

Tema 6

Funciones

Guillermo Peris Ripollés

Objetivos

Cuando finalice este tema, el alumno deberá ser capaz de:

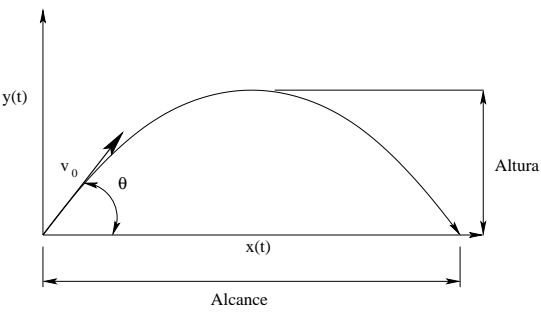
- Definir y utilizar funciones en `OCTAVE`.

Aplicación

Cuando finalice este tema, el alumno deberá ser capaz de resolver problemas como el siguiente, cuya resolución se indica a lo largo del propio tema.

Trayectoria de un proyectil

La siguiente figura muestra la trayectoria de un proyectil lanzado a una velocidad v_0 y un ángulo θ sobre un plano horizontal.



Suponiendo que pueden despreciarse todos los efectos de resistencia del aire, las ecuaciones del movimiento en los dos ejes vienen dadas por las ecuaciones siguientes:

$$x(t) = v_0 \cos(\theta)t$$
$$y(t) = v_0 \sin(\theta)t - \frac{1}{2}gt^2$$

donde $g = 9.81m/s^2$. Igualando $y(t) = 0$ podemos obtener el tiempo máximo de vuelo, que es el momento en el que el proyectil toca el suelo.

$$\text{Tiempo de vuelo} = \frac{2v_0 \sin(\theta)}{g}$$

A partir de estas ecuaciones, escribe:

1. Una función que calcule un vector de posiciones (x), en función del tiempo para una velocidad y ángulo inicial determinado.
2. Una función que calcule un vector de alturas (y), en función del tiempo para una velocidad y ángulo inicial determinado.
3. Una función que calcule el tiempo de vuelo para una velocidad y ángulo inicial determinado.
4. Una función que represente la trayectoria del proyectil a partir de los vectores de posición y altura.
5. Un programa que pida al usuario la velocidad y ángulo inicial del proyectil y represente su trayectoria, utilizando todas las funciones anteriores.

Contenidos

6.1. Definición y uso de funciones	6-3
6.2. Diseño de programas con funciones	6-5
6.3. Aplicación	6-7
6.4. Ejercicios prácticos	6-14

En la unidad de introducción a `OCTAVE` vimos cómo era posible ejecutar directamente órdenes en la ventana de `OCTAVE`, pero que en problemas más complejos resultaba más conveniente utilizar programas. Estos ficheros son útiles cuando deseamos repetir un conjunto de órdenes, cambiando (o no) algunos de los valores de las variables. Sin embargo, no son más que una mecanización de la introducción directa de órdenes. Asimismo, también hemos utilizado funciones `OCTAVE`, como por ejemplo `sqrt`, que recibían (o no) una serie de parámetros de entrada y devolvían (o no) un conjunto de parámetros de salida. En esta unidad, aprenderemos a definir nuestras propias funciones `OCTAVE`, lo cual nos permitirá escribir programas de una forma más estructurada y reutilizar código.

6.1. Definición y uso de funciones

Antes de detallar las reglas que deben seguir las funciones vamos a considerar un ejemplo sencillo: la función `estad(x)`, que se define a continuación, calcula la media y desviación típica de los datos contenidos en un vector:

```
function [media, std] = estad(x)
% ESTAD(x) Estadística simple
% Calcula la media y desviación típica de un vector x.

    n = length(x);
    media = sum(x)/n;
    v = x - media;
    std = sqrt(v*v'/(n-1)) ;
endfunction
```

La función `estad(x)` recibe un argumento de entrada que es un vector `x` y devuelve un argumento de salida que también es un vector `[media, std]` que contiene los valores de la media y desviación típica. La sintaxis general para la definición de funciones es la siguiente:

```
function [out1 out2 ... outM] = nombre_funcion (in1, in2, ...inN)
```

A esta línea se la conoce como declaración de la función. Fijémonos en que pueden existir varios argumentos de salida como elementos de un vector, por lo que deberán aparecer entre corchetes. En el caso de un solo argumento de salida los corchetes no son necesarios, y también es posible que no haya ningún argumento de salida. Del mismo modo, puede pasarse a la función una lista de argumentos de entrada separados por comas, e incluso no pasar ningún argumento. Cada uno de estos argumentos puede ser un escalar, vector o matriz.

El fichero que contenga a la función debe tener su mismo nombre, por lo que la función `estad` debe guardarse en un fichero de nombre `estad.m`. La primera línea del fichero debe ser necesariamente la declaración de la función.

Tras la declaración de la función, aparecen un conjunto de comentarios que documentan el uso de la función. Estas líneas son las que aparecerán cuando se pida ayuda sobre la función a `OCTAVE`:

```

>> help estad
estad is the user-defined function from the file
/home/user/estad.m

ESTAD(x) Estadística simple
Calcula la media y desviación típica de un vector x.

```

Por último, el resto de la función se denomina el *cuerpo de la función*. Fijémonos en que, en algún momento, deben definirse los valores de los argumentos de salida, en el ejemplo las variables `media` y `std`, utilizando los argumentos de entrada, en este caso el vector `x`.

La orden `endfunction` indica que la ejecución de la función ha terminado ¹.

Llamada a funciones

Una vez definida la función, para utilizarla debemos ejecutar *órdenes de llamada a la función*, de la misma forma que lo hacíamos con las funciones predefinidas por `OCTAVE`. Así, a continuación se muestra un ejemplo de llamada a la función `estad`:

```




>> vec = rand(1,5);
>> [a b] = estad(vec)
a = 0.24375
b = 0.20014

```

Fíjate en que los nombres de las variables que se le pasan a la función no tienen por qué coincidir con la definición de la función. Así, mientras que en la definición de `estad` el argumento de entrada era `x`, aquí hemos utilizado la variable `vec` para realizar la llamada, y lo mismo ocurre con los argumentos de salida.

Por último, recuerda que para que `OCTAVE` localice una de tus funciones, debes ejecutar el programa en el directorio en el que se encuentre la función (podemos evitar ésto, cambiando el valor de una variable de entorno de `OCTAVE`. Si te interesa saber cómo se hace, pregúntale a tu profesor).

Ejercicios

-  ▶ **1** Escribe la función `estad`, guárdala con el nombre `estad.m` en tu directorio de trabajo, y comprueba su funcionamiento con vectores de valores aleatorios (utiliza la función `rand(1,n)`, cambiando `n` para vectores de distinta longitud).
-  ▶ **2** Comprueba que obtienes las líneas de ayuda con la orden `help estad`.
-  ▶ **3** El área de un triángulo de lados a , b y c viene dada por la ecuación

$$area = \sqrt{s(s-a)(s-b)(s-c)} \quad ,$$

donde $s = (a + b + c)/2$. Escribe una función que acepte a , b y c como argumentos de entrada y devuelva el área del triángulo como salida.

¹ Mucho ojo para los usuarios de `Matlab`!!! Debeis usar `return` en lugar de `endfunction`.

6.2. Diseño de programas con funciones²

Ya hemos aprendido a escribir funciones, pero no está claro qué ventajas nos reporta trabajar con ellas. A grandes rasgos, el uso de funciones facilita la lectura del código y su propia programación. Veámoslo con un ejemplo.

Supongamos que en un programa deseamos leer dos números enteros y asegurarnos de que sean positivos. Podemos proceder repitiendo el bucle correspondiente dos veces:

```
% Leemos a
do
    a = input('Introduce un numero positivo: ');
    if a < 0
        disp('Error: el numero debe ser positivo');
    end
until a > 0
% Leemos b
do
    b = input('Introduce un numero positivo: ');
    if b < 0
        disp('Error: el numero debe ser positivo');
    end
until b > 0
end
```

o diseñar una función que lea un número asegurando que sea positivo,

```
function numero = lee_numero_positivo
% lee_numero_positivo: Pide un numero al usuario
%           y comprueba que sea positivo
do
    numero = input('Introduce un numero positivo');
    if numero < 0
        disp('Error: el numero debe ser positivo');
    end
until numero > 0
endfunction
```

y realizar dos llamadas a la misma,

```
a = lee_numero_positivo;
b = lee_numero_positivo;
```

Un programa que utiliza funciones es, por regla general, más legible que uno que inserta los procedimientos de cálculo directamente donde se utilizan... siempre que se escojan nombres de función que describan bien qué hacen éstas. Fijémonos en que este último programa es más fácil de leer que el original.

² Esta sección y la siguiente han sido adaptadas de los apuntes para Metodología y Tecnología de la Programación de primer curso de Ingeniería Informática, escritos por Andrés Marzal.

Las funciones son un elemento fundamental de los programas. Ahora ya sabemos *cómo* construir funciones, pero no *cuándo* conviene construirlas. Lo cierto es que no podemos decirlo: no es una ciencia exacta, sino una habilidad que se va adquiriendo con la práctica. De todos modos, sí podemos dar algunos consejos.

1. Por una parte, *todos los fragmentos de programa que se utilizan en más de una ocasión son buenos candidatos a definirse como funciones*, pues de ese modo evitamos tener que copiarlos en varios lugares. Evitar esas copias no sólo resulta más cómodo: también reduce considerablemente la probabilidad de que cometamos errores, pues acabamos escribiendo menos texto. Además, si acabamos cometiendo errores y hemos de corregirlos o si hemos de modificar el programa para ampliar su funcionalidad, siempre será mejor que el mismo texto no aparezca en varios lugares, sino una sola vez en una función.
2. No conviene que las funciones que definamos sean muy largas. En general, una función debería ocupar menos de 30 o 40 líneas (aunque siempre hay excepciones). *Una función no sólo debería ser breve, además debería hacer una única cosa... y hacerla bien*. Deberíamos ser capaces de describir con una sola frase lo que hace cada una de nuestras funciones. Si una función hace tantas cosas que explicarlas todas cuesta mucho, probablemente haríamos bien en dividir la función en otras funciones más pequeñas y simples. Recordemos que siempre podemos llamar a una función desde otra.

El proceso de identificar acciones complejas y dividir las en acciones más sencillas se conoce como *estrategia de diseño descendente* (en inglés, *top-down*). La forma de proceder es ésta:

- analizar primero qué debe hacer el programa y hacer un diagrama que represente las diferentes acciones que debe efectuar, pero sin entrar en los detalles de cada una de ellas;
- descomponer el programa en una posible función por cada una de esas acciones;
- analizar entonces cada una de esas acciones y ver si aún son demasiado complejas; si es así, aplicar el mismo método hasta que obtengamos funciones más pequeñas y simples.

Una estrategia de diseño alternativa recibe el calificativo de *ascendente* (en inglés, *bottom-up*) y consiste en lo contrario:

- detectar algunas de las acciones más simples que necesitaremos en el programa y escribir pequeñas funciones que las implementen;
- combinar estas acciones en otras más sofisticadas y crear nuevas funciones para ellas;
- seguir hasta llegar a una o unas pocas funciones que resuelvan el problema planteado.

Cuando se empieza a programar resulta difícil anticiparse y detectar a simple vista qué pequeñas funciones irán haciendo falta y cómo combinarlas apropiadamente. Será más efectivo empezar siguiendo la metodología descendente: dividir cada problema en subproblemas más y más sencillos que, al final, se combinan para dar solución al problema original. Cuando tengamos mucha más experiencia, descubriremos que al programar seguimos una estrategia híbrida, ascendente y descendente a la vez. Todo llega. Paciencia.

6.3. Aplicación

A continuación se muestran las funciones que se piden en la aplicación, así como el programa que las utiliza. A estas alturas del curso, no deberías de necesitar ninguna explicación para comprender este código.

```
function x = posicion(v0, theta, t)
% Funcion: x = posicion(v0, theta, t)
% Calcula la posición de un objeto
% lanzado con velocidad inicial v0
% y angulo theta, para distintos
% valores de tiempo t.

    x = v0*cos(theta*pi/180)*t;
endfunction
```

```
function y = altura(v0, theta, t)
% Funcion: y = altura(v0, theta, t)
% Calcula la altura de un objeto
% lanzado con velocidad inicial v0
% y angulo theta, para distintos
% valores de tiempo t.

    g = 9.81 ; % Constante gravedad en m/s2
    y = v0*sin(theta*pi/180)*t - 0.5*g*t.^2;
endfunction
```

```
function t = tvuelo(v0, theta)
% Funcion: t = tvuelo(v0, theta)
% Calcula el tiempo de vuelo de un objeto
% lanzado con velocidad inicial v0
% y angulo theta.

    g = 9.81 ; %Constante gravedad en m/s2.
    t = 2*v0*sin(theta*pi/180)/g;
endfunction
```

```
function trayectoria(x,y)
% Funcion: trayectoria(x,y)
% Dibuja la trayectoria de un objeto
% a partir de vectores de posición x e y.

    clg
    plot(x,y,';'), title('Trayectoria')
    xlabel('x(m)'), ylabel('y(m)')
    replot
endfunction
```

```
%=====
% Programa proyectil.m: Pide al usuario los valores
% iniciales de velocidad y ángulo
% de un proyectil, y representa
% su trayectoria.
%=====

% Pedimos los valores al usuario
v0 = input('Introduce el valor de velocidad inicial en m/s: ');
theta = input('Introduce el valor del ángulo inicial en grad: ');

% Calculamos el vector de tiempos
t = linspace(0,tvuelo(v0,theta),50);

% Hacemos la representacion
trayectoria(posicion(v0,theta,t), altura(v0,theta,t) )
```


Apéndice: Recursión

Desde una función podemos llamar a otras funciones. Ya lo hemos hecho en los ejemplos que hemos estudiado, pero ¿qué ocurriría si una función llamara a otra y ésta, a su vez, llamara a la primera? O de modo más inmediato, ¿qué pasaría si una función se llamara a sí misma?

La capacidad de que una función se llame a sí misma, directa o indirectamente, se denomina *recursión*. La recursión es un potente concepto con el que se pueden expresar procedimientos de cálculo muy elegantemente. No obstante, al principio cuesta un poco entender las funciones recursivas... y un poco más diseñar nuestras propias funciones recursivas.

La recursión es un concepto difícil cuando estamos aprendiendo a programar. Por ello, no debes asustarte si este material se te resiste más que el resto.

Cálculo recursivo del factorial

Empezaremos por presentar y estudiar una función recursiva: el cálculo recursivo del factorial de un número. Partiremos de la siguiente fórmula matemática para el cálculo del factorial:

$$n! = \begin{cases} n \cdot (n - 1)!, & \text{si } n > 1. \\ 1, & \text{si } n = 1. \end{cases}$$

Es una definición de factorial un tanto curiosa, pues se define en términos de sí misma! El primero de sus dos casos dice que para conocer el factorial de n hay que conocer el factorial de $n - 1$ y multiplicarlo por n . Entonces, ¿cómo calculamos el factorial de $n - 1$? En principio, conociendo antes el valor del factorial de $n - 2$ y multiplicando ese valor por $n - 1$. ¿Y el de $n - 2$? Pues del mismo modo... y así hasta que acabemos por preguntarnos cuánto vale el factorial de 1. En ese momento no necesitaremos hacer más cálculos: el segundo caso de la fórmula nos dice que el factorial de 1 vale 1.

Vamos a plasmar este mismo procedimiento en una función **OCTAVE**:

```
function f = factorial(n)
% FACTORIAL(n): Calcula el factorial de n de forma recursiva.
  if n > 1
    f = n * factorial(n-1) ;
  else
    f = 1 ;
  end
endfunction
```

Compara la fórmula matemática y la función **OCTAVE**. No son tan diferentes. **OCTAVE** nos fuerza a decir lo mismo de otro modo, es decir, con otra *sintaxis*. Más allá de las diferencias de forma, ambas definiciones son idénticas en el fondo.

Para entender la recursión, nada mejor que verla en funcionamiento. La Figura 6.1 nos muestra paso a paso qué ocurre si solicitamos el cálculo del factorial de 5. Con el anidamiento de cada uno de los pasos pretendemos ilustrar que el cálculo de cada uno de los factoriales tiene lugar mientras el anterior aún está pendiente de completarse. En el nivel más interno, **factorial(5)** está pendiente de que acabe **factorial(4)**, que a su vez está pendiente de que acabe **factorial(3)**, que a su vez está pendiente de que acabe **factorial(2)**, que a su vez está pendiente de que acabe **factorial(1)**. Cuando **factorial(1)** acaba, pasa el valor 1 a **factorial(2)**, que a su vez pasa el

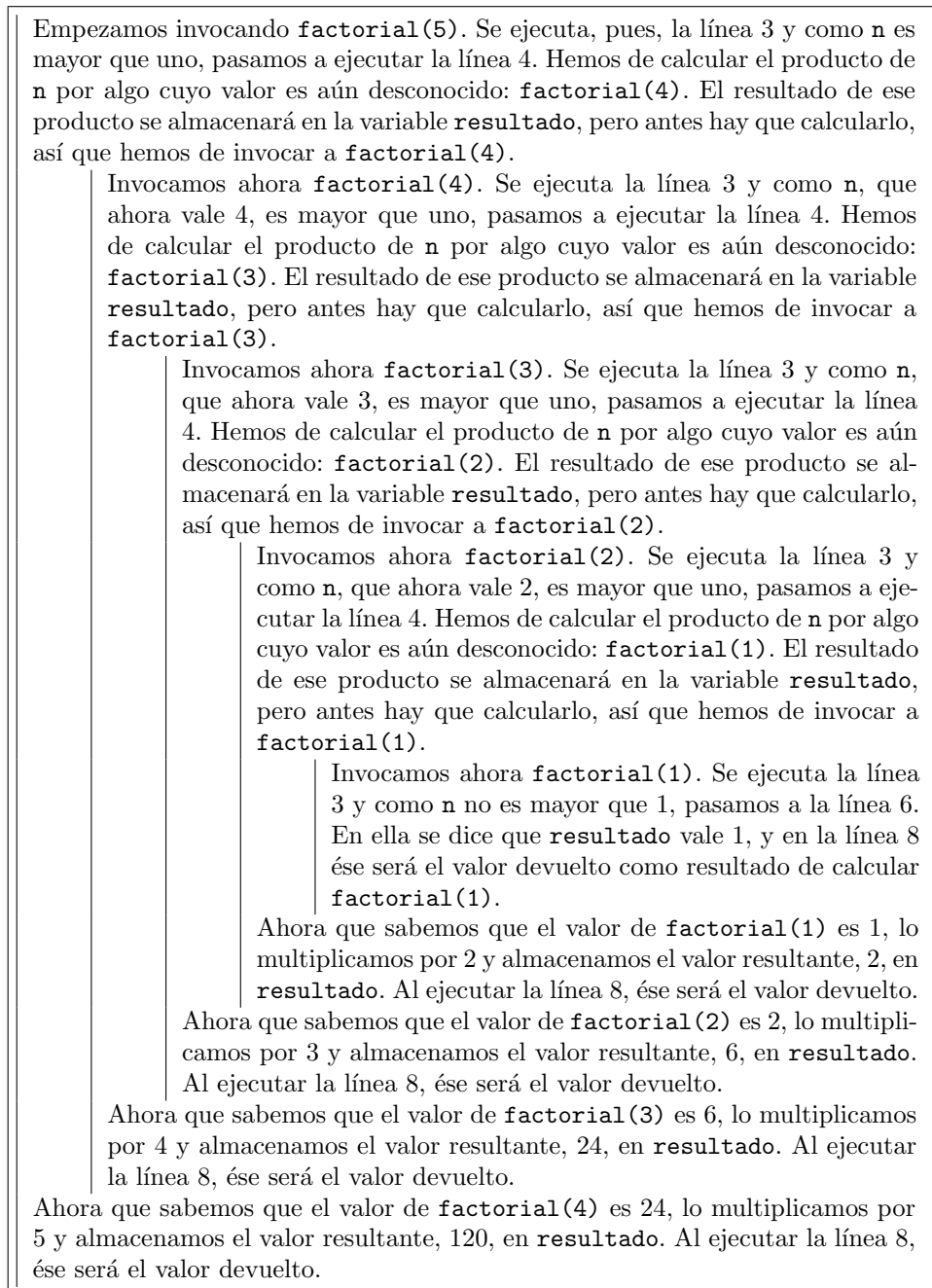


Figura 6.1: Traza del cálculo recursivo de `factorial(5)`.

valor 2 a `factorial(3)`, que a su vez pasa el valor 6 a `factorial(4)`, que a su vez pasa el valor 24 a `factorial(5)`, que a su vez devuelve el valor 120.

De acuerdo, la figura 6.1 describe con mucho detalle lo que ocurre, pero es difícil de seguir y entender. Veamos si la figura 6.2 nos es de más ayuda. En esa figura también se describe paso a paso lo que ocurre al calcular el factorial de 5, sólo que con la ayuda de unos muñecos.

- En el paso 1, le encargamos a Amadeo que calcule el factorial de 5. Él no sabe calcular el factorial de 5, a menos que alguien le diga el valor del factorial de 4.
- En el paso 2, Amadeo llama a un hermano clónico suyo, Benito, y le pide que calcule el factorial de 4. Mientras Benito intenta resolver el problema, Amadeo se echa a dormir (paso 3).

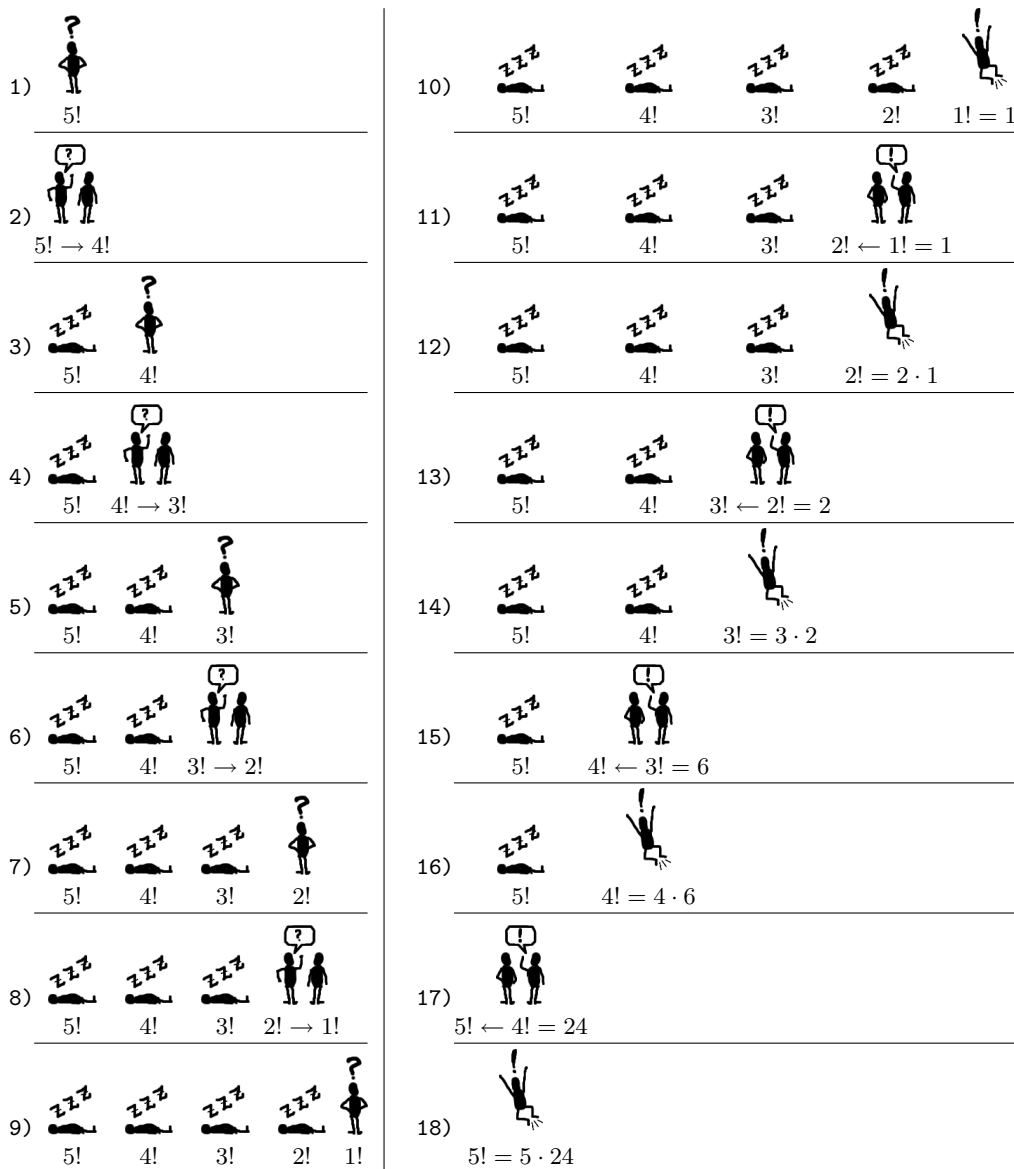





Figura 6.2: Cómec explicativo del cálculo recursivo del factorial de 5.

- Benito tampoco sabe resolver directamente factoriales tan complicados, así que llama a su clon Ceferino en el paso 4 y le pide que calcule el valor del factorial de 3. Mientras, Benito se echa a dormir (paso 5).
- La cosa sigue igual un ratillo: Ceferino llama al clon David y David a Eduardo. Así llegamos al paso 9 en el que Amadeo, Benito, Ceferino y David están durmiendo y Eduardo se pregunta cuánto valdrá el factorial de 1.
- En el paso 10 vemos que Eduardo cae en la cuenta de que el factorial de 1 es muy fácil de calcular: vale 1.
- En el paso 11 Eduardo despierta a David y le comunica lo que ha averiguado: el factorial de 1! vale 1.
- En el paso 12 Eduardo nos ha abandonado: él ya cumplió con su deber. Ahora es David el que resuelve el problema que le habían encargado: 2! se puede calcular multiplicando 2 por lo que valga 1!, y Eduardo le dijo que 1! vale 1.

- En el paso 13 David despierta a Ceferino para comunicarle que $2!$ vale 2. En el paso 14 Ceferino averigua que $3!$ vale 6, pues resulta de multiplicar 3 por el valor que David le ha comunicado.
- Y así sucesivamente hasta llegar al paso 17, momento en el que Benito despierta a Amadeo y le dice que $4!$ vale 24.
- En el paso 18 sólo queda Amadeo y descubre que $5!$ vale 120, pues es el resultado de multiplicar por 5 el valor de $4!$, que según Benito es 24.

Ejercicios

-  ▶ **4** Escribe la función `factorial` en `OCTAVE` y utilízala para calcular los factoriales de varios números enteros.
-   ▶ **5** También podemos formular recursivamente la suma de los n primeros número naturales:

$$\sum_{i=1}^n i = \begin{cases} 1, & \text{si } n = 1; \\ n + \sum_{i=1}^{n-1} i, & \text{si } n > 1. \end{cases}$$



Diseña una función `OCTAVE` recursiva que calcule el sumatorio de los n primeros números naturales. Comprueba que funciona correctamente comparando los resultados con los obtenidos con la orden `sum(1:n)`.

Hemos propuesto una solución recursiva para el cálculo del factorial, pero también podemos escribir una función que realice este cálculo *iterativamente*, utilizando un bucle `for`:

```
function f = factorial(n)
% factorial: Calcula el factorial de un entero n.
f = 1;
for i = 2:n
    f = f * i;
end
endfunction
```

Pues bien, para toda función recursiva podemos encontrar otra que haga el mismo cálculo de modo iterativo. Ocurre que no siempre es fácil hacer esa conversión o que, en ocasiones, la versión recursiva es más elegante y legible que la iterativa (al menos se parece más a la definición matemática). Por otra parte, las versiones iterativas suelen ser más eficientes que las recursivas, pues cada llamada a una función supone pagar una pequeña penalización en tiempo de cálculo y espacio de memoria.

Ejercicios

-   ▶ **6** Los números de Fibonacci son una secuencia de números enteros bastante curiosa, ya que la naturaleza está llena de situaciones en las que éstos aparecen. A continuación, se indican los primeros números de esta serie:

1 1 2 3 5 8 13 21 34 55 89

Los dos primeros números de la secuencia valen 1 y cada número a partir del tercero se obtiene sumando los dos anteriores. Podemos expresar esta definición matemáticamente así:

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & \text{si } n > 2. \\ 1, & \text{si } n = 1 \text{ o } n = 2. \end{cases}$$

- a) Escribe una función que acepte como argumento un entero n y devuelva F_n .
 - b) Escribe una función que lea una lista de enteros y calcule recursivamente el número de Fibonacci F_n de cada uno de ellos. Haz uso de la función definida anteriormente.
 - c) Calcula los 20 primeros números de Fibonacci, usando la función del apartado anterior.
 - d) Para los primeros 20 números de Fibonacci, calcula la relación F_n/F_{n-1} y represéntala en función de n . Esta relación tiende a $\frac{1+\sqrt{5}}{2}$. ¿Qué muestran tus resultados?
 - e) Escribe una función que calcule los primeros 20 números de Fibonacci sin recursión, utilizando bucles.
-

6.4. Ejercicios prácticos

Es conveniente que pienses y realices los ejercicios que han aparecido a lo largo de la unidad marcados con el símbolo \blacktriangleleft antes de acudir a la sesión de prácticas correspondiente. Deberás iniciar la sesión realizando los ejercicios marcados con el símbolo \blacksquare . A continuación, deberás hacer el mayor número de los ejercicios siguientes.

Ejercicios

► 7 1) Escribe una función `OCTAVE` que calcule la presión de un gas utilizando la ley de los gases ideales (LGI),

$$P = \frac{RT}{\hat{V}}$$

y otra función que calcule la presión con la ecuación de Soave-Redlich-Kwong (SRK),

$$P = \frac{RT}{\hat{V} - b} - \frac{\alpha a}{\hat{V}(\hat{V} + b)},$$

donde

$$\begin{aligned}\hat{V} &= V/n \\ a &= 0.42747R^2T_c^2/P_c \\ b &= 0.08664RT_c/P_c \\ R &= 0.082 \frac{\text{atm} \cdot \text{l}}{\text{mol} \cdot \text{K}}\end{aligned}$$

La constante α se calcula a partir del factor acéntrico de Pitzer ω utilizando dos ecuaciones adicionales:

$$\begin{aligned}m &= 0.48508 + 1.55171\omega - 0.1561\omega^2 \\ \alpha &= \left[1 + m \left(1 - \sqrt{T/T_c}\right)\right]^2\end{aligned}$$

Al escribir las funciones debes documentarlas convenientemente, de forma que después cualquier usuario pueda obtener ayuda sobre ellas con la orden `help`. Además, puedes definir las subfunciones que consideres necesarias para facilitar la programación.

2) Utiliza estas funciones para hacer una gráfica de la presión en función de la temperatura para un cilindro de gas de 820 l. que contiene 100 moles de CO_2 . El rango de temperaturas debe variar entre 20 y 400°C . La gráfica debe incluir las curvas predecidas por los modelos LGI y SRK. Para el CO_2 , los valores de las constantes requeridos para la ecuación SRK son:

$$\begin{aligned}\omega &= 0.225 \\ T_c &= 304.2\text{K} \\ P_c &= 72.9 \text{ atm}\end{aligned}$$

► **8** Un fluido compuesto de moléculas con momento dipolar μ , a una temperatura T , y sometido a un campo eléctrico E , presenta un momento dipolar medio (que se puede medir experimentalmente) dado por la expresión

$$\mu_{medio} = \mu\Lambda,$$

donde Λ es la *función de Langevin*, cuya ecuación es:

$$\Lambda = \coth(x) - \frac{1}{x},$$

siendo $x = \frac{\mu E}{kT}$, y $\coth(x)$ la cotangente hiperbólica de x ,

$$\coth(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}.$$

Por último, $k = 1.38 \cdot 10^{-23} \text{ J/K}$ es la constante de Boltzmann.

1. Escribe una FUNCIÓN `coth` que reciba como argumento de entrada un **vector** de valores y devuelva como argumento de salida un **vector** con las cotangentes hiperbólicas del vector de entrada.

Ayuda: Recuerda que la función que calcula e^x es `exp(x)`.

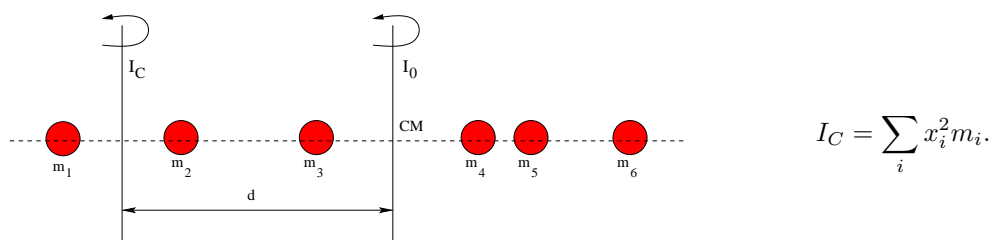
2. Escribe una FUNCIÓN `Langevin` que reciba como argumento de entrada un **vector** de valores x y devuelva como argumento de salida un **vector** con los valores de la función de Langevin Λ del vector de entrada.

3. Escribe un PROGRAMA que pida al usuario :

- el momento dipolar molecular μ ,
- el valor del campo eléctrico externo E , y
- el valor de la temperatura T en Kelvin.

y calcule el momento dipolar medio μ_{medio} .

► **9** El momento de inercia I_C de una distribución lineal de masas puntuales respecto a un eje de rotación (ver figura) viene dado por la siguiente ecuación:



donde x_i es la distancia de la partícula i al eje de rotación, y m_i su masa. Si conocemos el momento de inercia respecto al eje que pasa por el centro de masas, basta con que apliquemos el teorema de Steiner, según el cual

$$I_C = I_0 + M \cdot d^2$$

donde $M = \sum_i m_i$ es la masa total del sistema, y d la distancia del eje C al centro de masas. En este ejercicio se pretende que escribas un conjunto de funciones y un programa que permita el cálculo del momento de inercia de un sistema de partículas lineales. Para ello, sigue ordenadamente los siguientes pasos:

1. Escribe una función `CENTRODEMASAS` que, a partir de un **vector** de posiciones x y un **vector** de masas m devuelva como argumento de salida la posición del centro de masas x_{CM} del sistema de partículas, utilizando la ecuación:

$$x_{CM} = \frac{\sum_i (m_i \cdot x_i)}{\sum_i m_i}$$

2. Escribe una función `INERCIACM` que, a partir de un **vector** de posiciones x , un **vector** de masas m y el centro de masas x_{CM} devuelva como argumento de salida el momento de inercia del sistema respecto a un eje que pase por el centro de masas. La ecuación para este momento de inercia es:

$$I_{CM} = \sum_i (m_i \cdot (x_i - x_{CM})^2)$$

3. Escribe una función `INERCIA` que, a partir de un **vector** de posiciones x , un **vector** de masas m y la posición del eje de rotación x_0 , devuelva como argumento de salida el momento de inercia del sistema. Este momento de inercia se calcula con el teorema de Steiner:

$$I = I_{CM} + M(x_0 - x_{CM})^2$$

Nota: Ten en cuenta que esta función requiere el uso de las dos funciones anteriores.

4. Escribe un programa que, utilizando la función anterior, calcule el momento de inercia de un sistema de partículas lineales. Para ello, deberá pedirle al usuario un vector de masas, un vector de posiciones y la posición del eje de rotación. El programa también deberá comprobar que la longitud de los dos vectores sea la misma.
-