Métodos computacionales para la física estadística

Dr. David P. Sanders
Facultad de Ciencias, UNAM
dps@fciencias.unam.mx

Notas del curso del Posgrado en Ciencias Físicas

March 21, 2010

Capítulo 1

La física estadística computacional

La física estadística tiene como meta el establecer las propiedades a nivel *macroscópico* de un sistema compuesto por muchas partículas a nivel *microscópica*. Como se ven en los cursos básicos de esta materia, esta meta se puede lograr en ciertos casos de manera exacta, como son los gases ideales y los paramagnetos.

Pero rara vez se tratan en los cursos básicos sistemas que consisten en partículas *en interacción*. La razón por este hecho es sencillo: es, en general, *muy difícil* resolver problemas de este tipo. Sin embargo, esta clase de sistemas es exactamente donde pasan los fenómenos de interés físico, por ejemplo, las transiciones de fase, las propiedades emergentes, y las propiedades de transporte.

1.1. Métodos de Monte Carlo

Este curso se trata de otro enfoque para poder obtener información acerca de sistemas de partículas en interacción –un enfoque *computacional*.

Aún si las cantidades macroscópicas que nos interesan se pueden ver como cantidades deterministas, del punto de vista de la física estadística, están dadas por *promedios* de distintos tipos sobre cantidades microscópicas que fluctúan con cierta distribución de probabilidad.

Por lo tanto, estudiaremos principalmente métodos tipo *Monte Carlo* –así llamados por emplear muchos números aleatorios para simular al sistema[1].

1.2. Elección de idioma de programación

Los métodos tipo Monte Carlo suelen requerir la repetición de ciertas operaciones bastante sencillas millones de veces. Por lo tanto, es necesario contar con un lenguaje de programación sumamente eficiente para llevar a cabo simulaciones de este tipo.

Los dos candidatos más apropiados son Fortran 90 y C++. En este curso emplearemos C++, que cuenta con ciertos métodos más modernos de programación. Sin embargo, Fortran 90 es una buena opción también para este tipo de aplicaciones.

Unas razones por las cuales utilizamos C++ son los siguientes: es poderoso, flexible, rápido, y capaz de imponer estructura en proyectos grandes. Se pueden utilizar distintos estilos modernos de programación que permiten flexibilidad y eficiencia al mismo tiempo. Además, se está ocupando cada vez más para los proyectos de cómputo científico en todo el mundo.

Algunas desvantajas de C++ son: su sintaxis puede ser algo complicada; hay que recompilar después de cualquier cambio del programa; hay mucho que aprender (porque contiene muchas posibilidades); y los mensajes de error pueden ser difíciles de entender.

[1] Monte Carlo es una ciudad en el principado de Monaco, en el sur de Francia, famosa por el gran número de casinos que se encuentran ahí.

1.3. Entorno de análisis de datos

Una vez que se hayan generado datos, es necesario analizar los datos generados. Una buena opción moderna para eso es el idioma de programación interpretado Python, en conjunto con distintos paquetes y librerías de software libre que están disponibles para interactuar con Python. También ocuparemos el programa `gnuplot` para gráficas.

Todo el trabajo se llevará a cabo en un sistema operativo tipo Unix, que podría ser Linux ó BSD (la base de MacOS, por ejemplo). Los entornos tipo Unix proveen muchas herramientas de gran utilidad para el cómputo científico que no están disponibles en otros sistemas operativos. Todas las herramientas utilizadas en el curso serán de *software libre*.

1.4. Metas del curso

Las metas del curso son principalmente las siguientes dos:

- Aprender a utilizar un idioma de programación potente y moderno; y
- Desarrollar técnicas computacionales para estudiar problemas que surgen en la física estadística.

Parte I

Caminatas aleatorias y la sintaxis básica de C++

Capítulo 2

Sintaxis básica de C++

C++ es un lenguaje de programación *compilado* –es decir, para correr un programa, es necesario pasar a través de una etapa de compilación para convertirlo en código que la máquina puede utilizar. Aún si eso puede ser latoso, hace más eficiente (rápido) el procedimiento final de correr el programa.

C++ se desarrolló a partir del lenguaje C, por lo cual retiene la sintaxis básica del mismo. Sin embargo, C++ tiene conceptos de más alto nivel que C, los cuales adoptaremos desde el principio.

2.1. Primeros pasos

El primer programa clásico es el "¡Hola, mundo!", que en C++ se podría escribir como sigue:

```
#include <iostream>

int main() {
  std::cout << "Hola, mundo!" << std::endl;
  return 0;     // Regresar un valor al mundo externo
}
```

Para correrlo, guardamos el código en el archivo `primero.cpp` y lo compilamos y corremos con[1]:

```
$ g++ primero.cpp -o primero
$ ./primero
Hola, mundo!
```

Aquí, `g++` es el nombre del compilador de C++ del proyecto GNU, y `-o` indica que la salida ("output") va a ser un ejecutable que se llama `primero`. El comando `./primero` ejecuta este programa –el punto indica que se encuentra en el directorio actual.

En el código, el **`#include`** es una instrucción al compilador de *incluir* un archivo de código fuente predefinido, que en este caso provee los comandos `cout` y `endl`. El primero imprime lo que sigue después del `<<`, y el segundo da una nueva línea. Los dos están adentro del *espacio de nombres* `std` ("estándar"). Como se vuelve molesto teclear `std::` todo el tiempo, es conveniente utilizar el comando **`using namespace`** `std`, que da la siguiente versión mejorada del código:

```
#include <iostream>
using namespace std;

int main() {
```

[1] El signo `$` denota el "prompt" en la terminal donde se teclean los comandos.

```
  cout << "Hola, mundo!" << endl;
  return 0;
}
```

La línea `int main()` declara una *función* (subrutina) llamada `main`. Cada programa de C++ debe contener una función con este nombre, donde la ejecución del programa empezará. Las parentesis contienen los argumentos de la función (información que se manda a la función), e `int` indica que regresa un entero, que indicará si el programa terminó con exito (valor 0), como indica `return 0`, o no (valor no-cero).

Las llaves `{` y `}` encierran un *bloque* de código, que se trata como una unidad. En este caso, el bloque es el *cuerpo* de la función.

Nótese que cada instrucción, o comando, en C++, tiene que terminar con un *punto y coma*, `;` –el olvidar algun `;` llevará a un sinfin de errores al momento de compilar el programa!

Los *comentarios* se ponen después de `//`; el resto de la línea es un comentario.

El `'\n'` es un carácter especial, que también quiere decir que imprima una nueva línea; otro parecido es un "tabulador" (un espacio especial para alinear), dado por `'\t'`. (Los caracteres individuales se escriben entre apóstrofes, `'`, mientras que las cadenas de varios caracteres se escriben entre comillas, `"`.)

En el resto de este capítulo, normalmente daremos fragmentos de código. Para correrlos, es necesario incluirlos en un programa completo de este tipo.

2.2. Variables

Para llevar a cabo cálculos, necesitamos poder guardar datos en *variables*. Éstas se se tienen que *declarar*, es decir, especificar de qué tipo son, antes de utilizarse:

```
int num_particulas;        // declarar una variable a de tipo entero (integer)
double temperatura;    // declarar una variable real de doble precision
```

Una declaración reserva suficiente espacio en la memoria para guardar el valor de la variable, y le da información al compilador que puede utilizar para detectar errores, si intentamos manipular una variable de una manera que no corresponde a su tipo.

Principalmente usaremos `int` y `double`. Normalmente ya no se utilizan los números reales de precisión sencilla (`float`)[2], ya que las máquinas modernas están optimizadas para usar números de doble precisión.

Se asignan valores a las variables con =, que también se puede hacer al momento de declarar las variables:

```
num_particulas = 10;
temperatura = 1.5;

int a = 3;
double b = -1.3e5;     // notacion cientifica
```

Los valores de las variables, como la mayoría de los objetos, se pueden imprimir utilizando `cout`, como sigue. Lo que va dentro de comillas (`"`) se imprime exactamente tal cual. Los `<<` se pueden encadenar el número de veces que sea necesario (siempre respetando que el código sea legible):

```
cout << "a = " << a << "; b = " << b << endl;
```

[2]Eso ya ha cambiado con la actual programación de los GPUs, que están diseñados para funcionar en precisión sencilla.

Las declaraciones de variables se pueden hacer en *cualquier lugar* del programa[3]. El mejor estilo es el de declarar una variable lo más cercano a su primer uso.

2.3. Flujo de entrada

Para permitir al usuario que introduzca datos para guardarse en variables, se puede utilizar `cin`, de manera opuesta a `cout`:

```
int num_particulas;
cout << "Cuantas particulas? ";
cin >> num_particulas;

double temperatura, presion;
cout << "Valor de temperatura y presion? ";
cin >> temperatura >> presion;
cout << "Usando temperatura = " << temperatura
    <<  "; presion = " << presion << endl;
```

(Puede causar problemas el utilizar acentos dentro del código, por lo cual los evitamos.) Nótese que en este caso, si la entrada que damos es `3, 4`, entonces el valor de presión está equivocada. En un programa real, habría que checar este tipo de errores.

2.4. Condicionales

Muy a menudo, es necesario comprobar si alguna condición se satisfaga o no. El caso más sencillo se implementa con **`if`**:

```
int a = 3;
if (a > 0)  {      // NB: Las parentesis son obligatorias
  cout << "a es positivo" << endl;
}
```

Nótese que se utiliza un bloque, entre `{` y `}`, para delimitar el código que se ejecutará si la condición se satisface. Se puede agregar una alternativa que se ejecuta si la condición no se satisface:

```
else {
  cout << "a es negativo " << endl;
  cout << "Su valor absoluto es " << -a << endl;
}
```

Nótese que aquí hay dos comandos que se ejecutan como parte del bloque.

Las condiciones que se pueden probar incluyen:

igualdad	==
desigualdad	!=
AND	&&
OR	||
NOT	!
mayor que	>
mayor o igual que	>=
menor que	<
menor o igual que	<=

[3] A diferencia de C.

Para probar más de una condición simultáneamente, se utilizan `&&` (y) y `||` (ó). Las condiciones que se juntan se ponen dentro de otra paréntesis:

```
int a = 3, b = 5;

if ( (a < 0) && (b < 0) ) {
  cout << "a y b son negativos los dos" << endl;
}

if ( (a < 0) || (b < 0) ) {
  cout << "Al menos uno de a y b es negativo" << endl;
}
```

2.5. Bucles

El poder de las computadoras para llevar a cabo cálculos proviene de su capacidad de repetir las mismas operaciones una y otra vez, a través de los *bucles*.

2.5.1. `while`

El bucle más sencillo, y más básico, es el **`while`**, que se ejecuta *mientras* una condición se satisface. Normalmente se utiliza un **`while`** cuando no se sabe de antemano cuantas veces hay que iterar. Las condiciones se emplean de la misma manera como para un **`if`**.

Por ejemplo, una manera (¡no necesariamente la mejor!) de contar la mayor potencia de 2 menor que un número dado sería:

```
int valor = 63201;
int potencia = 0;

while (valor > 1) {
  cout << "valor = " << valor << endl;
  valor /= 2;      // equivalente a:  valor = valor / 2
  potencia ++;
}

cout << "La mayor potencia de 2 es " << potencia+1 << endl;
```

Nótese que la aritmética con variables se lleva a cabo de manera intuitiva. Es un procedimiento muy común el actualizar a una variable, reemplazando su valor actual con un valor nuevo que se calcula mediante el valor actual.

Para este fin, C++ provee una serie de operadores muy prácticos, que combinan un operador aritmético con un `=`. Por ejemplo, `a+=b` es una abreviación de `a = a+b`. Estas combinaciones tienen la ventaja de enfatizar el hecho de que se está actualizando el valor de una variable a partir del valor actual.

Además, hay operadores especiales `++` y `--` para incrementar y decrementar, respectivamente, el valor de una variable *entera* solamente.

2.5.2. `for`

Normalmente se utiliza un **`for`** cuando se sabe de antemano cuantas veces se quiere iterar. La sintaxis es:

```
for (inicializacion;  condicion;  actualizacion) {
  ...
```

```
}
```

donde

- inicialización: lo que hace al principio del bucle;
- condición: lo que se prueba al final de cada iteración;
- actualización: el comando que se ejecuta al final de cada iteración.

Por ejemplo, para sumar los números de 1 a 10:

```
int n = 10;
int total = 0;

for (int i = 0; i < n; i ++) {
  cout << i << "\t" << total << endl;    // '\t' es un caracter de tab
  total += i;
}
cout << "El total es " << total << endl;
```

Nótese que se declaró el entero `i` en el *paso de inicializacion* del bucle. Por lo tanto, esta variable existe *solamente dentro del bucle*. Así se pueden reutilizar nombres de variables comunes, sin tener que inventar nuevos nombres.

2.5.3. `do-while`

Un bucle **`do-while`** es el opuesto de un **`while`**: la condición se evalúa al final del bucle, así que el código se ejecuta al menos una vez. Se utiliza tal vez menos que los otros dos tipos de bucle.

```
int i = 0;
do {
  cout << i << endl;
  i++;
} while (i < 10);
```

2.6. Funciones matemáticas

Algunas funciones matemáticas ya están implementadas. Para utilizarlas, se requiere incluir la librería `cmath`:

```
#include <cmath>

cout << "El seno de 3.5 es " << sin(3.5);
cout << "La raiz cuadrada de 700 es " << sqrt(700);
```

La librería `cmath` incluye, entre otros, `exp`, `sin`, `cos`, `tan`, `asin` (seno inverso o arcseno), `acos`, `atan` y, `log`. También hay otras funciones evidentes, por ejemplo `log1p(x)` regresa $\log(1+x)$. Para tomar potencias existe `pow(x,y)`, que devuelve el valor de x^y.

Todas las funciones toman como argumentos números tipo **`double`**.

Para funciones más complicadas, por ejemplo funciones especiales, hay librerías disponibles como software libre. Una muy buena es el Gnu Scientific Library (GSL). Sin embargo, hay que reconocer que el interfaz de esta librería es de estilo C, no C++ propiamente.

Ejercicio: Implementa el *algoritmo Babilónico* para encontrar la raíz cuadrada de un numero y, como sigue. Si a_n es un estimado de la raíz cuadrada, entonces un mejor estimado es

$$a_{n+1} := \frac{1}{2}\left(a_n + \frac{1}{a_n}\right). \tag{2.1}$$

Pon esto adentro de una *iteración*, empezando desde cualquier condición inicial $a_0 \neq 0$. Compara el resultado obtenido a cada paso con el resultado exacto. Utiliza una gráfica para investigar la velocidad de convergencia del algoritmo.

2.7. Resguardo de datos

Ahora que hemos producido algunos datos, debemos procesarlos para entenderlos. La primera etapa es simplemente graficarlos para tener una idea intuitiva de la forma de los mismos.

Para poder hacerlo, hay dos maneras posibles: (i) mandar los datos a graficar directamente desde el mismo programa de C++; y (ii) guardar los datos en un archivo, que luego se graficará con otra herramienta. Posteriormente abarcaremos el inciso (i); por el momento, ocuparemos la herramienta `gnuplot` para graficar los datos.

Una opción para guardar los datos en un archivo es la de abrir un archivo dentro del programa y escribir ahí. Esta opción la veremos más adelante; por el momento, aprovecharemos un método que nos proporciona el *shell*, o *terminal* en Unix: podemos *redirigir* la salida de cualquier programa a un archivo.

Por ejemplo, para mandar la salida del programa `babilonico` a un archivo llamado `sqrt.dat`, podemos poner directamente en la línea de comandos el comando

```
$ babilonico > sqrt.dat
```

Ya no vemos la salida del programa en la pantalla, sino se guarda la misma en el archivo. En este caso, sobreescribirá el contenido del archivo. Para agregar los nuevos datos al contenido ya existente, hacemos

```
$ babilonico >> sqrt.dat
```

2.8. Ver contenido de un archivo

Para verificar que los datos se han guardado de manera adecuada, podemos copiar el contenido del archivo a la terminal con el comando

```
$ cat sqrt.dat
```

El comando `cat` tiene como fin el de "concatenar" su entrada a su salida. En este caso, la entrada es el archivo `sqrt.dat`, y la salida es la terminal.

Otro uso de `cat` es para crear un archivo, por ejemplo

```
$ cat > datos1.dat
```

creará un archivo `datos1.dat` con el contenido que se teclea en la terminal –aquí, la entrada es la terminal y la salida se ha redireccionado al archivo.

Finalmente, `cat` puede literalmente concatenar archivos, por ejemplo

```
$ cat datos1.dat datos2.dat >> datos3.dat
```

2.9. Graficamiento de datos con gnuplot

Ahora contamos con los datos en un archivo de texto sqrt.dat. Utilizaremos gnuplot para graficarlos.

gnuplot es un programa con el que podemos producir gráficas en 2 y 3 dimensiones, con calidad de publicación (¡si trabajamos un poco!) Posee una interfaz de línea de comandos, y también puede ser utilizado desde otro programa o al escribir "scripts" (programas con listas de comandos de gnuplot). Cuenta con gráficas interactivas, además de una gran variedad de formatos para exportar las gráficas producidas, incluyendo a Postscript y png. Hay distintas demostraciones de sus capabilidades en la página http://www.gnuplot.info/demo/.

En la línea de comandos ejecutamos

```
$ gnuplot
```

y así entramos en el entorno del programa.

2.9.1. Gráficas de funciones

Las gráficas más sencillas son las de funciones predefinidas, como

```
$ gnuplot
gnuplot> plot sin(x)
gnuplot> plot sin(x), cos(x)  # graficar dos funciones juntas
```

Para dibujar estas funciones, las evalua en distintos puntos

```
gnuplot> plot sin(x) with points    # abreviacion: w p
gnuplot> set samp 500     #aumentar el numero de puntos
gnuplot> replot     # o teclear 'e' en la ventna de la grafica
```

Se pueden definir variables y funciones

```
gnuplot> a=3
gnuplot> plot sin(a*x)
gnuplot> print a
3
```

2.9.2. Dibujar datos desde archivos

Para dibujar el contenido del archivo sqrt.dat, ponemos

```
gnuplot> plot "sqrt.dat"    # usa solo la primera (x) y segunda columna (y)
```

Por defecto, dibuja utilizando puntos.

```
gnuplot> plot "sqrt.dat"  with lines    # abreviacion: w l
gnuplot> plot "sqrt.dat", "" using 2:3    # abreviacion: u
```

Aquí, dibujamos dos veces el mismo archivo (al poner un nombre de archivo en vacío), y ocupamos las columnas 2 y 3 para la segunda gráfica. También se pueden mezclar datos y funciones:

```
gnuplot> plot "sqrt.dat"  w points, sqrt(x) w l   # abreviacion: w p
```

2.9.3. Exportación de las gráficas

`gnuplot` cuenta con un sistema de ayuda. Por ejemplo, podemos conocer las distintas "terminales" (formatos de salida) disponibles con

```
gnuplot> help set term
```

Para crear un archivo de Postscript (formato EPS) con la grafica, que cuente con etiquetas en los ejes, tenemos:

```
gnuplot> set xlabel "x"
gnuplot> set ylabel "sin(3x)"
gnuplot> set term post eps
gnuplot> set out "sin.eps"
gnuplot> replot
gnuplot> set out
gnuplot> !okular sin.eps    # manda un comando a la terminal
```

Un ejemplo interesante:

```
gnuplot> f(x) = sin(a*x)
gnuplot> plot a=1, f(x) title 'sin x', a=2, f(x) title 'sin 2x',
              a=3, f(x) t 'sin 3x'
```

Capítulo 3

Números aleatorios, funciones, y caminatas aleatorias

La meta de este capítulo es la de llegar a hacer una *simulación* de uno de los sistemas físicos más sencillos: una *caminata aleatoria*.

Por una simulación, nos referimos a un modelo computacional que reproduce las características principales de un sistema físico adentro de la computadora.

3.1. Caminatas aleatorias

Empecemos con el estudio de las caminatas aleatorias. En este tipo de modelo, una partícula brinca en distintas direcciones al azar. Este modelo es una aproximación, más o menos buena, del comportamiento de muchos sistemas físicos, por ejemplo el movimiento de una partícula grande "Browniana" en agua. El comportamiento "aleatorio" provee de los miles de colisiones que sufre la partícula con las moléculas a su alrededor, cuyo efecto total modelamos como impulsos aleatorios. Además, es una primera aproximación a una descripción del tipo de movimiento que ejecuta un organismo vivo, por ejemplo una célula, una hormiga, o un animal, viajando en su entorno.

Para simplicidad, consideremos inicialmente el caso donde el caminante brinca en momentos de tiempo *discretos* $n = 1, 2, \ldots$, correspondiendo a momentos de tiempo δt, $2\delta t$, etc., donde δt es un incremento de tiempo pequeño.

3.2. Caminata aleatoria discreta en 1 dimensión

Empecemos con una caminata aleatoria que vive en una una *red* en 1 dimensión, y una partícula –un caminante– que empieza en el orígen 0 al tiempo $n = 0$. En cada paso el caminante ejecuta una de tres acciones con distintas *probabilidades*

- brinca a la derecha con probabilidad p;
- brinca a la izquierda con probabilidad q; ó
- se queda en el mismo lugar con probabilidad $r := 1 - (p + q)$.

En el caso más sencillo, tiene que brincar, con $r = 0$. El caso más clásico es con probabilidades iguales de brincar en cualquier de las dos direcciones, es decir, $p = q = \frac{1}{2}$.

Este problema es uno de los problemas más clásicos en la física estadística, y muchos de sus propiedades se conocen de manera analítica.

3.3. Números aleatorios

Para abarcar la *simulación* de una caminata aleatoria, necesitamos poder hacer elecciones entre distintas opciones con cierta *probabilidad*. Para hacerlo, utilizaremos *números aleatorios*.

¿Cómo se pueden generar números aleatorios en una computadora? La respuesta es que ¡no se puede!, ya que una computadora puede llevar a cabo solamente operaciones deterministas. Sin embargo, podemos generar secuencias de números que se comportan "como si fueran" aleatorios; éstos se llaman números *pseudo-aleatorios*. Para hacerlo, ocuparemos secuencias generadas por iteraciones deterministas, con ciertas propiedades escogidas para dar la apariencia de aleatoriedad.

3.4. Generación de número aleatorios

Para el uso en una simulación seria, se emplearía una librería, por ejemplo el Gnu Scientific Library. (En particular, hoy en día se usa mucho el algoritmo llamado Mersenne Twister.) Sin embargo, para un uso básico, es útil saber cómo se pueden generar los números aleatorios en una computadora.

La manera más frecuente de generar números pseudo-aleatorios es a partir de una iteración $x_{n+1} = f(x_n)$, donde x_n es *entero*. Dependiendo de la función f se pueden obtener resultados que parezcan aleatorios.

Una de las secuencias que se solía usar era $f(x_n) = ax_n + b \pmod m$. Si se escogen bien la a, b y m, entonces esta iteración sencilla puede dar buenos resultados. Este método recae en el hecho de que los enteros en C++ son de un tamaño de 32 bits, entonces cualquier operación que da un resultado mayor que este tamaño se truncará.

Un ejemplo está dado por

$$a = 1664525; \quad b = 1013904223; \quad m = 2^{32}. \tag{3.1}$$

El valor de m es uno más que el máximo valor que se puede guardar en un entero de 32 bits, por lo cual se puede ejecutar de manera automática el módulo, al declarar las variables como

```
unsigned int x;
```

es decir, enteros que no pueden ser negativos.

Ejercicio: Implementa este método para generar números aleatorios enteros. ¿Cómo se puede convertir en un método para generar números aleatorios en el intervalo $[0,1]$? ¿En el intervalo $[0,1)$?

Ejercicio: Dibujar pares de estos números aleatorios en $[0,1]$ en un plano para verificar que no hay ninguna correlación que se pueda detectar a ojo.

Hoy en día se usa el método de "Mersenne Twister" (algo así como el torcedor de Mersenne) que usa los números primos de Mersenne, es decir, del tipo $2^{2^{n-1}}$. La secuencia que se genera es periódica, pero el periodo es de $2^{19937} - 1 \simeq 10^{600}$ números.

Para iniciar la secuencia de números aleatorios, se utiliza una *semilla* ("seed"), que es simplemente el valor inicial con el cual se empieza la iteración. Al utilizar otra semilla, se generará otra secuencia de números pseudo-aleatorios.

3.5. Funciones

Las *funciones* se pueden considerar como subprogramas, o subrutinas, que ejecutan una tarea reducida. Cada función debería de corresponder a una tarea dada.

Las funciones se *declaran* como es el caso de la función `main`. El caso más sencillo es:

```
void saludo() {
  cout << "Bienvenido al programa." << endl;
}
```

Aquí:

- `f` es el nombre de la función

- `()` es la lista de parámetros de la función
- el bloque `{ ··· }` es el cuerpo de la función –las operaciones que la función llevará a cabo
- **`void`** quiere decir que la función no regresa ninguna información.

El declarar una función *no implica* que la función se utilizará. Para utilizar una función, hay que *llamarla* desde otro lado del programa. Para hacerlo, se pone el nombre de la función, seguido por paréntesis que contienen los argumentos que se enviarán a la función, por ejemplo:

```
int cuad(int a) {
  return a * a;
}

int main() {
  int a = 17;
  cout << "El cuadrado de " << a << " es " << cuad(a);
}
```

Esta función acepta un *argumento* `a`, cuyo tipo debemos especificar. También regresa un entero.

Puede haber mas que un **`return`** en una función.

Una función debe ser corta, tener un nombre que representa lo que hace, y ser clara. Una vez que se ha comprobado que una función hace lo que debería de hacer, se puede olvidar del contenido de la función y reutilizar la misma dentro del mismo u otros programas. Si una función crece demasiado, es necesario dividirlo en distintas funciones, cada una de las cuales hace una tarea propia. Estas sub-funciones se pueden llamar desde la función principal.

De la misma manera, la función `main` *no* debería contener todo el programa, sino llamar a distintas funciones con nombres representativos, quienes llevan al cabo las distintas tareas requeridas.

Ejercicio: Calcular e imprimir las raíces cuadradas y las raíces cuárticas de los enteros de 1 a 100 utilizando el algoritmo Babilónico.

3.6. Rutinas para números aleatorios

En C++, ya existe una función `rand()` para generar números aleatorios *enteros*, que es suficiente para un uso "casual". Esta función está declarada en la librería `cstdlib`. Esta función regresa un número aleatorio entero entre 0 y una constante muy grande `RAND_MAX`, que normalmente está dado por de $2^{31}-1 \simeq 2\times 10^9$. (También viene definido en `cstdlib`.)

Para generar un número real en el intervalo $[0,1)$, uno pensaría en

```
double drand() {
  return rand() / RAND_MAX;
}
```

Sin embargo, aquí estamos dividiendo dos números enteros, lo cual da 0 (o tal vez 1 de vez en cuando), ya que operaciones entre enteros siempre regresan la parte entera de la respuesta. Por lo tanto, es necesario convertir uno de los enteros a precision doble:

```
double drand() {
  return double(rand()) / RAND_MAX;
}
```

Esto está perfecto para números en el intervalo $[0,1]$. Para el intervalo $[0,1)$ (es decir, que no incluye 1), hay que usar

```
double drand() {
  return double(rand()) / (RAND_MAX+1.);
}
```

Ejercicio:

- ¿Cómo se pueden generar números reales en el intervalo $[a,b)$?
- ¿Qué tal números enteros en $[i,j)$?
- ¿Cómo se pueden generar -1 ó 1 con probabilidad $\frac{1}{2}$ cada uno?
- ¿Cómo se puede generar 0, 1, 2 ó 3 con probabilidades 0.5, 0.25, 0.125 y 0.125, respectivamente?

3.7. Archivos de cabecera

Ya que tenemos el código para generar distintos tipos de números aleatorios, lo podemos colocar en un *archivo de cabecera*, llamado, por ejemplo, `alea.h`. En cada programa que utiliza este mismo código, incluimos la línea

```
#include "alea.h"
```

y entonces las funciones ahí definidas estarán disponibles. En general, los archivos de cabecera proveen código útil que se puede incluir en distintos programas para reutilizarse, sin tener que reescribirse cada vez.

Nótese que en C++, se puede ocupar el mismo nombre para dos funciones distintas, siempre y cuando sus argumentos sean distintos.

Ejercicio: ¿Cómo se podría generar "A" con $\frac{1}{3}$ ó "B" con $\frac{2}{3}$?

Ejercicio: Generar "A" con probabilidad $\frac{1}{3}$ y "B" con probabilidad $\frac{2}{3}$ un total de N veces. Contar el número de As y Bs que se obtienen realmente.

3.8. Simulación de una caminata aleatoria en una dimensión

Con las herramientas que hemos desarrollado hasta la fecha, no está difícil implementar la simulación de una caminata aleatoria en una red uni-dimensional.

La única pieza de información que el caminante requiere es su posición, que será un número entero. En cada paso, se escoge si brincará a la izquierda o a la derecha, o si se quedará en el mismo lugar, y se actualiza su posición de acuerdo con esta elección.

Ejercicio: Implementa una caminata aleatoria discreta en una dimensión en un programa llamado `caminata`. Dibujar su posición contra el tiempo.

3.9. Correr varias veces una simulación

Si tenemos una simulación de este tipo, ¿cómo podríamos correrlo varias veces y dibujar la salida de todas las corridas?

Una manera sería implementar en el mismo programa otro bucle, donde se especifica el número de veces que correrse. Pero otra manera, que puede ser más flexible, es implementar el bucle desde la línea de comandos. Utilizaremos `bash` para los ejemplos –la sintaxis cambia para distintos tipos de shell. (Para entrar en una sesión de `bash`, si se encuentra utilizando otro shell, simplemente se pone el comando `bash`.)

Muchos de los comandos que teclean en la línea de comandos son programas que `bash` corre. Uno de ellos en Linux es `seq`, que genera secuencias de números[1]:

```
$ seq 1 10
```

`bash` provee una manera de captar la salida de este comando con la sintaxis

```
$ echo $(seq 1 10)
```

donde el comando `echo` simplemente imprime su argumento y `$( ··· )` capta la salida del comando.

Ahora podemos *iterar* sobre una lista de palabras con un **`for`**:

```
$ for i in $(seq 1 10); do echo "Hola"; done
```

El valor de una variable se obtiene a través de `$`:

```
$ for i in $(seq 1 10); do echo $i; done
```

Ejercicio: Compara la salida de la siguientes dos comandos:

```
$ for i in $(seq 1 10); do echo $i; done
$ for i in seq 1 10; do echo $i; done
```

Ejercicio: Corre el programa `caminata` 100 veces y captar su salida a un archivo. Grafica la posición contra el tiempo para cara corrida. Para eso, es útil saber que `gnuplot` trata bloques de líneas de datos en un archivo como corridas distintas si están separadas por dos líneas en blanco. En la versión de `gnuplot` 4.3 y mayores[2], se puede iterar en estos distintos bloques (llamados *índices*) del archivo con la siguiente notación:

```
plot for [j=1:100] "caminata.dat" index j # abreviacion de index: i
```

3.10. Cambiar los números aleatorios

Si volvemos a correr un programa que utiliza los números aleatorios que provee `drand()`, encontraremos que los mismos números salen en cada corrida del programa. Eso puede ser útil –por ejemplo, para identificar dónde ocurre un error.

Sin embargo, normalmente querramos generar distintos números aleatorios en cada corrida. Para hacerlo, basta con incluir la línea

```
#include <ctime>

int main() {
  srand(time(0));
}
```

La función `srand()` inicializa la *semilla* de los números aleatorios con su argumento. En este caso, utilizamos el tiempo del sistema, dado por `time(0)`, para escoger una semilla distinta cada vez[3].

[1] En BSD (utilizado en MacOS), el equivalente es `jot 10 1`.

[2] El comando `gnuplot --version` indica qué versión es.

[3] Esta función regresa el tiempo en segundos desde el 1ero de enero de 1970. Por lo tanto, dará el mismo resultado si se corre un mismo programa varias veces antes de que se haya actualizado este número de segundos.

Capítulo 4

Contenedores de datos: vectores

Muy a menudo es importante poder guardar y manipular distintas variables que están relacionadas entre sí de una u otra manera.

4.1. Caminante en dos dimensiones

¿Cómo podríamos simular una caminata aleatoria en una red cuadrada en dos dimensiones? En cada paso, el caminante debería escoger una dirección al azar, por ejemplo con igual probabilidad.

La manera más fácil de hacerlo es extendiendo el método de un caminante en una dimensión, dividiendo el intervalo $[0,1)$ ahora en cuatro partes, cada una de las cuales corresponde a una dirección diferente, y hacer un `if` con varias opciones:

```
int x, y; # posicion de caminante en dos dimensiones
double r = drand();

if (r < 0.25) {
  x++;
}
if ( (r > 0.25) && (r < 0.5) ) {
  y++;
}
if ( (r > 0.5) && (r < 0.75) ) {
  x--;
}
if ( (r > 0.75) ) {
  y--;
}
```

Nótese que la única manera de expresar condicionales del estilo de $0.25 < x < 0.5$ en C++ es a través de una condicional de la forma `&&`.

Una manera más corta de expresar lo mismo es utilizando `else`, seguido por otro `if`:

```
if (r < 0.25) {
  x++;
}
else if (r < 0.5) {
  y++;
}
else if (r < 0.75) {
  x--;
```

```
}
else {
  y--;
}
```

Sin embargo, estos conjuntos de **`if`** y **`else`** son difíciles de leer, está fácil de cometer un error, y además es muy poco extendible. Por ejemplo, si ahora queremos estudiar un caminante en 3 –ó en d– dimensiones, entonces tenemos que reescribir el código por completo. Por lo tanto, necesitamos replantear el problema, para que sea fácilmente extendible a otras situaciones.

4.2. Colecciones de datos: arreglos

La posición en d dimensiones espaciales está representada por d coordenadas. La manera más sencilla de representar este vector es simplemente definiendo d variables enteras por separado, como hicimos con `x` y `y` en dos dimensiones. Sin embargo, esto no refleja la estructura matemática, y tampoco será fácil de extender a otras dimensiones.

Por lo tanto, es necesario buscar una *estructura de datos* capaz de representar a este conjunto de coordenadas. Matemáticamente, podemos pensar en un *vector* con d entradas; buscamos entonces el análogo en C++ de un vector, visto como un conjunto ordenado de números.

Este análogo es un *arreglo* –una estructura que puede contener un número dado de datos del mismo tipo con una orden dada. En C++, la manera más clásica –pero menos flexible– de declarar un arreglo de d entradas es

```
int posicion[2];
```

Aquí, `posicion` se declara como un arreglo de 2 enteros, los cuales se pueden accesar con

```
posicion[0] = 3;
posicion[1] = -17;
cout << "El vector es (" << posicion[0] << ", " << posicion[1] << ")\n"
```

Nótese que la numeración empieza en *cero* y termina en $d-1$ en C++.

4.3. Contenedores tipo `vector`

Sin embargo, esta manera de declarar arreglos resulta demasiado rígido. Otra manera más flexible, disponible solamente en C++, es utilizando la librería `vector`, que forma parte de la biblioteca estándar de C++. `vector` es un *contenedor*, que es cualquier estructura de datos que contiene distintos datos del mismo tipo. Se declara como sigue:

```
vector<int> posicion(2);
```

lo cual declara `posicion` como un arreglo que contiene enteros, y tiene tamaño inicial 2 (el número de elementos que puede contener). La sintaxis para accesar los elementos de un vector es igual a la para accesar un arreglo: `posicion[0]`, etc.

La ventaja de un `vector` es que es un arreglo *dinámico* –su tamaño puede cambiar *mientras el programa corre*. En cualquier momento, podemos utilizar el comando `resize` para cambiar el número de entradas en el arreglo:

```
vector<int> datos;
cout << datos.size() << endl;

datos.resize(10);
```

```
datos.resize(20);
```

En la primera línea, se declaró `datos` sin un tamaño. Por lo tanto, empieza como un arreglo vacío, con cero elementos, como nos muestra el comando `datos.size()`.

Estas dos funciones son los primeros ejemplos claros que hemos visto de la *programación orientada a objetos*. Un `vector` es un objeto, que tiene *propiedades* (variables) –por ejemplo, su tamaño, y los datos que contiene– y además tiene acciones (funciones), llamadas *métodos*. Tanto las propiedades como los métodos le pertenecen al objeto, y se accesan a través del operador '`.`'. Esto tiene la ventaja de que otros objetos pueden tener variables y funciones con los mismos nombres, sin crear ningún conflicto, ya que queda claro en cada momento cuál función o variable se requiere, al especificar a qué objeto pertenece.

Otro método sumamente útil de los `vector`s es `push_back()`, lo cual agrega un elemento al final del `vector`. Por lo tanto, no es necesario preocuparse por el tamaño:

```
vector<int> datos;
datos.push_back(5);
cout << "datos tiene " << datos.size() << " elementos\n";
```

Nótese que el nombre `vector` es un poco confuso del punto de vista matemática, ya que actúa simplemente como un contenedor de datos –no hay ninguna operación matemática definida para este tipo de `vector`. Visto sus capacidad de cambiar de tamaño, podemos pensar en `vector` más como una lista ordenada de objetos de un tipo dado que un vector matemático.

4.4. Caminatas aleatorias en dos dimensiones

Ahora podemos regresar a implementar una caminata aleatoria en dos dimensiones. La posición será un `vector` de dos coordenadas, las dos enteros. ¿Cuál es la dinámica del caminante?

Visto en coordenadas Cartesianas, el caminante tiene un vector de desplazamiento $\Delta\mathbf{x}$ en cada paso, desplazándose de $\mathbf{x}$ a $\mathbf{x}' = \mathbf{x} + \Delta\mathbf{x}$. En el caso más sencillo, este vector en dos dimensiones puede ser $(1,0)$, $(-1,0)$, $(0,1)$ ó $(0,-1)$. De manera similar, en 5 dimensiones, tendríamos vectores de desplazamiento del tipo $(0,-1,0,0,0)$. En cada caso, hay exactamente una coordenada que cambia en cada paso, y esta coordenada se incrementa o decrementa en 1.

Por lo tanto, podemos pensar que en cada paso, el caminante decide en qué dirección moverse: verticalmente u horizontalmente, en el caso de dos dimensiones. Luego decide si incrementar o decrementar la coordenada en esta dirección. Para eso, necesitamos poder escoger un entero al azar.

4.5. Más números aleatorios

Hasta ahora, hemos generado números aleatorios solamente en el intervalo $[0,1)$. Para generar números en el intervalo $[0,c)$, para algún valor de $c > 0$, simplemente escalamos por c. De ahí podemos ver que podemos generar números en el intervalo $[a,b)$ haciendo una traslación. Por lo tanto, podemos declarar una nueva función como sigue:

```
double drand2(double a, double b) {
  return a + (b-a) * drand();
}
```

Pasamos dos variables a la función como *argumentos*, cuyos valores emplea para calcular su respuesta. Además, aprovechamos el hecho de que ya contamos con la función `drand` para no repetir código.

De hecho, C++ cuenta con un mecanismo de *sobrecarga* de funciones. Eso quiere decir simplemente que podemos dar el mismo nombre a dos funciones distintas, siempre y cuando difieren en el número de argumentos

que tienen (para que el compilador puede distinguir cuál se requiere en cada situación). Por lo tanto, podemos reescribir la declaración como

```
double drand(double a, double b) {
  return a + (b-a) * drand();
}
```

Esto tiene la ventaja de que no es necesario recordar cuál nombre hay que utilizar en cada caso.

Ahora, para generar enteros al azar, podemos asociar el intervalo $[3,4)$ con 3, el intervalo $[4,5)$ con 4, etc. Así que si generamos 3.6, regresaremos el entero 3. Esto se logra con

```
int irand(int i, int j) {
  // Regresa un entero uniforme en $[i, j)$
  // NB: No incluye j como posibilidad

  return int( floor( drand(i, j) ) );
}
```

Aquí, dado dos enteros i y j, primero generamos un número aleatorio real entre i y j. Nótese que estos enteros se convierten de manera automática a `double`s al utilizarse como argumentos en la función `drand()`. Luego se encuentra el mayor entero que es menor o igual al resultado con la función `floor()`, que está definida en la librería `<cmath>`. Finalmente, se convierte el tipo de la salida de `floor()` (que regresa un `double`) a un entero. Nótese que el número j nunca se puede generar por esta función.

Ejercicio: Modifica la función `irand()` para generar un entero uniforme entre i y j que sí puede incluir j como posibilidad.

4.6. Caminante aleatorio en 2 dimensiones

Regresando al caminante aleatorio en 2 dimensiones, podemos reescribir la parte principal como sigue:

```
vector<int> posicion(2);
int d;

int direccion = irand(0,2);

double r = drand();
if (r < 0.5) {
  posicion[direccion] ++;
}
else {
  posicion[direccion] --;
}
```

Las condicionales en C++ son expresiones booleanas, que también toman valores. Por ejemplo,

```
int b = -1;
int a = (b < 0);
```

le asigna a `a` el valor del resultado de evaluar la condición `b < 0`. Estos valores se pueden utilizar para hacer cálculos.

Ejercicio: ¿Cómo se puede reescribir el caminante aleatorio en dos dimensiones máquina –como siempre, es cuestión de probar para ver cual opción es más rápida.

(i) Traza la trayectoria de una caminata aleatoria en 2 dimensiones. Utiliza el comando `set size ratio -1` para que los ejes se vean en la proporción correcta (1:1).

(iii) Traza la trayectoria de 100 caminatas aleatorias.

(iv) Dibuja las posiciones *finales* de 10000 caminatas aleatorias después de distintos números de pasos.

Ejercicio: Repite el ejercicio anterior para una caminata aleatoria en 3 dimensiones, utilizando el comando `splot` de `gnuplot` para dibujar en 3 dimensiones. (En las nuevas versiones de `gnuplot`, se puede utilizar `set view equal xyz` para obtener un efecto parecido a `set size ratio -1` en dos dimensiones.)

Capítulo 5

Referencias, archivos, argumentos de la línea de comandos y plantillas de función

En este capítulo, veremos algunas técnicas de C++ básico que nos ayudan a producir datos bien estructurados.

5.1. Calcular promedios en una función

Supongamos que nuestro programa produce unos datos, y quisiéramos calcular el promedio y otras propiedades estadísticas de estos datos. Tenemos tres posibilidades:

1. Actualizar una suma corrida cada vez que generamos un dato, y calcular el promedio al final;
2. Guardar los datos en un arreglo mientras el programa corre, y calcular las propiedades al final;
3. Guardar los datos en un archivo en el disco, y procesarlos posteriormente.

La primera opción tiene la ventaja que está más sencillo, pero se puede volver complicado calcular muchas cantidades, y hay que modificar el código adentro del programa para hacerlo. La tercera opción es tal vez la mejor, ya que los datos "crudos" están disponibles para calcular nuevas cantidades posteriormente.

Por el momento, consideremos la opción dos. Mientras generamos los datos, los vamos guardando en un arreglo, que en C++ se podría implementar con un `vector`:

```
vector<int> pos;
pos.push_back(3);
```

Podemos utilizar el método `push_back()` del `vector` para agregar más datos en cualquier momento.

Ejercicio: Los vectores tienen un método `capacity()` que reporta la cantidad de espacio que está reservado actualmente para que crezca el vector. Investiga cómo cambia esta capacidad al agregar cada vez más elementos con `push_back()`.

Al final del programa querramos calcular, por ejemplo, un promedio. Para poder reutilizar este código, quisiéramos ponerlo en una función:

```
double promedio(vector<int> v) {
  int suma = 0;
  for (int i=0; i < v.size(); i++) {
    suma += v[i];
  }
  return suma / double(v.size());
}
```

Nótese que pasamos un vector completo como argumento a la función.

5.2. Referencias

Para entender mejor el efecto de pasar argumentos a funciones, consideremos un caso muy sencillo:

```
void f(int a) {
  cout << "En f(): a = " << a << endl;
  a = 3;
  cout << "En f(): a = " << a << endl;
}

int main() {
  int b = 7;
  cout << "En main(): b = " << b << endl;
  f(b);
  cout << "En main(): b = " << b << endl;
}
```

¿Cuál será la salida de este programa? Podríamos esperar que el valor de `b` cambiaría, ya que es el argumento de la función `f()`, donde se modifica. Sin embargo, eso no es cierto –el valor de `b` no cambia.

La razón por esto es que la variable `a` se crea como una variable nueva que existe nada más adentro de la función `f()` –es decir, es una variable *local*. Su valor es una *copia* del valor de `b`, y por lo tanto, cuando se le asigna un nuevo valor, no afecta el valor de `b`.

Sin embargo, hay situaciones en las que *sí* queremos que las funciones modifiquen a las variables externas de ellas. Eso se logra utilizando una *referencia*:

```
void f(int& a) {
  a = 3;
}
```

Una referencia, declarada usando un ámpersand `&` después del tipo de la variable, declara un "alias", u otro nombre, de una misma variable. En este caso, `a` se vuelve otro nombre para `b` –se puede considerar como un *puntero* a `b`– y, por lo tanto, cuando se modifica `a` es ahora equivalente de modificar `b`.

Regresemos al caso de la función `promedio` de la sección anterior. Lo que acabamos de ver es que cuando declaramos la función usando

```
double promedio(vector<int> v) {
}
```

el vector `v` será una *copia* del vector original. Se copiarán todos los datos a un nuevo `vector`, solamente para calcular su promedio. Esto se puede evitar al utilizar una referencia, así que la declaración se vuelve

```
double promedio(vector<int>& v) {
}
```

Ahora `v` es simplemente otro nombre para el mismo objeto, y ya no se copia ninguna información.

Sin embargo, esto permitiría modificar a los datos originales, que tampoco queremos. Para evitar eso, ponemos la declaración siguiente:

```
double promedio(const vector<int>& v) {
```

```
}
```

La palabra **const** (“constante”) indica al compilador que esta función *no* está permitida modificar el contenido del objeto.

Parte II

Métodos de Monte Carlo con cadenas de Markov

Capítulo 6

El modelo de Ising y la física estadística

En este capítulo, introduciremos un modelo básico en la física estadística, el *modelo de Ising*, y reseñaremos los procedimientos básicos de la física estadística.

La meta de la física estadística es la de calcular las propiedades macroscópicas y termodinámicas de un sistema, a partir de una descripción microscópica del mismo. Por lo tanto, para cada modelo, cabe especificarlo al dar su *Hamiltoniano*, es decir, la función que determina la energía total del sistema cuando éste se encuentra en un microestado dado.

6.1. El modelo de Ising

El modelo de Ising es interesante ya que es un modelo sumamente sencillo de plantear, pero que exhibe distintos fenómenos de interés, tales como las transiciones de fase. Es un modelo de un imán cristalino, y consiste en muchas partículas, llamadas *espines*, que se encuentran en lugares fijos en una red cristalina. Los espines pueden tomar, en el caso más sencillo, dos valores diferentes, apuntándose hacia arriba o hacia abajo.

Los espines modelan los momentos magnéticos con los electrones sin pareja en un metal ferromagnético como el hierro. En tales materiales, hay una interacción entre espines cercanos, llamada la *interacción de intercambio*, que implica que éstos tienen una tendencia a alinearse con sus vecinos, ya que eso es un estado más energéticamente favorable.

Para convertir esta descripción en un modelo, consideremos una red de N sitios, con etiquetas $i = 1, \ldots, N$. En cada sitio i de la red hay un espín $\sigma_i \in \{\pm 1\}$, es decir, cada espín puede tomar los valores $+1$ y -1 (arriba y abajo, respectivamente). La configuración completa del sistema se denota por $\sigma := (\sigma_i)_{i=1,\ldots,N}$.

Para especificar la energía, consideraremos solamente interacciones entre pares de espines, especificadas a través de la energía de interacción $E(i,j)$ entre los espines σ_i y σ_j. Si los espines tienen el mismo valor, entonces asignaremos una energía favorable $E(i,j) = -J$, mientras que si tienen valores opuestos, la energía es $E(i,j) = J$. Además, solamente los espines *vecinos* podrán interactuar.

Entonces tenemos

$$E(i,j) = \begin{cases} -J\sigma_i\sigma_j, & \text{si } i \text{ y } j \text{ son sitios vecinos;} \\ 0, & \text{si no.} \end{cases} \tag{6.1}$$

Nótese que el producto $\sigma_i\sigma_j$ da justamente 1 si los espines están alineados, y -1 si no.

Por lo tanto la energía total de una configuración es

$$\mathscr{H}(\sigma) = E(\sigma) := -J \sum_{\langle i,j \rangle} \sigma_i\sigma_j. \tag{6.2}$$

Aquí, la notación $\langle \cdot \rangle$ denota una suma sobre pares de sitios vecinos en la red. Hacemos una distinción entre la energía de interacción $E(\sigma)$, y la energía total del sistema $\mathscr{H}(\sigma)$. En este caso, no hay otra contribución a la

energía, así que el Hamiltoniano está dado por la energía de interacción entre espines. Podemos pensar en esta suma como una suma sobre los *enlaces* entre los sitios.

6.2. Física estadística

Ya que hemos especificado un modelo de manera microscópica, la física estadística da una "receta" para calcular sus propiedades termodinámicas macroscópicas. Boltzmann y Gibbs mostraron que si ponemos nuestro sistema microscópico en contacto con un baño térmico a temperatura T, pero tal que no puede intercambiar materia con el baño, entonces el sistema se puede modelar utilizando el *ensamble*[1] *canónico*, en donde la frecuencia con la cual el sistema visita una configuración σ dada es

$$p(\sigma) \propto e^{-\beta \mathscr{H}(\sigma)}, \tag{6.3}$$

donde $\beta := 1/(k_B T)$ es la temperatura inversa, y k_B es la constante de Boltzmann. Tomaremos siempre unidades para las cuales $k_B = 1$.

Normalizando la distribución de probabilidad, usando que $\sum_{\sigma\in\Omega} p(\sigma) = 1$, donde Ω es el conjunto de estados posibles del sistema, obtenemos que

$$p(\sigma) = \frac{1}{Z(\beta)} e^{-\beta \mathscr{H}(\sigma)}, \tag{6.4}$$

donde

$$Z(\beta) := \sum_{\sigma\in\Omega} e^{-\beta \mathscr{H}(\sigma)} \tag{6.5}$$

se llama la *función de partición*, ya que involucra la manera en la cual se distribuye, o particiona, la probabilidad entre los distintos microestados del sistema. El promedio de un observable (es decir, una cantidad que podemos medir en el sistema) Q está dado por

$$\langle Q \rangle := \sum_{\sigma\in\Omega} Q(\sigma) p(\sigma). \tag{6.6}$$

Si conocemos $Z(\beta)$, entonces podemos derivar todas las cantidades termodinámicas. Por ejemplo, la energía interna macroscópica U se identifica –en el *límite termodinámico* $N \to \infty$, $V \to \infty$ con $\rho := N/V$ fija– con el promedio microscópico $\langle E \rangle$. Pero

$$\langle E \rangle = \sum_{\sigma} E(\sigma) p(\sigma) = \frac{1}{Z} E(\sigma) e^{-\beta E(\sigma)} = -\frac{1}{Z}\frac{\partial Z}{\partial \beta} = -\frac{\partial}{\partial \beta} \log Z(\beta). \tag{6.7}$$

Además, podemos decir que la energía libre $F(\beta)$ está dada por

$$F(\beta) = -k_B T \log Z(\beta). \tag{6.8}$$

De una manera similar, las cantidades macroscópicas termodinámicas se pueden escribir como funciones y derivadas de la función de partición $Z(\beta)$.

6.3. Los cálculos sin imposibles

En principio, la receta que provee la física estadística nos permite calcular cualquier cantidad macroscópica deseada, a partir del conocimiento del Hamiltoniano y la función de partición de un sistema. Sin embargo, en la práctica esta esperanza no se cumple, ya que los cálculos requeridos son *intratables*, debido al siguiente problema combinatórico.

Consideremos el modelo de Ising con N espines. El número total de configuraciones del sistema es $|\Omega| = 2^N$. Para un sistema pequeño, de 100×100 espines en dos dimensiones, tenemos que $N = 10000$, y por lo tanto $|\Omega| = 2^{10000} \simeq 10^{3000}$! Este número es realmente enorme, ¡aproximadamente 10^{3000} veces la edad del universo

[1] También llamado "conjunto representativo".

en segundos! Por lo tanto, no importa el tamaño de la computadora que tengamos, nunca podremos llevar al cabo el cálculo de la función de partición “exacta”.

La solución a esto es la de aproximar las sumas con sumas sobre un número reducido de configuraciones. Sin embargo, eso funcionará solamente si las configuraciones sobre las cuales se suma se escogen de manera apropiada para el problema, para que sean las “relevantes” o “dominantes”. La sigiuente parte del curso se tratará de las maneras de resolver este problema.

Capítulo 7

Muestreo

Como vimos en el capítulo anterior, es normalmente imposible llevar a cabo las sumas requeridas por la teoría de la física estadística para calcular la función de partición y los promedios deseados. Por lo tanto, es necesario *muestrear* ciertas configuraciones "representativas", es decir, escoger de manera estadística o aleatoria –pero al mismo tiempo, lista– las configuraciones sobre las cuales sumaremos.

La primera idea que podríamos tener es la de muestrear de manera uniforme sobre todas las configuraciones, escogiéndolas "al azar", es decir, con igual probabilidad, tal que cada espín tiene igual probabilidad de apuntar hacia arriba o hacia abajo. Sin embargo, está intuitivamente claro que eso dará configuraciones σ que tienen aproximadamente el mismo número de espines para arriba como para abajo, y por lo tanto, una magnetización $M(\sigma)$ y energía $E(\sigma)$ que se concentran alrededor de 0. Eso sería adecuado para investigar las propiedades del sistema a altas temperaturas, donde justamente las configuraciones tienen un peso estadístico, es decir, probabilidad $p(\sigma)$, más o menos uniforme. Sin embargo, a bajas temperaturas, el sistema se concentrará alrededor de sus estados bases, donde todos los espines se alínean, mientras que tales configuraciones casi nunca se generarán de manera uniforme.

7.1. Muestreo no-uniforme

Por lo tanto, es necesario introducir un muestreo no-uniforme, es decir, escoger distintas configuraciones según una distribución de probabilidad $p(\mu)$ (que no necesariamente es la de Boltzmann). Más adelante veremos cómo eso se puede hacer; por lo momento, supongamos que ya lo hemos logrado.

Consideremos un muestreo finito de configuraciones generadas según esta distribución de probabilidad, $(\mu_1, \ldots, \mu_M)$. En una corrida larga, esperamos que algunas de estas configuraciones son iguales. Enumeremos las *distintas* configuraciones como $\mu^{(1)}, \ldots, \mu^{(C)}$. Entonces $\mu^{(i)}$ debería aparecer $Mp(\mu^{(i)})$ veces, aproximadamente.

Si ahora queremos calcular, por ejemplo, la función de partición, pensaríamos primero en calcular

$$Z \stackrel{?}{\simeq} \sum_{i=1}^{M} e^{-\beta E(\mu_i)}. \tag{7.1}$$

Sin embargo, eso daría

$$Z \stackrel{?}{\simeq} \sum_{i=1}^{C} p(\mu^{(i)}) e^{-\beta E(\mu^{(i)})}. \tag{7.2}$$

Por lo tanto, las configuraciones aparecen en la suma pesada por la distribución $p(\mu)$ de muestreo que nosotros imponemos.

Para eliminar este efecto no-deseado, es necesario dividir por las p. Así obtenemos las siguientes expresiones aproximadas para las cantidades de interés:

$$Z \simeq \sum_{i=1}^{M} \frac{1}{p(\mu_i)} e^{-\beta E(\mu_i)}; \tag{7.3}$$

$$\langle Q \rangle \simeq \frac{\sum_{i=1}^{M} \frac{1}{p(\mu_i)} Q(\mu_i) e^{-\beta E(\mu_i)}}{\sum_{j=1}^{M} \frac{1}{p(\mu_j)} e^{-\beta E(\mu_j)}}. \tag{7.4}$$

Estos estimados convergen a los valores verdaderos cuando el número de datos M converge al infinito.

7.2. Muestreo de Boltzmann

¿Cuál distribución $p(\mu)$ nos dará los mejores resultados? Una posibilidad es la de intentar reproducir el sistema físico de la manera más fiel posible: muestreamos según la distribución de Boltzmann, $p(\mu) = \frac{1}{Z} e^{-\beta E(\mu)}$. Al sustituir esta distribución en el estimado, encontramos que

$$\langle Q \rangle \simeq \frac{1}{M} \sum_{i=1}^{M} Q(\mu_i). \tag{7.5}$$

Es decir, el promedio, tomando en cuenta las frecuencias de visita, se reduce a un promedio simple de los datos.

Sin embargo, esto nos lleva a otro problema: ¡no sabemos cómo generar distintas configuraciones según la distribución de Boltzmann! La solución, sugerida por primera vez por Metropolis et al. en los años 1950s, es que podemos generar esta –o cualquier otra– distribución usando un *proceso estocástico* que *converja* hacia esta distribución. En particular, emplearemos normalmente una *cadena de Markov*.

7.3. Cadenas de Markov

La idea es que generaremos una secuencia de configuraciones σ_t en el tiempo t. En cada paso, partiendo de la configuración σ actual, el sistema podrá escoger distintas configuraciones τ en el siguiente paso de tiempo $t+1$. Esta elección no será determinista, sino aleatorio: distintas configuraciones τ se podrán elegir con distintas *probabilidades de transición* $P(\sigma \to \tau)$. Eso se repite en cada paso de tiempo (discreto), así dando lugar a un proceso estocástico. Para que sea una cadena de Markov, se requiere además que las probabilidades $P(\sigma \to \tau)$ dependen solamente del estado actual σ. Una caminata aleatoria normal es un ejemplo.

Denotemos por $w_\sigma(t)$ la probabilidad de que el sistema se encuentre en el estado σ al tiempo t. Entonces tenemos que

$$w_\tau(t+1) = \sum_\sigma w_\sigma(t) P(\sigma \to \tau). \tag{7.6}$$

Eso es, la probabilidad de estar en cierto estado en el siguiente paso de tiempo es igual a la suma de las probabilidades de que estaba en otra configuración en el paso anterior, por la probabilidad de que la nueva configuración se escogió. Aunque la suma es, en principio, sobre todas las configuraciones del sistema, en la práctica se reduce a una suma sobre solamente las configuraciones de donde se puede alcanzar la configuración nueva τ.

Al examinar la forma del miembro de derecho en esta ecuación, nos percatamos de que tiene una forma parecida a una multiplicación de matrices. Por lo tanto, se puede reescribir como

$$\mathbf{w}(t+1) = \mathsf{P} \cdot \mathbf{w}(t). \tag{7.7}$$

Nótese que tenemos que definir las componentes de la matriz P como $P_{\tau\sigma} := P(\sigma \to \tau)$ para que salga esta ecuación.

Al iterar esta relación, tenemos que

$$\mathbf{w}(t) = \mathsf{P}^t \cdot \mathbf{w}(0). \tag{7.8}$$

Por lo tanto, el comportamiento asintótico en el tiempo de las probabilidades $w_\mu(t)$ depende de los valores y vectores propios de la matriz P de probabilidades de transición. Por ejemplo, si el valor propio más grande fuera menor que 1 en módulo, entonces todas las probabilidades se decaerían a 0. Sin embargo, dado que $\sum_\mu w_\mu(t) = 1$ para todo t, esto no es posible. De hecho, esta condición de normalización implica que el valor

propio de mayor módulo es 1, y los demás valores propios son menores que 1 en módulo. Por lo tanto, la distribución converge a un límite $\mathbf{w}(\infty)$ cuando $t \to \infty$, que satisface una invarianza bajo la acción de P:

$$w_\tau(\infty) = \sum_\sigma w_\sigma(\infty) P(\sigma \to \tau) \tag{7.9}$$

Es decir, esta distribución ya no cambia en el tiempo. Por lo tanto, la podemos considerar como una distribución de *equilibrio*. De hecho, es justamente la distribución –única– de equilibrio $p(\sigma)$ que genera esta cadena de Markov.

Distintas cadenas de Markov difieren en sus probabilidades de transición $P(\sigma \to \tau)$. Al cambiar estas probabilidades de transición, se cambia en general la distribución de equilibrio que se genera. Para poder generar una distribución de equilibrio dada, es crucial, por lo tanto, averiguar cómo se relacionan la P y la $p(\sigma)$ que generar.

7.4. Condiciones de balance detallado y ergodicidad

Reescribiendo la ecuación (7.9) en términos de la distribución de equilibrio $p(\sigma)$, obtenemos

$$p(\tau) = \sum_\sigma p(\sigma) P(\sigma \to \tau). \tag{7.10}$$

Pero también podemos escribir el miembro de izquierda como

$$p(\tau) = p(\tau) \sum_\sigma P(\tau \to \sigma) = \sum_\sigma p(\tau) P(\tau \to \sigma), \tag{7.11}$$

donde la primera igualdad utiliza la normalización que la suma vale 1 (“hay que ir a algún lado”). Finalmente, llegamos a la *condición de balance*

$$\sum_\sigma p(\tau) P(\tau \to \sigma) = \sum_\sigma p(\sigma) P(\sigma \to \tau). \tag{7.12}$$

La condición de balance es una condición *necesaria* para que la distribución $p(\sigma)$ sea una distribución invariante. Una manera posible de satisfacer esta ecuación es igualando de manera individual los términos de los dos lados. De ahí resulta una condición *suficiente*, pero no necesaria, llamada la *condición de balance detallado*:

$$p(\sigma) P(\sigma \to \tau) = p(\tau) P(\tau \to \sigma), \tag{7.13}$$

que se tiene para *todas* las configuraciones σ y τ. Podemos pensar que esta ecuación especifica que los flujos de probabilidad en equilibrio de una configuación a otra es igual al flujo de regreso.

Normalmente, queremos generar una distribución de equilibrio $p(\sigma)$ dada. Así que la versión de la condición de balance detallado que nos interesa es:

$$\frac{P(\sigma \to \tau)}{P(\tau \to \sigma)} = \frac{p(\tau)}{p(\sigma)}. \tag{7.14}$$

En el caso del muestreo de Boltzmann, la distribución de equilibrio que nos interesa es justamente la de Boltzmann. Por lo tanto, obtenemos que

$$\frac{P(\sigma \to \tau)}{P(\tau \to \sigma)} = \frac{p(\tau)}{p(\sigma)} = \frac{\frac{1}{Z} e^{-\beta \mathscr{H}(\tau)}}{\frac{1}{Z} e^{-\beta \mathscr{H}(\sigma)}} = e^{-\beta \Delta \mathscr{H}}, \tag{7.15}$$

donde hemos definido $\Delta\mathscr{H} := \mathscr{H}(\tau) - \mathscr{H}(\sigma)$, es decir, $\Delta\mathscr{H}$ es la diferencia de energías entre la configuración nueva y la anterior. Ésta es la condición que se tiene que satisfacer, entonces, para generar la distribución de Boltzmann. Nótese que los factores de la función de partición Z *se cancelan* de la ecuación. Es por eso que podemos generar la distribución adecuada, sin tener que conocer el valor de Z; ésta es la fuerza y la clave del método.

Ergodicidad

Otra condición importante que tiene que satisfacer el proceso es la de *ergodicidad*. Eso requiere que cualquier configuración está accesible desde cualquier otra, a través un número finito de pasos. Si no fuera el caso, entonces podría haber unas partes del espacio de configuraciones que nunca se pudiera alcanzar.

7.5. Regla de Metropolis

Nos queda la tarea de resolver la ecuación (7.15) para las probabilidades de transición $P(\sigma \to \tau)$ para una distribución de equilbrio $p(\sigma)$ dada. La manera que adoptaremos –que está lejos de ser única– se debe a Metropolis et al., quien lo inventaron a principos de los 1950s.

Metropolis et al. se dieron cuenta de que podemos reducir la dificultad de resolver la ecuación (7.15) al separar la probabilidad de transición $P(\sigma \to \tau)$ en dos partes:

$$P(\sigma \to \tau) = g(\sigma \to \tau)\, \alpha(\sigma \to \tau), \tag{7.16}$$

donde $g(\sigma \to \tau)$ es una probabilidad de *generar*, o *proponer* el cambio hacia la nueva configuración τ, y $\alpha(\sigma \to \tau)$ es la probabilidad de que *aceptemos* este cambio si se propone. Si no aceptamos el cambio, es decir que lo *rechazamos*, entonces el sistema se queda en la misma configuración σ. Por lo tanto, hay en general una probabilidad no-cero de quedarse en el mismo estado, lo que implica que $P(\sigma \to \sigma) \neq 0$.

La idea es que ahora nosotros imponemos las probabilidades $g(\sigma \to \tau)$ de proponer cambios que queramos, por ejemplo los que sean fáciles de implementar. Luego hace falta resolver para las $\alpha(\sigma \to \tau)$. Esto resulta ser más fácil si las $g(\sigma \to \tau)$ se escogen de manera que sean *simétricas*, ya que en este caso tenemos que

$$\frac{P(\sigma \to \tau)}{P(\tau \to \sigma)} = \frac{g(\sigma \to \tau)\alpha(\sigma \to \tau)}{g(\tau \to \sigma)\alpha(\tau \to \sigma)} = \frac{\alpha(\sigma \to \tau)}{\alpha(\tau \to \sigma)}. \tag{7.17}$$

Como ejemplo, consideremos el caso del modelo de Ising. La manera más sencilla de generar nuevas configuraciones es que cambiemos solamente un espín a la vez. Así que las probabilidades de generación $g(\sigma \to \tau)$ son $1/N$ si las configuraciones σ y τ difieren en solamente un espín, mientras que son 0 si no. En este caso, la probabilidad de regresar a la configuración anterior es igual, y entonces $g(\sigma \to \tau)$ sí es simétrica.

Solución de Metropolis et al. para probabilidades de aceptación

Finalmente, nos quedamos con la ecuación

$$\frac{\alpha(\sigma \to \tau)}{\alpha(\tau \to \sigma)} = e^{-\beta \Delta \mathscr{H}} \tag{7.18}$$

en el caso de la ecuación de Boltzmann. (En otro caso, el miembro de derecha de la ecuación será distinto, pero el siguiente argumento también se aplica.)

No queremos que las $\alpha(\sigma \to \tau)$ sean pequeñas, ya que en este caso el sistema se quedará mucho tiempo en cada configuración, y así no explorará bien las configuraciones posibles. Por lo tanto, queremos *maximizar* las $\alpha(\sigma \to \tau)$. Ya que son probabilidades, están acotadas entre 0 y 1. Por lo tanto, ponemos el más grande de $\alpha(\sigma \to \tau)$ y $\alpha(\tau \to \sigma)$ igual a 1. Si $\Delta\mathscr{H} > 0$, es decir que $\mathscr{H}(\tau) > \mathscr{H}(\sigma)$, entonces $e^{-\beta\Delta\mathscr{H}} < 1$, y entonces según la ecuación vemos que $\alpha(\sigma \to \tau) < \alpha(\tau \to \sigma)$. Así que ponemos $\alpha(\tau \to \sigma) = 1$, lo que implica que $\alpha(\sigma \to \tau) = e^{-\beta\Delta\mathscr{H}}$.

Finalmente, encontramos la *regla de Metropolis*: el cambio de la configuración σ a la configuración nueva τ se acepta con probabilidad

$$\alpha(\sigma \to \tau) = \begin{cases} e^{-\beta\Delta\mathscr{H}}, & \text{si } \Delta\mathscr{H} > 0 \\ 1, & \text{si no} \end{cases} = \min\{1, e^{-\beta\Delta\mathscr{H}}\}. \tag{7.19}$$

Nótese que el cambio siempre se acepta si la energía de la configuración nueva es menor o igual que la energía de la configuración anterior. Aunque eso sea sorprendente, resulta ser el método más eficiente, como acabamos de demostrar.

7.6. Algoritmo de Metropolis

Ya tenemos todos los ingredientes para simular el modelo de Ising –o, de hecho, cualquier modelo cuyo Hamiltoniano conocemos– a una temperatura fija:

1. Empezar con una configuración inicial σ.
2. Proponer una nueva configuración τ de manera simétrica –en el modelo de Ising, cambiar un espín.
3. Calcular el cambio de la energía total que resulta, $\Delta\mathscr{H} := \mathscr{H}(\tau) - \mathscr{H}(\sigma)$.
4. Si $\Delta\mathscr{H} \leq 0$, entonces aceptar el cambio –el sistema se queda en el nuevo estado τ.
   Si $\Delta\mathscr{H} > 0$, entonces aceptar el cambio con la probabilidad $e^{-\beta\Delta\mathscr{H}}$. Si no, entonces rechazar el cambio –el sistema se queda en el estado σ.
5. Repetir desde (2).

Lo que hemos mostrado en este capítulo es que el algoritmo anterior genera con éxito, en el límite de tiempo largo, las configuraciones σ con frecuencias que convergen a la distribución de Boltzmann $p(\sigma)$.

Capítulo 8

Mediciones

Después de contar con un programa que simula el modelo de Ising, estamos listos para poder *medir* las cantidades que nos interesan, es decir, promedios de observables Q. Por ejemplo, nos interesa medir la magnetización promedio $\langle M \rangle$, que identificamos con la magnetización macroscópica del sistema. A diferencia del enfoque de la física estadística usual, sacaremos este promedio literalmente como un promedio –de datos que medimos en la simulación.

8.1. Equilibración

En el capítulo anterior, vimos que en el caso donde generamos las configuraciones según la distribución de Boltzmann, el promedio $\langle M \rangle$ se calcula como un promedio simple de distintos datos. Sin embargo, estos datos se tienen que tomar *en equilibrio*, es decir cuando la distribución generada por la cadena de Markov ya se ha convergido a la de Boltzmann. Por lo tanto, es necesario esperar un *tiempo de equilibración* antes de empezar a tomar datos.

Para estimar el tiempo de equilibración, podemos dibujar una observable como función del tiempo, por ejemplo al magnetización instantánea $M(t) := M(\sigma(t))$, donde $\sigma(t)$ es la configuración alcanzada al tiempo t. Esta cantidad llegará a equilibrarse, fluctuando alrededor de cierto valor, después de un tiempo. (Para evitar que encontremos un estado metaestable, podemos lanzar distintas corridas desde distintas condiciones iniciales.) El tiempo de equilibración se puede estimar, entonces, desde la gráfica de $M(t)$ contra t.

8.2. Promedios en equilibrio

Después de que se ha equilibrado el sistema, empezamos a muestrear datos. A cada rato, medimos la observable $Q(t)$ que nos interesa, por ejemplo la magnetización instantánea, y guardamos esta información. Al final –después de muestrear lo que consideramos como suficientes datos– sacamos un promedio de estos datos, lo cual da un estimado de la $\langle Q \rangle$ deseada.

Además, es necesario calcular un estimado del *error* posible en la medición de $\langle Q \rangle$, es decir, el error que cometemos al calcular el promedio de población Q mediante un muestreo. Esto se conoce como el error estándar del promedio, y resulta ser un factor $1/\sqrt{N}$ menor que la desviación estándar $\sqrt{\langle Q^2 \rangle - \langle Q \rangle^2}$ que mide el tamaño de las fluctuaciones en Q alrededor de su promedio.

8.3. Variación de cantidades macroscópicas

El procedimiento anterior da como resultado un dato para cada observable, en cada simulación con temperatura T fija. Usualmente, nos interesa cómo varía el comportamiento del sistema cuando variamos los parámetros del mismo, por ejemplo la temperatura T y el campo externo h. Por lo tanto, hay que repetir las mediciones con distintas simulaciones a distintas temperaturas T, para extraer una gráfica de $\langle Q \rangle (T)$ como función de T.

8.4. Funciones de respuesta

Aparte de los promedios de cantidades que corresponden a funciones termodinámicas macroscópicas, hay cantidades adicionales que nos interesan. Por ejemplo, las funciones de respuesta macroscópicas χ –la susceptibilidad magnética– y c –el calor específico– también son importantes físicamente. Para extraerlas, podríamos derivar numéricamente los datos que ya tenemos. Sin embargo, eso no es una buena solución, ya que las derivadas numéricas suelen introducir mucho ruido. La alternativa adecuada es la de re-expresar estas funciones en términos de promedios de observables, las cuales podemos calcular directamente desde la simulación.

Hay otras funciones termodinámicas macroscópicas que ni siquiera se pueden expresar directamente como promedios, tales como la entropía S y energía libre F. A diferencia de la física estadística usual, estas cantidades son las más difíciles de calcular en simulaciones tipo Metropolis.

Parte III

Programación orientada a objetos

Capítulo 9

Programación orientada a objetos: clases

Hasta ahora, hemos tratado a C++ principalmente como una versión mejorada de C. Sin embargo, el punto principal de este lenguaje es que introduce la habilidad de crear nuevos tipos de *objeto*, que viven al lado de los tipos ya definidos, como `int` y `double`. Estos objetos nos proporcionan la manera de modelar manera más fiel en la computadora el problema que nos interesa.

9.1. Los objetos

Igual que las variables usuales, todo objeto en C++ tiene un *tipo*, lo cual se especifica a través de la declaración de una *clase*. Igual que los objetos en el mundo real, los objetos pueden tener propiedades internas, o *estados*, y maneras de *interactuar* con el mundo, a través de funciones.

9.1.1. Ejemplo: partículas

Supongamos que estamos modelando una partícula en 2 dimensiones. Podríamos representar la partícula a través de ciertas variables:

```
double x, y;    // position
double vx, vy;  // velocity
x += dt * vx;  // actualizamos su posicion
y += dt * vy;
```

Si ahora queremos introducir otra partícula, podríamos poner

```
double x1, y1;
double x2, y2;
double vx1, vy1;
double vx2, vy2;
```

Si además tienen masas y colores, entonces tenemos

```
double m1, m2;
int c1, c2;
```

Para muchas partículas introduciríamos mejor arreglos.

Ahora, ¿en dónde están las partículas? La respuesta tiene que ser “en ningún lado”, o más bien, “en todos lados”. El problema es que existen muchas variables que conceptualmente están relacionadas, pero no hemos logrado hacer la conección entre ellas en el programa.

Pero tenemos muchas variables que conceptualmente están relacionadas, todas *pertenecen* a una partícula, pero no hay manera de expresar esto en el lenguaje. (De hecho, en C hay 'struct'. Lo que vamos a ver es un 'struct' muchísimo ampliado.)

9.1.2. Clases

Para hacer esta conección, lo que queremos hacer es *agrupar* a las variables de una partícula en un grupo, u *objeto*, llamado `Particula`, y poder declarar variables de este tipo como `Particula p`. (Nótese que es una convención que los objetos definidos por el usuario tengan nombres que empiezan con mayúsculas.)

Para implementar un objeto de este tipo en C++, se declara una *clase*:

```
class Particula {
  double x;
  double y;

  double vx;
  double vy;
};   // no se olvide el ';' al final de la declaracion
```

Nótese que la clase se declara con llaves, y con un punto y coma al final. La declaración se coloca antes de la de `main()` y de cualquier otra función.

Esta declaración define un nuevo *tipo* `Particula`, que luego está disponible en el programa, pero todavía no hay ninguna partícula existente. Para crear una partícula de este tipo, creamos una *instancia* de la clase, o sea, un objeto que tiene este tipo:

```
Particula p1;
Particula p2;
```

La sintaxis es igual que la de declarar una variable de cualquier tipo pre-definido.

9.1.3. Acceso a datos de un objeto

Podemos pensar en los objetos como cajas que contienen ciertos datos, que corresponden a sus estados internos. Ahora es necesario poder accesar la información que se encuentra adentro del objeto. Para hacerlo utilizamos la siguiente sintaxis:

```
p1.x = 3.0;
p1.y = 2.0;
```

La sintaxis `p1.x` denota a la variable `x` que vive adentro del objeto llamado `p1`, de tipo `Particula`.

Pero hay un problema: por defecto, todo lo que hay adentro de una clase es *privado* (**`private`**) –eso es, estas variables sólo se pueden accesar o cambiar desde dentro de la clase. Para cambiar el acceso a publico, ponemos **`public`** antes de las partes que son públicas:

```
class Particula {
    double x;
    double y;

  public:
    double vx;
    double vy;
};
```

En este caso, las variables de posición siguen siendo privadas, mientras que las velocidades son públicas.

¿Para qué queremos que algunas variables sean privadas? Así, nosotros, como desarrolladores de la clase, podemos garantizar que no se pueden modificar estos datos desde afuera. Por ejemplo, al desarrollar una aplicación para un banco, no queremos que la cantidad de dinero disponible se pudiera modificar directamente.

Nótese que `p1.x` y `p2.x` son dos variables *distintas* que pertenecen a objetos distintos. Cada instancia de una clase tiene su propia copia de todas las variables de la clase.

9.1.4. Funciones adentro de los objetos: métodos

Hasta ahora, las clases han actuado solamente como un tipo de contenedor de datos, lo cual sí es un uso común de los objetos en C++. Sin embargo, los objetos en el mundo no sólo tienen propiedades, sino también pueden ejecutar acciones y pueden interactuar con su entorno–es decir, pueden hacer cosas. Para modelar esto, las clases también pueden contener funciones, llamadas *métodos*. Se declaran como funciones usuales, pero adentro de la clase. También pueden ser públicas o privadas:

```
class Particula {
  private:
    double x, y;
    double vx, vy;

  public:
    void mover(double dt) {
      x += dt*vx;
      y += dt*vy;
    }
};
```

Aquí hemos definido una función `mover()`. Al poner

```
Particula p1, p2;
p1.mover();
```

le decimos a la partícula llamada `p1` que *se mueva* –así, podemos interactuar con el objeto, e indirectamente cambiar sus estados. De igual forma, `p2.mover()` le dice a `p2` que ella se mueva. Todos los detalles de *cómo* se mueven las partículas están *escondidos* adentro del objeto. Incluso podemos cambiar por completo la estructura interna del objeto, sin tener que modificar el código afuera de la clase. Eso se llama *encapsulación de datos*, ya que hemos logrado esconder los datos adentro de una "cápsula" hermética que no podemos modificar de manera directa, sino solamente a través de las funciones que se proveen para este propósito.

Otro benificio de este enfoque es que podríamos tener otro objeto, por ejemplo `Disco`, que tuviera también un método llamado `mover`. Así que hemos evitado la necesidad de definir dos funciones con dos nombres distintos, `moverParticula()` y `moverDisco()`. De hecho, veremos más adelante que incluso puede ser útil tratar a un `Disco` como un *tipo e* `Particula`, lo cual se logra con una técnica llamada *herencia*.

9.2. Constructores

En nuestro ejemplo `Particula`, no hay manera de inicializar la posición y la velocidad de la partícula, puesto que ya no tenemos acceso directo a las variables constituyentes del objeto. Podríamos declarar una función pública, por ejemplo `inicializar()`, con este fin, que tomara argumentos representando la posición y velocidad iniciales del objeto. Sin embargo, no hay manera de obligar al usuario llamar a estas funciones.

C++ provee una solución a este problema: la *función constructora*, o *constructor*. Es una función especial que se llama *cada vez* que se crea una instancia de un objeto de un tipo dado. El constructor no tiene tipo de

regreso, y debe portar exactamente el mismo nombre como la clase. Se declara adentro de la declaración de la clase, y se utiliza para inicializar todas las variables de una clase, para lo cual hay una sintaxis especial:

```
class Particula {
  private:
    double x, y;
    double vx, vy;

  public:
    Particula() : x(0.0), y(0.0), vx(0.0), vy(0.0)
    { }
};
```

Muy a menudo en las clases simples, la función parece no tener ningún contenido –el contenido está en la *lista de inicialización*, que dice justamente cómo inicializar a las variables que pertenecen a la clase.

Acordémonos que en C++, dos funciones distintas pueden llevar el mismo nombre, siempre y cuando tienen listas de parámetros distintas. Así que podemos definir otro constructor que acepta como argumentos la posición y velocidad iniciales. También podemos especificar valores por defecto:

```
Particula(double xx=0.0, double yy=0.0, double vxx=0.0, double vyy=0.0)
    : x(xx), y(yy), vx(vxx), vy(vyy) {
}
```

Los argumentos se pasan al momento de crear un objeto:

```
Particula p1(3.0, 4.0);  // vx y vy son 0
Particula p3(3.0, 4.0, 0.1, 0.1);
```

9.3. Clases adentro de clases

Ya tenemos una representación de una partícula mucho más cercana al modelo matemático. Sin embargo, cuando pensamos en la posición, o velocidad, de una partícula, lo vemos como un vector. Este aspecto todavía no está representado.

¿Cómo podemos incorporar esta idea? Lo que requerimos es *otra clase*, `Vec`, que representa a un vector dos-dimensional:

```
class Vec {
  public:
    double x, y;

  Vec(double xx, double yy) : x(xx), y(yy)
  { }
};
```

Por ahora, pensaremos en `Vec` como sólo un contenedor de datos, por lo cual declaramos a sus variables como públicas[1].

Ponemos la declaración de `Vec` antes de la de `Particula`, para poder utilizar los `Vec` adentro de `Particula`:

[1]Eso también se puede hacer con la palabre clave **`struct`** en lugar de **`class`**. Un **`struct`** es un objeto cuyas variables son públicas por defecto. Una versión más reducida de este concepto (sin los métodos) también existe en C.

```
class Particula {
  Vec posicion;
  Vec velocidad;

  Particula(double xx=0.0, double yy=0.0, double vxx=0.0, double vyy=0.0)
    : posicion(xx, yy), velocidad(vxx, vyy)
  { }

  void mover(double dt) {
    posicion.x += dt * velocidad.x;
    posicion.y += dt * velocidad.y;
  }
};
```

Ahora sí hemos logrado una representación muy fiel del concepto de partícula que tenemos.

Falta un detalle: en la función `mover`, nos gustaría poder utilizar la notación *vectorial*

```
posicion += dt * velocidad;
```

para corresponder completamente con la fórmula matemática vectorial que escribiríamos. Eso es posible, pero requiere un concepto más complicado: la *sobrecarga de operadores*.