodos, que inicialmente estaba a 60 se sobreescibirá y pasa a valer 8. Si el número es 0, se genera un grafo dentro del programa.

- **0 posicion.dat:** Si el primer campo es 1, el programa busca el archivo indicado a continuación y lee las coordenadas iniciales de los nodos de la Red, a partir de las cuales comenzar la simulación. El archivo contiene en cada línea las coordenadas x, y de cada nodo, con el formato

```
fscanf(fin,"%lf %lf\n",&graph.r[site][0],&graph.r[site][1])
```

Si el campo numérico es 0, el programa calcula unas coordenadas random para empezar.

14.9.2. Ficheros de Output

Durante la ejecución se producen dos ficheros; el primero de ellos es `posicion_r.dat` que contiene las posiciones finales de todos los nodos en el plano. Supongamos que el fichero de input tiene la forma

```
60 N_par
1000 niter
1 dos_bolas.dat
0 posicion.dat
```

con el fichero `dos_bolas.dat`

```
0 2 1
0 3 2
0 4 10
0 5 10
0 6 10
0 7 10
0 8 10
0 9 10
0 10 10
10 11 10
10 12 10
10 13 10
10 14 10
10 15 1
10 16 10
10 17 10
10 18 10
10 19 10
10 20 1
30 31 10
30 32 10
31 32 10
```

Tras la ejecución el fichero `posicion_r.dat`, que contiene la posición final de todas la partículas, tendrá una forma similar (pues la secuencia de números aleatorios no será la misma en todos los ordenadores, y por tanto la simulación será diferente),

```
-5.954828 -4.946719 6.164414
-14.299615 -19.728722 1.224745
-10.433688 9.971095 1.224745
-20.444686 -9.389883 1.414214
-7.442840 -12.742571 2.449490
-6.273846 1.793580 2.449490
-11.803487 0.508414 2.449490
...
```

Este fichero nos da la posición de los nodos x, y en la primera y segunda columna, y en la tercera un número calculado como $\sqrt{1 + grado(Nodo_i)}$. Este número sera usado para el tamaño del simbolo en `gnuplot`: de este modo el área del simbolo será proporcional al grado del punto (en realidad sumamos 1 para que los nodos de grado 0 sean visibles). Queremos representarlos con `gnuplot` de modo que además de visualizar los nodos, veamos también las Links que los unen.

Para ello escribimos desde el propio código un fichero de comandos para `gnuplot`, que dibuje los nodos y una flecha entre nodos cuando hay links entre ellas. Dibujaremos las flechas en sentido del primer nodo al segundo según aparezcan en el fichero de input de la Red. Las flechas entre dos puntos se pueden dibujar con el comando `arrow`, que tiene la forma

```
set arrow from x_inicial,y_inicial to x_final,y_final
```

las flechas se visualizan al ejecutar el comando `plot`. El archivo que contiene todos los comandos y datos necesarios para visualizar la Red es `posicion.plt` y es de la forma

```
plot "posicion_r.dat" u 1:2:3 with points pt 6 ps variable
unset arrow
set arrow from -0000.46066358,-0000.03041999 to -0002.26457595,-0001.75139744
set arrow from -0000.46066358,-0000.03041999 to 00002.23090658,-0000.11284822
set arrow from -0000.46066358,-0000.03041999 to -0002.18803529,00002.17839440
set arrow from -0002.18803529,00002.17839440 to -0001.51212945,00004.62791121
set arrow from -0001.51212945,00004.62791121 to 00000.31712884,00005.52683303
set arrow from 00000.31712884,00005.52683303 to -0001.51212945,00004.62791121
replot
```

Remarcamos que este fichero debe ser construido desde el código en C. Dadas los ficheros de Input anteriores, la gráfica obtenida puede verse en la Figura 14.8.

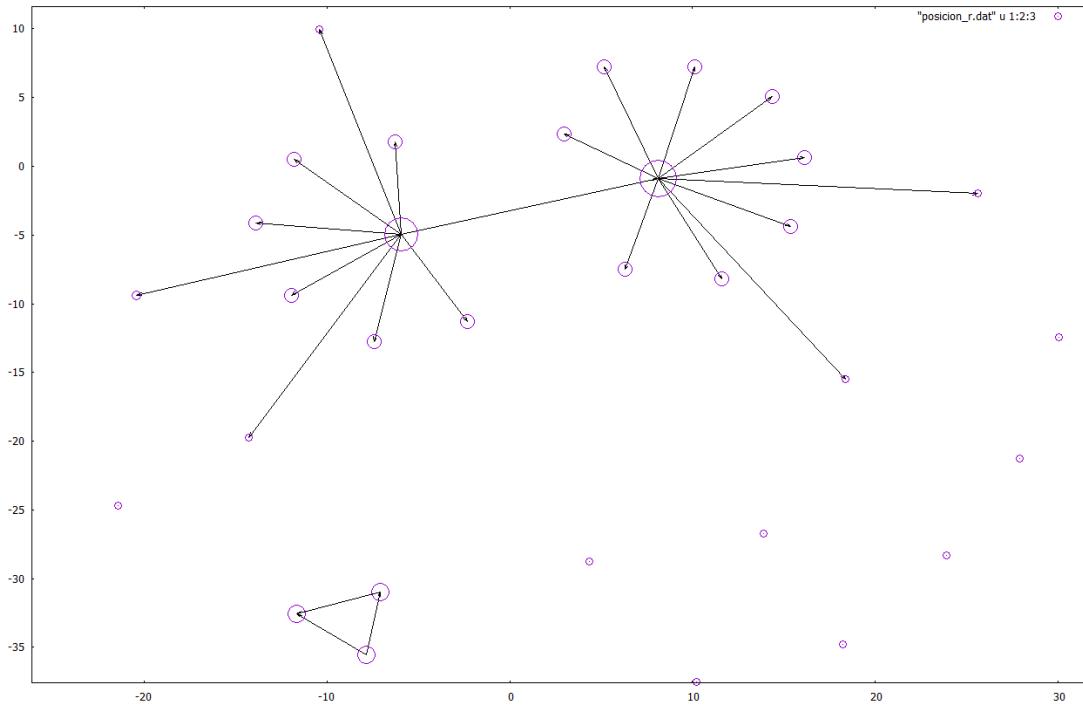


Figura 14.8: Geometría de la Red de Test dada en 14.9.2, tras la ejecución del algoritmo de posicionamiento, y la carga en gnuplot del archivo `posicion.plt`

14.9.3. Algoritmo de posicionamiento para Redes Complejas

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
//#define DEBUG

#define Dim_Max 2
#define N_par_Max 6000
#define NLinks_Max (N_par_Max*200)
```

```

#define fran ((double)rand() / ((double)RAND_MAX+1.0))

void lee_param(int ac,char **av);
void lee_coords(void);
void genera_coords(void);
void lee_graph(void);
void genera_graph(void);
void Calcula_Potential_FR(void);
void escribe_coords(void);
void compute_degree(void);

void Evoluciona(void);
void Acumula_Links(void);
double usada[N_par_Max][N_par_Max];

double F[N_par_Max][Dim_Max];
double epsilon;

struct Prm{
    int Dim;
    int N_par;
    int niter;
    int flag_graph;
    int flag_coords;
    char graph_file[256];
    char coords_file[256];
}Param;

struct grph{
    int N_par;
    int N_links;
    double wmean;
    double grado[N_par_Max];
    double size[N_par_Max];
    double r[N_par_Max][Dim_Max];
    int origen[NLinks_Max];
    int destino[NLinks_Max];
    double peso[NLinks_Max];
}graph;

struct P_FR{
    double area;
    double maxdelta;
    double coolexp;
    double temperature;
    double repulserad;
    double frk;
    double frk2;
}Potential_FR;

int Lado_Max;

int main(int argc,char **argv)
{
    int i,segundos;
    int mesfr;

    FILE *fevol;
    epsilon=0.000001; //para evitar la singularidad en el origen

    lee_param(argc,argv);

    if(Param.flag_graph==0)
        genera_graph();
    else
        lee_graph();

    compute_degree();
}

```

```

Lado_Max=sqrt((double)graph.N_par);

if(Param.flag_coords==0)
    genera_coords();
else
    lee_coords();

Calcula_Potential_FR();

mesfr=10;

fevol=fopen("Evol.dat","wt");

segundos=time(0);
for (i=Param.niter;i>0;i--)
{
    Potential_FR.temperature=Potential_FR.maxdelta*
        pow(i/(double)Param.niter,Potential_FR.coolexp);
    Evoluciona();

    fprintf(fevol, " %lf %lf %lf %lf\n",
    graph.r[0][0],graph.r[0][1],graph.r[1][0],graph.r[1][1]);

    if(i%mesfr==0)
    {
        printf("(t=%d)",(int)(time(0)-segundos));
        printf(" %d %lf %lf %lf %lf \n",i,graph.r[0][0],graph.r[0][1],graph.r[1][0],graph.r[1][1]);
        //getchar();
    }
}
printf("(Total Time= %d\n",(int)(time(0)-segundos));
fclose(fevol);

escribe_coords();

return 1;
}

// Esta función es el Core del programa

void Evoluciona(void)
{
    double distance,d2;
    int i,j,il,k;
    double factor,del[Dim_Max],ded;

    for(i=0;i<graph.N_par;i++)
        for(k=0;k<Param.Dim;k++)
            F[i][k]=0;

    for(i=0;i<graph.N_par;i++)
    {
        for(j=i+1;j<graph.N_par;j++)
        {
            for(k=0,d2=0;k<Param.Dim;k++) //Calculo de r cuadrado
            {
                del[k]=graph.r[i][k]-graph.r[j][k];
                d2+=del[k]*del[k];
            }
            d2+=epsilon; //Para evitar la singularidad en el origen
            distance=sqrt(d2); //r
        }
    }

    //Modulo del termino general:
    factor=Potential_FR.frik2*(1.0/d2-distance/Potential_FR.repulserad);

    for(k=0;k<Param.Dim;k++) //Vector fuerza general
    {
        F[i][k]+=del[k]*factor;
    }
}

```

```

        F[j][k]-=del[k]*factor;
    }
}
}
//recorre subconjunto de links:
for(il=0;il<graph.N_links;il++) //Termino de fuerza con links
{
    i=graph.origen[il];
    j=graph.destino[il];

    for(k=0,d2=0;k<Param.Dim;k++)
    {
        del[k]=graph.r[i][k]-graph.r[j][k];
        d2+=del[k]*del[k];
    }

    distance=sqrt(epsilon+d2);

    factor=distance/Potential_FR.frk*graph.peso[il]; //Modulo F Link

    for(k=0;k<Param.Dim;k++) //vector
    {
        F[i][k]-=del[k]*factor;
        F[j][k]+=del[k]*factor;
    }

} //fin bucle atraccion

for(i=0;i<graph.N_par;i++) //recorre subconjunto de nodos
{
    for(k=0,ded=0;k<Param.Dim;k++) //Modulo de la fuerza
    ded+=F[i][k]*F[i][k];

    ded=sqrt(ded+epsilon);

    if(ded>Potential_FR.temperature) //Una especie de Simmulated Annealing
    {
        ded=Potential_FR.temperature/ded;
        for(k=0;k<Param.Dim;k++)
            F[i][k]*=ded;
    }

    for(k=0;k<Param.Dim;k++)
        graph.r[i][k]+=F[i][k]; //Consumado el cambio
    } //fin bucle acumula y cambia
} //fin de funcion

void lee_param(int ac,char **av)
{
    char dummy[256],name[256];
    FILE *Dummy_File,*Input_File;
    /*
        Leemos el fichero que se indica en linea de comandos (parameters.dat default);
        En ese fichero leemos parametros para la simulacion
    */

    Param.Dim=2;
    switch(ac)
    {
        case 1:
            sprintf(name,"parameters.dat");
            printf("Opening parameters.dat for input\n");
            break;
        case 2:
            sscanf(av[1],"%s",name);
            break;

        default:
    }
}

```

```

    printf("uso: fr_new2 [parameters_file:Def:parameters.dat] \n");
    exit(1);
}

if((Input_File=fopen(name,"rt"))==NULL)
{
    printf("Error opening file %s for reading\n",name);
    exit(1);
}

if(fscanf(Input_File,"%d%s\n",&Param.N_par,dummy)!=2)
{
    printf("Error reading first data N_par=%d\n",Param.N_par);
    exit(2);
}

if(fscanf(Input_File,"%d%s\n",&Param.niter,dummy)!=2)
{
    printf("Error reading data niter=%d\n",Param.niter);
    exit(3);
}

if(fscanf(Input_File,"%d%s\n",&Param.flag_graph,Param.graph_file)!=2)
{
    printf("Error reading parameters for graph_file\n");
    exit(4);
}
else
{
    if(Param.flag_graph==1)
{
    if((Dummy_File=fopen(Param.graph_file,"rt"))==NULL)
    {
        printf("Error opening Links File=%s\n",Param.graph_file);
        exit(5);
    }
    else
        fclose(Dummy_File);
}
    }

if(fscanf(Input_File,"%d%s\n",&Param.flag_coords,Param.coords_file)!=2)
{
    printf("Error reading parameters for coords_file\n");
    exit(6);
}
else
{
    if(Param.flag_coords==1)
{
    if((Dummy_File=fopen(Param.coords_file,"rt"))==NULL)
    {
        printf("Error opening Coords File=%s\n",Param.coords_file);
        exit(7);
    }
    else
        fclose(Dummy_File);
}
    }
fclose(Input_File);
#endif DEBUG
printf("Leidos Parametros en lee_param:\n");
printf("N_par=%d\n",Param.N_par);
printf("flag_graph=%d\n",Param.flag_graph);
printf("flag_coords=%d\n",Param.flag_coords);
#endif
}

```

```

void lee_coords(void)
{
    FILE *fin;
    int site;

    if((fin=fopen(Param.coords_file,"rt"))==NULL)
    {
        printf("Error en apertura de fichero de coordenadas: %s\n",Param.coords_file);
        exit(30);
    }
    site=0;
    while(fscanf(fin,"%lf %lf\n",&graph.r[site][0],&graph.r[site][1])!=EOF)site++;

    if(site!=Param.N_par)
    {
        printf("Number of particles modified reading coords_file:\n");
        printf("N in Input_file=%d,N in coords_file=%d\n",Param.N_par,site);
        printf("All OK\n");
    }
    fclose(fin);
    Param.N_par=site;

#ifdef DEBUG
    printf("Coord r[0][0]=%lf, r[%d][1]=%lf\n",graph.r[0][0],graph.N_par-1,graph.r[graph.N_par-1][0]);
#endif
}

void genera_coords(void)
{
    int i,j;
    for(i=0;i<graph.N_par;i++)
        for(j=0;j<Param.Dim;j++)
            graph.r[i][j]=Lado_Max*(fran-0.5);
    for(j=0;j<Param.Dim;j++)
        graph.r[0][j]=0;
}

void lee_graph(void)
{
    //Formato del archivo: n_ini n_fin link_ini_fin
    //n_ini debe ser un indice de 0 a (n_par-1)
    int ilink,np,A,B;
    FILE *fin;
    graph.wmean=0;
    ilink=np=0;
    fin=fopen(Param.graph_file,"rt");
    while( fscanf(fin,"%d %d %lf\n",&A,&B,&graph.peso[ilink])!= EOF)
    {
        if(A!=B)//NO queremos loops
        {
            graph.origen[ilink]=A;
            graph.destino[ilink]=B;
            graph.wmean+=graph.peso[ilink];
#ifdef DEBUG
            printf("site_i=%d,site_j=%d,link[%d]=%lf\n",A,B,ilink,graph.peso[ilink]);
#endif
            ilink++;
            if(A>np)np=A;
            if(B>np)np=B;
        }
    }
    if((np+1)!=Param.N_par)
        printf("Sobreescrito el numero de particulas: input=%d, graph_file=%d\n",
        Param.N_par,np+1);
}

```

```

graph.N_par=np+1; //se sobrescribe lo leido en el fichero de parametros.
graph.N_links=ilink;
fclose(fin);

if(graph.N_links>0)
    graph.wmean/=ilink;

if(graph.N_par<=0)
{
    printf("Error: Grafo leido tiene 0 nodos\n");
    exit(10);
}

#ifndef DEBUG
printf("Grafo:\n");
printf("N_par=%d\n",graph.N_par);
printf("N_links=%d\n",graph.N_links);
printf("wmean=%lf\n",graph.wmean);
#endif
}

void genera_graph(void)
{
    int N_Links_Medio, ilink,i,j;
    int A,B;
    double Peso_Maximo_Link;

    graph.N_par=Param.N_par;
    ilink=0;
    for(i=0;i<graph.N_par;i++)
        for(j=0;j<graph.N_par;j++)
            usada[i][j]=0;

    N_Links_Medio=(sqrt(sqrt(graph.N_par)));
    if(N_Links_Medio<2)N_Links_Medio=2;
    if(Param.N_par>=1000)N_Links_Medio=1;
    Peso_Maximo_Link=20.0;
    for(i=0;i<graph.N_par*N_Links_Medio;i++) //random
    {
        do
    {
        A=graph.N_par*fran;
        while((B=graph.N_par*fran)==A);
    }while(usada[A][B]==1);

        graph.origen[ilink]=A;
        graph.destino[ilink]=B;
        usada[A][B]=1;
        graph.peso[ilink]=Peso_Maximo_Link*fran;
        graph.wmean+=graph.peso[ilink];
        ilink++;
    }

    graph.N_links=ilink;

    if(graph.N_links>0)
        graph.wmean/=ilink;

    if(graph.N_par<=0)
    {
        printf("Error: Grafo construido tiene 0 nodos\n");
        exit(10);
    }
}

void Calcula_Potential_FR(void)
{
    Potential_FR.area=(double)graph.N_par*graph.N_par;
    Potential_FR.maxdelta=sqrt(Potential_FR.area);
}

```

```

Potential_FR.coolexp=1.5;
Potential_FR.repulserad=graph.N_par*Potential_FR.area;
Potential_FR.frk=sqrt((double)graph.N_par)*pow(graph.wmean,1/3.0);
Potential_FR.frk2=Potential_FR.frk*Potential_FR.frk;
#ifndef DEBUG
    printf("En Calcula_Potential_FR:\narea=%lf, maxdelta=%f,repulserad=%lf, frk=%lf\n",
    Potential_FR.area,Potential_FR.maxdelta,Potential_FR.repulserad,Potential_FR.frk);
#endif
}
void escribe_coords(void)
{
    FILE *fout;
    int i,j,il,k;

    fout=fopen("posicion.dat","wt");
    for(i=0;i<graph.N_par;i++)
    {
        for(k=0;k<(Param.Dim-1);k++)
            fprintf(fout,"%lf ",graph.r[i][k]);
        fprintf(fout,"%lf\n",graph.r[i][Param.Dim-1]);//para evitar el blanco al final de linea
    }
    fclose(fout);

    fout=fopen("posicion_r.dat","wt");
    for(i=0;i<graph.N_par;i++)
    {
        for(k=0;k<Param.Dim;k++)
            fprintf(fout,"%lf ",graph.r[i][k]);

        fprintf(fout," %lf\n",sqrt(1+graph.grado[i]));
    }
    fclose(fout);
    fout=fopen("posicion.plt","wt");
    if(Param.Dim==2)
        fprintf(fout,"plot \"posicion_r.dat\" u 1:2:%d with points pt 6 ps variable\n",Param.Dim+1);
    else
        fprintf(fout,"splot \"posicion_r.dat\" u 1:2:3:%d with points pt 6 ps variable\n",Param.Dim+1);

    fprintf(fout,"unset arrow\n");
    for(il=0;il<graph.N_links;il++)
    {
        i=graph.origen[il];
        j=graph.destino[il];
        fprintf(fout,"set arrow from ");
        for(k=0;k<Param.Dim-1;k++)
            fprintf(fout,"%014.8f,\t",graph.r[i][k]);
        fprintf(fout,"%014.8f ",graph.r[i][Param.Dim-1]);

        fprintf(fout,"to ");
        for(k=0;k<Param.Dim-1;k++)
            fprintf(fout,"%014.8f,\t",graph.r[j][k]);
        fprintf(fout,"%014.8f\n",graph.r[j][Param.Dim-1]);
    }
    fprintf(fout,"replot\n");
    fclose(fout);
}
void compute_degree(void)
{
    int il;
    for(il=0;il<graph.N_par;il++)
        graph.grado[il]=0;

    for(il=0;il<graph.N_links;il++)
    {
        graph.grado[graph.origen[il]]+=graph.peso[il]/2.0; //Para grado out
        graph.grado[graph.destino[il]]+=graph.peso[il]/2.0; //Para grado in
    }
}

```


Capítulo 15

Neural Networks

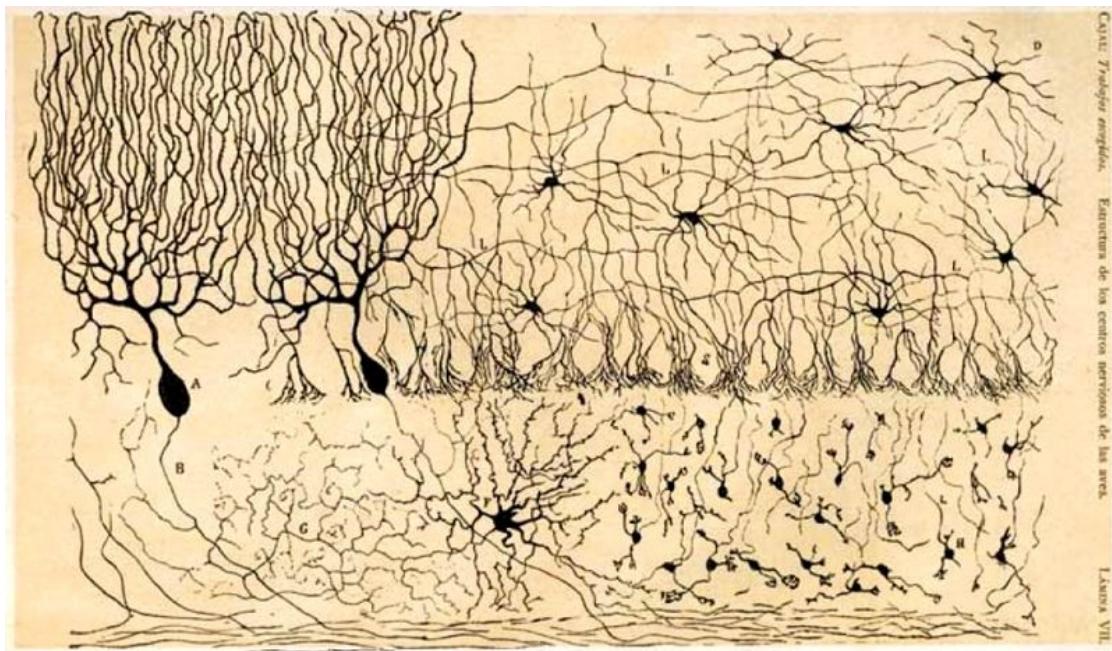


Figura 15.1: Uno de los primeros dibujos de la estructura de las Neuronas y sus conexiones, realizado por Santiago Ramón y Cajal

El descubrimiento de que las funciones cognitivas en el hombre (y en los animales) reside en el cerebro es relativamente reciente. Lo es aún más el conocimiento básico de su funcionamiento. En este aspecto hay que reconocer a Santiago Ramón y Cajal como descubridor del papel de las neuronas y sus conexiones en las funciones cerebrales, lo que inició lo que hoy conocemos como neurociencia. Despojada la inteligencia de connotaciones por encima de la Naturaleza, se ha ido tratando de comprender sus mecanismos, reproducirlos, imitarlos o mejorarlos.

Son ya muchos los aspectos en los que la Ciencia ha desarrollado tecnologías y herramientas que han logrado imitar e incluso superar al cerebro humano. La Memoria es uno de ellos. Mientras la memoria humana es limitada, selectiva y evanescente, hoy en día en un simple pendrive pueden almacenarse una enorme cantidad de datos (imágenes, caras, textos, números, canciones...) de forma casi ilimitada, con detalle y de forma permanente. Las operaciones matemáticas son otro aspecto particular del poder de las máquinas frente al cerebro. Incluso tareas como juegos (desde el trivial tres en raya hasta el rodeado de cierto aire místico como el Ajedrez) son acometidas ya por ordenadores con calidad al menos similar a los Humanos.

Sin embargo quedan muchos aspectos en los que se está muy lejos de las funcionalidades del

cerebro, concretamente en los aspectos cognitivos superiores: pensamiento abstracto, elaborar nuevas ideas, toma de decisiones complejas, generación de nuevos conceptos, avances científicos, arte, sentimientos, percepción de la realidad, autoconsciencia...

El cerebro humano es la consecuencia (última de momento....) de un proceso evolutivo de miles de millones de años, que comenzó con la primera célula (registros fósiles de hace 3500 millones de años), continuó con una mejora continua por la selección natural, y que recoge toda la complejidad de dicho periodo, lo que ha generado un órgano extremadamente complejo, versátil, potente, capaz de dominar el Planeta, comprender el Universo y sus Leyes, y en última instancia ser capaz de comprenderse a si mismo.

De todas las teorías y desarrollos tendentes a comprender, imitar o mejorar las funciones cerebrales, la Teoría de la Redes Neuronales es la que actualmente goza de mayor popularidad, y la expondremos aquí brevemente.

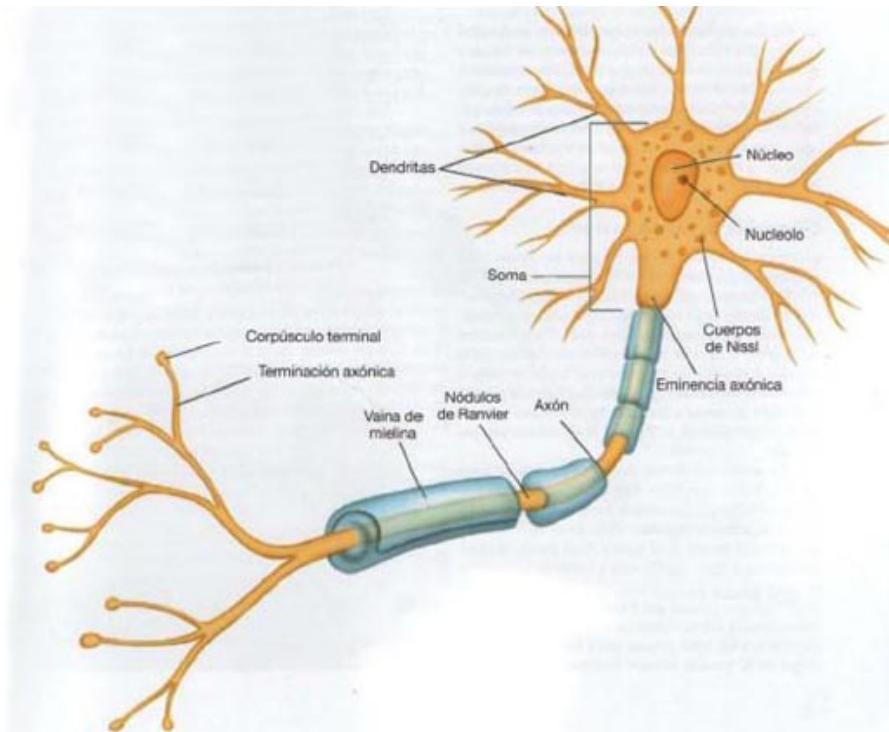


Figura 15.2: Dibujo básico de una Neurona cerebral.

15.1. Cerebro y Neuronas

En una versión supersimplificada podemos pensar en el cerebro como un conjunto de varios miles de millones de neuronas interconectadas entre sí. Cada neurona (Figura 15.2) consta principalmente de tres partes:

- Dendritas: Ramificaciones por las que la Neurona recibe Inputs eléctricos provenientes del Axón de otras Neuronas
- Soma: Cuerpo de la Célula, donde se sitúan el núcleo, los otros órganos celulares y dónde se realizan funciones lógicas con los Inputs y se genera el Output al Axón.
- Axón: Ramificación de salida del Soma, que lleva la señal eléctrica a otras Neuronas conectadas.

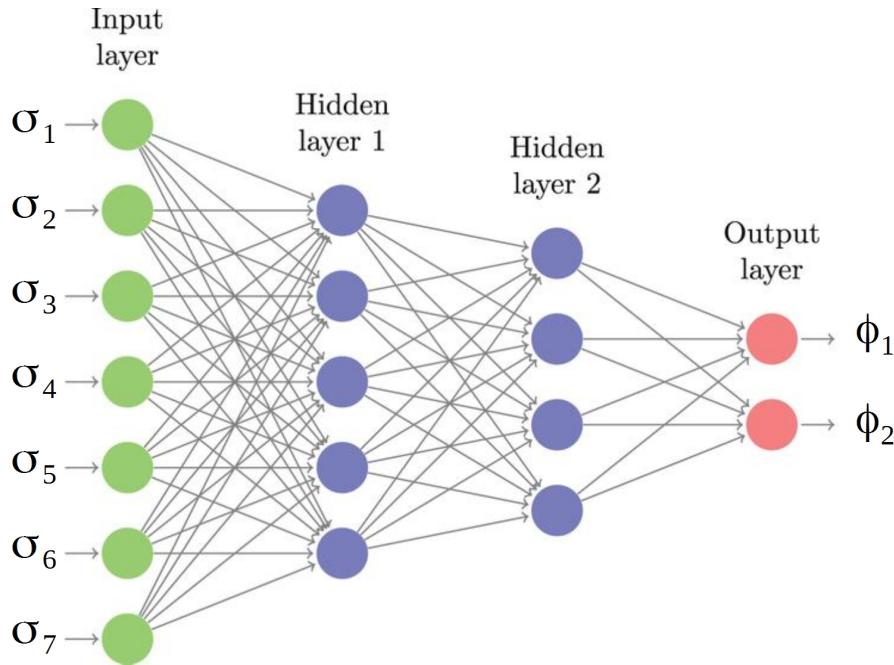


Figura 15.3: Esquema de una sencilla red neuronal. Los Inputs, o estímulos sensoriales, son los valores σ_i , que reciben las neuronas de lo que llamamos capa sensorial o de Input. A continuación las señales se van transmitiendo y procesando a lo largo de neuronas internas (Capas Internas) hasta llegar a las neuronas que generan la señal final ϕ_i (Output Layer, por ejemplo neuronas motoras)

Una neurona recibe una señal eléctrica (del orden de decenas de milivoltios) en sus ramificaciones de entrada (Dendritas), que se transmiten al cuerpo, o soma, de la misma. En el soma se realiza un cierto *procesamiento* de la señal, que también puede tener en cuenta el estado anterior (señales muy frecuentes pueden desactivar temporalmente a la neurona, por ejemplo) y finalmente *decide* si genera un potencial eléctrico a la salida, en el axón, que está conectado con varias dendritas de otras neuronas, como puede verse en la Figura 15.3. Este mecanismo en cada neurona de entrada y salida, produce grandes cascadas de actividad cuando recibimos un Input de los órganos sensoriales, cuando desencadenamos un cierto pensamiento...en general con cualquier actividad cerebral. El tiempo típico de cambio de los estados de las neuronas es del orden de unos pocos milisegundos. Las salidas de una neurona siempre llevan la misma señal (en el axón y sus ramificaciones).

Se configura así una Red Neuronal, que en algunos animales inferiores son de unas pocas neuronas, y en el hombre de varios miles de millones. El *Caenorhabditis elegans* (*C. elegans*) es un pequeño gusano transparente del que se conocen todas sus células una por una (tiene un total de 959); de ellas 302 son neuronas. Una cucaracha posee del orden de 10^6 neuronas.

Algunas neuronas son las que reciben las señales generadas en los órganos sensoriales (la señal de la retina, del oído, de la piel...), otras realizan el procesamiento de la información, otras envian señales a músculos... Otras neuronas *simplemente* generan el pensamiento abstracto, las ideas.

Las redes neuronales en el cerebro son ciertamente una Red y por tanto pueden estudiarse con la Teoría de Redes Complejas ya conocida por el alumno. De hecho pueden construirse esquemas funcionales de las neuronas y sus conexiones en el cerebro que corresponden con Redes Complejas similares a las estudiadas en el capítulo 14. En la Figura 15.4 puede verse un ejemplo.

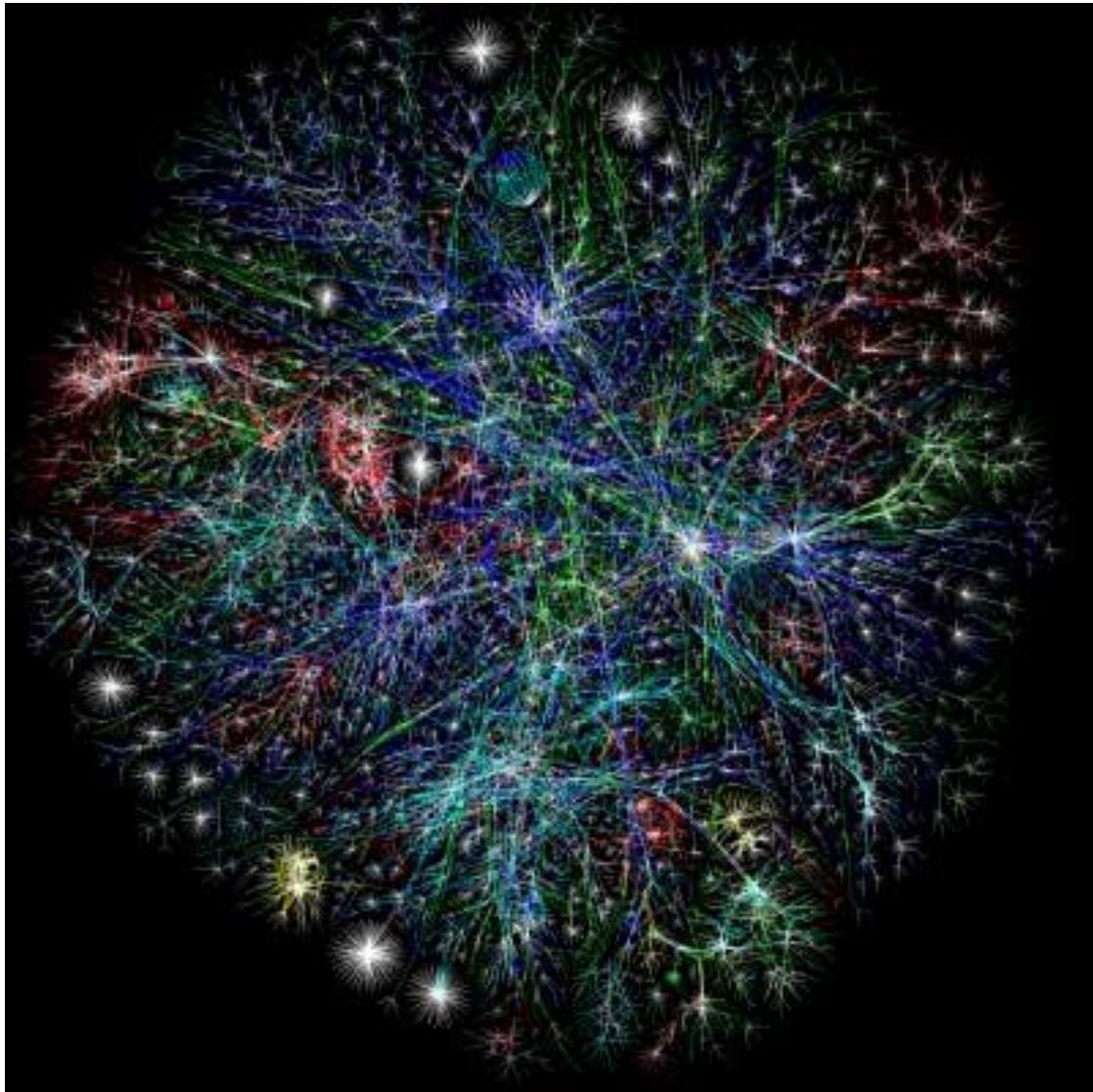


Figura 15.4: Red compleja que representa neuronas conectadas en un cerebro de un animal superior.

15.2. El perceptrón

El perceptrón es una realización relativamente básica de una neurona. El soma recibe inputs en muchas entradas diferentes, que procesa apropiadamente, generando una salida que es una función de los inputs.

El funcionamiento genérico del perceptrón (Ver Figura 15.5) se define del siguiente modo:

- En cada uno de sus inputs recibe un cierto valor $\sigma_i, i = 1, N$, que puede ser una variable analógica (un número Real) o una variable digital (0, 1 o $-1, 1$, por ejemplo)
- Cada Input llega al soma a través de una dentrita, que opera sobre el valor de entrada, enviando un potencial eléctrico dado por el producto del Input (σ_i) por un número dependiendo de la dentrita; llamaremos a este número, para la dentrita i , W_i . Supongamos N dentritas. Cuando W_i sea positivo diremos que la señal de la dentrita es estimuladora, y cuando W_i sea negativo, que la señal es inhibitoria. El soma recibe pues finalmente un potencial $W_i\sigma_i$ por cada dentrita.

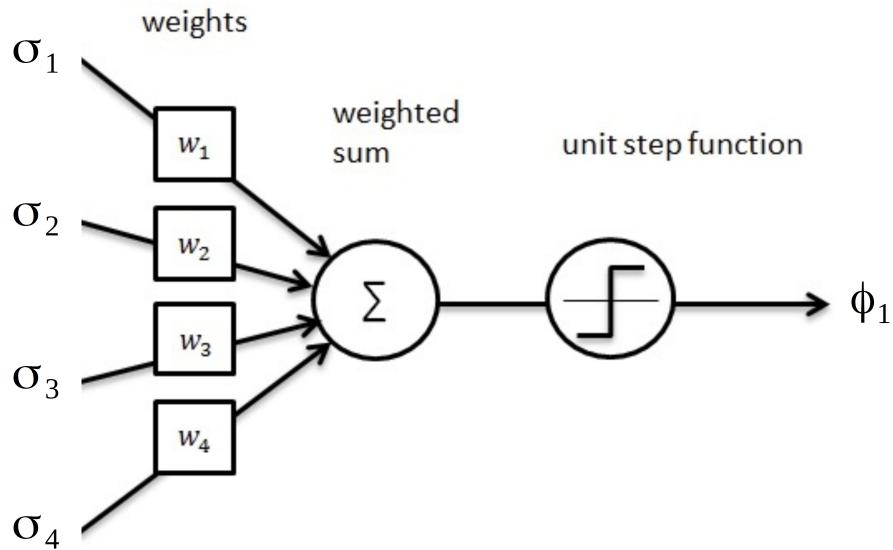


Figura 15.5: Esquema básico del perceptrón.

- El soma realiza la suma de todos los potenciales eléctricos que recibe de todas sus dendritas, calculando pues un potencial total $V_t = \sum_{i=1,N} W_i \sigma_i$
- Finalmente el soma genera un potencial eléctrico en su Axón, que es una función de V_t . Muchas neuronas usan una función escalón para ello, es decir $V_{Axon} = \theta(V_t - V_{Umbral})$. La función $\theta(x)$ está definida como

$$\theta(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0. \end{cases} \quad (15.1)$$

El valor de V_{Umbral} es, junto al valor de cada W_i , lo que caracteriza el funcionamiento de una neurona, *i.e.* los grados de libertad de nuestro sistema.

15.2.1. Funciones Lógicas con Perceptrones

Un simple perceptrón puede realizar funciones relativamente complejas como por ejemplo el reconocimiento de caracteres sencillos. Veáse por ejemplo la Figura 15.6.

En base a simples perceptrones, podemos reconstruir cualquier operación lógica al igual que los procesadores de silicio convencionales, o dicho de otra manera: una Red Neuronal puede hacer (al menos) lo mismo que cualquier ordenador. Para demostrar esto, basta con demostrar que podemos construir las funciones lógicas básicas (AND, NOT, OR y XOR por ejemplo) con neuronas.

Veámos por ejemplo como se puede construir una función XOR. Construimos la función XOR como $A \text{ XOR } B = (A \text{ OR } B) \text{ AND } (\text{NOT}(A \text{ AND } B))$.

Para construir el AND, conectamos los inputs a una neurona, con $W_A = 1$, $W_B = 1$ y Umbral $T = 3/2$.

Si $A = 1, B = 1$, tenemos que la entrada a la neurona vale 2, y como está por encima del umbral, la salida vale 1, correctamente.

Si las entradas son 0 y 1, la entrada al soma es 1, por debajo de $3/2$ con lo que la salida es 0. Y así sucesivamente.

Construimos luego el OR y el NOT, y todo integrado obtenemos la red neuronal con las W_i y los umbrales T que pueden verse en la Figura 15.7

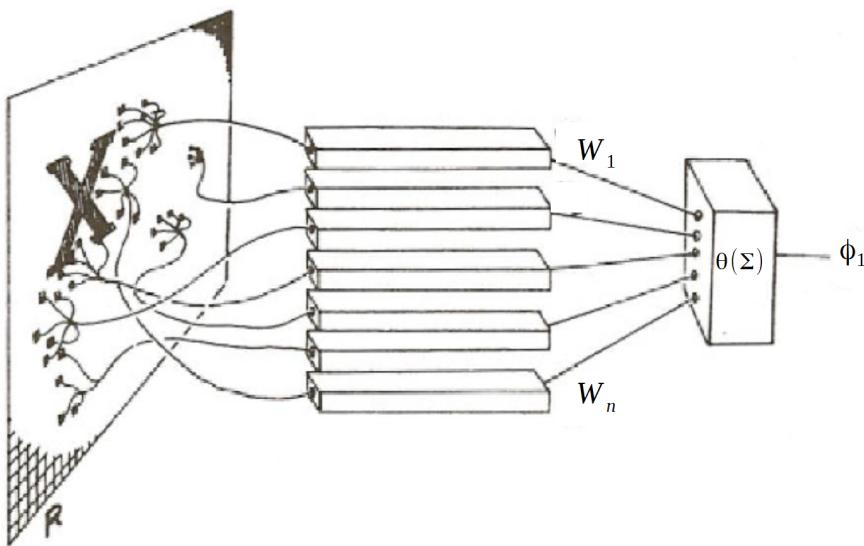


Figura 15.6: El perceptrón recibiendo una imagen para reconocer caracteres.

15.3. Reconocimiento de Patrones

Estudiemos ahora una funcionalidad básica de las Redes Neuronales, y que históricamente fue la que atrajo la atención sobre este modelo, como es el reconocimiento de imágenes. Reconocer caras, edificios, simples caracteres incluso, no es una tarea fácil usando los algoritmos tradicionales, y contiene aspectos muy interesantes, como la capacidad de identificar imágenes cambiadas de tamaño, perspectiva, con falta de detalles, etc... aspectos estos que se identifican de alguna manera con las capacidades cerebrales.

Consideremos una cierta imagen. Como es evidente podemos almacenarla como un patrón de bits. Reconocer pues una imagen consiste en dado un conjunto de patrones (las caras de todos nuestros conocidos), almacenar (memorizar) dichos patrones, y cuando veamos una imagen (nos encontramos con alguien por la calle...) buscar entre todas las imágenes (en nuestra memoria) qué imagen se corresponde con la que estamos viendo.

Consideremos pues que cada imagen viene dado por un vector de N componentes, cada una de las cuales puede ser $1, -1$.¹

15.3.1. Un solo Recuerdo

Supongamos que tenemos un sólo recuerdo, y queremos identificarlo. En este caso el problema se plantea en los siguientes términos: A la vista de una imagen parcial de un objeto que se parezca lo suficiente al conocido, debemos ser capaces de recordar el objeto completo, *i.e.* reconstruirlo. Es un proceso similar al que se produce en el cerebro cuando un sonido, una frase... desencadena un recuerdo completo.

Más concretamente, almacenando un vector $\vec{\xi}$ de N componentes, y partiendo de un vector inicial $\vec{\sigma}$, el proceso debe llevarnos de $\vec{\sigma}$ a $\vec{\xi}$ si ambos patrones se parecen lo suficiente. Efectivamente la frase *se parecen lo suficiente* es difusa... pero precisamente esa es una propiedad básica de algunas funciones cerebrales.

Bien, ahora el problema es: dado $\vec{\xi}$ (recuerdo), cómo calcular el valor del conjunto W_i para que cuando presentemos $\vec{\sigma}$ como Input, la red neuronal nos diga si se corresponde suficientemente con la imagen recordada.

¹Cuando estudiamos funciones lógicas tratamos con variables 0, 1. Cuando tratamos con Neuronas, Inputs, Conexiones, etc. a veces se consideran en su lugar variables $-1, 1$. Esto se hace para indicar que una conexión con $W = 1$ excita la neurona y una con $W = -1$ la inhibe, que es semejante al comportamiento real. Convertir variables $-1, 1$ en $0, 1$ es un trivial cambio lineal de variables.

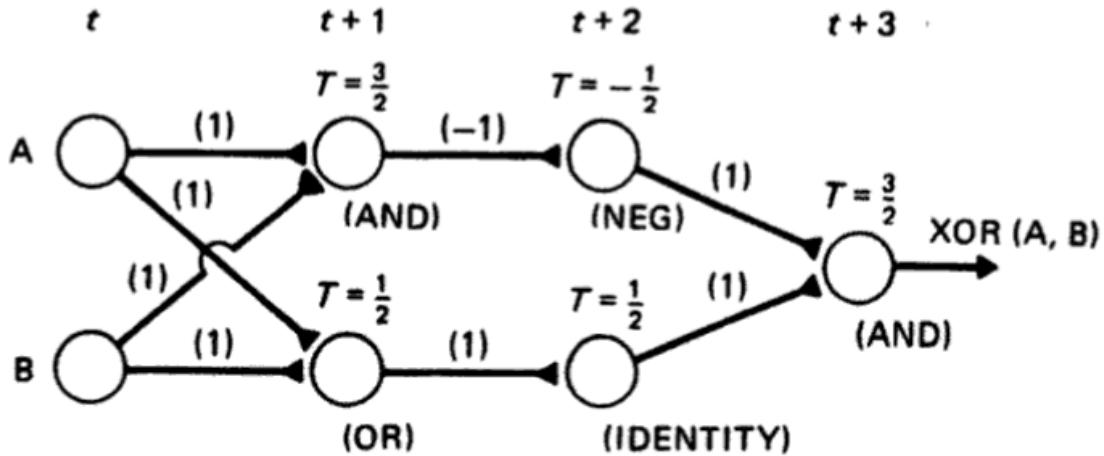


Figura 15.7: Red Neuronal para construir una función lógica, en este caso la salida es $A \text{ XOR } B$.

Consideremos N neuronas, cada una de las cuales recibe todos los Inputs. Esto es lo que llamamos un *Multi Perceptrón*, y puede verse en el Figura 15.8. Al tener más de una Neurona, para indexar correctamente el valor W de los Links entre un Input y una neurona, debemos utilizar dos índices, $W(m, n)$: el primer índice nos indica la neurona, el segundo la dentrita de entrada. En este caso, se nos presenta un vector de $\vec{\sigma}$ de N componentes, y a la salida queremos el vector $\vec{\xi}$ si el original se parece al almacenado, por tanto la entrada y la salida ($\vec{\sigma}$ y $\vec{\phi}$ tienen el mismo número de componentes).

Llaremos al output de la neurona j , $\phi_j, j = 1 \dots N$, al Input $\sigma_i, i = 1 \dots N$. La neurona k recibirá un Potencial igual a

$$V_k = \sum_j W(k, j) \sigma_j$$

Supongamos que el Umbral es Cero; el ouput de la Neurona i será finalmente

$$\phi_i = \theta\left(\sum_j W(i, j) \sigma_j\right)$$

Partiendo pues de una serie de Inputs que son -1 o 1 , obtenemos los outputs que son tambien valores -1 o 1 , dada nuestra definición de la función $\theta(x)$.

Veamos como podemos construir la matriz $W(i, j)$. Para ello usaremos algunas propiedades básicas de operadores lineales en espacios vectoriales.

Sabemos que un *Proyector* es un operador (matriz) que actuando sobre un cierto subespacio deja sus vectores dentro del subespacio, y sobre un vector general, lo convierte (Proyecta) en un vector de subespacio.

En este caso sin embargo las operaciones no son lineales y la situación es bien diferente, no obstante sirva como una aproximación para entender mejor el proceso.

Construyamos $W(i, j)$ del siguiente modo,²

$$W(i, j) = \frac{1}{N} \xi_i \xi_j$$

Recordando que $\xi_i^2 = 1 \forall i$ por construcción, y que $\langle \vec{\xi} | \vec{\xi} \rangle = N$ es fácil comprobar que la matriz W con índices $W(i, j)$ es el proyector sobre el subespacio generado por el vector $\vec{\xi}$, es decir $W(i, j) \equiv P_{\vec{\xi}}$. Efectivamente

$$W|\vec{\xi}\rangle = \frac{1}{N} |\vec{\xi}\rangle \langle \vec{\xi} | \vec{\xi} \rangle = |\vec{\xi}\rangle$$

²La matriz $W(i, j)$ podría escribirse tambien en notación de Dirac (*brackets*) como $W = \frac{1}{N} |\vec{\xi}\rangle \langle \vec{\xi}|$.

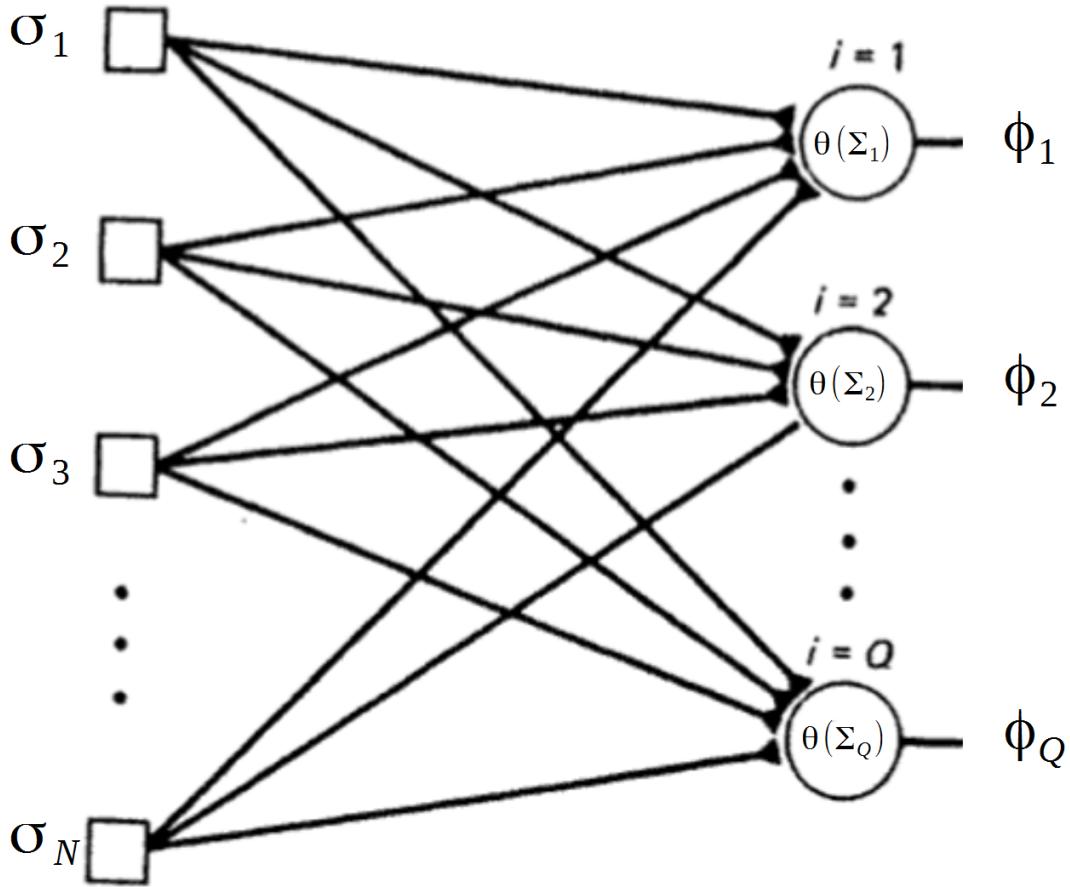


Figura 15.8: Multi Perceptrón ilustrando las conexiones para el reconocimiento de Imágenes. En el caso del texto, dado que el vector de salida debe reproducir el recuerdo almacenado, tendrá las mismas componentes que el vector de entrada; de este modo aquí $Q = N$

con la propiedad de que para valores de $\vec{\sigma}$ próximos a $\vec{\xi}$, cumple que

$$P_{\vec{\xi}} \vec{\sigma} \propto \vec{\xi}$$

Añadiendo la función θ eliminamos la proporcionalidad y caemos en $\vec{\xi}$ en cualquier caso... salvo que nuestro input $\vec{\sigma}$ esté en el subespacio ortogonal a $\vec{\xi}$ en cuyo caso este procedimiento nos lleva al vector nulo. Excepto en este caso, tenemos un procedimiento operativo para reconocer el patrón.

Para verlo explicitamente, consideremos un input dado por precisamente el recuerdo, es decir el input es el vector $\vec{\xi}$. El output vendrá dado por

$$\phi_i = \theta\left(\sum_j W(i, j)\xi_j\right) = \theta\left(\sum_j \frac{1}{N}\xi_i\xi_j\xi_j\right)$$

pero $\xi_j\xi_j = 1$, con lo cual

$$\phi_i = \theta\left(\sum_j \frac{1}{N}\xi_i\right) = \theta\left(\frac{1}{N}N\xi_i\right) = \xi_i$$

que es lo que queríamos: que la salida ϕ_i coincida con la entrada ξ_i , por tanto $\vec{\phi} = \vec{\xi}$.

15.3.2. Varios recuerdos

Supongamos que tenemos varios recuerdos o patrones para aprender a recordar; cada recuerdo lo etiquetamos con un índice superior μ , de modo que tenemos $\vec{\xi}^\mu, \mu = 1, N_p$, con N_p el número de patrones a recordar. Remarquemos que $\vec{\xi}^\mu$ es un vector, cuyas componentes son ξ_i^μ . Ahora construimos la matriz $W(i, j)$ como la suma de los proyectores anteriores sobre cada patrón o recuerdo.

$$W(i, j) = \frac{1}{N} \sum_{\mu=1}^{N_p} \xi_i^\mu \xi_j^\mu$$

y construimos el output como

$$\phi_i = \theta\left(\sum_j W(i, j) \sigma_j\right)$$

Dado un vector arbitrario a la entrada, aplicando la matriz $W(i, j)$ sobre el mismo deberíamos acercarnos a uno de los *recuerdos*. Realmente esto no se consigue en un solo paso: necesitamos iterar el proceso varias veces para lograr caer exactamente en uno de los recuerdos. Por tanto el proceso completo consiste en partir de un input dado $\vec{\sigma}$, aplicarle la matriz W para conseguir un output $\vec{\phi}$, y a continuación usar este vector de nuevo como input, iterando hasta conseguir un punto estable, o simplemente iterando un número fijo de veces.

15.3.3. Identificando el recuerdo

Como en general la salida no corresponde exactamente con ninguna entrada (puede haber algún bit de diferencia), para saber qué recuerdo hemos seleccionado, dada una salida, calculamos la *distancia* de la solución a cada uno de los recuerdos, y elegimos como seleccionado el de menor distancia. La distancia la podemos definir como el número de bits diferentes (Esto de corresponde con la *distancia de Hamming* entre la entrada y la salida). Llamaremos *overlap* entre dos vectores al número de componentes del vector iguales.

Por supuesto un vector inicial parecido a uno dado, a veces es mal identificado, pero esto es algo que también le sucede al cerebro humano.

Todos estos conceptos se recogen en el Código del Ejercicio [15.6.1](#).

15.4. Procesos de Aprendizaje Supervisado

El cerebro de los animales, y el del hombre especialmente, es capaz de aprender cosas nuevas. ¿Son nuestras redes neuronales capaces de aprender? Estudiaremos aquí el proceso de aprendizaje supervisado: en el caso humano una persona aprende a ir en bicicleta practicando y cayéndose; el proceso mental correcto para el equilibrio es el que nos mantiene de pie cuando montamos en la bicicleta; si hace algo mal se cae. Esto es aprendizaje supervisado. El proceso es similar al de aprender que beber agua, comer, huir del fuego... es bueno.

Estudiemos ahora un proceso muy sencillo como ejemplo.

Una Neurona recibe como Input $2N$ bits y debe aprender una cosa relativamente compleja: debe calcular el número binario correspondiente a los N primeros bits (lo llameremos a), el correspondiente a los N últimos (b), y devolver la diferencia ($a - b$) en decimal. Para ello debe aprender que la cadena de los N primeros bits representa un número en binario, otro la cadena de los N bits segundos, y a continuación debe aprender que queremos que *reste* los dos, todo ello con un perceptrón básico.

Pensemos en este problema como un proceso cerebral. La neurona puede ir cambiando sus valores de W_i y su valor de V_{Umbral} , de modo que a base de prueba y error (Evolución Natural) al final acierte con dar el resultado correcto. Si la vida del *propietario* de la Neurona dependiera de esto, es un buen mecanismo evolutivo donde los no adaptados mueren sin reproducirse; en este caso los valores no *adaptados* de W_i no se reproducen y los *adaptados* logran reproducirse en un ser más adaptado que su progenitor... y así sucesivamente.

Dado que en este caso en el output queremos un número entero arbitrario, podríamos resolver el problema con N outputs binarios, o con un simple output entero (*int*): para lograr esto último basta considerar valores enteros arbitrarios de W , no solo 1, -1, y eliminar la función θ a la salida. Usaremos esta opción en este ejemplo.

Los *grados de libertad* de que disponemos son los valores de W_i . Supongamos el umbral V_{Umbral} cero. El proceso de aprendizaje es encontrar el conjunto de $W_i, i = 1, 2N$ que resuelven el problema³. Es decir, si consideramos el perceptrón de la figura 15.5 con $2N$ entradas $W_j \in \mathcal{Z}$, y con inputs σ_i , sin la función salto a la salida. En este caso el output será:

$$\phi = \sum_j W_j \sigma_j$$

de modo que debemos conseguir $\phi = a - b$.

El proceso de aprendizaje supervisado consiste en partir de unos valores de W_i , y presentar una entrada de datos. Si la salida es correcta, perfecto; en caso contrario debemos modificar algunos de los W_i para mejorar la situación. En general llamamos *teacher* al proceso por el cual le decimos al perceptrón si está haciendo bien las cosas o no. La presencia de un teacher es imprescindible en el caso de aprendizaje Supervisado.

Esta fase de ir presentando datos a la entrada, que el sistema analiza, y que nosotros le decimos si está bien o mal, es el proceso de aprendizaje.

De modo que iremos presentando conjuntos en la entrada y cambiando las W_i para que la neurona vaya aprendiendo a hacer bien su trabajo.

Una vez finalizado este proceso, para verificar si realmente ha aprendido, debemos presentar al sistema una cantidad significativa de datos nuevos de entrada, y calcular la tasa de éxito: si la tasa de acierto es puramente la tasa de acierto al azar (o menor...), el sistema no ha aprendido nada; si la tasa es elevada relativa a la del azar, el sistema realmente ha aprendido.

Veremos a continuación cómo resolver este problema con lo ya conocido por el alumno.

15.4.1. Simulated Annealing

Este proceso iterativo de comenzar con unos valores W_i e irlos cambiando para que hagan mejor una cierta tarea, es en realidad un proceso de Optimización, y sabemos como hacerlo de modo eficiente. Podemos utilizar el algoritmo de Simmulated Annealing (Capítulo 13) para encontrar el mejor valor de las W_i .

En este caso de aprendizaje supervisado, la Energía para nuestro proceso es fácil de definir; sea E_{NN} el valor que nos devuelve nuestro perceptrón. Sea $E_{teacher}$ el valor exacto, definiendo entonces la Energía como

$$E = (E_{NN} - E_{teacher})^2$$

Nuestras variables son los valores de $W_i, i = 1, 2N$. Partimos de un conjunto elegido al azar, y vamos cambiando sus valores y aceptando o no los cambios según la Energía anterior, durante un proceso de enfriamiento.

En el Ejercicio 15.6.2 se plantea el problema y se presenta una versión básica del código.

15.4.2. Hebb Rule

Presentamos otro método para entrenar Redes Neuronales con aprendizaje supervisado. Consideraremos una leve modificación del problema anterior, donde dados los $2N$ bits de entrada, la salida no será la diferencia de los dos números, sino queremos que el perceptrón nos devuelva 1 si $a > b$, -1 si $a < b$ y 0 si $a = b$ (No preocuparse mucho por el *improbable* caso de $a = b$). Volvemos a considerar pues una salida digital en vez de entera.

En este proceso de aprendizaje supervisado, dados los valores de entrada a y b , el teacher nos devuelve -1, 0, 1 de acuerdo a la prescripción anterior.

³Es bastante simple resolver analíticamente el problema de modo que se pueden calcular exactamente los valores de W_i , si bien en este caso lo que queremos es que el sistema, sin más ayuda que el teacher, lo resuelva solo

Para resolver este problema usaremos la *Regla de Hebb*, que está basada en que en una red neuronal, si el input y el output tienen el mismo signo, esa señal debe ser reforzada (aumentar W_i). Si el input tiene diferente signo del output, entonces la señal debe ser disminuida (disminuir W_i).

En nuestro ejemplo, la salida de la neurona será

$$\phi = \theta(\sum_j W_j \sigma_j)$$

dónde debemos calcular el conjunto de valores W_j . La regla de Hebb puede ser escrita como

$$W_{new}[i] = \xi[i] * \vec{T}[\xi]$$

donde $\xi[i]$ es el input i a la neurona, y $\vec{T}[\xi]$ es el valor correcto para un input ξ , es decir lo que llamamos el valor del *Teacher*.

Iterando este proceso, recorriendo un número suficiente de valores de entradas para ir entrenando a la Red Neuronal, podemos acabar con un conjunto correcto de valores para las W_i .

En el ejercicio 15.6.3 y en el código asociado, puede verse una implementación concreta del problema de esta sección.

15.5. Attractores

Cuando consideramos neuronas que se conectan a sí mismas, introducimos un mecanismo que permite que dado un input, tras generarse el output, éste modifica el Input de nuevo, generando otro Input diferente, que genera otro Output, y así sucesivamente, en un proceso dinámico que puede converger a algún valor fijo o a un ciclo, o bien presentar comportamientos erráticos. Incluso partiendo de un Input fijo, la dinámica de Update de los potenciales, el orden usado, etc. cambian significativamente la evolución temporal de los estados de la neuronas.

Decimos que cuando este proceso acaba en un punto o un ciclo, estamos en presencia de un *Atractor*. La relación con la actividad cerebral es que los procesos cognitivos pasan una serie de estadios sucesivos; un recuerdo lleva a otros, o un pensamiento lógico a otros, y finalmente acabamos en algo concreto, que emularia el atractor.

Todo esto es implementable con Redes Neuronales.

Veamos un ejemplo. Consideremos una red neuronal con 4 neuronas e interacción entre ellas, de manera que en cada paso temporal (similar al ciclo elemental de una neurona), la neurona recibe un input (σ_i) y genera un output en el *ciclo* siguiente, (ϕ_i), según la expresión

$$\phi_i = \theta(\sum_j W(i, j) \sigma_j)$$

que podemos representar matricialmente

$$\vec{\phi} = \theta(W\vec{\sigma}) \tag{15.2}$$

donde la función $\theta(\vec{x})$, significa que se aplica a cada de sus componentes, es decir

$$(\theta(\vec{x}))_i \equiv \theta(\vec{x}_i)$$

Nos falta definir la matriz W ; tomemos por ejemplo

$$W = \begin{pmatrix} 0 & 1 & 1 & 1 \\ -1 & 0 & -1 & -1 \\ 1 & -1 & 0 & 1 \\ -1 & -1 & -1 & 0 \end{pmatrix} \tag{15.3}$$

Dado un conjunto de Inputs σ , generamos el Output, que es introducido de nuevo como Input, en un ciclo ininterrumpido. Es decir, calculamos el output a partir del input original y dicho output es utilizado com Input en el *ciclo neuronal* siguiente. Es decir en un ciclo neuronal,

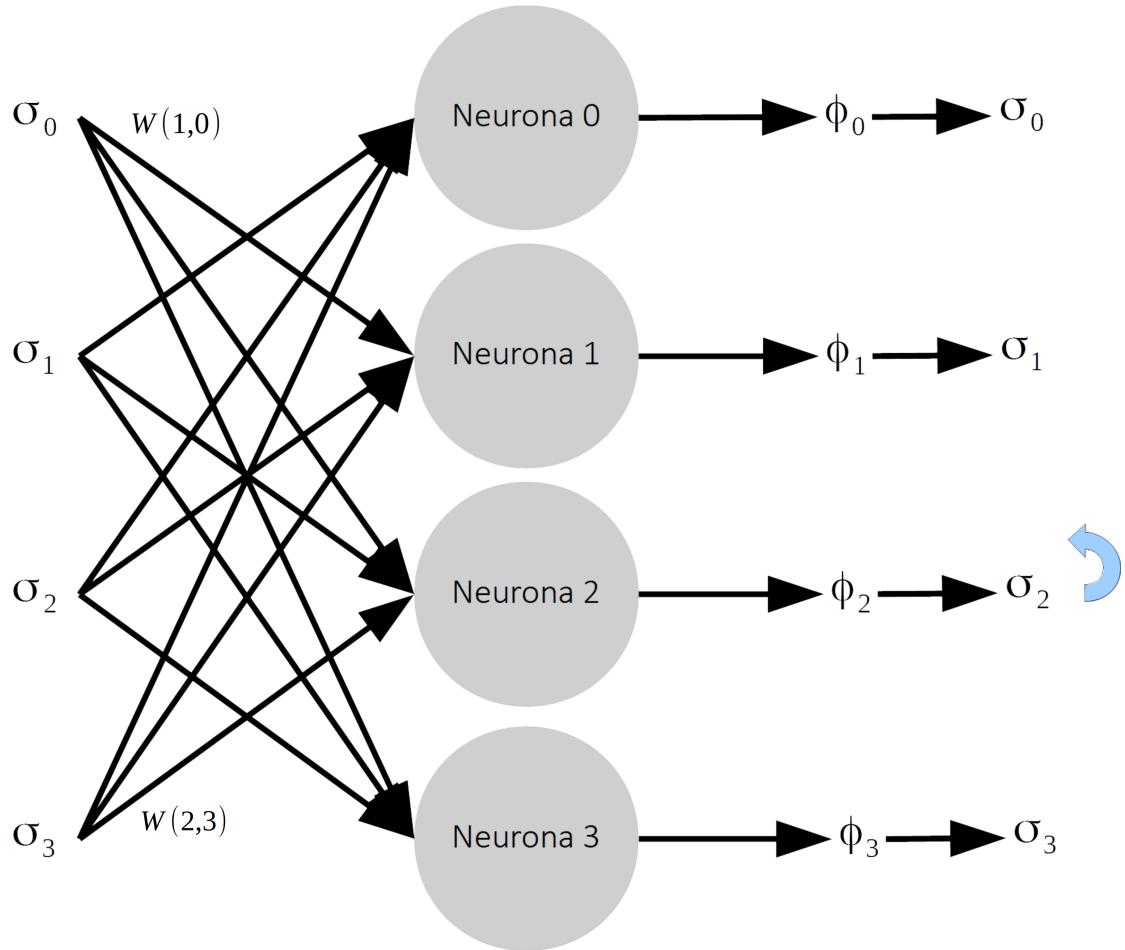


Figura 15.9: Red Neuronal utilizada para ilustrar el comportamiento de un Atractor Neuronal. Cada conexión de σ_i con la Neurona j tiene el valor $W(j, i)$. Dado un output en un momento dado, al cabo de un cierto tiempo δt se genera un output $\vec{\phi}$, que en el tiempo $+ \delta t$ siguiente, es usado como input, es decir, asignado a las variables $\vec{\sigma}$.

tomamos fijos todos los Inputs, y calculamos todos los outputs. Esto es conocido como *dinámica secuencial síncrona*.

Existen otras opciones, como por ejemplo la *dinámica asíncrona*, donde cambiamos uno solo de los outputs, por ejemplo σ_2 , y volvemos a calcular la salida: en general un cambio de un solo input puede producir cambios en todos los outputs. La dinámica asíncrona puede ser secuencial (donde los inputs se cambian por orden) o random (donde los inputs se cambian con un orden aleatorio).

Las diferentes dinámicas corresponden a diversas hipótesis sobre el funcionamiento neuronal; son más naturales las asíncronas *random*, donde no hay una dinámica de ciclos completos y el output que cambia cada vez se elige al azar.

No obstante, por simplicidad aquí usaremos la dinámica síncrona.⁴

Podemos estudiar ahora como se comporta el sistema, según el primer conjunto de datos presentados. En este caso los valores utilizados para representar el estado de la neurona son $\{-1, 1\}$. Consideremos como ejemplo el estado $v_0 \equiv (1, -1, -1, 1)$. Calculamos ahora el estado

⁴Dado que supondremos dinámica síncrona, la interacción de una neurona consigo misma no se produce en el mismo ciclo, sino sólo al ciclo siguiente: esto significa que nuestra matriz de conexiones W tiene la ligadura de que $W(i, i) = 0 \forall i$.

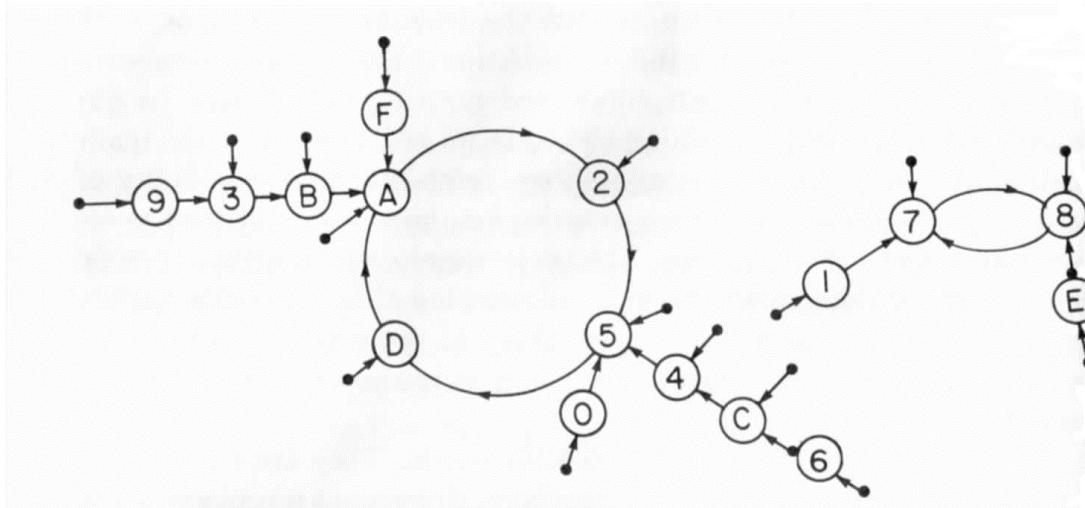


Figura 15.10: Evolucion en el Atractor Neural definido por la matriz W de la expresión 15.3. Cada número es la etiqueta (número binario asociado) del valor del vector $\vec{\sigma}$. La flecha con punto negro indica el valor de partida. Las flechas subsiguientes, la evolución según la dinámica sincrona. Por ejemplo si empezamos en el 2, obtenemos la secuencia 2, 5, D, A, 2, 5, D, A, ... repitiéndose indefinidamente.

neuronal siguiente; para ello primero calculemos

$$Wv_0 = (-1, -1, 3, 1)$$

y si ahora aplicamos la función θ a cada una de sus componentes, obtenemos para el siguiente estado neuronal $v_1 = (-1, -1, 1, 1)$.

Para simplificar la notación, etiquetamos a cada vector de entrada σ por el valor del número en Hexadecimal que representa, suponiendo que el -1 es un 0 ; es decir $\sigma = (1, 1, -1, 1) \equiv (1, 1, 0, 1) \equiv 13 \equiv D$, usando la base Hexadecimal.

Cuando aplicamos la expresión 15.2 al vector $D = (1, 1, 0, 1) \equiv (1, 1, -1, 1)$, obtenemos tras la proyección $(1, 0, 1, 0) \equiv A$; si ahora usamos A como entrada, obtenemos 2 como salida, etc.

El resultado final puede verse en la Figura 15.10. La interpretación es la siguiente. Si presentamos a la Red uno de los inputs $0, 2, 3, 4, 5, 6, 9, A, B, C, D$, la red neuronal evoluciona al *estado* cíclico $A, 2, 5, D$, que imaginemos desencadena el proceso de “buscar agua”. Si comenzamos por $1, 7, 8, E$, la red acaba en el estado cíclico $7, 8$, por ejemplo “esconderse”. Diferentes Inputs, nos llevan a dos situaciones posibles, correspondiendo a la respuesta cerebral diferente según el punto original. Este ciclo final cerrado es lo que da nombre a los Atractores, que corresponde al estado final del cerebro tras asociar un input sensorial con algo reconocido.

Remarcamos que el comportamiento concreto de este Atractor, depende exclusivamente de la matriz W ; otra matriz diferente da lugar a otra topología entre los estados. Grandes matrices (miles de filas y columnas) dan lugar a un rico comportamiento de atractores, ciclos, puntos fijos, etc. tal vez tan rico y poblado de recuerdos e ideas como el cerebro de un animal superior.

15.6. Ejercicios

15.6.1. Reconocimiento de Patrones

Escribir un programa que implemente los conceptos de reconocimientos de Patrones de la sección 15.3. En el apartado 15.7.1 se da un ejemplo, en el cual, usando `#define N_P 1` implementaremos el caso de un solo recuerdo, y con un valor mayor, varios recuerdos.

15.6.2. Funciones Matemáticas con un Perceptrón

Dados $2N$ bits de Input, un perceptrón debe identificar que los primeros N bits representan un número en binario, los N últimos otro número, y debe devolvernos la diferencia entre ellos. Usar el método de Simulated Annealing. En el apartado 15.7.2 puede verse un sencillo código que implementa esto.

15.6.3. Funciones Lógicas con un Perceptrón

Dados $2N$ bits de Input, un perceptrón debe identificar que los primeros N bits representan un número en binario a , los N últimos otro número b , calcular su diferencia $c = a - b$ y devolvernos 1 si $a > b$, 0 si $a = b$ y -1 si $a < b$. Usar la regla de Hebb para el proceso de aprendizaje. En el apartado 15.7.3 puede verse un código de ejemplo.

15.7. Código en C

15.7.1. Reconocimiento de patrones

```

/*
Neural Networks for pattern reconigtion
Alfonso Tarancon Lafita
Noviembre de 2018
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define EOL '\n'
#define N_N 100 // Numero de neuronas maximas
#define N_P 10 //Numero de patterns iniciales maximos
#define fran (rand()/(double)RAND_MAX+1)
double xi[N_P][N_N];
double J[N_N][N_N];

void read_input_patterns(char *na,int nn,int np);
void construct_J(int nn,int np);
double Calcula_Overlap(double *t,int n,int p,int *ind);
void A_por_vector(double *p,double *q,int n);
void Proyecta(double *p,int n);
void copia(double *p,double *q,int n);
void escribe_vector(double *p,int n);
void construct(double *t,int n, int f);
void Cambia_signo(double *p,int n);

int main(int argc, char **argv)
{
    int i,k;
    char name[256];
    int Niter,Ntest,ind,n_n,n_p,flag;
    double O;
    double test[N_N],test_n[N_N];
    switch(argc)
    {
        case 5:
            sscanf(argv[1],"%s",name);
            sscanf(argv[2],"%d",&n_n);
            sscanf(argv[3],"%d",&n_p);
            sscanf(argv[4],"%d",&flag);
            break;
        case 4:
            printf("Error en linea de comandos");
            exit(1);
        case 3:
            printf("Error en linea de comandos");
            exit(1);
        case 2:
            printf("Error en linea de comandos");
            exit(1);
        default:
            strcpy(name,"input1.dat");
            n_n=20;
            n_p=1;
            flag=0;
            break;
    }

    printf("Opening file: %s\n",name);
    read_input_patterns(name,n_n,n_p);
    Niter=20;
    construct_J(n_n,n_p);
    Ntest=10;

    for(k=0;k<Ntest;k++)

```

```

{
    construct(test,n_n,flag);
    Proyecta(test,n_n);
    escribe_vector(test,n_n);
    printf("Starting test %d...\n",k);

    for(i=0; i<Niter; i++)
    {
        A_por_vector(test,test_n,n_n);
        Proyecta(test_n,n_n);
        O=Calcula_Overlap(test_n,n_n,n_p,&ind);
        copia(test_n,test,n_n);
    }
    printf("final\n");
    escribe_vector(test,n_n);
    printf("Q=%lf,ind=%d\n",O,ind);
}
return 0;
}

void read_input_patterns(char *na,int n_n, int n_p)
{
    FILE *input;
    int i,j;
    int a;

    input=fopen(na,"rt");
    for(i=0; i<n_p; i++)
    {
        for(j=0; j<n_n; j++)
        {
            fscanf(input,"%d ",&a);
            xi[i][j]=a;
        }
    }

    for(i=0; i<n_p; i++)
    {
        for(j=0; j<n_n; j++)
        {
            printf("%d ",(int)xi[i][j]);
        }
        printf("\n");
    }
}

void construct_J(int nn,int np)
{
    int i,j,k,sum;
    for(i=0; i<nn; i++)
    {
        for(j=0; j<nn; j++)
        {
            for(sum=k=0; k<np; k++)
                sum+=xi[k][i]*xi[k][j];
            J[i][j]=sum/(double)np;
            // if(i==j)J[i][j]=0; La verdad que no es relevante a este nivel
        }
    }

    for(i=0; i<nn; i++)
    {
        for(j=0; j<nn; j++)
        {
            printf("%lf ",J[i][j]);
        }
        printf("\n");
    }
}

```

```

}

double Calcula_Overlap(double *t,int nn,int np,int *ind_max)
{
    double max,sum;
    int i,j;
    max=-99999;
    for(j=0; j<np; j++)
    {
        sum=0;

        for(i=0; i<nn; i++)
            sum+=t[i]*xi[j][i];
        printf("j=%d,0=%lf\n",j,sum);
        if(sum/nn<(-0.5)) //Para evitar problemas de atractor por simetria
        {
            Cambia_signo(t,nn);
            sum=-sum;
        }

        //sum=fabs(sum);
        if(sum>max)
        {
            max=sum;
            *ind_max=j;
        }
    }

    return max/nn;
}

void A_por_vector(double *p,double *q,int n)
{
    int i,j;
    for(i=0; i<n; i++)
    {
        q[i]=0;
        for(j=0; j<n; j++)
            q[i]+=J[i][j]*p[j];
    }
}

void Proyecta(double *p,int n)
{
    int i;
    for(i=0; i<n; i++)
    {

        if(p[i]>=0)p[i]=1;
        if(p[i]<0)p[i]=-1;
    }
}

void copia(double *p,double *q,int n)
{
    int i;
    for(i=0; i<n; i++)
        q[i]=p[i];
}

void escribe_vector(double *test,int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%2d ",(int)test[i]);
    printf("\n");
}

void construct(double *t,int n,int f)
{
}

```

```

int i;
printf("f=%d\n",f);
if(f==2)
    copia(xi[0],t,n);

if(f==1)
{

for(i=0;i<n;i++)
    t[i]=1;

t[n-2]=-1;
t[n-1]=-1;
}

if(f==0)
{
    for(i=0; i<n; i++)
    {
        fran;
        if(fran<0.5)
            t[i]=-1;
        else
            t[i]=1;
    }
}
void Cambia_signo(double *p,int n)
{
    int i;
    for(i=0; i<-n; i++)
        p[i]=p[i];
}

```

Ejemplo de Fichero de Input para el caso default.

```

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1 1 -1

```

15.7.2. Aprendizaje supervisado: Simulated Annealing

```

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

#define T_MAX 1000
#define N_MAX 100

#define Ran (rand()/(double)RAND_MAX+1)
#define DRan (2.0*(0.5-Ran))

int J[N_MAX];
int w[T_MAX][N_MAX];

int Evalua_ANN(int n,int *wt);
double Energy(int n);
void genera_muestra(int n);
void Inicializa(int n);
void Monte_Carlo(int n,double eps,int *to,int *ac);
int teacher(int n,int *wt);
void calcula_valor(int *wt,int *uno,int *dos,int n);
void Resultados(int n,int n_t);

double Beta;
int N_t;

```

```

FILE *fevol;

main()
{
    int i,j,N,Niter;
    double epsilon;
    int E_ANN,E_Exacta;
    int Error;
    double Beta_Max,Beta_Ini,delta_Beta;;
    int acept,tot;
    int ms,mesfr;
    double Acept_Anterior,Acept_Limit,En;
    int N_test_final;

    fevol=fopen("Evol.dat","wt");
    N=24; // Dentritas del Perceptron: en realidad datos de entrada
    N_t=100;//Muestras para entrenamiento

    Niter=200; //Iteraciones para convergencia de J_ij
    mesfr=400;
    Beta_Max=50;
    Beta_Ini=1;

    Beta=Beta_Ini;
    delta_Beta=((Beta_Max-Beta_Ini)/(Niter-1));
    Acept_Anterior=1.0;

    Inicializa(N);
    Resultados(N,N_t);

    epsilon=2.0; //Como no sabemos la escala, hacemos un cambio multiplicativo... o sea cambios exponenciales

    for(i=0;i<Niter;i++)
    {
        genera_muestra(N);

        tot=acept=0;
        for(ms=0;ms<mesfr;ms++)
            Monte_Carlo(N,epsilon,&tot,&acept);

        printf("i=%d,tot=%d,acept=%d ",i,tot,acept);
        if(((double)acept/tot<0.05)&&(i<0.95*Niter))
        {
            Beta=Beta*0.925;
            if(Acept_Anterior<0.1)
                Beta=Beta*0.6;

        }
        else
            Beta=Beta+delta_Beta;

        Acept_Limit=0.1*(double)(Niter-i)/Niter;

        if(Acept_Anterior>Acept_Limit)
            Beta=Beta*1.6;

        if(Beta<0.000001)
            Beta=0.0001;

        if(Beta>10000)
            Beta=1000;

        Acept_Anterior=(double)acept/tot;
        En=Energy(N);
        printf("Beta=%lf, E=%lf \n",Beta, sqrt(En)/N_t);
    }
}

```

```

fprintf(feval,"%lf %lf %lf \n",Beta,Acept_Anterior,En);

}

Resultados(N,N_t);

//Comprobacion con nuevos inputs
N_test_final=N_t;
double Error_tot=0;

genera_muestra(N);
for(i=0;i<N_test_final;i++)
{
    E_ANN=Evalua_ANN(N,w[i]);
    E_Exacta=teacher(N,w[i]);

    Error=(E_ANN-E_Exacta)*(E_ANN-E_Exacta);
    Error_tot+=Error;

    printf("i:%d E_ANN=%d,E_Exacta=%d, Error=%lf\n",i,E_ANN,E_Exacta,sqrt((double)Error));
}

printf("Error Total=%lf\n",sqrt(Error_tot)/N_test_final);

fclose(feval);
}

void Monte_Carlo(int n,double eps,int *to,int *ac)
{
    int i,k,is,i_changed,J_old;
    double E_i,E_f;
    double delta_J,exit_cero;
    double Delta_temp,Boltzmann;

    for(is=0;is<n;is++)
    {
        (*to)++;
        E_i=Energy(n);

        delta_J=DRan*eps;
        exit_cero=4*DRan;

        i_changed=is;

        J_old=J[i_changed];
        J[i_changed]=(int)(exit_cero+J[i_changed]*delta_J);

        E_f=Energy(n);

        Delta_temp=Beta*(E_f-E_i);

        Boltzmann=exp(-Delta_temp);

        if((Boltzmann<Ran)|| (Delta_temp==0))//OJO: si no se pone esto, acepta con igual energia, engagnando a la aceptacion...
        {
            J[i_changed]=J_old;
        }
        else
        {
            if(Delta_temp>0) //Para cambiar beta, solo si mejora la energia...
                (*ac)++;
        }
    }
}

```

```

}

void Resultados(int n,int n_t)
{
    int j;

    printf("\n Conexiones finales:\n");

    printf("J:");
    for(j=0;j<n;j++)
        printf("%d ",J[j]);
    printf("\n");
}

void genera_muestra(int n)
{
    int i,j;

    for(i=0;i<N_t;i++)
        for(j=0;j<n;j++)
    {
        if(Ran<0.5)
            w[i][j]=1;
        else
            w[i][j]=0;
    }
}

void Inicializa(int n)
{
    int i;

    for(i=0;i<n;i++)
    {
        J[i]=1;//n*Dran;
    }
    /* Solucion exacta
       for(i=0;i<n/2;i++)
    {
        J[i]=pow(2,i);
    }

       for(i=n/2;i<n;i++)
    {
        J[i]=-pow(2,i-n/2);
    }
    */
}

int teacher(int nn,int *wt)
{
    int uno,dos;
    calcula_valor(wt,&uno,&dos,nn);

    return (uno-dos);

}

void calcula_valor(int *wt,int *uno,int *dos,int n)
{
    int i,sum1,sum2,fact;
}

```

```

fact=1;
for(sum1=i=0; i<n/2; i++)
{
    sum1+=(wt[i]*fact);
    fact*=2;
}
fact=1;
for(sum2=i=0; i<n/2; i++)
{
    sum2+=(wt[i+(n/2)]*fact);
    fact*=2;
}
*uno=sum1;
*dos=sum2;
}

double Energy(int n)
{
    int i,E0,E1,Ener;

    for(Ener=i=0;i<N_t;i++)
    {
        E0=Evalua_ANN(n,w[i]);
        E1=teacher(n,w[i]);

        Ener+=(E0-E1)*(E0-E1);
    }
    //printf("*****E0=%lf,E1=%lf\n",E0,E1);

    return sqrt(Ener)/N_t;
}

int Evalua_ANN(int n,int *wt)
{
    int Ener;
    int i;

    Ener=0; //Primera capa
    for(i=0;i<n;i++)
        Ener+=J[i]*wt[i];

    return Ener;
}

```

15.7.3. Aprendizaje supervisado: Hebb rule

```

/*
Neural Networks for logical function
Alfonso Tarancón Lafita
Noviembre de 2018
*/

/*
Para mostrar las cuestiones conceptuales del perceptrón...
No es un buen algoritmo para convergencia...
*/

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

```

```

#define N_N 100 // Numero maximo de Inputs
#define fran (rand()/(double)RAND_MAX+1)
int xi[N_N];
int J[N_N];

void construct_J(int nn);
void construct_xi(int nn);
void calcula_valor(int *uno, int *dos, int n);
int xi_J(int nn);
int teacher(int nn);
double Norm;

int main(int argc, char **argv)
{
    int i,k,n_teachers,n_test,res_teacher;
    int delta,out0;
    char name[256];
    int n_n;
    int aciertos,fallos;

    FILE *fevol;

    switch(argc)
    {
    case 4:
        sscanf(argv[1],"%d",&n_n);
        sscanf(argv[2],"%d",&n_teachers);
        sscanf(argv[3],"%d",&n_test);
        break;
    default:
        printf("Use: name bits_1_number n_teachers n_iter n_test\nDefault: ");
        n_n=20;
        n_teachers=500000000;
        n_test=1000;
    }

    printf("Using %d %d %d\n",n_n,n_teachers,n_test);
    strcpy(name,"Evol.dat");
    printf("Opening file for output: %s\n",name);
    fevol=fopen(name,"wt");

    construct_J(n_n);

    printf("Starting...\n");
    aciertos=fallos=0;
    for(i=0; i<n_teachers; i++)
    {
        construct_xi(n_n);
        res_teacher=teacher(n_n);

        out0=xi_J(n_n);
        delta=-1;
        if(out0==res_teacher)
            delta=1;

        if(res_teacher==0)
        {
            for(k=0;k<n_n;k++)
                J[k]+=(int)(0.5-fran)*4;

            delta=1;
        }
        else
        {
            if(delta== -1)
            {

```

```

        for(k=0;k<n_n;k++)
            J[k]+=res_teacher*xi[k];//Hebb Rule
    }

}

if(delta==1)
    aciertos++;
else
    fallos++;



if(i%100000==0 && i>0)
{
    printf("i=%d,Aciertos/Fallos=(%d/%d): J=%d,%d,aciertos,fallos);
    fprintf(fevol,"%d %d\n",aciertos,fallos);
    aciertos=fallos=0;
    for(k=0;k<n_n;k++)
        printf(" %d",J[k]);
    printf("\n");
}

}

fclose(fevol);

aciertos=0;
for(i=0;i<n_test;i++)
{
    construct_xi(n_n);
    res_teacher=teacher(n_n);
    if(res_teacher==0)
    {
        aciertos++;
        continue;
    }

    out0=xi_J(n_n);
    if(out0==res_teacher)
        aciertos++;
}
printf("Test Final: Intentos: %d, Aciertos=%d (%f %% )\n",n_test,aciertos,aciertos/(double)n_test);

return 0;
}

void construct_J(int nn)
{
    int i,JMax=1000;
    for(i=0; i<nn; i++)
    {
        J[i]=(0.5-fran)*JMax;
    }
}

void construct_xi(int nn)
{
    int i;
    for(i=0; i<nn; i++)
    {
        if(fran<0.5)
            xi[i]=0;
        else
            xi[i]=1;
    }
}

int xi_J(int nn)
{

```

```

int i;
double sum;
for(sum=i=0; i<nn; i++)
{
    sum+=J[i]*xi[i];
}
if(sum>0)
    return 1;
else
    return -1;
}

int teacher(int nn)
{
    int uno,dos;
    calcula_valor(&uno,&dos,nn);
    if(uno==dos)
        return 0;
    if(uno>dos)
        return 1;
    else
        return -1;
}

void calcula_valor(int *uno,int *dos,int n)
{
    int i,sum1,sum2,fact;
    fact=1;
    for(sum1=i=0; i<n/2; i++)
    {
        sum1+=(xi[i]*fact);
        fact*=2;
    }
    fact=1;
    for(sum2=i=0; i<n/2; i++)
    {
        sum2+=(xi[i+(n/2)]*fact);
        fact*=2;
    }
    *uno=sum1;
    *dos=sum2;
}

```


Capítulo 16

Ejemplo de Examen

El examen consta de 2 sesiones. Una teórica y una práctica, que se realizan habitualmente en sesión de Mañana (Teoría) y Tarde (Prácticas, en el Aula de Informática utilizada durante el curso para prácticas, aunque los alumnos que lo deseen pueden acudir con su propio ordenador).

16.1. Examen Teórico

Consta de 5 preguntas. La nota máxima del examen es 10, que luego se multiplica por 0.6 para obtener la nota final. En el apéndice pueden verse dos ejemplos de examen. Las preguntas deben responderse en el espacio enmarcado en negro entre preguntas. La parte de atrás de las hojas, en blanco, puede utilizarse para realizar operaciones, cálculos, pruebas, etc. y no será corregida. En el examen no se dispondrá de folios adicionales. No podrán utilizarse apuntes, libros, calculadoras, móviles u otros dispositivos electrónicos.

16.2. Examen Práctico

Con una semana (aproximadamente) de anterioridad a la realización del examen se anunciará debidamente un problema similar a los ejercicios desarrollados en prácticas, que los alumnos deberán programar previamente y presentar en el momento del examen práctico.

Se puntuará la resolución correcta aportada con una puntuación de hasta 1 punto. A continuación se indicará la realización de una modificación bien en el código o bien en el tratamiento de los datos. La correcta realización de esta modificación será puntuada con un máximo de 2 puntos.

Al iniciar el examen práctico los alumnos recibirán por escrito las cuestiones; para la primera parte (Comprobación) se formula una pregunta que se puede obtener ejecutando el programa sin modificarlo. Para la segunda parte (Modificación) se indica la modificación correspondiente y se pide un resultado concreto a obtener tras la modificación.

A continuación puede verse un ejemplo de convocatoria en la que se indica además de la fecha y hora, la prueba práctica concreta a desarrollar por el alumno. Se incluye también la hoja que se entrega a los alumnos durante dicha prueba práctica. Se incluyen también las respuestas requeridas (que, evidentemente, no recibe el alumno...). Puesto que hay varios grupos, las preguntas en cada grupo son diferentes, si bien de dificultad similar.

Como ayuda incluimos también un posible código para resolver el problema, y las modificaciones.

Se muestra también un examen teórico con las preguntas resueltas, para que el alumno vea el nivel que se requiere.

16.3. Ejemplo de Convocatoria de Examen

Convocatoria del Examen de Física Computacional

Día 18 de Junio de 2019

Teoría: 17:00 Horas, Aula Magna , Edificio A (Físicas)

Prácticas: Laboratorio de Informática, Edificio C (Geológicas), Distribuidos en 3 Turnos:

- 9:00 horas: De CCCCC a HHHHH
- 10:30 horas: De IIIII a RRRRR
- 12:00 horas: De SSSSS a ZZZZZ y de AAAAA a BBBBB

Trabajo a presentar:

Una partícula de masa m y carga q se mueve bajo la influencia de un campo magnético dado por

$$\vec{B}(x, y, z) = K \frac{(-y, x, 0)}{\sqrt{x^2 + y^2}}$$

cuando (x, y, z) esta en el interior de un toro centrado en el origen, y generado por una circunferencia en el plano (x, y) de radio R_T , sobre la que se añade una circunferencia de radio R_t . Es decir, un toro cuya superficie cumple la ecuación

$$(x, y, z) = ((R_T + R_t \sin \phi) \cos \theta, (R_T + R_t \sin \phi) \sin \theta, R_t \cos \phi)$$

En el interior del toro no actúa ninguna otra fuerza, pero en el exterior el Campo Magnético es nulo y la partícula sufre la fuerza de la gravedad g en el eje z .

Escribir un programa que simule el movimiento de la partícula. Usar el Algoritmo de Euler, con h suficientemente pequeño para que con todos los parámetros anteriores del orden de 1, el error en la energía tras 1 segundo de evolución sea menor del 1 por ciento. Estudiar las trayectorias que se producen tanto numéricamente como visualizándolas con `gnuplot`, tanto dentro como fuera del Toro.

16.4. Ejemplo de Examen Práctico para cada grupo

Comprobación Inicial. Turno 1

Apellidos y Nombre:

Considerar

$m=0.2$
 $q=10$
 $K=10$
 $g=-10$
 $R_T=1$
 $R_t=0.5$

Partir de la situación inicial: $(x, y, z) = (1, 0, 0.52)$ y la velocidad inicial $(v_x, v_y, v_z) = (0, 0, 0)$. Calcular la posición, velocidad y la Energía Cinética de la partícula al cabo de 1.2 segundos.

$$\vec{r} = (1.343969, 0.000000, 0.387456)$$

$$\vec{v} = (1.406565, 0.000000, -0.785021)$$

$$E = 0.259468$$

Modificación

Modificar el campo magnético para que sea de la forma en todo el espacio:

$$\vec{B}(x, y, z) = K \frac{(x, y, z)}{\sqrt{x^2 + y^2 + z^2}}$$

La fuerza de gravedad actúa ahora en todo el espacio. Con los parámetros pero con la posición inicial $(x, y, z) = (10, 0, 0)$ y la velocidad inicial $(v_x, v_y, v_z) = (10, 0, 0)$ anteriores calcular la posición, velocidad y Energía cinética al cabo de 1 segundo.

$$\vec{r} = (20.000031, -0.027735, -0.000138)$$

$$\vec{v} = (10.000045, -0.042726, 0.004577)$$

$$E = 10.000276$$

Comprobación Inicial. Turno 2

Apellidos y Nombre:

Considerar

```
m=0.2
q=10
K=10
g=-10
R_T=1
R_t=0.5
```

Partir de la situación inicial: $(x, y, z) = (1, 0, 0)$ y la velocidad inicial $(v_x, v_y, v_z) = (10, 10, 0)$. Calcular el tiempo en segundos que la partícula está dentro del Toro.

$$t = 2.302 \text{ segs}$$

Modificación.

Añadir una fuerza viscosa de ecuación

$$\vec{F}(\vec{v}) = -D\vec{v}$$

Considerar $D=2$, y con los parámetros anteriores calcular la posición, velocidad y Energía cinética al cabo de 0.15 segundos.

$$\vec{r} = (0.713372, 0.699033, 0.025383)$$

$$\vec{v} = (-0.097172, 3.032607, -0.867834)$$

$$E = 0.995928$$

Comprobación Inicial. Turno 3

Apellidos y Nombre:

Considerar

$m=0.2$
 $q=10$
 $K=10$
 $g=-10$
 $R_T=1$
 $R_t=0.5$

Partir de la situación inicial:

$x=1.2$
 $y=0$
 $z=0$
 $v_x=4$
 $v_y=4$
 $v_z=0$

Calcular el tiempo que le cuesta dar una vuelta completa alrededor del eje z .

$$t = 1.88 \text{ segs}$$

Modificación

Sustituir la región Toroidal donde no hay gravedad, por el interior de una Esfera de Radio 2. Con los parámetros anteriores pero cambiando la velocidad a $v = (12, 12, 0)$, calcular el momento que la partícula abandona la esfera.

$$t = 6.447 \text{ segs}$$

16.5. Ejemplo de Código para la prueba práctica

```

#include <math.h>
#include <stdio.h>
//#define DEBUG
double h;

void Evoluciona_dt(double *pos, double *vel);
void Calcula_Fuerza(double *R,double *V,double *F);
void Escribe_resultados(double time,double *r,double *v);
void Calcula_Fuerza_Modificacion(double *r,double *v,double *F);

FILE *fin,*fout;
double Q,M,B_C,R_Toro,r_Toro;
double G;
#define TRUE 1
main()
{
    //>
    double R[3],V[3];
    double tiempo;
    int i,j;
    int mesfr=200000,Niter=1000;

    fin=fopen("parameters.dat","rt");

    fscanf(fin,"%lf\n",&M);
    fscanf(fin,"%lf\n",&Q);
    fscanf(fin,"%lf\n",&R[0]);
    fscanf(fin,"%lf\n",&R[1]);
    fscanf(fin,"%lf\n",&R[2]);
    fscanf(fin,"%lf\n",&V[0]);
    fscanf(fin,"%lf\n",&V[1]);
    fscanf(fin,"%lf\n",&V[2]);
    fscanf(fin,"%lf\n",&B_C);
    fscanf(fin,"%lf\n",&R_Toro);
    fscanf(fin,"%lf\n",&r_Toro);
    fscanf(fin,"%lf\n",&G);

    fclose(fin);

    fout=fopen("Evol.dat","wt");

    h=0.00000001; //Paso temporal de la ecuacion diferencial discreta
    tiempo=0;

    printf("#      t          T          V          E_t \n");

    for(i=0;i<Niter;i++)
    {
        for(j=0;j<mesfr;j++)
            Evoluciona_dt(R,V);
        tiempo+=mesfr*h;
        Escribe_resultados(tiempo,R,V);
    }
    fclose(fout);

}

void Evoluciona_dt(double *r, double *v)
{
    int i;
    double F[3];

```

```

double D=0.0;

//Calcula_Fuerza_Modificacion(r,v,F);

Calcula_Fuerza(r,v,F);
#ifndef DEBUG
printf("En evoluciona, h=%f\n",h);
printf("Evol: R=%f %f %f, F=%f %f %f, v=%f %f %f\n",r[0],r[1],r[2],F[0],F[1],F[2],v[0],v[1],v[2]);
#endif

for(i=0;i<3;i++)
{
    r[i]+=h*v[i];
    v[i]=v[i]+F[i]*h/M-D*v[i]*h/M;
}

void Calcula_Fuerza(double *r,double *v,double *F)
{
    double B[3],Dist,Norm,R_G[3],A[3];

    Norm=R_Toro/sqrt(r[0]*r[0]+r[1]*r[1]+0.0000001);
    R_G[0]=r[0]*Norm;
    R_G[1]=r[1]*Norm;
    R_G[2]=0;

    A[0]=r[0]-R_G[0];
    A[1]=r[1]-R_G[1];
    A[2]=r[2]-R_G[2];

    Norm=sqrt(A[0]*A[0]+A[1]*A[1]+A[2]*A[2]);

    // Esfera
    /*
    Norm=sqrt(r[0]*r[0]+r[1]*r[1]+r[2]*r[2]);
    */

    if(Norm < r_Toro)
    {
        Dist=sqrt(r[0]*r[0]+r[1]*r[1]);
        B[0]=-B_C*r[1]/Dist;
        B[1]=B_C*r[0]/Dist;
        B[2]=0;

        F[0]=Q*(v[1]*B[2]-v[2]*B[1]);
        F[1]=Q*(v[2]*B[0]-v[0]*B[2]);
        F[2]=Q*(v[0]*B[1]-v[1]*B[0]);
    }
    else
    {
        F[0]=F[1]=0;
        F[2]=-M*G;
    }

#ifdef DEBUG
    printf("F=%f %f %f\n",F[0],F[1],F[2]);
#endif
}

```

```

#endif

}

void Calcula_Fuerza_Modificacion(double *r,double *v,double *F)
{
    double B[3],Dist,Norm,R_G[3],A[3];

    Dist=sqrt(r[0]*r[0]+r[1]*r[1]+r[2]*r[2]);
    B[0]=B_C*r[0]/Dist;
    B[1]=B_C*r[1]/Dist;
    B[2]=B_C*r[2]/Dist;

    F[0]=Q*(v[1]*B[2]-v[2]*B[1]);
    F[1]=Q*(v[2]*B[0]-v[0]*B[2]);
    F[2]=Q*(v[0]*B[1]-v[1]*B[0]);
    F[2]=-M*G;

#ifdef DEBUG
    printf("F=%f %f %f\n",F[0],F[1],F[2]);
#endif

}

void Escribe_resultados(double time,double *r,double *v)
{
    double Energia;
    Energia=0.5*M*(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);

#ifdef DEBUG
    printf("r=(-10.31f,-10.31f,-10.31f),v=(-10.31f,-10.31f,-10.31f),E=-10.31f\n",
          r[0],r[1],r[2],v[0],v[1],v[2],Energia);
#endif

    fprintf(fout,"%f %f %f %f %f %f %f %f\n",time,r[0],r[1],r[2],v[0],v[1],v[2],Energia);

    //getchar();
}

```

16.6. Ejemplo de Examen teórico

Damos el enunciado con el código de la solución. Además del código se valora los comentarios del alumno sobre los puntos más delicados de la solución.

Física Computacional (Teoría Online)
22 de Junio de 2020

Nombre:**Apellidos:**

Pregunta 1: Sea un conjunto de N números enteros $\{x_i\}, i=0, \dots, N-1, x_i \geq 0 \forall i$. Escribir completamente una función que recibe N y el conjunto de los $\{x_i\}$ y devuelve un índice $i \in [0, N-1]$, de manera que la probabilidad de la X_i correspondiente cumpla $p(x_i) \propto x_i$. La función debe devolver también un flag indicando si se devuelve un número correcto (flag=0) o no es posible devolverlo (flag=1) debido a que algún X_i sea negativo. También se debe soportar correctamente el caso en que todos los sean X_i cero.

```

int Q1(int n,int *dat,int *flag)
{
    float omega;
    int i,sum,tot;
    *flag=0;
    for(sum=i=0;i<n;i++)
    {
        if(dat[i]<0){*flag=1; return 0;}
        sum+=dat[i];
    }

    if(sum<=0){return (n*fran);}//Aquí, todos los datos son 0. Devolvemos un índice al azar.

    omega=fran*sum;
    i=0;
    tot=dat[0];
    while(omega>tot && i<n)
    {
        i++;
        tot+=dat[i];
    }
    return i;
}

```

Pregunta 2: Sea la curva $y=ax^2, x \in [-10, 10]$. Escribir completamente una función que reciba a y p y devuelva los vectores unitarios tangente y normal a la curva en el punto $x=p$. Elegir los signos de modo que la componente X de ambos vectores no sea negativa.

```
void Vector_t_n(double a,double x,double *t, double *n)
{
    double Norm;
    Norm=1/sqrt(1+4*a*a*x*x);

    t[0]=Norm;
    t[1]=Norm*2*a*x;

    if((a*x)<0)
    {
        n[0]=-Norm*2*a*x;
        n[1]=Norm;
    }
    else
    {
        n[0]=Norm*2*a*x;
        n[1]=-Norm;
    }
}
```

Pregunta 3: Un archivo tipo texto contiene un grafo dirigido y no pesado en el formato “nodo_inicial nodo_final” por línea, por ejemplo “8 5” (Ambos int Indicando que hay una link que va del nodo 8(out) al nodo 5(in)). El índice de los nodos está en el intervalo [0, N_Max]. Escribir completamente una función que recibe el nombre del archivo, lea todas sus líneas y devuelva: Número de Links, Grado_In y Grado_Out de cada nodo. Debe devolver además un flag con valor 0 si todo ha ido bien y con valor 1, si el archivo no existe o está vacío.

```
void Lee_grafo(char *name,int *n_links,int *g_in,int *g_out,int *f)
{
FILE *Gr;
int i,ii,io;

//Abre el archivo
if((Gr=fopen(name,"rt"))==NULL)
{
    *f=1;
    return;
}
for(i=0;i<N_Max;i++)
{
    g_out[i]=g_in[i]=0;
}
i=0;
while( fscanf(Gr,"%d %d\n",&io,&ii)!=EOF )
{
    g_out[io]++;
    g_in[ii]++;
    i++;
}

*n_links=i;
if(*n_links==0)
{
    *f=1;
    return;
}
*f=0;
}
```

Pregunta 4: Sea una polea de Radio R sin masa y sin rozamiento de la que cuelgan dos masas m_1, m_2 bajo los efectos de la gravedad g . La cuerda es sin masa y de longitud L . Escribir completamente una función que recibe R, L, g, m_1, m_2 , la posición y velocidad de la masa 1 (y_1, v_1), un tiempo t , y devuelve la posición y velocidad de ambas masas (y_1, y_2, v_1, v_2) al cabo de ese tiempo. Dado que el problema tiene solución exacta (que el alumno puede calcular fácilmente), debe usarse dicha solución y no un algoritmo de integración numérica, como Euler o Verlet. El eje y es tomado positivo hacia abajo y tiene su origen en el centro de la polea. La función debe devolver también un flag que vale 1 si una de las masas choca contra la polea (la coordenada y alcanza la coordenada y del centro de la polea) y cero en caso contrario. En el caso del choque, no debe devolver posiciones ni velocidades.

```
void Polea(double R,double Lon,double g,double m1,double m2,
           double *y1,double *y2,double *v1,double *v2, double ti,int
*flag_choque)
{
    double a;
    double Pi;

    Pi=2*asin(1.0);

    //Solucion exacta a=(m_1 -m_2) / (m_1+m_2) * g

    *flag_choque=0;
    a=g*(m1-m2)/(m1+m2);

    *y1=*y1+(*v1)*ti+0.5*a*ti*ti;
    *v1=*v1+a*ti;

    *v2=-(*v1);
    *y2=Lon -(*y1)-Pi*R;

    if((*y1)<0 || ((*y2)<0))
        *flag_choque=1;
}
```

Pregunta 5 : Sea el modelo de Ising en una red cúbica en $d=3$ con condiciones de contorno periódicas y tamaño $L_x \times L_y \times L_z$. Considerar la red numerada con un sólo indice $n \in [0, V-1]$, con $V = L_x \times L_y \times L_z$ y n corriendo primero en x , luego en y y luego en z . Escribir una función que reciba L_x, L_y, L_z y devuelva seis vectores xp, xm, yp, ym, zp, zm de manera que sean los offsets para pasar de n al punto siguiente en la dirección $+x, -x, +y, -y, +z, -z$ respectivamente.

```
void Calcula_Offsets(int Lx,int Ly,int Lz,int *xp,int *xm,int *yp,int
*ymp,int *zp,int *zm)
{
    int i;

    for(i=0;i<Lx;i++)
    {
        xp[i]=1;
        xm[i]=-1;
    }
    xp[Lx-1]==-(Lx-1);
    xm[0]=(Lx-1);

    for(i=0;i<Ly;i++)
    {
        yp[i]=Lx;
        ym[i]=-Lx;
    }
    yp[Ly-1]==-Lx*(Ly-1);
    ym[0]=Lx*(Ly-1);

    for(i=0;i<Lz;i++)
    {
        zp[i]=Lx*Ly;
        zm[i]=-Lx*Ly;
    }
    zp[Lz-1]==-Lx*Ly*(Lz-1);
    zm[0]=Lx*Ly*(Lz-1);
}
```

Física Computacional (Teoría)
18 de Junio de 2019

Nombre:

Apellidos:

Pregunta 1: Escribir completamente una función que recibe un entero en el intervalo [0,9999] y devuelve la suma de sus cuatro dígitos; por ejemplo si recibe 3841 devuelve 16.

```
int suma(int a)
{
    int sum,d0,d1,d2,d3,prev;

    d0=a/1000;
    prev=a-1000*d0;
    d1=prev/100;
    prev=prev-100*d1;
    d2=prev/10;
    prev=prev-10*d2;
    d3=prev;
    sum=d0+d1+d2+d3;
    return sum;
}
```

Pregunta 2: Escribir completamente una función que devuelve un número aleatorio en el intervalo [1,3], distribuido proporcionalmente a $f(x)=a^x, a>1$. Usar el método de la función de distribución. Nota: $a^x=e^{x\ln(a)}$

```
double alea(double a)
{
double x,omega;
//p(x)=C*a^x x \in [1,3]
//C=log(a)/(a^3-a) ; a>1
//P(x)=(log(a)/(a^3-a))*a^x
//Tau(x)=(a^x-a)/(a^3-a)
//omega=(a^x-a)/(a^3-a) ->
// x=log(a+(a^3-a)*omega)/log(a)

omega=fran;
x=log(a+(a*a*a-a)*omega)/log(a);

return x;
}
```

Pregunta 3: Sobre un plano situamos dos cargas Q y $-Q$, la primera en el punto $(0,0)$ y la segunda en $(0,a)$. Escribir completamente una función que recibe un punto del plano (x,y) y devuelve el campo eléctrico sobre dicho punto ($K = 9 \times 10^9$).

$$F(x,y) = K \frac{Qq}{r_1^2} \frac{(x,y)}{r_1} - K \frac{Qq}{r_2^2} \frac{(x,y-a)}{r_2}$$

```
void Campo(double K, double Q, double a, double x, double y, double *E)
{
    double R1,R2, R13,R23;
    R1=x*x+y*y;
    R13=R1*sqrt(R1);

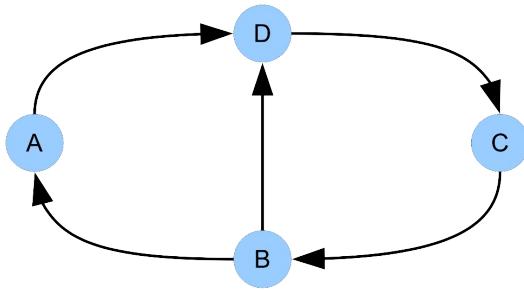
    R2=x*x+(y-a)*(y-a);
    R23=R2*sqrt(R2);
    printf("R1=%lf, R2=%lf\n",R1,R2);
    E[0]=K*(Q*x/R13)-K*(Q*x/R23);
    E[1]=K*(Q*y/R13)-K*(Q*(y-a)/R23);

}
```

Pregunta 4: Sea $p(x,y)$ una función densidad de probabilidad correctamente normalizada y acotada en un rectángulo entre los puntos $(x,y)=[1,2]$ y $(x,y)=[3,5]$. Escribir un segmento de código que calcule un valor de (x,y) distribuidos según $p(x,y)$ usando el método de la altura.

```
Genera x entre 1 y 3; Genera y entre 2 y 5;
Genera omega entre 0 y el máximo
Si omega es menor de p(x,y) elige el valor de (x,y). Si no, repite el proceso.
```

Pregunta 5 : Sea una red dirigida de 4 nodos con las conexiones (de nodo saliente a nodo entrante): (A,D), (B,A), (B,D) (D,C), (C,B). Calcular el page rank de cada nodo, fijando el page rank máximo a 1.



$$A = B/2$$

$$D = A + (B/2)$$

$$C = D$$

$$B = C$$

Tomando $D=1$, tenemos $D=1, C=1, B=1, A=1/2$