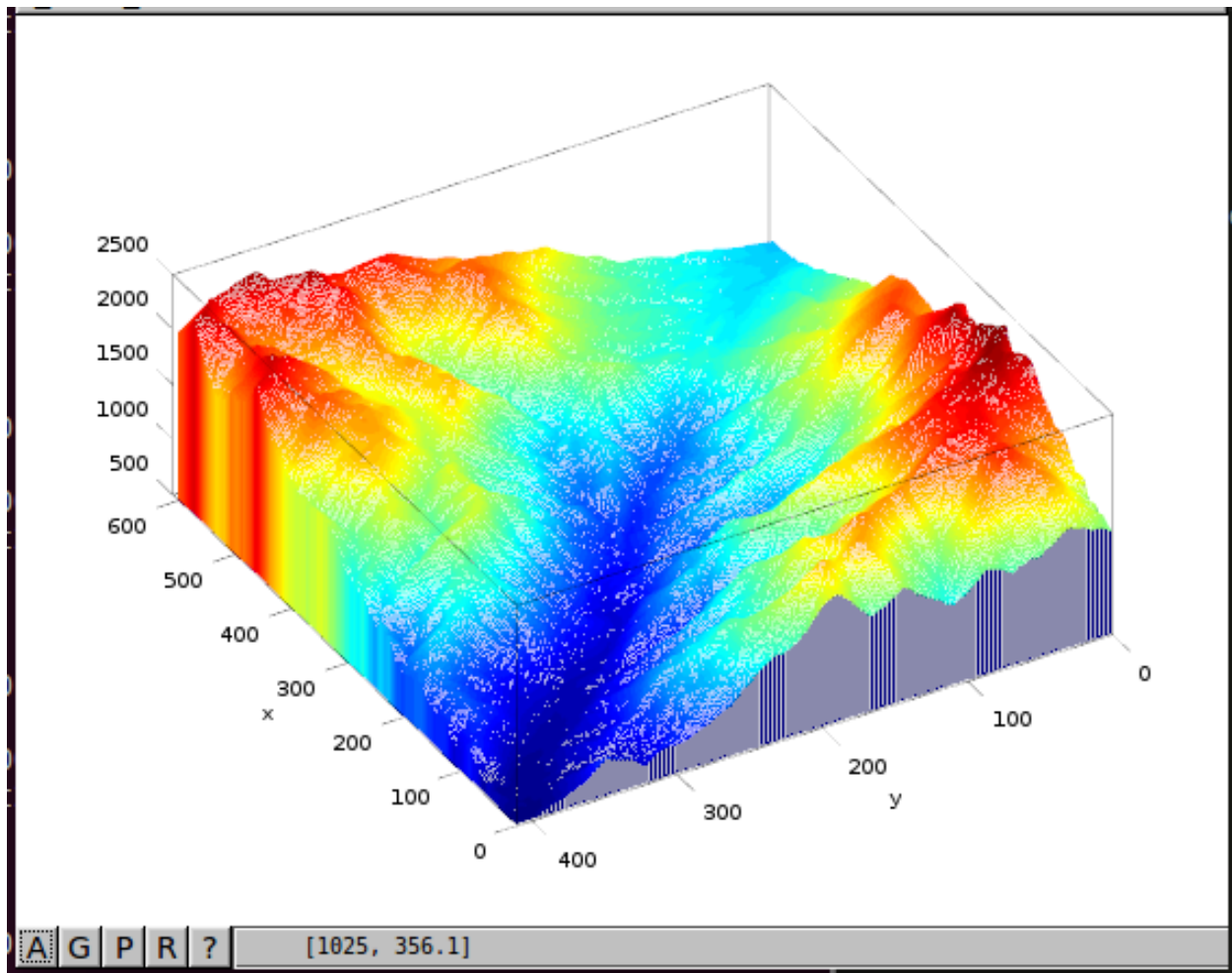

Introducción a Octave Documentation

Release 1.0

Santiago Higuera

June 05, 2016

1	Primeros pasos con Octave	3
2	Vectores y Matrices	11
3	Funciones predefinidas en Octave	21
4	Programación en Octave	25
5	Polinomios	29
6	Ejercicios resueltos	31
7	Gráficos con Octave	39
8	Lectura y escritura de ficheros de texto	51
9	Utilidades	57
10	Personalización del arranque de Octave	59
11	Tortugas y fractales	61
12	Modelos Digitales de Elevaciones	73
13	Acceso a recursos de Internet	77
14	Utilización de clases java desde Octave	79
15	Acerca de este documento	81
16	Licencia	83



Primeros pasos con Octave

Contents

- *Primeros pasos con Octave*
 - *Arranque de la consola de Octave*
 - *Cálculos elementales*
 - *Asignación de variables*
 - *Nombres de variables*
 - *Variables predefinidas en Octave*

1.1 Tipos de datos

Octave incluye un buen número de tipos de datos básicos predefinidos, entre los que podemos mencionar los escalares enteros, reales y complejos, los valores lógicos *verdadero-falso* y las cadenas de caracteres.

1.1.1 El tipo *double*

El tipo de datos por defecto en *Octave* es el tipo **double**, esto es, cuando tecleamos un número en la consola, *Octave* lo interpretará como un número **double**. Los números **double** son números en coma flotante.

La palabra *double* se refiere a **double precision**, en contraposición a la *single precision* de los números de tipo *single* que se verán más adelante.

Octave reconocerá un número como *double* al teclearlo con o sin punto decimal. También se reconocen las notaciones denominadas científicas, que se teclean finalizando el número en la letra ‘e’ y la potencia de diez deseada precedida del signo menos si ha lugar.

```
102
102.0
1.02e2
1020e-1
```

Para convertir otro tipo de dato numérico a *double*, o simplemente para crear un *double* de valor determinado se puede utilizar la función **double(x)**, que devuelve un número *double* de valor x.

Octave proporciona dos constantes que nos permiten conocer los valores máximo y mínimo que puede tener un número *double*, se trata de las constantes **realmax** y **realmin**.

```
octave:31> realmax
ans = 1.7977e+308
octave:32> realmin
ans = 2.2251e-308
octave:33>
```

1.1.2 Números complejos

Octave opera con números complejos. Para teclear un número complejo se escribira la parte real, el signo '+' y la parte imaginaria seguida sin espacios por la letra 'i'. Se puede utilizar la letra 'i' en minúscula o mayúscula y también la letra 'j', mayúscula o minúscula. Octave interpreta en todos los casos que se refiere a $\sqrt{-1}$. Veamos algún ejemplo:

```
1 + 0i
1-2I
1 + 3.0e-2j
1.0 + 2.0J
```

La función **complex()** nos devuelve un número complejo construido con la parte real e imaginaria pasadas como argumento. Ejemplos de utilización de la función **complex()**:

```
complex(2); % ans=2+0i Si solo se pasa un parámetro se toma como la parte real
complex(2,3); % ans=2+3i Si se pasan dos parámetros son la parte real y la imaginaria
```

Entre dos números complejos se pueden utilizar las operaciones suma, resta, producto y división:

```
w = 1 + i; % Nota: el punto y coma final indica a Octave que no imprima el resultado
z = 2 + 2i;
c1 = w+z % El resultado será: c1=3 + 3i
c2 = w-z % El resultado será: c2=-1 - 1i
c3 = w*z % El resultado será: c3=0 + 4i
c4 = w/z % El resultado será: c4=0.50000
```

También hay algunas funciones predefinidas para operar con números complejos. Si **z** es un número complejo, podemos utilizar las siguientes funciones:

- **real(z)** Nos devolverá la parte real del complejo **z**
- **imag(z)** Nos devolverá la parte imaginaria
- **conj(z)** Nos devolverá el número complejo conjugado de **z**
- **abs(z)** Nos devolverá el módulo de **z**
- **angle(z)** Nos devolverá el argumento del complejo **z** (en radianes)

1.1.3 Números naturales y números enteros

Octave ofrece varios tipos de datos para trabajar con números enteros, **integer (int)**, y con números naturales, **unsigned integer (uint)**.

Cuando se utiliza un solo byte para el almacenamiento, se tienen los tipo de datos **int8** y **uint8***, según se refiera a enteros positivos y negativos o a enteros solo positivos. En el primer caso los valores serán entre -128 y 127. En el caso de **uint8** los valores podrán estar comprendidos entre 0 y 255.

Para definir un número entero de un byte hay que utilizar las funciones **int8()** o **uint8()**, como en el siguiente ejemplo:

```
n1 = int8(-100)
% El resultado será: n1=-100
n2 = uint8(25)
% El resultado será: n2=25
```

De manera análoga al caso de los enteros de un byte de almacenamiento, *Octave* ofrece variables y funciones de creación para variables enteras, con o sin signo, de 2 bytes, 4 bytes y 8 bytes. El listado completo de tipos de datos enteros y sus correspondientes funciones de creación es:

- **int8, int8()**
- **int16, int16()**
- **int32, int32()**
- **int64, int64()**
- **uint8, uint8()**
- **uint16, uint16()**
- **uint32, uint32()**
- **uint64, uint64()**

Podemos conocer los valores máximo o mínimo de cualquiera de los tipos de datos anteriores a través de las funciones **intmax()** e **intmin()**. Cuando se invocan las funciones sin argumentos nos devuelven los valores correspondientes al tipo **int32**. Cuando se les pasa como argumento una cadena de texto con el nombre del tipo de dato, nos devolverán el valor máximo o mínimo del tipo de datos solicitado. Por ejemplo:

```
octave:39> intmax
ans = 2147483647
octave:40> intmax('int32')
ans = 2147483647
octave:41> intmin
ans = -2147483648
octave:42> intmin('int32')
ans = -2147483648
octave:43> intmax('uint8')
ans = 255
octave:44> intmax('uint16')
ans = 65535
octave:45> intmin('uint16')
ans = 0
octave:46> intmax('int16')
ans = 32767
octave:47>
```

Warning: Los tipos de datos enteros, a excepción de los tipos de 8 bytes, admiten las operaciones **suma, resta, producto, división y potenciación**, obteniendo como resultado enteros del mismo tipo que los operados. Hay que tener cuidado con el tipo de datos del resultado cuando aplicamos estas operaciones a datos de distinto tipo.

1.1.4 Valores logical

Un valor de tipo **logical** solo puede tener dos valores: **verdadero (true)** o **falso (false)**. *Octave* nos presentará un `'0'` para el valor **false** y un `'1'` para el valor **true**.

Para crear una variable de tipo *logical* se utiliza la función **logical()**, que recibe un número como argumento. Si el número que se pasa como argumento es `0`, la variable lógica creada tendrá el valor *false*, cero. Si el número que se pasa como argumento es distinto de cero, la variable *logical* creada tendrá el valor *true*, uno.

```
bool1 = logical(0)
% El resultado será: bool1 = 0
bool2 = logical(1)
% El resultado será: bool2 = 1
bool3 = logical(-2)
% El resultado será: bool3 = 1
bool4 = logical(2500)
% El resultado será: bool4 = 1
```

1.2 Operadores

1.2.1 Operadores relacionales y de igualdad

Utilizados entre dos valores numéricos devuelven una variable *logical*, *true* (`1`) o *false* (`0`). Los siguientes son operadores válidos:

<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual a
~=	No igual a

Algunos ejemplos podrían ser:

```
2 < 3 % ans=1
2 > 3 % ans=0
2 >= 3 % ans=0
2 == 3 % ans=0
8 == (2^3) % ans=1
8 ~= 3 % ans=1
```

Hay que poner especial cuidado con el orden en que opera *Octave* cuando se concatenan varias operaciones seguidas en la misma línea. En general, cuando en una misma línea aparecen varios operadores relacionales, *Octave* opera la línea de izquierda a derecha. Veamos un ejemplo:

```
x = 6;
bool = 1 < x < 5
% El resultado será: bool = 1
x = 3
```

```
bool = 1 < x < 5
% El resultado será: bool = 1
```

En el ejemplo anterior, la expresión $1 < x < 5$ siempre devolverá el resultado *true*, *1*, independientemente del valor que tenga x . La explicación es la siguiente: *Octave* opera la expresión $1 < x < 5$ de izquierda a derecha; primero opera $1 < x$, que será *0* (false) o *1* (true) en función del valor de x . El resultado de esta operación lo compara con 5 , dando siempre como resultado *1*.

Para evitar errores en la operaciones, siempre es aconsejable hacer uso de los paréntesis. La expresión anterior se evaluaría correctamente escribiendo: $(1 < x) \& (x > 6)$

1.2.2 Operadores lógicos

Operan entre dos variables *logical* y el resultado también es un valor *logical*. Los siguientes son los operadores que se pueden utilizar:

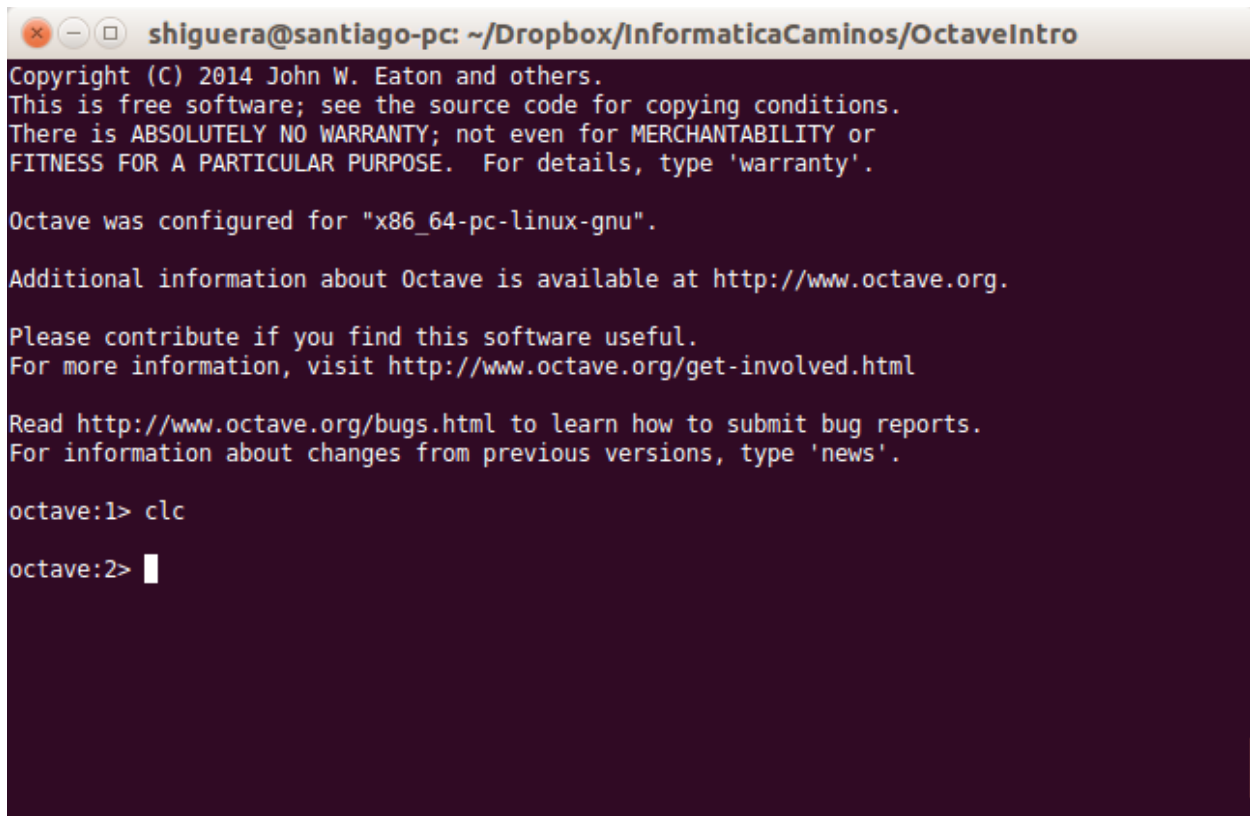
- **OR** Se utiliza el símbolo `|` (barra vertical). El resultado será cierto si es cierto el valor de al menos una de las dos variables *logical* que se operan
- **AND** Se utiliza el símbolo `&`. El resultado de la operación es cierto si son ciertas las dos variables *logical* que se operan.
- **OR EXCLUSIVO** Se utiliza la función `xor()`. En este caso el resultado es cierto si lo es una de las dos variables, pero no si son falsas o verdaderas las dos.
- **NOT** Este operador opera sobre una sola variable *logical*. El resultado será el contrario del valor de la variable.

```
x=3;
bool1 = (1<x) & (x<6)
% El resultado será: bool1 = 1
bool2 = xor((1<x), (x<6))
% El resultado será: bool1 = 0

x=7;
bool3 = (1<x) | (x<6)
% El resultado será: bool3 = 1
bool4 = ~(3<x)
% El resultado será: bool4 = 0
bool5 = xor((1<x), (x<6))
% El resultado será: bool5 = 1
```

1.3 Arranque de la consola de Octave

Partimos de un ordenador con Octave instalado. Abriremos un terminal para poder ejecutar instrucciones del sistema operativo. En *Linux* simplemente hay que teclear `'octave'`, en *Windows* el programa que tenemos que ejecutar para abrir la consola de Octave es un programa llamado `octave.exe` y que está en el directorio `'bin'` de la instalación de Octave. Si todo ha ido bien, nos aparecerá la consola de *Octave* con el *prompt* indicando algo parecido a `octave:1>`.



```
shiguera@santiago-pc: ~/Dropbox/InformaticaCaminos/OctaveIntro
Copyright (C) 2014 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1> clc
octave:2> █
```

Para salir de la consola de Octave hay que teclear ‘exit’ o ‘quit’

1.4 Cálculos elementales

Octave se puede utilizar como una calculadora. Se dispone de las operaciones aritméticas suma, resta, multiplicación, división y exponenciación. Pruebe a teclear las siguientes operaciones en la consola de Octave:

```
1+1
2-1
2*3.5
3.0/2.0
2^5
```

Tras cada operación hay que pulsar la tecla *intro*. La consola nos mostrará la palabra *ans*, el signo igual y el resultado de la operación. La palabra *ans* quiere decir *answer*, respuesta.

Octave dispone de funciones para calcular raíces cuadradas, logaritmos naturales, logaritmos decimales y funciones trigonométricas. Pruebe las siguientes operaciones en la consola de Octave:

```
sqrt(25)
log(10)
log10(10)
sin(90*pi/180)
cos(0)
tan(45*pi/180)
asin(1)*180/pi
acos(0)
atan(-1)*180/pi
```

Observamos un par de cosas en las expresiones anteriores:

- La utilización de la constante predefinida *pi*. Octave tiene varias constantes con valores predefinidos, una de ellas es el número *pi*=3.14...
- Las funciones trigonométricas trabajan con ángulos en radianes.

Podemos calcular exponenciales del número *e*, por ejemplo, la ecuación de *Euler*:

$$e^{i\pi} + 1 = 0$$

la podríamos comprobar tecleando algo así:

```
exp(pi*i)+1
```

Vemos que Octave entiende la variable *i* como el número complejo $i = \sqrt{-1}$. Podemos ver algunas de las variables que tiene predefinidas:

```
e
i
j
pi
ans
```

Cada vez que hacemos un cálculo, el resultado se guarda en una variable llamada *ans* que podemos utilizar en el siguiente cálculo.

1.5 Asignación de variables

Podemos almacenar valores en memoria mediante la asignación de dichos valores a *nombres válidos* de variables. El símbolo utilizado para la asignación es el símbolo **igual** '='.

En el ejemplo siguiente se asigna valor a las variables *x* e *y*. Estos valores quedan almacenados en la memoria de *Octave* y se pueden utilizar en operaciones posteriores utilizando en las expresiones el nombre de variable elegido. Se puede reasignar el valor de una variable en cualquier momento. El valor en memoria será el último asignado a la variable.

```
x = 2.5
y = 3
x+y
% El resultado será: ans = 5.5
x = 4
x+y
% El resultado será: ans = 7
```

Vemos que el esquema de la *instrucción de asignación en *Octave* es situar el nombre de variable, a continuación el símbolo igual y a la derecha del símbolo igual el valor que queremos asignar a la variable.

A la derecha del símbolo igual puede aparecer un valor numérico tecleado explícitamente o cualquier expresión válida de *Octave*. En el siguiente ejemplo se asignan distintos valores a variables:

```
x = pi/2;
y = sin(x)
% El resultado será: y = 1
```

Note: La variable predefinida **ans** de *Octave* guarda el resultado de la última operación realizada en la consola de *Octave*, siempre que esa operación no sea de asignación. Si realizamos una asignación, el valor de la variable *ans* no varía.

Note: Si al hacer una asignación no queremos que muestre el resultado de la asignación en consola tenemos que finalizar la sentencia con **punto y coma** ‘;’.

1.6 Nombres de variables

Los nombres de variables pueden contener letras, números y caracteres underscore (guión bajo), pero el primer carácter tiene que ser letra.

Warning: No hay que utilizar como nombres de variables los nombres de variables predefinidas, funciones o comandos de *Octave*.

Note: Las letras que se pueden utilizar en los nombres de variables son las del alfabeto inglés. Los caracteres locales, (ñ, letras acentuadas), no se deben utilizar en el nombre de variables o funciones.

Ejemplos de nombres válidos de variables:

```
x
x2
XX_2
ultimoValor
n_factorial
```

Ejemplos de nombres no válidos de variables:

```
6x
end
n!
```

El número máximo de caracteres que puede tener el nombre de una variable se puede consultar con la función **namelengthmax()**:

```
namelengthmax()
% El resultado en mi consola: ans = 63
```

Note: Los nombres que dan una idea de para qué sirven las variables hacen que la legibilidad del código fuente de los programas mejore mucho. Es más fácil de seguir y comprender un programa, (un tercero o el mismo programador al cabo de unas semanas o meses), cuando los nombres de variables y funciones se eligen adecuadamente. Por ejemplo, la variable *numFilas* nos dice más que la variable *n*, y puede ser de gran ayuda para *seguir* el hilo del programa en una serie de bucles y sentencias *if* anidadas. Un criterio habitual es denominar a una variable con más de una palabra, poniendo la inicial de la primera palabra en minúsculas y las iniciales del resto de palabras en mayúsculas. Así podremos utilizar nombres de variables como *contadorVehiculos* o *ultimaFila*.

1.7 Variables predefinidas en Octave

Vectores y Matrices

Contents

- *Vectores y Matrices*
 - *Vectores*
 - *Rangos*
 - *Definición de vectores a partir de rangos*
 - *La función linspace()*
 - *Lectura de las componentes de un vector*
 - *Concatenación de vectores*
 - *Funciones utilitarias para trabajar con vectores*

2.1 Matrices

En *Octave* es fácil definir una matriz. No es necesario determinar las dimensiones a priori, *Octave* las adaptará a los datos introducidos.

Para introducir una matriz por teclado bastará ir tecleando el valor de las componentes encerradas entre corchetes. Se teclea cada fila separada de la anterior por **punto y coma** ‘;’, y , dentro de cada fila, los elementos se separarán con **coma** ‘,’ o espacio. Por ejemplo, el siguiente código *Octave*:

```
A = [1, 2; 3, 4]
```

dará lugar a que *Octave* asigne a la variable *A* la matriz de dos filas y dos columnas siguiente:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Note: Para separar elementos dentro de una fila se puede utilizar indistintamente la *coma* o la *barra espaciadora*. Para separar una fila de otra se puede utilizar indistintamente el *punto y coma* o cambiar de línea en pantalla pulsando la tecla *intro*.

2.1.1 Rangos y matrices

Todo lo dicho en el apartado de Vectores en relación a cómo definir series de datos equiespaciados a base de *rangos* o con la función *linspace()* es aplicable a la definición de matrices. Habrá que tener la precaución de que las filas deben

ser todas con el mismo número de columnas:

```
A = [1:10; 11:20]
% El resultado será la matriz A= [1 2 3...9 10 ; 11 12 ... 19 20]
```

2.1.2 Lectura de las componentes de una matriz

Para leer el valor de una componente de una matriz se pone el nombre de la variable que guarda la matriz y a continuación, entre paréntesis, la fila y la columna de la componente buscada. La primera fila y la primera columna tienen el índice 1, y también aquí podemos referirnos abreviadamente al índice de la última fila o columna de la matriz utilizando la cláusula **end**:

```
A = [1,2; 3, 4]
A(1,1)
% El resultado será: ans = 1
A(1,end)
% El resultado será: ans = 2
A(end,end)
% El resultado será: ans = 4
```

Analogamente a lo explicado en el caso de los vectores, podemos utilizar rangos para extraer una serie de valores de la matriz. En este caso el resultado será una matriz (submatriz) con el número de filas y columnas acorde a los rangos solicitados:

```
A = [1,2,3,4; 5,6,7,8; 9,10,11,12; 13,14,15,16]
A(2:3, 2:3)
% El resultado será la matriz [6, 7; 10, 11]
A([2 4],[2 4])
% El resultado será la matriz [6, 8; 14, 16]
```

En el caso de las matrices existe una construcción especial para referirse a todos los elementos de una fila o columna y es utilizando los **dos puntos ':'** como índice:

```
A = [1,2,3,4; 5,6,7,8; 9,10,11,12; 13,14,15,16]
A(:,3)
% El resultado será la columna 3
A(2,:)
% El resultado será la fila 2
```

Internamente *Octave* almacena las matrices como un vector columna, poniendo una columna tras la otra. Es posible referirse a un elemento de una matriz mediante un único índice que en este caso hará referencia a la posición del elemento en el vector columna resultante del almacenamiento de la matriz en una única columna:

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


2.1.3 Direccionamiento indirecto de matrices a partir de vectores

Los índices de los elementos que queremos extraer de una matriz los podemos indicar mediante un vector, explícitamente o que esté guardado en memoria:

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
v1 = [1 2]
v2 = [1 2]
A(v1,v2)
% El resultado será la matriz [1,2; 4,5]
A([1 2],[1 2])
% El resultado será la matriz [1,2; 4,5]
```

2.1.4 Eliminación de filas o columnas

Podemos eliminar una fila o columna de una matriz mediante la siguiente construcción, que asigna el valor [] a los elementos correspondientes:

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
A(1,:) = []
% Se elimina de la matriz A la primera fila. A queda como matriz 2x3
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
A(:,3) = []
% Se elimina de la matriz A la tercera columna. A queda como matriz 3x2
```

2.1.5 Operaciones con matrices

- **Suma +:**
- **Resta -:**
- **Producto *:**
- **Producto elemento a elemento .*:**
- **Exponenciación ^:**
- **Exponenciación elemento a elemento .^:**
- **División /:**
- **División elemento a elemento ./:**
- **División invertida \:**
- **División invertida elemento a elemento .\:**
- **Operador traspuesta ':**

2.1.6 Matrices predefinidas

Hay varias funciones utilitarias que permiten construir matrices de tipos particulares:

- **eye(n)** Forma la matriz identidad cuadrada de dimensión n
- **zeros(n)** Forma una matriz de ceros cuadrada de dimensión n
- **zeros(m,n)** Forma una matriz de **ceros** de m filas y n columnas

- **ones(*n*)** Forma una matriz de **unos** cuadrada de dimensión n
- **ones(*m,n*)** Forma una matriz de **unos** de m filas y n columnas

2.1.7 La función *size()*

La función *size()*, recibe como parámetro una matriz y nos devuelve un vector de dos componentes con el número de filas y el número de columnas de la matriz. La forma de la función es:

Un ejemplo de utilización podría ser:

```
octave:74> A = [1 2 3; 4 5 6; 7 8 9]
A =

     1     2     3
     4     5     6
     7     8     9

octave:75> dims=size(A)
dims =

     3     3

octave:76> 
```

La función *size()* admite un segundo parámetro que si vale *1* nos devolverá el número de filas y si vale *2* nos devolverá el número de columnas:

```
octave:79> A=[1:4;5:8;9:12]
A =

     1     2     3     4
     5     6     7     8
     9    10    11    12

octave:80> numfilas=size(A,1)
numfilas = 3
octave:81> numcolumnas=size(A,2)
numcolumnas = 4
octave:82> 
```

2.1.8 Inversa, determinante y traza de una matriz

Existen funciones específicas para calcular la inversa, el determinante y la traza de una matriz cuadrada:

- `inv()`
- `det()`
- `trace()`

Si la matriz es singular se producirá un error que será indicado por *Octave*

2.1.9 Funciones `max()`, `min()`, `sum()` y `prod()`

La función **`max()`**, cuando se aplica a un vector, nos devuelve el valor del máximo elemento del vector, cuando se aplica a una matriz, nos devuelve un vector fila con el maximo elemento de cada columna de la matriz.

La función **`min()`** funciona de manera análoga a la función *max()* pero devolviendo valores mínimos.

La función **`sum()`**, cuando se aplica a una matriz, devuelve un vector fila en la que cada elemento es la suma de los elementos de la columna correspondiente de la matriz pasada como argumento.

La función **`prod()`** devuelve en una fila el producto de los elementos de las columnas de la matriz original.

2.1.10 Aplicación de funciones a matrices

En general, cuando pasemos una matriz o un vector como argumento de una función, nos devolvera una matriz o vector de las mismas dimensiones con la función aplicada elemento a elemento. Por ejemplo:

```
octave:98> v = [-pi:pi/4:pi]
v =
   -3.14159   -2.35619   -1.57080   -0.78540    0.00000    0.78540    1.57080    2.35619    3.14159

octave:99> sin(v)
ans =
   -0.00000   -0.70711   -1.00000   -0.70711    0.00000    0.70711    1.00000    0.70711    0.00000

octave:100> 
```

2.1.11 Aplicación de operadores lógicos a matrices

Los operadores lógicos (`&`, `|`, `xor()`) o relacionales (`<`, `>`, `<=`, `>=`, `==`, `~=`) se pueden utilizar cuando uno o los dos operandos son matrices.

Cuando aplicamos un operador lógico o relacional a una matriz, el operador se aplicará elemento a elemento, obteniendo como resultado una matriz de las mismas dimensiones y cuyas componentes son valores *logical*, resultado de aplicar la operación al elemento.

```
A = [1, 2; 3, 4];
B = [1, 0, 3, 0];
C = A & B
% El resultado será: C = [1 0; 1 0]
```

```
D = A > 2
% El resultado será: D = [0 0; 1 1]
```

Se pueden utilizar las matrices resultado para generar submatrices mediante referenciación indirecta, esto es, utilizando la matriz de valores *logical* obtenida como matriz que define los índices que queremos extraer de la matriz original, de la siguiente manera:

```
A = [1, 2; 3, 4];
B = [1, 0, 3, 0];
C = A & B
% El resultado será: C = [1 0; 1 0] (Matriz de elementos logical)
A(C)
% El resultado será: ans = [1; 3] (Vector columna de doubles)
D = A > 2
% El resultado será: D = [0 0; 1 1] (Matriz de elementos logical)
A(D)
% El resultado será: ans = [3; 4] (Vector columna de doubles)
E = A > 1
% El resultado será: E = [0, 1; 1 1] (Matriz de elementos logical)
% El resultado será: ans = [3; 2; 4] (Vector columna de doubles)
```

El resultado de la referenciación indirecta aplicada a una matriz es un vector columna obtenido en el orden en que almacena *Octave* la matriz, que es con las columnas almacenadas una tras otra.

Cuando la matriz que operamos es un vector fila, la referenciación indirecta nos devuelve un vector fila, no columna.

2.1.12 Funciones utilitarias para matrices

- **repmat(A,m,n)** Devuelve una matriz resultado de copiar la matriz A en *m* filas y *n* columnas. Si A es un escalar, el resultado será una matriz *m*×*n* con valor A en todos los elementos
- **diag(A)** Siendo A una matriz, devuelve un vector columna con los elementos de la diagonal de A
- **diag(v)** Siendo v un vector, devuelve una matriz diagonal con los elementos de v ocupando la diagonal.
- **blkdiag(A,B)** Crea una matriz diagonal de submatrices (por bloques) a partir de las matrices A y AB

2.2 Vectores

Octave trabaja con vectores de datos, tanto con vectores fila como con vectores columna.

Para definir un vector fila por teclado solo hay que teclear los elementos separados por un espacio o por una coma y encerrados entre corchetes.

Analogamente, un vector columna se define tecleando la lista de elementos separados por punto y coma ';' o por la pulsación de la tecla *intro*:

```
octave:11> v = [ 1 2 3 4 5]
v =

    1    2    3    4    5

octave:12> w = [6; 7; 8]
w =

     6
     7
     8

octave:13> 
```

2.3 Rangos

En *Octave* podemos definir un tipo de datos especial, llamado **Rango**, consistente en una colección ordenada (una *serie*) de números equiespaciados.

Para definir un *rango* debemos indicar a *Octave* el valor inicial del rango, el incremento entre valores sucesivos y el máximo valor que pueden alcanzar los valores de la serie. La construcción para definir un *rango* coloca los tres valores separados por ‘:’:

```
% rango = inicio : incremento : máximo_valor
rango = 1:2:10
% El resultados será: 1 3 5 7 9
```

Cuando se omite el *incremento*, *Octave* asume que el incremento es la unidad:

```
% rango = inicio : máximo_valor
rango = 1:4
% El resultado será: 1 2 3 4
```

Se pueden establecer rangos de valores decrecientes. Para ello el valor de inicio deberá ser mayor que el valor final y el incremento un valor negativo:

```
rango = 4:-1:0
% El resultado será: 4 3 2 1 0
```

Para especificar los valores inicio, incremento o máximo de un rango se pueden utilizar funciones, o cualquier otra expresión de *Octave* que devuelva un resultado válido, por ejemplo:

```
rango = 0:intmax('uint8')
% El resultado serán los primeros 255 números naturales: rango = 0 1 2 ... 255
rango2 = 0:2:2^3
% El resultado será: rango2 = 0 2 4 6 8
```

2.4 Definición de vectores a partir de rangos

Una de las utilizaciones de los *rangos* es la definición de las componentes de un vector. Veamos un ejemplo. Supongamos que queremos un vector denominado *az* cuyas componentes sean los ángulos 0, 90, 180, 270. Podríamos definirlo de la siguiente manera:

```
rango = 0:90:270
az = [rango]
% El resultado será: az=[0 90 180 270]
```

De hecho, no es necesario definir primero el rango y luego el vector. La manera habitual de hacerlo es definiendo el rango directamente dentro de los corchetes:

```
az = [0:90:270]
% El resultado será: az=[0 90 180 270]
```

2.5 La función *linspace()*

La función ***linspace()*** es también muy útil para definir vectores. Devuelve un vector con *n* números equiespaciados entre unos números de inicio y final. Los argumentos que se le pasan a la función son el número de inicio de la serie, el número final de la serie y el número total de elementos que tendrá la serie:

```
% v = linspace( inicio, fin, num_elementos)
v = linspace(1,10,10)
% El resultado será: v=[1 2 3 4 5 6 7 8 9 10]
```

La función *linspace()* permite también realizar series descendentes, y también podemos utilizar expresiones válidas de *Octave* para los parámetros:

```
w = linspace(10,5,5)
% El resultado será: w=[10 9 8 7 6]
z = linspace(0,pi/2,4)
% El resultado será: z=[0.00000 0.52360 1.04720 1.57080]
```

2.6 Lectura de las componentes de un vector

Cuando queramos referirnos a una componente concreta de un vector, utilizaremos el nombre del vector seguido del índice de la componente encerrado entre paréntesis. En *Octave*, el índice de la primera componente de un vector es el **1**. Veamos un ejemplo refiriéndonos al vector definido en el epígrafe anterior *az* = [0 90 180 270] :

```
az(1)
% El resultado será: 0
az(2)
% El resultado será: 90
az(4)
% El resultado será: 270
```

Octave proporciona un *truco* para extraer la última componente de un vector, sin tener que explicitar el índice, y es utilizar la palabra **end**. En el ejemplo anterior es equivalente escribir *az(4)* que *az(end)*:

```
az(4)
% El resultado será: 270
az(end)
% El resultado será: 270
```

También podemos definir las componentes a extraer mediante un rango. En el siguiente ejemplo extraemos la segunda y tercera componente del vector *az*. El resultado de la operación es un vector de dos componentes, *[90 180]*, que se lo asignamos a un nuevo vector que hemos llamado *es*:

```
es = az(2:3)
% El resultado será: es=[90 180]
```

Otro ejemplo de extracción de un *rango* de componentes de un vector es el siguiente, en el que primeramente definimos un vector *v* y a continuación extraemos algunas de sus componentes a un nuevo vector *W*:

```
v = [1:2:10] % Se define un vector y se asigna a la variable v
% El resultado será: v=[1 3 5 7 9]
w = v(1:2:5) % Se extraen ciertas componentes del vector v que se asignan a w
% El resultado será: w=[1 5 9]
```

Note: Es importante distinguir entre las sentencias utilizadas para definir un vector, que utilizan **corchetes** `[]`, y las utilizadas para extraer componentes de un vector, que utilizan el nombre del vector seguido de las componentes a extraer entre **paréntesis** `()`.

2.7 Concatenación de vectores

Podemos unir dos vectores, esto es, poner las componentes de un vector a continuación de las de otro para formar un único vector. Para ello se definen entre corchetes los elementos a concatenar separados por *coma* o por *espacio*. Veamos un ejemplo:

```
a = [1 2 3];
b = [4 5 6];
c = [a b]
% El resultado será: c = [1 2 3 4 5 6]
```

Podemos hacer otro tipo de concatenaciones, por ejemplo, concatenar un vector con un rango, o un número con un vector, o un vector con un número. De hecho, cuando definimos un vector tecleando sus componentes, lo que hacemos es concatenar varios números:

```
a = [1 2 3];
c = [a 5]
% El resultado será: c = [1 2 3 5]
d = [-5:0 a 4:5]
% El resultado será: d = [-5 -4 -3 -2 -1 0 1 2 3 4 5]
d = [-5 -4 -3 -2 -1 0 a 4 5]
% El resultado será: d = [-5 -4 -3 -2 -1 0 1 2 3 4 5]
```

Se pueden concatenar de manera similar vectores columna. En este caso las distintas *subcomponentes* se separarán por punto y coma.

2.8 Funciones utilitarias para trabajar con vectores

Hay varias funciones útiles a la hora de trabajar con vectores. Siendo *v* un vector:

- **length(v)** Devuelve el número de componentes del vector *v*
- **max(v)**, **min(v)** Devuelve el valor máximo/mínimo de entre las componentes de *v*
- **sum(v)**, **prod(v)** Devuelve la suma/producto de las componentes de *v*

- **norm(v)** Devuelve el módulo del vector v
- **dot(v, w)** Devuelve el producto escalar de los vectores v y w
- **cross(v, w)** Devuelve el vector producto vectorial de v y w (Las dimensiones de v y w deben ser congruentes como máximo tres)
- **sort(v)** Devuelve un vector con las componentes de v ordenadas de menor a mayor

Funciones predefinidas en Octave

Octave tiene una extensa librería de funciones predefinidas que abarcan numerosos campos de la ciencia matemática.

3.1 Funciones para cadenas de caracteres

Note: La documentación de Octave y Matlab para manejar cadenas de caracteres se puede encontrar en:

- **Octave:**
- **Matlab:**
- **strrep** : Reemplaza una subcadena dentro de una cadena

```
cad = '3,78'
newcad = strrep(cad, ',', '.');
% result : newcad = 3.78
```

–**strtrim(cad)**: Quita los caracteres *espacio* del inicio y final de una cadena

- **C = strsplit(str, delimiter)**: Devuelve un *CellArray* con las subcadenas que resultan de dividir la cadena original *str* con el separador *delimiter*.

```
cad = '3,78'
c = strsplit(cad, ',');
pentera = str2num(c{1}) % result 3 double
pdec = str2num(c{2}) % result 78 double
```

Note: Recordar que para extraer elementos de un *CellArray* se utilizan llaves, no paréntesis. La función *size()* con *CellArrays* funciona igual que con matrices normales.

- **str2num(cad)**: Convierte en número double la cadena *cad*

3.2 La función *rand()*

La función **rand()** se utiliza para generar numeros pseudoaleatorios. Se pueden generar scalares o matrices de números pseudoaleatorios. Si llamamos a la función sin argumentos nos devolverá un numeros pseudoaleatorios uniformemente distribuidos entre 0 y 1.

Si pasamos un número entero n como parámetro, la función nos devolverá una matriz cuadrada de dimensión n de números pseudoaleatorios comprendidos entre 0 y 1. Si le pasamos dos números enteros m y n la matriz resultante tendrá m filas y n columnas.

El número que está utilizando *Octave* como *semilla* generadora de los números pseudoaleatorios se puede ver con `rand('seed')`. También podemos modificar la *semilla* con `rand('seed', x)`, donde x es la nueva *semilla* que queremos utilizar.

```
octave:86> rand()
ans = 0.93631
octave:87> rand
ans = 0.81563
octave:88> rand
ans = 0.32130
octave:89> rand
ans = 0.93011
octave:90> rand('seed')
ans = 6.8447e-310
octave:91> rand(3)
ans =

    0.651459    0.192528    0.251040
    0.935898    0.048588    0.682341
    0.070046    0.383789    0.748951

octave:92> rand(4,2)
ans =

    0.860424    0.847851
    0.515507    0.823250
    0.136358    0.464736
    0.087084    0.860010

octave:93> 
```

Los números generados por `rand()` son doubles comprendidos entre 0 y 1. Si queremos generar números comprendidos en un determinado intervalo debemos operar los números obtenidos de `rand()` para adaptarlos al intervalo buscado:

```
M = 1
rand() + M
% El resultado estará comprendido entre M y M+1 (en este caso entre 1 y 2)
%
M = 10
rand() * M
% El resultado estará comprendido entre 0 y M
%
A = 10
B = 20
```

```
A + (B-A)*rand()
% El resultado estará entre A y B
```

Para generar numeros enteros en un determinado intervalo se puede utilizar la función **randi()**:

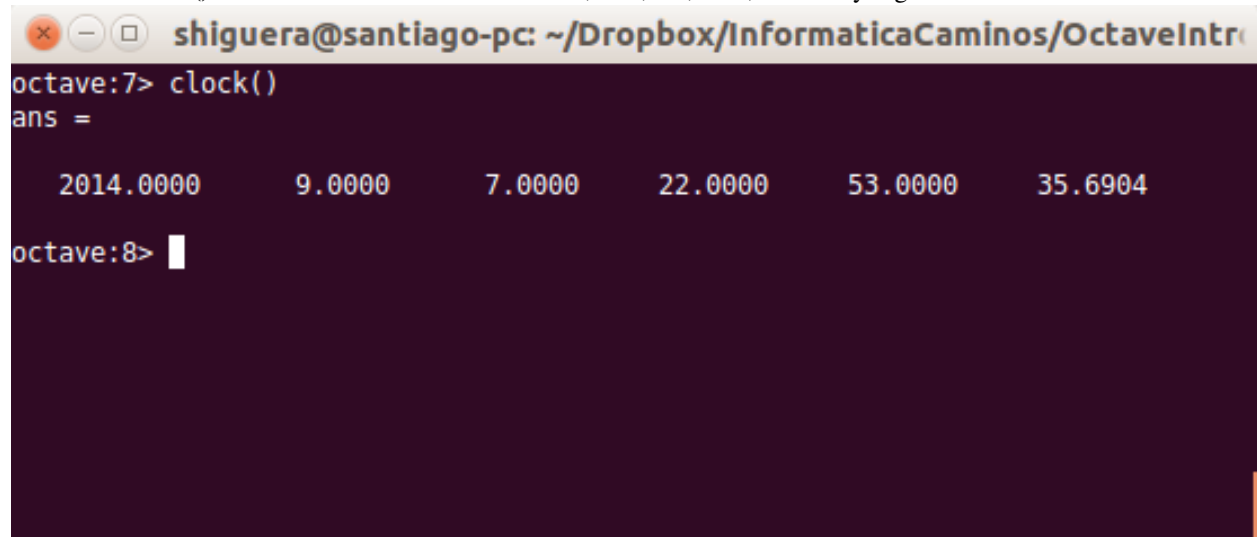
```
randi(10)
% Genera un número entero entre 1 y 10
```

También se pueden conseguir enteros mediante la función **rand()**:

```
A = 10
B = 20
round(A + (B-A)*rand())
% El resultado será un entero (double con parte decimal cero) entre A y B
int32(round(A + (B-A)*rand()))
% El resultado será un entero int32 entre A y B
```

3.3 Utilidades para medir el tiempo

La función **clock()** nos devolverá un vector con el año, mes, día, hora, minutos y segundos:



The screenshot shows a terminal window titled "shiguera@santiago-pc: ~/Dropbox/InformaticaCaminos/OctaveIntro". The prompt is "octave:7> clock()". The output is "ans =", followed by a row of six numbers: "2014.0000", "9.0000", "7.0000", "22.0000", "53.0000", and "35.6904". The prompt then changes to "octave:8>".

Las función **date()** nos devuelve la fecha como una cadena de texto.

La función **time()** nos devuelve el número de segundos transcurridos desde las 00:00 horas UTC del día 1 de enero de 1970.

La función **etime()** (*elapsed time*) nos devolverá el tiempo transcurrido entre dos fechas:

```
t0 = clock();
# unas cuantas computaciones más tarde...
tiempo_transcurrido = etime(clock(), t0);
```

La función **is_leap_year()** nos devolverá *1 (verdadero)* si el año pasado como argumento es bisiesto.

Programación en Octave

4.1 Scripts

Podemos escribir una serie de instrucciones de Octave en un fichero con extensión *.m* y luego ejecutarlo desde la consola. Al fichero con las instrucciones de Octave se le llama un *script* o programa. Para ejecutarlo tendremos que situarnos en el directorio donde esté el fichero y teclear su nombre sin extensión en la consola de Octave.

El lenguaje *m* es un lenguaje interpretado, esto es, Octave lee los ficheros *m* y los va ejecutando línea a línea, sin una compilación previa.

Para escribir los programas utilizaremos un editor de texto. La versión actual de Octave trae un editor incorporado y también copia en el directorio de instalación una versión del editor *Notepad++*.

Note: Se puede ejecutar un fichero *.m* situado en otro directorio distinto del directorio de trabajo. Para ello tendremos que teclear la ruta completa del fichero. También puede suceder que el fichero esté grabado en algún directorio del *path* de Octave. Entonces no será necesario teclear la ruta completa, sino simplemente el nombre del fichero.

4.2 El programa *Hola mundo*

Es tradicional comenzar el aprendizaje de cualquier lenguaje de programación con el típico programa *Hola Mundo*. Se trata de escribir un programa que muestre por pantalla la frase *Hola mundo*. El ejercicio permite comprobar que uno sabe crear el fichero de instrucciones, escribir alguna instrucción en él y hacer que el ordenador ejecute el programa.

En nuestro caso vamos a crear un fichero en el mismo directorio de trabajo en el que nos encontremos, llamado **'hello.m'**, y vamos a escribir en él una única instrucción:

```
disp('Hola, mundo');
```

La función *disp()* nos permite mostrar una variable por pantalla, en este caso una cadena de texto. Tendremos que grabar el archivo en el disco con el nombre *'hello.m'* y después, desde la consola, teclear *hello*. Si todo va bien, el programa se ejecutará, nos mostrará el mensaje *'Hola mundo'* y finalizará, con lo que volveremos a tener el *prompt* de la consola a disposición para seguir tecleando instrucciones.

Desde el editor integrado en la versión gráfica de Octave podemos crear un nuevo fichero, escribir en él la instrucción y grabarlo con el nombre elegido.

Note: A mi me gusta utilizar alguna frase que incluya algún acento o alguna ñ para comprobar que funciona la codificación de caracteres. Podemos probar con una frase del tipo *Hola, ¿qué tal?*, que incluye el signo de apertura de

interrogación y una e acentuada. A mí, con la versión 3.8 de Octave funcionando bajo linux me funciona perfectamente la codificación de caracteres.

4.3 Líneas de Comentarios

Las líneas de comentarios son líneas que no serán ejecutadas por Octave al correr el programa. Sirven para documentar las rutinas.

Los comentarios pueden ser de una línea. Para ello se deben preceder del símbolo ‘#’ o del símbolo ‘%’. El comentario será desde el símbolo *comentario* hasta el final de la línea. Puede ocupar toda la línea, si el símbolo *comentario* se pone al principio de la misma, o comentar una línea concreta, desde el símbolo *comentario* hasta el final de la línea. Veamos un ejemplo:

```
# Cuenta atrás para cohetes
disp (3);
disp (2);
disp (1);
disp ("Blast Off!"); # El cohete abandona la plataforma
```

Warning: *MATLAB* solo admite el símbolo porcentaje ‘%’ como indicador de comentarios, no admite el símbolo ‘#’.

Se pueden comentar bloques de código. Para ello hay que encerrar el bloque a comentar entre los símbolos ‘#{’ y ‘#}’. Estos símbolos tienen que aparecer como únicos caracteres en su línea. ‘#{’ aparecerá como línea previa al código a comentar, y ‘#}’ como línea posterior. Una utilidad de lo anterior es ‘desactivar’ bloques de código durante el proceso de depuración de los programas, por ejemplo.

```
# Cuenta atrás para cohetes
disp (3);
#{
disp (2);
disp (1);
#}
disp ("Blast Off!"); # El cohete abandona la plataforma
```

En esta versión del programa las líneas de *disp(2)* y *disp(1)* no se ejecutarán, y se pasará directamente del *disp(3)* al lanzamiento.

Comentar una línea de código es habitual durante la fase de creación y depuración de los programas. Ello nos permite deshabilitar ciertas partes del código para estudiar el funcionamiento del resto.

4.4 La instrucción input

La instrucción **input()** nos permite solicitar al usuario algunos datos que se necesitan para los cálculos posteriores. Por ejemplo, si queremos hacer un programa que devuelva el valor de elevar un número al cubo podríamos hacer lo siguiente:

```
x = input('Teclée un número: ');
y = x^3;
disp(y)
```

La función *input()* recibe una cadena de texto a modo de mensaje que muestra por consola, situando el cursor a continuación a la espera de que el usuario teclée el valor solicitado. Una vez el usuario pulsa la tecla *intro*, el valor

teclado será asignado a la variable elegida, en el ejemplo anterior la variable x . El programa, entonces, se seguirá ejecutando en la línea siguiente a la de la sentencia *input* y podrá utilizar el valor de x en los cálculos posteriores. En el ejemplo, se crea una nueva variable y que almacena el valor de x^3 y a continuación se muestra el valor de y por pantalla mediante la sentencia *disp()*.

Note: Cuando una instrucción se acaba con un punto y coma se omite la salida por pantalla de dicha instrucción. En el ejemplo, en la segunda instrucción, donde asignamos el valor x^3 a la variable y , de no poner el punto y coma, se mostraría por pantalla el valor asignado a y . Se puede comprobar repitiendo la ejecución del programa con el punto y coma suprimido

Note: En Matlab no se pueden meter subfunciones dentro de los scripts, solo funciones anónimas, por ejemplo: `f=@(x) x.^2; x=[1 2 3]; x2=f(x);`

Polinomios

En Octave los polinomios se representan por un vector de coeficientes ordenados de la mayor potencia hacia la menor. Por ejemplo, el polinomio $2x^3 - 3x^2 + 2$ se representa por el vector $[2, -3, 0, 2]$.

Sea p un vector que define los coeficientes de un polinomio. Hay varias funciones para operar con polinomios:

- **polyout(p, 'x')**: Expresa el polinomio en términos de la variable x
- **y = polyval(p, x)**: Devuelve el valor del polinomio para el valor x . Si x es un vector o matriz, devuelve un vector o matriz con la evaluación del polinomio p en cada elemento.
- **b = conv(p, q)**: Devuelve el vector de coeficientes que define el polinomio b , producto de los polinomios p y q .
- **[b, r] = deconv(p, q)**: Realiza la división de los polinomios p y q , devolviendo el polinomio cociente b y el polinomio resto r , de forma que se cumple: $p = conv(q, b) + r$
- **r = roots(p)**: Devuelve un vector r con las raíces del polinomio p
- **d = polyder(p)**: Devuelve un vector con los coeficientes del polinomio que resulta de derivar p
- **s = polyint(p)**: Devuelve un vector con los coeficientes del polinomio que resulta de integrar p
- **p = polyfit(x, y, n)**: Devuelve el polinomio de grado n que mejor ajusta por mínimos cuadrados el conjunto de puntos (x_i, y_i)

Ejercicios resueltos

6.1 Ejercicio 1 (Función size())

Escribir una función que devuelva *logical 1* si la variable que recibe como argumento es un vector fila o columna, y *logical 0* en otro caso (argumento escalar o argumento matriz).

```
function is = isVector(v)
% Devuelve logical 1 si la variable recibida
% es un vector y logical 0 en otro caso.

% Comprobación del número de argumentos
if (nargin ~= 1)
    is = logical(0);
    return;
end

if (xor(size(v,1)==1, size(v,2)==1))
    % Es un vector
    is = logical(1);
else
    % No es un vector
    is = logical(0);
    return;
end

end
```

6.2 Ejercicio 2 (Bucles for)

Escribir una función que calcule la suma s de los elementos de un vector que recibe como argumento. La función comprobará que el argumento recibido es un escalar o un vector. En caso de recibir un argumento no válido, la función devolverá *NaN*.

Note: Octave tiene una función de librería que realiza los cálculos pedidos en el enunciado, se trata de la función *sum()*. En la solución del ejercicio no se podrán utilizar funciones de librería, el propósito del ejercicio es que el alumno utilice bucles y operadores aritméticos corrientes.

```
function s = sumvector(v)
% Suma las componentes de un vector

    if(nargin ~= 1)
        s = NaN;
        return;
    end

    if( (size(v,1) ~= 1) & (size(v,2) ~= 1) )
        s = NaN;
        return;
    end

    s = 0;
    for index = 1: length(v)
        s = s + v(index);
    end

end
```

6.3 Ejercicio 3 (Bucles *for*. Algoritmo de la suma)

Escribir una función que devuelva la suma de todos los elementos de un vector o matriz que recibe como argumento. En el caso de recibir un escalar o un vector la función pedida sería equivalente a la función `sum()` de la librería. En el caso de recibir una matriz como argumento, el resultado de la función pedida sería equivalente a la expresión `sum(sum())`. En la solución no se utilizarán funciones de librería, solamente operadores aritmeticos sencillos.

```
function s = sumall(A)
% Suma todos los elementos de un vector o una matriz

filas = size(A,1);
cols = size(A,2);

s = 0;
for ind1 = 1: filas
    for ind2 = 1: cols
        s = s + A(ind1,ind2);
    end
end

end
```

6.4 Ejercicio 4 (Bucles *for*. . Algoritmo del producto)

Escribir una función que devuelva el producto de todos los elementos de un vector o matriz que recibe como argumento. En el caso de recibir un escalar o un vector la función pedida sería equivalente a la función `prod()` de la librería. En el caso de recibir una matriz como argumento, el resultado de la función pedida sería equivalente a la expresión `prod(prod())`. En la solución no se utilizarán funciones de librería, solamente operadores aritmeticos sencillos.

```
function p = prodall(A)
% Multiplica todos los elementos de un vector o una matriz

filas = size(A,1);
cols = size(A,2);
```

```

p = 1;
for ind1 = 1: filas
    for ind2 = 1: cols
        p = p * A(ind1,ind2);
    end
end
end

```

6.5 Ejercicio 5 (Bucles for)

Escribir una función que realice el *puntoproducto* .* de dos vectores que recibe como argumento. La función comprobará que el número de argumentos recibidos es correcto, y que se trata de dos vectores. Si los argumentos no son correctos, la función devolverá *NaN*. En la solución solo se permitirá la utilización de bucles y operadores aritméticos sencillos. La función devolverá el resultado correcto independientemente de que los vectores recibidos sean fila-fila, col-col, fila-col o col-fila.

```

function pp = puntoprod(v, w)
% Calcula el punto producto de dos vectores que recibe como argumento

% Comprobación del número de argumentos recibidos
if (nargin ~= 2)
    pp = NaN;
    return;
end

% Comprobación de la dimensión de los vectores
if (size(v,1) ~= 1 & size(v,2) ~= 1)
    pp = NaN;
    return;
elseif (size(w,1) ~= 1 & size(w,2) ~= 1)
    pp = NaN;
    return;
elseif (length(v) ~= length(w))
    pp = NaN;
    return;
end

pp = 0;
for index = 1:length(v)
    pp = pp + v(index) * w(index);
end

```

6.6 Ejercicio 6 (Bucles for)

Escribir una función que calcule la matriz *C* producto de dos matrices *A* y *B* que recibe como argumentos. La función comprobará, en primer lugar, que el número de argumentos recibidos es el adecuado y que las dimensiones de las matrices *A* y *B* son congruentes. En caso de argumentos incorrectos la función devolvera una matriz vacía []. Si los argumentos son correctos, la función calculará la matriz *C* sin hacer uso de funciones de librería, utilizando bucles y operadores aritméticos sencillos.

```

function C = prodmat(A, B)
% prodmat.m : Calcula la matriz C, producto de las matrices A y B

```

```
% Comprobación del número de argumentos recibidos
if (nargin ~= 2)
    C = [];
    return;
end

% Comprobación de la dimensión de las matrices
% El número de columnas de A debe ser igual al de filas de B
% Si las dimensiones no son las correctas el programa finaliza
if (size(A,2) ~= size(B,1))
    C = [];
    return;
end

% Preparo la matriz solución con las dimensiones adecuadas
% Si son A(m,n) y B(n,p) las dimensiones de C seran C(m,p)
C = zeros(size(A,1),size(B,2));

% Cálculo del producto mediante dos bucles for anidados
for fila = 1: size(A,1)
    for col = 1:size(B,2)
        sum = 0;
        for(index=1:size(A,2))
            sum = sum + A(fila, index) * B(index, col);
        end
        C(fila,col) = sum;
    end
end
end
```

6.7 Ejercicio 7 (Strings, Funciones)

La variable `result = '11X121XX1X21X1'` contiene los resultados de los catorce partidos de futbol de la quiniela de una jornada. La variable `apuesta = '112X211X12X1XX'` tiene los resultados apostados en una determinada quiniela. Se pide: (a) Calcule en una sola expresión cuantos resultados hay acertados. (b) Cuales son los numeros de orden de los partidos que se han acertado. (C) Cuáles son los resultados acertados.

```
numAciertos = sum(result == apuesta)
ans = 8

partidosAcertados = find(result == apuesta)
ans =     1     2     5     6     8     9    12    13

resultadosAcertados = result(result == apuesta)
ans = 1121X11X
```

6.8 Ejercicio 8 (Función rand())

Desarrolle una función llamada `dado()` que devuelva un número aleatorio entre uno y seis, todos ellos con la misma probabilidad de ocurrencia.

```
function r = dado()
% dado() : devuelve un numero r al azar entre 1 y 6
% Fórmulas útiles para números aleatorios
% Número entre A y A+1 : A + rand();
% Número entre A y B : A + (B-A) * rand()
% Número entero entre A y B : round(A + (B-A) * rand())
n = round(1 + (6-1) * rand());
end
```

6.9 Ejercicio 9 (nargin, función rand())

Modifique la función *dado()* del ejercicio anterior de manera que en caso de recibir un número *n* como argumento devuelva un vector de *n* componentes cada una de ellas con el resultado de una tirada de dado. Si no recibe argumentos seguirá devolviendo un único número entre 1 y 6. Si el parámetro *n* recibido como argumento no es entero, se redondeará al entero más próximo.

```
function r = dado(n)
% dado() : devuelve un numero r al azar entre 1 y 6
if(nargin == 0)
r = round(1 + (6-1) * rand());
return;
end

if(size(n,1) ~=1 | size(n,2) != 1 | n<=0)
r = round(1 + (6-1) * rand());
return;
end

n = round(n);
r = [];
for ind = 1 : n
r(ind) = round(1 + 5 * rand());
end
end
```

6.10 Ejercicio 10 (Algoritmo de máximo)

Desarrolle una función que reciba un vector *V* de números reales y devuelva un vector de dos componentes *[max, n]*, la primera, *max*, con el valor máximo de las componentes del vector *V* de entrada y la segunda, *n*, con la posición que ocupa el máximo dentro del vector *V*. El problema se resolverá mediante bucles y el algoritmo de máximo.

```
function [max, pos] = maxx(V)
max = V(1);
pos = 1;
for k = 1 : length(V)
if V(k) > max
max = V(k);
pos = k;
end
end
end
```

Note: En el bucle anterior, se comienza en 1 a propósito. Repetimos el conteo del primer elemento, pero la rutina sigue funcionando si se recibe un vector de una sola componente, esto es, un escalar.

6.11 Ejercicio 11 (Diagonal secundaria)

Desarrolle una función que reciba una matriz como argumento y devuelva un vector fila con los elementos de la diagonal secundaria.

```
function dg = diag2(A)
% diag2(A) : Devuelve un vector fila con las componentes de la diagonal secundaria

numfilas = size(A,1);
numcolumnas = size(A,2);

% El primer elemento de la diagonal secundaria es A(1,numcolumnas)
% El segundo es A(2, numcolumnas-1)
% El último es A(numfilas, numcolumnas-numfilas) si numcolumnas > numfilas
% El último es A(numfilas, 1) si numcolumnas <= numfilas
dg = [];
k = 0;
while ((numcolumnas - k) > 0 & (k+1) <= numfilas )
    dg(k+1) = A(k + 1, numcolumnas - k );
    k = k +1;
end
end
```

Otra solución del problema utilizando un bucle *for* en lugar de un bucle *while*:

```
function dg = diag2(A)
% diag2(A) : Devuelve un vector fila con las componentes de la diagonal secundaria

numfilas = size(A,1);
numcolumnas = size(A,2);

dg = [];
for k = 1 : numfilas
    if (numcolumnas - k + 1) == 0
        break;
    end
    dg(k) = A(k, numcolumnas - k +1);
end
end
```

Otra solución, observando que los elementos de la diagonal secundaria siempre suman $1 + \text{numcolumnas}$:

```
function dg = diag23(A)
% diag2(A) : Devuelve un vector fila con las componentes de la diagonal secundaria

numfilas = size(A,1);
numcolumnas = size(A,2);

% Los índices de los elementos de la diagonal secundaria
% siempre suman 1 + numcolumnas

dg = [];
```



```
contador = 0;
for fil = 1 : numfilas
    for col = 1 : numcolumnas
        if (fil + col) == (1 + numcolumnas)
            contador = contador + 1;
            dg(contador) = A(fil, col);
        end
    end
end
end
```

6.12 Ejercicio 12 (Recursividad, Factorial de un número)

Podemos calcular el factorial de un número mediante la función *factorial()*, también a partir de la función *prod(1:n)*. Lo que se pide en este problema es desarrollar una función que calcule el factorial de un número natural mediante recursividad, esto es, mediante una función que se llame a si misma.

```
function fac = recurfac(n)
% Calcula el factorial de un número n mediante recursividad
if(n == 0)
    fac = 1;
else
    fac = n * recurfac(n-1);
end
end
```

Gráficos con Octave

Contents

- *Gráficos con Octave*
 - *Gráficos de curvas en coordenadas polares*
 - *La función plot3()*
 - *La función mesh()*

7.1 Ejercicios resueltos: Gráficos

7.1.1 Ejercicio 1 (Gráfica de líneas)

Se pide realizar un script que dibuje la gráfica de la función $\sin()$ en el intervalo $[-\pi, \pi]$ con línea de color azul de dos puntos de grueso. Se añadirán al gráfico un título y etiquetas para los ejes x e y .

7.1.2 Ejercicio 2 (Representación de curvas con ecuaciones paramétricas)

Las ecuaciones paramétricas de una elipse de semiejes a y b , centrada en el origen de coordenadas son:

$$x = a \cos(t)$$

$$y = b \sin(t)$$

Se pide desarrollar una función que reciba como argumentos los valores de a y de b y que dibuje la gráfica de la elipse con la línea de color rojo y 3 puntos de gruesa.

```
function plotellipse(a,b)
% Dibuja la elipse de semiejes a y b centrada en el origen

t = [-pi:0.1:pi+0.1];
x = a * cos(t);
y = b * sin(t);
plot(x,y,'r', 'linewidth',3)
end
```

7.1.3 Ejercicio 3 (Representación de funciones paramétricas)

Dada la función:

$$x = \cos(at) - \cos^3(bt)$$

$$y = \sin(ct) - \sin^3(dt)$$

Se pide desarrollar una función que reciba como argumentos a , b , c , d y dibuje el gráfico de la función. Pruébese la función para los juegos de valores $[1,80,1,80]$, $[80,1,80,1]$, $[80,1,1,80]$ y $[1,100,1,50]$. Nota: Se debe utilizar un número elevado de puntos para obtener una buena gráfica, del orden de cien mil puntos.

```
function plot1(a,b,c,d)

    t=linspace(-pi,pi,1e6);
    x = cos(a*t) - cos(b*t).^3;
    y = sin(c*t) - sin(d*t).^3;
    plot(x,y)
    title(sprintf('a=%d b=%d c=%d d=%d', a,b,c,d));
end
```

7.1.4 Ejercicio 4 (Polinomios)

Desarrolle una función denominada *polyplot(p)* que reciba un polinomio como argumento y lo plotee en el intervalo comprendido entre la menor de las raíces menos uno y uno más la mayor de las raíces. El título del gráfico será el polinomio desarrollado. Si el polinomio no tiene raíces, el programa escribirá en pantalla un mensaje que muestre el polinomio e informe de la inexistencia de raíces.

```
function polyplot(p)
% Representa el polinomio p en el intervalo comprendido entre sus raíces

    r = roots(p);
    if length(r) == 0
        polyout(p,'X')
        fprintf("No tiene raíces\n")
        return;
    end

    maxr = max(r);
    minr = min(r);

    x = linspace(minr-1, maxr+1,100);
    y = polyval(p, x);
    plot(x,y,"-r", "linewidth", 2);

    hold on;

    y2 = zeros(1,100);
    plot(x,y2);

    title(sprintf('%s',polyout(p,'x')));
    hold off
end
```

7.1.5 Ejercicio

Las ecuaciones paramétricas de un hiperboloide de una hoja son:

$$x = a \cosh(\theta) \cos(\phi)$$

$$y = b \cosh(\theta) \sin(\phi)$$

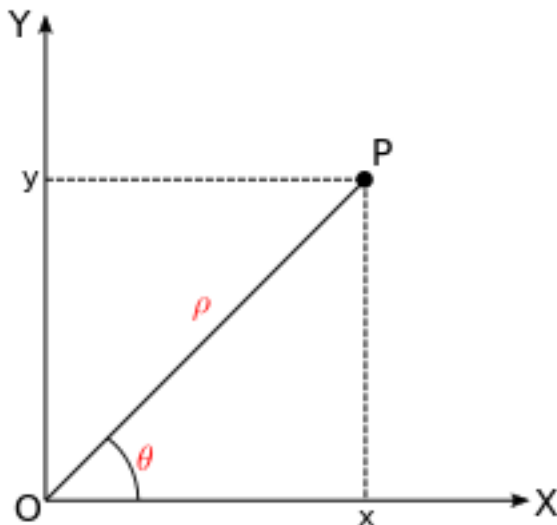
$$z = c \sinh(\theta)$$

Se pide representar la función con $\theta \in \mathbb{R}, 0 < \phi \leq 2\pi$

7.1.6 Solución

7.2 Gráficos de curvas en coordenadas polares

En coordenadas polares, un punto P del plano queda definido mediante el valor ρ de la distancia desde el punto P al origen de coordenadas y el valor del ángulo θ que va desde el eje x , en sentido antihorario, hasta la línea que une el origen de coordenadas con el punto P (ver figura).



Se le llama ecuación polar a la ecuación que define una curva expresada en coordenadas polares. En muchos casos se puede especificar tal ecuación definiendo ρ como una función de θ . La curva resultante consiste en una serie de puntos en la forma $(\rho(\theta), \theta)$. Cuando una función está expresada en coordenadas polares, al origen de coordenadas se le denomina también **polo**.

$$\rho = \rho(\theta)$$

Para transformar las coordenadas polares a coordenadas cartesianas hay que aplicar las siguientes ecuaciones:

$$x = \rho \cos(\theta)$$

$$y = \rho \sin(\theta)$$

Vamos a explicar cómo representar gráficamente con *Octave* una función expresada en coordenadas polares. Utilizaremos para ello la siguiente función:

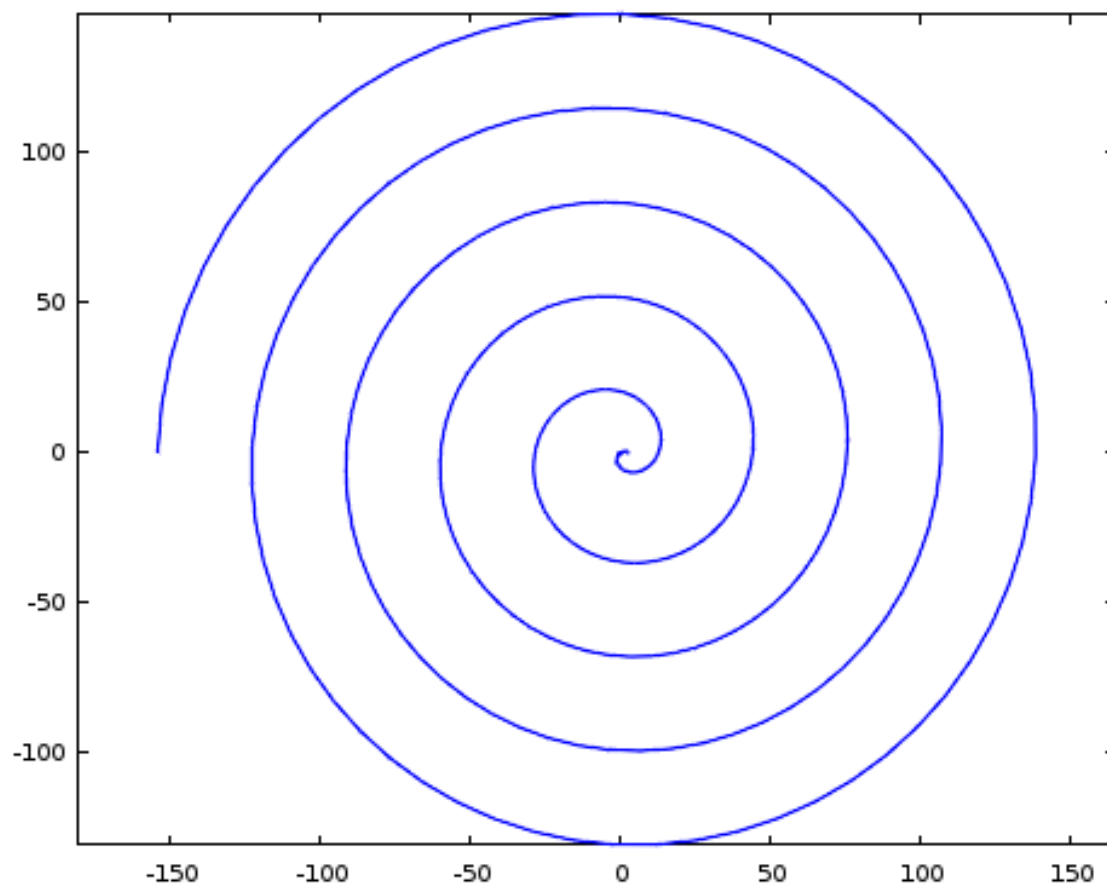
$$\rho(\theta) = 3 - 5\theta \quad \theta \in [0, 10\theta]$$

Lo primero es discretizar el parámetro θ . A continuación se obtiene el vector de valores del radio ρ . Una vez que tenemos los dos vectores con los valores de ρ y θ podemos obtener los vectores con los valores de x e y , haciendo la transformación a coordenadas cartesianas. Finalmente utilizaremos la función `plot()` con los valores de x e y obtenidos. El código *Octave* es el siguiente:

```
% Discretizamos ro y theta
theta = linspace(0, 10*pi, 300);
ro = 3 - 5 .* (theta);

% Transformamos a cartesianas
x = ro .* cos(theta);
y = ro .* sin(theta);

% Ploteamos
plot(x,y)
axis equal
```



Algunos ejemplos de funciones en coordenadas polares son:

Circunferencia: la expresión general de una circunferencia con centro en el punto (ρ_0, θ_0) y radio a es:

$$\rho^2 + 2\rho\rho_0\cos(\theta - \theta_0) + \rho_0^2 = a^2$$

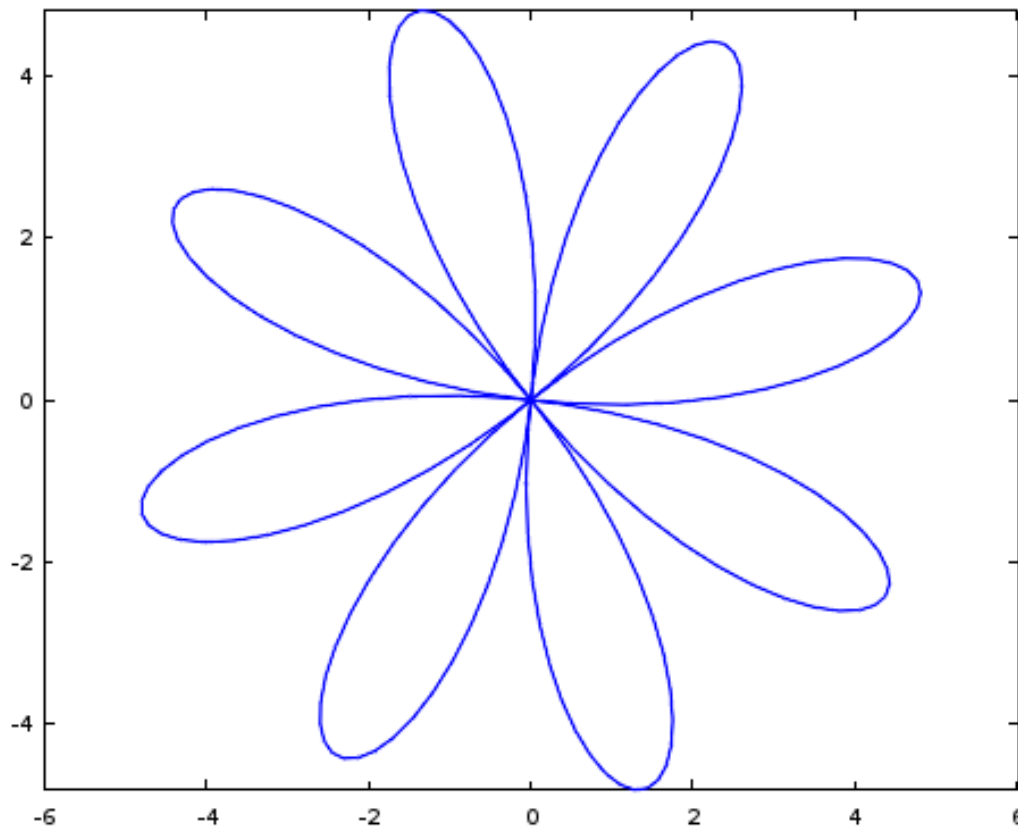
Cuando la circunferencia tiene centro en el origen de coordenadas, la expresión se simplifica:

$$\rho(\theta) = a$$

Rosa polar: es una expresión matemática con forma de flor. Su expresión es:

$$\rho(\theta) = a \cos(k\theta + \theta_0)$$

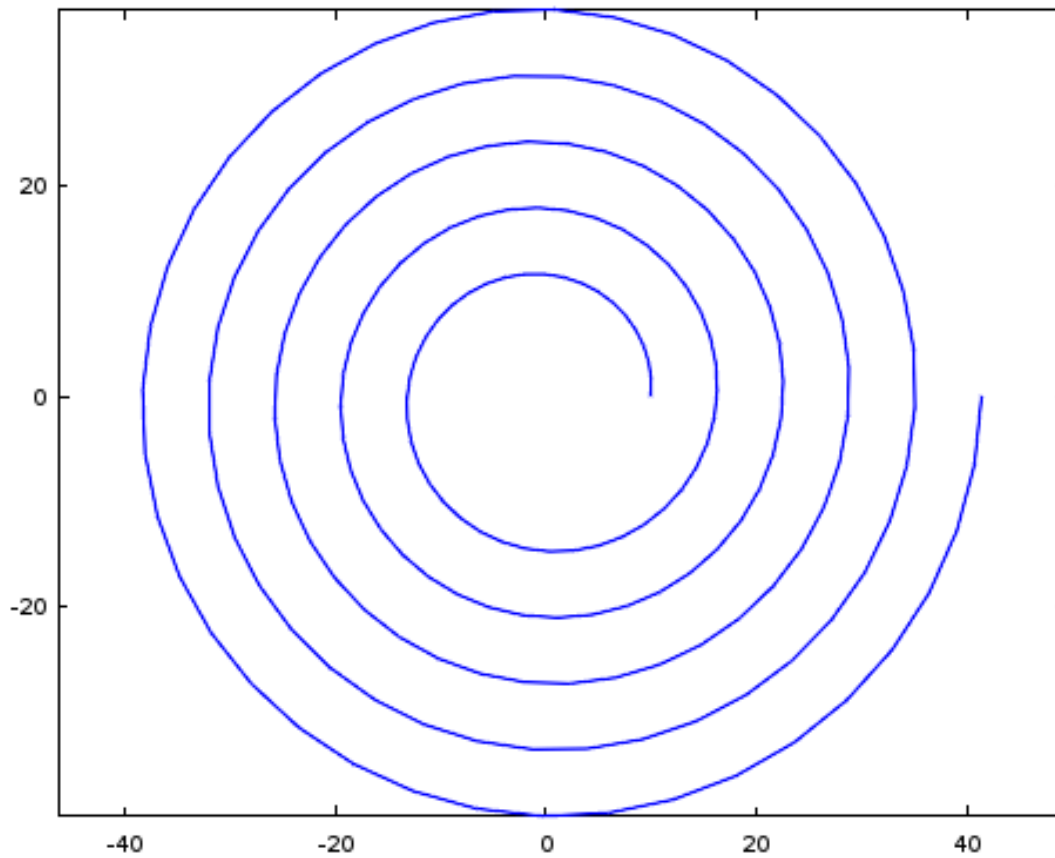
donde θ_0 es cualquier valor constante, incluido el cero. Si k es un valor entero, si es impar, la flor tendrá k pétalos y si k es par, la flor tendrá $2k$ pétalos. Si k es racional pero no entero, la gráfica es similar a una rosa pero con los pétalos solapados. Nótese que estas ecuaciones nunca definen una rosa con 2, 6, 10, 14, etc. pétalos. La variable a representa la longitud de los pétalos de la rosa.



Espiral de Arquímedes: La espiral de Arquímedes es una famosa espiral descubierta por Arquímedes, la cual puede expresarse también como una ecuación polar simple. Se representa con la ecuación:

$$\rho(\theta) = a + b\theta$$

Un cambio en el parámetro a producirá un giro en la espiral, mientras que b controla la distancia entre los brazos, la cual es constante para una espiral dada. La espiral de Arquímedes tiene dos brazos, uno para $\theta > 0$ y otro para $\theta < 0$. Los dos brazos están conectados en el polo. La imagen especular de un brazo sobre el eje vertical produce el otro brazo. Esta curva fue una de las primeras curvas, después de las secciones cónicas, en ser descritas en tratados matemáticos. Además es el principal ejemplo de curva que puede representarse de forma más fácil con una ecuación polar.



Elipse: la expresión polar de una elipse centrada en el origen de coordenadas y de semiejes a y b es:

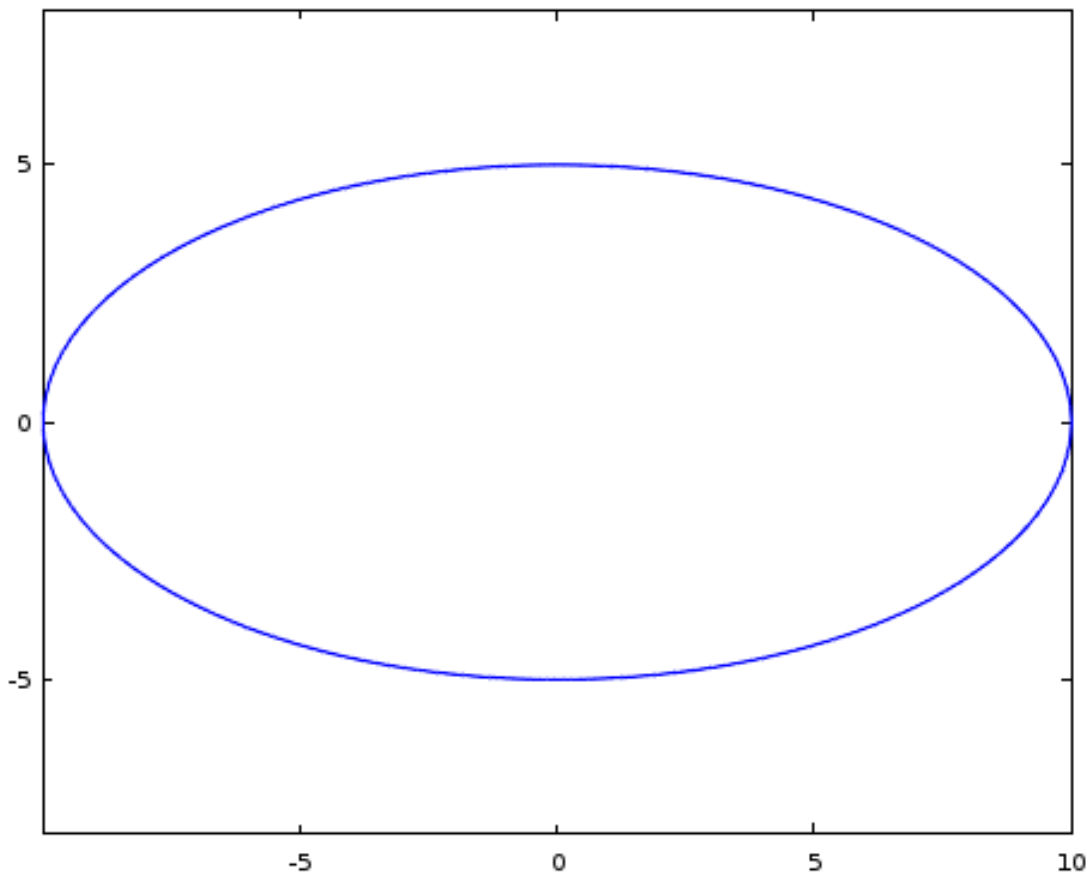
$$\rho(\theta) = \frac{1}{\sqrt{\frac{\cos^2(\theta)}{a^2} + \frac{\sin^2(\theta)}{b^2}}} \quad \theta \in [0, 2\pi]$$

Se define la excentricidad e de una elipse mediante la siguiente expresión:

$$e = \sqrt{1 - \frac{b^2}{a^2}}$$

La ecuación polar de la elipse en función de su excentricidad queda de la siguiente manera:

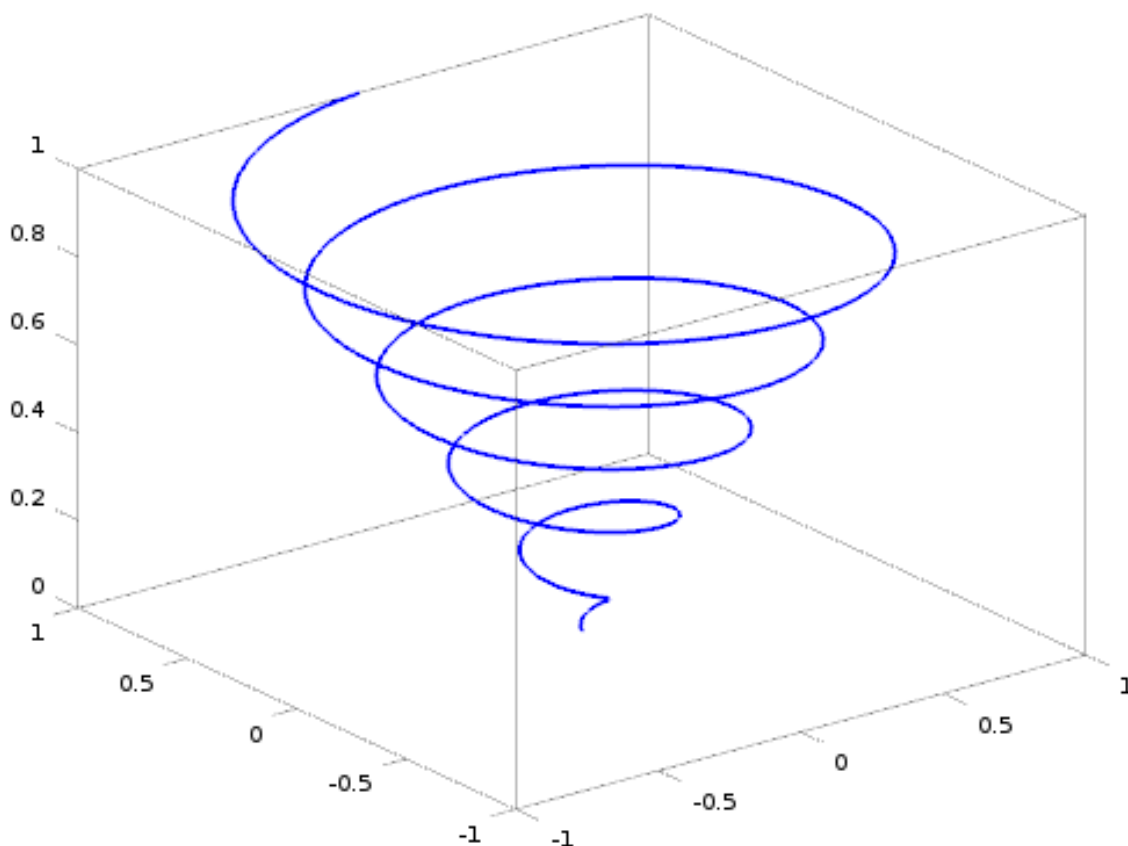
$$\rho(\theta) = \frac{b}{\sqrt{1 - e^2 \cos^2(\theta)}}$$



7.3 La función *plot3()*

- **plot3()**: La función *plot3()* dibuja curvas en tres dimensiones, o ternas de valores arbitrarios sin necesidad de que formen una superficie. Por ejemplo, el código siguiente dibuja una espiral en tres dimensiones:

```
t = 0:0.1:10*pi;  
r = linspace (0, 1, length(t));  
z = linspace (0, 1, length(t));  
plot3 (r.*sin(t), r.*cos(t), z, 'linewidth',2);
```



- **view(*az*, *el*)**: Situa el punto de vista en el azimut *az* y la elevación *el*. Haciendo *view(2)* o *view(3)* establece los valores por defecto para dos y tres dimensiones.

7.4 La función *mesh()*

La función *mesh()* dibuja un gráfico de malla en la forma $Z=f(X,Y)$, tomando la *Z* como elevación y haciendo el color del mallado proporcional al valor de *Z*. Si *Z* es una matriz cuyas dimensiones son $[m, n]$. El vector de las coordenadas *x* tendrá que tener *m* elementos, y el vector de las componentes *y* tendrá *n* elementos.

Hay varias formas de llamar a la función *mesh()*:

- **mesh(*Z*)**: dibuja el gráfico de malla 3D correspondiente a $Z=Z(x,y)$ haciendo $x=1:m$ e $y=1:n$ con $[m, n] = \text{size}(Z)$. El color es proporcional al valor de *Z*.
- **mesh(*X*, *Y*, *Z*)**: dibuja el gráfico de malla 3D con el color determinado por el valor de *Z*. Si *X* e *Y* son vectores las dimensiones tienen que cumplir: $m=\text{length}(X)$, $n=\text{length}(y)$ y $[m, n]=\text{size}(Z)$. En este caso, los puntos de la superficie serán $(X(j), Y(i), Z(i,j))$. Si *X*, *Y*, *Z* son matrices, sus dimensiones serán iguales. En este caso los puntos del mallado serán: $(X(i,j), Y(i,j), Z(i,j))$
- **meshgrid(*x*,*y*)**: Dados dos vectores de coordenadas *x* e *y*, la función *meshgrid()* devuelve una pareja de matrices *Mx*, *My* que se corresponden con la red de puntos de intersección de las dos series *x* e *y*. Estas matrices se pueden pasar directamente a la función *mesh()* y también se pueden utilizar para calcular los valores de *Z* si conocemos su expresión en la forma $Z=f(X,Y)$.

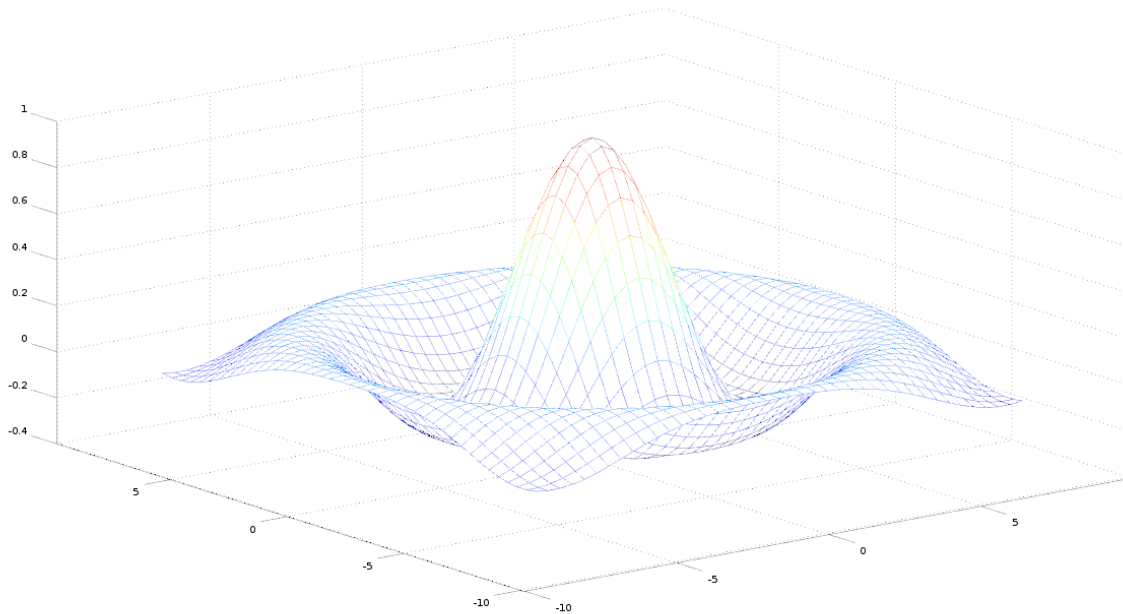
Si mallamos los vectores $x=[0 \ 1 \ 2 \ 3]$ e $y=[0 \ 1 \ 2 \ 3]$ con la función $[xx, yy]=\text{meshgrid}(x,y)$, el resultado será el siguiente:

```
xx =
    0  1  2  3
    0  1  2  3
    0  1  2  3
    0  1  2  3
yy =
    0  0  0  0
    1  1  1  1
    2  2  2  2
    3  3  3  3
```

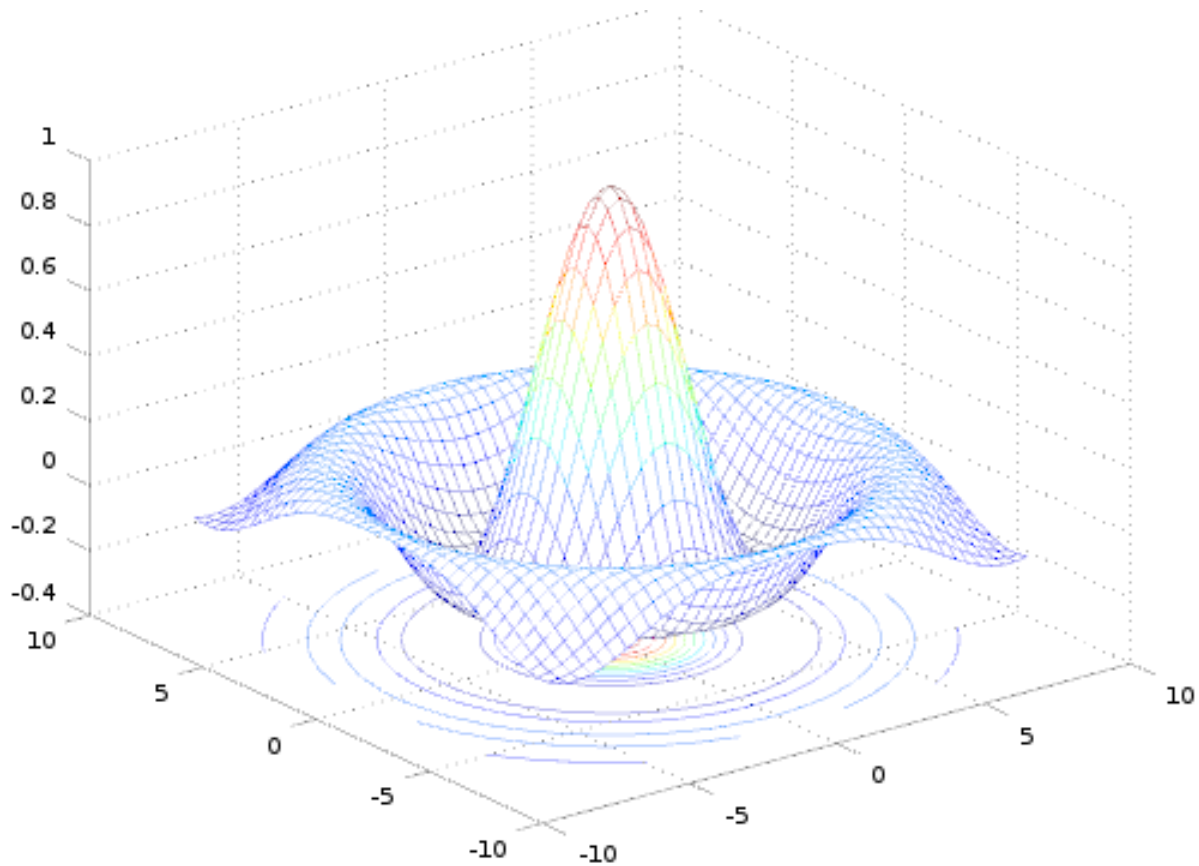
Por ejemplo:

```
tx = linspace (-8, 8, 41)';
ty = linspace (-8, 8, 41)';
[xx, yy] = meshgrid (tx, ty);
r = sqrt (xx .^ 2 + yy .^ 2) + eps;
tz = sin (r) ./ r;
mesh (tx, ty, tz);
```

Produce el gráfico del conocido *sombrero* que se ve en la figura:



- **meshc()**: Es similar a *mesh()* pero dibuja además las curvas de nivel sobre el plano *x-y*: El resultado se puede ver en la siguiente figura:



Para representar una superficie expresada en ecuaciones paramétricas se procede de la siguiente forma. Supongamos que las ecuaciones de la superficie expresada en función de dos parámetros son las siguientes:

$$x = f_1(\theta, \phi)$$

$$y = f_2(\theta, \phi)$$

$$z = f_3(\theta, \phi)$$

Generaremos unos rangos para el valor de cada parámetro y utilizaremos a continuación la función *meshgrid()* para generar las matrices Θ , Π con los mallados de valores de los parámetros. A continuación podemos calcular las matrices X , Y , Z que servirán para pasarle a la función *mesh()*.

Veamos un ejemplo. Sea el paraboloide elíptico definido por las siguientes ecuaciones:

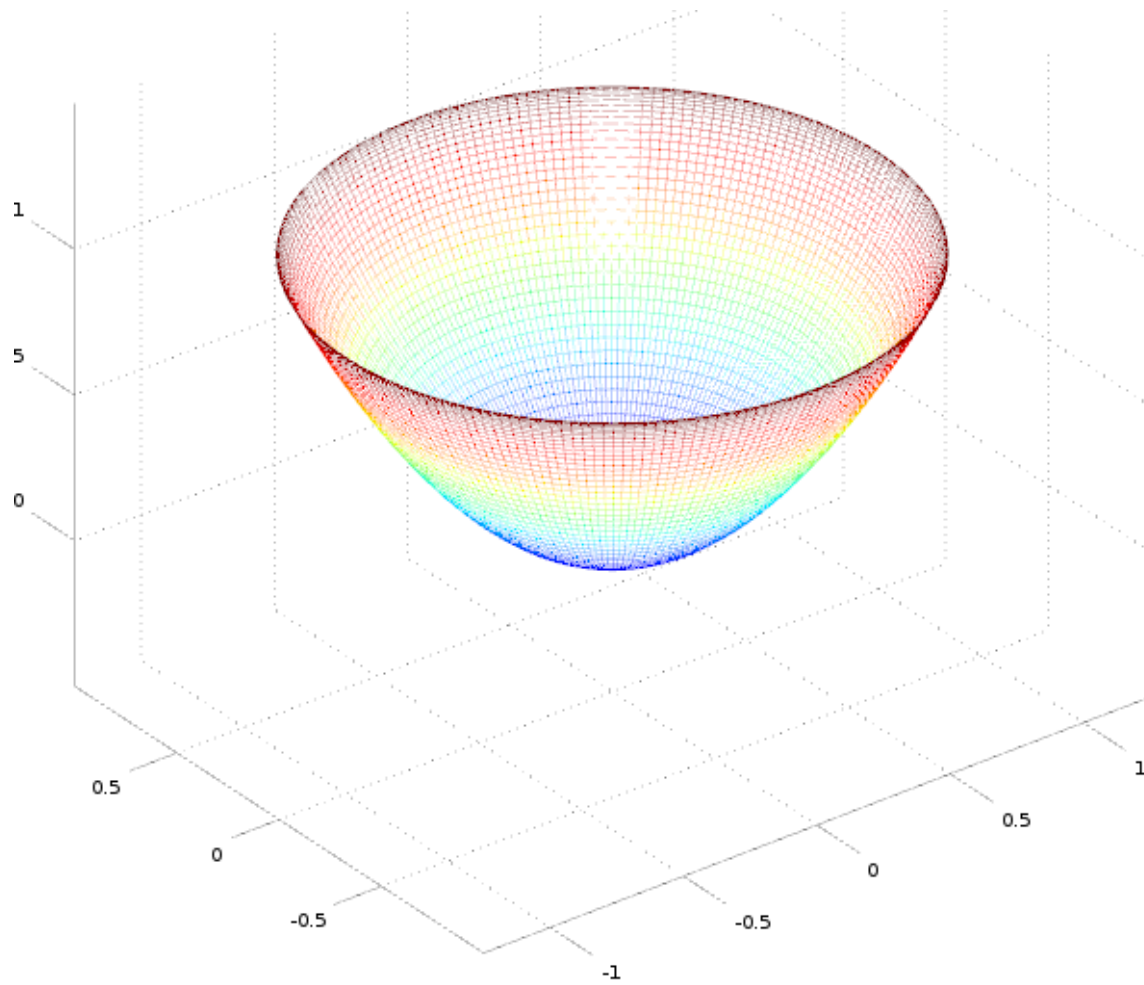
$$x(t, u) = a \cos(t) \cos(u)$$

$$y(t, u) = b \cos(t) \sin(u)$$

$$z(t, u) = (\cos(t))^2$$

Se pide representarlo para $a = 1, b = 1, 0 < t < 2\pi, 0 < u < 2\pi$

El resultado se puede ver en la siguiente figura:



Lectura y escritura de ficheros de texto

Contents

- *Lectura y escritura de ficheros de texto*
 - *Comandos utilitarios trabajando con ficheros*
 - *Los comandos **load** y **save***
 - *La función **fopen()***
 - *La función **fclose()***
 - *Lectura de ficheros línea a línea*
 - *Lectura de datos formateados*
 - *Escritura de datos formateados*

8.1 Ejercicios Lectura y escritura de ficheros

Note: Los ficheros de datos necesarios para la realización de los ejercicios se pueden descargar desde la plataforma moodle de la asignatura.

Note: Se denominan ficheros **CSV**, (*Comma Separated Values*), a los ficheros de texto formados por filas de datos, separados entre sí por una ‘,’ (*coma*). Cada fila está separada de la siguiente por un salto de línea, carácter ‘**\n**’. Está permitido utilizar de separador otro caracter en lugar de la coma. También está permitido que el fichero tenga una o más líneas de encabezamiento. Este tipo de ficheros pueden ser leídos directamente por las hojas de cálculo.

8.1.1 Ejercicio 1: Escribir un fichero CSV

Desarrollar una función que escriba una matriz de *doubles* en un fichero CSV. La función recibirá dos parámetros: una cadena de texto con el nombre del fichero que se quiere escribir, y una matriz de *doubles* con los datos a escribir en el fichero. La función devolverá un valor *logical true* si la grabación se realiza correctamente y un *false* en caso de que se produzcan errores durante la grabación. Los números se escribirán en el fichero utilizando el ‘.’ (punto) como separador de decimales y la ‘,’ (coma) como separador de datos dentro de cada fila.

8.1.2 Ejercicio 2: Leer un fichero CSV

Desarrollar una función que permita leer el contenido de un fichero CSV escrito por la función del ejercicio anterior. El fichero tendrá los valores de una matriz de *doubles* separados por *coma*, utilizando el *punto* como separador de decimales. La función devolverá la matriz de *doubles* si el fichero se leyó adecuadamente o una matriz vacía en el caso de que no sea posible leer el fichero.

8.1.3 Ejercicio 3: Leer ficheros CSV

El fichero `'slopesProfile_5m.csv'` contiene la abscisa s y la pendiente p de puntos cada cinco metros de un tramo de la carretera M-607 de Madrid. Se pide desarrollar una función que lea el fichero, almacenando los valores leídos en sendos vectores s y p . La función calculará e imprimirá en pantalla: (1) el número de puntos leídos, (2) la longitud del tramo y la pendiente media del tramo. En el caso de que la función no pudiera leer el fichero, mostrará un mensaje en pantalla informando del error, y terminará la ejecución.

8.1.4 Ejercicio 4: Leer ficheros CSV

Desarrollar una función que lea el fichero `'slopesProfile_5m.csv'` y dibuje el gráfico de pendientes.

8.1.5 Ejercicio 5: Contador de líneas

Desarrolle una función de nombre `'lcount()'` que reciba como parametro el nombre de un fichero y devuelva el número de líneas que tiene el fichero.

8.1.6 Ejercicio 6: Escribir ficheros CSV

El perfil longitudinal de cierto tramo de carretera se rige por las siguientes ecuaciones:

$$5920.93 \leq x \leq 6100.06 \quad Z = 808.75791 + 0.013 * X \quad (8.1)$$

$$6100.06 < x \leq 6231.94 \quad Z = -1425.760105 + 0.745638 * X - 0.000060 * X^2 \quad (8.2)$$

$$6231.94 < x \leq 6297.27 \quad Z = 908.722208 + -0.0032 * X \quad (8.3)$$

Se pide desarrollar una función que reciba como parámetros el nombre del fichero y el espaciado de puntos deseado y escriba el fichero CSV correspondiente a los puntos (S_i, Z_i) del perfil espaciados según el parámetro recibido.

8.1.7 Ejercicio 7: Leer ficheros CSV

Desarrollar una función que lea los ficheros generados por la función del ejercicio anterior y dibuje el gráfico de perfil correspondiente.

Note: La documentación de Octave y de Matlab correspondiente a este capítulo se puede encontrar en los siguientes enlaces:

- **Octave:**
 - **Matlab:**
-

8.2 Comandos utilitarios trabajando con ficheros

Cuando estamos trabajando con ficheros de texto hay varios comandos de *Octave* que conviene conocer.

El comando **type** permite mostrar el contenido de un fichero de texto en la consola. La forma de utilizar el comando es la siguiente: **type filename**, siendo *filename* el nombre de un fichero existente. Si el nombre del fichero incluye su ruta de directorio, se buscará el fichero en la ruta especificada. Si solo se incluye el nombre del fichero, se buscará en el directorio de trabajo.

El directorio de trabajo en el que estamos posicionados se puede consultar con **pwd()**, que muestra en pantalla la ruta del directorio de trabajo donde estamos posicionados.

El contenido del directorio de trabajo se puede mostrar mediante el comando **dir**, al estilo windows, o mediante el comando **ls**, al estilo linux. Los dos comando admiten una *mascara* que permite especificar el tipo de fichero buscados. Por ejemplo podríamos utilizar *dir *.m* para mostrar un listado de los ficheros con extensión *.m*. En la máscara, el asterisco hace la función de comodín. Podríamos leer el comando anterior como: '*dir lo-que-sea punto m*'.

También podemos cambiar el directorio de trabajo desde la consola de *Octave*. Para ello se utiliza el comando **cd**, '*change directory*', que admite un parámetro a continuación para especificar la ruta a la que se quiere acceder. Hay un atajo que hay que conocer. Si tecleamos **cd ..**, *cd punto punto*, nos moveremos al directorio padre del directorio en el que estemos posicionados. Para acceder a un determinado directorio utilizaremos: **cd dirname**, donde *dirname* es el nombre o la ruta completa del directorio en el que queremos posicionarnos.

Los comandos anteriores admiten ser utilizados en forma de función.

8.3 Los comandos load y save

Podemos guardar variables en ficheros de texto mediante el comando *Octave* **save**. Las variables guardadas las podremos recuperar luego mediante el comando **load**. El comando *save* se puede utilizar de distintas maneras:

- **save filename**: Guarda todas las variables del *workspace* en el fichero *filename*, donde *filename* es el nombre del fichero y opcionalmente su ruta completa.
- **save filename v1 v2**: Se pueden especificar qué variables concretas queremos guardar en el fichero. En este ejemplo, solo se guardarán en el fichero *filename* las variables de nombre *v1* y *v2*.
- **save options filename v1 v2**: donde *options* son una o más opciones separadas por espacio. Las opciones nos permiten especificar, por ejemplo, que se guarden los ficheros en distintos formatos binarios o que se guarden los datos en formato comprimido *zip*.

La forma de operar se puede ver en el siguiente ejemplo, donde primero definimos una matriz y a continuación la guardamos en un fichero '*matriz.txt*'. Por último se muestra el contenido del fichero mediante *type()*:

8.4 La función fopen()

Para poder realizar operaciones de lectura y/o escritura en los ficheros de texto es necesario primero abrir el fichero con **fopen()**. Una vez abierto el fichero, se procede a realizar las distintas operaciones de lectura y/o de escritura en el mismo. Cuando se termina de operar con el fichero hay que cerrarlo con la función **fclose()**.

El procedimiento habitual para leer datos de un fichero se resume en el siguiente esquema:

```
file = fopen('mifichero.txt', 'r');

% Operaciones de lectura del fichero

fclose(fid);
```

La función **fopen()** tiene la siguiente signatura:

- **file = fopen(filename, permission)** Abre el fichero de nombre *filename* con el tipo de acceso especificado por *permission*. Devuelve *file*, que es un número, un *puntero*, que identifica al fichero en posteriores operaciones de lectura escritura. Si se producen errores al intentar abrir el fichero, la función *fopen()* devuelve *-1*. El puntero será un número mayor o igual a 3. Los parametros son los siguientes:
 - *filename*: cadena de texto con el nombre del fichero que se quiere abrir. Si incluye la ruta del fichero se utilizará, si no se buscará o creará el fichero en el directorio de trabajo.
 - *permission*: especifica el modo de apertura del fichero. Lo habitual es abrir el fichero para escribir en él o para leer de él, si bien es también posible abrir el fichero en modo *lectura y escritura*. En los modos de escritura es posible añadir texto a un fichero existente o bien crear uno nuevo para la operación, sobrescribiendo el existente si lo hubiera. *permission* es una cadena de texto que puede tener los siguientes valores:
 - * 'r': Abre el fichero para lectura. Es el modo por defecto, si se utiliza la función *fopen()* sin el argumento *permission*
 - * 'r+': Abre el fichero en modo *lectura-escritura*
 - * 'w': Abre o crea un nuevo fichero en modo *escritura*. Si existe se sobrescribe
 - * 'w+': Abre o crea un fichero para *lectura-escritura*. Si existe, se sobrescribe
 - * 'a': Abre o crea un nuevo fichero para *escritura*. Si existe el fichero, añade al final del mismo.
 - * 'a+': Abre o crea un nuevo fichero para *lectura-escritura*. Si existe, se añade al final del mismo.

Note: Hay dos modos más de apertura de ficheros: 'A' y 'W' que no realizan el *flush* automático de los datos.

Note: La función *fopen()* admite un tercer y cuarto parámetros que permiten especificar el orden de lectura escritura de los bytes (Big-endian, Little-endian) y la codificación de caracteres del fichero (UTF-8, ISO-8859-1,...).

Note: **fids = fopen('All')**: Devuelve un vector con los punteros a TODOS los ficheros abiertos que haya.

8.5 La función fclose()

Sirve para cerrar los ficheros cuando se ha terminado de trabajar con ellos.

- **fclose(file)**: Cierra el fichero *file*. Devuelve 0 si pudo cerrar el fichero o *-1* si hay errores.
- **fclose('All')**: Cierra todos los ficheros que haya abiertos. Devuelve 0 si pudieron cerrar los ficheros o *-1* si hay errores.

8.6 Lectura de ficheros línea a línea

Una vez abierto el fichero para lectura podemos ir leyendo una línea cada vez, que recibiremos como una cadena de texto, con o sin carácter fin de línea, según la función de lectura utilizada:

- **line = fgets(file)**: Lee una línea del fichero *file*, incluyendo el caracter de fin de línea, y la devuelve en forma de cadena de texto.

- **line = fgets(file, nchar):** Lee al menos *nchar* caracteres de la siguiente línea del fichero *file*. (Si se alcanza el final de línea o el final del fichero devuelve lo que haya leído hasta ahí).
- **line = fgetl(file):** Lee la siguiente línea del fichero *file*, sin incluir el caracter fin de línea, y la devuelve como una cadena de caracteres.

8.7 Lectura de datos formateados

Octave ofrece la posibilidad de leer los datos contenidos en el fichero, uno a uno, y devolverlos en el formato seleccionado, pudiendo de esta manera leer, no solo cadenas de texto, sino también números enteros o doubles.

La función para leer datos formateados es **fscanf()**. La función *fscanf()* admite varias signatures. Una forma habitual de utilizar la función es la siguiente:

fscanf()

8.8 Escritura de datos formateados

En el ejemplo siguiente creamos una matriz *A* de tres filas y cuatro columnas. A continuación la escribimos en un fichero con una simple indicación de formato numérico. Cuando hacemos *type* al fichero recién creado podemos ver la forma en que *Octave* ejecuta la función *fprintf()*.

```
octave:44> A = [11 12 13 14; 21 22 23 24; 31 32 33 34]
A =

    11    12    13    14
    21    22    23    24
    31    32    33    34

octave:45> file = fopen('prueba.txt','w')
file = 14
octave:46> fprintf(file, '%d ', A)
octave:47> fclose(file)
ans = 0
octave:48> type prueba.txt
11 21 31 12 22 32 13 23 33 14 24 34
octave:49> 
```

Como vemos la sentencia *fprintf(file, '%d ', A)* escribe todos los elementos de la matriz *A* en el fichero, utilizando el formato indicado: *numero entero más espacio*. Hay que fijarse en el orden en que Octave coge los elementos de *A* para escribirlos en el fichero. Vemos que coge los elementos de la matriz por columnas, esto es, primero el $A(1,1)$, a continuación el $A(2,1)$, y así hasta agotar la primera columna, leyendo a continuación las siguientes columnas de la misma manera hasta agotar la matriz.

Lo que hace Octave es leer las matrices en el orden el que las tiene guardadas en memoria, y *Octave* guarda las matrices en memoria por columnas.

Si en lugar del formato especificado en el ejemplo anterior, ejecutamos la sentencia `fprintf(file, '%d %d %d %d\n', A)`, en la que decimos a *Octave* que cada cuatro números inserte un carácter fin de línea `\n`, *Octave* escribirá tres filas y cuatro columnas, pero no en el orden de filas y columnas en el que tenemos definida la matriz:

```
octave:55> file = fopen('prueba.txt','w')
file = 14
octave:56> fprintf(file, '%d %d %d %d\n', A)
octave:57> fclose(file)
ans = 0
octave:58> type prueba.txt
11 21 31 12
22 32 13 23
33 14 24 34

octave:59> █
```

Una vez más *Octave* rellena el formato indicado cogiendo los elementos de la matriz por columnas.

9.1 Utilización de paquetes adicionales

Octave-Forge [1] es el repositorio donde se centraliza el desarrollo colaborativo de paquetes de funcionalidades adicionales para *Octave*.

Los paquetes de *Octave-Forge* amplían la funcionalidades estándar de *Octave*, añadiendo utilidades y funciones específicas para distintos ámbitos de aplicación. Todos los paquetes de *Octave-Forge* se pueden instalar mediante el mecanismo de instalación de paquetes incorporado en el propio *Octave*. Para instalar un paquete, desde la consola de *Octave*, teclear:

```
pkg install -forge package_name
```

donde *package-name* es el nombre del paquete a instalar. *Octave* descargará e instalará el paquete, quedando las funciones contenidas en el mismo disponibles para su utilización.

Hay multitud de paquetes con funcionalidades adicionales que se pueden incorporar a la instalación de *Octave*. La lista de paquetes disponibles se puede consultar en [2]. Algunos ejemplos de paquetes adicionales que podemos instalar:

- *image*: procesamiento de imágenes
- *io*: funciones de entrada-salida en distintos formatos: *csv*, *xml*, *xls*, *json* y otros.
- *java*: interface para utilizar funciones *java*
- *mechanics*: incluye, entre otras, funciones utilitarias para cálculo de estructuras
- *octproj*: transformaciones entre proyecciones cartográficas
- *splines*: funciones para manipulación de splines

Podemos comprobar la lista de paquetes que tenemos instalados haciendo:

```
pkg list
```

Además de la lista de paquetes anterior, hay una lista adicional de paquetes no incluidos en el sistema de paquetes *pkg*, pero cuyo código fuente se puede consultar y descargar en [3].

[1] Octave-Forge: <http://octave.sourceforge.net/index.html>

[2] Octave-Forge: Lista de paquetes: <http://octave.sourceforge.net/packages.php>

[3] Paquetes adicionales: http://sourceforge.net/p/octave/_list/hg

9.2 Utilidades para interactuar con el Sistema Operativo

Podemos ejecutar comandos del sistema operativo mediante la función **system(cmd, flag)**. El primer argumento será una cadena de caracteres con el comando a ejecutar. El segundo parámetro es opcional, y si está presente, la salida del comando se devolverá como una cadena de texto. Si este segundo parámetro no está presente la salida del comando se enviará a la salida estándar. El valor de este segundo parámetro es cualquiera, por ejemplo un `'1'`.

Podemos recuperar el valor de una variable del sistema mediante la función **getenv('var_name')**, por ejemplo, en Linux, para recuperar el *path*:

```
octave:20> getenv('PATH')
ans = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/u
sr/local/games:/home/shiguera/apps:/usr/lib/jvm/jdk1.6.0_38/bin:/home/shiguera/
apps/apache-maven-3.1.1/bin:/home/shiguera/development/android-sdk-linux/tools:/
home/shiguera/development/android-sdk-linux/platform-tools:/opt/lampp/bin:/home/
shiguera/apps/osmosis-0.40/bin:/usr/lib/x86_64-linux-gnu/octave/3.8.1/site/exec/
x86_64-pc-linux-gnu:/usr/lib/x86_64-linux-gnu/octave/api-v49+/site/exec/x86_64-p
c-linux-gnu:/usr/lib/x86_64-linux-gnu/octave/site/exec/x86_64-pc-linux-gnu:/usr/
lib/x86_64-linux-gnu/octave/3.8.1/exec/x86_64-pc-linux-gnu:/usr/bin
octave:21> █
```

Podemos conocer el directorio actual mediante **pwd()**, cambiar de directorio mediante **cd <dir_name>**, listar el contenido de un directorio mediante **dir** o **ls** y crear un nuevo directorio mediante **mkdir <di_name>**.

La función **computer()** nos informará del ordenador en el que se está ejecutando *Octave*.

Personalización del arranque de Octave

Octave busca, y ejecuta si existen, varios ficheros antes del arranque. Estos ficheros pueden contener cualquier comando válido de *Octave*, incluidas deefiniciones de funciones. Siendo *octave-home* el directorio de instalación de *Octave*, y *user-directory* el directorio del usuario, los ficheros que Octave ejecutará en el arranque serán:

- *octave-home/share/octave/site/m/startup/octaverc* : Afecta a todos los usuarios del ordenador y todas las versiones instaladas de *Octave*
- *octave-home/share/octave/version/m/startup/octaverc*: Afecta a todos los usuarios de una determinada versión de *Octave*
- *user-directory/.octaverc* : Afecta al **Octave* ejecutado por un determinado usuario.
- *.octaverc* : Al arrancar *Octave* desde un determinado directorio que contenga un fichero *.octaverc* se ejecutará dicho fichero, con lo que podemos personalizar el arranque de un determinado proyecto.

La función **dump_prefs()** se puede utilizar para ver las personalizaciones que se están utilizando en una determinada sesión.

Tortugas y fractales

Las gráficas tortuga fueron añadidas al lenguaje de programación Logo por Seymour Papert a finales de la década de 1960 para apoyar la versión de Papert del robot tortuga, un simple robot controlado desde el puesto de trabajo del usuario diseñado para llevar a cabo funciones de dibujo asignadas mediante una pequeña pluma retráctil en su interior o adjuntada al cuerpo del robot.

La tortuga tiene tres atributos:

- Una posición
- Una orientación
- Una pluma, teniendo atributos como color, ancho y arriba y abajo.

La tortuga se mueve con comandos relativos a su posición, como «avanza 10 » y «gira a la izquierda 90 »

Las gráficas tortuga se adaptan bien al dibujo de fractales.

Vamos a explicar cómo construir una tortuga para *Octave* o *Matlab* y luego vamos a desarrollar algún ejemplo clásico de dibujo de fractales.

Lo primero es construir la tortuga. Nuestra tortuga admitirá diferentes comandos identificados cada uno de ellos por una cadena de caracteres. En principio vamos a utilizar dos comandos: 'AVANZA' y 'GIRA'. En ambos casos nos hace falta un valor que defina la magnitud del avance o del giro de la tortuga. La función *turtle()* admitirá, por tanto dos parámetros, quedando su signatura definida de la siguiente manera:

turtle(command, value)

La versión *clásica* de la tortuga de *Papert* admitía dos comandos para el giro: *Gira izquierda* y *Gira derecha*. Nosotros vamos a utilizar un único comando *GIRA*, distinguiendo el sentido de giro mediante el signo del *value*.

La arquitectura de la función *turtle()* estará organizada mediante una función principal, que distribuye los comandos recibidos, y una serie de funciones auxiliares para cada comando. La función principal se encarga de interpretar el comando recibido y pasarlo a la función auxiliar correspondiente que es la que realmente ejecutará el comando. El primer listado de nuestra función *turtle()* podría ser el siguiente:

```
function turtle(command, value)
% Tortuga básica para gráficos

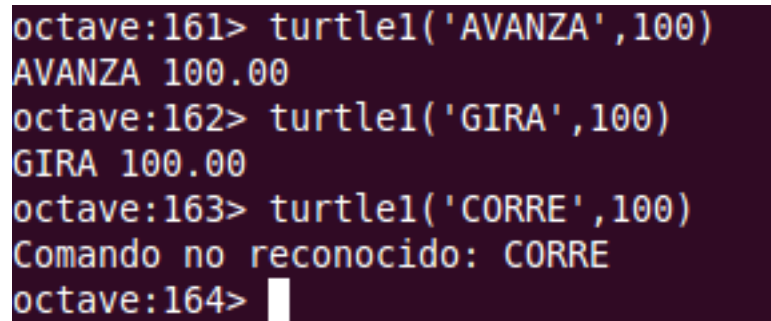
    if strcmp(command, 'GIRA')
        gira(value);
    elseif strcmp(command, 'AVANZA')
        avanza(value);
    else
        dispError(command);
    end
end
```

```
function gira(value)
    fprintf('GIRA %f\n', value)
end

function avanza(value)
    fprintf('AVANZA %f\n', value)
end

function dispError(command)
    fprintf('Comando no reconocido: %s\n', command)
end
```

Podemos probar la función *turtle()* con comandos válidos y no válidos. Por supuesto la tortuga todavía no dibuja nada, lo único que hace es imprimir un mensaje con el comando recibido, de forma que podamos comprobar que la arquitectura *función principal* y *subfunciones* funciona adecuadamente. Se trata solo del *esqueleto* que nos permitirá ir dotando de funcionalidad a la tortuga.



```
octave:161> turtle1('AVANZA',100)
AVANZA 100.00
octave:162> turtle1('GIRA',100)
GIRA 100.00
octave:163> turtle1('CORRE',100)
Comando no reconocido: CORRE
octave:164> 
```

Inicialmente nuestra tortuga mantendrá los valores correspondientes a la posición y la orientación. La tortuga necesita mantener el valor de sus variables de estatus entre una ejecución y otra. Para ello utilizaremos unas variables globales llamadas *x*, *y* y *angulo*. La función *turtle()* se limita a declarar dichas variables como globales. Será el programa principal que utilice la función *turtle()* el encargado de definir las.

Vamos a añadir dos comando utilitarios adicionales: el comando *INIT* y el comando *DISP*. El comando *INIT* se encargará de inicializar las variables de estatus, poniendo a cero los valores de *x*, *y* y *angulo*. El comando *DISP*, por su parte, mostrará en pantalla el valor actual de las variables de estatus de la tortuga.

Con estas nuevas consideraciones, el listado de nuestra tortuga quedará de la siguiente manera:

```
function turtle(command, value)
% Tortuga básica para gráficos
% Necesita la existencia de unas variables globales x, y, angulo

    global x y angulo;

    if strcmp(command, 'GIRA')
        gira(value);
    elseif strcmp(command, 'AVANZA')
        avanza(value);
    elseif strcmp(command, 'DISP')
        disp();
    elseif strcmp(command, 'INIT')
        init();
    else
        dispError(command);
    end
end
```

```

function gira(value)
    fprintf('GIRA %f\n', value)
end

function avanza(value)
    fprintf('AVANZA %f\n', value)
end

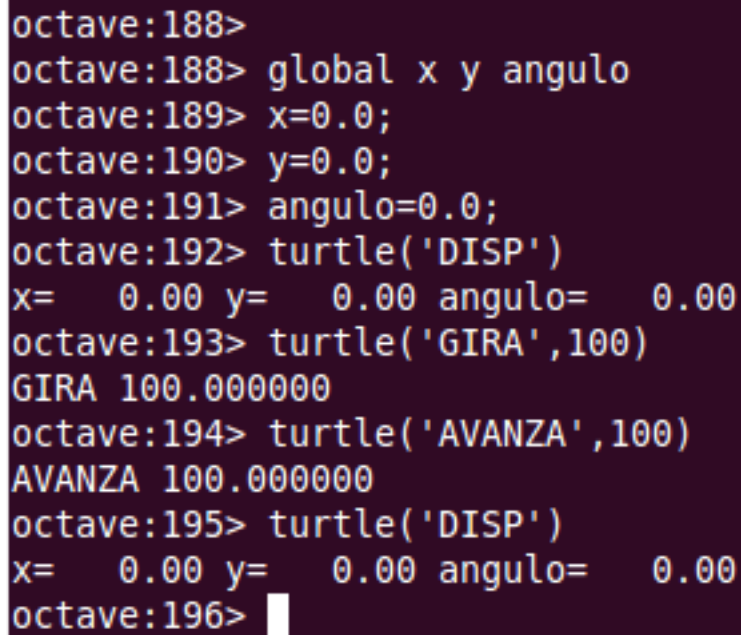
function disp()
    global x y angulo;
    fprintf('x= %6.2f y= %6.2f angulo= %6.2f\n', x, y, angulo)
end

function init()
    global x y angulo;
    x=0.0; y=0.0; z=0.0;
    disp();
end

function dispError(command)
    fprintf('Comando no reconocido: %s\n', command)
end

```

Si tratamos de ejecutar la función *turtle()* desde consola nos dará un error, a menos que declaremos como globales y asignemos un valor inicial a las variables *x*, *y* y *angulo*. Podemos ver la secuencia de comandos en la siguiente imagen:



```

octave:188>
octave:188> global x y angulo
octave:189> x=0.0;
octave:190> y=0.0;
octave:191> angulo=0.0;
octave:192> turtle('DISP')
x=  0.00 y=  0.00 angulo=  0.00
octave:193> turtle('GIRA',100)
GIRA 100.000000
octave:194> turtle('AVANZA',100)
AVANZA 100.000000
octave:195> turtle('DISP')
x=  0.00 y=  0.00 angulo=  0.00
octave:196>

```

Primero definimos desde consola las variables *x*, *y*, *angulo* y luego llamamos a la función *turtle()* varias veces con distintos comandos. La última ejecución del comando *DISP* muestra ceros en las variables de estatus, a pesar de haber realizado un giro y un avance. Esto es debido a que aún no hemos implementado las rutinas correspondientes. Lo que si podemos comprobar es el funcionamiento correcto del mecanismo de variables globales para *x*, *y*, *angulo*.

Vamos a implementar la función *gira()*. Esta función debe actualizar el valor de la variable global *x*, sumándole el valor recibido. Habrá que ajustar el nuevo valor del *angulo* al intervalo *0-360*. El listado de la función 'GIRA' quedará de la siguiente forma:

```
function gira(value)
    global angulo;
    fprintf('GIRA %f\n', value)
    angulo = angulo + value;
    if abs(angulo) > 360
        angulo = rem(angulo, 360);
    end
    if angulo == 360 | angulo == -360
        angulo = 0;
    end
    if angulo < 0
        angulo = 360 + angulo;
    end
end
```

Podemos probar la nueva implementación de la función *gira()* con unas cuantas sentencias desde la consola de *Octave*. (Tendremos que haber realizado anteriormente la declaración y definición de las variables globales *x*, *y*, *angulo*). El resultado se ve en la siguiente figura:

```
octave:222>
octave:222> turtle('INIT')
x=  0.00 y=  0.00 angulo=  0.00
octave:223> turtle('GIRA',100)
GIRA 100.000000
octave:224> turtle('DISP')
x=  0.00 y=  0.00 angulo= 100.00
octave:225> turtle('GIRA',-200)
GIRA -200.000000
octave:226> turtle('DISP')
x=  0.00 y=  0.00 angulo= 260.00
octave:227> turtle('GIRA',200)
GIRA 200.000000
octave:228> turtle('DISP')
x=  0.00 y=  0.00 angulo= 100.00
octave:229> 
```

La implementación de la función *avanza()* requiere algunos cálculos trigonométricos para calcular los incrementos en *x* e *y*. El listado queda de la siguiente manera:

```
function avanza(value)
    global x y angulo;
    fprintf('AVANZA %f\n', value)
    incx = value * cosd(angulo);
    incy = value * sind(angulo);
    x = x + incx;
    y = y + incy;
end
```

Podemos volver a probar la tortuga con comandos 'AVANZA'. En la figura siguiente se puede comprobar el resultado.

Obsérvese que los ángulos de la tortuga se miden desde el eje x en sentido levógiro. La dirección positiva del eje y corresponde al ángulo 90 .

```
octave:253> turtle('INIT')
x=  0.00 y=  0.00 angulo=  0.00
octave:254> turtle('GIRA',90)
GIRA 90.000000
octave:255> turtle('AVANZA',100)
AVANZA 100.000000
octave:256> turtle('DISP')
x=  0.00 y= 100.00 angulo=  90.00
octave:257> turtle('GIRA',90)
GIRA 90.000000
octave:258> turtle('AVANZA',100)
AVANZA 100.000000
octave:259> turtle('DISP')
x= -100.00 y= 100.00 angulo= 180.00
octave:260> turtle('GIRA',90)
GIRA 90.000000
octave:261> turtle('AVANZA',100)
AVANZA 100.000000
octave:262> turtle('DISP')
x= -100.00 y=  0.00 angulo= 270.00
octave:263> turtle('GIRA',90)
GIRA 90.000000
octave:264> turtle('AVANZA',100)
AVANZA 100.000000
octave:265> turtle('DISP')
x=  0.00 y=  0.00 angulo=  0.00
octave:266>
```

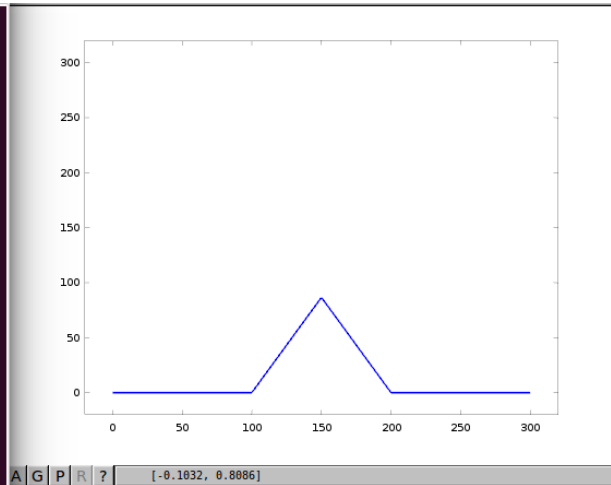
Una vez comprobado que la matemática de la tortuga funciona correctamente, podemos implementar las sentencias necesarias para que la tortuga dibuje los gráficos. En esta versión básica de tortuga la única función que realmente dibujará en pantalla será la función 'AVANZA'. Daremos por hecho que el programa principal que utilice nuestra función *turtle()* será el encargado de abrir y configurar la ventana de gráficos, haciendo un *hold on* para que la ventana de gráficos reciba los sucesivos comandos *plot()* enviados por la tortuga. Nuestra función 'turtle()' lo único que hace es enviar el comando *plot()* necesario para dibujar el segmento comprendido entre la posición de la tortuga al inicio del comando y la posición final calculada en base al ángulo y magnitud del avance de la tortuga. La función *avanza()* podría quedar resuelta de la siguiente forma:

```
function avanza(value)
    global x y angulo;
    fprintf('AVANZA %f\n', value)
    oldx = x;
    oldy = y;
    incx = value * cosd(angulo);
    incy = value * sind(angulo);
    x = oldx + incx;
    y = oldy + incy;
    xx = [oldx, x];
    yy = [oldy, y];
    plot(xx, yy, 'linewidth',2)
end
```

Podemos probar el resultado desde la consola de *Octave*. La secuencia de comandos y el gráfico que se obtendría

serían los siguientes:

```
octave:319> hold on
octave:320> xlim([-20,320])
octave:321> ylim([-20,320])
octave:322> turtle('INIT')
x= 0.00 y= 0.00 angulo= 0.00
octave:323> turtle('AVANZA', 100)
AVANZA 100.000000
octave:324> turtle('GIRA', 60); turtle('AVANZA',100)
GIRA 60.000000
AVANZA 100.000000
octave:325> turtle('GIRA', -120); turtle('AVANZA',100)
GIRA -120.000000
AVANZA 100.000000
octave:326> turtle('GIRA', 60); turtle('AVANZA',100)
GIRA 60.000000
AVANZA 100.000000
octave:327> █
```



Con esto tenemos creada la implementación básica de tortuga que nos permitirá realizar las primeras pruebas de dibujo. La forma habitual de utilizar la función `turtle()` será desde otras funciones que la utilizarán para realizar sus figuras. En el listado definitivo de la función `turtle()` que se muestra a continuación, se han suprimido las sentencias `fprintf()` de las funciones `avanza()` y `gira()`, para que no interfieran durante el dibujo de segmentos sucesivos.

```
function turtle(command, value)
% Tortuga básica para gráficos
% Necesita la existencia de unas variables globales x, y, angulo

    global x y angulo;

    if strcmp(command, 'GIRA')
        gira(value);
    elseif strcmp(command, 'AVANZA')
        avanza(value);
    elseif strcmp(command, 'DISP')
        disp();
    elseif strcmp(command, 'INIT')
        init();
    else
        dispError(command);
    end
end

function gira(value)
    global angulo;
    % fprintf('GIRA %f\n', value)
    angulo = angulo + value;
    if abs(angulo) > 360
        angulo = rem(angulo, 360);
    end
    if angulo == 360 | angulo == -360
        angulo = 0;
    end
    if angulo < 0
        angulo = 360 + angulo;
    end
end

function avanza(value)
```

```

global x y angulo;
% fprintf('AVANZA %f\n', value)
oldx = x;
oldy = y;
incx = value * cosd(angulo);
incy = value * sind(angulo);
x = oldx + incx;
y = oldy + incy;
xx = [oldx, x];
yy = [oldy, y];
plot(xx, yy, 'linewidth',2)
end

function disp()
global x y angulo;
fprintf('x= %6.2f y= %6.2f angulo= %6.2f\n', x, y, angulo)
end

function init()
global x y angulo;
x=0.0; y=0.0; angulo=0.0;
disp();
end

function dispError(command)
fprintf('Comando no reconocido: %s\n', command)
end

```

Podemos utilizar nuestra tortuga para dibujar algunas figuras sencillas. El siguiente listado muestra cómo utilizar la tortuga para dibujar un círculo, por ejemplo:

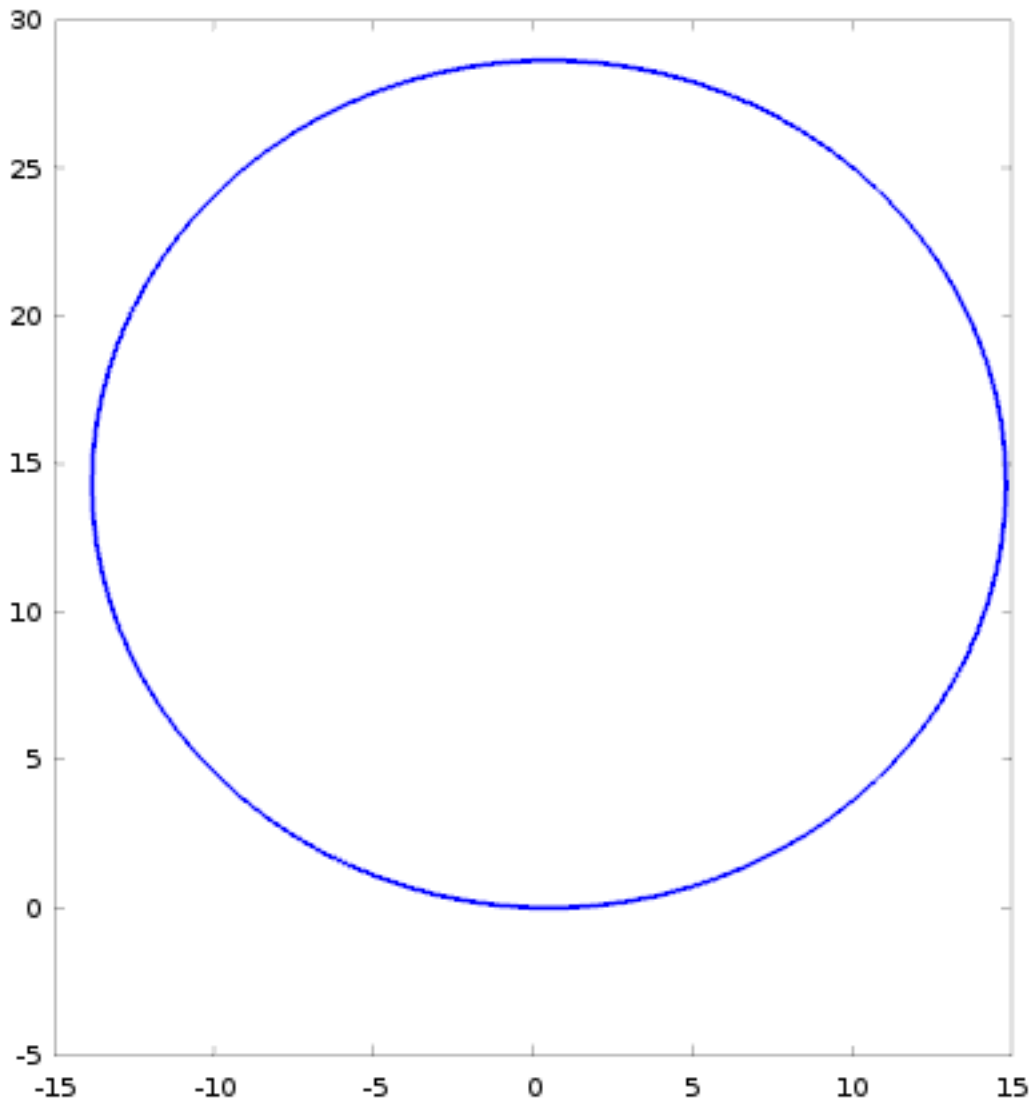
```

function circle()
% Dibuja un círculo utilizando la función turtle()
global x y angulo;
turtle('INIT');

close();
hold on;

for i = 0 : 90
    turtle('AVANZA', 1);
    turtle('GIRA',4);
end
end

```



En el siguiente ejemplo se utiliza la tortuga para dibujar una serie de triángulos:

```
function triang()
    % Dibuja triángulos
    global x y angulo;
    turtle('INIT');

    close();
    hold on;

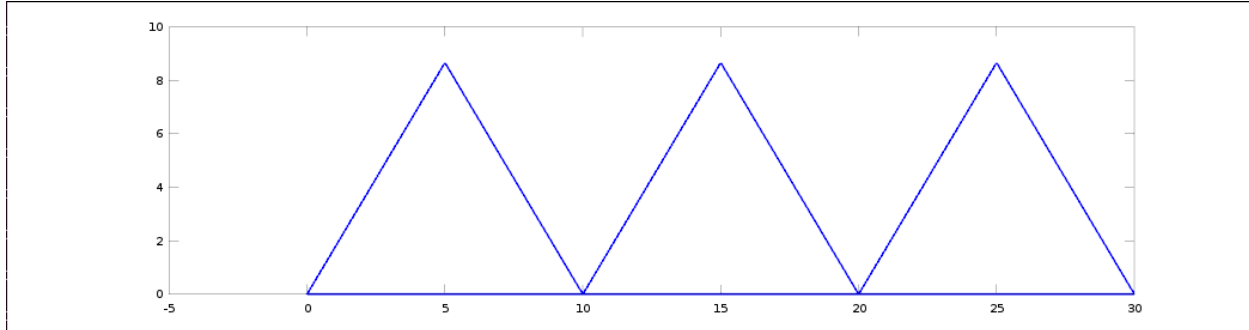
    for i = 1 : 3
        turtle('AVANZA', 10);
        turtle('GIRA', 120);
        turtle('AVANZA', 10);
    end
```



```

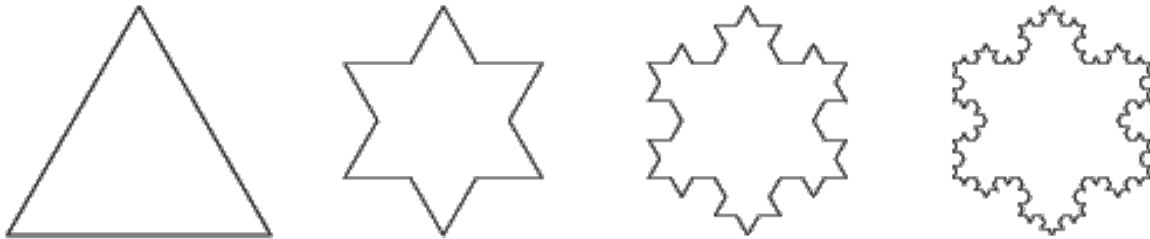
turtle('GIRA', 120);
turtle('AVANZA', 10);
turtle('GIRA', 120);
turtle('AVANZA', 10);
end
end

```



11.1 El copo de nieve de Koch

La curva fractal ‘*Koch snowflake*’, también conocida como la ‘*isla de Koch*’, fue descrita por primera vez por Helge von Koch en 1904. Se construye partiendo de un triángulo equilátero. En cada lado del triángulo se sustituye la tercera parte central por un nuevo triángulo equilátero de lado un tercio del primero. Este proceso se repite indefinidamente.



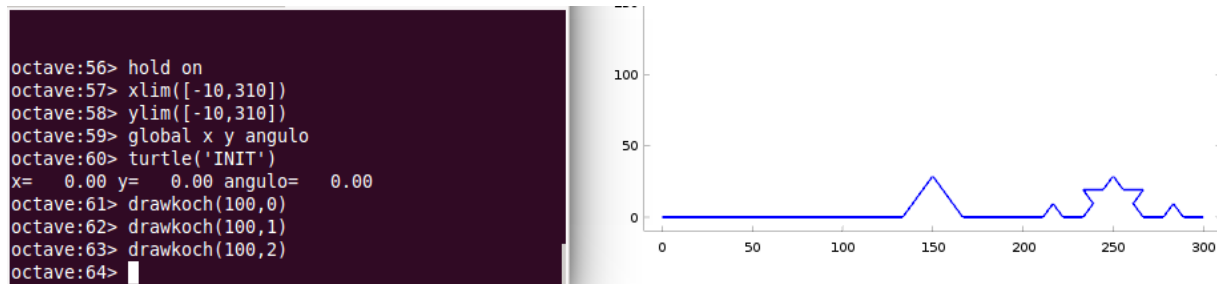
El resultado es una curva cuya longitud tiende a infinito, pero encerrando un área que tiende a una cantidad finita. Podemos implementar una función que dibuje una curva de Koch mediante la utilización de la tortuga creada en el apartado anterior. Para ello crearemos en primer lugar la función `drawkoch()` que dibuja, de forma recursiva, el segmento básico de la curva de Koch para un determinado nivel de profundidad:

```

function drawkoch(length, depth)
    if depth == 0
        turtle('AVANZA', length);
    else
        drawkoch(length/3, depth-1);
        turtle('GIRA', 60);
        drawkoch(length/3, depth-1);
        turtle('GIRA', -120);
        drawkoch(length/3, depth-1);
        turtle('GIRA', 60);
        drawkoch(length/3, depth-1);
    end
end

```

Podemos probar la función recién creada enlazando los tres primeros niveles de segmentos de Koch. La secuencia de instrucciones y el resultado se puede apreciar en la siguiente figura:



Para dibujar el copo de nieve completo, con un determinado nivel de profundidad tendremos que dibujar tres tramos de Koch, cada uno girado -120 grados respecto del anterior. Podemos implementar una función que denominaremos `koch()`, que utiliza la función anterior para dibujar el copo de nieve. El código de la función `koch()` será el siguiente:

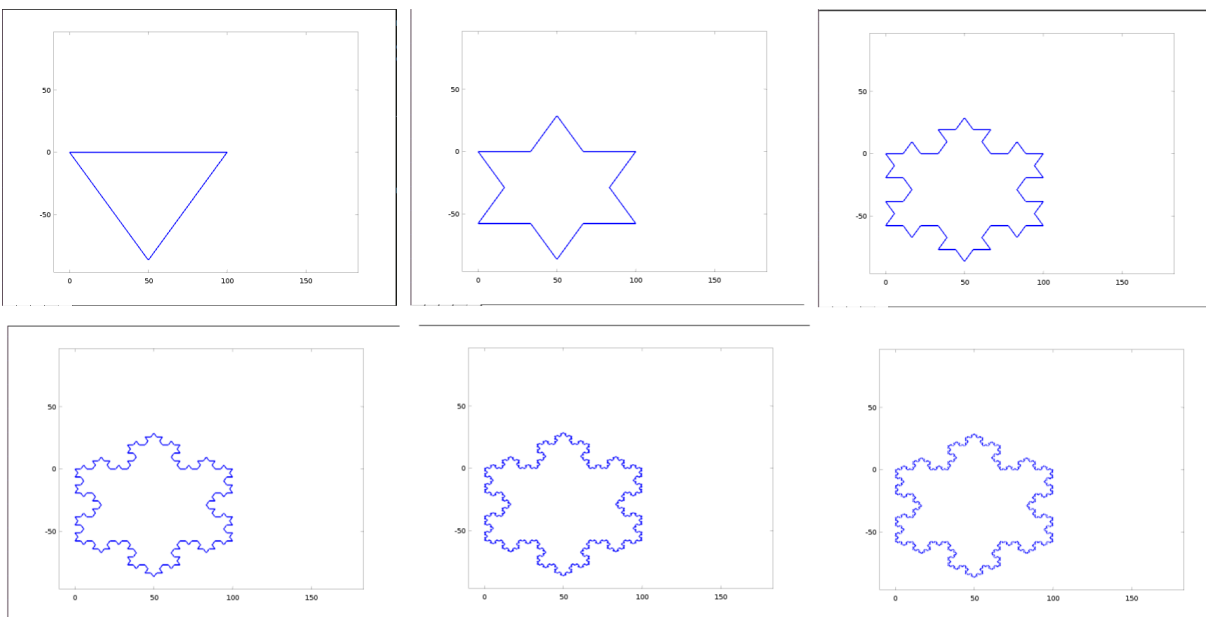
```
function koch(length, level)
% Dibuja la curva de Koch para un nivel determinado

hold off;
close;
hold on;
xlim([-10, 2*length*sind(60)+10]);
ylim([-10-length*sind(60), 10+length*sind(60)]);

global x y angle;
turtle('INIT');

drawkoch(length, level);
turtle('GIRA', -120);
drawkoch(length, level);
turtle('GIRA', -120);
drawkoch(length, level);
end
```

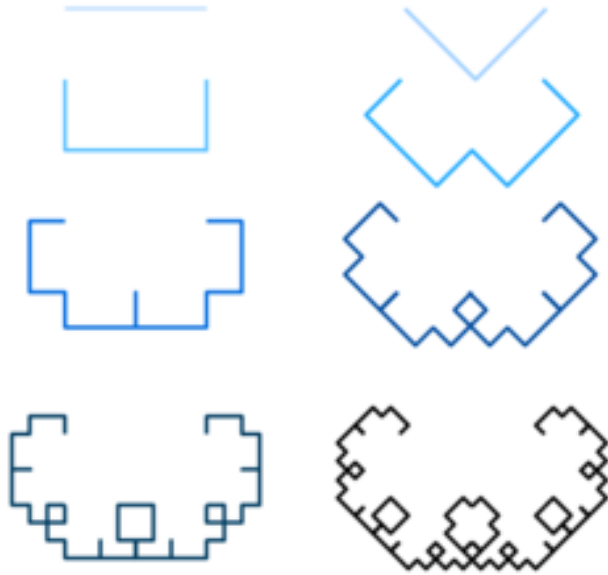
El resultado de la función para los primeros niveles se puede ver en la siguiente figura:



11.2 La curva C de Lévy

El último ejemplo que vamos a mostrar es el de la curva C de Lévy, descrita por primera vez por Ernesto Cesàro en 1906 y Georg Faber en 1910, pero que lleva el nombre del matemático francés Paul Lévy, que fue el primero que proporcionó un método geométrico para construirla.

La curva C comienza con un segmento recto de cierta longitud. Utilizando este segmento como hipotenusa, se construye un triángulo isósceles con ángulos 45° , 90° y 45° . Sustituimos el segmento original por los dos catetos del triángulo. El proceso se repite indefinidamente con cada uno de los segmentos que se generan. En la figura siguiente se muestran los primeros niveles de la curva C:

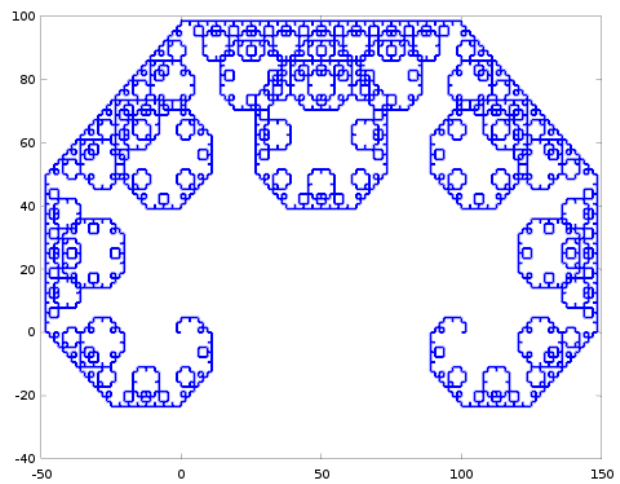


El código que se muestra a continuación dibuja la curva C para un determinado nivel de profundidad:

```
function ccurve(length,level)
  if level == 0
    turtle('AVANZA',length);
  else
    turtle('GIRA',45);
    ccurve(length/sqrt(2), level-1);
    turtle('GIRA',-90);
    ccurve(length/sqrt(2), level-1);
    turtle('GIRA',45);
  end
end
```

El resultado para el nivel de profundidad 12 se muestra en la siguiente figura:

```
octave:112> hold on
octave:113> global x y angulo
octave:114> turtle('INIT')
x=  0.00 y=  0.00 angulo=  0.00
octave:115> ccurve(100,12)
octave:116> 
```

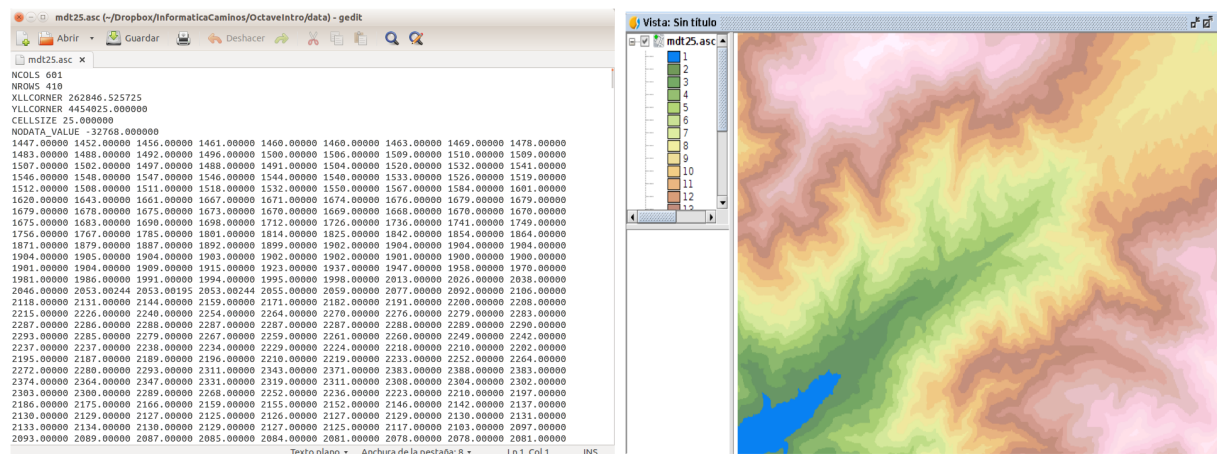


Modelos Digitales de Elevaciones

12.1 Introducción

Actualmente los Modelos Digitales de Elevaciones, **MDE**, sustituyen a los antiguos mapas de curvas de nivel. Consiste en dividir el terreno mediante una cuadrícula rectangular y asignar una cota a cada rectángulo resultante de la división. Esto permite representar el *MDE* mediante una matriz. Un formato habitual para este tipo de *mapas* es el formato *ESRI ASC*. Consiste en un fichero de texto en el que se guardan los valores de las alturas en formato coma flotante junto con unas líneas al principio que indican el número de filas y columnas del *MDE*, las coordenadas de la esquina inferior izquierda del mapa, (*Lower Left Corner*), el tamaño en metros del lado de los rectángulos de la rejilla, (*paso de malla*), y a veces algún dato más.

La figura siguiente muestra la cabecera de uno de estos ficheros **.asc** según se ve al abrirlo con un editor de texto y al abrirlo con el programa gvSIG:



Actualmente es posible descargar modelos digitales de elevaciones a nivel mundial con paso de malla 30 metros, desde el portal de la NASA <http://asterweb.jpl.nasa.gov/gdem.asp>. En España está disponible el *MDE* con paso de malla 5 metros. Se puede descargar desde el portal del Instituto Geográfico Nacional <http://centrodedescargas.cnig.es/CentroDescargas>.

12.2 Problema

Se pide desarrollar una rutina en *Octave-Matlab* que lea el fichero *mdt25.asc* y realice una representación tridimensional del mismo.

12.3 Solución

Si abrimos el fichero *mdt25.asc* con un editor de texto, veremos que tiene seis líneas de metadatos al principio, y luego vienen las verdaderas líneas de datos. Los metadatos del fichero *mdt25.asc* son los siguientes:

- *NCOLS, NROWS*: Número de filas y columnas de la rejilla rectangular en que se ha dividido el terreno, en este caso 601x410
- *XLLCORNER, YLLCORNER*: Coordenadas *UTM* de la esquina inferior izquierda del *MDE*
- *CELLSIZE*: Tamaño del lado de la celda, en este caso cuadrada de lado 25 metros
- *NODATA_VALUE*: Las celdas que tengan este valor quiere decir que no hay dato de altura de la celda.

En la solución que vamos a desarrollar aquí no haremos uso de los metadatos, salvo para saber el número de filas y columnas de la matriz, dato que miraremos abriendo el fichero con un editor de texto. La solución propuesta abre el fichero *mdt25.asc*, se salta las primeras seis líneas del fichero utilizando *fgets()* y a continuación lee los valores de las alturas en una matriz. Dada la forma de trabajar de la función *fscanf()*, que va leyendo valores y rellenando por columnas, es necesario leer la matriz con tamaño [columnasxfilas] y luego trasponerla. A continuación se muestra el código de la rutina Octave, y de la figura resultante:

```
function readmdt(filename)
% Lee un fichero raster en formato ASC

file = fopen(filename,'r');
if file <3
    fprintf('Error al abrir fichero %s', filename)
    return;
end

for i=1:6
    metadato = fgets(file);
    fprintf("%s", metadato)
end

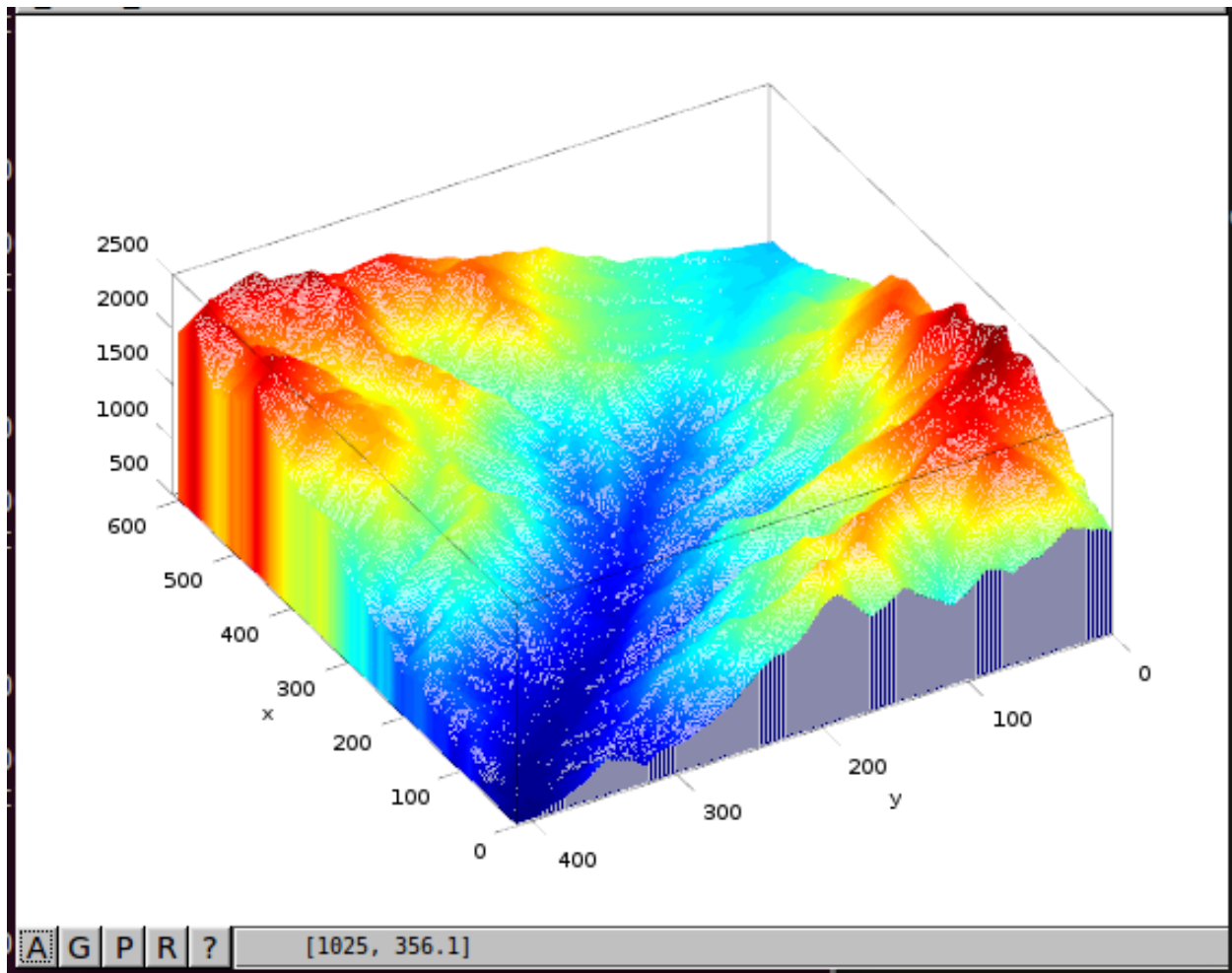
filas = 410;
columnas = 601;
A = fscanf(file, '%f', [columnas, filas]);
A = A';

hold off;
close;
hold on;
xlim([0,610]);
ylim([0,410]);
xlabel('x');
ylabel('y');
view(240,60);

meshz(A);

fclose(file);

end
```



Acceso a recursos de Internet

- **str = urlread(URL)** Hace una petición POST. Se pueden pasar argumentos con *urlread(url,name, value,...)*. Para hacer una petición *GET* hay que hacer: *urlread(URL, 'Get', {'param1', 'param2',...})*.
- **stat = web(url)** Abre una página web en el navegador

Utilización de clases java desde Octave

Note: Podemos acceder e interactuar con clases java desde *Octave*. Necesitaremos tener instalado el paquete *java*. Podemos comprobar si está instalado haciendo un *help javaObject* desde la consola de *Octave*. Si encuentra el *help* tendremos instalado el paquete. En caso contrario, habrá que instalarlo siguiendo las instrucciones de la sección *Utilización de paquetes adicionales*

El paquete *java* nos proporciona una API con varios métodos que podemos utilizar para crear objetos de clases *java* y acceder a sus propiedades y métodos.

Podemos instanciar un objeto de una clase *java* utilizando el método *javaObject()*, que admite como primer parámetro el nombre cualificado de la clase a la que pertenece el objeto a instanciar. El método *javaObject()* admite como parámetros adicionales los que queramos pasar al constructor de la clase.

En el siguiente ejemplo creamos un objeto *java.util.ArrayList* al que añadimos varios números utilizando su método *add()*. A continuación accedemos al contenido del *ArrayList* con el método *get()* de la clase *java* utilizado desde dentro de la función *fprintf()* de *Octave*. Por último imprimimos el tamaño del *ArrayList* utilizando su método *size()*:

```
list = javaObject('java.util.ArrayList')

list.add(4);
list.add(3);
list.add(5);

fprintf('%d %d %d \n',
        list.get(0), list.get(1), list.get(2))

fprintf('%d \n', list.size());
```

La salida en pantalla la podemos ver en la siguiente imagen:

```
octave:20> list = javaObject('java.util.ArrayList')
list =

<Java object: java.util.ArrayList>

octave:21> list.add(4);
octave:22> list.add(3);
octave:23> list.add(5);
octave:24> fprintf('%d %d %d \n', list.get(0), list.get(1), list.get(2))
4 3 5
octave:25> fprintf('%d \n',list.size())
3
octave:26> █
```

Podemos observar la manera de acceder a los métodos del objeto *java* creado mediante el código *objeto.punto.método* habitual en *java*.

- `javaclasspath()`
- `javaaddpath()`
- `javarmppath()`

Acerca de este documento

Este documento ha sido creado en Septiembre de 2014 por Santiago Higuera, profesor de informática de la Escuela de Ingenieros de Caminos de Madrid (España).

El objetivo es crear un manual de introducción a la utilización de . Se está realizando como material de apoyo a las clases de la asignatura Informática del primer curso de los estudios de Grado en Ingeniería Civil y Territorial que se imparten en la .

La documentación está realizada con la herramienta y el código fuente está alojado en . Si estás interesado en colaborar en la elaboración del manual, puedes ponerte en contacto con el autor por correo electrónico a .

Licencia

Excepto donde quede reflejado de otra manera, la presente documentación se halla bajo licencia [Creative Commons Reconocimiento Compartir Igual](#)

