7. Bucles

Es frecuente que al realizar un algoritmo sea preciso repetir un paso durante un número determinado de veces. Estas repeticiones se conocen como *bucles* y en Matlab se implementan mediante los comandos `for` y `while`, dependiendo de si se conoce o no por anticipado el número de repeticiones del bucle.

Bucle For:

El comando `for` permite repetir el bucle el número de veces que se especifique mediante los términos de un vector. Su sintaxis es de la forma:

```
for variable=vector
    < Hacer cosas >
end
```

Por ejemplo, el siguiente código genera los cuadrados de los números del 1 al 10:

```
cuadrados=[];
for i=1:10
    cuadrados=[cuadrados,i^2];
end
cuadrados
```

cuyo resultado es:

```
cuadrados =

     1     4     9    16    25    36    49    64    81   100
```

NOTA: Hemos usado un bucle `for` en este caso simplemente para ilustrar como funciona dicho bucle. En realidad Matlab/Octave nos permite calcular los cuadrados de los números del 1 al 10 de una manera más sencilla simplemente mediante:

```
>> [1:10].^2
```

```
ans =

    1     4     9    16    25    36    49    64    81   100
```

En este caso estamos haciendo lo que se conoce como *aritmética vectorial*. En lugar de repetir mediante un bucle la misma operación para cada término de un vector, simplemente aplicamos dicha operación al vector completo. No todos los problemas prácticos son vectorizables como en este caso.

El vector de índices sobre los que se ejecuta el `for` no tiene por qué ser necesariamente una sucesión de valores correlativos. Por ejemplo, supongamos que se desea construir una función tal que, dado un valor x, devuelva:

$$f(x) = x + \frac{x^2}{2} + \frac{x^4}{4} + \frac{x^5}{5} + \frac{x^7}{7}$$

Una forma de hacerlo utilizando un bucle `for` sería la siguiente:

```
function y=f(x)
  y=0;
  for k=[1 2 4 5 7]
    y=y+(x^k)/k;
  end %for
end %function
```

```
>> f(3)
```

```
ans =  388.78
```

Aunque, al igual que en el caso anterior existe la posibilidad de utilizar aritmética vectorial para realizar el cálculo; en tal caso la función podría programarse así:

```
function y=g(x)
  k=[1 2 4 5 7];
  y=sum((x.^k)./k);
end %function
```

Vemos que el resultado es el mismo que utilizando el bucle `for`:

```
>> g(3)
```

```
ans =  388.78
```

En general siempre resulta más eficiente desde el punto de vista del tiempo de cómputo utilizar la aritmética vectorial antes que un bucle de programación.

Ejemplo: Sucesión de Fibonacci

La sucesión de Fibonacci es una sucesión clásica en matemáticas que se utiliza en múltiples aplicaciones. Comienza con los números 0 y 1 y a partir de ahí cada término se obtiene como suma de los dos anteriores. A continuación mostramos una función que implementa un bucle `for` para generar los primeros n términos de la sucesión de Fibonacci, con $n \geq 2$:

```
function [F]=Fibonacci(n)
  % Esta función calcula los n primeros términos de la sucesión de Fibonacci
  if n!=floor(n)
    F=[];
    disp('Aviso: n debe ser un  valor entero')
  elseif n<=0
    F=[];
    disp("Aviso: n debe ser mayor o igual que cero");
  elseif n==1
    F=[0];
  elseif n==2
    F=[0 1];
  else
    F=[0 1];
    for k=3:n
      F(k)=F(k-1)+F(k-2);
    end %for
  end %if
end %function
```

Para construir la sucesión de Fibonacci de esta forma (es decir, de manera recurrente, calculando cada término como la suma de los dos anteriores) no es posible utilizar aritmética vectorial.

Bucle While:

El comando `while` hace que se repita un bucle mientras se cumpla una condición; el bucle se detiene cuando dicha condición deja de cumplirse. Su sintaxis es de la forma:

```
while condicion
    < Hacer cosas >
end
```

Usaremos un bucle `while` en lugar de un bucle for cuando no sea posible determinar **a priori** el número de veces que hay que iterar el bucle.

Ejemplo: Supongamos que dado un valor x_{max} se desea construir una función que determine el mayor valor de n tal que

$$0^2 + 1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2 \leq x_{max}$$

La función podría programarse usando un bucle while del siguiente modo:

```
function [n]=cuadradosNmax(xmax);
  % Esta función determina el mayor entero n tal que la suma de los
  % cuadrados de los valores enteros de 0 a n es menor o igual que xmax
  if xmax<0
     disp("Aviso: ha especificado una suma máxima negativa");
     n=[];
   else
     n=0;
     suma=0;
     while (suma<xmax)
       n=n+1;
       suma=suma+n^2;
       if (suma>xmax) n=n-1;
      end %while
  end %if
end %function
```

Así, por ejemplo, el mayor valor de n tal que $\sum_{k=1}^{n} k^2 < 100$ sería

```
>> cuadradosNmax(100)
```

```
ans =  6
```

Ejemplo: Construir una función que devuelva todos los términos de la sucesión de Fibonacci menores

que un valor x preespecificado.

La función podría implementarse como:

```
function [F]=FibonacciMenorQue(x)
  % Esta función calcula los términos de la sucesión de Fibonacci
  % menores que x
  if x<=0
    F=[];
    disp("Aviso: x debe ser mayor que cero");
  elseif x<=1
    F=[0];
  else
    F=[0 1];
    k=2;
    while F(k)<x
      k=k+1;
      F(k)=F(k-1)+F(k-2);
    end %while
    F=F(1:(k-1));
  end %if
end %function
```

```
>> FibonacciMenorQue(100)
```

```
ans =

    0    1    1    2    3    5    8   13   21   34   55   89
```

Otra implementación un poco más elegante sería la siguiente:

```
function [F]=FibonacciMenorQueB(x)
  % Esta función calcula los términos de la sucesión de Fibonacci
  % menores que x utilizando un bucle while
  if x<=0
    F=[];
    disp("Aviso: x debe ser mayor que cero");
  else
    F=[0];
    k=1;
    siguiente=1;
    while siguiente<x
      k=k+1;
      F(k)=siguiente;
      siguiente=F(k)+F(k-1);
    end %while
  end %if
end %function
```

```
>> FibonacciMenorQueB(100)
```

```
ans =

    0    1    1    2    3    5    8   13   21   34   55   89
```