programación, fractales y métodos numéricos

(sólo para físicos de amplio criterio)

J. Fernando Rojas R.

Facultad de Ciencias Físico Matemáticas
BUAP

julio, 2005

... y se iba por las casas de los chinos pobres, repartiendo sacos de arroz y pescado seco, y hablando con los viejos y los niños, y leyendo, en aquellos libros que empiezan por la última página, lo que Confucio dijo de los perezosos, que eran peor que el veneno de las culebras, y lo que dijo de los que aprenden de memoria sin preguntar por qué, que no son leones con alas de paloma, como el hombre debe ser, sino lechones flacos, con cola de tirabuzón y las orejas caídas...

José Martí

Prólogo

Estas notas de programación, en realidad, parecieran notas de "todología", y hay más de una razón para que esto sea así. Una de ellas, como siempre, es la de mayor peso en esta idea y corresponde a un evento: un curso de métodos numéricos para físicos, que podríamos calificar como exitoso en el sentido amplio de la palabra. El curso fue construido en el verano del 2005 día a día revisando y discutiendo el material, los temas, las ideas.

A veces uno descubre -redescubre- que la libertad es una parte fundamental de nuestra vida como individuos, como sociedad, como artesanos, científicos o artistas: como seres humanos. A raíz de lecturas, interpretaciones y diálogos sobre el tema y sobre la visión de los sistemas complejos y la autoorganización como eventos naturales -que, por cierto, nos empeñamos en olvidar- fuimos estructurando un curso en el que se conjugaron varios elementos que tal vez valga la pena nombrar.

La idea se vino gestando en mi cabeza años atrás con diferentes cursos y motivaciones, con diferentes enfoques de lo mismo, de los mismos tópicos, con distintas formas de asumir mi papel y tratar de que los estudiantes asumieran el suyo de manera activa. En pocas palabras la idea central es que todos y cada uno de los estudiantes de cualquier curso lo tomen como suyo porque es suyo, de nadie más.

En estas notas no hay más pretención que la de aportar un material que motive hacia el uso -en estas fechas indispensable- de la computadora en la resolución de problemas que van, desde simples problemas numéricos en el sentido tradicional del término, hasta tópicos de interés reciente que cada día más afectan a todos los campos del conocimiento, de nuestra forma de asumir la vida y el mundo.

Índice

Prólogo . . . 5

Índice . . . 7

1 ¿Quién es *Zeropio*? . . . 9

1.1 ¿Quién es *Zeropio*? . . . 9
1.2 Cómo se hace *un* programa para *Zeropio*... . . . 11
1.3 ... y cómo se hacen *dos* . . . 13
1.4 Ensamblador... . . . 15

2 Gráficas con *gnuplot* . . . 17

2.1 ¿Para qué? . . . 17
2.2 *gnuplot* interactivo y programado . . . 17
2.3 Modo paramétrico contra modo normal . . . 19
2.4 Algunos ejercicios . . . 20

3 Programación y algoritmos . . . 21

3.1 Estructuras de programación . . . 21
3.2 Algoritmos . . . 22
3.2.1 ¿Qué hace por "dentro" *gnuplot*? . . . 22
3.3 Estructuras de datos . . . 27
3.3.1 Arreglos (vectores y matrices) . . . 27
3.3.2 Otras estructuras . . . 29
3.3.3 Objetos . . . 29
3.4 Elementos de *C++*, *Fortran* y *Pascal* . . . 29

4 Sistemas dinámicos y métodos numéricos . . . 31

4.1 Los sistemas dinámicos . . . 31
4.2 Osciladores, sistemas dinámicos y Euler . . . 33
4.2.1 Otra vez los *flujos*: puntos fijos y raíces de funciones . . . 38
4.2.2 Puntos fijos, raíces de funciones y mapeos dinámicos: el caos . . . 40

5 Dinámica, Fractales, Simulación . . . 43

5.1 Los números aleatorios: volúmenes y áreas . . . 43
5.2 Los fractales matemáticos: Sierpinski . . . 46
5.3 Regreso a los Sistemas Dinámicos: Mapas de Poincaré y Caos . . . 48

Bibliografía . . . 49

Índice de materias . . . 51

Capítulo 1
¿Quién es *Zeropio*?

Es probable que tú ya sepas programar y, si es así, este capítulo no servirá -supongo- de gran cosa, pero podría resultar entretenido.

1.1 ¿Quién es *Zeropio*?

Bueno, *Zeropio* es un modelo simplificado construido a partir de una computadora real y pretende resolver en parte el problema que ocurre cuando queremos aprender a manejar un auto y no sabemos qué tiene dentro del cofre.

Se trata de *ver* explícitamente lo que ocurre dentro de una computadora, cómo funcionan sus partes y cómo se relacionan entre sí. Estarás de acuerdo en que si sabemos cómo funciona el embrague de un coche, el acelerador, etc. nos será mucho más sencillo manejarlo: sabríamos lo que cada una de nuestras acciones produce en el interior del móvil. Podríamos decir que en este capítulo se trata de manipular las partes internas de una computadora aunque, desde luego, esto se hace de manera muy diferente al caso de un coche: vamos a conocer las instrucciones que *Zeropio* sabe y puede ejecutar y además cuáles son las partes de su "mecanismo" que se activan y forman parte de cada una de las acciones.

Primero vamos a suponer algunas cosas sobre *Zeropio* análogas en mayor o menor medida a las que ocurren con una computadora real:

- solamente sabe leer *unos* y *ceros* (usa el sistema binario para representar números internamente[1.1])
- cuando se enciende lo primero que hace es comenzar a ejecutar órdenes (binarias, desde luego) que de algún modo se encuentran en su memoria (RAM, Random Access Memory)
- su tarea fundamental es transformar información y para esto solamente sabe efectuar dos operaciones básicas: sumar y comparar

No sabemos si sea obvio, pero parece que si una de estas máquinas solamente usa dos dígitos y solamente puede sumar y comparar, entonces habrá que hacer implementaciones ingeniosas que permitan a la computadora volverse tan indispensable -como lo es hoy- para casi cualquier tarea administrativa, tecnológica o científica.

1.1. Veremos que no solamente los números: también, con ingenio, se pueden representar muchos otros objetos usando este sistema de numeración. En particular la representación del número decimal (sobre nuestra acostumbrada base de diez dígitos) 728.4 nos indica la siguiente operación:

$$728.4_{10} = 7 \times 10^2 + 2 \times 10^1 + 8 \times 10^0 + 4 \times 10^{-1}$$

De manera análoga, tomando ahora alguna cantidad en representación binaria, podemos evaluar su valor numérico de la siguiente forma:

$$101100.01_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 44.25_{10}$$

En ambos casos (al igual que se haría en cualquier otra base) seguimos la convención de que la posición de cada dígito con respecto al punto decimal nos indica un término que es el producto del dígito por una potencia de la base usada.

De esto se sigue, que el número más grande que se puede representar en codigo binário usando 8 dígitos es: $11111111_2 = 255_{10}$, que es un número natural. La representación interna (en una computadora) de números enteros requiere un bit (el más significativo) para definir el signo de la cantidad. Por esta razón el número entero mayor y menor que se puede manejar en binario usando 8 bits será: $\pm\, 1111111_2 = \pm\, 127_{10}$ (el primer bit se usa para definir el signo y corresponde al valor $1 \times 2^7 = 128_{10}$).

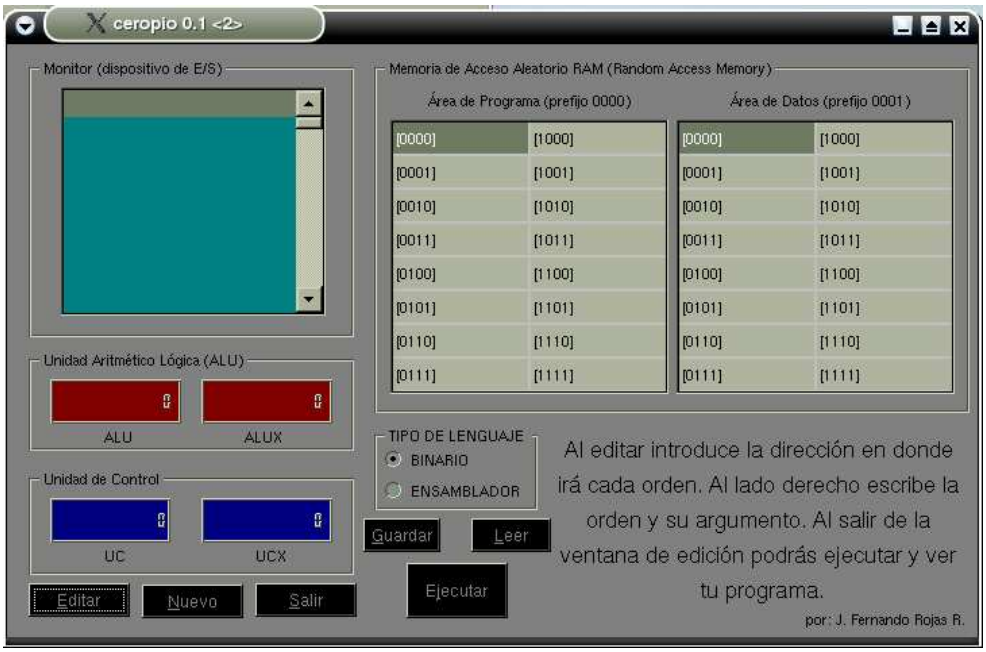


Figura 1.1. Esta es la imagen de *Zeropio*

Vamos a suponer que *Zeropio* conoce un conjunto mínimo de instrucciones u órdenes en lenguaje binario (ceros y unos); que cada orden en la memoria (o dato) ocupa un *byte*[1.2], y que, en el caso de las órdenes, éstas serán interpretadas tomando 4 bits para la orden específica y los 4 bits restantes, en su caso, como argumento de la orden dada:

ORDEN	ARGUMENTO	especificaciones
0000	****	inicio de programa
0001	dddd	copiar valor en [dddd] a ALU
0010	dddd	sumar valor en [dddd] al contenido de ALU
0011	dddd	restar valor en [dddd] al contenido de ALU
0100	dddd	multiplicar valor en [dddd] al conenido de ALU
0101	dddd	copiar contenido de ALU en [dddd] RAM-datos
0110	dddd	leer dato del teclado y guardar en [dddd]
0111	dddd	escribir valor de [dddd] en monitor
1000	pppp	salto incondicional a *pppp*
1001	pppp	¿mayor que 0? cambiar ejecución a *pppp*
1010	pppp	¿menor que 0? cambiar ejecución a *pppp*
1011	pppp	¿igual a 0? cambiar ejecución a *pppp*
1100	dddd	incrementar en 1 el valor *dddd*
1101	pppp	salto a *pppp* con retorno
1110	****	retorno de subprograma
1111	****	fin de programa

Tabla 1.1. El conjunto de órdenes de *Zeropio*. Los * indican que el valor de los 4 bits no tiene importancia. Los bits llenos con "d" indican una dirección de la memoria RAM asociada con la parte de Datos, los que tienen "p" indican una dirección de memoria en la parte de Programas.

1.2. Un *byte* es un conjunto de 8 bits, ocho estados *cero* o *uno* consecutivos que sirven para representar datos (numéricos o de otro tipo) en los diferentes dispositivos de la computadora: | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | representa el número entero 6 (en notación binaria).

Una de las principales partes de *Zeropio* es la Memoria RAM que, esquemáticamente puede verse así:

PROGRAMAS	PROGRAMAS	DATOS	DATOS
[0000]	[1000]	[0000]	[1000]
[0001]	[1001]	[0001]	[1001]
[0010]	[1010]	[0010]	[1010]
[0011]	[1011]	[0011]	[1011]
[0100]	[1100]	[0100]	[1100]
[0101]	[1101]	[0101]	[1101]
[0110]	[1110]	[0110]	[1110]
[0111]	[1111]	[0111]	[1111]

RAM (en las celdas vacías se asume que hay ceros)

y que es el lugar donde se guardan datos e instrucciones. Como puede verse en la figura la RAM tiene dos partes: una dedicada a almacenar connjuntos de órdenes y la otra para almacenar información (datos, en este caso, numéricos). Los dígitos binarios que aparecen del lado izquierdo de cada celda se conocen como la *dirección* en RAM y, obviamente, sirve para identificar la localización de una orden o dato.

Una vez que el programa queda «escrito» en la RAM puede ser ejecutado. Cada orden será «leída», interpretada y ejecutada por el procesador. Aquí entra en juego la *Unidad de Control* (UC) que se encarga de guardar la dirección de la siguiente instrucción a ser ejecutada

UC
0000

en un celda llamada *registro*. Cuando una orden se ejecuta el registro guarda la dirección de la orden siguiente, por ejemplo, al incio puede tener la dirección 0000 que es la del inicio del programa y, cuando lee la orden que se encuentra en la dirección 0000, el registro de la UC se modifica por 0001 que es la dirección de la siguiente instrucción.

Las operaciones aritméticas y lógicas se efectúan en la *Unidad Aritmético-Lógica* (ALU) y la forma de efectuarlas es como sigue:

ALU
0000

- para hacer una operación aritmética hay que enviar, al registro de la ALU, un primer dato (para lo cual se usa la orden 0001) y, después, usar una orden de operación (suma: 0010, resta: 0011, producto: 0100) que tomará el valor almacenado en la dirección respectiva (el argumento de la orden) y lo operará con el valor que ya se encuentra en la ALU dejando ahí el resultado,
- las operaciones lógicas básicamente son comparaciones (recordemos que la computadora solamente sabe sumar y comparar). *Zeropio* solamente sabe comparar el valor del registro de la ALU con cero (mayor: 1001, menor: 1010 o igual: 1011 a cero). Si la comparación es cierta entonces cualquiera de estas órdenes modifica el registro de la UC poniendo ahí el argumento (los últimos 4 bits) de la orden: la orden 1001 1000 modificará el registro de la UC al valor 1000 si el valor en el registro de la ALU es mayor que cero.

1.2 Cómo se hace *un* programa para *Zeropio*...

El primer programa de ejemplo es muy sencillo y evaluará la suma de dos números dados por nosotros a la computadora para escribir el resultado al final. El programa ahí dentro es éste

PROGRAMAS	PROGRAMAS	DATOS	DATOS
[0000] 0000 0000	[1000]	[0000]	[1000]
[0001] 0110 1001	[1001]	[0001]	[1001] 00000011
[0010] 0110 1011	[1010]	[0010]	[1010]
[0011] 0001 1001	[1011]	[0011]	[1011] 00000100
[0100] 0010 1011	[1100]	[0100]	[1100]
[0101] 0101 1111	[1101]	[0101]	[1101]
[0110] 0111 1111	[1110]	[0110]	[1110]
[0111] 1111 0000	[1111]	[0111]	[1111] 00000111

y las órdenes serán ejecutadas secuencialmente siguiendo el orden de las *direcciones* en RAM: desde 0000 (que es la primera dirección en RAM del área de programas) hasta, en este caso, la dirección 0111 que contiene la orden de terminar la ejecución del programa y que es la celda número 7.

Veamos lo que ocurre cuando este programa es ejecutado por la computadora siguiendo la secuencia de las direcciones:

→ Dirección [0000], orden 0000: inicio del programa. La orden aparentemente no tiene sentido pero ¿qué pasaría si, como ocurre en la realidad, hubiera más de un programa ejecutándose en la RAM?
UC: 0001

→ Dirección [0001], orden 0110, argumento 1001: leer y guardar en RAM. En este caso la computadora esperará a que el usuario ingrese un valor que, una vez convertido a su representación binaria, será guardado en la dirección 1001. Pensemos por ejemplo en el número 3 y tendremos lo que se ve en el esquema de arriba.
UC: 0010 RAM (datos [1001]): 00000011

→ Dirección [0010], orden 0110, argumento 1011: leer y guardar en RAM. En este caso se asume, según el esquema de arriba, que se ingresó el número 4.
UC: 0011 RAM (datos [1011]): 00000100

→ Dirección [0011], orden 0001, argumento 1001: enviar valor a ALU. El valor contenido en la dirección 1001 (nuestro 3) será enviado a la ALU para ser "operado".
UC: 0100 ALU: 3 (00000011)

→ Dirección [0100], orden 0010, argumento 1011: sumar valor a ALU. El valor de la dirección 1011 (nuestro 4) será sumado al valor que ya está en la ALU, de manera que ahora tiene almacenado un 7 (00000011+00000100=00000111).
UC: 0101 ALU: 7 (00000111)

→ Dirección [0101], orden 0101, argumento 1111: guardar el valor de ALU en RAM. El valor de la suma (nuestro 7) será almacenado (como se ve en el esquema) en la dirección 1111.
UC: 0110 RAM (datos [1111]): 00000111

→ Dirección [0110], orden 0111, argumento 1111: escribir el valor de RAM. Escribirá el valor contenido en la dirección 1111 del área de datos en un dispositivo de *salida*. Al hacerlo hará un cambio de representación de binario a decimal.
UC: 0111

→ Dirección [0111], orden 1111, argumento 1111: terminación del programa. En este caso la orden es la importante y el argumento no es necesario (puede ser cualquier valor) y le indican al computador que el programa ha llegado a su fin. Cuando esto ocurre la RAM tendrá el estado mostrado arriba.
UC: 1000

Una orden del tipo *salto* modifica directamente el registro de la UC permitiendo que se «rompa» la secuencia normal de ejecución, por ejemplo

PROGRAMAS	PROGRAMAS	DATOS	DATOS
[0000] 0000 0000	[1000] 1111 0000	[0000]	[1000]
[0001] 0110 1001	[1001]	[0001]	[1001]
[0010] 0110 1011	[1010]	[0010]	[1010]
[0011] 0001 1001	[1011]	[0011]	[1011]
[0100] 0010 1011	[1100]	[0100]	[1100]
[0101] 0101 1111	[1101]	[0101]	[1101]
[0110] 0111 1111	[1110]	[0110]	[1110]
[0111] 1000 0001	[1111]	[0111]	[1111]

en la dirección [0111] aparece una orden de salto incondicional a la dirección (del tipo 'pppp', ver tabla de instrucciones) [0001]. Esta orden modifica el registro de la UC guardando en él este valor y, por lo tanto, la siguiente orden a ser ejecutada será 0110 1001 (que se encuentra en la dirección [0001]). ¿Qué es lo que hace este programa..?

Ejercicio 1.1. Construir los siguientes programas para *Zeropio*:

i. Que multiplique dos números,

ii. que lea un número y lo eleve al cuadrado,

iii. que lo eleve al cubo,

iv. que termine cuando la suma sea cero (por el momento no podemos usar números negativos),

v. que termine, cuando el primer sumando sea cero, al hacer una suma de tres números

vi. que escriba la secuencia de N a M de uno en uno

1.3 ... y cómo se hacen *dos*

Como dijimos antes, *Zeropio* tiene una parte de la RAM asignada para almacenar programas. Podríamos preguntarnos si en este apartado de la RAM cabe y/o se ejecuta un solo programa cada vez. Bueno, la respuesta es NO: en una computadora real siempre hay muchos programa ejecutándose simultáneamente y esto parece ser más o menos obvio si nos fijamos en cualquier computadora que esté funcionando: se mueve el ratón, recibe y muestra información en el navegador web, muestra la hora y la fecha, hace sonar un disco compacto, nos permite escribir un texto y mantener un diálogo escrito o hablado con alguien en otro lugar del planeta.

En particular (y para nuestros fines) es importante que un programa diseñado para alguna acción específica pueda ser utilizado por otro programa: la idea consiste en que no tendremos que "reprogramar" esa parte que ya está hecha. Por ejemplo, si tenemos un programa que "sabe" sumar dos números o bien que "sabe" generar los pares de datos que representan la solución de una ecuación diferencial, entonces no será necesario volver a programar esas dos tareas: ya están hechas. Entonces lo que resta es hacer queHay cuestiones de tipo técnico que iremos desarrol nuestro programa, el que requiere alguna de estas tareas para (seguramente) resolver otro problema, pueda llamar a escena al otro en el momento en que lo requiera[1.3].

Un ejemplo que podríamos construir aquí consiste en un programa que llama a un subprograma[1.4] cuya tarea es elevar un número (el *argumento* del subprograma) al cuadrado:

1.3. Esto es mucho más elemental que la ejecución de todas las tareas que mencionamos antes, pero es una buena analogía.

1.4. Que también suele llamarse *función*, *subrrutina* o *procedimiento*. Los nombres y otras cuestiones más específicas dependen del lenguaje que se esté usando.

PROGRAMAS	PROGRAMAS	DATOS	DATOS
[0000] 0000 0000	[1000] 0100 1111	[0000] 00001001	[1000]
[0001] 0110 1001	[1001] 1110 0000	[0001]	[1001] 00000011
[0010] 0001 1001	[1010]	[0010]	[1010]
[0011] 1101 0111	[1011]	[0011]	[1011]
[0100] 0101 0000	[1100]	[0100]	[1100]
[0101] 0111 0000	[1101]	[0101]	[1101]
[0110] 1111 0000	[1110]	[0110]	[1110]
[0111] 0101 1111	[1111]	[0111]	[1111] 00000011

→ Dirección 0000, orden 0000: inicio del programa. La orden aparentemente no tiene sentido pero ¿qué pasaría si, como ocurre en la realidad, hubiera más de un programa ejecutándose en la RAM?
UC: 0001

→ Dirección 0001, orden 0110, argumento 1001: leer y guardar en RAM. En este caso la computadora esperará a que el usuario ingrese un valor que, una vez convertido a su representación binaria, será guardado en la dirección 1001. Pensemos por ejemplo en el número 3 y tendremos lo que se ve en el esquema de arriba.
UC: 0010 RAM (datos [1001]): 00000011

→ Dirección[0010], orden 0001, argumento 1001: copia el valor de [0010] en la ALU (en este ejemplo se trata de que el valor de la ALU sirve al subprograma: es un *parámetro*)
UC: 0011 ALU: 00000011

→ Dirección [0011], orden 1101, argumento 0111: esta orden es más compleja que las vistas antes, de hecho hace varias tareas:
i) copia el valor de la UC al *registro auxiliar* UCX, esto es: guarda la dirección *de retorno* a la cual volverá después
UC: 0100 -> UCX: 0100, ALU: 00000011, ALUX: 00000000
ii) copia el argumento 0111 en la UC: la nueva dirección de ejecución del programa (un salto a la dirección donde comienza el subprograma)
UC: 0111, UCX: 0100, ALU: 00000011, ALUX: 00000000
iii) copia el valor de la ALU en el auxiliar ALUX que será usado, a partir del hecho de que la UCX tenga un valor no nulo, para efectuar las operaciones aritméticos-lógicas (suma, resta, multiplicación y saltos condicionados)
UC: 0111, UCX: 0100, ALU: 00000011, ALUX: 00000011

→ Dirección[0111], orden 0101, argumento 1111: copiar el valor de ALUX en RAM [1111] (nota que se usa ALUX en lugar de ALU y esto es porque UCX está «ocupado»)
UC: 1000, UCX: 0100, ALU: 00000011, ALUX: 00000011, RAM (datos [1111]): 00000011

→ Dirección [1000], orden 0100, argumento 1111: multiplicar el valor en [1111] de RAM con el valor de ALUX
UC: !001, UCX: 0100, ALU: 00000011, ALUX: 00001001, RAM (datos [1111]): 00000011

→ Dirección [1001], orden 1110: retorno (*return)*: aquí concluye el subprograma. El resultado del subprograma está en ALUX ($9_{10} = 00001001_2$). La orden de retorno cumple las siguientes tareas:
i) *mueve*[1.5] el valor de ALUX a ALU: ahora sabemos que el resultado del subprograma se encuentra en ALU,
UC: 1010, UCX: 0100, ALU: 00001001, ALUX: 00000000, RAM (datos [1111]): 00000011
ii) mueve el valor de UCX a UC: con esta acción la ejecución regresa a la dirección *de retorno* que había sido guardada
UC: 0100, UCX: 0000, ALU: 00001001, ALUX: 00000000, RAM (datos [1111]): 00000011

1.5. Debes notar la diferencia entre *copiar* y *mover*: copiar implica duplicar el valor de un lugar en el otro, mientras que mover implica que el valor «desaparece» del lugar que ocupaba dejando ceros.

→ Dirección [0100], orden 0101, argumento 0000: copiar el valor de ALU en RAM (datos [0000]) que es resultado «devuelto» por el subprograma
UC: 0101, UCX: 0000, ALU: 00001001, ALUX: 00000000, RAM (datos [0000]): 00001001

→ Dirección [0101], orden 0111, argumento 0000: escribe el resultado (que tenemos en RAM [0000])
UC: 0110

→ Dirección [0110], orden 1111: fin de programa
UC: 0111

1.4 Ensamblador...

Antes de decir algo: cuando uno lee la palabra «ensamblador» (en términos computacionales) se refiere a dos cosas: i) el *lenguaje* de bajo nivel escrito con palabras mnemónicas y direcciones binarias usando sistema hexadecimal y ii) el *programa* (o aplicación) que lee una serie de instrucciones escritas, precisamente en lenguaje ensamblador, para traducirlas a lenguaje binario que, como ya sabemos, es el único que la computadora interpreta.

Pensemos que podemos construir un programa que puede leer una orden como `READ 1A` y traducirla en la orden binaria 0110 1001

La posibilidad de tener un programa que lea un conjunto de órdenes en forma de palabras y las traduzca a binario permite al programador memorizar las órdenes de manera que la construcción de un programa se hace más rápidamente y con menos errores. Esto permite, también, implementar órdenes nuevas que serán traducidas no ya a una sola orden sino a un conjunto de ellas, por ejemplo:

BINARIO	ENSAMBLADOR
0000	INIT
0001	SEND
0010	ADDD
0011	REST
0100	MULT
0101	STOR
0110	READ
0111	WRIT
1000	JUMP
1001	JMPU
1010	JMPD
1011	JMPZ
1100	INCR
1101	SUBP
1110	RETN
1111	STOP

Si además de esto pudiésemos sustituir las direcciones binarias una notación más compacta el problema se simplificaría más todavía. Una forma de hacer esto es cambiando de *base* la numeración[1.6] así que, observando que en la numeración en *hexadecimal* (base 16) un dígito es equivalente a cuatro dígitos binarios

1.6. El compilador se encargará de traducir las direcciones de esta nueva representación a la binaria.

Capítulo 2

Gráficas con *gnuplot*

2.1 ¿Para qué?

En este apartado trataremos con la generación de gráficas en 2 y 3 dimensiones usando un graficador de licencia GNU (software libre) que tiene versiones para Windows y Linux.

Una cuestión importante para quienes no han programado antes es la referente a la sintaxis, es decir, a la forma en que se dan las órdenes a la computadora[2.1] así que en esta sección nos ocuparemos de esto dando un repaso, además, a las funciones y sus gráficas, a las propiedades de las funciones y sus transformaciones así como a las curvas y superficies paramétricas. No nos vamos a detener en cuestiones muy puntuales y/o técnicas sobre *gnuplot* pues esa información se encuentra en los manuales (también disponibles en la red internet) y en la ayuda local que se puede invocar con la orden `help`.

2.2 *gnuplot* interactivo y programado

El graficador "gnuplot" es un programa de aplicación compacto y versátil que se puede obtener por medio de internet e instalar prácticamente en cualquier sistema operativo.

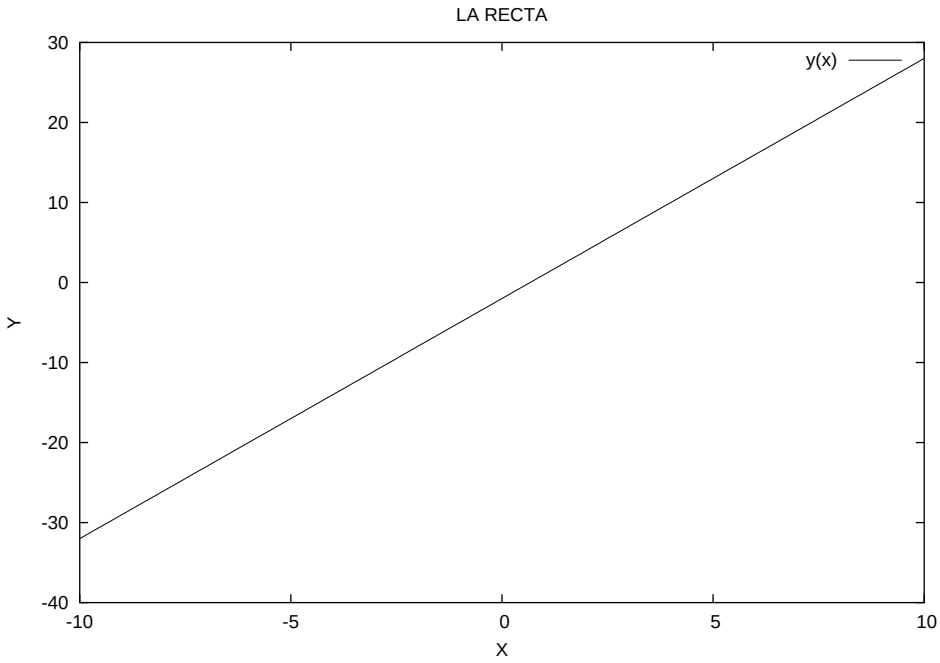
En este primer ejemplo vamos a ver las siguientes cosas: cómo poner título a la gráfica, cómo poner etiqueta (un texto) a cada uno de los ejes, cómo definir una función y cómo graficarla[2.2]:

```
   This is a TeXmacs interface for GNUplot.

GNUplot] set title "LA RECTA"
         set xlabel "X"
         set ylabel "Y"
         y(x) = 3*x-2
         plot y(x)
```

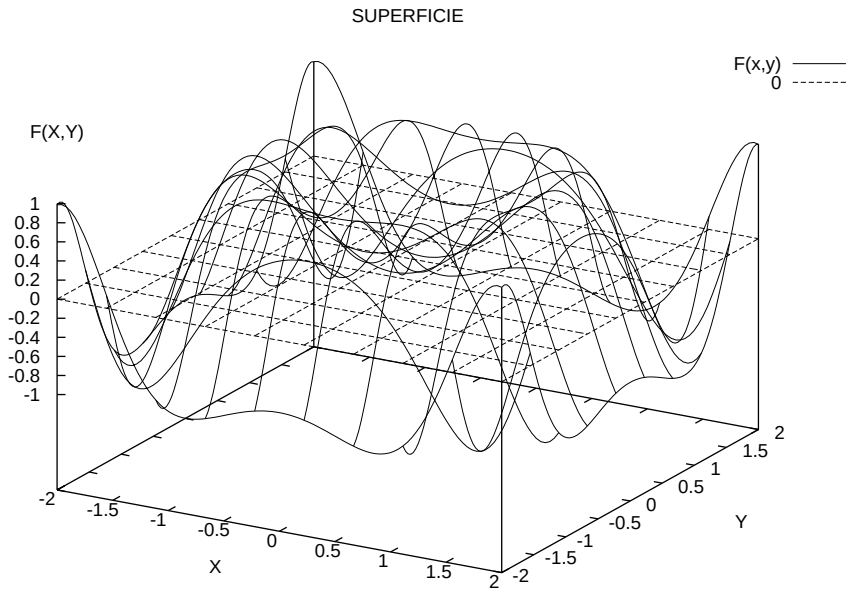
2.1. Dar órdenes a la computadora es una forma popular de decirlo. En realidad uno se refiere a la forma explícita en la cual uno debe escribir las órdenes, las sentencias, las instrucciones que deberá interpretar algún programa ya sea de aplicación o bien del sistema operativo en general.

2.2. En estos ejemplos estamos usando una interface de *TeXmacs* con *gnuplot*. En el modo interactivo cada línea u orden son independientes y no existe un orden predefinido para ellas salvo el lógico.



En esta gráfica el rango [-10,10] es el rango por defecto para cualquier variable de *gnuplot* y, en este otro ekemplo, veremos la gráfica de una superficie junto con el plano XY (que se logra haciendo que se grafique $Z = f(x, y) = 0$). Aparecen dos líneas en las que definimos los rangos visibles de nuestra superficie.

```
GNUplot] set title "SUPERFICIE"
         set xlabel "X"
         set ylabel "Y"
         set zlabel "F(X,Y)"
         F(x,y)=sin(x**2+y**2)
         set xrange [-2:2]
         set yrange [-2:2]
         splot F(x,y),0
```



En realidad esta sección es un pretexto para comenzar a hacer «programas» que realmente no lo son: más bien son una secuencia de órdenes que no requieren estructura. Sin embargo resulta útil hacer un repaso de geometría al mismo tiempo que una iniciación a lo que sería «programar» un graficador.

Cualquier editor de texto simple es útil para guardar los conjuntos de órdenes (*scripts*) de los ejemplos anteriores. *gnuplot* los «ejecuta» con la orden `load ''nombredelscript.gp''`. La extensión en este caso es usada como abreviatura de *gnuplot*.

Las variables que son reconocidas como tales, para construir funciones, gráficas, etc., son x y y.

2.3 Modo paramétrico contra modo normal

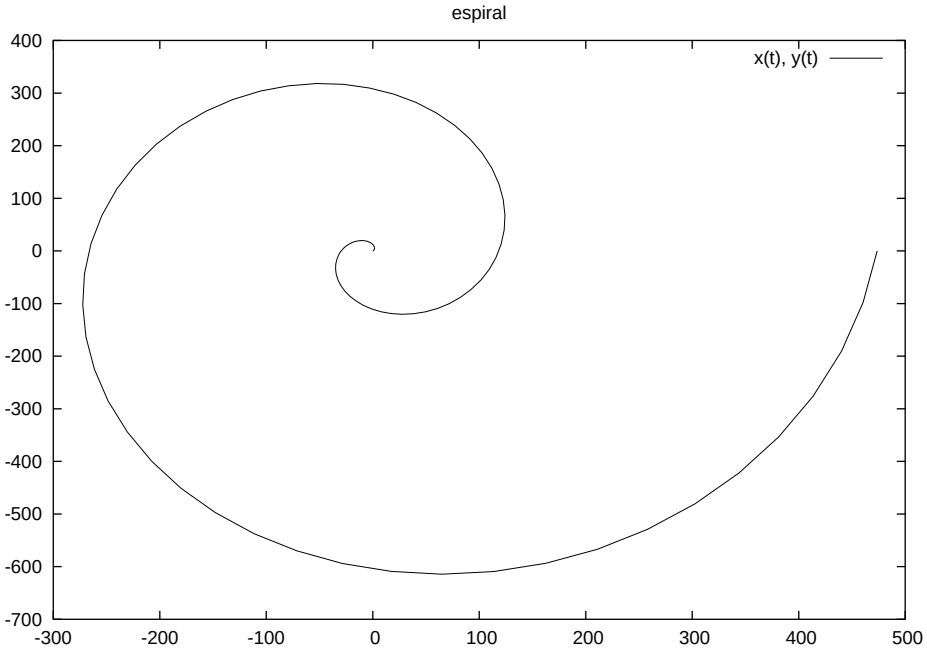
¿Qué pasa si en lugar de tener a las variables x y y como variables independientes resultan ser dependientes de un parámetro? Veamos el caso del movimiento de un proyectil

$$x(t) = x_0 + v_{0x}t \tag{2.1}$$

$$y(t) = y_0 + v_{0y}t + \frac{1}{2}at^2 \tag{2.2}$$

las coordenadas $x(t)$ e $y(t)$ son las componentes del vector de posición del objeto disparado. En cada instante de tiempo, es decir, para cada valor del parámetro t podemos evaluar los valores respectivos de las componentes de $\boldsymbol{r}(t)$. Lo que tenemos aquí es, por un lado, los valores explícitos de las componentes de $\boldsymbol{r}(t)$ para un tiro parabólico; por otro lado podemos verlo como el par de ecuaciones paramétricas que definen una curva (parábola en este caso) en el plano cartesiano. Para el caso del proyectil t resulta ser el tiempo: el caso más general es que t (o cualquier otro símbolo) es un *parámetro* del cual dependen nuestras variables.

```
GNUplot] set title "espiral"
         set parametric
         A(t)=3*t**2
         B(t)=5*t**2
         x(t)=A(t)*cos(t)
         y(t)=B(t)*sin(t)
         plot [0:4*pi] x(t),y(t)
```



2.4 Algunos ejercicios

1. Dado un conjunto de funciones elementales realizar operaciones aritméticas con ellas y hacer las gráficas correspondientes (antes de hacerlas en el graficador hay que imaginar cómo son: esto lo hace un buen ejercicio).
2.

Capítulo 3
Programación y algoritmos

Una cosa es hacer un programa elemental que efectúa un cálculo o, incluso, una lista de cálculos y otra cosa es desarrollar un programa en el que, por ejemplo, a partir de cierto modelo y/o información de entrada, se genere un conjunto de datos (en memoria, en archivos en disco, en acciones específicas vía *hardware*, etc.) y a partir de ellos se genere otro tipo de información (estadística, por ejemplo) y de ahí se pase a otra aplicación para darle un formato específico o para darle un tratamiento a la información,... etc. etc.

Se hace necesario definir claramente cuáles son las estructuras con las que contamos, en cualquier lenguaje de programación, porque a) nos permitirá identificar de manera rápida y sencilla las estructuras necesarias para la solución del problema específico y b) asociado con esto, nos permite construir algoritmos claramente *estructurados*.

3.1 Estructuras de programación

Podemos hablar de cuatro conceptos, de cuatro estructuras básicas, que nos permiten construir cualquier algoritmo computacional y, por los mismo, cualquier programa. Se vale mencionar aquí, además, que el lenguaje en el cual sea construído el programa, si bien es importante[3.1], es lo de menos: todos tienen implementadas las órdenes necesarias para poder trasladar un algoritmo, con las cuatro estructuras básicas, a un programa.

Cuando hicimos el primer programa para *Ceropio*, lo que hicimos fue una construcción en forma de *secuencia* de órdenes: desde el *inicio* del programa, pasando por la *lectura* de datos, las operaciones necesarias para efectuar la suma, la escritura del resultado y la orden de *terminación*. Es obvio aquí que la estructura básica que usamos para construir el programa fue la llamada *secuencia*. Sin embargo cuando ampliamos este ejemplo, modificando la orden de *terminación* por una orden de *salto incondicional* al punto en el que se leían los datos de entrada, construímos una estructura diferente: la *iteración*. Entonces teníamos ya un programa que efectúa repetidamente una tarea, pero que tiene el inconveniente de que no termina: sería algo así como un algoritmo (o programa) «inconcluso». Digamos que, de alguna forma, no tiene a dónde llegar. Con esto en mente construímos también una modificación adicional al mismo programa de suma de dos números: ahora el programa termina cuando la suma (o uno de los sumandos) sea cero. He ahí que hemos formulado una nueva estructura básica de cualquier algoritmo de programación: la *decisión* o *bifurcación* (el nombre lo dice todo: se elige un camino u otro).

La última estructura de programación es la llamada *modularidad*. Esta estructura tiene que ver con otra cuestión importante en el desarrollo de la solución computacional de un problema:

3.1. Los diferentes lenguajes de programación son concebidos para orientarlos a cierto tipo de aplicaciones (manejar bases de datos, efectuar cálculos de precisión, etc.) y bajo diferentes criterios (optimización en código o en rapidez, sencillez para programar, diferentes tipos y construcciones de datos, etc.), pero todos los programas se construyen usando las estructuras básicas.

3.2 Algoritmos

Una forma de organizar las cuatro estructuras de programación es colocándolas en diferente orden. Tal vez después de una estructura de decisión aparezca una estructura de iteración, el llamado a un subprograma o función, etc.[3.2] sin embargo hay otra forma de organizar estas estructuras y es incluyendo unas dentro de otras de tal modo que una iteración puede quedar dentro de una de las opciones de una estructura de decisión, una decisión dentro de una iteración que a su vez se encuentra dentro de otra y así. Cuando tenemos una estructura dentro de otra similar, por ejemplo un ciclo (o iteración) dentro de otro, decimos que tenemos (en este caso) *ciclos anidados*.

En esta sección vamos a desarrollar y analizar algunos algoritmos mínimos, pero suficientes para lo que como físicos requerimos.

3.2.1 ¿Qué hace por "dentro" *gnuplot*?

Ya hemos utilizado el graficador más de una vez y hemos encontrado diferentes situaciones: desde una gráfica que podríamos calificar de exitosa (donde los rangos corresponeden, así como los títulos, etiquetas, tipo de coordenadas, etc.) hasta el simple mensaje de error.

Hay dos cuestiones básicas en la estructura interna de *gnuplot* y son, por un lado, la evaluación explícita[3.3] de funciones y por otro lado el mapeo de puntos de un subespacio de $\mathbb{R}^2$ en una región rectangular del monitor: en una ventana. Vamos a referirnos a cómo se evalúan las funciones ahí dentro generando una secuencia de pares ordenados:

Lo primero que requerimos un intervalo de definición de la(s) variable(s). Pensemos en una sola variable independiente x que se encuentra definida en el intervalo $[a, b]$:

El intervalo será dividido haciendo una partición uniforme de tamaño h de manera que el primer punto a evaluar sea a, luego $a+h$, $a+2h$, ... y así hasta $a+nh=b$. Hay que notar aquí que el primer punto es $a=a+0h$, es decir, tenemos en realidad $n+1$ puntos. Por esta razón si queremos evaluar n puntos del intervalo[3.4] dado tenemos que evaluar el tamaño de "paso" como

$$h=\frac{b-a}{n-1} \tag{3.1}$$

así que los valores que nuestra variable x irá tomando dentro del intervalo serán

$$x_0=a,\ \ x_1=x_0+h=a+h,\ \ x_2=x_0+2h,\ ...,\ \ x_{n-1}=x_0+(n-1)h=b \tag{3.2}$$

o bien

$$x_1=a,\ \ x_2=x_0+h=a+h,\ \ x_3=x_0+2h,\ ...,\ \ x_n=x_0+(n-1)h=b \tag{3.3}$$

La expresión (3.3) tiene la ventaja de que el índice de x_i va de 1 a n, pero los coeficientes de h tienen la forma $i-1$. Por el otro lado, la forma anterior (3.2) lleva los índices de 0 a $n-1$, pero los coeficientes de h coinciden con los índices de x_i.

Ahora bien, se trata de generar, por el momento, un conjunto de pares de puntos (que después serán representados en un gráfico) por ejemplo, para una función cualquiera $g(x)$. Vamos a construir el algoritmo que nos permita generar este conjunto:

- **algoritmo**

3.2. Obviamente la estructura que rige por encima de todas es la secuencia: los programas siempre serán ejecutados en *secuencia*.

3.3. *gnuplot* también hace gráficas a partir de archivos de datos, pero aquí nos referimos al caso de la evaluación interna de funciones.

3.4. El valor de n nos conviene poderlo definir de entrada. Las razones explícitas aparecen sobre todo con los métdos de integración numérica.

1. inicio
2. introducir a, b
3. introducir n
4. evaluar $h \leftarrow (b-a)/(n-1)$
5. $i \leftarrow 0$
6. repetir mientras $x_i < b$
   $x_i \leftarrow a + i\,h$
   $y_i \leftarrow g(x_i)$
   escribir x_i, y_i
   $i \leftarrow i + 1$
7. fin

- **programa en C++** Versión «simple»: aquí simplemente implementamos el programa correspondiente al algoritmo *sin comentarios, notas, etc.*

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

int i;
double xi,yi;
double g(double);

void main()
{
   cout << ''Valor de a:'' << end;
   cin >> a;
   cout << ''Valor de b:'' << end;
   cin >> b;
   cout << ''Número de puntos:'' << end;
   cin >> n;
   h = (b-a)/double(n-1);
   i = 0;
   do {
      xi = a + (double)i * h;
      yi = g(xi);
      cout << xi << `` `` << yi << endl;
      i = i + 1;
   } while (xi <= b);
}

double g(double algo)
{
   return sin(algo)*3.0 + exp(-algo);
}
```

- **otra vez programa en C++** Versión «completa»: la implementación del programa ahora es más clara (a pesar de que pueda verse más aparatosa) pues cuenta con todo tipo de indicaciones. Además presenta una estructura que separa cada bloque indicando qué se hace en esa parte (los comentarios que inician con //// sirven como guía únicamente; los otros deberían ser algo normal en cualquier programa)

```
// programa para generar un conjunto de puntos para evaluar una función
// en un intervalo dado [a,b]
//   fernando rojas r.
```

```
// Julio, 2005

#include <iostream>  //// libreria de E/S (entrada/salida)
#include <stdlib.h>  //// librria estándar de C y C++
#include <math.h>    //// librería de funciones matemáticas

// declaración de variables

int i;            // contador (subíndice de la variable x)
double xi,yi;     // variables para los pares de puntos
double g(double); // declaración de la función g(x) que bla bla

// ** funcion principal ** //

void main()
{
  // entrada de datos
   cout << ''Valor de a:'' << end;
   cin >> a;
   cout << ''Valor de b:'' << end;
   cin >> b;
   cout << ''Número de puntos:'' << end;
   cin >> n;

  // ciclo de evaluación de la función
   h= (b-a)/double(n-1);
   i=0;
   do {
      xi = a + (double)i * h;
      yi = g(xi);
      cout << xi << `` `` << yi << endl;
      i = i + 1;
   } while (xi <= b);
}

// la función g(x)  que bla bla...     //// aquí se implementa g(x)

double g(double algo)
{
   return sin(algo)*3.0 + exp(-algo);  //// devuelve  3 sen x + e^(-x)
}
```

Nota 3.1. Compilar. ¿Cómo hacemos para que este programa pueda ejecutarse?

Antes que nada necesitamos una aplicación que nos permita *editar* el programa que queremos ejecutar. Existen varias aplicaciones como *pico*, *nano, emacs*, *nedit*,

De la misma forma que hay aplicaciones como ***pico*** y *gnuplot*, existen aplicaciones cuya función consiste en leer un *programa fuente* (que se encuentra en un archivo de texto) y transportarlo (traducirlo) al lenguaje binario que, como sabemos ya, es el que la computadora puede interpretar. La aplicación que realiza esta función se llama **compilador** y la forma explícita de hacer esto es la siguiente:

- En *C++*:

```
g++ nombre_del_archivo_fuente.cpp  -o  nombre_del_archivo_ejecutable -lm
```

donde `g++` es el nombre del compilador C++, `-o` es un parámetro para definir qué nombre queremos darle al archivo ejecutable (si no ponemos esta parte el ejecutable se llamará, por defecto, `a.out`; la opción `-lm` sirve para indicar al compilador que debe incluir la librería matemática. Si el programa está escrito en lenguaje *C* el compilador se llama `gcc` y la extensión del archivo fuente será .c (de otro modo no lo reconocerá el compilador).

– En *Fortran*:

```
f77 nombre_del_archivo_fuente.cpp  -o  nombre_del_archivo_ejecutable -lm
```

los parámetros funcionan de la misma forma que en ***C*** o en *C++* .

Uno puede enviar los datos generados a un archivo manipulando, desde «dentro» del programa un archivo de texto que quedará en el disco duro:

- **Versión «simple» modificada para enviar datos a un archivo de texto**

```
#include <iostream>
#include <fstream>                          // 0.
#include <stdlib.h>
#include <math.h>

using namespace std;

int i;
double xi,yi;
double g(double);

void main()
{
   ofstream fout(''datos.dat'');            // 1.
   cout << ''Valor de a: '' << end;
   cin >> a;
   cout << ''Valor de b: '' << end;
   cin >> b;
   cout << ''Número de puntos: '' << end;
   cin >> n;
   h = (b-a)/double(n-1);
   i = 0;
   do {
      xi = a + (double)i * h;
      yi = g(xi);
      cout << xi << ``  `` << yi << endl;
      fout << xi << ``  `` << yi << endl;  // 2.
      i = i + 1;
   } while (xi <= b);
   fout.close();                            // 3.
}

double g(double algo)
{
   return sin(algo)*3.0 + exp(-algo);
}
```

Las líneas del programa que están numeradas son las necesarias para a) crear un archivo de texto en el disco duro, b) «usarlo» (escribir en él o bien leer de él) y finalmente c) asegurar que el conjunto de datos que se guardan temporalmente en un *buffer*[3.5] queden todos en el archivo. La línea con el comentario 0 incluye una librería («header» en C, C++) que utilizamos para manipular archivos en disco; en la línea 1 simultáneamente declaramos a *fout* como un «flujo» (*stream*: "chorro" o "haz") de salida en disco (*of*: output-file) y creamos el archivo "datos.dat" que a partir de este momento aparecerá en el directorio de archivos con tamaño 0 bytes[3.6]. La línea 2: es la que corresponde, para nuestro caso, en «usar» el archivo: escribirá sobre él con el mismo formato que, según la línea anterior a ésta, lo hará en el monitor, identificado por *cout* como la salida por defecto. La línea 3 asegura que los datos que pudieran quedar en el *buffer* vayan todos al archivo en disco y el archivo se declare cerrado.

Una posibilidad adicional, entre otras muchas, es que nuestro programa se encargue de hacer la tarea completa, es decir, para el ejemplo dado aquí, nuestro programa podría además de evaluar los pares de datos (puntos) para ser graficados, crear un *script* en un archivo de texto para *gnuplot* y también llamar al graficador para ver nuestros resultados. La idea es más o menos simple: hay que crear un archivo de texto adicional donde se guardará el conjunto de órdenes del graficador y, luego, llamar al graficador con el nombre de este archivo. Esto lo podemos hacer con un conjunto de líneas dentro del programa principal:

- **El mismo programa que evalúa puntos de una función $g(x)$. Ahora genera un *script* y hace un llamado al sistema operativo para ejecutar *gnuplot* (versión sin documentar):**

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

using namespace std;

int i;
double xi,yi;
double g(double);

void main()
{
  ofstream fout(''datos.dat'');
  cout << ''Valor de a: '' << end;
  cin >> a;
  cout << ''Valor de b: '' << end;
  cin >> b;
  cout << ''Número de puntos: '' << end;
  cin >> n;
  h = (b-a)/double(n-1);
  i = 0;
  do {
    xi = a + (double)i * h;
    yi = g(xi);
    cout << xi << ''  '' << yi << endl;
    fout << xi << ''  '' << yi << endl;
    i = i + 1;
  } while (xi <= b);
```

3.5. Una «región» en la RAM donde se guarda temporalmente la información antes de ser enviada al disco duro. La razón es técnica: la velocidad de acceso a RAM es mucho mayor que la de acceso al disco duro.

3.6. Puedes usar la orden `ls -la *.dat` por ejemplo, que muestra todos los atributos de los archivos.

```
    fout.close();

    // generación de script para gnuplot

    // esta línea declara script1 como el identificador de un archivo de salida en
    // disco: «datos.gp»
    // usando el constructor ofstream y, al mismo tiempo, crea el archivo
    ofstream script1 («datos.gp»);

    // estas líneas de texto corresponden al script que necesitamos para gnuplot
    script1 << "set title 'mi función'  " << endl;
    script1 << "set xlabel 'x'  " << endl;
    script1 << "set ylabel 'y'  " << endl;
    script1 << "set xrange [" << a << ":" << b << "]" << endl;
    script1 << "plot 'datos.dat'  " << endl;
    script1 << "pause -1 'presiona enter...'  " << endl;

    // aquí se asegura que el archivo se declare cerrado. La información del buffer será
    // puesta en el disco duro
    script1.close();

    // llamada al sistema: gnuplot + script

    system("gnuplot datos.gp");
}

double g(double lax)
{
   return sin(lax)*3.0 + exp(-lax);
}
```

Existe la posibilidad adicional de que desde un programa podamos construir gráficos e incluso de que podamos ver cómo se va dibujando. AGREGAR G2.H

3.3 Estructuras de datos

Es decir que no todos los datos que se manejan dentro de un programa son *escalares*[3.7], también hay tipos de datos que poseen una estructura interna y que son, de forma general, los *arreglos* o *arrays* (útiles para manejo de listas, vectores, matrices, etc.), los *registros* o *estructuras* (que son tipos de datos más complicados porque en su estructura interna hay información de diferentes tipos), los *string* o *cadenas de caracteres* (que pueden verse como un arreglo o lista en la que cada lugar es ocupado por una letra, dígito o signo cualquiera). Hay otro tipo de datos como las *listas*, las *colas*, y los *árboles* que se aplican en diferentes áreas como por ejemplo en software que hace cálculo simbólico (Mathematica, Maple, etc.). Existen los *objetos* también y son estructuras más complejas que las anteriores pues, además de tener una estructura interna, tienen definidas acciones específicas: un avión (el objeto) vuela, aterriza, gira, etc. (las acciones) y tiene alas, combustible, timón, etc. (las propiedades, la estructura).

3.3.1 Arreglos (vectores y matrices)

3.7. Es decir, que tienen un valor simple entero, de punto flotante, caracter, booleano, etc.

La primera estructura útil que veremos por medio de un ejemplo consiste en un arreglo de valores tipo *double* en la cual vamos a mantener los valores generados x_i así como $y_i = g(x_i)$. Mientras el programa se está ejecutándose (*corriendo*) vamos a tener todos los valores generados en RAM (probablemente para obtener valores extremos o ceros de $g(x)$, promedios de las x_i o de los valores de $y_i = g(x_i)$, etc.).

De la misma manera que un vector en matemáticas es un arreglo de n números (componentes del vector) y que una matriz de $n \times m$ es un arreglo de $n \cdot m$ números acomodados en n filas y m columnas, en la mayoría de lenguajes de programación se contemplan estructuras de datos similares llamadas *arreglos*. La forma directa de manipular arreglos en casi cualquier lenguaje es por medio de los sunínidices.

Veamos el mismo caso de la evaluación de una función, pero ahora vamos a guardar los valores de x_i y de y_i en arreglos para evaluar, por ejemplo, sus promedios x_m, y_m (en el ejemplo mantenemos el "acceso" a un archivo en disco: ambas cosas se pueden hacer):

- **Ahora se trata de evaluar una función y de mantener todos los valores en RAM para efectos de evaluar promedios**

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

using namespace std;

#define  n  1000   // Definición de la constante  n.

double x[n],       // Dos arreglos tipo double de tamaño n cuyos
       y[n],       // índices irán de  0  a  n-1 (999).
       x_m = 0.0,  // Las variables x_m, y_m para los promedios
       y_m = 0.0;  // correspondientes.

double g(double);

void main()
{
   ofstream fout(''datos.dat'');
   cout << ''Valor de a: '' << end;
   cin >> a;
   cout << ''Valor de b: '' << end;
   cin >> b;

   // notar que n fue declarada más arriba como constante usando #define
   // y que i se declara ''dentro'' de la función principal main() (ver abajo)
   // lo cua implicai que i no será ''vista'' fuera del boloque de main()

   h = (b-a)/double(n-1);
   int i = 0;
   do {
      x[i] = a + (double)i * h;
      y[i] = g(x[i]);
      cout << x[i] << ``  `` << y[i] << endl;
      fout << x[i] << ``  `` << y[i] << endl;
      i++;
   } while (i < n);
```

```
   fout.close();

   // evaluación de promedios (x_m, y_m: se hicieron cero en la declaración):
   for (int i=0; i<n; i++)
   {
      x_m += x[i];
      y_m += y[i];
   }
   x_m /= double(n);
   y_m /= double(n);
   cout << ''Promedios: '' << x_m << '', '' << y_m << endl;

   // y las lista de valores...
   cout << ``de los valores:'' << endl;
   for (int i=0; 1<n; i++)
     cout << x[i] << ``  `` << y[i] << endl;
}

double g(double algo)
{
   return sin(algo)*3.0 + exp(-algo);
}
```

- •

Ejercicio 3.1. Modificar el programa anterior para que evalúa los mismos promedios sin utilizar arreglos.

Las matrices se pueden ver como arreglos de arreglos (vectores de vectores): De hecho se pueden declarar variables de este tipo que requieran más de 2 subíndices. Vamos a poner el ejemplo de la traza de una matriz de 3×3:

- **la traza de una matriz de 3×3**

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;

double a[3][3]= {1.0, 2.0, 3.5,
                 4.8, 2.4, -1.0,
                 1.2, 3.3, 9.9};

void main()
{
  cout << ``la traza de la matriz A='' << endl;
  for (int i=0; i<3; i++)
  {
    for (int j=0; j<3; j++)
      cout << a[i][j] << ``  ``;
    cout << endl;
  }

  double traza = 0.0;
  for (int i=0; i<3; i++)
     traza += a[i][i];
```

```
    cout << `` es: `` << traza << endl;
  }
```

- •

==========

- Expansión (contracción) a lo largo del eje X: $\begin{pmatrix} C & 0 \\ 0 & 1 \end{pmatrix}$
- Expansión (contracción) a lo largo del eje Y: $\begin{pmatrix} 1 & 0 \\ 0 & C \end{pmatrix}$
- Reflexión respecto a la recta $y = x$: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
- Reflexión respecto al eje X: $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
- Reflexión respecto al eje Y: $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$
- Corte a lo largo del eje X: $\begin{pmatrix} 1 & C \\ 0 & 1 \end{pmatrix}$
- Corte a lo largo del eje X: $\begin{pmatrix} 1 & 0 \\ C & 1 \end{pmatrix}$

Operaciones elementales (por renglones) con matrices

- $R_1 \to c\,R_1$: $\begin{pmatrix} c & 0 \\ 0 & 1 \end{pmatrix}$ opera así $\begin{pmatrix} c & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} cx & cy \\ z & w \end{pmatrix}$
- $R_2 \to c\,R_2$: $\begin{pmatrix} 1 & 0 \\ 0 & c \end{pmatrix}$ opera así $\begin{pmatrix} 1 & 0 \\ 0 & c \end{pmatrix}\begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} x & y \\ cz & cw \end{pmatrix}$
- $R_1 \to R_1 + c\,R_2$: $\begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix}$ opera así $\begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} x+cz & y+cw \\ z & w \end{pmatrix}$
- $R_2 \to R_2 + c\,R_1$: $\begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix}$ opera así $\begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix}\begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} x & y \\ z+cx & w+cy \end{pmatrix}$
- $R_1 \leftrightarrows c\,R_1$: $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ opera así $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} z & w \\ x & y \end{pmatrix}$

3.3.2 Otras estructuras

Existen otro tipo de estructuras algunas de las cuales reciben nombres diferentes dependiendo del lenguaje particular. La idea central de varias de estas estructuras es que no se trata de un valor *escalar* (de un solo tipo) ni tampoco de un *arreglo* de valores del mismo tipo (asociados por un nombre único y que se distinguen por el subíndice): se trata de un conjunto definido por un nombre único pero que, dentro de sí, guarda diferentes tipos de datos (esto incluye arreglos, escalares, apuntadores, etc.). Aquí, y para nuestros fines, nos vamos a referir primero a la estructura *struct*[3.8].

Un ejemplo relacionado con

3.3.3 Objetos

3.3.4 Los objetos se pueden pensar como una generalización de los registros (*struct* en C, C++) y además de contener datos de diferentes tipos tienen procedimientos y funciones formando parte de ellos.

3.8. En *Pascal*, por ejemplo, esta misma estructura se llama *registro*.

Capítulo 4
Sistemas dinámicos y métodos numéricos

4.1 Los sistemas dinámicos

Una forma[2] de estudiar la evolución de cualquier sistema (no necesariamente físico) es bajo la óptica de los llamados *Sistemas Dinámicos* (SD) y, una de las ideas importantes en este enfoque, es que uno pueda *ver*, de alguna manera la dinámica del sistema: cómo se comporta en general, para cualquier tiempo. Una de las implicaciones directas de esta idea es que en las ecuaciones que representan, precisamente, la dinámica del sistema, el tiempo no aparece y decimos entonces que es sistema es *autónomo*.

Por un lado las ecuaciones a las que estamos acostumbrados en el área de la física son ecuaciones diferenciales de segundo orden las cuales siempre, como se puede ver por ejemplo al pasar de la ecuación de Lagrange a las ecuaciones de Hamilton[4.1], se pueden transformar en ecuaciones diferenciales de primer orden haciendo las definiciones adecuadas. Para SD las ecuaciones son de la forma general

$$\begin{aligned} \dot{x_1} &= F_1(x_1, x_2, ..., x_m) \\ \dot{x_2} &= F_2(x_1, x_2, ..., x_m) \\ &... \\ \dot{x_m} &= F_m(x_1, x_2, ..., x_m) \end{aligned} \tag{4.1}$$

donde $\dot{x_i} \equiv dx_i/dt$ es la derivada temporal de x_i. Las cantidades x_i son las variables del sistema[4.2] y las funciones $F_{(i)}$ son cualquier función de todas las variables x_i. Esta es la forma general que asumen las ecuaciones de evolución en SD y -esto es importante- en ellas *el tiempo no aparece en forma explícita* sino solamente a través de la derivada[4.3]. En resumen tenemos, con este enfoque, la posibilidad de analizar la dinámica de un sistema que evoluciona en el tiempo a través de soluciones "fijas", estáticas, es decir, de imágenes o mapas que nos dicen mucho sobre el comportamiento del sistema. Las ecuaciones (4.1) corresponden a un SD de *dimensión m*.

Antes de entrar explícitamente a la solución numérica de la dinámica de algún sistema vamos a analizar qué es lo que pasa con un *flujo* representado en un *espacio fase ampliado*[4.4] y con una sola variable dinámica: pensemos en la expresión

$$\dot{x} = F(x) = x^3 \tag{4.2}$$

4.1. Esta idea se puede consultar en cualquier texto de Mecánica Teórica y consiste en pasar de una ecuación diferencial de segundo orden, que "vive" en el espacio de coordenadas (generalizadas) y sus derivadas temporales (velocidades generalizadas), a un par de ecuaciones diferenciales de primer orden que "viven" en el espacio de momentos y coordenadas.

4.2. Las variables del sistema son las cantidades medibles, del mismo, para las cuales podemos definir una ecuación que describa su evolución. En este sentido cada una de las x_i bien puede ser una coordenada, velocidad, población de un lugar o especie, concentración de una sustancia, costo de un producto, ingreso, presión arterial, etc. etc.

4.3. De hecho, cuando hay una dependencia explícita del tiempo, al escribir las ecuaciones de evolución de un sistema, se puede considerar a t como otra más de las variables $x_{n+1} = t$. Con esta definición $\dot{x}_{n+1} = f_{n+1}(x_1, x_2, ..., x_{n+1}) = 1$ sería una ecuación diferencial adicional del sistema que de esta forma se ha vuelto autónomo.

4.4. Le llamamos aquí espacio fase ampliado porque no coincide exactamente con la idea de espacio fase tradicional que consiste en un sistema de ejes perpendiculares asociados con los momentos y las coordenadas del sistema. En el enfoque de los SD los ejes están asociados a las variables x_i y sus derivadas temporales $\dot{x}_i$.

como un flujo en el que la velocidad *generalizada*[4.5] es proporcional a la variable x. El espacio fase de este flujo será representado en un sistema de ejes cartesianos asociados con x y con $\dot{x}$ respectivamente. La gráfica es una curva de la forma $y = x^3$. No olvidemos que el eje $''y''$ en este caso es la derivada temporal de x, así que podemos hacer un análisis cualitativo del flujo, digamos, *ver* su dinámica: cuando $x > 0$ lo que tenemos es, dada la expresión (4.2), que $\dot{x} > 0$ y esto significa que el movimiento del flujo es hacia la derecha, es decir, si la variable x es positiva entonces el flujo será de tal forma que x crece. De la misma forma, si $x < 0$ entonces la velocidad (la razón de crecimiento de x) será negativa: el flujo será de manera que x decrece, se mueve hacia la izquierda en el espacio fase.

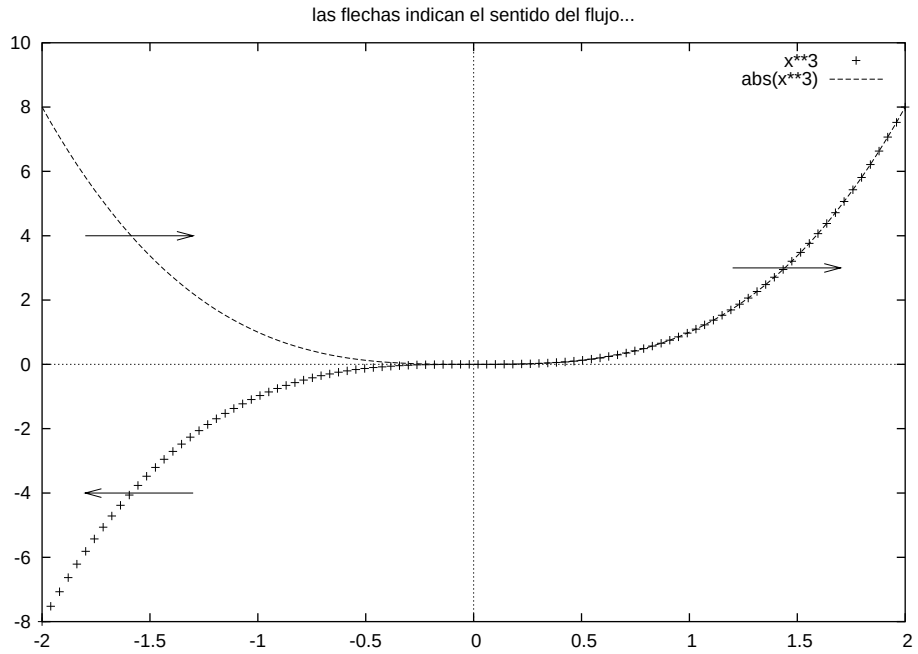
Hay un punto interesante que es el origen. ¿Qué pasa con el flujo ahí (cuando $x = 0$)? ¿Y qué pasa si nos desplazamos en el espacio fase un poquito a la derecha? ¿Y si lo hacemos a la izquierda? ¿El origen *atrae* o *repele* al flujo?

Decimos que los puntos en los que $\dot{x} = 0$ se llaman *puntos fijos* (PF) y, para el caso unidimensional simplemente son las raíces de la función $F(x)$. Obviamente en esos puntos el flujo no cambia (la variable x permanece constante), pero es interesante ver qué ocurre cerca del punto fijo $x = x^\star$ (donde $F(x^\star) = 0$), por ejemplo, para el caso (4.2) y para el caso

$$\dot{x} = \left| x^3 \right| \tag{4.3}$$

cuya única diferencia consiste en tomar el valor absoluto de x^3:

```
GNUplot] set zeroaxis
         set xrange [-2:2]
         set arrow 1 from -1.8,4 to -1.3,4
         set arrow 2 from 1.2,3 to 1.7,3
         set arrow 3 from -1.3,-4 to -1.8,-4
         set title "las flechas indican el sentido del flujo..."
         plot x**3 with points, abs(x**3)
```



La curva con línea punteada corresponde al flujo (4.2) y la de símbolos "+" al representado por (4.3). Ambos flujos tienen el punto fijo $x^\star = 0$, sin embargo el flujo (4.3), fuera de este punto, siempre es tal que la variable x crece ($\dot{x} > 0$). Podríamos decir, desde el punto de vista gráfico, que el flujo se mueve o se des-

4.5. Recordemos que hablamos de una *velocidad* de manera muy general, dado que x es una variable (cualquiera) del sistema.

plaza hacia la derecha. Si ponemos nuestro sistema en condiciones iniciales a la izquierda del punto fijo ($x < 0$) tendremos un efecto de *atracción* hacia $x^\star$. Por el contrario, si el punto inicial del SD es tal que $x > 0$, entonces el PF *repele* al flujo: se alejará de $x^\star$ hacia la derecha. En este caso decimos que se trata de un punto fijo *semiestable*: atrae por un lado y repele por el otro.

El caso (4.2) que discutimos arriba tiene el mismo PF $x^\star = 0$, pero en este caso se trata de un *repulsor* o de un PF *inestable*: no importa de qué lado del PF inicie el sistema siempre será repelido alejándose del PF.

Ejercicio 4.1. Analizar el comportamiento de los SD unidimensionales (flujos) siguientes

i. $\dot{x} = -x^3$

ii. $\dot{x} = x^2$

iii. $\dot{x} = r - x^2$, para $r < 0$, $r > 0$, ¿Qué pasa en la transición de valores de r? Construye una gráfica de $x^\star$ como función de r. ¿?

iv. $\dot{x} = r - \text{sen } x$. Analiza la dinámica (puntos fijos y estabilidad) para valores de r al rededor de 1. Construye la gráfica de $x^\star$ como función de r para $-\pi \leqslant x \leqslant \pi$. Este caso es peculiar: busca la solución analítica de la ecuación diferencial (por cualquier medio) para $x(t)$ (en el peor de los casos para $t(x)$). ¿La expresión te dice algo sobre la dinámica del flujo?

Ejercicio 4.2. Establece un criterio, a partir de los ejercicios, para saber si un PF es *atractor*, *repulsor*, o *semiestable*. (Tiene que ver con $dF(x)/dx$ en las cercanías del PF).

4.2 Osciladores, sistemas dinámicos y Euler

Vamos a pensar en el caso, primero, de un oscilador armónico unidimensional (sí, ya sabemos que ésas cosas aparecen por todos lados y por eso la traemos como ejemplo ahora). Como en todo sistema mecánico la ecuación de movimiento es diferencial de segundo orden:

$$m\ddot{x} = -kx \tag{4.4}$$

Se trata de una ecuación diferencial de segundo orden como muchas conocidas en la física. Pero si hacemos, por ejemplo

$$\begin{aligned} x_1 &= \dot{x} \\ x_2 &= \ddot{x} \end{aligned}$$

entonces podemos construir el sistema de ecuaciones de primer orden

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\frac{k}{m}x_1 = -\omega^2 x_1 \tag{4.5}$$

que son un par de ecuaciones de la forma (4.1). En este caso las dos funciones $F_1(x_1, x_2) = x_2$ y $F_2(x_1, x_2) = -\omega^2 x_1$ son lineales aunque, como veremos adelante, este no es el caso general.

Es más o menos obvio (si no lo parece analízalo un rato) que las soluciones $x_i(t)$ son funciones armónicas[4.6] de la forma general $A \,\text{sen}\, \omega t + B \cos \omega t$ y, a pesar de conocer las soluciones analíticas, vamos a implementar una solución aproximada para el sistema de ecuaciones diferenciales (4.3) bajo la idea de que en el caso más general uno podría tener al oscilador sometido a un conjunto de fuerzas externas como fricción, ruido, algún tipo de campo, etc.

→ **Método de Euler.**

- El problema consiste en tener una ecuación diferencial genérica de la forma

$$\frac{dw}{dv} = f(v, w) \tag{4.6}$$

4.6. Dejando las constantes por un lado, una función es la derivada de la otra y ésta, a su vez es "menos" la derivada de la primera.

y encontrar una solución aproximada usando técnicas numéricas[1]. Lo primero que podríamos hacer es, dado que $f(v,\ w)$ es una función conocida de las variables v y w, escribir una aproximación para la derivada de y en función de x:

$$\frac{dw}{dv} \cong \frac{w(v+h)-w(v)}{h} \tag{4.7}$$

donde h es una cantidad pequeña[4.7]. Resulta que tomando los lados derechos de las dos ecuaciones (4.6) y (4.7)

$$\frac{w(v+h)-w(v)}{h} = f(v,w). \tag{4.8}$$

- Por otro lado, como mencionamos antes, podemos usar una notación para los n valores de v dentro de un intervalo de interés $[a,b]$ con una partición uniforme de tamaño h:

$$v_0 = a,\ v_1 = a+h = v_0+h,\ \ldots,\ v_i = v_0+ih \tag{4.9}$$

donde $i = 0,\ 1,\ 2\ldots,\ n-1$ mientras que $h = (b-a)/(n-1)$. Podemos agregar la notación siguiente para fines de comodidad

$$f_i \equiv f(v_i, w_i) \tag{4.10}$$

y entonces la expresión (4.8) se transforma en

$$\boxed{w_{i+1} = w_i + h f_i} \tag{4.11}$$

que permite evauar el conjunto de puntos (v_k, w_k), solución numérica de la ecuación diferencial (4.6), a partir de la condición inical $(v_0,\ w_0)$. La expresión (4.11) es la **ecuación de Euler** para resolver este tipo de ecuaciones diferenciales.

$\rightarrow$ •

Con esta idea podemos escribir un algoritmo que permita resolver, mediante un programa, *una* ecuación diferencial del tipo (4.6):

- **Primer caso**: definimos el **intervalo de solución** de la ecuación ($[a,b] = [v_0, v_{n-1}]$)[4.8]

  1. introducir v_0 , w_0 (el punto (v_0,w_0) es la condición inicial)
  2. introducir v_{n-1} , h
  3. $v_i \leftarrow v_0$, $w_i \leftarrow w_0$
  4. escribir v_i , w_i
  5. repetir mientras $v_i <= v_{n-1}$
     $w_i \leftarrow w_i + h f(v_i, w_i)$
     $v_i \leftarrow v_i + h$
     escribir v_i , w_i
  5. fin

  Hay que notar que en este algoritmo no usamos un "contador": solamente la condición sobre los valores que va adquiriendo v_i.

- **Segundo caso**: definimos el **número de puntos** (iteraciones) n a realizar[4.9], así como el tamaño de "paso" h. El valor final del intervalo de solución es $v_0 + (n-1)h$:

  1. introducir v_0 , w_0
  2. introducir n , h
  3. $v_i \leftarrow v_0$, $w_i \leftarrow w_0$
  4. escribir v_0, w_0 (el punto inicial)
  4. repetir para i desde 1 a $n-1$ (los $n-1$ puntos restantes)

4.7. Tan pequeña como pueda manejar la computadora sin que se vuelva cero.

4.8. La notación x_{n-1} como final del intervalo es solamente con la idea de recordar que n puntos en el intervalo irán de 0 a $n-1$. La línea 5 del algoritmo puede sustituirse por un ciclo post-condicionado: `5. repetir hasta que` $v_i > v_{n-1}$.

4.9. Obviamente el *contador* (la variable i) puede tomar valores entre 1 y n. En este caso habría que renombrar $x_i = x_1 + (i-1)h$, es decir las x_i irán de x_1 a x_n. En este caso será conveniente, antes del ciclo, introducir una línea que mande a escribir x_0 y y_0 que son el punto inicial de nuestro problema.

$w_i \leftarrow w_i + h\, f(v_i, w_i)$
$v_i \leftarrow v_0 + i\, h$
escribir v_i , w_i

5. fin

En este algoritmo sí aparece de manera explícita un "contador" (la variable i) del cual depende explícitamente x_i.

- •

Ejercicio 4.3. ¿Es necesario, en los algoritmos anteriores, usar v_0 y v_i, o w_0 y w_i? La tarea es modificar ambos algoritmos de manera que el paso 3. sea omitido así como las variables v_i y w_i.

Ejercicio 4.4. Resolver analíticamente y hacer los programas (ambos casos) para resolver

i. $\frac{dy}{dx} = -x^2 y$

ii. $\frac{dP}{dt} = \csc P$

Probar con diferentes condiciones iniciales.

Ejercicio 4.5. Haciendo la expansión en serie de Taylor de $y(x+h)$ con respecto a x, demostrar que el **error en la expresión de Euler** *es del orden de* h (se escribe así: $\mathcal{O}(h)$). (Guía: de la expansión hay que despejar $y'(x)$ y luego eliminar los términos que «sobran» para obtener (4.7)).

En la solución del oscilador armónico debemos tener en cuenta dos cosas: a) que tenemos dos ecuaciones diferenciales que se deben resolver simultáneamente, así que el algoritmo debe ser modificado para tal efecto, y b) en la fórmula de Euler (4.11) la variable independiente no aparece explícitamente: lo hace a través de $f_i = f(v_i, w_i)$ y del incremento o "paso" h. Esto último nos permite encontrar las soluciones x_i del oscilador armónico (ya sea (4.5) u otro) *sin la variable independiente*. Esto es de particular interés en sistemas dinámicos pues nos referimos siempre al espacio fase de variables x y sus derivadas temporales.

Para hacer coincidir la notación con (4.1) vamos a reescribir la ecuación que define el método de Euler (4.11) para una de las variables x_i que depende de t:

$$\boxed{x_{i,n+1} = x_{i,n} + dt\, F_{i,n}} \tag{4.12}$$

en la que $x_{i,n} = x_{i,n}(t)$ es la variable del sistema (tal como fueron definidas en (4.1)) en la que el primer índice señala a la variable en sí (x_1, x_2, etc.). El segundo índice está relacionado con la aplicación del método: se refiere al punto «que sigue» $(n+1)$ en términos del ya conocido (n). Sustituimos aquí, solamente por claridad en los algoritmos, el «paso» h por dt, para señalar que la variable independiente es el tiempo t.

En la expresión (4.12), de manera general, $F_{i,n} = F_i(x_{1,n}, x_{2,n}, ..., x_{m,n}, \boldsymbol{t})$ pero, bajo el enfoque de los Sistemas Dinámicos, $F_{i,n} = F_i(x_{1,n}, x_{2,n}, ..., x_{m,n})$ *no depende explícitamente de* t, siguiendo también la notación de (4.1): una o más variables $x_i(t)$ cuya dependencia del tiempo aparece solamente de manera implícita a través de la derivada temporal y de sus relaciones con las otras variables (a través de la(s) función(es) del lado derecho[4.10]).

El algoritmo para la solución del oscilador armónico bajo la óptica de los sistemas dinámicos sería, tomando como ejemplo el segundo caso:

- **Segundo caso**: definimos el **número de puntos** (iteraciones) n a realizar y el tamaño de "paso"

1. introducir $x_{1,0}, x_{2,0}$
2. introducir n , dt
3. escribir $x_{1,0}, x_{2,0}$
4. repetir para i de 1 a $n-1$
   $x_{1,i} \leftarrow x_{1,0} + dt\, F_1(x_{1,0}, x_{2,0})$
   $x_{2,i} \leftarrow x_{2,0} + dt\, F_2(x_{1,0}, x_{2,0})$
   escribir $x_{1,i}$, $x_{2,i}$
   $x_{1,0} \leftarrow x_{1,i}$

4.10. En el caso general (4.10) la expresión del lado derecho es una función de *ambas variables* (dependiente e independiente).

$x_{2,0} \leftarrow x_{2,i}$

5. fin

En este caso asumimos que, como las condiciones iniciales son a $t_0 = 0$, no escribimos en valor inicial de t, en el punto 1. estamos definiendo, por lo tanto, las condiciones iniciales $(t_0,\ x_{1,0})$ y $(t_0, x_{2,0})$ solamente con los valores iniciales $x_{1,0}$ y $x_{2,0}$ o bien, en el caso general, el conjunto de las $\{x_{i,0}\}$.

- •

Hay que notar en el algoritmo anterior que después de la línea *escribir* $x_{1,i},\ x_{2,i}$ se actualizan los valores de $x_{1,0}$ y de $x_{2,0}$ con los nuevos obtenidos arriba ($x_{1,i}$ y $x_{2,i}$). En este caso se hace necesario hacerlo así, pero ¿Por qué? Bueno, la respuesta es la siguiente: en el momento de evaluar $x_{1,i}$ requerimos el valor de $x_{1,0}$ para evaluar f_1 y, en el momento de evaluar $x_{2,i}$ también requerimos el valor de $x_{1,0}$ para evaluar f_2. Por esta razón no podemos modificar directamente $x_{1,0}$ ni $x_{2,0}$ y usamos las variables auxiliares $x_{1,i}$ y $x_{2,i}$

Ejercicio 4.6. Escribir los algoritmos correspondientes a la solución de las ecuaciones diferenciales del oscilador armónico (4.5). Hacer los programas y las gráficas (obviamente la curva obtenida en el plano XY será una elipse[4.11]

Ejercicio 4.7. Modificar los programas para que generen un *script* para *gnuplot* que muestre la solución (poner títulos, etiquetas en los ejes, etc.). Hay que tener en cuenta que si se requieren los valores iniciales para mandarlos como valores al graficador deberán estar «guardados» en otras variables:

1. introducir $x_{1,\text{ini}}, x_{2,\text{ini}}$ // aquí guardamos "para siempre" los valores iniciales correspondientes a $t = 0$
2. introducir n , dt
3. $x_{1,0} \leftarrow x_{1,\text{ini}}$; $x_{2,0} \leftarrow x_{2,\text{ini}}$
4. escribir $x_{1,0}, x_{2,0}$
5. repetir para i de 1 a $n-1$
   $x_{1,i} \leftarrow x_{1,0} + dt F_1(x_{1,0}, x_{2,0})$
   $x_{2,i} \leftarrow x_{2,0} + dt F_2(x_{1,0}, x_{2,0})$
   escribir $x_{1,i}$, $x_{2,i}$
   $x_{1,0} \leftarrow x_{1,i}$
   $x_{2,0} \leftarrow x_{2,i}$
6. fin

Ahora sí podemos reutilizar los valores iniciales para enviarlos al graficador.

Ejercicio 4.8. Modificar el programa anterior para que genere un archivo de datos con 5 columnas: t, x_1, x_2, $\dot{x}_1$, $\dot{x}_2$. (Recordar aquí que $\dot{x}_1$ y $\dot{x}_2$ están dadas por f_1 y f_2). Hacer las gráficas de

i. $x_1(t)$, $x_2(t)$ ¿Oscilan? ¿Eso se esperaba?

ii. $\dot{x}_1(t)$ ¿Y ahora? ¿Esperabas lo que viste?

iii. x_1, x_2, t tridimensional, no importa el orden de las columnas.

iv. $x_1, x_2, \dot{x}_1$ lo mismo que en iii.

Tenemos que el método de Euler para resolver ecuaciones diferenciales del tipo (4.6) tiene un error del orden de h, es decir, el tamaño del «paso» (incremento en la variable independiente) determinará el orden del error que tendremos en la solución numérica de la ecuación. En todo caso uno buscará siempre en los métodos numéricos dos cosas principalmente: precisión (sobre todo) y sencillez. No siempre (o casi nunca) encuentra uno las dos cosas simultáneamente, pero siempre tiene la opción de elegir de acuerdo a las necesidades específicas, a las condiciones del problema y de la solución deseada.

→ **Método de Euler Modificado**.

- El método original de Euler consiste en usar una aproximación a la derivada $\frac{dx}{dt} \simeq \frac{x(t+dt)-x(t)}{dt}$. Sabemos (de un ejercicio anterior) que el error en esta aproximación es del mismo orden que dt. Vamos a mostrar aquí que la siguiente expresión da una mejor aproximación a la derivada que la original de Euler (4.11):

$$\frac{dw}{dv} \cong \frac{w(v+h) - w(v-h)}{2h} \tag{4.13}$$

4.11. Tal vez resulte conveniente revisar la parte de curvas paramétricas.

- Para demostrar que tiene un error de orden $\mathcal{O}(h^2)$ consideremos las siguientes dos expansiones en serie de Taylor

$$w(v+h) = w(v) + h\,w'(v) + \frac{h^2}{2!}w''(v) + \frac{h^3}{3!}w'''(v) + \frac{h^4}{4!}w^{(\mathrm{iv})}(v) + \frac{h^5}{5!}w^{(v)}(v)... \tag{4.14}$$

y

$$w(v-h) = w(v) - h\,w'(v) + \frac{h^2}{2!}w''(v) - \frac{h^3}{3!}w'''(v) + \frac{h^4}{4!}w^{(\mathrm{iv})}(v) - \frac{h^5}{5!}w^{(v)}(v)... \tag{4.15}$$

entonces hagamos la resta de (4.14) y (4.15):

$$w(v+h) - w(v-h) = 2h\,w'(v) + 2\frac{h^3}{3!}w'''(v) + 2\frac{h^5}{5!}w^{(v)}(v)...$$

y despejemos $w'(v)$:

$$\begin{aligned} w'(v) &= \frac{1}{2h}\left(w(v+h) - w(v-h) - 2\frac{h^3}{3!}w'''(v) - 2\frac{h^5}{5!}w^{(v)}(v)...\right) \\ &= \frac{w(v+h) - w(v-h)}{2\,h} - \frac{h^2}{3!}w'''(v) - \frac{h^4}{5!}w^{(v)}(v)... \\ &= \frac{w(v+h) - w(v-h)}{2\,h} + \mathcal{O}(h^2) \end{aligned}$$

donde el orden de error está en el primer término que hemos "cortado" de la serie de potencias de h (puesto que $h < 1$). Esto es, si en nuestro problema el valor de h es 0.01 el método de Euler nos dará un error del mismo orden que h, mientras que el método de Euler de segundo orden (o "modificado") tendrá un error del orden de 0.0001 lo cual es muy significativo.

- El algoritmo para poder implementar este método en un programa se construye de la misma forma que antes

$$\boxed{w_{n+2} = w_n + 2h\,f(v_{n+1}, w_{n+1})} \tag{4.16}$$

En este caso el costo de la precisión en el método consiste en el hecho de que, como puede verse en (4.16) el valor w_{n+2} requiere de los dos valores anteriores w_{n+1} y w_n, es decir que, para evaluar w_2 requerimos los dos valores anterioes w_0 y w_1.

→ •

Ejercicio 4.9. Mostrarse a sí mismo, de manera gráfica, que la aproximación de segundo orden para la derivada se "parece" más a la pendiente de la curva en el punto $(v,\ w(v))$. (Trazar en un sistema de ejes cartesianos la curva de una función (más o menos pronunciada para efectos visuales) y definir ahí los puntos $(v-h, w(v-h))$, $(v, w(v))$ y $(v+h, w(v+h))$, trazar la tangente en $(v, w(v))$ y comparar con la recta que une los otros dos puntos. Las rectas que unen el punto central con cada uno de los otros dos corresponden al método de primer orden, así que el método de orden 2 es como un promedio de las pendientes de estas dos rectas).

Ejercicio 4.10. Llegar explícitamente a la expresión del Método de Euler de segundo orden (4.16) a partir de (4.13) aplicado a la ecuación diferencial (4.6).

Para la solución de un SD escribimos la fórmula de Euler de segundo orden en la misma forma que (4.12):

$$x_{i,n+2} = x_{i,n} + 2dt\,F_{i,n+1} \tag{4.17}$$

donde, como antes, $F_{i,n+1}$ es el valor de la función F_i evaluada en el conjunto de valores $\{x_i\}_{i=1,...,m}$ en el paso $n+1$ que denotamos $\{x_{i,n+1}\}$.

Ahora tenemos el problema de obtener, después de la condición inical dada por todos los valores $\{x_{i,0}\}_{1=1,...,m}$ en $t=0$, un punto adicional (el conjunto de los $\{x_{i,1}\}_{i=1,...,m}$) para poder aplicar la relación recursiva (4.17). La solución consiste en usar el método de Euler de primer orden una sola vez en el algoritmo con lo que, dado el conjunto $\{x_{i,0}\}$, se puede obtener una aproximación para $\{x_{i,1}\}$ y, con estos dos conjuntos de valores, se puede obtener el conjunto $\{x_{i,2}\}$:

- **Euler de segundo orden**:

1. introducir $x_{1,0}, x_{2,0}, ...$ (las condiciones iniciales)
2. introducir n , dt (número de pasos y tamaño de paso)
3. escribir $x_{1,0}, x_{2,0}, ...$ (escribimos el "punto" inicial m-D)
4. $x_{i,1} \leftarrow x_{i,0} + dt\, F_{i,0}$ (método Euler de 1er orden para un punto[4.12])
5. escribir $x_{1,1}, x_{2,1}, ...$
4. repetir para i de 1 a $n-1$ (ahora sí, el de orden 2, para varias variables)
   $x_{1,2} \leftarrow x_{1,0} + 2dt\, F_1(\{x_{i,1}\})$
   $x_{2,2} \leftarrow x_{2,0} + 2dt\, F_2(\{x_{i,1}\})$
   ...
   escribir $x_{1,2}$, $x_{2,2}, ...$
   $x_{1,0} \leftarrow x_{1,1}$; $x_{1,1} \leftarrow x_{1,2}$
   $x_{2,0} \leftarrow x_{2,1}$; $x_{2,1} \leftarrow x_{2,2}$
   ...
5. fin

Los puntos suspensivos en términos de programación implicarían el uso de arreglos para no escribir líneas repetitivas. El segundo subíndice puede usarse para los nombres de tres arreglos `x0[v]`, `x1[v]` y `x2[v]`, y el primer subíndice sería el índice "normal" que se usa dentro del programa (`v`) y señalaría a cada una de las variables así como a las funciones F_i. Por ejemplo (el paso 4):

4. repetir para i de 1 a $n-1$ (i sigue corriendo sobre los pasos del prog.)
   repetir para v de 1 a m (la v se refiere a la variable del sistema)
   $x_{v,2} \leftarrow x_{v,0} + 2dt\, F_v(\{x_{v,1}\})$ (se evalúan todas las $x_{i,2} = x_{i,0} + 2dt\, F_{i,1}$)
   repetir para v de 1 a m
   escribir $x_{v,2}$ (se escriben todos los nuevos valores de las x_i)
   repetir para v de 1 a m
   $x_{v,0} \leftarrow x_{v,1}$ (se actualizan los dos puntos del método)
   $x_{v,1} \leftarrow x_{v,2}$

Hay que notar que tenemos aquí 3 ciclos "metidos" (*anidados*) en otra estructura iterativa: la variable i corre sobre el número de paso de la solución numérica, la variable v (que en el algoritmo es reutilizada) corre sobre todas las variables del SD cuyos valores, para efectos del método, están guardados en 3 arreglos (o vectores) de dimensión m.

• •

La función *Euler()* se ve algo aparatosa, así que requiuere calma. Vamos a exponer aquí un ejemplo que usa la aproximación de segundo orden para un SD de dimensión arbitraria. En esta función usamos un arreglo de apuntadores para el conjunto de funciones F_i que corresponden a cada variable dinámica x_i.

- **Solución de un SD de 2 dimensiones**

  El SD que vamos a resolver en el ejmplo es el siguiente (un oscilador con un término no lineal y un parámetro μ):

$$\begin{aligned} \dot{x_1} &= -x_2 + \mu x_1^2 \\ \dot{x_2} &= x_1 + \mu x_1^2 \end{aligned} \tag{4.18}$$

  así que en el programa (función *main()*) declaramos una constante que define la dimensión del SD, un vector donde se definen las condiciones iniciales $\{x_{i,0}\}$

4.12. En el caso de más de dos variables dinámicas este paso requiere, de preferencia, una iteración sobre un arreglo (vector) para las x_i.

- **La función *Euler()***
  Los arreglos dentro de la función fueron declarados de tamaño `dim+1` para dejar más claro el manejo de las variables dinámicas: de 1 a `dim` (y no de 0 a `dim-1`).

```
// Método de Euler 2o orden para resolver un sistema de
// ecuaciones diferenciales acopladas de orden dim: dx_i/dt=f({x_i})

void euler(double X[], int dim,     // vector {x_{i,0}} y tamaño (dimensión de SD)
           double h, int np,        // paso y número de puntos (o tiempo)
           double (*f[])(double *), // arreglo de funciones via apuntador array
           char *narch)             // archivo (cadena de caracteres)
{
  double x0[dim+1], x1[dim+1], x2[dim+1];
  ofstream out(narch);

  for (int i=1; i<=dim; i++) // condiciones iniciales al array local
  {                          // se guarda en x0[] y se envía a archivo
    x0[i] = X[i];
    out << x0[i] << ``  '';
  }
  out << endl;

  for (int j=1; j<=dim; j++) // segundo punto fase (Euler orden 1)
  {                          // se evalúa y se envía a archivo
    x1[j] = x0[j] + h * (*f[j])(x0);  // x0 es un vector!
    out << x1[j] << ``  '';
  }
  out << endl;

  for (int i=1; i<=dim; i++) // Euler orden 2: i corre sobre el paso
  {
     for (int j=1; j<=dim; j++)  // j corre sobre las variables dinámicas
     {
        x2[j] = x0[j] + 2.0 * h * (*f[j])(x1);  // x1 es un vector también
        out << x2[j] << ``  '';
     }
     out << endl;

     for (int j=1; j<=dim; j++)  // actualizacion {x0}<-{x1}; {x1}<-{x2}
     {
        x0[j] = x1[j];
        x1[j] = x2[j];
     }
  } // for Euler 2
  out.close();
}
```

- **y el programa principal:**
  En el programa principal también se dimensionaron los arreglos o vectores con tamaño `dim+1` para manejar los subíndices de 1 a `dim`

```
/*
    Solucion de un SD de dimension variable
    usando el metodo de Euler modificado
```

```
    F. Rojas
    2005
*/

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

using namespace std;

#define DIM 2         // la dimension del SD

double mu = 0.5;

double f(double x[]); // notar que los parámetros son arreglos
double g(double x[]);

int main()
{
  double (*laf[DIM+1])(double *); // arreglo de apuntadores a funciones

  double X0[DIM+1]; // vector (arreglo) de condiciones iniciales

  double dt = 0.001;  // paso
  int puntos = 15000;

  // asignamiento de funciones
  laf[1] = f;
  laf[2] = g;

  // condiciones iniciales
  X0[1] = 0.1;
  X0[2] = -0.2;
  euler(X0,DIM,dt,puntos,laf,"eulerDIM1.dat");
}

double f(double x[])
{ return -x[2] + mu*x[1]*x[1]; }

double g(double x[])
{ return x[1] + mu*x[1]*x[1]; }
```

• •

Las dos funciones *euler()* (versión con arreglos y sin arreglos) pueden coxistir simultáneamente y con el mismo nombre. Esta es una característica de C++ y se denomina *sobrecarga de funciones*. La única condición que se tiene para tener dos funciones con el mismo identificador (o nombre) es que el conjunto de parámetros sea diferente: es la forma de identificar una "versión" de otra.

La solución del SD del ejemplo anterior se muestra abajo: para diferentes condiciones iniciales el oscilador tiende hacia un *atractor* que es donde se mantendrá oscilando para tiempos largos.

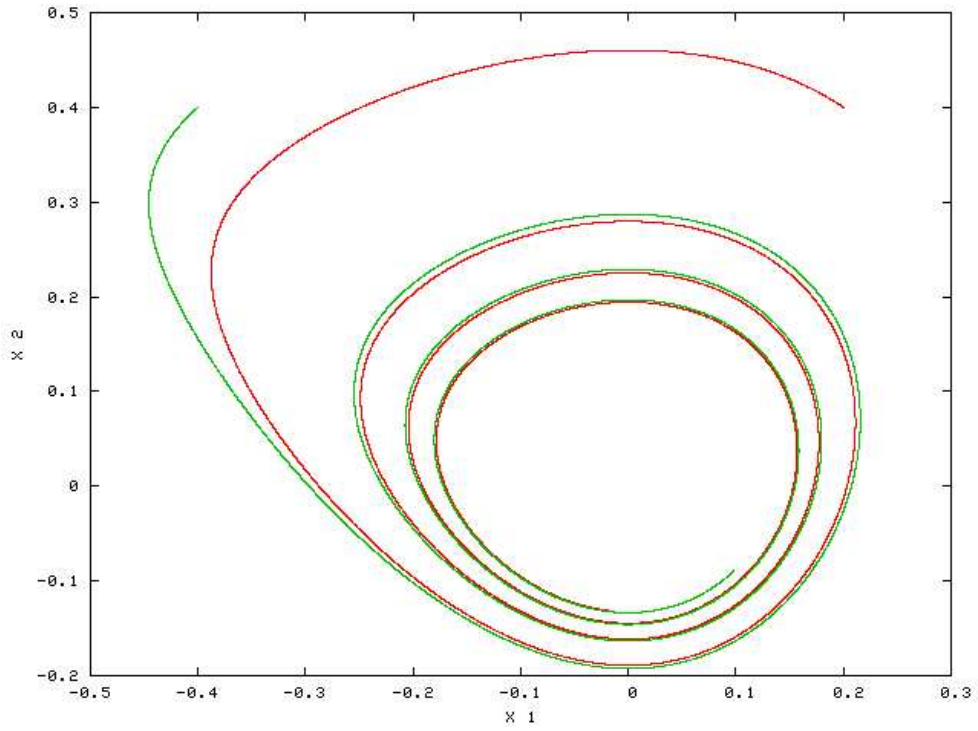


Figura 4.1. Condiciones iniciales: (0.2, 0.4) y (-0.4, 0.4), $\mu = 1.5$. El oscilador no lineal (•) muestra un *atractor* bidimensional semi elíptico.

Un método importante (aunque un poco más elaborado) es el de Runge-Kutta. No vamos a exponer toda le deducción aquí, pero la idea es la siguiente:

→ **Método de Runge-Kutta de cuarto orden**[4.13]

- Una forma alterna de modificar el método de Euler consiste en tomar una *primera aproximación* a x_{n+1} usando la forma convencional (4.11) o, usando la formulación de SD (donde sabemos que la variable independiente t no aparece explícitamente en las F_i), tendríamos la expresión (4.12). Pensemos en esta expresión como *esa primera aproximación* para un SD de orden uno (un flujo)

$$\tilde{x}_{n+1} = x_n + dt\, F(x_n)$$

una nueva aproximación (también de segundo orden) consiste en tomar un "promedio" de la pendiente entre $F(x_n)$ y $F(\tilde{x}_{n+1})$:

$$x_{n+1} = x_n + \frac{dt}{2}[F(x_n) + F(\tilde{x}_{n+1})].$$

La idea, a diferencia del método de Euler modificado de *dos puntos* (ver (4.17)), es tener siempre x_{n+1} en términos de x_n.

- Si uno continúa con el procedimiento llega al algoritmo que aparece en cualquier texto de métodos numéricos:

$$\begin{aligned} k_1 &= dt\, F(x_n) \\ k_2 &= dt\, F(x_n + \frac{1}{2}k_1) \\ k_3 &= dt\, F(x_n + \frac{1}{2}k_2) \\ k_4 &= dt\, F(x_n + k_3) \end{aligned} \tag{4.19}$$

4.13. Obviamente hay aproximaciones de orden mayor, pero el costo de la precisión (como suele suceder) implica que el programa efectúe muchas más operaciones y llamadas a funciones.

y, finalmente

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{4.20}$$

- En la solución de un sistema de ecuaciones diferenciales acopladas (como es el caso de los SD) la solución se ve muy aparatosa pues hay que definir k_1, k_2, etc. para cada una de las variables x_i. La razón es que para evaluar el *punto* (del espacio *fase*) $\{x_{i,n+1}\}$ se requiere, además del *punto* $\{x_{i,n}\}$, todos los valores de todas las $k_{1,2,3\,y\,4}$. Lo recomendable sería, por ejemplo, definir arreglos (vectores) `k1[dim]`, k2[dim], k3[dim], k4[dim] donde `dim` es la dimensión del SD. En ese caso el algoritmo quedaría así

$$\begin{aligned} k_{i,1} &= dtF(\{x_{i,n}\}) \\ k_{i,2} &= dtF(\{x_{i,n} + \frac{1}{2}k_{i,1}\}) \\ k_{i,3} &= dtF(\{x_{i,n} + \frac{1}{2}k_{i,2}\}) \\ k_{i,4} &= dtF(\{x_{i,n} + k_{i,3}\}) \end{aligned} \tag{4.21}$$

y entonces, para cada variable dinámica x_i

$$x_{i,n+1} = x_{i,n} + \frac{1}{6}(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4}) \tag{4.22}$$

$\rightarrow$ •

Ejercicio 4.11. Sería un buen y útil ejercicio resolver alguno de los problemas ya hechos sobre osciladores usando este algoritmo.

Ejercicio 4.12. Y más útil resultaría, tomando como modelo la función *Euler()*, construir la propia función *Runge_ Kutta()* para un número arbitrario de ecuaciones diferenciales simultáneas.

4.2.1 Otra vez los *flujos*: puntos fijos y raíces de funciones

Vamos a regresar a los *flujos* (SD unidimensionales) y a suponer que tenemos un flujo en el cual no podemos evaluar los puntos fijos de manera analítica, por ejemplo podemos considerar el flujo ilustrativo

$$\dot{x} = x - \cos x \tag{4.23}$$

Es obvio que no podemos conocer los PF analíticamente. A primera vista lo único que podemos hacer es una estimación haciendo la gráfica de $F(x) = x - \cos x$ y viendo dónde corta al eje x (otra forma equivalente sería graficar x y $\cos x$ y ver en qué punto se cortan). Pero existen técnicas numéricas por medio de las cuales uno puede encontrar las raíces (o ceros) de una función. Estas técnicas varían en precisión, velocidad y aplicabilidad, pero para nuestro ejemplo podemos probar con una calculadora cualquiera: hay que ponerla en modo *radianes* (obviamente el argumento de cos en (4.23) no puede estar en grados) y escribir un número cualquiera. Entonces oprimir la tecla [cos] una y otra vez sin perder de vista el resultado que nos muestra. Lo que vamos a observar es que el resultado *siempre* se va a ir acercando al número 0.739... cambiando cada vez menos dígitos hasta que, en la pantalla de la calculadora, el número deja de cambiar. Cuando la cantidad "deja de cambiar" (allá dentro podría seguir cambiando dígitos, pero no lo vemos) estamos seguros de que el número mostrado y su *coseno* son el mismo, por lo menos con tantos dígitos como nuestra calculadora muestre. Es decir, si la pantalla nos puede mostrar 3 dígitos decimales, veremos que el valor 0.739 quedará fijo aunque, seguramente, allá dentro podrá seguir cambiando en fracciones más pequeñas que milésimas.

Vamos a reconstruir en forma algorítmica lo que hicimos en la calculadora: primero elegimos un número arbitrario x_0 como valor inicial. De ahí en adelante presionamos [cos] para evaluar x_1, luego repetimos para obtener x_2 y así sucesivamente, es decir que vamos a aplicar una fórmula recursiva como

$$x_{n+1} = \phi(x_n) \tag{4.24}$$

que, en nuestro caso particular sería $x_{n+1} = \cos x_n = \phi(x_n)$ ($x_1 = \cos x_0$, $x_2 = \cos x_1 = \cos(\cos(x_0))$, $x_3 = \cos x_2 = ...$). La expresión (4.24) es típica de los procedimientos iterativos en métodos numéricos. De hecho se puede comparar, por ejemplo, con el método de Euler (4.12).

El procedimiento se detiene cuando nos aburre ver el mismo número en la calculadora, cuando deja de cambiar, cuando x_{n+1} y $\cos x_n$ son -para fines prácticos, el mismo número, cuando hemos resuelto hasta donde nos permite la tecnología (o bien lo decidimos nosotros) la ecuación $x = \cos x$ (que es equivalente a $x - \cos x = 0$) con alguna precisión determinada. El método de *Punto Fijo* o de *Iteración Simple* es el siguiente:

→ **Método de *Iteración Simple* o de *Punto Fijo* para raíces de funciones**

- Dada la función cuya raíz se busca, se iguala a cero y se despeja una x para obtener una expresión de la forma (4.24). En el ejemplo teníamos $F(x) = x - \cos x$ y obtuvimos $x = \phi(x)$.
- Como no todos los métodos convergen de la misma forma (es decir, por seguridad) hay que dar un valor inicial, x_0, al procedimiento iterativo. Pero este primer valor debe estar cerca de la raíz verdadera para asegurar convergencia del método[4.14].
- Implementar la iteración definida por (4.24) y detener cuando $|x_{n+1} - x_n| < \text{TOL}$ donde TOL es una cantidad (TOLerancia) que define la distancia máxima aceptable entre x_{n+1} y x_n. Por ejemplo, si deseamos obtener una precisión de 6 dígitos decimales habrá que definir TOL como 1.0×10^{-6}.
- ¿Cuándo el método converge? Llamemos $x_0 = \alpha + \epsilon_0$ donde α es la raíz de la función y ϵ_0 el error en nuestra primera aproximación a α. Del mismo modo $x_n = \alpha + \epsilon_n$ para cada valor de x_n. Para la iteración propuesta el error debe disminuir (las x_n deben acercarse a α) $\Longleftrightarrow$ $e_0 > \epsilon_1 > \epsilon_2 ... > \epsilon_n ...$ así que de (4.24)

$$\begin{aligned} \alpha + \epsilon_{n+1} &= \phi(\alpha + \epsilon_n) \\ &= \phi(\alpha) + \epsilon_n \phi'(\alpha) + \epsilon_n^2 \phi''(\alpha) + ... \end{aligned} \tag{4.25}$$

pero como $\alpha = \phi(\alpha)$

$$\begin{aligned} \epsilon_{n+1} &= \epsilon_n \phi'(\alpha) + \epsilon_n^2 \phi''(\alpha) + ... \\ &= \epsilon_n \phi'(\alpha) + \mathcal{O}(\epsilon_n^2) \end{aligned} \tag{4.26}$$

así que, salvo un error del orden de ϵ_n^2 se cumple la relación

$$\epsilon_{n+1} = \epsilon_n \phi'(\alpha) \tag{4.27}$$

y, como el error debe ir decreciendo cuando n aumenta entonces debe cumplirse que $|\phi'(\alpha)| < 1$. Este es el criterio principal de convergencia del método.

- Aunque suele chocar con lo expuesto es bueno discutir lo que pasa en esta última expresión: en la ecuación (4.27) aparece α que *es una cantidad desconocida* a la cual nos estamos aproximando haciendo uso de algoritmos y técnicas. Uno acude siempre a la suposición de que la función en cuestión (ϕ en este caso) es bien portada y, entonces, uno asume que su derivada no cambia mucho si en kugar de α usamos un valor cercano, por ejemplo x_0.

→ •

Ejercicio 4.13. Encontrar las raíces de las siguientes funciones, buscando valores aproximados opr medio de gráficos:

i. $F(x) = x - e^x$

4.14. No es, de hecho, el único criterio para asegurar que un procedimiento *converge*. Adelante veremos otros criterios formalmente.

4.2.2 Puntos fijos, raíces de funciones y mapeos dinámicos: el caos

Además del uso de ecuaciones diferenciales para describir la dinámica de muchos sistemas, se puede hacer a través de los llamados *mapeos* que, formalmente, no son otra cosa que expresiones de la forma (4.24) pero que, además, tienen el sentido de representar la evolución en el tiempo (discreto) de algún sistema.

Un ejemplo sencillo (y trillado) es el modelo de crecimiento de población propuesto por Malthuss[4] (y que llevaba a conclusiones catastróficas sobre el futuro nuestro):

$$x_{n+1} = K x_n \tag{4.28}$$

donde K representa la razón de crecimiento de la población ($K = 1.27$ representa el 27% de crecimiento para el año siguiente). En este caso es obvio que la población del año "siguiente" depende de la del año anterior y con un factor de crecimiento, pero el modelo no considera el hecho de que la gente muere por diversas razones (enfermedades, accidentes, guerra, falta de recursos de toda índole, etc.). Parece obvio que, a menos que $K = 1$ el mapeo (4.28) no tiene puntos fijos: si $K < 1$ la población desaparece y si $K > 1$ entonces crecerá indefinidamente. Para efectos de cálculo se toma a x entre 0 y 1 de modo que el valor $x = 1$ representa el 100% de un valor de referencia escalado. La expresión (4.28) puede recordarnos, por su forma, la de crecimiento exponencial $dx/dt = Kx$.

En 1845 P. Verhulst propone la llamada *ecuación logística*[2][6] que originalmente es una corrección al modelo de Malthuss: la población no solamente crece, de hecho debe depender de la población actual, pero también de la "despoblación" que entra como el factor $1 - x$ en el modelo:

$$x_{n+1} = K x_n (1 - x_n) \tag{4.29}$$

El valor de K va de 0 a 4. La ecuación logística se ve bastante sencilla y predecible: es una relación cuadrática que "abre" hacia abajo, no más.

Bueno, vamos a experimentar con ella. Primero: usando un contador y un valor inicial x_0 evaluemos $x(t)$. Hay que notar aquí que el tiempo tiene unidades arbitrarias (por ejemplo, para una dinámica de población la unidad de tiempo puede ser un año, una generación, etc.), entonces el tiempo es simplemente el contador

- **Ecuación logística** (algoritmo pensado con subprograma o función)
  1. Leer x_0, n // x_0 debe estar entre 0 y 1; n del orden de 50 es suficiente
  2. para i de 0 a n
     escribir i, x_0 // i son unidades arbitraria de tiempo
     $x_0 \leftarrow F(x_0)$
  3. fin

  La función F(x):
  1. calcula r $\leftarrow$ Kx(1-x)
  2. devuelve r
  3. fin

  En este algoritmo la sentencias "escribir" está antes que la operación con el objeto de que sea escrito el valor inicial antes de hacer cualquier operación.

- •

Ejercicio 4.14. Hacer el programa para poder graficar $x(t)$ del mapeo logístico. (Lo que vas a observar es que para $K < 1$ la población (o la variable correspondiente, x) decae. Para K entre 1 y 3 el mapeo tiende a un PF y para $K > 3$...)

¿Cómo encontramos los PF de un mapeo? De la misma manera que ocurre en el método de Iteración Simple el PF se encuentra cuando x_{n+1} y $\phi(x_n)$ coinciden, es decir

$$x^\star = \phi(x^\star)$$

lo cual significa que nuestra variable x no se mueve para ese valor $x^\star$. En el caso del mapeo logístico (4.29) los PF se encuentran resolviendo la ecuación

$$x = Kx(1 - x)$$

que da una ecuación cuadrática

$$Kx^2 + (1-K)x = 0$$

que tiene un punto fijo obvio, $x_1^\star = 0$, y $x_2^\star = \frac{K-1}{K}$.

Ejercicio 4.15. Encontrar explícitamente los puntos fijos del mapeo logístico.

Ejercicio 4.16. Verificar con el programa que para $K = 2$ el punto fijo es $1/2$.

Si suponemos un mapeo cualquiera de la forma general[4.15]

$$x_{n+1} = F(x_n) \tag{4.30}$$

entonces, de manera análoga a lo que ocurre con la expresión (4.24), el criterio para que un *mapeo* como (4.30) tenga *un* PF es el mismo y está expresado en la relación que asegura la convergencia (4.27): $|F'(x^\star)| < 1$.

Ejercicio 4.17. Demostrar que el mapeo logístico tiene PF estables para $x^\star = \frac{K-1}{K}$ si $1 < K < 3$.

*** ¿Ya notaste que puedes usar la función que encuentra raíces por Iteración Simple para determinar el PF de un mapeo? (cuando existe, claro) ***

Habras notado ya que en los mapeos ocurre algo análogo a lo que pasa con las ecuaciones de flujos: si uno mueve los parámetros que aparecen ahí al rededor de valores críticos la dinámica del sistema cambia de manera radical.

el espacio fase del mapeo será x_{n+1} contra x_n

4.15. Llamamos $F(x)$ a la función que define el *mapeo* por consistencia con la notación de los SD. Recordemos que la función $\phi(x)$ viene en realidad de "despejar" x en la expresión $F(x) = 0$.

Capítulo 5
Dinámica, Fractales, Simulación

Este capítulo está dedicado a las estrructuras fractales y a su estrecha relación, por un lado, con las estructuras que la naturaleza crea y, por otro, con la dinámica de los sistemas. Como en otros capítulos en este también introducimos partes de programación y de estructuras de datos.

5.1 Los números aleatorios: volúmenes y áreas

Antes de pasar a la construcción de figuras fractales vamos a manejar un poco los numeros "aleatorios" que son generados por una función *random()*[5.1].

Para comenzar, cuando decimos "aleatorios" (entre comillas) lo decimos porque son generados vía un algoritmo, lo cual implica que hay una "receta" que los "produce". De hecho hay una gran cantidad de "recetas" de programación para generar secuencias de números aleatorios (de aquí en adelante le quitamos las comillas). En particular nosotros vamos a usar la función *random()* que genera, en cada llamada, un número seudo-aleatorio que proviene del anterior (generado en la anterior llamada). No nos vamos a meter con problemas de "aleatoriedad" aquí, pero más adelante, con el Método de Montecarlo [5], justificaremos el uso de estas secuencias de números. Por el momento vamos a utilizar sus propiedades.

La función *random()* en C++ genera números aleatorios de tipo *entero* (long int) en el intervalo de (0,RAND_MAX[5.2]). La función y la constante están definidas en *stdlib.h* que se incluye en los programas C++. En el ejemplo anterior se asume que hemos construído la función *urandom()* que devuelve un número tipo *double* de la forma (double)random()/(double)RAND_MAX. Esta función devuelve un número "aleatorio" *uniformemente distribuído*[5.3] en el intervalo [0,1).

- **La función *urandom()***

```
double urandom()
{
   return (double)random()/(double)RAND_MAX;
}
```

- •

Esta función (de preferencia) la vamos a incluir en el archivo *comosellame.h* (la biblioteca de funciones que estamos construyendo) de manera que puede ser llamada desde cualquier programa posteriormente.

5.1. Esta función o su equivalente aparece en cualquier lenguaje de programación y/o paquetería (hojas de cálculo, etc.).

5.2. Uno debe tener cuidado con la versión de g++ que está utilizando. Al parecer las versiones tienen definida esta constante como 32767. En todo caso uno puede incluir una línea `#define RAND_MAX 2147483647` en su programa o en su biblioteca de funciones *.h*: una forma de verificar el valor de la constante es escribiéndolo (en cualquier programa se puede agrgar la línea `cout << RAND_MAX << endl;`).

5.3. Números aleatorios Uniformemente Distribuídos en un intervalo [a,b] significa que, dados dos subuintervalos de [a,b] que sean del mismo tamaño, la probabilidad de que uno de estos números "caiga" en uno o en otro es la misma. Esto asegura que si "llenamos" un espacio con puntos generados así el espacio será llenado de manera uniforme.

La función *random()* requiere una *semilla* (seed) que utiliza para iniciar una secuencia. Cada valor diferente de la semilla sirve para iniciar una secuencia diferente de números aleatorios. La función *srandom(int)*. En el ejemplo que sigue vamos a "llenar" de puntos aleatorios una región rectangular (en el ejemplo no incluimos las declaraciones, *#include*, etc.):

- **Puntos aleatorios en una región rectangular $[a, b] \times [c, d]$**

```
void main()
{
  double x,y;
  ofstream ran(''random.dat'');
  srandom(time(NULL)); // iniciamos secuencia con semilla del reloj interno
  for (int i=1; i<10000; i++)
  {
    x = a + (b-a)*urandom();
    y = c + (d-c)*urandom();
    ran << x << ''  '' << y << endl;
  }
  ran.close();
}
```

- •

Nota 5.1. En el programa ejemplo anterior las expresiones para x y y se obtienen del siguiente razonamiento que, por cierto, es análogo a lo que se hace para parametrizar el segmento [a,b] a partir de un parámetro $t \in [0, 1]$: i) La función *random()* devuelve números entre 0 y 1 (al igual que t); ii) Para generar, por tanto, números entre 0 y d, hay que multiplicar *d*random()*. Aquí d es el tamaño del intervalo (del mismo modo que `(b-a)` o `(d-c)` en el programa) y iii) Finalmente, como los intervalos no siempre comienzan en 0, hay que sumar el extremo inferior del intervalo (en este caso `a` y `c` respectivamente):

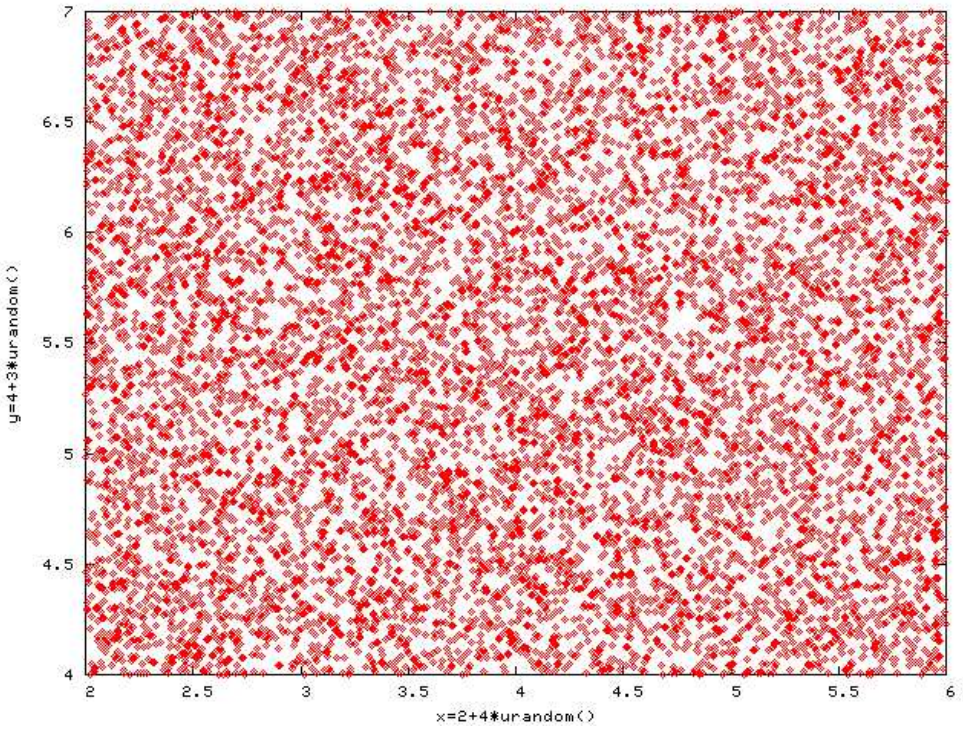


Figura 5.1. Puntos generados utilizando la función *urandom()* para el rectángulo mostrado: $[2, 6] \times [4, 7]$

Ejercicio 5.1. Modificar el programa para "llenar" un cubo $[a, b] \times [c, d] \times [e, f]$. Hacer la gráfica.

Ejercicio 5.2. Llenar con puntos aleatorios un volumen esférico de radio R. (Uno tiene que preguntar si la distancia del punto generado $\sqrt{x^2+y^2+z^2}$ es menor que R, si es así guardar el punto o dibujarlo).

Veamos ahora un problema más o menos simple como lo es medir el volumen de una piedra. Este problema es más o menos equivalente a calcular la integral de una función muy complicada. La solución consiste en tener un recipiente (donde quepa la piedra) con agua, meter la piedra en él y ver cuánto aumentó el volumen de agua que, finalmente, corresponde a la cantidad de agua que "cabe" en la piedra. De esta misma forma podemos evaluar el área de una figura si la encerramos en un rectángulo (cuya área A es conocida) y vemos "la cantidad de agua" que cae dentro de la figura con respecto al total de agua dentro del rectángulo. Veamos: si queremos el área de un círculo de radio R lo podemos colocar dentro del rectángulo $[-R, R] \times [-R, R]$ (o bien $[0, 2R] \times [0, 2R]$, o cualquier otro rectángulo que pueda contener a la "piedra", es al gusto). Si "dejamos caer" N puntos aleatorios uniformemente distribuídos el rectángulo sabemos que muchos de ellos "caerán" dentro del círculo y otros no. Si n puntos "caen" dentro del círculo (y $N - n$ fuera, obviamente) debe haber una proporción entre n y el área del círculo a igual a la proporción entre N y el área del rectángulo $A = 2R \times 2R = 4R^2$:

$$\frac{n}{a} = \frac{N}{A} \tag{5.1}$$

de donde se tiene que

$$a \approx \frac{n}{N} A = 4\,\frac{n}{N}\,R^2.$$

En el programa de ejemplo anterior simplemente hay que redefinir `a, b, c`, d adecuadamente y decidir cuándo un punto está dentro o no del círculo:

- **Puntos aleatorios para evaluar el área de un círculo** centrado en el origen

```
void main()
{
 double x,y;
 srandom(time(NULL)); // iniciamos secuencia con una semilla del reloj interno
                      // para asegurar secuencias diferentes cada vez
 n = 0;
 for (int i=1; i<=N; i++)  // asegurar que son N puntos
 {
   x = -R + 2.0*R*urandom();
   y = -R + 2.0*R*urandom();
   dist = sqrt(x*x + y*y);
   if (dist < R) n++;   // incrementa el contador de puntos dentro del círculo
 }
 area = 4.0*R*R * (double)n/(double)N;
 cout << '' area aproximada: '' << area << endl;
}
```

  Hay que tener en cuenta que hay variables, declaraciones, #include, etc. que no están escritos en el ejemplo.

- •

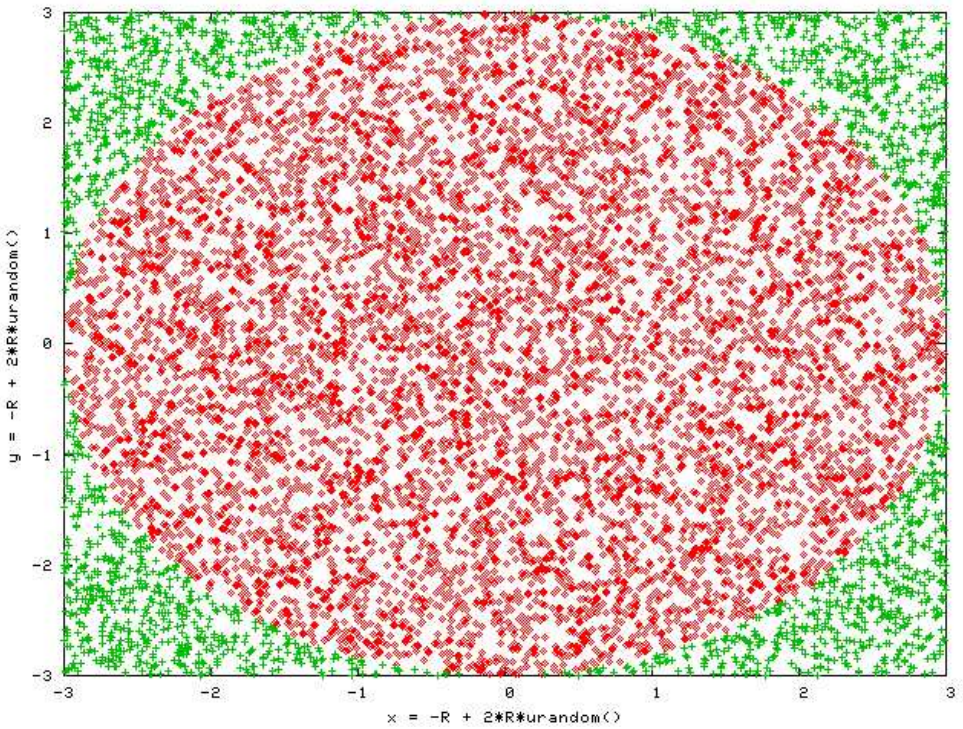


Figura 5.2. El mismo algoritmo del conteo de puntos "dentro" del círculo, seleccionando datos en diferentes archivos.

Ejercicio 5.3. Llenar con puntos aleatorios un espacio cúbico que contenga a una esfera de radio R. La distinción entre los que pertenecen a la esfera y los que no (están fuera) se hace midiendo la distancia al centro de la esfera (como en el ejemplo) y guardando los puntos que "caen" dentro de la esfera en un archivo y los que no en otro. Graficar ambos. (Debe verse una esfera dentro de un cubo).

Ejercicio 5.4. Mostrar, a partir de la relación (5.1) para el caso 3D, y escribiendo explícitamente los volúmenes a y A, que se puede hacer una estimación de π con la expresión $\pi \cong \frac{6\,n}{N}$.

Ejercicio 5.5. Modificar el programa del área del círculo para hacer una estimación de π a partir de una esfera dentro de un cubo. Ejecutarlo para diferentes valores de N y verificar que el error del cálculo es del orden de $1/\sqrt{N}$. En realidad esta estimación se puede hacer dentro del mismo programa comparando la constante M_PI (predefinida) con el valor obtenido para π, por ejemplo, si app_pi es el valor estimado por el programa y `dif=(M_PI-app_pi)`, debe cumplirse que $\frac{\text{dif}}{\sqrt{N}}$ es de orden $\mathcal{O}(1)$.

Nota 5.2. Esta técnica se usa sobre todo para evaluar integrales múltiples: uno "encierra" la fuunción a integrar en un hipercubo y genera números aleatorios contando, cada vez, si el punto está "dentro" o "fuera" de manera equivalente a lo que hicimos con la esfera.

Ejercicio 5.6. Evaluar la integral $\int_0^\pi \text{sen}\, x\; dx$ usando puntos aleatorios. (El resultado es 2 y puedes estimar el error asociado con el método de Montecarlo).

Ejercicio 5.7. ¿Podrías sugerir un algoritmo para evaluar una integral a partir de las sumas de Riemann?

5.2 Los fractales matemáticos: Sierpinski

A diferencia de la geometría de Euclides (líneas rectas, planos, círculos, etc.) la geometría fractal no se construye de manera directa (no hay reglas, compases, etc.) sino a través de *algoritmos*. Digamos que las formas primarias son conjuntos de procedimientos de transformación (rotación, traslación, deformación). Un fractal (una figura fractal) es resultado de un conjunto finito de estas transformaciones (llamadas transformaciones afines) aplicadas de manera recursiva sobre un mismo objeto original. Resulta impresionante lo que se puede lograr con un pequeño conjunto de estas transformaciones: la hoja de un helecho, la planta completa, el bosque, las montañas, y un conjunto innumerable de cosas más.

Hay una técnica alternativa para construir estos objetos y para la cual se utilizan, precisamente, los números aleatorios: basta con elegir un punto arbitrario en la figura y a partir de él los demás puntos son generados seleccionando aleatoriamente una de las transformaciones afines que dan lugar al fractal correspondiente [2][3][?].

Para ir calentando motores lo primero que vamos a hacer es construir uno de los fractales más conocidos: el triángulo de Sierpinski. A simple vista, siguiendo la secuencia de dibujos, puede verse cuál es la idea de la construcción. Se trata de lo que llamamos un algoritmo recursivo: se aplica a sí mismo una y otra vez:

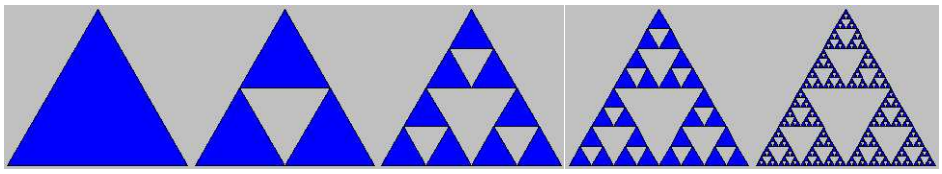


Figura 5.3. Aplicación de un algoritmo recursivamente: cortar los lados por la mitad y eliminar el triángulo central.

Hay un algoritmo para construir esta figura (no se trata directamente de la construcción del triángulo y la aplicación de la regla, sino de un algoritmo alternativo[5.4]):

- **Triángulo de Sierpinski** (el algoritmo es una síntesis de la idea de elegir aleatoriamente una transformación afín: esto se traduce en la selección al azar de un vértice)

  1. Tomamos tres vértices de un triángulo.
  2. Marcamos un punto cualquiera en el interior del triángulo.
  3. Elegimos un vértice del triángulo al azar, y marcamos el punto medio entre el vértice elegido y el punto marcado.
  4. Dibujamos el punto directamente (si es posible) o lo guardamos en un archivo y regresamos al paso 3 (tantas veces como se desee).

- •

Este algoritmo es más "literal" que otros que hemos escrito aquí (y no por eso menos útil). La razón es que vamos a introducir una estructura de datos, que no hemos usado antes, mediante un programa ejemplo:

- **Sierpinski en 2-D**

```
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include "numeric.h"

typedef struct    // Una estructura de datos (struct) usando typedef que permite
{                 // declarar tipos de datos al usuario.
  double x,y;     // Los elementos de la estructura (las coordenadas eneste caso).
} punto;          // El nombre que le damos a la estructura: punto.

int i,v,
    n_puntos = 20000;
```

5.4. En realidad uno puede hacer la construcción del triángulo de Sierpinski usando tranformaciones afines, por ejemplo: i) reducirlo a la mitad, ii) trasladar copias del resultado a la derecha y arriba hasta hacer coincidir sus vértices con los del triángulo original y iii) aplicar esto recursivamente.

```
punto V[3],p_i;   // Declaración del arreglo V de 3 elementos (los vértices) y el punto p_i
                  // ambas del tipo punto.
punto puntomedio(punto M,punto N);
main()
{
  // definición de los vértices del triángulo
  V[0].x = 0.0;
  V[0].y = 0.0;
  V[1].x = 2.0;
  V[1].y = 0.0;
  V[2].x = 4.0;
  V[2].y = 2.0;

  ofstream koch("sierp2D.dat");

  // un punto dentro del triángulo
  p_i.x = 0.5;
  p_i.y = 0.34;

  i = 1;
  do {
    koch << p_i.x << ''  '' << p_i.y <<endl;
    // urandom da un número aleatorio entre 0 y 1, así que v toma los valores 0, 1, 2:
    v = int(urandom()*3.0);
    p_i = puntomedio(p_i, V[v]);
  } while (i++ <= n_puntos);
  koch.close();
}

/* ____________
  La función puntomedio recibe dos argumentos del tipo punto (en los parámetros M y N) y
  devuelve un valor del mismo tipo (punto) que es el punto medio entre M y N
____________ */
punto puntomedio(punto M, punto N)
{
   punto m;
   m.x = (M.x+N.x)/2.0;
   m.y = (M.y+N.y)/2.0;
   return(m);
}
```

- •

Ejercicio 5.8. El ejercicio obvio: modificar todo lo necesario en el programa para hacer un triángulo de Sierpinski en tres dimensiones (tetrahedro de Sierpinski). (Hay que redefinir la estructura *punto* para 3-D y, de la misma manera, modificar la función *puntomedio()* para las 3 coordenadas. También hay que dar 4 vértices).

Nota 5.3.

5.3 Regreso a los Sistemas Dinámicos: Mapas de Poincaré y Caos

Apéndice A
Elementos de C++

Aunque el contenido de estas notas está orientado al lenguaje *C++*, hemos puesto en esta sección una tabla de equivalencias y de diferencias entre varios lenguajes de programación. Por un lado las órdenes equivalentes en lo referente a las estructuras básicas de programación, por otro los tipos de datos escalares que manejan cada uno de los lenguajes y los operadores principales.

A.0.1 Estructura básica

- **un programa básico**

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

void main()
{
  cout << ''¡Hola!'' << endl;
}
```

- **un equivalente con una función (subprograma) llamadoa desde *main()***

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

void print_hola()
{
  cout << ''¡Hola!'' << endl;
}

void main()
{
  print_hola();
}
```

A.0.2 Control de flujo

A.0.3 Funciones

A.0.3.1 Funciones tipo void y de otros tipos

A.0.3.2 Funciones con parámetros y apuntadores

A.0.4 Arreglos (vectores y matrices)

A.0.5 Clases (objetos)

A.0.6 Sobrecarga de funciones y operadores

Bibliografía

[1] Richard L. Burden and J. Douglas Faires. *Análisis Numérico*. International Thomson Editores, 6th edition, 1998.

[2] Gary William Flake. *The Computational Beauty of Nature*. The MIT Press, 2001.

[3] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals (New Frontiers of Science)*. Springer-Verlag, 1992.

[4] Moisés José Sametband. *Entre el Orden y el Caos: La Complejidad*. Fondo de Cultura Económica, 2a. edition, 1999.

[5] I. M. Sóbol. *Método de Montecarlo*. Editorial MIR, Moscú, 2nd. edition, 1983.

[6] Steven H. Strogatz. *Nonlinear Dynamics and Chaos*. Westview Press, 1994.

Índice de materias

error en la expresión de Euler 35
Euler 34