Condicionales y bucles

Instrucción 'if'

Esta instrucción nos permite preguntar una condición respecto a los cálculos que estamos manejando y obrar en consecuencia en función de si dicha condición se cumple o no. La forma básica de esta instrucción es la siguiente:

```
if <condición>
<instrucciones>
end
```

De esta forma, si la condición se cumple entonces las instrucciones se ejecutarán. Si no se cumple, las instrucciones se ignorarán.

Veamos como ejemplo sencillo cómo implementaríamos la función del valor absoluto $|x|$ si ésta no apareciera implementada en Octave:

```
function y=abs(x)
if x<0
y=-x;
end
if x>=0
y=x;
end
endfunction
```

Los operadores más usados con estos bucles son los que se muestran a continuación:

Operador	Interpretación
$<$	menor que
$<=$	menor que o igual a
$>$	mayor que
$>=$	mayor que o igual a
$==$	igual a
$\sim=$	distinto de

En Octave se pueden agrupar varias condiciones para una misma instrucción `if` y que la instrucción se ejecute cuando se cumplan todas, o se cumpla alguna. En tal caso, habrá que poner cada condición entre paréntesis y separarlas por el símbolo lógico correspondiente. Los conectores lógicos que usa Octave aparecen en la siguiente tabla:

Símbolo	Operador lógico
$\sim$	no
&	y
\| | o |

Pongamos por ejemplo que queremos calcular las raíces de una parábola y que queremos que el cálculo se produzca solamente cuando tengamos la certeza de que las raíces existen y son reales. Como sabemos que las raíces de la parábola $ax^2 + bx + c$ vienen dadas por

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

podemos programar el código como sigue:

```
if (b^2-4*a*c >= 0) & (a ~=0)
x1=(-b+sqrt(b^2-4*a*c))/(2*a);
x2=(-b-sqrt(b^2-4*a*c))/(2*a);
end
```

Instrucciones 'if' anidadas

El comando `else` nos permitirá introducir una instrucción en el caso de que la condición dada por el comando `if` correspondiente no se cumpla. Así, por ejemplo, la función para el valor absoluto podría implementarse de la siguiente forma:

```
function y=abs(x)
if x<0
y=-x;
else
y=x;
end
endfunction
```

Este procedimiento funciona si sólo tenemos que considerar dos situaciones: que una condición se cumpla o que no. En caso de que haya más de dos condiciones que determinen qué instrucciones ejecutar, deberemos usar el comando `elseif`. El siguiente método computa todas las soluciones posibles reales de una ecuación de segundo grado $ax^2 + bx + c = 0$:

```
function sol=cuad(a,b,c)
if a ~=0
if b^2-4*a*c>=0
x1=(-b+sqrt(b^2-4*a*c))/(2*a);
```

```
x2=(-b-sqrt(b^2-4*a*c))/(2*a);
disp('La ecuación tiene dos soluciones reales, que son')
disp(x1)
disp('y')
disp(x2)
else
disp('La ecuación no tiene soluciones reales.')
end
elseif b~=0
disp('La ecuación es una ecuación lineal con solución')
disp(-b/a)
elseif c~=0
disp('La ecuación es una identidad falsa que por lo tanto no tiene solución.')
else
disp('La ecuación es una identidad verdadera y por lo tanto cualquier número es solución.
end
endfunction
```

Bucles

Un bucle es una estructura de programación que permite la repetición controlada de un conjunto de instrucciones. Los bucles que veremos a continuación son los más utilizados no sólo en Octave, sino en la inmensa mayor parte de los lenguajes de programación.

- Bucle `for`. Este bucle ejecuta una determinada instrucción tantas veces como se lo indique el usuario. Su estructura es la siguiente:

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

  Así, el índice `i` creado ex-profeso para este bucle recorrerá ordenadamente las coordenadas del vector `vector` y para cada una de ellas ejecutará las instrucciones indicadas por el usuario. En particular, el usuario deberá de cuidarse de que el nombre del índice escogido no coincide con el de ninguna de las variables creadas para el programa, o habrá problemas de cruce de referencias. Por otra parte, no hace falta darle el nombre de `i`: cualquier otro nombre lo hará funcionar igual.

  Normalmente el vector de índices será una colección finita de números consecutivos empezando en 1: `i=1:n`, pero el bucle funcionará con cualquier vector que se proponga.

  Si queremos realizar un código que construya el vector

  $$v = \left(1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{10}\right)$$

operaríamos de la siguiente manera:

```
for i=1:10
v(i)=1/i;
end
```

Así, el vector v se iría creando en cada iteración de `i`. Hagamos otro programa, que proporcione la suma de los elementos de un vector (lo cual ya podemos hacer con el comando `sum`):

```
function x=sum(v)
x=0;
for i=1:length(v)
x=x+v(i);
end
endfunction
```

Notemos que hemos tenido que iniciar la suma con `x=0;` para poder tener la variable creada, o el programa daría problemas. Esta técnica es muy socorrida para casos donde tengamos que realizar sumas o productos con elementos cuyo límite no podemos controlar (vectores de cualquier longitud, matrices de cualquier dimensión...).

Como segundo ejemplo, y nuevamente aunque Octave ya tiene una función para ello, vamos a implementar una función que nos calcule el factorial de un número.

```
function x=fact(n)
if (n ~=floor(n))|(n ~=abs(n))
display('necesito un enterno no negativo').
elseif n==0
x=1;
else
x=1;
for i=1:n
x=x*i;
end
end
endfunction
```

Veamos otro programa que tome una matriz cuadrada y lo que haga sea reflejar la triangular superior sobre la inferior, es decir, cree una matriz simétrica a partir de los elementos de la triangular superior:

```
function B=TriangSim(A)
[n,m]=size(A)
if n~=m
display('La matriz introducida no es simétrica.')
else
for i=2:n
for j=1:i-1
B(i,j)=A(j,i);
end
end
end
endfunction
```

Por último, hagamos un programa que calcule el término $n-$ésimo de la sucesión de Fibonacci (que, como todo el mundo sabe, es una sucesión que se inicia con 0 y 1 y después calcula el siguiente término mediante la suma de los dos anteriores: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$).

```
function x=Fib(n)
if (n~=floor(n))|(n~=abs(n))
display('Necesito que el número sea un entero positivo.')
elseif n=1
x=0;
elseif n=2
x=1;
else
x1=0;
x2=1;
for i=3:n
x=x1+x2;
x1=x2;
x2=x;
end
end
endfunction
```

- Bucle `while`. Los bucles `while` ejecutan unas intrucciones mientras que determinada condición lógica se verifica. La sintaxis es como sigue:

```
while <condición>
<instrucciones>
end
```

El riesgo que pueden tener estos bucles es que la condición no deje de verificarse, y el programa se vea atrapado en un bucle infinito, en cuyo caso deberemos esperar que el comando `break` pueda sacarnos de éste. En caso de que no pueda, tendremos que reiniciar el programa.

Nótese que un bucle `for` puede transformarse en un bucle `while`. Pongamos el ejemplo de la suma de las coordenadas de los vectores. Si queremos usar un bucle `while` podríamos escribir

```
function x=sum(v)
x=0;
i=0;
while i<n
i=i+1;
x=x+v(i);
end
endfunction
```

Método de Jacobi para aproximación de soluciones de ecuaciones lineales.

Con lo visto anteriormente tenemos las herramientas para hacer programas complicados, entre los que se encuentran los métodos iterativos, es decir, algoritmos que consisten en la repetición de un mismo proceso varias veces, de tal forma que se construye una sucesión. Nos vamos a detener en el llamado método de Jacobi, el cual permite encontrar soluciones de sistemas del tipo $Ax = b$ (recordemos que `inv(A)*b` podía empezar a dar problemas al poco que la matriz fuese complicada). El método es el que sigue:

1. Empezamos con un valor cualquiera x_0.
2. x_n lo obtenemos del término anterior, x_{n-1}, de la siguiente forma: la coordenada $k-$ésima la obtenemos a partir de la fórmula

$$x_n(i) = \frac{1}{a_{ii}} \left(- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_{n-1}(j) + b_i \right), \quad i = 1, \ldots, n.$$

Si consideramos ese método, la sucesión $\{x_n\}$ creada verifica que tiende a la solución del sistema $Ax = b$. En particular, se verifica que la distancia entre dos elementos consecutivos tiende a cero: $\lim_{n\to\infty} \|x_n - x_{n+1}\| = 0$.

Así, si queremos implementar el método con 100000 iteraciones, podríamos escribir el siguiente código:

```
for k=1:100000
y=x;
n=length(x);
for i=1:n
cuant=0;
for j=1:n
cuant=cuant+A(i,j)*y(j);
end
x(i)=1/A(i,i)*(-cuant+A(i,i)*y(i)+b(i));
end
end
```

Ejercicio. Adaptar el código anterior para un programa cuyas variables sean el número de iteraciones, N, la matriz del sistema, A, el vector de términos independientes, b, y el valor inicial, x. Incluir además un parámetro de control, d, de forma que si $\|x_n - x_{n-1}\| < d$ entonces el programa interrumpa el proceso (porque consideraremos que estamos lo suficientemente cerca de la solución).

Es decir, el programa debería tener como cabecera algo parecido a

```
function z=Jacobi(N,A,b,x,d)
```