Tutorial Archinformal de Octave 0.7.34

Diego Bravo Estrada

Julio 2023

Este es un tutorial confeccionado durante algunas sesiones de auto-aprendizaje; con suerte puede ser útil a algunos lectores. Escribir al autor a "diegobravoestrada arroba hotmail punto com" para correcciones y sugerencias.

1 Inicio

Octave es un programa que proporciona un lenguaje orientado a cálculo numérico basado en operaciones matriciales[1]. Los usuarios de Windows pueden descargar y ejecutar un instalador de `https://octave.org/download.html`; en las distribuciones Linux normalmente se proporciona como uno de los paquetes disponibles en sus respectivos repositorios de software.

1.1 Iniciar octave:

```
diego@inspiron:~$ octave
GNU Octave, version 3.2.3
Copyright (C) 2009 John W. Eaton and others.
This is free software; see the source code for copying conditions.
...
For information about changes from previous versions, type 'news'.
```

Normalmente se lanzará el entorno gráfico de Octave. Si se desea trabajar en modo texto, ejecutar del siguiente modo:

```
diego@inspiron:~$ octave --no-gui
octave:1>
```

[1]La sintaxis básica es muy similar a la de Matlab, por lo que es posible desarrollar funciones que se ejecutan idénticamente en ambos aplicativos. En este documento no hemos intentado lograr tal compatibilidad, lo cual puede ser un ejercicio interesante para el lector.

1.2 Usando Matrices

Crear una matriz:

```
octave:1> P=[4,5; 5,2]
P =
4 5
5 2
```

Si se agrega un punto y coma al final, no se imprime el valor asignado...

```
octave:2> Q=[1,1; 1,3];
```

"q" no es lo mismo que "Q":

```
octave:3> q
error: 'q' undefined near line 3 column 1
```

Imprimir la matriz recién creada:

```
octave:3> Q
Q =
1 1
1 3
```

Otras funciones incorporadas permiten crear matrices. Por ejemplo, una matriz de números aleatorios:

```
octave:4> ALEATORIOS = rand(5,2)
ALEATORIOS =
0.422600 0.044483
0.567010 0.917974
0.401833 0.954165
0.897529 0.601186
0.321889 0.726979
```

Multiplicación por un escalar:

```
octave:6> P
P =
4 5
5 2
octave:7> 2*P
ans =
8 10
10 4
octave:8> 3 * P
ans =
12 15
15 6
```

Multiplicación de matrices:

```
octave:11> P*Q
ans =
9 19
7 11
octave:12> Q*P
ans =
9 7
19 11
```

Transpuesta (en realidad, conjugado complejo de la transpuesta; la función `transpose()` proporciona la transpuesta estricta.)

```
octave:21> (P*Q)'
ans =
9 7
19 11
```

1.3 Resolución de ecuaciones Lineales

Al doble de lo que posee Pablito le falta 6 para alcanzar al quíntuple de lo que tiene Juanita. Pero a Juanita le falta 9 para alcanzar a Pablito. Cuánto tienen?

Solución:

$$\begin{aligned} 2 * P + 6 &= 5 * J \\ J + 9 &= P \end{aligned}$$

En matrices:

$$\begin{pmatrix} 2 & -5 \\ -1 & 1 \end{pmatrix} * X = \begin{pmatrix} -6 \\ -9 \end{pmatrix}$$

Usando el operador "backslash" de Octave:

```
octave:29> A=[2,-5;-1,1]
A =
2 -5
-1 1
octave:30> B=[-6;-9]
B =
-6
-9
octave:31> A\B
ans =
17
8
```

Respuesta Pablito tiene 17, Juanita 8.

1.4 Intervalos numéricos

Una manera sencilla de crear una matriz de $1xN$ cuyos elementos están en progresión aritmética...

```
octave:36> N=[1:10]
N =
1 2 3 4 5 6 7 8 9 10
```

Con salto:

```
octave:38> N=[1:2:10]
N =
1 3 5 7 9
octave:40> N = [4:5:80]
N =
4 9 14 19 24 29 34 39 44 49 54 59 64 69 74 79
```

Igualdad con un elemento (no confundir con comparación de dos matrices):

```
octave:41> N == 69
ans =
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

Como se ve, se retorna una matriz de ceros y unos; los unos corresponden al elemento que es igual a lo especificado.

La función `find()` permite ubicar los índices de elementos distintos de cero de una matriz. Por ejemplo:

```
octave:45> find([0,0,9,0,3,0,1,2,0,0])
ans =
3 5 7 8
```

Para el ejemplo de más arriba, cómo ubicar el índice del elemento 69?

```
octave:46> find(N==69)
ans = 14
```

De modo inverso, cómo obtener el elemento decimocuarto?

```
octave:47> N(14)
ans = 69
```

Se sugiere consultar las funciones relacionadas `any()` y `all()`.

1.5 Búsquedas con lookup()

`Find()` no es el mejor método para esto; `lookup()` permite hallar la posición del elemento que se desea, siempre que la matriz esté ordenada. Si no se encuentra el elemento, proporciona la posición del más cercano anterior. Para el ejemplo:

```
octave:52> lookup(N,19)
ans = 4
octave:53> lookup(N,21)
ans = 4
octave:54> lookup(N,25)
ans = 5
```

También se pueden ubicar las posiciones de más de un elemento:

```
octave:56> lookup(N,[19,45])
ans =
4 9
```

Notar que las posiciones encontradas se retornan en una matriz con las mismas dimensiones que la matriz de los elementos deseados, y no de la fuente de elementos.

```
octave:57> lookup(N,[19,45]')
ans =
4
9
octave:59> lookup(N',[19,45])
ans =
4 9
```

1.6 Extracción de elementos de una matriz

Partimos de un "cuadrado mágico" con la función `magic`():

```
octave:48> K=magic(4)
K =
 16  2  3 13
  5 11 10  8
  9  7  6 12
  4 14 15  1
```

Obtener la segunda fila:

```
octave:49> K(2,:)
ans =
  5 11 10 8
```

La tercera columna:

```
octave:50> K(:,3)
ans =
  3
 10
  6
 15
```

El “cuadrado interno”:

```
octave:51> K(2:3,2:3)
ans =
  11 10
   7 6
```

Las filas 2 a 4, sólo la tercera columna:

```
octave:52> K(2:4,3)
ans =
  10
   6
  15
```

Otra forma de generar rangos, con linspace. Genera N elementos igualmente espaciados en un intervalo. Por ejemplo, de 2 a 4, 11 elementos:

```
octave:56> linspace(2,4,11)
ans =
2.0000 2.2000 2.4000 2.6000 2.8000 3.0000 3.2000 3.4000 3.6000 3.8000 4.0000
octave:57> linspace(2,4,10)
ans =
2.0000 2.2222 2.4444 2.6667 2.8889 3.1111 3.3333 3.5556 3.7778 4.0000
```

1.7 Unión de matrices

Dadas:

```
octave:10> A=[1,2;3,4]
A =
1 2
3 4
octave:11> B=[5,5;6,6;7,7]
B =
5 5
6 6
7 7
```

Podemos obtener una matriz total uniendo las filas de ambas (esto funciona debido a que ambas tienen el mismo número de columnas.)

```
octave:13> [A;B]
ans =
  1 2
  3 4
  5 5
  6 6
  7 7
```

Para unir las columnas de ambas, es lógico que el número de filas debe ser el mismo:

```
octave:14> [A,B]
error: number of rows must match (3 != 2) near line 14, column 4
```

Pero podemos a unir A con la traspuesta de B:

```
octave:14> [A,B']
ans =
  1 2 5 6 7
  3 4 5 6 7
```

En resumen, con $[X, Y]$ unimos las matrices a lo largo de las columnas; con $[X; Y]$ se unen a lo largo de sus filas.

La función `zeros()` proporciona una matriz de ceros:

```
octave:17> zeros(3,7)
ans =
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
  0 0 0 0 0 0 0
```

La función `eye()` proporciona una matriz identidad:

```
octave:20> eye(3)
ans =
Diagonal Matrix
1 0 0
0 1 0
0 0 1
```

Podemos combinar lo anterior así (notar la unión de tres matrices):

```
octave:21> [[A,B',A];zeros(3,7)]
ans =
1 2 5 6 7 1 2
3 4 5 6 7 3 4
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

Otro ejemplo:

```
octave:38> Q=ones(4)
Q =
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
octave:39> R=rand(3,2)
R =
0.391289 0.853896
0.534633 0.739454
0.052724 0.515341
octave:40> Q(1:2,:)
ans =
1 1 1 1
1 1 1 1
octave:41> [Q(1:2,:),R']
ans =
1.000000 1.000000 1.000000 1.000000 0.391289 0.534633 0.052724
1.000000 1.000000 1.000000 1.000000 0.853896 0.739454 0.515341
```

1.8 Tipos de datos

Usualmente Octave utiliza números de punto flotante de doble precisión (también conocido como "`double`" en lenguaje C) que es la representación usual para los números reales en cálculo científico. Ocasionalmente puede ser conveniente forzar el uso de otros tipos de datos más restringidos:

```
octave:58> R=(100*(rand(4,4).*rand(4,4))).^2
R =
  1.6480e+02 2.9748e+00 7.1926e+00 4.1811e+01
  5.6098e+03 8.5593e+01 2.3651e+00 3.2293e+02
  3.5274e+01 9.6808e+01 2.1813e+03 2.5410e+03
  3.4960e+02 7.5738e+00 1.2520e-02 6.1105e+03
octave:61> RI8=int8(R)
RI8 =
  127  3   7  42
  127 86   2 127
   35 97 127 127
  127  8   0 127
octave:62> RI16=int16(R)
RI16 =
  165   3    7   42
  5610 86    2  323
  35   97 2181 2541
  350   8    0 6111
octave:63> RI32=int32(R)
RI32 =
```

```
  165   3    7   42
  5610 86    2  323
  35   97 2181 2541
  350   8    0 6111
```

El tipo se conserva:

```
octave:67> class(RI16)
ans = int16
octave:68> 2.3+RI16
ans =
  167  5    9   44
 5612 88    4  325
   37 99 2183 2543
  352 10    2 6113
```

Aquí se aprecia que el valor 2.3 (todas las constantes numéricas son double por omisión) es promovido al nivel inferior del otro operador (`int16`), en lugar de promover al `int16` hacia double como ocurre en lenguaje C. En general, este es el comporamiento de Octave.

Para forzar a `double`:

```
octave:69> 2.3+double(RI16)
ans =
  167.3000  5.3000    9.3000   44.3000
 5612.3000 88.3000    4.3000  325.3000
   37.3000 99.3000 2183.3000 2543.3000
  352.3000 10.3000    2.3000 6113.3000
```

Octave opera internamente en doble precisión, pero los resultados se muestran en precisión simple por omisión, por ejemplo:

```
octave:9> a=rand(2,3);
octave:10> a
a =
  0.67448 0.71599 0.48106
  0.17609 0.97179 0.65884
```

Para mostrar los valores en doble precisión:

```
octave:11> format long
octave:12> a
a =
  0.674481809361969 0.715987465166574 0.481057898720585
  0.176085436236263 0.971787720305162 0.658836932094040
octave:13> format short
octave:14> a
a =
  0.67448 0.71599 0.48106
  0.17609 0.97179 0.65884
```

Para eliminar variables de la memoria utilizamos "`clear`"[2]. El comando "`whos`" permite apreciar las variables actualmente definidas:

```
octave:16> clear a
octave:17> a
error: 'a' undefined near line 17 column 1
octave:17> whos
Variables in the current scope:
Attr Name Size Bytes Class
==== ==== ==== ===== =====
     M    7x1     56 double
     NT   7x1     56 double
     T    7x1     56 double
     ans  1x30    30 char
Total is 51 elements using 198 bytes
```

1.9 Tamaño de las Matrices

Para obtener las dimensiones de una matriz utilizamos la función `size()`:

```
octave:18> N=ones(2,3)
N =
1 1 1
1 1 1
octave:19> size(N)
ans =
2 3
octave:20> length(N)
ans = 3
octave:21> rows(N)
ans = 2
octave:22> columns(N)
ans = 3
```

Notar que `length()` devuelve la dimensión de mayor extensión.

El rango de una matriz se puede obtener con `rank()`:

```
octave:25> A=[2,3,4;4,6,8;4,9,12]
A =
2 3 4
4 6 8
4 9 12
octave:26> rank(A)
ans = 2
```

En el ejemplo mostrado, la segunda fila es igual al doble de la primera, por lo que las tres no son linealmente dependientes.

[2] `clear` sin argumentos elimina todas las variables de memoria.

De otro lado, las funciones `min()` y `max()` calculan los valores mínimo y máximo respectivamente de un vector[3]. Para una matriz, encuentra los valores mínimos y máximos de las columnas (y devuelve un vector fila con estos valores.) Por ejemplo:

```
octave:27> min(A)
ans =
2 3 4
octave:28> max(A)
ans =
4 9 12
octave:29> max(max(A))
ans = 12
```

Para ordenar los elementos de un vector utilizamos `sort()`. En una matriz, ordena los elementos de cada columna:

```
octave:37> B=int8(10*rand(5,6))
B =
7 7 7 7 4 9
4 6 8 4 2 2
8 4 1 8 2 1
9 10 9 0 7 2
4 9 2 2 6 10
octave:38> sort(B)
ans =
4 4 1 0 2 1
4 6 2 2 2 2
7 7 7 4 4 2
8 9 8 7 6 9
9 10 9 8 7 10
octave:42> sort(B(1,:))
ans =
4 7 7 7 7 9
octave:44> sort(B(1,:), 'descend')
ans =
9 7 7 7 7 4
```

[3]Con la palabra "vector" aquí nos referimos simplemente a una "tupla", y más precísamente a una matriz de $N * 1$ o de $1 * N$. Continuaremos con este uso informal de "vector como tupla", puesto que este documento está más orientado hacia la ingeniería que a las ciencias matemáticas.

1.10 Ejercicio

1 Resolver el sistema de ecuaciones lineales

$$\begin{aligned} 89x - 21y - 31z &= 421 \\ 12.3x + 91z &= 12 \\ -x - y + z &= 94 \end{aligned}$$

Solución: Formamos las matrices A y B:

```
octave:48> A=[89,-21,-31 ; 12.3,0,91 ; -1,-1,1]
A =

   89.00000  -21.00000  -31.00000
   12.30000    0.00000   91.00000
   -1.00000   -1.00000    1.00000

octave:49> B=[421;12;94]
B =

   421
    12
    94
```

Obtenemos la solución:

```
octave:50> A\B
ans =

  -13.2117
  -78.8707
    1.9176
```

Otra forma menos eficiente:

```
octave:52> inv(A)*B  ans =
  -13.2117    -78.8707      1.9176
```

2 Definiendo y utilizando funciones

A continuación definimos una función que recibe tres argumentos de entrada, los cuales permiten obtener un resultado (valor de retorno de la función.) A diferencia de muchos lenguajes de programación que proporcionan una

sentencia "`return`" para este fin[4], aquí los valores se retornan asignándolos a ciertas variables señaladas en la definición de la función (en nuestro caso, la hemos denominado "`retorno`".)

```
octave:7> function retorno = notanbasica(primero, segundo, tercero)
> printf("Invocando a notanbasica\n");
> retorno = primero + segundo * tercero;
> printf("...Termina notanbasica\n");
> endfunction
octave:8> notanbasica(1,2,3)
Invocando a notanbasica
...Termina notanbasica
ans = 7
```

Asignando el resultado a una variable:

```
octave:10> a = notanbasica(1,2,3);
Invocando a notanbasica
...Termina notanbasica
octave:11> a
a = 7
```

En la práctica las funciones deben ser almacenadas en un archivo independiente (convencionalmente con extensión `.m`). El entorno gráfico permite la edición de archivos de funciones con soporte a la sintaxis. Cuando se utiliza Octave en modo texto, es usual que el usuario utilice su propio editor de texto, o que lo despliegue mediante el comando `edit` de Octave. En tal caso puede ser conveniente configurar el editor a ser desplegado, lo que se consigue creando o modificando el archivo `$HOME/.octaverc`, de modo tal que contenga:

```
edit editor "vi %s"
```

o para un editor en entorno gráfico que se lanza asincrónicamente a la sesión (como una tarea que no detiene a Octave):

```
edit editor "gvim %s"
edit mode "async"
```

Recalcamos que esto último normalmente no es necesario para los usuarios que emplean Octave en su entorno gráfico usual.

[4]En Octave también se dispone de una sentencia "`return`", pero (a diferencia de lo que ocurre en muchos lenguajes), sólo fuerza el retorno de la función: el valor a retornar siempre se pasa mediante la "variable de retorno".

2.1 Funciones con argumentos variables[5]

Utilizar la variable especial `varargin`:

```
function retorno = prueba(varargin)
    v = [varargin{:}];
    retorno(1) = sum(v);
    retorno(2) = prod(v);
endfunction
```

`varargin` es un "cell array", es decir una colección que soporta valores de diverso tipo. Con {:} se obtiene una lista de todos los valores, y con [] se crea una matriz de esta lista.

```
octave:21> prueba(3,4)
ans =

    7   12
```

La variable especial `nargin` se puede utilizar para validar que el usuario ha introducido cierto número de parámetros:

```
function [ ret ] = prueba2 (valores)
    if(nargin != 1)
        error("E1: Debe pasarse una matriz de 1x2");
    endif
    if(! ismatrix(valores))
        error("E2: Debe pasarse una matriz de 1x2");
    endif
    if(!isequal(size(valores), [1,2]))
        error("E3: Debe pasarse una matriz de 1x2");
    endif
    ret(1) = sum(valores);
    ret(2) = prod(valores);
endfunction

octave:33> prueba2
error: E1: Debe pasarse una matriz de 1x2
error: called from:
error:   /home/diego/octave/prueba2.m at line 4, column 3
octave:33> prueba2(3,4,5)
error: E1: Debe pasarse una matriz de 1x2
error: called from:
error:   /home/diego/octave/prueba2.m at line 4, column 3
octave:33> prueba2(3)
error: E3: Debe pasarse una matriz de 1x2
error: called from:
```

[5]Esta sección puede obviarse en una primera lectura. A nuestro criterio, son contados los casos en los que esta sintaxis es beneficiosa versus la complejidad adicional que conlleva.

```
error:   /home/diego/octave/prueba2.m at line 10, column 3
octave:33> prueba2("33")
error: E2: Debe pasarse una matriz de 1x2
error: called from:
error:   /home/diego/octave/prueba2.m at line 7, column 3
octave:33> prueba2([3,4])
ans =

    7   12
```

2.2 Número variable de valores de retorno

Esto puede resultar algo sorprendente[6], pero se usa ampliamente en Octave (y Matlab.) El usuario puede solicitar un número variable de valores de respuesta en el momento de la invocación mediante una sintaxis como esta:

```
[x,y,z] = f(...) # solicita 3 argumentos de salida
[x,y] = f(...) # solicita sólo 2
```

Para esto, la función debe emplear la variable especial `nargout`. Ejemplo:

```
function [ a,b,c,d ] = prueba3 ()
    printf("Se solicito %d valores\n", nargout);
    a=3;
    b=4;
    c=5;
    d=6;
endfunction

octave:3> prueba3
Se solicito 0 valores
ans =  3
octave:4> [x,y] = prueba3
Se solicito 2 valores
x =  3
y =  4
octave:5> [x,y] = prueba3;
Se solicito 2 valores
octave:6> [x,y,z,w,k] = prueba3
Se solicito 5 valores
x =  3
y =  4
z =  5
w =  6
```

[6] Al menos si se compara con la mayoría de lenguajes de propósito general donde la función únicamente opera en base a los argumentos de entrada, siguiendo una estructura más fiel a la definición de las funciones matemáticas.

```
error: element number 5 undefined in return list
octave:6> [x,y,z,w] = prueba3
Se solicito 4 valores
x =  3
y =  4
z =  5
w =  6
```

2.3 Introducir comentarios y proporcionar ayuda

En Octave existen distintas formas de introducir comentarios[7]; son especialmente comunes las líneas que se inician con un signo de porcentaje (%) y las líneas que se inician con un signo de número (#); la primera forma es compatible con Matlab. Un grupo ininterrumpido de líneas de comentario se considera un bloque de comentarios.

El primer bloque de comentarios dentro de una función corresponde a la ayuda de la misma:

```
function [ ret ] = prueba_ayuda (x,y)
% Utilizar: prueba_ayuda(x,y)
%
% Esta funcion permite hallar la media armonica de dos
% numeros

% Aqui mas comentarios de un segundo bloque
ret = 2*x*y/(x+y);

endfunction
```

luego, esta ayuda puede ser obtenida mediante el comando "`help`":

```
octave:31> help prueba_ayuda
'prueba_ayuda' is a function from the file /home/diego/octave/prueba_ayuda.m

 Utilizar: prueba_ayuda(x,y)

 Esta funcion permite hallar la media armonica de dos
 numeros
```

[7] Ver la documentación oficial de Octave más más detalles; al momento de escribir estas líneas, esta información se encuentra en `https://docs.octave.org/v8.2.0/Comments.html`.

2.4 Ejercicio

Proporcionar una secuencia de 10000 números aleatorios binarios a partir de `rand()`. Aplicar los tres primeros tests de calidad recomendados por NIST/CSRC[8]:

a) Sesgo binario Dada la secuencia de números binarios pseudo-aleatoria $e_1, e_2, ...e_n$ evaluar los valores $X_i = 2e_i - 1$ (que produce +1 y -1)

Evaluar

$$S = \frac{\sum X_i}{\sqrt{n}}$$

Evaluar la función de error complementario

$$P = erf(\frac{S}{\sqrt{2}})$$

Si $P < 0.01$ entonces se concluye que la secuencia no es aleatoria.

b) Test de frecuencia por bloque Dada la secuencia de números binarios pseudo-aleatoria $e_1, e_2, ...e_n$ y un tamaño de bloque "M", se particiona la secuencia en $N = [n/M]$ bloques sin "overlapping", descartándose los elementos no utilizados[9].

Determinar la proporción J_i de "unos" dentro de cada M-bloque usando la ecuación:

$$J_i = \frac{\sum_{j=1}^{M} e_{i-1}M + j}{M}$$

para $i = 1...N$.

Se calcula

$$\chi^2 = 4M \sum_{i=1}^{N} (J_i - \frac{1}{2})^2$$

[8]Tomado de "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" (NIST/CSRC) versión 2008; existe una versión actualizada en `https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final`. El que una secuencia de números pseudoaleatorios apruebe estos tests no garantiza su calidad; estos tests sólo funcionan en modo negativo: permiten detectar ciertas secuencias presumiblemente "de mala calidad".

[9]$[x]$ se refiere al máximo entero, implementado por la función `floor()` de Octave.

para $i = 1...N$.

Calcular $P = igamc(\frac{\chi^2}{2}, {}^{N}/_{2})$, donde $igamc(x, a)$ se obtiene de la función gamma incompleta:

$$igamc(x, a) = 1\text{–}gammainc(x, a)$$

Si $P < 0.01$, la secuencia no es aleatoria.

c) Tests de "carreras" Una "carrera" de longitud k consiste de k bits idénticos y está precedida y sucedida por bits de valor opuesto. Dados $e_1, e_2, ...e_n$ se obtiene la proporción:

$$J = \frac{\sum_{j=1}^{n} e_j}{n}$$

Se calcula los valores $r(k)$ con $k \in [1..(n-1)]$, donde $r(k) = 0$ si $e(k) = e(k+1)$ y $r(k) = 1$ en caso contrario.

$$V_n = 1 + \sum_{k=1}^{n-1} r(k)$$

Se obtiene el valor P:

$$P = erfc\left(\frac{|V_n - 2nJ(1-J)|}{2\sqrt{2n}J(1-j)}\right)$$

Si $P < 0.01$, la secuencia no es aleatoria.

Solución: Sea N un conjunto de números obtenidos con `rand()` como en los ejemplos:

```
N=rand(10000,1);
```

Obtenemos datos binarios:

```
octave:24> E=N>0.5;
```

a) Crearemos la función `Frequency()` que recibe estos valores:

```
function [ret] = Frequency(E)
        X=2*E-1;
        suma = sum(X);
        S=abs(suma)/sqrt(size(E,1));
        P=erfc(S/sqrt(2));
        ret = P;
```

```
endfunction

octave:56> Frequency(E)
ans =  0.40091
```

Como $P > 0.01$ entonces no descartamos la aleatoridad de la secuencia.

b) Creamos la función `BlockFrequency(E,M)` que recibe la sencuencia y el tamaño de bloque. El aspecto más importante de esta implementación consiste en la creación de la matriz EM que contiene todos los N intervalos en columnas de M filas. Para esto empleamos `reshape()`, sin embargo, esta función requiere que la fuente de los datos sea exactamente de tamaño MxN, por lo que previamente obtenemos el vector E2 con esta cantidad de elementos, posiblemente descartándose los del final de E.

```
function [ ret ] = BlockFrequency (E,M)
        n = size(E, 1);
        N = floor(n/M);
        E2 = E(1:N*M);
        EM = reshape(E2,M,N);
        j = sum(EM)/M;
        CHI2 = 4 * M * sum((j' - 1/2).^2);
        ret = 1 - gammainc(CHI2 / 2, N / 2);
endfunction
```

Probaremos la función (nota: el documento de NIST aconseja que $M > 0.01n$)

Tomaremos $M = 150$:

```
octave:71> BlockFrequency(E,150)
ans =  0.16724
```

como $P > 0.01$ entonces la aleatoridad de la secuencia no se descarta.

Una forma alternativa menos eficiente pero que puede ser más flexible y simple consiste en iterar los N intervalos (en lugar de crear la matriz auxiliar con `reshape`):

```
function [ ret ] = BlockFrequency2(E,M)
        n = size(E, 1);
        N = floor(n/M);
        for i = [1:N]
                j(i) = sum(E([(i - 1) * M + 1: (i - 1) * M + M]))/M;
        endfor
        CHI2 = 4 * M * sum((j - 1/2).^2);
        ret = 1 - gammainc(CHI2 / 2, N / 2);
endfunction
```

c) Crearemos la función `Runs(E)` que recibe la secuencia.

Aquí el paso clave consiste en aprovechar la no-igualdad de elementos (operador `!=`)[10] para obtener unos y ceros, evitando un loop. Para esto creamos dos matrices desplazadas en un elemento extremo al inicio y al final: $[E; 0]$ y $[0; E]$. El resultado produce una matriz de $(n + 1)$ elementos, descartándose los elementos extremos (tomamos desde "2" hasta "n".)

```
function [ ret ] = Runs (E)
        n = size(E, 1);
        J=sum(E)/n;
        rextra = [E;0]!=[0;E];
        r=rextra(2:size(E,1));
        Vn=sum(r)+1;
        P=erfc(abs(Vn-2*n*J*(1-J))/(2*sqrt(2*n)*J*(1-J)));
        ret = P;
endfunction

octave:113> Runs(E)
ans =  0.66503
```

como $P > 0.01$ la aleatoridad de la secuencia no se descarta.

3 Funciones gráficas

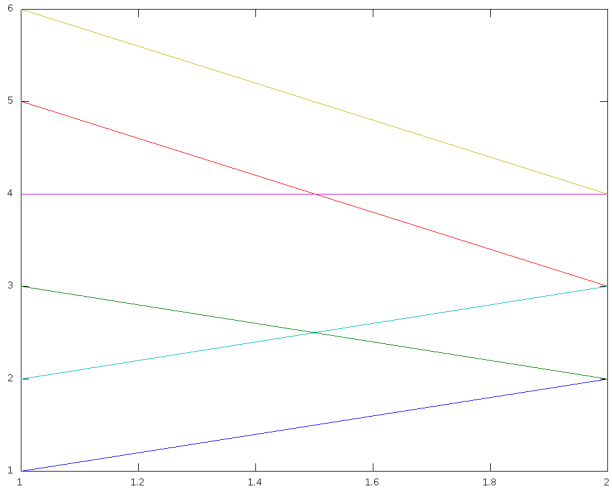
3.1 Ploteo de valores

Una lista de valores puede ser trazada a lo largo del eje Y, utilizando los números naturales como eje X. Dicho de otro modo, si se desea dibujar una matriz, los valores se representan en el eje Y, mientras que los índices de éstos se representan en el eje X.

Los valores de una matriz a trazarse normalmente deben ubicarse a lo largo de columnas. En el siguiente ejemplo, plot asume seis series de valores, cada una de sólo dos elementos:

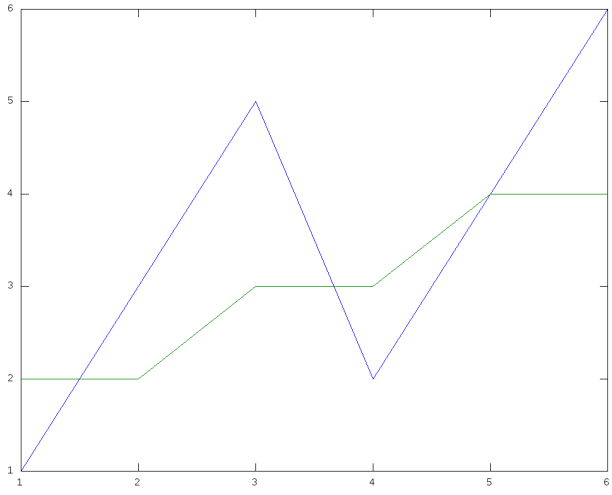
```
octave:62> A=[1,3,5,2,4,6;2,2,3,3,4,4]
A =

   1   3   5   2   4   6
   2   2   3   3   4   4

octave:63> plot(A)
```

[10] También puede emplearse el operador `~=` que es compatible con Matlab.



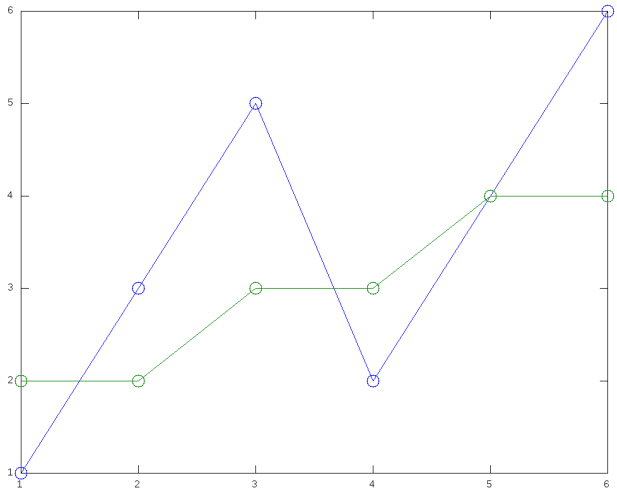
Más frecuente es el ploteo de muchos valores en pocas series:

```
octave:68> plot(A')
```



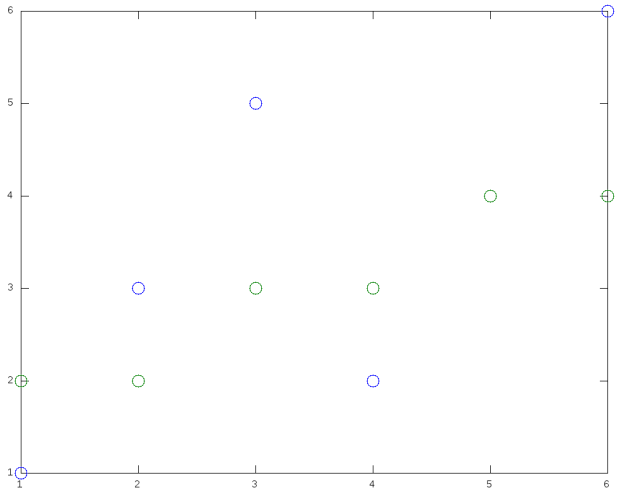
El argumento textual que sigue a los valores determina el formato del gráfico. En este caso es de líneas ("-") y tiene señalización con círculos ("o"). A continuación es posible agregar propiedades, por ejemplo, el tamaño de la señalización (markersize.)

```
octave:87> plot(A', "-o", "markersize", 20);
```



Estilo de puntos (sin líneas):

```
octave:90> plot(A', ".o", "markersize", 20);
```



Los gráficos se asocian a una "figura", que puede considerarse una "ventana física"; por omisión se utiliza la número "1". Si se hace otro dibujo (por ejemplo con `plot`) se reemplazará la figura actual. Para generar varios gráficos en simultáneo, se requiere activar más figuras utilizando figure(). Para grabar la figura actual se puede utilizar `print(‘‘nombre.png’’)`; para grabar

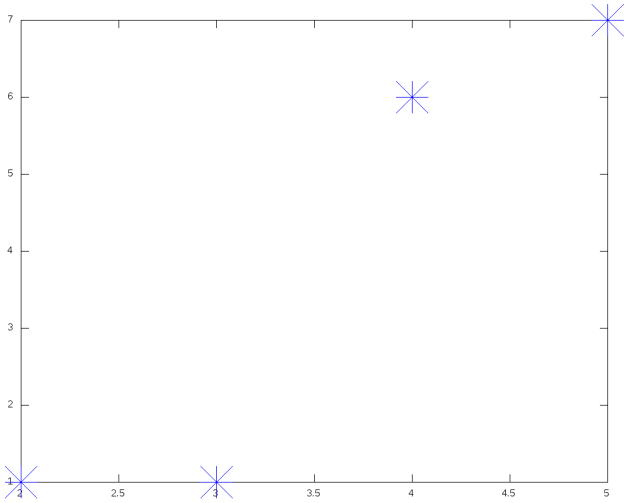
la figura “n”, usar `print(n, ‘‘nombre.png’’)`; diversas extensiones y formatos de archivo gráfico son válidas.

3.2 Ejes

Partimos del gráfico de algunos puntos sueltos:

```
octave:18> A=[2,3,4,5]';
octave:19> B=[1,1,6,7]';
octave:20> plot(A,B,'.*',"markersize", 50)
```

Obtenemos puntos que están sobre los ejes y se hacen difíciles de visualizar:

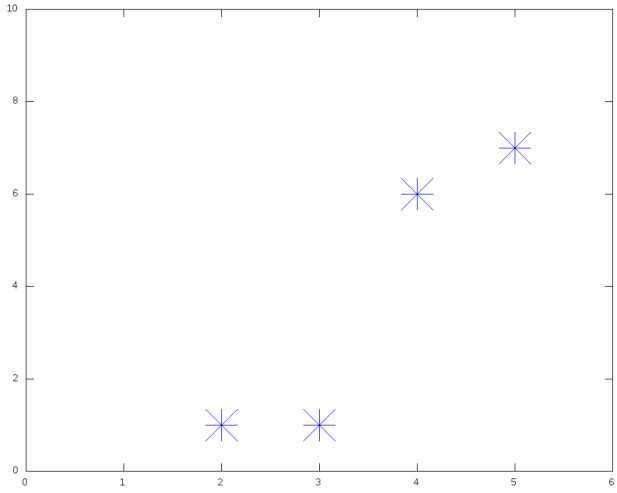


En el ejemplo, hemos forzado esto con un “markersize” muy elevado, de lo contrario los puntos serían casi indistinguibles.

Podemos mejorar esto definiendo el rango para los ejes de coordenadas a visualizar utilizando “axis”; por ejemplo:

```
octave:24> axis([0,6,0,10])
```

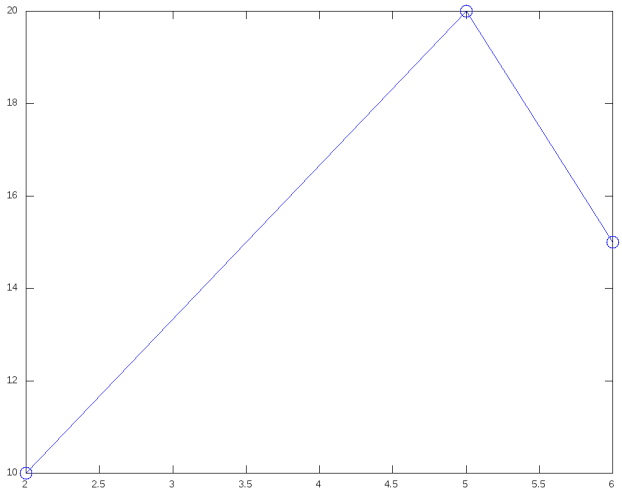
Permite obtener el siguiente gráfico:



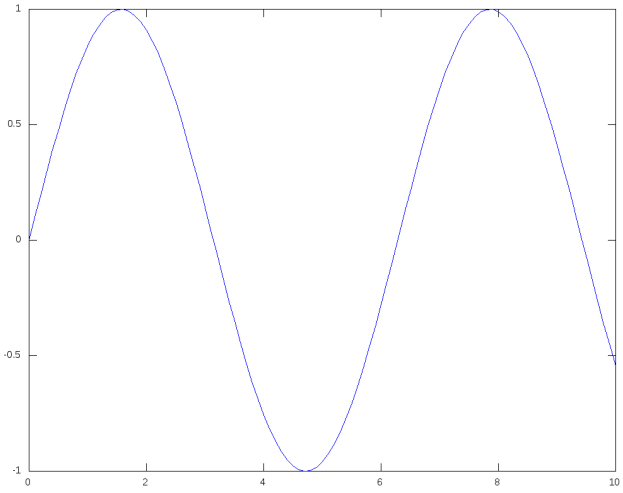
3.3 Ploteo de funciones

Es similar al anterior pero el eje X se fija a los valores de un dominio.

```
octave:93> plot([2;5;6],[10;20;15], "-o", "markersize", 20);
```

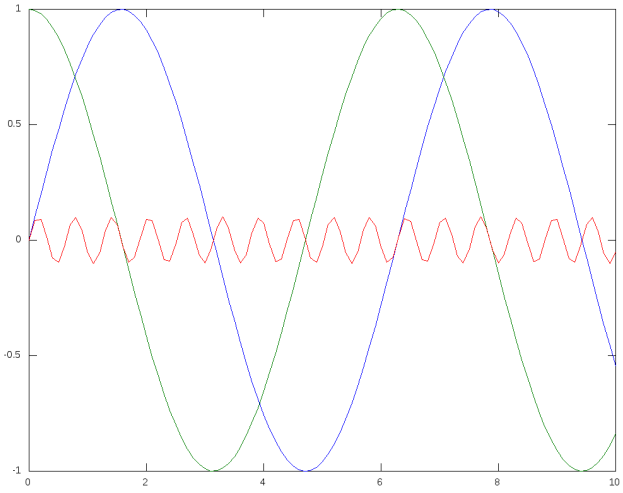



```
octave:100> x=[0:0.1:10]';
octave:101> y=sin(x);
octave:102> plot(x,y);
```



Como siempre, las columnas determinan series de valores...

```
octave:107> plot(x,[sin(x),cos(x),sin(10*x)/10]);
```



3.4 Grafico de varias series de datos

Veamos otro ejemplo: se tiene un sistema de procesamiento de formularios en batch. Se ha hecho pruebas de funcionamiento controlándose el tiempo y

el consuma de memoria del proceso para diversos tamaños del lote de datos de entrada. Los resultados se muestran en la tabla siguiente:

Tamaño de muestra	Tiempo de proceso	Memoria consumida (mb)
5000	50s	80
10000	20s	120
20000	50s	165
40000	1m 20s	310
80000	2m 40s	710
120000	3m 50s	900
150000	5m	1200

Se sabe que este proceso puede en el futuro requerir operar con lotes más grandes, por lo que se se desea obtener un gráfico que ilustre la tendencia en el tiempo y consumo de memoria.

Para esto, creamos las matrices:

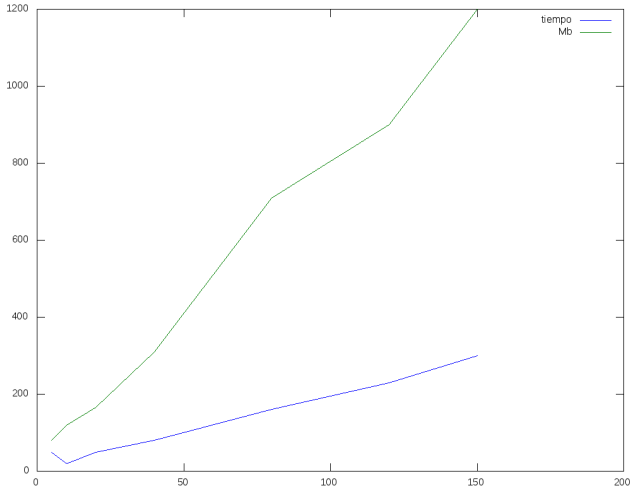
- TM = Tamaño de muestra en millares
- T = Tiempo en segundos
- M = Memoria en Mb

```
TM=[5; 10; 20; 40; 80; 120; 150];
T=[50 ; 20; 50; 60 + 20; 2*60 + 40; 3*60+50; 5*60];
M=[80; 120; 165; 310; 710; 900; 1200];
```

Finalmente lanzamos el ploteo como dos series de datos con la leyenda respectiva:

```
plot(TM,T,'-;tiempo;', TM,M,'-;Mb;');
```

Apreciar que los títulos se especifican encerrándolos entre punto y comas (`;titulo;`).



3.5 Gráficos tridimensionales

Malla de dominio (mesh) Los gráficos tridimensionales son esencialmente de dos tipos: líneas y superficies. Para el último caso, es conveniente comprender la malla de dominio (x, y) que permite llevar a cabo la graficación.

Considérese una función $z = z(x, y)$ graficada como superficie. La estrategia de graficación de Octave consiste en aproximar esta superficie mediante una "malla" de puntos representativos (más puntos en la malla, más aproximación.)

Comoquiera que los puntos z son función de los pares (x, y), la estructura de la malla se define mediante el conjunto de dichos pares.

Debemos tener en cuenta que la malla consiste no tanto de los puntos (x, y) sino de las interconexiones de éstos, del mismo modo que una red se compone de pequeñas "celdas".

Lo último significa que no es suficiente disponer de un conjunto de pares sueltos (x, y), sino de información complementaria acerca de la mutua cercanía entre éstos. De este modo se consigue pasar de un conjunto de vértices con esta estructura:

```
+         +         +
```

```
        +       +       +
```

A una malla con esta estructura:

```
      +-------+-------+
       \       \       \
        \       \       \
         +-------+-------+
```

Para esto Octave exige exige que los elementos "x" y los elementos "y" se dispongan separadamente en matrices rectangulares A y B de las mismas dimensiones. Para una misma posición (i, j), el punto (x, y) es $(A(i, j), B(i, j))$.

Gracias a esta representación, Octave puede deducir los puntos (x,y) mutuamente "adyacentes", simplemente a través del caracter adyacente de las posiciones (i, j) de las matrices de elementos.

Octave proporciona la función "meshgrid" que permite obtener rápidamente las matrices A y B. Por ejemplo, considérese el rango x en $[1, 4]$, e y en $[5, 7]$. Tenemos:

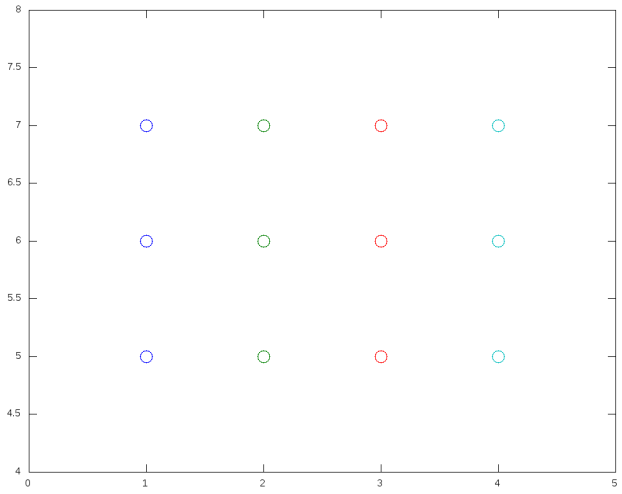
```
octave:26> [A,B]=meshgrid([1:4],[5:7])
A =

   1   2   3   4
   1   2   3   4
   1   2   3   4

B =

   5   5   5   5
   6   6   6   6
   7   7   7   7
```

Nuestra malla es de dimensión $3x4$ y tiene por tanto 12 puntos.

Los puntos generados pueden graficarse fácilmente con plot:

```
octave:26> [A,B]=meshgrid([1:4],[5:7])
A =

   1   2   3   4
   1   2   3   4
   1   2   3   4

B =

   5   5   5   5
   6   6   6   6
   7   7   7   7
```

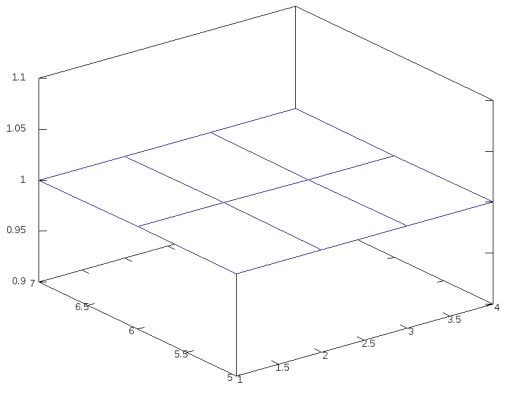
```
octave:29> plot(A,B,'.o',"markersize", 20)
octave:31> axis([0,5,4,8])
```



Pero lo interesante es utilizar estos puntos como base para una superficie. Partiremos de una superficie plana con $z = 1$:

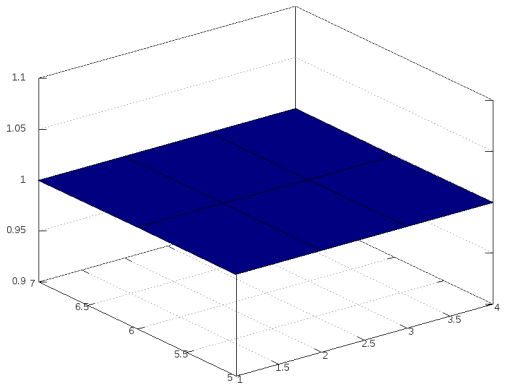
```
octave:35> Z=ones(size(A))
Z =

   1   1   1   1
   1   1   1   1
   1   1   1   1

octave:36> mesh(A,B,Z)
```

Con `mesh()` obtenemos la malla tridimensional:



O mediante `surf()`, la malla aparece coloreada (aquí de un único color dado que "Z" es constante):

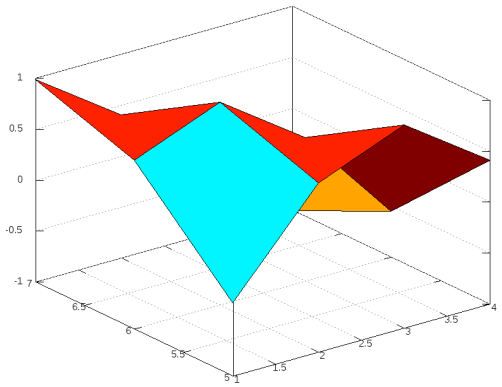
```
octave:38> surf(A,B,Z)
```



Una función un poco más compleja ilustra la necesidad de una malla más fina:

```
octave:40> Z=sin(A+B)
Z =
```

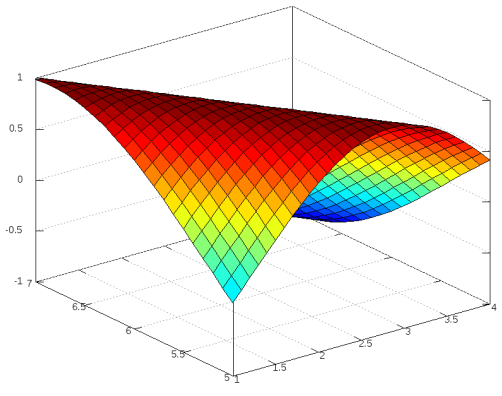
```
   -0.27942    0.65699    0.98936    0.41212
    0.65699    0.98936    0.41212   -0.54402
    0.98936    0.41212   -0.54402   -0.99999

octave:41> surf(A,B,Z)
```



Lo cual se consigue fácilmente mediante un paso más fino en el dominio:

```
octave:45> [A,B]=meshgrid([1:0.1:4],[5:0.1:7]);
octave:46> Z=sin(A+B);
octave:47> surf(A,B,Z)
```

El resultado proporciona información más útil:



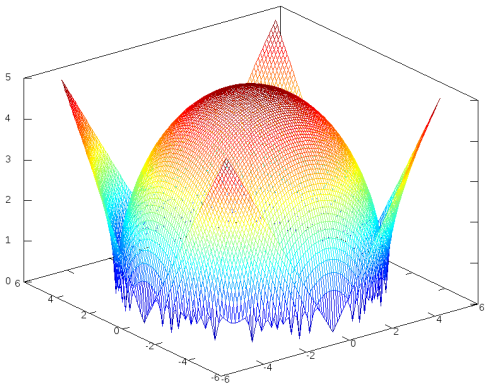
Ejercicio: Probar el clásico "sombrero" ejecutando "`sombrero`" en la línea de comandos.

3.6 Dominios más sofisticados

En ocasiones no es conveniente utilizar un dominio `rectangular(x,y)` como el definido anteriormente. Por ejemplo, para graficar un hemisferio $Z = \sqrt{R^2 - x.^2 – y^2}$ podríamos tener problemas:

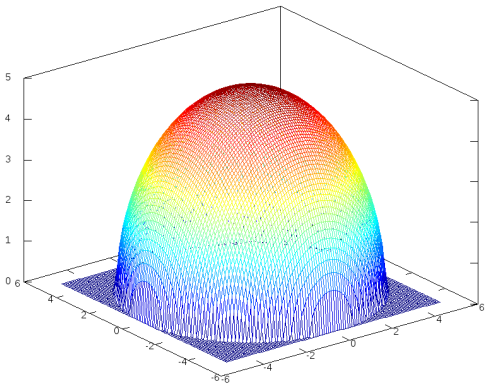
```
octave:50> [A,B]=meshgrid([-5:0.1:5],[-5:0.1:5]);
octave:51> Z=sqrt(25-A.^2-B.^2);
octave:52> mesh(A,B,Z)
error: octave_base_value::array_value(): wrong type argument ‘complex matrix’
```

Una solición inmediata podría ser tomar el valor absoluto, con lo que obtenemos una extraña figura:



Otra solución consiste en considerar sólo la parte real de Z:

```
octave:61> mesh(A,B,real(Z))
```



Aquí el único inconveniente es la malla en $Z = 0$ que permanece para los valores originalmente complejos, y esto gracias a que eran imaginarios puros por coincidencia.

Podríamos eliminar los valores complejos utilizando el valor especial NA (valor no disponible), el cual evita el gráfico de algunos puntos de la malla.

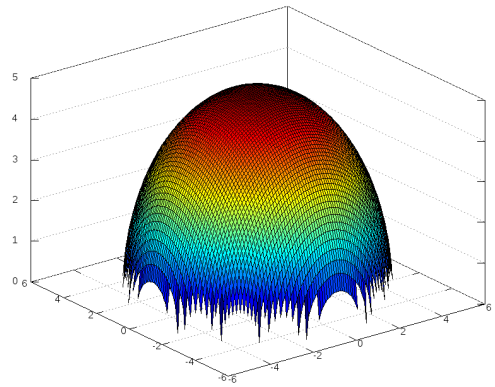
Encontremos primero los puntos "indeseables", aquellos con parte imaginaria:

```
octave:87> INDESEABLE=(imag(Z)!=0);
```

Sus elementos "1" servirán para dehabilitar valores en una copia de nuestra matriz Z:

```
octave:88> ZLIMPIO=Z;
octave:89> ZLIMPIO(INDESEABLE)=NA;
octave:90> surf(A,B,ZLIMPIO)
```

con esto obtenemos una figura sin malla en $Z = 0$, pero con problemas de continuidad en su base, debido a que la malla se ha roto sin considerar su estructura:



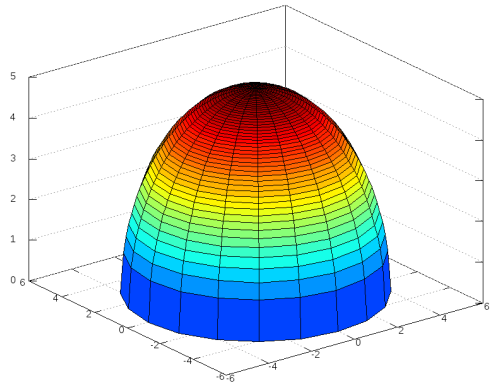
Una solución más sofisticada consiste en generar un dominio no rectangular. Para el presente caso, una representación polar puede ser más conveniente.

Para esto, redefinimos las matrices de dominio del siguiete modo:

```
octave:93> T=[0:pi/10:2*pi];
octave:94> R=[0:0.1:5];
octave:95> [AT,AR]=meshgrid(T,R);
octave:96> X=AR.*cos(AT);
octave:97> Y=AR.*sin(AT);
octave:98> Z=sqrt(25-X.^2-Y.^2);
octave:99> surf(X,Y,Z)
error: octave_base_value::array_value(): wrong type argument 'complex matrix'
...
```

Por limitaciones de precisión algunos valores han de contener valores complejos, pero han de ser muy cercanos a cero. En tal caso, podemos considerar sólo la parte real:

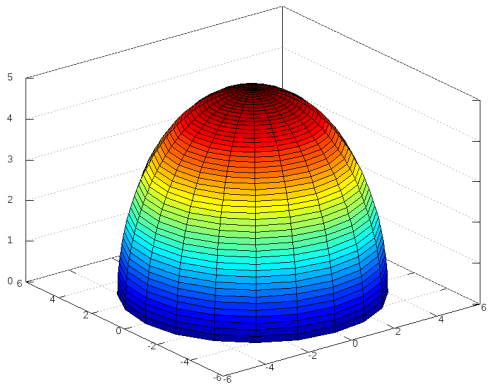
```
octave:99> surf(X,Y,real(Z))
```



Como se aprecia, la malla está más acorde a la forma de la superficie. Finalmente, podemos mejorar la variación del radio "R" a fin de evitar que en la zona superior se acumulen muchas celdas de la malla. En tanto deseamos una malla que ascienda uniformemente, podemos considerar la variación el ángulo PHI que subtiende el eje Z y cada circunferencia de la malla, desde cero a $PI/2$. El radio R en el plano corresponde al radio de la circunferencia (5) multiplicado por el seno de PHI:

```
octave:102> PHI=[0:pi/100:pi/2];
octave:103> R=5*sin(PHI);
octave:104> [AT,AR]=meshgrid(T,R);
octave:105> X=AR.*cos(AT);
octave:106> Y=AR.*sin(AT);
octave:107> Z=sqrt(25-X.^2-Y.^2);
octave:108> surf(X,Y,real(Z))
```

con esto, el gráfico resulta muy satisfactorio:



4 Integración y derivación

Por ejemplo, sea la función:

```
function [ ret ] = fun5 (x)
    ret = 4*exp(-x/2) + x.^3 / 100;
endfunction
```

Integramos en $[0, 10]$:

```
octave:16> quad("fun5",0,10)
ans =  32.946
```

Por integración exacta, tenemos:

```
# antiderivada de fun5
function [ ret ] = fun5_ad (x)
    ret = -8*exp(-x/2) + x .^ 4 / 400;
endfunction
```

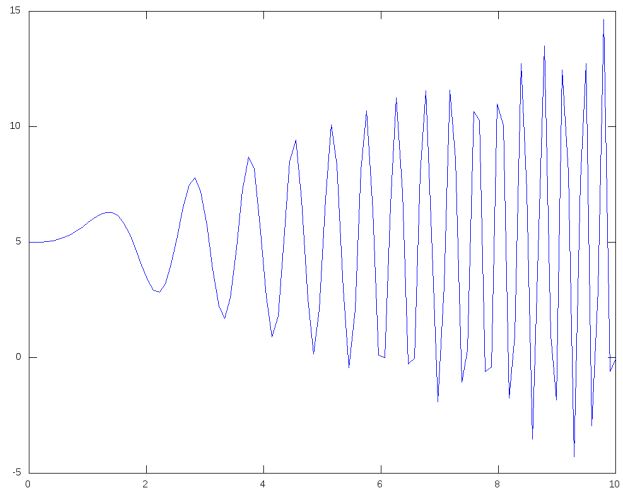
El cálculo exacto arroja:

```
octave:19> fun5_ad(10)-fun5_ad(0)
ans =  32.946
```

Integremos una función más compleja:

```
function [ ret ] = fun6 (x)
    ret = x.*sin(x.*x) + 5;
endfunction
```

Para el rango $[0-10]$, gráficamente tenemos:



Integrando con `quad`():

```
octave:60> quad("fun6",0,10)
ans =  50.069
```

De otro lado, encontrando su antiderivada:

```
octave:61> type fun6_ad

function [ ret ] = fun6_ad (x)
    ret = -cos(x*x)/2 + 5 * x;
endfunction
```

Con lo que obtenemos:

```
octave:62> fun6_ad(10)-fun6_ad(0)
ans =  50.069
```

Una integral impropia Considérese la función:

$$f(x) = \begin{cases} -log(-x+1) & 0 < x < 1 \\ -log(x–1) & 1 < x < 2 \end{cases}$$

Se requiere integrar en $[0,2]-1$, donde claramente hay una discontinuidad en $x = 1$.

Podemos definir la función así:

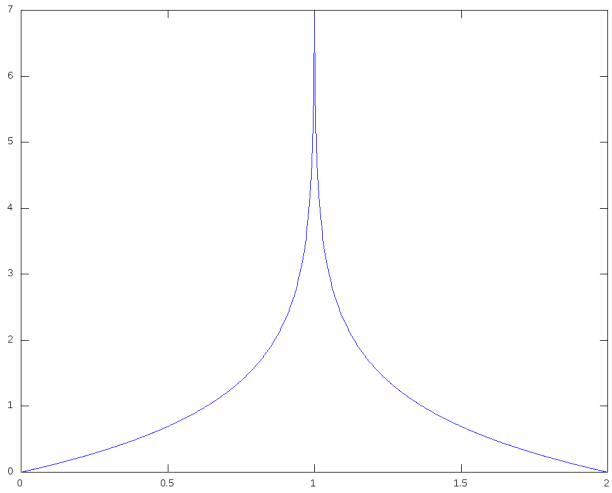
```
function ret = fun8 (t)
    ret = zeros(size(t)(1),1);
    for i = 1:size(t)
        tactual = t(i);
        if (tactual < 1)
            ret(i) = -log(-tactual + 1);
        else
            ret(i) = -log(tactual-1);
        endif
    endfor
endfunction
```

De este modo:

```
octave:91> fun8(1)
ans = Inf
```

Graficando:

```
octave:92> t=linspace(0,2,1000)';
octave:93> plot(t,fun8(t));
```



`Quad` tiene problemas:

```
octave:95> quad("fun8",0,2)
ans = Inf
```

Sorprendentemente, quad obvia los extremos, por lo que (en este ejemplo) conviene integrar en dos tramos:

```
octave:96> quad("fun8",0,1)
ans =  1.0000
```

```
octave:97> quad("fun8",1,2)
ans =  1.0000
```

Para estos casos se indica la función `quadgk`, que es más resistente a los intervalos infinitos y a las singularidades:

```
octave:98> quadgk("fun8",0,2)
ans =  2.0000
octave:99> format long
octave:100> quadgk("fun8",0,2)
ans =  1.99999801639871
```

La solución exacta es dos, como se puede calcular mediante la antiderivada del logaritmo.

Otra manera de definir la función consiste en eliminar el caso $t = 1$:

```
function ret = fun9 (t)
    ret = zeros(size(t)(1),1);
    for i = 1:size(t)
        tactual = t(i);
        if (tactual < 1)
            ret(i) = -log(-tactual + 1);
        elseif(tactual > 1)
            ret(i) = -log(tactual-1);
        else
            ret(i) = NA;
        endif
    endfor
endfunction
```

El valor especial "NA" significa ausencia de valor, y algunas funciones lo pueden aprovechar:

```
octave:5> quad("fun9",0,2)
 ABNORMAL RETURN FROM DQAGP
ans = NA
octave:6> quadgk("fun9",0,2)
ans =  2.0000
```

Derivación Para obtener las derivadas de una función empleamos "`diff`()". Con `diff` se obtiene un vector con las diferencias relativas entre sus elementos:

```
diff([x1;x2;x3;x4...xn]) = [x2-x1;x3-x2;...xn-xn-1]
```

Por ejemplo:

```
octave:2> A=[4;5;6;8;10;12;15;19];
octave:3> diff(A)
```

```
ans =
1
1
2
2
2
3
4
```

Ejemplo Hallar y graficar la derivada de $sin(x^2)$ para $[0, 6]$.

Obtenemos el dominio y la función:

```
octave:37> x=[0:0.01:6]';
octave:38> y=sin(x.*x);
```

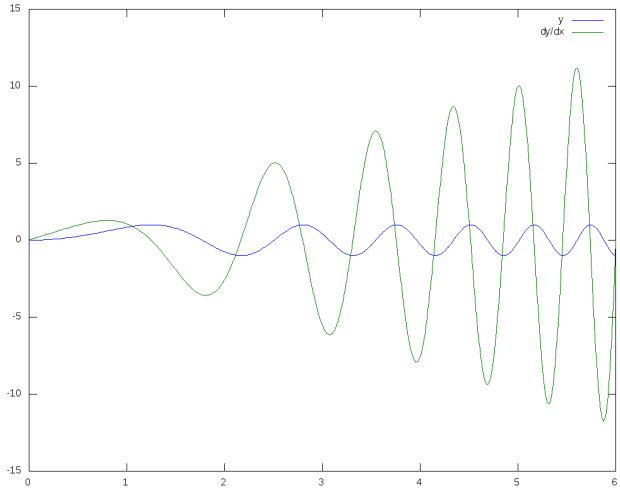
Obtenemos la derivada. Nótese que `dy1` tiene un elemento menos que "x" e "y":

```
octave:39> dy1=diff(y)./diff(x);
octave:41> size(dy1)
ans =
600 1
octave:42> size(x)
ans =
601 1
```

Graficamos ambas funciones, extendiendo la derivada con un punto arbitrario (un cero):

```
octave:46> plot(x,y,'-;y;',x,[dy1;0],'-;dy/dx;')
```

El resultado:



Como ejercicio el lector podría graficar la derivada exacta: $y' = 2x\cos(x^2)$

Ejercicio: Considérese la función $f : \mathbb{R} \to \mathbb{R}$ definida en $[0, 2)$ del siguiente modo:

para $[0,\ 1] \to f(x) = x$
para $[1,\ 1 + \frac{1}{2}] \to f(x) = 2(x - 1)$
para $[1 + \frac{1}{2},\ 1 + \frac{1}{2} + \frac{1}{4}] \to f(x) = 4(x - 1 - \frac{1}{2})$
para $[1 + \frac{1}{2} + \frac{1}{4},\ 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}] \to f(x) = 8(x - 1 - \frac{1}{2} - \frac{1}{4})$
etc.
Proporcionar una regla explícita, graficar e integrar en el rango $[0,\ 1.95]$.

Solución Primero proporcionaremos una definición explícita para $f(x)$. Lo primero es hallar una fórmula más directa para la regla anterior. Utilizando las progresiones geométricas: $S = a(r^n - 1)/(r - 1)$

$$1 + \frac{1}{2} + \frac{1}{4} + ... + 1/2^{(n-1)} = 2(1 - 1/[2^n]) = p(n)$$

así, los intervalos son:
$n = 1 \to [0,\ p(1)] \to f(x) = 2^0.(x)$
$n = 2 \to [p(1),\ p(2)] \to f(x) = 2^1.(x–p(1))$
$n = 3 \to [p(2),\ p(3)] \to f(x) = 2^2.(x–p(2))$
...
Para $n \to [p(n-1),\ p(n)] \to f(x) = 2^{(n-1)}.(x - p(n-1))$

De la expresión para $p(n)$ se puede resolver n:

$$n = -\ln(1 - p/2)/\ln(2)$$

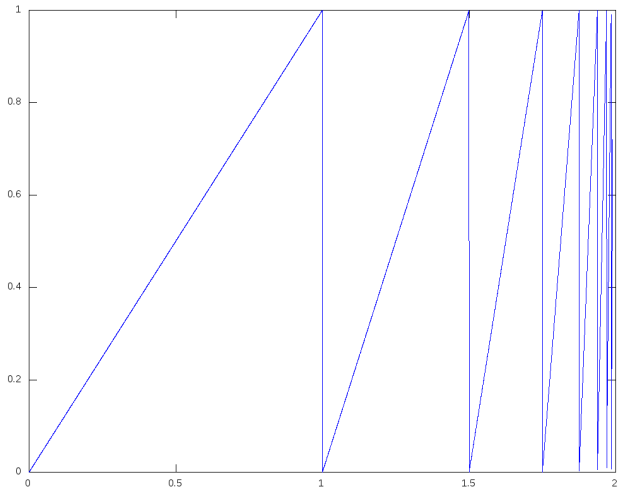
Puesto que los intervalos anteriores tienen la forma $[p(n-1),\ p(n)]$, para cualquier valor x, es factible hallar el intervalo "n" al que pertenece mediante:

```
n = floor( 1 - ln(1-x/2)/ln(2))
```

Esto permite obtener una definición de la función:

```
function [ ret ] = fun7 (x)
    n = floor(1 -log(1-x./2)/log(2));
    n = n - 1;
    p = [2*(1 - 1/(2.^n))]';
    ret = 2.^(n).*(x - p);
endfunction
```

A continuación un gráfico de la función así definida:



Con esto podemos realizar la integración:

```
octave:79> quad("fun7",0,1.95)
ans = 0.97125
```

Es interesante el ejercicio de realizar la integración mediante antiderivadas.
Al interior del intervalo n:
$[p(n-1),\ p(n)] \rightarrow f(x) = 2^{(n-1)}.(x\text{-}p(n-1))$
Su antiderivada es:
$F(x) = 2^{(n-1)}.(x^2/2\text{–}p(n-1).x) + CTE$

Con lo que la integral en el intervalo $[p(n-1), x]$ resulta:

$$SI(n,x) = 2^{(n-1)}.(x^2/2 - p(n-1).x) - 2^{(n-1)}.(p(n-1)^2/2 - p(n-1).p(n-1)) =$$

$$2^{(n-1)}.(x^2/2\text{-}p(n-1).x + p(n-1)^2/2) = 2^{(n-2)}(x - p(n-1))^2$$

Para calcular el total para un intervalo, es fácil calcular el área de cada triángulo; puesto que la altura es constante, el área es la mitad de la base, es decir, $\frac{1}{2}.(1/(2^{(n-1)}) = 1/(2^n)$; otra forma consiste en usar la fórmula anterior usando $x = p(n)$.

Asimismo, el área total incluyendo los primeros n intervalos es: $1/2 + 1/4 + ... + 1/(2^n)$. Por la fórmula anteriormente enunciada para progresiones geométricas: $ST(n) = 1 - 1/(2^n)$

Esta fórmula tiende a 1 conforme n tiende a infinito, lo cual coincide con la intuición pues la curva siempre está llenando la mitad del rectángulo del gráfico de $2x1$.

La integral de 0 a x vendrá dada por: $SI(n,x) + ST(n-1) = 2^{(n-2)}(x - p(n-1))^2 + 1 - 1/(2^n) =$

$$2^{(n-2)}(x - 2(1 - 1/[2^{n-1}]))^2 + 1 - 1/(2^{n-1})$$

Para el caso de $x = 1.95$, el intervalo que contiene a x es:

```
n = floor(1-ln(1-1.95/2)/ln(2)) = 6
```

Reemplazando el valor de n, tenemos:

```
octave:49> 2.^(n-2)*(x - 2*(1 - 1./[2.^(n-1)]))^2 + 1 - 1./(2.^(n-1))
ans =  0.97125
```

5 Polinomios

Se representan como una matriz de coeficientes que corresponden a potencias decrecientes. Por ejemplo:

$$f(x) = x^2 - 5x + 6$$

Se representa por:

```
P = [ 1 , -5 , 6];
```

Para evaluar $f(7)$:

```
octave:11> polyval(P,7)
ans = 20
```

Sus raíces:

```
octave:12> roots(P)
ans =
3
2
```

También puede retornar raíces complejas:

```
octave:13> roots([1,1,1])
ans =
-0.50000 + 0.86603i
-0.50000 - 0.86603i
octave:14> roots([1,1,1,1,1])
ans =
0.30902 + 0.95106i
0.30902 - 0.95106i
-0.80902 + 0.58779i
-0.80902 - 0.58779i
```

Producto de dos polinomios:

```
octave:29> conv([1,1,1],[2,0,1])
ans =
2 2 3 1 1
```

Para encontrar el máximo común divisor se utiliza `polygcd()`. A modo de ejemplo, simplifiquemos la expresión:

$$E(x) = \frac{x^6 + 4x^5 - 5x^4 - 3x^3 + 5x^2 - 6x + 18}{2x^6 + 4x^5 - 10x^4 - 7x^3 + 19x^2 + 2x - 3}$$

Usando Octave:

```
octave:41> R=[ 1 , 4 , -5 , -3 , 5 , -6 , 18];
octave:42> S=[2 , 4 , -10 , -7 , 19 , 2 , -3];
octave:43> G=polygcd(R,S)
G =
1 -1 -2 3
```

Para dividir polinomios utilizamos "`residue()`". El cociente viene en "k":

```
octave:55> [r, p, k, e] = residue (R, G);
octave:56> k
k =
1 5 2 6
```

Para estar seguros de que la división fue exacta, vemos que el residuo es cero:

```
octave:57> r
r =
-0 - 0i
-0 - 0i
0 + 0i
```

Del mismo modo:

```
octave:58> [r2, p2, k2, e2] = residue (S, G);
octave:59> k2
k2 =
2 6 0 -1
octave:60> r2
r2 =
-0 - 0i
-0 - 0i
0 + 0i
```

Por tanto:

$$E(x) = \frac{x^3 + 5x^2 + 2x + 6}{2x^3 + 6x^2 - 1}$$

5.1 Derivada e integral de polinomios:

Se utilizan las funciones `polyderiv()` y `polyint()`:

```
octave:65> polyderiv([1,5,0,0])
ans =
3 10 0
octave:66> polyint([3,10,0])
ans =
1 5 0 0
```

Una representación más "estándar" de los polinomios para efectos de visualización se obtienen con "`polyout()`":

```
octave:1> polyout([1, 6, -6, -5])
1*s^3 + 6*s^2 - 6*s^1 - 5
octave:2> polyout([1, 6, -6, -5], 'x')
1*x^3 + 6*x^2 - 6*x^1 - 5
```

5.2 Ajuste de datos por interpolación de polinomios

Empezaremos "fabricando" un ejemplo de problema a interpolar. Partimos de un polinomio cualquiera:

```
octave:73> P=[1.1, -5, 6.5]
P =
1.1000  -5.0000  6.5000
```

y generamos un conjunto de valores:

```
octave:74> x=linspace(0,5,10);
octave:75> polyval(P,x)
ans =
6.50000  4.06173  2.30247  1.22222  0.82099  1.09877  2.05556  3.69136  6.00617  9.00000
```

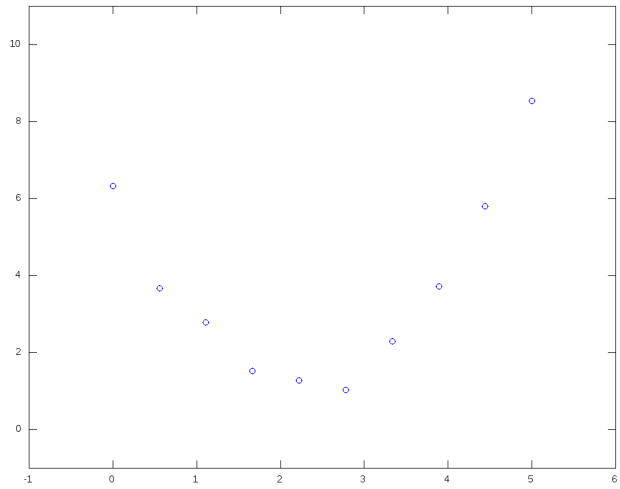
Generamos algo de "ruido" para agregar a los valores anteriores:

```
octave:77> R=(rand(10,1)-0.5)'
R =

  -0.177775  -0.396235   0.490894   0.296503   0.461531  -0.060640
0.237439   0.024887  -0.207178  -0.458858
```

Y obtenemos nuestra "data inicial":

```
octave:78> DATA=polyval(P,x) + R
DATA =

   6.3222   3.6655   2.7934   1.5187   1.2825   1.0381   2.2930   3.7162
5.7990   8.5411

octave:82> plot(x,DATA,'o', "markersize",10)
octave:83> axis([-1,6,-1,11])
```

A partir de estos datos trataremos de obtener un polinomios que la represente:



Buscaremos un polinomio de grado dos:

```
octave:86> PFIT=polyfit(x,DATA,2)
PFIT =
0.99264  -4.50941  6.23895
```

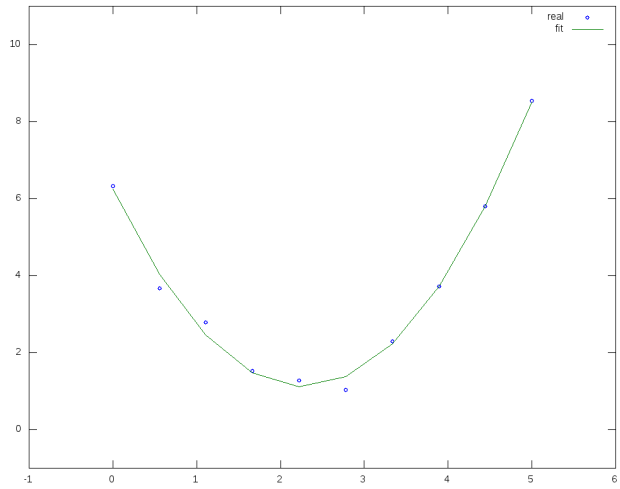
Compárese `PFIT` con P:

```
octave:87> P
P =
1.1000  -5.0000  6.5000
```

Como se aprecia, hay bastante similitud, pero nada más. En general, cuantos más puntos deberíamos esperar una mayor similitud.

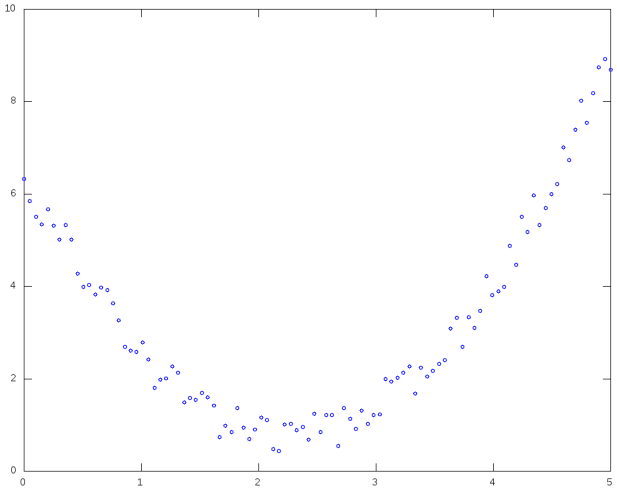
Observemos cómo se ajusta el polinomio a la data:

```
octave:88> plot(x,DATA,'o;real;',x,polyval(PFIT,x),'-;fit;')
octave:89> axis([-1,6,-1,11])
```



Probemos con más puntos:

```
octave:91> x=linspace(0,5,100);
octave:95> R=(rand(100,1)-0.5)';
octave:96> DATA=polyval(P,x) + R;
octave:97> plot(x,DATA,'o')
```

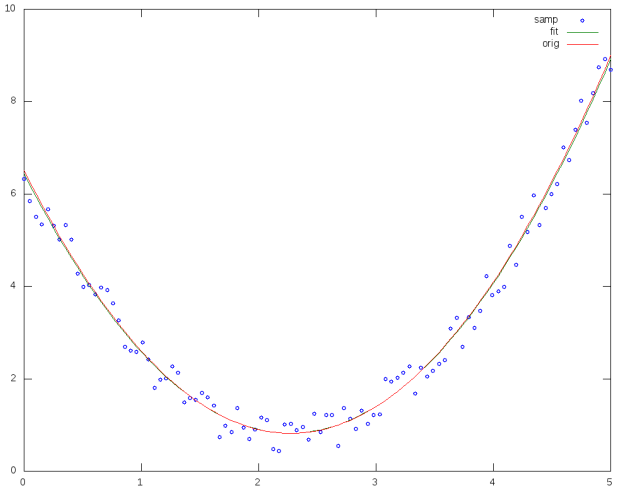


El ajuste es mucho más cercano:

```
octave:99> PFIT=polyfit(x,DATA,2)
PFIT =
1.0868  -4.9362  6.4287
```

Finalmente, veamos el gráfico que ilustra los puntos de la "muestra", el polinomio ajustado, así como el polinomio que sirvió como base para crear la muestra:

```
octave:102> plot(x,DATA,'o;samp;',x,polyval(PFIT,x),'-;fit;',x,polyval(P,x),'-;orig;')
```



6 Ecuaciones diferenciales

Ejemplo ilustrativo

Obtener $f(1)$ y $f(10)$ dada la ecuación diferencial:

$$\frac{df}{dx} = x + cos(x)$$

con condición inicial $f(0) = 1$.

Este ejemplo es trivial. Por integración directa es evidente que $f = x^2/2 + sin(x) + 1 \rightarrow f(1) = 2,34$; $f(10) = 50,456$

Mediante Octave, debemos especificar una función tal que retorne $df/dx = E(f, x)$:

```
function [ ret ] = xfun (f,x)
    ret = x + cos(x)
endfunction
```

Definiremos la condición inicial `f0=1`, además de crear 101 puntos (matriz columna) en el rango $[0-10]$, lo cual corresponde a una longitud de intervalo de $1/100$; esto permite obtener un punto exactamente en $x = 1$. Finalmente se resuelve la ecuación usando `lsode`():

```
f0=1;
x=linspace(0,10,101)';
```

```
f=lsode("xfun", f0, x);
```

Los valores vienen dados en "`x`" y "`f`":

```
[x,f]
ans =
0.00000 1.00000
0.10000 1.10483
...
0.80000 2.03736
0.90000 2.18833
1.00000 2.34147
1.10000 2.49621
...
9.70000 47.77323
9.80000 48.65352
9.90000 49.54746
10.00000 50.45598
```

Es decir, mediante Octave hallamos que $f(1) = 2.34147$ y $f(10) = 50.45598$. Esta aproximación no requirió de antiderivadas.

Nota También se podría haber utilizado un puntero a la función `xfun()`, lo que puede ser necesario en aplicaciones más sofisticadas.

```
f = lsode(@xfun, f0, x);
```

A continuación otro ejemplo que incluye a la propia función como parte de la ecuación diferencial. En realidad, se trata de la misma función del ejemplo anterior ($f = x^2/2 + sin(x) + 1$) pero con otra ecuación diferencial:

$$\frac{df(x)}{dx} = \cos(x) + \sqrt{2f(x) - 2\sin(x) - 2}$$

```
function [ ret ] = xfun2 (f,x)
    ret = cos(x) + sqrt(2 * f - 2 * sin(x) - 2);
endfunction
```

con la misma condición de frontera $f(0) = 1$. Notar que ahora no es factible realizar una antiderivación. Siguiendo el mismo procedimiento de Octave obtenemos:

```
octave:3> f=lsode("xfun2", f0, x);
warning: lsode: ignoring imaginary part returned from user-supplied function
octave:4> [x,f]
ans =

    0.00000    1.00000
```

```
...
    0.80000    1.71736
    0.90000    1.78332
    1.00000    1.84145
    1.10000    1.89119
    1.20000    1.93202
    1.30000    1.96354
...
    9.70000    0.72822
    9.80000    0.63350
    9.90000    0.54244
   10.00000    0.45596
```

Como se aprecia, el resultado para $x = 1$ y $x = 10$ son 1.84145 y 0.45596, lo que dista tremendamente de los valores correctos hallados en el ejemplo anterior (no olvidar que se trata de la misma función.) El mensaje de alerta (`warning: lsode: ignoring imaginary part...`) nos da una pista del problema: es claro que en los alrededores de $x = 0$ la raíz cuadrada puede recibir valores negativos, lo que nos sugiere evitar este punto inicial. Si partimos de $x = \delta > 0$ es razonable suponer que se pueden evitar los valores negativos en el radical en tanto $f(x)$ debería crecer más velozmente que $\sin(x)$. Sin embargo, no disponemos de la condición inicial $f(\delta)$, por lo que debemos plantear una aproximación usual:

$$f(\delta) \approx f(0) + \frac{df(0)}{dx}\delta$$

Afortunadamente el cálculo de $\frac{df(0)}{dx}$ es inmediato partiendo de la misma ecuación diferencial:

$$\frac{df(0)}{dx} = \cos(0) + \sqrt{2f(0) - 2\sin(0) - 2} = 1$$

Luego $f(\delta) \approx 1 + \delta$. Tomando $\delta = 0.1$, hemos considerado ahora 100 nodos (99 intervalos) a fin de que el se "capture" el punto $x = 1$:

```
octave:28> delta=0.1; f0=1 + delta;
octave:29> x=linspace(delta,10,100)';
octave:30> f=lsode("xfun2", f0,x); [x,f]
ans =

    0.10000    1.10000
...
    0.80000    1.97530
    0.90000    2.11810
    1.00000    2.26307
```

```
    1.10000    2.40963
    1.20000    2.55728
    1.30000    2.70563
....
    9.70000   46.98363
    9.80000   47.85574
    9.90000   48.74151
   10.00000   49.64185
```

Este resultado está mucho más cerca de la respuesta correcta, y ha desaparecido el mensaje de alerta. Una mayor exactitud se puede conseguir elevando el número de intervalos así como reduciendo δ a 0.01:

```
octave:31> delta=0.01; f0=1 + delta;
octave:32> x=linspace(delta,10,1000)';
octave:33> f=lsode("xfun2", f0,x); [x,f]
ans =

    0.010000    1.010000
...
    0.980000    2.301502
    0.990000    2.316786
    1.000000    2.332087
    1.010000    2.347404
    1.020000    2.362736
...
    9.970000   50.087881
    9.980000   50.179012
    9.990000   50.270297
   10.000000   50.361735
```

En este último ejemplo se puede visualizar los resultados con más comodidad así:

```
octave:42> f(find( (x==1) | (x == 10)))
ans =

    2.3321
   50.3617
```

o también:

```
octave:43> f(lookup(x,[1,10]))
ans =

    2.3321
   50.3617
```

6.1 Ecuaciones diferenciales de orden superior

Sea la ecuación diferencial sobre $x(t)$:

$$100x'' + x = t^2 + sin(t/10) – 7t + 204$$

con condiciones iniciales $x(0) = 4\ ;\ x'(0) = -6.9$
Previamente, analicemos su solución exacta:

$$x(t) = t^2 + sin(t/10) – 7t + 4$$

En el rango $[0, 10]$ podemos obtener algunos valores representativos:

```
function [ ret ] = xfun3_sol (t)
    ret = t.*t + sin(t/10) - 7 * t + 4;
endfunction

octave:280> [[0:10]', xfun3_sol([0:10]')]
ans =

    0.00000    4.00000
    1.00000   -1.90017
    2.00000   -5.80133
    3.00000   -7.70448
    4.00000   -7.61058
    5.00000   -5.52057
    6.00000   -1.43536
    7.00000    4.64422
    8.00000   12.71736
    9.00000   22.78333
   10.00000   34.84147
```

Haciendo $x \to x_1$ y $x' = x_1' = x_2$, reescribimos la ecuación:

$$100x_2' + x_1 = t^2 + sin(t/10) – 7t + 204$$

es decir, formamos el sistema:

$$\begin{cases} x_2' = (t^2 + sin(t/10) – 7t + 204 – x_1)/100 \\ x_1' = x_2 \end{cases}$$

Así, tenemos dos ecuaciones de primer orden de la forma $df/dt = E(f, t)$. Este sistema de ecuaciones se procesa del mismo modo que el caso anterior:

```
function [ ret ] = xfun3 (x, t)
    ret = zeros(2,1);
    ret(1) = x(2);
```

```
    ret(2) = (t * t - 7 * t + 204 - x(1))/100;
endfunction

octave:285> f0=[4,-6.9]';
octave:286> t=linspace(0,10,101)';
octave:287> f=lsode("xfun3", f0, t);
```

Visualizamos los resultados en los puntos correspondientes:

```
octave:290> [[0:10]',f(lookup(t,[0:10]'))]
ans =

    0.00000    4.00000
    1.00000   -1.90017
    2.00000   -5.80133
    3.00000   -7.70448
    4.00000   -7.61058
    5.00000   -5.52057
    6.00000   -1.43536
    7.00000    4.64422
    8.00000   12.71736
    9.00000   22.78333
   10.00000   34.84147
```

Otro ejemplo: resolver en $y(x)$ para $x \in [0, 2]$:

$$y''' + y''–6y' = 2 + 2x - 6x^2$$

condiciones iniciales: $y(0) = -4,\ y'(0) = -10,\ y''(0) = -20.$

Para resolverlo consideramos los cambios de variable $y = y_1;\ y' = y_2;\ y'' = y_3$, y reescribimos:

$$y''' = 2 + 2x - 6x^2–y'' + 6y' = 2 + 2x - 6x^2–y_3 + 6y_2$$

Así, la función con las derivadas $[y', y'', y''']$ es:

```
function [ ret ] = xfun4 (y,t)
    ret(1) = y(2);
    ret(2) = y(3);
    ret(3) = 2 + 2 * t - 6 * (t.^2) + 6 * y(2) - y(3);
endfunction

f0=[-4,-10,-20]'
f0 =

   -4
  -10
  -20
t=linspace(0,2,101)';
f=lsode("xfun4", f0, t);
```

La solución resulta:

```
octave:339> [[0:0.1:2]',f(lookup(t,[0:0.1:2]'),1)]
ans =

     0.00000    -4.00000
     0.10000    -5.10668
     0.20000    -6.45646
     0.30000    -8.10159
     0.40000   -10.10637
     0.50000   -12.54974
     0.60000   -15.52859
     0.70000   -19.16167
     0.80000   -23.59450
     0.90000   -29.00524
     1.00000   -35.61195
     1.10000   -43.68141
     1.20000   -53.53990
     1.30000   -65.58638
     1.40000   -80.30859
     1.50000   -98.30272
     1.60000  -120.29736
     1.70000  -147.18289
     1.80000  -180.04724
     1.90000  -220.21968
     2.00000  -269.32420
```

Es interesante comparar la aproximación (2da columna) con la solución exacta (3ra):

```
function [ ret ] = xfun4_sol (t)
    ret = (t.^3)/3 - 5 * exp(2*t) + 1;
endfunction

octave:341> [[0:0.1:2]',f(lookup(t,[0:0.1:2]'),1) , xfun4_sol([0:0.1:2]')]
ans =

     0.00000    -4.00000    -4.00000
     0.10000    -5.10668    -5.10668
     0.20000    -6.45646    -6.45646
     0.30000    -8.10159    -8.10159
     0.40000   -10.10637   -10.10637
     0.50000   -12.54974   -12.54974
     0.60000   -15.52859   -15.52858
     0.70000   -19.16167   -19.16167
     0.80000   -23.59450   -23.59450
     0.90000   -29.00524   -29.00524
     1.00000   -35.61195   -35.61195
     1.10000   -43.68141   -43.68140
     1.20000   -53.53990   -53.53988
```

```
1.30000   -65.58638   -65.58636
1.40000   -80.30859   -80.30857
1.50000   -98.30272   -98.30268
1.60000  -120.29736  -120.29732
1.70000  -147.18289  -147.18283
1.80000  -180.04724  -180.04717
1.90000  -220.21968  -220.21959
2.00000  -269.32420  -269.32408
```

7 Control de flujo e interacción con el sistema

Octave proporciona sentencias de control de flujo típicas. Por ejemplo:

```
octave:21> for k = [1:10]; printf("elemento: %d\n", k); endfor
elemento: 1
elemento: 2
elemento: 3
elemento: 4
elemento: 5
elemento: 6
elemento: 7
elemento: 8
elemento: 9
elemento: 10
```

`printf()` es similar a la correspondiente en lenguaje C; también se puede emplear la menos pontente pero más sencilla `disp()`.

```
octave:32> f = 1; while f <= 5 ; disp('F es'), disp(f) ; f = f + 1; endwhile
F es
 1
F es
 2
F es
 3
F es
 4
F es
 5
```

De la mano de estas funciones tenemos "`clc`" (limpiar la pantalla) e "`input()`", que permite solicitar un valor del teclado. `Input` tiene una sintaxis inusual: cuando si se trata de obtener un texto debe agregarse un argumento 's', de lo contrario asumirá que el texto introducido es una expresión matemática a ser evaluada.

De otro lado cuando se requiere presentar un menú de opciones, se puede emplear `menu()`.

El siguiente "script" se ha almacenado en un archivo llamado `sistema_de_archivos.m`, e ilustra las funciones mencionadas:

```
while true
    clc
    disp('Sistema␣de␣Archivos');
    disp('--------------------');
    disp('');
    p = pwd;
    printf("Directorio actual: %s\n\n", p);
    o = menu('Opciones␣disponibles',
        'Cambiar␣de␣directorio',
        'Listar␣directorio',
        'Mostrar␣archivo',
        'Crear␣archivo',
        'Salir');
    salir = false;
    switch o
           case 1
        nd = input('A␣que␣directorio?', 's');
        cd(nd);
           case 2
        fls = dir('.');
        for t = [1:size(fls)]
            if ( fls(t).isdir == 1 )
                printf('[%s]\n' , fls(t).name)
            else
                disp(fls(t).name)
            endif
        endfor
           case 3
        nom = input('Nombre?', 's');
        system(['cat␣' , nom]);
           case 4
        nom = input('Nombre?', 's');
        system(['gvim␣' , nom]);
           case 5
        salir = true;
           otherwise
             error ("Opcion incorrecta");
          endswitch

    if (salir)
        break
    endif
endwhile
```

Este script ilustra lo mencionado más arriba así como el uso de la estructura `switch... case ... otherwise... endswitch`, que es similar a

la de otros lenguajes de programación.

Adicionalmente se presentan algunas funciones para interactuar con el sistema. `system()` permite lanzar un programa externo. Nótese que empleamos `['cat ', nom]` para contatenar estos textos (existe una función relacionada llamada `strcat()` que el lector puede consultar.)

La función `cd()` permite cambiar de directorio; el directorio actual se obtiene de la variable especial `pwd`.

El ejemplo anterior ilustra además la función `dir()` la cual permite obtener información acerca del contenido de un directorio, lo que se explica a continuación.

7.1 Estructuras y celdas

El operador "." permite definir miembros de estructuras:

```
octave:33> pox.a=33;
octave:34> pox.b=55;
octave:35> pox
pox =
{
  a =  33
  b =  55
}
octave:36> pox.a
ans =  33
octave:37> class(pox)
ans = struct
```

De otro lado, mientras las matrices permiten almacenar listas de valores del mismo tipo, los "cell" permiten heterogeneidad:

```
octave:45> e={'a','b',44;'j',12,'k'};
octave:46> e
e =

{
  [1,1] = a
  [2,1] = j
  [1,2] = b
  [2,2] =  12
  [1,3] =  44
  [2,3] = k
}

octave:56> e{2,1}
ans = j
```

```
octave:57> e{2,2}
ans =  12
```

El caso de `dir()` es interesante pues retorna un array de estructuras cuyos elementos son cells. Así tenemos:

```
octave:59> T=dir('/tmp');
octave:60> T
T =
{
  26x1 struct array containing the fields:

    name
    date
    bytes
    isdir
    datenum
    statinfo
}
```

Podemos obtener un elemento:

```
octave:69> T(11)
ans =
{
  name = ch6.pdf
  date = 07-Apr-2011 11:41:26
  bytes =  228752
  isdir = 0
  datenum =  7.3457e+05
  statinfo =
  {
    dev =  2053
    ino =  134432
    mode =  33024
    modestr = -r--------
    nlink =  1
    uid =  1000
    gid =  1000
    rdev = 0
    size =  228752
    atime =  1.3022e+09
    mtime =  1.3022e+09
    ctime =  1.3022e+09
    blksize =  4096
    blocks =  448
  }
}
```

y el nombre de un archivo:

```
octave:74> T(11).name
ans = ch6.pdf
```

Para almacenar la lista de nombres debemos emplear un cell array. El "cell array" puede considerarse una matriz con capacidad de almacenar elementos de distinto tipo y tamaño. En nuestro caso, los elementos son strings de distinto tamaño.

```
N={T.name};
octave:147> N
N =

{
  [1,1] = .
  [1,2] = ..
  [1,3] = .ICE-unix
  [1,4] = .X0-lock
...
```

Análogamente a las matrices, podemos acceder a sus elementos con `{fila,columna}`:

```
octave:150> N{1,12}
ans = eclipse
```

También a rangos de elementos:

```
octave:151> N{1,12:14}
ans = eclipse
ans = hsperfdata_diego
ans = jna3140732564658228979.tmp
```

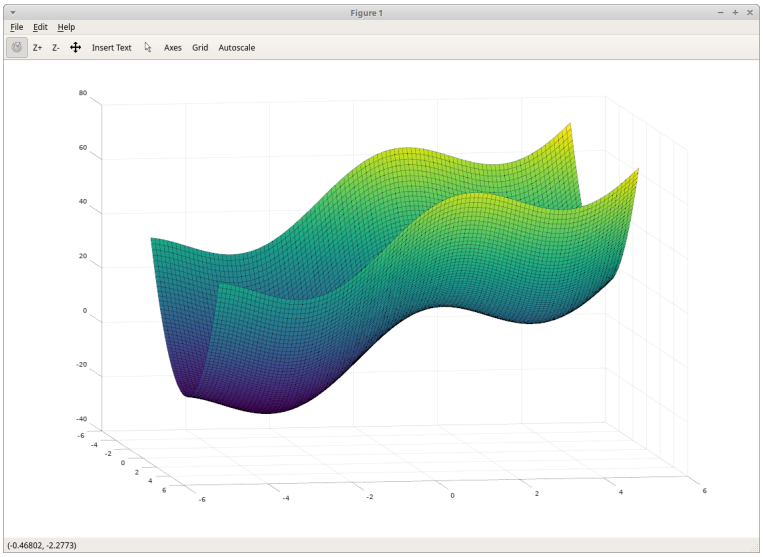
Y recorrerlos en un loop `for...endfor`:

```
octave:148> for i = N ; printf("Archivo : %s\n", i{1,1}) ; endfor
Archivo : .
Archivo : ..
Archivo : .ICE-unix
Archivo : .X0-lock
Archivo : .X11-unix
...
```

8 Optimización de funciones

Ahora considérese la siguiente función con dominio en $[-5,5] * [-5,5]$:

```
octave:112> [X,Y] = meshgrid([-5:0.1:5],[-5:0.1:5]);
octave:113> Z = 2*X.^2 + 10*sin(Y+1) + 5*Y;
octave:114> surf(X, Y, real(Z))
```



Esta función tiene claramente un mínimo local alrededor de $(0, 3)$ y un mínimo global alrededor de $(0, -4)$. Busquemos estos mínimos con Octave: la función de dos variables se registrará en `tomin.m`:

```
function [ ret ] = tomin (x,y)
    ret = 2*x.^2+10*sin(y+1)+5*y;
endfunction
```

ahora emplearemos `fminunc()`; esta función recibe como argumento una función a minimizar; sin embargo, se espera que los argumentos sean proporcionados en un vector, por lo que emplearemos una función auxiliar que extraiga los elementos del vector e invoque a `tomin()`; naturalmente, esto se puede evitar reescribiendo `tomin()` para que reciba un único argumento que representa a un vector con dos elementos.

Notar finalmente que se proporciona el "punto inicial" para la búsqueda:

```
octave:8> [x, fval, info, output, grad, hess] = fminunc(@(v) tomin(v(1),v(2)),[0;3])
x =

   0.00000
   3.18879

fval =  7.2837
info =  3
output =

  scalar structure containing the fields:

    iterations =  5
```

```
    successful =  4
    funcCount =  20

grad =

   0.0000e+00
   3.1055e-06

hess =

   1.00000   0.00000
   0.00000   8.65978
```

Este ejemplo muestra la información retornada por la función; el valor `info=3` significa que las iteraciones han llegado a un punto en que la mejora entre evaluaciones es menor que el atributo "`TolFun`" (no empleado en este ejemplo en forma explícita) que por omisión es $1e-7$.

El otro mínimo (el global) se puede encontrar partiendo desde otro punto:

```
octave:9> [x, fval, info, output, grad, hess] = fminunc(@(v) tomin(v(1),v(2)),[0;-4])
x =

   0.00000
  -3.09439

fval = -24.132
info =  3
...
```

Si no se especifican los argumentos de salida, sólo se retorna el punto mínimo:

```
octave:31> fminunc(@(v) tomin(v(1),v(2)),[0;-4])
ans =

   0.00000
  -3.09439
```

Una sintaxis alternativa empleando `num2cell()` para el mismo ejemplo va a continuación:

```
octave:32> fminunc(@(v)tomin(num2cell(v){:}),[0;-4])
ans =

   0.00000
  -3.09439
```

8.1 Optimización con sqp()

El mismo resultado puede ser obtenido con `sqp()`, la cual permite agregar restricciones a la optimización; observar que el punto inicial es el primer argumento:

```
octave:36> [x, obj, info, iter, nf, lambda] = sqp([0;3],@(v) tomin(num2cell(v){:}))
x =

   0.00000
   3.18879

obj =  7.2837
info =  104
iter =  5
nf =  9
lambda = [](0x1)

octave:40> [x, obj, info, iter, nf, lambda] = sqp([0;-4],@(v) tomin(num2cell(v){:}))
x =

   0.00000
  -3.09440

obj = -24.132
info =  101
iter =  6
nf =  9
lambda = [](0x1)
```

Para apreciar el poder de `sqp()`, obsérvese lo que ocurre con otro punto inicial $(0, 0)$:

```
octave:38> [x, obj, info, iter, nf, lambda] = sqp([0;0],@(v) tomin(num2cell(v){:}))
x =

  -6.3853e-08
  -9.3776e+00

obj = -55.548
info =  104
iter =  10
nf =  14
lambda = [](0x1)
```

El mínimo hallado corresponde (aproximadamente) a $(0, -9.37)$, el cual está fuera del dominio que hemos definido inicialmente. La función `sqp()` permite especificar valores máximos y mínimos para las coordenadas del mínimo:

```
octave:41> [x, obj, info, iter, nf, lambda] = sqp([0;0],
           @(v) tomin(num2cell(v){:}),[],[],[-5;-5],[5;5])
x =

   0.00000
  -3.09440

obj = -24.132
info =  101
iter =  9
nf =  12
lambda =
   0
   0
   0
   0
```

Observar que hemos pasado matrices vacías en los argumentos tercero y cuarto; los valores mínimos van en el quinto argumento, y los máximos en el sexto.

Una forma más general consiste en emplear el cuarto argumento, proporcionándole una función que retorna un vector cuyos valores deben ser mayores que cero (restricciones para el espacio de búsqueda); como nuestro dominio es cuadrado, este equivale a las desigualdades $-5 < x < 5$, es decir $|x| < 5$, equivalente a $5 - |x| > 0$ (análogamente para la coordenada "y"); esto se puede escribir en una función del siguiente modo:

```
octave:55> constr=@(v) [5-abs(v(1));5-abs(v(2))];
```

y se proporciona como cuarto argumento a Octave (notar que ya no empleamos ni el quinto ni el sexto):

```
octave:56> [x, obj, info, iter, nf, lambda] = sqp([0;0],
           @(v) tomin(num2cell(v){:}),[],constr)
x =

   1.9378e-08
  -3.0944e+00

obj = -24.132
info =  104
iter =  12
nf =  22
lambda =

   0
   0
```

9 Cálculo Simbólico

Octave proporciona la extensión "Symbolic Package" para ejecutar operaciones de forma simbólica; esto puede requerir de un proceso de instalación complementario[11].

Una vez instalado el paquete "symbolic", éste debe cargarse:

```
octave:1> pkg load symbolic
```

Las variables empleadas para cálculo simbólico deben especificarse en forma explícita. Por ejemplo:

```
octave:2> syms x a
```

Una ecuación tal como

$$2x - 5a = 0$$

puede resolverse simbólicamente mediante:

```
octave> solve(2*x-5*a==0,x)
ans = (sym)
  5.a
  ---
   2
```

9.1 Derivación e Integración

A partir de variables simbólicas, es posible definir funciones en forma simbólica. Mediante `diff()` e `int()` es posible calcular derivadas e integrales:

```
octave> f=3*log(5*x+1)
f = (sym) 3.log(5.x + 1)
octave> g=diff(f)
g = (sym)

      15
  -------
  5.x + 1

octave> h=sin(x)^3+cos(4*x)
h = (sym)
```

[11] En Linux Debian 12 viene dado por el paquete `octave-symbolic`; notar que el año 2021 este paquete tuvo problemas de compilación con sus dependencias (ver `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=980707`), lo que afectó a distribuciones dependientes; por ejemplo Ubuntu 22.04.

```
     3
  sin (x) + cos(4.x)

octave> int(h)
ans = (sym)

                  3
  sin(4.x)   cos (x)
  -------- + ------- - cos(x)
     4          3
```

9.2 Matrices simbólicas

La operación con matrices también se puede efectuar simbólicamente. El siguiente ejemplo define una matriz de rotación en el plano y se calcula su inversa. Esta inversa es luego explícitamente simplificada:

```
octave> syms theta
octave> rot=[cos(theta),-sin(theta);sin(theta),cos(theta)]
rot = (sym 2x2 matrix)

  |cos(theta)  -sin(theta)|
  |                       |
  |sin(theta)  cos(theta) |

octave> rotinv=inv(rot)
rotinv = (sym 2x2 matrix)

  |       2                   |
  |1 - sin (theta)            |
  |---------------  sin(theta)|
  |   cos(theta)              |
  |                           |
  |  -sin(theta)    cos(theta)|

octave> rotinv=simplify(rotinv)
rotinv = (sym 2x2 matrix)

  |cos(theta)   sin(theta)|
  |                       |
  |-sin(theta)  cos(theta)|
```

9.3 Substitución de Valores

La función subs() permite efectuar la substitución de valores en forma efectiva:

```
octave> subs(rot,theta,sym(pi)/6)
ans = (sym 2x2 matrix)

  |\/3       |
  |--    -1/2|
  |2         |
  |          |
  |      \/3 |
  |1/2   --- |
  |       2  |

octave> subs(rot,theta,sym(pi)/7)
ans = (sym 2x2 matrix)

  |    pi       pi |
  |cos(--) -sin(--)|
  |    7        7  |
  |                |
  |    pi      pi  |
  |sin(--) cos(--) |
  |    7       7   |
```

Este ejemplo ilustra el uso de `sym(.)` para definir constantes. Es importante notar que debe emplearse `sym(pi)/7` y no `sym(pi/7)`, pues en el último caso se efectuará la división (en punto flotante) la cual podría perder su forma "simbólica".

El siguiente ejemplo ilustra que la sustitución puede ser mediante nuevas expresiones simbólicas más complejas:

```
octave> k=subs(rot,theta,x+theta)
k = (sym 2x2 matrix)

  |cos(theta + x)  -sin(theta + x)|
  |                               |
  |sin(theta + x)  cos(theta + x) |

octave> kinv=subs(rotinv,theta,x+theta)
kinv = (sym 2x2 matrix)

  |cos(theta + x)   sin(theta + x)|
  |                               |
  |-sin(theta + x)  cos(theta + x)|

octave> k*kinv
```

```
ans = (sym 2x2 matrix)

  |   2                  2                                          |
  |sin (theta + x) + cos (theta + x)              0                 |
  |                                                                 |
  |                                 2                    2          |
  |            0              sin (theta + x) + cos (theta + x)|

octave> simplify(k*kinv)
ans = (sym 2x2 matrix)

  |1  0|
  |    |
  |0  1|
```