CAPITULO 1 INTRODUCCIÓN

1.1. MODO INTERACTIVO Y PROGRAMAS

Cuando en *Mathematica* escribimos una operación, comando, etc., y le pedimos que lo evalúe, para usar el resultado en cálculos posteriores, estamos trabajando con el ordenador de un modo *interactivo*. Sin embargo, si tenemos que resolver algún problema que implique realizar cálculos de cierta complejidad, es posible que tengamos que realizar la misma operación varias veces, variando solamente los datos.
Así, por ejemplo, si tenemos que hacer varias operaciones como las siguientes:

$2+2^2+2^3+2^4$
$3+3^2+3^3+3^4+3^5$

podríamos ahorrar tiempo si definimos una función tal que, dados dos enteros positivos k y n, calcule la suma

$1+n^2+.....+n^k$

Esto se puede hacer, por ejemplo, definiendo la función

```
SumaPotencias[n_, k_] := Sum[n^{i}, {i, 1, k}]
```

Entonces comenzaremos a escribir *programas*.
Además de ser una calculadora númerica y simbólica con la que podemos trabajar interactivamente, *Mathematica* es también un lenguaje de programación de alto nivel de gran potencia. A diferencia de otros lenguajes como Pascal o C, no está restringido a un número reducido de tipos de datos, lo que permite que podamos realizar todo tipo de cálculos simbólicos.

Mathematica soporta varios estilos de programación:

1. Programación tradicional, con procedimientos, distintos tipos de bucles e iteraciones, condicionales, recursividad, etc. En este sentido es muy similar a lenguajes de programación como C o PASCAL.

2. Programación funcional, con funciones puras, operadores funcionales y listas. Muy similar a como trabaja LISP.

3.Programación basada en reglas de transformación, con modelos y orientación al objeto, tendencia que siguen actualmente muchos de los lenguajes de programación, como C++.

De estos tres tipos de programación debemos elegir la que más se adecúe a nuestros propósitos. Sin embargo, debido a la naturaleza simbólica de *Mathematica*, la programación funcional y la programación basada en reglas de

transformación dan lugar a programas más eficientes y más fáciles de entender.

Cuando programamos se pretende, además de resolver el problema, que el programa sea comprendido por una persona que no lo ha escrito y fácilmente optimizable. La programación estructurada se basa en la división del programa en módulos, de modo que cada módulo sea lo más independiente posible de los demás. Un módulo debe tener un tamaño máximo de dos folios en código fuente, si se escribe una instrucción por línea.

En la programación estructurada se pretende que nuestro programa tenga un diseño TOP-DOWN, a excepción de que haya un bucle; con todo esto se pretende evitar la programación "spaghetti" que origina GOTO y que inicialmente se usaba en BASIC. De este modo al dividir el programa en subprogramas, éste se entiende más fácilmente y es más sencillo depurar errores, permitiendo además utilizar estas subrutinas en programas posteriores.

Los programas escritos en *Mathematica* se llaman **paquetes** (en inglés, packages), y son archivos de disco que tienen extensión con nombres de la forma nombre.m. Más adelante hablaremos de la diferencia entre los paquetes y los libros de notas (notebooks), cuya extensión es nb.

1. 2. VARIABLES LOCALES Y GLOBALES

Con la noción de estructura en los programas surge también la necesidad de variables locales a los bloques, variables que son conocidas para el bloque particular, pero no para el programa principal.
Consideremos la función **SumaPotencias** definida en la sección anterior y vamos a realizar con ella las operaciones siguientes:

```
SumaPotencias[x, 5]
SumaPotencias[i, 5]
```

$$\{x + x^2 + x^3 + x^4 + x^5\}$$

$$\{3413\}$$

El resultado de la primera operación es el esperado, pero la segunda nos dará el resultado 3413, en lugar de $i+i^2+i^3+i^4+i^5$

El motivo de este extraño comportamiento de la función es que la variable **i** que sirve de índice de la suma y la que hemos usado como argumento de la función son la misma, por lo que el resultado que obtenemos es precisamente

$1+2^2+3^3+4^4+5^5=3413$.

Para evitar este tipo de errores, en *Mathematica* disponemos de las funciones **Block** y **Module**, cuya sintaxis es la siguiente:

Module[{x, y, ... }, cuerpo]
Block[{x, y, ... }, cuerpo]

Module crea variables locales que se pierden cuando termina **Module**, mientras que **Block** hace locales los valores de las variables; cuando termina **Block** las variables recuperan los valores globales que tuvieran.

1.2.1. ¿ Cómo funciona Module?

La variable **$ModuleNumber** del sistema cuanta el número de veces que **Module** es usado. Cada vez que se usa **Module** para cada una de las variables declaradas como locales se crea un nuevo símbolo que representa a cada una de las variables. El nuevo nombre se toma del especificado seguido de $ y por un número de serie que se toma de **$ModuleNumber**.
Veamos algunos ejemplos:

■ El resultado de

```
Module[{t}, Print[t]]
```

```
t$1
```

■ Así pues, podríamos haber evitado el error de la función **SumaPotencias** si la hubiésemos definido como

```
SumaPotencias[n_, k_] := Module[{i},
          Sum[n^{i}, {i, 1, k}]
         ]
```

```
SumaPotencias[i, 5]
```

$$\{i + i^2 + i^3 + i^4 + i^5\}$$

■ Cuando usamos la función **Integrate** necesitamos una variable de integración con un nombre concreto, y dicha variable no debe entrar en conflicto con otras variables que tengamos definidas. Si definimos la función

```
p[n_] := Integrate[f[s] s^n, {s, 0, 1}]
```

y evaluamos

```
p[s + 1]
```

$$\int_0^1 s^{1+s} f[s] \, ds$$

habrá un conflicto entre las dos variables **s**. Esto se podría haber evitado definiendo

```
p[n_] := Module[{s}, Integrate[f[s] s^n, {s, 0, 1}]]
```

ya que en ese caso el resultado de evaluar **p[s+1]** sería

```
p[s + 1]
```

$$\int_0^1 s\$19^{1+s} f[s\$19] \, ds\$19$$

1.2.2. Block

Block no crea nuevas variables cuando estamos usando alguna que ya existía en el programa principal. Simplemente hace local el *valor* de dicha variable.
■ Compárense los resultados de las dos operaciones siguientes:

```
m = i^2;
Block[{i = a}, i + m]
Module[{i = a}, i + m]
```

$a + a^2$

$a + i^2$

Como se ve, la primera da como resultado a+a^2, porque **Block** usa la variable **i** que habíamos definido antes, mientras que **Module** crea una nueva, y el resultado de la segunda operación es a+i^2.

■ En versiones de *Mathematica* anteriores a la 2.0, **Block** era la única función de la que se disponía para trabajar con variables locales, ya que **Module** no existía. Actualmente los programadores en *Mathematica* utilizan normalmente **Module**, reservando **Block** para usos especiales, como puede ser la modificación temporal de los valores de alguna variable del sistema; si, por ejemplo, estamos usando algún algoritmo que necesite más recursividad de la permitidapor defecto, para evitar errores podríamos escribir algo así:

```
func[x_,y_,...]:=
  Block[{$RecursionLimit = Infinity},
    Module[{a,b,...},
      (* procedimiento muy recursivo *)
    ]
  ]
```

■ El mecanismo que usa **Block** para manejar las variables locales es el mismo que se usa en los procesos iterativos de *Mathematica*. La razón de que dichos procesos usen implícitamente **Block** y no **Module** es que muchas veces queremos operar con expresiones que dependen de una variable y queremos que ésa sea precisamente la variable de la iteración. Si, por ejemplo, queremos calcular $\sum_{k=1}^{10}$ k+k^2, podemos proceder del modo siguiente:

```
expr = k + k^2;
Sum[expr, {k, 1, 10}]
```

440

Si *Mathematica* usase para este proceso una nueva variable **k$1** distinta de **k**, el resultado de la operación anterior sería 10k+10k^2 en lugar de 440.

1.3. CONTEXTOS

Sí a una función le damos el nombre **f**, al pasar cierto tiempo desde que la hayamos definido no sabremos como funciona salvo que miremos su definición, pues su nombre no nos da ninguna información sobre ella. Por este motivo es siempre una buena idea dar a las variables y a las funciones nombres tan explícitos como sea posible; pero si queremos que el nombre nos lo diga casi todo esto puede resultar largo y engorroso.

Por esta razón existe en *Mathematica* la noción de **contexto**. La idea es que el nombre de cualquier símbolo se compone de dos partes: el nombre del contexto en el que está definido y el "nombre corto" del símbolo, en la forma

```
nombre de contexto`nombre de simbolo
```

Lo normal es tener todos los símbolos que se refieren a un mismo tema dentro del mismo contexto, dentro del cual podemos referirnos a ellos por su nombre corto.

Para entender cómo funcionan los contextos podemos compararlos con la estructura de archivos del disco del MS-DOS (o las carpetas del Windows o del sistema operativo de Macintosh). Para no tener todos los archivos mezclados, los distribuimos en directorios dentro del disco; en cada directorio están los archivos que tienen que ver con un mismo tema. Así, podemos tener en un disco C: directorios como c:\dos, \c:\windows,c:\word, etc., y cada directorio puede tener a su vez subdirectorios. Si un archivo está en el directorio c:\word\cartas y tiene el nombre alfredo.doc, su nombre completo será c:\word\cartas\alfredo.doc. Dentro de directorios diferentes puede haber archivos distintos con el mismo nombre, ya que sus nombres largos son diferentes.

Los contextos en *Mathematica* tienen una estructura muy similar a ésta. Todos los símbolos que definimos están dentro de un contexto, llamado **Global`**, dentro del cual podemos definir subcontextos para incluir dentro de ellos nuestras variables y funciones, de modo que no se mezclen unas con otras.

La variable del sistema **$Context** tiene como valor el contexto dentro del cual estamos trabajando en ese momento.

Siguiendo con el ejemplo anterior, así como en MS-DOS tenemos la variable **PATH** que dice al sistema operativo en qué directorios ha de buscar los archivos, en *Mathematica* existe la variable **$ContextPath**, cuyo valor es la lista de los contextos en los que tiene que buscar los símbolos.
El valor por defecto de **$ContextPath** es **{Global`,System`}**. Dentro del contexto **System`** están todos los símbolos que el *Mathematica* ya tiene definidos (variables y funciones del sistema). Dentro del contexto **Global`** todas las variables y funciones que nosotros definimos a lo largo de una sesión.

Los siguientes comandos son útiles para manejar contextos:

Context[simb] devuelve el contexto del símbolo simb
Begin["cont`"] cambia al contexto cont
End[ ] vuelve al contexto anterior

1.4. PAQUETES

Como ya hemos dicho, los programas escritos en *Mathematica* se llaman **paquetes**. Generalmente, cada paquete contiene símbolos (funciones, variables, etc.) para ser utilizados desde fuera del paquete. Los paquetes de *Mathematica* son lo equivalente a las librerias de C o de otros lenguajes de programación. La estructura de un paquete es la siguiente:

```
BeginPackage["nombre`"]
.....(programa)
EndPackage[]
```

Todos los símbolos definidos en el paquete son puestos en un contexto cuyo nombre es el del paquete; cuando se lee el paquete, su nombre se añade a **$ContextPath**.

Recuerda que para leer un paquete hay dos posibilidades:

```
Needs[nombre del paquete]
ó
<<nombre del paquete
```

1.5. RELACIÓN ENTRE PAQUETES Y LIBROS DE NOTAS

1.5. RELACIÓN ENTRE PAQUETES Y LIBROS DE NOTAS

El programa *Mathematica* consta, como sabemos, de dos partes: el núcleo *(kernel)*, que es el que se encarga de realizar los cálculos, y la presentación *(front end)*, que es la parte del programa mediante la cual el usuario se comunica con el núcleo. La presentación de *Mathematica* para Windows y para el entorno System 7 de Macintosh se basa en lo que llamaremos *libros de notas*. Los libros de notas pueden contener, además de lo que normalmente hay en un paquete (código que pueda entender el núcleo de *Mathematica*), texto y gráficos, lo que permite mejorar sustancialmente tanto la entrada de los datos como la presentación de los resultados.

Así como la extensión de los nombres de los paquetes en el disco es **.m**, la de los libros de notas es: **.nb** en la versión 3.0, **.ma** en la versión 2.0, **.mat** en la versión 2.1, para que podamos distinguir fácilmente unos de otros.

Es muy sencillo transformar un paquete en un libro de notas: basta con abrirlo desde el programa en Windows o Macintosh y grabarlo otra vez en disco.
El proceso de conversión de un libro de notas en paquete es algo más complicado: hay que borrar todo lo que haya en el libro de notas que haga referencia a gráficos y presentación y dejar exclusivamente código fuente del núcleo de *Mathematica*.

1.6. EJERCICIOS

1.6. EJERCICIOS

■ 1
El algoritmo de Newton es un método iterativo para la busqueda de soluciones de ecuaciones. A continuación tienes definida una función NewtonIte que calcula una solución aproximada para f=0, fijando el número de iteraciones como argumento de la función.

```
NewtonIte[eq_, ci_, ite_] :=
Module[{x0 = ci, i, x1 = ci, a},
   For[i = 0, i < ite, i++,
     x0 = x1;
      a = (D[eq, x] /. {x -> x0});
     If[a != 0, x1 =, N[x0 - (eq /. {x -> x0})/a],
       Return["ERROR: se anula la derivada de
                    la funcion"]
     ]];
    Print["solucion aproximada:"];
    Print[x1];
    Print["error:"];
    Print[Abs[x1 - x0]];
    x1
   ]
```

Implementa el algoritmo de Newton fijando un error máximo permitido, en vez del número de iteraciones. Observa que ahora tienes una condición de parada para el bucle; el comando **While** te servirá

■ 2.
A continuación responde a las siguientes preguntas sobre contextos:
• ¿Cuál es el contexto actual?

• ¿Qué contextos son conocidos por el programa en este momento?

•¿Cuál es el contexto de Sin?

•Cambia el contexto actual al contexto de nombre hoy

•¿cuál es ahora el contenido de $ContextPath?

•¿cuál es el contexto de x?

■ 3.
Desde el cuadro de diálogo del menú Open abre el siguiente paquete: **Calculus`VectorAnalysis`**. (Este paquete contiene lo necesario para hacer cálculo vectorial en R^3; lo usaremos más adelante). Observa su estructura:

```
BeginPackage[Calculus`VectorAnalysis`]
(*funcionamiento de las
funciones alli descritas*)
Begin["Private`"]
(*Definicion de las funciones del
paquete *)
Protect[funcion] (*se protegen las funciones del paquete*)
End[](*terminamos el contexto Private*)
EndPackage[](*terminamos el contexto del paquete*)
```

Copiando la estructura del paquete anterior construye en tu disco un paquete que se llame Newton , donde incluiras las dos funciones Newton construidas anteriormente.

CAPITULO 2

CAPITULO 2
PROGRAMACIÓN TRADICIONAL

2.1. BUCLES E ITERACIONES

2.1. BUCLES E ITERACIONES

Los bucles e iteraciones son fundamentales en muchos lenguajes de programación. En *Mathematica*, por similitud con lenguajes como el Pascal y el C, se puede programar usando bucles, aunque, como veremos más adelante, su uso se puede evitar casi siempre.

Hay dos tipos de bucles: aquellos en los que una instrucción o grupo de instrucciones es repetido un número fijo de veces y otros en los que dichas instrucciones se repiten mientras cierta condición sea satisfecha; tenemos el ciclo **Do** dentro del primer tipo y **While** y **For** en el segundo.