Tema 5

Instrucciones repetitivas

Guillermo Peris Ripollés

Objetivos

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

- Entender el funcionamiento de los bucles `for`, `while` y `do-until`, utilizándolos en las situaciones que corresponda de forma adecuada.
- Comprender el anidamiento de bucles, aplicándolo en los problemas en que resulten convenientes.

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.

Cálculo de la energía de repulsión nuclear en una molécula

La energía de repulsión nuclear entre dos átomos i y j de una molécula viene dada por la expresión

$$E_{ij} = \frac{Z_i \cdot Z_j}{R_{ij}} \quad ,$$

donde Z_i es la carga nuclear del átomo i y R_{ij} es la distancia que separa los átomos i y j.

1. Escribe un programa en `octave` que pida al usuario un vector de cargas nucleares Z y una matriz de distancias R_{ij}, y con ellos calcule la matriz E_{ij} de energías nucleares según la ecuación anterior. Deberás tener en cuenta que los elementos de la diagonal de la matriz E_{ij} deben ser infinito (ya que la distancia entre un átomo y sí mismo es 0).
2. Añade al programa anterior las líneas necesarias para calcular la energía total de repulsión.
3. Añade al programa anterior las líneas necesarias para decidir qué átomos experimentan una mayor repulsión.

Contenidos

5.1. Introducción

Un bucle es una estructura de programación que permite la repetición controlada de un conjunto de instrucciones. Este tipo de estructuras, y en particular las instrucciones *for* y *while*, son utilizadas de forma generalizada en la inmensa mayoría de lenguajes de programación, lo cual hace interesante su estudio. Sin embargo, en muchos casos los bucles pueden (y deben) evitarse en OCTAVE, ya que hay estructuras típicas de este lenguaje más eficaces que los propios bucles. No obstante, sí que existen ocasiones en las que es necesario el uso de bucles, por lo que estudiaremos estas instrucciones con detalle en esta unidad.

5.2. Bucles for

Los bucles `for` se utilizan cuando nos interesa repetir un bloque de instrucciones un número predeterminado de veces. La estructura general del bucle `for` es la siguiente[1]:

```
for i = vector
    instrucciones
end
```

El conjunto de *instrucciones* se repite para cada elemento de *vector*, denominándose *iteración* cada una de estas repeticiones. En cada iteración, i toma ordenadamente el valor de cada elemento de *vector*. Al igual que ocurría con las instrucciones `if`, resulta conveniente realizar un sangrado de línea en las instrucciones del interior del bucle. Veamos un ejemplo sencillo:

```
≫for i = [1:5]
     j = 2*i;
     disp(j)
end
   2
   4
   6
   8
   10
```

En este ejemplo, observamos como el valor de i toma en cada iteración, y de forma ordenada, los valores del vector `[1 2 3 4 5]`, y dichos valores se utilizan para realizar cálculos en el interior del bucle. El operador ':' para la creación de listas es ampliamente utilizado en los bucles `for`. En este caso, los bucles tendrán la siguiente forma particular:

```
for k = [inicial:incremento:final]
    instrucciones
end
```

aunque los corchetes no son realmente necesarios. Por ejemplo, supongamos que queremos calcular los cuadrados de los primeros 5 números impares. Podríamos hacerlo utilizando una instrucción `for`...

[1] En lugar de recorrer un vector en cada iteración del bucle, podemos recorrer una matriz. En ese caso, en cada ciclo `i` es una de las columnas de la matriz.

```
≫for i = 1:2:9
     j = i^2;
     disp(j)
end
   1
   9
   25
   49
   81
```

...aunque en este caso sería más eficiente no hacerlo:

```
≫j = [1:2:9].^2;
≫disp(j')
   1
   9
   25
   49
   81
```

Dentro de un bucle podemos modificar el valor de la variable que recorre el vector, aunque al terminar la iteración seguirá tomando el siguiente valor que le corresponda. En el siguiente ejemplo, intentamos terminar el bucle cambiando el valor de la variable i, sin ningún efecto en el resultado final:

```
≫for i = 1:2:9
     j = i^2;
     disp(j)
     i = 10;
end
   1
   9
   25
   49
   81
```

De todas formas, para evitar posibles errores *no conviene cambiar el valor de la variable de iteración en el interior del bucle.*

Veamos un par de ejemplos de bucles `for` para entender mejor el funcionamiento de estas estructuras. Supongamos que queremos escribir un programa que sume los elementos de un vector (lo cual hace la función `sum`, pero nos olvidaremos momentáneamente de eso). Para ello, tomamos el vector y sumamos a una variable `suma` cada uno de sus elementos:

```
≫v = 1:2:9;
≫for x = v
     suma = suma + x;
end
error: 'suma' undefined...
```

¿Qué ha pasado? El código parece correcto: sumamos todos los elementos a la variable `suma`, pero aparece un mensaje de error. La causa es que no hemos *inicializado* la variable `suma`: en la primera iteración, cuando `v = 1`, tenemos la instrucción `suma = suma + 1` pero, ¿cuál es el valor inicial de `suma` al que sumarle 1? Para obtener el resultado esperado, debemos definir `suma` con un valor inicial de 0, como se muestra en el código correcto:

```
≫v = 1:2:9;
≫suma = 0 ;
≫for x = v
     suma = suma + x;
end
≫disp(suma);
  25
```

Supongamos ahora que nos interesa calcular la media de los valores de un vector. En ese caso, debemos dividir la suma de los elementos del vector por el número de elementos. Para ello, debemos *contar* el número de elementos, para lo cual utilizaremos un *contador*, que inicializaremos a 0 antes del bucle (aún no hemos tomado ningún valor del vector) e incrementaremos en 1 cada vez que encontremos un nuevo elemento. Finalmente, dividiremos la suma de los elementos por el número de éstos para obtener así la media. Fijémonos en que esta división se realiza al terminar el bucle, y no en su interior. El código OCTAVE se reproduce a continuación:

```
≫v = 1:2:9;
≫suma = 0 ;
≫contador = 0 ;
≫for x = v
     suma = suma + x;
     contador = contador + 1;
end
≫media = suma/contador;
≫disp(media);
  5
```

Esta técnica general se utiliza en muchos algoritmos en los que se necesita sumar, contar elementos, o calcular medias. OCTAVE puede realizar este cálculo más eficientemente. De hecho, en este caso, no hace falta utilizar un contador, ya que la orden `length(v)` nos daría el número de elementos de `v` y, por lo tanto, el número de iteraciones. Más aún, la función `mean` calcula con una sola instrucción la media de un vector:

```
≫mean(1:2:9)
ans =
  5
```

Por supuesto, internamente la función `mean` recorre los elementos del vector sumándolos, pero lo hace a mayor velocidad que el bucle `for`.

Ejercicios

▶ 1 Escribe un programa que calcule el producto de los elementos del vector `[1:2:9]` haciendo uso de un bucle `for` y sin hacer uso de las funciones `length` y `prod`.

▶ 2 Modifica el programa anterior para que calcule la media geométrica de los elementos de un vector utilizando bucles y contadores, es decir, sin hacer uso de funciones predefinidas como `length`, `prod`, etc. Recuerda que la media geométrica de una serie de n valores $X_1 X_2 \cdots X_n$ se calcula como Media geom. $= \sqrt[n]{X_1 X_2 \cdots X_n}$ y que $\sqrt[n]{x} \equiv x^{1/n}$.

5.3. Bucles while

El bucle `while` es una estructura que se utiliza para repetir un conjunto de instrucciones mientras se cumpla una condición lógica determinada. La estructura general de este bucle es la siguiente:

```
while condición
    instrucciones
end
```

Mientras (en inglés, *while*) la *condición* es verdadera, se ejecutan las instrucciones, tras lo cual se vuelve a comprobar la *condición*. En el momento en que ésta es falsa, se termina el bucle. Fijémonos en que alguna de las variables de *condición* debe cambiar durante las instrucciones, de lo contrario el valor de *condición* sería siempre el mismo y el bucle entraría en un *ciclo infinito*. Veamos un ejemplo sencillo de bucle `while`:

```
i = 0 ;
while i < 3
   disp(i) ;
   i = i + 1;
end
disp('Terminado');
```

La ejecución de este código da lugar a la siguiente salida:

```
0
1
2
Terminado
```

Analicemos este ejemplo con un poco más de detalle: en primer lugar, se inicializa el valor de `i` a 0. Como se cumple la condición del bucle (`i<3`) se ejecutan sus instrucciones, mostrándose el valor 0 y aumentando en uno el valor de `i`. Se vuelve a comprobar la expresión del bucle, que vuelve a ser verdadera, por lo que se muestra el valor 1 e `i` pasa a valer 2. Se repite este último paso, se muestra el valor 2 e `i` pasa a valer 3. Pero ahora, al no cumplirse la condición del bucle, se termina el ciclo, pasando a ejecutarse la instrucción que imprime la palabra `Terminado`.

Para las condiciones de los bucles `while` se pueden utilizar todos los operadores y funciones lógicas vistas en la unidad anterior. Fijémonos en que los bucle `for` se van a utilizar cuando se quiera repetir un conjunto de intrucciones un número predeterminado de veces, mientras que en el bucle `while` se busca el cumplimiento de una condición para la finalización del ciclo.

Ejercicios

▶ **3** Determina la salida de los siguientes códigos:

a)
```
i = 0 ;
while i <= 3
   disp(i) ;
   i = i + 1;
end
disp('Terminado');
```

b)
```
i = 0 ;
while i < 10
   disp(i) ;
   i = i + 2;
end
disp('Terminado');
```

c)
```
i = 3 ;
while i < 10
   disp(i) ;
   i = i + 2;
end
disp('Terminado');
```

d)
```
i = 1 ;
while i < 100
   i = i * 2;
   disp(i) ;
end
disp('Terminado');
```

5.4. Bucles do-until

Los bucles `do-until` son muy similares a los bucles `while`, en lo que respecta a que la finalización del bucle está ligada al cumplimiento de una condición. Sin embargo, existen dos diferencias importantes: en los bucles `do-until` la condición se comprueba **al final** de la estructura, y se ejecutan las órdenes **hasta** que se cumple la condición (y no **mientras**, como ocurre con `while`). La estructura general de este bucle es la siguiente:

```
do
    instrucciones
until condición
```

Fijémonos en una diferencia importante que surge como consecuencia de evaluar la condición al final: *las instrucciones del interior del bucle se ejecutan como mínimo una vez*. Veamos un ejemplo equivalente al visto anteriormente para `while`, en el que se imprimían los números del 0 al 2:

```
i = 0 ;
do
   disp(i) ;
   i = i + 1;
until i > 2
disp('Terminado');
```

Ejercicios

▶ **4** Reescribe las líneas de código del ejercicio anterior con bucles `do-until`, de forma que muestren la misma salida que aquellos.

5.5. Selección del tipo de bucle

En muchas ocasiones, intuimos que debemos utilizar algún tipo de bucle, pero no sabemos con certeza cuál de ellos es el más adecuado. Antes de ver ejemplos que ilustran las diferencias prácticas entre los tres bucles estudiados, recordemos brevemente sus características más significativas:

- Bucle `for`: Repetición de un conjunto de instrucciones un número predeterminado de veces.
- Bucle `while`: Incumplimiento de una condición al inicio del bucle para la finalización del ciclo. Las instrucciones del bucle se ejecutan 0 o más veces.
- Bucle `do-until`: Cumplimiento de una condición al final del bucle para la finalización del ciclo. Las instrucciones del bucle se ejecutan 1 o más veces.

Vamos a aplicar los tres bucles a la implementación del desarrollo en serie de funciones. Por ejemplo, supongamos que queremos calcular $1/(1-x)$ con un desarrollo de Taylor de la función:

$$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i = 1 + x + x^2 + x^3 + x^4 + \dots \quad .$$

Si quisiéramos calcular un número determinado de términos de la serie (por ejemplo 8) utilizaríamos un bucle `for`, como se muestra en el siguiente código con x =0.6:

```
≫x = 0.6;
≫suma = 0 ;
≫for i = 0:7
     suma = suma + x^i;
end
≫disp(suma) ;
suma =
   2.4580
≫disp(1/(1-x))
   2.5000
```

Pero, ¿y si nos interesa obtener el resultado con una cierta precisión? En ese caso, debemos comprobar que la diferencia entre los dos últimos valores calculados sea menor que un cierto valor pequeño, digamos 0.00001. En ese caso resulta más conveniente la utilización de un bucle `while` o `do-until`:

```
≫x = 0.6;
≫precision = 0.00001 ;
≫suma = 1 ;
≫termino = 1 ;% Inicializacion
≫while abs(termino) > precision
      termino = termino*x;
      suma = suma + termino;
endwhile
≫suma
suma = 2.5000
```

```
≫x = 0.6;
≫precision = 0.00001 ;
≫suma = 1 ;
≫termino = 1 ;% Inicializacion
≫do
      termino = termino*x;
      suma = suma + termino;
until abs(termino) < precision
≫suma
suma = 2.5000
```

Varios comentarios al respecto de este código. En primer lugar, utilizamos un producto en lugar de una potencia para el cálculo de cada término [2] de la serie,

[2] ¿Por qué se toma el valor absoluto del valor de `termino` en la condición del `while`?

calculándolo multiplicando por x el término anterior. Utilizamos un contador i para comprobar el número de iteraciones necesarias para converger a la precisión requerida. Por último, utilizamos una variable `suma` para acumular los términos obtenidos hasta el momento.

Ejercicios

▶ **5** Escribe y ejecuta el programa anterior para el cálculo de $1/(1-x)$ en x =0.6.

▶ **6** Escribe un programa que calcule $1/(1+x^2)$, siendo x un valor introducido por el usuario, mediante el siguiente desarrollo en serie de Taylor:

$$\frac{1}{1+x^2} = \sum_{i=0}^{\infty}(-1)^i x^{2i} = 1 - x^2 + x^4 - x^6 + \dots \quad .$$

La precisión también deberá ser introducida por el usuario. Fijémonos en que el signo de cada término de la serie cambia alternativamente entre + y −. Calcula el desarrollo para $x = 0.01$.

5.6. Bucles anidados

Los bucles pueden anidarse, es decir, podemos introducir un bucle dentro de otro, de forma que para cada iteración del bucle externo se ejecutan todas las iteraciones del bucle interno. Veamos un ejemplo.

Supongamos que queremos escribir un programa que, tras leer un número entero N, diga qué números entre 1 y N son primos. Una implementación burda de la condición de primo implica el análisis de todos los posibles divisores desde 2 hasta $N/2$. Así pues, en este caso podemos utilizar dos bucles `for`: uno que recorra todos los enteros i desde 1 hasta N, y otro que (para cada i) recorra todos los posibles divisores desde 2 hasta $i/2$.

```
N = input('Introduce el valor de un numero entero: ');
for i = 1:N
   primo = 1;
   for div = 2:i/2
      if rem(i,div) == 0
         primo = 0;
      end
   end
   if primo == 1
      disp(i);
   end
end
```

Inicialmente, decidimos que todos los enteros son primos (`primo = 1`), pero si encontramos un numero que divide al entero con un resto 0 (`if rem(i,div) == 0`) entonces el número ya no es primo (`primo = 0`). Fijémonos en que aunque encontremos un divisor para el número analizado, el bucle `for` interno sigue ejecutándose. En este caso resulta útil la sentencia `break`, que suspende la ejecución de un bucle cuando se cumple una cierta condición. De esta forma, una versión mejorada del programa anterior sería la siguiente:

```
N = input("Introduce el valor de un número entero: ");
for i = 1:N
   primo = 1;
   for div = 2:i/2
      if rem(i,div) == 0
         primo = 0;
         break;% Terminamos el bucle for interno
      end
   end
   if primo == 1
      disp(i);
   end
end
```

No obstante, podemos evitar el uso de la orden `break` mediante un bucle condicional:

```
N = input("Introduce el valor de un número entero: ");
for i = 1:N
   div = 2;
   while (div*div <= i) & (rem(i,div) ~= 0)
      div++;
   end
   if div*div >i
      disp(i);
   end
endfor
```

En el siguiente ejemplo, se utilizan dos bucles *for* para crear una matriz `A` en la que el elemento (i, j) se calcula como la suma de los cuadrados de los números de fila y columna, o sea, $A_{ij} = i^2 + j^2$. Fijémonos en que se inicializa la matriz (creando, por ejemplo, una matriz de ceros) antes de poder utilizarla[3].

```
≫A = zeros(5) ;
≫for i = 1:5
     for j = 1:5
          A(i,j) = i^2+ j^2;
     end
end
≫disp(A);
  2   5  10  17  26
  5   8  13  20  29
 10  13  18  25  34
 17  20  25  32  41
 26  29  34  41  50
```

[3] Realmente, en OCTAVE no es necesaria esta inicialización, ya que se crean las matrices y se modifica su tamaño cuando es necesario.

Ejercicios

▶ 7 Escribe y ejecuta el programa para el cálculo de números primos con $N = 100$.

▶ 8 Dados los vectores `x = [4 1 6 -1 -2 2]` e `y = [6 2 -7 1 5 -1]`, escribe el código OCTAVE que calcularía matrices según las siguientes fórmulas:

a) $a_{ij} = y_j/x_i$

b) $b_{ij} = x_i/(2 + x_i + y_j)$

c) $c_{ij} = 1/max(x_i, y_j)$

▶ 9 Escribe un programa que transponga una matriz cuadrada (crea una aleatoria con `rand(5)`. Comprueba que funciona comparando el resultado con el operador transposición (').

5.7. Aplicación

En la aplicación de esta unidad vamos a utilizar bucles para calcular la energía de repulsión nuclear entre los átomos de una molécula. Para calcular esta energía, utilizaremos la ecuación

$$E_{ij} = \frac{Z_i \cdot Z_j}{R_{ij}}$$

donde Z_i es la carga nuclear del átomo i y R_{ij} es la distancia que separa los átomos i y j. Por lo tanto, necesitamos pedir al usuario que introduzca un vector con las cargas de los N átomos, y una matriz con las distancias internucleares entre átomos. Esta matriz de distancias tiene una serie de características especiales: por un lado, es simétrica (la distancia entre los átomos i y j es independiente del orden de las etiquetas) y presenta ceros en la diagonal (la distancia entre un átomo y sí mismo es nula).

```
Z = input('Introduce el vector de cargas nucleares: ');
R = input('Introduce la matriz de distancias: ');
```

A continuación, vamos a calcular cada uno de los elementos de la matriz $E(i, j)$. Si tenemos en cuenta que los elementos de la diagonal van a ser todos infinito (recuerda: la diagonal de la matriz de distancias se comopone de ceros), podemos evitarnos el cálculo de la diagonal (y de paso, algunos posibles errores o avisos). Podemos inicializar la matriz E de la siguiente forma (piénsalo):

```
N = length(Z);
E = diag(zeros(1,N)*Inf);
```

Como tenemos que calcular la matriz $E(i, j)$ para todos los valores posibles de i y j, debemos utilizar un bucle doble:

```
for i = 1:N
   for j = 1:N
      E(i,j) = Z(i)*Z(j)/R(i,j);
   end
end
```

Pero si tenemos en cuenta que no necesitamos calcular la diagonal, y que la matriz es simétrica, basta con que calculemos uno de los triangulos de la matriz. Para ello, variamos el índice j desde $i+1$ hasta N, calculamos con la ecuación el elemento $E(i,j)$, e igualamos $E(j,i)$ al mismo valor :

```
for i = 1:N
   for j = i+1:N
      E(i,j) = Z(i)*Z(j)/R(i,j);
      E(j,i) = E(i,j);
   end
end
```

En el segundo apartado se nos pide la energía total de repulsión. Para obtenerla, debemos sumar todas las energías de repulsiónentre átomos **sin repeticiones**, es decir, no podemos sumar la energía de repulsión $E(i,j)$ y la energía $E(j,i)$, puesto que representan la misma interacción. Esto se consigue sumando sólo una de las energías. Además, debemos inicializar la variable que acumula la suma antes del bucle:

```
Etotal = 0;
for i = 1:N
   for j = i+1:N
      E(i,j) = Z(i)*Z(j)/R(i,j);
      E(j,i) = E(i,j);
      Etotal = Etotal + E(i,j);
   end
end
```

Por último, la búsqueda del máximo se consigue almacenando el valor mayor calculado en cada momento, y los índices que lo han generado. Basta con añadir unas líneas al código anteriormente escrito. A continuación, se muestra el código completo, con comentarios (fíjate en la inicialización de `Etotal` a $-\infty$ para la búsqueda del máximo). Después se muestra un ejemplo de uso para la molécula de agua.

```
%*****************************************************
% Programa : repulsion.m
%*****************************************************

% Pedimos datos al usuario
Z = input('Introduce el vector de cargas nucleares: ');
R = input('Introduce la matriz de distancias: ');

% Inicializamos E con infinitos en la diagonal
%  y ceros en el resto de elementos (esto no es necesario).
N = length(Z);
E = diag(zeros(1,N)*Inf);
Etotal = 0;
imin = -1;
jmin = -1;
Emax = -Inf;

for i = 1:N
    for j = i+1:N
        E(i,j) = Z(i)*Z(j)/R(i,j);
        Etotal = Etotal + E(i,j);
        E(j,i) = E(i,j);
        % Buscamos máximo
        if E(i,j) > Emax
                Emax = E(i,j);
                imin = i; jmin = j;
        end
    end
end

fprintf('La energía total de repulsión es %g\n', Etotal')
fprintf('Los atomos que mas se repelen son %g y %g.\n', imin,jmin)
```

```
≫ octave -q repulsion.m
Introduce el vector de cargas nucleares: [16 1 1]
Introduce la matriz de distancias: [0.0000 0.9895 0.9895; 0.9895
0.0000 1.5163; 0.9895 1.5163 0.0000]
La energía total de repulsión es 32.9991
Los atomos que mas se repelen son los atomos 1 y 2.
```

Ejercicios

10 Escribe y ejecuta el programa anterior.

5.8. Ejercicios prácticos

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

Ejercicios

▶ **11** Escribe un programa de nombre `pi1.m` que calcule el valor de π utilizando la siguiente serie matemática:

$$\frac{pi^2 - 8}{16} = \sum_{n=1}^{\infty} \frac{1}{(2n-1)^2(2n+1)^2} \quad .$$

¿Cuántas iteraciones son necesarias para obtener π de forma que la precisión asociada al sumatorio sea 1e-12?

▶ **12** Otra forma de calcular pi se basa en el siguiente método:

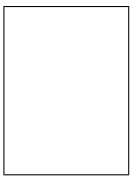
- Inicialización: $a = 1$, $b = 1/\sqrt{2}$, $t = 1/4$ y $x = 1$.
- Repite las siguientes órdenes hasta que la diferencia entre a y b se encuentre por debajo de una cierta precisión:
  1. $y = a$
  2. $a = (a + b)/2$
  3. $b = \sqrt{(b * y)}$
  4. $t = t - x(y - a)^2$
  5. $x = 2 * x$
- Con los valores resultantes de a, b y t, se calcula la estimación de π como:

$$\pi = \frac{(a+b)^2}{4t}$$

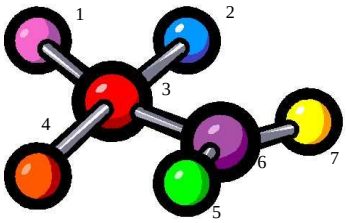
Implementa este programa (`pi2.m`) con OCTAVE y calcula el número de iteraciones necesarias para obtener una precisión de 1e-12. Compara este resultado con el del ejercicio anterior.

▶ **13** ¿Cuál es el valor de la variable `ires` tras la ejecución de este código?

```
ires = 0 ;
for index1 = 10:-2:4
    for index2 = 2:2:index1
       if index2 == 6
          break
       end
       ires = ires + index2;
    end
end
```



▶ 14 Podemos representar el conjunto de enlaces de una molécula de N átomos mediante una matriz D de $N \times N$, en la que el elemento D_{ij} es un 1 si los átomos i y j están enlazados y 0 en caso contrario. Veámoslo con un ejemplo.



$$\begin{bmatrix} 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \end{bmatrix}$$

Fíjate en que la matriz es simétrica (si un átomo está enlazado con otro, el otro lo está a su vez con el primero), y que se ha tomado el convenio de poner ceros en la diagonal, porque no hace falta indicar que un átomo está *enlazado* consigo mismo.

Escribe un programa que pida al usuario una matriz de enlaces y le diga qué pares de átomos están enlazados. Para ello, deberás tener en cuenta lo siguiente:

- El programa deberá verificar que la matriz es simétrica. Recuerda que una matriz es simétrica si es igual a su transpuesta.
- Sólo debe indicar cada par de átomos enlazados *una sóla vez*, y no una por cada átomo del par.

Para facilitarte la escritura del programa, a continuación se muestra un ejemplo de su uso.

```
% Ejemplo de uso:
≫ enlaces
Introduce la matriz de enlaces: [0 0 1 0 0 0 0
                                 0 0 1 0 0 0 0
                                 1 1 0 1 0 1 0
                                 0 0 1 0 0 0 0
                                 0 0 0 0 0 1 0
                                 0 0 1 0 1 0 1
                                 0 0 0 0 0 1 0]
Los atomos 1 y 3 estan enlazados.
Los atomos 2 y 3 estan enlazados.
Los atomos 3 y 4 estan enlazados.
Los atomos 3 y 6 estan enlazados.
Los atomos 5 y 6 estan enlazados.
Los atomos 6 y 7 estan enlazados.
```

▶ 15 Queremos calcular y dibujar la curva de valoración de un ácido fuerte con una base fuerte. Para ello, vamos a escribir un PROGRAMA que pida al usuario los siguientes valores:

- V_a: Volumen inicial de ácido.
- C_a: Concentración del ácido.
- C_b: Concentración de la base.
- V_b: Un *vector* con distintos valores del volumen de la base añadida.

Con estos valores, el PROGRAMA deberá calcular el pH para cada valor de V_b, utilizando para ello las siguientes ecuaciones:

$$pH = \begin{cases} -\log_{10}\left(\dfrac{\Delta}{V_a + V_b} + 10^{-7}\right) & \text{si } \Delta \geq 0 \\ 14 + \log_{10}\left(\dfrac{-\Delta}{V_a + V_b}\right) & \text{si } \Delta < 0 \end{cases}$$

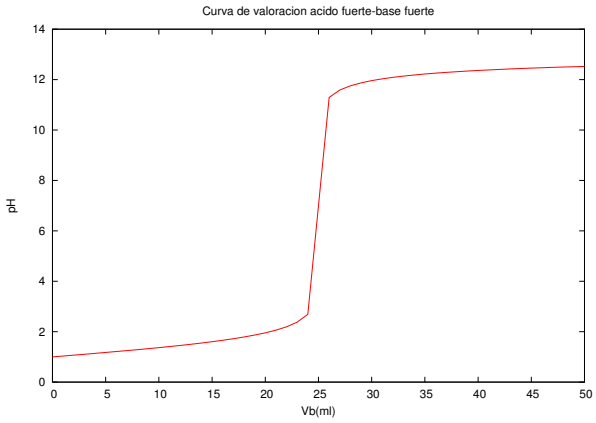
donde Δ se define como

$$\Delta = C_a V_a - C_b V_b \quad .$$

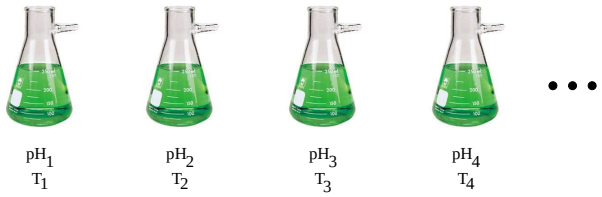
Además, el PROGRAMA deberá representar el pH en función del volumen de base añadido V_B.

Ayuda: La función para calcular el logaritmo decimal en `Octave` es `log10`.

```
% Ejemplo de uso:
≫ pH
Introduce el volumen inicial de ácido (ml): 25
Introduce la concentracion inicial de ácido (M): 0.1
Introduce la concentracion inicial de base (M): 0.1
Introduce un vector de volúmenes de base (ml): [0:50]
```



▶ 16 Se tienen N disoluciones numeradas de 1 a N y se mide el pH y la temperatura de cada disolución.



a) Queremos escribir un programa que pida al usuario el número de disoluciones N, y los valores de temperatura y pH de las N disoluciones, y muestre en pantalla:

- La temperatura media de las disoluciones claramente ácidas (pH < 6.5).
- La temperatura media de las disoluciones claramente básicas (pH > 7.5).
- La temperatura media de las disoluciones neutras ($6.5 \leq pH \leq 7.5$).

A continuación tienes un ejemplo de cómo debería usarse el programa:

```
% Ejemplo de uso:
≫ pH
.........
La temperatura media de las disoluciones ácidas es: 276.33 K
La temperatura media de las disoluciones neutras es: 238.00 K
La temperatura media de las disoluciones básicas es: 280.50 K
```

Puedes pedirle los datos al usuario como creas conveniente.

b) Añade al programa anterior las líneas que consideres oportunas para que evalúe cuál es la disolución de mayor temperatura, y el pH de ésta.

c) Puede que al escribir el programa anterior no hayas tenido en cuenta que pudiera no haber disoluciones en uno de los grupos de pH (por ejemplo, podría no haber disoluciones ácidas). En ese caso, al ejecutar el programa se generaría un error al calcular la media (por una división por cero). Si no lo has hecho ya, incluye las modificaciones necesarias para que, en caso de que no haya disoluciones de un grupo, el programa se lo indique al usuario. Por ejemplo,

```
% Ejemplo de uso:
≫pH
.........
La temperatura media de las disoluciones acidas es: 276.33 K
No hay disoluciones neutras.
La temperatura media de las disoluciones basicas es: 280.50 K
```