■ 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.

2.1.1. EL CICLO DO

La estructura más general del ciclo **Do** es la siguiente:

Do[instrucciones,{i,i1,i2,d}]

donde
i es la variable que controla el bucle, que consiste en repetir las instrucciones, con el valor inicial i1 para la variable i hasta que i=i2, donde cada vez i se incrementa en d. Se puede suprimir el valor inicial i1 (en cuyo caso se supone igual a 1) y el incremento d (que por defecto también es 1).
La estructura de este bucle es similar a la del DO de FORTRAN o Basic y a la del For del Pascal.

Ejemplos:

1) Podemos escribir los cinco primeros números primos usando la instrucción

```
Do[Print[Prime[i]], {i, 1, 5}]
```

```
2

3

5

7

11
```

o simplemente

```
Do[Print[Prime[i]], {i, 5}]
```

```
2

3

5

7

11
```

2) La función **fibonacci** que definimos a continuación calcula el n-ésimo número de Fibonacci, para un valor de n dado;
En este caso suprimimos incluso la variable que controla el bucle.

```
fibonacci[n_] :=
 Module[{fn1 = 0, fn2 = 1},
  Do[{fn1, fn2} = {fn2, fn1 + fn2}, {n - 1}];
  fn1
 ]
```

```
General::spell1 :
 Possible spelling error: new symbol name "fibonacci" is similar
   to existing symbol "Fibonacci". More…
```

```
fibonacci[7]
```

```
8
```

■ 3) Tanto los valores de la variable que controla el bucle como el incremento pueden ser simbólicos en lugar de números:

```
Do[Print[i], {i, 2 a, 4 a, a}]
```

```
2 a

3 a

4 a
```


■ 2.1.2. EL CICLO WHILE

La instrucción

While[condición,proceso]

realiza el proceso especificado mientras la condición sea cierta. Si la primera vez la condición es falsa, el proceso no se realiza ninguna vez. Es similar al ciclo **While** de Pascal y C.

■ Ejemplo: Consideremos las instrucciones siguientes:

```
n = 117;
While[PrimeQ[n] == False, n++]

n
```

```
127
```

Después del bucle, el valor de **n** será el primer número primo mayor que 117, que en este caso es 127.
Si hacemos

```
n = 127;
While[PrimeQ[n] == False, n++];
n
```

```
127
```

En este caso el proceso no se realiza ninguna vez, porque 127 es primo

■ 2.1.3. EL CICLO FOR

• Es similar al ciclo **For** del C y no tiene equivalente en los lenguajes de programación más conocidos. Su sintaxis es

For[c0,condición,incremento,proceso]

Esta instrucción evalúa la condición inicial **c0** y luego la **condición**; si ésta es cierta, evalúa el proceso y luego el incremento, donde aquí **incremento** significa simplemente una o varias instrucciones que modifican la condición inicial **c0**. Luego vuelve a evaluar la condición y se repite lo anterior hasta que ésta sea falsa.

• Se puede sustituir por **While**, porque lo anterior es equivalente a

co;While[condición,proceso;incremento]

• Entre los programadores en C y *Mathematica* es mucho más habitual usar el ciclo For que el Do, dada su gran flexibilidad. La instrucción Do[proceso,{i,1,n}] se suele sustituir por

```
For[i=1,i<=n,i++,proceso]
```

■ Ejemplo: Calculamos la suma de los 10 primeros naturales

```
ac = 0; n = 10;
For[i = 1, i <= n, i++, ac += i];
ac
```

```
55
```

La instrucción ac+=i es equivalente a ac=ac+i.
Las variables del tipo de ac se suelen denominar acumuladores, y las del tipo de i , contadores

■ 2.1.4. EL CICLO UNTIL

A diferencia del Pascal, que posee el ciclo **Repeat...Until**, en *Mathematica* no hay un ciclo similar al While, pero que evalúe la condición al final. Si se quiere, este ciclo se puede construir usando While o For. Una instrucción

```
Until[proceso,test]
```

que ejecute el proceso hasta que el test dé el resultado True es equivalente a

```
While[True,Proceso;if[test,Break[]]
```

y también a

```
For[t=False,!t,t=test,proceso]
```


2.2. CONDICIONALES

La evolución de un programa está controlada por las funciones condicionales.
Estos comandos tienen como principal misión realizar tareas paralelas en las programas según se cumplan o no ciertas condiciones.

2.2.1. ¿Qué entiende Mathematica por condición?

2.2.1. ¿Qué entiende *Mathematica* por condición?

• Una condición es una **expresión booleana**, es decir, una expresión cuya evaluación puede dar como resultado True o False.

• Debemos tener cuidado porque *Mathematica* no es un lenguaje restrictivo a la hora de definir expresiones booleanas. Cualquier expresión simple (formada por una única sentencia) puede ser considerada como una expresión booleana dando un resultado sin producir error.

• En este sentido se entiende que sólo un grupo especial de expresiones serán susceptibles de producir un valor **True** al ser evaluadas como booleanas; por ejemplo las expresiones que contengan relaciones de pertenencia, igualdad, desigualdades, etc. El resto de las expresiones simplemente producirán un resultado **False**.

• *Mathematica* dispone de un comando para evaluar expresiones booleanas.
Este comando es

TrueQ[expresion] devuelve True si expresión es cierta y False si expresión es falsa

```
TrueQ[7 < 8]
```

```
True
```

```
TrueQ[x^2 + 7 x]
```

```
False
```

Mathematica considera como falsa cualquier expresión que no pueda ser evaluada a True o False.

• Los operadores lógicos y relacionales más corrientes son:

x==y	igual lógico
x!=y	distinto
x>=y	mayor o igual que
x>y	mayor que
!p	no p
p&&q	conjuncción lógica
p‖q	disyuncción lógica

Es aconsejable cuando se usen estos operadores la utilización de paréntesis para indicar prioridad la evaluación

•En *Mathematica* disponemos de tres condicionales: **If, Which , Switch**.

2.2.2. EL CONDICIONAL If

2.2.2. EL CONDICIONAL If

La sintaxis de **If** es:

If[condición,proceso1,proceso2,proceso3]

Esta instrucción realiza el primer proceso si la condición es cierta, el segundo si es falsa y el tercero si no se sabe; se pueden omitir los dos últimos procesos

EL CONDICIONAL Which

2.2.3. EL CONDICIONAL Which

En los casos en que sea necesario controlar el programa por más de una condición se dispone del siguiente comando:

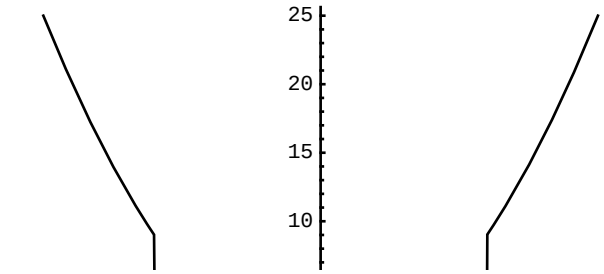
Which[condición1,proceso1,.....condiciónn,proceson]

que evalúa consecutivamente las condiciones 1,2...,n hasta que encuentra una cierta y realiza el proceso asociado.

• Poniendo True como última condición se consigue que devuelva la última expresión si ninguna de las condiciones ha resultado cierta.

• **Which** se puede utilizar para definir funciones a trozos:

```
f[x_] := Which[-2 <= x <= 2, x^2, -3 <= x <= 3, 4, True, x^2]
Plot[f[x], {x, -5, 5}]
```




```
- Graphics -
```

2.2.4. EL CONDICIONAL Switch

2.2.4. EL CONDICIONAL Switch

La sintaxis para este comando es la siguiente:

Switch[exp,form1,value1,form2,value2,...]

form1, form2.....son "plantillas" o modelos para expresiones. Las estudiaremos de modo detallado en el capítulo 4.

Se compara **exp** con **form1**, si el resultado es **True** , **Switch** devuelve **value1**, si el resultado es **False**, compara **exp** con **form2**, así sucesivamente.

• Es posible añadir un último argumento que se devuelve en el caso de todos **False**.

Switch[exp,form1,value1,form2,value2,...,_,def]

Ejemplo:

```
Switch[17, 0, a, 1, b, _, q]
```

```
q
```

da como resultado q ya que 17, no es ni 0 ni 1, y por tanto se devuelve el valor por defecto.

■ Ejemplo:

```
f[x_] := Switch[x, _Real, x^2, _Complex, 2 I]
```

_Real es una plantilla para los números reales, _Complex es una plantilla para los números complejos.

```
f[2.0]
```

```
4.
```

```
f[2]
```

```
Switch[2,
_Real, 2^2,
_Complex, 2 i]
```

No devuelve 2^2 porque 2 no se adapta al modelo _Real sino _Integer

```
f[3 I]
```

```
2 I
```


2.3. ESTRUCTURAS DE CONTROL DE FLUJO

A veces conviene salir de un bucle antes de terminar de ejecutarlo, si bien no es muy recomendable hacerlo sistemáticamente. Para ello *Mathematica* incorpora funciones de control del flujo, similares a las de otros lenguajes como el C.
Estas funciones son:

Break[]	sale del bucle más interno
Continue[]	continua en el siguiente paso del bucle
Return[expr]	devuelve exp y sale de todos los bucles
Goto[nombre]	va a la instrucción marcada con Label[nombre]

• Sólo se puede usar Goto cuando el Label especificado ocurre dentro del mismo procedimiento; es por esto habitual encerrar un Goto y su Label correspondiente entre paréntesis, para indicar que forman parte de un mismo procedimiento. En general el uso de Goto reduce el grado de estructura que se puede percibir fácilmente en un programa, y hace que este sea más dificil de entender. Se recomienda su uso sólo en aquellos casos en que sea la única posibilidad.

■ Ejemplo 1: La función raiz que definimos a continuación comprueba si un número es positivo, en cuyo caso nos da su raíz cuadrada, o negativo, y en este caso nos da un mensaje de error.

```
raiz[x_] :=
 If[x < 0, Return["numero negativo"], Return[x^(1 / 2)]]
```

```
raiz[-3]
```

```
numero negativo
```

■ Ejemplo2:

```
(q = 2; Label[hola]; Print[q]; q += 3; If[q < 6, Goto[hola]])
```

```
2
5
```


2.4. FUNCIONES DE ENTRADA Y SALIDA

A lo largo de una sesión o del desarrollo de un programa puede ser necesario que se requiera de algún dato suministrado por el usuario o que bien este necesite algún dato intermedio. Contamos para ello con funciones de entrada y salida de datos. Estas funciones son similares a sus equivalentes en otros lenguajes de programación.

■ 2.4.1. Funciones de entrada

Supongamos que queremos realizar un programa que pidiendo al usuario un número límite calcule los cubos desde 1 hasta el límite, necesitamos entonces una función de entrada que pida al usuario ese número límite. Esta función es

Input[]

Cuando tecleamos Input[ ] aparece un cuadro de diálogo de núcleo de *Mathematica* en que el que se nos requiere un dato.

• Existe la posiblilidad de indicar en Input una frase que haga referencia al tipo de dato requerido, o del uso que de él se va a hacer; escribimos esta frase entre comillas dentro de Input.
■ Ejemplo:

```
l = Input["Teclee un numero limite"];
For[i = 1, i <= l, i++, Print[i^3]]
```

```
1
8
27
64
125
216
343
```

Tomando l=5, escribimos los cinco primeros cubos

■ 2.4.2. Funciones de salida

Hemos visto ya dos funciones de salida de datos

Print[exp] **Return[exp]**

En la mayoria de los casos no necesitaremos usarlas porque *Mathematica* devuelve en pantalla el Out del último In ejecutado. Las usaremos cuando necesitemos de algún dato intermedio. Si queremos escribir alguna frase la escribimos entre comillas, del mismo modo que haciamos para Input.

2.5. EJERCICIOS

NOTA:
Para diseñar un algoritmo seguiremos los siguientes pasos:

a) Sobre el papel se decide la estructura del algoritmo (tenemos una condición de parada para el bucle, conocemos el número de pasos, sería conveniente este u otro condicional....)

b) Formalización del algoritmo para concretar y precisar los pasos básicos.

c) Se prueba el algoritmo para asegurarse de que no tiene errores y en caso de que los tuviese depurarlos. En ningún caso debe variarse sustancialmente la estructura del algoritmo

d) Se pueden intentar algunas mejoras: mejorar la presentación en la entrada y salida de datos, añadir mensajes de información al usuario,...

■ **1.** ¿Es posible en el ejemplo 2 de la sección 2.3. evitar el uso de Goto?

```
(q = 2; Label[hola]; Print[q]; q += 3; If[q < 6, Goto[hola]])
```

```
2
5
```

■ **2.** Realizar un programa que pidiendo al usuario un número límite l, calcule el producto de los l primeros enteros.

■ **3.** Realizar un programa que pidiendo al usuario un número límite calcule los cubos de los números naturales mientras el cubo se mantenga menor que el límite.

■ **4.** Observa los siguientes bucles, ¿se te ocurre otra manera de describirlos?

t=1; Do[t*=k;Print[t];If[t>19,Break[]],{k,10}]

t=1;Do[t*=k;Print[t];If[k<3,Continue[]];t+=2,{k,10}]

■ **5.** Observa el siguiente programa que adivina un número pensado por el usuario del 1 al que el usuario teclee (límite). Para ello pide al usuario que piense un número , después el programa propone un número, al que el usuario contesta 1, si el número propuesto es demasiado alto, 2 si es demasiado alto o 3 si ha acertado, así sucesivamente hasta que acierte. Si el ordenador no acierta antes de $\log_2$(limite) + 1 debe llamar al usuario tramposo.

```
Clear[l];
Clear[i];
Clear[num];
Clear[n];
l = Input["Escribe un numero limite y piensa un
numero entre 1 y ese numero
limite"];
n = Log[2, l];
num = Round[l / 2];
For[i = 1, i <= (n // N) + 1, i++,
 Print[num];
 opcion = Input["escriba 1 si el numero pensado
                es mayor que el propuesto,
                2 si es menor,3 si es igual"];
 If[opcion == 3,
  Print["\n El numero pensado es:"];
  Print[num];
  Break[]
  ];
 num = Round[Which[opcion == 1, num += (l - 1) / 2^(i + 1),
        opcion == 2, num -= (l - 1) / 2^(i + 1)
       ]]
 ]
If[(i == Round[N[n]] + 1) && ((opcion == 1) ||
 (opcion == 2)), Print["tramposo"]]
```

• Observa como está escrito el programa: el uso de tabulaciones permite distinguir claramente los bucles y condiciones anidadas. Esto que puede parecer una trivialidad facilita mucho la lectura y permite de un golpe de vista comprender su estructura.

• Es importante también que notes como en los condicionales usamos ==, este es el igual lógico, distinto de = que usabamos para la asignación de valores a variables.

• Borrar los valores de las variables que vas a usar es una buena costumbre, evita errores si estas tuvieran ya asignado algún valor.

Responde a las siguientes preguntas:

• Trata de explicar cual es el algoritmo de busqueda del número que hemos usado. Prueba con un ejemplo; esto te ayudará.

• ¿Por qué el bucle principal es un bucle For?

• ¿Por qué tenemos un condicional Which?

• ¿Se te ocurre otra forma de implementar el algoritmo?

■ 6. Escribe un programa que pida al usuario un número límite y que calcule todos los primos desde 1 hasta el límite. Para ello ten en cuenta lo siguiente:

•Admite como primos 1 y 3

•Para comprobar si un número es primo debes dividirlo por todos los primos anteriores, si la división es exacta en algún momento ese número no es primo.

• Los pares no son primos

• La función que da el resto de la división de m entre n es Mod[m,n]

• Es conveniente que los primos que vayas obteniendo en cada paso del bucle los vayas acumulando en una lista l. El valor inicial de l es l={3}. Para ir añadiendo primos a la lista usa la función AppendTo.

• Observa que el bucle principal debe ser un bucle For

■ 7. Realizar un programa que devuelva la tabla de amortización mes a mes de un prestamo dinerario. El capital inicial, el interés (rédito), y la cuota mensual se solicitan del usuario mediante la función Input.

■ 8. El método de Newton es un método iterativo para la busqueda de soluciones de ecuaciones. El algoritmo es el siguiente: para calcular una solución de f(x)=0 se parte de una semilla inicial x0, se calcula la recta tangente a la curva y=f(x) en (x0,f(x0)), y se corta dicha recta con el eje de las x's. Se obtiene x1. Luego partiendo de x1 se repite el proceso. Si la función f es lo "suficientemente regular" la iteración del proceso conduce a una raíz de f(x). Realizar un programa que calcule soluciones aproximadas de ecuaciones utilizando como método númerico el método de Newton.

(a) La ecuación, la semilla incial y el número de iteraciones del proceso se solicitan del usuario mediante Input.

(b) La ecuación, la semilla inicial y el error máximo permitido se solicitan al usuario mediante Input.

■ 9. El método de las secantes es un método iterativo para la busqueda de soluciones de ecuaciones que se puede utilizar cuando falla el de Newton porque la derivada es cero en alguna de las iteraciones. El algoritmo es el siguiente: Para calcular una solución de f(x)=0 se parte de dos semillas x0 y x1y se corta la recta que pasa por (x0,f(x0)) y (x1,f(x1))) con el eje de las x's. Se obtiene x2. Luego tomando x0=x1, x1=x2, se repite el proceso. Si la función f es lo suficientemente regular el método converge permitiendonos obtener soluciones aproximadas de f(x)=0.

Realizar un programa que calcule soluciones aproximadas de ecuaciones utilizando como método númerico el método descrito anteriormente.

(a) La ecuación, la semillas inciales y el número de iteraciones del proceso se solicitan del usuario mediante Input.

(b) La ecuación, la semilla iniciales y el error máximo permitido se solicitan al usuario mediante Input.

■ 10. Dada una función f(x) diremos que x0 es un punto fijo de f si f(x0)=x0. Partiendo de un x0 cualquiera e iterando f en x0 (x1=f(x0), x2=f(x1),x3=f(x2),...) se obtiene una aproximación de un punto fijo de f (No decimos nada sobre las condiciones que debe verificar f para que el método converja, cosa que, obviamente, no siempre es cierta).

Realizar un programa que calcule puntos fijos de funciones utilizando como método númerico el método descrito anteriormente.

(a) La función, la semilla inicial y el número de iteraciones del proceso se solicitan del usuario mediante Input.

(b) La función, la semilla inicial y el error máximo permitido se solicitan al usuario mediante Input.

■ 11. El método de la bisección es un método númerico para resolver ecuaciones f(x)=0 en [a,b] de modo aproximado siendo f una función continua en [a,b] y f(a)f(b)<0. Está basado en el teorema de Bolzano:
Evaluamos la función en el punto intermedio c del intervalo [a,b]. Si f(c)=0, c es la raíz buscada, si f(c)≠0 elegimos el subintervalo [a,c] o [b,c] en el f tome valores de signo opuesto. Así, repetimos el proceso. En cada paso c da una mejor aproximación de la raíz.

(a) Programar el método de la bisección: la función f; varmin, el extremo inferior del intervalo; varmax, el extremo superior del intervalo y el número de iteraciones ite se solicitan del usuario mediante Input.

(b) Realizar los cambios necesarios en el programa anterior al solicitar del usuario un error máximo permitido en vez del número de iteraciones.

(c) Introduce los cambios necesarios en (b) para que aparezca el mensaje "convergencia demasiado lenta" si el número de iteraciones es mayor que 100.

■ 12. **El algoritmo de ordenación rápida.** Es un algoritmo de ordenación de listas en el que cada elemento es comparado con uno fijo de la lista, por ejemplo el primero. Si un elemento es menor que el primero, se guarda en una lista que llameremos lista1, y si no se guarda en una lista que llameremos lista2. Con cada una de estas listas se repite el proceso hasta que en las sucesivas listas sólo quede un elemento.

■ 13. **Ordenación por selección.** Este método de ordenación consiste en buscar el elemento más pequeño de la lista a ordenar y ponerlo en primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segundo lugar, y así sucesivamente hasta colocar el último elemento.

■ 14. **El método de la burbuja.** Este método de ordenación consiste en comparar pares de elementos adyacentes e

intercambiarlos entre sí hasta que estén todos ordenados. Los elementos más pequeños van "burbujeando" hasta alcanzar las primeras posiciones.

■ 13. El método de los **rectángulos** es un método numerico para calcular integrales definidas. Consiste en aproximar el área bajo la curva y=f(x) entre x=a y x=b por una suma de áreas de rectángulos del siguiente modo: Fijamos una partición P={x0=a,x1, ...xn=b} del intervalo en n partes iguales. Aproximamos el área bajo la curva en el intervalo $[x_{i-1}, x_i]$ por el área del rectángulo de base x_i-x_{i-1} y altura f(c_i) siendo c_i el punto medio del intervalo $[x_{i-1}, x_i]$. Cuanto más fina sea la partición P, más se aproxima el área de los rectángulos al área bajo la curva. Comenzamos dividiendo el intervalo a la mitad, después en tres partes,....

(a) Programar el método anterior. La función f; el número de iteraciones ite y los extremos del intervalo a y b se solicitan del usuario mediante Input. Será más fácil si en cada paso del algoritmo en vez de pasar de n a n+1 divisiones del intervalo pasas de n a 2n (divides cada subintervalo a la mitad).

(b) Prueba el programa con $\int_0^1 \frac{x}{(x+1)(x+2)}\,dx$ y 20 iteraciones. Compara con el resultado exacto.

(c) ¿Que cambios harías en el programa (a) si quieres especificar un error máximo permitido en vez del número de iteraciones?

■ 14. El método de los **trapecios** es un método numerico para calcular integrales definidas. Consiste en aproximar el área bajo la curva y=f(x) entre x=a y x=b por una suma de áreas de trapecios del siguiente modo: Fijamos una partición P={x0=a,x1, ...xn=b} del intervalo en n partes iguales. Aproximamos la curva por la poligonal construida uniendo los puntos (x_i,f(x_i)) con (x_{i+1},f(x_{i+1})) y el área bajo la curva por el área bajo por la suma de las áreas de los trapecios. El área de cada trapecio se calcula con la fórmula

$$\frac{(b-a)}{n}\ \frac{f(x_i)+f(x_{i+1})}{2}$$

Cuanto más fina sea la partición del intervalo, más se aproxima el área de los trapecios al área bajo la curva.

(a)Programar el método anterior solicitando del usuario la función f, los extremos de integración y un error máximo permitido.

(b)Probar el programa anterior para calcular $\int_1^3 e^x \operatorname{sen}(x)\,dx$con un error menor que 1/1000.

■ 15. El método de **Simpson** es un método numerico para calcular integrales definidas. Consiste en aproximar la curva por segmentos parabólicos y el área bajo la curva y=f(x) entre x=a y x=b por el área bajo las parábolas del siguiente modo: Fijamos una partición P={x0=a,x1, ...xn=b} del intervalo en n partes iguales. Aproximamos el área bajo la curva en el intervalo $[x_{i-1}, x_i]$ por el área bajo el segmento parabólico que paroxima la curva en el intervalo $[x_{i-1}, x_i]$. Este área se calcula por la fórmula h/6 (f(x_{i-1})+f(x_i)+4f(c_i)), siendo c_i el punto medio del intervalo $[x_{i-1}, x_i]$ y h la longitud de los subintervalos en que dividimos el intervalo [a,b]. Cunato más fina sea la partición más se aproxima el área bajo los segmentos parábolicos al área bajo la curva.

(a)Programar el método anterior solicitando del usuario la función f, los extremos de integración y un error máximo permitido.

(b)Probar el programa anterior para calcular $\int_1^3 e^x \operatorname{sen}(x)\, dx$ con un error menor que 1/1000. Compara con el resultado exacto

CAPÍTULO 3
PROGRAMACIÓN FUNCIONAL

En muchos lenguajes de programación los bucles e iteraciones presentados en el capítulo anterior son posibles. Sin embargo aunque *Mathematica* ofrece este tipo de programación tradicional, es preferible reducir su uso continuado.

La verdadera potencia de *Mathematica* se hace patente cuando hacemos cálculo simbólico. Por esta razón resulta más eficaz diseñar nuestros programas bajo modelos de programación funcional y reglas de transformación. En este capítulo nos ocuparemos de la programación funcional, lo que nos permitirá tratar todo tipo de expresiones, en particular las funciones, como datos.

Veremos como incluso en problemas númericos (Método de Newton...) la programación funcional es una alternativa a bucles e iteraciones.

La programación funcional se basa :

■ Por un lado, en la noción de expresión ("TODO ES UNA EXPRESION"). Esto hace de *Mathematica* un lenguaje muy flexible, donde las funciones pueden operar sobre todo tipo de datos.

■ Por otro, en operadores funcionales, funciones que a su vez actúan sobre otras funciones.

3.1.LISTAS

Las listas constituyen una de los elementos más comunes y también uno de los más potentes en *Mathematica*. Las listas sirven para agrupar elementos de cualquier tipo. Por ejemplo: a={x,2,f[x]}

Es importante manejar con soltura las listas, ya que su estructura es común a la totalidad de las expresiones. A continuación hacemos un breve recorrido por las instrucciones básicas para manejar listas.

■ 3.1.1. Construyendo listas

Dos de las instrucciones más corrientes para construir listas son Table y Array.

■ 3.1.1.1. Table

Table[f, {i, imin, imax, d}] generamos la lista {f[imin],...f[imax]}
Table[f,{i,imax,d}] generamos la lista {f[1],...f[imax]}
Table[f,{i,imax}] generamos la lista {f[1],...f[imax]} , el paso d se supone1
Table[f,{i,imin,imax,di},{j,jmin,jmax,dj}] generamos listas multidimensionales

Table

■ Ejemplo: generamos la lista formada por los diez primeros números primos