PRÁCTICA 3. MATLAB

Tipos de datos, optimización de código.

1. TIPOS DE DATOS

A parte de los formatos que ya se han visto en prácticas anteriores de Matlab tenemos los siguientes tipos de datos que se pueden utilizar y que aportan otro tipo de funcionalidad.

1.1 Caracteres

En Matlab las cadenas de caracteres, o simplemente cadenas, son simplemente texto entrecomillado y se manejan como vectores filas.

```
>> prueba='Esto es una cadena de caracteres'
prueba =
   Esto es una cadena de caracteres
```

Al ser vectores se pueden concatenar cadenas como los vectores numéricos:

```
>> [prueba ‘concatenada’]
ans =
     Esto es una cadena de caracteres concatenada
```

Incluso se puede trabajar de forma matricial haciendo que cada fila sea un nuevo string. El problema en este caso es que la longitud de la cadena tiene que ser siempre la misma, sino da error de dimensión.

```
>> [Prueba; Prueba]
ans =
   Esto es una cadena de caracteres
   Esto es una cadena de caracteres
```

Al ser matrices (aunque normalmente trabajaremos con vectores fila) se pueden acceder a los caracteres aislados de igual forma que accedemos a los elementos de las matrices mediante indexación cumpliéndose las mismas premisas de acceder a elementos aislados o a varios simultáneamente.

```
>> ultimochar=Prueba(end)
ultimochar =
   s

>> pruebainversa = Prueba(length(Prueba):-1:1)
pruebainversa =
   seretcarac ed anedac anu se otsE
```

Los arrays de caracteres en realidad se trata de una traducción que hace Matlab en formato ASCII de valores de tipo int8. Por tanto se puede trabajar con variables definidas como cadenas de texto con operaciones matemáticas. Lo único es que cuando se hace una operación de esta clase los datos se convierten en números y dejan de ser cadenas de texto.

```
>> pruebainversa + 10
ans =
  Columns 1 through 9
   125   111   124   111   126   109   107   124   107
  Columns 10 through 18
   109    42   111   110    42   107   120   111   110
  Columns 19 through 27
   107   109    42   107   120   127    42   125   111
  Columns 28 through 32
    42   121   126   125    79
```

Mediante las funciones `int8` y `char` se pueden convertir las cadenas a arrays numéricos y viceversa.

```
>> char(Prueba + 10)
ans =
   O}~y*o}*□xk*mknoxk*no*mk|km~o|o}
```

Además la función `char` permite generar matrices de caracteres aunque estos inicialmente no tengan la misma longitud, para ello rellena las cadenas de cada fila con espacios en blanco.

```
>> A = char('Cad', 'Perro', 'Fuente');
>> size(A)
ans =
     3    6
```

Funciones útiles para trabajar con strings

- `strcmp:` compara cadenas y retorna 1 para cierto y 0 para falso

  ```
  booleano = strcmp('cadena1', 'cadena 2);
  ```

- `strrep:` find-and-replace

  ```
  cadena = strrep(cadena,'busca','sustituye');
  ```

- `findstr:` busca una cadena dentro de otra

  ```
  posicion = findstr('busca', cadena);
  ```

- `strcat:` concatena 2 o más cadenas

```
texto = strcat(cadena1, cadena2, cadena3);
```

- `sprintf:` construye una cadena a partir de variables.

```
Cadena = sprintf('Tengo %6.2f EUR',mi_dinero);
```

Otra función útil para trabajar con cadenas largas, sobre todo cuando se leen textos es la función `regexp`, esta función trabaja con expresiones regulares para buscar patrones en cadenas de caracteres. Esta función recibe como entrada una cadena de texto y una expresión a buscar en la cadena y devuelve un resultado en formato cell.

```
out = regexp(cadena,expresion,operación)
```

Las opciones de esta variable son bastante numerosas, y funciona de forma parecida a otros lenguajes de programación. Por ejemplo para separar las palabras de una cadena de texto por los espacios entre palabras

```
out = regexp(cadena,' ','split')
```

Con este formato devuelve cada cadena de texto que haya entre dos caracteres de tipo espacio en un elemento de un cell array. Las opciones de esta función son numerosas por lo que para ver otros ejemplos y otras opciones es conveniente consultar la ayuda de Matlab.

1.2 Fechas

Hasta la versión R2014b Matlab no tenía un tipo de variable reservado para fechas y horas, para trabajar con fechas y horas Matlab utiliza los formatos conocidos aplicando un formato concreto para que las fechas sean entendibles por el usuario. En las versiones nuevas de Matlab existe un tipo de dato

Matlab representa las fechas y las horas en tres formatos dependiendo del tipo de variable que se utilice:

- Vectores: se utilizan vectores de seis elementos numéricos para representar las fechas. [año mes dia hora minutos segundos]

```
>> [2016 03 09 01 12 00]
```

- String: se utiliza un array de caracteres para representar las fechas. El formato del string es configurable por medio de modificadores de formato, aunque Matlab tiene varios formatos predefinidos.

```
>> ’09-Mar-2016 01:12:00’
```

- Número: también se puede utilizar un formato numérico que tiene un formato fijo y que es el número de días transcurridos desde el 0 de Enero de 0000

```
>> 736398.05
```

Tanto para declarar una variable de tipo fecha en cualquiera de sus formatos como para convertir de un formato a otro se utilizan las siguientes funciones:

`datenum`	convierte la fecha a formato numérico
`datevec`	convierte la fecha a formato vector
`datestr`	convierte la feche a formato str

Cuando se define una fecha como string hay que establecer el formato en el que se pasa la variable a la función para que Matlab entienda cual es cada uno de los componentes. Este formato para declarar una variable fecha se puede pasar o bien mediante un número de formato que Matlab tiene registrado como formatos predeterminados o bien se puede establecer mediante un string que especifica el formato de fecha que se va a utilizar. Existen algunos formatos que Matlab es capaz de reconocer aunque no se le pasen parámetros, pero se corre el riesgo de que Matlab los malinterprete, sobre todo, en el caso de malinterpretación del día - mes

```
>> datestr('01-Mar-2000', 1)
>> datestr('01-Mar-2000', 'dd-mmm-yyyy')
```

Las fechas en Matlab no tienen referencia de zonas horarias ni de cambios por horarios de verano por lo que hay que tenerlo presente cuando se está trabajando con fechas. Además si se quiere trabajar con fechas de forma matemática es necesario que las fechas estén en formato numérico ya que en caso contrario no se puede hacer ningún tipo de conversión. En Matlab lo habitual es trabajar con el formato string en fechas cuando se hace la lectura y escritura de ficheros y se plotean gráficos. Para el resto de operaciones los cálculos se realizan en formato numérico.

Otras funciones útiles para trabajar con fechas en Matlab son:

`now`	devuelve el instante actual en formato numérico.
`date`	devuelve la fecha únicamente en formato string.
`clock`	devuelve la fecha y hora en formato de vector.

Ejemplo

Queremos trabajar con la fecha 09/03/2016 y desfasarla dos días, cuatro horas. Luego mostraremos el resultado por pantalla

```
>> Fecha = datestr('09/03/2016', 'dd/mm/yyyy')
Fecha =
       03/09/2016
>> Fechanum = datenum(Fecha)
Fechanum =
     736398
>> Fechanum = Fechanum + 2.25
Fechanum =
       736400.25
>> datestr(Fechanum)
ans =
       11-Mar-2016 06:00:00
>> datevec(Fechanum)
ans =
  Columns 1 through 4
       2016           3          11           6
  Columns 5 through 6
         0      0
```

1.3 Structs

Una estructura es una agrupación de datos de tipo diferente bajo un mismo nombre. Estos datos denominados campos llevan asociados un nombre interno que permite diferenciarlas. La ventaja de las estructuras es que permiten asociar bajo un mismo nombre de variable datos de diferente naturaleza.

En Matlab no es posible tener en una matriz datos de diferente naturaleza, si se crea una matriz de tipo numérico solo se pueden introducir números y la matriz siempre tiene que tener la misma estructura. Una estructura permite almacenar combinar diferentes tipos de datos y así poder acceder a distintas características definitorias de una variable.

Por ejemplo se puede crear una variable que almacene diferentes datos de relativos a un alumno como puede ser por un lado el nombre que sería una cadena de caracteres y por otro el número de carnet. Para crear una estructura no es necesario definir previamente el modelo o el patrón de la estructura y se pueden ir añadiendo los campos a medida que es necesario definirlos.

```
>> Alumno.nombre='Miguel'
Alumno =
      nombre: 'Miguel
>> Alumno.carnet=75482
Alumno =
      nombre: 'Miguel'
      carnet: 75482
>> Alumno
Alumno =
      nombre: 'Miguel'
      carnet: 75482
```

Cuando se escribe la variable de la estructura se devuelven todos los campos de la estructura, en el caso de que solo queramos acceder a un campo concreto se utiliza el operador punto (.)

```
>> Alumno.nombre
ans =
      Miguel
```

La estructura también puede crearse por medio de la función struct() que permite definir varios campos de una estructura de forma simultánea. Cada campo se define por medio de un par por el que se indica siempre nombre del campo indicado como string y el valor del mismo que se pasará en la forma correspondiente.

```
>> Alumno = struct('nombre', 'Ignacio', 'carnet', 76589)
Alumno =
      nombre: 'Ignacio'
      carnet: 76589
```

Dentro de una estructura se pueden almacenar matrices e incluso otras estructuras, haciendo el acceso a elementos internos también por medio del operador punto.

```
>> clase = struct ('curso', 'primero', 'grupo', 'A',
'alumno', struct ('nombre', 'Juan', 'edad', 19))
clase =
    curso: 'primero'
    grupo: 'A'
    alumno: [1x1 struct]
>> clase.alumno.edad
ans =
    17
```

También es posible la creación de arrays de estructuras de esta forma se pueden crear varios elementos con la misma estructura, lo que nos permite agrupar varios datos cuando tienen los mismos campos. La forma de crear estas estructuras es utilizando un índice en la variable de la estructura.

```
>> Alumno(10) = struct ('nombre', 'Ignacio', 'carnet',
76589)
Alumno =
      1x10 struct array with fields:
      nombre
      carnet
```

En este caso los datos se guardan en el elemento 10, si los otros elementos del array no se han inicializado, automáticamente se crean con elementos vacíos.

También es posible acceder a los campos de un struct de forma dinámica haciendo el acceso al campo por medio de otra variable que es la que va a contener el campo concreto. De esta forma se puede cambiar el campo por programación al que se accede.

```
>> campo = ‘nombre’;
>> Alumno(10).(campo)
ans =
      Ignacio
```

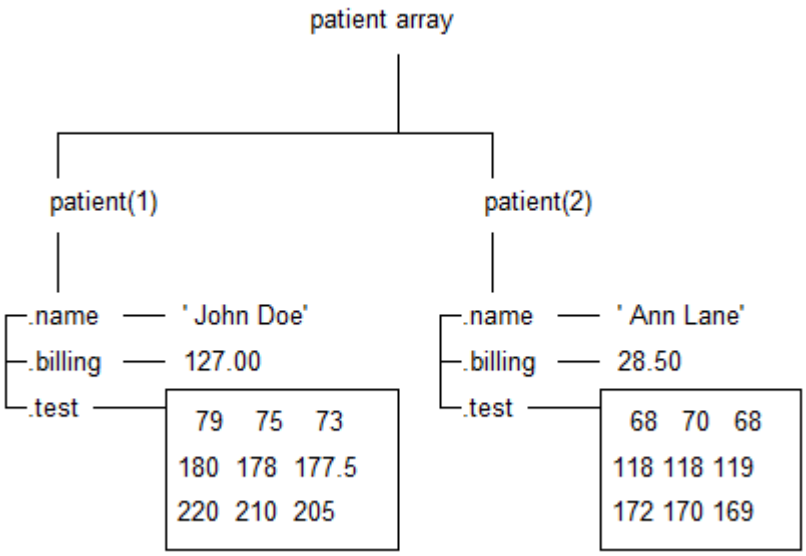
Funciones útiles

`fieldnames`	devuelve un cell con los nombres de los campos.
`isfield`	permite saber si un campo existe en una estructura
`isstruct`	permite saber si ST es o no una estructura
`rmfield`	elimina el campo indicado de la estructura

`getfield`	devuelve el valor del campo especificado
`setfield`	da el valor indicado al campo de la estructura

Ejemplo

Queremos crear una variable que guarde datos de varios pacientes de acuerdo a la siguiente estructura.




```
% Creamos los datos del primer paciente
patient(1).name = 'John Doe';
patient(1).billing = 127.00;
patient(1).test = [79, 75, 73; 180, 178, 177.5; 220, 210, 205];

% Creamos los datos del segundo paciente
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68, 70, 68; 118, 118, 119; 172, 170, 169];
```

1.4 Cell arrays

Un cell array es un tipo de dato que internamente funciona como una matriz pero que permite que cada uno de los elementos sea una variable de cualquier tipo. En un array ordinario todos los elementos son números o cadenas de caracteres. Sin embargo, en un cell, se pueden combinar números, strings, estructuras e incluso otros elementos de tipo cell.

Un cell array tiene una función parecida a las estructuras que hemos visto con anterioridad. La idea es poder combinar datos de diferente naturaleza bajo una misma variable con el fin de facilitar el acceso a los datos. A diferencia de las estructuras que la ordenación se hace por medio de campos que tienen un nombre asociado, los cell se

ordenan en base a su posición en la estructura matricial (el objetivo es que cada fila sea un dato y cada columna una característica de ese dato)

Un cell se crea de forma similar a las matrices pero en lugar de utilizar corchetes [] para añadir los elementos se utilizan llaves {}. En este caso se crea un cell cuyas dimensiones son 2 filas y tres columnas con diferentes elementos.

```
>> myCell = {1, 2, 3; 'text', rand(5,10,2), {11; 22; 33}}
myCell =
    [   1]    [            2]    [       3]
    'text'    [5x10x2 double]    {3x1 cell}
```

También es posible crear cell con componentes vacíos o se puede inicializar un cell si sabemos el tamaño de antemano.

```
 >> C = {} % Se crea un cell vacio
>> emptyCell = cell(3,4) % Se crea un cell vacio de dimensiones 3 x 4
```

El problema de trabajar con cells es que al igual que las estructuras es solo una forma de ordenar los datos, es decir, no se pueden realizar operaciones matriciales con esos datos directamente sino que hay que extraer esos datos de su posición para poder utilizarlos (existen funciones que admiten como entrada variables de tipo cell que son tratados internamente).

```
>> myCell .^ 2
Undefined function 'power' for input arguments of type 'cell'.
```

Al igual que con las matrices se pueden acceder a los elementos de un cell por medio del indexado de subíndices, es decir, por medio de coordenadas de filas y columnas podemos obtener un elemento de un cell aislado o podemos obtener varios elementos.

```
>> myCell(1,2)
ans =
    [2]
>> myCell(:,1)
ans =
    [1]
    'text'
```

Sin embargo, cuando realizamos este tipo de indexación el resultado que obtenemos sigue siendo un dato de tipo cell array, es decir, sirve únicamente para hacer selecciones de datos. Para poder acceder al valor interno que está almacenado en un elemento de un cell utilizamos las llaves {}.

```
>> A = myCell{1,2}*2
A =
     2
>> A = myCell(1,2)*2
Undefined function 'mtimes' for input arguments of type
'cell'.
```

La primera instrucción devuelve el valor almacenado en el cell y se pueden realizar una operación matemática mientras que la segunda devuelve un cell por lo que no puede someterse a una operación matemática. Cuando se accede a los valores que almacenan los elementos de un cell solo se puede acceder a uno cada vez, si se intentan acceder a varios solo permanece el primer valor al que se accede.

Para modificar el valor almacenado en un elemento el acceso hay que hacerlo también por medio de llaves. Hay dos opciones:

```
>> myCell{(1,1)} = 5;
>> myCell(1,1) = {5};
```

En el caso que el acceso se haga sin llaves Matlab devuelve un error:

```
>> myCell(1,1) = 5
Conversion to cell from double is not possible.
```

También es posible almacenar un struct en un cell y acceder a los campos que contiene ese struct.

```
Cell{i,j}.Campo
```

Funciones útiles

`cell`	crea un cell array vacío de m filas y n columnas
`celldisp`	muestra el contenido de todas las celdas
`iscell`	indica si la variable es un cell
`num2cell`	convierte un array numérico en un cell array
`cell2struct`	convierte un cell array en una estructura

`struct2cell`	convierte una estructura en un cell array
`cell2mat`	convierte un cell en una matriz

Ejemplo

Replicamos el problema de la estructura en un variable de tipo cell.

```
% Creamos los datos del primer paciente
patient(1,1) = {'John Doe'};
patient(1,2) = {127.00};
patient(1,3) = {[79, 75, 73; 180, 178, 177.5; 220, 210, 205]};

% Creamos los datos del segundo paciente
patient(2, :) = {'Ann Lane'; 28.50; [68, 70, 68; 118, 118, 119; 172,
170, 169]};
```

2. OPTIMIZACIÓN DE CÓDIGO

En Matlab existen una serie de medidas para optimizar el código y reducir los tiempos de ejecución. Las medidas más eficaces que se utilizan son:

- Variables globales.
- Reserva de memoria de las matrices.
- Vectorización de las operaciones.
- Enmascaramiento de las operaciones.

A parte de estas medidas, otra medida que también reduce los tiempos de ejecución es evitar en todo lo posible la impresión de variables por pantalla. Olvidarse del punto y coma al final de cada instrucción supone bastante tiempo de ejecución en Matlab.

Variables globales

El uso de variables globales para llamadas a funciones está recomendado cuando sobre una variable se van a hacer varias operaciones sobre una variable y las dimensiones de la misma son elevadas.

```
global variable = []
```

En Matlab cuando se llama a una función el paso de las variables se hace por valores, Matlab no siempre copia esta variable en el stack sino que hace un paso por referencia, por lo que esta variable sólo se modifica cuando se hacen operaciones sobre

la misma. Si la variable finalmente no se altera internamente, es decir, sólo se lee Matlab no gasta memoria en hacer otra variable para utilizarla dentro de la función.

El problema surge cuando esa variable se va a modificar internamente, ya que tiene que copiar la variable entera para luego trabajar sobre ella y dejar la otra inalterada, el problema es cuando la matriz es grande que la copia es ineficiente por el gran uso de memoria (una variable grande se asignaría varias veces dentro de la función para luego ser eliminadas una vez acabada la función.

Por este motivo cuando se trabaja con variables muy grandes en Matlab y estas se van a modificar internamente en una función es recomendable utilizar variables globales, en caso contrario en Matlab es mejor no utilizar estas variables ya que Matlab no las gestiona bien.

Reserva de memoria de las matrices

Matlab no requiere que las variables se inicialicen ni que se reserve memoria para trabajar con ellas. La siguiente operación:

```
>> A = zeros(100); A = [A ones(100)]
```

genera automáticamente una variable llamada A en la que se reserva espacio para una matriz de 100 x 100 y a continuación se concatena otra matriz de 100 x 100 de unos por lo que al final la matriz A tiene unas dimensiones de 200x100. En ningún momento hemos tenido que decirle a Matlab las dimensiones finales de la matriz sino que Matlab automáticamente ha realizado esta operación.

Esta capacidad de Matlab para hacer crecer el tamaño de las matrices es útil sobre todo cuando hacemos crecer una matriz dentro de un bucle porque no sabemos la dimensión final de la matriz.

Sin embargo, cuando sabemos de antemano el tamaño de la matriz conviene inicializar la matriz con el tamaño que va a tener y se rellenan los valores de la matriz mediante la asignación a los subíndices correspondientes, al igual que se hace en otros lenguajes. Esta asignación evita el tiempo de redimensionamiento de las matrices.

Ejemplo

Calculo de una matriz de Hilbert (es una matriz cuadrada cuyos campos constituyen una fracción de la unidad $H_{ij} = 1 / (i + j - 1)$).

```
%% No se define a=zeros(n)

function a = hilb0(n)
    for j=1:n;
      a(1:n,j) = 1./((0:n-1)+j)';
    end;

% tiempo de ejec.  39.418 sg

      %%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Se define a=zeros(n)

function a = hilb00(n)
    a = zeros(n);
    for j=1:n;
      a(1:n,j) = 1./((0:n-1)+j)';
    end;

% tiempo de ejec.  0.1990 sg
```

Vectorización de las operaciones

Los bucles en Matlab son bastante ineficientes, cada iteración en un bucle `for` o `while` suponen ralentizar bastante el código. En la medida de lo posible es necesario reducir los bucles en la ejecución de código de Matlab y realizar las operaciones por matrices o vectores.

Muchas de las operaciones en Matlab pueden repensarse en forma vectorial reduciendo los tiempos de ejecución, operaciones como `sum, diff, prod`… permiten realizar operaciones que evitan tener que hacer un bucle que recorra las filas de las columnas y se pueden hacer sobre la matriz.

Ejemplo

En el caso del cálculo de la matriz de Hilbert del problema anterior si se rellena cada elemento de forma individual o se realiza el relleno de forma vectorial rellenando cada fila.

```
%% Se define cada elemento de la matriz de forma individual por
% medio de bucles anidados

function a = hilb1(n)
    for j=1:n;
      for j=1:n;
      a(i,j) = 1/(i+j-1);
      end;
    end;
```

```
% tiempo de ejec.  48.825 sg

      %%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Se define cada fila de la matriz de forma vectorial de manera
% que cada fila se hace en un único paso.

function a = hilb100(n)
    a = zeros(n);
    for j=1:n;
      a(1:n,j) = 1./((0:n-1)+j)';
    end;

% tiempo de ejec.  0.672 sg
```

Enmascaramiento de operaciones

El enmascarado de operaciones consiste en utilizar condiciones lógicas para evitar el uso de bucles condicionales. Se trata de obtener matrices de tipo lógico de 1 y 0 en base a operaciones lógicas que nos indican si ese elemento de la matriz cumple o no un determinado criterio.

Cuando se realiza una operación sobre una matriz a la que se le pasa una máscara, la operación solo se aplica a los elementos en los que la máscara es igual a 1. Esto nos permite realizar operaciones sobre todos los elementos de una matriz que cumplen una determinada condición sin necesidad de recorres dichos elementos evitando el uso de bucles.

Las operaciones de enmascarado se vieron en la práctica anterior, junto con la función find, en las nuevas versiones de Matlab, cuando la operación se va aplicar directamente sobre la matriz no es necesario utilizas la función find, ya que se puede indexar directamente con la matriz lógica, con lo que esta función se reserva únicamente cuando la posición del elemento tiene importancia y necesitamos obtener dicha posición.

Ejemplo

Calcular el resultado de la siguiente función:

$$f(x) = \begin{cases} 1 + \sin(2\pi x), & \text{si } |x| > 0.5, \\ 0, & \text{si } |x| \leq 0.5. \end{cases}$$

```
Ejecución de la función mediante bucles condicionales
comprobando si cada elemento cumple la condición anterior
```

```
% x=rand(1000,1); tic; mask1(x); toc
function f = mask1(x)
f=zeros(size(x));
    for i = 1:length(x)
      if abs( x(i)) > 0.5
        f(i) = 1 + sin(2*pi*x(i));
      end;
end;


% tiempo de ejecución 0.8218 sg

%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Ejecución de la función mediante operación de enmascarado
%%
% x=rand(1000,1); tic; mask2(x); toc
function f = mask2(x)
    f=zeros(size(x));
    mask = abs(x) > 0.5;
    f(mask) = 1+ sin(x(mask)*2*pi);


% tiempo de ejecución 0.0451 sg
```

Medición de tiempos de ejecución

En Matlab existen dos métodos fundamentalmente para medir el tiempo que tarda en ejecutarse un código determinado:

- Las funciones `tic toc:` miden el tiempo en segundos transcurrido entre las llamadas. Sirven únicamente para mostrar por pantalla el tiempo de ejecución de una serie de instrucciones

```
>> tic; inv(inv(inv(randn(1000)))); toc
Elapsed time is 10.015000 seconds.
```

- La función `cputime`: indica el tiempo de CPU en segundos. Este tiempo es absoluto de CPU por lo que se hace es llamar a la función en el instante de inicio de medición, almacenarlo en una variable y luego sacar la diferencia de tiempo llamando otra vez a la función cuando se termina el código que queremos evaluar. La ventaja de esta función es que nos permite almacenar el valor en una variable para trabajar luego con ella.

```
>> t=cputime; inv(inv(inv(randn(1000)))); e=cputime-t
e=
  9.5137
```

Profiler

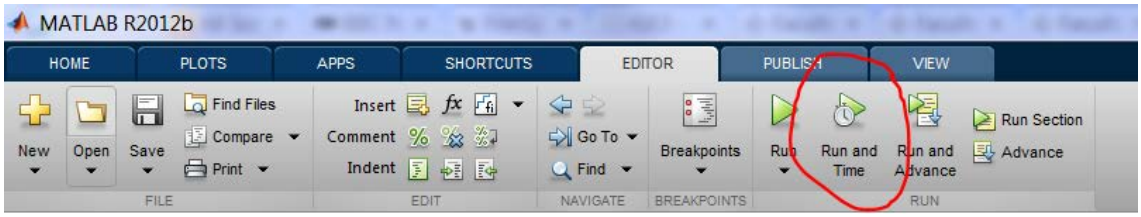
El **profiler** es una herramienta de debugging que permite conocer el tiempo de ejecución que emplea Matlab en cada una de las instrucciones que se utilizan en un script o una función.

Al llamar al comando `profile` y después ejecutar la función Matlab devuelve un informe detallado indicando el tiempo empleado por cada función y por las funciones internas a las que llama dicha función o script.

Para llamar al comando `profile` hay dos opciones, llamarlo desde la línea de comandos:

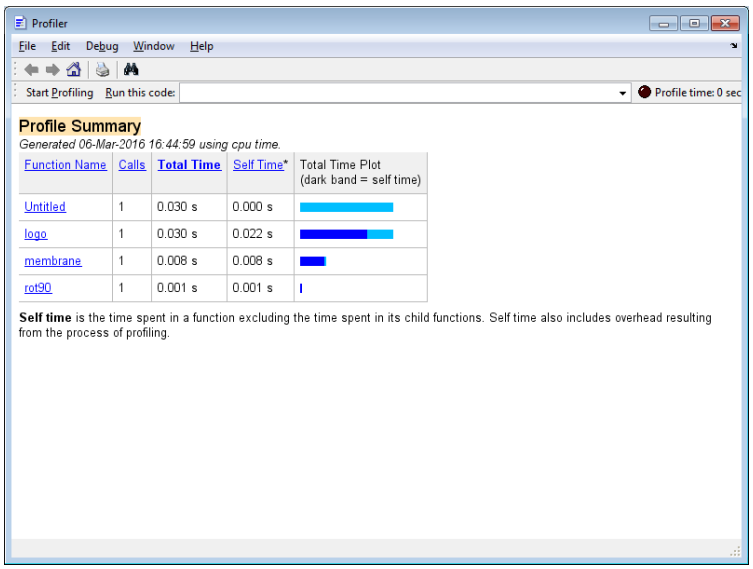
```
>> profile on;
>> función a evaluar
>> profile report
>> profile off;
```

La segunda opción es desde la ventana de ejecución del script es llamar a la función "**Run and time**" ya que esta función solo evalúa el tiempo de ejecución del script que ejecuta, mientras que desde la línea de comandos evalúa más funciones que Matlab ejecuta y que no tienen que ver necesariamente con el código evaluado.



Ejemplo

Evaluación con profiler la función log de Matlab que devuelve una imagen con el logo.



El informe nos muestra una gráfica con el tiempo empleado por cada función indicando si se debe a funciones internas o a la propia, el número de llamadas que se realizan y el tiempo empleado en segundos.

Analizando este informe que devuelve automáticamente Matlab se puede determinar que partes del código son las más costosas y dedicar el tiempo de depuración del código a elementos concretos

Problemas

1. Manejo de strings (I)

Escribir una función que convierta un número romano a su equivalente decimal. Los valores de los números romanos son:

I	1
V	5
X	10
L	50
C	100
D	500

Se utilizará como representación el sistema en el que la posición de los caracteres importa, es decir, el valor 4 sería IV.

Probar con los números LVIII y CDXLIV

2. Manejo de strings (II)

Escribir una función que reciba una frase como parámetro y sea capaz de asignar a cada palabra un valor numérico. Para ello convertir la frase en un cell array en el que cada elemento sea una palabra de la frase. Se considerará que la frase no tendrá ningún carácter especial. La función devolverá un cell en el que la primera fila se índica cada palabra y en la segunda fila el número de veces que se repite esa palabra.

Utilizar las funciones `unique, regexp` y `strcmp`

Probar los resultados con las frases

'Primera frase de prueba de este ejercicio'

'Otra frase para la prueba de la frase de la practica'

3. Optimización de código

Implementar una función que calcule los cuadrados de los 1000 y 10000 primeros números naturales de 3 formas distintas:

- Utilizando un bucle for sin declarar las variables.
- Utilizando un bucle for declarando previamente la variable.
- Calculando el cuadrado del vector correspondiente (operación componente a componente).

Calcular, con la ayuda de la instrucción de Matlab *cputime* el tiempo que se tarda en realizar el cálculo con cada uno de los tres métodos. La función a implementar debe tener como variable de entrada el número N, y como salida el tiempo que se tarda con cada uno de los tres métodos (t1, t2, t3).

4. Estructuras

Escribir un script en el cual se van a crear dos estructuras cuya composición es la siguiente.

1.- Nombre (string)

2.- Tiempos (real array)

3.- Calidad (boolean array).

Esta estructura almacenará datos del rendimiento de trabajadores en un proceso industrial y utilizaremos como ejemplo los siguientes datos:

```
Bernard, [25.8, 34.6, 22.9, 33.3], [true, true,
false, true].
Joe, [18.7, 19.9, 23.4, 18.0, 18.7, 20.0]; [false,
true, true, false, true, false];
```

Dentro de esa función se calculará el tiempo medio que tarda en fabricar un trabajador una pieza que pasa el control de calidad. Este valor debe guardarse en un nuevo campo de la estructura llamado Tmean.

Una vez que se ha creado este campo se han de convertir a cell y fusionarlas en una única variable en la que las columnas son los campos y las filas los datos. Sobre esta variable de tipo cell hay que calcular el tiempo medio en fabricar una pieza que no ha pasado el control de calidad y almacenarla en el cell.

5. Profiler

Ejecutar las siguientes líneas de código:

```
fileData = rand(100000,100);
save('data.mat', 'fileData');
```

En la misma carpeta crear la siguiente función

```
function perfTest
    for iter =1:100
        newData = subFunc(iter);
        result(iter) = max(newData)+rand(1);
    end
    disp(max(result))
end

function result = subFunc(iteration)
    file = load('data.mat');
    result = sin(file.fileData(:,iteration));
end
```

Esta función lee el fichero anterior creado anterior, calcula el seno de cada valor, devuelve el máximo de cada fila y le añade a ese máximo un valor aleatorio.

Utilizando el **profile** de Matlab optimizar lo máximo posible la función (si es necesario eliminar la subfunción declarada.