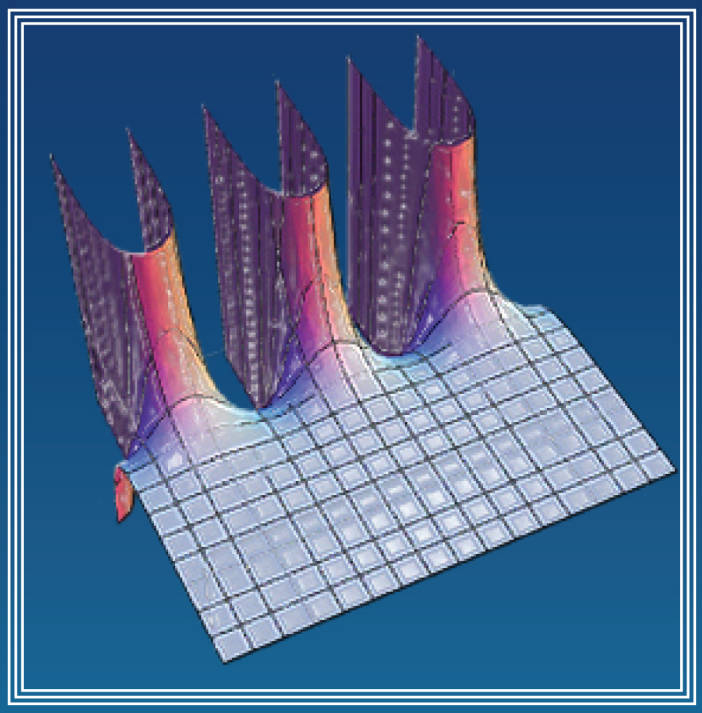
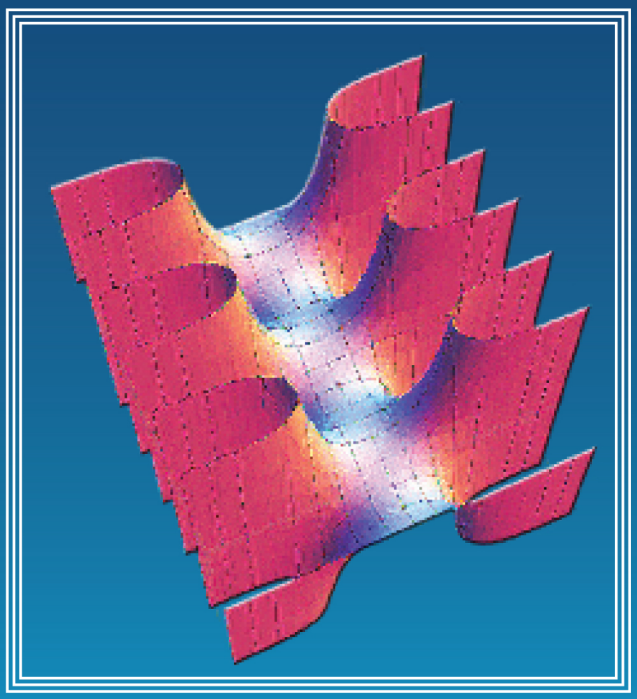
Colección CBI

Mathematica Esencial

con aplicaciones





Leonardo Dagdug
Orlando Guzmán

UNIVERSIDAD AUTONOMA METROPOLITANA
Casa abierta al tiempo
UNIDAD IZTAPALAPA División de Ciencias Básicas e Ingeniería

Mathematica esencial

con aplicaciones

Colección CBI

Mathematica esencial

con aplicaciones

Leonardo Dagdug Lima

Área de Mecánica estadística

UAM-Iztapalapa

Orlando Guzmán López

Área de Física de líquidos

UAM-Iztapalapa

Mathematica es un programa de cómputo desarrollado por Wolfram Research Incorporated.

UNIVERSIDAD AUTONOMA METROPOLITANA

Casa abierta al tiempo

UNIDAD IZTAPALAPA División de Ciencias Básicas e Ingeniería

L. D. L.
A Karina,
Fer, Rodri y Santi,
a Gustavo, Mariana,
Michelle y Stephanie.

A mis Padres

O. G. L.
Para León Máximo,
quien me enseñó a escribir
en ratos libres.

Primera edición, 2010

Primera reimpresión, 2016

Prolongación Canal de Miramontes No. 3855,
Col. Ex Hacienda San Juan de Dios.
Delegación Tlalpan C.P. 14387 México D.F.

Universidad Autónoma Metropolitana Unidad Iztapalapa
División de Ciencias Básicas e Ingeniería

ISBN 978-607-28-2107-1

ISBN Vol 978-607-28-2128-6

Impreso en México/*Printed in Mexico*

Agradecemos a M. V. Vázquez
la revisión del manuscrito.

O. G. L.
Agradece a los alumnos de
Física Computacional
de la UAM-I por sus
comentarios y críticas a
una versión preliminar del libro.

Contenido

Introducción

Mathematica como una lengua extranjera

Este libro fue escrito con la intención de ser útil a todas aquellas personas que desean acercarse por primera vez al programa *Mathematica,* conocer sus aspectos esenciales y aplicarlos con provecho. Por lo tanto, está dirigido por igual a estudiantes de licenciatura y posgrado, así como a científicos, ingenieros y otros profesionales técnicos.

Sin embargo, también fue nuestra intención desde un inicio dar oportunidad a aquellos que tienen un grado superficial de conocimiento sobre este software, para ampliar su dominio sobre los comandos de *Mathematica*, así como de sus paquetes de aplicaciones para cómputo simbólico, numérico y visualización. En particular, insistimos en incluir una introducción a la programación usando *Mathematica*, porque pensamos que la capacidad de combinar todos los aspectos de la computación científica en un sólo ambiente aporta a nuestros estudiantes y colegas ventajas enormes para adquirir, analizar y presentar información.

Cuando el programa *Mathematica* salió a la venta a finales de los años 80, se anunciaba como "un sistema para hacer matemáticas por computadora". Si por "hacer matemáticas" entendemos investigar e inventar nuevas ideas matemáticas, tal vez el anuncio sea exagerado; pero, por otra parte, si lo entendemos (a la manera de los físicos y los ingenieros) como "realizar y presentar cálculos" entonces *Mathematica* es bastante exitoso. No es que antes de *Mathematica* los físicos y los ingenieros no supieran cómo hacer cálculos con una computadora; los lenguajes de computación disponibles en ese entonces estaban orientados casi exclusivamente a hacer operaciones numéricas. Para realizar operaciones algebraicas con símbolos, aún los lenguajes de computación como FORTRAN o C eran (y siguen siendo) muy poco útiles.

Si hicieramos una analogía entre los lenguajes de computación y los medios de comunicación modernos, diríamos que la distancia entre los primeros lenguajes de programación y *Mathematica*, es más o menos la misma que hay entre un telegrama y una página web. El código Morse de los telegramas es muy simple y no requiere mucha tecnología; por el contrario, una página web puede transmitir mensajes mucho más complejos que un telegrama, pero requiere de tecnología más compleja. Digamos que hay un balance entre la simplicidad del lenguaje y la expresividad que se puede alcanzar por medio de éste.

Mathematica ofrece un lenguaje con gran capacidad para resolver problemas matemáticos: realizar calculos numéricos, manipular expresiones simbólicas, visualizar datos y codificar algoritmos (es decir, programar). El precio por esta expresividad es, no obstante, una gran cantidad de comandos.

Para el principiante, la mera existencia de cientos de comandos para aprender puede parecer más que un rasgo positivo, como una desventaja del lenguaje. Para poner las cosas en perspectiva, consideremos una lengua que ya conoces, el español: de seguro posees un vocabulario de varios miles de palabras, con las cuales es posible comprender la mayoría de los mensajes que oyes y lees cada día. Imagina lo aburrido y cansado que sería el español si sólo tuviésemos 200 palabras para comunicarnos. Con esto queremos decir que para una lengua que usamos cotidianamente, un vocabulario amplio es mejor que uno reducido, ya que a la larga uno ahorra tiempo expresando de manera concisa lo que queremos expresar.

Para animarte con otra analogía, piensa en *Mathematica* como si fuera un idioma extranjero que vas a aprender. En vez de aprender inglés o francés, vamos a aprender a "hablar en *Mathematica*". Si has tomado clases de idiomas, sabrás que cuando sólo se enfatiza el aprendizaje del vocabulario y de la gramática, aprender a hablar en el nuevo idioma es complicado y tedioso. Si, por el contrario, tuviste la ocasión de practicarlo con otras personas que ya lo hablaban, estarás de acuerdo que es mucho más dinámico, y nunca piensas cuándo fue que memorizaste esta palabra o aquella.

Así, si pensamos en *Mathematica* como una lengua extranjera, los comandos se asemejan a palabras aisladas (el vocabulario). Cuando se combinan, obtenemos oraciones con cierto significado, y cuando las ponemos juntas podemos expresar ideas complejas. Por ejemplo, si escribimos

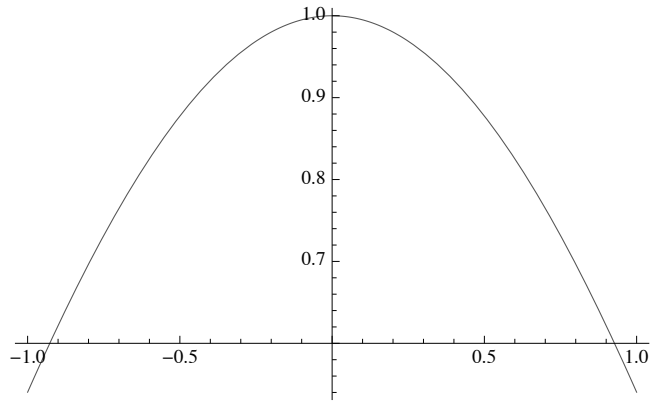
```
Cos[x]
```

```
Cos[x]
```

no obtenemos mucho: "Coseno de x". Sin embargo, si escribimos : "Grafica el coseno de x, tomando valores desde -1 hasta +1", es como una oración completa.

```
In[1]:= Plot[Cos[x], {x, -1, 1}]
```

Out[1]=



Es importante señalar que los aspectos de *Mathematica* descritos en este libro son, en su mayoría, independientes de la versión particular que se utilice. Sin embargo, debido a que con el paso del tiempo se han introducido nuevos comandos y nuevas capacidades para *Mathematica*, deseamos indicar aquí que este libro fue escrito y desarrollado usando sobre todo las versiones 5 y 6, y que fue terminado usando la versión 7.0.0 En los pocos casos en que se requiera una versión específica para usar determinado comando, o paquete, así lo indicaremos en el texto (un ejemplo importante de esta situación son las visualizaciones interactivas llamadas "objetos dinámicos", introducidas con la versión 6 y que se discuten en las secciones sobre graficación). De otro modo, simplemente daremos por entendido que estamos discutiendo los temas esenciales y que seguramente se conservarán en las nuevas versiones de *Mathematica*.

Por último, para terminar la analogía con los lenguajes naturales, recordemos la siguiente afirmación atribuida a Max Weinreich (1945): "Un lenguaje es un dialecto con una armada y un ejército". La armada de *Mathematica* es su capacidad de hacer cálculos simbólicos, y su ejército es su estilo de programación, que permite combinar sus capacidades numéricas, simbólicas y de visualización dentro de un solo sistema.

L. Dagdug y O. Guzmán

Iztapalapa, México, D.F a 20 de abril de 2009

Capitulo 1

Operaciones Básicas

Una primera sesión con Mathematica

En esta sección mostraremos algunas de las principales capacidades y características de *Mathematica* para dar una idea al lector de qué tipo de operaciones se pueden llevar a cabo, así como dar una primera impresión del uso de los comandos, es decir, las órdenes que se dan al programa y por medio de las cuales se llevan a cabo las tareas deseadas. En los capítulos subsecuentes haremos una descripción más completa de cada uno de los comandos y las capacidades de *Mathematica*.

Es recomendable seguir en el programa mismo los ejemplos aquí discutidos. Al introducir los comandos debe tenerse sumo cuidado en teclear correctamente las letras mayúsculas o minúsculas porque, a diferencia de otros lenguajes de computación, *Mathematica* distingue entre ambos tipos. Así, no es lo mismo escribir **`Cos[0]`** que **`cos[0]`**. Además también debe ponerse atención al tipo de paréntesis usados: *Mathematica* utiliza paréntesis cuadrados (corchetes) **`[]`**, para especificar los argumentos de una función o de un comando, paréntesis redondos **`()`** para agrupar operaciones, y llaves **`{}`** para agrupar los elementos de una lista.

Aunque *Mathematica* puede usarse a través de una interfaz de texto simple, es más conveniente usarlo a través de una interfaz gráfica (tal como Windows® en una PC, o X Window® en un sistema Unix®). Así como MSWord® opera sobre archivos llamados "documentos" cuyos nombres tienen la extension .doc, *Mathematica* utiliza archivos llamados "cuadernos" cuya extensión es .nb (de la palabra inglesa *notebook*). Cuando uno inicia el programa sin especificar un archivo, éste automáticamente crea un cuaderno en blanco llamado `notebook-1.nb`. En los cuadernos de *Mathematica* se escriben los comandos o instrucciones que deseamos llevar a cabo; posteriormente, se pide al programa que ejecute dichas instrucciones. *Mathematica* interpreta las instrucciones recibidas y responde con los resultados correspondientes en la línea siguiente (o varias líneas si se han suministrado varios comandos en secuencia). Explicaremos esto con un ejemplo simple, vamos a iniciar una sesión escribiendo la expresión **`1+1`**. La pantalla mostrará lo siguiente en la línea de comandos:

```
In[1]:= 1 + 1
Out[1]= 2
```

Para pedir al programa que evalúe la expresión (es decir, que ejecute el comando), necesitamos pulsar simultáneamente las teclas **Shift** y **Enter** (en adelante, la operación suimultánea de teclas se indicará como **Shift+Enter**). (En realidad, cada sistema operativo tiene una manera diferente para pedir la evaluación de comandos: en una PC se puede usar la tecla **Insert** o la tecla **Enter** del teclado numérico; en una Macintosh, la tecla **Enter** cumple el mismo propósito. Sin embargo, dado que la combinación **Shift+Enter** funciona en todos los sistemas operativos, es conveniente acostumbrarse a ella.) Enseguida el resultado será impreso en el siguiente renglón, y lo que se observará en su computadora tendrá la siguiente forma:

```
In[2]:= 1 + 1
Out[2]= 2
```

La etiquetas "**In[n]:=**" y "**Out[n]=**" a la izquierda de la expresión a evaluar y del resultado, respectivamente, son añadidas automáticamente por *Mathematica* en el momento de la evaluación. Una vez que la evaluación de la última expresión ha terminado, se puede continuar con una nueva. Cada vez que se evalúa una nueva instrucción, las etiquetas **In[n]** y **Out[n]** aparecerán con números consecutivos indicando el número de evaluaciones que el programa ha efectuado hasta ese punto.

En *Mathematica* todas las operaciones a realizar se representan internamente como una expresión de la forma **Comando[Argumento1,Argumento2,...]**. Para dar un ejemplo sencillo, tomaremos las operaciones básicas de suma, resta, división y multiplicación. Éstas se pueden efectuar por medio de los símbolos de operación habituales (**+**, **-**, *****, **/**) así como por los comandos: **Plus[]**, **Subtract[]**, **Times[]** y **Divide[]**, respectivamente. La multiplicación también se puede indicar dejando un espacio vacío entre las cantidades a multiplicar, es decir, **a*b** es equivalente a **a b**.

Plus[*elementos separados por comas***]**	Suma de todos los elementos.
Subtract[*minuendo, sustraendo***]**	Resta de dos elementos.
Times[*elementos separados por comas***]**	Producto de todos los elementos.
Divide[*numerador, denominador***]**	División de dos elementos.

Es importante notar que, por convención, los comandos de *Mathematica* inician con una mayúscula y que los argumentos se escriben entre corchetes. Así, para efectuar la suma 1+1 por medio del comando **Plus[ ]**, se escriben las cantidades a sumar como argumentos (separados por comas) dentro de los corchetes:

```
In[3]:= Plus[1, 1]
Out[3]= 2
```

Una operación muy importante en *Mathematica* es la composición o anidación de una operación dentro de otra. En otras palabras, *Mathematica* nos permite introducir comandos dentro de otros comandos. Por ejemplo, si se quiere realizar la suma 4+4 y posteriormente restarle 1 al resultado, tal operación se puede llevar a cabo introduciendo el comando **Plus[]** dentro del comando **Subtract[]**:

```
In[4]:= Subtract[Plus[4, 4], 1]
Out[4]= 7
```

Mathematica primero evalúa el comando interior (la suma) y, una vez que obtiene el resultado, procede evaluar el comando exterior (la resta) utilizando el primer resultado como parte de sus argumentos. Evidentemente este último ejemplo se pudo haber llevado a cabo con solo teclear `4+4-1`, pero sólo pretendemos mostrar la importante propiedad de composición de comandos en *Mathematica*. En general, cuando se quieren hacer operaciones numéricas simples es mejor utilizar la notación de operadores tal como lo haríamos en una calculadora:

```
In[5]:= 756.89 * 32 098.005 / 45 643
Out[5]= 532.276
```

Sin embargo, *Mathematica* no sólo opera con expresiones numéricas, también puede operar con expresiones *simbólicas*. Por ejemplo, si queremos obtener una aproximación decimal a 5/7 y evaluamos esta expresión en Mathematica, el resultado es pasmoso:

```
In[6]:= 5 / 7
Out[6]= 5/7
```

Mathematica solo escribe el resultado como una fracción, la misma que introdujimos inicialmente. La razón es que *Mathematica* es capaz de usar el valor 5/7 como el número racional exacto, y no sólo como una aproximación. Si lo que se desea es una aproximación decimal, es posible obtenerla utilizando el comando **N[]**

```
In[7]:= N[5 / 7]
Out[7]= 0.714286
```

N[*expresión, número de decimales***]**	Aproximación numérica a una expresión con un número determinado de cifras decimales.

El comando **N[]** acepta opcionalmente, como segundo argumento, el número de decimales deseado en la aproximación numérica:

```
In[8]:= N[5 / 7, 30]
Out[8]= 0.714285714285714285714285714286
```

Una ventaja de que *Mathematica* opere con expresiones simbólicas es que la precisión numérica de los cálculos (es decir, el número de cifras significativas) no está limitada por el número de dígitos con los que nuestra computadora opera. Por tanto, es posible llevar a cabo cálculos aritméticos con gran precisión y desplegarlos en pantalla con todos sus dígitos. Por ejemplo, el número 300! tiene más de 600 dígitos:

```
In[9]:= 300!
Out[9]= 306 057 512 216 440 636 035 370 461 297 268 629 388 588 804 173 576 999 416 ⁖
        776 741 259 476 533 176 716 867 465 515 291 422 477 573 349 939 147 888 ⁖
        701 726 368 864 263 907 759 003 154 226 842 927 906 974 559 841 225 476 ⁖
        930 271 954 604 008 012 215 776 252 176 854 255 965 356 903 506 788 725 ⁖
        264 321 896 264 299 365 204 576 448 830 388 909 753 943 489 625 436 053 ⁖
        225 980 776 521 270 822 437 639 449 120 128 678 675 368 305 712 293 681 ⁖
        943 649 956 460 498 166 450 227 716 500 185 176 546 469 340 112 226 034 ⁖
        729 724 066 333 258 583 506 870 150 169 794 168 850 353 752 137 554 910 ⁖
        289 126 407 157 154 830 282 284 937 952 636 580 145 235 233 156 936 482 ⁖
        233 436 799 254 594 095 276 820 608 062 232 812 387 383 880 817 049 600 ⁖
        000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 ⁖
        000 000 000 000 000 000
```

Una vez más, la habilidad de *Mathematica* para operar con símbolos también hace posible resolver ecuaciones y sistemas de ecuaciones en varias variables.

`Solve[{`*ecuaciones separadas por comas***`},{`***variables separadas por comas***`}]`**
Solución de un sistema de ecuaciones.

Por ejemplo, el sistema *2ax+3y=25*, *ax-4=32* (donde *a* es una variable conocida), se puede resolver con el comando **`Solve[]`**:

```
In[10]:= Solve[{2 a x + 3 y == 25 , a x - 4 == 32}, {x, y}]
Out[10]= {{y → -47/3, x → 36/a}}
```

En este ejemplo podemos observar varias cosas: primero, el uso de las llaves **`{}`** para agrupar las dos ecuaciones del sistema dentro de una *lista* de *Mathematica*. El segundo argumento del comando **`Solve[]`** también es una lista, **`{x,y}`**, que contiene las dos incógnitas. En segundo lugar, la notación de *Mathematica* para especificar una ecuación (un doble signo de igualdad **`==`**) es similar a la de otros lenguajes de programación, como C. Más adelante veremos cómo esta notación permite distinguir las ecuaciones de una asignación tal como *x=3*.

Efectuar manipulaciones algebraicas con *Mathematica* es simple, a continuación se muestra como factorizar y posteriormente como desarrollar un polinomio con coeficientes enteros.

```
In[11]:= Factor[x^4 - 2 x^2 + 1]
Out[11]= (-1 + x)^2 (1 + x)^2
In[12]:= Expand[(x - 1)^2 (x + 1)^2]
Out[12]= 1 - 2 x^2 + x^4
```

Factor[*polinomio***]**	Factorización de un polinomio.
Expand[*polinomio***]**	Desarrollo de un polinomio.

Como el lector nuevamente habrá notado, las cantidades se agrupan por medio de paréntesis redondos, mientras que la operación **a n** calcula la n-ésima potencia de *a*. Además, en este ejemplo, la multiplicación entre los dos términos entre paréntesis ha sido efectuada dejando un espacio entre ambas. Esta notación para efectuar multiplicaciones es la más cómoda y simple, ya que solo basta teclear una vez la barra espaciadora.

Mathematica también es útil para obtener, simplificar y verificar identidades trigonométricas:

```
In[13]:= Simplify[Sec[x]^2 - Tan[x]^2]
Out[13]= 1
```

Simplify[*polinomio***]**	Simplificación de un polinomio.

Las operaciones del cálculo diferencial e integral (tales como derivadas, derivadas parciales e integrales tanto definidas como indefinidas) también se pueden llevar a cabo fácilmente con *Mathematica*.

D[*expresión, variable***]**	Derivada de la expresión en la varialble.

Por ejemplo, la derivada de la función *f(x)=x/(x+1)* con respecto a *x* se puede obtener con el comando **D[]**

```
In[14]:= D[x / (x + 1), x]
Out[14]= -x/(1 + x)^2 + 1/(1 + x)
```

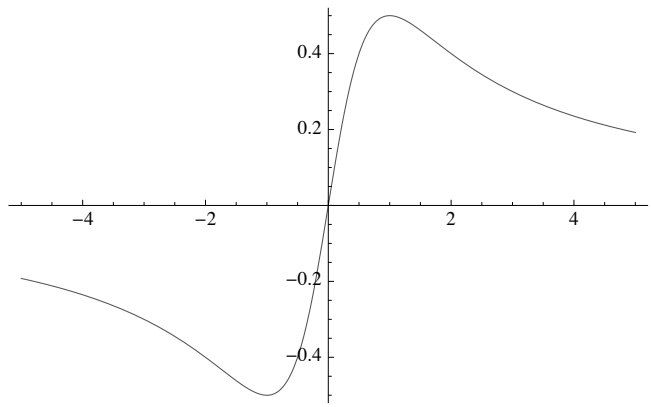
Una integral indefinida como $\int (x+1)^{-1}\, dx$ se puede obtener por medio del comando **Integrate[]**

Integrate[*expresión, variable***]**	Integral de la expresión en la varialble.

```
In[15]:=
In[16]:= Integrate[1 / (x + 1), x]
Out[16]= Log[1 + x]
```

Mathematica posee una gran cantidad de comandos para visualizar gráficas y datos en 2 y 3 dimensiones. El más sencillo de todos es el comando para graficar **Plot[]**, el cual permite graficar una función de una sola variable en un intervalo dado. Por ejemplo, si se quiere graficar la función $f(x)=x/(x^2+1)$ en el intervalo $-5 \leq x \leq 5$, se puede hacer de la siguiente manera:

```
In[17]:= Plot[x / (x^2 + 1), {x, -5, 5}]
```



Plot[*expresión,***{***variable, límite inferior, límite superior***}]**
Gráfica de la expresión en el intervalo dado de la variable.

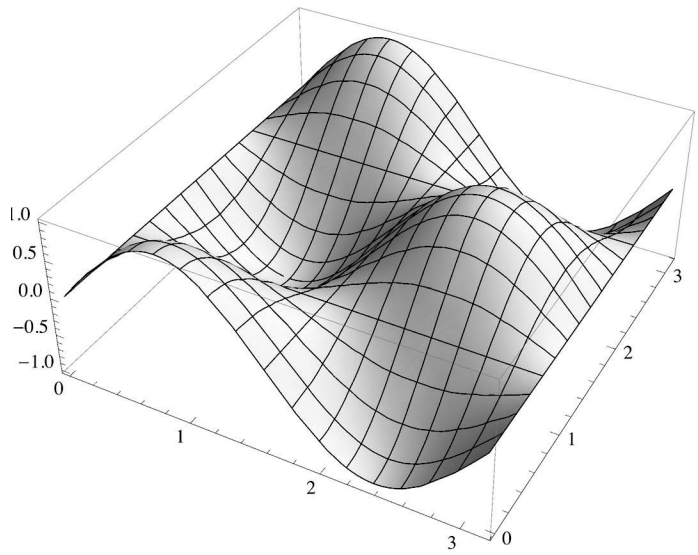
El comando **Plot3D** opera de manera análoga, pero con funciones de dos variables.

Plot3D[*expresión,***{***variable1, límite inferior, límite superior***},** **{***variable2, límite inferior, límite superior***}]**	Gráfica en tres dimensiones de una función de dos variables.

Así, para obtener la gráfica en tres dimensiones de la función $f(x,y)=\sin(2x)\cos(2y)$ en el intervalo $0 \leq x \leq \pi$ y $0 \leq y \leq \pi$, evaluamos la expresión siguiente:

In[18]:= **Plot3D[Sin[2 x] Cos[2 y], {x 0, π}, {y, 0, π}]**

Out[18]=



A partir de la versión 6.0, una vez evaluadas, las gráficas tridimensionales pueden rotarse usando el mouse, sin tener que volver a ejecutar la instrucción. (En las versiones anteriores, para generar una vista desde un punto de vista nuevo, es necesario especificar con un vector el nuevo punto de vista y volver a evaluar.)

En los capítulos del 6 al 9 discutiremos con profundidad las posibilidades que ofrece *Mathematica* para graficar. Por el momento es importante que el lector tenga en mente la estructura del comando **`Plot`** para graficar funciones. Este comando se utilizará recurrentemente a lo largo de los capítulos 2 al 5.

Para terminar con esta primera sesión, daremos un ejemplo para ilustrar cómo se pueden escribir programas en *Mathematica*, tal como se hace en otros lenguajes de programación como: C, Fortran, Pascal, etcétera.

La manera más simple de programar en *Mathematica* consiste en definir una nueva función o comando.

`nombreDeLaFunción[x_,y_]:=` *expresión*	Define una función de las variables *x, y*.

Se considera un buen hábito de programación en *Mathematica* el definir nombres de variables (o funciones) de usuario comenzando por minúsculas, reduciendo así la posibilidad de conflicto con nombres reservados. Ahora veamos un ejemplo, si queremos definir la función *g(x,y)=exp(-x^y)* podemos escribir

In[19]:= **g[x_, y_] := Exp[-x^y]**

En esta definición, la expresión a la izquierda del símbolo de asignación := representa la manera en que se invoca el nuevo comando o función. Los guiones bajos que aparecen como sufijos de x y y indican que éstas variables se deben sustituír por los valores dados por el usuario al momento de evaluar la función (note que los guiones bajos ya no aparecen en la expresión de la derecha). En este caso, la intención es que el usuario escriba **`g[2.0,3.0]`** para obtener el resultado $exp(-2^3)$:

```
In[20]:= g[2.0, 3.0]
Out[20]= 0.000335463
```

El ejemplo siguiente consiste en simular el número de veces que se obtiene cara (o cruz) al lanzar una moneda al aire un determinado número de veces. La simulación del lanzamiento de la moneda se hará obteniendo un número aleatorio entre 0 y 1 mediante el comando **`RandomReal[{0,1}]`**. Para agrupar todos los comandos necesarios para llevar a cabo la tarea utiizaremos el comando **`Module`**.

`Random`*tipo***`[{`***min,max***`}]`**	Da un número aleatório del tipo que se escoja entre el rango seleccionado.
El *tipo* puede ser **`Real`**, **`Integer`** o **`Complex`**.	

Ya que la probabilidad de obtener *cara* o *cruz* es la misma e igual a 1/2, si el número aleatorio obtenido es menor o igual a 0.5, entonces se considera que la moneda ha caído en *cara*, de lo contrario se tomará como *cruz*.

Este es el programa **`caraOcruz[ ]`** en *Mathematica*:

```
In[21]:= caraOcruz[cuantos_] :=
          Module[{t, cara = 0, n = 0},
           While[n ≤ cuantos,
            t = RandomReal[{0, 1}];
            If[t < 0.5, cara = cara + 1];
            n = n + 1;
           ];
           Print[cara, " caras observadas en ", cuantos, " lanzamientos"]
          ]
```

En la primer línea del código, se le da nombre a la rutina y a su argumento, que en este caso es el número de lanzamientos de la moneda (**`cuantos`**). También en este renglón se encuentra el comando **`Module[]`**, el cual sirve para aislar las variables locales **`t`**, **`cara`** y **`n`** dentro del módulo. Esto permite evitar problemas: si en la misma sesión hemos usado variables con los mismos nombres, éstas interferirán con nuestros cálculos si no usamos el comando **`Module[]`**.

En la segunda línea se inicializan las variables locales. En este caso, número aleatorio generado para simular el resultado del lanzamiento es **t**, el contador **cara** nos dirá el número de veces que el lanzamiento cae en *cara*, así como el numero de veces que se han llevado a cabo dichos lanzamientos, está representado por **n**. En la tercera línea se inicia un ciclo que se ejecutará mientras se cumpla la condición de que **n** sea menor que **cuantos**. En el cuarto renglón se genera un número aleatorio entre 0 y 1 con el comando **RandomReal[]**. En el quinto renglón se decide si es *cara* o *cruz* el resultado del lanzamiento: si **t** es menor o igual a 0.5 el resultado es *cara* y sumamos 1 al contador **cara**. En la siguiente línea se incrementa el contador **n** que almacena el número total de lanzamientos llevados a cabo. En el último renglón, una vez que se ha terminado la ejecución del ciclo, se imprime el mensaje que nos dice cuántas veces se obtuvo *cara* utilizando el comando **Print[]**. Es importante señalar que cada vez que se corra el programa c**araOcruz[]** se obtendrán diferentes valores, ya que cada vez se generarán diferentes números aleatorios.

Para correr nuestro código efectuando 100 lanzamientos, invocamos el comando c**araOcruz[100]**,

```
In[22]:= caraOcruz[100]
45 caras observadas en 100 lanzamientos
```

Siguiendo la filosofía de *Mathematica*, es importante recalcar que nuestro código es ahora un nuevo comando que puede combinarse con los comandos predefinidos en *Mathematica*. Esta virtud hace que sea sencillo extender las capacidades de *Mathematica* escribiendo programas especializados. De hecho, *Mathematica* se distribuye con una gran cantidad de extensiones (llamadas paquetes estándar) que se pueden cargar usando comandos como

```
In[23]:= << "NonlinearRegression`"
General::obspkg :
  NonlinearRegression` is now obsolete. The legacy version being loaded may conflict
        with current Mathematica functionality. See the
        Compatibility Guide for updating information. >>
```

En los siguientes capítulos el lector podrá aprender a usar con mayor detalle los principales comandos incluidos en *Mathematica*.

Ayudas

Es muy importante poder escribir los comandos con la redacción y estructura adecuada, por tal motivo iniciaremos esta sección con un par de recomendaciones útiles para evitar cometer errores al escribir comandos y sobre todo para introducir la estructura adecuada cuando todavía no la hemos memorizado.

Mathematica puede terminar de escribir un comando por nosotros si se escriben las primeras letras del nombre del comando. Ya sea porque el comando es muy largo o porque no se recuerda el nombre completo, bastará con teclear una o dos letras en la línea de comandos y usar el menú. Por ejemplo, digamos que queremos escribir el comando **`Simplify`**, entonces escribimos las dos primeras letras:

```
In[24]:= Si
Out[24]= Si
```

Posteriormente se selecciona el menú ***Edit***, y allí se selecciona ***Complete Selection*** (en adelante, la selección en secuencia de varios elementos de menú se denotará por cadenas del tipo ***Edit>Complete Selection***). A continuación *Mathematica* muestra en una paleta las siguientes opciones:

SiegelTheta
Sign
Signature
SignPadding
Simplify
Sin
Sinc
SingleEvaluation
SingleLetterStyle
SingularValueDecomposition
SingularityValueList
SingularValues
Sinh
SinhIntegral
SinIntegral
SixJSymbol

Para elegir el comando deseado se selecciona la opción apropiada con el cursor y se pulsa el botón del mouse. Este método es muy útil cuando se tiene que escribir un comando con un nombre muy largo, pues reduce el tiempo de escritura y evita errores tipográficos comunes.

Una situación diferente ocurre cuando lo que no se recuerda es la estructura o sintaxis del comando. *Mathematica* provee tres sistemas de ayuda independientes.

Al primer método de ayuda se accede desde la línea de comandos, mediante el comando **`Information[]`**. Digamos que deseamos obtener información acerca de **`ArcTan`**. En ese caso simplemente escribimos

```
In[25]:= Information[ArcTan]
```

ArcTan[z] gives the arc $\tan^{-1}(z)$ of the complex number z.
ArcTan[x, y] gives the arc tangent of $\frac{y}{x}$, taking into account which quadrant the point (x, y) is in. ≫

```
Attributes[ArcTan] = {Listable, NumericFunction, Protected, ReadProtected}
```

Así como el operador **+** es una representación simbólica del comando **Plus**, el comando **Information[]** se representa como el símbolo **?**:

```
In[26]:= ? ArcTan
```

ArcTan[z] gives the arc tangent $\tan^{-1}(z)$ of the complex number z.
ArcTan[x, y] gives the arc tangent of $\frac{y}{x}$, taking into account which quadrant the point (x, y) is in. ≫

Si se pulsa el botón derecho del ratón sobre el símbolo **>>** al final de la última línea, aparecerá la ayuda de *Mathematica*. Por otro lado, si sólo recordamos una parte del nombre de un comando, todavía podemos buscar información sobre éste. Digamos que recordamos que estamos buscando un comando que empieza con **Arc**, pero no sabemos qué sigue. Entonces escribimos:

```
In[27]:= ? Arc*
```

▼ System`

ArcCos	ArcCot	ArcCsc	ArcSec	ArcSin	ArcTan
ArcCosh	ArcCoth	ArcCsch	ArcSech	ArcSinh	ArcTanh

Obtenemos una lista de todos los símbolos de *Mathematica* que satisfacen el patrón dado y pulsando el botón del ratón sobre el comando deseado se despliega la información de dicho comando.

El segundo método es similar al de completar el nombre de un comando: Primero escribimos el nombre del comando que queremos usar, posteriormente seleccionamos ***Edit>Make Template***. *Mathematica* pegará la estructura del comando en su forma más simple. Por ejemplo, escribamos **Plot** en la línea de comandos y seleccionamos ***Edit>Make Template*** del menú, obtendremos lo siguiente:

```
In[28]:= Plot[f, {x, xmin, xmax}]
```

Plot::plln : Limiting value x_{min} in {x, x_{min}, x_{max}} is not a machine-size real number. ≫

```
Out[28]= Plot[f, {x, xmin, xmax}]
```

Ahora es posible editar la celda con los valores que el usuario desee, remplazando **f** por la función ***f(x)*** deseada, y las variables x_{min} y x_{max} por los límites inferior y superior, respectivamente, de la variable independiente ***x***.

Finalmente, como el último método de ayuda, uno puede buscar información sobre cualquier comando de *Mathematica* en los manuales incluidos, en formato electrónico. Para tener acceso a estos manuales basta seleccionar ***Help>Documentation Center*** del menú. Aparecerá un índice de contenidos donde podemos hallar el comando buscado. Los manuales contienen descripciones detalladas sobre los comandos, y describen sus argumentos, sus resultados, y en algunos casos hasta los métodos usados por los comandos para llevar a cabo su tarea.

Todos estos sistemas de ayuda son muy útiles ya que al combinarlos es posible escribir expresiones complicadas en poco tiempo y con un mínimo de errores, sin embargo, a medida que uno se familiariza con los comandos de uso frecuente, es posible prescindir de estas ayudas, de la misma manera que la mayoría del tiempo no buscamos en un diccionario cómo se escribe cada palabra de nuestro vocabulario, sino ocasionalmente, cuando se nos presenta alguna duda.

Cómo controlar la ejecución de los comandos

Hay ocasiones en que es necesario detener la ejecución de un comando, ya sea porque se ha cometido algún tipo de error o porque el programa ha dejado de responder. En estos casos es útil empezar intentando bloquear la evaluación del comando; si esto no funciona, podemos intentar detener por completo el kernel. Como último recurso (después de guardar el archivo con el que estamos trabajando), podemos cerrar *Mathematica* por completo.

Para suspender la ejecución de un comando, tenemos seleccionar ***Evaluation>Abort Evaluation*** del menú, así podremos continuar con la sesión sin tener que perder las definiciones de usuario y algún resultado obtenido hasta ese momento. Si esto no funciona, entonces podemos forzar el cierre de la sesión; esto último se hace mediante la selección de la opción ***Evaluation>Quit Kernel>Local*** del menú. Una ventana de diálogo nos advertirá que estamos a punto de cerrar la sesión y perder todas nuestras operaciones. Al pulsar el botón del mouse sobre el botón sobre el botón **Quit**, aceptamos el cierre de la sesión; el kernel se cerrará, pero el archivo se mantendrá intacto. Por último, si no tenemos más remedio (por ejemplo, la ventana no responde al teclado ni al mouse), será necesario cerrar por completo el programa. Si no hemos guardado con anterioridad nuestro cuaderno de *Mathematica*, perderemos todo nuestro trabajo. Por esta razón es recomendable guardar con frecuencia nuestros archivos (oprimiendo la secuencia de teclas **Ctrl+S**), y si el trabajo es muy importante, mantener copias del archivo con versiones sucesivas. Para el uso profesional, existen programas que respaldan automáticamente los archivos de *Mathematica* y, más aún, programas para el control de revisiones como *CVS*® (*Concurrent Version System*) para administrar el desarrollo de aplicaciones basadas en *Mathematica*.

Manipulación de archivos

Comúnmente cualquier usuario de programas de cómputo requiere transferir datos hacia o desde un archivo. Esta tarea se da con alta frecuencia como usuario avanzado de *Mathematica*. En muchas ocaciones se utiliza *Mathematica* para manipular los datos obtenidos de un experimento o generados por algún otro programa. También es común cuando se genera una gráfica o una imagen que se quiera exportar en un formato predeterminado para ser utilizada posteriormente. A continuación explicaremos brevemente como llevar a cabo estas tareas.

Para exportar e importar listas de datos númericos se utilizan los comandos **Import** y **Export** con la siguiente sintaxis:

Export[{"*file***", *lista*, "List"}]**	Exporta *lista* al archivo "*file*" .
Import[{"*file***", "List"}]**	Importa el archivo de datos "*file*" como una lista.

Primero hagamos que *Mathematica* indique cuál es el directorio de trabajo actual, a continuación podemos definir esta ruta o cualquier otra como nuestro directorio de trabajo. En nuestros ejemplos definiremos como directorio de trabajo el mismo directorio que *Mathematica* tiene por omisión:

```
In[29]:= dir = Directory[]
Out[29]= /Users/leo
In[30]:= SetDirectory[ dir]
Out[30]= /Users/leo
```

En el siguiente ejemplo exportaremos una lista de datos con el nombre *datos.dat* en nuestro directorio definido y posteriormente la importaremos.

```
In[31]:= Listaex = {1, 2, 3, 4}
Out[31]= {1, 2, 3, 4}
In[32]:= Export["datos.dat", Listaex, "List"]
Out[32]= datos.dat
In[33]:= Lista2 = Import["datos.dat", "List"]
Out[33]= {1, 2, 3, 4}
```

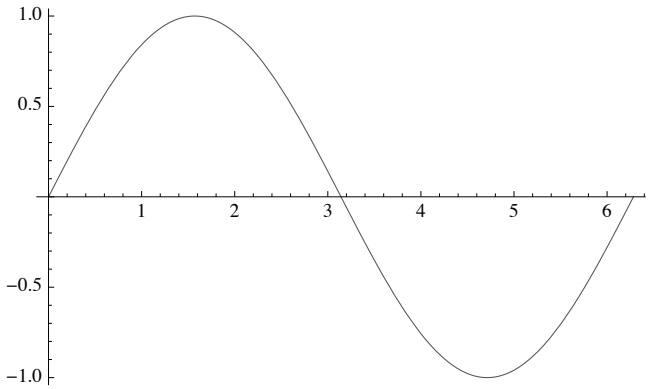
Para exportar gráficas se utiliza el comando **Export**. Los formatos a los que *Mathematica* puede exportar un gráfico es muy grande, entre ellos están: ps, eps, pdf, tiff, gif, jpeg, png, bmp, etcétera. Un archivo en cualquiera de estos formatos también puede importarse para ser usado en *Mathematica*.

Export[{"*file.ext***", *gráfica*, "*formato*"}]**	Exporta *gráfica* al archivo "*file.ext*" .

A continuación mostraremos como exportar una gráfica a formato pdf:

```
In[34]:= grafica1 = Plot[Sin[x], {x, 0, 2 Pi}]
```

Out[34]=




```
In[35]:= Export["sen.pdf", grafica1, PDF]
Export::inselem : There are insufficient elements to export to PDF format. >>
Out[35]= $Failed
In[36]:= Import["sen.pdf"]
Import::nffil : File not found during Import. >>
Out[36]= $Failed
```

Exportando expresiones matemáticas a Tex, Fortran y C

Cuando se utiliza TEX o se programa en Fortran o C, escribir expresiones matemáticas complicadas es largo y tedioso. Con *Mathematica* podemos escribir estas expresiones y posteriormente utilizar los comandos, **TeXForm**, **FortranForm** y **CForm**, para exportarlos en los formatos de TEX, Fortran o C. Estos comandos tienen una sintaxis muy simple.

TeXForm[*expresión***]**	Imprime la *expresión* en el formato Tex.
FortranForm[*expresión***]**	Imprime la *expresión* en el formato de Fortran.
CForm[*expresión***]**	Imprime la *expresión* en el formato de C.

A continuación se ejemplifica el uso de estos comandos. Suponga que desea escribir, en los formatos arriba mencionados, la fórmula de Maxwell para la distribución de velocidades. Esta ecuación calcula el número de moléculas N_v en un gas ideal cuya velocidad varía entre v y dv, donde N es el número total de moléculas presentes en el sistema, m la masa molecular, T la temperatura absoluta y k la constante de Boltzmann, a saber,

$$N_v\, d\,v = 4\,\pi\, N \left(\frac{m}{2\,\pi\, k\, T}\right)^{3/2} v^2\, e^{-\left(\frac{m\,v^2}{2\,k\,T}\right)}\, d\,v$$

Con el comando **TeXForm** podemos ver esta expresión en el formato TEX,

```
In[37]:= TeXForm[4 π N (m/(2 π k T))^(3/2) v^2 Exp[(-m v^2)/(2 k T)]]
```

```
Out[37]//TeXForm=
\sqrt{\frac{2}{\pi }} N v^2 \left(\frac{m}{k T}\right)^{3/2} e^{-\frac{m v^2}{2 k T}}
```

El comando **FortranForm** nos permite ver esta expresión en el formato de fortran,

```
In[38]:= FortranForm[4 π N (m/(2 π k T))^(3/2) v^2 Exp[(-m v^2)/(2 k T)]]
```

```
Out[38]//FortranForm=
(N*Sqrt(2/Pi)*(m/(k*T))**1.5*v**2)/E**((m*v**2)/(2.*k*T))
```

Finalmente con el comando **CForm** podemos ver esta expresión en el formato de C,

```
In[39]:= CForm[4 π N (m/(2 π k T))^(3/2) v^2 Exp[(-m v^2)/(2 k T)]]
```

```
Out[39]//CForm=
(N*Sqrt(2/Pi)*Power(m/(k*T),1.5)*Power(v,2))/Power(E,(m*Power(v,2))/(2.*k*T))
```

Capítulo 2

Capacidades aritméticas

En este capítulo explicaremos básicamente cómo sustituir una calculadora científica con *Mathematica*; nos concentraremos en las operaciones más simples de la aritmética como sumas, restas, multiplicaciones, divisiones, elevar a potencias y sacar raíces, los logaritmos, así como las funciones trigonométricas, exponencial y sus inversas.

Con esto no queremos incitar a nuestros lectores a tirar su calculadora a la basura, sino más bien queremos comenzar a familiarizarnos con *Mathematica* en un contexto en el que el lector se sienta cómodo y pueda concentrase en el modo en que se hacen las operaciones.

En el capítulo anterior explicamos cómo los comandos **`Plus`**, **`Subtract`**, **`Times`** y **`Divide`** realizan las cuatro operaciones básicas de la aritmética, y cómo éstos comandos tienen una abreviatura que corresponde a la notación algebraica habitual. Además, en *Mathematica*, las expresiones aritméticas siguen las reglas de precedencia de dichas operaciones: La multiplicación y la division tienen precedencia sobre la suma y la resta, y las operaciones de igual precedencia se realizan primero de izquierda a derecha. Asi, evaluando la siguiente expresion obtenemos

```
In[1]:= 10 + 6 / 3 - 5 * 2
Out[1]= 2
```

Como con las calculadoras, los paréntesis (redondos) sirven para agrupar explicítamente operaciones con independencia de las reglas de precedencia. En virtud de los paréntesis, la siguiente expresion toma un valor totalmente distinto al de la expresión anterior:

```
In[2]:= (10 + 6) / (3 - 5) * 2
Out[2]= -16
```

Nuevamente, tenemos que hacer la aclaración de que en *Mathematica* debemos distinguir entre numeros exactos y aproximados. Por ejemplo, si evaluamos la expresión siguiente, el resultado es la misma expresión sin cambios:

```
In[3]:= 3 / 5
Out[3]= 3/5
```

La razón de tan "extraño" comportamiento es que *Mathematica* distingue entre el numero racional que es exactamente 3/5 y el número de precisión flotante 0.6. Como hemos explicado en el capítulo anterior, si queremos evaluar aproximadamente el valor de un número exacto podemos usar el comando **`N`**:

```
In[4]:= N[3 / 5]
Out[4]= 0.6
```

N[*expresión***]**	Aproximación numérica a una expresión.

Existen algunas convenciones para los nombres de los comandos de *Mathematica*. La más importante es la que especifica que *todos* los comandos preprogramados en este software comienzan con una letra mayúscula, por ejemplo, **N**, **Plus**, **Times**, etcétera. Esto es especialmente útil para poder distinguir rápidamente cuándo un comando lo programamos nosotros y cuándo proviene del sistema en sí mismo: si tenemos cuidado de que todos los comandos que nosotros definimos empiecen con una letra minúscula no habrá posibilidad de confusión. En este libro no seguimos esta regla al 100%, porque creemos que lo más importante es que el código de nuestros documentos y programas sea claro y fácil de mantener: si en alguna ocasión lo natural es usar letras mayúsculas, así lo hacemos, corriendo un pequeño riesgo que en versiones futuras los diseñadores de *Mathematica* escojan (por puro azar) el mismo nombre que escogimos para nuestro comando local.

Continuando con las funcionalidad básica de una calculadora, *Mathematica* provee métodos para calcular potencias y extraer raíces. Si queremos obtener el cuadrado de un numero *x*, usamos la expresion **x^2**. Por ejemplo,

```
In[5]:= 2^3
Out[5]= 8
```

Para obtener la raíz de *x*, usamos el comando **Sqrt[x]**:

```
In[6]:= Sqrt[100]
Out[6]= 10
```

Sqrt[*argumento***]**	Da la raíz cuadrada del argumento.

Una de las funciones elementales es la función exponencial, e^x. En *Mathematica*, la notación para esta función es **Exp[x]**.

Exp[*argumento***]**	Eleva elnúmero *e* al argumento.

Por ejemplo,

```
In[7]:= Exp[1.0]
Out[7]= 2.71828
```

La inversa de la función exponencial es el logaritmo natural, que en *Mathematica* se escribe así

```
In[8]:= Log[2.71828128]
Out[8]= 1.
```

Log[*argumento***]**	Da el logaritmo del argumento.

En este ejemplo vemos que operar con números aproximados introduce errores, si repetimos el ejemplo anterior usando aritmetica exacta, obtenemos el siguiente resultado:

```
In[9]:= Log[Exp[1]]
Out[9]= 1
```

En general, es conveniente tratar de usar expresiones exactas en los cálculos y sólo sustituir los valores aproximados en el paso final. Por supuesto, hay veces en que este procedimiento no es muy eficiente, por ejemplo cuando realizan cálculos exactos toma tanto tiempo que es imposible llegar al resultado final, mientras que un cálculo numérico puede terminar mucho más rápido. Aún en estos casos es importante estimar la magnitud del error asociado a las aproximaciones numéricas que se hicieron, de manera que se pueda mantener bajo control el error en los resultados.

Continuando con las funciones elementales, tenemos todas las funciones trigonométricas: **Sin[x]**, **Cos[x]**, **Tan[x]**, **Cot[x]**, **Sec[x]** y **Csc[x]**.

Sin[*ángulo en radianes***]**	Da el seno del ángulo en radianes.
Cos[*ángulo en radianes***]**	Da el coseno del ángulo en radianes.
Tan[*ángulo en radianes***]**	Da la tanente del ángulo en radianes.
Sec[*ángulo en radianes***]**	Da la secante del ángulo en radianes.
Csc[*ángulo en radianes***]**	Da la cosecante del ángulo en radianes.
Cot[*ángulo en radianes***]**	Da la cotangente del ángulo en radianes.

A diferencia de las calculadoras convencionales, *Mathematica* define también la cotangente, secante y cosecante. Es importante señalar que estas funciones están diseñadas para aceptar como argumentos ángulos medidos en radianes. Es muy fácil cometer el error de usar ángulos medidos en grados, en dichos casos la respuesta parecerá equivocada aunque no habrá ninguna advertencia o mensaje de error. Por citar un ejemplo, digamos que nos equivocamos y tratamos de averiguar el valor de cos 90° sin convertir el ángulo a radianes, esperando obtener por respuesta cero:

```
In[10]:= Cos[90.0]
Out[10]= -0.448074
```

Desde el punto de vista de *Mathematica*, 90 radianes es un ángulo perfectamente válido (corresponde aproximadamente a 5156.6° o bien, si reducimos este ángulo al segundo cuadrante, a 116.6°), y por tanto la función coseno devuelve un valor definido. Es muy importante recordar siempre que, tal y como establecen las convenciones del cálculo diferencial e integral, todos los ángulos se han de medir en radianes. Si queremos calcular el coseno de 90 grados *Mathematica* nos permite hacerlo de la siguiente manera, usando el símbolo **Degree**:

```
In[11]:= Cos[ 90 Degree]
Out[11]= 0
```

Las funciones trigonométricas inversas están disponibles en *Mathematica* mediante los siguientes comandos: **ArcSin[x]**, **ArcCos[x]**, **ArcTan[x] ArcCot[x]**, **ArcSec[x]** y **ArcCsc[x]**. La función **ArcTan** acepta también la sintaxis **ArcTan[x, y]** donde **x** representa el cateto adyacente y **y** el cateto opuesto al ángulo buscado. Esta variante es útil porque permite distinguir el cuadrante apropiado para el ángulo con base en los signos de los dos argumentos, lo cual no es posible hacer cuando sólo se conoce la razón *y*/*x*.

ArcSin[*ángulo en radianes***]**	Da el arco seno del ángulo en radianes.
ArcCos[*ángulo en radianes***]**	Da el arco coseno del ángulo en radianes.
ArcTan[*ángulo en radianes***]** **ArcTan[***x,y***]**	Da el arco tanente del ángulo en radianes. Da el arco tangente del cateto adyacente y opuesto. *x, y*.
ArcSec[*ángulo en radianes***]**	Da el arco secante del ángulo en radianes.
ArcCsc[*ángulo en radianes***]**	Da el arco cosecante del ángulo en radianes.
ArcCot[*ángulo en radianes***]**	Da el arco cotangente del ángulo en radianes.

Una clase de funciones muy importante en las aplicaciones numéricas son los llamados "generadores de números aleatorios". Un generador de números aleatorios es simplemente una función que devuelve números al azar, pero que siguen una distribución predefinida. Por ejemplo, para simular el resultado de tirar un dado, requerimos un generador de números aleatorios que produzca un número entero entre uno y seis con una distribución uniforme; es decir, que todos los números tengan la misma probabilidad de ocurrir. En *Mathematica*, podemos acceder a un generador como el descrito mediante la instrucción

```
In[12]:= RandomInteger[{1, 6}]
Out[12]= 6
```

RandomReal[{*min,max***}]**	Da un número aleatorio real en el intervalo especificado.
RandomInteger[{*min,max***}]**	Da un número aleatorio entero en el intervalo especificado.
RandomComplex[{*min,max***}]**	Da un número aleatorio complejo en el intervalo especificado.

Un problema técnico, pero que de ninguna manera es ignorable, es que en realidad por definición no es posible esperar que un programa de computadora genere números al azar: si tenemos el listado de las instrucciones del programa y sabemos los contenidos de la memoria de la computadora, en principio podemos predecir cuál será la secuencia de números supuestamente "aleatorios" que el programa va a generar. En general, lo más que podemos esperar es producir una secuencia de numeros "pseudoaleatorios", es decir, una lista de números que pasen una gran lista de pruebas estadísticas para detectar correlaciones entre elementos sucesivos. Sin embargo, si inicializamos la memoria del generador al mismo estado, siempre obtendremos la misma secuencia. Esta circunstancia es a veces ventajosa: por ejemplo, cuando se esta verificando un programa a veces es útil repetir una lista de números aleatorios para evaluar si el programa sigue funcionando igual antes y después de haber sido modificado. En *Mathematica*, podemos inicializar el generador de números aleatorios dando una "semilla" en la forma de un número que nosotros elegimos.

SeedRandom[*número***]**	Asinación de una semilla.

Por ejemplo, si tomamos como semilla el número 2389659,

```
In[13]:= SeedRandom[2 389 659]
```

y generamos tres números aleatorios

```
In[14]:= RandomReal[]
Out[14]= 0.279749

In[15]:= RandomReal[]
Out[15]= 0.302547

In[16]:= RandomReal[]
Out[16]= 0.498767
```

```
In[19]:= RandomReal[]
Out[19]= 0.302547
In[20]:= RandomReal[]
Out[20]= 0.498767
```

Podríamos seguir con una larga lista de otras funciones elementales (funciones hiperbólicas, de combinaciones y probabilidad, etcétera), pero en lugar de hacer esto simplemente el lector puede consultar el manual en línea de *Mathematica* (accesible a través del menu *Help*) que contiene descripciones de todas y cada una de las funciones disponibles.

Ahora, queremos llamar la atención del lector a un grupo de funciones que normalmente las calculadoras científicas no proveen, y sin embargo encuentran aplicaciones en varios campos de la física y la ingeniería.

Nos referimos a las llamadas "funciones especiales", las cuales no son más ni menos elementales que las funciones trigonométricas. Con la llegada de sistemas como *Mathematica*, familiarizarse y usar estas funciones es más fácil que nunca. Entre estas funciones tenemos los polinomios de Legendre (`LegendreP`), de Hermite (`HermiteH`), de Laguerre (`LaguerreL`), las funciones de Bessel (`BesselJ`, `BesselY`), etcétera. Volveremos a referirnos a estas funciones en el capítulo dedicado a la solución de ecuaciones diferenciales.

Para finalizar con este capítulo, queremos presentar la manera de "guardar en la memoria" los resultados de un cálculo en *Mathematica*. Digamos que queremos realizar una operación en la que queremos conocer la diferencia entre dos cantidades, $u = x - y$, así como el cociente $v = y/x$. Supongamos que tenemos fórmulas explícitas para x y para y, y además los valores que entran dentro de cada fórmula:

$x = mg$, con m=0.25, g=9.81

y=6 π η R, con $\eta = 0.01$, $R = 0.10$

En este momento no es importante qué representan estas fórmulas, son tan solo un ejemplo de la necesidad de almacenar el resultado de las operaciones correspondientes. En lugar de tener que reescribir los números correspondientes a x y y cada vez que los necesitemos, decidimos asignar el valor del primer cálculo a la variable x:

```
In[21]:= x = 0.25 * 9.81
Out[21]= 2.4525
```

y el valor del segundo cálculo a y:

```
In[22]:= y = 6 * π * 0.01^2
Out[22]= 0.00188496
```

Ahora podemos usar los símbolos `x` y `y` para referirnos a los valores que les hemos asignado:

```
In[23]:= x - y
Out[23]= 2.45062
```

```
In[24]:= y / x
Out[24]= 0.000768585
```

Para borrar las asignaciones de una variable dada, podemos usar el comando **Clear**:

Clear[x]	Borra la asignación a la variable *x*.

```
In[25]:= Clear[x]
```

A partir de ahora, el símbolo x es interpretado como una variable genérica, ya que no tiene un valor numérico definido:

```
In[26]:= x - y
Out[26]= -0.00188496 + x
```

Este tipo de asignación representa la forma más básica de asociar un valor con un símbolo. En capítulos siguientes aprenderemos otras formas que permiten controlar, entre otras cosas, cuándo se aplica la definición y si el lado derecho de la definición se debe tomar literalmente o se debe considerar como una expresión algebraica en la cual se han de sustituir valores en el momento en que de hecho se utiliza la definición.

Además de distinguir entre números exactos y aproximados, *Mathematica* también distingue, entre otros tipos, números enteros, reales y complejos. En particular, un número complejo se puede representar en la forma siguiente

```
In[27]:= 1 + 2 I
Out[27]= 1 + 2 i
```

El símbolo **I** (i mayúscula) representa el número imaginario $i = \sqrt{-1}$. A partir de la versión 3, una alternativa al símbolo **I** es usar la siguiente secuencia de teclas: *Esc*,*i*,*i*,*Esc*. Esto introduce el símbolo **i** que significa exactamente lo mismo que **I**, pero que visualmente es más parecido al símbolo tradicional para la raíz de -1. Otras constantes similares son **E** (la base de los logaritmos naturales) y **Pi** (la razón de la circunferencia al diámetro de un círculo). También para estas constantes existe una secuencia alternativa que produce sinónimos de **E** y Pi: *Esc*,*e*,*e*,*Esc* resulta en el símbolo e, mientras que *Esc*,*p*,*Esc* resulta en la letra griega π.

Algunas funciones relacionadas a los números complejos son **Re** e **Im**, que calculan la parte real e imaginaria (respectivamente) de su argumento. La función **Abs** calcula el módulo (valor absoluto) de un número complejo, mientras que **Arg** calcula el ángulo (llamado argumento) que el número complejo forma con el eje real.

Re[*expresión***]**	Parte real de un número complejo.
Im[*expresión***]**	Parte imaginaria de un número complejo.

Abs[*expresión***]**	Módulo (valor absoluto) de un número complejo.
Arg[*expresión***]**	Argumento (ángulo con el eje real) de un número complejo.
Conjugate[*expresión***]**	Complejo conjugado de un número complejo.

A continuación se presentan alguos ejemplos numéricos de como hacer las operaciones básicas con números complejos.

```
In[28]:= (2 + 3 i) + (1 + 2 i)
Out[28]= 3 + 5 i

In[29]:= (2 + 3 i) - (1 + 2 i)
Out[29]= 1 + i

In[30]:= (2 + 3 i) / (1 + 2 i)
Out[30]= 8/5 - i/5
```

Finalmente en los siguientes ejemplos obtendremos la parte real , imaginaria, al argumento, el módulo y el complejo conjugado de un número complejo.

```
In[31]:= z = 5 + 2 i
Out[31]= 5 + 2 i

In[32]:= Re[z]

Out[32]= 5

In[33]:= Im[z]

Out[33]= 2

In[34]:= Abs[z]
Out[34]= Sqrt[29]
         Sqrt[29]
         Sqrt[29]

In[35]:= Conjugate[z]
Out[35]= 5 - 2 i
```

Ejercicios

1. Calcule el valor de la "razón áurea": $a = \frac{1+\sqrt{5}}{2}$

2. Calcule el valor *aproximado* de las expresiones siguientes:

$a_1 = \frac{1}{1+10}$

$a_2 = \frac{1}{1+\frac{1}{1+10}}$

$a_3 = \frac{1}{1+\frac{1}{1+\frac{1}{1+10}}}$

$a_4 = \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+10}}}}$

$a_5 = \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+10}}}}}$

Compare los números recién obtenidos con la razón áurea.

3. Un dodecahedro es un polihedro regular con doce lados pentagonales. El volumen del dodecahedro es igual a $V = a^3 \ \frac{15+7\sqrt{5}}{4}$. Supongamos que el lado de dichos pentágonos tiene una longitud a=2 m, ¿cuál es el volumen del dodecahedro?

4. La esfera más pequeña que encierra completamente al dodecahedro del ejercicio anterior tiene un radio r dado por $r = a \ \frac{\sqrt{3}\left(1+\sqrt{5}\right)}{4}$. ¿Qué porcentaje del volumen de dicha esfera encierra el dodecahedro?

5. Calcular el valor de la función seno de 60^0.

6. Encontrar un número pseudo aleatorio de 15 dígitos entre π y 2π.

7. Usar el comando **`PrimeQ`** para averiguar si 156,875,438,767 es o no es un número primo.

8. Calcular el logaritmo en base 10 de e^5.

9. Calcular $\sqrt{3} - \sqrt{2}$ con 50 dígitos.

10. Encontrar el valor numérico de seno de 15^0.

11. Obtener la tangente hiperbólica del logaritmo natural de x.

12. Encontrar el valor aproximado de cuántos grados hay en un radián.

13. Encontrar la raíz cuadrada de 3+4i.

14. Calcular $\log_2 100$.

15. ¿Cuál es entero más cercano a $\sqrt{159}$?

Capítulo 3

Capacidades algebraicas

En este capítulo discutiremos la capacidad de Mathematica para hacer operaciones algebraicas. El uso de los comandos de este capítulo puede reducir el tiempo empleado al hacer cálculos, todos ellos tienen una estructura simple y son fáciles de recordar. Su importancia radica en que nos ayudarán a manipular las soluciones que obtengamos en *Mathematica,* para simplificarlas o ponerlas en una forma más conveniente.

Factorizando y desarrollando polinomios

Si se quiere factorizar un polinomio como por ejemplo $a^2+2\,a\,b+b^2$, se utiliza el comando **Factor[]** introduciendo la expresión a simplificar en los paréntesis cuadrados, como se muestra a continuación. (Es importante dejar un espacio entre las variables *a* y *b* para indicar la multipicación, si no se deja el espacio parecerá que estamos introduciendo una variable llamada *ab*)

```
In[1]:= Factor[a^2 + 2 a b + b^2]
Out[1]= (a + b)^2
```

Factor[*expresión***]**	Factoriza la expresión.

Evidentemente Mathematica nos será util en la medida en que pueda factorizar expresiones que a primera vista no resulten una tarea fácil de efectuar a mano, como se muestra en el siguiente ejemplo

```
In[2]:= Factor[16 + 50 x^2 + 24 x^3 + 75 x^5]
Out[2]= (8 + 25 x^2) (2 + 3 x^3)
```

Mathematica, por defecto, lleva a cabo la factorización utilizando sólo números reales. Sin embargo, en ocaciones es necesario el uso de números complejos para efectuar dicha tarea. Si se quiere que *Mathematica* factorice también utilizando números imaginarios se tiene que agregar la opción **GaussianIntegers→True** en el comando **Factor**.

Factor[*expresión*,**GaussianIntegers→True]**	Factoriza la expresión utlizando número imaginarios.

Por ejemplo, si se quiere factorizar la expresión x^2+9, esta operación sólo se puede llevar a cabo utilizando números complejos, ya que las raíces del polinomio son *-3i* y *+3i*. Conociendo las raices sabemos que podemos factorizar x^2+9 como *(x+3i)(x-3i)*. Al utilizar solamente el comando **Factor** en la expresión anterior se obtiene como respuesta sin cambios la misma expresión , a saber,

```
In[3]:= Factor[x^2 + 9]
Out[3]= 9 + x^2
```

Sólo al incluir la opción **GaussianIntegers→True**, para incluir números imaginarios, obtenemos el resultado esperado.

```
In[4]:= Factor[x^2 + 9, {GaussianIntegers -> True}]
Out[4]= {(-3 i + x) (3 i + x)}
```

Para que la factorización también se pueda efectuar utilizando funciones trigonométricas o hiperbólicas y múltiplos de ángulos, es necesario agregar la opción **Trig→True**. En el siguiente ejemplo mostraremos la diferencia entre usar y no usar la opción arriba mencionada en una factorización en la que se involucran funciones trigonométricas.

```
In[5]:= Factor[(9 / 2) - (1 / 2) Cos[2 x] + 4 Sin[x]]
Out[5]= 1/2 (9 - Cos[2 x] + 8 Sin[x])
In[6]:= Factor[(9 / 2) - (1 / 2) Cos[2 x] + 4 Sin[x], Trig → True]
Out[6]= (2 + Sin[x])^2
```

En el primer caso se observa que la factorización afectuada por el programa da como resultado una función tan complicada como la original, que para fines prácticos no es de utilidad. En el segundo caso se obtiene una expresión que luce mucho más simple que la original.

En muchas ocasiones desarrollar un polinomio a mano se convierte en una tarea tediosa, pero usando el comando **Expand**, *Mathematica* es capaz de hacerlo por nosotros.

Expand[*expresión***]**	Desarrolla la expresión.

Por ejemplo, encontrar la quinta potencia de *2x+3y* no parece complicado, pero definitivamente es laborioso, Con *Mathematica* obtendremos la respuesta inmediatamente.

```
In[7]:= Expand[(2 x + 3 y)^5]
Out[7]= 32 x^5 + 240 x^4 y + 720 x^3 y^2 + 1080 x^2 y^3 + 810 x y^4 + 243 y^5
```

Es importante señalar que la opción **Trig→True** también puede ser útil con el comando **Expand**. Como una abreviatura, los comandos **TrigExpand[ ], TrigFactor[ ] y TrigReduce[ ]**, asumen automáticamente **Trig→True**.

Existe una gran cantidad de situaciones en las cuales se quiere escribir una expresión algebraica en la forma más simple posible. Decir exactamente que significa la expresión más simple, no es una taréa fácil, pero el procedimiento puede reducirse al observar las diferentes expresiones obtenidas tras la simplificación, identificar cual de ellas tiene el mínimo número de términos, y escojer una.

Para simplificar expresiones con *Mathematica* se utiliza el comando **Simplify[]**.

Simplify[*polinomio***]**	Simplificación de un polinomio.

Este comando intenta dar como respuesta la forma más simple de la expresión, como por ejemplo cancelando los factores comunes del numerador y denominador, como a continuación se muestra.

```
In[8]:= Simplify[(1 - x^5) / (1 - x)]
```

$$\text{Out[8]= } 1 + x + x^2 + x^3 + x^4$$

En los casos en los cuales se encuentran exponentes en la expresión que se quiere manipular, es conveniente utilizar el comando **PowerExpand[]** para simplificar dichas expresiones . Por ejemplo, $\sqrt{x^2}$ no puede ser simplificada por medio de los comandos **Simplify** o **Expand**, como se muestra a continuación.

PowerExpand[*expresión***]**	Desarrolla la expresión cuando en ésta se encuentran exponentes.

```
In[9]:= Simplify[Sqrt[x^2]]
```

$$\text{Out[9]= } \sqrt{x^2}$$

```
In[10]:= Expand[Sqrt[x^2]]
```

$$\text{Out[10]= } \sqrt{x^2}$$

Por otro lado, el comando **PowerExpand** sí simplifica la expresión anterior,

```
In[11]:= PowerExpand[Sqrt[x^2]]
Out[11]= x
```

A continuación discutiremos brevemente el uso de los comandos **Together**, **Apart** y **Collect**, los cuales también pueden ser de utilidad en la manipulación algebraica de polinomios. El comando **Together** combina términos sobre un mismo común denominador. **Apart** separa las fracciones, y **Collect** factoriza coeficientes de igual grado en la variable que se especifica.

Together[*expresión***]**	Combina términos sobre un mismo común denominador.
Apart[*expresión***]**	Separa fracciones.
Collect[*expresión***]**	Factoriza coeficientes de igual grado en la variable que se especifica.

En los siguientes ejemplos se pone de manifiesto la función de estos comandos.

```
In[12]:= Together[x / y + z / w]

Out[12]= (w x + y z) / (w y)

In[13]:= Apart[(y z + x w) / (y w)]

Out[13]= x/y + z/w

In[14]:= Collect[a x + b x + c y + d y, x]

Out[14]= (a + b) x + c y + d y
```

El comando **FullSimplify**, intenta encontrar un número mayor de posibilidades de simplificación que el comando **Simplify**. Por otro lado la ejecución del comando puede en ocasiones tomar un tiempo considerable.

FullSimplify[*polinomio***]**	Simplificación de un polinomio intentando un número mayor de posibilidades.

En el ejemplo que a continuación se presenta, el lector podrá comparar la eficiencia de operación entre los comandos **Simplify** y **FullSimplify**. Para ello utilizamos estos comandos junto con el comando **Timing.**

Timing[*operación***]**	Tiempo en que la computadora tarda en llevar a cabo la operación.

Éste último nos indica el tiempo que la computadora tarda en llevar a cabo la operación.

```
In[15]:= y := ArcTan[x]
```

```
In[16]:= expresion1 := (Exp[y] - Exp[-y]) / (Exp[y] + Exp[-y])

In[17]:= Timing[Simplify[expresion1]]

Out[17]= {0.013941, (-1 + E^(2 ArcTan[x])) / (1 + E^(2 ArcTan[x]))}

In[18]:= Timing[FullSimplify[expresion1]]

Out[18]= {0.0889, Tanh[ArcTan[x]]}
```

Observamos que **`FullSimplify`** es en general más tardado que **`Simplify`**, porque el primero busca más formas de simplificar la expresión que el segundo.

Para evitar que el valor de la variable *y* nos estorbe en el resto de la sesión, podemos borrarlo usando el comando **`Clear[ ]`**

```
In[19]:= Clear[y]
```

Para finalizar esta sección hablaremos de algunos comandos por medio de los cuales es posible extraer sólo los términos deseados de una expresión algebraica. Los comandos más útiles son **`Coefficient[]`**, **`Exponent[]`** y **`Part[]`**. El primero despliega todos los términos que contienen el coeficiente indicado. El segundo muestra el exponente máximo de la variable seleccionada en la expresión. Finalmente el comando Part da el *n*-ésimo término de la expresión.

`Coefficient[`*expresión, variable***`]`**	Despliega todos los términos que contiene la expresión en lavariable.
`Exponent[`*expresión, variable***`]`**	Despliega el exponente máximo de la expresión en lavariable.
`Part[`*expresión, n***`]`**	Despliega el *n-ésimo* término de la expresión.

A continuación se dan ejemplos en los cuales se muestra la sintaxis de estos comandos. Supongamos que tenemos la expresión *1+3x+6x^2+8y^2+22xy^2+3y^3*. Los coeficientes de *x* se obtienen de la siguiente manera,

```
In[20]:= Coefficient[1 + 3 x + 6 x^2 + 8 y^2 + 22 x y^2 + 3 y^3, x]

Out[20]= 3 + 22 y^2
```

El máximo exponente de *y* se despliega escribiendo,

```
In[21]:= Exponent[1 + 3 x + 6 x^2 + 8 y^2 + 22 xy^2 + 3 y^3, y]

Out[21]= 3
```

Finalmente si se quiere obtener el tercer término de la expresión se hace de la siguiente manera,

```
In[22]:= Part[1 + 3 x + 6 x^2 + 8 y^2 + 22 xy^2 + 3 y^3, 3]
Out[22]= 6 x^2
```

Solución de ecuaciones y sistemas de ecuaciones, analítica y numéricamente

Mathematica también es capaz de manipular ecuaciones e incluso resolverlas. Esto significa, encontrar los valores para los cuales una igualdad se cumple. *Mathematica* utiliza el comando **Solve[]** para encontrar las soluciones de una ecuación. El primer argumento en este comando es la ecuación a resolver, mientras que en el segundo argumento se indica la variable sobre la cual se quiere resolver la ecuación. Esto además asigna un valor constante a todos los restantes símbolos que aparecen en la ecuación. Ambos argumentos se deberán encontrar separados por una coma.

Solve[{*ecuaciones separadas por comas***},{***variables separadas por comas***}]**
Solución de un sistema de ecuaciones.

Por ejemplo, la solución de la ecuación $ax^2+8bx+166=0$, tomando camo variable a x, la encontramos de la siguiente manera,

```
In[23]:= Solve[{a x^2 + 8 b x + 166 == 0}, x]
```

$$\text{Out[23]=}\quad \left\{\left\{x \to \frac{-4\,b - \sqrt{2}\,\sqrt{-83\,a + 8\,b^2}}{a}\right\}, \left\{x \to \frac{-4\,b + \sqrt{2}\,\sqrt{-83\,a + 8\,b^2}}{a}\right\}\right\}$$

La solución obtenida es simplemente el par de valores que satisfacen la igualdad. Si se quiere tener un resultado más simple de leer, se puede aplicar el comando **TableForm** al comando **Solve.**

TableForm[*expresión***]** Da formato de columnas a una expresión.

Al aplicar esta secuencia de comandos al ejemplo anterior, el resultado se desplegaría de la siguiente forma,

```
In[24]:= TableForm[Solve[{a x^2 + 8 b x + 166 == 0}, x]]
Out[24]//TableForm=
```

$$x \to \frac{-4\,b-\sqrt{2}\,\sqrt{-83\,a+8\,b^2}}{a}$$

$$x \to \frac{-4\,b+\sqrt{2}\,\sqrt{-83\,a+8\,b^2}}{a}$$

Es muy probable que en muchas ocasiones, el lector necesite manipular la solución de la ecuación, por ejemplo, si se quiere comprobar que el resultado sea correcto. Para ello es necesario introducir el valor de la solución en la ecuación resuelta. A continuación se explicará como hacer dicha tarea, para ello iniciaremos explicando la regla de transformación. Una regla de transformación sirve para asignar un valor particular a una variable. Si se quiere asignar temporalmente el valor de 2 a *x*, se hace creando la regla de transformación x→2. Ahora bien, para aplicar una regla de transformación a una expresión se tiene que escribir **/.** entre la expresión y la regla de transformación. Para dar otro ejemplo, utilizaremos la regla de transformación para determinar si la solución a una ecuación obtenida por Mathematica es la correcta.
Primero encontraremos la solución a la ecuación 2x+5=9 utilizando el comando **Solve**,

```
In[25]:= Solve[{2 x + 5 == 9}, x]
Out[25]= {{x → 2}}
```

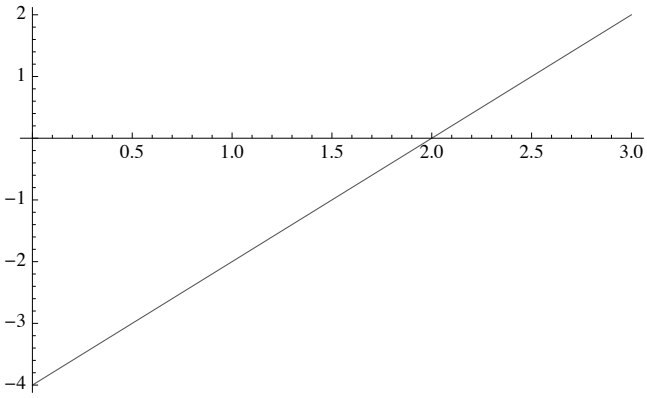
Para comprobar que la solución a la ecuación es la correcta, aplicamos la regla de transformación obtenida como solución a la ecuación original

```
In[26]:= 2 x + 5 == 9 /. x → 2
Out[26]= True
```

La respuesta obtenida significa que tras la substitución de *x=2*, *Mathematica* encontró que sí se cumple la igualdad. Finalmente mostramos la gráfica de la función para ver el comportamiento de esta y ratificar que la raíz encontrada es la correcta,

```
In[27]:= Plot[{2 x + 5 - 9}, {x, 0, 3}]
```

Out[27]=



Para la gran mayoría de ecuaciones de quinto grado y mayores, difícilmente se encontrará una solución analítica. Para estos casos es conveniente utilizar el comando **NSolve**. Este comando nos dará una solución numérica aproximada de los valores que cumplen la igualdad. A continuación daremos un ejemplo de como usar el comando. Respecto a la sintaxis, es la misma que la del comando **Solve**.

NSolve[{*ecuaciones separadas por comas***},{***variables separadas por comas***}]**
Solución de un sistema de ecuaciones numericamente.

```
In[28]:= TableForm[NSolve[x^6 + x^5 + x^3 + 1, x]]
Out[28]//TableForm=
x → -1.32472
x → -1.
x → 0. - 1. i
x → 0. + 1. i
x → 0.662359 - 0.56228 i
x → 0.662359 + 0.56228 i
```

Con lo discutido anteriormente será simple plantear en el lenguaje de *Mathematica* la solución para un sistema de ecuaciones. En este caso también se usa el comando **Solve**. En caso de que se quiera resolver numericamente el sistema, se puede usar el comando **NSolve**. El primer argumento será la lista de ecuaciones a resolver y el segundo argumento deberá contener la lista de incógnitas. En ambos casos los listados se tienen que escribir entre llaves. En el siguiente ejemplo se encuentra la solución de un sistema de dos ecuaciones con dos incógnitas (x,y). Como una de las ecuaciones es cuadrática y la otra lineal, se encuentran dos soluciones diferentes:

```
In[29]:= TableForm[Solve[{x^2 + y^2 == 16, x - 4 == y}, {x, y}]]
Out[29]//TableForm=
x → 0  y → -4
x → 4  y → 0
```

Proyecto. La razón áurea

Cuando *Mathematica* despliega una gráfica, lo hace en un marco cuya relación entre el tamaño del eje de las ordenadas y el de las abscisas es $(1+\sqrt{5})/2$. El valor de esta expresión es aproximadamente 1.618 y es conocida como la razón áurea. *Mathematica* utiliza esta relación porque de este modo se construye el rectángulo con las proporciones más agradables a la vista. A un rectángulo con estas proporciones se le llama rectángulo perfecto. Se dice que un rectángulo es perfecto si tiene la propiedad de que al quitarle un cuadrado obtenemos un rectángulo (más pequeño) con las mismas proporciones del rectángulo original. Este hecho era bien conocido por los griegos en el siglo VI, quienes lo utilizaron al construir el Partenón. En la naturaleza tambien se observa esta relación entre las proporciones estructurales de seres vivos, un ejemplo de ello es la proporción que existe entre las diferentes capas de la concha de un Nautilus.
En este ejemplo utilizaremos el comando **`Solve`** para resolver la ecuación algebráica con la cual se obtiene la relación áurea. Para obtener la razón áurea pensemos en un rectángulo cuyo lado más largo tiene una longitud $1+x$, y el más corto de una unidad. (Ver figura, rectángulo fromado por la parte clara más abscura. La longitud mayor del rectángulo se puede tomar como formada por una unidad, en claro, y la obscura de longitud x. De modo tal que tenemos un rectangulo con lados $1+x$ y 1. Ahora si le quitamos al rectángulo original uno de longitud x por 1, nos quedará otro rectángulo de lados 1 y x , rectángulo obscuro. Entonces, para que tengamos un rectángulo perfecto se tiene que cumplir que la razón entre los lados mayores y menores de ambos rectángulos tiene que ser igual, esto es $(1+x)/1=1/x$. Ésta relación define la razón áurea.

In[30]:=
```
Show[Graphics[{Blue, Rectangle[{1, 0}, {GoldenRatio, 1}]}],
 Graphics[{Orange, Rectangle[{0, 0}, {1, 1}]}]]
```

Out[30]=

Si queremos resolver esta ecuación para la variable x, multiplicamos ambos lados por x y reacomodando obtenemos finalmente $x^2+x-1=0$. Utilizando *Mathematica* para resolver la ecuación cuadrática podemos obtener facilmente la solución,

```
In[31]:= Solve[x^2 + x - 1 == 0, x]
```

$$\text{Out[31]= } \left\{\left\{x \to \frac{1}{2}\left(-1-\sqrt{5}\right)\right\}, \left\{x \to \frac{1}{2}\left(-1+\sqrt{5}\right)\right\}\right\}$$

Ya que la raíz cuadrada de 5 es mayor que uno, la segunda expresión es la única que nos da una longitud positiva y por lo tanto es la solución a nuestro problema. Según la igualdad que planteamos tenemos dos fromas de calcular la relación áurea. Una es sumar la unidad a la raíz positiva y dividir el resultado entre uno, *(1+x)/1*, y la segunda es dividir 1 entre la raíz posotiva, *1/x*. A continuación se obtiene el valor numérico con díez decimales de la relación áurea,

```
In[32]:= N[1 + 1/2 (-1 + Sqrt[5]), 10]
Out[32]= 1.618033989

In[33]:= N[1/(1/2 (-1 + Sqrt[5])), 10]
Out[33]= 1.618033989
```

Mathematica tiene definido este valor en el comando **`GoldenRatio`**, el cual podemos comprobar que es igual al obtenido,

```
In[34]:= N[GoldenRatio, 10]
Out[34]= 1.618033989
```

Ahora el lector podrá entender como fue generado el rectángulo de la figura que se muestra al inicio del Proyecto. El procedimiento puede hacerse un gran número de veces. Esto es, ahora tomar el rectangulo oscuro y posteriormente extraerle un cuadrado de tal forma que se preserve la razón áurea. Esto procedimiento se puede hace cuantas veces se quiera.

Proyecto. La máquina de Atwood.

En este ejemplo nuestra tarea será encontrar la tensión en la cuerda y la aceleración de las masas si se tienen dos masas, m_1 y m_2, unidas por una cuerda que pasa por una polea sin fricción y de masa despreciable. Supongamos que la aceleración de la primer masa se toma como positiva hacia arriba. T representa la tensión en la cuerda, a la aceleración de las masas y g la ecleración debida a la gravedad. Entonces la ecuación de moviento de m_1 es $T-m_1g=m_1$ a y la de m_2 es, $T-m_2g=-m_2a$. Para encontrar la solución al problema se tiene que resolver un sistema de dos ecuaciones con dos incógnitas. Con *Mathematica* lo hacemos utilizando el comando **`Solve`**. Si además queremos que se despliege el resultado más simple utlizamos el comando **`Simplify`**. Finalmente obtenemos la solución de la siguiente forma,

```
In[35]:= Simplify[Solve[{T - m1 g == m1 a, T - m2 g == -m2 a}, {a, T}]]
```

$$\text{Out[35]=}\ \left\{\left\{a \to \frac{g\,(-m_1 + m_2)}{m_1 + m_2},\ T \to \frac{2\,g\,m_1\,m_2}{m_1 + m_2}\right\}\right\}$$

En este ejemplo es evidente como en una sóla línea con *Mathematica* podemos resolver un problema que al menos nos tomaría una página completa de álgebra.

Solución a ecuaciones trascendentales

Una ecuación trascendental es aquella que no se puede resolver usando operaciones aritméticas, elevando a potencias y extrayendo raíces. Un ejemplo es *tan(x)=-x*. Cuando se quieren resolver ecuaciones trascendentales no es de gran ayuda el comando **Solve**. Por ejemplo, si se quieren encontrar las soluciones a la ecuación anterior con el comando **Solve** se obtiene el siguiente resultado:

```
In[36]:= Solve[Tan[x] == -x, x]

Solve::tdep :
  The equations appear to involve the variables to be solved for in an
      essentially non-algebraic way. >>

Out[36]= Solve[Tan[x] == -x, x]
```

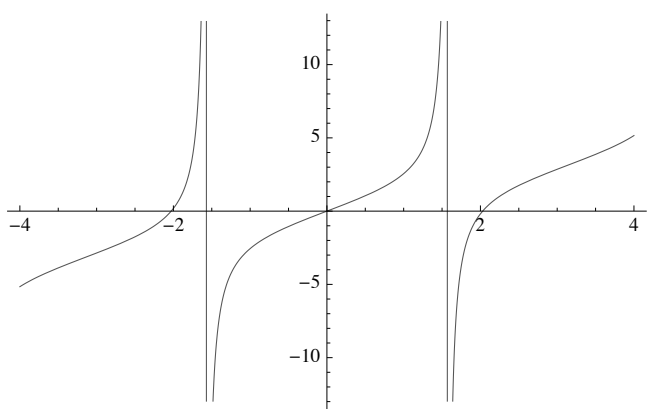
Para poder encontrar numéricamente las raíces de la ecuación trascendental se puede utilizar el comando **FindRoot**. Este comando intentará encontrar numéricamente la primer raíz a partir de un valor numérico inicial, el cual tiene que ser indicado por el usuario.

FindRoot[{*expresión* **==** *expresión***},{***variable, número inical***}]**
Solución de una ecuacion trascendantal cerca del número de inicio.

Lo más prudente para tener una idea de cual es el valor inicial a escoger es graficar primero la función.

```
In[37]:= Plot[Tan[x] + x, {x, -4, 4}]
```

Out[37]=



Por ejemplo, si queremos encontrar una raíz cerca de 0, la sintaxis del comando quedará de la siguiente manera.

```
In[38]:= FindRoot[Tan[x] == -x, {x, 0}]
Out[38]= {x → 0.}
```

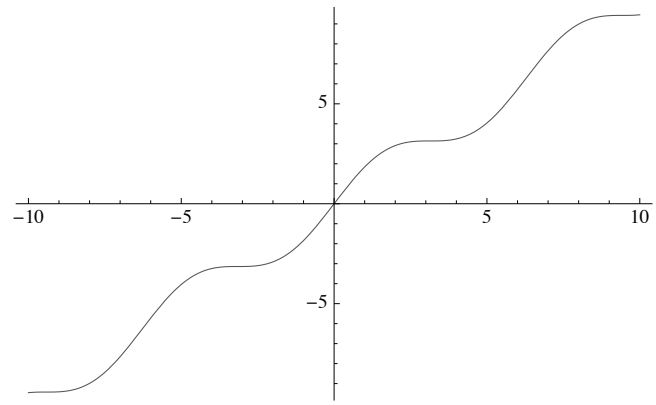
El primer argumento dentro del comando es la ecuación a resolver. En el segundo argumento, entre llaves, encontramos la variable sobre la cual queremos encontrar solución a la ecuación, el segundo elemento indica el número en el cual iniciar la busqueda.
Si se quiere encontrar otra raíz en el ejemplo anterior, digamos la más cerca a dos, se indica de la siguiente manera.

```
In[39]:= FindRoot[Tan[x] == -x, {x, 2}]
Out[39]= {x → 2.02876}
```

La ecuación *x+senx=0* tiene una única solución en *x=0* como podemos observar en la siguiente gráfica.

```
In[40]:= Plot[x + Sin[x] , {x, -10, 10}]
```

Out[40]=



El lector puede observar que sucede si se toma un punto inicial muy lejos de la solución con el comando **FindRoot**.

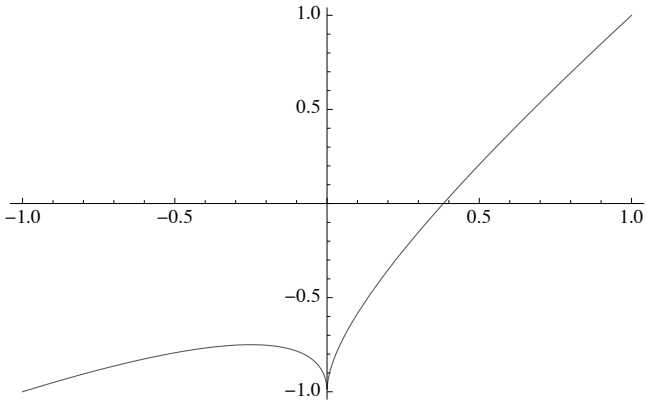
```
In[41]:= FindRoot[x + Sin[x] == 0, {x, 100}]
Out[41]= {x → -4.47869 × 10^-20}
```

En los cálculos numéricos aproximados, las respuestas dependen de la precisión numérica de la computadora donde se hace el cálculo. En este caso, *Mathematica* da como resultado un número extraordinariamente pequeño y cercano a cero, que es la solución exacta.

En el caso en que el método de Newton (que es el predeterminado por *Mathematica*) sea incapaz de obtener una solución con el camando **FindRoot**, el método de la secante se puede intentar. El método de Newton utiliza el valor de la intersección obtenida de la línea tangente del punto inicial, por lo que el método no es de utilidad si no se puede calcular la derivada de la función. Por otro lado, el método de la secante, aunque más lento, utiliza valores de la función en dos puntos, calculando la intersección de la línea secante. Por ejemplo, la ecuación $\sqrt{|x|} + x - 1 = 0$ tiene una solución entre 0 y 1 como se muestra en la gráfica.

```
In[42]:= Plot[Sqrt[Abs[x]] + x - 1, {x, -1, 1}]
```

Out[42]=



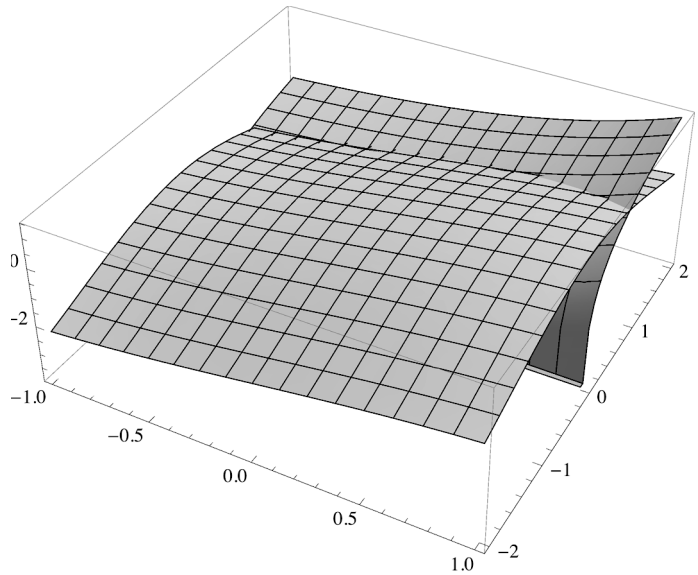
Mathematica no puede obtener la raiz por el método de Newton ya que no puede calcular la derivada de la función. Especificando dos valores iniciales en lugar de uno, en el comando **FindRoot**, indica a *Mathematica* que use el método de la secante, a saber,

```
In[43]:= FindRoot[Sqrt[Abs[x]] + x - 1 == 0, {x, {0, 1}}]
Out[43]= {x → {0.381966, 0.381966}}
```

FindRoot también es útil cuando se tiene un sistema de ecuaciones trascendentales . Por ejemplo, si se quiere encontrar el punto de intersección entre $e^x+lny=2$ y $senx+cosy=1$. Este sistema de ecuaciones tiene solución cerca de los puntos $x=0.5$ y $y=1$, los cuales se pueden determinar a partir de una gráfica.

```
In[44]:= Plot3D[{Exp[x] + Log[y] - 2, Sin[x] + Cos[y] - 1},
          {x, -1, 1}, {y, -2, 2}]
```

Out[44]=



Para resolver el sistema utilizamos la siguiente sintaxis,

```
In[45]:= FindRoot[{Exp[x] + Log[y] == 2, Sin[x] + Cos[y] == 1}, {x, 0.5}, {y, 1}]
Out[45]= {x → 0.624295, y → 1.14233}
```

Finalmente comentaremos que poder controlar el número de dígitos que se desean puede ser de gran utilidad al encontrar soluciones a ecuaciones numéricamente. Esto se hace introduciendo la opción **`WorkingPrecision`**. Esta opción es seguida de una flecha y del número de dígitos que se desean. Por ejemplo,

```
In[46]:= FindRoot[Tan[x] == -x, {x, 2}, WorkingPrecision → 50]
Out[46]= {x → 2.0287578381104342235769711247347143761083800287594}
```

En este caso el número de dígitos utilizados por *Mathematica* para desplegar la solución es de 50.

Proyecto. La ecuación de estado de Dieterici en la forma del virial.

La ecuación de estado para un gas nos dice cúal es la presión de dicho gas a una densidad y temperatura dados. Para un gas ideal, la ecuación de estado es simplemente $p=\rho kT$, donde $k=R/N_A$ es la constante de Boltzmann. Los gases reales sólo siguen la ecuación de estado ideal para densidades muy pequeñas y generalmente se requiere una ecuación de estado más complicada para describirlos. Un ejemplo es la ecuación de estado virial: $p/\rho kT=1+B\rho+C\rho^2+\cdots$. Los coeficientes B, C, etc., se llaman coeficientes viriales y son funciones únicamente de la temperatura.

Existen muchas otras ecuaciones de estado, otro ejemplo es la ecuación de Dieterici. En este ejemplo identificaremos los tres primeros coeficientes en el desarollo del virial para la ecuación de estado de Dieterici. Para llevar a cabo esta tarea, utilizaremos algunos de los comandos descritos en esta sección. La ecuación de estado de Dieterici $p(v-b)=RT\ exp[-a/RTv]$ es válida sólo cuando el gas se aparta poco de su comportamiento como gas ideal, pero es de suma utilidad ya que representa cualitativamente la ecuación de estado de un gas en todo el intervalo gas líquido. Las constantes a y b son constantes particulares de cada gas, cuya interpretación física puede darse en términos de las fuerzas intermoleculares que actuan entre moléculas que componen el mismo gas.

Dividiendo por ρkT ambos miembros de la ecuación de estado de Dieterici, recordando que la densidad molar $\rho=N_A/v$, se tiene que $p/\rho kT=exp[-a\rho/kT]/(1-(b/\rho))$. Desarrollando el binomio y la exponencial de esta última ecuación obtenemos que

$$p\rho/kT=[1+b/\rho+(b/\rho)^2+\cdots][\,1-(a\rho/kT)+(a\rho/kT)^2+\cdots].$$

Ahora, utilizando *Mathematica* podemos identificar el término independiente, que nos daría el primer coeficiente del virial, e identificando los terminos que son independientes y los que van como potencias de ρ obtendremos los coeficientes del virial respectivamente.
Primero definimos el desarrollo del binomio y de la exponencial hasta segundo orden

```
In[47]:= DB = 1 + b / ρ + (b / ρ)^2
```

$$\text{Out[47]}= 1+\frac{b^2}{\rho^2}+\frac{b}{\rho}$$

```
In[48]:=
DE = 1 - (a ρ / (k T)) + (1 / 2) ((a ρ) / (k T))^2
```

$$\text{Out[48]}= 1-\frac{a\,\rho}{k\,T}+\frac{a^2\,\rho^2}{2\,k^2\,T^2}$$

A continuación escribimos la ecuación de Dieterici introduciendo los desarrollos

```
In[49]:= ED = DB DE
```

$$\text{Out[49]}= \left(1+\frac{b^2}{\rho^2}+\frac{b}{\rho}\right)\left(1-\frac{a\,\rho}{k\,T}+\frac{a^2\,\rho^2}{2\,k^2\,T^2}\right)$$

Utilizando el comado **Expand** podemos apreciar toda la expresión

```
In[50]:= Expand[ED]
```

$$\text{Out[50]}= 1 + \frac{a^2 b^2}{2 k^2 T^2} - \frac{a b}{k T} + \frac{b^2}{\rho^2} + \frac{b}{\rho} - \frac{a b^2}{k T \rho} + \frac{a^2 b \rho}{2 k^2 T^2} - \frac{a \rho}{k T} + \frac{a^2 \rho^2}{2 k^2 T^2}$$

Para obtener el coefeciente *B* del desarrollo virial, tenemos que encontrar el coeficiente que va como ρ, para ello utilizaremos el comando **Coefficient**,

```
In[51]:= Coefficient[ED, ρ]
```

$$\text{Out[51]}= \frac{a^2 b}{2 k^2 T^2} - \frac{a}{k T}$$

Utilizando el comando Simplify podemos obtener una expresión más simple que la anterior

```
In[52]:= Simplify[%]
```

$$\text{Out[52]}= \frac{a\,(a\,b - 2\,k\,T)}{2\,k^2\,T^2}$$

A continuación obtendremos del mismo modo que en el caso anterior el tercer coeficiente del virial, *C*.

```
In[53]:= Coefficient[ED , ρ^2]
```

$$\text{Out[53]}= \frac{a^2}{2 k^2 T^2}$$

Finalmente podemos concluir que los dos primeros coeficientes del virial para la ecuación de estado de Dieterici son: $B=a\,(a\,b-2\,k\,T)/2\,k^2\,T^2$ y $C=a^2/2\,k^2\,T^2$.

Proyecto. Difracción de Fraunhofer para una abertura circular

La difracción es la desviación que sufren las ondas alrededor de bordes y esquinas. Se produce cuando una porción de un frente de ondas se ve cortado o interrumpido por una barrera u obstáculo. Por ejemplo, si hacemos incidir un rayo de luz por una placa con un pequeño orificio circular y del otro lado ponemos una pantalla, podremos ver sobre esta una serie de regiones de cero intensidad y otras de máxima intensidad, este diagrama de difracción tiene una forma muy peculiar. En el centro se observa un círculo luminoso rodeado por anillos oscuros y brillantes intercalados. En este ejemplo mostraremos como calcular los dos primeros anillos de cero intencidad del patron de difracción. Para ello tendremos que resolver una ecuación trascendental. La intensidad de luz, ϕ, reflejada en una pantalla al hacer incidir luz por una objeto con una abertura circular en el centro, es proporcional a $J_1[(2\pi a/\lambda)\,\text{sen}\alpha]/\text{sen}\alpha$ en donde J_1 es la función de Bessel de la primera especie de orden 1, λ es la longuitud de onda incidente, y a es el radio de la abertura. α es el ángulo medido a partir del centro de la abertura al cual se quiere saber la intensidad.

Supongamos que la luz que incide es color verde con una longuitud de onda $\lambda = 5.5\text{x}10^{-5}$ cm a una abertura circular de radio $a=0.5$ cm. Si queremos saber en que regiones de la pantalla la intensidad es cero, lo que tenemos que encontrar son los valores de α para los cuales la ecuación de la intensidad se hace cero. Aunque a primera vista aparantemente el problema resulta complicado, en realidad la solución se obtendría simplemente si se encuentran los dos primeros valores para los cuales la función de Bessel $J_1(x)$ se hace cero y se igualan al argumento, $x=2\pi a\ sen\alpha\ \lambda$. Al despejar α de esta última igualdad encontraremos a que ángulos se obtienen los dos primeros discos de sombra.
Primero definimos el valor de las constantes.

```
In[54]:= a = 0.5 cm
Out[54]= 0.5 cm

In[55]:= λ = 5.5 10^-5 cm
Out[55]= 0.000055 cm
```

Ahora encontremos las dos primeras raíces de la función de Bessel. Estos dos primeros valores están cerca de 3 y 7, esto se puede observar de hacer la gráfica de la función.

```
In[56]:= FindRoot[BesselJ[1, x] == 0, {x, 3}]
Out[56]= {x → 3.83171}

In[57]:= FindRoot[BesselJ[1, x] == 0, {x, 7}]
Out[57]= {x → 7.01559}
```

Finalmente, encontramos los valores de los ángulos de igualar el argumento de la función de Bessel a cada una de las dos raíces y de despejar α. *Mathematica* nos da los valores de estos ángulos en radianes.

```
In[58]:= ArcSin[(3.83171 λ) / (2 Pi a)]
Out[58]= 0.0000670819
In[59]:= ArcSin[(7.01559 λ) / (2 Pi a)]
Out[59]= 0.000122822
```

Se deja al lector como ejercicio convertir estos ángulos a grados.

Ejercicios

1.-Factorizar $a+2\sqrt{a}\sqrt{b}+b$

2.-Factorizar x^2-3

3.-Expandir $(a-1)(b^2+1)/(b-5)^5$, y encontrar el valor de la expresión si $a=7$ y $b=11$.

4. Resuelva $x^4+3x^3+x^2-x=1$, utilizando *Mathematica* y a mano, comprobar que las soluciones sean correctas.

5. Resolver $2sen^2x+1=3senx$ para $senx$ y para x.

6. Resuelva el siguiente sistema de ecuaciones, comprobar que las soluciones sean correctas, $(x+y)/(1+xy)=a$ y $(x-y)(1-xy)=b$.

7. El doble de un número supera en 9 al triple de otro, mientras que 12 veces el segundo excede en 12 unidades el séptuplo del primero. Hallar ambos números.

8. Dos cargas fijas, de $1x10^{-6}$ C y $-3x10^{-6}$ C, están separadas 10 cm entre sí. En dónde se debe colocar una tercera carga para que no sienta fuerza alguna?

9. Si la base de un rectángulo disminuye 2 cm y la altura aumenta 2, su área se incrementa en 16 cm^2. Si la base aumenta 5 cm y la altura disminuye 3, al área aumenta 15 cm^2. Encontrar el área del rectángulo original.

10. En cierto número de tres cifras el dígito de las centenas es una unidad menor que el de las decenas y la suma de los tres dígitos es 17. Si se intercambian los dígitos de las unidades y las centenas, el número disminuye en 495. Encontrar el número original.

11. Encontrar una línea recta que pasa por los puntos (2,5) y (7,9).

12. Resolver el siguiente sistema de ecuaciones y encontrar la solución cuando $z=1$, $z=2$, y $z=3$. $w+x+y+z=3$, $2w+3x+4y+5z=3$, y $w-x+y-z=4$.

13. Encontrar con 20 cifras significativas un número el cual al sumarlo a su cuadrado y a su cubo es igual a treinta.

14. Encontrar el punto de intersección de las funciones $f(x)=x^3-7x^2+2x+20$ y $g(x)=x^2$.

15. Encontrar una solución a la ecuación $senx=2$.

16. Encontrar los puntos de intersección de la parábola $y=x^2+x-10$ con el círculo $x^2+y^2=25$.

17. Encontrar una solución del sistema de ecuaciones, $x+y+2=6$, $senx+cosy+tanz=1$, y $e^x+\sqrt{y}+1/z=5$, en la cercanía del punto (1,2,3).

18. Resolver para x, $e^{2x}+e^x=3$.

19. Obtener una solución aproximada con al menos veinte dígitos después del punto de la ecuación $100/x=x/(x+1)$, cerca de 5000.

20. Resolver la ecuación $e^{-x}=x$ por el método de Newton y de la secante para cien iteraciones y comparar la precisión de ambos resultados. Lea el manual en línea para saber como especificar el método y el número máximo de iteraciones al usar el comando **`FindRoot`**.

Capítulo 4

Trabajando con listas y tablas

Construcción de un listado

Las listas están formadas por una colección de objetos. *Mathematica* ofrece una gran variedad de comandos para manipularlas, en este capítulo explicaremos el uso de los más útiles.
En *Mathematica* las listas se construyen poniendo los elementos entre llaves y separados por comas, alternativamente el comando **`List`** se puede utilizar para indicar los elementos que constituyen el listado.

`List[`*elementos separados por comas*`]`**	Define una lista.

A continuación generaremos una lista que contenga las vocales.

```
In[1]:= List[a, e, i, o, u]
Out[1]= {a, e, i, o, u}
```

A las listas se las puede nombrar, lo cuál es útil en la manipulación de éstas. Por ejemplo, supongamos que tenemos un conjunto de números formados por los enteros del 1 al 5, a los que queremos elevar al cuadrado y también encontrar su raíz cuadrada. Todas las operaciones sobre la lista se aplican a cada miembro de ésta, como se observa a continuación,

```
In[2]:= lista1 = {1, 2, 3, 4, 5}
Out[2]= {1, 2, 3, 4, 5}
In[3]:= lista1^2
Out[3]= {1, 4, 9, 16, 25}
In[4]:= Sqrt[lista1]
Out[4]= {1, √2, √3, 2, √5}
```

En ocasiones se desean generar listas de números, de un número *m* a otro *n* con un incremento *d,* para ello *Mathematica* cuenta con el comando **`Range`**.

`Range[`*m,n,d*`]`**	Lista de enteros de *n* a *m* en incrementos de *d*.

Si queremos hacer una lista que contenga los enteros del 6 al 10, podemos hacerlo de la siguiente manera:

```
In[5]:= lista2 = Range[6, 10, 1]
Out[5]= {6, 7, 8, 9, 10}
```

También es posible efectuar operaciones entre listas, estas operaciones se llevaran a cabo entre cada uno de los elementos respetando el orden de los elementos de las listas involucradas. Supongamos que queremos sumar, multiplicar, dividir y elevar los elementos de la **lista1** con los de la **lista2**. Todos las operaciones se llevaran a cabo entre los primeros miembros de la primera y de la segunda lista, entre el segundo miembro de la primera lista con el de la segunda, y así sucesivamente.

```
In[6]:= lista1 + lista2
Out[6]= {7, 9, 11, 13, 15}
In[7]:= lista1 * lista2
Out[7]= {6, 14, 24, 36, 50}
In[8]:= lista1 / lista2
Out[8]= {1/6, 2/7, 3/8, 4/9, 1/2}
In[9]:= lista1 ^ lista2
Out[9]= {1, 128, 6561, 262 144, 9 765 625}
```

Mathematica también ofrece comandos parecidos a **Range** con los que se pueden generar listados de caracteres. El comando **Characters** genera un listado de los caracteres de las cadenas de texto introducidas entre comillas.

Characters["*cadena*"**]**	Lista con elementos de cada letra de la cadena de texto.

Por ejemplo:

```
In[10]:= Characters["Mathematica"]
Out[10]= {M, a, t, h, e, m, a, t, i, c, a}
```

Si queremos generar una lista con letras de la *a* a la *g*, lo hacemos de la siguiente manera:

```
In[11]:= CharacterRange["a", "g"]
Out[11]= {a, b, c, d, e, f, g}
```

Con el comando **CharacterRange** se puede generar una lista que ponga una secuencia de letras especificando con cuáles se quiere iniciar y finalizar. Por ejemplo, si se quiere una lista con las 27 letras del alfabeto se hace de la siguiente forma,

CharacterRange["*a*","*z*"**]**	Lista con letras de la *a* a la *z*.

Manipulación de un listado

Iniciaremos esta sección explicando comandos que son capaces de extraer información de una lista, ellos son: **Length**, **First** y **Last**. Como sus nombres en inglés lo indican, el primero de ellos nos dice cuantos elementos constituyen al listado, el segundo nos devuelve el elemento en la primera posición, y el último de ellos nos indica cual es el último elemento de la lista.

Length[*lista***]**	Número de elementos que constituyen una lista.
First[*lista***]**	Despliega el primer elemento de una lista.
Last[*lista***]**	Despliega el último elemento de una lista.

En el siguiente ejemplo aplicaremos estos comandos a la primera lista que generamos,

```
In[12]:= Length[lista1]
Out[12]= 5
In[13]:= First[lista1]
Out[13]= 1
In[14]:= Last[lista1]
Out[14]= 5
```

Si se desea extraer algunos de los miembros de un listado en particular, el comando **Part** realiza dicha tarea. Como primer argumento del comando se pone el nombre de la lista de interés, el segundo argumento dentro del comando lo forma el *n*-ésimo elemento a partir del primero si éste es positivo.

Part[*lista, n***]**	Despliega el *n*-ésimo elemento de *lista* contando desde el primero si *n* es positivo. Si *n* es negativo la posición se cuenta a partir del último elemento.

En el ejemplo siguiente encontraremos utilizando este comando el primer y el segundo elemento de la **lista2**, generada en un ejemplo anterior,

```
In[15]:= Part[lista2, 1]
Out[15]= 6
```

```
In[16]:= Part[lista2, -3]
Out[16]= 8
```

Si el segundo argumento en el comando **Part** es negativo, nos regresa el *n*-ésimo elemento a partir del último elemento de la lista. A continuación mostraremos como obtener el penúltimo y último elemento de la **lista2**,

```
In[17]:= lista2
Out[17]= {6, 7, 8, 9, 10}
In[18]:= Part[lista2, -2]
Out[18]= 9
In[19]:= Part[lista2, -1]
Out[19]= 10
```

Es muy útil conocer las formas abreviadas del comando **Part**; ambas son muy similares a la notación usada en el lenguaje C para denotar los elementos de un arreglo *(array)* de varios números, pues utilizan un sufijo que consiste en un entero encerrado entre dos corchetes: La sintaxis es la siguiente:

*lista***[[** *n***]]**	Notación abreviada para referirse al *n*-ésimo elemento de una lista, escribiendo a continuación del símbolo para la lista dos paréntesis cuadrados que encierran al entero *n*.
lista⟦*n*⟧	Otra notación más compacta para referirse al *n*-ésimo elemento de una lista. Para escribir los caracteres de corchetes dobles hay que teclear ***Esc*** seguido de dos juegos de corchetes y luego ***Esc*** de nuevo.

Si se desea extraer varios elementos de una lista esto puede hacerse con el comando **Take**. En este comando se puede especificar de qué elemento *m* a qué elemento *n* se quiere llevar a cabo la extracción para formar una nueva lista. Utilizaremos este comando y la **lista1** para extraer de ella una nueva lista en la que no se incluyan ni el primero, ni el último elemento; esto es, del segundo al cuarto elemento de **lista1**.

Take[*lista, {n,m}***]**	Despliega del *n*-ésimo al *m*-ésimo elementos de *lista*.

```
In[20]:= Take[lista1, {2, 4}]
Out[20]= {2, 3, 4}
```

Take también puede ser utilizado para tomar sólo un término de la lista. Por ejemplo, si quisiéramos tomar el tercer miembro de la **lista2**, lo haríamos de la siguiente manera:

```
In[21]:= Take[lista2, {3}]
Out[21]= {8}
```

Si lo que se quiere es borrar un elemento definitivamente de un lista, se utiliza el comando **Delete** especificando el elemento a borrar.

Delete[*lista, n***]**	Borra definitivamente el *n*-ésimo elemento de una lista.

Por ejemplo:

```
In[22]:= Delete[{1, 2, 3, 4}, 2]
Out[22]= {1, 3, 4}
```

Si ahora lo que quisiéramos hacer es obtener una nueva lista de una lista existente, borrando elementos de esta última, se utiliza el comando **Drop** exactamente con la misma estructura que el comando **Take**. Utilizaremos los mismo ejemplos que en el párrafo anterior para ver como funciona el comando **Drop**.

Drop[*lista, {n,m}***]**	Borra definitivamente del *n*-ésimo al *m*-ésimo elementode una lista.

```
In[23]:= Drop[lista1, {2, 4}]
Out[23]= {1, 5}
```

En este caso lo que hizo el comando **Drop** fue darnos los elementos de la **lista1** quitando los elementos del segundo al cuarto.

```
In[24]:= Drop[lista2, {3}]
Out[24]= {6, 7, 9, 10}
```

En este segundo caso, el comando sólo quitó el tercer elemento de la lista **lista2**.

Si lo que se quiere es insertar un nuevo elemento a una lista se utiliza el comando **Insert**. En este comando es necesario especificar el nombre de la lista, el elemento a insertar y la posición que se quiere que ocupe el nuevo elemento.

Insert[*lista, elemento, n***]**	Inserta a *elemento* como *n*-ésimo elemento de *lista*.

Por ejemplo si queremos agregar el número 11 al final de la **lista2**, se hace de la siguiente manera,

```
In[25]:= Insert[lista2, 11, 6]
Out[25]= {6, 7, 8, 9, 10, 11}
```

También se pueden utilizar los comandos **Append** y **Prepend** para insertar nuevos elementos en una lista ya existente. El primero de los comandos inserta el elemento deseado a la derecha del último elemento, mientras que **Prepend** inserta este elemento a la izquierda del primero.

Append[*lista, elemento***]**	Inserta a *elemento* al final de *lista*.
Prepend[*lista, elemento***]**	Inserta a *elemento* al principio de *lista*.

Podemos repetir el ejemplo anterior ahora usando el comando **Append**:

```
In[26]:= Append[lista2, 11]
Out[26]= {6, 7, 8, 9, 10, 11}
```

Ahora a **lista2** le podemos insertar un 5 antes del primer elemento, que es el 6, con el comando **Prepend** como se muestra a continuación:

```
In[27]:= Prepend[lista2, 5]
Out[27]= {5, 6, 7, 8, 9, 10}
```

Rotar los elementos de una lista se puede hacer con *Mathematica* utilizando el comando **RotateLeft** o **RotateRight**. **RotateLeft** rota el primer elemento de la lista al final de la misma. Esta acción se puede llevar a cabo el número de veces deseado.

RotateLeft[*lista, n***]**	Rota el primer elemento al final de la lista *n* veces.

```
In[28]:= RotateLeft[lista2, 1]
Out[28]= {7, 8, 9, 10, 6}
In[29]:= RotateLeft[lista2, 2]
Out[29]= {8, 9, 10, 6, 7}
```

Como el lector podrá observar, en los dos primeros ejemplos se muestra cómo mover el primer miembro de la lista al final de ella, mientras que en el segundo se movieron los dos primeros. Rotar todos los elementos una vez, haciendo que los elementos roten el número de veces correspondiente al número de elementos del que consta el listado, devuelve los elementos de la lista en el orden original

```
In[30]:= RotateLeft[lista2, Length[lista2]]
Out[30]= {6, 7, 8, 9, 10}
```

El comando **RotateRight** rota el último elemento de la lista al inicio de la misma.

RotateRight[*lista, n***]**	Rota el primer elemento al final de la lista *n* veces.

Como en el siguiente ejemplo.

```
In[31]:= RotateRight[lista2, 1]
Out[31]= {10, 6, 7, 8, 9}
In[32]:= RotateRight[lista2, 2]
Out[32]= {9, 10, 6, 7, 8}
```

Para rotar todos los elementos una vez, lo hacemos igual que en el ejemplo anterior, con el mismo efecto:

```
In[33]:= RotateRight[lista2, Length[lista2]]
Out[33]= {6, 7, 8, 9, 10}
```

Dos comandos de gran utilidad para ordenar elementos de una lista son **Sort** y **Reverse**. El primero ordena los números, de acuerdo con su valor, de menor a mayor y en forma lexicográfica las letras, poniendo al final las letras mayúsculas. El comando **Reverse** ordena en sentido contrario a **Sort**. Si en la lista se encuentran números y letras, primero se acomodan las letras en el orden especificado anteriormente.

Sort[*lista***]**	Ordena los elementos de una lista de menor a mayor.
Reverse[*lista***]**	Ordena los elementos de una lista de mayor a menor.

Esto queda de manifiesto en el siguiente ejemplo,

```
In[34]:= Sort[{2, 5, 1, a , A}]
Out[34]= {1, 2, 5, a, A}
In[35]:= Reverse[{2, 5, 1, a , A}]
Out[35]= {A, a, 1, 5, 2}
```

A continuación mostraremos como *Mathematica* es capaz de concatenar dos listas, para ello se utiliza el comando **Join** y en su argumento las listas a concatenar.

Join[*lista1, lista2***]**	Concatena listas.

Si se quieren concatenar las listas **lista1** y **lista2**, se hace de la siguiente manera,

```
In[36]:= Join[lista1, lista2]
Out[36]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Para terminar esta subsección discutiremos cómo utilizar algunos comandos que nos permiten manejar las listas como conjuntos. Para ello primero definamos nuestro conjunto universo,

```
In[37]:= universo = Range[10]
Out[37]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Ahora definamos un par de conjuntos:

```
In[38]:= A = {1, 4, 2, 5, 3, 7, 5, 9}
Out[38]= {1, 4, 2, 5, 3, 7, 5, 9}
In[39]:= B = {4, 5, 4, 4, 2, 3, 7, 8}
Out[39]= {4, 5, 4, 4, 2, 3, 7, 8}
```

Si se quiere hacer la unión de estos conjuntos se emplea el comando **Union**,

Union[*conjunto1, conjunto2***]**	Unión de conjuntos.

```
In[40]:= Union[A, B]
Out[40]= {1, 2, 3, 4, 5, 7, 8, 9}
```

Es importante distinguir entre el resultado de **Union** (unión de conjuntos) y el de **Join** (concatenación de listas)**.** El primero combina los elementos de sus argumentos, pero elimina los que están repetidos, mientras que **Join** los pone uno tras otro, sin importar cuántas veces aparezcan en las listas de sus argumentos.

Si se quiere encontrar la intersección de ambos conjuntos se utiliza el comando **Intersection**,

Intersection[*conjunto1, conjunto2***]**	Intersección de conjuntos.

```
In[41]:= Intersection[A, B]
Out[41]= {2, 3, 4, 5, 7}
```

Si se quiere encontrar el conjunto complemento se utiliza el comando **Complement**,

Complement[*conjuntoUniverso,conjunto1***]**	Complemento del *conjunto1* en el conjunto *conjuntoUniverso*.

```
In[42]:= Complement[universo, A]
Out[42]= {6, 8, 10}
In[43]:= Complement[universo, B]
Out[43]= {1, 6, 9, 10}
```

Este último ejemplo muestra como obtener el complemento de la unión de **A** con **B**.

```
In[44]:= Complement[universo, A, B]
Out[44]= {6, 10}
```

Construcción de una tabla

Cuando los elementos de una lista pueden ser generados por una fórmula matemática, el comando **Table** nos proporciona una forma simple de hacer la lista. La forma más general del comando es la siguiente,

Table[*expr*, **{***n*, n_o, n_1, *m***}]**	Evaluación de la expresión *expr* de n_0 a n_1 en incrementos de tamaño *m*.

En la expresión*n* será evaluadoda de n_o a n_1 en incrementos de tamaño *m*. Si *m* es omitido de la estructura del comando, por omisión *Mathematica* toma incrementos de tamaño 1. En el siguiente ejemplo construiremos una lista de diez monomios con potencias sucesivas del 1 al 10:

```
In[45]:= Table[x^n, {n, 1, 10}]
Out[45]= {x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^10}
```

En el siguiente ejemplo generaremos también monomios pero ahora con potencias pares sucesivas del 1 al 10:

```
In[46]:= Table[x^n, {n, 2, 10, 2}]
Out[46]= {x^2, x^4, x^6, x^8, x^10}
```

Ya que la tabla queda definida como un listado, podemos aplicar todos los comandos utilizados para manipular los listados en las tablas. Por ejemplo, si queremos encontrar el número de elementos que conforman la tabla generada anteriormente, utilizamos el comando **Length**:

Length[*tabla***]**	Número de elementos que constituyen una tabla.

```
In[47]:= Length[Table[x^n, {n, 1, 10}]]
Out[47]= 10
```

También es importante resaltar que la expresión que se evalua dentro del comando **Table** (para formar una lista) puede ser a su vez una lista, es decir, **Table** puede generar listas de dos, tres y más dimensiones; como en el ejemplo siguiente, en el cuál generaremos una tabla bidimensional:

```
In[48]:= Table[{n / 2, Cos[n / 2]}, {n, -2, 2}]
```

$$\text{Out[48]= } \left\{\{-1, \text{Cos}[1]\}, \left\{-\frac{1}{2}, \text{Cos}\left[\frac{1}{2}\right]\right\}, \{0, 1\}, \left\{\frac{1}{2}, \text{Cos}\left[\frac{1}{2}\right]\right\}, \{1, \text{Cos}[1]\}\right\}$$

Cundo se tiene una expresión con dos o más variables a evaluar, el comando **Table** se utiliza de la siguiente manera:

```
In[49]:= Table[n * k, {n, 1, 2, 1}, {k, 1, 5, 1}]
Out[49]= {{1, 2, 3, 4, 5}, {2, 4, 6, 8, 10}}
```

Es importante señalar el orden en que se mueven los índices en este ejemplo. Primero, *Mathematica* asigna a *n* los valores a tomar y posteriormente los valores de *k*. Por lo tanto la primer lista de la lista se genera tomando *n=1* y multiplicando por todos los valores que toma *k*. Posteriormente, para generar la siguiente lista, *Mathematica* toma *n=2* y también se multiplica por todos los valores que puede tomar *k*. Como el lector habrá notado ya, en este ejemplo lo que hemos generado es una lista de listas. Tenemos en primera instancia una lista compuesta por dos elementos, pero a su vez cada elemento es una lista de cinco elementos. *Mathematica* representa tablas y matrices como una lista de listas sin hacer internamente distinción entre ellas. Comúnmente es más conveniente utilizar el comando **MatrixForm** o **TableForm** ya que visualmente nos permite ver con claridad las estructura que hemos creado. Una tabla o matriz formado por *m* hileras y *n* columnas, es una lista formada por *m* sublistas de *n* elementos cada una.

TableForm[*tabla***]** o bien, *tabla***//TableForm**	Da formato tabular (de columnas) a *tabla*.

A continuación mostraremos cómo se despliega una tabla con el comando **TableForm**.

```
In[50]:= TableForm[Table[n * k, {n, 0, 4, 2}, {k, 0, 10, 2}]]
Out[50]//TableForm=
0  0  0   0   0   0
0  4  8   12  16  20
0  8  16  24  32  40
```

Por omisión el comando alinea los elementos de la tabla a la izquierda. Esto puede ser controlado introduciendo dentro del comando la opción **TableAlignments->**. Las opciones son: **Left**, **Right** y **Center**.

TableForm[*tabla,* **TableAlignments->** *Center o Left o Right***]**	Alinea los elementos de una tabla.

Por ejemplo, si queremos centrar los elementos de la tabla anterior:

```
In[51]:= TableForm[Table[n * k, {n, 0, 4, 2}, {k, 0, 10, 2}],
           TableAlignments → Center]
```

Out[51]//TableForm=

```
0  0   0   0   0   0
0  4   8   12  16  20
0  8   16  24  32  40
```

En ocasiones es necesario etiquetar cada una de las hileras y/o las columnas, para ello se utiliza la opción. **TableHeadings**. Su estructura es la siguiente,

TableForm[*tabla,* **TableHeadings->{***etiquetas de hileras, etiquetas de columnas***}]**
Etiqueta cada una de las hileras o las columnas de una tabla.

En caso de que no se quieran imprimir algunas de las etiquetas, se escribe la palabra **None**.

En el siguiente ejemplo haremos una lista en la cual compararemos la temperatura Celsius, de -40 a 40 ºC, con su equivalente en Fahrenheit.

```
In[52]:= f = (9 / 5) C + 32
```

$$\text{Out[52]=}\quad 32 + \frac{9\,C}{5}$$

```
In[53]:= TableForm[Table[{C, N[f, 4]}, {C, -40, 40, 10}],
           TableHeadings → {None, {"Celsius", "Fahrenheit"}}]
```

Out[53]//TableForm=

Celsius	Fahrenheit
-40	-40.00
-30	-22.00
-20	-4.000
-10	14.00
0	32.00
10	50.00
20	68.00
30	86.00
40	104.0

Cuando se quiere evaluar una función definida por el usuario para cada elemento de una lista, esto se lleva a cabo en *Mathematica* con el comando **Map**.

Map[*función, lista***]**	Evalúa la función con cada uno de los elementos de *lista* como argumento.

```
In[54]:= Map[func, {a, b, c, d, e}]
Out[54]= {func[a], func[b], func[c], func[d], func[e]}
```

Supongamos que hemos definido la función $f(x)=x^2+1$ y enseguida queremos evaluarla de $x=0$ a $x=20$, variando x en incrementos de 2 unidades; ésto se hace de la siguiente forma:

```
In[55]:= lista3 = Table[n, {n, 0, 20, 2}]
Out[55]= {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
In[56]:= f1[x_] = x^2 + 1
Out[56]= 1 + x^2
In[57]:= Map[f1, lista3]
Out[57]= {1, 5, 17, 37, 65, 101, 145, 197, 257, 325, 401}
```

En *Mathematica* muchas de las funciones y comandos incorporados, poseen el atributo *Listable*, esto hace que al tener una lista como argumento, el resultado sea a su vez un listado con el resultado de aplicar la función a cada elemento de la lista original. De este modo no es necesario utilizar el comando **Map**.

Attributes[*función***]**	Atributos de una función.

Por ejemplo si queremos ver los atributos de la función f1, utilizamos el comando **Attributes**:

```
In[58]:= Attributes[f1]
Out[58]= { }
```

Lo que nos muestra *Mathematica* es que no tiene ninguno. Ahora supongamos que queremos darle la propiedad *Listable*. Esto lo hacemos con el comando **SetAttributes.**

SetAttributes[*función***]**	Asigna atributos a una función.

```
In[59]:= SetAttributes[f1, Listable]
```

Ahora podemos verificar que se le asignó el nuevo atributo de la función:

```
In[60]:= Attributes[f1]
Out[60]= {Listable}
```

A continuación repetiremos el ejemplo anterior insertando en el argumento de **f1** la **lista3**:

```
In[61]:= f1[lista3]
Out[61]= {1, 5, 17, 37, 65, 101, 145, 197, 257, 325, 401}
```

Proyecto. La viscosidad del agua

En el siguiente proyecto mostraremos cómo a partir de una serie de datos experimentales, que nos muestran la viscosidad del agua a diferentes temperaturas, es posible encontrar una ecuación capaz de describir su comportamiento. Es sabido que los datos experimentales entre viscosidad y temperatura son bien descritos por una ecuación exponencial tipo Arrhenius,

$$\eta = \eta_0 \, e^{-A/T}$$

en donde η es la viscosidad dada en unidades del Sistema Internacional en Pa·s, η_0 es una viscosidad de referencia, A una constante y T la temperatura absoluta. Para comparar los datos experimentales con la ecuación teórica es pertinente hacer un cambio de variables, de manera que las nuevas variables (X, Y) estén relacionadas de manera lineal. A la relación tipo Arrhenius se le saca el logaritmo natural en ambos lados de la ecuación, con lo cual tenemos que escogiendo $Y = \log \eta$ y $X = 1/T$ se tendrá que

$$Y = \log \eta_0 - AX.$$

Debido a esto, si calculamos el logaritmo natural de los datos obtenidos para la viscosidad experimental, y los graficamos contra el inverso de la temperatura absoluta, si obtenemos un comportamiento lineal, será sencillo encontrar el ajuste a una ecuación tipo Arrhenius. Además, al comparar estos datos transformados con la ecuación teórica podremos encontrar los valores de η_0 y A que nos dan el mejor ajuste. Los datos experimentales son los siguientes:

```
In[62]:= datos = {{0, 17.3}, {10, 13.07}, {20, 10.02},
        {30, 7.98}, {40, 6.53}, {50, 5.47}, {60, 4.67},
        {70, 4.04}, {80, 3.54}, {90, 3.15}, {100, 2.82}};
```

La primer entrada de cada par es la temperatura en la escala Celsius y la segunda la viscosidad en Pa·s. Utilizando el comando **TableForm** podemos ver los datos de manera más clara.

```
In[63]:= TableForm[datos, TableHeadings → {None, {"T/ºC", "ρ/Pa·s"}}]
```

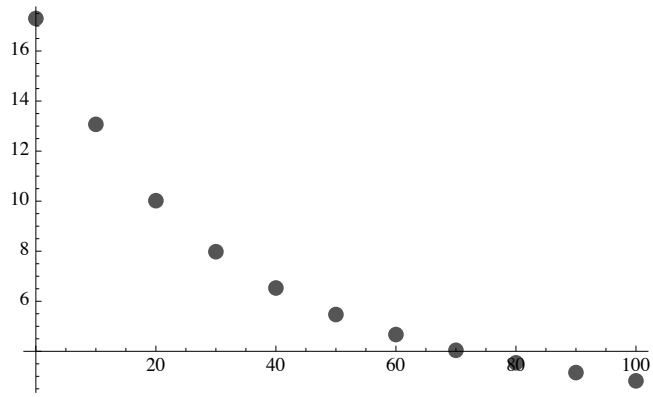
Out[63]//TableForm=

T/ºC	ρ/Pa·s
0	17.3
10	13.07
20	10.02
30	7.98
40	6.53
50	5.47
60	4.67
70	4.04
80	3.54
90	3.15
100	2.82

A continuación graficaremos los datos para ver la forma de la variación de la temperatura respecto a la viscosidad.

```
In[64]:= ListPlot[datos, PlotStyle → PointSize[0.025]]
```

Out[64]=



Para transformar nuestra lista de datos experimentales de tal modo que podamos compararlos con la ecuación teórica, es necesario primero definir la función de transformación. Esta función tendrá primero que pasar la temperatura a Kelvin y posteriormente calcular su recíproco. Con respecto a la viscosidad tendrá que calcular el logaritmo natural. Dicha función la podemos definir de la siguiente manera:

```
In[65]:= ftv[{temp_, vis_}] := {1 / (temp + 273.15), Log[vis]}
```

Ahora podemos aplicar esta función a los datos experimentales mapeando la función sobre nuestro listado. Además podremos guardar el resultado en una nueva lista a la que llamaremos **datos2**.

```
In[66]:= datos2 = Map[ftv, datos]

Out[66]= {{0.00366099, 2.85071},
          {0.0035317, 2.57032}, {0.00341122, 2.30458},
          {0.0032987, 2.07694}, {0.00319336, 1.87641},
          {0.00309454, 1.69928}, {0.00300165, 1.54116},
          {0.00291418, 1.39624}, {0.00283166, 1.26413},
          {0.00275368, 1.1474}, {0.00267989, 1.03674}}
```

Una vez más podemos utilizar el comando **TableForm** para visualizarlos.

```
In[67]:= TableForm[datos2, TableAlignments -> Center,
          TableHeadings → {None, {"T^-1/K^-1", "Log[η/Pa s]"}}]
```

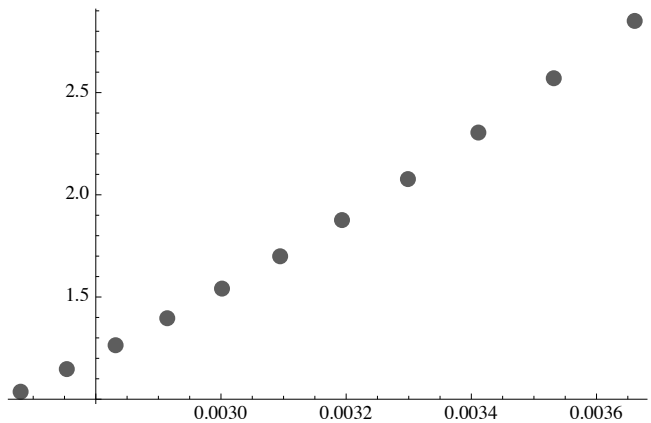
Out[67]//TableForm=

T^{-1}/K^{-1}	Log[η/Pa s]
0.00366099	2.85071
0.0035317	2.57032
0.00341122	2.30458
0.0032987	2.07694
0.00319336	1.87641
0.00309454	1.69928
0.00300165	1.54116
0.00291418	1.39624
0.00283166	1.26413
0.00275368	1.1474
0.00267989	1.03674

A continuación graficaremos los datos para ver si éstos tienen una relación lineal, y obtendremos la ecuación ajustada.

```
In[68]:= G1 = ListPlot[datos2, PlotStyle → {Red, PointSize[0.025]}]
```

Out[68]=

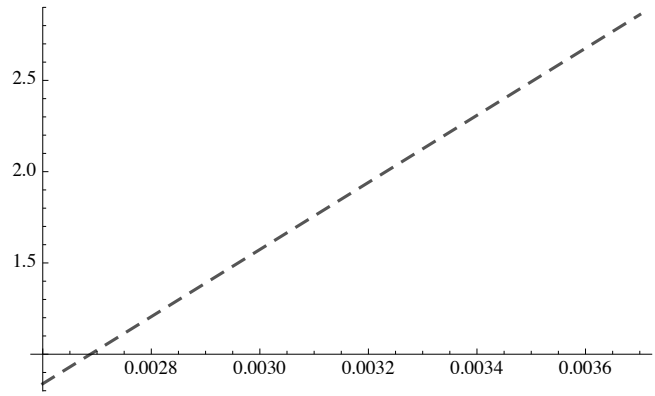



```
In[69]:= Fit[datos2, {1, T}, T]

Out[69]= -3.9459 + 1837.82 T
```

Ahora graficaremos la ecuación que nos ha dado el comando **Fit** para después compararla con los datos experimentales.

```
In[70]:= G2 = Plot[-3.94 + 1837.82 T, {T, 0.0026, 0.0037},
            PlotStyle → {Dashing[{0.02}], Thickness[0.005]}]
```

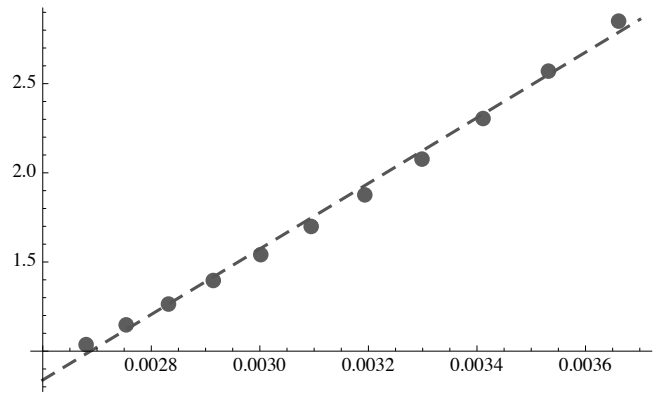
Out[70]=



Finalmente podemos obsevar la gráfica de los puntos experimentales junto con la predicción teórica. Esto lo hacemos con el comando **Show**.

```
In[71]:= Show[G1, G2]
```

Out[71]=



Como podemos ver al ajuste es muy bueno. Sin embargo, puede verse que los datos de los extremos pasan por arriba de la predicción teórica, mientras que los del medio lo hacen por debajo: esto significa que existe algún tipo de error sistemático entre los datos experimentales y la ecuación ajustada. Lo cual nos hace pensar que se podría lograr un mejor ajuste por medio de una modelo distinto del modelo usado (tipo Arrhenius).

Para escribir finalmente la ecuación teórica observamos que

$$\ln \eta = \ln \eta_0 - A / T.$$

De nuestro ajuste identificamos inmediatamente que $\ln \eta_0$ =-3.9459 y que *-A=1837.82*. De estas expresiones podemos obtener inmediatamente al valor númerico de A; sin embargo, para obtener el valor numérico de η_0, todavía es necesario sacar el inverso del logaritmo natural de la primera igualdad.

```
In[72]:= Solve[Log[η0] == -3.9459, η0]

Out[72]= {{η0 → 0.0193338}}
```

Con este resultado ahora podemos escribir la ecuación teórica:

$$\eta = 0.0193338\, e^{-1837.82/T}.$$

Finalmente, comentaremos que la manipulación del listado de datos experimentales puede hacerse de otra forma: si a la función de transformación le damos atributos de **Listable**. Para ello definamos una nueva función de dos variables.

```
In[73]:= f2[temp_, vis_] := {1 / (temp + 273.15), Log[vis]}

In[74]:= SetAttributes[f2, Listable]
```

Ahora tenemos que definir una lista para la temperatura y otra para la viscosidad.

```
In[75]:= temperatura = Range[0, 100, 10]

Out[75]= {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

In[76]:= viscosidad = {17.3, 13.07, 10.02,
          7.98, 6.53, 5.47, 4.67, 4.04, 3.54, 3.15, 2.82}

Out[76]= {17.3, 13.07, 10.02, 7.98, 6.53, 5.47, 4.67, 4.04, 3.54, 3.15, 2.82}

In[77]:= f2[temperatura, viscosidad]

Out[77]= {{0.00366099, 2.85071},
          {0.0035317, 2.57032}, {0.00341122, 2.30458},
          {0.0032987, 2.07694}, {0.00319336, 1.87641},
          {0.00309454, 1.69928}, {0.00300165, 1.54116},
          {0.00291418, 1.39624}, {0.00283166, 1.26413},
          {0.00275368, 1.1474}, {0.00267989, 1.03674}}
```

Aún mejor, sin tener que definir listados adicionales.

```
In[78]:= datos3 = datos /. {temp_, visc_} → {1 / (temp + 273.15), Log[visc]}

Out[78]= {{0.00366099, 2.85071}, {0.0035317, 2.57032},
          {0.00341122, 2.30458}, {0.0032987, 2.07694}, {0.00319336, 1.87641},
          {0.00309454, 1.69928}, {0.00300165, 1.54116}, {0.00291418, 1.39624},
          {0.00283166, 1.26413}, {0.00275368, 1.1474}, {0.00267989, 1.03674}}
```

Ambas opcones dan exactamente los mismos resultados obtenidos anteriormente.

```
In[79]:= f2[temperatura, viscosidad] == Map[ftv, datos] == datos3

Out[79]= True
```

Ejercicios

1. Construir una lista que contenga los enteros múltiplos de 7 no mayores de 100.

2. Encontrar el tamaño de la tabla anterior, el primero y el último miembros.

3. Sumar todos los miembros de la lista del ejercicio 1.

4. Sumarle a la lista del ejercicio 1 los múltiplos de 7 entre 100 al 200.

5. Crear una tabla de los 100 primeros números de la serie de Fibonacci y sumarlos.

6. Construir una lista del 1 al 20 en orden descendente.

7. Construir una lista de números de 0 a 2π en incrementos de $\pi/6$.

8. Construir una tabla que contenga en la primer columna los enteros del 1 al 10, y en las dos columnas consecutivas las raíces cuadrada y cúbica de estos números.

9. Construir una matriz de 3 por 3 cuyas entradas las formen números enteros consecutivos.

10. Construir una matriz de 5 por 5 que en la diagonal tenga los primeros 5 números primos y fuera de la diagonal solamente ceros.

Capítulo 5

Gráficas de funciones en *Mathematica*

Dos preguntas básicas antes de graficar

Antes de graficar cualesquier objeto, debemos hacernos las siguientes preguntas:

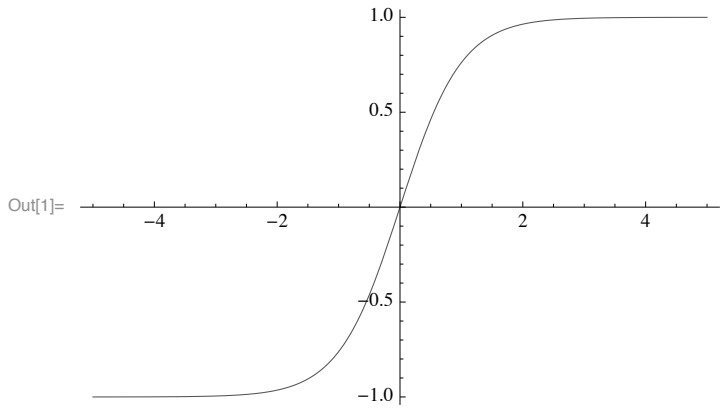
¿Vamos a graficar una función o datos de una lista?
¿En cuántas dimensiones vamos a graficar (2D or 3D)?

La siguiente es una tabla de las posibles respuestas a estas preguntas. Como *Mathematica* dispone de familias de comandos para cada combinación de respuestas, es importante discutir cada familia por separado:

	2D	3D
Función	**Plot**	**Plot3D**
Lista	**ListPlot**	**ListPlot3D**

Por ejemplo, si queremos una gráfica de la función $y = tanh(x)$, notamos que vamos a graficar una función en 2D. Leyendo la tabla anterior, vemos que el comando adecuado es **Plot**:

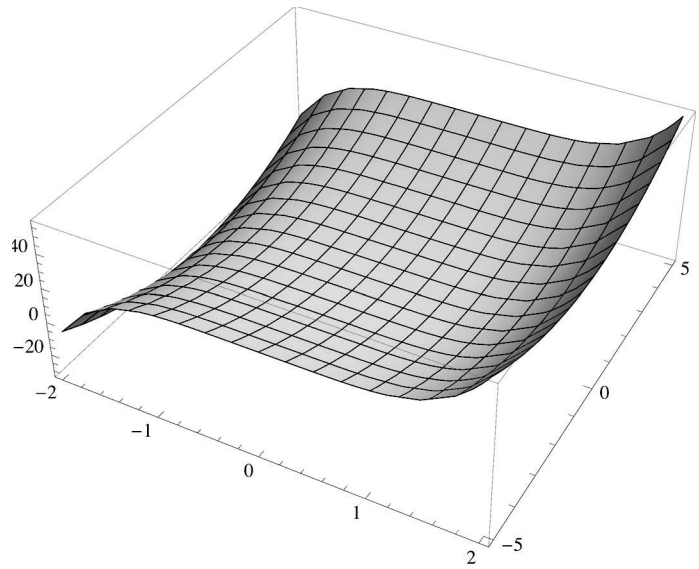
```
In[1]:= Plot[Tanh[x], {x, -5, 5}]
```



Para dar otro ejemplo, si queremos graficar la función $z = x^5 + y^2$, la gráfica será en 3D. Leyendo en la tabla encontrarmos que el comando apropiado es **Plot3D**:

```
In[2]:= Plot3D[x^5 + y^2, {x, -2, 2}, {y, -5, 5}]
```

Out[2]=



A lo largo de este capítulo y del próximo también, conoceremos otras formas para graficar nuestros datos.

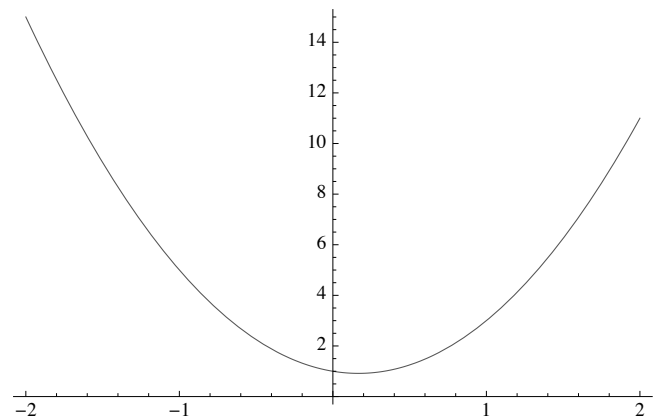
Graficando funciones con el comando `Plot`

La gráfica de una función permite el examen visual de su comportamiento en un cierto intervalo. Por ejemplo, si queremos conocer cuál es el comportamiento de la función $f(x) = 3x^2 - x + 1$ en el intevalo [-2,2], podemos hacerlo usando el comando **`Plot:`**.
La sintaxis básica del comando **`Plot`** es la siguiente:

`Plot[`*expr, {var, min, max}, opciones***`]`**	Grafica el objeto *expr* en el intervalo *{min, max}.*

```
In[3]:= Plot[3 x^2 - x + 1, {x, -2, 2}]
```

Out[3]=

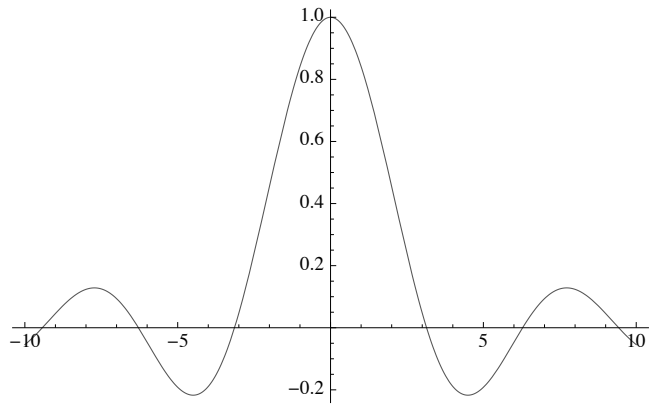


En esta gráfica podemos ver que $f(x)$ tiene un mínimo cerca de $x = 0.2$, pero no tiene raíces reales.

Como segundo ejemplo mostraremos la gráfica de la función $sen(x) / x$ en el intervalo $[-10, 10]$,

```
In[4]:= Plot[Sin[x] / x, {x, -10, 10}]
```

Out[4]=



En esta gráfica podemos ver que la función tiene varios puntos extremos, y que a pesar de que numerador y denominador son cero cuando $x = 0$, tiene un valor finito: $f(0) = 1$. Este hecho tal vez no sea evidente la primera vez que nos encontramos con esta función.

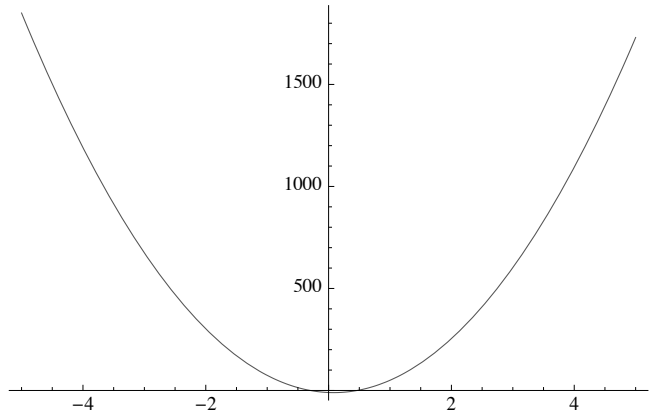
Esta forma básica del comando **`Plot`** nos permite ver la forma general de la función. Sin embargo, habrá ocasiones en que necesitamos tener mayor control sobre la forma en que se hace la gráfica como enfocarla en una región en especial, añadir etiquetas para los ejes, etcétera. Por esta razón los comandos gráficos tienen una gran cantidad de opciones (**`PlotRange`** y **`AxesLabel`** son las opciones que permiten resolver los dos ejemplos recién mencionados). Aquí sólo describiremos las que se usan más frecuentemente.

Para mostrar las posibilidades de *Mathematica* vamos a usar un ejemplo clásico: la búsqueda de raíces de ecuaciones no lineales. Comencemos con una ecuación cuadrática de la forma $f(x) = a\,x^2 + b\,x + c$. Si nos enfrentamos con una ecuación en particular, digamos, $72\,x^2 - 12\,x - 10 = 0$, podemos preguntarnos ¿dónde se encuentran las raíces?

Para ecuaciones cuadráticas existe una solución general analítica, $x = \frac{-b \pm \sqrt{b^2 - 4\,ac}}{2\,a}$. En ocasiones antes de obtener la solución analítica es probable que se quiera conocer en forma aproximada dónde se encuentran las soluciones. Para ello usemos el comando **`Plot`** proponiendo un intervalo arbitrario:

```
In[5]:= Plot[72 x^2 - 12 x - 10, {x, -5, 5}]
```

Out[5]=



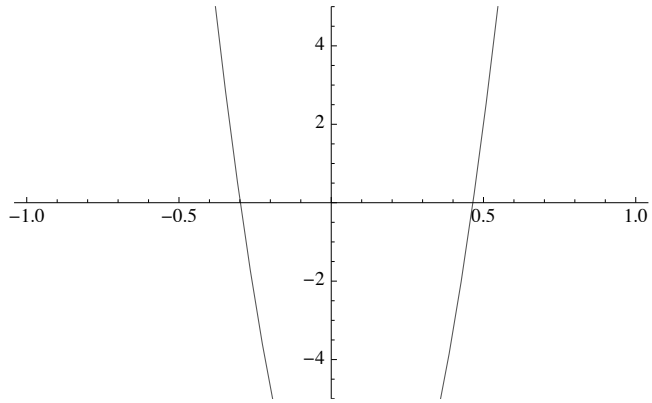
Podemos ver que la función toma valores muy grandes, y que si hay soluciones deben estar dentro del intervalo [-1,1]. Haciendo un segundo intento por encontrar las raices podemos usar la opción **PlotRange** para controlar la región visualizada.

PlotRange->{*intervalo horiz, intervalo vert***}**	Controla la región visualizada de la gráfica.

El primer intervalo es para el eje horizontal y el segundo para el vertical, y cada intervalo se puede especificar como **All**, **Automatic**, o una lista con los valores extremos del intervalo:

```
In[6]:= Plot[72 x^2 - 12 x - 10, {x, -1, 1},
          PlotRange → {Automatic, {-5, 5}}]
```

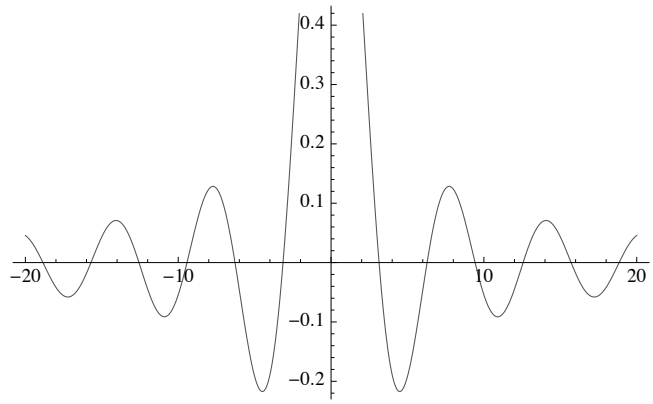
Out[6]=



Las formas **All** y **Automatic** son, sobre todo, útiles para acotar el eje vertical. **Plot** evalúa la función dada en una serie de puntos y la opción **Automatic** determina automáticamente la región "más interesante" a partir de los valores extremos de la función y de las regiones con mayor curvatura. Para mostrar todo el recorrido de la función en el contradominio se usa la forma **All.** Veamos la diferencia entre ambos comandos en las siguientes dos gráficas:

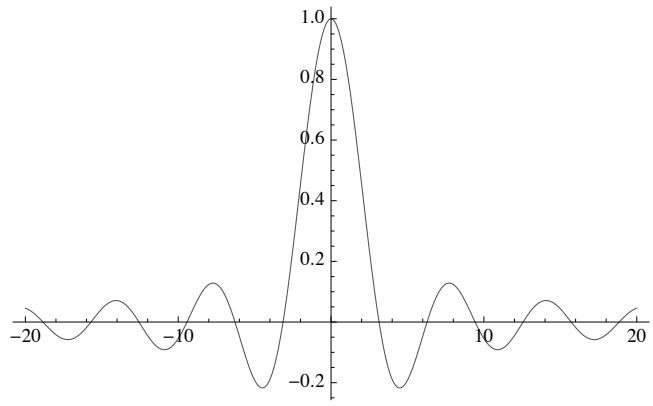
```
In[7]:= Plot[Sin[x] / x, {x, -20, 20}, PlotRange → Automatic]
```

Out[7]=




```
In[8]:= Plot[Sin[x] / x, {x, -20, 20}, PlotRange → All]
```

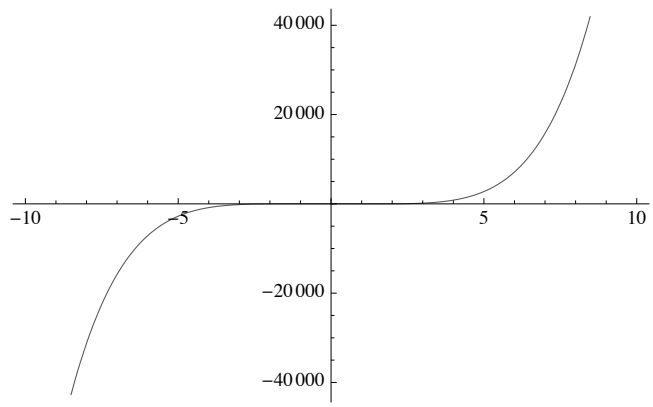
Out[8]=



Volviendo al tema de las raíces de ecuaciones, consideremos una ecuación quíntica, para la cual no hay solución en términos de funciones elementales. En el siguiente ejemplo trataremos de mostrar una estrategia que nos permita encontrar las raíces de la función $f(x) = x^5 - 3\,x^3 + x - 1 = 0$. Para orientarnos, comenzaremos graficando la función en un intervalo arbitrario:

```
In[9]:= Plot[ x^5 - 3 x^3 + x - 1, {x, -10, 10}]
```

Out[9]=

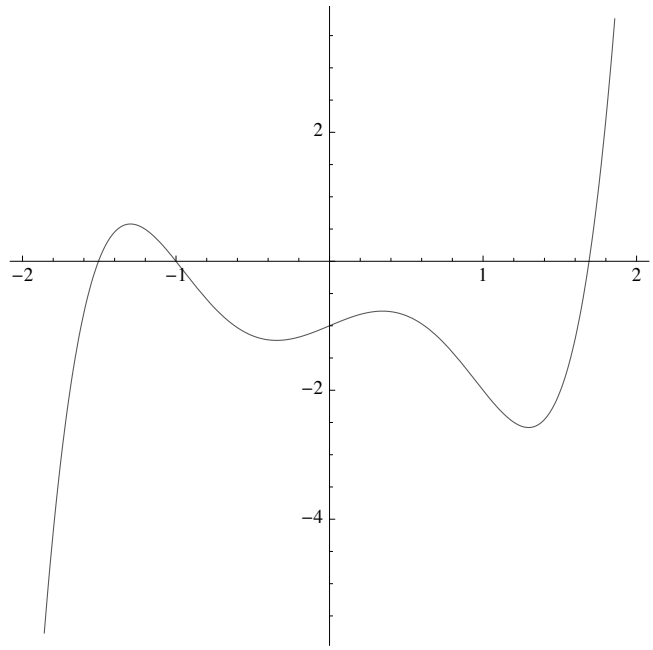


Aparentemente, podrían existir raíces reales en el intervalo $-2 \le x \le 2$. Para mostrar con mayor detalle la gráfica de la función en la región que nos interesa, vamos a modificar las proporciones de la gráfica usando la opción **AspectRatio**. Esta opción especifica el cociente del tamaño vertical de la gráfica al tamaño horizontal; por omisión *Mathematica* toma el valor **1/GoldenRatio**≈0.62. Ahora haremos ambas dimensiones de la gráfica iguales,

AspectRatio-> *n*	Especifica el cociente del tamaño vertical al tamaño horizontal de la gráfica.

```
In[10]:= Plot[ x^5 - 3 x^3 + x - 1, {x, -2, 2}, AspectRatio → 1]
```

Out[10]=



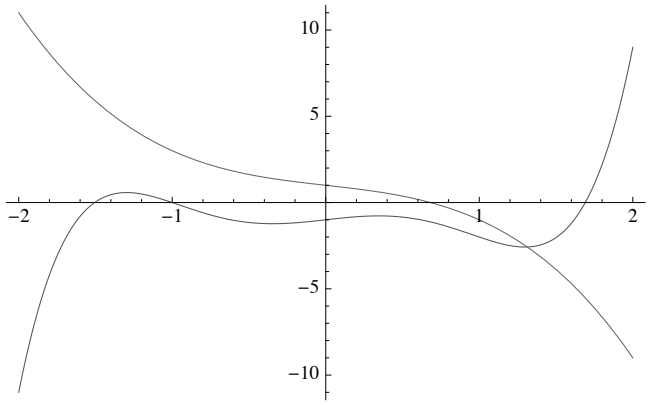
De la gráfica podemos ver que hay por lo menos tres raíces, una cerca de $x = -1.5$, otra cerca de $x = -1$ y la última cerca de $x = 1.7$. Si el usuario se interesa por obtener una aproximación numérica de las cinco raíces puede usar el comando **NSolve**,

```
In[11]:= NSolve[ x^5 - 3 x^3 + x - 1 == 0, x]
Out[11]= {{x → -1.50507}, {x → -1.}, {x → 0.407392 - 0.476565 i},
          {x → 0.407392 + 0.476565 i}, {x → 1.69028}}
```

Ahora supongamos que queremos mostrar dos funciones diferentes en una sola gráfica. El comando **Plot** puede graficar varias funciones a la vez, si ponemos como primer argumento las funciones dentro de una lista. En el siguiente ejemplo mostraremos cómo llevar a cabo esta tarea graficando las siguientes funciones, $f(x) = x^5 - 3x^3 + x - 1$ y $g(x) = -x^3 - x + 1$.

```
In[12]:= Plot[{x^5 - 3 x^3 + x - 1, -x^3 - x + 1}, {x, -2, 2}]
```

Out[12]=



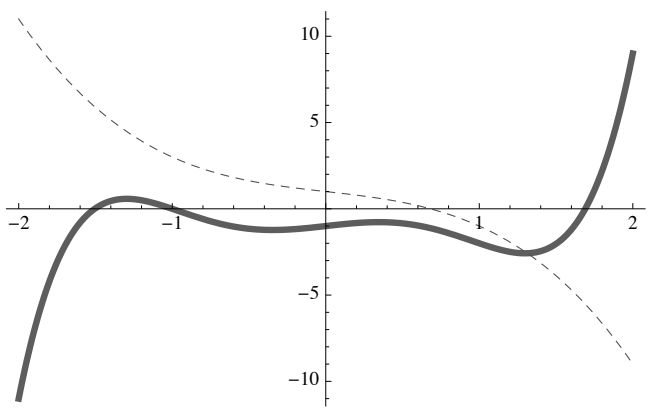
Como el lector podrá notar no es simple a primera vista distinguir entre las dos gráficas. ¿Cómo se puede identificar a cada función? Para hacerlo se usa la opción **PlotStyle**.

PlotStyle->{*opciones***}**	Estilo con que se dibujará la función.

Esta opción indica el estilo con que se dibujará la función, esto puede incluir el color, ancho de la línea e incluso si la línea es continua o a trazos. En nuestro ejemplo, haremos que la primera función se dibuje como una línea roja gruesa (**{Thickness[0.01], Red}**), mientras que la segunda en azul, a trazos (**{Dashing[{0.012, 0.012}], Blue}**):

```
In[13]:= Plot[{x^5 - 3 x^3 + x - 1, -x^3 - x + 1}, {x, -2, 2},
  PlotStyle → {
    {Thickness[0.01], Red},
    {Dashing[{0.012, 0.012}], Blue}
   }
]
```

Out[13]=



Cuando necesitamos identificar claramente a cada función en una figura, podemos especificar el estilo de línea para cada una de ellas. Esto se consigue con la opción **PlotStyle-> {***estilo1, estilo2,...***}**. Los estilos se aplican en el mismo orden en que aparecen las funciones a graficar en el comando **Plot**.

Los estilos de línea más frecuentes son colores o líneas punteadas. Para especificar un color, desde la versión 5.1 en adelante, usamos el nombre del color deseado: **Black**, **Blue**, **Brown**, **Cyan**, **Gray**, **Green**, **Magenta**, **Orange**, **Pink**, **Purple**, **Red**, **White** o **Yellow**.

```
PlotStyle -> {Red, Blue}
```

Para tener mayor control sobre el color, podemos usar el formato **RGBColor[***rojo, verde, azul***]**, donde *rojo*, *verde* y *azul* son números entre 0 y 1 que dan las intensidades de cada color primario que se han de combinar para producir el color deseado.

Por ejemplo, si queremos indicar una línea roja y otra verde, usamos la expresión

```
PlotStyle -> {RGBColor[1, 0, 0], RGBColor[0, 1, 0]}
```

Otra manera de indicar un color es el formato **Hue[***tono, saturacion, brillo***]**. La escala de *tono* es cíclica: empieza en 0 con el color rojo, 0.1 corresponde al naranja, 0.4 al verde, 0.6 al azul, 0.8 al violeta y vuelve al color rojo para 1. Otra manera de ver esto es que sólo la parte fraccionaria del *tono* determina el color con que se graficará finalmente la función. Las escalas de *saturación* y *brillo* van desde el mínimo para 0 hasta el máximo para 1.

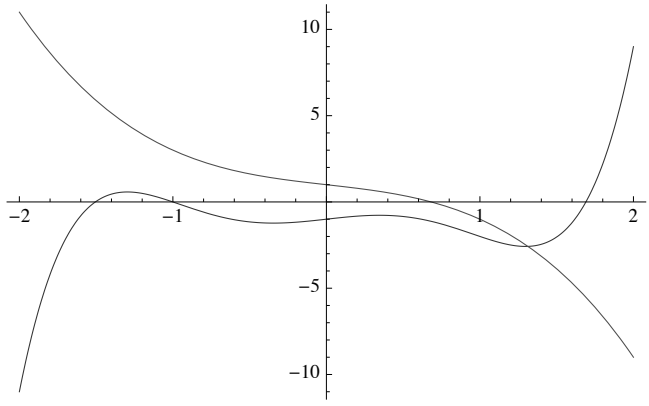
Con este formato, si queremos indicar una línea roja y otra azul, podemos usar

```
PlotStyle -> {Hue[0.0, 1, 1], RGBColor[0.6, 0.6, 1]}
```

Volviendo a nuestro ejemplo, podemos hacer que la ecuación quíntica graficada se vea como una línea muy roja pero no muy brillante, en tanto que la cúbica se vea como una brillante línea azul:

```
In[14]:= Plot[{x^5 - 3 x^3 + x - 1, -x^3 - x + 1}, {x, -2, 2},
           PlotStyle → {Hue[0.0, 1, 0.7], RGBColor[0, 0, 1]}]
```

Out[14]=



Por cierto, si se omiten los valores de saturación y brilo (**Hue[***tono***]**), *Mathematica* usa por omisión la máxima saturación y el máximo brillo.

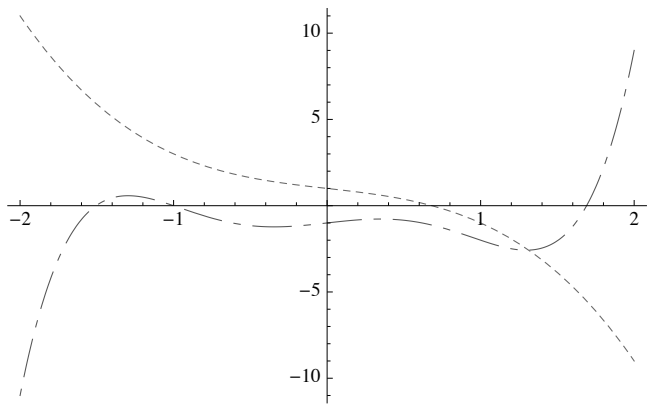
La segunda forma habitual de distinguir entre varias gráficas es usar líneas discontinuas. Para esto usamos **Dashing[{*segmento1*, *segmento2*, ...}]**. Los valores *segmento1*, *segmento2*, etcétera representan fracciones respecto del ancho total de la gráfica. Por lo tanto, si queremos (a) una línea punteada y (b) una línea con trazos cortos y largos, podemos obtenerlas usando

```
In[15]:= PlotStyle -> {
          Dashing[{0.01, 0.02, 0.06, 0.02}],
          Dashing[{0.01, 0.01}]
         }
Out[15]= PlotStyle → {Dashing[{0.01, 0.02, 0.06, 0.02}], Dashing[{0.01, 0.01}]}
```

Una vez más, recurriendo a nuestro ejemplo, la ecuación quíntica se verá ahora como una línea a trazos cortos y largos, mientras que la cúbica se verá como una línea a trazos cortos.

```
In[16]:= Plot[{x^5 - 3 x^3 + x - 1, -x^3 - x + 1}, {x, -2, 2},
          PlotStyle →
           {Dashing[{0.01, 0.02, 0.06, 0.02}], Dashing[{0.01, 0.01}]}
         ]
```

Out[16]=



Es importante señalar que se puede combinar el estilo de color y el de tipo de línea. La forma para hacer esto es poner ambos estilos dentro de una lista. Por ejemplo,

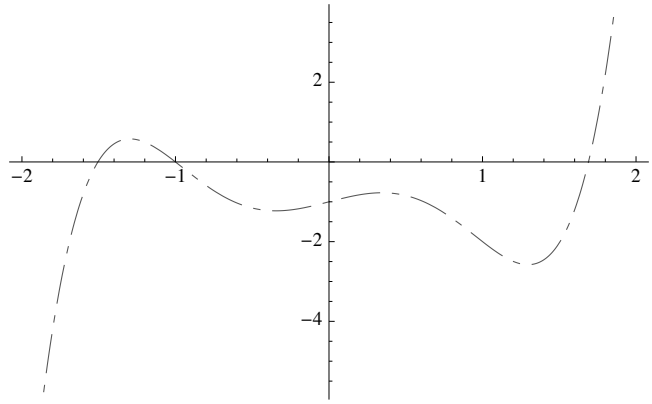
```
In[17]:= PlotStyle->
         {
          { Hue[0], Dashing[ { 0.01, 0.01}] },
          Hue[0.4]
         }
Out[17]= PlotStyle → {{Hue[0], Dashing[{0.01, 0.01}]}, Hue[0.4]}
```

Otra forma de poner varias gráficas dentro de una misma figura es generar cada una por separado y luego superponerlas usando el comando **Show**:

Primero, generamos una gráfica para la primera función (es decir, la quíntica). Para usarla después más fácilmente, asignamos el resultado a la variable *antes*.

```
In[18]:= antes = Plot[x^5 - 3 x^3 + x - 1, {x, -2, 2},
           PlotStyle → Dashing[{0.01, 0.02, 0.06, 0.02}]
         ]
```

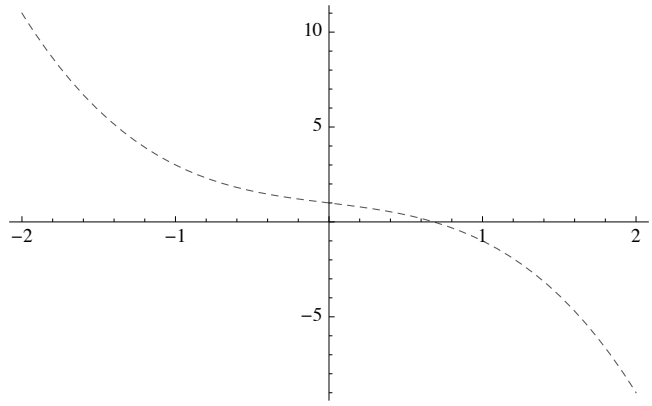
Out[18]=



Luego generamos la segunda gráfica, es decir, la cúbica:

```
In[19]:= despues = Plot[ -x^3 - x + 1, {x, -2, 2},
           PlotStyle → Dashing[{0.01, 0.01}]
         ]
```

Out[19]=

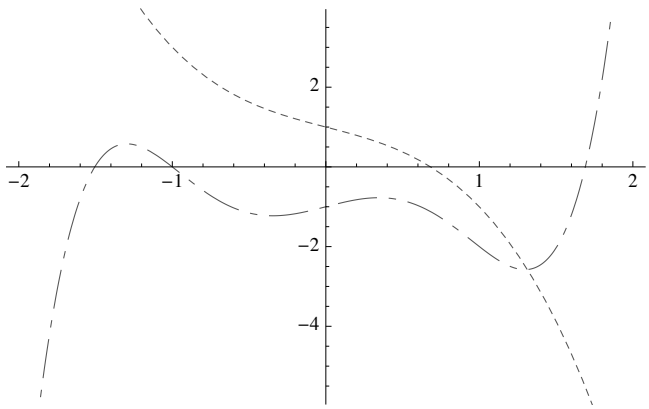


Finalmente, combinamos las gráficas mediante el comando **Show**:

Show[{*figura1, figura2,..***},***opciones***]**	Muestra el listado de figuras con las opciones indicadas.

```
In[20]:= Show[antes, despues]
```

Out[20]=



Este método permite combinar cualquier número de gráficas. En cada una de ellas se deberá especificar por medio de la opción **PlotStyle** un estilo que permita distinguir a cada gráfica de las demás.

Existen una gran variedad de comandos que permiten realizar tipos especializados de gráficas, por ejemplo, **PolarPlot**, **LogPlot**, **LogLinearPlot**, **ParametricPlot**, etcétera. Todas ellas comparten muchas de las opciones y formas de especificar los atributos de las etiquetas, ejes, etcétera, por lo que en vez de discutir cada una de ellas aquí sugerimos al lector que consulte la documentación en línea. En cambio, a continuación presentamos en detalle cómo controlar muchos atributos de las gráficas que en la práctica es necesario modificar si queremos usar directamente una gráfica de *Mathematica* en una publicación profesional.

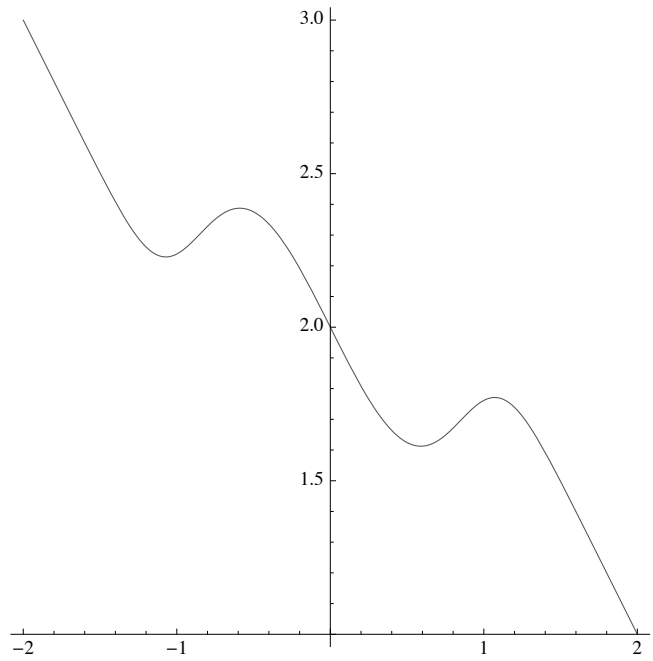
Preparando una gráfica para publicación

Normalmente, las opciones por omisión de *Mathematica* nos sirven para darnos una buena idea de cómo se comporta una función, lo cual es muy útil cuando estamos en medio de un cálculo. Sin emabargo, una vez que necesitamos una gráfica de muy buena calidad, como las que se usan en publicaciones, normalmente tendremos que usar más opciones para: rotular los ejes, ajustar el tamaño de la gráfica, ajustar el tipo de letra, entre las operaciones más comunes. En esta sección usaremos el comando **Show** para mostrar cómo obtener una gráfica con calidad de publicación.

Supongamos que necesitamos una gráfica de una función para ilustrar un artículo o un libro, digamos que se trata de la función $y = tanh(x^3) - x + 2$, en el intervalo $-2 \leq x \leq 2$

```
In[21]:= figura1 = Plot[Tanh[x^3] - x + 2, {x, -2, 2}, AspectRatio → 1]
```

Out[21]=

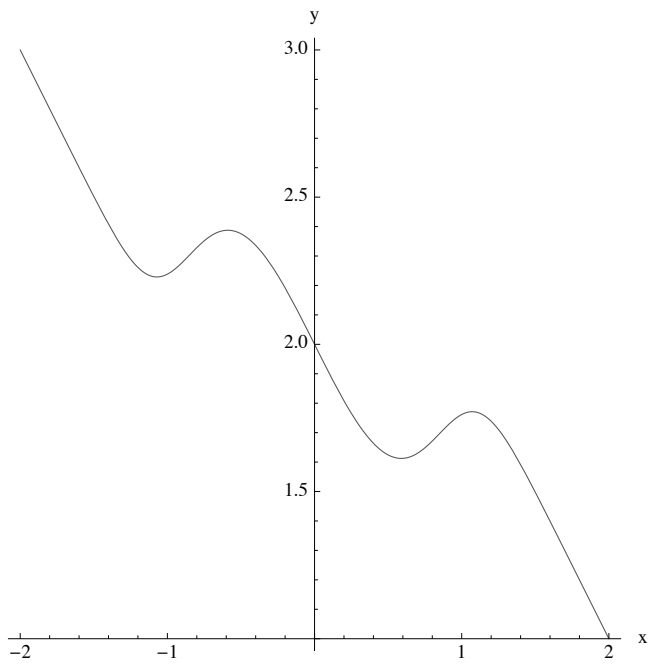


Lo primero que queremos mostrar es como identificar los ejes horizontal y vertical. Como primer argumento usaremos la variable *figura1*, en el comado **Show** para que se muestre la figura que hemos realizado en el ejemplo anterior. El segundo argumento que utilizaremos será la opción **AxesLabel→ {*ejeX*, *ejeY*}** para etiquetar los ejes

AxesLabel->{*etiqueta eje x, etiqueta eje y***}** Rotula los ejes con el nombre deseado.

In[22]:=
```
Show[figura1, AxesLabel → {"x", "y"}]
```

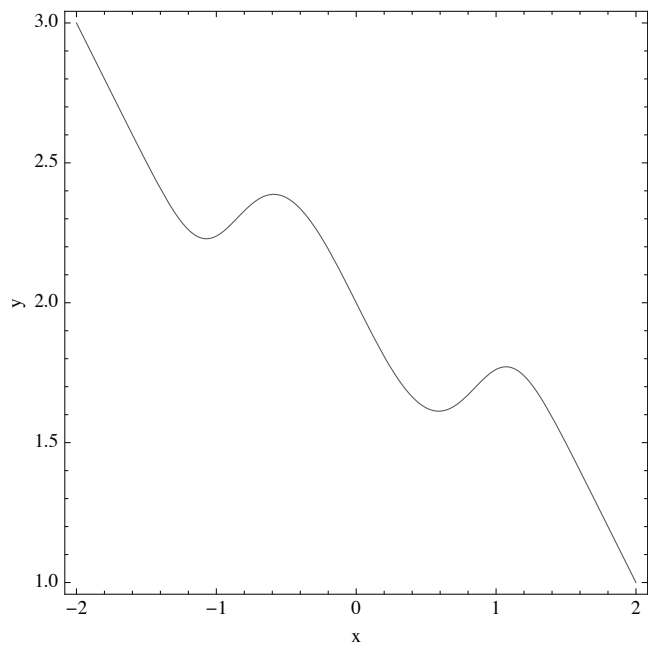
Out[22]=



Una alternativa a rotular los ejes consiste en agregar un marco a la gráfica (**Frame -> True**) y rotular los lados de ese marco (**FrameLabel-> {"x","y"}**). Como los ejes y el marco son independientes y se combinan, a veces es útil eliminar los ejes si estamos usando el marco:

In[23]:=
```
figura2 =
 Show[figura1, Frame → True, FrameLabel → {"x", "y"}, Axes → None]
```

Out[23]=



Por el contrario, para indicar que todos los ejes deben mostrarse, se utiliza el comando **Axes**

Axes->{*opción eje X, opción eje Y***}**	Se indica si los ejes correspondientes deben mostrarse (**True**) o no (**False**).

Entre las combinaciones posibles, podemos pedir que el eje horizontal se muestre, pero no el vertical:

```
In[24]:= Axes->{ True, False }
Out[24]= Axes → {True, False}
```

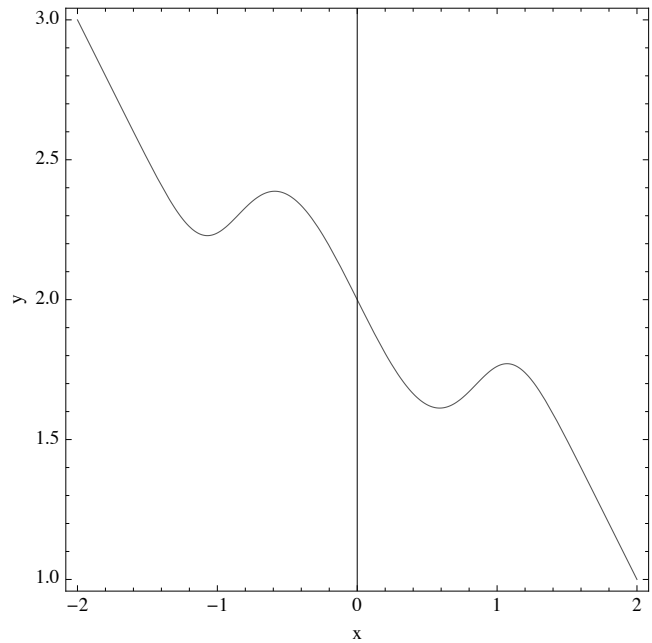
o mostrar el eje vertical, pero no el horizontal:

```
In[25]:= Axes->{ False, True }
Out[25]= Axes → {False, True}
```

Veamos el resultado de esta última forma para nuestra gráfica número 2:

```
In[26]:= Show[figura2, Axes → {False, True}]
```

Out[26]=



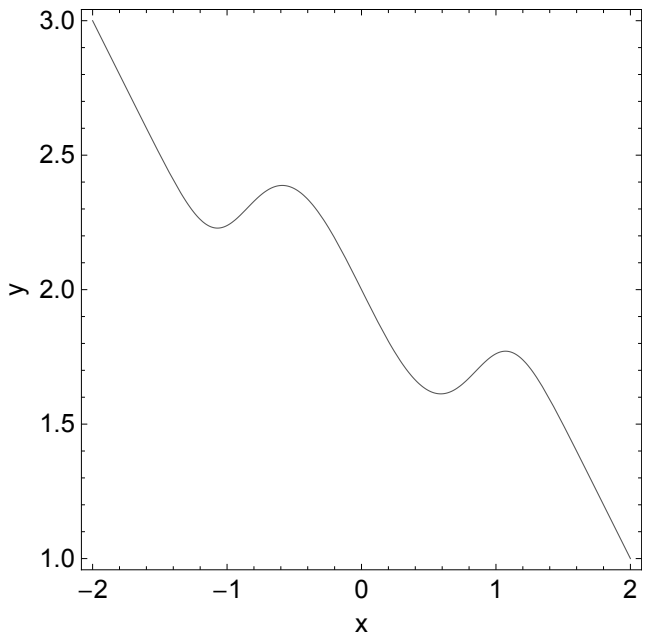
Generalmente el tipo de letra que *Mathematica* usa por omisión (Times) no es apropiado para publicación, debido a que las letras son demasiado delgadas para leerlas fácilmente. Una solución común consiste en usar tipos de letras sin patines (*sans-serif*), por ejemplo las familias Helvetica o Arial. El comando **Plot** permite modificar globalmente el tipo de letra usado para las coordenadas y los rótulos de los ejes usando la opción **BaseStyle**. La forma básica especifica, en el lado derecho de la opción, una lista que indica el tipo de letra y el tamaño en puntos:

BaseStyle->{FontFamily → "*Tipo de letra*"**, FontSize →** *n***}**	Cambia al tipo de letra deseado con tamaño *n*.

En nuestro ejemplo usaremos letra de tipo Arial con un tamaño de 12 puntos:

```
In[27]:= figura3 =
          Show[figura2, BaseStyle → {FontFamily → "Arial", FontSize → 12}]
```

Out[27]=

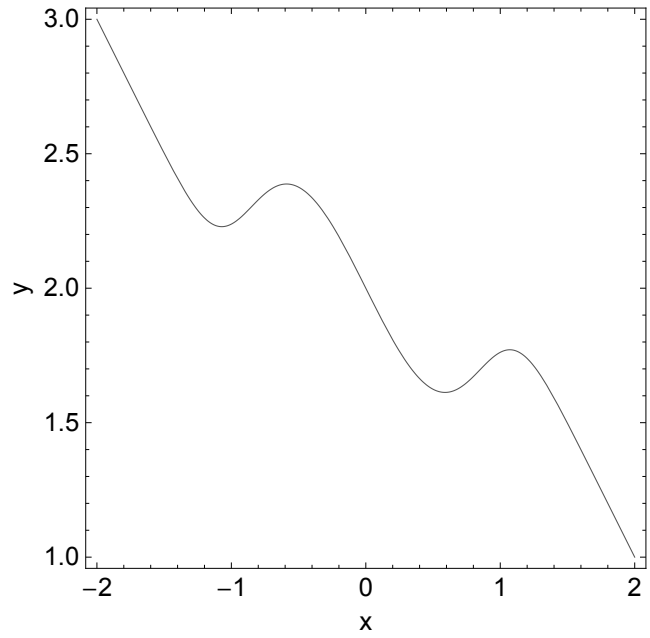


Por último, en las gráficas para publicación es conveniente que las curvas que estamos mostrando no sean muy delgadas. Al preparar la figura 1, pudimos haber incluido en la opción **PlotStyle** un comando **Thickness** para hacer la curva más gruesa. Aún ahora no es demasiado tarde para hacerlo, puesto que *Mathematica* posee dos opciones llamadas **Prolog** y **Epilog**. Las instrucciones que se incluyen en la opción **Prolog** se llevan a cabo inmediatamente después de dibujar los ejes, cajas y marcos pero antes de dibujar las funciones. Así, añadiendo un comando **Thickness** dentro de **Prolog** conseguimos hacer que la curva se dibuje más gruesa (el argumento de **Thickness** es el ancho de la línea expresado como porcentaje de la anchura total de la gráfica):

Prolog->{*opciones***}**	Instrucciones que se llevan a cabo inmediatamente antes de dibujar las funciones.

```
In[28]:= figura4 = Show[figura3, Prolog → Thickness[0.01]]
```

Out[28]=

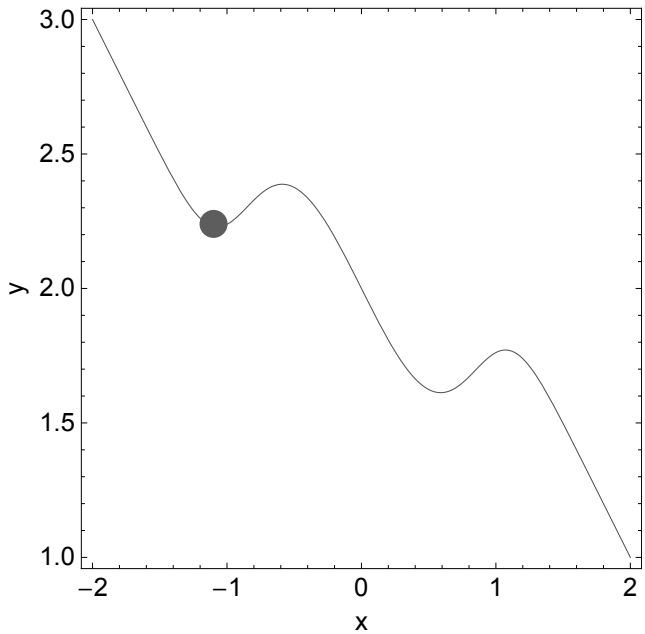


De manera parecida, las instrucciones gráficas que se incluyen en la opción **`Epilog`** se llevan a cabo inmediatamente después de dibujar la parte principal de la gráfica, es decir, después de terminar de dibujar todas las funciones. En nuestro ejemplo, añadiremos un punto rojo cerca del primer mínimo de la función

`Epilog->{`*opciones***`}`**	Instrucciones que se llevan a cabo inmediatamente después de dibujar la parte principal de la gráfica.

In[29]:=
```
Show[figura4,
 Epilog→ {PointSize[0.05], Red, Point[{-1.1, 2.24}]}
]
```

Out[29]=

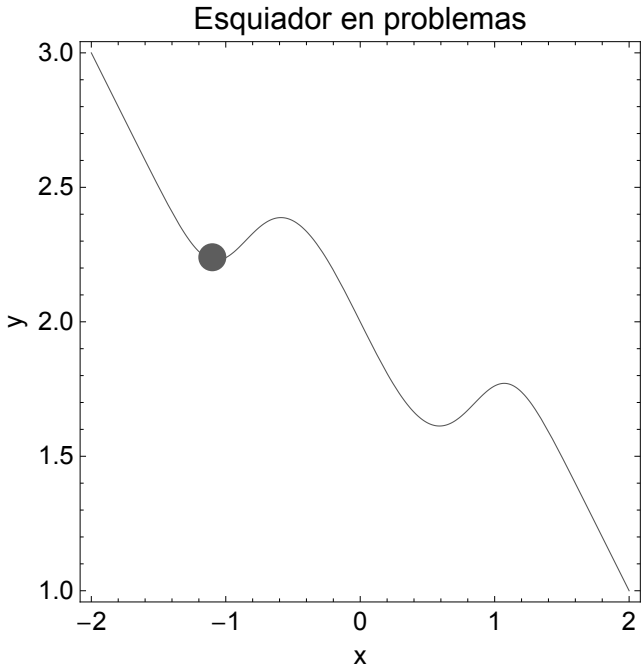


Algunas veces necesitaremos añadir una etiqueta global para la gráfica, para esas ocasiones usamos la opción **`PlotLabel`**,

`PlotLabel ->` *"Título"*	Etiqueta para la gráfica en conjunto.

```
In[30]:= Show[figura4,
          Epilog → {PointSize[0.05], Red, Point[{-1.1, 2.24}]},
          PlotLabel → "Esquiador en problemas"]
```

Out[30]=



Aunque es posible cortar y pegar la figura usando únicamente la interfaz gráfica y el sistema operativo, generalmente, para aprovechar la figura con la máxima calidad es necesario exportarla a un archivo en un formato apropiado. En la sección siguiente nos ocupamos de cómo llevar ésto a cabo.

Exportando gráficas con `Export` y *Save Cell As*

Para conseguir que las gráficas de *Mathematica* se puedan reproducir a la máxima resolución posible, debemos estar al tanto de los formatos comunes entre *Mathematica* y los que maneja la publicación a la que enviaremos nuestro manuscrito.
También es posible que seamos nosotros mismos los que vamos a producir e imprimir la publicación (por ejemplo para un reporte, para un cartél, etcétera). En ese caso, lo que necesitamos saber es cuáles son los formatos comunes entre *Mathematica* y el programa a donde queremos insertar las gráficas.

Si el programa de destino es capaz de usar la función de "copiar y pegar", lo más sencillo es señalar la gráfica en la ventana de *Mathematica* y luego copiarla usando el menú ***Edit>Copy***. (También podemos usar la combinación de teclas ***Ctrl+C*** para copiar la figura.)

A continuación, en el programa de destino usaremos el menú "Editar" y "Pegar", o la combinación de teclas equivalente (típicamente ***Ctrl+V***). Para mayor control, haciendo click sobre la gráfica con el botón derecho del mouse aparecerá un menú contextual, si se selecciona la opción ***Copy As*** ésta nos permite definir el formato en que se copiará la gráfica.

En un sistema Windows los formatos más pertinentes para gráficas son *Bitmap* y *Metafile*. El primero copia la imagen como mapa de bits, ésto usa mucho espacio y sólo proporciona una resolución fija. La opción *Metafile* es generalmente mejor, puesto que da una descripción de los elementos básicos de la gráfica (líneas, texto, puntos, etc) y por lo tanto tiene la ventaja de funcionar bien en distintas resoluciones.

En muchos programas, tales como *Word* y *PowerPoint*, las gráficas pueden ser ajustadas después de ser insertadas: usando el botón derecho del mouse uno accede al menú contextual para la figura insertada. Seleccionando "editar las propiedades de la figura", es posible ajustar el tamaño, el centrado e incluso ocultar secciones de la imagen.

En otras ocasiones es preferible exportar primero la figura de *Mathematica* a un archivo con un formato aceptado por el programa de destino. Por ejemplo, si queremos exportar una gráfica a *Word*, es posible que queramos usar un formato *Windows MetaFile* (WMF), *Enhanced MetaFile* (EMF), *Joint Photographic Experts Group* (JPEG), *Bitmap* (BMP) o *Encapsulated PostScript* (EPS).

Hay por lo menos dos métodos sencillos para exportar una gráfica desde *Mathematica* a un archivo. El primer método usa tan solo la interfáz gráfica (*FrontEnd*) y por lo tanto no requiere que esté activo el Kernel. En el segundo caso, el Kernel evalúa una expresión del tipo **`Export`** y tiene la ventaja de facilitar la automatización de la tarea.

Método 1. Se selecciona la gráfica que se quiere exportar, luego se usa el menú ***Edit*** (o ***File***, si se trata de la versión 6) y su opción ***Save Selection As*** para escoger entre los formatos siguientes: EPS, BMP, EMF y WMF. Después de indicar en la caja de diálogo a dónde se guardará el archivo exportado, se puede ir al programa de destino a importar la gráfica siguiendo las instrucciones en el respectivo manual.

Método 2. Se requiere que el Kernel esté activo y que la sesión todavía tenga en memoria la gráfica que se quiere exportar. Esto significa que no basta con abrir un archivo donde hayamos guardado la gráfica de una sesión anterior, sino que debemos producir la gráfica antes de poder exportarla. Una vez que la gráfica esté en memoria, usamos el comando

`Export[` *"archivo.ext", expr, "formato"* **`]`**	Exporta la expresión *expr* en el formato deseado con nombre *archivo* y extensión *ext*.

En el comando anterior, el primer argumento es una cadena de caracteres encerrada entre comillas, con el nombre del archivo en donde se exportará la grafica. El segundo elemento es una expresión que produce una gráfica, ya sea cualquier comando gráfico válido (**`Plot`**, **`ListPlot`**, **`ListPlot3D`**, etcétera) o una variable donde hayamos almacenado previamente nuestra figura. Por último, el tercer argumento es otra cadena de caracteres (delimitada por comillas) que especifique alguno de los formatos manejados por el comando **`Export`**:

Resolución Variable

Extensión	Tipo
APS	*Mathematica* abbreviated PostScript
EPS	Encapsulated PostScript
PDF	Adobe Acrobat Portable Document Format
PICT	Macintosh PICT Format
SVG	Scalable Vector Graphics
WMF	Windows MetaFile

Resolución Fija

Extensión	Tipo
AVI	Audio Video Interleave Format
BMP	Bitmap
DICOM	DICOM Medical Imaging Format
GIF	Graphics Interchange Format
JPEG	Joint Photographic Expert Group Format
PNG	PNG Format
WMF	Windows MetaFile
TIFF	TIFF Format

En muchas ocasiones estos dos métodos permitirán insertar la figura deseada en otros programas, sin embargo en diversas ocasiones (por ejemplo, cuando se está colaborando en un libro o colección y se requiere que todas las figuras sean producidas por un mismo software) será necesario exportar los datos procesados por *Mathematica* y regenerar las figuras en un graficador independiente. Para esas ocasiones es útil tambien la forma del comando **Export**["*archivo.dat*", *datos*, "*Table*"], donde el segundo argumento es una tabla de datos con la que se pueda regenerar la gráfica. Al especificar como formato de salida "*Table*", el contenido de "*archivo.dat*" será una tabla delimitada por tabuladores; este formato es leído por la mayoría de programas graficadores y también hojas de cálculo como *Excel* y *OpenOffice.org Calc*.

Haciendo animaciones

Frecuentemente queremos observar cómo cambia una serie de figuras a medida que se modifica un parámetro. El parámetro más común sería el tiempo, pero también es posible pensar en otros parámetros para diversos sistemas, por ejemplo, para un reactor químico podría ser la temperatura y para un circuito eléctrico podría ser la resistencia de uno de sus componentes.

En esos caso puede ser útil producir animaciones donde el parámetro que se varía se convierte en el eje del tiempo de la animación. Para producir animaciones con *Mathematica* podemos seguir dos rutas, la primera es a través de la interfaz gráfica y la segunda a través del Kernel.

En la interfaz gráfica, para animar una serie de gráficas contiguas basta con seleccionarlas y seleccionar en el menú *Cell* la opción *Animate Selected Graphics*. La interfaz gráfica responde generando una animación de las imágenes seleccionadas y presentando una pequeña barra de herramientas en la parte baja de la ventana, con las que se puede adelantar, regresar, detener la animación y cambiar su resolución.

Para preparar animaciones usando el Kernel se requiere, como en el caso del comando **`Export`**, que las gráficas que se quieren combinar estén disponibles en la memoria de la sesión.

`ShowAnimation[` *{grafica1, grafica2,...}* **`]`**	Muestra las animaciones especificadas entre paréntesis.

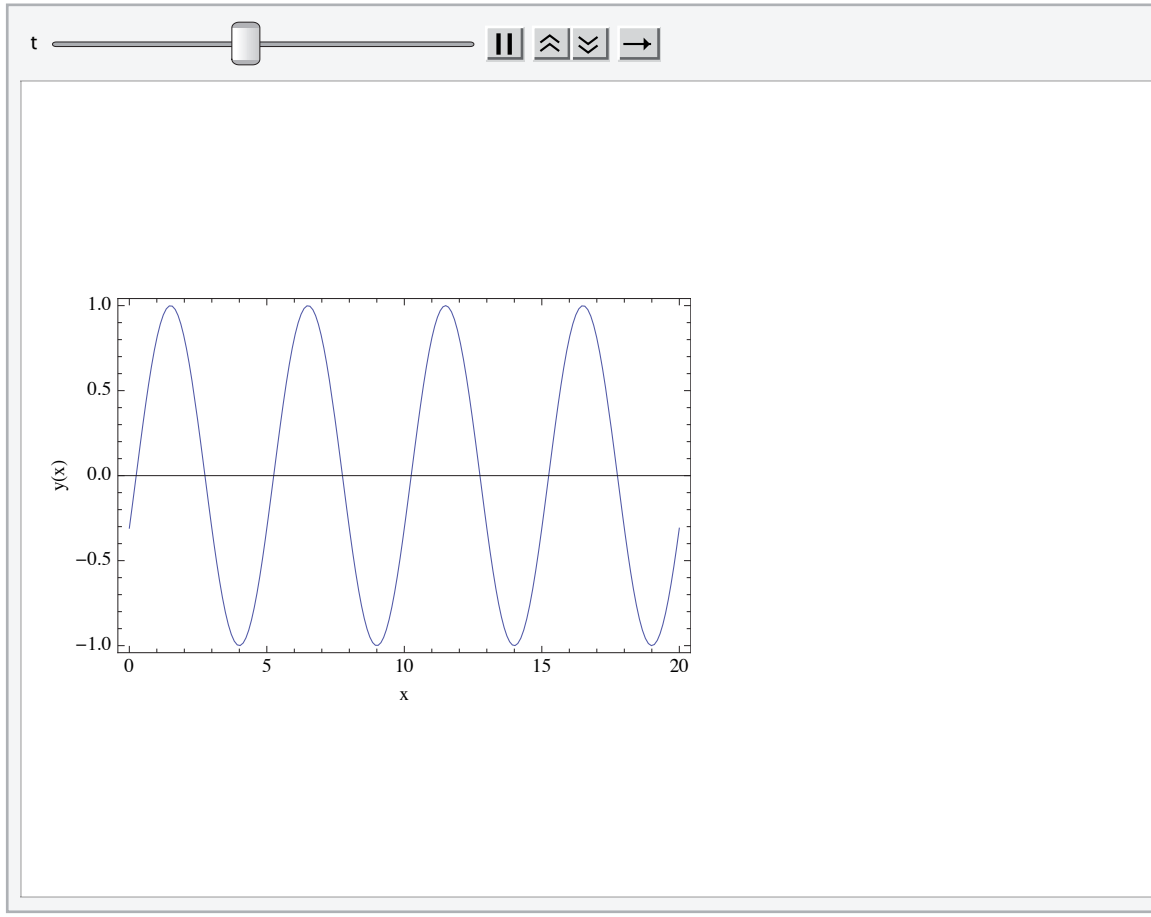
Para graficar funciones, sin embargo, existen los comandos **`MoviePlot`**, **`MoviePlot3D`** y otros similares que evitan la necesidad de generar primero cada cuadro de la animación. A partir de la versión 6 de *Mathematica* también existen los comandos **`Animate`** y **`Manipulate`** que producen animaciones con controles gráficos para detener, avanzar o repetir la animación (la documentación en línea de *Mathematica* es útil para aprender más sobre estos "objetos dinámicos"). La sintaxis de ambos comandos es parecida a la de **`Table`**:

`Animate[` *expr, {x, xmin, xmax, Δx}* **`]`**	Genera una animación de la expresión dada como función de *x*.

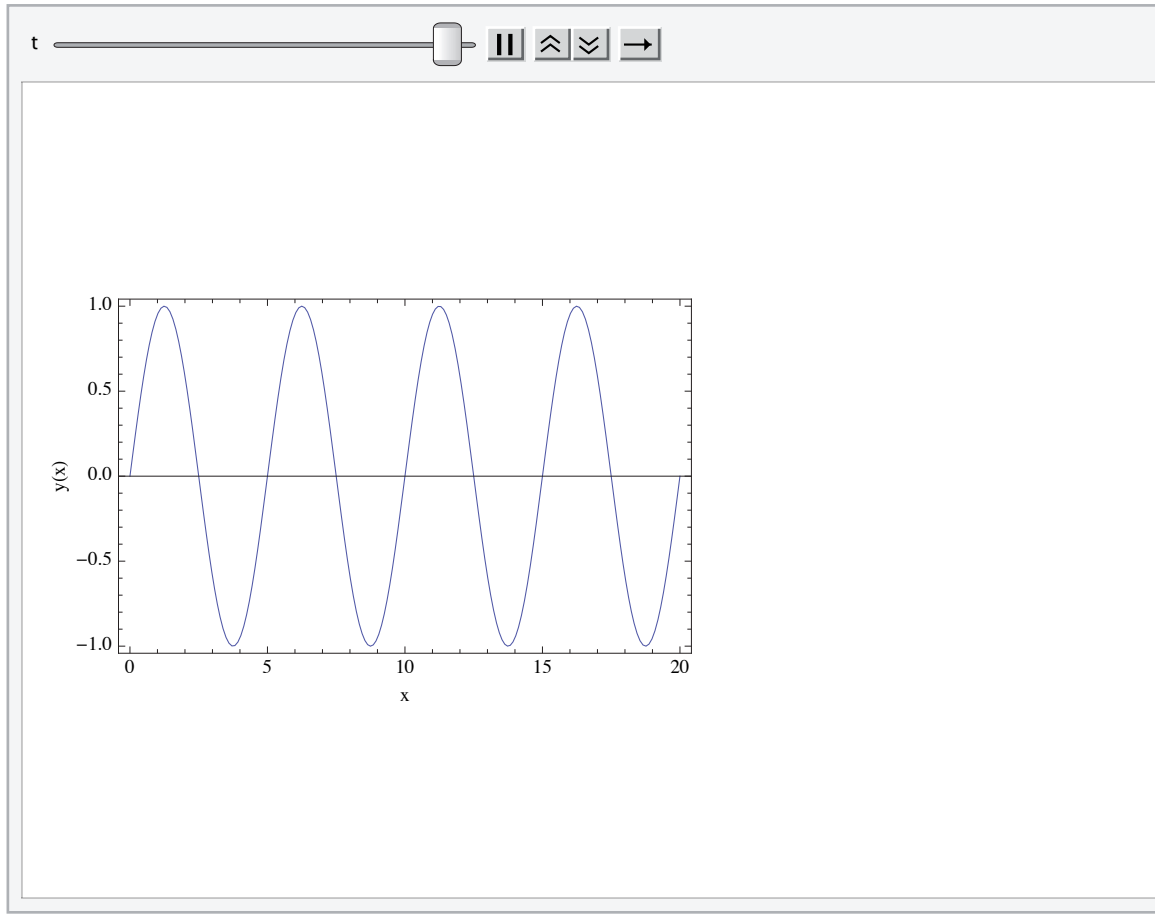
In[31]:=

In[32]:=
```
Animate[ Plot[Sin[2 π (x/5 - t/10)], {x, 0, 20},
  Frame → True, FrameLabel → {"x", "y(x)"}],
 {t, 0, 10, 0.5}]
```

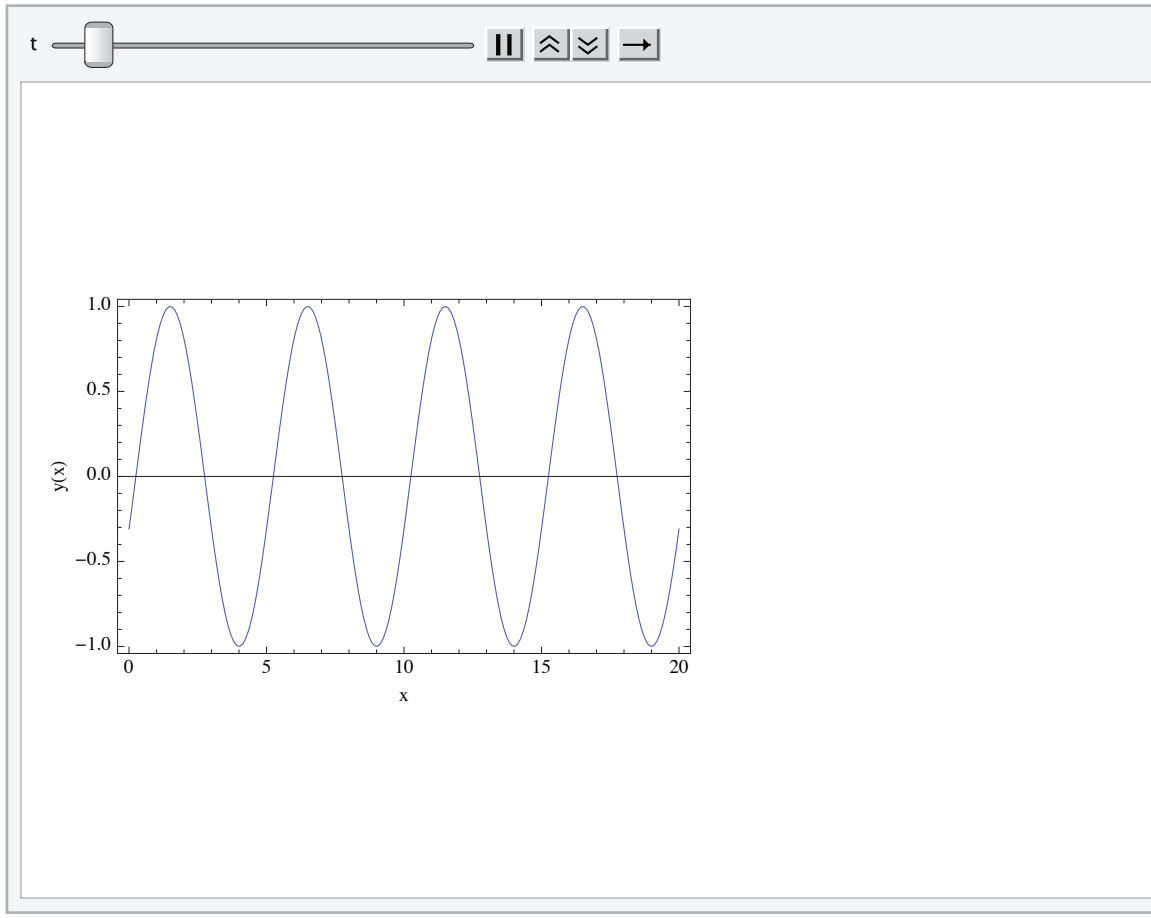
Out[32]=



In[33]:=



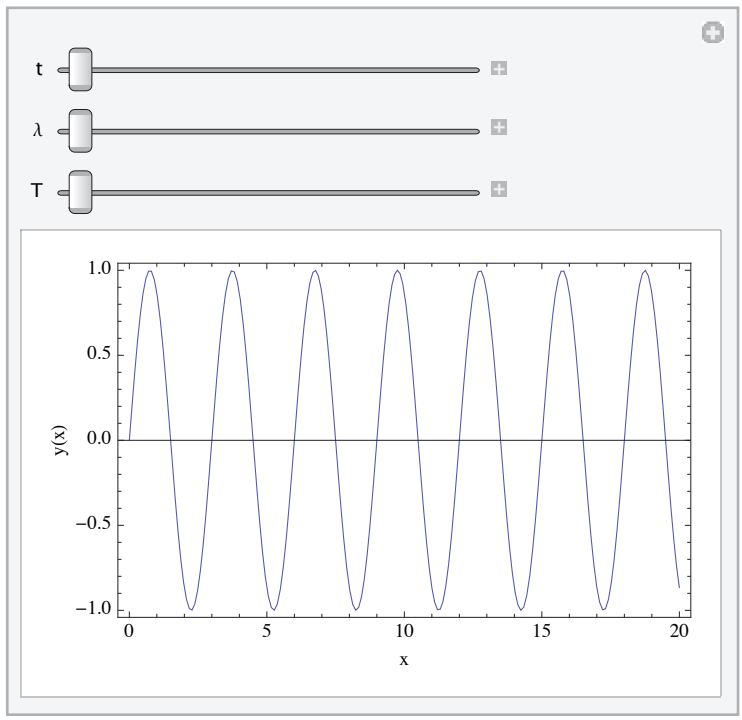
Out[33]=



`Manipulate[`*expr, {x, xmin, xmax, Δx}, {parametro1, min, max}, ...***`]`**
Genera una animación de la expresión dada, con controles para modificar los parámetros especificados.

In[34]:=
```
Manipulate[ Plot[Sin[2 π (x/λ - t/T)], {x, 0, 20},
  Frame → True, FrameLabel → {"x", "y(x)"}],
 {t, 0, 10, 0.5}, {λ, 3, 7}, {T, 10, 20}]
```

Out[34]=



Junto al control de cada variable hay un símbolo + que al presionarse revela controles detallados para modificar el valor del parámetro y animar la gráfica. Es importante indicar aquí que los documentos de *Mathematica* que contienen "objetos dinámicos" generarán una advertencia correspondiente cada vez que se abren, para alertar al usuario de que necesitará una versión reciente del programa para poder usar y modificar estos objetos. A propósito, el programa *Mathematica Player* se distribuye gratuitamente en la internet y permite visualizar y usar documentos de *Mathematica* que incluyan objetos dinámicos, pero no permite modificarlos.

Ejercicios

1.-Graficar la función $y = x^2$ en el intervalo $-5 \le x \le 5$.

2. Graficar las funciones $y = x^2$ y $y = 2x + 10$ en el intervalo $-5 \le x \le 5$.

3. Grafica las funciones $y = x^2$, $y = x^3$ y $y = x^4$ en el intervalo $0 \le x \le 1$.

4. Graficar las funciones $y = x$, $y = -x$, y $y = x \operatorname{sen} x$ en el intervalo $-6\pi \le x \le 6\pi$.

5. Graficar en el intervalo $0 \le x \le 4$, la función $f(x) = \begin{cases} x^2 & \text{si } x \le 2 \\ 8 - 2x & \text{si } x > 2 \end{cases}$.

6. Graficar en el intervalo $-\pi \le x \le \pi$, las funciones $\operatorname{sen} x$, $\operatorname{sen} 2x$, y $\operatorname{sen} 3x$. Para cada línea utilizar un color y estilo diferente.

7. Graficar en el intervalo $-\pi \le x \le \pi$, las funciones x^2, $2x^2$, y $3x^2$. Para cada linea utilizar un color y estilo diferente.

8. Agregar leyendas a la gráfica anterior.

9. Agregar nombres en los ejes y título a la gráfica anterior.

10. Poner en una caja con cuadrícula la gráfica de las funciones $\operatorname{sen} x$, $\operatorname{sen} 2x$, y $\operatorname{sen} 3x$ en el intervalo $-\pi \le x \le \pi$.

11. Construir la animación del movimiento de una partícula en tiro parabólico usando el comando **`Manipulate`**, cambiando los parámetros de velocidad inicial y ángulo de salida.

Capítulo 6

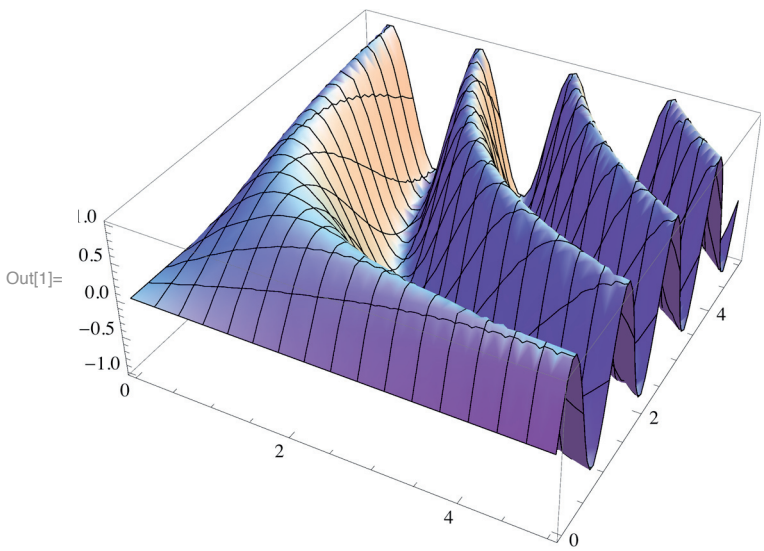
Graficando funciones de dos variables

Una vez que se adquiere práctica con el comando **Plot**, resulta sencillo transladar lo que sabemos al comando **Plot3D**. A diferencia del primero, **Plot3D** grafica funciones de dos variables y genera gráficas de superficies tridimensionales. Los datos corresponderán a los valores de una función $z = f(x, y)$, por lo que nos referiremos a estos datos como puntos de una superficie (la superficie $F(x, y, z) = f(x, y) - z = 0$).

Un ejemplo clásico es graficar la función $\sin(x\,y)$:

```
In[1]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5}]
```

Out[1]=



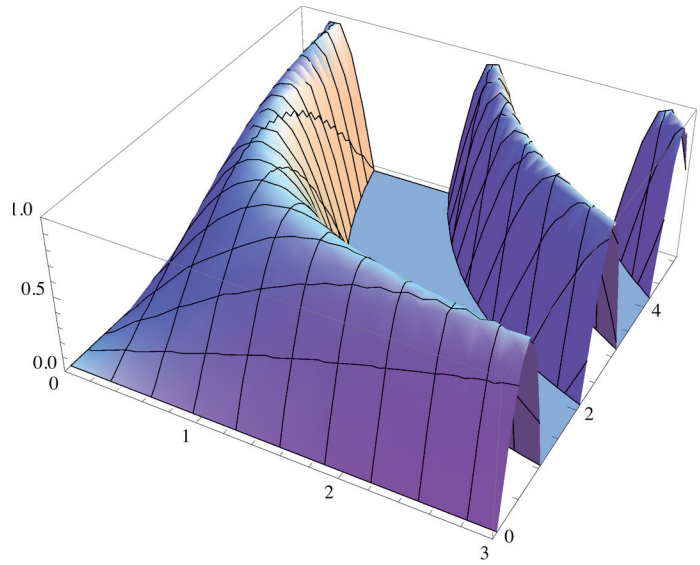
Plot3D[*expr, {var1, min, max}, {var2, min, max}, opciones***]** Grafica la función *f(x,y)=expr*.

La sintaxis de **Plot3D** es muy parecida a la de **Plot**: primero se especifica la función a graficar, luego una lista con la primera variable independiente, su valor mínimo y su valor máximo. La diferencia consiste en que **Plot3D** necesita una segunda lista, la cual tiene la misma forma: segunda variable independiente, valor mínimo y valor máximo. En el caso de nuestro ejemplo, las dos listas son: **{x, 0, 5}** y **{y, 0, 5}**.

Como en el caso de **Plot**, luego de la descripción del dominio de graficación pueden incluirse diversas opciones. Una que nos resulta familiar es **PlotRange**, sólo que ahora debemos especificar dos intervalos, uno para cada variable independiente. Modificando nuestro ejemplo, indicamos con la opción **PlotRange** que sólo queremos mostrar la región con $0 \le x \le 3$, y sólo la parte positiva de la gráfica $0 \le z \le 1$:

```
In[2]:= Plot3D[Sin[x y], {x, 0, 5}, {y, 0, 5},
         PlotRange → {{0, 3}, Automatic, {0, 1}}]
```

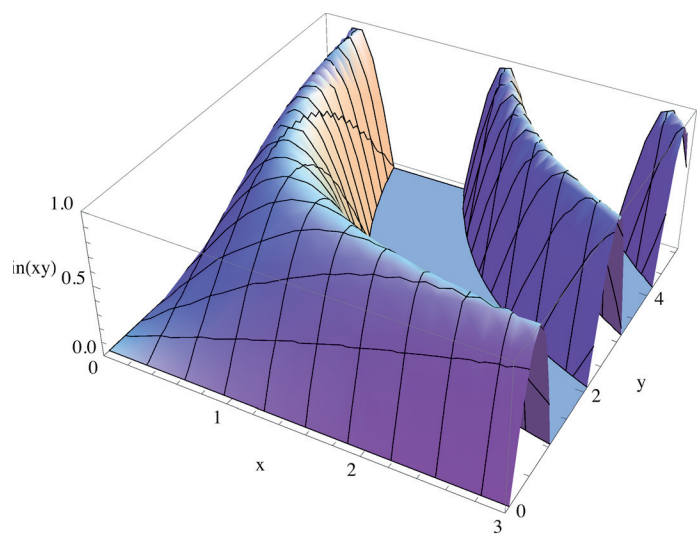
Out[2]=



Para rotular los ejes e identificar más fácilmente a las variables, usamos tambien una opción que es común a **`Plot`**, **`AxesLabel`**:

```
In[3]:= Plot3D[Sin[x y], {x, 0, 5}, {y, 0, 5},
         PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"}]
```

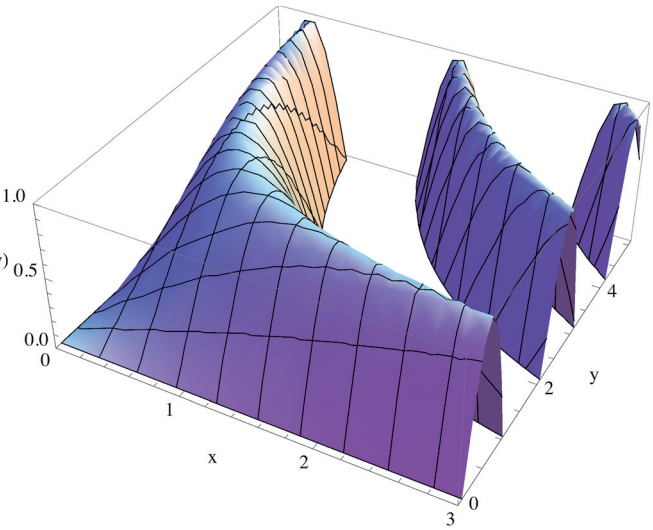
Out[3]=



En esta última gráfica se puede observar que hay una especie de valle donde la función es igual a cero (Ese valle coincide de hecho con la región donde la función es negativa). Por nuestros conocimientos de trigonometría sabemos que ese no es el comportamiento de la función seno, por lo tanto podemos concluír que **`Plot3D`** está arbitrariamente llenando los huecos con polígonos horizontales. Si se quiere que esto no suceda se puede desactivar con la opción **`ClippingStyle->None`**

```
In[4]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
         PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"},
         ClippingStyle → None]
```

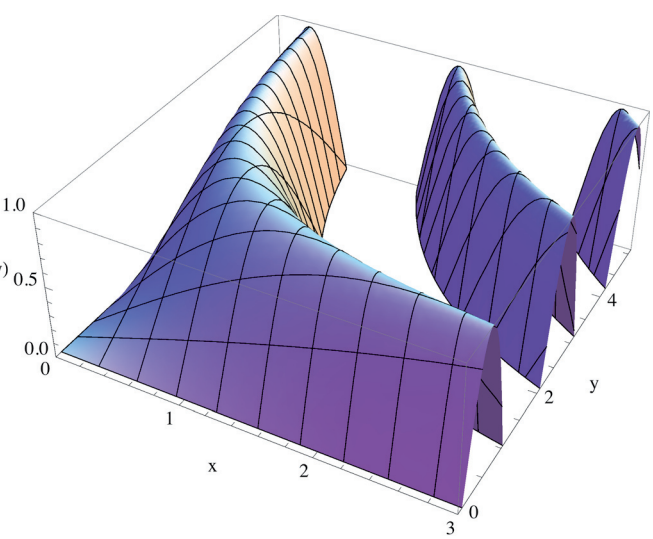
Out[4]=



Viendo la última gráfica notamos que la resolución no es muy alta: podemos ver que la segunda y tercer "olas" de la gráfica están representadas con muy pocos polígonos. Una manera de resolver esto es aumentar el número de puntos donde se evalúa la gráfica mediante la opción **`PlotPoints`**. El valor por omisión de **`PlotPoints`** es de 30 puntos, así que para aumentar la resolución debemos usar **`PlotPoints-> n`**, donde $n > 30$

```
In[5]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
         PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"},
         ClippingStyle -> None,
         PlotPoints → 50]
```

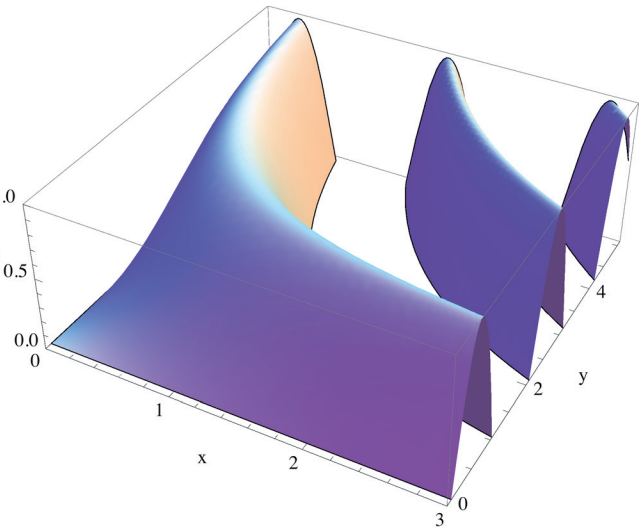
Out[5]=



Ahora podemos ver más claramente las "olas" de la función, pero lamentablemente el enrejado que decora la gráfica oscurece mucho la superficie cuando la resolución es alta. Una manera de evitar este efecto es usar la opción **Mesh->False** y así eliminar por completo el enrejado.

```
In[6]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
         PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"},
         ClippingStyle -> None,
         PlotPoints → 50,
         Mesh → False
        ]
```

Out[6]=

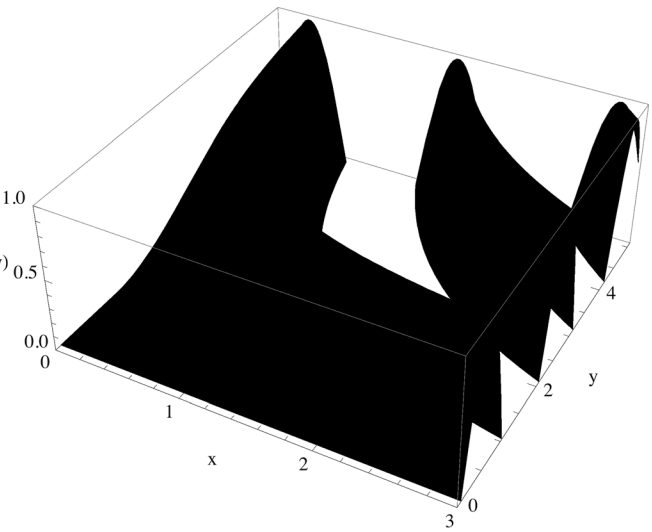


Un concepto importante para trabajar con las superficies que produce **Plot3D** es el de la iluminación: la idea es que después de evaluar la función en los intervalos deseados, **Plot3D** construye un "modelo" de la superficie tal y como se vería al ser iluminado por una serie de "lámparas", cada una con su propia posición, dirección y tono de luz.
El modelo de iluminación estándar de *Mathematica* generalmente da buenos resultados para contrastar las diversas partes de una superficie, aunque también sea posible especificar directamente bajo qué iluminación deseamos que se muestre una gráfica.

En particular, podemos especificar que no haya ninguna iluminación. En ese caso, la gráfica se muestra en blanco y negro: los puntos más bajos aparecen en blanco y los puntos más altos en negro.

```
In[7]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
         PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"},
         ClippingStyle -> None,
         PlotPoints → 50,
         Mesh → False,
         Lighting → None
        ]
```

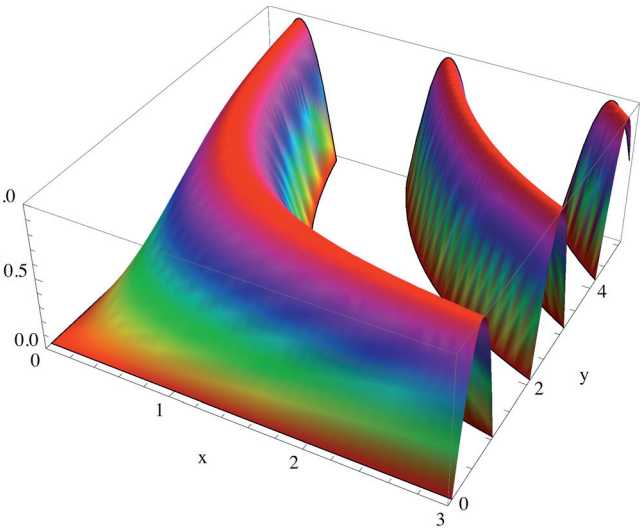
Out[7]=



Un efecto parecido pero a colores se obtiene si añadimos la opción **ColorFunction → Hue**. En este caso los diversos niveles de la gráfica quedan indicados por los colores, en orden periódico de frecuencia (rojo, naranja, amarillo, verde, cian, azul, violeta, rojo,...):

```
In[8]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
          PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"},
         ClippingStyle -> None,
         PlotPoints → 50,
         Mesh → False,
         Lighting → Automatic,
         ColorFunction → Hue
        ]
```

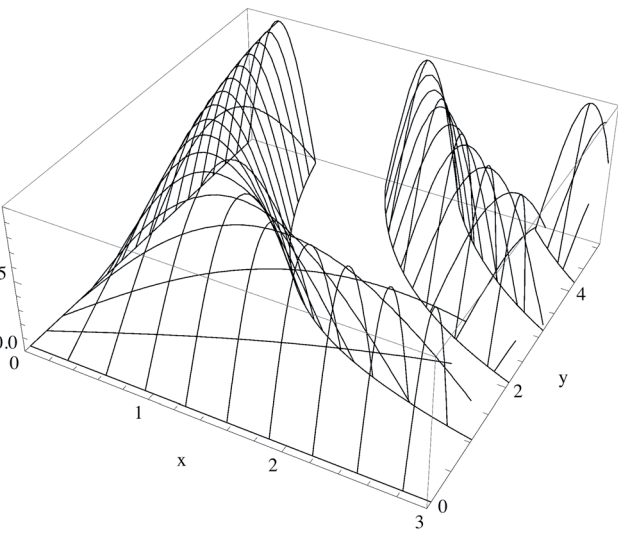
Out[8]=



Muy raras veces es necesario mostrar el enrejado como si la superficie fuera transparente. Para esas ocasiones se usa la opción **`HiddenSurface-> False`**. Sin embargo, en la mayoría de los casos es más claro dejar a la superfice "opaca", por esta razón el valor default de **`HiddenSurface`** es **`True`**. A partir de la versión 6, en lugar de **`HiddenSurface`** se debe usar la opción **`PlotStyle → FaceForm[None].`**

```
In[9]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
          PlotRange → {{0, 3}, Automatic, {0, 1}},
         AxesLabel → {"x", "y", "sin(xy)"},
         ClippingStyle -> None,
         PlotPoints → 50,
         Mesh → True,
         PlotStyle → FaceForm[None]
        ]
```

Out[9]=

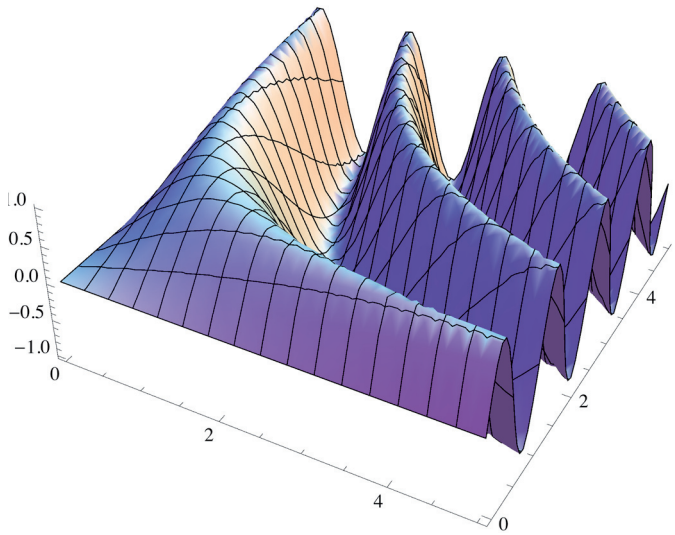


Hay otros dos elementos para gráficas 3D que nos gustaría mencionar. El primero es el control sobre la caja donde se dibuja la gráfica, el segundo es el control del punto de vista desde el cual se presenta la gráfica.

De manera parecida a la opción **`Frame`** para gráficas en 2D, la opción **`Boxed`** permite incluir o eliminar la caja con la que *Mathematica* enmarca una gráfica 3D. El valor por omisión de **`Boxed`** es **`True`**, por lo que para eliminar la caja se usa la opción **`Boxed->False`**:

```
In[10]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5},
           Boxed → False]
```

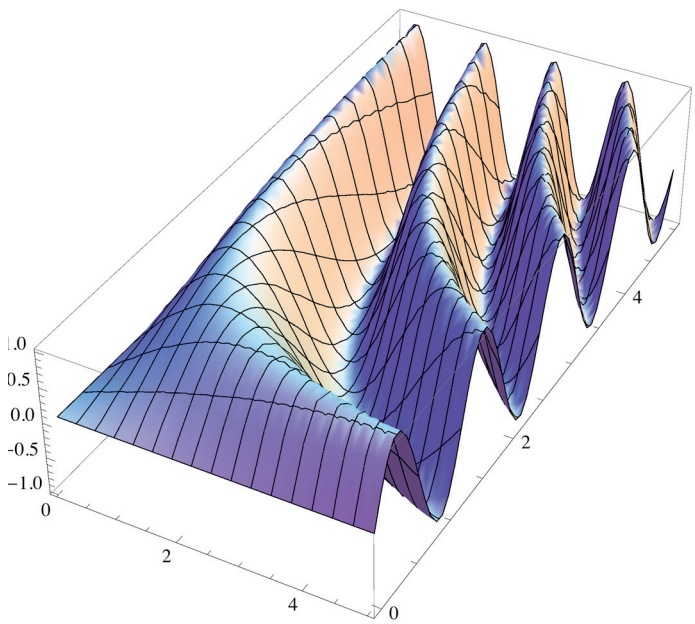
Out[10]=



Otra forma de controlar la caja producida por **`Plot3D`** se logra mediante la opción **`BoxRatios`**. Esta opción se corresponde con **`AspectRatio`** en una gráfica 2D, es decir, nos permite especificar el tamaño relativo de las dimensiones de la caja a lo largo de los tres ejes de la gráfica. Así, **`BoxRatios->{1, 2, 0.5}`** indica que el tamaño de la caja en la dirección *y* es el doble del tamaño en la dirección *x*. Análogamente, el tamaño en la dirección *z* es la mitad del de la dirección *x*.

```
In[11]:= Plot3D[Sin[ x y], {x, 0, 5}, {y, 0, 5}, BoxRatios → {1, 2, 0.5}]
```

Out[11]=



Regresando al tema del punto de vista desde el cuál se observa la gráfica, la idea es que la opción **`ViewPoint->{`***x*,*y*,*z*`}`** indica un punto en el espacio (*x*, *y*, *z*) desde donde el observador ve la superficie graficada. El valor por omisión de **`ViewPoint`** es **`{1.3, -2.4, 2}.`**

Algunos valores comunes se presentan en la siguiente tabla, donde "enfrente" significa mirando a lo largo del eje *x*

`{0, -2, 0}`	enfrente
`{0, -2, 2}`	enfrente y arriba
`{0, -2, -2}`	enfrente y abajo
`{-2, -2, 0}`	esquina inferior izquierda
`{2, -2, 0}`	esquina superior derecha
`{0, 0, 2}`	directamente arriba

Generalmente es mucho más fácil escoger las coordenadas deseadas para **`ViewPoint`** usando el selector de la interfaz gráfica. Para acceder a él, conviene situar el cursor en el punto donde se va a insertar la opción **`ViewPoint`**. Luego, en el menú ***Input*** se selecciona el elemento ***3D ViewPoint Selector***. Aparecerá una nueva ventana, la cual contiene del lado derecho una serie de botones (***Close Dialog***, ***Cancel***, ***Paste***, ***Defaults***, ***Help***) y del lado izquierdo un esquema de una caja mostrando la manera en que se observaría una gráfica para el punto de vista actual.

Arrastrando el mouse sobre dicho esquema se modificará el punto de vista, hasta encontrar la orientación deseada. También se puede ajustar el punto de vista moviendo las tres barras que controlan las coordenadas según los sistemas cartesiano o esférico. Una vez que se haya encontrado el punto de vista apropiado, se pulsa el botón Paste y el selector copia el valor de la opción en el punto donde se encuentra el cursor. Adicionalmente, desde la versión 6, se puede rotar las gráficas 3D haciendo click sobre ellas y desplazando el mouse sin soltar el botón.

Finalmente, mencionaremos algunas opciones más que funcionan de manera análoga tanto para **`Plot3D`** como para **`Plot`**: **`DisplayFunction`**, **`TextStyle`**, **`Prolog`** y **`Epilog`.** Además el comando **`Show`** también puede usarse para combinar gráficas de la misma dimensionalidad, es decir, combinar gráficas 3D con otras gráficas 3D, pero no con gráficas 2D.

Con esto terminamos nuestra discusión de gráficas de funciones y expresiones simbólicas, y pasamos a discutir las gráficas que se pueden realizar a partir de datos tales como listas y tablas.

Ejercicios

Graficar utilizando las diferentes opciones mostradas en este capítulo las siguientes funciones:

1.- $z = sen(x - y)$

2.- $z = x^2\, y^2\, Exp\left[x^2 + y^2\right]$

3.- $x^2 = y$

4.- En la misma gráfica: $z = x^2 + y^2$ y $z = 16 - \left(x^2 + y^2\right)$

5.- $z = sen(x)\, sen(y)$

Capítulo 7

Graficando listas de datos en 2D

ListPlot

Así como el comando **Plot** representa la forma más simple de generar una gráfica a partir de una función, el comando **ListPlot** es tal vez la forma más sencilla de obtener una gráfica a partir de una lista de datos. Ambos comandos comparten muchas opciones para controlar el aspecto de la gráfica, por lo que después de describir la sintaxis propia de **ListPlot**, indicaremos brevemente cuáles opciones se usan del mismo modo que en **Plot** y cuáles son nuevas.

ListPlot[*lista,opciones***]**	Grafica una lista de datos.

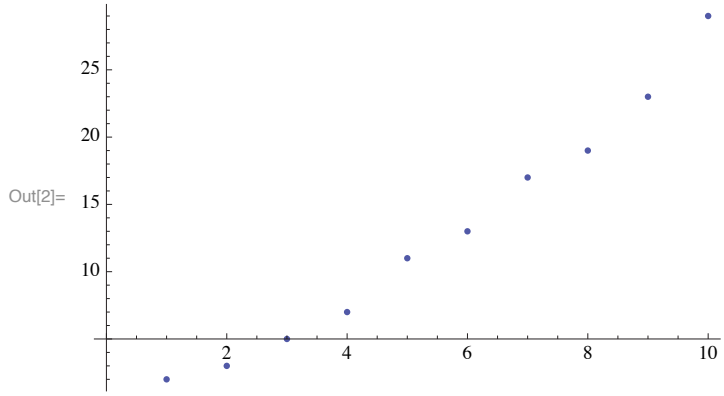
Aquí, *datos* puede ser una lista de valores $\{y_1, y_2, \ldots\}$. Por ejemplo,

```
In[1]:= datos1 = Table[ Prime[n], {n, 1, 10}]
Out[1]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

es una lista con los primeros diez números primos. Para construir una gráfica a partir de esta lista, usamos el comando **ListPlot**

```
In[2]:= grafica1 = ListPlot[datos1]
```

Out[2]=



Los puntos de la gráfica corresponden a los datos en la lista *datos1*, donde se ha asociado implícitamente las abcisas 1, 2, 3, ..., 10 para cada dato, en forma sucesiva.

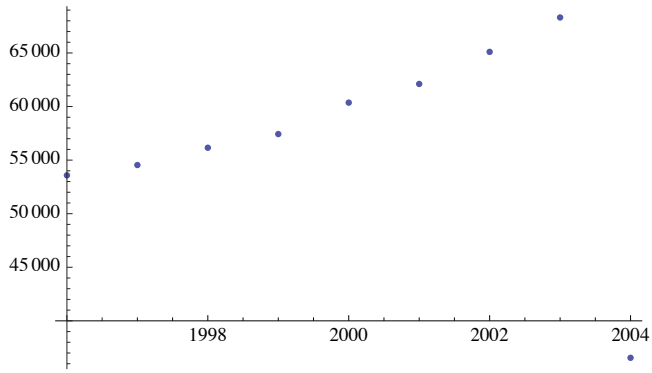
Sin embargo, y más frecuentemente, la lista *datos* también puede ser una lista de pares ordenados $\{\{x_1, y_1\}, \{x_2, y_2\}, \ldots\}$. Para ejemplificar ésto usamos la siguiente tabla, que da el número de artículos sobre cáncer que aparecieron en el mundo entre 1996 y 2004 [1]:

```
In[3]:= datos2 =
          {
           {1996, 53 580},
           {1997, 54 541},
           {1998, 56 154},
           {1999, 57 431},
           {2000, 60 365},
           {2001, 62 109},
           {2002, 65 103},
           {2003, 68 313},
           {2004, 36 551}
          };
```

A pesar de que ahora tenemos una lista de parejas de datos, la sintaxis es exactamente la misma que la del primer ejemplo:

```
In[4]:= grafica2 = ListPlot[datos2]
```

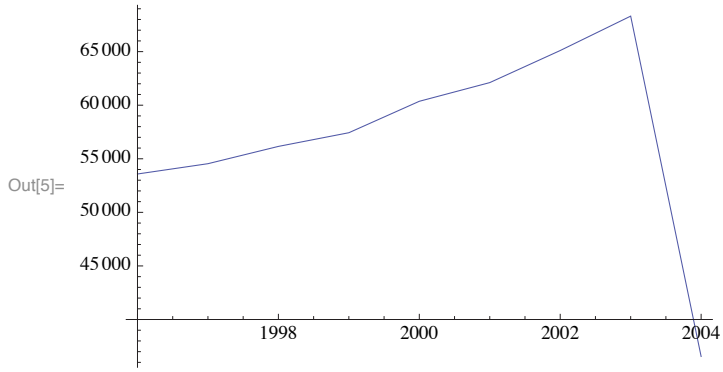
Out[4]=



También es posible unir los puntos de la serie mediante segmentos rectos, esto se indica usando la opción **`Joined.`** Los valores **`True`** y **`False`** (opción por omisión) corresponden, respectivamente, a unir los datos o representarlos únicamente como puntos:

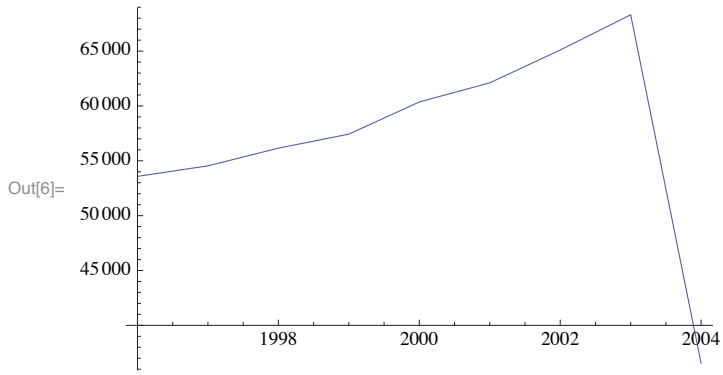
`Joined→True`	Une los puntos graficados mediante segmentos de recta.

```
In[5]:= grafica3 = ListPlot[datos2, Joined → True]
```



Para mayor brevedad, en la versión 6 también se puede usar un nuevo comando ListLinePlot:

```
In[6]:= ListLinePlot[datos2]
```



Opciones comunes entre `ListPlot` y `Plot`

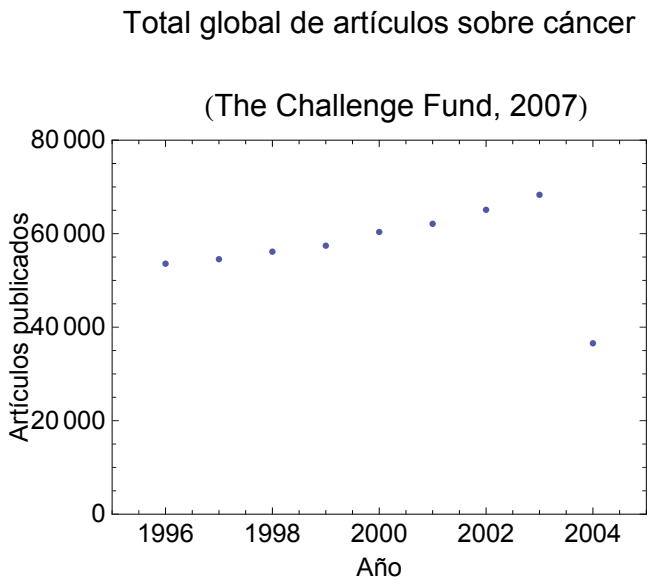
Los diseñadores de *Mathematica* se han preocupado por que haya cierta consistencia entre los comandos que realizan funciones análogas, ya sea con expresiones simbólicas o con expresiones numéricas. Por esta razón muchas de las opciones relevantes a **`Plot`** también se aplican al comando **`ListPlot`.** Por ejemplo, las siguientes opciones tienen exactamente el mismo efecto en ambos comandos:

AspectRatio	Razón de la altura al ancho de la gráfica.
Axes	Controla si se dibujan o no los ejes.
AxesLabel	Indica cómo etiquetar los ejes.
DisplayFunction	Se puede usar para suprimir el despliegue de la gráfica.
Epilog	Comandos gráficos a dibujar **antes** de la gráfica principal.
Frame	Indica si se debe dibujar un marco.
FrameLabel	Indica cómo etiquetar los lados del marco.
PlotLabel	Especifica una etiqueta para la gráfica.
PlotRange	Lista que da los intervalos para graficar las abcisas y las ordenadas.
PlotStyle	Especifica el estilo con que se dibujan las series de datos.
Prolog	Comandos gráficos a dibujar **después** de la gráfica principal.
BaseStyle	Especifica el tipo de letra por defecto para la gráfica.

Combinando estas opciones (explicadas en el capítulo dedicado a **Plot**), tenemos el siguiente ejemplo:

```
In[7]:= grafica3 = ListPlot[datos2, AspectRatio → 0.7,
          Axes → None,
          Frame → True,
          FrameLabel → {"Año", "Artículos publicados"},
          PlotLabel → "Total global de artículos sobre
             cáncer\n\n     (The Challenge Fund, 2007)",
          PlotRange → {{1995, 2005}, {0, 80 000}},
          Prolog → PointSize[0.02`],
          BaseStyle → {FontFamily → "Arial", FontSize → 12}
        ]
```

Out[7]=



Al graficar una serie de datos, aparte de detectar patrones generales, pueden reconocerse datos que se apartan notoriamente de dichas tendencias generales. En este ejemplo, por supuesto, el último punto (que corresponde al año de 2004) claramente está por debajo de lo que se esperaría a partir de extrapolar el comportamiento de todos los años anteriores. Antes de concluir que la productividad científica en el año 2004 se redujo catastróficamente, sería prudente averiguar si la manera de cuantificar los artículos para dicho año fue igual que para los demás. Por ejemplo, si el estudio se realizó a mediados del año 2004 no es posible que incluya los artículos publicados en los meses posteriores a la conclusión del estudio.

Manipulando objetos `Graphics` producidos por `ListPlot` y `Plot`

Los comandos **`Plot`** y **`ListPlot`** tienen en común que sus resultados son objetos de un tipo especial llamado **`Graphics`**. Esto quiere decir que los comandos **`Show`**, **`ShowAnimation`** y **`Export`** pueden ser usados indistintamente. Además los procedimientos para copiar, pegar y guardar todos los objetos **`Graphics`** usando la interfaz gráfica son los mismos, independientemente de cuál comando se usó para producirlos.

`Show`	Se usa para mostrar y combinar objetos de tipo gráfico.
`ShowAnimation`	Produce animaciones a partir de una secuencia de objetos gráficos.
`Export`	Se usa para guardar objetos gráficos en archivos de tipo JPEG, GIF, EPS.
Copy As	Elemento de menú en la interfaz gráfica que copia el gráfico seleccionado.
Save Cell As	Elemento de menú en la interfaz gráfica que guarda gráficos como archivos.

Como ejemplo de cómo se pueden combinar los objetos Graphics producidos por **`Plot`** y **`ListPlot`**, vamos a ajustar la serie **`datos2`**. Primero, eliminamos el último punto por apartarse radicalmente de la tendencia general.

```
In[8]:= datos3 = Drop[datos2, -1]
Out[8]= {{1996, 53 580}, {1997, 54 541}, {1998, 56 154}, {1999, 57 431},
  {2000, 60 365}, {2001, 62 109}, {2002, 65 103}, {2003, 68 313}}
```

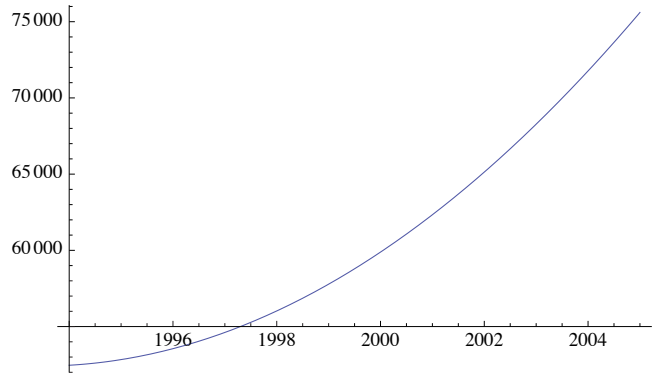
Al observar la gráfica de los puntos se observa una tendencia general que presenta cierta curvatura. Por lo tanto, escogemos un polinomio cuadrático como el modelo para ajustar los datos:

```
In[9]:= f[x_] = Fit[datos3, {1, x, x^2}, x]
Out[9]= 6.88981 × 10^8 - 691 199. x + 173.369 x^2
```

El propósito de definir una función a partir del ajuste recién hecho es que así podemos graficarla más fácilmente. Ahora graficamos el modelo recién ajustado,

```
In[10]:= grafica4 = Plot[ f[x], {x, 1994, 2005}]
```

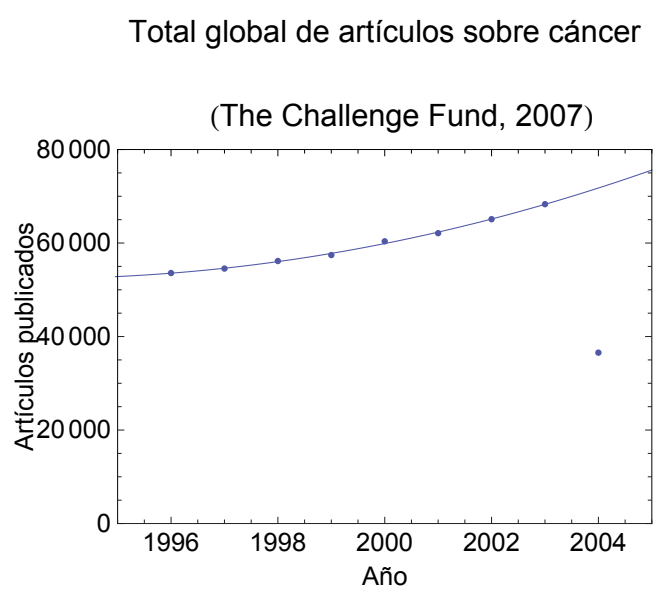
Out[10]=



Finalmente, combinamos los dos objetos **Graphics** obtenidos, a saber, la gráfica de los datos preparada con **ListPlot** y la gráfica del modelo preparada con,

```
In[11]:= grafica5 = Show[grafica3, grafica4]
```

Out[11]=



Ahora podemos cuantificar la magnitud de la caída aparente en el número de artículos publicados con respecto a la tendencia de los años anteriores. Ahora procesamos el valor esperado para 2004 usando el modelo obtenido del ajuste:

```
In[12]:= f[2004]
Out[12]= 71 768.2
```

Comparemos este valor con el dato consignado en la lista oiginal

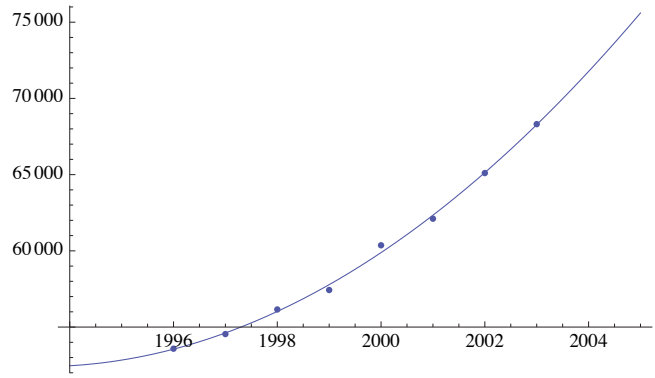
```
In[13]:= f[2004] - datos2[[-1, 2]] / f[2004] * 100
Out[13]= 49.0708
```

En efecto, el valor reportado para 2004 es prácticamente la mitad del valor esperado y mantenemos nuestra sospecha de que algo puede estar errado en esta serie de datos.

Por otra parte, debemos observar que el orden en que aparecen los objetos **Graphics** determina el aspecto final de la gráfica combinada. La primera gráfica que aparece en el comando **Show** es la que determina el valor para las opciones al desplegar la gráfica combinada. Por esta razón, si ponemos primero al ajuste en la lista de argumentos de **Show**, se pierden las opciones para rotular la gráfica, etcétera,

```
In[14]:= grafica6 = Show[grafica4, grafica3]
```

Out[14]=



Otros comandos para graficar listas de datos

Existen otros comandos que producen gráficas a partir de una serie de datos en 2D. Aquí sólo nos referiremos a algunos de los más usados en aplicaciones de ciencias básicas e ingeniería.

BarChart	Genera gráficos de barra.
TextListPlot	Usa el valor de la variable *y* como símbolo para el punto (*x*, *y*).
PieChart	Genera gráficos de pastel (*pie*).
ErrorListPlot	Incluye barras de error.
MultipleListPlot	Grafica múltiples series de datos con símbolos diferentes.
ListPlotVectorField	Grafica datos provenientes de un campo vectorial.

Para poder usar los comandos de la tabla anterior es necesario cargar primero su definición, las cuales están contenidas dentro de un paquete estándar.

Excepto para los dos últimos, todos los comandos de la lista se cargan con la instrucción.

```
In[15]:= << "BarCharts`"; << "Histograms`"; << "PieCharts`"
```

```
General::obspkg :
  BarCharts` is now obsolete. The legacy version being loaded may conflict with current
      Mathematica functionality. See the
      Compatibility Guide for updating information. >>

BarChart3D::shdw :
  Symbol BarChart3D appears in multiple contexts {BarCharts`, System`}; definitions in context
      BarCharts` may shadow or be shadowed by other definitions. >>

General::obspkg :
  Histograms` is now obsolete. The legacy version being loaded may conflict with current
      Mathematica functionality. See the
      Compatibility Guide for updating information. >>

Histogram3D::shdw :
  Symbol Histogram3D appears in multiple contexts {Histograms`, System`}; definitions in context
      Histograms` may shadow or be shadowed by other definitions. >>

General::obspkg :
  PieCharts` is now obsolete. The legacy version being loaded may conflict with current
      Mathematica functionality. See the
      Compatibility Guide for updating information. >>

General::stop : Further output of General::obspkg will be suppressed during this calculation. >>
```

Para cargar a **ErrorBarPlot** y a **VectorFieldPlots** se usan, respectivamente, las instrucciones

```
In[16]:= << "ErrorBarPlots`"
```

y

```
In[17]:= << "VectorFieldPlots`"
```

```
General::obspkg :
  VectorFieldPlots` is now obsolete. The legacy version being loaded may conflict with
      current Mathematica functionality. See the
      Compatibility Guide for updating information. >>
```

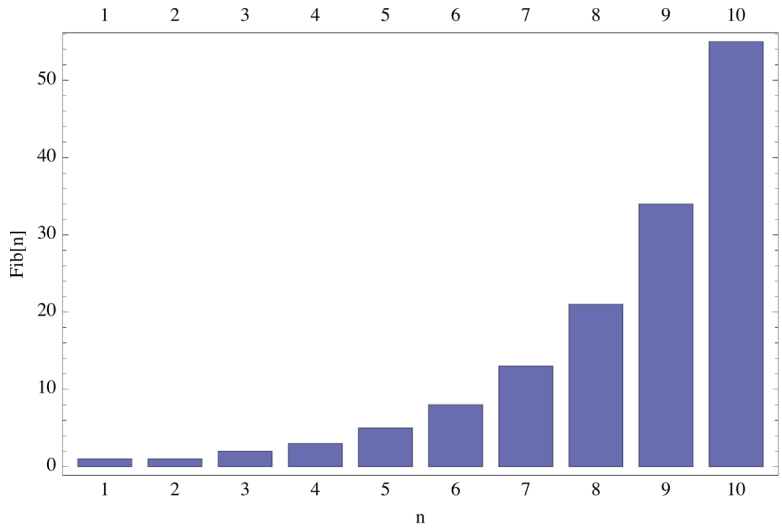
A continuación daremos breves ejemplos de como usar cada uno de estos comandos, usando las opciones que ya conocemos para los objetos **Graphics**.

BarChart[*datos* **]**	Grafica los datos como una gráfica de barras.

En el siguiente ejemplo generaremos una gráfica de barras para la secuencia de Fibonacci

```
In[18]:= grafica7 = BarChart[ Table[ Fibonacci[n], {n, 1, 10}],
           Axes → None,
           Frame → True,
           FrameLabel → {"n", "Fib[n]"} ]
```

Out[18]=

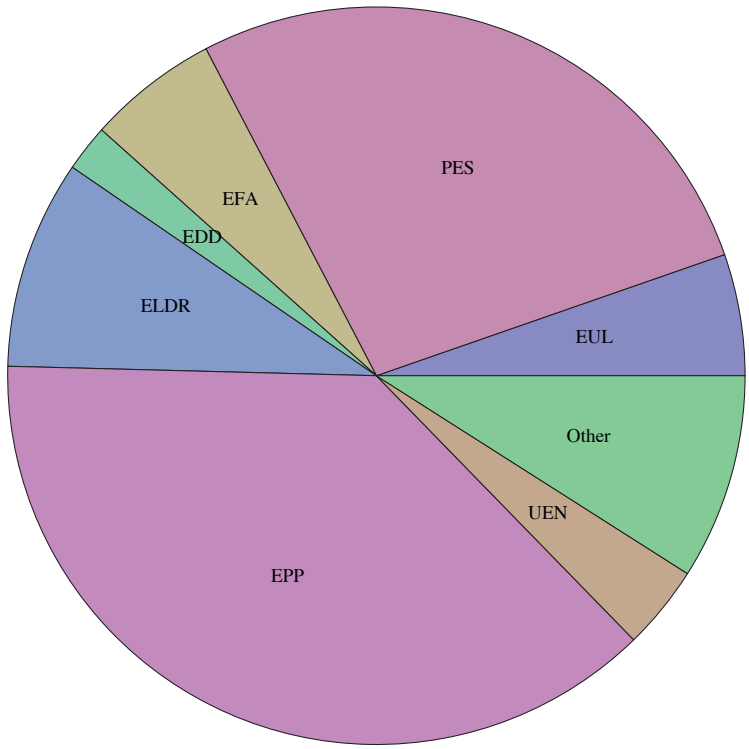


A continuación mostraremos como hacer una gáfica de pastel, para ello tomaremos la composición del Parlamento Europeo tras la elección de 2004.[2] La lista da el número de representantes de cada uno de los grupos parlamentarios, identificados por sus siglas.

PieChart[*datos* **]**	Grafica los datos como una gráfica de pastel (*pie*).

```
In[19]:= grafica9 = PieChart[
           {39, 200, 42, 15, 67, 276, 27, 66},
           PieLabels →
            {"EUL", "PES", "EFA", "EDD", "ELDR", "EPP", "UEN", "Other"}
          ]
```

Out[19]=



Proyecto. Ecuación de estado para un gas

En termodinámica es frecuente representar la relación entre la presión p de un fluido, su densidad ρ y su temperatura T mediante una serie de potencias:

$$\frac{p}{\rho\, k_B\, T} = 1 + B(T)\,\rho + C(T)\,\rho^2 + \ldots$$

A esta ecuación se le conoce con el nombre de la ecuación de estado del virial, y las funciones $B(T)$, $C(T)$, ... reciben el nombre de coeficientes viriales. En particular, $B(T)$ es el segundo coeficiente virial, $C(T)$ es el tercer coeficiente virial, etcétera.

El interés por los coeficientes viriales es doble: si se conocen dichos coeficientes, desde el punto de vista práctico, se pueden predecir todas las propiedades termodinámicas de un gas diluído; desde el punto de vista fundamental, se puede obtener información sobre las fuerzas intermoleculares.

La siguiente lista contiene datos experimentales del tercer coeficiente virial del argón [3], el primer elemento de cada terna es la temperatura T en la escala Kelvin, el segundo es precisamente el tercer coeficiente virial $C(T)$ en unidades de $\text{cm}^6\ \text{mol}^{-2}$y el tercero es la incertidumbre en $C(T)$:

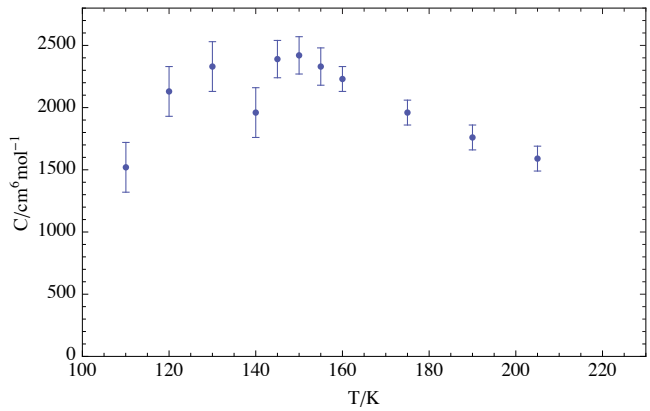
In[20]:=
```
tercerCoeficienteVirial =
  {
   {110.04, 1520, 200},
  {120, 2130, 200},
  {130, 2330, 200},
  {140, 1960, 200},
  {145, 2390, 150},
  {150, 2420, 150},
  {155, 2330, 150},
 {160, 2230, 100},
 {175, 1960, 100},
 {190, 1760, 100},
 {205, 1590, 100}
 };
```

ErrorListPlot[$\{\{x_1,\ y_1,\ \delta y_1\},\ \ldots\}$ **]** Grafica datos incluyendo barras de error (incertidumbre).

A continuación prepararemos una gráfica a partir de los datos experimentales anteriores, en la cual incluímos barras verticales para representar la incertidumbre experimental, esto se hace con el comando **ErrorListPlot**.

In[21]:=
```
grafica10 = ErrorListPlot[ tercerCoeficienteVirial,
   Axes → None,
  Frame → True,
  FrameLabel → {"T/K", "C/cm^6mol^-1"},
   PlotRange → {{100, 230}, {0, 2800}}]
```

Out[21]=



Podemos observar que, a pesar de la incertidumbre de los datos individuales, claramente $C(T)$ presenta un máximo a la temperatura $T_M \approx 140$K. Esta temperatura es similar a la energía de interacción característica entre dos átomos de argón (ϵ) dividida entre la constante de Boltzmann: $T_\epsilon = \frac{\epsilon}{k_B} \approx 146$K. Para averiguar si este parecido es una casualidad o representa una relación general deberíamos analizar el tercer coeficiente virial de un conjunto amplio de moléculas.

Proyecto. Aproximación de Stirling para el logaritmo de la función factorial

La aproximación de Stirling tradicional consiste en aproximar el logaritmo de la función $n!$ como n Log[n] -n. A continuación usaremos una gráfica para verificar si esta aproximación es válida para valores de n suficientemente grandes. Primero, generamos un par de listas, una con los valores calculados directamente de la función factorial y otra con los valores obtenidos de la aproximación,

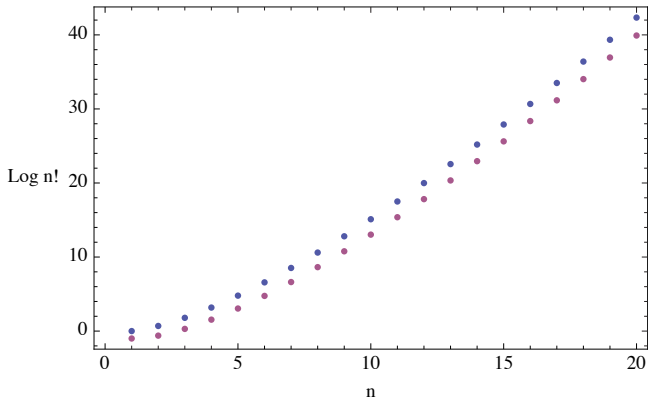
```
In[22]:= datos1 = Table[{n, N[Log[n!]]}, {n, 1, 20}]

Out[22]= {{1, 0.}, {2, 0.693147}, {3, 1.79176}, {4, 3.17805},
          {5, 4.78749}, {6, 6.57925}, {7, 8.52516}, {8, 10.6046},
          {9, 12.8018}, {10, 15.1044}, {11, 17.5023}, {12, 19.9872},
          {13, 22.5522}, {14, 25.1912}, {15, 27.8993}, {16, 30.6719},
          {17, 33.5051}, {18, 36.3954}, {19, 39.3399}, {20, 42.3356}}

In[23]:= datos2 = Table[{n, N[ n Log[n] - n ] }, {n, 1, 20}]

Out[23]= {{1, -1.}, {2, -0.613706}, {3, 0.295837}, {4, 1.54518},
          {5, 3.04719}, {6, 4.75056}, {7, 6.62137}, {8, 8.63553},
          {9, 10.775}, {10, 13.0259}, {11, 15.3768}, {12, 17.8189},
          {13, 20.3443}, {14, 22.9468}, {15, 25.6208}, {16, 28.3614},
          {17, 31.1646}, {18, 34.0267}, {19, 36.9443}, {20, 39.9146}}
```

A continuación utilizaremos el comando **`MultipleListPlot`** para poder presentar las dos series en una sola gráfica.

`MultipleListPlot[` {*lista1*, *lista2*, ...}**`]`**	Grafica multiples series de datos con símbolos distintos.

In[24]:=
```
ListPlot[{datos1, datos2},
 Frame → True, FrameLabel → {"n", "Log n!"},
 RotateLabel → False]
```

Out[24]=

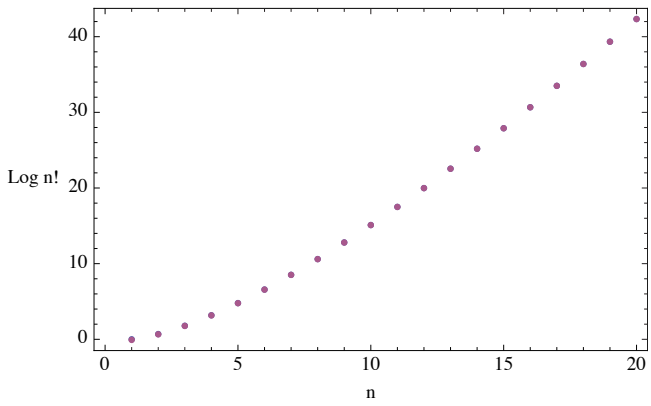


En la gráfica podemos ver que existe una diferencia sistemática entre ambas. La forma tradicional de la aproximación de Stirling omite un término Log[2π]/2. Si incluimos este término, la aproximación mejora notablemente.

In[25]:=
```
datos3 =
 Table[{n, N[(n + 1/2) Log[n] - n + 1/2 Log[2 π]]}, {n, 1, 20}];
```

In[26]:=
```
ListPlot[{datos1, datos3}, Frame → True,
 FrameLabel → {"n", "Log n!"}, RotateLabel → False]
```

Out[26]=



Al usar la mecánica estadística para estudiar las propiedades macroscópicas de un sistema de *n* partículas, *n* toma valores del orden de 10^{23} y por esa razón la constante omitida en la aproximación tradicional no es importante.

Referencias

[1] The Challenge Fund. *Research of cancer in developing countries*.
http://www.cancerworld.org/CancerWorld/moduleStaticPage.aspx?id_stato=1&id_sito=6&id=1274,
(consultado el 28 de enero de 2007)

[2] Wikipedia. *European Parliament election, 2004*
http://en.wikipedia.org/wiki/European_Parliament_election%2C_2004
(consultado el 28 de enero de 2007)

[3] J.H.Dymond, R.C.Wilhoit, K.N.Marsh, K.C.Wong, M.Frenkel. *Virial Coefficients of Pure Gases*.
Springer (2002)

Ejercicios

1. Graficar el cuadrado de los enteros positivos del 1 al 10.

2. Graficar una tabla de 100 números enteros aleatorios entre 0 y 100.

3. Graficar los primeros 50 números primos.

4. Graficar los 100 primeros números de la serie de Fibonacci.

5. Graficar los puntos (0,0), (2,7), (3,5) y (4,11) y unirlos por una línea.

6. Graficar los valores de n! para n=1, 2, 3, 4 y 5.

7. Graficar los valores de 2^n para n=1, 2, 3, 4 y 5.

8. Poner en una sola gráfica las correspondientes a los ejercicios 6 y 7, usando símbolos diferentes para cada serie de datos.

9. Repetir la gráfica anterior uniendo los puntos con líneas.

10. Hacer una gráfica de barras y una de pastel donde se representen los días de cada mes del año.

Capítulo 8

Graficando listas de datos en 3D

Hemos visto ya cómo usar el comando **Plot3D** para graficar funciones de dos variables independientes (es decir, para obtener gráficas tridimensionales). Ahora vamos a presentar comandos útiles para graficar datos en tres dimensiones: es importante recordar que los datos corresponderán a los valores de una función $z = f(x, y)$, por lo que nos referiremos a estos datos como puntos de una superficie (la superficie $F(x, y, z) = f(x, y) - z = 0$).

El primer comando para graficar datos en 3D que abordaremos es **ListPlot3D**. Este comando cumple respecto a **Plot3D** un papel análogo al de **ListPlot** respecto a **Plot**, es decir, las versiones con prefijo **List** sirven para graficar listas de datos**.**

Es importante recordar que los datos generalmente representan alguna información que tenemos sobre el sistema o proceso que estamos tratando de entender. Frecuentemente, obtener los datos involucra un procedimiento largo y costoso: por ejemplo, hacer una serie de experimentos, realizar una serie de encuestas o entrevistas, recopilar información de varias fuentes, etcétera. Por esta razón los datos generalmente estarán disponibles en ciertas zonas del dominio de definición de las variables independientes, formando un conjunto discreto. De ahí la importancia de poder graficar listas de datos discretos.

Para concentrarnos en los aspectos gráficos, supondremos que sabemos cómo traducir el formato en que fueron capturados los datos al de una lista de *Mathematica*. (Véase la información en este libro sobre los comandos **Import** y **Export**).

Aquí simplemente simularemos que tenemos algunos datos experimentales (por ejemplo, digamos que medimos el nivel z del agua en un canal poco profundo, como función de la posición (x, y) en el momento en que se abre una compuerta y se deja pasar el agua desde un nivel más alto.) Sin pretender ser muy realistas, digamos que la superficie del agua tiene el aspecto de una tangente hiperbólica (es decir, en lados opuestos de la compuerta el nivel es similar, pero cerca de la compuerta varía rápidamente con la posición). Para simular el efecto de que se trata de una medición, superponemos a esta tangente hiperbólica un ruido de fondo aleatorio:

```
In[1]:= superficie1 = Table[Tanh[x + y] + RandomReal[],
          {x, -10, 9}, {y, -10, 9}];
```

Obsérvese que hemos preparado la tabla incluyendo solamente los valores de z, dejando fuera los valores de la posición (x, y). Ahora vamos a visualizar esta superficie en 3D.

Gráfica de superficies tridimensionales con ListPlot3D

El comando **ListPlot3D** toma por argumento un arreglo rectangular de números reales, que representan las alturas de la superficie a graficar (es decir, se trata de una matriz z_{ij}). El arreglo no incluye las coordenadas de posición, porque el comando **ListPlot3D** presupone que las coordenadas de la posición son iguales a los índices i y j de la matriz de datos.

En nuestro ejemplo, el arreglo que vamos a pasar a **ListPlot3D** es una matriz de 20 x 20 elementos,

Dimensions[*expresión***]**	Da una lista con las dimensiones de *expresión*.

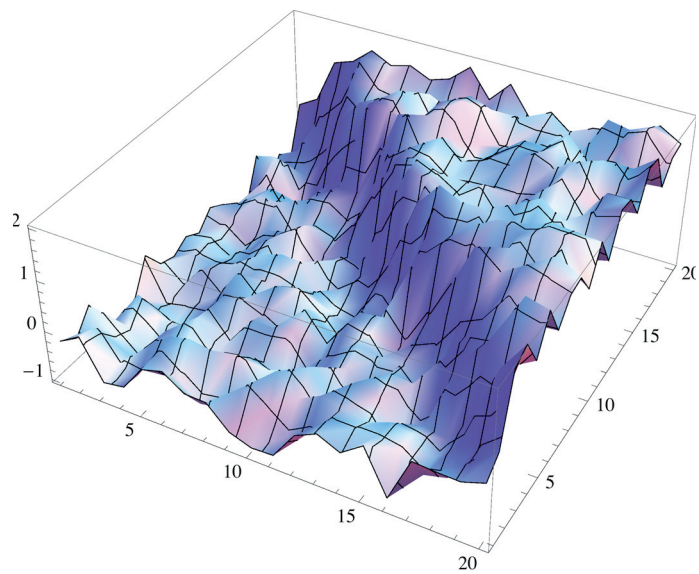
```
In[2]:= Dimensions[superficie1]
Out[2]= {20, 20}
```

ListPlot3D[*datos,opciones***]**	Grafica una lista de datos de dos variables.

Evaluando **ListPlot3D** con nuestra superficie de ejemplo, obtenemos la siguiente gráfica,

```
In[3]:= ListPlot3D[ superficie1]
```

Out[3]=



Nótese que la gráfica varía según los números aleatorios que fueron obtenidos al momento de definir la superficie.

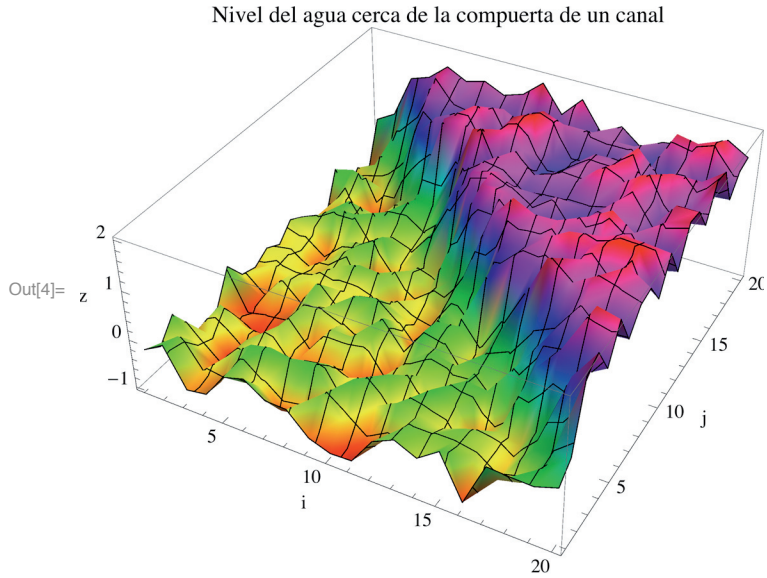
Mucho de lo que sabemos respecto de cómo controlar el aspecto de las gráficas hechas con **Plot3D** se puede aplicar también a **ListPlot3D**. En particular, las opciones siguientes son comunes entre los dos comandos (hay más):

AspectRatio, Axes, AxesLabel, AxesStyle, ClipFill, ColorFunction, DisplayFunction, Epilog, Lighting, LightSources, Mesh, PlotLabel, PlotRange, Prolog, Shading, TextStyle, Ticks, ViewPoint.

Una versión más sofisticada de la misma gráfica, usando varias de estas opciones, podría ser,

```
In[4]:= ListPlot3D[ superficie1,
         AxesLabel → {"i", "j", "z"}, ColorFunction → Hue,
         PlotLabel → "Nivel del agua cerca de la compuerta de un canal"]
```

Out[4]=



Observemos que ahora los ejes están rotulados y que el color de los elementos de la superficie ahora dependen del valor de z (usando **`ColorFunction-> Hue`**). El valor mínimo corresponde a tonos rojizos-amarillos, el valor máximo corresponde a tonos azules-violetas; el máximo y el mínimo coinciden en el tono rojo. También añadimos un título a la gráfica. Este es simplemente un ejemplo para señalar que podemos aplicar todo lo que sabemos para modificar, combinar y animar gráficas también a las gráficas producidas con **`ListPlot3D`**.

Ahora presentaremos algunas formas alternativas a **`ListPlot3D`** para visualizar listas de datos organizados en tres dimensiones. Con esto queremos decir que aunque las figuras no tratan de dar la impresión de ser una vista en 3D, sí hacen énfasis en que los datos están organizados en un arreglo rectangular que corresponde a dos variables independientes.

Gráfica de densidad

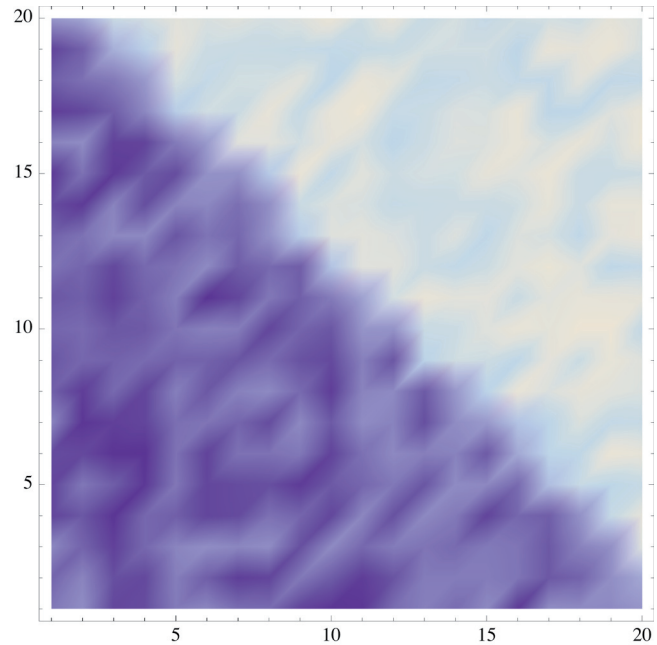
A veces conviene presentar la información de la tercera dimensión como un color o un tono en escala de grises, usando el comando **`ListDensityPlot`**. Los mapas de bits corresponden justamente a este tipo de representación: el tono que le toca a cada pixel en la posición (x, y) se especifica como un número entre 0 y 1, donde 0 representa el negro y el 1 el blanco, con todos los valores intermedios representando intensidades del gris: 0.25 representaría una mezcla de 25% blanco y 75% negro. A este tipo de gráficas las llamaremos gráficas de densidad, porque podemos imaginarnos que son una forma de representar la densidad de "blanco" que hay que usar para cada pixel.

`ListDensityPlot[`*datos,opciones***`]`**	Gráfica una lista de datos de dos variables representada por medio de tonos.

Aplicando **ListDensityPlot** a nuestra superficie de ejemplo obtenemos lo siguiente,

```
In[5]:= ListDensityPlot[ superficie1]
```

Out[5]=



Aunque no pretende darnos una imagen en perspectiva de 3D, esta gráfica es bastante informativa. Por ejemplo, nos muestra inmediatamente la "línea divisoria" entre los niveles altos y bajos de agua en el canal. También nos muestra claramente que hay una variación (aleatoria) del nivel, a partir del hecho de que los tonos de cada lado de la división no son uniformes.

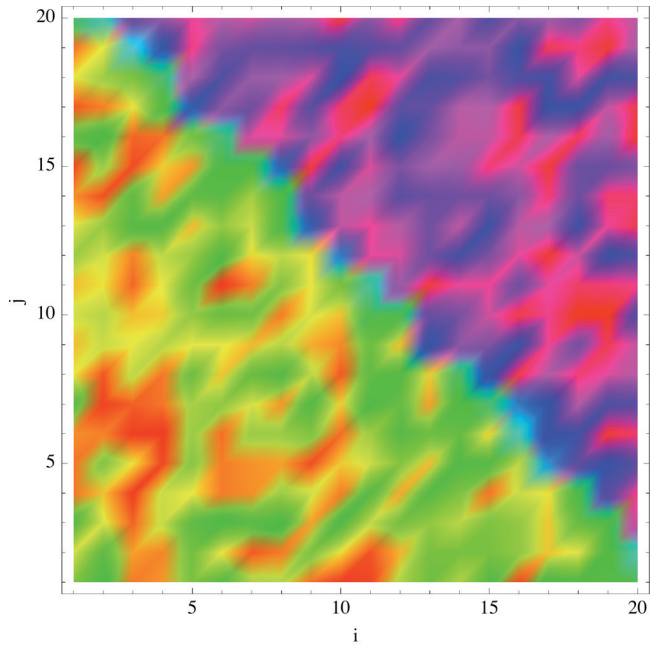
Entre las opciones de **ListDensityPlot** encontramos muchas conocidas:

AspectRatio, Axes, AxesLabel, ColorFunction, DisplayFunction, Epilog, Frame, FrameLabel, Mesh, PlotLabel, PlotRange, Prolog, RotateLabel, TextStyle, Ticks.

A continuación se muestra una versión ligeramente modificada del ejemplo anterior, usando color y nombrando los ejes.

```
In[6]:= ListDensityPlot[ superficie1,
          FrameLabel → {"i", "j "},
          ColorFunction → Hue]
```

Out[6]=

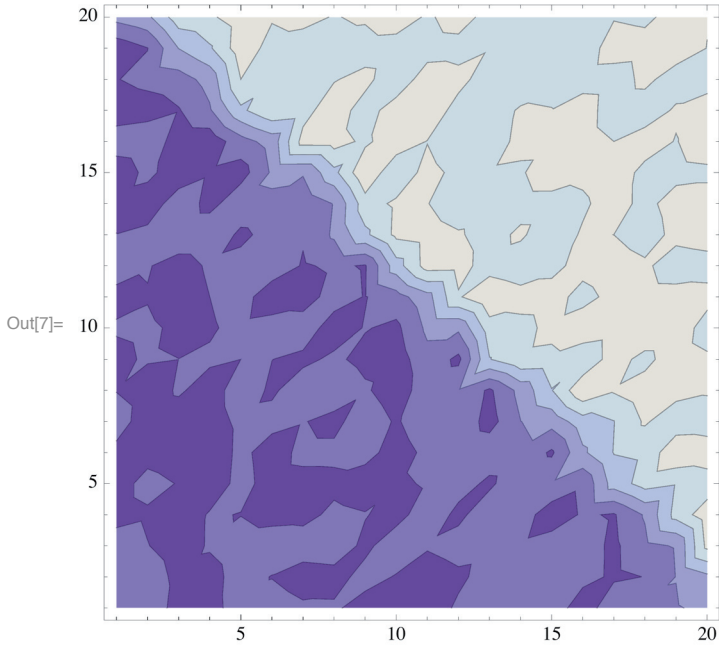


Gráfica de contornos

Un modo alterno de investigar el comportamiento de nuestros datos en tres dimensiones, consiste en utilizar una gráfica de contornos, mediante el comando **`ListContourPlot`**.
Una gráfica de contornos es básicamente una colección de curvas de nivel, es decir, primero se escogen una serie de valores de la altura $\{h_1,\ h_2,\ ...\}$ y se grafican las curvas (contornos) que tienen altura constante $z = h_n$. Luego, el interior de cada contorno se colorea para distinguirlo de los demás:

`ListContourPlot[`*datos,opciones***`]`**	Gráfica una lista de datos de dos variables representada por medio de contornos.

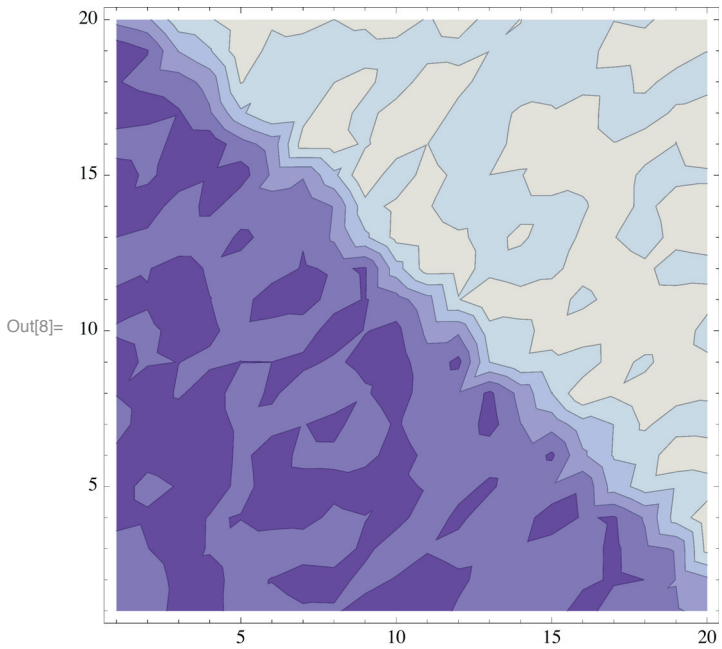
```
In[7]:= ListContourPlot[ superficie1]
```



Aparte de las opciones comunes que ya conocemos, algunas opciones de **ListContourPlot3D** particularmente útiles son **Contours, ContourLines, ContourStyle, ContourSmoothing.**

Contours sirve para controlar cuántas y cuáles curvas de nivel se usarán en la gráfica. Si queremos especificar que queremos únicamente 5 curvas de nivel, usamos el comando

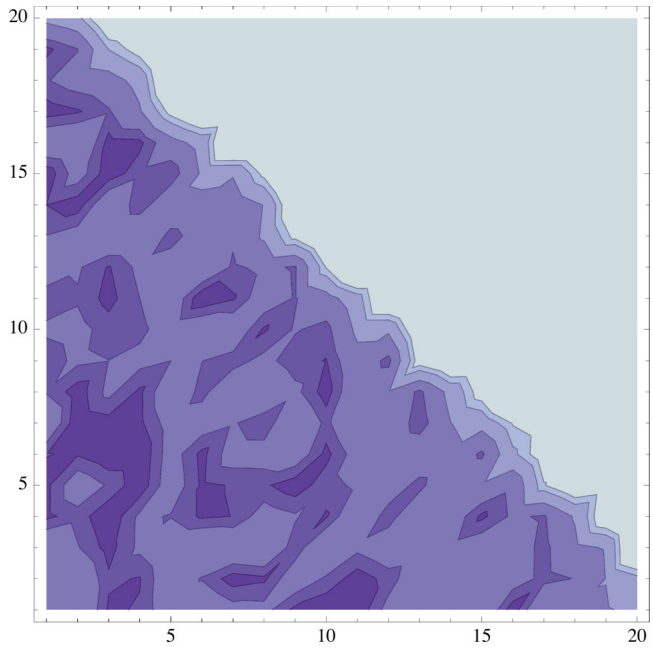
```
In[8]:= ListContourPlot[ superficie1, Contours → 5]
```



Si además queremos especificar cuáles niveles queremos usar, entonces los nombramos explícitamente en una lista,

```
In[9]:= ListContourPlot[ superficie1,
         Contours → {-0.75, -0.5, 0, 0.5, 0.75}]
```

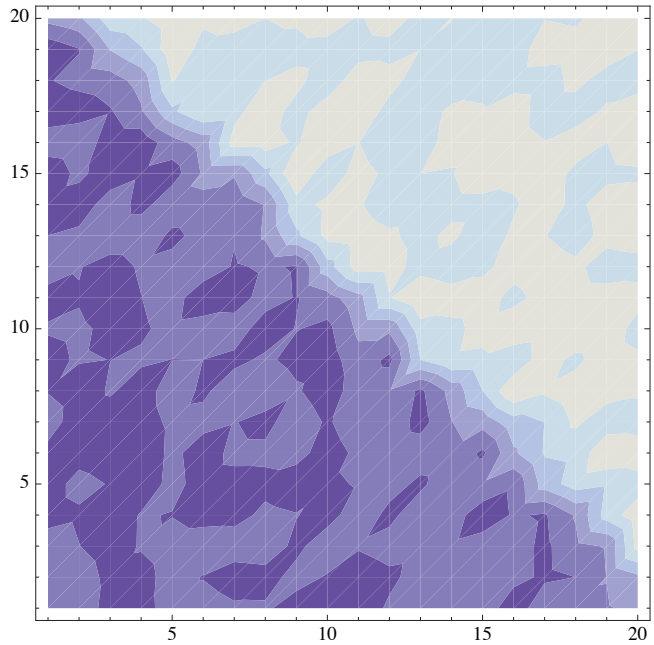
Out[9]=



También se puede controlar el estilo con que se presentarán las curvas de nivel. En primer lugar, con la opción **`ContourStyle`** pueden ocultarse, o mostrarse explícitamente, las curvas de nivel. La opción por defecto es mostrarlas (**`ContourStyle -> True`**), pero podemos desactivarlas de la manera siguiente

```
In[10]:= ListContourPlot[superficie1, Contours → 5, ContourStyle → None]
```

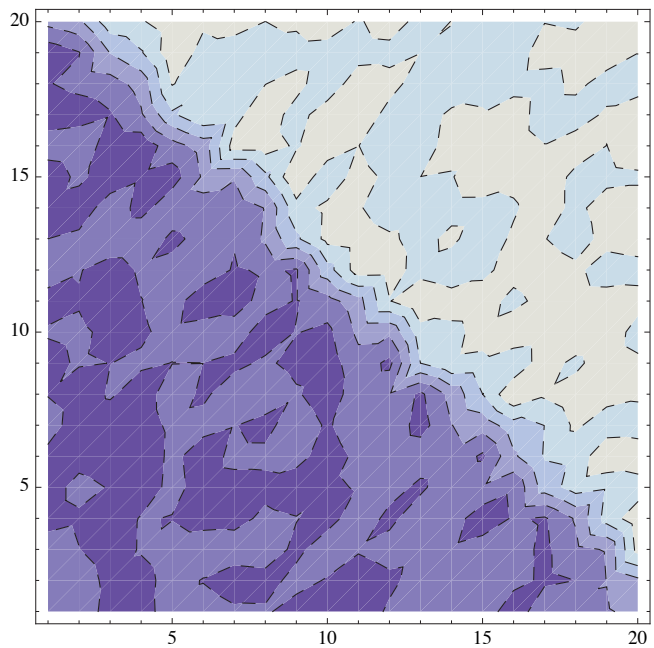
Out[10]=



En segundo lugar, la opción **`ContourStyle`** sirve para especificar el estilo de las curvas de nivel mostradas en la gráfica, de manera análoga a la opción **`PlotStyle`** en **`Plot`**. Se puede especificar el estilo para todas las curvas o estilos diferentes para cada curva de nivel.

```
In[11]:= ListContourPlot[ superficie1, Contours → 5,
          ContourStyle → Dashing[{0.02, 0.02}]
         ]
```

Out[11]=



Finalmente mencionaremos que la opción **ContourSmoothing** sirve para suavizar las curvas de nivel. Dado que sólo tenemos datos limitados sobre la superficie, las curvas de nivel son poligonales bastante irregulares. Para mejorar su apariencia, se pueden utilizar técnicas de interpolación entre los datos de la lista y así suavizar los contornos. La opción **ContourSmoothing** realiza este procedimiento por defecto, pero se puede desactivar especificando **ContourSmoothing -> False.**

Interpolación de una lista de datos en 3D

Ya que hablamos de la posibilidad de usar interpolación para suavizar las gráficas de listas de datos, convinene mencionar que podemos hacer dicha interpolación nosotros mismos, usando el comando **Interpolation**, para luego generar una gráfica con **Plot3D**. Una ventaja de usar esta técnica es que las coordenadas de la gráfica corresponderán a los valores de las variables independientes (x, y) y no a los índices de la matriz z_{ij}.

Para preparar la interpolación, primero generamos una tabla con ternas de datos (x, y, z).

```
In[12]:= tablaDeDatos = Table[
           {x, y, superficie1[[11 + x, 11 + y]]}, {x, -10, 9}, {y, -10, 9}];
```

Esta tabla contiene 20 x 20 ternas,

```
In[13]:= Dimensions[tablaDeDatos]
Out[13]= {20, 20, 3}
```

El comando **Interpolation** opera, sin embargo, sobre una lista que contenga una secuencia de ternas, $\{(x_1,\ y_1,\ z_1), (x_2,\ y_2,\ z_2),\ ...,\ (x_N,\ y_N,\ z_N)\}$. Para poner la tabla de datos en esta forma, aplicamos el comando **Flatten** hasta el nivel 1:

```
In[14]:= superficieConCoordenadas = Flatten[ tablaDeDatos, 1] ;
In[15]:= Dimensions[superficieConCoordenadas]
Out[15]= {400, 3}
```

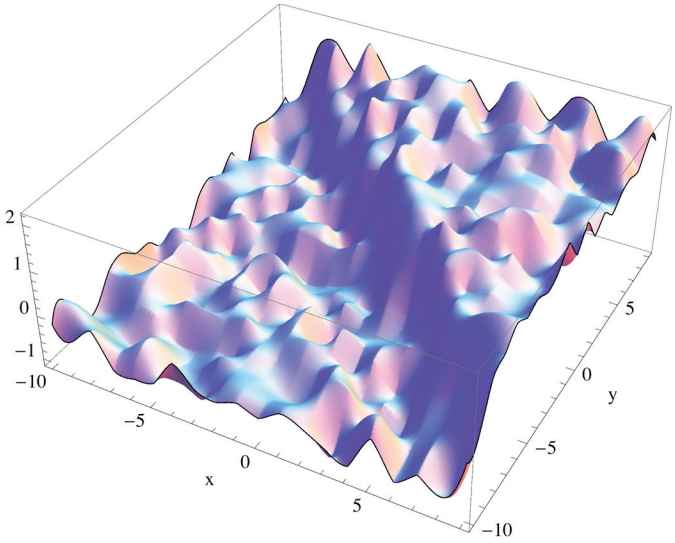
Ahora generamos la interpolación deseada:

```
In[16]:= superficieInterpolada = Interpolation[ superficieConCoordenadas]
Out[16]= InterpolatingFunction[{{-10., 9.}, {-10., 9.}}, <>]
```

Aplicando lo que hemos aprendido para graficar funciones con el comando **Plot3D**, obtenemos la siguiente figura.

```
In[17]:= Plot3D[ superficieInterpolada[x, y], {x, -10, 9}, {y, -10, 9},
  Mesh → False, PlotPoints → 100, AxesLabel → {"x", "y", "z"}]
```

Out[17]=



Existen muchas más opciones y comandos para generar gráficos, las que presentamos aquí fueron seleccionadas tanto porque son usadas con mucha frecuencia en la práctica, como porque nos permiten ilustrar que han sido diseñadas para que se comporten de manera unificada, y por lo tanto sea fácil aprender a usar más tipos especializados de gráficas a partir de estos pocos ejemplos.

Ejercicios

1. Graficar $z = \text{sen}(x - y)$ en el intervalo $-\pi \le x \le \pi$ y $-\pi \le y \le \pi$.

2. Graficar $z = x^2\, y^2 \exp\left[-\left(x^2 + y^2\right)\right]$ en el intervalo $-2 \le x \le 2$ y $-2 \le y \le 2$.

3. Graficar $z = x^2 - y^2$ en el intervalo $-5 \le x \le 5$ y $-5 \le y \le 5$.

4. Graficar $z = \exp\left[-x^2 - y^2\right]$ en el intervalo $-1 \le x \le 1$ y $-1 \le y \le 1$.

5. Graficar $z = \text{sen}\, x\, \text{sen}\, y$ en el intervalo $-2\pi \le x \le 2\pi$ y $-2\pi \le y \le 2\pi$.

6. Graficar la intersección de la parábola $z = x^2 + y^2$ con el plano $y + z = 12$.

7. Graficar $z = 3\sqrt{x^2 + y^2}$ en coordenadas cilíndricas en el intervalo $0 \le r \le 3$ y $0 \le \theta \le 2\pi$.

8. Graficar $x^2 + y^2 + (z - 9)^2 = 9$ en coordenadas cilíndricas en el intervalo $0 \le r \le 3$ y $0 \le \theta \le 2\pi$.

9. Poner las dos gráficas anteriores en una sola.

10. Obtener las gráficas de contorno y densidad de una tabla de números aleatorios de 10 por 10.

Capítulo 9

Cálculo diferencial e integral

9.1 Cálculo diferencial

Límites

El cálculo diferencial y el cálculo integral se introducen generalmente mediante la idea de límite. La definición de límite formaliza la intuición de que para funciones suaves, a medida que la variable se acerca a un valor dado, la función tiende a cierto valor definido. Por ejemplo, la función

```
In[1]:= f[x_] := 1 + 2 x
```

debe tender al valor límite 3 a medida que la variable x tiende a 1. En *Mathematica* el comando **Limit** permite evaluar este límite para cualquier expresión que dependa de x

```
In[2]:= Limit[1 + 2 x, x → 1]
Out[2]= 3
```

Limit[*expresión, variable → valor***]**	Evalua el límite de *expresión* cuando *variable* tiende al *valor* especificado.

Por supuesto, *Mathematica* puede calcular límites incluso en casos en que la función en sí misma no está definida para el valor específico al que se hace tender la variable independiente. Un caso muy conocido es el de la siguiente función, la cual no está definida para el valor $x=0$

```
In[3]:= f[x_] := Sin[x] / x
In[4]:= f[0]
```

Power::infy : Infinite expression $\frac{1}{0}$ encountered. ≫

∞::indet : Indeterminate expression 0 ComplexInfinity encountered. ≫

```
Out[4]= Indeterminate
```

Limit es capaz de evaluar el límite en el caso $x \to 0$

```
In[5]:= Limit[ Sin[x] / x, x → 0]
Out[5]= 1
```

Limit también opera sobre expresiones simbólicas, es decir, trata de establecer el límite de funciones en las que aparecen otras variables además de la que se hace tender al valor especificado,

```
In[6]:= Limit[ Sin[ a x] / x, x → 0]
Out[6]= a
```

Por último, en algunos casos (por ejemplo, cerca de una discontinuidad) es importante especificar desde qué dirección se toma el límite: para ello se usa la opción **Direction**, que puede tomar los valores -1 y 1 según se quiera que el límite se calcule desde la izquierda o desde la derecha, respectivamente.

Derivada de una expresión

La definición matemática de la derivada $f'(x) = \frac{df}{dx}$ de una función $f(x)$ consiste en el siguiente límite

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Afortunadamente, *Mathematica* incorpora un comando especializado para calcular derivadas, este es el comando **D**.

D[*f, variable* **]**	Evalua la derivada de *f* respecto a la variable especificada.

Por ejemplo, si queremos calcular la derivada del siguiente polinomio, $f(x) = 1 + 2\,x + 3\,x^2$, usamos la expresión

```
In[7]:= D[ 1 + 2 x + 3 x^2, x]
Out[7]= 2 + 6 x
```

Adicionalmente, si la expresión que quiere derivarse es una función de un sólo argumento, digamos,

```
In[8]:= f[x_] := 1 + 2 x + 3 x^2
```

También puede usarse una notación más compacta que consiste en poner el caracter apóstrofe ' (que se lee "prima") entre el nombre de la función y los corchetes del argumento:

```
In[9]:= f'[x]
Out[9]= 2 + 6 x
```

Esta notación permite además evaluar la derivada en cualquier valor deseado (numérico o simbólico),

```
In[10]:= f'[x]
Out[10]= 2 + 6 x
In[11]:= f'[1]
Out[11]= 8
```

También se puede aplicar a las funciones predeterminadas de *Mathematica* :

```
In[12]:= Sin'[x]
Out[12]= Cos[x]
In[13]:= Sin'[0.5]
Out[13]= 0.877583
```

Derivadas de expresiones simbólicas

Un aspecto muy poderoso en *Mathematica* es que el comando **D** y la notación ' (prima) funcionan igualmente bien cuando la expresión a derivar contiene constantes simbólicas además de constantes numéricas. Por ejemplo, en los cursos de cálculo se aprende la regla de derivación para potencias de la variable independiente:

```
In[14]:= f[x_] := x^n
In[15]:= f'[x]
Out[15]= n x^(-1+n)
```

Otras regla muy conocida es la regla de la cadena,

```
In[16]:= f[x_] := g[h[x]]
In[17]:= f'[x]
Out[17]= g'[h[x]] h'[x]
```

Esto hace que Mathematica sea muy útil para calcular o revisar derivadas de expresiones largas.

Proyecto. Cálculo de la rapidez mínima de un proyectil en caída libre

Imaginemos que se dispara un proyectil con una rapidez inicial v_0 a un ángulo θ sobre la horizontal y que a continuación el proyectil se encuentra en caída libre. En este ejemplo calcularemos la rapidez mínima del proyectil y el tiempo en que ocurre ésto.

A partir de las ecuaciones de movimiento en las coordenadas horizontal y vertical,

$$r(t) = \{x, y\} = \left\{ v_0 \cos\theta\, t,\ v_0 \sin\theta\, t - \frac{gt^2}{2} \right\}$$

se puede calcular la velocidad instantánea tomando la derivada del vector de posición r con respecto al tiempo,

```
In[18]:= r[t_] := {v0 Cos[θ] t, v0 Sin[θ] t - 1/2 g t^2}

In[19]:= v[t_] = D[ r[t], t]
Out[19]= {Cos[θ] v0, -g t + Sin[θ] v0}
```

La rapidez del proyectil es simplemente la raíz cuadrada de la norma del vector velocidad:

$$v = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{v_x^2 + v_y^2}$$

```
In[20]:= rapidez[t_] = Sqrt[ v[t]. v[t] ]
Out[20]= Sqrt[Cos[θ]^2 v0^2 + (-g t + Sin[θ] v0)^2]
```

Ahora podemos encontrar el mínimo de la rapidez, derivando la expresión anterior con respecto al tiempo e igualando a cero el resultado,

```
In[21]:= condicion = D[rapidez[t], t] == 0
                    g (-g t + Sin[θ] v0)
Out[21]= - ------------------------------------------ == 0
           Sqrt[Cos[θ]^2 v0^2 + (-g t + Sin[θ] v0)^2]
```

Resolviendo esta condición para t, encontramos que el tiempo al que ocurre el valor mínimo de la rapidez es igual a $t_{\min} = \frac{v_0}{g} \sin\theta$.

```
In[22]:= Solve[condicion, t]
                  Sin[θ] v0
Out[22]= {{t → ------------}}
                      g
```

Evaluando la rapidez para dicho tiempo, encontramos el valor mínimo buscado,

```
In[23]:= rapidezMinima = rapidez[t] /. t → Sin[θ] v0 / g
Out[23]= √(Cos[θ]^2 v0^2)
```

Sabiendo que el ángulo de disparo está entre 0° y 90°, y que la rapidez inicial v_0 es positiva, podemos simplificar la respuesta anterior,

```
In[24]:= Simplify[ rapidezMinima, Assumptions → {v0 > 0,  0 < θ < π / 2 }]
Out[24]= Cos[θ] v0
```

Este resultado final coincide exactamente con la magnitud de la componente horizontal de la velocidad del proyectil. Una manera de entender esto es observando que la rapidez $\sqrt{v_x{}^2 + v_y{}^2}$ de un proyectil en caída libre sólo cambia debido a la variación de la componente vertical de la velocidad; en consecuencia, en el punto más alto de la trayectoria donde dicha componente es cero, el proyectil alcanza su mínima rapidez.

Derivada total y derivada parcial

En los ejemplos anteriores, hemos usado el comando **`D[f,x]`** para derivar la expresión **`f`** en términos de **`x`** *tomando como constantes todas las variables distintas de* **`x`**. A veces, sin embargo, deseamos obtener la derivada suponiendo que todas (o algunas) de las variables de hecho dependen de x.

Por ejemplo, digamos que tenemos la expresión $f=a\,x^2 + b\,x$ y que en dicha expresión todas las variables dependen de x. Para calcular la derivada total $\frac{\mathrm{d}f}{\mathrm{d}x}$, considerando estas dependencias implícitas de *a* y *b*, empleamos el comando **`Dt`** (derivada total):

`Dt[`*f, variable* **`]`**	Evalua la derivada total de f respecto a la variable especificada.

```
In[25]:= Dt[ a x^2 + b x, x]
Out[25]= b + 2 a x + x^2 Dt[a, x] + x Dt[b, x]
```

Observemos que las derivadas totales de a y b con respecto a x aparecen simbólicamente como **`Dt[a,x]`** y **`Dt[b,x]`**.

El comando **`D[f,x]`** es de hecho una derivada parcial $\frac{\partial f}{\partial x}$, es decir, el resultado de derivar la expresión f con respecto a x manteniendo todas las demás variables constantes:

```
In[26]:= D[a x^2 + b x, x]
Out[26]= b + 2 a x
```

Debido a que **D[f,x]** representa una derivada parcial, existe una notación alternativa para esta operación: $\partial_x f$. Esta notación recuerda la notación tradicional del cálculo, $\frac{\partial f}{\partial x}$.

En cierto sentido, **D[f, x]** y **Dt[f,x]** son polos opuestos respecto de cómo se tratan las variables que aparecen la expresión f. En **D** todas las variables distintas de x se toman como constantes, mientras que en **Dt** todas las variables se toman como dependientes de x.

Cuando al calcular una derivada necesitamos tener mayor control sobre cuáles variables deben permanecer constantes y cuáles deben tomarse como dependientes de x, podemos usar cualquiera de los comandos **D** o **Dt**, dando la opción correspondiente **Constants** o **NonConstants**.

Por ejemplo, digamos que tenemos la expresión $a\,x^2 + b\,y^2$ y que deseamos calcular la derivada con respecto a x, sabiendo que de hecho a y b son constantes pero que y depende de x. Una primera forma de calcularla es,

```
In[27]:= D[a x^2 + b y^2, x, NonConstants → {y}]
Out[27]= 2 a x + 2 b y D[y, x, NonConstants → {y}]
```

La segunda forma es usar el comando **Dt**, pero especificando que a y b son constantes

```
In[28]:= Dt[a x^2 + b y^2, x, Constants → {a, b}]
Out[28]= 2 a x + 2 b y Dt[y, x, Constants → {a, b}]
```

Proyecto. Derivación implícita

Supongamos que se tiene una ecuación de la forma

$$a\,x^2 + b\,x\,y + y^2 - r^2 == 0$$

en la cuál no es simple despejar una variable en términos de otra, y de todas maneras queremos calcular la derivada de *y(x)*. Una manera de hacer esto es mediante derivación implícita, esto es, derivando ambos lados de la igualdad, recordando el hecho de que y es una función de *x*. En *Mathematica* esto es justamente el sentido de la derivada total, **Dt**, si especificamos que *a* , *b* y r son constantes,

```
In[29]:= ecuacionDiferencial =
          Dt[a x^2 + b x y + y^2 - r^2, x, Constants → {a, b, r}] == 0
Out[29]= 2 a x + b y + b x Dt[y, x, Constants → {a, b, r}] +
           2 y Dt[y, x, Constants → {a, b, r}] == 0
```

Ahora podemos despejar la derivada deseada en la ecuacion diferencial recién obtenida,

```
In[30]:= Solve[ecuacionDiferencial, Dt[y, x, Constants → {a, b, r}]]
```

$$\text{Out[30]= } \left\{\left\{\text{Dt[y, x, Constants} \to \text{\{a, b, r\}}] \to \frac{-2\,a\,x - b\,y}{b\,x + 2\,y}\right\}\right\}$$

Derivadas de orden superior

Frecuentemente es necesario calcular la segunda derivada, tercera derivada, o derivadas de órdenes superiores. *Mathematica* permite calcular fácilmente el resultado de estas operaciones. La notación de primas sigue operando para derivadas de orden superior para funciones de un solo argumento,

```
In[31]:= {Cos'[x], Cos''[x], Cos'''[x], Cos''''[x]}
Out[31]= {-Sin[x], -Cos[x], Sin[x], Cos[x]}
```

Cuando el orden de la derivada es alto, es mejor usar la forma completa (no abreviada),

Derivative[n][*f, variable* **]**	Evalua la derivada *n*-ésima total de *f* respecto a la variable especificada.

```
In[32]:= Derivative[n][f][x]
```

$$\text{Out[32]= } f^{(n)}[x]$$

Es importante observar que en la notación **f'[x]** (y de la misma manera en la forma completa **Derivative[f][x]**) el argumento **[x]** no indica la variable respecto a la que se está derivando, sino el punto en el que se quiere evaluar la derivada. Para aclarar esto, consideremos el caso de la función Cos[k x]. La primera derivada de f respecto a x es -k Sin[k x]. Sin embargo, observemos el resultado de la siguiente expresión

```
In[33]:= Cos'[k x]
Out[33]= -Sin[k x]
```

La razón de que no obtengamos el factor k esperado es que el significado de **Cos'[k x]** es: "Calcula la derivada de la función **Cos** respecto a su único argumento, y evalúa el resultado en el punto **k x**."
Para obtener la derivada deseada, es probablemente más claro usar el comando **D**:

```
In[34]:= D[Cos[k x], x]
Out[34]= -k Sin[k x]
```

Derivadas parciales de orden superior y mezcladas

Para obtener derivadas parciales de orden superior, la notación del operador **D** cambia ligeramente: el segundo argumento se transforma en una lista de dos elementos, el primero indica la variable de derivación y el segundo elemento indica el orden de la derivada deseada.

D[*f*,{ *variable*,*n*} **]**	Calcula la derivada repetida *n* veces de *f* respecto a la variable especificada.

```
In[35]:= D[func[x], {x, n}]
Out[35]= func^(n)[x]
```

Por ejemplo, la derivada repetida de segundo orden con respecto a **x** de **Cos[x y]** se obtiene como,

```
In[36]:= D[Cos[x y], {x, 2}]
Out[36]= -y^2 Cos[x y]
```

También es posible calcular derivadas parciales mezcladas, es decir, donde la derivación se hace sucesivamente respecto a variables diferentes. En este caso, después de la expresión a derivar, se añade una secuencia de las variables respecto de las cuales se quiere derivar,

```
In[37]:= D[Cos[x y], x, y]
Out[37]= -x y Cos[x y] - Sin[x y]
```

Por supuesto, también es posible calcular derivadas parciales mezcladas de orden superior. En este caso, después de la expresión a derivar se añade una secuencia de parejas ordenadas, especificando cada variable y orden de derivación deseado:

```
In[38]:= D[Cos[x y], {x, 2}, {y, 3}]
Out[38]= 6 x^2 y Cos[x y] + 6 x Sin[x y] - x^3 y^2 Sin[x y]
```

Esta flexibilidad para obtener todo tipo de derivadas parciales es muy práctica y útil en muchos campos de la ciencia y la ingeniería.

Series de Taylor

Una de las aplicaciones del cálculo diferencial es la de obtener una expresión aproximada de una función en la forma de una serie infinita llamada serie de Taylor. Típicamente, esta serie converge al valor de la función original dentro de cierto intervalo (finito). La fórmula es la siguiente:

$$f(x) = f(x_0) + \sum_{n=1}^{\infty} f^{(n)}(x_0) \frac{(x-x_0)^n}{n!}$$

Mathematica puede evaluar derivadas de cualquier orden, por lo que es fácil evaluar las derivadas que aparecen en la fórmula. De hecho, *Mathematica* contiene el comando **`Series`** que evalúa los primeros n términos de la serie de Taylor de f alrededor del punto x_0 .

`Series[`*f*,{ *variable*,x_0, *n*} **`]`**	Evalúa los primeros *n* términos de la serie de Taylor de *f* al rededor de x_0.

Por ejemplo, la función exponencial tiene una serie de Taylor, alrededor de $x_0 = 0$, muy sencilla,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots.$$

La cual se puede obtener con *Mathematica* de la siguiente forma,

```
In[39]:= Series[ Exp[x], {x, 0, 5}]
```

$$\text{Out[39]= } 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O[x]^6$$

Las series de Taylor también pueden emplearse con funciones de varias variables, en este caso, el cambio en la notación consiste en añadir listas que indiquen la variable respecto a la que se hace el desarrollo en serie, el punto alrededor del que se hace el desarrollo y el orden hasta el cual se trunca la serie.

```
In[40]:= Series[ Exp[x y], {x, 0, 5}, {y, 0, 2}]
```

$$\text{Out[40]= } 1 + \left(y + O[y]^3\right) x + \left(\frac{y^2}{2} + O[y]^3\right) x^2 + O[y]^3 x^3 + O[y]^4 x^4 + O[y]^5 x^5 + O[x]^6$$

Esta capacidad es útil, por ejemplo, al tratar de representar expresiones complicadas en intervalos pequeños mediante expresiones más simples (polinomios). También sirve para desarrollar teorías en las que ciertos efectos "pequeños" pueden tratarse aproximadamente mediante órdenes sucesivos de la serie de Taylor. Es importante mencionar, sin embargo, que las series de Taylor pueden no corresponder con el comportamiento verdadero de la función que tratan de representar cuando se truncan o cuando se utilizan más allá de su radio de convergencia. Para aprender más sobre estos temas, se recomienda estudiar el cálculo diferencial e integral de funciones de variable compleja.

Asignación diferida e inmediata y el cálculo de derivadas

Hemos visto varias maneras de obtener la derivada de una expresión simbólica, el resultado es por supuesto otra expresión simbólica. ¿Cómo podemos evaluar eficientemente dicha derivada para uno o varios valores específico de x, digamos x=1? En *Mathematica* hay por lo menos tres maneras.

La primer forma es probablemente la más simple: evaluamos la derivada simbólicamente y a continuación aplicamos una regla del tipo **x →** *valor*,

```
In[41]:= D[1 + 2 x + 3 x^2, x] /.  x → 1
Out[41]= 8
```

Un problema con este enfoque, sin embargo, es que con frecuencia necesitamos a la derivada misma como una función que se pueda evaluar para muchos valores distintos de x.

A continuación daremos los pasos conducentes que pueden ser útiles en estos casos. Primero asignamos el resultado de la derivada a una función nueva, digamos, $f_1[x]$, y a continuación evaluamos la función recién definida para el valor de x que nos interesa.

Por ejemplo primero definimos nuestra función de partida

```
In[42]:= f[x_] := 1 + 2 x + 3 x^2
```

Luego definimos una nueva función como el resultado de la operación de derivación,

```
In[43]:= f1[x_] = D[f[x], x]
Out[43]= 2 + 6 x
```

Ahora podemos evaluar la derivada para cualquier valor de x,

```
In[44]:= f1[1]
Out[44]= 8
```

Es importante notar que en la asignación del resultado de la derivada usamos el operador de asignación inmediata (**=**) en vez del acostumbrado operador de asignación diferida (**:=**). ¿Cuál es la razón para ello? Si usamos asignación diferida, al calcular el valor de **f1[1]**, *Mathematica* primero sustituye el lado derecho de la definición de **f1[x]** usando en vez de **x** el valor **1**. Es decir, *Mathematica* trata de calcular la siguiente expresión:

```
In[45]:= D[f[1], 1]
         General::ivar :  1 is not a valid variable. ≫
Out[45]= ∂1 6
```

Desafortunadamente, al sustituir **1** como valor de **x**, la operación de derivada ya no tiene sentido y obtenemos tan sólo un mensaje de error (**1** no es una variable válida).

En resumen, cuando definamos funciones directamente en términos del operador **D** aplicado a otras funciones, es importante recordar que debemos usar asignación inmediata (**=**) en vez de asignación diferida (**:=**)

La tercera forma de evaluar derivadas es una combinación de las dos anteriores: usando una regla de remplazo para evitar el problema de tratar de derivar respecto a un valor numérico. Primero, nos percatamos que la variable de derivación es una variable muda, depués podemos reemplazarla por cualquier otro símbolo y tendremos el mismo resultado.

```
In[46]:= D[f[x], x] /. x → 1
Out[46]= 8
```

es equivalente a

```
In[47]:= D[f[w], w] /. w → 1
Out[47]= 8
```

Por lo tanto podemos definir nuestra función derivada aplicando el comando **D** con una variable muda, y luego remplazando ésta con el valor en donde vamos a evaluar la derivada

```
In[48]:= f1[x_] := D[f[w], w] /. w → x
```

Observemos que ahora sí estamos usando asignación diferida (**:=**), pero que ahora ya no tendremos problemas. Por supuesto, el resultado final es el mismo que el obtenido con los otros dos métodos.

```
In[49]:= f1[x]
Out[49]= 2 + 6 x
```

La ventaja del tercer método es que si por cualquier razón, modificamos la definición de la función base $f[x]$ la derivada $f_1[x]$ se modificará automáticamente, mientras que con el segundo método habria que volver a ejecutar la asignación inmediata (**=**) para que se realizara el cambio.

9.2 Cálculo integral de una variable

Integrales en 1D

En *Mathematica*, la integral $\int f(x)\,dx$ de una función $f(x)$ con respecto a x se calcula con el comando **Integrate**.

Integrate[*f, variable* **]**	Evalúa la integral de *f* con respecto la variable especificada.

Por ejemplo,

```
In[50]:= Integrate[x^2, x]
```

$$\text{Out[50]=}\ \frac{x^3}{3}$$

Si se usa la interfaz gráfica, se puede usar la paleta llamada ***Basic Input*** para hacer el mismo cálculo usando una notación más parecida a la usada en los libros de texto:

$$\text{In[51]:=}\ \int x^2\, dx$$

$$\text{Out[51]=}\ \frac{x^3}{3}$$

Independientemente de qué estilo usemos, es importante recordar que a cualquier integral indefinida se le puede añadir una constante arbitraria y obtener otra integral indefinida, en el sentido de que las derivadas de ambas expresiones serán iguales al integrando. *Mathematica* simplemente devuelve una entre una infinidad de funciones que difieren a lo más por una constante.

Aunque calcular integrales y derivadas son en cierta medida operaciones inversas, difieren importantemente en su complejidad. Por un lado, existen métodos generales que permiten calcular la derivada de una expresión hecha de funciones elementales (potencias, logaritmos, trigonométricas, etcétera) de manera sistemática. Por otro lado, no existe un método general que garantice encontrar la integral indefinida de una expresión hecha de funciones elementales.

Por ejemplo, la derivada de la siguiente función $g(x)$

```
In[52]:= g[x_] := Log[Sin[1 - x^2]]
```

es el producto de x y una función trigonométrica,

```
In[53]:= D[g[x], x]
Out[53]= -2 x Cot[1 - x^2]
```

Sin embargo, la integral indefinida de $g(x)$ no se puede expresar en términos de funciones elementales.

```
In[54]:= Integrate[ g[x], x]
```

$$\text{Out[54]=}\ \int \text{Log}\left[\text{Sin}\left[1 - x^2\right]\right] dx$$

Antes del desarrollo de programas de álgebra simbólica era común recurrir a manuales o tablas de integrales para evalúar los casos más complicados de integración indefinida. A lo largo de muchos años de desarrollo *Mathematica* ha adquirido la capacidad de poder encontrar la integral indefinida de casi todos los casos cubiertos en dichos manuales y, lo que es más importante, para conseguirlo se tuvieron que desarrollar algoritmos generales que permiten encontrar otras nuevas integrales indefinidas que no se conocían antes.

Integrales definidas

En muchas aplicaciones es necesario evalúar integrales definidas del tipo

$$\int_a^b f(x)\,dx = \mathrm{F(b)} - \mathrm{F(a)}$$

donde $F(x)$ es una primitiva (integral indefinida) del integrando $f(x)$. En estos casos, el comando usado para evalúar la integral indefinida también es **`Integrate`**, la única diferencia consiste en la forma del segundo argumento. Mientras que en la integral indefinida sólo se indicaba la variable de integración, para obtener una integral definida indicamos una lista de tres elementos: la variable de integración, el límite inferior y el límite superior de integración.

`Integrate[`*f, variable,{variable,LimInf,LimSup}* **`]`**
Evalúa la integral de *f* con respecto a la variable, entre los límites especificados.

Por ejemplo,

```
In[55]:= Integrate[x^2, {x, 0, 10}]
Out[55]= 1000/3
```

El argumento, por supuesto, puede ser cualquier expresión simbólica o numérica,

```
In[56]:= int1 = Integrate[ Sin[2 x] Cos[2 x], {x, α, β}]
Out[56]= 1/8 (Cos[4 α] - Cos[4 β])
```

En ocasiones, más que el valor preciso de la integral definida para una combinación de parámetros dada estamos más interesados en comprender el comportamiento general de la integral a medida de que la variable independiente recorre su dominio, o del efecto de modificar uno u otro parámetro. Es en dichas ocasiones que tiene mayor valor el poder hallar y comprender una expresión de las integrales definidas en términos de funciones conocidas. Al aplicar nuestro conocimiento sobre estas funciones sabemos que esperar de la integral definida (y del fenómeno o sistema que estamos estudiando).

Integrales múltiples

Aunque comprender el concepto de las integrales múltiples toma algún tiempo en un curso regular de cálculo, una vez que se entiende su significado es inmediato usar *Mathematica* para calcularlas. Por ejemplo, digamos que necesitamos calcular una integral múltiple indefinida

$$\int \left(\int \left(\int f(x, y, z) \ \mathrm{dz} \right) \mathrm{dy} \right) \mathrm{dx}$$

donde *f(x,y,z) = z·Log(x·y)*. Una vez más, usamos el comando **Integrate** y modificamos un poco la sintaxis. Ahora, tras el integrando añadiremos tantos argumentos como variables de integración se necesiten. Así, para nuestro ejemplo escribimos,

```
In[57]:= Integrate[ Log[x y] z, x, y, z]
```

$$\text{Out[57]= } \frac{1}{2} x\, y\, z^2\, (-2 + \mathrm{Log}[x\, y])$$

Otro caso es el de una integral definida múltiple. Digamos que debemos evalúar

$$\int_0^1 \left(\int_0^2 \left(\int_0^3 f(x, y, z)\, \mathrm{dz} \right) \mathrm{dy} \right) \mathrm{dx}$$

En este caso, en vez de indicar una sola lista después del integrando, se escribirán tantas listas como variables de integración haya, cada una contendrá la variable respectiva y los dos límites del intervalo de integración correspondiente,

```
In[58]:= Integrate[ Log[x y] z,  {x, 0, 1}, {y, 0, 2}, {z, 0, 3}]
Out[58]= -18 + Log[512]
```

Vale la pena hacer algunas aclaraciones, el orden de integración implícito para el comando **Integrate** es que las variables indicadas más a la derecha son las que se realizan primero (es decir, son las integrales anidadas a nivel más profundo). En nuestro ejemplo, esto quiere decir que primero se realiza la integración en *z*, luego la integración en *y*, dejando al final la integración en *x*.
Modifiquemos ahora un poco el ejemplo anterior, digamos que el límite de integración para z dependa de y,

```
In[59]:= Integrate[ Log[x y] z,  {x, 0, 1}, {y, 0, 2}, {z, 0, y / 2}]
```

$$\text{Out[59]= } \frac{1}{9} (-4 + \mathrm{Log}[8])$$

Esta integral se calcula sin problemas porque la integral sobre z se hace primero, lo cual produce una expresión que depende de *y* que puede integrarse sin dificultades en el siguiente nivel de integración. Sin embargo, si queremos hacer que el límite superior para la *y* dependa de *z* debemos modificar el orden de integración,

```
In[60]:= Integrate[ Log[x y] z,  {x, 0, 1}, {z, 0, 3}, {y, 0, z / 2}]
```

$$\text{Out[60]=}\quad \frac{3}{2}\left(-7+\text{Log}\left[\frac{27}{8}\right]\right)$$

Si no tenemos cuidado y dejamos sin modificar el orden de integración, encontraremos erróneamente la siguiente expresión,

```
In[61]:= Integrate[ Log[x y] z,  {x, 0, 1}, {y, 0, z / 2}, {z, 0, 3}]
```

$$\text{Out[61]=}\quad \frac{9}{4}\,z\left(-2+\text{Log}\left[\frac{z}{2}\right]\right)$$

Este resultado no es siquiera una integral definida, ya que todavía depende de las variables (mudas) de integración. ¿Cómo llegó *Mathematica* a esta expresión que no corresponde a lo que estamos buscando? En primer lugar, *Mathematica* realiza la integración sobre *z*, obteniendo una expresión que ya no depende de *z*. Luego, al realizar la siguiente integral (sobre *y*), *Mathematica* sustituye el valor *z/2* como límite superior de la integración, como si fuera cualquier variable. Esta dependencia se propaga hasta la expresión final. Obsérvese que *Mathematica* no nos da ningún mensaje de error, porque desde el punto de vista del programa es perfectamente posible que alguien use la variable *z* como variable muda dentro de la integral y despues otra vez como otra variable que denota un límite superior de integración. Aunque esto es *posible,* es mucho más *probable* que se trate de un error. Debido a este riesgo, es recomendable inspeccionar cuidadosamente tanto los límites de integración cuanto el orden de integración en las integrales múltiples.

Integración numérica

Mathematica tiene la capacidad de aproximar numéricamente el valor de integrales definidas, tanto unidimensionales como en múltiples dimensiones (es decir, para una y más variables de integración). El comando apropiado es **`NIntegrate`** y si sintaxis es prácticamente igual a la usada para integrales definidas con **`Integrate`**.

`NIntegrate[`*f, {variable,LimInf,LimSup}* **`]`**
Evalúa numericamente la integral de *f* con respecto a *variable*, entre los límites especificados.

Es indispensable que la función *f(x)* devuelva valores numéricos cuando se sustituye *x* por cualquier valor en el intervalo de integración. Un error común es que *f(x)* dependa de un parámetro al cual no se le ha asignado un valor numérico.

Por ejemplo, la función

```
In[62]:= g[x_] := Log[Sin[1 - x^2]]
```

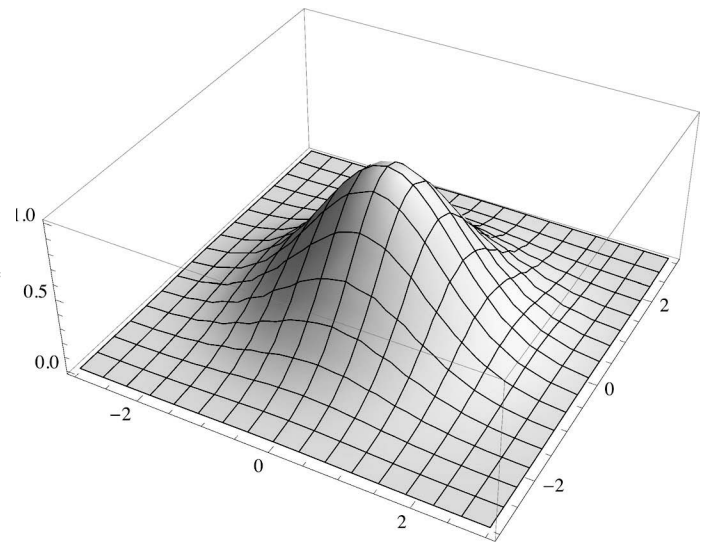
no tiene una integral primitiva conocida en términos de funciones elementales, de modo que si queremos integrarla lo tenemos que hacer numéricamente. Usando el comando **NIntegrate** podemos evalúar el valor numérico de $\int_0^1 g(x)\,dx$

```
In[63]:= NIntegrate[ g[x], {x, 0, 1}]
Out[63]= -0.704981
```

NIntegrate también puede evaluar integrales múltiples numéricamente. La sintaxis, una vez más, es parecida a la usada en **Integrate** para un caso análogo. Por ejemplo, la superficie siguiente corresponde a una Gaussiana en 2D.

```
In[64]:= Plot3D[Exp[-(x^2 + y^2)/2], {x, -3, 3}, {y, -3, 3}]
```

Out[64]=



El volumen bajo la superficie mostrada se puede evaluar numéricamente como una integral múltiple,

```
In[65]:= NIntegrate[Exp[-(x^2 + y^2)/2], {x, -3, 3}, {y, -3, 3}]
Out[65]= 6.2493
```

El proceso de integración numérica puede ser lento, por lo que en ocasiones es conveniente medir el tiempo de ejecución con el comando **Timing** y estimar si el costo computacional es aceptable.

```
In[66]:= Timing[
          NIntegrate[Exp[-(x^2 + y^2)/2], {x, -3, 3}, {y, -3, 3}]
         ]
Out[66]= {0.006114, 6.2493}
```

Dado que todos los métodos de integración numérica son aproximados, al Evaluar integrales numéricas debemos buscar tener una estimación del error cometido y de la eficiencia del método usado. Aunque *Mathematica* utiliza algoritmos para adaptar el método usado al caso particular de integral que está realizando, no debemos dar por sentado que los resultados arrojados están libre de error.
Una discusión sobre las opciones que permiten controlar detalladamente el método numérico usado por **NIntegrate** requiere conocimientos básicos de análisis numérico, por lo que simplemente las enlistamos aquí y referimos al lector al manual en línea de *Mathematica*.

```
AccurayGoal
MaxPoints
MaxRecursion
Method
MinRecursion
PrecisionGoal
WorkingPrecision
```

Ejercicios

1. Calcule la integral de $y=sen(x^2)$.

2. Haga una tabla de la integral indefinida de $y=sen^n\, x$ para valores enteros de n desde 1 a 10.

3. Obtenga la integral de *y=sen(sen(x))* de 0 a 1 con veinte cifras significativas.

4. Calcule la integral de $y = x^n$ de 1 a infinito para $n > 1$.

5. Calcule el área contenida por las curvas $f(x) = 1 - x^2$ y $g(x) = x^4 - 3\, x^2$.

6. Obtener el volumen de una esfera de radio R utilizando una integral.

7. Calcular al área entre la parábola $y = 9 - x^2$ y el eje x. También calcular la altura (y) a la cual se tienen 2/3 del área total.

8. Calcular el volumen para un tazón esférico si tiene la ecuación $x^2 + y^2 + z^2 = a$ y $z \leq 0$.

9. Calcule la integral *1/(1+x)* de menos infinito a infinito en forma exacta y luego numéricamente.

10. Calcular en forma exacta y numéricamente el área de un cono que se abre hasta una circunferencia con radio igual a 1.

Capítulo 10

Ecuaciones Diferenciales Ordinarias

Una ecuación que contiene las derivadas de una o más variables dependientes con respecto a una o más variables independientes, es una ecuación diferencial. A diferencia de las ecuaciones algebraicas usuales, en una ecuación diferencial la incógnita es una función, en vez de ser un número particular.

Las ecuaciones diferenciales se clasifican de acuerdo con su tipo, orden y linealidad. Según el tipo: si una ecuación sólo contiene derivadas ordinarias de una o más variables dependientes con respecto a una sola variable independiente, entonces se dice que es una ecuación diferencial ordinaria. Una ecuación que contiene las derivadas parciales de una o más variables dependientes, respecto de dos a más variables independientes, se llama ecuación en derivadas parciales. Según el orden: el orden de una ecuación diferencial es el de la derivada de mayor orden en la ecuación. Según la linealidad: se dice que una ecuación diferencial de la forma

$$y^{(n)} = f\left(x, y, y', \ldots, y^{(n-1)}\right)$$

es lineal cuando f es una función lineal de $y, y', \ldots, y^{(n-1)}$. Cuando una función ϕ, definida en algún intervalo I, se sustituye en una ecuación diferencial y transforma esa ecuación en una identidad, se dice que ϕ es una solución de la ecuación en el intervalo.

En este capítulo expondremos cómo *Mathematica* puede ayudarnos a encontrar ϕ para ecuaciones diferenciales ordinarias, tanto analítica como numéricamente.

Solución analítica

En *Mathematica* se utiliza el comando **DSolve** para resolver ecuaciones diferenciales. Al igual que en una ecuación algebraica la igualdad se denota con un doble igual, **==**. Este comando tiene la siguiente estructura,

DSolve[*ecuación*, **y[x]**, **x]**	Da la solución general, **y[x]**, a la ecuación diferencial *ecuación*, cuya variable independiente es **x**.

A continuación mostraremos como utilizar el comando para resolver la ecuación diferencial $\frac{dx}{dy} = x + y$.

```
In[1]:= DSolve[y'[x] == x + y[x], y[x], x]
Out[1]= {{y[x] → -1 - x + ℯ^x C[1]}}
```

En el siguiente ejemplo se muestra una forma alternativa en la que se puede escribir la ecuación a resolver,

```
In[2]:= DSolve[D[y[x], {x, 1}] == x + y[x], y[x], x]
Out[2]= {{y[x] → -1 - x + ℯ^x C[1]}}
```

Esta última forma puede ser muy útil cuando se quiere resolver una ecuación diferencial de orden más grande que uno. Por ejemplo si se quiere resolver $x^2 \frac{d^2y}{dx^2} + x \frac{dy}{dx} + (x^2 - 4) y = 0$, que es una ecuación tipo Bessel, la notación anterior será útil,

```
In[3]:= DSolve[
          x^2 D[y[x], {x, 2}] + x D[y[x], {x, 1}] + (x^2 - 4) y[x] == 0, y[x], x]
Out[3]= {{y[x] → BesselJ[2, x] C[1] + BesselY[2, x] C[2]}}
```

La solución devuelta por *Mathematica* está expresada en términos de las funciones de Bessel de primera y segunda especie.

Si se conoce algún valor de y junto con cualesquiera de sus derivadas en algún punto, la labor de encontrar una solución se conoce como problema con valores iniciales o de frontera. *Mathematica* puede incluir estas condiciones y darnos una solución única al problema. Por ejemplo, si se quiere resolver la ecuación diferencial $\frac{dx}{dy} = x + y$ con la siguiente condición inicial $y(0) = 2$, se hace del siguiente modo,

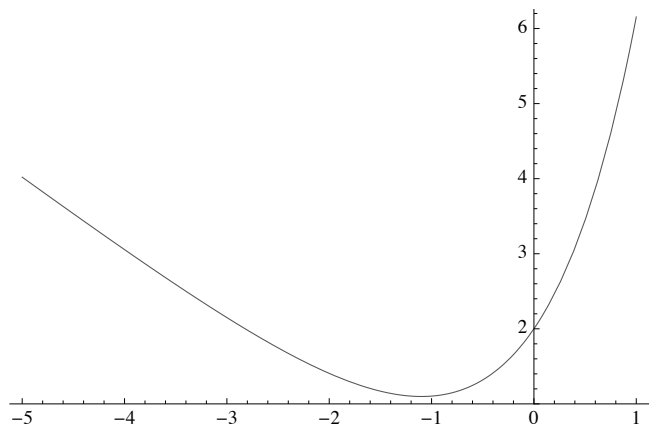
```
In[4]:= DSolve[{y'[x] == x + y[x], y[0] == 2}, y[x], x]
Out[4]= {{y[x] → -1 + 3 ℯ^x - x}}
```

Cuando las ecuaciones diferenciales incluyen derivadas de orden superior, o para sistemas de ecuaciones diferenciales, es necesario dar más de una condición inicial, ésto se consigue agregando a la lista del primer argumento todas las condiciones iniciales necesarias.

En ocaciones una vez que es encontrada la solución se quiere manipularla, por ejemplo, en ocaciones es necesario graficarla, para ello es conveniente asignarle una variable a la solución,

```
In[5]:= sol = DSolve[{y'[x] == x + y[x], y[0] == 2}, y[x], x]
Out[5]= {{y[x] → -1 + 3 ℯ^x - x}}
In[6]:= Plot[y[x] /. sol, {x, -5, 1}]
```

Out[6]=



A continuación daremos un ejemplo en el que además de la condición inicial se tiene un valor a la frontera. Supongamos que se quiere obtener la solución de la ecuación diferencial $y'' + y = 20\cos(x)$, con $y(0) = 0$ y $y(\pi/2) = 1$. Para resolver este problema sólo es necesario introducir el par de condiciones en el comando **DSolve**,

```
In[7]:= DSolve[{y''[x] + y[x] == 20 Cos[x], y[0] == 0, y[Pi / 2] == 1}, y[x], x]
Out[7]= {{y[x] → -10 Cos[x] + 10 Cos[x]^3 +
        Sin[x] - 5 π Sin[x] + 10 x Sin[x] + 5 Sin[x] Sin[2 x]}}
```

Para simplificar el resultado es conveniente utilizar el comando **Simplify**

```
In[8]:= Simplify[
         DSolve[{y''[x] + y[x] == 20 Cos[x], y[0] == 0, y[Pi / 2] == 1}, y[x], x]]
Out[8]= {{y[x] → (1 - 5 π + 10 x) Sin[x]}}
```

Si se quiere encontrar una familia de soluciones a una ecuación diferencial, variando por ejemplo, una constante, con *Mathematica* esta tarea puede ser simple. Por ejemplo, encontrar las solución a la ecuación diferencial $\frac{dx}{dy} = 2x + y$, con la condición inicial y(0)=k, en el intervalo de -3 a 3 con incremento en una unidad, se puede hacer de la siguiente manera,

```
In[9]:= Table[DSolve[{y'[x] == 2 x + y[x], y[0] == k}, y[x], x], {k, -3, 3, 1}]
Out[9]= {{{y[x] → -2 - ℯ^x - 2 x}}, {{y[x] → -2 (1 + x)}}, {{y[x] → -2 + ℯ^x - 2 x}},
         {{y[x] → 2 (-1 + ℯ^x - x)}}, {{y[x] → -2 + 3 ℯ^x - 2 x}},
         {{y[x] → 2 (-1 + 2 ℯ^x - x)}}, {{y[x] → -2 + 5 ℯ^x - 2 x}}}
```

Las ecuaciones diferenciales ordinarias simultáneas aparecen en problemas que tienen relación con varias variables dependientes que son función de la misma variable independiente. Con el comando **DSolve** también es posible resolver un sistema de ecuaciones diferenciales simultáneas poniendo éstas entre llaves. A continuación discutiremos un problema simple en el cual se pondrá de manifiesto la necesidad del uso de este comando.

DSolve[{*ec1, ec2, ...***},{***y_1**[x],**y_2**[x], ...}, x]**
Da la solución general al sistema de ecuaciones diferenciales de variable independiente **x**.

En la más simple de las reacciones entre una especie A y una B con coeficientes de volocidad k_1 y k_2 se puede escribir un sistema de ecuaciones diferenciales simultáneas que describan cómo cambia la concentración de cada una de las especies en el tiempo. En este caso las constante de velocidad k_1 nos indican la probabilidad con la que una molécula de la especie A se transforma en una de la especie B y viceversa. En esta reacción el cambio de cada especie en el tiempo está dada por el sistema $\frac{dA}{dt} = -k_1 A + k_2 B$, $\frac{dB}{dt} = -k_2 B + k_1 A$. Las condiciones iniciales las podemos escribir en general de la siguiente manera $A(0) = a_0$ y $B(0) = b_0$, donde a_0 y b_0 representan las concentraciones iniciales de cada una de las especies. Este sistemas de ecuaciones lo podemos resolver facilmente con el comando **DSolve**,

```
In[10]:= Simplify[DSolve[{A'[t] == -k1 A[t] + k2 B[t],
    B'[t] == k1 A[t] - k2 B[t], A[0] == a0, B[0] == b0}, {A[t], B[t]}, t]]
```

$$\text{Out[10]= } \left\{\left\{A[t] \to \frac{\left(1 - e^{-t\,(k_1+k_2)}\right)\, b_0\, k_2 + a_0\, \left(e^{-t\,(k_1+k_2)}\, k_1 + k_2\right)}{k_1 + k_2},\right.\right.$$
$$\left.\left. B[t] \to \frac{\left(1 - e^{-t\,(k_1+k_2)}\right)\, a_0\, k_1 + b_0\, \left(k_1 + e^{-t\,(k_1+k_2)}\, k_2\right)}{k_1 + k_2}\right\}\right\}$$

El comando **Simplify** se ha utilizado para que *Mathematica* nos de una solución más simple.

Proyecto. Tiro parabólico

En el siguiente ejemplo encontraremos la posición de una partícula disparada por un cañon (que se encuentra a una altura h del piso haciendo un ángulo θ con la horizontal) que se encuentra sujeta al campo gravitacional terrestre y que por lo tanto describe un movimiento parabólico. Las ecuaciones de Newton en este caso son, $x'' = 0$ y $y'' = -g$. Además las velocidades iniciales estan dadas por $x'(0) = v_0 \cos(\theta)$ y $y'(0) = v_0\, \text{sen}(\theta)$, y su posición incial por $x = 0$ y $y = h$. La solución al problema la podemos encontrar introduciendo toda esta información en el comando **DSolve,**

```
In[11]:= DSolve[{x''[t] == 0, y''[t] == -g, x'[0] == v0 Cos[θ],
    y'[0] == v0 Sin[θ], x[0] == 0, y[0] == h}, {x[t], y[t]}, t]
```

$$\text{Out[11]= } \left\{\left\{x[t] \to t\, \text{Cos}[\theta]\, v_0,\ y[t] \to \frac{1}{2}\, \left(2\, h - g\, t^2 + 2\, t\, \text{Sin}[\theta]\, v_0\right)\right\}\right\}$$

Estas son las ecuaciones que describen al movimiento tanto en el eje X como en el Y.

Proyecto. Velocidad de escape

En este ejemplo obtendremos la velocidad inicial que requiere una masa m para escapar del campo gravitacional de la Tierra en dirección radial. Para ello tendremos que encontrar la velocidad incial que haga que la partícula nunca regrese, esto es, que la partícula siempre tenga velocidad positiva. De la segunda ley de Newton y de su ley de gravitación sabemos que la aceleración de una partícula bajo la influencia del campo gravitacional terrestre es inversamente proporcional al cuadrado de la distancia de la partícula al centro de la tierra. Si r representa la distancia de la partícula desde el centro de la tierra, R el radio de la tierra, v la velocidad de la partícula y a su aceleración, entonces sabemos que $a = \frac{\mathrm{dv}}{\mathrm{dt}} = \frac{k}{r^2}$. Si la aceleración que experimenta la masa debida a la fuerza gravitacional es $-g$, entoces el valor de la constate la podemos calculcular de esta última ecuación y está dada por, $k = -\mathrm{gR}^2$. Con el valor de la constante y utilizando la regla de la cadena tenemos que $v\frac{\mathrm{dv}}{\mathrm{dr}} = -\frac{\mathrm{gR}^2}{r^2}$. Ésta ecuación diferencial contiene la información de la velocidad de la partícula en todo momento, su solución por tal motivo es de gran utilidad. Si v_0 representa la velocidad de escape entonces al resolver la ecuación diferencial con la siguiente condición incial $v = v_0$, donde $r = R$ nos dira la velocidad de la partícula en todo momento.

```
In[12]:= Clear[v];
         Simplify[DSolve[{v[r] v'[r] == -g R^2 / r^2, v[R] == v0}, v[r], r]]

         DSolve::bvnul :
           For some branches of the general solution, the given boundary conditions lead to an
                 empty solution. >>

         DSolve::bvnul :
           For some branches of the general solution, the given boundary conditions lead to an
                 empty solution. >>

Out[12]= { }
```

La solución que estamos buscando es aquella en que la partícula siempre tiene velocidad positiva, lo cual nos lleva a escoger sólo la segunda solución. Además para que la raíz tenga solución real se tiene que cumplir que $-2\,\mathrm{gR} + v^2{}_0 \geq 0$, esta condición se cumple simpre que $v_0 \geqslant \sqrt{2\,\mathrm{gR}}$. Finalmente al introducir los valores aproximados correspondientes a la aceleración de gravedad y el radio de la Tierra encontramos la velocidad de escape,

```
In[13]:= g = 9.8

Out[13]= 9.8

In[14]:= R = 6 370 000

Out[14]= 6 370 000
```

```
In[15]:= Sqrt[2 g R]
Out[15]= 11 173.7
```

De modo que para que un objeto de masa *m* escape de la atracción de la tierra requiere al menos de una velocidad de aproximadamente 11173.7 m/s.

Proyecto. Corriente eléctrica

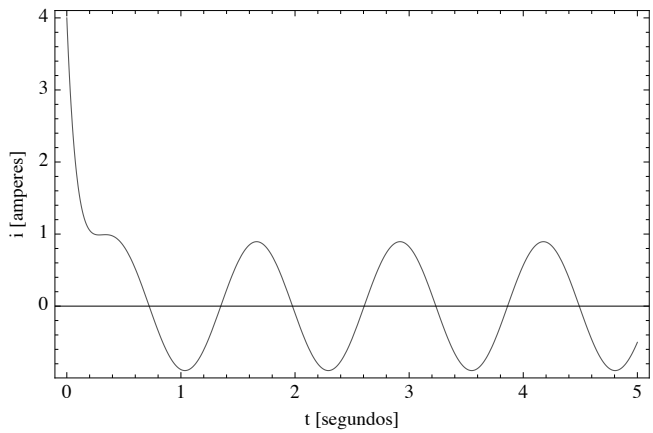
La ecuación que gobierna la cantidad de corriente I a través de un cable con inductancia y resistencia en la que se aplica una diferencia de potencial V, es $L\frac{dI}{dt} + R\,I = V$. Las unidades de V, I, L y R son volts, amperes, henries y ohms respectivamente. Supongamos que queremos calcular la intensidad de corriente a cada tiempo si se tiene una resistencia de *R=10* ohms, una inductancia *L=1* henry, *V(t)=10 sen 5t* y una corriente incial de *I(0)=4* amperes. Para ello es necesario resolver la ecuación diferencial anterior, lo cual podemos hacer con ayuda del comando **DSolve**,

```
In[16]:= L = 1;
         R = 10;
         sol = DSolve[{L i'[t] + R i[t] == 10 Sin[5 t], i[0] == 4}, i[t], t]
Out[18]= {{i[t] → -2/5 E^(-10 t) (-11 + E^(10 t) Cos[5 t] - 2 E^(10 t) Sin[5 t])}}
```

La gráfica de la solución es la siguiente.

```
In[19]:= Plot[i[t] /. sol, {t, 0, 5}, Frame → True,
          FrameLabel → {"t [segundos]", "i [amperes]"}]
```

Out[19]=



Solución numérica

Aunque algunas ecuaciones diferenciales ordinarias se pueden resolver y expresar su solución analítica en términos de funciones elementales, en general éste no es el caso. Para resolver esta clase de ecuaciones diferenciales ordinarias con *Mathematica* es necesario usar un comando que lo haga en forma aproximada y numérica. Para ello *Mathematica* cuenta con el comando **NDSolve**. En este comando además de introducir le ecuación diferencial y sus condiciones iniciales también es necesario indicar el intervalo en el que se quiere encontrar la solución. La estructura del comando es la siguiente,

NDSolve[*{ecuación, condiciones inciales}*, **y[x]**, **{xmin,xmax}]**
Da la solución numérica, a la ecuación diferencial de variable independiente **x**.

Mathematica resuelve numericamente las ecuaciones diferenciales utilizando técnicas que construyen soluciones aproximadas en un número finito de puntos. Posteriormente interpolando construye una función que pase por los puntos y por lo tanto la solución la da en términos de interpolación de funciones, que para utilizar se tiene que asignar un nombre.

Supongamos que queremos resolver la ecuación diferencial $\frac{dy}{dx} = x^2 + \sqrt{y}$ con la siguiente condición inicial, $y(0) = 1$ en el intervalo $1 \le x \le 2$. Si utilizamos el comando **DSolve** se obtiene lo siguiente,

```
In[20]:= DSolve[{y'[x] == 2 x^2 + Sqrt[y[x]], y[0] == 1}, y[x], x]
Out[20]= DSolve[{y'[x] == 2 x^2 + Sqrt[y[x]], y[0] == 1}, y[x], x]
```

Ya que *Mathematica* no puede encontrar la solución en términos de funciones elementales, es necesario resolverla numéricamente. Para ello impondremos el intervalo entre 1 y 2.

```
In[21]:= sol = NDSolve[{y'[x] == 2 x^2 + Sqrt[y[x]], y[0] == 1}, y, {x, 1, 2}]
Out[21]= {{y → InterpolatingFunction[{{1., 2.}}, <>]}}
```

Con el comando **FullForm** se pueden desplegar todos los puntos que *Mathematica* calculó. Si se quiere utilizar la solución, ya sea para calcular algún punto o simplemente para graficarlos se tiene que asignar algun nombre a la función de interpolación, digamos f,

```
In[22]:= f = sol[[1, 1, 2]]
Out[22]= InterpolatingFunction[{{1., 2.}}, <>]
```

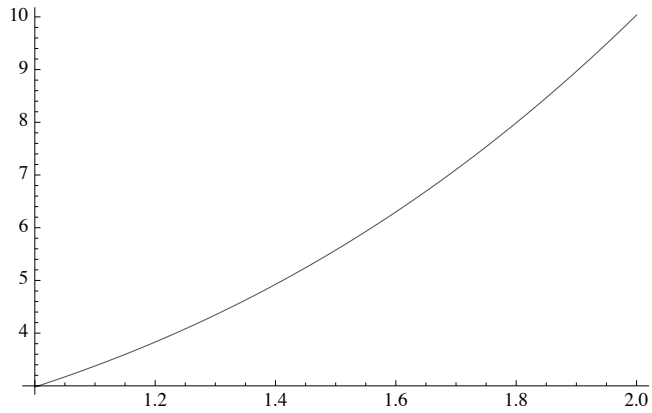
Por ejemplo si se quieire saber el valor de la solución en el punto $x = 1.5$, se obtiene de la siguiente manera,

```
In[23]:= f[1.5]
Out[23]= 5.57546
```

La función se puede graficar utilizando el comando **Plot**,

```
In[24]:= Plot[f[x], {x, 1, 2}]
```

Out[24]=



También suele ser deseable el preparar una tabla que nos muestre la solución, ésto lo hacemos con el comando **Table**,

```
In[25]:= TableForm[Table[{x, f[x]}, {x, 1, 2, 0.1}]]
```

Out[25]//TableForm=

```
1.   2.97817
1.1  3.37694
1.2  3.8313
1.3  4.346
1.4  4.92582
1.5  5.57546
1.6  6.29963
1.7  7.10299
1.8  7.9902
1.9  8.96587
2.   10.0346
```

Con este comando también es posible resolver sistemas de ecuaciones. Supongamos que queremos resolver el siguiente sistema de ecuaciones $x' = -2x^2 - y$, $y' = y - y$, sujeto a las siguientes condiciones iniciales $x(0) = 0.2$ y $y(0) = 0.1$, en el intervalo $0 \le t \le 10$. Esto se hace de la siguiente manera,

```
In[26]:= sol = NDSolve[{x'[t] == -2 x[t]^2 - y[t], y'[t] == x[t] - y[t],
            x[0] == 0.2, y[0] == 0.1}, {x[t], y[t]}, {t, 0, 10}]
Out[26]= {{x[t] → InterpolatingFunction[{{0., 10.}}, <>][t],
           y[t] → InterpolatingFunction[{{0., 10.}}, <>][t]}}
```

Proyecto. Cálculo de precisión

En este ejemplo construiremos una tabla que nos permita apreciar el grado de precisión de una solución numérica respecto con la analítica. Para ello resolveremos por los dos métodos la ecuación diferencial $y' = \frac{5}{2} + \frac{1}{2}\, y^2$ sujeta a la condición inicial $y(0) = 1$ en el intervalo $0 \le x \le 1$.

Primero calculamos la solución exacta,

```
In[27]:= ecu1 = DSolve[{y'[x] == 2.5 + 0.5 y[x]^2, y[0] == 1}, y[x], x]
Solve::ifun :
  Inverse functions are being used by Solve, so some solutions may not be found; use Reduce
      for complete solution information. >>
Out[27]= {{y[x] → 2.23607 Tan[0.5 (0.841069 + 2.23607 x)]}}

In[28]:= s1[x_] = ecu1[[1, 1, 2]]
Out[28]= 2.23607 Tan[0.5 (0.841069 + 2.23607 x)]
```

A continuación calculamos una solución numérica,

```
In[29]:= ecu2 = NDSolve[{y'[x] == 2.5 + 0.5 y[x]^2, y[0] == 1}, y[x], {x, 0, 1}]
Out[29]= {{y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]}}

In[30]:= s2[x_] = ecu2[[1, 1, 2]]
Out[30]= InterpolatingFunction[{{0., 1.}}, <>][x]
```

Finalmente comparamos las dos soluciones,

```
In[31]:= TableForm[Table[{x, s1[x], s2[x]}, {x, 0, 1, 0.1}],
          TableHeadings → {None, {"x", "analítica", "numérica"}}]
```

Out[31]//TableForm=

x	analítica	numérica
0.	1.	1.
0.1	1.31718	1.31718
0.2	1.67929	1.67929
0.3	2.10808	2.10808
0.4	2.63839	2.63839
0.5	3.33065	3.33065
0.6	4.30095	4.30095
0.7	5.80573	5.80573
0.8	8.54876	8.54876
0.9	15.4174	15.4174
1.	69.3587	69.359

Como se puede observar las dos soluciones son prácticamente iguales.

Ejercicios

1.- La velocidad de una partícula viene dada por $v = 12\,t + 6$, donde t se expresa en segundos y v en metros por segundo. a) Hacer una gráfica de v en función de t y hallar el área limitada por la curva en el intervalo de $t = 0$ s a $t = 10$ s. b) Hallar la función de posición x(t). Utilízala para calcular el desplazamiento durante el intervalo de $t = 0$ s a $t = 10$ s.

2.- La velocidad de una partícula en metros por segundo viene dada por $v = 10\,t^2 - 10$, donde t se expresa en segundos. Encontrar la función de posición.

3.- Si la aceleración de una partícula viene dada por $a = 3\,C \cdot t$, en m/s^2, encontrar la función de velocidad y de posición.

4.- Para modelar el fenómeno de la desintegración radiactiva, se supone que la tasa con que los núcleos de una sustancia se desintegran es proporcional a la cantidad de núcleos, $A(t)$, de la sustancia que queda al tiempo t, $\frac{dA}{dt} = kA$. Resolver la ecuación diferencial.

5.- Cuando se analiza la diseminación de una enfermedad contagiosa, es razonable suponer que la tasa o razón con que se difunde no sólo es proporcional a la cantidad de personas, $x(t)$, que la han adquirido en el momento t, sino también a la cantidad de sujetos, $y(t)$, que no han sido expuestos todavía al contagio. El modelo del fenómeno cumple la ecuación diferencial $\frac{\mathrm{dx}}{\mathrm{dt}} = \mathrm{kx}(n + 1 - x)$. Resolverla con la condición inicial en la que la enfermedad inicia con una persona contagiada, $x(0) = 1$.

6.- Según la ley empírica de Newton del enfriamiento, la rapidez con que se enfría un objeto es proporcional a la diferencia entre su temperatura y la del medio que le rodea, que es la temperatura ambiente, por ejemplo. Si $T(t)$ representa la temperatura del objeto al tiempo t, T_m la temperatura constante del medio que lo rodea y dT / dt es la rapidez con que se enfría el objeto, la ley de Newton del enfriamiento se traduce en el enunciado matemático $\frac{\mathrm{dT}}{\mathrm{dt}} = k(T - T_m)$. Resolver la ecuación diferencial.

7.- En ciertas circunstancias, un cuerpo que cae, de masa m, se encuentra con una resistencia del aire que es proporcional a su velocidad instantánea v. Resolver la ecuación diferencial del problema si ésta viene dada, de acuerdo a la segunda ley de Newton, por $m\,\frac{dv}{dt} = \; mg - kv$.

8.- Graficar la trayectoria de un proyectil que es disparado desde una altura de 10 metros, haciendo un ángulo con la horizontal de 25 grados, si su velocidad inicial es de 25 m/s. No tomar en cuenta la resistencia del aire. Repita el problema suponiendo que la resistencia del aire es proporcional a la velocidad en ambas direcciones. Tomar k como un entero positivo que va de 5 a 10 con incrementos de una unidad (véase el problema 7).

9.- Resolver numéricamente la siguiente ecuación diferencial $\frac{d^2y}{\mathrm{dx}^2} + y = 0$, con las siguientes condiciones iniciales $y(0)=0$, y $y'(0) = \; 1$.

10. Repita el problema 8 resolviendo las ecuaciones diferenciales numéricamente y comparar los resultados.

Capítulo 11

Programando en *Mathematica*

Diferentes estilos de programación

Un buen lenguaje de computación es aquel que permite al programador resolver rápida y correctamente el problema en el que está trabajando. En la actualidad existe una gran variedad de lenguajes de programación, cada uno de ellos ofrece ventajas para resolver problemas de cierta clase.

Mathematica provee un lenguaje de programación muy poderoso, la razón es que dicho lenguaje tiene acceso a todos los comandos propios de *Mathematica*. Esto es, tenemos oportunidad de combinar operaciones numéricas, simbólicas y gráficas dentro de un mismo ambiente.

Para ilustrar los distintos estilos de programación, este capítulo se divide en tres secciones: en la primera sección, veremos ejemplos resueltos exclusivamente mediante la programación tradicional (propio de lenguajes como Fortran o C), es decir, mediante iteraciones y llamadas a subprogramas.
En la segunda sección presentamos un ejemplo que ocurre frecuentemente en la práctica de la ingeniería y las ciencias físicas: la necesidad de ajustar una gran cantidad de series de datos, mediante una regresión lineal para cada una de ellas.
Finalmente, presentamos algunos proyectos algo más complejos, que sirvan para ilustrar más posibilidades de combinar las capacidades simbólicas, numéricas y gráficas con el estilo y estructuras de control características de *Mathematica*.

Programación tradicional

En el estilo de programacion tradicional, uno trata de dividir un problema en subproblemas más sencillos. A su vez, cada subproblema se vuelve a dividir en subproblemas cada vez más simples, hasta que se obtienen tareas que se pueden realizar directamente a partir de los elementos del lenguaje de programación.
Como primer ejemplo, nos proponemos calcular la n-ésima potencia de una matriz cuadrada **A**. En este ejemplo, podemos encontrar una solución si consideramos que calcular una potencia corresponde a realizar una multiplicación n veces, almacenando el resultado de cada multiplicación en una variable auxiliar y repitiendo esta operación n veces. Por ejemplo, si $n = 7$,

$$A^n = A\ A\ A\ A\ \ldots\ A$$

En Mathematica la multiplicación entre matrices se efectua poniendo un punto entre las matrices a multiplicar, por ejemplo

```
A.B
```

El elevar una matriz a su *n*-ésima potencia se reduce a la tarea de repetir el proceso de multiplicar la matriz por sí misma *n* veces. En la programación tradicional esto se consigue mediante las estructuras de control de iteraciones. Las más conocidas, ya que se heredan de lenguajes tradicionales como Fortran o C++, son las instrucciones **Do**, **For** y **While.** En este capítulo también mostraremos la utilidad de los comandos **If** y **With**. *Mathematica* tiene la capacidad de unir un grupo de comandos como el código de un programa en Fortran o C++, para ello se utiliza el comando **Module**, este comando es muy útil y discutiremos su uso.

A contnuación mostraremos como encontrar la potencia de una matriz utilizando cada uno de los comandos, **Do**, **For** y **While**, para ejemplificar su uso y estructura.

Do[*expresión*, **{***i, imin, imax, pasos***}]**	Evalua *expresión* para *i,* desde *imin* hasta *imax,* incrementando *i* en pasos de tamaño *pasos* cada vez.

La iteración que realiza el comando **Do** consiste en evaluar la expresión usando los valores del iterador *i*, comenzando en el valor *imin* e incrementándolo en la cantidad *pasos*, hasta llegar al valor final *imax*.

A continuación calcularemos el efecto de rotar los ejes de un sistema coordenado tres veces de la siguiente forma. Imaginemos que tomamos al sistema cordenado del eje *z* y lo rotamos 90^0 de modo tal que el eje *x* termine en el eje *y* y *y* en *-x*. Encontramos la posición final de los ejes al calcular A^3 si

$$A = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Esta tarea la podemos llevar a cabo escribiendo un código utilizando los comandos **Do**, **For** y **While**, como a continuación mostraremos.

Lo primero que tenemos que hacer es definir la matriz **A**.

```
In[7]:= A = {
           {0, 1, 0},
           {-1, 0, 0},
           {0, 0, 1}
         };
       MatrixForm[A]
```

Out[8]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

En términos de nuestro ejemplo, aplicaríamos el comando **Do** como sigue: primero, debemos inicializar una variable auxiliar para que almacene el resultado intermedio de los productos. Lo más apropiado, en este caso es definir una variable llamada **An** y hacer que tome el valor inicial de la matriz **A**,

```
In[9]:= An = A
Out[9]= {{0, 1, 0}, {-1, 0, 0}, {0, 0, 1}}

In[10]:= Do[An = A.An, {i, 2, 3}]

In[11]:= MatrixForm[An]
Out[11]//MatrixForm=
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Esta matriz nos dice que después de tres rotaciones el eje *X* esta en *-Y*, *Y* en *X*, y *Z* permanece en su lugar original.

Como el lector habrá notado iniciamos el ciclo con **i=2**. Esto es debido a que cuando se lleva cabo el primer ciclo se tiene la matriz al cuadrado, y esto es cuando **n=2**.

También se puede combinar los comandos **Do**, **For** y **While** con el comando **With**. Esto nos permite escribir un código más claro. En el ejemplo anterior en lugar de tomar el ciclo **Do** de **i=2** a **3**, lo podemos tomar de **i=2** a **i=n**, en donde se podemos especificar dentro del comando **Wiht** el valor de **n**.

With[{*x*=x_0, *y*=y_0,...**},***expresión***]**	Especifica que en *expresión*, *x* tendrá que ser cambiado por x_0, etc.

```
In[12]:= An = A
Out[12]= {{0, 1, 0}, {-1, 0, 0}, {0, 0, 1}}

In[13]:= With[{n = 3},
          Do[An = A.An, {i, 2, n}]
          ]
```

Al terminar el ciclo, la variable **An** tiene almacenada la potencia deseada,

```
In[14]:= MatrixForm[An]
Out[14]//MatrixForm=
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Una manera diferente de resolver nuestro problema sería usar el comando **For**. La sintaxis más simple de dicho comando es la siguiente,

For[*inicio, prueba, incremento, expresión***]**
Se ejecuta *inicio*, y si se cumple *prueba*, se evalúa la *expresión* y se ejecuta *incremento*.

La idea de los creadores de este comando fue la de emular el comportamiento del comando *for* del lenguaje de programación C. Por tanto, la manera en que **For** funciona en *Mathematica* es la siguiente: Primero se evalúa la expresión *inicio*, Luego se evalúa la expresión *prueba* y si el valor da **True,** se evalúa *expresión* seguida del *incremento*. Este ciclo se repite hasta que la *prueba* deje de dar **True**.

En este caso, de nuevo inicializamos la variable auxiliar (como la matriz identidad) antes del inicio del ciclo **For**.

La sintaxis del ciclo ahora es,

```
In[15]:= An = A
Out[15]= {{0, 1, 0}, {-1, 0, 0}, {0, 0, 1}}

In[16]:= With[{n = 3},
           For[i = 2,
            i ≤ n,
            ++i,
            An = A.An
           ]
          ];
```

Una vez más, el resultado queda almacenado en la variable **An**.

```
In[17]:= MatrixForm[An]
Out[17]//MatrixForm=
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A diferencia de **Do**, **For** permite incorporar la inicialización de variables dentro del propio comando. La forma de hacer esto es separar mediante puntos y comas las distintas instrucciones de inicialización y separar este bloque de instrucciones de los demás argumentos de **For** mediante una coma,

```
In[18]:= With[{n = 3},
           For[i = 2; An = A,  (* instrucciones de inicialización,
            separadas por punto y coma *)
            i ≤ n,           (* prueba *)
            ++i,             (* incremento de la variable de control *)
            An = A.An]
          ];
In[19]:= MatrixForm[An]
Out[19]//MatrixForm=
```

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

El cuerpo de la iteración también puede constar de múltiples instrucciones, siempre y cuando estén separadas mediante punto y coma.

El comando **For** es más flexible que el comando **Do**, puesto que podemos ejecutar en la expresión de *inicio* operaciones que no son las mismas que las que se ejecutan en el cuerpo de la iteración. Además, la prueba puede ser cualquier expresión que dependa del iterador (es decir, la variable de control), y no necesariamente limitarse a que el iterador tome cierto valor.

Por último, cuando uno no sabe de antemano cuántas iteraciones se requieren para resolver un problema, en el estilo tradicional de programar se usan comandos del tipo de **While**.

While[*prueba, expresión***]**	Si se cumple *prueba,* evalúa *expresión*.

El comando **While** es más simple que **Do** o **For**. Si la prueba es **True**, se evalúa el cuerpo de la iteración y se repite el proceso hasta que la prueba resulte **False**. **While** es especialmente apropiado cuando no sabemos de antemano cuántas iteraciones se requerirán para completar la tarea.
Con este comando debemos asegurarnos de dos cosas: (a) de inicializar todas las variables auxiliares antes del comienzo de la iteración y (b) asegurarse de modificar la variable de control para que la prueba pueda llegar a ser **False** al final de las iteraciones necesarias. Así, el programa quedaría como sigue,

```
In[20]:= i = 2;
         An = A;  (* inicialización fuera del ciclo While *)

         With[{n = 3},
           While[
            i ≤ n,
            An = A.An;   (* cuerpo de la iteración,
            con múltiples instrucciones *)
            ++i
            (* incremento manual de la variable de control *)
           ]
          ];

         MatrixForm[An]
```

Out[23]//MatrixForm=

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Aunque en ocasiones es posible que no deseemos incrementar manualmente el valor de la variable de control, la mayoría de las veces sí vamos a querer hacer este incremento. De no hacerlo así, estaríamos provocando un ciclo infinito, ya que si la variable de control no cambia dentro del interior del ciclo **While**, no hay manera de que la prueba llegue a ser **False** si la primera vez resulto ser **True**.

Junto con la instrucción **If** para tomar decisiones, los comandos de iteración **Do**, **For** y **While** son el corazón de la programación tradicional. A continuación se muestra la estructura del comando **If**,

If[*prueba,* *instrucciones_verdadero,* *instrucciones_falso,* *instrucciones_otro***]**	Si *prueba* es verdadero se ejecuta *instrucciones_verdadero*; si prueba es falso, se ejecuta *instrucciones_falso*; si ninguna de las anteriores aplica se ejecuta *instrucciones_otro*.

Se presenta el siguiente ejemplo para mostrar la estructura del comando **If**,

```
In[24]:= If[2 == 2, Print["a"], Print["b"], Print["c"]]
a
In[25]:= If[2 == 3, Print["a"], Print["b"], Print["c"]]
b
In[26]:= If[7, Print["a"], Print["b"], Print["c"]]
c
```

En el primer caso ya que la afirmación es verdadera se ejecuta la primer orden, en el segundo la segunda y finalmente ya que no se tiene ninguna condición que cumplir se ejecuta la tercera orden.

El comando **Module** nos permite agrupar comandos como en un código, su estructura es la siguiente,

Module[*{condiciones iniciales,variables locales}, Código* **]**
Agrupa un conjunto de comandos.

Es evidente que en *Mathematica* cualquier grupo de tareas se puede hacer ejecutando comando por comando, pero también es cierto que a veces queremos ejecutar el grupo de comandos varias veces, para estos casos resulta muy útil el comando **Module**.

En un primer ejemplo mostraremos cómo construir un código que nos permita calcular el factorial de un número.

```
In[27]:= fact[x0_] := Module[
           {y = 1, x = x0},
           While[x > 1,
            y = x y;
            x = x - 1];
           Print["El factorial de ", x0, " es ", y];
          ]
```

```
In[28]:= fact[6]
El factorial de 6 es 720
```

El que nuestro código esté correcto lo podemos verificar fácilmente utilizando el que nos ofrece *Mathematica*.

```
In[29]:= Factorial[6]
Out[29]= 720
```

En el siguiente ejemplo escribiremos un código que nos permita saber, de una lista de calificaciones, el número de estudiantes, la calificación más baja, la más alta y el promedio. El procedimiento lo aplicaremos a un par de listas.

```
In[30]:= Resultados[datos_] := Module[
           {listaordenada, n},
           listaordenada = Sort[datos];
           n = Length[datos];
           Print["El número de estudiantes es:", n];
           Print["La calificación más alta es:", Last[listaordenada]];
           Print["La calificación más baja es:", First[listaordenada]];
           Print["El promedio es:",
            N[Sum[datos[[i]], {i, 1, n}] / n]];
         ]

In[31]:= salon1 = {6, 7, 7, 6, 5, 7, 8, 9, 10, 4, 3, 7, 8}
Out[31]= {6, 7, 7, 6, 5, 7, 8, 9, 10, 4, 3, 7, 8}

In[32]:= Resultados[salon1]
El número de estudiantes es:13
La calificación más alta es:10
La calificación más baja es:3
El promedio es:6.69231

In[33]:= salon2 = {10, 10, 9, 4, 6, 7, 5, 8, 9, 7, 8, 8, 9, 10, 5, 6, 9}
Out[33]= {10, 10, 9, 4, 6, 7, 5, 8, 9, 7, 8, 8, 9, 10, 5, 6, 9}

In[34]:= Resultados[salon2]
El número de estudiantes es:17
La calificación más alta es:10
La calificación más baja es:4
El promedio es:7.64706
```

Proyecto. Difusión en una dimensión con dos paredes absorbentes usando Programación Tradicional.

En este proyecto escribiremos el código del simulación de le proceso de difusión de una concentración que inicie en el origen de un tubo en una dimensión y que a una distancia `L` y `-L` tiene paredes absorbentes. Nuestro objetivo final será calcular el tiempo promedio en que las partículas tardan en salir del sistema. Como es bien sabido la ecuación de difusión describe la evolución de la consentración ante un gradiente inicial de ésta y en una dimensión esta dada por,

$$\frac{\partial c}{\partial t} = D\,\frac{\partial^2 c}{\partial x^2}.$$

En esta ecuación c representa la concentración y $D \equiv \Delta x^2 \,/\, 2\,\Delta t$ es el coeficiente de difusión. Δx representa el desplazamiento promedio de las partículas en un tiempo Δt. La solución de esta ecuación diferencial para un pulso c_0 de concentración que inicia en el origen esta dada por,

$$c(x,\,t) = \frac{c_0}{\sqrt{4\,\pi \mathrm{Dt}}}\,\exp\left[-x^2 \,/\, 4\,\mathrm{Dt}\right]$$

Esta distribución es gaussiana y tiene una desviación cuadrática media $\sigma^2 = 2\,\mathrm{Dt}$ y media $\mu = 0$. En el lenguaje de *Mathematica* esto puede escribirse como:

```
NormalDistribution[0, Sqrt[2 Dt]]
```

Ya que el movimiento de las partículas Brownianas es aleatorio podemos encontrar su desplazamiento mediante la generación de un número aleatorio con las propiedades de la distribución que es solución de la ecuación diferencial,

```
RandomReal[NormalDistribution[0, Sqrt[2 Dt]]]
```

Para resolver nuestro problema tenemos que iniciar una caminata para la partícula Browniana en el origen y que termine a una distancia `L` y `-L`. Cada paso sucesivo lo encontramos con

```
RandomReal[NormalDistribution[0, Sqrt[2 Dt]]]
```

Finalmente si el número de pasos totales antes de salir por las paredes absorbentes lo multiplicamos por el tiempo de paso tendremos el tiempo en que la partícula sale del sistema. Si este procesos lo hacemos para un gran número de partículas, promediando el tiempo en que las partícualas salen del sistema, podremos calcular el tiempo promedio de escape. A continuación mostramos los principales elementos que tiene que contener el código:

Los parámetros que podemos variar en nuestro sistema son el tiempo de paso `dt`, la longitud `L`, el número de partículas `nrw`, la semilla que inicia los número aleatorios `iseed`, y los valores que al iniciar nustro código tienen que tomar algunas variables como, el tiempo de duración del proceso `t`, el número de pasos `n`, y la posición `x`.

Los pasos de la partícula Browniana los generamos del siguiente modo,

```
x = x + Random[NormalDistribution[0, Sqrt[2 dt]]]
```

Utilizaremos un ciclo `Do` para llevar a cabo las iteraciones correspondientes a los desplazamientos de la partícula y un `If` para determinar si la partícula ha salido del sistema. Para indicar en el código si la partícula continúa caminando, o ha salido, utilizaremos un par de etiquetas `begin` y `end`.

Finalmente para poder agrupar todos los comandos necesarios para llevar la tarea acabo utilizaremos el comando `Module`.

```
In[35]:= tao[dt_, L_, nrw_, iseed_] := Module[{n, x, tn},
           RandomSeed[iseed];
           (* inicialización de losnúmero aleatorios *)
           tn = 0; (* inicialización del tiempo total *)
           Do[
            n = 0; (* inicialización del número de pasos *)
            x = 0; (* inicialización de la posición *)
            Label[begin]; (* Si la partícula
             sige dentro del sistema continua de nuevo *)
            x = x + Random[NormalDistribution[0, Sqrt[2 dt]]];
            (* generación de la caminata de la partícula browniana *)
            n = n + 1; (* conteo del número de pasos
             antes de salir del sistema *)
            If[
             L ≥ x ≥ -L,
             (* verificando si la partícula termina su caminata *)
             Goto[begin], (* si está dentro del
              sistema continua con la caminata *)
             Goto[end];   (* si está fuera del sistema
              termina con la caminata *)
            ];
            Label[end];
            tn = tn + n;
            (* contando el número de pasos totales al salir *)
            , {k, 1, nrw, 1}];
           Print[N[tn * dt / nrw]];
           (* Imprime el valor del tiempo promedio de salida *)
          ]
```

Ahora podemos correr nuestro código, para ello tomaremos mil partículas, `L=1`, un tiempo de paso de 10^{-2}, y una semilla de `1`. El lector tendrá que tener paciencia ya que esto le tomara un buen rato a la computadora. A un procesesador a 2.66 GHz le toma aproximadamente 40 segundos. Este tiempo se incrementa considerablemente cuando el tiempo de paso se disminuye.

```
In[36]:= tao[10^-2, 1, 10 000, 1]
0.592227
```

El valor teórico para el tiempo promedio está dado por $\tau = (1/2)\left(L^2 - x_0^2\right)$, que en nuestro caso da un valor de 0.5, lo cual representa un error de alrededor del 15%. Para mejorar este resultado es necesario tomar un tamaño de paso Δt de al menos 10^{-3}. Con este tiempo de paso el valor obtenido es de 0.485 con un error del 3%.

Proyecto. Ajuste de múltiples series de datos

En este proyecto supondremos que tenemos una gran cantidad de series de datos, cada una de ellas consiste de parejas ordenadas (x, y), donde x representa la variable independiente y y la variable dependiente. Queremos obtener un ajuste de regresión por mínimos cuadrados para cada una de las series, obtener los coeficientes de cada regresión y graficar las curvas recién ajustadas junto con los datos correspondientes, para juzgar visualmente qué tan bueno es el ajuste.
Una estrategia para programar la solución es resolver el problema para una sola serie, generando un subprograma especialmente para un solo caso, y después construir una iteración que aplique ese subprograma tantas veces como series haya. A continuación ponemos en marcha esta estrategia.

Resolviendo un caso particular del problema

Primero, debemos leer los datos almacenados en un archivo, por lo que fijamos el directorio de trabajo de *Mathematica* en el directorio que contiene todos nuestros datos. ***Para trabajar con estos archivos es necesario primero generarlos en el ápendice que se encuentrá al final del capítulo***.

```
In[37]:= dir = Directory[];
         SetDirectory[dir]
Out[38]= /Users/leo
```

Para leer los datos dentro del primer archivo, usamos el comando

```
In[39]:= datos = ReadList["serie1.dat", Number, RecordLists → True];
```

ReadList[*nombre_archivo, tipo, opciones***]**	Lee una lista de datos (del tipo dado) desde un archivo.

Supongamos que sabemos que la función que debemos ajustar es de la forma $y = m\,x + b$. En este caso, para obtener el ajuste podemos usar el comando

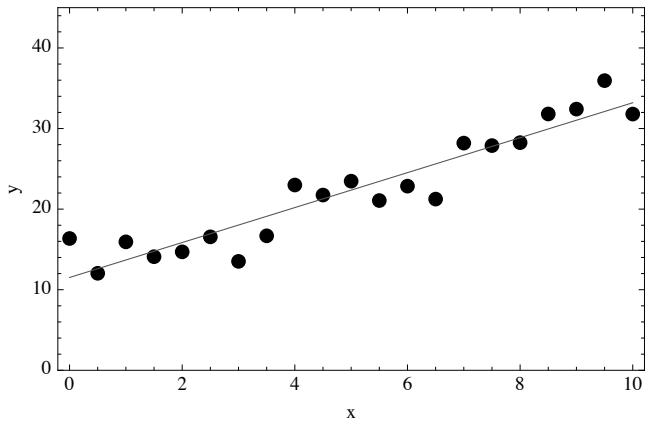
Fit[*datos, funciones, variable independiente***]**	Ajusta una combinación lineal de las funciones dadas a los datos, usando la variable independiente señalada.

```
In[40]:= ajuste[x_] = Fit[datos, {1, x}, x]
Out[40]= 11.5186 + 2.16813 x
```

En este caso asignamos a la ecuación que nos da *Mathematica*, como el mejor ajuste a la función **ajuste[x_]**
Ahora graficaremos la curva ajustada junto con los datos, en el intervalo [0,10]. Una manera de conseguir esto es:

```
In[41]:= grafica = Plot[ajuste[x], {x, 0, 10},
           Frame → True, FrameLabel → {"x", "y"},
           PlotRange → {All, {0, 45}},
           Axes → None, Prolog → {PointSize[0.025], Map[Point, datos]}]
```

Out[41]=



En esta gráfica se han usado opciones del comando **Plot** tales como **Frame**, **FrameLabel**, **Prolog**; todas ellas se discuten en el capítulo 6 dedicado a las gráficas. Por el momento, simplemente queremos destacar que es posible usar a *Mathematica* para generar un ajuste y visualizarlo al mismo tiempo.

Para extraer los coeficientes de un ajuste usamos el comando **CoefficientList**; por ejemplo,

CoefficientList[*expresión, variable***]**	Da una lista de los coeficientes de potencias sucesivas de la variable en la expresión dada.

```
In[42]:= CoefficientList[ajuste[x], x]
Out[42]= {11.5186, 2.16813}
```

En nuestro ejemplo, podemos identificar cuatro tareas:

1. Leer datos desde el archivo correspondiente,
2. Ajustar los datos con una función lineal,
3. Graficar la curva de ajuste con los datos,
4. Extraer los coeficientes del ajuste.

Como hemos visto, cada una de estas tareas se puede realizar directamente mediante un comando de *Mathematica*, de manera que no necesitamos considerar subprogramas adicionales. (Compárese esta situación con la que tendríamos en un lenguaje tradicional, tipo Fortran o C, en donde para realizar cada una de las tareas 1 a 4 necesitaríamos suministrar subprogramas adicionales.) Continuando con nuestro ejemplo, vamos a construir subrutinas que realicen las tareas 1 a 3 (la cuarta tarea no requiere un subprograma.)

LeerDatos

Básicamente, esta función debe tomar como argumento el nombre del archivo que se va a leer, y debe devolver una lista que contenga los datos leídos. El código necesario es,

```
In[43]:= LeerDatos[archivo_] := ReadList[archivo,
           Number, RecordLists → True]
```

AjustarDatos

Esta función debe tomar como argumento una lista de pares ordenados (x, y) y la variable independiente, para devolver un polinomio ($m\,x + b$),

```
In[44]:= AjustarDatos[datos_, x_] := Fit[datos, {1, x}, x]
```

Graficar Ajuste y Datos

La función debe tomar tres argumentos (el polinomio ajustado, una lista con la variable de ajuste y sus valores mínimo y máximo, así como los datos originales), y devolverá un objeto del tipo **Graphics**

```
In[45]:= GraficarAjusteYDatos[polinomio_, {x_, xmin_, xmax_}, datos_] :=
          Plot[polinomio, {x, xmin, xmax},
           Frame → True, Axes → None,
           FrameLabel → {ToString[x], "y[" <> ToString[x] <> "]"},
           Prolog → {PointSize[0.012], Map[Point, datos]}
          ]
```

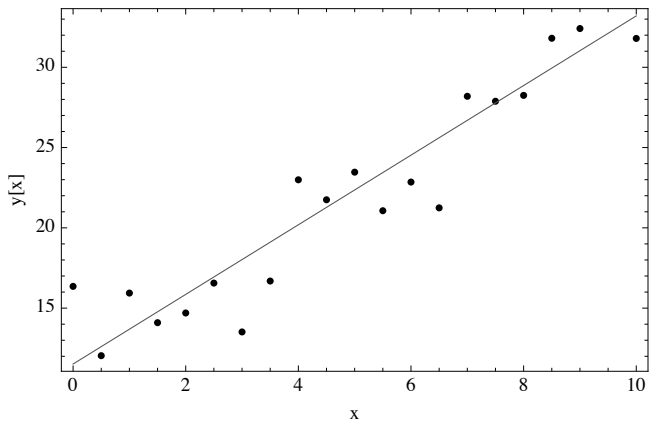
Programa completo para una sola serie

Ahora estamos en condiciones de ensamblar todas las funciones en un solo programa que resuelva el problema completo para cualquier serie. Para asegurarse que no vamos a tener colisión entre las variables que usamos en el programa y las variables que hayamos usado con anterioridad en la sesión, vamos a utilizar el comando **Module**.

```
In[46]:= AjustarSerie[archivo_, {x_, xmin_, xmax_}] :=
          Module[{datos, y, grafica},
           datos = LeerDatos[archivo];
           y = AjustarDatos[datos, x];
           grafica = GraficarAjusteYDatos[y, {x, xmin, xmax}, datos];
           Print[grafica];
           CoefficientList[y, x]
          ]
```

Veamos como funciona:

```
In[47]:= AjustarSerie["serie1.dat", {x, 0, 10}]
```




```
Out[47]= {11.5186, 2.16813}
```

El primer número de la lista, que se depliega al final, corresponde a la ordenada al origen *b* y el segundo corresponde a *m*, es decir a la pendiente.

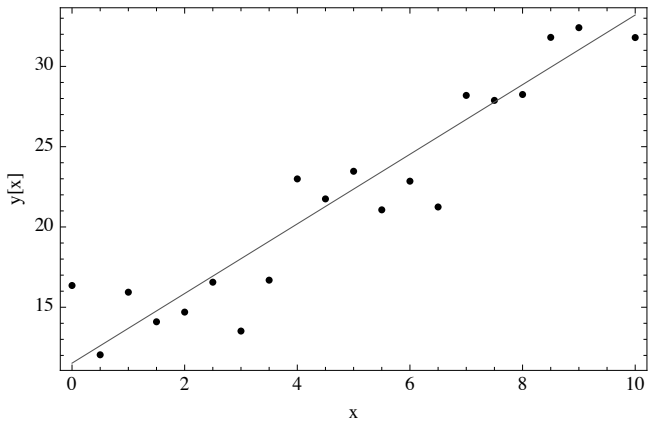
Ahora que tenemos un programa que resuelve el problema de ajustar y presentar la curva de ajuste para una serie de datos, ilustraremos cómo se resolvería el problema de aplicar este programa a un gran número de series diferentes. Pongamos, por ejemplo, que tenemos 100 series de datos con los nombres "serie1.dat", "serie2.dat", ..., "serie100.dat". En el estilo tradicional, para aplicar el programa sobre cada una de las series realizamos una *iteración*, es decir, construimos un ciclo que se repite de manera secuencial tantas veces como series de datos queremos procesar. En *Mathematica* podemos elegir entre varios comandos para ejecutar estos ciclos. Repetimos, entre los más importantes tenemos: **`Do`**, **`For`** y **`While`**.

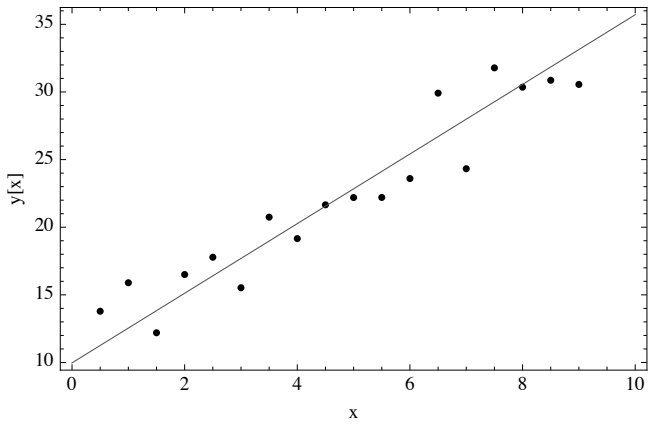
En términos de nuestro ejemplo, aplicaríamos el comando **`Do`** como sigue:

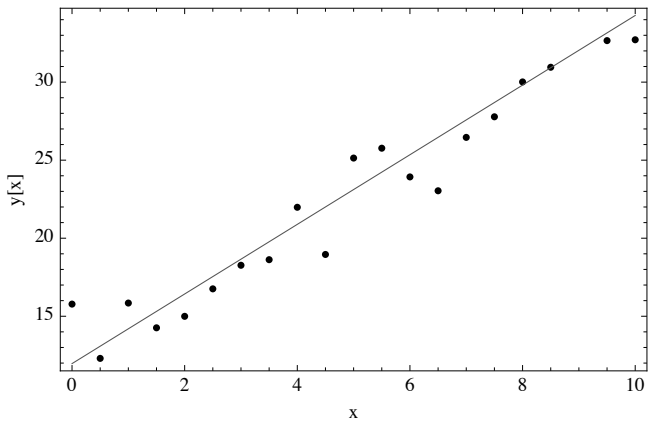
```
In[48]:= Do[
          AjustarSerie["serie" <> ToString[i] <> ".dat"],
          {i, 1, 100}
         ]
```

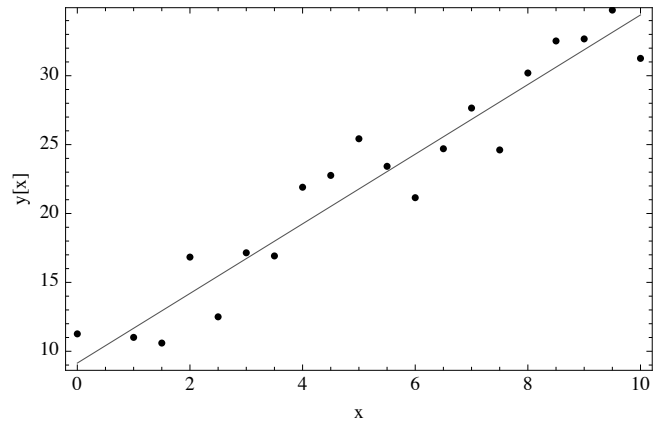
Una manera diferente de resolver nuestro problema sería usar el comando **For**.

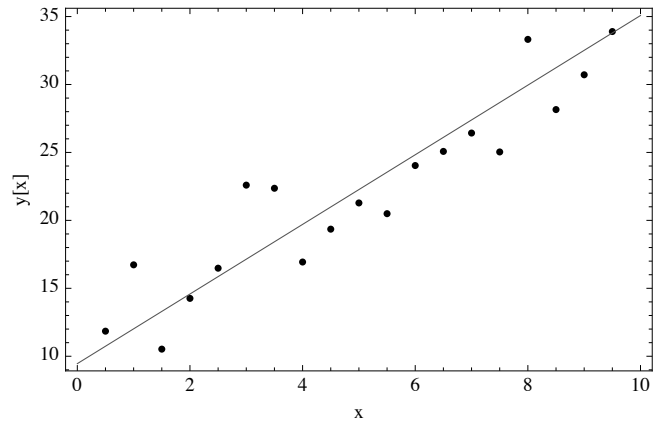
```
In[49]:= For[ i = 1, i ≤ 10, i++ ,
           AjustarSerie["serie" <> ToString[i] <> ".dat", {x, 0, 10}]
         ]
```

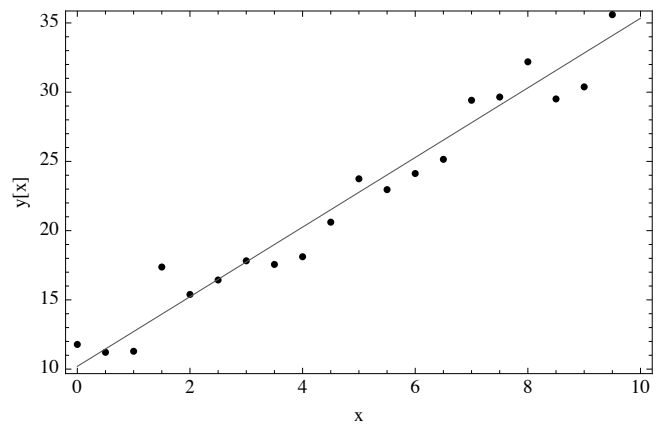


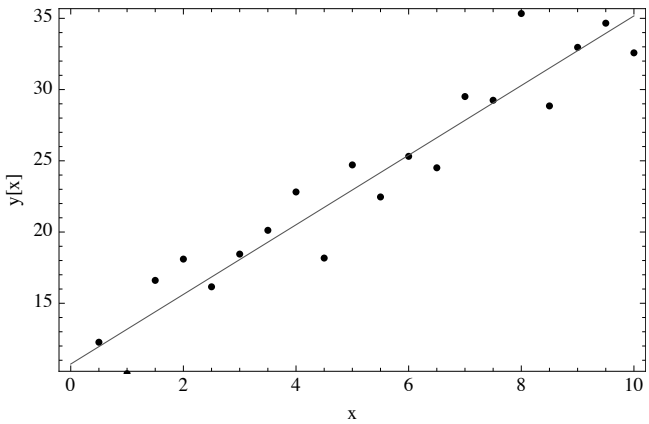


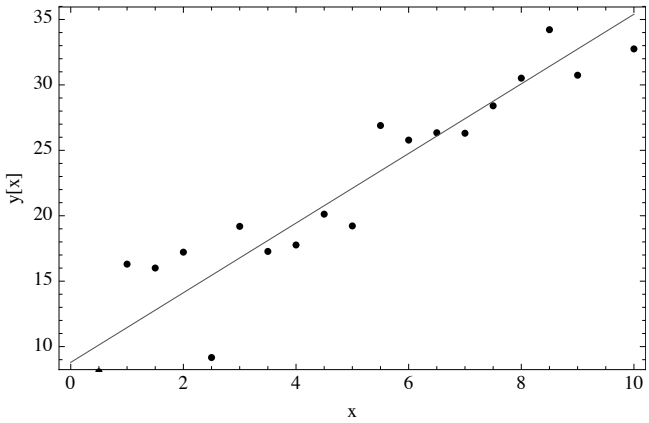


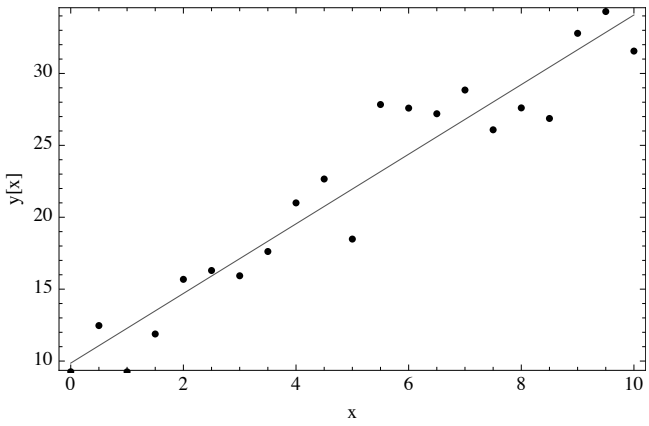

y[x]
x
10
15
20
25
30
0
2
4
6
8
10

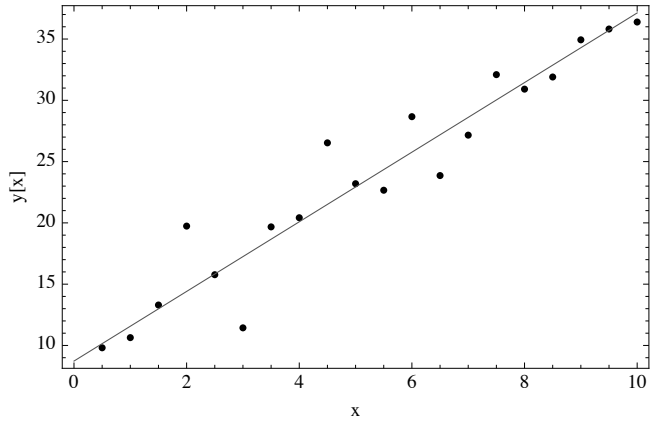

y[x]
x
10
15
20
25
30
35
0
2
4
6
8
10


y[x]
x
10
15
20
25
30
35
0
2
4
6
8
10


y[x]
x
0
2
4
6
8
10
15
20
25
30
35

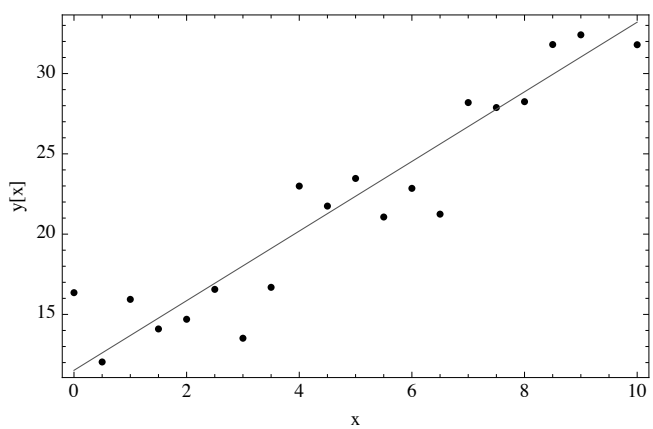

y[x]
x
0
2
4
6
8
10
10
15
20
25
30
35

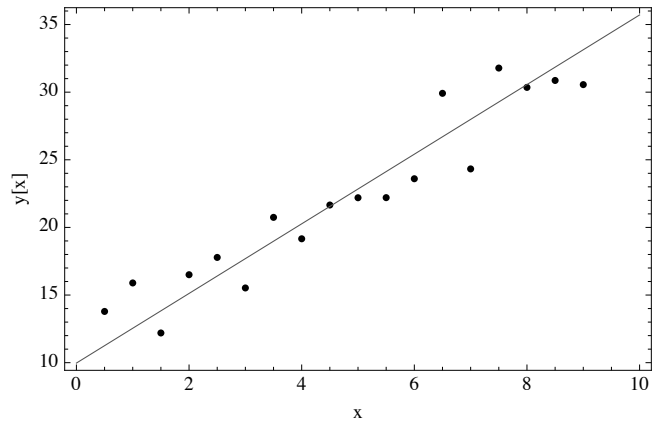

y[x]
x
0
2
4
6
8
10
10
15
20
25
30

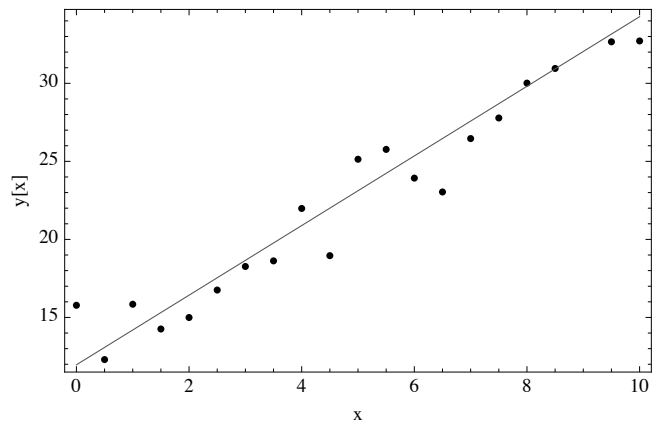


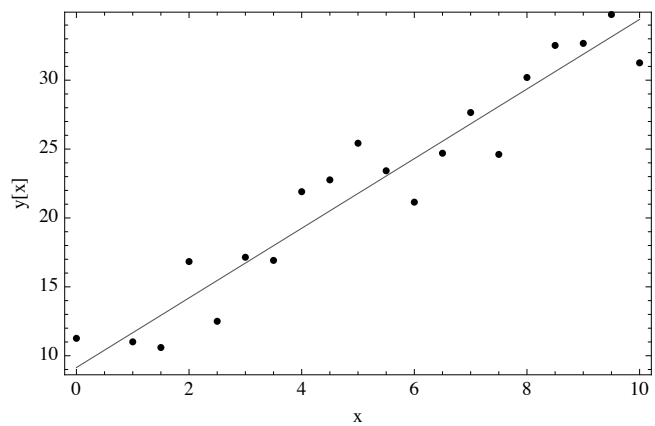
En nuestro ejemplo, como sabemos de antemano que tenemos 100 series, el comando **`While`** no es adecuado, pero para mostrar cómo funciona la iteración **`While`** supondremos que no tenemos idea de cuantos archivos de datos hay. En ese caso, una estrategia podría ser verificar secuencialmente si cada archivo está presente (usando el comando **`FileType`**), comenzando por el archivo *serie1.dat*. El programa deberá detenerse la primera vez que fallemos en localizar un archivo. Antes de empezar, debemos inicializar el iterador a 1, y en el cuerpo de la iteración debemos asegurarnos de incrementarlo en cada paso:

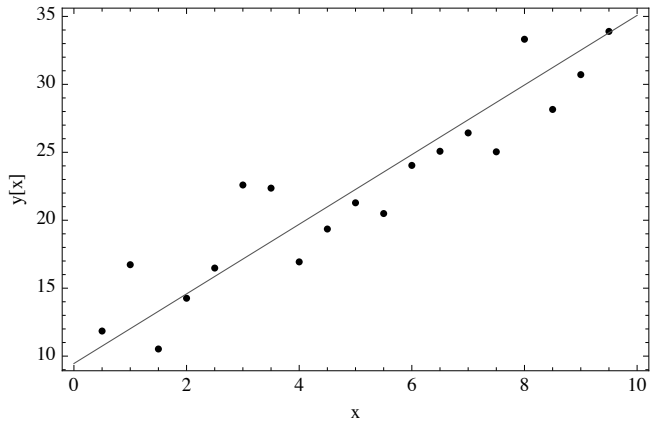
```
In[50]:= (* inicializar *)
         i = 1;

         While[

          (* prueba *)
          FileType[ estaSerie = "serie" <> ToString[i] <> ".dat"] == File,

          (* cuerpo *)
          i++;
          resultado = AjustarSerie[estaSerie, {x, 0, 10}]

         ]
```

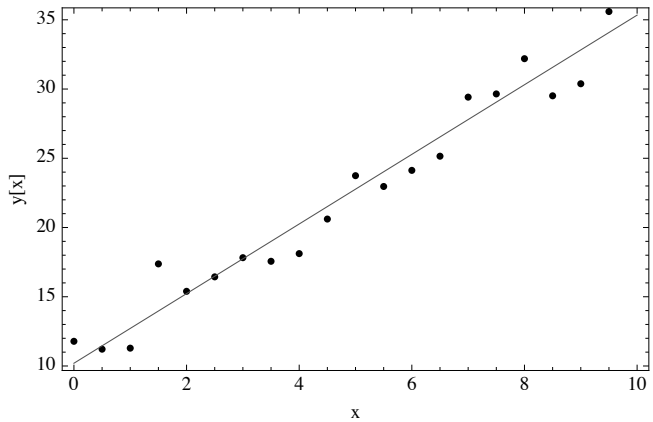


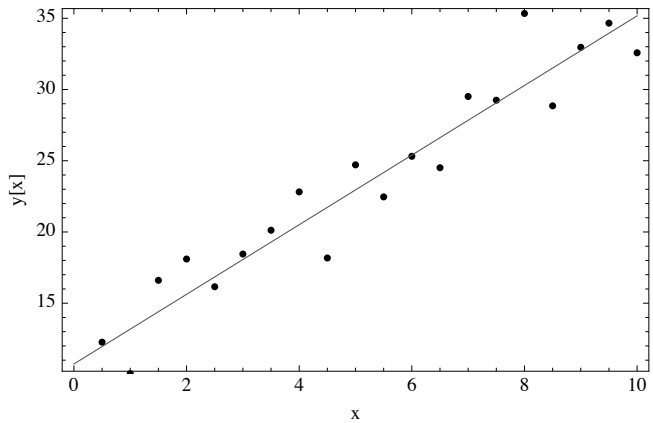

y[x]
x
35
30
25
20
15
10
0
2
4
6
8
10

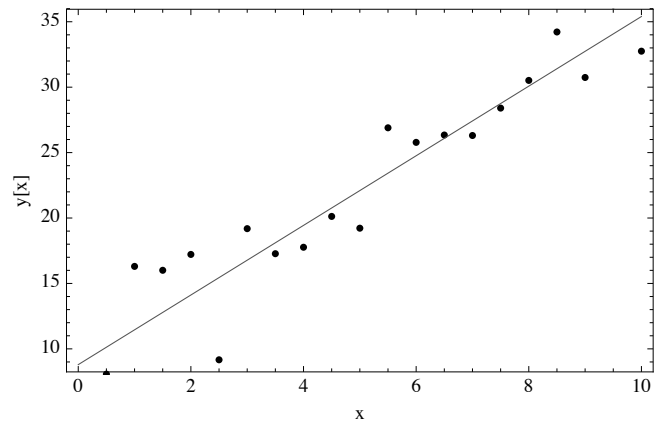

y[x]
x
30
25
20
15
0
2
4
6
8
10

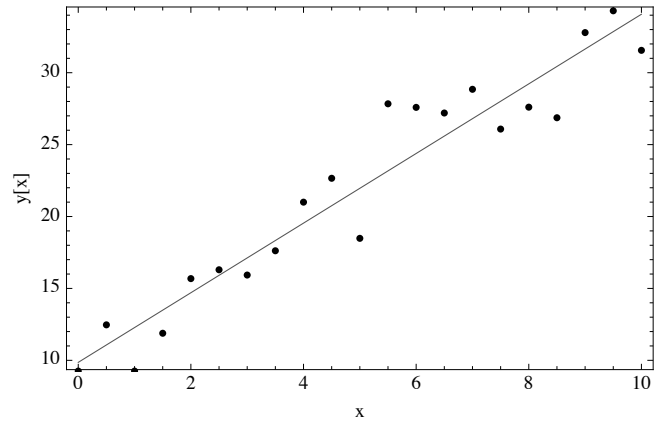

y[x]
x
30
25
20
15
10
0
2
4
6
8
10

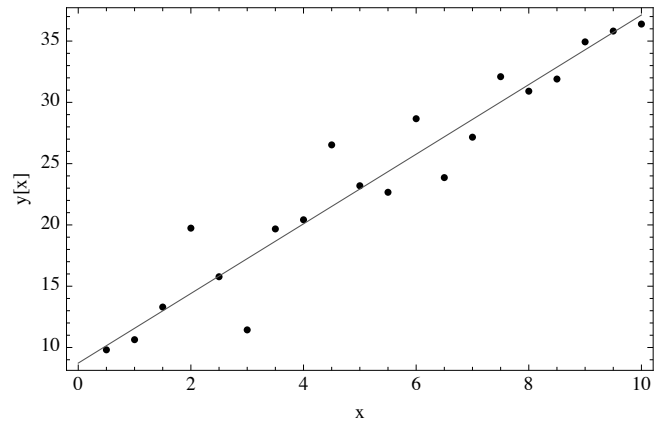

y[x]
x
35
30
25
20
15
10
0
2
4
6
8
10


y[x]
x
35
30
25
20
15
10
0
2
4
6
8
10


y[x]
x
35
30
25
20
15
0
2
4
6
8
10


y[x]
x
35
30
25
20
15
10
0
2
4
6
8
10


y[x]
x
30
25
20
15
10
0
2
4
6
8
10


y[x]
x
35
30
25
20
15
10
0
2
4
6
8
10

Programación estilo *Mathematica*

Aunque la programación tradicional (basada en **If** para la toma de decisiones y en **Do**, **For** y **While** para las iteraciones) es suficiente en principio para programar cualquier computadora para que ponga en práctica nuestros algoritmos, en la práctica el lenguaje de *Mathematica* nos da otras formas de expresar esos mismos algoritmos en una forma más *tersa*.
Por ejemplo, al definir la función **f[x_]:= x+1** en el fondo estamos escribiendo un pequeño programa (o subrutina) para *Mathematica*, sólo que la sintaxis es tan próxima a la notación matemática que podemos hacer la codificación casi sin ningún esfuerzo. Nos encontraremos con esta naturalidad en muchos otros casos. En consecuencia, el estilo de programación de *Mathematica* contiene comandos que simplifican la codificación.

Proyecto. Generación de números aleatorios enteros y movimiento Browniano

Como ya hemos visto en ciertas aplicaciones es necesario contar con una sucesión de números aleatorios. Un ejemplo es en las simulaciones de movimiento Browniano, donde se supone que una partícula microscópica experimenta colisiones aleatorias con las moléculas del disolvente que la contiene.

Tomaremos un modelo sencillo suponiendo que el movimiento browniano se da en una dimensión, donde cada desplazamiento sucesivo tiene una magnitud $|\Delta x|=1$, con probabilidad $p = 1/2$. Si nos preguntamos, tras haber dado n pasos, cúal es la distribución $P(x \mid n)$, el valor medio y la varianza de la posición x, es una tarea que podemos resolver en *Mathematica*. Este modelo reproducirá todo lo que ya supusimos en nuestro estudio anterior de movimiento Browniano.

Primero queremos un programa que genere una serie de números aleatorios con la misma probabilidad emulando el movimiento de la partícula, saltos a la izquierda o derecha a partir del punto en el que se encuentra. Una manera de hacerlo es usar el comando **RandomInteger** para generar enteros aleatorios. En general, si queremos producir números enteros distribuidos uniformemente en el intervalo $[a, b]$, debemos usar el comando **RandomInteger[{a,b}]**. En nuestro caso, deseamos producir tan solo el valor -1 o el valor 1 (sin pasar por cero) para decidir en cada caso si la partícula se mueve a la izquierda o a la derecha. Una forma de hacer esto es evaluar

```
2*Random[Integer,{0,1}]-1
```

cuando el número aleatorio es cero, la expresión vale -1, pero si el número aleatorio es uno, entonces (al sumar 2 a -1) obtenemos el valor +1.

```
In[52]:= RandomSteps[n_] := Table[ 2 RandomInteger[{0, 1}] - 1, {i, 1, n}]
```

Por ejemplo, una serie de 10 pasos aleatorios se ve así,

```
In[53]:= RandomSteps[10]
Out[53]= {-1, -1, -1, -1, -1, -1, 1, -1, 1, -1}
```

Para obtener la posición final, sólo tenemos que modificar la lista de pasos, usando el comando **Apply**,

```
In[54]:= xRandom[n_] := Apply[ Plus, RandomSteps[n]]
```

Ahora generamos una serie de m caminatas al azar, todas de n pasos, para formar la distribución $P(x \mid n)$,

```
In[55]:= xPopulation[n_, m_] := Table[ xRandom[n], {i, 1, m}]
```

Esta es una muestra de la posición final de 100 caminatas al azar, cada una de ellas con 50 pasos,

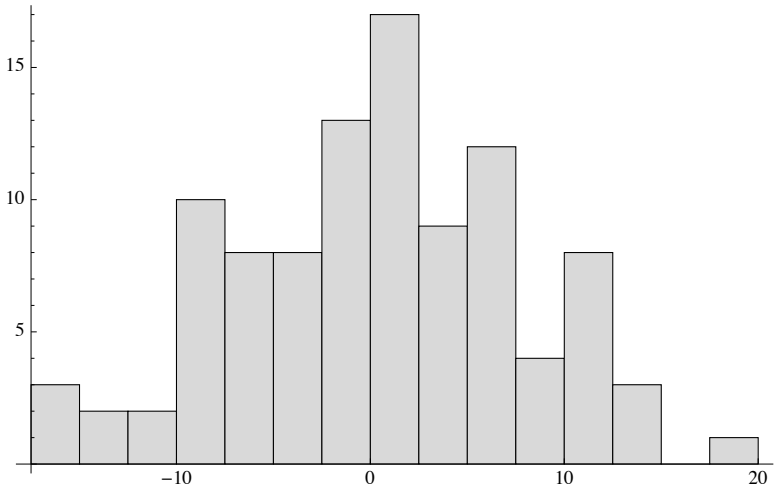
```
In[56]:= sample[100] = xPopulation[50, 100]
Out[56]= {6, -2, -4, 6, -2, 10, -14, -2, 6, -2, -8, 0, 0, 0, 6, 2, 4, -8,
  4, -6, -8, 4, 6, -6, 14, 0, -10, 10, -6, 0, 4, 6, -12, -2, 0,
  6, -6, 8, 0, 0, 2, -4, -4, -10, 4, -8, 4, 14, 10, -2, -4, -4,
  -2, -6, -2, 6, 2, 2, -12, 18, 6, 12, 12, -2, 6, -4, 8, -8, 8,
  4, -10, 12, 4, 2, -14, -2, 4, 2, -16, 12, -2, 8, -4, 6, -6,
  -4, 0, -6, 0, -16, -2, -16, 14, 2, 10, -2, -10, -6, 6, -8}
```

La cual podemos visualizar en un histograma; antes de utilizar el comando Histogram, es necesario correr el siguiente comando,

```
In[57]:= Needs["Histograms`"]
```

```
General::obspkg :
  Histograms` is now obsolete. The legacy version being loaded may conflict with current
      Mathematica functionality. See the
      Compatibility Guide for updating information. >>

BarChart3D::shdw :
  Symbol BarChart3D appears in multiple contexts {BarCharts`, System`}; definitions in context
      BarCharts` may shadow or be shadowed by other definitions. >>

Histogram3D::shdw :
  Symbol Histogram3D appears in multiple contexts {Histograms`, System`}; definitions in context
      Histograms` may shadow or be shadowed by other definitions. >>
```

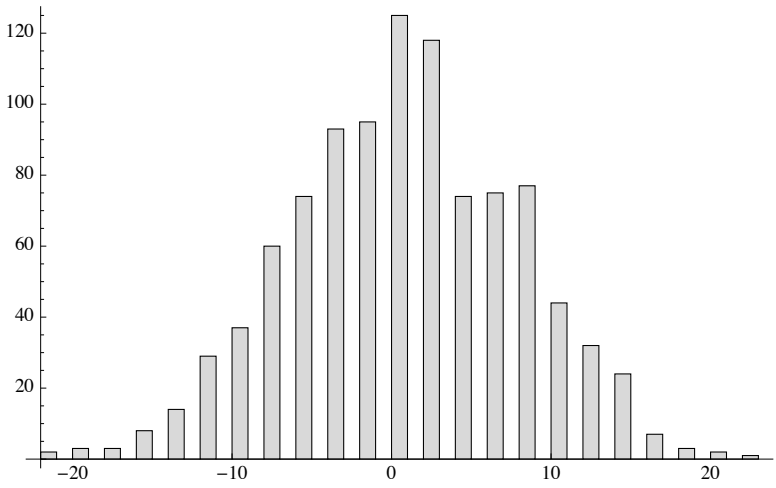
```
In[58]:= Histogram[ sample[100]]
```



Histogram[*datos***]**	Grafica un histograma de los datos calculando automáticamente los subintervalos.

En realidad, nos interesa la forma de la distribución cuando el tamaño de la muestra $m >> 1$. Por ejemplo, tomemos $m = 1000$

```
In[59]:= sample[1000] = xPopulation[50, 1000] ;
         Histogram[sample[1000]]
```



Esta es aproximadamente una distribución gaussiana. Si queremos la media y la varianza, usamos justamente los comandos Mean y Variance

```
In[61]:= N[ Mean[sample[1000] ]]
Out[61]= 0.3
```

```
In[62]:= N[Variance[sample[1000] ]]
Out[62]= 51.1852
```

Mean[*datos***]**	Calcula la media (promedio) de los datos.
Variance[*datos***]**	Calcula la varianza (cuadrado de la desviación estándar) de los datos.

Ahora intentaremos lo siguiente, queremos graficar la media y la varianza de la distribución para m=1000 como función de n

Primero construimos una lista de muestras de caminatas al azar. Cada muestra consiste de 1000 caminatas, cada una de ellas con el mismo número de pasos n. Las distintas muestras tienen valores de n entre 50 y 400.

```
In[63]:= caminatas = Table[{n, xPopulation[n, 1000]}, {n, 50, 400}];
```

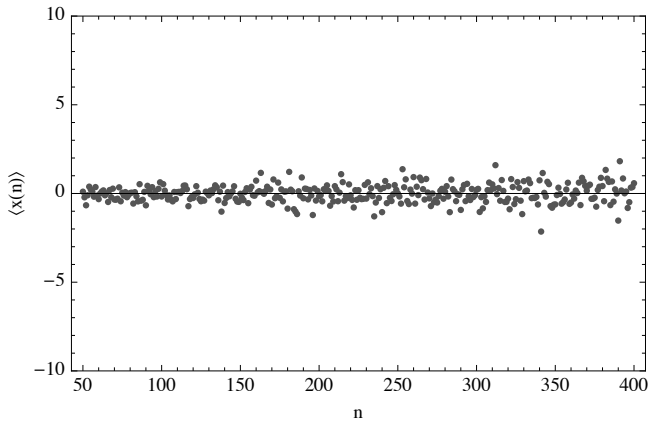
Luego, aplicamos una función para transformar dicha lista en otra cuyos elementos son las parejas ordenadas del tamaño de la caminata y el desplazamiento medio correspondiente: $\{n, \langle x \rangle\}$,

```
In[64]:= calculateMean[{n_, population_}] := {n, Mean[population]}
In[65]:= meanData = Map[ calculateMean, caminatas];
```

Al graficar los datos de la posición media contra n, observamos que $\langle x \rangle$ es independiente de n

```
In[66]:= ListPlot[ meanData, PlotRange → {All, {-10, 10}},
          Frame → True,  FrameLabel → {"n", "⟨x(n)⟩"}]
```

Out[66]=



Analizando los coeficientes de una regresión lineal, observamos que los datos son consistentes con la hipótesis simple de que $\langle x \rangle = 0$

```
In[67]:= Needs["LinearRegression`"]

General::obspkg :
  LinearRegression` is now obsolete. The legacy version being loaded may conflict with
        current Mathematica functionality. See the
        Compatibility Guide for updating information. >>

In[68]:= Regress[ meanData, {1, n}, n]

Out[68]= {ParameterTable →
            |  Estimate      SE            TStat     PValue
          1 | -0.0873072   0.0651126    -1.34087  0.180836,
          n | 0.000341486  0.000263868  1.29416   0.196467

  RSquared → 0.00477605, AdjustedRSquared → 0.0019244,
  EstimatedVariance → 0.250905, ANOVATable →

            DF     SumOfSq     MeanSq      FRatio     PValue
   Model    1      0.420225    0.420225    1.67484    0.196467 }
   Error    349    87.5657     0.250905
   Total    350    87.9859
```

`Regress[`*datos, funciones, variableIndependiente***`]`**
Realiza un ajuste lineal, devolviendo el análisis estadístico de los parámetros del ajuste.
Antes se tiene que correr **`Needs["LinearRegression`"].`**

En esta tabla, observamos las entradas de **`ParameterTable`**. Las filas corresponden a las estadísticas de los coeficientes de cada término del ajuste (que aquí corresponden al término constante y al número de pasos n). Las columnas de la tabla corresponden al estimador del coeficiente, al error estadístico de dicho estimador (es decir, su incertidumbre estadística), las dos últimas columnas nos indican qué tan probable es la hipótesis de que el coeficiente sea de hecho cero, pero hayamos obtenido un estimador diferente debido al azar.

Al leer la tabla, notamos que ambos estimadores son del orden de la incertidumbre estadística. En efecto, la hipótesis simple $\langle x \rangle = 0$ es razonable.

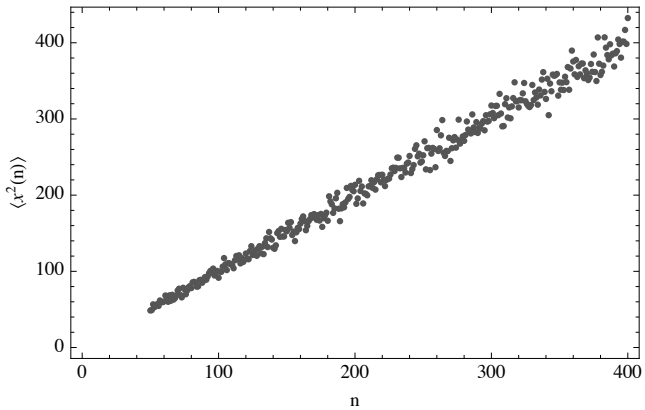
Por otro lado, el análisis correspondiente para la varianza comienza por aplicar una función a los datos para obtener $\{n,\ \mathrm{Var}(x)\}$. Dado que la media de la posición es cero, $\mathrm{Var}(x) = \left\langle x^2(n) \right\rangle$. Ahora calculamos la varianza que corresponde a cada número de pasos n.

```
In[69]:= calculateVariance[{n_, population_}] :=
          {n, Variance[population]}

In[70]:= varianceData = Map[ calculateVariance, caminatas];
```

Al graficar los datos de la varianza contra n, observamos que $\mathrm{Var}(x)$ es lineal con n

```
In[71]:= ListPlot[ varianceData,
          Frame → True, FrameLabel → {"n", "⟨x²(n)⟩"}]
```

Out[71]=



El análisis de regresión lineal revela que los datos son consistentes con $\langle x^2 \rangle = n$ además de que la pendiente es precisamente $2D$, que en este lo caso hemos tomado como 1.

```
In[72]:= Regress[ varianceData, {1, n}, n]
```

Out[72]=

```
{ParameterTable →
```

	Estimate	SE	TStat	PValue
1	0.764786	1.32199	0.578513	0.563291
n	0.998793	0.00535732	186.435	$1.527680356311807 \times 10^{-351}$

```
,
RSquared → 0.990059, AdjustedRSquared → 0.990031,
EstimatedVariance → 103.427, ANOVATable →
```

	DF	SumOfSq
Model	1	$3.59491 \times$
Error	349	36 095.9
Total	350	$3.63101 \times$

A diferencia del caso de $\langle x \rangle$, sólo el coeficiente del término constante es del mismo orden que su incertidumbre estadística. Por el contrario, el coeficiente del término n es aproximadamente 170 veces mayor a su incertidumbre, lo cual hace que sea muy razonable la hipótesis simple $\langle x^2 \rangle = n$.

Esto equivale a decir que el desplazamiento cuadrático medio de una partícula en movimiento Browniano crece linealmente con el número de pasos que realiza.

Para caminatas al azar más generales (en más dimensiones, con pasos en cualquier dirección y de tamaño no unitario) de todos modos se conserva el importante resultado de que el desplazamiento cuadrático medio es proporcional al número de pasos dados.

Proyecto. Cinética enzimática

Las proteínas son macromoléculas formadas por unidades llamadas aminoácidos, en números que van desde unos pocos hasta miles de ellos (la hemoglobina consta de alrededor de 5000). La composición de aminoácidos de cada proteína es como una huella digital y se relaciona estrechamente con la forma que adquiere la proteína en el espacio y de ahí con la función que desempeñará en el organismo vivo. Comúnmente, las proteínas con estructura tridimensional tienen algún tipo de función como catalizador de ciertas reacciones bioquímicas. A las proteínas de esta categoría se las llama *enzimas*.

Los factores más importantes que influyen en la velocidad de las reacciones catalizadas por enzimas son: las concentraciones de enzima libre, ligando (sustratos, productos, inhibidores y activadores), la concentración de iones de hidrógeno (pH), la fuerza iónica (suma del producto de la concentración molal de la especie j y su carga iónica, sobre todas las especies presentes en la mezcla) y la temperatura. Con el cuidado necesario, el análisis del efecto de estos factores puede enseñarnos mucho acerca de la naturaleza de las reacciones enzimáticas. La *cinética enzimática* es la rama de la Enzimología que se encarga de este estudio.

La reacción enzimática más simple consiste en la transformación de una única molécula de sustrato en una molécula de producto. Ya desde 1882 se conocía el papel del complejo enzima-sustrato (ES) como intermediario de la reacción. En 1913 L. Michaelis y M. Menten propusieron el siguiente esquema de reacción,

$$E + S \underset{k_2}{\overset{k_1}{\rightleftharpoons}} ES \overset{k_3}{\rightarrow} P + E$$

En la ecuación química E es la concentración molar de enzima, S de sustrato P de producto, y ES es la concentración del complejo enzima-sustrato. Este modelo sólo es válido cuando la concentración del sustrato es mayor que la concentración de la enzima, y para condiciones de estado estacionario, es decir, que la concentración del complejo enzima-sustrato permanece constante. Para determinar la velocidad máxima de una reacción enzimática, la concentración de sustrato (S) se aumenta hasta alcanzar una velocidad constante de formación de producto. En este proyecto escribiremos un código que sea capaz de ayudarnos a determinar si los datos experimentales de concentración de sustrato vs. velocidad de una reacción, para un sistema de enzima-sustrato, siguen una cinética de Michaelis y Menten o no, así como estimar los parámetros cinéticos de la enzima: la constante K_M y la velocidad máxima de reacción, $V_{\max}$.

La ecuación cinética que da la variación en el tiempo de la concentración del complejo ES (paso limitante de la reacción global), corresponde a un balance detallado (la generación menos el consumo) y se escribe como,

$$\frac{dES}{dt} = k_1 \times E \times S - k_2 \times ES - k_3 \times ES$$

En general no se encuentran soluciones analíticas para esta ecuación, aunque sí es posible resolverla numéricamente. No obstante, Michaelis y Menten introdujeron la hipótesis del *equilibrio rápido*, en la que se establece que las concentraciones de E, S y ES se equilibran muy rápido en comparación con la velocidad de disociación de ES en $P + E$. La velocidad instantánea de producción de P depende de ES

$$v = k_3 \times ES$$

Y la concentración total de enzima se reparte entre la forma libre y la enzima ligada al sustrato.

$$E_T = E + ES$$

Multiplicando la expresión anterior por $k_1 \cdot S$, y reacomodando términos, obtenemos

$$k_1 \times E \times S = k_1 \times E_T \times S - k_1 \times ES \times S$$

En las condiciones del equilibrio rápido, la velocidad de producción de *ES* es igual a la velocidad de disociación,

$$k_1 \times E \times S = \; k_2 \times ES + k_3 \times ES$$

igualando las últimas expresiones,

$$k_1 \times E_T \times S - k_1 \times ES \times S = \; k_2 \times ES + k_3 \times ES$$

y despejando para ES,

$$ES = \frac{k_1 \times E_T \times S}{k_1 \times S + k_2 + k_3}$$

o bien, si definimos $K_M = \frac{k_2 + k_3}{k_1}$, la constante de Michaelis,

$$ES = \frac{E_T \times S}{K_M + S}$$

Usando la expresión recién obtenida para *ES* en la expresión para la velocidad de la reacción,

$$v = \frac{k_3 \times E_T \times S}{K_M + S}$$

obtenemos la conocida ecuación de Michaelis y Menten. El término $k_3 \cdot E_T$ se define como una *velocidad máxima* ($v_{\max}$), y es una asíntota para la gráfica de *v*. En ese caso, toda la enzima disponible está saturada con sustrato.

Determinar si una enzima dada sigue una cinética Michaeliana requiere de dos pasos adicionales; el primero consiste en reescribir la ecuación de Michaelis-Menten en una forma lineal, del tipo $y = m\,x + b$, con ésto el segundo paso es aplicar un análisis de regresión a un conjunto de datos cinéticos medidos experimentalmente (*v s. S*) y determinar el grado de ajuste entre éstos y el modelo.

Existen varias formas lineales de la ecuación de Michaelis y Menten (una curva hiperbólica) de uso frecuente, éstas son:

La ecuación doble recíproca de Lineweaver-Burke:

$$\frac{1}{v} = \frac{K_M}{v_{\max}} \frac{1}{S} + \frac{1}{v_{\max}}$$

La ecuación de Hanes-Woolf:

$$\frac{S}{v} = \frac{1}{v_{\max}} S + \frac{K_M}{v_{\max}}$$

La ecuación de Eadie-Scatchard:

$$v = -K_M \frac{v}{S} + v_{\max}$$

La ecuación de Woolf-Augustinsson-Hofstee:

$$\frac{v}{S} = \frac{1}{K_M} v + \frac{v_{\max}}{K_M}$$

En nuestro proyecto deseamos escribir un programa que reciba como argumento una lista de parejas ordenadas ({*concentración de sustrato*, *velocidad*}) y efectúe las transformaciones necesarias para aplicar un análisis de regresión usando cualquiera de los modelos lineales de la ecuación de Michaelis-Menten, permitiendo al usuario elegir cuál de ellos se utilizará, entregando como salida una gráfica del modelo ajustado (una recta con los puntos experimentales) junto a una lista con los parámetros de la enzima y la bondad del ajuste.

Varios elementos que intregrarán el programa, ya se han discutido en ejemplos previos: la obtención de una ecuación ajustada a partir de una lista de parejas ordenadas, la extracción de los parámetros del ajuste y la graficación simultánea de los datos y la ecuación ajustada.

La manipulación de la lista original de datos puede hacerse mediante diversas técnicas, que se han presentado a lo largo de este libro. Una de las técnicas más poderosas, por su sencillez y efectividad, es la que usa el reconocimiento de *patrones* de *Mathematica*. Por ejemplo, al definir una función simple, digamos

```
f[x_] := x^2 + 3
```

el guión bajo que se coloca inmediatamente después del argumento de `f` entre los corchetes, le dice a *Mathematica* que cualquier objeto que se coloque entre los corchetes se sustituya por el nombre `x` en la expresión de la derecha. Así, "`x_`" no sólo es un objeto en sí, sino un *patrón* que se ajusta a cualquier objeto (números, listas, otras expresiones, etcétera) definido en *Mathematica*.

Podemos usar patrones para indicar a *Mathematica* cómo manipular elementos individuales de una lista, sin tener que definir variables adicionales. Tomemos una lista que contenga datos experimentales de concentración vs. velocidad de reacción para un cierto sistema de enzima-sustrato

```
In[73]:= data = {{8.33 * 10^-6, 13.8 * 10^-9}, {1.00 * 10^-5, 16.0 * 10^-9},
    {1.25 * 10^-5, 19.0 * 10^-9}, {1.67 * 10^-5, 23.6 * 10^-9},
    {2.00 * 10^-5, 26.7 * 10^-9}, {2.50 * 10^-5, 30.8 * 10^-9},
    {3.33 * 10^-5, 36.3 * 10^-9}, {4.00 * 10^-5, 40.0 * 10^-9},
    {5.00 * 10^-5, 44.4 * 10^-9}, {6.00 * 10^-5, 48.0 * 10^-9},
    {8.00 * 10^-5, 53.4 * 10^-9}, {1.00 * 10^-4, 57.1 * 10^-9},
    {2.00 * 10^-4, 66.7 * 10^-9}};
  TableForm[data,
   TableHeadings → {None, {"[S], mol·l^-1", "v, mol·l^-1·min^-1"}}]
```

Out[74]//TableForm=

[S], mol·l^{-1}	v, mol·l^{-1}·min^{-1}
8.33×10^{-6}	1.38×10^{-8}
0.00001	1.6×10^{-8}
0.0000125	1.9×10^{-8}
0.0000167	2.36×10^{-8}
0.00002	2.67×10^{-8}
0.000025	3.08×10^{-8}
0.0000333	3.63×10^{-8}
0.00004	$4. \times 10^{-8}$
0.00005	4.44×10^{-8}
0.00006	4.8×10^{-8}
0.00008	5.34×10^{-8}
0.0001	5.71×10^{-8}
0.0002	6.67×10^{-8}

Si deseamos calcular los recíprocos de `[S]` y `v`, podríamos indicarle a *Mathematica* lo siguiente

```
In[75]:= newdata = data /. {s_, v_} → {1 / s, 1 / v};
```

es decir, "a todos los elementos de `data` que concuerden con el patrón `{s_,v_}`, aplícales la transformación `{1/s,1/v}`". En formato tabular, la transformación aplicada a la lista `data` da como resultado

```
In[76]:= TableForm[newdata, TableHeadings -> {None, {"1/[S]", mol^-1·l", "1/v", mol^-1·l·min"}}]
```

Out[76]//TableForm=

$\frac{1}{[S]}$, $mol^{-1} \cdot l$	$\frac{1}{v}$, $mol^{-1} \cdot l \cdot min$
120 048.	7.24638×10^7
100 000.	6.25×10^7
80 000.	5.26316×10^7
59 880.2	4.23729×10^7
50 000.	3.74532×10^7
40 000.	3.24675×10^7
30 030.	2.75482×10^7
25 000.	2.5×10^7
20 000.	2.25225×10^7
16 666.7	2.08333×10^7
12 500.	1.87266×10^7
10 000.	1.75131×10^7
5000.	1.49925×10^7

Ahora podemos escribir el código de la función que llamaremos **michaelis**, el cual nos ayudará a concluir si la cinética en estudio sigue el modelo de Michaelis-Menten.

```
In[77]:= michaelis[lista_, modelo_] :=
 Module[{datos, abcisas, limInfX, limSupX, encabezado,
   labelX, labelY, ecuacion, gteo, gdata, parametros, rsq},
  (*Selección del modelo*)
  If[modelo == "lb",
   datos = lista /. {s_, v_} → {1 / s, 1 / v};
   encabezado = "Lineweaver-Burk";
   labelX = "1/S";
   labelY = "1/v"];
  If[modelo == "hw",
   datos = lista /. {s_, v_} → {s, s / v};
   encabezado = "Hanes-Woolf";
   labelX = "S";
   labelY = "S/v"];
  If[modelo == "wah",
   datos = lista /. {s_, v_} → {v / s, v};
   encabezado = "Woolf-Augustinsson-Hofstee";
```

```
 labelX = "v/S";
 labelY = "v"];
If[modelo == "es",
 datos = lista /. {s_, v_} → {v, v / s};
 encabezado = "Eadie-Scatchard";
 labelX = "v";
 labelY = "v/S"];
(*Ecuación ajustada*)
ecuacion = Fit[datos, {1, x}, x];
(*Coeficiente de correlación al cuadrado:
  bondad del ajuste*)
rsq = Regress[datos, {1, x}, x][[2]];
(*Obtención de los parámetros de Michaelis-Menten*)
parametros = CoefficientList[ecuacion, x];
If[modelo == "lb",
 Vmax = 1 / parametros[[1]];
 Km = parametros[[2]] * Vmax];
If[modelo == "hw",
 Vmax = 1 / parametros[[2]];
 Km = parametros[[1]] * Vmax];
If[modelo == "wah",
 Vmax = parametros[[1]];
 Km = -1 * parametros[[2]]];
If[modelo == "es",
 Km = -1 / parametros[[2]];
 Vmax = parametros[[1]] * Km];
(*Gráfica del modelo con los parámetros ajustados*)
abcisas = datos /. {x_, y_} → {x};
limInfX = Min[abcisas];
limSupX = Max[abcisas];
gteo = Plot[ecuacion, {x, limInfX, limSupX},
  PlotStyle → {Blue, Dashing[{0.005}]}];
(*Gráfica de los datos experimentales*)
gdata = ListPlot[datos,
  PlotStyle → {Red, PointSize[.015]}];
(*Impresión de resultados*)
Print[Show[{gdata, gteo},
```

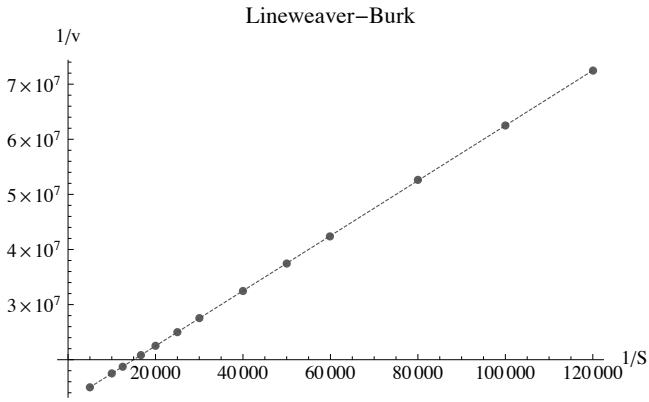
```
    PlotLabel → encabezado,
    AxesLabel → {labelX, labelY}]];
  {vmax → Vmax, km → Km, rsq}
]
```

Como utilizamos el comando **Regress** para obtener un parámetro de la bondad del ajuste (el coeficiente de correlación al cuadrado), debemos llamar al paquete *LinearRegression.m* antes de correr nuestro primer ejemplo, mediante la instrucción,

```
In[78]:= Needs["LinearRegression`"]
```

Ahora podemos correr el ejemplo, con el modelo de Lineweaver.Burk:

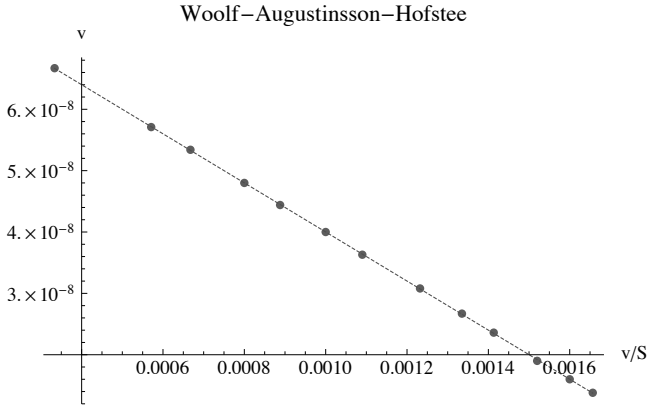
```
In[79]:= michaelis[data, "lb"]
```




```
Out[79]= {vmax → 7.99923 × 10^-8, km → 0.0000399887, RSquared → 0.999992}
```

y otro más, con el modelo de Woolf-Augustinsson-Hofstee,

```
In[80]:= michaelis[data, "wah"]
```




```
Out[80]= {vmax → 8.0004 × 10^-8, km → 0.0000399982, RSquared → 0.999968}
```

Hay enzimas que no obedecen la ecuación de Michaelis-Menten. Se dice que su cinética no es Michaeliana. En estos casos pequeñas variaciones del sustrato se traducen en grandes variaciones en la velocidad de reacción. Estos casos se presentan por ejemplo, cuando al ligarse un substrato cambia la afinidad de los demás sitios de enlace, como sucede con la hemoglobina. A cotinuación se muestran los datos experimentales de velocidad vs. concentración, obtenidos para una enzima para la cuál tendremos qué averiguar si sigue una cinética de Michaelis-Menten o no.

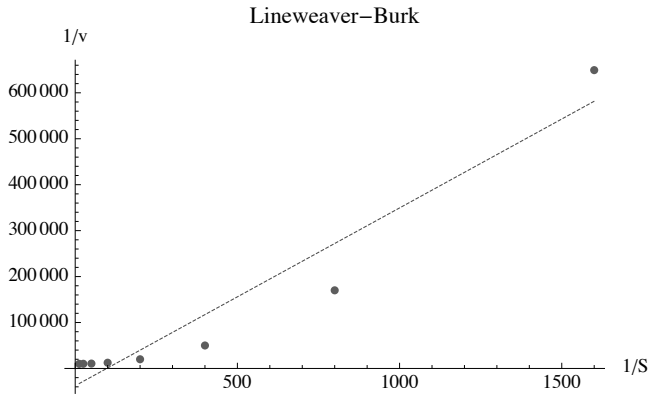
```
In[81]:= data2 = {
           {6.25 * 10^-4, 1.54 * 10^-6},
           {12.5 * 10^-4, 5.88 * 10^-6},
           {25.0 * 10^-4, 20.0 * 10^-6},
           {50.0 * 10^-4, 50.0 * 10^-6},
           {100.0 * 10^-4, 80.0 * 10^-6},
           {200.0 * 10^-4, 94.12 * 10^-6},
           {400.0 * 10^-4, 98.46 * 10^-6},
           {800.0 * 10^-4, 99.61 * 10^-6}
         };
         TableForm[data2, TableHeadings → {None, {"[S], mol·l^-1", "v, mol·l^-1·min^-1"}}]
```

Out[82]//TableForm=

[S], $mol \cdot l^{-1}$	v, $mol \cdot l^{-1} \cdot min^{-1}$
0.000625	1.54×10^{-6}
0.00125	5.88×10^{-6}
0.0025	0.00002
0.005	0.00005
0.01	0.00008
0.02	0.00009412
0.04	0.00009846
0.08	0.00009961

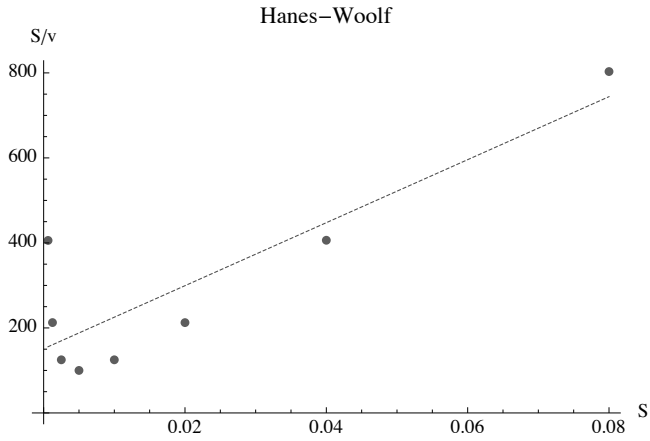
Con nuestro código ahora podemos analizar los datos experimentales para concluir si siguen una cinética de Michaelis-Menten.

```
In[83]:= michaelis[data2, "lb"]
```



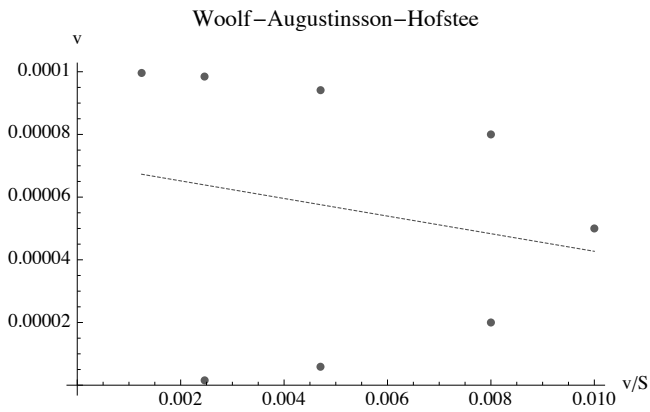

```
Out[83]= {vmax → -0.0000265811, km → -0.0102881, RSquared → 0.930087}
```

```
In[84]:= michaelis[data2, "hw"]
```



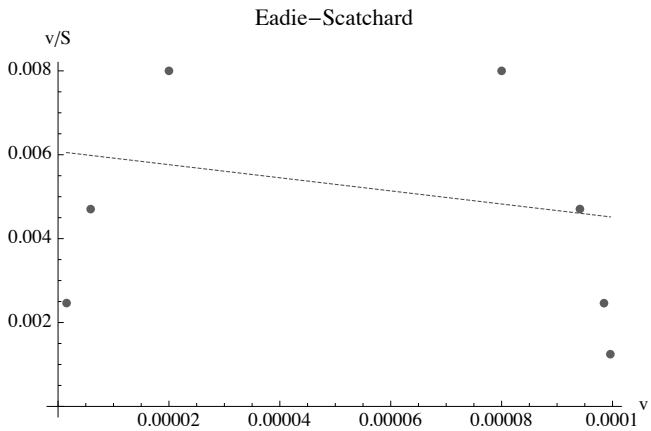

```
Out[84]= {vmax → 0.000134922, km → 0.0203912, RSquared → 0.750543}
```

```
In[85]:= michaelis[data2, "wah"]
```




```
Out[85]= {vmax → 0.0000707971, km → 0.0028082, RSquared → 0.0439058}
```

```
In[86]:= michaelis[data2, "es"]
```




```
Out[86]= {vmax → 0.000388637, km → 0.0639597, RSquared → 0.0439058}
```

Finalmente podemos concluir que la enzima en estudio no sigue una cinética de Michaelis-Menten. Nótese cómo el comportamiento de los datos experimentales se aleja de la linealidad. En el primer caso, con el modelo de Lineweaver-Burk, aunque se obtiene un valor del coeficiente de correlación cercano a la unidad, los parámetros cinéticos calculados tienen signo negativo, lo cual carece de significado físico, aunado a la confirmación visual del comportamiento no lineal de los datos experimentales permite afirmar sin lugar a dudas el comportamiento no michaeliano de la enzima. Como en este ejemplo, en la práctica deben considerarse criterios alternativos encaminados a brindar evidencia clara y apoyar objetivamente nuestras conclusiones.

Proyecto. Cómo determinar si todos los elementos de una lista son idénticos

En este proyecto nos proponemos hacer un programa que determine si todos los elementos de una lista son idénticos. Dichos elementos pueden ser de cualquier tipo, tanto cadenas de caracteres como expresiones numéricas o simbólicas.

Para dar un ejemplo, el siguiente programa genera aleatoriamente expresiones simbólicas del tipo $ax^b y^c$, donde a, b y c son enteros aleatorios distribuidos uniformemente en el intervalo [0, 2],

```
In[87]:= RandomInteger[{1, 10}]
Out[87]= 6
In[88]:= RandomInteger[{0, 2}]
Out[88]= 1
In[89]:= Clear[x, y]
In[90]:= FortuneCookies[n_] :=
          Table[
           RandomInteger[{1, 10}] x^RandomInteger[{0,2}],
           {j, 1, n}
          ]
In[91]:= cookies = FortuneCookies[5]
Out[91]= {7, 9 x^2, 3 x, 3 x, 9 x^2}
```

Para probar nuestros próximos programas, necesitaremos una lista más larga de expresiones aleatorias. Por el momento usaremos una lista de 5000 expresiones.

```
In[92]:= manyCookies = FortuneCookies[5000];
```

Un programa basado en lenguajes de programación tradicionales como Fortran o C tendría que recurrir a especificar cómo distinguir cada uno de estos tipos de datos, para luego revisar secuencialmente cada una de sus partes y decidir si se trata de elementos iguales o no. En *Mathematica* podemos utilizar otros métodos.

Una de las ideas del lenguaje de programación de *Mathematica* es aplicar una operación simultáneamente a todos los elementos de una lista. Para aprovechar esto, debemos conocer algunos comandos que hacen una prueba para saber si dos expresiones son iguales. Algunos de estos comandos son **SameQ**, **Equal** y **Unequal**. Otros comandos útiles son los comandos relacionados con conjuntos, tales como **Union**, **Intersection**, etcétera. A continación veremos como aplicarlos a la solución de nuestro problema.

El comando **SameQ[expr1, expr2]** se evalúa como **True** (verdadero) si las dos expresiones **expr1** y **expr2** son idénticas.

SameQ[*expr1, expr2***]**	*Determina si dos expresiones son idénticas.*

Así,

```
In[93]:= SameQ[x^2, x^2]
Out[93]= True
```

y, por el contario,

```
In[94]:= SameQ[x^2, y^2]
Out[94]= False
```

Lo más natural es aplicar el comando **SameQ** a la lista de nuestro problema para saber si todos los elementos son idénticos, pero cuidado, si lo aplicamos simplemente como,

```
In[95]:= SameQ[cookies]
Out[95]= True
```

el resultado no es el esperado. ¿Cuál es la razón? El problema consiste en que **SameQ** funciona con cualquier número de argumentos, con un argumento, siempre devuelve **True,**

```
In[96]:= SameQ[x^2]
Out[96]= True
```

Por eso, cuando evaluamos **SameQ** con nuestra lista **cookies** como único argumento siempre obtendremos el resultado **True**, como si todos los elementos de la lista fueran identicos.

Por el contrario, dando dos o más argumentos **SameQ** si realiza las comparaciones entre los argumentos,

```
In[97]:= SameQ[x^2, x^2, x^2]
Out[97]= True
```

```
In[98]:= SameQ[x^2, x^2, y^2]
Out[98]= False
```

Este es el funcionamiento que nosotros queremos. Ahora nuestro problema es, cómo podemos producir el comando **SameQ[elem1, elem2, ...]** a partir de nuestra lista **{elem1, elem2, ...}**?
Recordando lo que sabemos sobre manipulación de listas, recordamos que el comando **Apply** nos permite realizar justamente esta transformación,

```
In[99]:= Apply[SameQ, cookies]
Out[99]= False
```

Probemos de nuevo, usando ahora la lista más larga

```
In[100]:= Apply[SameQ, manyCookies]
Out[100]= False
```

Llegamos así a nuestro primer programa para resolver el problema. En *Mathematica*, es tradicional que las funciones que devuelven sólo valores lógicos (**True** o **False**) terminen en Q mayúscula,

```
In[101]:= ListSameQ[list_] := Apply[SameQ, list]
```

Probemos ahora el programa con listas cortas, donde es probable que en efecto los elementos sean idénticos. Primero generemos 20 listas de 2 elementos, éstas seran las listas con las que vamos a probar el programa,

```
In[102]:= moreCookies = Table[ FortuneCookies[2], {i, 1, 20}]
Out[102]= {{10, 8}, {4, 10}, {9 x^2, 10 x}, {4 x, 6}, {x, 3},
           {2 x^2, 1}, {5 x, x^2}, {4 x^2, 8}, {x, 9 x^2}, {10 x, 7 x^2},
           {6, 9}, {9, 5 x}, {3 x, 5}, {5 x^2, 8}, {2 x, 6},
           {5, 7 x}, {2 x, 10}, {7 x^2, x}, {7 x, 10}, {9 x^2, 10 x}}
```

A simple vista es difícil decir cuáles listas son idénticas. Aplicando el programa, podemos ver que hay exactamente dos listas cuyos elementos son idénticos,

```
In[103]:= Map[ ListSameQ, moreCookies]
Out[103]= {False, False, False, False, False, False,
           False, False, False, False, False, False, False,
           False, False, False, False, False, False, False}
```

Es importante señalar que en la ejecución de los comandos de esta sección, el resultado estará sujeto a cambios debido a que el comando **RandomInteger** (en el programa **FortuneCoookies**) dará resultados distintos cada vez que se evalúe.

Si, por otra parte, es deseable reproducir el ejemplo varias veces y obtener cada vez el mismo resultado, debe especificarse una semilla que inicialice la secuencia de números pseudoaleatorios, mediante el comando **SeedRandom** (véase el ejemplo de difusión en una dimensión en este capítulo).

El comando **Equal** es similar al comando **SameQ**, pero es más cauteloso. Cuando **Equal** no conoce los valores de las variables involucradas en las expresiones, no devuelve ni **True** ni **False**. Por el contrario, el resultado permanece sin evaluar, y se escribe como una ecuación,

Equal[*expr1,expr2***]**	Expresa la ecuación *expr1* == *expr2* , que puede ser **True**, **False** o quedarse sin evaular.

Por ejemplo,

```
In[104]:= Apply[ Equal, cookies]
Out[104]= 7 == 9 x^2 == 3 x
```

Por esta razón, el comando **Equal** no nos sirve para resolver el problema.

El comando **Union** permite manipular conjuntos. En *Mathematica*, los conjuntos se representan simplemente como listas, pero se les imponen restricciones adicionales. Una de las más importantes es que los conjuntos no pueden tener elementos repetidos. Por eso, cuando se hace la operación de unión entre conjuntos los elementos idénticos se eliminan para dejar un sólo elemento de cada tipo.

Union[*lista***]**	Devuelve los elementos de la lista, eliminando las duplicaciones.

```
In[105]:= Union[{1, 2, 3}, {3, 4}]
Out[105]= {1, 2, 3, 4}
```

Esta propiedad de la unión de conjuntos es la que nos permite aprovechar el comando **Union** para verificar si los elementos de una lista son idénticos o no, ya que cuando se le da al comando **Union** un sólo argumento, éste elimina los elementos duplicados,

```
In[106]:= Union[{1, 2, 3, 3}]
Out[106]= {1, 2, 3}
```

Así, para resolver nuestro problema bastaría aplicar **Union** a la lista en cuestión. Si el resultado es un conjunto con un sólo elemento es porque todos los elementos de la lista eran idénticos. Así llegamos al segundo programa para resolver el problema,

```
In[107]:= IdenticalByUnion[list_] := Length[ Union[list]] ≤ 1
```

Corremos de nuevo los diagnósticos y los comparamos con los del primer programa. Primero, para la lista corta de cinco elementos,

```
In[108]:= IdenticalByUnion[cookies]
Out[108]= False
In[109]:= ListSameQ[cookies]
Out[109]= False
```

Finalmente, para el conjunto de 20 listas con dos elementos cada una,

```
In[110]:= Map[IdenticalByUnion, moreCookies]
Out[110]= {False, False, False, False, False, False,
           False, False, False, False, False, False, False,
           False, False, False, False, False, False, False}
In[111]:= Map[ListSameQ, moreCookies]
Out[111]= {False, False, False, False, False, False,
           False, False, False, False, False, False, False,
           False, False, False, False, False, False, False}
```

Una observación final para este problema. No tuvimos cuidado de verificar que las expresiones simbólicas fuesen equivalentes algebraicamente, es decir, nuestros programas toman por diferentes a expresiones como $a^2 + 2\,a\,b + b^2$ y $(a + b)^2$. Una manera de corregir esto sería aplicar comandos del tipo de **`Simplify`** antes de realizar las comparaciones.

Proyecto. Velocidad terminal y la profundidad de un pozo

El último proyecto de programación estilo *Mathematica* tiene que ver con el problema de la caída de un cuerpo dentro de un medio que le ejerce cierta fuerza de resistencia. Desde los primeros cursos universitarios de física, uno se familiariza con el problema de la caída libre. Un ejemplo de este tipo de problemas es el siguiente: Una persona deja caer una piedra desde el borde de un pozo y escucha que ésta choca contra el agua tras un intervalo de tiempo t_1. ¿A qué profundidad h por debajo del borde del pozo se encuentra el nivel del agua?

Para analizar este problema integraremos las capacidades simbólicas, gráficas y de programación de *Mathematica*.

Si ignoramos el efecto de la fricción del aire y suponemos que la velocidad del sonido $c \gg h / t_1$, la ecuación que describe el movimiento de la piedra será

$$m \frac{\mathrm{dv}}{\mathrm{dt}} = m\,g$$

donde la velocidad $v(t) = \frac{\mathrm{dx}}{\mathrm{dt}}$. Para las condiciones iniciales $x(0) = 0$ y $v(0) = 0$, la solución es,

$$x(t) = \frac{1}{2} g t^2$$

Por lo tanto, la solución al problema será,

$$h = \frac{1}{2} g t_1^2$$

Digamos que queremos un programa que grafique la solución y devuelva una tabla de las alturas correspondientes a los tiempos $t_1 = 1, 1.5, 2. .., 5$s. El siguiente programa nos permite hacerlo,

```
In[112]:= CaidaLibre[g_, {tMin_, tMax_, Δt_}] :=
  Module[{tablaDeAlturas, grafica},
    grafica = Plot[1/2 g t^2, {t, tMin, tMax},
   Frame → True, FrameLabel → {"t/s", "h/m"}];
    tablaDeAlturas = Table[
   {
    PaddedForm[t, {3, 2}],
    PaddedForm[1/2 g t^2, {6, 2}]
   }, {t, tMin, tMax, Δt}];
    TableForm[ tablaDeAlturas,
  TableHeadings → {None, {"t/s", "h/m"}}]
 ]
```

El comando **`Module`** sirve para encapsular las variables y el código de un programa, protegiéndolos de conflictos con variables del mismo nombre que hayamos usado durante la sesión. Esto significa que en nuestro programa, las variables **`tablaDeAlturas`** y **`grafica`** son completamente diferentes e independientes de cualquier variable que hayamos usado antes.

Continuando con el cuerpo del programa, lo primero que se hace es generar la gráfica de la función $\frac{1}{2} g t^2$, asignándola a la variable **`grafica`**. En segundo lugar, formamos una tabla de parejas ordenadas $\{t, \frac{1}{2} g t^2\}$; para hacer más fácil de leer esta tabla, usamos el comando **`PaddedForm[`***expr*, {*dígitos*, *decimales*}**`]`** para escribir las entradas de la tabla con un número definido de dígitos y decimales.

`PaddedForm[`*expresión, {dígitos, decimales}***`]`**
Representa la expresión dada usando el número de dígitos y decimales especificados.

Finalmente, usamos el comando **`TableForm`** para poner un encabezado a las columnas de nuestra tabla. El resultado final es el siguiente,

```
In[113]:= CaidaLibre[9.81, {0, 5.0, 0.5}]
```

Out[113]//TableForm=

t/s	h/m
0.00	0.00
0.50	1.23
1.00	4.91
1.50	11.04
2.00	19.62
2.50	30.66
3.00	44.15
3.50	60.09
4.00	78.48
4.50	99.33
5.00	122.63

Vemos que la profundidad del pozo crece rápidamente con el tiempo necesario para oír el chapuzón. ¡Se requieren tan sólo 4.5 segundos para caer una altura de 100 m en caída libre!

Una manera de entender cualitativamente el movimiento de una partícula es mediante retratos fase. Para esto, hay que resolver la ecuación de movimiento para diversos valores de las condiciones iniciales, y graficar la pareja ordenada $\{x(t),\ v(t)\}$ como función del tiempo para cada condición inicial $\{x_0,\ v_0\}$. Esto lo conseguimos con el siguiente programa,

```
In[114]:= OrbitasFase[{tMin_, tMax_}, {xMin_, xMax_, Δx_},
    {vMin_, vMax_, Δv_}] := Table[Module[{sol, g},
     g = 9.81;

     sol = DSolve[ {x''[t] == g , x'[0] == v0, x[0] == x0}, x, t];
     ParametricPlot[
      Evaluate[{x[t], x'[t]} /. First[sol]], {t, tMin, tMax} ,
       PlotStyle → Hue[ 9/10 (x0 - xMin)/(xMax - xMin) , 1/2 + 1/2 (v0 - vMin)/(vMax - vMin), 1 ],
       DisplayFunction → Identity]
    ],
    {v0, vMin, vMax, Δv}, {x0, xMin, xMax, Δx}
   ]
```

Aquí estamos generando una tabla cuyos elementos son el resultado de evaluar un comando **Module.** Dentro del comando **Module**, resolvemos una ecuación diferencial cuyas condiciones iniciales son las variables a iterar para formar la tabla (**v0** y **x0**). Por lo tanto, para este programa las condiciones iniciales forman una retícula regular.

El resto es simplemente graficar la solución y su derivada (esto es, la posición de la partícula y su velocidad) en el espacio con coordenadas (*x*, *v*). Para poder distinguir las trayectorias, a cada una la graficamos con un color diferente: ésto se consigue al variar linealmente los argumentos del comando **Hue[***tono*, *saturación*, *brillo***]**. Tal vez lo más interesante es el uso de la opción **DisplayFunction→ Identity** para evitar que se despliegue cada una de las gráficas. En lugar de eso, primero obtenemos una lista de todas las órbitas en el espacio fase, las reunimos en una sola gráfica y *sólo entonces* dejamos que se desplieguen.

DisplayFunction → *valor*	Permite controlar el despliegue de la gráfica, usando como valores **$DisplayFunction** e **Identity**, respectivamente.

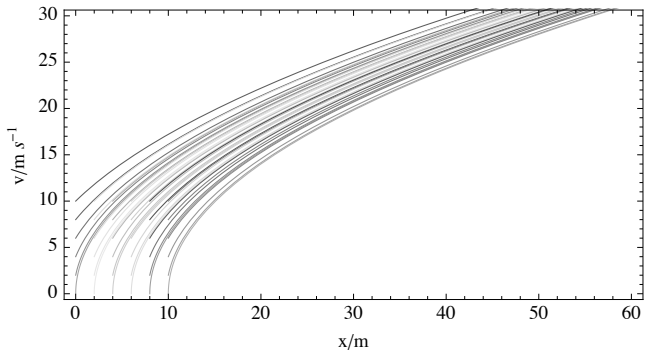
Al correr el programa, guardamos el resultado en una variable.

```
In[115]:= orbitas = OrbitasFase[{0, 5}, {0, 10, 2}, {0, 10, 2}];
```

Ahora usamos el comando **Show** para desplegar todas las funciones simultáneamente, usando la opción **DisplayFunction→ $DisplayFunction.**

```
In[116]:= Show[ Flatten[orbitas], DisplayFunction → $DisplayFunction,
           Frame → True, FrameLabel → {"x/m", "v/m s^-1"},
           PlotRange → {{0, 60}, {0, 30}}]
```

Out[116]=



El retrato fase así obtenido es bastante aburrido, todas las órbitas son parábolas. Sin embargo, ¿qué pasa si no podemos ignorar la fricción del aire?, ¿cuál será la expresión que reemplaza a $\frac{1}{2} g t_1^2$? y ¿cómo se verá el retrato fase?

Comencemos por establecer las ecuaciones de movimiento. Ahora, aparte de la fuerza de gravedad, aparecerá en las ecuaciones un término debido a la resistencia del aire. Cuando el objeto que cae es se mueve lentamente, la fuerza de resistencia F_r debida al aire es proporcional a la velocidad de caída.

$$F_r = -k\,v$$

Más aún, si el objeto que cae es aproximadamente esférico, el coeficiente de resistencia k puede calcularse mediante la solución de las ecuaciones de la hidrodinámica. El resultado se conoce como la *Ley de Stokes*:

$$k = 6\,\pi\,\eta\,R$$

donde R es el radio del objeto y η es la viscosidad dinámica del aire ($\eta = 1.8 \times 10^{-5}$ Pa s, para condiciones de temperatura y humedad típicas. Con esto queremos recalcar que la ley de fuerza para la resistencia del aire es relativamente simple cuando la velocidad de caída es pequeña. Así, la nueva ecuación de movimiento será

$$m\,\frac{\mathrm{dv}}{\mathrm{dt}} = -\,k\,v\,+m\,g$$

Antes de buscar la solución, nos conviene familiarizarnos un poco con la nueva ecuación. Por ejemplo, ¿hay alguna solución de equilibrio en que $\frac{\mathrm{dv}}{\mathrm{dt}} = 0$?

En efecto, existe una solución de este tipo, cuando la fuerza de resistencia balancea exactamente la fuerza de gravedad. Esto ocurre para un valor especial de la velocidad,

$$v_T = \frac{m\,g}{k}.$$

A este valor especial se le llama *velocidad terminal,* porque es valor al que tiende asintóticamente la velocidad del objeto que cae experimentando la resistencia del medio.

Si despejamos el valor del coeficiente de resistencia k en función de v_T y sustituímos en la ecuación de movimiento, encontraremos que ésta se simplifica a,

$$\frac{\mathrm{dv}}{\mathrm{dt}} = g\left(1 - \frac{v}{v_T}\right)$$

Podemos encontrar la solución recurriendo al comando **`DSolve`**,

```
In[117]:= sol = DSolve[ {v'[t] == g (1 - v[t]/vT), v[0] == 0}, v, t]

Out[117]= {{v → Function[{t}, ℯ^(-g t/vT) (-1 + ℯ^(g t/vT)) vT]}}

In[118]:= v[t] /. First[ sol ]

Out[118]= ℯ^(-g t/vT) (-1 + ℯ^(g t/vT)) vT
```

`DSolve[`*edo, y, t***`]`**	Resuelve las ecuaciones diferenciales ordinarias *edo*, devolviendo la solución *y(t)* como función de *t*.

También podemos usar un cambio de variable: $v = v_T\,(1 + u)$. Así, al sustituir en la ecuación de movimiento se tendrá

$$v_T\,\frac{\mathrm{du}}{\mathrm{dt}} = g\left(1 - \frac{v_T(1+u)}{v_T}\right) = -\,g\,u$$

lo anterior es equivalente a

$$\frac{\mathrm{du}}{\mathrm{dt}} = \frac{u}{\tau}$$

En otras palabras, el tiempo característico del problema es $\tau = v_T / g$. Tomando en cuenta la definición de v_T, obtenemos también

$$\tau = \frac{m}{k} = \frac{2}{9}\frac{\rho R^2}{\eta}$$

En la segunda igualdad hemos sustituído la ley de Stokes y escrito la masa en términos de la densidad y el volumen ($m = \frac{4}{3}\pi R^3$). Regresando a la ecuación de movimiento, encontramos que la solución a

$$\frac{du}{u} = -\frac{dt}{\tau}$$

es

$$u(t) = A \exp\left(-\frac{t}{\tau}\right)$$

regresando a la variable original e integrando $\frac{\mathrm{dx}}{\mathrm{dt}} = v$, obtenemos la solución general:

$$v(t) = v_T\left(1 + A \exp\left(-\frac{t}{\tau}\right)\right)$$
$$x(t) = v_T\left(t - A\,\tau \exp\left(-\frac{t}{\tau}\right) + B\right)$$

Para encontrar el valor de las constantes A y B, recurrimos a las condición iniciales: $x(0) = x_0$ y $v(0) = v_0$:

$$v(0) = v_T(1 + A) = v_0$$
$$x(0) = v_T(-A\,\tau + B) = x_0$$

La solución es,

```
In[119]:= Clear[vT, v0, A, B, τ]
```

```
In[120]:= Simplify[ Solve[ {vT (1 + A) == v0, vT (- A τ + B) == x0}, {A, B}] ]
```

$$\text{Out[120]= } \left\{\left\{\mathtt{B} \to \frac{\mathtt{x0} + \mathtt{v0}\ \tau - \mathtt{vT}\ \tau}{\mathtt{vT}},\ \mathtt{A} \to -1 + \frac{\mathtt{v0}}{\mathtt{vT}}\right\}\right\}$$

Sustituyendo estas constantes en la solución general, obtenemos la solución específica para cada par de condiciones iniciales,

$$x(t) = v_T\, t + (v_T - v_0)\,\tau\left(e^{-t/\tau} - 1\right) + x_0$$

Ahora hacemos el cambio de variables $\frac{t}{\tau} \to t$, $\frac{x}{v_T\,\tau} \to x$, $\frac{v_0}{v_T} \to v$, y obtenemos

$$x(t) = t + (1 - v_0)\ (e^{-t} - 1) + x_0$$

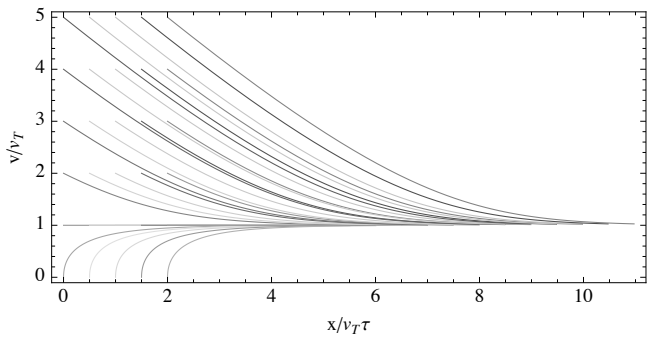
En estas unidades, la profundidad del pozo es simplemente $h = x(t_1) = t + e^{-t} - 1$, ya que el guijarro parte del origen de coordenadas con velocidad cero.

Podemos dibujar el retrato fase para este sistema usando prácticamente el mismo programa que usamos para dibujar el de la caída libre, aunque ahora usaremos la solución explícita que acabamos de encontrar para cada condición inicial,

```
In[121]:= x[t_, x0_, v0_] := t + (1 - v0) (Exp[-t] - 1) + x0
```

```
In[122]:= v[t_, v0_] := 1 - (1 - v0) Exp[-t]

In[123]:= OrbitasFaseConResistencia[{tMin_, tMax_},
            {xMin_, xMax_, Δx_}, {vMin_, vMax_, Δv_}] := Table[
            ParametricPlot[ Evaluate[{x[t, x0, v0], v[t, v0]}],
             {t, tMin, tMax} ,
              PlotStyle → Hue[ 9/10 (x0 - xMin)/(xMax - xMin) , 1/2 + 1/2 (v0 - vMin)/(vMax - vMin), 1 ],
              DisplayFunction → Identity],
            {v0, vMin, vMax, Δv}, {x0, xMin, xMax, Δx}
           ]

In[124]:= orbitasConResistencia =
            OrbitasFaseConResistencia[{0, 5}, {0, 2, 0.5}, {0, 5, 1}];
          Show[ Flatten[orbitasConResistencia],
           DisplayFunction → $DisplayFunction, Frame → True,
           FrameLabel → {"x/v_Tτ", "v/v_T"}, PlotRange → All]
```

Out[125]=



¿Bajo qué circunstancias serán parecidas las dos estimaciones que se pueden hacer para h? Es decir, ¿bajo que condiciones es

$$x_0 + v_0\, t + \tfrac{1}{2} g\, t^2$$

una buena aproximación a

$$x(t) = x_0 + \; v_T \; t \; + (v_T - v_0)\, \tau \left(e^{-t/\tau} - 1\right)?$$

Una manera de averiguarlo es recordar que τ es un tiempo característico, que nos indica cuanto tiempo es necesario esperar antes de que se noten los efectos de la fricción del aire. Por lo tanto, conviene examinar el límite de nuestra segunda expresión en el límite que $t \ll \tau$. Cuando se cumple esta última condición, el argumento de la exponencial en $x(t)$ es un número pequeño, por lo que el valor de la función exponencial puede aproximarse por los primeros términos de su serie de Taylor: $\exp(z) \approx 1 + z + \frac{1}{2} z^2$.

```
In[126]:= Simplify[ Series[ x0 + vT t + (vT - v0) τ (ℯ^(-t/τ) - 1), {t, 0, 2}]]
```

$$\text{Out[126]= } x_0 + v_0\, t + \frac{(-v_0 + v_T)\, t^2}{2\,\tau} + O[t]^3$$

Si $v_0 \ll v_T$, entonces

$$x(t) \approx x_0 + v_0\, t + \frac{v_T}{2\,\tau}\, t^2$$

Como $v_T = m\,g/k$ y $\tau = m/k$, la aproximación de caída libre es apropiada sólo cuando $v_0 \ll m\,g/k$. Esta condición simplemente significa que para que la aproximación sea válida, la fuerza de gravedad $m\,g$ debe ser mucho mayor que la fuerza de resistencia inicial $k\,v_0$.

```
In[127]:= x[t, 0, v0]
Out[127]= t + (-1 + ℯ^(-t)) (1 - v0)
```

Finalmente, repetimos nuestro programa que genera una gráfica y una tabla a partir de nuestras soluciones. Para facilitar la comparación con el caso de caída libre, en nuestro programa generamos una tabla con tres columnas: el tiempo, la distancia de caída libre y la distancia de caída con fricción. De manera parecida, a la hora de hacer la tabla graficamos la trayectoria de caída libre (usando un estilo de línea a trazos) y la trayectoria de caída con fricción (con estilo de línea roja),

```
In[128]:= CaidaConFriccion[v0_, {tMin_, tMax_, Δt_}] :=
 Module[{tablaDeAlturas, grafica},
  tablaDeAlturas = Table[
   {
    PaddedForm[t, {3, 2}],
    PaddedForm[1/2 t^2 + v0 t, {6, 2}],
    PaddedForm[x[t, 0, v0], {6, 2}]
   }, {t, tMin, tMax, Δt}];
   TableForm[ tablaDeAlturas,
  TableHeadings → {None, {"t/τ", "h_libre/vT τ" , "h/vT τ"}}]
 ]
```

```
In[129]:= PlotCaidaConFriccion[v0_, {tMin_, tMax_, Δt_}] :=
           Module[{tablaDeAlturas, grafica},

            grafica = Plot[{1/2 t^2 + v0 t, x[t, 0, v0]}, {t, tMin, tMax},
              Frame → True, FrameLabel → {"t/τ", "h/v_T τ"},
              PlotStyle → {Dashing[{0.012, 0.012}], Hue[0]}]
           ]
```

Volviendo por fin al problema de la profundidad del pozo, las siguientes ejecuciones de nuestro programa muestran la profundidad del pozo como función del tiempo hasta oír el chapuzon. El primer diagrama y la primera tabla corresponden al caso de que la piedra tenga una una velocidad inicial pequeña comparada con la velocidad terminal. Se observa que por un rato (0.5 τ) la solución prácticamente coincide con la de una caída libre.

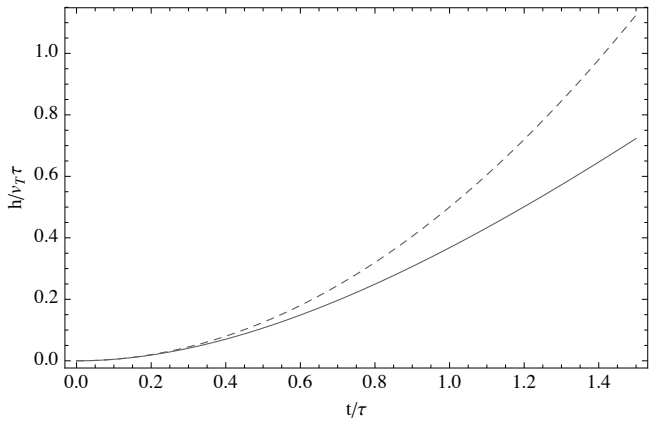
```
In[130]:= CaidaConFriccion[0, {0.0, 1.5, 0.1}]
```

Out[130]//TableForm=

t/τ	$h_{libre}/v_T\tau$	$h/v_T\tau$
0.00	0.00	0.00
0.10	0.01	0.00
0.20	0.02	0.02
0.30	0.05	0.04
0.40	0.08	0.07
0.50	0.13	0.11
0.60	0.18	0.15
0.70	0.25	0.20
0.80	0.32	0.25
0.90	0.41	0.31
1.00	0.50	0.37
1.10	0.61	0.43
1.20	0.72	0.50
1.30	0.85	0.57
1.40	0.98	0.65
1.50	1.13	0.72

In[131]:=
```
PlotCaidaConFriccion[0, {0.0, 1.5, 0.1}]
```

Out[131]=



Sin embargo, la solución con $v_0 > v_T$ rápidamente se aparta de la de caída libre (en 0.2 τ)

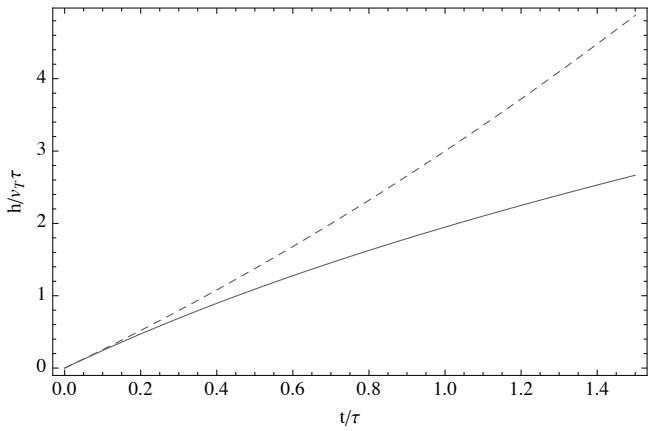
In[132]:=
```
CaidaConFriccion[2.5, {0, 1.5, 0.1}]
```

Out[132]//TableForm=

t/τ	$h_{libre}/v_T\tau$	$h/v_T\tau$
0.00	0.00	0.00
0.10	0.26	0.24
0.20	0.52	0.47
0.30	0.80	0.69
0.40	1.08	0.89
0.50	1.38	1.09
0.60	1.68	1.28
0.70	2.00	1.46
0.80	2.32	1.63
0.90	2.66	1.79
1.00	3.00	1.95
1.10	3.36	2.10
1.20	3.72	2.25
1.30	4.10	2.39
1.40	4.48	2.53
1.50	4.88	2.67

```
In[133]:= PlotCaidaConFriccion[2.5, {0, 1.5, 0.1}]
```

Out[133]=



Ejercicios

1. Encontrar 20! utilizando exclusivamente variables auxiliares, la operación de multiplicación y un ciclo **Do**.

2. Encontrar 20! utilizando exclusivamente variables auxiliares, la operación de multiplicación y un ciclo **For**.

3. Escribir un código que pueda calcular el área bajo una curva y poner todos los pasos dentro del comando **Module**.

4. Imprimir todos los enteros del 1 al 1000 que no son múltiplos de de 2, 3 o 5.

5. Para cada entero n de 1 a 10, inclusive, imprimir $n/2$ si el número n es par, y $2n$ si es impar.

6. Con un ciclo **Do** encontrar la suma de los números primos de 1 a 250.

7. Con un ciclo **For** encontrar la suma de los 50 primeros números de Fibonacci.

8. Para los enteros del 1 al 20, inclusive, formar una tabla de dos columnas con los siguientes elementos: en la primera columna, los números que son primos y en la segunda aquellos números que no lo son.

9. Hacer un programa que calcule el promedio de 100 números enteros aleatorios entre 1 y 100, inclusive. ¿Cuánto esperaría que fuese el valor de dicho promedio?

10. Imprimir una tabla de dos columnas, en la primera de ellas los dígitos (0 ... 9) y en la segunda las frecuencia con la que aparece dicho dígito entre las primeras 1000 cifras decimales de π.

Apéndice
Generación de datos sintéticos para explicar ajustes por mínimos cuadrados

Primero le preguntamos a *Mathematica* que nos indique cuál es el directorio de trabajo actual, posteriormente podemos definir este u otro si así lo queremos. A continuación definimos el directorio de trabajo el mismo que se tiene por omisión:

```
In[134]:= dir = Directory[]
Out[134]= /Users/leo
In[135]:= SetDirectory[ dir]
Out[135]= /Users/leo
```

Calcularemos la variable dependiente con una parte sistemática (una recta) y una parte aleatoria (ruido Gaussiano, con media 0 y desviacion estándar 2.5),

```
In[136]:= y[x_] := 10 + 2.5 x + RandomReal[NormalDistribution[0, 2.5]]
```

A continuación se muestra el programa para calcular y guardar en archivos las *n* series de datos,

```
In[137]:= SaveSeries[n_] :=
           Module[{data},
             data = Table[ {x, y[x]}, {x, 0, 10, 0.5}];
            Export["serie" <> ToString[n] <> ".dat", data]
           ]
```

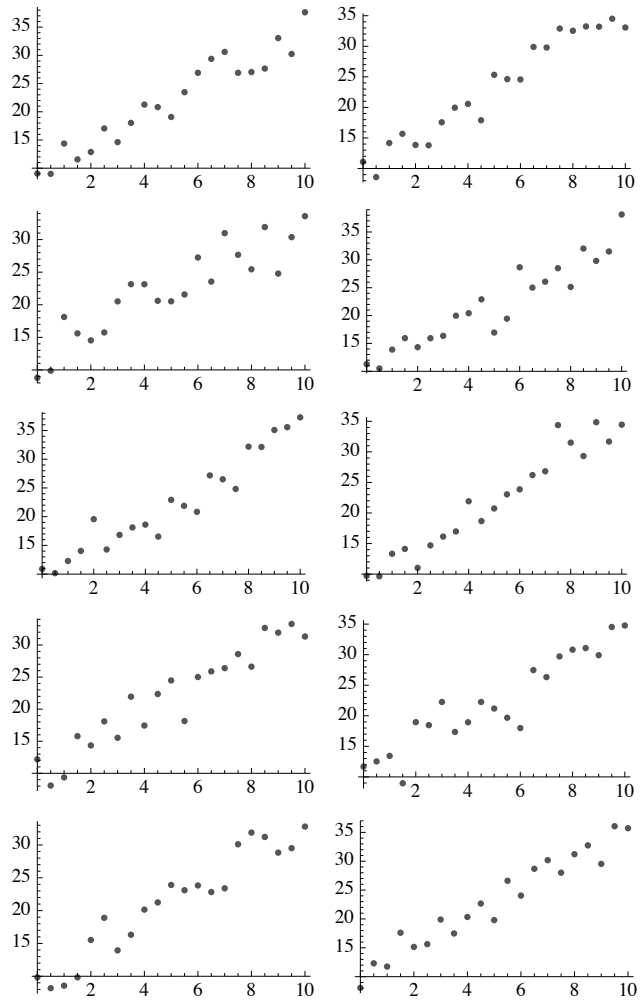
Con este comando guardaremos 10 series de datos sintéticos para ajuste. Los resultados, siendo aleatorios, cambiarán de una ejecución a otra, pero la tendencia será parecida,

```
In[138]:= Table[ SaveSeries[i], {i, 1, 10}]
Out[138]= {serie1.dat, serie2.dat, serie3.dat, serie4.dat, serie5.dat,
           serie6.dat, serie7.dat, serie8.dat, serie9.dat, serie10.dat}
```

Finalmente leemos y graficamos las series recién generadas. La mayor parte presenta una tendencia lineal apreciable todavía,

In[139]:=
```
GraphicsArray[ Partition[
  Map[ ListPlot[ Import[#] ] &, FileNames["serie*.dat"]],
  2]
]
```

Out[139]=



Índice

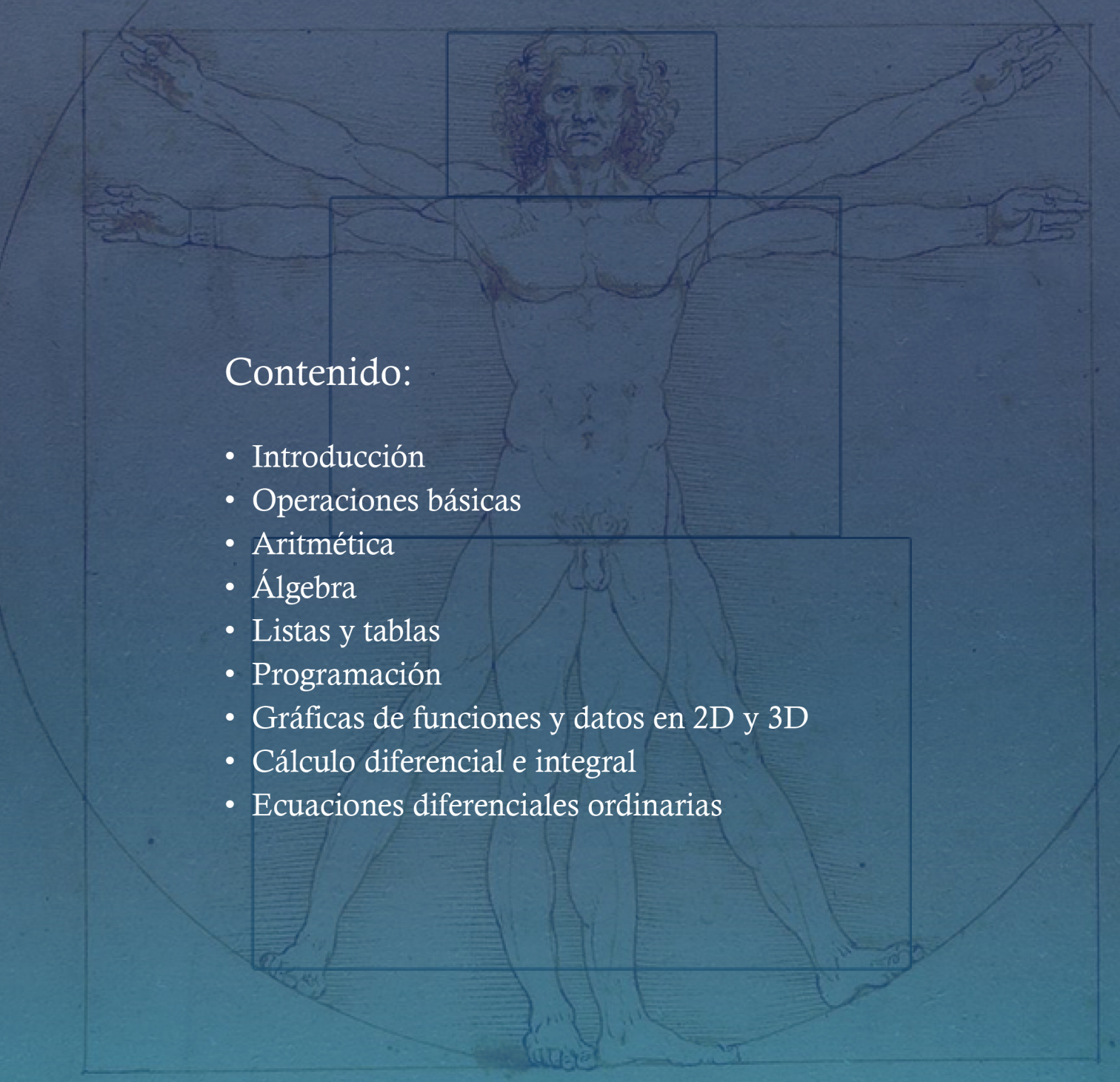
Mathematica esencial
Universidad Autónoma Metropolitana
Unidad Iztapalapa

Departamento de Física
División de Ciencias Básicas e Ingeniería

Impreso en los talleres de AFA Impresos, Canal Nacional No. 59,
Col. San Andrés Tomatlán, Del. Iztapalapa, C. P. 98720, México, D. F.,

La edición consta de 500 ejemplares

Mayo de 2016



Contenido:

- Introducción
- Operaciones básicas
- Aritmética
- Álgebra
- Listas y tablas
- Programación
- Gráficas de funciones y datos en 2D y 3D
- Cálculo diferencial e integral
- Ecuaciones diferenciales ordinarias

ISBN 978-607-477-240-1