física computacional

(dinámica, geometría, caos)

FERNANDO ROJAS R.

Correo-e: fernandorojasr@gmail.com

18 febrero 2008

Aspectos:

- Una de las principales consideraciones en la elaboración de estas notas está relacionada con el gran éxito que el enfoque de los Sistemas Complejos ha tenido en el planeta. En este caso, solamente como una aclaración, el «éxito» tiene el sentido del logro: de la capacidad de establecer algunos criterios, ideas y principios que la naturaleza ha mostrado, que no son formales ni producto únicamente del pensamiento o de la razón, para poder describir eventos que ocurren a nuestro alrededor todo el tiempo.
- La complicación vino al decidir qué se pone primero y qué después: al final todo está concatenado con todo. Todos los aspectos de esta nueva visión del mundo y de la naturaleza están vinculados en más de un sentido y esta es la razón que no permite establecer un orden. Por lo menos no un orden convencional, no un orden simple de esos a los que estamos acostumbrados.
- El texto está orientado al uso de software libre, en particular al graficador `gnuplot` y al lenguaje de programación `python`. Éste último por muchas razones:
    i. no requiere experiencia en programación: sus estructuras son simples de manejar
    ii. es un lenguaje orientado a objetos y a la construcción de módulos que se pueden agregar en cualquier momento a otras aplicaciones
    iii. tiene una gama enorme de recursos libres en la red (además de ser libre)
    iv. permite construir aplicaciones interactivas (con GUI) de manera muy sencilla y rápida
    v. permite (de hecho lo hacen librerías como `numpy` o `scipy`) en caso necesario, hacer ligas con funciones en `C++` o en `fortran`

Índice

1 Oscilaciones en la naturaleza 7

1.1 Introducción 7
1.2 Dinámica en una dimensión: Espacio Fase y Puntos Fijos . . 8
1.3 Solución de Euler de Ecuaciones Diferenciales de Primer Orden 10
1.4 Sistemas Dinámicos de dos o más dimensiones 16
1.5 Mayor precisión: Runge-Kutta 24
1.6 Los modelos del tipo Lotka-Volterra 26
1.6.1 Series de Tiempo 27
1.7 Puntos Fijos, Raíces... y errores 27
1.8 Discusión 31
1.9 Ejercicios 32
1.9.1 de afirmación 32
1.9.2 de aplicación 35

2 Dinámica Discreta 37

2.1 Introducción 37
2.2 Mapeos 38
2.3 Autómatas Celulares 42
2.3.1 Ecuación de Difusión 44
2.4 Ejercicios 45

3 Azar y Determinismo 47

3.1 Introducción 47
3.2 Números seudo-aleatorios 47
3.3 Estimaciones simples usando `random()` 47
3.4 Estructuras Recursivas y Fractales 52
3.5 Movimiento Browniano y Difusión 52
3.6 Procesos Estocásticos. Montecarlo 53
3.7 Distribuciones Teóricas y Reales 55
3.8 Ejercicios 59

4 Geometría de la Naturaleza 59

4.1 Introducción 61
4.2 Estructuras Recursivas 61
4.3 Sistemas de Reacción-Difusión 64
4.4 Formación de Patrones 65
4.5 Ejercicios 66

Apéndice A Elementos de `gnuplot` 66

A.1 Implementación directa (interactiva) de órdenes en `gnuplot` 77
A.2 Modo programado de `gnuplot` 77

Apéndice B Programación `python` 81

B.1 Implementación interactiva de tareas en `python` 85
B.2 Modo programado de `python` 85
B.3 Estructuras de programación 87
B.4 Algoritmos 88
B.4.1 ¿Qué hace por "dentro" *gnuplot*? 91
B.5 Estructuras de datos 93
B.5.1 Arreglos (vectores y matrices) . . . . . ?

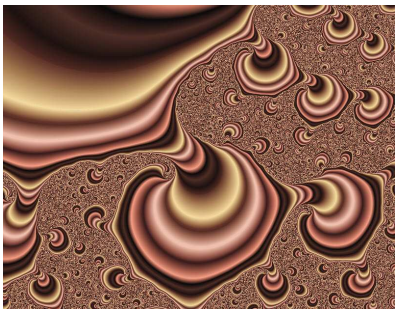
Índice de materias . . . . . ?

Bibliografía . . . . . ?

Capítulo 1

Oscilaciones en la naturaleza

En la naturaleza se combinan dos movimientos de manera continua: el devenir y el repetirse, el avanzar y el retroalimentarse. Esto tiene su analogía con las estructuras algorítmicas. El teorema de KAM implica que los sistemas se quedan en un toroide en su espacio fase: girando de alguna forma...



1.1 Introducción

Sobre todo en las formulaciones de Lagrange y Hamilton de la Mecánica Teórica se emplea el concepto de *espacio fase* que es, básicamente, un espacio conveniente formado por tantos ejes ortogonales entre sí como variables utiliza la correspondiente formulación.

El concepto a nivel de teoría se extiende a áreas como la Mecánica Cuántica o la Mecánica Estadística y permite modelar, dentro de sus propios marcos, sistemas de dimensiones atómicas y

sistemas conformados por muchas partículas. La visión, dentro de este material, se extiende a sistemas que en general están fuera de equilibrio, como los sistemas vivos y que, en general, no requieren de una formulación teórica para poder ser descritos o modelados.

De la misma manera que en la Mecánica Teórica, aquí vamos a hacer uso del llamado *espacio fase* y lo vamos a construir de manera análoga. La diferencia real es el enfoque asociado con los *Sistemas Dinámicos* (**SD** a partir de ahora) bajo el cual las variables que determinan la dinámica del sistema en cuestión no son necesariamente variables asociadas con alguna formulación de tipo teórico. En adelante veremos la utilidad que tiene un análisis de la dinámica en este espacio. El concepto de *Punto Fijo* (**PF**) es importante también pues permite hacer análisis cualitativo de la dinámica correspondiente e incluso algunas predicciones sobre la misma.

1.2 Dinámica en una dimensión: Espacio Fase y Puntos Fijos

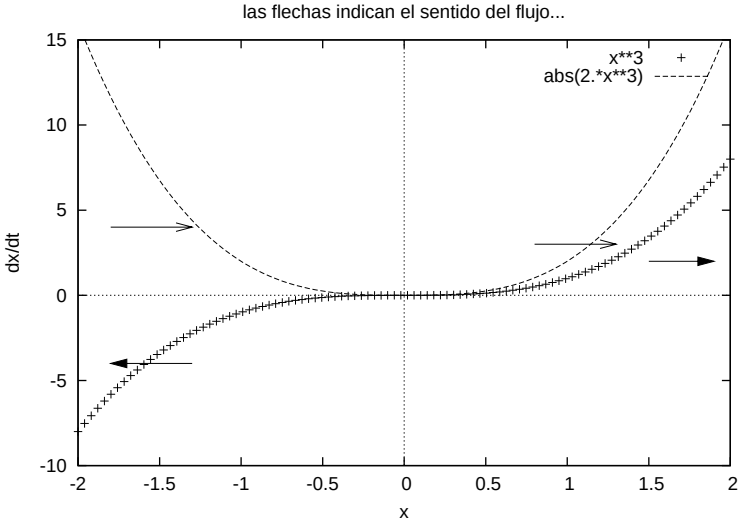
Vamso a comenzar pensando en dos dinámicas diferentes que se diferencian solamente en un signo. La imagen es el espacio fase y representa las variables x y sus derivadas temporales $\dot{x}$ en un sistema de ejes ortogonales. La dinámica en este enfoque de SD tiene la forma de las ecuaciones de Hamilton [] (ecuaciones diferenciales de primer orden en el tiempo) aunque hay que notar que las variables dinámicas en este caso no están asociadas necesariamente a coordenadas y momentos generalizados ni tampoco, como dijimos antes, a una formulación teórica.

Si lo que nos interesa en el momento es un SD en una sola dimensión lo que tendremos es una dinámica definida por una sola ecuación diferencial

$$\dot{x} = F(x) \tag{1.1}$$

```
This is a TeXmacs interface for GNUplot.
```

```
set zeroaxis
set size 0.8,0.8
set arrow 1 from -1.8,4 to -1.3,4
set arrow 2 from 0.8,3 to 1.3,3
set arrow 3 from -1.3,-4 to -1.8,-4 filled
set arrow 4 from 1.5,2 to 1.9,2 filled
set title "las flechas indican el sentido del flujo..."
set xlabel "x"
set ylabel "dx/dt"
plot [-2:2][-10:15]x**3 with points, abs(2.*x**3)
```



La gráfica nos muestra el espacio fase de dos sistemas dinámicos parecidos, pero diferentes:

$$\dot{x} = x^3 \tag{1.2}$$

y

$$\dot{x} = 2|x^3| \tag{1.3}$$

donde introdujimos el factor 2 simplemente para que no quedaran sobrepuestas las curvas. Como la función x^3 es impar tiene una rama hacia arriba y la otra, de forma simétrica, hacia abajo (la curva con '+'). En otro caso corresponde a la función $|x^3|$ que, al perder el signo, se vuelve simétrica respecto del eje y (la curva punteada).

La discusión sobre la dinámica del sistema representado ahí solamente depende de las condiciones iniciales elegidas. Tomemos por ejemplo el caso expresado en (1.3): ambas ramas de la curva están hacia arriba por lo que $\dot{x} > 0$ siempre, excepto en un punto: el origen. Es en este punto el único sitio del espacio fase donde $\dot{x} = 0$, donde x no varía. Este punto se llama *Punto Fijo* (**PF**) del sistema. Ahora bien, si elegimos arbitrariamente cualquier punto de la curva asociada a la expresión (1.3) como condición inicial de nuestra dinámica y ésta se encuentra determinada de antemano por la misma expresión, el único resultado viable que podemos obtener es que el sistema evoluciona de manera que su variable x *siempre* crece. Es decir, donde quiera que nos coloquemos sobre la curva en el espacio fase, siempre tendremos que $\dot{x} > 0$ y por lo tanto x crece: decimos que *el flujo se mueve hacia al derecha*. Y claro: hay un punto ahí en el cual esto no ocurre: si elegimos el origen de corrdenadas como condición inicial de nuestra dinámica el sistema permanecerá por siempre ahí, sin moverse, sin cambios (por lo menos sin cambios en la variable x pues $\dot{x} = 0$).

El mismo análisis puede hacerse directamente para el SD expresado en (1.2): la diferencia ahora será que si elegimos el punto de inicio a la derecha del **PF** el sistema evolucionará de forma que x crece (*el flujo se mueve a la derecha*) y, en caso contrario, si elegimos una condición inici a la izquierda del PF el *flujo se moverá a la izquierda*, de manera que como $\dot{x} < 0$ entonces x tiende a decrecer.

Nota 1.1. Hay que mantener en mente esta idea: la dinámica, como se vio y se verá en otros ejemplos, depende de las condiciones iniciales.

El problema se complica por lo tanto si por un lado complicamos la expresión que defina el flujo (el SD) o si agregamos variables al sistema. En la realidad ambos son muy comunes como puede apreciarse en los ejercicios anteriores: la solución analítica se va complicando.

1.3 Solución de Euler de Ecuaciones Diferenciales de Primer Orden

Analicemos primero algunos aspectos importantes en la solución de SD de cualquier dimensión (número de variables): una primera idea

es el hecho de que tenemos que resolver ecuaciones diferenciales de primer orden y, en general, no lineales.

La forma más usual de pensar en la solución de un problema dinámico es buscando una función que nos diga cómo depende la variable dinámica explícitamente del tiempo. Antes hemos mencionado que hay formas diferentes de ver el problema y que algunas, como es el caso del enfoque de los SD, nos proporcionan información adicional cualitativa sobre el sistema. Por el momento veremos una técnica numérica que nos permite aproximar, en general, la solución de una ecuación diferencial de primer orden:

→ **Método de Euler.**

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

$$\frac{dw}{dv} = f(v, w) \tag{1.4}$$

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:

$$\frac{dw}{dv} \cong \frac{w(v+h) - w(v)}{h} \tag{1.5}$$

donde h es una cantidad pequeña[1.1]. Resulta que tomando los lados derechos de las dos ecuaciones (1.4) y (1.5)

$$\frac{w(v+h) - w(v)}{h} = f(v, w). \tag{1.6}$$

- 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:

$$v_0 = a,\ v_1 = a + h = v_0 + h,\ ...,\ v_i = v_0 + i\,h \tag{1.7}$$

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

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

$$f_i \equiv f(v_i, w_i) \tag{1.8}$$

y entonces la expresión (1.6) se transforma en

$$\boxed{w_{i+1} = w_i + h\, f_i} \tag{1.9}$$

que permite evaluar el conjunto de puntos (v_k, w_k), solución numérica de la ecuación diferencial (1.4), a partir de la condición inical (v_0, w_0). La expresión (1.9) es la **ecuación de Euler** para resolver este tipo de ecuaciones diferenciales.

→ •

Con esta idea podemos escribir un algoritmo que permita resolver, mediante un programa, *una* ecuación diferencial del tipo (1.4):

- **Primer caso**: definimos el **intervalo de solución** de la ecuación $([a, b] = [v_0, v_{n-1}])$[1.2]

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 $v_i <= v_{n-1}$
   $w_{i+1} \leftarrow w_i + h\, f(v_i, w_i)$
   $v_{i+1} \leftarrow v_i + h$
   $i \leftarrow i + 1$
   escribir v_i , w_i
6. fin

1.2. 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` $v_i > v_{n-1}$.

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[1.3], 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. escribir v_0, w_0 (el punto inicial)
  4. repetir para i desde 0 a $n-1$ (los $n-1$ puntos restantes)
     $w_{i+1} \leftarrow w_i + h\, f(v_i, w_i)$
     $v_{i+1} \leftarrow v_i + h$
     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.

- •

Ejemplo 1.2. El programa python correspondiente al *segundo caso* usando listas para los valores de v y w:

```
v_o, w_o = 2.3, 4.2   # información de entrada
n, h = 1000, 0.002    # puntos y precisión

# la función de lado derecho de la ec. dif.
def F(v,w):
   return -v*w**2

v = [v_0 + h*k for k in range(n)] # generar listas o vectores
w = [0 for k in range(n)] # w es una lista vacía (con ceros)

w[0] = w_0                # condición inicial
print v[0], w[0]          # a los vectores
```

1.3. 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.

```
for i in range(n-1):      # ciclo principal
   w[i+1] = w[i] + h*F(v[i], w[i])
   print v[i+1], w[i+1]
```

Nota que a) en las dos primeras líneas del programa se hacen asiganciones dobles de valores: `v_o, w_o = 2.3, 4.2` indica que `v_0` tomará el valor `2.3` del mismo modo que `w_0` tendrá el valor `4.2`; b) el vector `v` ya contiene todos los valores de la variable independiente desde el momento de construirlo; c) con `w` no se puede hacer lo mismo pues cada valor en una de las celdas depende del valor en la celda anterior; d) ¿Por qué aparece `print v[i+1],w[i+1]` en la última línea en lugar de, por ejemplo `print v[i],w[i]`?

Ejemplo 1.3. El mismo ejemplo anterior con variantes: se usa lectura de datos (el usuario los da), no se utilizan vectores (listas) y se genera un archivo de datos (en itálicas):

```
v_0 = input(``v inicial: '')
w_0 = input(``w inicial: ``)

n, h = 1000, 0.002    # puntos y precisión

# la función de lado derecho de la ec. dif.
def F(v,w):
   return -v*w**2

print v_0, w_0        # valor de inicio

# abrir archivo
f = open(``euler1.dat'',''w'')
f.write( ``%f  %f \n'' % (v_0, w_0) )

for i in range(n-1):      # ciclo principal
   w_1 = w_0 + h*F(v_0, w_0)
   v_0 = v_0 + h
   w_0 = w_1
   # escribir en archivo
   f.write( ``%f  %f \n'' % (v_0, w_0) )
   # y en pantalla
   print v_0, w_0

# cerrar archivo
f.close()
```

→ **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 (1.9):

$$\boxed{\frac{dw}{dv} \cong \frac{w(v+h)-w(v-h)}{2h}} \tag{1.10}$$

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

$$\boxed{w(v+h) = w(v) + h\,w'(v) + \frac{h^2}{2!}w''(v) + \frac{h^3}{3!}w'''(v) + ...} \tag{1.11}$$

y

$$\boxed{w(v-h) = w(v) - h\,w'(v) + \frac{h^2}{2!}w''(v) - \frac{h^3}{3!}w'''(v) + ...} \tag{1.12}$$

entonces hagamos la resta de (1.11) y (1.12)[1.4]:

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

y despejemos $w'(v)$:

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

1.4. Si en lugar de hacer la resta hacemos la suma usando el mismo procedimiento se puede obtener una expresión para la segunda derivada $w''(v_n) = \frac{w(v_{n+1}) - 2w(v_n) + w(v_{n-1})}{h^2} + \mathcal{O}(h^2)$.

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

$$\boxed{w_{n+2} = w_n + 2h\, f(v_{n+1}, w_{n+1})} \tag{1.13}$$

En este caso el costo de la precisión en el método consiste en el hecho de que, como puede verse en (1.13) 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.

→ •

1.4 Sistemas Dinámicos de dos o más dimensiones

Como habíamos mencionado al inicio del capítulo, es un hecho que los sistemas en la naturaleza suelen tener varias o muchas variables y que, del mismo modo, hay parámetros que inciden en la dinámica correspondiente.

Una forma de estudiar la evolución de *cualquier* sistema[1.5] [2, 6] es bajo la óptica de los llamados *Sistemas Dinámicos* (SD). Una de las ideas importantes en este enfoque es que, como dijimos antes, uno pueda *ver* la dinámica del sistema: cómo se comporta en general y para cualquier tiempo, es decir, se trata de tener una imagen general de la dinámica del sistema sin tener de manera explícita la solución en el tiempo. Una de las implicaciones directas de esta idea es que en las ecuaciones que representan la dinámica del sistema el tiempo no aparece[1.6] y decimos entonces que el sistema es *autónomo*.

1.5. *Cualquier sistema* implica cualquier porción de nuestro universo, es decir, no se trata solamente de sistemas físicos o inanimados, se trata también de sistemas biológicos e, incluso, de sistemas que son producto del pensamiento (la literatura, las finanzas, etc.).

Las ecuaciones a las que estamos acostumbrados en la gran mayoría de modelos de la física son ecuaciones diferenciales de segundo orden, pero éstas siempre (como se puede ver por ejemplo al pasar de la ecuación de Lagrange a las ecuaciones de Hamilton[1.7]) se pueden transformar en ecuaciones diferenciales de primer orden haciendo las definiciones adecuadas. Para un SD de *dimensión m* las ecuaciones son de la forma general

$$\begin{aligned} \dot{x_0} &= F_0(x_0, x_1, ..., x_{m-1}) \\ \dot{x_1} &= F_1(x_0, x_1, ..., x_{m-1}) \\ &... \\ \dot{x}_{m-1} &= F_m(x_0, x_1, ..., x_{m-1}) \end{aligned} \tag{1.14}$$

donde $\dot{x}_i \equiv dx_i/dt$ es la derivada temporal de x_i. Las cantidades x_i son las variables del sistema[1.8] y las funciones F_i son expresiones algebraicas que involucran, en principio, a todas las variables x_i, es decir, al conjunto $\{x_m\} \equiv \{x_0, x_1, ..., x_{m-1}\}$. Esta es la forma general que asumen las ecuaciones de evolución en un SD y -esto es importante- en ellas *el tiempo no aparece en forma explícita* sino solamente a través de la derivada. 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 y que son representadas en un espacio fase ampliado.

1.6. Siempre cabe la posibilidad de definir una «nueva» variable del sistema $x_{m+1} = t$ y, por lo tanto, una nueva ecuación diferencial $\dot{x}_{m+1} = 1$. Ver la ecuación (1.14).

1.7. 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.

1.8. 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.

Tomemos como primer caso al Oscilador Armónico masa-resorte en tres formulaciones diferentes:

1. La formulación de Newton, implica simplemente escribir la segunda Ley de Newton de forma adecuada para la fuerza de Hooke (que es le debida a un material elástico o resorte, al menos en primera aproximación):

$$\ddot{x} = -\frac{k}{m}\, x \tag{1.15}$$

de modo que si definimos $x_1 = x$ y $x_2 = \dot{x} = \dot{x_1}$ tendremos que $\dot{x_2} = -\frac{k}{m}\, x_1$. Así que, finalmente,

$$\dot{x_1} = x_2 \tag{1.16}$$

y

$$\dot{x_2} = -\, x_1 \tag{1.17}$$

sin normalizamos k/m a 1. Este par de ecuaciones tienen la forma de un SD en el que $F_1(x_1,\, x_2) = x_2$ y también $F_2(x_1,\, x_2) = -\, x_1$.

2. La formulación de Lagrange. En este caso se trata de evaluar la energía cinética y la potencial en términos de q y de $\dot{q}$ de manera que se pueda escribir $\mathcal{L}(q,\, \dot{q}) = \mathcal{K}(q,\, \dot{q}) - \mathcal{U}(q,\, \dot{q})$ que es la función Lagrangiana en términos de coordenadas generalizadas y de velocidades generalizadas.

3. La formulación de Hamilton. Este caso es un poco diferente: se requieren las coordenadas generalizadas y los momentos generalizados (que ya no son las derivadas temporales de las coordenadas). Así, después de tener la función Hamiltoniana en términos de las q y los p: $\mathcal{H}(q,\, p) = \mathcal{K}(q,\, p) + \mathcal{U}(q,\, p)$, se busca la solución (la dinámica) del sistema a partir de las ecuaciones generales de movimiento

$$\begin{aligned} \dot{q} &= \frac{\partial \mathcal{H}}{\partial p} \\ \dot{p} &= -\frac{\partial \mathcal{H}}{\partial q} \end{aligned}$$

que obviamente tienen la forma de las ecuaciones (1.16) y (1.17) para q y p como variables dinámicas.

En el caso del oscilador armónico tenemos que $\mathcal{K} = \frac{p^2}{2m}$ y $\mathcal{U} = \frac{1}{2}kq^2$. La suma de ambos es el Hamiltoniano $\mathcal{H}$ que, al aplicarle las ecuaciones de Hamilton dan

$$\begin{aligned} \dot{q} &= \frac{p}{m} \\ \dot{p} &= -kq \end{aligned}$$

que corresponden con las definiciones de velocidad y fuerza de Hooke.

Ahora lo que tenemos es un sistema mecánico con su ecuación de movimiento de segundo orden transformada en dos ecuaciones de primer orden. Esta transformación permite usar la formulación de SD en esto que realmente sería un Sistema Hamiltoniano. Si nos fijamos bien, basta con cambiar $q, p \to x, mv$ para tener las expresiones tradicionales. El par de ecuaciones diferenciales, haciendo que $k/m = 1$ sirven para un oscilador «universal» y tienen la forma de (1.16) y (1.17).

Vamos a resolver el par de ecuaciones usando el método de Euler y algunas características propias de `python` pero, antes de comenzar,

Nota 1.4. veamos que hay una diferencia entre una ecuación diferencial de primer orden genérica y una *ecuación diferencial de primer orden sin variable independiente* del lado derecho. En el primer caso (comparar con (1.4)) la ecuación sería de la forma

$$\frac{dx}{dt} = F(x,t) \tag{1.18}$$

mientras que en el segundo caso se tendría (comparar con las ecuaciones (1.14))

$$\frac{dx}{dt} = F(x). \tag{1.19}$$

Es importante saber que si en alguna de las ecuaciones (1.14) aparece el tiempo explícitamente del lado derecho el sistema de ecuaciones no podrá resolverse[1.9].

Pensemos en una estructura algorítmica y de datos que nos permita resolver la cuestión:

$X_{0,0}$	$X_{0,1}$
$X_{1,0}$	$X_{1,1}$
$X_{2,0}$	$X_{2,1}$
...	...
$X_{t,0}$	$X_{t,1}$
$X_{t+1,0}$	$X_{t+1,1}$
...	...

X es el nombre de la matriz que vamos a utilizar. El primer índice nos dice en qué paso de tiempo estamos y el segundo índice se refiere a la variable[1.10]. De este modo la primera columna de la matriz tiene los valores de x_1 evolucionando en el tiempo y la segunda columna los de x_2. Los índices correspondientes son *cero* y *uno* por cuestiones de manejo interno del lenguaje. La primera fila de la matriz entonces representa el estado inicial en el espacio fase: el primer punto de la dinámica representada ahí. Las filas subsiguientes representan la evolución de todas las variables del sistema.

Ejemplo 1.5. El oscilador armónico tomando como base el ejemplo 1.3:

```
# método de Euler para Sistemas Dinámicos
# F. Rojas

# definimos cantidades imporantes del sistema
DIM = 2    # número de variables (columnas de la matriz)
np = 10000 # número de 'pasos' de tamaño dt
dt = 0.001 # tamaño de paso de tiempo

X = [[0 for i in range(DIM)] for t in range(np)] # la matriz
```

1.9. 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.

1.10. Hay que tener cuidado con los subíndices pues en varios lenguajes los arreglos, listas, matrices, etc. tienen como primer subíndice el *cero* (como en este caso).

```
   # lectura valores de inicio
   X[0][0] = input(``X_0 inicial: '')
   X[0][1] = input(``X_1 inicial: ``)

   # las funciones de lado derecho de las ecs. 1.16 y 1.17
   def F0(X):
      return X[1]

   def F1(X):
      return -X[0]

   # un vector o lista de funciones
   F = [F0, F1]

   # abrir archivo
   f = open(``euler1.dat'',''w'')

   for t in range(np-1):      # ciclo principal Euler
      # el método de Euler para todas las variables (columnas)
      # el índice t indica al fila; X[t] es la fila completa
      for i in range(DIM):
         X[t+1][i]=X[t][i] + dt*F[i](X[t])

   # escribir matriz en archivo
   for t in range(np):
      # escribir en archivo las 4 coordenadas del espacio fase
      f.write( ``%f %f %f %f \n'' % (X[t][0], X[t][1], F[0](X[t]),
F[1](X[t])) )

   # cerrar archivo
   f.close()
```

El bloque de escritura y evaluación (ciclo de Euler) de la matriz fueron separados: siempre es más seguro trabajar con módulos independientes. Se escriben en el archivo *todas* las variables que ineter-vienen X_0, X_1, $\dot{X}_0$, *y* $\dot{X}_1$. Los valores de las derivadas los dan las funciones `F0` y `F1` en cada paso de tiempo.

Es un hecho demostrado que en dos o más dimensiones también hay atractores y tienen diferentes formas para diferentes dinámicas y número de variables. En el caso de dos dimensiones solamente puede haber PF o bien atractores bidimensionales, trayectorias en el espacio fase que atraen la dinámica del sistema. En este caso el evento es que el sistema no depende del todo de sus condiciones iniciales: siempre su dinámica lo lleva a visitar algún atractor y a quedarse ahí dando vueltas.

Ejemplo 1.6. El ejemplo consiste de un oscilador (puede reconocerse en las ecuaciones) más un término cuyo peso está determinado por el parámetro μ:

$$\dot{x_1} = -x_2 + \mu x_1^2$$

$$\dot{x_2} = x_1 + \mu x_1^2$$

Este caso, como muchos otros, es interesante porque presenta un atractor bidimensional: diferentes condiciones iniciales llevan al oscilador a una misma trayectoria en el espacio fase de la cual no sale:

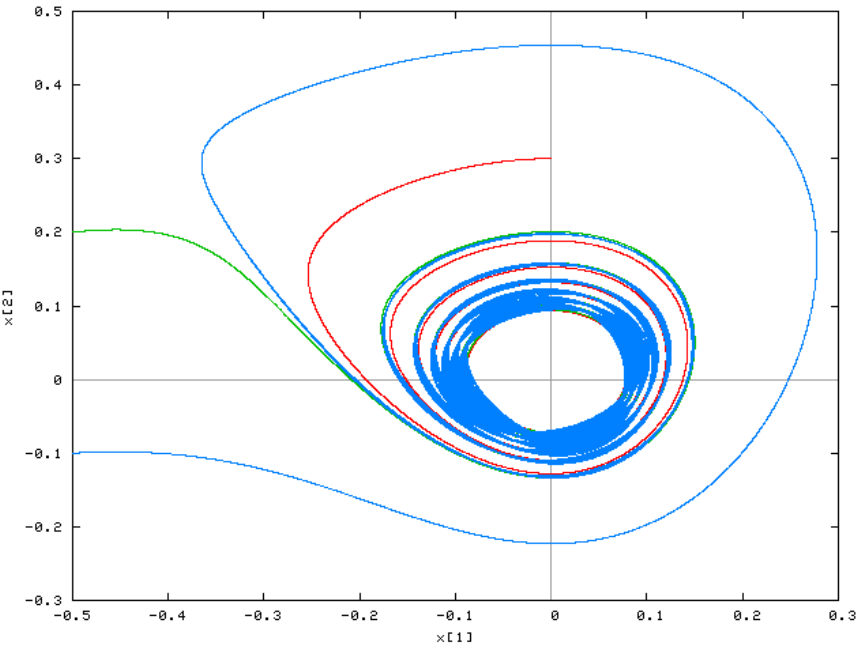


Figura 1.1. Atractor bidimensional. Inicios: (0.0,0.3), (-0.5,0.2), (-0.5,-0.1)

En este caso $\mu = 2.2$, los tres puntos de inicio de las trayectorias no inciden en la forma final del atractor: inesperadamente todas las trayectorias se dirigen hacia el centro y se quedan al rededor del origen. Nuestro sistema oscila anarmónicamente sin tocar el origen del espacio fase. Sin embargo ocurren otras cosas inesperadas.

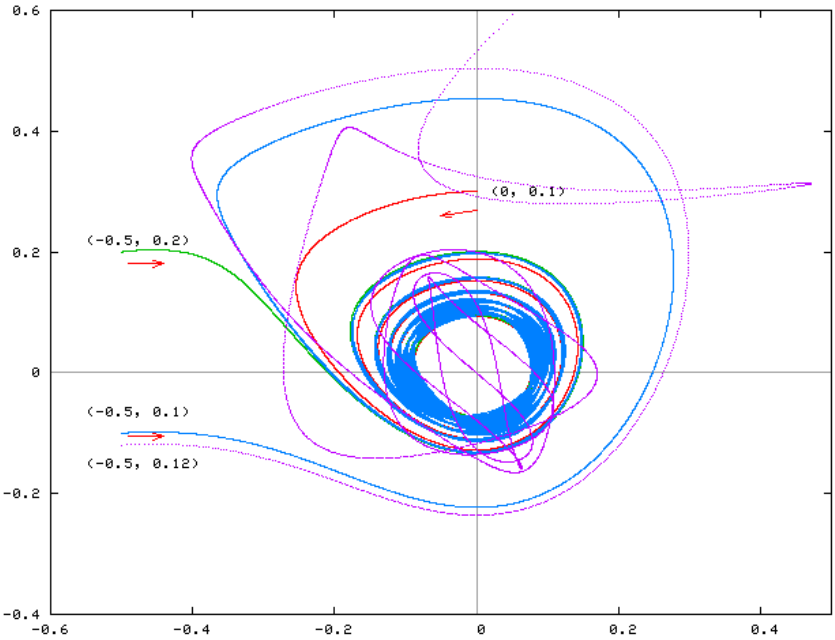


Figura 1.2. La trayectoria irregular inicia en (-0.5,-0.12)

Puede verse ahora una nueva trayectoria que inicia más externamente que las tres iniciales. La diferencia en el punto de inicio es de dos centésimas (-0.5, -0.12) (comparar con la línea inmediatamente arriba que inicia en (-0.5, -0.1)). La línea más externa se acerca a la región del atractor, pero una vez ahí su trayectoria comienza a hacer giros y saltos indescriptibles[1.11].

Nota 1.7. En este caso decimos que el sistema es *Caótico*: su comportamiento dinámico depende muy sensiblemente de las condiciones iniciales.

Este tipo de comportamiento en las ecuaciones diferenciales *no lineales* es lo que dió origen allá por los 1970 a la llamada Teoría del Caos y a la idea del *Efecto Mariposa*. En los ejercicios hay algunos

1.11. En realidad la línea curva con rizos y ésas cosas no es la trayectoria. Las líneas realmente van saltando de extremo a extremo de las curvas hasta que salen de la región mostrada y se alejan rápidamente.

ejemplos y, en particular, se encuentra el original de Lorenz relacionado con la dinámica de los cambios climáticos[1.12].

1.5 Mayor precisión: Runge-Kutta

En esta sección mostramos una técnica de mayor precisión para la solución numérica de sistemas de ecuaciones diferenciales. No haremos una discusión muy detallada del método y sí algunas observaciones.

→ **Método de Runge-Kutta de *cuarto* orden**[1.13]

- Una forma alterna de modificar el método de Euler consiste en tomar una *primera aproximación* a x_{n+1} usando la forma convencional (1.9) o, usando la formulación de SD (donde sabemos que la variable independiente t no aparece explícitamente en las F_i), tendríamos la expresión (1.9). 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 (?)), es tener siempre x_{n+1} solamente en términos de x_n.

1.12. Algunos casos de SD no pueden resolverse (o al menos cuestan mucho trabajo) usando métodos de Euler. Los métodos más precisos a veces resultan imprescindibles. Ver la sección siguiente para estudiar el método de Runge-Kutta.

1.13. 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.

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

$$\begin{aligned} k_1 &= dtF(x_n) \\ k_2 &= dtF(x_n + \frac{1}{2}k_1) \\ k_3 &= dtF(x_n + \frac{1}{2}k_2) \\ k_4 &= dtF(x_n + k_3) \end{aligned} \tag{1.20}$$

y, finalmente

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{1.21}$$

- 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*) $\{x\}_{n+1}$ se requiere, además del *punto* $\{x\}_n$, todos los valores de todas las $k_{1,2,3,4}$. Lo recomendable sería, por ejemplo, definir arreglos (vectores) `k1[dim]`, k2[dim], k3[dim], k4[dim] donde `dim` es la dimensión del SD[1.14]. En ese caso el algoritmo quedaría así

$$k_{1,i} = dt\, F_i(\{x\}_n) \tag{1.22}$$

$$k_{2,i} = dt\, F_i(\{x\}_n + \frac{1}{2}\{k_1\}) \tag{1.23}$$

$$k_{3,i} = dt\, F_i(\{x\}_n + \frac{1}{2}\{k_2\}) \tag{1.24}$$

$$k_{4,i} = dt\, F_i(\{x\}_n + \{k_3\} \tag{1.25}$$

y entonces, para cada variable dinámica x_i

$$x_{i,n+1} = x_{i,n} + \frac{1}{6}(k_{1,i} + 2k_{2,i} + 2k_{3,i} + k_{4,i}) \tag{1.26}$$

1.14. Al igual que en el ejemplo de la función *Euler()*, se pueden definir `k1[dim+1], k2[dim+1], k3[dim+1], k4[dim+1]` con el fin de manejar subíndices de `1` a `dim` en lugar de tener índices de `0` a `dim-1`.

No hay que perder de vista que i) la expresión $\{x\}$ representa al conjunto de variables del SD que, en términos de algoritmos (o programas) representamos con un arreglo o vector de dimensión m (lo mismo vale para $\{k_j\}$), ii) cada subíndice i corresponde a cada una de las ecuaciones diferenciales del SD, tanto para las variables $x_{i,n}$ como para las constantes k_j y para las funciones $F_i(\{x\})$; iii) las funciones $F_i(\{x\})$ son funciones de *todas* las variables x_i, es decir, del conjunto $\{x\}$; iv) los símbolos $\{x\}_n + \frac{1}{2}\{k_{1,2,3,4}\}$ representan aquí la suma de los vectores $\{x\}$ con $\{k_{1,2,3,\acute{o}\,4}\}$ *elemento a elemento* (en un ciclo *for* esto sería escrito como `x_aux[j] = x0[j] + 0.5*k1[j]`, por ejemplo).

→ •

1.6 Los modelos del tipo Lotka-Volterra

El modelo de Lotka-Volterra que se puede encontrar prácticamente en cualquier texto de Ecuaciones Diferenciales es un primer ejemplo de SD en el que no hay interveción de formulaciones teóricas. De hecho uno puede construir «con las manos» el modelo de la dinámica de dos especies, depredador y presa:

Definamos dos variables para nombrar a las dos especies, digamos que x e y. Según el modelo primitivo de Malthus la población deberá crecer de manera proporcional a la población que ya existe[1.15]:

$$\begin{aligned} \dot{x} &= \alpha_1\, x \\ \dot{y} &= \alpha_2\, y, \end{aligned}$$

donde los coeficientes α_i son característicos de tal o cual especie. Ahora bien si tenemos x predadores y y presas, la «ganancia» de población de los depredadores dependerá del número de encuentros reales entre elementos de las dos especies, es decir, es proporcional al

1.15. El grado de certeza de esta afirmación es proporcinal al grado de certeza de poder encontrar una especie aislada de las demás y de los recursos. Malthus, en su momento, causó grandes controversias pues afiormaba que la humanidad estaba destinada a desaparecer por falta de alimentos: según el modelo la población crece exponencialmente y al parecer la producción de alimentos crece más lentamente. El detalle está en que ninguna especie está aislada.

producto xy. De la misma forma ocurrirá con las presas, aunque esta vez se trate de «pérdida». De este modo tenemos una forma general de los modelos de Lotka-Volterra:

$$\dot{x} = \alpha_1 x + \beta_1 xy$$

$$\dot{y} = \alpha_2 x - \beta_2 xy$$

donde, nuevamente, los parámetros β_i dependen explícitamente de las especies en cuestión y, probablemente, habría que determinarlos de manera experimental.

1.6.1 Series de Tiempo

1.7 Puntos Fijos, Raíces... y errores

¿Cuál es la idea de esta sección? Bueno, resulta por un lado que los PF no son otra cosa que las raíces de las funciones del «lado derecho» para SD unidimensionales o para conjuntos de funciones si se tarata de un SD de dimensión mayor que uno. En el caso de más dimensiones los PF, de existir, definen el comportamiento del sistema en las vecindades de los mismos y, en algunos casos, aún más allá.

Vamos a partir de un ejemplo particular de flujo en el cual no podemos evaluar los puntos fijos de manera analítica, por ejemplo podemos considerar el flujo

$$\dot{x} = x - \cos x \tag{1.27}$$

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 esta función corta al eje x (otra forma equivalente sería graficar $f_1(x) = x$ y $f_2(x) = \cos x$ 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[1.16]. 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

1.16. Hay que notar aquí que los *ceros* a los que nos referimos son las raíces de la función de la derecha.

de cos en (1.27) 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

$$x_{n+1} = \phi(x_n) \tag{1.28}$$

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 (1.28) es típica de los procedimientos iterativos en métodos numéricos. De hecho se puede comparar, por ejemplo, con el método de Euler (1.9) que encuentra soluciones de ecuaciones diferenciales.

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 (1.28) que es la fórmula relacionada con este método. 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[1.17].

- Implementar la iteración definida por (1.28) y detener cuando $\left|x_{n+1} - x_n\right| < \text{TOL}$ 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 (dado que las x_n deben acercarse cada vez más a α) $\iff e_0 > \epsilon_1 > \epsilon_2 ... > \epsilon_n ...$ así que de (1.28)

$$\begin{aligned} \alpha + \epsilon_{n+1} &= \phi(\alpha + \epsilon_n) \\ &= \phi(\alpha) + \epsilon_n \phi'(\alpha) + \epsilon_n^2 \phi''(\alpha) + ... \end{aligned} \tag{1.29}$$

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

$$\begin{aligned} \epsilon_{n+1} &= \epsilon_n \phi'(\alpha) + \epsilon_n^2 \phi''(\alpha) + ... \\ &= \epsilon_n \phi'(\alpha) + \mathcal{O}(\epsilon_n^2) \end{aligned} \tag{1.30}$$

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

$$\epsilon_{n+1} = \epsilon_n \phi'(\alpha) \tag{1.31}$$

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.

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

- Aunque suele chocar con lo expuesto es bueno discutir lo que pasa en esta última expresión: en la ecuación (1.31) 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.

→ •

Vamos a construir aquí una versión de la función *Punto_Fijo()* que nos va a servir para usar este método y también, más adelante, para hacer análisis de la dinámica de un sistema que no se plantea por medio de ecuaciones diferenciales, sino de *mapeos*. Por el momento en esta función necesitamos poner como parámetros el punto inicial x_0, la *tolerancia* (que define la precisión deseada para el valor de la raíz) y la dirección de la función $\phi(x)$:

Ejemplo 1.8. La función `Punto_Fijo()`, en este caso usando una lista que guarda temporalmente la secuencia de valores aproximados a la raíz:

```
def Punto_Fijo(F, xo, TOL):
    x = [0]*100
    x[0] = xo
    dist, n = 1.+TOL, 0
    while dist > TOL:
        x[n+1] = F(x[n])
        dist = abs(x[n+1]-x[n])
        n += 1
    return x[n+1]
```

en este otro caso no se usan listas:

```
def Punto_Fijo(F, xo, TOL):
    dist = 1.+TOL
    while dist > TOL:
        x1 = F(xo)
```

```
        dist = abs(x1-xo)
        xo = x1
    return x1
```

•

Un método mejor siempre es posible en métodos numéricos aunque, como hemos visto, la precisión tiene su costo en sencillez y velocidad (o número de cálculos). El método más popular para encontrar raíces de funciones es el método de *Newton-Raphson*:

→ **Método de *Newton-Raphson* para raíces de funciones**

1. Dada la función cuya raíz se busca, digamos $F(x) = x - \cos x$, uno busca, como en el método de *iteración simple* o *punto fijo*, una expresión de la forma (1.28).
2. Como siempre hay que dar un valor inicial, x_0, cercano a la raíz verdadera.
3. Implementar la iteración, de la forma (1.28), y detener cuando $|x_{n+1} - x_n| < \text{TOL}$. Desde luego, en este caso, la función $\phi(x_n)$ tiene una forma específica:

$$x_{n+1} = x_n - \frac{F(x_n)}{F'(x_n)} \tag{1.32}$$

   en la que obviamente la función $\phi(x_n) = x_n - F(x_n)/F'(x_n)$.
4. El error de este método (que es el más utilizado) es de orden $\mathcal{O}(h^2)$. Para visualizar esto retomamos la forma

$$\begin{aligned} \alpha + \epsilon_{n+1} &= \phi(\alpha + \epsilon_n) \\ &= \alpha + \epsilon_n - F(\alpha + \epsilon_n)/F'(\alpha + \epsilon_n) \end{aligned} \tag{1.33}$$

   y procedemos de manera análoga al caso del método de Euler.

→ •

1.8 Discusión

1.9 Ejercicios

1.9.1 de afirmación

Ejercicio 1.1. Una modificación al SD expresado en (1.2) puede consistir en introducir un *parámetro de control*:

$$\dot{x} = r - x^3$$

En este caso el parámetro de control es $r \geqslant 0$. Su nombre proviene del hecho de que la dinámica del sistema va a ser modificada cuando r tome diferentes valores.

i. Encontrar los PF para este SD, es decir, los valores posibles de x que hagan $\dot{x} = 0$. Hay que notar desde ya que se trata de una función explícita de r, es decir, los PF dependen directamente del parámetro de control.

ii. Hacer la gráfica en el espacio fase y discutir la dinámica a la derecha e izquierda del punto $\sqrt[3]{r}$.

Ejercicio 1.2. La misma tarea que en el ejercicio anterior, pero ahora con dos parámetros de control positivos:

$$\dot{x} = r - ax^3.$$

Ejercicio 1.3. Dado el SD:

$$\dot{x} = r - a\sin(x)$$

con $r, a > 0$.

i. Encontrar la solución analítica $x(t)$

ii. Hacer la gráfica de $x(t)$ ($x(t)$ contra t)

iii. Hacer la gráfica en el espacio fase para diferentes valores de r fijando a.

iv. ¿Qué relación debe haber entre r y a para que no existan puntos fijos?

v. ¿Cuál es la diferencia de comportamiento de dos PF vecinos? ¿Cómo cambia la dinámica cerca de cada uno de ellos?

Ejercicio 1.4. 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 1.5. Llegar explícitamente a la expresión del Método de Euler de segundo orden (1.13) a partir de (1.10) aplicado a la ecuación diferencial (1.4).

Ejercicio 1.6. Hacer la suma de (1.11) y (1.12) y despejar $w''(v)$ para obtener $w''(v_n) = \frac{w(v_{n+1}) - 2w(v_n) + w(v_{n-1})}{h^2} + \mathcal{O}(h^2)$. Que es una expresión útil en el caso de ecuaciones diferenciales parciales.

Ejercicio 1.7. Construir una función `Euler(X, F, dim, np, h, arch = 'euler.dat')` de modo que el programa del ejemplo 1.5 se reduzca a:

```
# método de Euler para Sistemas Dinámicos
# F. Rojas

# definimos cantidades imporantes del sistema
DIM = 2    # número de variables (columnas de la matriz)
np = 10000 # número de 'pasos' de tamaño dt
dt = 0.001 # tamaño de paso de tiempo

X = [0 for i in range(DIM)] # la matriz
# lectura valores de inicio
X[0] = input(``X_0 inicial: '')
X[1] = input(``X_1 inicial: ``)

# las funciones de lado derecho de las ec. 1.16 y 1.17
def F0(X):
   return X[1]

def F1(X):
   return -X[0]

# un vector o lista de funciones
F = [F0, F1]

# llamar Euler:
Euler(X, F, DIM, NP, dt, 'miarchivo.dat')
```

Hay que notar que ya no se requiere aquí la matriz completa: lo único que requerimos es la primera fila (la condición inicial). La matriz, en todo caso, será construida y usada dentro de la nueva función `Euler`. La función puede estar en otro archivo de texto. Ver ejercicios adelante.

Ejercicio 1.8. La misma idea pero para la función `Euler2`, en donde se implemente el método de Euler de segundo orden.

Ejercicio 1.9. Colocar las dos funciones `Euler()` y `Euler2()` en un archivo independiente `SisDinam.py` por ejemplo. De este modo al inicio del archivo se pondrá la línea `from SisDinam import Euler` que implica que uno puede llamar a la función `Euler()` «importada» de `SisDinam.py`

Ejercicio 1.10. Se trata de retomar las ecuaciones de movimiento del oscilador armónico y agregarle términos relacionados con *fricción* como función de la velocidad x_2, y/o *fricción* relacionada con la posición x_1. Resolver numéricamente y discutir el espacio fase obtenido.

Ejercicio 1.11. * Resolver los dos problemas anteriores usando una nueva función `Runge_Kutta(X,F,dim,n,h,archivo='datos.dat')`. La nueva función debería tener la forma aproximada siguiente:

```
def Runge_Kutta(X,F,dim,n,h=1.e-3, arch='datos.dat'):
   DIMR = range(dim) # lista de valores 0...dim-1
   X0 = X[:]  # copia de X en la variable local X0

   def writeX(X): # función dentro de función para escribir fila
      for j in DIMR:
         print >> ff, X[j],  # coma para que escriba en seguida
      print >> ff

   ff = open(arch,''w'')
   writeX(X0)

   # método de Runge-Kutta
   for t in range(n):
      k1 = [ h*F[j](X0) for j in DIM ]
      k1x = [ X0[i]+0.5*k1[i] for i in DIM ]
      k2 = [ h*F[j](k1x) for j in DIM ]
      k2x = [ X0[i]+0.5*k2[i] for i in DIM ]
      k3 = [ h*F[j](k2x) for j in DIM ]
      k3x = [ X0[i]+k3[i] for i in DIM ]
      k4 = [ h*F[j](k3x) for j in DIM ]
      X1 = [ X0[i] + 1./6.*(k1[i] + 2*k2[i] + 2*k3[i] + k4[i]) |
             for i in DIM ]
      writeX(X1)
      del X0
      X0 = X1[:]
      del k1, k2, k3, k4, k1x, k2x, k3x, X1
  ff.close()
```

Ejercicio 1.12. Usando expansiones en serie de Taylor para $F(x)$ y $F'(x)$ en la expresión (4) demostrar que el método de Newton-Raphson tiene un error de orden $\mathcal{O}(h^2)$ (ver el caso de Euler arriba).

Ejercicio 1.13. Buscar una justificación geométrica del método de Newton-Raphson.

Ejercicio 1.14. Encontrar las raíces usando `Punto_Fijo()` y el método de Newton-Raphson de las siguientes funciones, buscando valores iniciales aproximados por medio de gráficos:

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

ii. $F(x) = Kx(1-x)$ para diferentes valores de K.

iii. $F(x) = x - a\cos(x)$

1.9.2 de aplicación

Ejercicio 1.15. * A partir del ejemplo implemetar la solución numérica para encontrar el atractor que encontró Lorenz al tratar de ecuaciones para el clima

$$\begin{aligned} \dot{x}_1 &= P(x_2 - x_1) \\ \dot{x}_2 &= Rx_1 - x_2 - x_1x_3 \\ \dot{x}_3 &= x_1x_2 - Bx_3 \end{aligned}$$

los parámetros que se acostumbran graficar en la literatura son $P = 10$, $R = 28$, $B = 8/3$.

Ejercicio 1.16. * Lo mismo que en el caso anterior. Las ecuaciones de Rossler son:

$$\begin{aligned} \dot{x}_1 &= -(x_2 + x_3) \\ \dot{x}_2 &= x_1 - 0.2x_2 \\ \dot{x}_3 &= 0.2 + x_3(x_1 - 5.7) \end{aligned}$$

y están relacionadas con cinética química. Los mecanismos de reacción química que involucran dos componentes típicamente ocurren a velocidades relacionadas con el producto de las concentraciones. Rossler descubrió muchas ecuaciones que mostraban caos y este es el ejemplo más conocido.

Ejercicio 1.17. * En este problema[1.18] se verá lo fácil que es el análisis del problema de la epidemia en el espacio fase. Aquí $x(t) \geqslant 0$ denota el tamaño de la población sana y $y(t) \geqslant 0$ el tamaño de la población infectada. El modelo es

$$\begin{aligned} \dot{x} &= -kxy \\ \dot{y} &= kxy - ly \end{aligned}$$

donde $k, l > 0$. (Aquí se omite $z(t)$ que representa el número de muertes pues no afecta la dinámica en el plano xy.

i. Encontrar y clasificar los PF

ii. Dibujar las *nulclinas*[1.19]

iii. Encontrar una cantidad que se conserve en el sistema (Construir una ecuación diferencial para dy/dx, separar las variables e integrar ambos lados)

iv. Dibujar el espacio fase xy

v. Elegir un punto de inicio (x_0, y_0). Se dice que ocurre una *epidemia* si $y(t)$ se incrementa inicialmente. ¿Bajo qué condición ocurre esto?

1.18. Tomado de la referencia [6].

1.19. Las curvas que dependen de x e y que corresponden a $\dot{x}, \dot{y} = 0$.

Ejercicio 1.18. * Relatividad general y órbitas planetarias[1.20]. La ecuación relativista para la órbita de un planeta alrededor del sol es

$$\frac{\partial^2 u}{\partial \theta^2} + u = \alpha + \varepsilon u^2$$

donde $u = 1/r$ y r, θ son las coordenadas polares del planeta en su plano de movimiento. El parámetro α es positivo y se puede encontrar explícitamente de la mecánica clásica de Newton; el término εu^2 es la corrección de Einstein. Aquí ε es un parámetro positivo muy pequeño.

i. Reescribir la ecuación como un SD en el espacio fase (u, v) donde $v = du/d\theta$

ii. Encontrar los puntos de equilibrio del sistema

iii. Demostrar que el punto de equilibrio corresponde a una órbita circular.

Ejercicio 1.19. * Hardenberg [7] propone un modelo de la dinámica de un sistema que involucra la densidad de biomasa vegetal y la humedad local generada por lluvias:

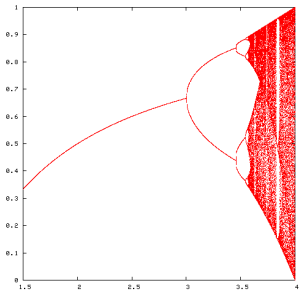
$$\begin{aligned} \dot{n} &= \frac{\gamma}{1+\sigma n} n - n^2 - \mu n + \nabla^2 n \\ \dot{w} &= p - (1-\rho n) w - w^2 p + \delta \nabla^2 (w - \beta n) - v \frac{\partial (w - \alpha n)}{\partial x} \end{aligned}$$

en donde n es la densidad de biomasa, w la humedad local, p (el parámetro de control en el modelo) es la cantidad de lluvia promedio anual. Los valores de los otros parámetros son obtenidos de las referencias en el artículo original: $\gamma = 1.6$, $\sigma = 1.6$, $\mu = 0.2$, $\rho = 1.5$, $\alpha = 3.0$, $\beta = 3.0$, $\delta = 100$. El problema incluyendo la parte espacial será revisitado en otro capítulo. Por el momento, atediendo solamente a la dinámica, encontrar los PF y dibujar el espacio fase del modelo de Hardenberg. Superponer para diferentes valores de p (0.1, 0.2, 0.3, 0.5, 0.7, 0.9)

1.20. Tomado de la referencia [6].

Capítulo 2
Dinámica Discreta

Los diagramas de bifurcaciones se aprecian sobre todo en la evolución de los mapeos. Son importantes pues muestran una cara de la naturaleza que no se conocía y tiene que ver con la impredecibilidad: no es posible saberlo todo, ni predecirlo todo.



2.1 Introducción

Estamos acostumbrados, como parte de nuestra cultura, a una forma de pensar en la que el continuo forma la base de toda construcción mental o material. Aún cuando sabemos que al final encontraremos átomos, moléculas o, incluso, partículas más elementales, generalmente ocurre que, al enfrentar problemas y buscar o construir soluciones, se piensa antes que nada en una formulación continua del problema y, por consistencia, en una solución del mismo tipo, sin embargo «*La invención de computadoras digitales rápidas y el desarrollo de la teoría de sistemas dinámicos, hacia finales del Siglo XX, trajo aparejado la redefinición o creación de entidades matemáticas nuevas, que son de gran utilidad para el estudio de fenómenos discretos que ocurren en el espacio. Ejemplo de ellos son los autómatas celulares, los mapeos acoplados y las redes neuronales. Estos sistemas, pese a su extrema simplicidad, son capaces de exhibir una gran complejidad de conductas dinámicas tales como el caos determi-*

nista, la autoorganización y la formación de patrones espaciales, sincronización, procesamiento de información y cómputo emergente, etcétera. En esta línea, he estudiado problemas de formación de patrones en morfogénesis, usando autómatas celulares elementales e híbridos de autómatas celulares con redes neuronales para explorar la dinámica temporal de insectos sociales, finalmente, he construido modelos basados en mapeos acoplados para el estudio de propiedades espaciales de ecosistemas complejos.»[2.1].

Es un hecho que, a partir de los años 1970, cuando nacía la *Teoría de las Catástrofes* creada por el ganador del Premio Fields René Thom, comenzó una crítica generalizada hacia cierto tipo de «actitudes adquiridas» por el pensamiento científico. Una de las críticas que hace Thom es lo que él mismo llama *reduccionismo* (en una entrevista publicada en los años 1970 [??]) y se refiere principalmente a dos cosas: primero a lo que él llama *atomismo* y que consiste en suponer que si uno conoce los elementos de un sistema, es decir, los individuos o las partes (los *átomos*), entonces será capaz de describir al sistema completo (hoy decimos que *el todo no es la suma de sus partes*, precisamente por evidencias como las que menciona O. Miramontes arriba). La otra parte de la crítica se refiere al pensamiento *diferencial*. El que mencionamos al inicio de esta introducción y que está asociado a las funciones *bien portadas*, continuas, diferenciables y que supone que son suficientes para describir a los fenómenos o a la naturaleza.

En este capítulo vamos a discutir algunas cuestiones relacionadas sí, con dinámica, pero con la visión *discreta* del asunto.

2.2 Mapeos

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 (1.28) pero que, además, tienen el sentido de representar la evolución en el tiempo (discreto) de algún sistema.

2.1. La cita es de Octavio Miramontes (Sistemas Complejos, IFUNAM), tomada de su página web: *http://scifunam.fisica.unam.mx/mir/biol.html*

La forma general de los mapeos con varias variables significativas para el sistema es análoga al caso de un SD

$$\begin{aligned} x_{1,n+1} &= F_1(x_{1,n}, x_{2,n}, ..., x_{m,n}) \\ x_{2,n+1} &= F_2(x_{1,n}, x_{2,n}, ..., x_{m,n}) \\ &... \\ x_{m,n+1} &= F_m(x_{1,n}, x_{2,n}, ..., x_{m,n}). \end{aligned} \tag{2.1}$$

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

$$x_{n+1} = Kx_n \tag{2.2}$$

donde K representa la razón de crecimiento de la población ($K = 1.27$ representa el 27% de crecimiento para el periodo 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 (2.2) no tiene puntos fijos: si $K < 1$ la población desaparece y si $K > 1$ entonces crecerá indefinidamente. Para efectos de cálculo se normaliza con x entre 0 y 1 de modo que el valor $x = 1$ representa el 100% de un valor de referencia escalado. La expresión (2.2) 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:

$$x_{n+1} = 4Kx_n(1 - x_n) \tag{2.3}$$

El factor 4 se introduce para que el valor de K sea de 0 a 1. 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 discretas de tiempo
     $x_0 \leftarrow F(x_0)$
  3. fin

  La función F(x):
  1. calcula r $\leftarrow$ 4Kx(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.

- •

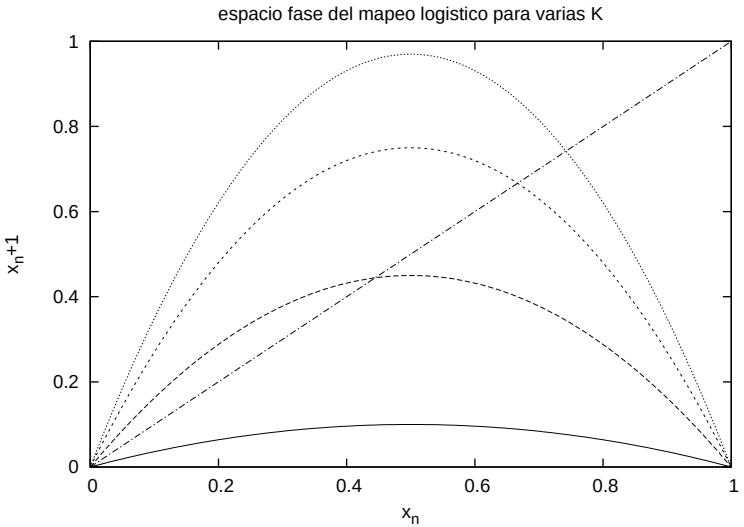
Ejercicio 2.1. Hacer el programa para poder graficar $x(t)$ del mapeo logístico a partir del algoritmo. (Lo que vas a observar es que para $K < 1$ la población (o la variable correspondiente, x) decae. Para K entre 1 y 3 el mapeo tiende a un PF y para $K > 3$...)

De la misma manera que en el caso de los SD, en los mapeos podemos ver la dinámica en un espacio fase particular

Habrás 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 cerca de ciertos valores críticos la dinámica del sistema cambia de manera radical.

En el caso de un mapeo el espacio fase del mapeo será x_{n+1} contra x_n. Para el caso del mapeo logístico x_{n+1} es una parábola "encerrada" en el cuadrado $[0,1] \times [0,1]$:

```
GNUplot] set title "espacio fase del mapeo logistico para
         varias K"
         set size 0.8,0.8
         set xlabel "x_n"
         set ylabel "x_n+1"
         set nokey
         F(x) = 4*K*x*(1.-x)
         plot [0:1] K=0.1,F(x), K=0.45,F(x), K=0.75,F(x),
         K=0.97,F(x), x
```



La recta diagonal es la recta formada por todos los Puntos Fijos del mapeo (¿es obvio?.. es ahí donde están todos los casos posibles $x_{n+1} = x_n$). Como puede verse de las órdenes de `gnuplot` las parábolas corresponden a la función $F(x) = 4Kx(1-x)$ para diferentes valores de K. Todos los PF se encuentran en las intersecciones de $F(x)$ con la recta x de los PF para cada valor de K.

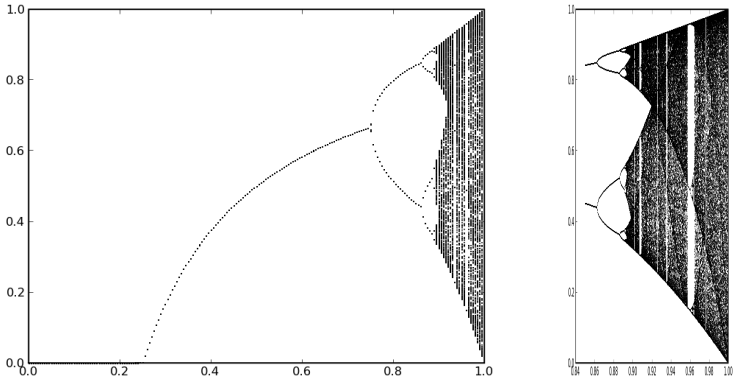


Figura 2.1. Diagrama de Bifurcaciones. mapeo Logístico

¿Cómo encontramos los PF de un mapeo? A diferencia del caso continuo (en donde los PF se encuentran cuando la derivada -o el lado derecho, F- es cero), en el caso discreto los PF se encuentran de la misma manera que ocurre en el método de Iteración Simple: cuando x_{n+1} y $\phi(x_n)$ coinciden, es decir, satisfacen

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

lo cual significa que nuestra variable x no se mueve para el valor $x = x^\star$. En el caso del mapeo logístico (2.3) los PF se encuentran resolviendo la ecuación

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

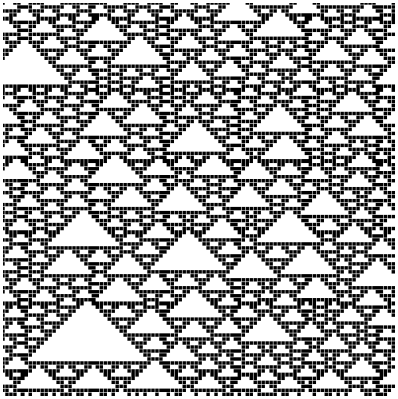
que es la relación que deben satisfacer los PF. Tras alguna manipulación se tiene:

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

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

2.3 Autómatas Celulares

Los AC son objetos distribuidos espacialmente cuyas celdas cambian su estado a uno nuevo que depende solamente de sus vecinos. Obviamente la forma de las celdas no importa (cuadradas es lo más fácil), lo importante es que evolucionan, computan, modelan fenómenos, generan patrones, interactúan, muestran caos, etc. etc.



En la imagen tenemos la evolución en el tiempo (hacia arriba) de un Autómata Celular unidimensional: una matriz «lineal», arreglo o vector en el que cada una de sus celdas puede tener estados 0 ó 1. En este caso se encuentran aleatoriamente dispuestos en el estado inicial del AC. La línea horizontal de cuadrados blancos y negros más abajo es el estado inicial. Cada línea horizontal hacia arriba es un paso en la evolución de la línea inicial siguiendola regla 126. La imagen muestra un patrón que uno no podría esperar. Un patrón de una complejidad que no podría esperarse si se sabe que la regla de evolución es tan simple como definir cuál será el nuevo estado de la celda j si sus vecinas $j-1$ y $j+1$ tienen tal o cual estado. Por ejemplo, la *regla 90* se establece así:

111	110	101	100	011	010	001	000	la regla:
0	1	0	1	1	0	1	0	$01011010_2 = 90_{10}$

que en colores diría, leyendo las dos primeras columnas: si una celda es negra y tiene vecinas negras, su nuevo estado es blanco; si es negra y su vecina izquierda es negra pero la derecha es blanca, su nuevo estado será negro; etc. Como puede verse en la última columna el nombre de *regla 90* viene de la traducción de binario a decimal de los dígitos que definen a la regla. Hay que notar que para dos vecinos (se dice en este caso *vecindad r=1*) se tienen cuando más $2^3 = 8$ posibles combinaciones 0 y 1.

Hasta el momento hemos encontrado diferentes tipos de comportamiento dinámico: *atractores puntuales*, para los que la dinámica converge a un estado de equilibrio caracterizado por la constancia de todas las variables a lo largo del tiempo; comportamiento *periódicos* o cuasiperiódicos que son predecibles y que muestran cambios regulares a lo largo del tiempo. Finalmente hemos visto también el comportamiento caótico tanto en los SD de tres o más variables como en la dinámica discreta de una sola variable: sistemas que exhiben propiedades dinámicas enormemente complejas.

Discusión. El caos determinista, que es caracterizable por la correlación del atractor o sus exponentes de Lyapunov [5], muestra una propiedad muy notable: *su alta sensibilidad a las condiciones iniciales*. Lo remarcamos aquí porque en parte puede ser la condición física, real, que vuelve impredecible al sistema. Pensando en el tiro de un dado y en el hecho de que el evento es puramente mecánico uno puede preguntarse ¿dónde y cómo *entró* el azar en el sistema?

Una conjetura del área de la Biología: «*Son muchos los ejemplos en la naturaleza de comportamientos complejos que surgen a partir de componentes e interacciones simples. Si se da una descripción adecuada y completa de dichos componentes, es posible construir un mundo artificial donde interactúen organismos artificiales y cumplan los mismos principios que los sistemas naturales reales. Más aún, si existen leyes biológicas universales, entonces en ese mundo artificial deberían surgir los mismos comportamientos que en el mundo natural de forma espontánea y, por tanto, ser un mundo antural en sí mismo.*» La conjetura anterior requiere de comprobación y requiere de encontrar un mecanismo para la construcicón de mundos artificiales que soporten todas las leyes biológicas descritas y por describir. En este sentido los Autómatas Celulares llaman poderosamente la atención

2.3.1 Ecuación de Difusión

El modelo de Enstein para el movimiento Browniano da origen a una ecuación diferencial parcial conocida hoy como la *ecuación de Difusión* tradicional:

$$\frac{\partial u}{\partial t} = \mathcal{D}\frac{\partial^2 u}{\partial x^2} \tag{2.4}$$

o, en el caso de tres dimensiones

$$\frac{\partial u}{\partial t} = \mathcal{D}\nabla^2 u \tag{2.5}$$

donde $\mathcal{D}$ es el coeficiente de difusión.

Así que en la parte espacial estamos habando de sitios vecinos y la ecuación de difusión puede reescribirse en forma discreta como

$$x[n, t+1] = x[n, t] + D(x[n+1, t] + x[n-1, t] - 2x[n, t])$$

donde hemos hecho $dt = h = 1$ desescalando el espacio y el tiempo. De este modo el nuevo estado del sitio n en el arreglo x depende de su estado anterior $x[n, t]$ y de los estados anteriores de sus vecinos:

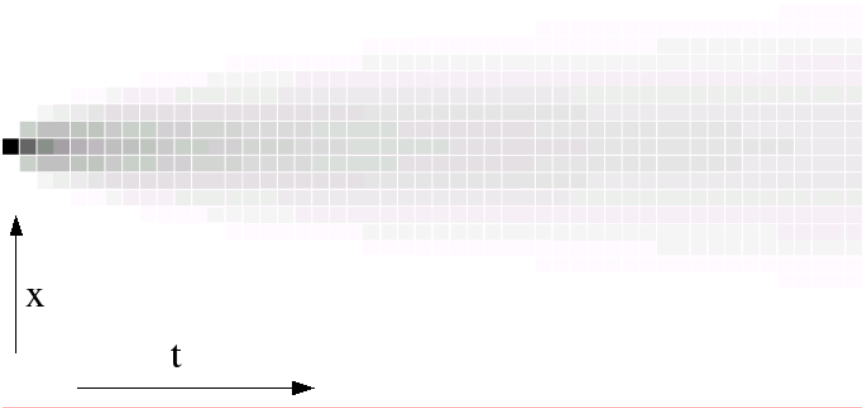


Figura 2.2. Solución de la Ec. de Difusión

2.4 Ejercicios

Ejercicio 2.2. Demostrar que el mapeo logístico tiene PF estables para $x^\star = \frac{K-1}{K}$ si $1 < K < 3$.

Capítulo 3
Azar y Determinismo

3.1 Introducción

Nosotros podemos inventar lo que sea. Como especie hemos construído y destruído ideas y objetos materiales a lo largo de nuestra historia. Es parte de nosotros como especie, de nuestra necesidad de crear

$$\sum_{i=1}^{n} p_i = 1 \tag{3.1}$$

3.2 Números seudo-aleatorios

Desde hace muchos años se busca un algoritmo capaz de generar una secuencia de números que satisfagan una condición de aleatoriedad (una prueba estadística) y que tengan un *periodo* suficientemente grande como para poder hacer cualquier tipo de simulación o cálculo.

Una de las formas más comunes de generar números cuyas secuencias nos permitan *simular situaciones en las que la ignorancia predomina, es la siguiente*: Existen varias fórmulas para obtener una secuencia de números aleatorios, una de las más sencillas es la denominada fórmula de congruencia: se trata de una fórmula iterativa, en la que el resultado de una iteración se utiliza en la siguiente para generar un nuevo número de la secuencia: x=(a*x+c) mod m

$$x = (a\,x + c) \bmod m$$

donde *a, c, m*, son constantes cuyos valores elige el creador de la rutina, así por ejemplo tenemos {a=24298 c=99491 m=199017} o {a=899 c=0 m=32768}.

3.3 Estimaciones simples usando `random()`

Estimación de volumen. Veamos un problema más o menos simple como lo es medir el volumen de una piedra. Este problema es 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 (o se derramó de) el volumen de agua que, finalmente, corresponde a la cantidad de agua que «cabe» en la piedra. Se trata entonces de hacer una analogía numérica con este procedimiento: 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.

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»). Si «dejamos caer» N puntos aleatorios *uniformemente distribuídos*[3.1] dentro del rectángulo, muchos de ellos caerán dentro del círculo y otros no. Si suponemos que n de los puntos lo logran, entonces tendremos $N - n$ fuera. De este modo llegamos a la siguiente conclusión: debe haber una proporción entre n y el área del círculo a igual a la proporción entre N y el área del rectángulo $A = 2R \times 2R = 4R^2$, es decir,

$$\frac{n}{a} = \frac{N}{A} \tag{3.2}$$

de donde se tiene que

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

3.1. Esto significa, a *grosso modo*, que todos los números generados tienen la misma probabilidad de salir. Requerimos que esto sea así por la situación real: no se considera que el agua en el recipiente tenga diferentes densidades en diferentes sitios dentro del recipiente. Eso complicaría el problema.

Nota 3.1. Esta técnica se usa sobre todo para evaluar integrales múltiples: uno «encierra» a la función a integrar en un hipercubo y genera números aleatorios uniformemente distribuídos contando, cada vez, si el punto está «dentro» o «fuera» de manera equivalente a lo que hicimos con la esfera (ver ejercicios).

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: OJOOJOOJOOJOOJO

Difusión. Veamos un problema de difusión más o menos elemental[3.2] en el cual se aplica esta idea: dos recipientes unidos por un agujero (o bien uno solo dividido en dos partes iguales). Inicialmente, situamos un conjunto de N moléculas de un gas en el recipiente A de la izquierda, tal como se ve en la figura. Al abrir la compuerta que comunica ambas mitades, observaremos que las moléculas van pasando desde la izquierda hacia la derecha hasta que se establece el equilibrio, en el que habrá aproximadamente $N/2$ moléculas en cada mitad. Este equilibrio es dinámico, ya que continúan pasando moléculas desde el primer al segundo recipiente y viceversa, pero en promedio el flujo neto es cero. A estas oscilaciones en el número de moléculas en torno al equilibrio les denominamos fluctuaciones. Podremos comprobar que el proceso inverso, en el que partiendo con la mitad de las moléculas en cada parte, es muy difícil, aunque no imposible, que todas las moléculas se agrupen en una de las dos mitades. Si el número total de moléculas es muy grande esta probabilidad decae más y más.

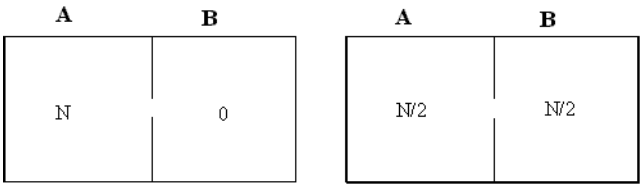


Figura 3.1. Estados inicial y final (promedio) de los dos recipientes conectados.

3.2. Más adelante y en el capítulo ? trataremos más ampliamente lo relacionado con fenómenos de difusión.

Supongamos (razonablemente) que la probabilidad de que una molécula pase de la mitad A a la mitad B es proporcional al número N_1 de moléculas en A, y por la misma razón, la probabilidad de que una molécula pase de la mitad B a la mitad A sea proporcional al número N_2 moléculas en B. Obviamente tenemos que asegurar la condición (3.1) así que $P_A = \frac{N_1}{N_1+N_2}$ y $P_B = \frac{N_2}{N_1+N_2}$ serán las probabilidades de que una partícula salga de A o de B respectivamente. Como las cantidaes N_1 y N_2 son dependientes (si una incrementa en una cantidad la otra disminuye igual) tomaremos como referencia el lado A y usaremos una variable aleatoria γ (obviamente uniforme en [0,1)) para decidir si una partícula pasa o no al lado B.

Este ejemplo tiene solamente dos valores de p_i, lo cual simplifica el problema en la cuestión de las probabilidades acumuladas: $0 \leqslant \gamma < p_1$ o bien $p_1 \leqslant \gamma < 1$.

La idea es "generar" un número aleatorio γ y verificar si $0 \leqslant \gamma < P_A$. Entonces una partícula pasará al lado B. Es obvio que la otra parte será $P_A \leqslant \gamma < P_A + P_B$ (el otro intervalo) asociada con el evento del paso de una partícula del lado B al A.

Definiremos una clase que denominaremos *Difusion*, con cuatro miembros dato, el número `N1` de moléculas en el recipiente A, y el número `N2` de moléculas en el recipiente B, el total de moléculas NP y el tiempo t. Una función miembro denominada *evolucion()* calcula el número de partículas en cada recipiente en función del tiempo, de acuerdo con el resultado del sorteo de una variable aleatoria γ uniformemente distribuida en el intervalo [0, 1). En el constructor se establece el estado inicial del sistema: `N1` de moléculas en el recipiente A, y `N2` de moléculas en el recipiente B. Se pone el contador de tiempo t a cero.

- **Difusión entre dos cajas usando *objetos***

```
from random import *
```

```
class Difusion:  # clase

   def __init__(self, N1, N2):  # constructor
      self.N1 = N1
      self.N2 = N2
      self.NP = N1 + N2

   def evolucion(self,dt): # dinámica de difusión
      p = float(self.N1)/float(self.NP)
      gamma = random()
      if gamma < p:
         N1, N2 = N1-1, N2+1
      else:
         N1, N2 = N1+1, N2-1
      self.t = self.t+dt
```

Que nos permitiría ver cómo evoluciona la cantidad de partículas en A y en B si enviamos la información a un archivo de datos.

- mientras que **el programa** `python` quedaría más o menos así:

```
from difusion import *

N = 500
dt = 10
dif = Difusion(N,0)
print ‘‘tiempo  izquierda  derecha’’
n1 = N;
i = 0;
# estado inicial:
print i*dt, "  -  ", n1, "  -  ", N-n1

# la secuencia
for i in range(200)
   n1 = dif.evolucion(dt);
   print i*dt, "  -  ", n1, "  -  ", N-n1
```

En la primera línea se supone que la clase `Difusion` se encuentra en el archivo `difusion.py`

• •

3.4 Estructuras Recursivas y Fractales

3.5 Movimiento Browniano y Difusión

Sabemos que el movimiento Browniano se debe a las interacciones de una partícula pequeña con las moléculas de un fluido que vienen de diferentes direcciones, en cualquier cantidad e intervalo. Lo que nosotros podemos observar es el efecto neto de este conjunto de interacciones: desplazamientos de tamaño aleatorio X_i en cada intervalo Δt. En el caso unidimensinal podemos pensar en desplazamientos de tamaño Δx a la derecha o izquierda de la posición del momento con la misma probabilidad $1/2$. Es decir: nuestras variables aleatorias X_i pueden tomar valores $\pm\,\Delta x$ con igual probabilidad y todas la X_i son estadísticamente independientes.

El desplazamiento neto de la partícula será la suma de todas estas variables aleatorias después de n intervalos de tiempo

$$X = \sum_{i=1}^{n} X_i \tag{3.3}$$

Este es el modelo de *camino aleatorio* para el movimiento Browniano. De acuerdo con este modelo todos los promedios son

$$< X_i > = 0$$

dado que, de acuerdo con lo dicho, para cada i se tiene $< X_i > = (1/2)\Delta x + (1/2)(-\Delta x) = 0$. El teorema de la suma promedio indica entonces que

$$< X > = \sum_{i=1}^{n} < X_i > = 0 \tag{3.4}$$

es decir, el desplazamiento medio tiene promedio nulo. La desviación media cuadrática o variancia de X será

$$\text{var}\{X\} = \text{var}\{X_1\} = \text{var}\{X_2\} = ... = \Delta x^2 \tag{3.5}$$

Un ejemplo de partículas Brownianas en `python` usando la librería `Tkinter`:

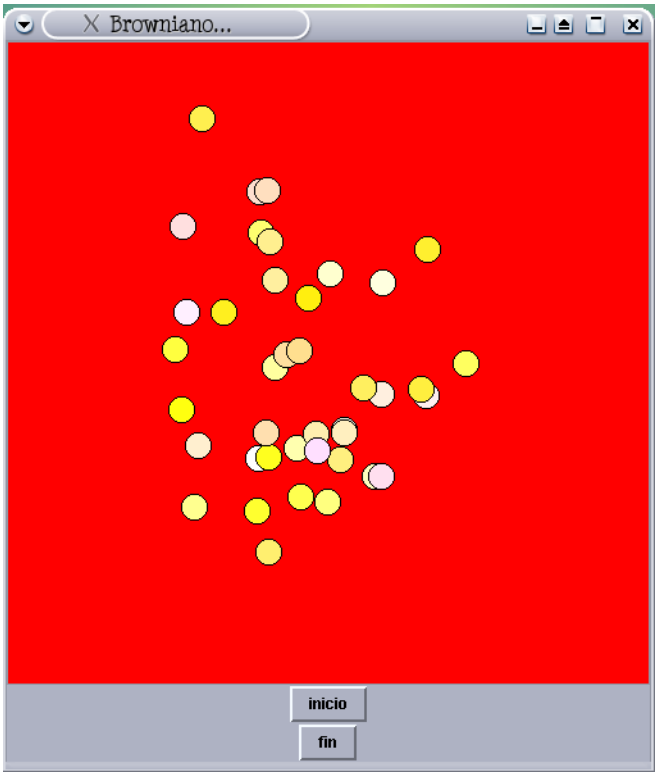


Figura 3.2. Imagen del programa `browniano.py`

- El código de `browniano.py` «simple» con la condición de que los pasos no miden Δx (toman valores continuos entre -5 y +5):

```
#!/usr/bin/python
#-*- coding: iso-latin-1 -*-
from Tkinter import *
from random import *
from Lcolores import *

def mover():
  for i in range(40):
    dx=-5+10*random()
    dy=-5+10*random()
    c.move(p[i],dx,dy)
  w.after(100,mover)

init_colorsD()
seed()

w=Tk() # modo gráfico
w.title('Browniano...')
c=Canvas(w,width=500,height=500,bg='red')
c.pack()
b1=Button(w,text='inicio',command=mover).pack()
b2=Button(w,text='fin',command=w.destroy).pack()

MEDIA,SIGMA = 250, 20 # las partículas
p =[0]*40
for i in range(40):
  x = gauss(MEDIA, SIGMA)
  y = gauss(MEDIA, SIGMA)
  p[i]=c.create_oval(x-10,y-10, x+10, y+10,
                     fill=colors[i])

# llamado principal
w.mainloop()
```

- ... y usando objetos:

```
#!/usr/bin/python
```

```
#-*- coding: iso-latin-1 -*-

from Tkinter import *
from random import *
from Lcolores import *

class Browniano: # definición de la clase

  def __init__(self): # constructor init_colorsD()
     seed()
     self.w = Tk() # parte gráfica
     self.w.title('Browniano...')
     self.c = Canvas(self.w,width=500,height=500,bg='red')

     self.c.pack()
     self.b1 = Button(self.w,text = 'inicio',
                      command = self.mover).pack()
     self.b2 = Button(self.w,text = 'fin',
                      command = self.w.destroy).pack()

     MEDIA,SIGMA = 250, 20 # las partículas
     self.p = [0]*40
     for i in range(40):
        x = gauss(MEDIA, SIGMA)
        y = gauss(MEDIA, SIGMA)
        self.p[i] = self.c.create_oval(x-10,y-10,x+10,y+10,
                                       fill=colors[i])
  ### __init__

  def mover(self): # la dinámica
     for i in range(40):
        dx=-5+10*random()
        dy=-5+10*random()
        self.c.move(self.p[i],dx,dy)
     self.w.after(100,self.mover)
  ### mover
```

```
### fin declaración de clase


# ‘‘programa’’ principal
brown = Browniano() # se declara el objeto
brown.w.mainloop()  # llama a ejecución
```

• •

El teorema de la suma de variancias para variables estadísticas independientes lleva finalmente a que $< X^2 > = n\Delta x^2$ y como el tiempo total debe ser $t = n\Delta t$ entonces

$$< X^2 > = \left(\frac{\Delta x^2}{\Delta t} \right) t \tag{3.6}$$

que es la característica central del movimiento Browniano: la variancia del desplazamiento neto X es proporcional al tiempo que dura el desplazamiento.

Existen muchas formas de generalizar este modelo para el movimiento Browniano. Las más simples consistirían en ampliar la dimensionalidad del modelo, que los pasos Δx sean diferentes en diferentes direcciones o bien que sean diferentes las probabilidades en cada dirección.

3.6 Procesos Estocásticos. Montecarlo

Los métodos de simulación se están usando cada día más en diferentes ámbitos de la vida: la aplicación de estos métodos para resolver modelos físicos formales, la modelación de un sistema de cajas registradoras en una tienda comercial, la dinámica de colonias de insectos o de bacterias, crecimiento de tumores o dinámica del corazón, vida artificial, inteligencia, adaptación y evolución, etc. etc.

En el área de la física existe un método de simulación conocido como *Dinámica Molecular* cuya idea central consiste en resolver las ecuaciones de movimiento de todas las partículas que formen el sistema. Supongo que al mencionar esto ya hemos dicho dónde y cómo se aplica, sin embargo tiene sus detalles formales y prácticos (algún autor lo denomina el *arte de la simulación*).

Otro de los métodos de gran interés, precisamente por la gran cantidad de aplicaciones que tiene, es el llamado *método de Montecarlo* en el cual nos vamos a centrar: una introducción a este método y a sus bases formales para ser utilizado en *sistemas estadísticos en equilibrio*. Después vamos a discutir brevemente algunos elementos *procesos estocásticos* que, por cierto, están más relacionados con las ideas de *sistemas complejos*, *irreversibilidad*, *entropía* y cosas por el estilo. Trataremos de reformular algunos problemas clásicos en el lenguaje de las variables aleatorias.

Una *variable aleatoria*[3.3] se puede entender como una magnitud que, bajo ciertas condiciones, puede tomar un conjunto (finito o no) de valores dentro de un intervalo (también finito o no). "No importa si la variación al azar es intrínseca e inevitable o un artefacto de *nuestra ignorancia*" [4]. Una característica de estas variables es el hecho de que cada uno de los valores que puede tomar tiene asociada una probabilidad (o bien, en el caso de variables aleatorias *continuas*, lo que se tiene es una función densidad de probabilidad cuya integral en un intervalo define la probabilidad correspondiente).

Uno puede definir una variable aleatoria por medio de una tabala de valores como sigue

$$X = \begin{pmatrix} x_1 & x_2 & \dots & x_n \\ p_1 & p_2 & \dots & p_n \end{pmatrix} \tag{3.7}$$

3.3. Mucho del material aquí expuesto se puede consultar con más detalle en la referencia [ ] o en la página *http://www.sc.ehu.es/sbweb/fisica/cursoJava/numerico/montecarlo/montecarlo.htm*, que son unas notas basadas en el mismo texto orientadas al lenguaje `java`. Algunas figuras fueron tomadas de ahí.

de modo que se asume que la variable aleatoria X puede tomar uno de los n valores x_i y cada valor tiene la probabilidad de ocurrir p_i. donde, como siempre,

$$\sum_{i=1}^{n} p_i = 1 \tag{3.8}$$

$$\sum_{i=1}^{n} p_i = 1 \tag{3.9}$$

Un ejemplo más o menos obligatorio -y obvio- es el de un dado que tiene 6 resultados posibles {1,2,3,4,5,6} y todos son igualmente probables con probabilidad 1/6. Como una experiencia podemos hace girar una ruleta y apuntar el número del sector que coincide con la flecha (ver figura). En la ruleta de la izquierda de la figura los resultados posibles son {0, 1, 2, 3, 4, 5, 6, 7}, y la probabilidad de cada resultado es 1/8, es decir,

$$X_1 = \begin{pmatrix} o & 1 & \dots & 7 \\ \frac{1}{8} & \frac{1}{8} & \dots & \frac{1}{8} \end{pmatrix}$$

En la ruleta de la derecha de la figura los posibles resultados son {0, 1, 2, 3}, pero las probabilidades respectivas ya no son iguales. Podemos pensarlas proporcionales al ángulo del sector y

$$X_2 = \begin{pmatrix} o & 1 & 2 & 3 \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{8} & \frac{1}{8} \end{pmatrix}.$$

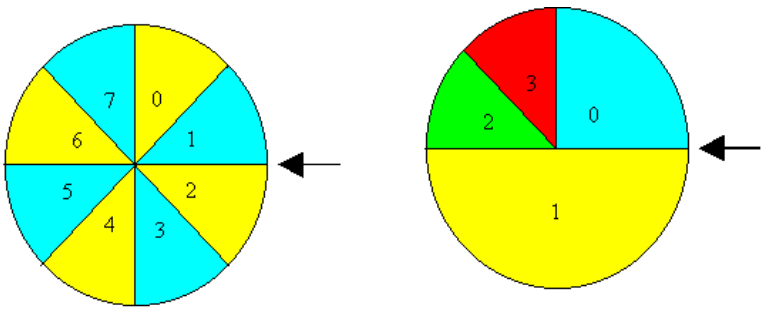


Figura 3.3. Representación gráfica de las variables aleatorias ζ_1 y ζ_2.

Los dos primeros ejemplos se refieren a variables aleatorias *uniformemente distribuídas*. Aun cuando son *discretas* nos referimos explícitamente al hecho de que tienen la misma probabilidad de tomar cualquiera de sus valores posibles.

En general, para simular un proceso físico, o hallar la solución de un problema matemático, es necesario usar una gran cantidad de números aleatorios los cuales (para nuestros fines y usando nuestros medios) serán obtenidos por medio de algún algoritmo. De manera práctica resulta conveniente emplear los denominados números pseudoaleatorios: se trata de números que se obtienen a partir de un número denominado *semilla*, y la aplicación reiterada de una fórmula, obteniéndose una secuencia de números $\{z_1,\ z_2,\ ...,\ z_n\}$ que imitan bastante bien[3.4] los valores de una variable *uniformemente distribuida* en el intervalo [0, 1). De esto ya hablamos cuando describimos la función `random()` que se usa en muy diversas aplicaciones (aquí mostramos algunas).

Tenemos ahora el problema de cómo simular la ruleta situada a la derecha de la figura usando una función del tipo `random()`. ¿Cómo generamos valores con una distribución definida -no uniforme- a partir de una variable γ uniformemente distribuída? (Hay que notar que esto equivale a poder hacer cualquier distribución a partir de `random()`).

Hay una forma de hacerlo procediendo del siguiente modo: se hallan las probabilidades de cada resultado, proporcionales al ángulo de cada sector y se apuntan en la segunda columna (ver la tabla abajo), la suma total debe de dar la unidad. En la tercera columna, se escriben las probabilidades acumuladas:

x_i	p_i	p acum.
0	0.250	0.250
1	0.500	0.750
2	0.125	0.875
3	0.125	1.000

(3.10)

3.4. Por lo menos satisfacen los requerimentos formales de aleatoriedad.

De la tabla y pensando en la variable aleatoria γ que -sin decirlohemos definido como *uniformemente distribuída en [0,1)*, parecería más o menos obvio que cada valor de γ definirá un valor de X_2 dependiendo del valor que tome y de "dónde caiga" con respecto a las probabilidades acumuladas de la tercera columna de (1.3).

Otra forma de verlo es la siguiente: en el eje Y están las probabilidades acumuladas (columna 3) y en el eje X los valores de X_2 (columna 1 de la tabla). Los valores de γ (uniformemente distribuídos sobre el eje Y) definen valores x_i para X_2:

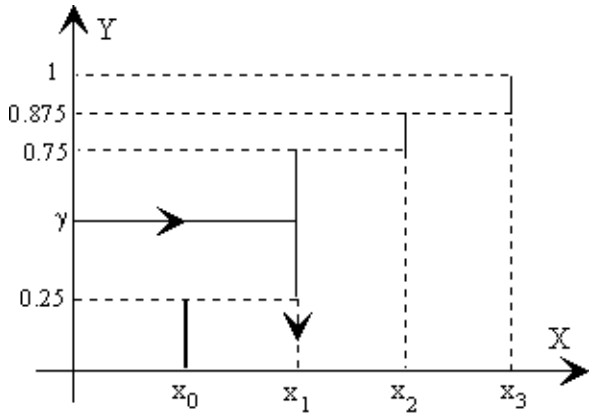


Figura 3.4. Un valor de γ define el valor que tomará ζ_2 (en eje Y).

En la figura se ilustra cómo un valor de γ define el valor que tomará ζ_2 (en eje Y). Las líneas continuas verticales «suman» comparando su valor con las probabilidades acumuladas $\sum p_i$.

Puede verse que si $0 \leqslant \gamma < 0.25 = p_0$ el valor de X_2 será x_0. De la misma manera cuando consideramos $0.25 \leqslant \gamma < 0.75 = p_0 + p_1$, a X_2 le corresponderá el valor $x_1 = 1$ (el mostrado con la flecha) y, para el caso en el que $p_0 + p_1 = 0.75 \leqslant \gamma < 0.875 = p_0 + p_1 + p_2$, X_2 será x_2. En resumen, los valores x_i que va adquiriendo X_2 se obtienen a partir de γ bajo la condición[3.5]

$$\sum_{j=0}^{i-1} p_j \leqslant \gamma < \sum_{j=0}^{i} p_j \tag{3.11}$$

Tenemos que notar aquí que la relación (3.11) define el índice i y con él el valor de la variable aleatoria X, es decir, si el número aleatorio γ cumple la relación (3.11) para algún i, entonces X tomará el valor x_i. Esta relación permite definir el algoritmo para el caso continuo, es decir, para obtener números aleatorios con una distribución continua específica a partir de γ: si X es una variable aleatoria continua definida en el intervalo $[a,b]$ y $p(x)$ es su densidad de probabilidad[3.6] entonces la probabilidad de que X tome un valor entre a y x será

$$P\{a \leqslant X < x\} = \int_a^x p(x)dx$$

que es el equivalente de la probabilidad acumulada (3.11) y entonces la variable uniforme γ nos puede decir qué valores x toma X si resolvemos la ecuación

$$\gamma = \int_a^x p(x)dx \tag{3.12}$$

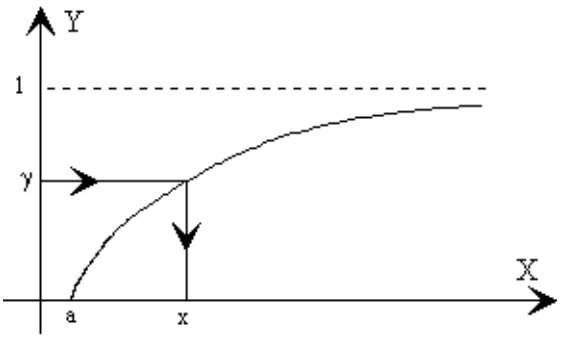


Figura 3.5. Valor x de la variable ζ determinado por los valores de γ.

3.5. Obviamente la sumatoria de la izquierda pierde sentido si $i - 1 < 0$, entonces se toma como *cero*.

3.6. Es decir que la probabilidad de que $a' \leqslant \zeta \leqslant b'$ (dentro de $[a,b]$) está dada por $P\{a' \leqslant \zeta \leqslant b'\} = \int_{a'}^{b'} p(x)dx$.

Por ejemplo, una variable uniformemente distribuida en $[a,b)$ tendrá la densidad de probabilidad $p(x) = \frac{1}{b-a}$ (constante[3.7]) para $a \leqslant x < b$. La integral (3.12) será

$$\gamma = \int_a^x \frac{dx}{b-a} = \frac{x-a}{b-a}$$

de donde

$$x = a + \gamma(b-a)$$

que es un resultado discutido en el Apéndice correspondiente a `gnuplot` como analogía con expresiones paramétricas: si $\gamma \in [0,\ 1)$ podría considerarse, con buena aproximación, que toma el lugar de un parámetro $t \in [0, 1]$ y, por lo tanto, que los puntos que se generan con los dierentes valores de γ también se generan con t o viceversa.

Por cierto, en la sección 3.3 se discutió el problema de la difusión con un enfoque particular que implicaba solamente la dinámica de partículas para ir de un lado al otro. De más de una manera este caso ocurre si, y sólo si el sistema, cerrado, tiende a un estado en el que su densidad es uniforme en todo el volumen. este resultado tiene una justificación en términos de probabilidad y conteo: entendamos un *macroestado* como en estado en el que el sistema está aislado y tiene N partículas (distribuídas en A y en B). Un *microestado* es un estado particular, uno posible, en el que se tienen N_1 partículas de gas en el lado A y N_2 en el lado B. ¿Cuántos microestados diferentes hay con esta condición? o, en otras palabras ¿De cuántas formas puedo acomodar N_1 partículas en A, y N_2 en B, si tengo $N{=}N_1 + N_2$? Esta pregunta es equivalente a preguntar cuántos subconjuntos de tamaño N_1 puedo formar a partir de las N partículas (obviamente el resto de partículas siempre será $N - N_!$ y estarán en B). La cantidad está dada por

$$\binom{N}{N_1} = \frac{N!}{N_1!(N-N_1)!}$$

3.7. El valor $p(x) = \frac{1}{b-a}$ está dado por la condición (3.9).

que es la *combinatoria* de N tomadas de a N_1 a la vez (ver ejercicio 3.13).

3.7 Distribuciones Teóricas y Reales

3.8 Ejercicios

Ejercicio 3.1. Modificar el programa para "llenar" un paralelogramo $[a, b] \times [c, d] \times [e, f]$. Hacer la gráfica 3D.

Ejercicio 3.2. 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).

Ejercicio 3.3. 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 3.4. Mostrar, a partir de la relación (3.2) para el caso 3D, y escribiendo explícitamente los volúmenes a y A, que se puede hacer una estimación de π con la expresión $\pi \cong \frac{6n}{N}$.

Ejercicio 3.5. 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 $\mathcal{O}(1)$.

Ejercicio 3.6. 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 3.7. ¿Podrías sugerir un algoritmo, usando `random()`, para evaluar una integral a partir de las sumas de Riemann?

Ejercicio 3.8. Pensemos en el átomo de H. Tiene un núcleo y normalmente un electrón. De acuerdo con la teoría cuántica este electrón ocupa una región espacio energética con alguna probabilidad dada por $|\psi|^2 = \psi\,\psi^\star$. Si pudiésemos trazar una esfera con centro en el núcleo, cuyo radio fuese el valor más alto de la coordenada $r(\theta,\ \phi)$ en coordenadas esféricas, podríamos evaluar el volumen relativo que ocupa el electrón dentro de esta esfera. Buscar la solución del átomo de Hidrógeno (en cualquier libro clásico de Mecánica Cuántica) y evaluar esta fracción de volumen.

Ejercicio 3.9. Verificar la expresión 3.5 usando la definición de desviación media cuadrática.

Ejercicio 3.10. Programa para generar una distribución de valores con densidad de probabilidad $p(x) = e^{-x}$ en el intervalo $[0, \infty]$. Verifivar gráficamente la distribución. Esto implica generar un *histograma* de frecuencias: un arreglo con el número de puntos que tienen valores dentro de un subintervalo. (Usar la idea de (3.12): no se requiere construir una tabla de probabilidades).

Ejercicio 3.11. Un pequeño reto consiste en construir la función *Combinatoria(N,n)* optimizada: resulta que para valores grandes de N los factoriales se vuelven números gigantes y suele pasar que no se puedan calcular en la computadora. En el caso de las combinaciones se trata de un cociente de factoriales que permite eliminar productos (y operaciones también). Algo importante en la construcción de la función es que no conviene usar tipos `int` o `long int`: no son suficientes. La idea consiste en hacer el menor número de operaciones usando el hecho de que $\frac{N!}{n!(N-n)!} = \frac{(n+1)\cdot(n+2)\cdot\ldots\cdot N}{1\cdot 2\cdot\ldots\cdot(N-n)}$ y de que hay el mismo número $N-n$ de factores arriba y abajo (lo cual permite que se pueda evaluar la expresión como un producto de cocientes que son mucho más manejables que los factoriales "completos").

Ejercicio 3.12. Adicionalmente a la función del problema anterior se le puede agregar, en términos de optimización (menos cálculos y menos tiempo de ejecución), la forma de elegir de manera adecuada el factor en el denominador que será "eliminado". En el ejercicio eliminamos $n!$ del denominador y quedan $N-n$ factores. Si n es pequeño (comparado con N) quedan *muchos* factores...

Ejercicio 3.13. Una vez que tenemos una función construida que evalúe las combinatorias podemos construir un histograma que, precisamente, nos va a mostrar la forma de la distribución gaussiana (más y más pronunciada para valores de N "grandes", por ejemplo para $N = 500$ el máximo es del orden de 1.1674E+149). La distribución que vas a observar justifica el postulado de *igual probabilidad a priori*: los valores más probables (los microestados que aparecen más veces en la evolución del sistema) lo son por muchos órdenes de magnitud. Así uno puede pensar que, en promedio, el sistema «está» en ese estado. Hay que graficar la combinatoria contra N_1, como muestra la gráfica. Hay que notar que el microestado que aparece muchísimas más veces que los demás tiene 250 partículas en cada lado, y es del orden de 1.2×10^{149} contra prácticamente cero para valores fuera del rango 200-300. Por ejemplo el microestado con 225 partículas en el lado A aparece del orden de 4×10^{148} veces y sigue cayendo de manera abrupta ¿Qué pasará con 10^{23} partículas?

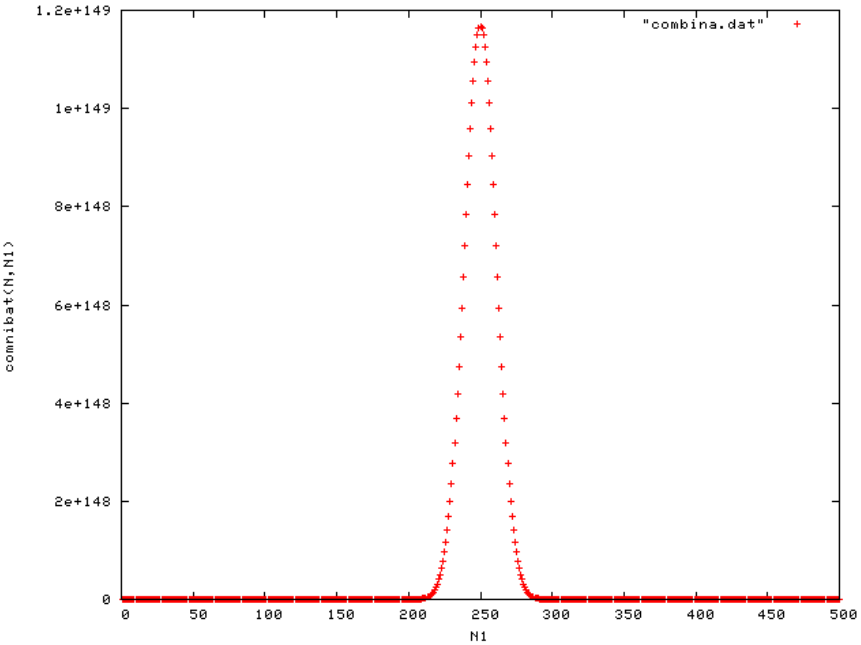


Figura 3.6. El microestado con mayor frecuencia de ocurrir tiene 250 partículas/lado

Capítulo 4
Geometría de la Naturaleza

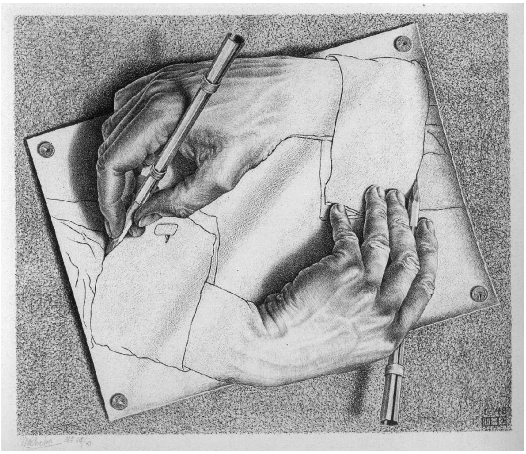
La naturaleza es rica en patrones que se forman de manera natural. Desde las nubes o las manchas en las pieles de diferentes animales, las imágenes sobre los caracoles o las huellas digitales. Las estructuras de árboles, venas, neuronas, ríos, etc.

4.1 Introducción

algo de historia, thom etfc

4.2 Estructuras Recursivas

Allá por los años 1980 se puso en boga la idea de la recursividad en lo referente a las estructuras de programas de cómputo. La recursividad está relacionada con la idea de una función o un programa que se aplique sobre sí mismo, que se pueda llamar desde *dentro* y ejecutarse.



Se establecieron nexos entre la música de Bach, los dibujos recursivos de Escher y estas estructuras algorítmicas. Se habla de estructuras lingüísticas, axiomáticas e incluso de enzimas y estructura del ADN[4.1]. Aunque no se trata aquí de profundizar en el tema, proba-

4.1. Gödel, Escher, Bach An Eternal Golden Braid. Reseña del libro de Douglas R. Hofstadter, por Manuel de la Herrán Gascón.

¿Puede un sistema comprenderse a sí mismo? Investigar este misterio es una aventura que recorre la matemática, la física, la biología, la psicología, y, muy especialmente, el lenguaje.

Sorprendentes paralelismos ocultos entre los grabados de Escher y la música de Bach nos remiten a las paradojas clásicas de los antiguos griegos y a un teorema de la lógica matemática moderna que ha estremecido el pensamiento del siglo XX: el de Kurt Gödel.

¿Es posible definir que es la evidencia? ¿Es posible formular leyes que indiquen cómo asignar un sentido a las situaciones? Es probable que no, pues toda regulación rígida tendría, indudablemente, excepciones, y no reglas [...] Entonces, si después de todo la evidencia es algo tan intangible, ¿por qué estoy tan prevenido contra formas nuevas de interpretación de la misma? [...] Todos los teoremas limitativos de la metamatemática y de la teoría de la computación insinúan que, una vez alcanzado determinado punto crítico en la capacidad de representar nuestra propia estructura, llega el momento del beso de la muerte: se cierra la posibilidad de que podamos representarnos alguna vez a nosotros mismos de forma integral. El Teorema de la Incompletitud de Gödel; el Teorema de la Indecibilidad, de Church; el problema de la Detención, de Turing; el Teorema de la Verdad, de Tarski: todos ellos tienen las resonancias de ciertos antiguos cuentos de hadas, advirtiéndonos que "perseguir el autoconocimiento es iniciar un viaje que... nunca estará terminado, no puede ser trazado en un mapa, núnca se detendrá, no puede ser descrito".

blemente se entienda más la idea si retomamos de la referencia [3] un ejemplo de *enunciados autorreferenciales*:

- Este enunciado contiene cinco palabras.
- Este enunciado carece de sentido porque es autorreferencial.
- Este enunciado sin verbo.
- Este enunciado es falso[4.2].
- El enunciado que estoy escribiendo es el enunciado que usted está leyendo.

De esta misma manera se pueden construir funciones (procedimientos, subprogramas) autorreferenciales o recursivos. En este caso el sentido es que hay un punto dentro del código donde se hace referencia al mismo. Un ejemplo muy común es la construcción recursiva de la función `factorial`. El primer caso es el más o menos obvio: un ciclo,

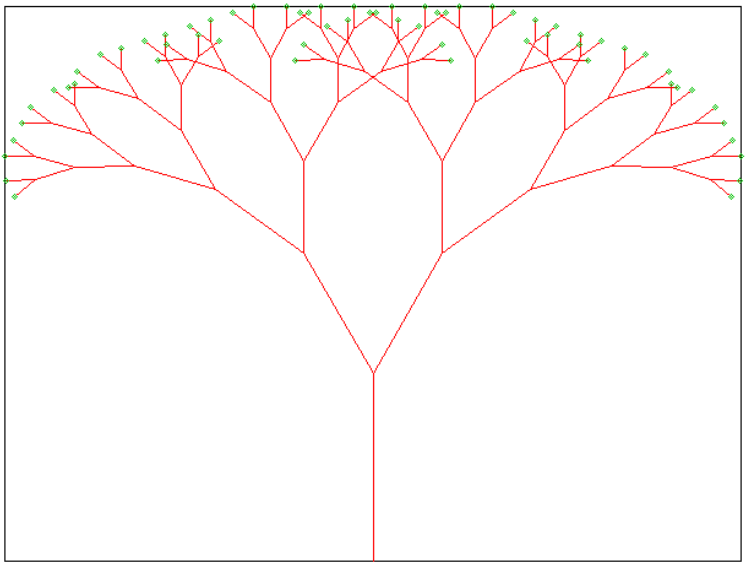
```
def factorial(n):
   if n <= 1: return 1
   fac = 1
   for i in range(n):
      fac = fac * i
   return fac
```

en el que la primera línea de la función evita operaciones innecesarias además de prevenir con la definición del cero factorial, no así con el caso de argumentos negativos, sin embargo lo que nos interesa por el momento es una versión *recursiva* de la función:

```
def factorial(n):
   if n <= 1: return 1
   else: return n*factorial(n-1)
```

De esta misma forma uno puede pensar en la estructura de las plantas, digamos, una planta sencilla que tiene un tronco y dos ramas simétricas en forma de `Y`. Decir a un niño que dibuje esto es muy sencillo y, si después de esto le decimos que cada rama tiene también sus ramitas y éstas a su vez un par de ramas más y más pequeñas, seguramente veremos algo más realista que esto

4.2. Paradoja de Epiménides.



que fue construído con un algoritmo recursivo, precisamente construyendo `Y`'s a partir de una primera línea que, en el lenguaje de las `IFS`[4.3], se llama *axioma*.

La idea es simple: necesitamos darle a nuestro algoritmo la información de dónde inicia la línea, cuál es su longitud y qué ángulo forma, por ejemplo, con la horizontal. Cuando se haga referencia a sí misma, tendrá que dar la información adecuada a las nuevas ramas:

```
from random import *
from math import *

def arbol1(xi, yi, ang, L, iter):
   xf = xi+L*cos(ang)
   yf = yi+L*sin(ang)
   # escribir las coordenadas de inicio y fin de línea
   print xi, yi
   print xf, yf
```

4.3. Iterated Function Systems.

```
    print          # más una línea en blanco
    if iter > 0:  # llamado recursivo:
       arbol1(xf, yf, ang+pi/6, 2*L/3.0, iter-1)
       arbol1(xf, yf, ang-pi/6, 2*L/3.0, iter-1)
    ###

# el programa:
longini = 50
anguloini = pi/2
Xini, Yini = 0, 0
iteraciones = 8
# llamado a la función recursiva
arbol1(Xini, Yini, anguloini, longini, iteraciones)
```

la línea en blanco después de escribir las coordenadas de inicio y fin de la rama permite, por un lado, separar cada rama de otra y, por otro lado, si uno grafica con `gnuplot` un archivo construído así, cada linea blanca implica un nuevo bloque de datos así que uno puede hacer una sola gráfica y cada rama será trazada de manera independiente (de otro modo serían unidas «punta» con «cola» y la figura sería una madeja indescibrable).

En un capítulo anterior discutimos sobre la razón de existir de los números «aleatorios» en los modelos (la ignorancia) y bueno, no será la excepción si ahora suponemos elementos aleatorios en algunas partes del «crecimiento» del árbol. Por ejemplo, sustituyendo estas líneas

```
if iter > 0:  # llamado recursivo:
      arbol1(xf, yf, ang+pi/6, 2*L/3.0, iter-1)
      arbol1(xf, yf, ang-pi/6, 2*L/3.0, iter-1)
```

por estas otras:

```
if iter > 0:  # llamado recursivo:
      r1, r2 = random(), random()
      arbol1(xf, yf, ang+pi/6+r1, 2*L/3.0, iter-1)
      arbol1(xf, yf, ang-pi/6+r2, 2*L/3.0, iter-1)
```

estamos «produciendo» viento en nuestro árbol original.

4.3 Sistemas de Reacción-Difusión

4.4 Formación de Patrones

4.5 Ejercicios

Apéndice A

Elementos de `gnuplot`

Una de las primeras cosas que uno espera al utilizar un graficador es, entre otras cosas, que nos permita hacer gráficos de manera sencilla y robusta, que nos permita manipular espacios, nombres, etiquetas, datos, formatos de impresión o de inclusión en documentos, etc.

El caso de `gnuplot` es otro proyecto de *software libre* o *freeware* de *GNU*. Y tiene, sin ostentación, las características que mencionamos arriba y que son suficientes para el trabajo[A.1]. En este mismo sentido se propone un formato que difiere mucho de ser un manual: más bien se trata de la construcción de ejemplos de los más simple a lo más complejo con la idea de que pueda, literalmente, *verse* el resultado de tal o cual acción.

A.1 Implementación directa (interactiva) de órdenes en `gnuplot`

Sin compromiso con la parte formal del software y/o de las definiciones y custiones técnicas, vamos a comenzar con algunos ejemplos

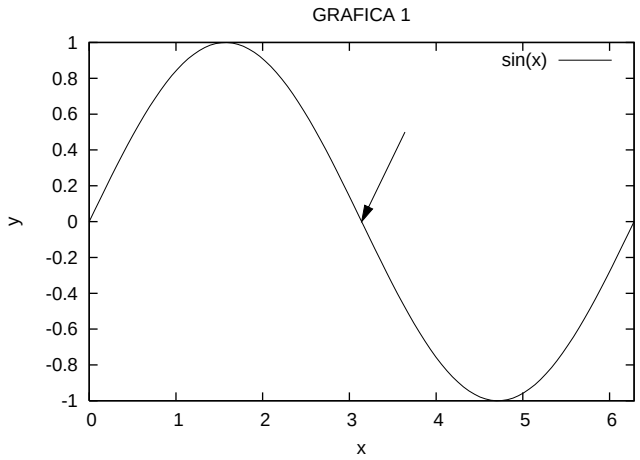
A.1. Existen libros completos, comerciales, como el de la referencia [2] *The computational beauty of nature*.

típicos simples y luego algunos más complicados.

Pensemos en una gráfica de la función $\sin(x)$ en un periodo completo en $[o, 2\pi]$ con un título «GRAFICA 1» y etiquetas adecuadas en los ejes. Nos ineteresa, además, señalar el punto al centro en el que la curva corta al eje[A.2] x:

```
   This is a TeXmacs interface for GNUplot.

GNUplot] set title "GRAFICA 1"
         set xlabel "x"
         set ylabel "y"
         set arrow from pi+0.5,0.5 to pi, 0.0 filled
         set size 0.7, 0.7
         plot [0:2*pi] sin(x)
```



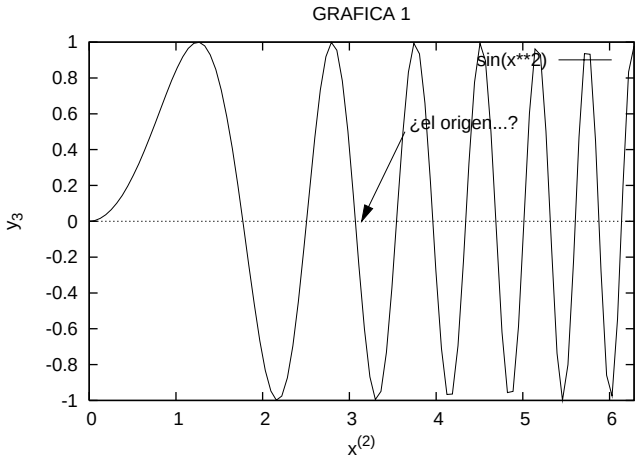
Una nueva versión modificada del caso anterior: esta vez vamos a acomodar un texto o *etiqueta* dentro de la gráfica, en donde señala la

A.2. En todas las gráfica incluiremos la línea `set size 0.7, 0.7`. El software con el que fue escrito el presente material es `TeXmacs` y permite insertar sesiones de gnuplot, octave, axiom e incluso Maple o Mathematica. En el caso de las gráficas de `gnuplot` éstas son generadas dentro del texto en una sesión especial así que, para efectos visuales, tenemos que incluir esa línea.

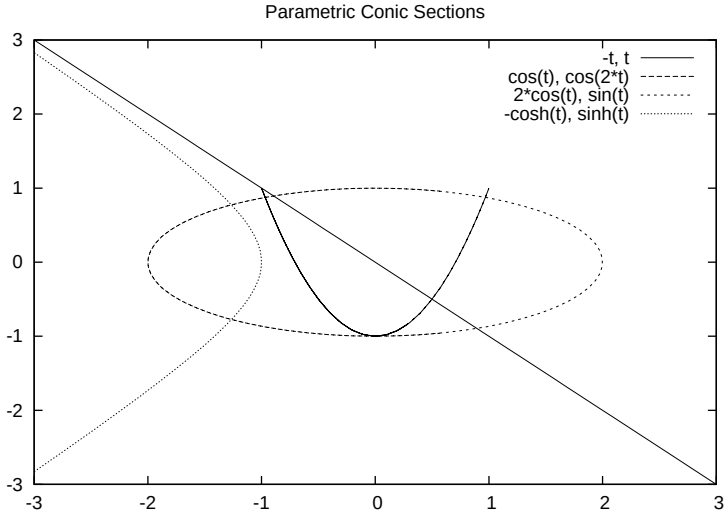
flecha. Además de que podremos ver los ejes y texto con acento y subíndices:

```
   This is a TeXmacs interface for GNUplot.

GNUplot] set title "GRAFICA 1"
         set xlabel "x^{(2)}"
         set ylabel "y_3"
         set arrow 1 from pi+0.5,0.5 to pi, 0.0 filled
         set label 1 "£el origen...?" at pi+0.55, 0.55
         set size 0.7, 0.7
         set zeroaxis
         plot [0:2*pi] sin(x**2)
```

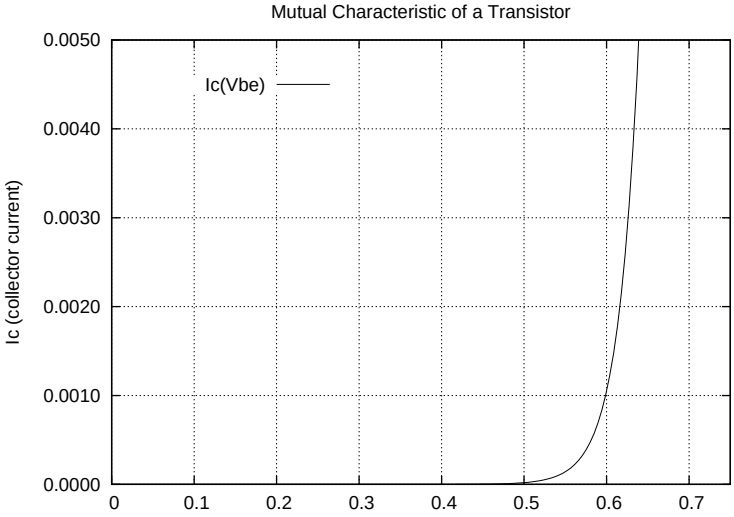



```
GNUplot] set xrange [-3:3]
         set yrange [-3:3]
         set size 0.8, 0.8
         set parametric
         set title "Parametric Conic Sections"
         plot -t,t,cos(t),cos(2*t),2*cos(t),sin(t),-
         cosh(t),sinh(t)
```



This is a TeXmacs interface for GNUplot.

```
GNUplot] # Transistor Bipolar (NPN)
         set size 0.8,0.8
         Ies = 4e-14
         Ico = 1e-09
         Ie(Vbe)=Ies*exp(Vbe/kT_q)
         Ic(Vbe)=alpha*Ie(Vbe)+Ico
         alpha = 0.99
         kT_q = 0.025
         set dummy Vbe
         set grid
         set offsets
         set nolog
         set nopolar
         set samples 160
         set title "Mutual Characteristic of a Transistor"
         set xlabel "Vbe (base emmitter voltage)"
         set xrange [0 : 0.75]
         set ylabel "Ic (collector current)"
         set yrange [0 : 0.005]
         set key .2,.0045
         set format y "%.4f"
         plot Ic(Vbe)
```



A.2 Modo programado de gnuplot

Apéndice B

Programación `python`

Este apéndice no pretende otra cosa que ser una guía mínima del lenguaje de programación *python*[B.1]. Se incluirán elementos del lenguaje que serán utilizados en los ejemplos y (seguramente) en los ejercicios propuestos.

La intención es manejar python como un lenguaje de programación «*natural*» (ya que se presta bastante bien). Tiene muchas características ventajosas con respecto a otros lenguajes de programación para quienes no son expertos e, incluso, para quienes lo son (puede programarse usando objetos o no aunque se recomienda que sí, no requiere declarar las variables con los tipos de datos correspondientes, los argumentos de las funciones no tienen tipo predeterminado, etc.). Es gratis y existe para todos los sistemas operativos, tienen una cantidad enorme de librerías y paquetes para construir aplicaciones (gráficos, acceso a páginas web o ftp, cálculos numéricos y científicos, modo interactivo muy simple de construir, etc.)

Se trata en fin de hacer una especie de *construcción* del lenguaje a partir de necesidades específicas que aparecen tanto en este apéndice como en el texto general. Los métodos numéricos aparecen aquí como ejercicios y ejemplos cuando no se implementaron en los distintos capítulos del texto.

B.1. Se puede bajar e instalar desde http://www.python.org

B.1 Implementación interactiva de tareas en python

La primera tarea que se acostumbra es la de escribir la frase "*¡hola, mundo!*", así que comenzaremos con ella. Al iniciar `python` en modo interactivo aparece el símbolo `>>>` que indica que `python` está esperando una orden[B.2].

```
>>> print ''¡hola, mundo!''
¡hola, mundo!
```

Comenzaremos por órdenes directas que nos muestren el «funcionamiento» general de `python`

```
>>> d = [2.32, 'pollito', -4, [1,2,3], 'hola']
>>> for k in d:
        print k
2.32
'pollito'
-4
[1, 2, 3]
'hola'
```

En este ejemplo `d` es una lista que muestra una vez más las diferencias entre `python` y otros lenguajes. Las listas son estructuras muy importantes en `python` y, como puede apreciarse, una lista puede albergar diferentes objetos (de diferentes tipos) a la vez.

- Hay que notar que abajo de la orden `for` hay una sangría: la orden `print k` está recorrida hacia la derecha con respecto a la orden `for`. Esta sangría es MUY IMPORTANTE en `python` pues define completamente la estructura de cualquier aplicación construida con este lenguaje[B.3].

B.2. `C++` no presenta la opción de funcionar de manera interactiva por ser un lenguajeque, a diferencia de `python` que es *interpretado*, se tiene que *compilar*.

Ejercicio B.1. Probar qué es lo que ocurre si la orden `print k` se recorre al mismo nivel (a la izquierda) que la orden `for`.

Ejercicio B.2. Probar agregando más *objetos* a la lista qué ocurre si se sustituye `print k` por `print 2*k` o bien por `print k**2`.

Ejercicio B.3. Probar la secuencias siguiente:

```
for k in d:
  print k
  print 2*k
```

y comparar, *fijándose en la sangría*, con la secuencia:

```
for k in d:
  print k
print 2*k
```

¿Cuáles son las diferencias entre os dos códigos?

A diferencia de la mayoría de lenguajes, `python` permite asignamiento múltiple. Veamos un ejemplo que pretende sumar dos cantidades dadas de manera fija (como *constantes*)

```
>>> a, b = 3, 5
>>> c, d = a + b
>>> print c, d
```

Una versión modificada podría hacer la situación más interactiva:

```
>>> a = input(‘‘dame a: ‘‘)
>>> b = input(‘‘dame b: ‘‘)
>>> c = a + b
>>> print c
```

B.3. No importa el tamaño de la sangría: es al gusto de quien programa, sin embargo no hay que olvidar que sangría *cero* no es sangría.

Este conjunto de órdenes ya podría considerarse un *programa*.

Operadores Aritméticos

operador	descripción	efecto
m+n	suma	aritmético
m-n	diferencia	”
m*n	producto	”
m/n	cociente	”
m**n	exponente	”
m>>n	desplaza n a la derecha	binario
m<<n	desplaza n a la izquierda	”
~m	complemento	”
m & n	AND binario	”
m | n	OR binario	”

Dado que `python` utiliza las listas como uno de sus objetos principales y básicos, veamos algunas formas de manipular listas así como algunas funciones elementales. Una lista se puede implementar *por construcción*[B.4]

```
>>> x1 = [j*0.1 for j in range(50)]
```

que es equivalente a

```
>>> x2=[]  # que es una lista vacía, sin elementos
>>> for j in range(50):
        x2.append(j*0.1)  # la función agregar de lista
```

probar ahora

B.4. Esto es una analogía a lo que se hace para describir conjuntos: se dan los elementos explícitamente o bien la descripción de los mismos.

```
>>> print x1
>>> print x2
```

Una lista de este tipo siempre es útil en trabajo numérico: uno siempre requiere una variable que toma valores en algún intervalo definido para poder, resolver, evaluar, modelar o simular eventos. Las dos listas anteriores contienen el mismo conjunto de valores, aunque la primera es más rápida en intuitiva de construir (además de requerir menos espacio y tiempo para escribirla). Una nota importante es que uno no necesariamente requiere un *lista* de valores (en el sentido de una estructura como las que `python` emplea), puede ser que simplemente requiera que esa lista pueda ser escrita y no que esté almacenada en la memoria. Hay más de una forma de hacer esto, incluso para escribir los datos en archivos que el propio programa genere :

```
for s in range(10):
  print 0.1*s, ''----'', ''0.1*''+str(s)
```

Aquí hemos suprimido el símbolo >>> en el entendimiento de que el código `python` puede ser escrito en forma interactiva (*dentro* de `python`) o bien en un archivo de texto que despés `python` leerá y ejecutará. En este ejemplo s irá tomando cada vez los valores de la lista `range(10)` (del 0 al 9) y la orden `print` escribirá el producto de la operación `0.1*s` para cada valor de `s`.

Ejercicio B.4. Implementar listas de las dos formas vistas antes (con la función `append()` y por construcción) con las siguiuentes secuencias:

i. -1.5, -2.5, -3.5, -4.5, -5.5, ..., -55.5

ii. 0.1, 0.2, 0.3, ..., 0.9

iii. 1., 0.5, 0.25, ... $1/2^n$, 20 términos

Ejercicio B.5. Repetir en inciso iii para n=10, 20, 30 y comparar el resultado de `sum(a)` con 2 (suponiendo que `a` es la lista generada)

B.2 Modo programado de `python`

Antes que nada hay que saber que uno puede hacer correr un programa...

Una cosa es hacer un programa elemental que efectúa un cálculo de manera secuencial o, incluso, una lista de cálculos (en la misma forma) 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. A partir de la información generada por un programa se puede tener acceso a otro tipo de información más ordenada, comprensible, resumida, asimilable por nosotros (estadística, por ejemplo).

En la realización de cualquier programa, más o menos elaborado, es necesario conocer las estructuras mínimas de programación con las que contamos (esto vale para 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*.

De manera esquemática podemos decir que un programa tiene dos tipos de estructuras: las *lógicas*[B.5] y las *de datos*. Junto con un adecuado *algoritmo* (la secuencia de pasos a realizar para el fin deseado), tendremos un programa útil.

Estructuras necesarias para construir un programa:

Estructuras lógicas:	Estructuras de Datos:
secuencia	escalares
decisión	
iteración	estructurados
modularidad	

B.5. Aquí nos referimos a las estructuras *propias* de la programación.

B.3 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[B.6], 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 las primeras «pruebas» en la sección B.1 podrías decirse que estábamos haciendo pequeños programas: pequeñas secuencias de órdenes que deberían cumplir con una tarea específica. Lo que hicimos fue una construcción en forma de *secuencia* de órdenes: desde el *inicio* de un programa, pasando por la *lectura* de datos, las operaciones necesarias para efectuar una suma, la escritura del resultado y la orden de *terminación*[B.7]. Es obvio aquí que la estructura básica que usamos para construir el programa fue la llamada *secuencia*. Sin embargo cuando construimos una lista de cualquiera de las dos formas vistas en B.1, además de que construímos una estructura de datos (la *lista*), usamos una estructura lógica de rogramación diferente: la *iteración*, representada en ese momento por la orden `for`.

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). OJO

B.6. 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.

B.7. Esta orden en muchos lenguajes es explícita, en otros como `C++` es implícita y en python es inexistente, al menos formalmente: las tareas terminan cuando no hay más líneas con órdenes.

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 y es el hecho de que cualquier problema se puede analizar y resolver por partes. En un programa cada una de esas partes puede ser implementada en la forma de un subprograma (llamado subrutina. procedimiento, función, módulo, etc. en diferentes lenguajes)

B.4 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.[B.8] 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.

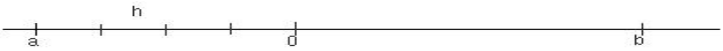
B.4.1 ¿Qué hace por "dentro" *gnuplot*?

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 corresponden, así como los títulos, etiquetas, tipo de coordenadas, etc.) hasta el simple mensaje de error.

B.8. Obviamente la estructura que rige por encima de todas es la secuencia: los programas siempre serán ejecutados en *secuencia*.

Hay dos cuestiones básicas en la estructura interna de *gnuplot* y son, por un lado, la evaluación explícita[B.9] 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 $a + n\,h = b$. 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[B.10] dado tenemos que evaluar el tamaño de "paso" como

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

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

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

o bien

$$x_1 = a,\ \ x_2 = x_1 + h = a + h,\ \ x_3 = x_1 + 2h, ...,\ \ x_n = x_1 + (n-1)\,h = b \tag{B.3}$$

B.9. *gnuplot* también hace gráficas a partir de archivos de datos, pero aquí nos referimos al caso de la evaluación interna de funciones.

B.10. El valor de n nos conviene poderlo definir de entrada. Las razones explícitas aparecen sobre todo con los métdos de integración numérica.

La expresión (B.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 (B.2) lleva los índices de 0 a $n-1$, pero los coeficientes de h coinciden con los índices de x_i así que, por esta razón, tomaremos como convención la dada por (B.2).

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

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

```
from math import *

nump = 2000

x = [0 for i in range(nump)] # vectores para los valores
y = x[:]                     # y es una copia del vector x
```

```
def g(algo):
   return sin(algo)*3.0 + exp(-algo)

a = input(``valor de a: '')  # lectura de intervalo
b = input(``valor de b: '')
h = float(b-a)/float(nump-1) # tamaño de paso

i = 0
while x[i] < b:
   x[i] = a + i*h
   y[i] = g(x[i])
   print x[i], y[i]
   i = i + 1
```

- **otra versión** `python`. Esta vez construimos las listas ya con los valores respectivos de `(x,y)` con la generación de un archivo de datos

```
from math import *

# definición del intervalo deseado
xI, xF = 2.0, 4.5

# número de puntos a evaluar
np = 200

# evaluar las divisiones del intervalo
h = (xF-xI)/(np-1)

# definimos la función que se evalúa
def g(algo):
   return sin(algo)*3.0 + exp(-algo)

# los valores de x en una lista:
x = [xI+j*h for j in range(np)]

# con estos valores construimos la lista y
y = [g(x[i]) for i in range(np)]

# escribir los datos en parejas
```

```
ff = open(''datos.dat'',''w'')     ### 1
for k in range(np):
   print x[k], y[k]  # a pantalla
   ff.write( ''%f  %f \n'' % (x[k], y[k]) )  ### 2

ff.close()  ### 3
```

Las líneas del programa que están numeradas (con un comentario a la derecha) son las necesarias para 1) crear un archivo de texto en el disco duro, 2) "usarlo" (escribir en él, leer de él o bien agregarle información) y finalmente 3) asegurar que el conjunto de datos que se guardan temporalmente en un *buffer*[B.11] queden todos en el archivo. En la línea 1 simultáneamente declaramos a `ff` como un «flujo» (*stream*: "chorro", "haz", "flujo") de salida a disco (`''w''`: write, archivo de escritura) y creamos el archivo "datos.dat" que a partir de este momento aparecerá en el directorio de archivos con tamaño 0 bytes[B.12]. La línea 2: es la que corresponde, para nuestro caso, en «usar» el archivo: escribirá sobre él con el mismo formato que aparece dentro de los argumentos de `write()` como una cadena de caracteres. `''%f %f...''` indica que serán escritos dos números de *punto flotante*, es decir, que tienen punto decimal (ver apéndice técnico). 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.

Ahora podemos entender la razón por la cual a veces un graficador puede mostrar curvas de funciones aun en la región donde la función tiene singularidades y otras veces no: todo depende de si los puntos x_i en los que la función es evaluada están muy cerca de (o coinciden con) la singularidad de la función. Este mismo problema se presenta en el programa anterior para funciones con puntos singulares.

B.11. 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.

B.12. Puedes usar la orden `ls -la *.dat` por ejemplo, que muestra todos los atributos de los archivos.

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. Este ejemplo tiene además la característica de no usar arreglos para los valores de x e y. En su lugar se usan únicamente dos variables (`xi` y `yi`) lo cual implicaría, para un programa que maneja muchos datos, un ahorro de memoria RAM:

- **El mismo programa en `python` 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):**

```
from math import *

# definición del intervalo deseado
xI, xF = 2.0, 4.5

# número de puntos a evaluar
np = 200

# evaluar las divisiones del intervalo
h = (xF-xI)/(np-1)

# definimos la función que se evalúa
def g(x):
   return sin(algo)*3.0 + exp(-algo)

# los valores de x en una lista:
x = [xI+j*h for j in range(np)]

# con estos valores construimos la lista y
y = [g(x[i]) for i in range(np)]

# escribir los datos en parejas
ff = open(''datos.dat'',''w'')
for k in range(np):
   print x[k], y[k]  # a pantalla
   ff.write( ''%f  %f \n'' % (x[k], y[k]) )  # a archivo
```

```
ff.close()  # cerrar archivo

# generación de script y ejecución de gnuplot
script = open(‘‘gnudatos.gp’’,’’w’’)
script.write(‘‘set title ’mi función’’)
scritp.write(‘‘set xlabel ‘x‘’’)
script.write(‘‘set ylabel ‘y‘’’)
script.write(‘‘set xrange [%f:%f]’’ % (a,b))
script.write(‘‘plot ‘datos.dat‘’’)
script.write(‘‘pause -1 ‘presione enter...‘’’)
script.close()
os.execl(‘‘gnuplot’’,’’gnudatos.gp’’)
```

B.5 Estructuras de datos

Es decir que no todos los datos que se manejan dentro de un programa son *escalares*[B.13], 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).

De manera particular en `python` se tienen los **datos tipo** *escalar* que son *enteros*, de *punto flotante*, *caracter* o *booleano*; y los datos con estructura interna que en `python` pueden reducirse a las *cadenas* de caracteres, las *listas* y los *registros* (que requieren de una librería especial).

B.13. Es decir, que tienen un valor simple entero, de punto flotante, caracter, booleano, etc.

B.5.1 Arreglos (vectores y matrices)

La primera estructura útil que hemos visto 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 $y_i = g(x_i)$. Es decir que mientras el programa se está ejecutándo (*corriendo*) vamos a tener todos los valores generados en un bloque que llamamos *arreglo* en RAM. Así como todos los tipos de datos *escalares* (sin estructura interna) tienen sus propias aplicaciones, los *arreglos* y todas las otras *estructuras* de datos tienen sus posibles aplicaciones. En el caso de los arreglos se pueden usar para obtener valores extremos o ceros de $g(x)$, promedios de las x_i o de los valores de y_i, para guardar información estadística o bien los valores de una o más matrices y/o vectores de diensiones diferentes, 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 los lenguajes de programación se contemplan estructuras de datos similares que, como ya dijimos, se llaman *arreglos* (así que de aquí en adelante *arreglo* es lo mismo que vector o matriz). La forma directa de manipular arreglos en casi cualquier lenguaje (incluyendo el matemático) es por medio de los subíndices.

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**

```
from math import *

# definición del intervalo deseado
xI, xF = 2.0, 4.5

# número de puntos a evaluar
np = 200
```

```
# evaluar las divisiones del intervalo
h = (xF-xI)/(np-1)

# definimos la función que se evalúa
def g(x):
   return sin(algo)*3.0 + exp(-algo)

# los valores de x en una lista:
x = [xI+j*h for j in range(np)]

# con estos valores construimos la lista y
y = [g(x[i]) for i in range(np)]

# escribir los datos en parejas
ff = open(''datos.dat'',''w'')
for k in range(np):
   print x[k], y[k]  # a pantalla
   ff.write( ''%f  %f \n'' % (x[k], y[k]) )  # a archivo

ff.close()  # cerrar archivo

# evaluación de promedios x_m, y_m
x_m, y_m = 0.0, 0.0
for i in range(np):
   x_m = x_m + x[i]
   y_m = y_m + y[i]
x_m = x_m/float(np)
y_m = y_m/float(np)
print ''Promedios: '', x_m, y_m
```

Sin embargo `python` cuenta con funciones para manejo de listas como `sum(x)` y `len(x)` de modo que el promedio del vector (o *lista*) `x` podría evaluarse simplemente con:

`x_m = sum(x)/float(len(x))`.

• •

Ejercicio B.6. Modificar el programa anterior para que encuentre (además de los promedio) los valores máximo y mínimo de los y_i.

Ejercicio B.7. Modificar el programa anterior para que evalúa los mismos promedios *sin* utilizar arreglos.

Ejercicio B.8. Verificar que el promedio de las x es $(a+b)/2$ y que el promedio de las y, `y_m`$= < y >$, debe satisfacer

$$< y >= \frac{1}{b-a} \int_a^b g(x)dx$$

Hay que tener en cuenta que las cantidades que nos arrojen los programas no son exactamente iguales (Hay que elegir una función cuya integral sea conocida, para fines de comparación).

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

.....

Índice de materias

Espacio Fase 7
Euler 12
Hamilton 7
Lagrange 7
Mecánica 7
PF 8
Punto Fijo 8
SD 8
Sistema Dinámico 8

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 (Computer Explorations of Fractals, Chaos, Complex Systems and Adaptation)*. The MIT Press, 2001.

[3] Douglas R. Hofstadter. *GÖDEL, ESCHER, BACH: una eterna trenza dorada*. CONCACyT, 1982.

[4] Don S. Lemons. *An Introduction to Stochastic Processes in Physics*. The John Hopkins University Press, 2002.

[5] Ricard V. Solé and Susanna C. Manrubia. *Orden y Caos en Sistemas Complejos*. Ediciones UPC. Cataluña, España, 1993.

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

[7] J. von Hardenberg, E. Meron, M. Shachak, and Y. Zarmi. Diversity of Vegetation Patterns and Desertification. *Phys. Rev. Lett.*, 87(19), 2001.