Introducción al Octave

Alberto F. Hamilton Castro
Dpto. de Ingeniería de Sistemas y Automáticas y Arquitectura y Tecnología de Computadores
Universidad de La Laguna

30 de abril de 2013

1. Licencia

Índice

1. Licencia **1**

2. Introducción **3**

2.1. Instalación . . . 3

3. Tipos de datos **3**

3.1. Números . . . 3

3.2. Matrices . . . 4

3.2.1. Clasificación . . . 4

3.3. Matrices *string* . . . 5

4. Entorno de trabajo **5**

4.1. Sentencias . . . 5

4.2. Espacio de trabajo . . . 5

5. Definición de matrices **6**

5.1. Definición explícita . . . 6

5.2. Rangos . . . 6

5.3. Funciones que generan matrices . . . 7

5.4. Indexación . . . 8

6. Operaciones sobre matrices **9**
6.1. Aritméticas . . . 9
6.1.1. Monarias . . . 9
6.1.2. Binarias . . . 10
6.2. De relación . . . 10
6.3. Lógicas . . . 11

7. Funciones **11**
7.1. Funciones matemáticas . . . 11
7.1.1. Matriciales . . . 11
7.1.2. Elemento-a-elemento . . . 12
7.2. Funciones de reorganización de matrices . . . 13
7.3. Funciones de análisis de datos . . . 13
7.4. Funciones de comprobación de condiciones . . . 14

8. Polinomios **14**

9. Gráficos **14**
9.1. Gráficas en dos dimensiones . . . 15
9.2. Comandos para el control de la gráfica . . . 16
9.3. Etiquetas en la gráfica . . . 16
9.4. Gráficas tridimensionales . . . 17
9.5. Múltiples gráficas . . . 17
9.6. Múltiples ventanas . . . 17

10. Ficheros de comandos **17**
10.1. Ficheros de función . . . 17
10.2. Ficheros de *script* . . . 17

11. Otros Comandos de interés **18**
11.1. Manejo de identificadores . . . 18
11.2. Generales del entorno . . . 18
11.3. Manejo del directorio actual . . . 19
11.4. Control del tiempo . . . 19

12. Comandos entrada/salida **19**
12.1. Por terminal . . . 19
12.2. Por fichero . . . 20
12.3. Entrada y salida tipo C . . . 20

13.Control de flujo **20**
13.1. Estructura if . . . 20
13.2. Estructura while . . . 21
13.3. Operadores lógicos de circuito-corto . . . 21
13.4. Estructura for . . . 21
13.5. Sentencia break . . . 22
13.6. Sentencia continue . . . 22
13.7. Definición de Funciones . . . 22

14.Estructuras **23**
14.1. Funciones propias de las estructuras . . . 23
14.2. Recorrido de los campos de una estructura . . . 24

15.Funciones sobre *string* **24**
15.1. Funciones que crean *string* . . . 24
15.2. Funciones de búsqueda y reemplazo . . . 24
15.3. Funciones de conversión de *string* . . . 24
15.4. Funciones lógicas . . . 25

2. Introducción

Octave es un paquete que permite la programación en alto nivel para el cálculo numérico. Tiene una sintaxis similar al paquete comercial MATLAB, al cual es prácticamente compatible hasta las versiones 4.

Octave fue pensado originariamente para ser un software de acompañamiento de un libro de texto sobre reactores químicos escrito por James B. Rawlings de la Universidad de Wisconsin-Madison y John G. Ekerdt de la Universidad de Texas.

Se caracteriza porque el tipo de datos básico es la matriz matemática (de 2 dimensiones), para las cuales tiene implementadas gran cantidad de operaciones. También puede manejar cadena de caracteres (*string*) y otros contenedores de datos más complejos como estructuras, arrays de celdas o listas.

Se utilizan identificadores de cualquier longitud para nombrar las variables y funciones. En estos se distinguen las mayúsculas de las minúsculas, es decir, en identificador `Variable` es distinto de `variable`.

Debido a la facilidad para la creación de funciones con número variable de parámetros de entrada y salida, ha podido ampliarse con conjuntos de funciones (*toolbox*) para abordar numerosos problemas del campo de las ciencias y la ingeniería: calculo numérico, estadística, procesamiento de señales, control de sistemas, etc.

Posee capacidades para realizar gráficas bidimensionales bastante completas gracias a la utilización de la aplicación Gnuplot. También son posibles las gráficas tridimensionales algo más sencillas.

Al ser un software libre, acogido a la GNU GPL (licencia publica general de GNU), los autores dan libertad a cualquier usuario para utilizar, compartir, mejorar y redistribuir (con o sin modificaciones) la aplicación. Su código fuente está disponible.

El documento actual se refiere al la versión 3.6 de Octave, aunque la mayoría del contenido es también válido para las versiones anteriores.

2.1. Instalación

La página principal de la aplicación es www.octave.org desde la cual se puede descargar la última versión para distintos sistemas operativos. Además, para la mayoría de distribuciones del sistema operativo GNU/Linux, está preparada para su instalación directa a través del sistema de gestión de paquetes correspondientes, quedando completamente integrada en el sistema.

3. Entorno de trabajo

3.1. Sentencias

Cuando se arranca la aplicación aparecerá una línea de entrada (*prompt*), típicamente `octave:1>`, donde se podrán teclear y editar las expresiones a evaluar. Al pulsar el salto de línea, y si la expresión está completa, será interpretada y ejecutada por la aplicación:

- Si la sentencia tiene errores se indicará con un mensaje.
- Si la sentencia está incompleta (paréntesis, corchetes, comillas, etc. no cerrados) aparecerá un *prompt* reducido, típicamente `>`. Se puede seguir escribiendo el comando en varias líneas. Cuando, al pulsar el salto de línea, la sentencia está completa se evaluará dando error o una sentencia correcta. Si no se desea continuar con la entrada de la sentencia, la combinación de teclas `<control>+c` terminará la entrada.
- Si la sentencia es correcta se evaluará. Si no se realiza asignación a una variables, el resultado se almacenará en la variable `ans`.

  El resultado se mostrará por pantalla. Si este ocupa más líneas de las disponibles en el terminal se utilizará un paginador, que nos permitirá avanzar y retroceder por la información (normalmente con las flechas de cursor). Con la tecla *q* se saldrá del paginador.

  Si la sentencia termina con un punto y coma (`;`) no se mostrará el resultado por pantalla, aunque se seguirá almacenando en las variables asignadas, o en `ans`.

En la mayoría de los entornos es posible recuperar y editar las sentencias anteriores utilizando las flechas de cursor del teclado.

Las funciones `clc()` o `home()` borran la pantalla y sitúan el cursor al comienzo del terminal.

Cuando se necesiten varias sentencias para obtener el resultado buscado lo más conveniente es utilizar ficheros de comandos (véase sección 10).

3.2. Espacio de trabajo

Cuando se ejecuta una sentencia de asignación (*identificador* = *expresión*) el identificador representa una variable que recoge el resultado de la expresión.

Los identificadores pueden estar compuestos por letras, dígitos (del `0` al `9`) y subrayados (`_`) pero no pueden comenzar por números. Pueden tener cualquier longitud, aunque no se recomienda que sean mayores de 30 caracteres.

Una variable, una vez definida, se pueden utilizar en cualquier expresión y permanece en el espacio de trabajo (*workspace*) mientras no sea borrada ni se salga del Octave.

Algunos comandos interesantes son:

`who` muestra las variables definidas en el espacio de trabajo.

`clear` borra las variables selecionadas (todas si no se especifica ninguna).

`quit` sale del Octave.

Más información sobre éste y otros comandos para manejar las variables en la sección 11.1. Es posible salvar y cargar variables en ficheros, como se indica en la sección 12.2.

A diferencia de otros lenguajes de programación, NO es necesario declarar el tipo ni tamaño de las variables y estas se pueden redimensionar (cambiar su tamaño) dinámicamente.

4. Tipos de datos

4.1. Números

El octave maneja por defecto los números como complejos en coma flotante según el estándar IEEE. Para indicar la coma decimal se utiliza en punto (`.`) y para el exponente se utiliza la letra `e` como ocurre en la mayoría de lenguajes informáticos.

> Así para escribir el número $2,84 \times 10^{-15}$ pondremos `2.84e-15`

Para indicar la parte compleja, se añade junto al número (sin espacios) la letra *i* ó *j*.

> Para representar el número complejo con parte real 4 e imaginaria 2,5 escribiremos `4+2.5i` o `4+2.5j`.

Si se desea representar la unidad imaginaria no debe usarse las letras *i* ó *j* en solitario, ya que, si está definida la variable correspondiente, se utilizaría el valor de la variable sin ningún tipo de advertencia. Por ese motivo debe usarse `1i` ó `1j`.

También es posible, en esta norma, la representación de los números infinitos tanto positivo como negativo (`Inf`, `+Inf`, `-Inf`), y los *No-Números* (`NaN`), resultado de la división 0/0.

Es posible indicar que un número no está definido usando la constante `NA`.

Cuando se utiliza un número en un expresión de lógica booleana, el valor 0 se considera falso y un valor distinto de 0 se considera verdadero (como en el lenguaje C).

4.2. Matrices

Como ya se comentó, el tipo básico de datos es la matriz bidimensional de números complejos en punto flotante. Cada matriz tiene un cierto número de filas (n) y un cierto número de columnas (m).

Las matrices deben ser *rectangulares*. Deben cumplir que:

- Todas las filas tengan el mismo número de columnas
- Todas las columnas tengan el mismo número de filas

es decir, no puede haber huecos en la matriz. Evidentemente, ambas condiciones son equivalentes. Si se cumple una de ellas, se cumple también la otra.

4.2.1. Clasificación

Atendiendo a las dimensiones de la matriz, tenemos los siguientes grupos:

cuadrada tiene el mismo número de filas que de columnas (n=m).

vector tiene una sola fila o columna (n=*1* ó m=*1*).

> **vector fila** vector que tiene una única fila (n=*1*).
> **vector columna** vector que tiene una única columna (m=*1*).

escalar matriz con un único elemento, es decir, una fila y una columna (n=*1* y m=*1*).

matriz vacía no tiene elementos, alguna de sus dimensiones es cero (n=*0* ó m=*0*).

La matrices pueden tener cualquier tamaño y pueden ser ampliadas o reducidas dinámicamente, es decir, no es necesario declarar el tamaño previamente, como ocurre en otros lenguajes de programación, ya que el entorno se encarga de conseguir la memoria necesaria.

Para conocer el tamaño de una matriz se dispone de las siguientes funciones:

`columns(A)` devuelve el número de columnas de la matriz.

`rows(A)` devuelve el número de filas de la matriz.

`size(A)` devuelve vector fila de dos elementos, primero el número de filas y segundo el número de columnas

`size(A,dim)` donde `dim` puede ser 1 ó 2, devuelve el número de filas si `dim` es 1, o el número de columnas si `dim` es 2.

`length(v)` devuelve la longitud del vector. Si se aplica a una matriz devuelve el valor de la dimensión más grande.

`isempty(A)` devuelve 1 si A es una matriz vacía.

5. Definición de matrices

Las matrices pueden generarse, a groso modo:

- mediante definición explícita
- como resultado de operaciones
- como resultado devuelto por una función

5.1. Definición explícita

Se realiza indicando los elemento entre corchetes (`[ ]`):

- La coma (`,`) o el espacio (uno o varios) se utiliza para separar los elementos de una fila.
- El punto y coma (`;`) o el salto de línea se utiliza para separar una fila de la siguiente.

Por lo general, es preferible la utilización de la coma y punto y coma en vez del espacio y el salto de línea, ya que estos últimos pueden llevar a errores difíciles de localizar.

Ejemplo: para representar la matriz $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ escribiremos:

```
A=[1, 2, 3; 4, 5, 6; 7, 8, 9]
```

Los elementos que forman la matriz pueden ser:

- constantes: `9 -1.4e-7 3.17-0.5e-2j`
- otras matrices, siempre y cuando de lugar a una matriz de dimensiones correctas
- expresiones que den lugar a escalares o matrices: `25.3*18.9 -0.5+3.33`

Ejemplo: para construir la matriz $B = \begin{pmatrix} 1 & 2 & 3 & 11 \\ 4 & 5 & 6 & 12 \\ 7 & 8 & 9 & 13 \end{pmatrix}$ podemos escribir:

```
B=[A, [11; 6*2; 10+3]]
```

ya que con esto añadimos a la matriz A una columna con los elementos 11, 12 y 13 para generar B.
En cambio

```
C=[A, [10, 11] ]
```

da error porque se pretendería hacer la matriz $C = \begin{pmatrix} 1 & 2 & 3 & 10 & 11 \\ 4 & 5 & 6 & & \\ 7 & 8 & 9 & & \end{pmatrix}$ que no tiene el mismo número de columnas en todas sus filas.

La matriz vacía se define con dos corchetes sin elementos: `vacia = []`

5.2. Rangos

Genera vector fila de números equiespaciados (de una sucesión aritmética). Tiene dos formas:

`inicial : incremento : límite`

`inicial : límite` Es igual que el caso anterior suponiendo `incremento=1`

El vector generado estará formado por:

`[inicial, inicial+incremento, inicial+2*incremento, inicial+3*incremento, ....]`

El último valor que aparecerá será el elemento de la sucesión PREVIO al que sea:

- mayor que `límite`, si `incremento` es positivo
- menor que `límite`, si `incremento` es negativo

Por ello, el valor `límite` aparecerá en el vector solo si pertenece a la sucesión. Es decir, puede no aparecer en el vector.

Si el rango no es realizable devolverá la matriz vacía `[]`. Un rango NO es realizable cuando:

- Siendo el `incremento` positivo, es `límite` $<$ `inicial`, ya que por mucho que incrementemos `inicial` nunca llegaremos al `límite`
- Siendo el `incremento` negativo, es `límite` $>$ `inicial`, ya que por mucho que decrementemos `inicial` nunca llegaremos al `límite`

Ejemplos:

el rango `1:2:8` es equivalente a `[1, 3, 5, 7]`
el rango `10:-3:-5` es equivalente a `[10, 7, 4, 1, -2, -5]`
el rango `1:-1:5` dará lugar a la matriz vacía `[]`
el rango `[2.2:5.9]` dará lugar al vector `[2.2, 3.2, 4.2, 5.2]`

5.3. Funciones que generan matrices

Existe una serie de funciones que se utilizan para generar matrices de ciertas características.

Como comentamos en la introducción, las funciones de Octave permiten un número variable de parámetros. La mayoría de ellas tienen un comportamiento distinto según el número y el tipo de los parámetros con que son invocadas. De esta manera con un mismo identificador (nombre de la función) se pueden conseguir resultados distintos. Veamos esto en la función `zeros`:

`zeros` devuelve una matriz con todos sus elementos a 0. Los parámetros sirven para indicar el tamaño de dicha matriz. Las posibilidades que teney no un errormos son:

- `zeros(n)` invocada con 1 parámetro que es un escalar entero positivo `n`. Devuelve una matriz cuadrada de dimensión `n` (`n` filas y `n` columnas) con todos los elementos a 0.
- `zeros(n,m)` invocada con 2 parámetro que son escalares enteros positivos `n` y `m`. Devuelve una matriz de `n` filas y `m` columnas con todos los elementos a 0.
- `zeros([n,m])` invocada con 1 parámetro que es un vector de dos enteros positivo `n` y `m`. Devuelve una matriz de `n` filas y `m` columnas con todos los elementos a 0.

Otras funciones, que tienen unas opciones de invocación idénticas a `zeros`, son :

`ones` devuelve un matriz con todos sus elementos a 1. Formas de invocación: `ones(n)`, `ones(n,m)`, `ones([n,m])`.

`eye` devuelve un matriz con todos sus elementos a 0 salvo los de la diagonal principal (primera diagonal). Si se genera una matriz cuadrada el resultado será la matriz identidad de esa dimensión. Formas de invocación: `eye(n)`, `eye(n,m)`, `eye([n,m])`.

`rand` devuelve un matriz con elementos aleatorios tomados de una distribución uniforme entre 0 y 1. Formas de invocación `rand(n)`, `rand(n,m)` y `rand([n,m])`

`randn` devuelve un matriz con elementos aleatorios tomados de una distribución gausiana de media 0 y varianza unidad 1. Formas de invocación `randn(n)`, `randn(n,m)` y `randn([n,m])`

Otras funciones interesantes de este grupo son:

`linspace(inicial,final,n)` devuelve un vector de elementos equiespaciados a partir del valor `inicial` y hasta el `final`. Habrá tantos elementos como indique `n`, o 100 si este parámetro no se especifica. A diferencia de los rangos, en que se conoce la separación de los elementos pero no su número, en este caso se conoce el número de elementos pero no su separación.

`logspace(inicial,final,n)` similar a `linspace` pero con una separación logarítmica entre los elementos del vector. En este caso `inicial` y `final` representan los exponentes de 10 para el valor inicial y final del vector. Es decir, se devuelven `n` números de una sucesión geométrica entre $10^{inicial}$ hasta 10^{final}. En el caso de que `final` valga π el valor final será π y no 10^{π}.

5.4. Matrices *string*

Un strig es una secuencia de caracteres encerrada entre comillas dobles (") o simples (’). Dará lugar a un vector fila. Es preferible utilizar las comillas dobles para evitar la confusión con el operador transposición. Además entre comillas dobles es posible incluir secuencias de escape, como las de el lenguaje de programación C, precedidas por la barra invertida (\).

Los *string* se manejan como cualquier matriz, por ejemplo se pueden concatenar para crear *string* más largos.

> Ejemplo: si ejecutamos
>
> `["El perro ","ladra mucho"]`
>
> nos devolverá el string
>
> `"El perro ladra mucho"`.

En el caso de definir una matriz con *string* en varias filas, las filas más pequeñas se rellenan automáticamente con blancos para llegar a la longitud de la más larga.

> Ejemplo: si hacemos `S=["Hola";"tu"];` tendremos que `size(S)` es `[2,4]`

Para saber si una matriz es string existe la función `ischar(S)`.

> Ejemplo: `ischar(S)` devolverá 1.

En la sección 15 se comentan varios grupos de funciones sobres string.

6. Indexación

Cuando tenemos una variable matriz, representada por su identificador, es posible especificar submatrices de la misma para operar con ellas. Esto se realiza mediante las expresiones de indexación que consisten en:

- *Identificador(indice_filas, indice_columnas)* para el caso de matrices
- *Identificador(indice_elemento)* para el caso de vectores

Los posibles tipos de índices, tanto para las filas como para las columnas, son:

- Escalares
- Dos puntos
- Vector de enteros
- Expresión lógica

Estas variantes se pueden combinar libremente, es decir, en una indexación se puede utilizar un vector de enteros para seleccionar las filas y un escalar para seleccionar las columnas.

Escalar Entero que selecciona sólo la fila/columna indicada. Las filas/columnas se numeran **comenzando por le número 1.** Si entero es menor que 1 o mayor que el número de filas/columnas de la matriz se producirá un error.

La expresión `A(3,1)` representa al elemento de la tercera fila, primera columna. En la matriz `A` anterior su valor es 7.

La expresión `A(1,4)` devolverá un error ya que `A` sólo tiene 3 columnas.

: selecciona todas las filas/columnas

`A(2,:)` representa toda la segunda fila

`A(:,1)` representa toda la primera columna

`A(:,:)` representa a todas las filas y columnas, es decir, a toda la matriz

vector de enteros se selecciona cada una de las filas/columnas indicadas por los elementos del vector y en el orden indicado por este. Los números deben ser enteros entre 1 y el número de filas/columnas.

`A(2, [3,1])` representa la fila 2 columnas 3 y 1. En nuestro ejemplo devolverá `[6,4]`

`A( [2,1,2] , 1)` selecciona el primer elemento de la fila 2ª la 1ª y nuevamente la 2ª. En nuestro ejemplo devolverá el vector columna `[4; 1; 4]`

`A(3:-1:1 , :)` devuelve la matriz con el orden de las filas invertido. En nuestro ejemplo `[7,8,9;4,5,6;1,2,3]`

`A([1,7] , :)` devuelve error ya que `A` sólo tiene 3 filas.

`A(: , [1.1,1.5])` devuelve error ya que los índices para las columnas no son enteros.

lógico cuando el índice es un vector de ceros y unos resultante de una operación de relación o lógica, se seleccionan únicamente la fila/columna donde la relación se cumple (tienen el elemento correspondiente a 1).

`A(A(:,1)>5,3)` se selecciona el tercer elemento de las filas para las cuales su primer elemento es mayor que 5. En nuestro caso será el 9, ya que la relación `A(:,1)>5` se cumple sólo para el 3er elemento.

En el caso de vectores, se puede usar un único índice que se aplicará automáticamente a la dimensión más larga (distinta de 1).

Para el vector columna `vc=[1;7;-5;0.5]` se tiene que `vc([1,3,2])` devolverá en vector columna `[1;-5;7]`

Para el vector fila `vf=[3,-10,14,7.2]` se tiene que `vf(2:4)` devolverá el vector fila `[-10,14,7.2]`

Si a una matriz bidimensional se le aplica un único índice, la matriz se convierte en vector columna concatenando un columna debajo de otra antes de aplicar el índice (indexación tipo Fortran).

`A([1,3,8])` devolverá el vector `[1,7,6]`

Estas indexaciones se pueden utilizar también al lado izquierdo de una asignación para modificar parte de una matriz.

`A([1,3],[1,2])=zeros(2,2)` hará que la matriz `A` quede $\begin{pmatrix} 0 & 0 & 3 \\ 4 & 5 & 6 \\ 0 & 0 & 9 \end{pmatrix}$

Estas expresiones de indexación, sobretodo cuando se utilizan rangos y expresiones de relación, son uno de los elementos más potentes de este lenguaje. Permiten realizar operaciones muy complejas de una manera muy compacta. En otros lenguajes de programación, estas operaciones supondrían la realización de bucles iterativos y expresiones condicionales (bucles *for* y condiciones *if*).

Ejemplos:

> Dado el vector `datos` con gran número de valores, si queremos quedarnos sólo con uno de cada 5 valores bastará con la siguiente expresión `datos(1:5:length(datos))`.
>
> Sea la matriz `Tiempo_T` de dos columnas en la que la primera están los dato de tiempo y la segunda de temperatura. Para obtener los instantes de tiempo en que la temperatura superó un determinado valor `Tmin`, bastará con la expresión `Tiempo_T( Tiempo_T(:,2)>Tmin , 1)` .

7. Operaciones sobre matrices

Los operadores y funciones definidas se pueden clasificar en dos tipos:

matriciales operan sobre la matriz como un todo, según está definido matemáticamente. Las dimensiones de los operandos y el resultado están establecidas por las leyes matemáticas.

elemento-a-elemento operan sobre cada uno de los elementos de las matrices intervinientes de manera independiente. Las matrices se utilizan simplemente como un contenedor de datos. Las matrices operadas han de tener las mismas dimensiones que, a su vez, serán las dimensiones del resultado.

Cuando uno de los operandos es un escalar la operación se realiza entre el escalar y cada elemento del otro operando (matriz). Es decir, es como si el escalar se convirtiera en una matriz de la misma dimensión que el otro operando, con todos sus elementos con el mismo valor que el escalar y luego se hiciera la operación elemento-a-elemento.

7.1. Aritméticas

7.1.1. Monarias

X.' trasposición, intercambia filas por columnas.

X' trasposición compleja conjugada, cambia filas por columnas y cambia el signo de la parte imaginaria. Si todos los elementos son reales, es equivalente a la trasposición.

-X Cambio de signo de todos los elementos de la matriz.

7.1.2. Binarias

X+Y Suma de matrices, las dimensiones deben coincidir.

X.+Y Suma elemento-a-elemento, equivalente a +

X-Y Resta de matrices, las dimensiones deben coincidir.

X.-Y Resta elemento-a-elemento, equivalente a -

X*Y Multiplicación de matrices, las dimensiones internas (nº de columnas de X y filas de Y) deben coincidir.

X.*Y Multiplicación elemento-a-elemento, las dimensiones deben coincidir.

X/Y División de matrices, es decir, multiplicación por la inversa por la derecha: $X \cdot Y^{-1}$

X./Y División de cada elemento de **X** por el correspondiente elemento de **Y**

X\Y División de matrices por la izquierda, es decir, multiplicación por la inversa por la izquierda $X^{-1} \cdot Y$

X.\Y División de cada elemento de **Y** por el correspondiente elemento de **X**

X^P Potencia de matrices, sólo definido matemáticamente en los casos en que **X** ó **P** sea un escalar. Si **P** es entero y **X** matriz cuadrada es equivalente a la multiplicación de **X** por si mismo **P** veces.

XP** ídem que **X^P**

X.^P Potencia elemento-a-elemento, las dimensiones deben coincidir.

X.P** ídem que **X.^P**

7.2. De relación

Se realizan la comparación de los valores elemento-a-elemento. Los operandos han de tener las mismas dimensiones y se devuelve una matriz de las mismas dimensiones con los elementos a 1 donde se cumple la relación y a 0 donde no se cumple.

El resultado de estos operadores se pueden utilizar como índices de filas o columnas de una matriz.

Los operadores disponibles son:

X<Y Cierto si el valor de **X** es menor que el de **Y**

X<=Y Cierto si el valor de **X** es menor o igual que el de **Y**

X>Y Cierto si el valor de **X** es mayor que el de **Y**

X>=Y Cierto si el valor de **X** es mayor o igual que el de **Y**

X==Y Cierto si el valor de **X** es igual que el de **Y**

X!=Y Cierto si el valor de **X** es distinto que el de **Y**

X~=Y equivalente a **X!=Y**

X<>Y equivalente a **X!=Y**

7.3. Lógicas

Realizan la operación booleana elemento-a-elemento suponiendo 0 como falso y distinto de 0 como verdad. El resultado devuelto es 1 para verdad y 0 para falso.

El resultado de estos operadores se pueden utilizar como índices de filas o columnas de una matriz.

Los operadores son:

X&Y y-lógico, será verdad sólo si ambos elementos son verdad.

X|Y o-lógico, será falso sólo si ambos elementos son falsos.

!X no-lógico, será falso si el elemento es verdad y será verdad si el elemento es falso.

~X equivalente a **!X**

8. Funciones

Se invocan indicando su nombre y argumentos de entrada entre paréntesis. Las funciones pueden devolver varios resultados en una lista de salida. La forma más general de invocación de una función es la siguiente:

[argSal1,argSal2,...]=nombre(argEnt1,argEnt2,....)

Los argumentos de entrada siempre se pasan por valor, es decir, se evalúa la expresión correspondiente y el resultado se pasa a la función. Por ello pueden ser constantes, identificadores de variables existentes o expresiones. No es posible modificar dentro de la función el valor de una variable que se coloque como argumento de entrada.

En cambio, en la lista de argumentos de salida deben figurar únicamente identificadores de variables (o variables indexadas), a los cuales se les asignará el valor del resultado. Si no se indica lista de salida el resultado devuelto por la función es el correspondiente al primer argumento de salida. En este caso la invocación a la función se puede usar como parte de una expresión:

```
M=[ones(10,15), zeros(10,20)];
```

Una función no puede modificar ninguna variable definida en el espacio de trabajo, salvo que aparezca en su lista de salida. Es decir, las funciones no tienen ningún efecto lateral.

Muchas funciones tienen un comportamiento distinto dependiendo del número de argumentos de entrada o de salida con el que sean invocadas. Incluso el comportamiento puede variar con las dimensiones de los argumentos de entrada (diferente si es escalar, vector, matriz cuadrada, etc.). Para conocer los detalles del funcionamiento de una función se debe hacer uso del comando `help` *nombre*.

8.1. Funciones matemáticas

8.1.1. Matriciales

Son funciones que están definidas matemáticamente para una matriz como un todo.

8.1.1.1. Trascendentes

expm(A) Exponencial de una matriz cuadrada, se calcula por desarrollo de la serie de Taylor.

logm(A) Logaritmo neperiano de una matriz cuadrada.

sqrtm(A) Raíz cuadrada de una matriz cuadrada.

8.1.1.2. Generales

det(A) determinante de una matriz cuadrada.

trace(A) traza de la matriz: suma de los elementos de su diagonal principal.

inv(A) inversa de una matriz cuadrada no singular ($|A| \neq 0$).

inverse(A) equivalente a **inv(A)**

landa=eig(A) autovalores de la matriz cuadrada.

[v,landa]=eig(A) devuelve los autovectores y autovalores de la matriz cuadrada.

rank(A) rango de la matriz.

8.1.2. Elemento-a-elemento

Funciones que se aplican a cada elemento de la matriz y el resultado se devuelve en una matriz de las mismas dimensiones. En este caso las matrices se utilizan simplemente como un contenedor de datos. Están definidas todas las habituales en cualquier lenguaje de programación o librería matemática.

8.1.2.1. Aritmética compleja

abs(X) módulo del número complejo, si son reales es equivalente al valor absoluto.

arg(X) angle(X) argumento del número complejo.

conj(X) complejo conjugado, parte imaginaria cambiada de signo.

imag(X) parte imaginaria como número real.

real(X) sólo parte real de los elementos.

8.1.2.2. Utilitarias

ceil(X) floor(X) fix(X) round(X) redondeos.

rem(X,Y) resto de la división.

sign(X) signo de los elementos: 1 si positivo, -1 si negativo, 0 si 0.

8.1.2.3. Trascendentes

exp(X) log(X) log10(X) log2(X) exponencial y logaritmos.

pow2(X) para cada elemento se calcula 2^x.

sqrt(X) raíz cuadrada.

sin cos tan sec csc cot trigonométricas ordinarias.

asin acos atan asec acsc acot trigonométricas inversas.

sinh cosh tanh sech csch coth trigonométricas hiperbólicas.

asinh acosh atanh asech acsch acoth trigonométricas hiperbólicas inversas.

atan2(x,y) arcotangente de y/x con el argumento correcto entre $-\pi$ y π.

8.2. Funciones de reorganización de matrices

transpose(A) equivalente a `A.'`

fliplr(A) (vuelta izquierda-derecha) devuelve la matriz con las columnas en el orden inverso.

flipud(A) (vuelta arriba-abajo) devuelve la matriz con las filas en el orden inverso.

rot90(A,n) (rotación 90^o) devuelve la matriz rotada 90^o en el sentido de las agujas del reloj el número de veces indicado por **n**. Si **n** se omite se supone 1.

reshape(A,n,m) convierte la matriz en una de **n** filas y **m** columnas. Los elementos se toman por columnas.

vec(A) devuelve un vector columna formado por las columnas de **A** puestas una debajo de la otra. Es equivalente a `A(:)`

8.3. Funciones de análisis de datos

Si queremos almacenar, en una matriz, datos resultado de experimentos o medidas, se deben colocar los del mismo tipo en la misma columna. Por ejemplo, si obtenemos la evolución de un sistema ante una entrada, lo más conveniente es colocar en una columna los instantes de tiempo, en otra los valores de la entrada y en la tercera los valores de la salida (para cada uno de los instantes de tiempo).

$$Datos = \begin{bmatrix} t_0 & u_0 & y_0 \\ t_1 & u_1 & y_1 \\ t_2 & u_2 & y_2 \\ \vdots & \vdots & \vdots \\ t_N & u_N & y_N \end{bmatrix}$$

Existe una serie de funciones para el análisis de datos que, por este motivo, realizan la operación por columnas en el caso de que su argumento sea una matriz. En caso de vectores, realizan la operación sobre todos los elementos del vector.

Las funciones más interesantes de este grupo son:

mean(X) media de los elementos de cada columna.

median(X) mediana de los elementos de cada columna.

std(X) desviación estándar de los elementos de cada columna.

[M,i]=max(X) devuelve en **M** el máximo de cada una de las columnas de **X**. En **i** se devuelve, para cada columna, el índice de la fila en la que se encontró el máximo.

[m,i]=min(X) devuelve en **m** el mínimo de cada una de las columnas de **X**. En **i** se devuelve, para cada columna, el índice de la fila en la que se encontró el mínimo.

[S,i]=sort(X) devuelve en **S** los elementos de **X** ordenados de manera creciente por columnas. En **i** se devuelve los índices, de la matriz original, que da lugar a la nueva ordenación. `S=[X(i(:,1),1) X(i(:,2),2), ... ]`

sum(X) suma de los elementos de cada columna.

prod(X) producto de los elementos de cada columna.

sumsq(X) suma del cuadrado de los elementos de cada columna.

cumsum(X) suma acumulada de los elementos de cada columna.

cumprod(X) producto acumulado de los elementos de cada columna.

cov(X,Y) calcula la matriz de covarianza entre dos observaciones.

corrcoef(X,Y) calcula la matriz de correlación entre dos observaciones

8.4. Funciones de comprobación de condiciones

any(X) realiza la o-lógica sobre los elementos de cada columna de la matriz y devuelve vector fila.

all(X) realiza la y-lógica por columnas de la matriz.

isinf(X) matriz lógica con 1 donde existan Inf.

isnan(X) matriz lógica con 1 donde existan NaN (*not a number*).

finite(X) matriz lógica con 1 donde existan elementos no infinitos ni NaN.

isna(X) matriz lógica con 1 donde existan números no definidos (NA).

[i,j,v]=find(X) devuelve los índices de fila (**i**) y columna (**j**) de los elementos que son verdad ($\neq 0$). En **v** el valor de dichos elementos.

find(X) es equivalente a `find(vec(X))`.

9. Polinomios

En octave los polinomios se representan mediante un vector con los coeficientes del polinomio en orden descendente. Dado el vector `p=[c1,c2,...,cN]`, éste representa al polinomio $p(x) = c_1x^{N-1} + c_2x^{N-2} + ... + c_{N-1}x + c_N$

Las funciones que realizan operaciones en polinomios son:

polyout(p,x) muestra por pantalla una representación del polinomio **p**, usando como carácter para la variable independiente el indicado en string **x**, que por defecto vale "s".

roots(p) obtiene las raíces del polinomio.

poly(A) siendo **A** matriz cuadrada, devuelve el polinomio característico.

poly(r) siendo **r** un vector, devuelve el polinomio cuyas raíces son los elementos de **r**.

polyval(p,X) evalúa el polinomio **p** para todos los elementos de **X**.

polyvalm(p,A) para **A** matriz cuadrada, evalúa el polinomio **p** en sentido matricial.

conv(p,q) devuelve el producto de dos polinomios.

[c,r]=deconv(x,y) devuelve el cociente y el resto de la división del polinomio **x** entre el **y**.

polyderiv(p) devuelve el polinomio derivada.

polyfit(x,y,n) devuelve el polinomio de orden **n** que mejor se ajusta, en mínimos cuadrados, a los puntos formados por **(x,y)**.

polyreduce(p) reduce, si es posible, el vector del polinomio **p**, quitando los ceros a la izquierda.

[r,p,k,e]=residue(num,den) calcula la descomposición en fracciones simples del cociente del polinomio **num** entre el **den**. **r** son los residuos (numeradores), **p** son los polos (denominadores), **k** es el cociente (si grado del numerador mayor que el del denominador), y **e** son los exponentes para cada denominador. Es decir:

$$\frac{num(s)}{den(s)} = k(s) + \frac{r(1)}{(s-p(1))^{e(1)}} + \frac{r(2)}{(s-p(2))^{e(2)}} + ...$$

10. Gráficos

Para la realización de los gráficos el Octave invoca a el paquete Gnuplot.

10.1. Gráficas en dos dimensiones

Los comandos de Octave son:

plot función que produce gráficas bidimensionales con escalas lineales en ambos ejes. Existen varias formas de invocar a plot:

- **plot(x,y)** donde tanto **x** como **y** son vectores, se representarán los elementos de **y** frente a los de **x**, es decir, se representarán los puntos *(x(1),y(1)), (x(2),y(2)), ...* Las longitudes de **x** e **y** han de coincidir.
- **plot(y)** donde **y** es un vector, representa las componentes del vector frente a sus índices. Equivale a `plot([1:length(y)],y)`.
- **plot(x,Y)** donde **x** es vector e **Y** matriz, se representan las columnas de **Y** frente a los valores de **x**. En caso de que la longitud de las columnas no coincida con la longitud de **x**, se intentará con las filas.
- **plot(X,y)** donde **X** es matriz e **y** vector, se representará el vector **y** frente a las columnas de **X**. En caso que la longitud de las columnas no coincida con la longitud de **y**, se intentará por las filas.
- **plot(X,Y)** donde tanto **X** como **Y** son matrices, las columnas de **Y** se representan frente a la columna correspondiente de **X**, por ello ambas dimensiones han de coincidir.

plot(x1,y1) donde **x1** e **y1** son escalares, se representará un único punto.

plot(X,Y,formato) donde **X** e **Y** son de cualquiera de las formas anteriores, **formato** especifica la forma en que se representará la línea.

plot(X1,Y1,fm1,X2,Y2,fm2,....) es posible combinar varias llamadas a plot colocando los argumentos necesarios unos a continuación de otros. El argumento de formato se puede omitir.

Los posibles formatos del comando **plot** son los siguientes[1]

'-' segmento uniendo los datos, formato por defecto.

'.' puntos pequeños en cada dato.

'@' puntos en cada dato.

'-@' líneas uniendo cada dato con punto en cada dato.

'x' aspas en cada dato.

'+' cruces en cada dato.

'o' circulo en cada dato.

'n' donde n es dígito entre 1 y 6, indica color. Algunos colores se pueden especificar por su inicial inglesa: *r*, *g*, *w*, etc.

'nm' donde **n** y **m** son dígitos de 1 a 6, **n** indica color y **m** estilo de punto.

La relación entre el número, el color y el estilo de línea es la siguiente:

Número	Color	Letra color	Tipo de punto	Símbolo punto
1	rojo	r	círculo	o
2	verde	g	cruces	+
3	azul	b	cuadrado	
4	magenta	m	aspa	x
5	cian	c	triangulo	
6	marrón		asterisco	*

semilogx(arg) recibe los mismos argumentos que plot pero utiliza una escala logarítmica en el eje x.

semilogy(arg) recibe los mismos argumentos que plot pero utiliza una escala logarítmica en el eje y.

loglog(arg) recibe los mismos argumentos que plot pero utiliza una escala logarítmica en los eje x e y.

polar(angulo,modulo) hace trazo bidimensional utilizando el ángulo y la distancia al origen para situar los puntos.

errorbar(arg) se le ha de pasar junto a los puntos los errores cometidos en cada dimensión (x e y) y representa las barras de error en cada dimensión. Existen también las funciones equivalentes **loglogerr**, **semilogxerr, semilogyerr.**

bar(x,y) donde **x** e **y** son vectores de las mismas dimensiones, produce el diagrama de barras de **y** frente a **x**. **x** debe tener valores en orden ascendente.

bar(y) donde **y** es vector, representa los valores de **y** frente a sus índices.

[xb,yb]=bar(x,y) no realiza la representación, sino que devuelve los vectores **xb** e **yb** que se pueden utilizar posteriormente con **plot(xb,yb)**

hist(x,y) representa el histograma con los datos suministrados.

stairs(x,y) donde **x** e **y** son vectores de datos, produce el trazo en escalera típico de la salida de un retenedor de orden 0. **x** ha de tener valores en orden ascendente.

[xb,yb]=stairs(x,y) no realiza la representación, sino que devuelve los vectores **xb** e **yb** que se pueden utilizar posteriormente con **plot(xb,yb)**.

[1] Las especificaciones de formato parecen variar mucho con la versión y el sistema operativo de ejecución.

10.2. Comandos para el control de la gráfica

axis([minx maxx miny maxy]) especifica los límites entre los que se representa la gráfica. Los rangos para el eje y se pueden omitir, es decir, se puede fijar el rango para el eje x y para los ejes x e y.

axis vuelve al estado de autoescalado.

replot actualiza la gráfica según las modificaciones que se hayan introducido posteriormente al **plot**, sobre todo en los ejes. En ciertas versiones este comando no es necesario ya que los cambios se aplican inmediatamente.

clf borra la ventana de gráficos actual .

hold on mantiene el gráfico actual en la ventana, los sucesivos se añadirán al actual.

hold off desactiva la permanencia del gráfico, los siguientes borraran la ventana. Este es el estado por defecto.

hold cambia el estado actual del **hold** entre **on** y **off**.

ishold() devuelve 1 si el gráfico está mantenido y 0 en caso contrario.

10.3. Etiquetas en la gráfica

grid coloca rejilla en la gráfica.

title(string) coloca el título a la gráfica.

xlabel(string) ylabel(string) coloca las etiquetas en los distintos ejes.

10.4. Gráficas tridimensionales

[X,Y]=meshdom(x,y) dados los vectores con los puntos a considerar en el eje x e y genera las matrices **X** e **Y** necesarias para la función mesh.

mesh(X,Y,Z) a partir de las matrices **X** e **Y** de meshdom, y la matiz de con los valores z en esos puntos, genera la gráfica tridimensional.

axis([minx maxx miny maxy minz maxz]) especifica los límites entre los que se representa la gráfica. Los rangos para el eje y y z se pueden omitir, es decir, se puede fijar el rango para el eje x, los ejes x e y, o los ejes x, y y z.

10.5. Múltiples gráficas

Es posible tener varias subgráficas en la misma ventana:

subplot(f,c,a) divide la ventana en **f** filas y **c** columnas de subgráficas y sitúa como gráfica actual la **a**-ésima.

subplot(fca) ídem que el anterior pero los tres parámetros se pasan como los tres dígitos decimales de un único parámetro (compatibilidad con MATLAB).

10.6. Múltiples ventanas

Es posible abrir varias ventanas gráficas.

figure(n) sitúa la ventana actual para los siguientes comandos como la n-ésima, abriéndola si no existe. La ventana que se abre por defecto es la número 1.

11. Ficheros de comandos

Es posible ejecutar comandos que están en un fichero. Estos ficheros son de texto normal y deben de tener la extensión *'.m'*, por lo que suelen conocer como *ficheros-m*.

En estos ficheros es posible poner comentarios que comienzan por el carácter # y continúan hasta el final de la línea. Las líneas de comentario que aparecen juntas al principio del fichero o tras la definición de una función representan la ayuda, y son presentadas si ejecutamos: `help` *nombre*

Cuando se invoca a una función o nombre, Octave la busca en el espacio de trabajo actual. Si no existe allí, busca si existe un fichero con ese nombre y la extensión *'.m'* en el directorio actual y el los directorios de una lista (`LOADPATH`). Si existe lo carga en memoria y lo ejecuta. En las siguientes invocaciones Octave sólo comprueba si el fichero ha sido actualizado, si es así lo carga nuevamente, en caso contrario utiliza la copia existente en memoria.[2]

11.1. Ficheros de función

Son ficheros que contienen únicamente la definición de una función (véase sección 13.7). Se deben llamar con el mismo nombre que la función.

11.2. Ficheros de *script*

Estos ficheros contiene comandos normales que se ejecutan como si fueran tecleados directamente en el entorno, es decir, trabajan sobre el espacio de trabajo global, por lo que pueden acceder a las variables existentes, modificarlas o crear nuevas.

Para invocarlo basta poner el nombre del fichero, sin la extensión *'.m'*.

Es posible definir funciones dentro de un fichero *script*, pero el primer comando ejecutable no puede ser una definición de función ya que, en caso contrario, el fichero se considerará de función.

12. Otros Comandos de interés

12.1. Manejo de identificadores

Existe un conjunto de funciones para manejar los identificadores en el espacio de trabajo, las principales son:

who ***opciones patron*** lista los identificadores existentes que encajan con el patrón, o todos si éste no se indica. El patrón puede utilizar los comodines: * ? [*lista*]. Las opciones posibles son:

- **-all** lista todos los símbolos: variables internas y funciones internas además de las funciones y variables del usuario.
- **-builtins** lista variables y funciones internas.
- **-functions** lista las funciones definidas por el usuario.
- **-variables** lista las variables definidas por el usuario.
- **-long** presenta más detalles sobre cada uno de los identificadores: tipo, tamaño, etc.

whos es equivalente a **who -long**

clear ***opciones patrón*** borra los identificadores del usuario que concuerdan con el patrón, o todos si no se especifica patrón. La única opción posible es:

- **-x** borra todos los identificadores excepto los que concuerdan con el patrón.

[2] Esto puede ocasionar problemas si se está trabajando sobre un disco en un servidor de ficheros y la hora del servidor y el puesto no están suficientemente sincronizadas.

exist("nombre") devuelve: 1 si existe una variable con **nombre**, 2 si existe fichero de función con **nombre**.m, 3 si es el nombre de un fichero '.oct' o '.mex' en el path de Octave, 5 si es el nombre de una función propia, 7 si es el nombre de un directorio, o 103 si es el nombre de una función no asociada a un fichero (introducida por línea de comandos), 0 en caso contrario.

document(*identificador*,texto) fija el texto como documentación del identificador.

12.2. Generales del entorno

quit sale del octave.

quit(estado) sale del octave devolviendo el código de estado indicado (valor entero).

exit(estado) ídem que **quit**.

help *concepto* presenta la ayuda disponible sobre el tema solicitado, función, comando, etc.

diary *opción* permite registrar los comandos y respuestas que aparecen en el terminal. Las opciones posibles son:

on comienza la grabación en el fichero *diary* del directorio actual.

off detiene la grabación

fichero comienza la grabación en el fichero indicado.

sin opción cambia el estado actual de grabación

echo *opción* controla si los comandos son presentados antes de su ejecución. Las opciones posibles son:

on presente las comandos que se ejecutan en los ficheros *script*.

off desactiva la presentación de comandos.

on all activa la presentación de comandos en *scripts* y ficheros de función.

eval(string) ejecuta el comando representado por ***string*** en el espacio de trabajo actual.

12.3. Manejo del directorio actual

El directorio actual es utilizado para la búsqueda de los ficheros-m y para salvar los ficheros de datos. Las principales funciones relacionadas con el sistema de ficheros son:

pwd() devuelve en un *string* el directorio actual

chdrir *nuevoDir* cambia el directorio actual al especificado, o al directorio raíz del usuario si no se especifica ninguno.

cd ídem que **chdir**

dir lista los ficheros del directorio actual.

ls ídem que **dir**

12.4. Control del tiempo

Las principales funciones de este grupo son:

clock() devuelve un vector de 6 componentes con el valor del año, mes, día, hora, minuto y segundo actuales.

pause(segundos) suspende la ejecución del programa el número de segundos indicado. Si no se indican segundos se espera hasta que se pulse una tecla.

sleep(segundos) suspende la ejecución del programa el número de segundos indicado.

tic() toc() permiten cronometrar la duración de una serie de comandos. **tic()** pone a cero el cronómetro mientras que **toc()** devuelve el número de segundos que han transcurrido.

13. Comandos entrada/salida

13.1. Por terminal

disp(x) presenta el contenido del parámetro pero sin indicar su nombre, a diferencia de lo que ocurre si evaluamos directamente su nombre.

format *opción* permite controlar la forma en que se realiza la presentación de los números. Las principales opciones son:

short trata de representar, en punto fijo, 5 cifras significativas en un máximo de 10 caracteres. Si no es posible conseguirlo en todos los elementos de una matriz utiliza exponente.

long trata de representar, en punto fijo, 15 cifras significativas en un máximo de 24 caracteres. Si no es posible conseguirlo en todos los elementos de una matriz utiliza exponente.

short e ídem que **format short** pero en punto flotante (presentando exponente).

long e ídem que **format long** pero en punto flotante (presentando exponente).

input(mensaje) presenta contenido del *string* mensaje y espera que se teclee en el terminal una expresión, la cual es evaluada y devuelta.

input(mensaje,"s") ídem que antes, pero lo tecleado se devuelve en un *string* y no es evaluado.

menu(titulo,opcion1,...) presenta un menú con titulo y las distintas opciones y devuelve el número de la opción elegida por el usuario.

kbhit() lee una única pulsación en el terminal y devuelve el carácter pulsado en un *string*.

13.2. Por fichero

Estas funciones permiten salvar y recuperar variables a fichero en distintos formatos:

save *opciones fichero* v1 v2 ... salva, en el *fichero*, las variables indicadas, o todas las del espacio de trabajo actual si no se indica ninguna. Las variables se pueden indicar utilizando comodines (? * [*lista*]). Las opciones sirven para indicar el formato en que se salvaran las variables:

-ascii formato de texto de Octave (formato por defecto).

-binary formato binario de Octave.

-float-binary formato binario de Octave, pero sólo en simple precisión.

-mat4-binary formato binario de MATLAB versión 4.

-mat-binary formato binario de MATLAB versión 6.

-mat7-binary formato binario de MATLAB versión 7.

load *opciones fichero* v1 v2 ... carga las variables especificadas del *fichero*, o todas si no se especifican variables. Octave detecta el formato del fichero a cargar. Por defecto Octave se niega a sobreescribir una variable existente en el espacio de trabajo. La opción más interesante es:

-force si es necesario, sobreescribe las variables en memoria.

13.3. Entrada y salida tipo C

Están disponibles un conjunto completo de funciones de acceso a ficheros tanto de texto como binarios, cuyo nombre, sintaxis y funcionalidad sigue los estándares del lenguaje de programación C. De esta manera es posible un control total sobre el manejo de ficheros.

Las más destacadas son: **printf**, **sprintf**, **fprintf**, **fscanf**, **fread**, **fwrite**. Para más información consultar el manual o la ayuda.

14. Control de flujo

Las estructuras de control de flujo permiten la ejecución condicional y la repetición de un conjunto de comandos.

Las condiciones que aparecen en estas estructuras son expresiones que, como todas las de octave, devuelven una matriz. Habitualmente estas expresiones son de relación o lógicas, pero es posible cualquier expresión.

La condición será cierta si todos los elementos de la matriz resultado son distintos de cero, es decir, se realiza la y-lógica de todos los elementos. La condición es falsa si algún elemento es cero.

14.1. Estructura if

Es similar a la que aparece en otros lenguajes de alto nivel. Se construye de la siguiente manera:

```
if condición
    sentencias-entonces
else
    sentencias-caso-contrario
endif
```

La parte `else` es opcional. La *sentencias-entonces* se ejecutan si se cumple la condición, en caso contrario se ejecutan las *sentencias-caso-contrario*.

En estas *sentencias* puede aparece, como es lógico, otras estructuras de control. También es posible concatenar condiciones de la siguiente manera:

```
if condición-1
    sentencias-entonces-1
elseif condición-2
    sentencias-entonces-2
⋮
else
    sentencias-caso-contrario
endif
```

14.2. Estructura while

Se construye de la siguiente manera:

```
while condición
    sentencias
endwhile
```

Se ejecutan las *sentencias* mientras la condición sea verdadera, esta condición se evalúa tras cada bucle. Si la condición es falsa la primera vez, las sentencias no se ejecutan ni una sola vez.

Esta estructura se debe utilizar cuando el número de iteraciones que se deben realizar dependen, de alguna manera, del resultado de las *sentencias*.

14.3. Operadores lógicos de circuito-corto

En las estructuras de control condicionales (if, while) es interesante terminar la evaluación de las expresiones de condición desde que se conozca que el resultado va a ser falso o verdad:

exp1 && exp2 y-lógica de circuito corto. Se evalúa la **exp1**, si el resultado tiene algún elemento falso (0) el operador devolverá 0 sin evaluar **exp2**. Si todos los elementos de **exp1** son verdad se pasa a evaluar **exp2**. Si el resultado de **exp2** tiene todos los elementos verdad el resultado será verdad (1) caso contrario será falso (0).

exp1 || exp2 o-lógica de circuito corto. Se evalúa la **exp1**, si el resultado tiene todos los elementos a verdad (!=0) el operador devolverá 1 sin evaluar **exp2**. Si algún elemento de **exp1** es falso se pasa a evaluar **exp2**. Si el resultado de **exp2** tiene todos los elementos a false el resultado será false (0), caso contrario será verdadero (1).

14.4. Estructura for

Se construye de la siguiente manera:

```
for var = expresión
    sentencias
endfor
```

En este caso se evalúa la *expresión* al comenzar. Las *sentencias* se ejecutan tantas veces como columnas tenga el resultado de la *expresión*, asignándose a la variable *var*, en cada iteración, una de las columnas, comenzando por la primera. La expresión **sólo se evalúa una vez**, antes de comenzar, y no después de cada iteración como en el caso de la estructura **while**.

Si queremos un bucle que se ejecute N veces y tenga una variable contador `i`, lo hacemos de la siguiente manera:

```
for i=1:N
    sentencias
endfor
```

como el rango genera un vector fila, `i` tendrá los valores 1, 2, ..., N, en las sucesivas iteraciones.

14.5. Sentencia break

Salta fuera del bucle **for** o **while** más interno que la contiene, por lo tanto, sólo debe situarse dentro de dichos bucles.

14.6. Sentencia continue

Salta inmediatamente a la siguiente iteración del bucle **for** o **while** más interno que la contiene, por lo tanto, sólo debe situarse dentro de dichos bucles. En el caso del bucle **while**, se pasa a evaluar la condición.

14.7. Definición de Funciones

Esta estructura permite definir un conjunto de sentencias que puede ser invocada posteriormente escribiendo el nombre de la función.

Se construye de la siguiente manera:

```
function [sal1, sal2,...] = nombre(ent1, ent2, ...)

    sentencias

endfunction
```

La especificación de parámetros de salida (*sal1, sal2, ...*) y entrada (*ent1, ent2, ...*) es opcional y depende del comportamiento deseado para la función.

Consideraciones importantes:

- Todas las variables y parámetros que aparecen en la función son locales a esta, es decir, existen en un espacio de trabajo distinto del existente en el entrono. Tampoco podemos acceder al espacio de trabajo del entorno.
- La única manera de pasar datos a una función es mediante los parámetros de entrada, que **se pasan por valor**, es decir, no se modifican las variables utilizadas en la invocación aunque se modifiquen los parámetros correspondientes dentro de la función. Por ello los parámetros de entrada pueden ser expresiones, cuyo resultado es el que pasa a la función.
- La única manera de recibir datos de una función es mediante los parámetros de salida. Los parámetros de salida en la invocación deben ser identificadores de matrices o expresiones de indexación.
- En la definición de la función se indican únicamente los parámetros de entrada y salida máximos que soporta la función, pero la función se puede invocar con un número menor de parámetros, incluso sin ninguno.
- Los parámetros de entrada sólo estarán definidos si en la invocación se han indicado suficientes parámetros de entrada. Intentar acceder a un parámetro de entrada que no está definido produce un error.
- Los parámetros de salida nunca están definidos inicialmente. Sólo lo estarán una vez que aparezcan a la izquierda de una asignación, a partir de ese momento se pueden utilizar normalmente.
- No hay inconveniente en asignar valores a parámetros de salida que no se han solicitado en la invocación. Dichos valores se perderán, salvo el del primer parámetro que es el valor que devuelve la función aunque no se le soliciten parámetros de salida.

Existen una serie de funciones que son de utilidad dentro de una función:

nargin devuelve el número de parámetros de entrada en la invocación actual de una función.

nargout devuelve el número de parámetros de salida en la invocación actual de una función.

return termina la ejecución de la función de manera normal, como si se hubiera llegado al final de la misma.

error(s) termina la función con error, presentando como mensaje el *string* **s**. Si éste no termina con '\n' se presenta la traza completa de las funciones que se han atravesado hasta llegar al error.

warning(s) saca el mensaje s precedido por 'warning: ' y continua la ejecución. Sirve para hacer advertencias al usuario.

keyboard(mensaje) presenta el mensaje y permite que el usuario introduzca expresiones que son evaluadas y su resultado presentado. Se sigue presentando el mensaje hasta que se de el comando **exit** o **quit**. Si se ejecuta dentro de una función, las variable manejadas son las locales, por lo que es útil para depuración.

Basándonos en las funciones **nargin** y **nargout**, y estructuras de control de flujo, podemos hacer que nuestra función se comporte de manera distinta según el número de parámetros de entra o salida con que es invocada en cada momento.

15. Estructuras

Las estructuras que son *arrays* asociativos con índices de tipo string, es decir, son variables que pueden contener matrices, u otras estructuras, identificadas por su nombre, al estilo de la estructuras de C. Son equivalentes a las matrices en cuanto a la asignación y paso como parámetros de entrada o salida a funciones.

Para crear o acceder a los campos de estas estructuras se coloca el nombre de la estructura, un punto '.' y el nombre del campo. Al igual que ocurre con las matrices, no es necesario declarar a un identificador como estructura, el hecho de aparecer a la izquierda de un punto le confiere ese tipo.

Ejemplo de definición de varios campos de una estructura:

```
estru.a=1
estru.b=[1,2,3;4,5,6]
estru.c="Hola"
```

Las estructuras se presentan indicando el nombre y contenido de sus campos entre llaves ({}). Cuando existen estructuras anidadas por defecto sólo se representa hasta el segundo nivel. Este comportamiento se puede modificar modificando la variable **struc_levels_to_print**.

15.1. Funciones propias de las estructuras

is_struct(est) devuelve 1 si su parámetro es una estructura.

struct_contains(est,nombre) devuelve 1 si la estructura **est** contiene el campo **nombre** (que ha de ser un *string*).

struct_elements(est) devuelve una matriz de *strings* con los nombres de todos los campos de la estructura.

15.2. Recorrido de los campos de una estructura

Existe un formato especial del bucle **for** que permite recorrer una estructura. Se construye de la siguiente manera:

```
for[ valor, nombre]= expresión
     sentencias
endfor
```

donde *expresión* devuelva una estructura. En este caso, en cada iteración se le asigna a la variable *valor* el contenido de uno de los campos de la estructura y en la variable *nombre* el nombre de dicho campo. Se repite para todos los campos de la estructura.

16. Funciones sobre *string*

16.1. Funciones que crean *string*

Las funciones más destacadas de este grupo son:

setstr(X) convierte una matriz numérica a una matriz de *string*, remplazando cada elemento con el carácter correspondiente según el código ASCII.

int2str(n) devuelve en un *string* la representación del entero **n**. No es muy flexible, es preferible **sprintf**.

num2str(x) devuelve en un *string* la representación del número **x**. No es muy flexible, es preferible **sprintf**.

blanks(n) Devuelve un *string* de **n** blancos.

16.2. Funciones de búsqueda y reemplazo

delblank(s) borra los espacios finales del *string*.

substr(s,inicio,largo) devuelve la subcadena de **s** desde **inicio** y longitud **largo**, es equivalente a `s(inicio:inicio+largo-1)`.

findstr(s,t,NoSolapar) devuelve un vector con todas las posiciones en que la cadena **t** aparece en la cadena **s**. Por defecto se consideran las ocurrencias solapadas, pero si se especifica el 3er. parámetro a 1 se cambia este comportamiento.

index(s,t) devuelve el índice del primer lugar donde aparece la cadena **t** en la cadena **s**, o 0 si no aparece.

rindex(s,t) devuelve el índice del último lugar donde aparece la cadena **t** en la cadena **s**, o 0 si no aparece.

split(s,t) devuelve una matriz *string* con las piezas en que se puede dividir **s** usando como separador la cadena **t**.

strrep(s,x,y) remplaza en **s** todas las ocurrencias de la cadena **x** por la **y**.

16.3. Funciones de conversión de *string*

toascii(s) devuelve en una matriz los códigos ASCII de los caracteres de **s**.

str2num(s) convierte la cadena de caracteres en el número representado.

tolower(s) pasa a minúsculas las letras de **s**.

toupper(s) pasa a mayúsculas las letras de **s**.

Existe un conjunto de funciones que sirven para realizar cambios de base numérica, representando la binaria y hexadecimal en *string*:

bin2dec(s) devuelve el número decimal equivalente al número binario representado por la cadena **s**.

hex2dec(s) devuelve el número decimal equivalente al número hexadecimal representado por la cadena **s**.

dec2bin(n) devuelve una cadena con la representación binaria del número **n**.

dec2hex(n) devuelve una cadena con la representación hexadecimal del número **n**.

16.4. Funciones lógicas

strcmp(s1,s2) compara los dos *string* y devuelve 1 si son iguales.

isstr(s) devuelve 1 si se trata de una matriz *string* y 0 en caso contrario.

Otras funciones son elemento-a-elemento y devuelven un vector lógico de la misma longitud que el *string* poniendo a 1 los elementos correspondiente a los caracteres que cumplen la condición. Las funciones son:

isalnum isalpha isascii iscntrl isdigit isgraph islower isprint ispunct isspace isupper isxdigit