


Herramientas Computacionales para Matemática Aplicada

Curso 2020



**Elementos de programación
en bash**

Cuántos lenguajes de programación hay? ...

A [\[edit \]](#)

- [A# .NET](#)
- [A-0 System](#)
- [A+](#)
- [A++](#)
- [ABAP](#)
- [ABC](#)
- [ABC ALGOL](#)
- [ACC](#)
- [Accent](#)
- [Ace DASL \(Distributed Application Specification Language\)](#)
- [Action!](#)
- [ActionScript](#)
- [Actor](#)
- [Ada](#)
- [Adenine](#)
- [Agda](#)
- [Agilent VEE](#)
- [Agora](#)
- [AIMMS](#)
- [Aldor](#)
- [Alef](#)
- [ALF](#)
- [ALGOL 58](#)
- [ALGOL 60](#)
- [ALGOL 68](#)
- [ALGOL W](#)
- [Alice](#)
- [Alma-0](#)
- [AmbientTalk](#)
- [Amiga E](#)
- [AMOS](#)
- [AMPL](#)
- [AngelScript](#)
- [Apex](#)
- [APL](#)
- [App Inventor for Android's visual block language](#)
- [AppleScript](#)
- [APT](#)
- [Arc](#)
- [ARexx](#)
- [Argus](#)
- [Assembly language](#)
- [AutoHotkey](#)
- [AutoLISP / Visual LISP](#)
- [Averest](#)
- [AWK](#)
- [Axum](#)

B [\[edit \]](#)

- [B](#)
- [Babbage](#)
- [Ballerina](#)
- [Bash](#)
- [BASIC](#)
- [bc](#)
- [BCPL](#)
- [BeanShell](#)
- [Batch file \(Windows/MS-DOS\)](#)
- [Bertrand](#)
- [BETA](#)
- [BLISS](#)
- [Blockly](#)
- [BlooP](#)
- [Boo](#)
- [Boomerang](#)
- [Bosque](#)
- [Bourne shell \(including bash and ksh\)](#)

C [\[edit \]](#)

- [C](#)
- [C-- \(C minus minus\)](#)
- [Ch](#)
- [Chapel](#)
- [CLU](#)
- [CMS-2](#)
- [Coq](#)
- [Coral 66](#)

Lenguajes de alto y bajo nivel:

```
org 100h  
mov msg[2], 34H  
mov dx, offset msg  
mov ah, 9  
int 21h  
ret  
msg db "Hola, Mundo $"
```



Ej. programa “Hola Mundo” en ensamblador. El ensamblador es un lenguaje de bajo nivel y el código depende de la arquitectura de procesador.

```
print "Hola, Mundo"
```



Ej. programa “Hola Mundo” en Python. Python es un lenguaje de alto nivel y el código es independiente de la arquitectura de procesador, es **portable**.

Lenguajes compilados e interpretados:

Un **lenguaje interpretado** es un **lenguaje de programación** para el que la mayoría de sus implementaciones ejecuta las instrucciones directamente, sin una previa **compilación** del programa a instrucciones en **lenguaje máquina**. El **intérprete** ejecuta el programa directamente, traduciendo cada sentencia en una secuencia de una o más subrutinas ya compiladas en código máquina.

Un lenguaje compilado es aquel cuyo código fuente, escrito en un lenguaje de alto nivel, es traducido por un compilador a un archivo ejecutable entendible para la máquina en determinada plataforma. Con ese archivo se puede ejecutar el programa cuantas veces sea necesario sin tener que repetir el proceso por lo que el tiempo de espera entre ejecución y ejecución es ínfimo.

Aspectos generales básicos de un lenguaje de programación:

Un lenguaje de programación debe permitir, como mínimo:

- Definir variables
- Definir funciones
- Introducir comentarios en el código
- Incluir mecanismos de control de flujo (estructuras condicionales)
- Incluir mecanismos para realizar ciclos (iteraciones).

Condicionales [\[editar \]](#)

Las sentencias condicionales son estructuras de código que indican que, para que cierta parte del programa se ejecute, deben cumplirse ciertas premisas; por ejemplo: que dos valores sean iguales, que un valor exista, que un valor sea mayor que otro... Estos condicionantes por lo general solo se ejecutan una vez a lo largo del programa. Los condicionantes más conocidos y empleados en programación son:

- **If:** Indica una condición para que se ejecute una parte del programa.
- **Else if:** Siempre va precedido de un "If" e indica una condición para que se ejecute una parte del programa siempre que no cumpla la condición del if previo y sí se cumpla con la que el "else if" especifique.
- **Else:** Siempre precedido de "If" y en ocasiones de "Else If". Indica que debe ejecutarse cuando no se cumplan las condiciones previas.

Bucles [\[editar \]](#)

Los bucles son parientes cercanos de los condicionantes, pero ejecutan constantemente un código mientras se cumpla una determinada condición. Los más frecuentes son:

- **For:** Ejecuta un código mientras una variable se encuentre entre 2 determinados parámetros.
- **While:** Ejecuta un código mientras que se cumpla la condición que solicita.

Hay que decir que a pesar de que existan distintos tipos de bucles, todos son capaces de realizar exactamente las mismas funciones. El empleo de uno u otro depende, por lo general, del gusto del programador.

Funciones [\[editar \]](#)

Las funciones se crearon para evitar tener que repetir constantemente fragmentos de código. Una función podría considerarse como una variable que encierra código dentro de si. Por lo tanto, cuando accedemos a dicha variable (la función) en realidad lo que estamos haciendo es ordenar al programa que ejecute un determinado código predefinido anteriormente.

Todos los lenguajes de programación tienen algunos elementos de formación primitivos para la descripción de los datos y de los procesos o transformaciones aplicadas a estos datos (tal como la suma de dos números o la selección de un elemento que forma parte de una colección). Estos elementos primitivos son definidos por reglas sintácticas y semánticas que describen su estructura y significado respectivamente.

Historia de bash:

Bash

GNU Bash o simplemente **Bash** (**B**ourne-again **s**hell) es un lenguaje de comandos y **shell de Unix** escrito por **Brian Fox** para el **Proyecto GNU** como un reemplazo de **software libre** para el **shell Bourne**.^{1 2} Lanzado por primera vez en 1989,³ se ha utilizado ampliamente como el shell de **inicio de sesión** predeterminado para la mayoría de las distribuciones de **Linux** y **MacOS Mojave** de **Apple** y versiones anteriores. Una versión también **está disponible para Windows 10** y **Android**.⁴ También es el shell de usuario predeterminado en **Solaris 11**.⁵

Bash es un **procesador de comandos** que generalmente se ejecuta en una **ventana de texto** donde el usuario escribe comandos que causan acciones. Bash también puede leer y ejecutar comandos desde un archivo, llamado **script de shell**. Al igual que todos los shells de Unix, es compatible con el **agrupamiento** de nombres de archivo (coincidencia de comodines), **tuberías**, **here documents**, **sustitución de comandos**, **variables** y **estructuras de control** para pruebas de **condición** e **iteración**. Las **palabras reservadas**, la **sintaxis**, las **variables de ámbito dinámico** y otras características básicas del lenguaje se copian de **sh**. Otras características, por ejemplo, **el historial**, se copian de **csh** y **ksh**. Bash es un shell compatible con **POSIX**, pero con varias extensiones.

El nombre de la shell es un acrónimo de Bourne-again shell, un juego de palabras con el nombre de la shell de Bourne que reemplaza⁶ y la noción de "nacer de nuevo".^{7 8}



Tareas del shell:

The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program. Shell scripts are interpreted, not compiled. The shell reads commands from the script line per line and searches for those commands on the system (see [Section 1.2](#)), while a compiler converts a program into machine readable form, an executable file - which may then be used in a shell script.

Apart from passing commands to the kernel, the main task of a shell is providing a user environment, which can be configured individually using shell resource configuration files.

Estructuras condicionales IF - ELSE :

```
#!/bin/bash
#Test IF-ELSE

#valor "a adivinar" por el usuario ...
#Importante: al definir variables, no dejar espacios en blanco a ninguno de
#los lados del signo '='
Var=15

echo ' Adivina el valor numérico de la variable: Introduce un entero'
read A
echo "'Valor leído: $A'"
echo -e "\n"
if [ $A -eq $Var ]; then
echo 'Acertaste!'
elif [[ $A -le ${Var+3} && $A -gt $Var ]]; then

echo 'Estuviste cerca, pero te pasaste...'
elif [[ $A -ge ${Var-3} && "$A" -lt $Var ]]; then

echo 'Estuviste cerca, pero te faltó un poco...'
else
echo 'Erraste por bastante...|'
fi

exit 0
```

Notar que ES CRUCIAL dejar espacios entre []:

```
if [ ${a} -eq 1 ] <-- ES INCORRECTO
```

```
if [ $ {a} -eq 1 ] <-- ES CORRECTO
```


Estructuras condicionales IF - ELSE :

```
fede@fedelaptoplenovo:~/tmp$ ./script_prueba.sh
Adivina el valor numérico de la variable: Introduce un entero
3
Valor leído: 3
```

Erraste por bastante...

```
fede@fedelaptoplenovo:~/tmp$ ./script_prueba.sh
Adivina el valor numérico de la variable: Introduce un entero
13
Valor leído: 13
```

Estuviste cerca, pero te faltó un poco...

```
fede@fedelaptoplenovo:~/tmp$ ./script_prueba.sh
Adivina el valor numérico de la variable: Introduce un entero
16
Valor leído: 16
```

Estuviste cerca, pero te pasaste...

```
fede@fedelaptoplenovo:~/tmp$ ./script_prueba.sh
Adivina el valor numérico de la variable: Introduce un entero
15
Valor leído: 15
```

Acertaste!

Operadores de comparación para cadenas:

Operator	Description
-z string	True if the length of string is zero
-n string	True if the length of string is non-zero
string1 == string2 or string1 = string2	True if the strings are equal; a single = should be used with the test command for POSIX conformance. When used with the [[command, this performs pattern matching as described above (compound commands).
string1 != string2	True if the strings are not equal
string1 < string2	True if string1 sorts before string2 lexicographically (refers to locale-specific sorting sequences for all alphanumeric and special characters)
string1 > string2	True if string1 sorts after string2 lexicographically

Operadores de comparación para numéricos:

Operator	Description
arg1 -eq arg2	True if arg1 equals arg2
arg1 -ne arg2	True if arg1 is not equal to arg2
arg1 -lt arg2	True if arg1 is less than arg2
arg1 -le arg2	True if arg1 is less than or equal to arg2
arg1 -gt arg2	True if arg1 is greater than arg2
arg1 -ge arg2	True if arg1 is greater than or equal to arg2

Ciclos 'for' en bash

```
~/tmp/simple_for.sh - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

simple_for.sh x
1 #!/bin/bash
2 #Test simple for
3
4 for i in lunes martes miércoles jueves viernes "fin de semana"; do
5     #statements
6     echo $i
7 done
8 exit 0
9
```

```
fedede@fededeLaptopLenovo: ~/tmp
File Edit View Search Terminal Help
fedede@fededeLaptopLenovo:~/tmp$ ./simple_for.sh
lunes
martes
miércoles
jueves
viernes
fin de semana
fedede@fededeLaptopLenovo:~/tmp$
```

Ciclos 'for' en bash al estilo 'C':

```
#!/bin/bash

#Para hacer ciclos (loops) en bash, usar las sentencias
#for y do, como en

N=10
SumaCuadrados=0

for((a=1; a <= N ; a++))
do
SumaCuadrados=$((SumaCuadrados+a*a))
done

echo "La suma de los cuadrados de los primeros numeros \
naturales hasta $N es $((SumaCuadrados))"
```

Ciclos while:

```
~/tmp/simple_while.sh - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

simple_while.sh x
1 #!/bin/bash
2 #Test simple while
3 i=10
4 echo "imprimimos cuenta regresiva!"
5 while [[ $i -ge 0 ]]; do
6     echo $i
7     sleep 1
8     i=$((i-1))
9 done
10 echo "Listo!"
11 exit 0
12
```

```
fedede@fededeLaptopLenovo: ~/tmp
File Edit View Search Terminal Help
fedede@fededeLaptopLenovo:~/tmp$ ./simple_while.sh
imprimimos cuenta regresiva!
10
9
8
7
6
5
4
3
2
1
0
Listo!
fedede@fededeLaptopLenovo:~/tmp$
```


Funciones en bash:

Si, por ejemplo, en el archivo `PruebaFuncionesBash.sh` generamos una función de bash tal como

```
Calendario_Y_Hora()  
{  
    echo "Imprimiendo Calendario y Hora"  
    cal  
    date  
}
```

y luego hacemos

```
. PruebaFuncionesBash.sh
```

o

```
source PruebaFuncionesBash.sh  
tendremos Calendario_Y_Hora como un comando mas de bash (por  
supuesto solo en la terminal en la que estamos trabajando)
```

```
fed@fedelaptopLenovo:~/tmp$ source PruebaFuncionesBash.sh  
fed@fedelaptopLenovo:~/tmp$ Calendario_Y_Hora  
Imprimiendo Calendario y Hora  
      Marzo 2020  
do lu ma mi ju vi sa  
  1  2  3  4  5  6  7  
  8  9 10 11 12 13 14  
15 16 17 18 19 20 21  
22 23 24 25 26 27 28  
29 30 31  
  
mar mar 24 14:00:47 -03 2020
```

El comando sed:

Para reemplazar una cadena de texto de un archivo por otra desde la línea de comandos usar el comando sed:

```
sed "s/cadena_a_reemplazar/nueva_cadena/" archivo_in > archivo_out
```

con la opción -i se modifica directamente el archivo de entrada, sin necesidad de generar otro archivo.

Por ejemplo

```
sed -i "s/TEK/#TEK/" Archivo.dat
```

podría servir para comentar todas las líneas que comiencen con TEK

```
sed "s/cadena_a_reemplazar/nueva_cadena/g" archivo
```

La opción g se pone para indicar una sustitución global de la cadena (en algunas versiones de sed es indispensable agregar una terminación como esta a la opción "s")

Para insertar por ejemplo una cadena en la línea n-esima de un archivo usar

```
sed -i '<n>i New Line in the file!' <filename>
```

Por ejemplo, para insertar en la primera línea poner

```
sed -i '1i Top of the file!' <filename>
```

Para eliminar las líneas 2 a 7 del archivo

```
sed -i '2,7 d' <filename>
```

El comando awk:

Para operar sobre archivos con columnas desde línea de comandos, usar el comando awk, como por ej.

```
awk '{print $3 "\t" $4}' archivo_in.txt
```

o (redirigiendo la salida)

```
awk '{print $3 "\t" $4}' archivo_in.txt > archivo_out.txt
```

Aquí \$1, \$2, etc se refieren a la primera columna, segunda columna, etc.

Por ejemplo para retener solo las columnas 4 y 5 de aquellas filas en las que los campos de la primera columna sean mayores que 0, redirigiendo la salida, ejecutar

```
awk '$1>0 {print $4 "\t" $5}' archivo_in.txt > archivo_out.txt
```

Operaciones aritméticas en punto flotante

Para poder en una variable la salida de un comando usar el comando entre ``
Por ejemplo

```
segundosUTC=`date +%s`
```

pone en la variable segundosUTC el numero de segundos transcurridos desde el 1 de enero de 1970 a las 0 hs.

Si queremos hacer operaciones aritmeticas sobre una variable usamos \$(())
por ejemplo:

```
a=$((segundosUTC+20))
```

Para hacer operaciones en punto flotante hay que usar el comando bc (bash solo trata de forma nativa con la aritmetica de enteros) por un pipe del comando echo.
Por ejemplo,

```
b=`echo "3.51*$a" | bc -l`
```

Pasando parámetros al script:

```
#!/bin/bash
#Los scripts de Bash reciben los argumentos que le pasa la shell
#como $1, $2, ..., $n. Se puede obtener el número total de argumentos con el símbolo $#.

#Usando $# es posible verificar el número de argumentos entregados al
#script antes de realizar alguna acción con ellos:

if [ $# -lt 2 ]; then
    echo "El script espera al menos dos argumentos."
    exit 1
fi

#Otra forma de acceder a los argumentos es a través del array $@, por medio
#del cual se puede iterar sobre todos los argumentos dados:

for arg in "$@"
do
    echo "$arg"
done
```

Pasando parámetros al script:

Ejecutando el script varias veces con distintos argumentos obtenemos:



```
fede@fedelaptoplenovo:~/tmp$ ./script_args.sh
El script espera al menos dos argumentos.
fede@fedelaptoplenovo:~/tmp$ ./script_args.sh 23
El script espera al menos dos argumentos.
fede@fedelaptoplenovo:~/tmp$ ./script_args.sh 23 hola
23
hola
fede@fedelaptoplenovo:~/tmp$ ./script_args.sh 23 hola chau
23
hola
chau
fede@fedelaptoplenovo:~/tmp$ ./script_args.sh 23 hola chau 33.2
23
hola
chau
33.2
fede@fedelaptoplenovo:~/tmp$
```


Alternativa al if-else: sentencias case-esac

```
~/tmp/cases.sh - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

cases.sh x
1  #!/bin/bash
2  #Test case esac
3
4  echo "Quieres saber el directorio actual? (sS/nN)
5  read Resp
6  case $Resp in
7      [sS])
8          #la respuesta es sí
9          echo "El directorio actual es"
10         pwd
11         ;;
12         [nN])
13             #la respuesta es no
14             echo "Ok, entendido!"
15             ;;
16         *)
17             #opción por default
18             echo "Entrada no válida"
19             ;;
20     esac
21
22     exit 0
23
```

Line 18, Column 28 Tab Size: 4 Bourne Again Shell (bas

```
File Edit View Search Terminal Help
fed@fedelaptopLenovo:~/tmp$ ./cases.sh
Quieres saber el directorio actual? (sS/nN)
s
El directorio actual es
/home/fede/tmp
fed@fedelaptopLenovo:~/tmp$ ./cases.sh
Quieres saber el directorio actual? (sS/nN)
S
El directorio actual es
/home/fede/tmp
fed@fedelaptopLenovo:~/tmp$ ./cases.sh
Quieres saber el directorio actual? (sS/nN)
n
Ok, entendido!
fed@fedelaptopLenovo:~/tmp$ ./cases.sh
Quieres saber el directorio actual? (sS/nN)
N
Ok, entendido!
fed@fedelaptopLenovo:~/tmp$ ./cases.sh
Quieres saber el directorio actual? (sS/nN)
fg
Entrada no válida
fed@fedelaptopLenovo:~/tmp$
```

Operadores útiles para el manejo de archivos:

Algunos de los operadores disponibles:

- -e ~> verdadero si el archivo existe pero está vacío.
- -s ~> verdadero si el archivo existe y contiene datos.
- -d ~> verdadero si se trata de un directorio.
- -L ~> verdadero si se trata de un enlace simbólico.
- -w ~> verdadero si es un archivo con permiso de escritura.
- -x ~> verdadero si es un archivo ejecutable.

```
#!/bin/bash
```

```
if [ "$#" -ne "1" ]; then  
    echo -e "Uso del script:\t$0 NombreArchivo\n"  
    exit 1  
else
```

```
NombreArchivo=$1  
fi
```

```
if [ -s $NombreArchivo ]  
then  
    echo "El archivo $NombreArchivo existe y contiene datos"  
elif [ -e $NombreArchivo ]  
then  
    echo "El archivo $NombreArchivo existe y está vacío"  
else  
    echo "No existe un archivo llamado $NombreArchivo"  
fi
```

Operadores útiles para el manejo de archivos:

hola

Open ▾



prueba_vacio.dat
~/tmp

Save



fed@fedelaptoplenovo: ~/tmp

File Edit View Search Terminal Help

```
fed@fedelaptoplenovo:~/tmp$ ./script_acceso_archivos.sh
```

```
Uso del script: ./script_acceso_archivos.sh NombreArchivo
```

```
fed@fedelaptoplenovo:~/tmp$ ./script_acceso_archivos.sh inexistente
```

```
No existe un archivo llamado inexistente
```

```
fed@fedelaptoplenovo:~/tmp$ ./script_acceso_archivos.sh prueba.dat
```

```
El archivo prueba.dat existe y contiene datos
```

```
fed@fedelaptoplenovo:~/tmp$ ./script_acceso_archivos.sh prueba_vacio.dat
```

```
El archivo prueba_vacio.dat existe y está vacío
```

```
fed@fedelaptoplenovo:~/tmp$
```

Cambiando el color de la fuente en bash

En general, el esquema es:

```
\e[CODIGOm(texto)\e[0m (el texto sin los paréntesis)
```

Si ponemos, por ejemplo,

```
echo -e "\e[1;31m"
```

sin cadena de texto, nos cambiará a rojo el color de la fuente de la consola, hasta tanto no la volvamos a cambiar

Los códigos de algunos de los colores son:

* Negro = 0;30

* Cian = 0;36

* Azul = 0;34

* Verde = 0;32

* Rojo = 0;31

* Gris oscuro = 1;30

* Cyan resaltado = 1;36

* Azul resaltado = 1;34

* Verde resaltado = 1;32

* Rojo resaltado = 1;31

Caracteres especiales en bash:

```
~/tmp/caract_esp.sh - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

simple_while.sh x caract_esp.sh x

1 #!/bin/bash
2 #Ejemplo caracteres especiales en bash
3 #para imprimir caracteres especiales en bash, se pueden usar
4 #caracteres utf-8, por ej.
5 #imprime la letra griega pi
6 echo -e "\xcf\x80"
7
8 #imprime la letra i acentuada
9 echo -e "\xc3\xad"
10
11 #Para imprimir en colores usar los caracteres de escape, como en:
12 echo -e "\e[0;31m\xcf\x80\e[0m"
13 #que imprimira la letra griega pi en color rojo, el ultimo
14 #caracter de escape \e[0m lo que hace es devolvernos al color
15 #original por default de la consola. Otro ejemplo:
16
17 echo -e "Esto es \e[0;31mrojo\e[0m y esto es \e[1;34mazul resaltado\e[0m"
18
19
```

fedefede@fedelaptoplenovo: ~/tmp

```
File Edit View Search Terminal Help
fedefede@fedelaptoplenovo:~/tmp$ ./caract_esp.sh
π
í
π
Esto es rojo y esto es azul resaltado
fedefede@fedelaptoplenovo:~/tmp$
```

Line 17, Column 38

Accediendo al valor de retorno de los comandos de bash:

~/tmp/chequear_comando.sh - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

chequear_comando.sh x

```
1 #!/bin/bash
2 type "$1" > /dev/null
3
4 # $? nos da el código de retorno del último comando ejecutado
5 #0 si fue exitoso
6 #1 error leve
7 #2 error grave
8
9 if [[ $? -eq 0 ]]; then
10     echo "El comando $1 existe y es:"
11     type $1
12 else
13     echo "El comando $1 NO existe..."
14 fi
```

```
fed@fedelaptopLenovo:~/tmp$ type --help
type: type [-afptP] name [name ...]
    Display information about command type.
```

For each NAME, indicate how it would be interpreted if used as a command name.

Options:

- a display all locations containing an executable named NAME; includes aliases, builtins, and functions, if and only if the '-p' option is not also used
- f suppress shell function lookup
- P force a PATH search for each NAME, even if it is an alias, builtin, or function, and returns the name of the disk file that would be executed

fed@fedelaptopLenovo: ~/tmp

File Edit View Search Terminal Help

```
fed@fedelaptopLenovo:~/tmp$ ./chequear_comando.sh gedit
El comando gedit existe y es:
gedit is /usr/bin/gedit
fed@fedelaptopLenovo:~/tmp$ ./chequear_comando.sh geditrf
./chequear_comando.sh: line 2: type: geditrf: not found
El comando geditrf NO existe...
fed@fedelaptopLenovo:~/tmp$ ./chequear_comando.sh ls
El comando ls existe y es:
ls is /bin/ls
fed@fedelaptopLenovo:~/tmp$ ./chequear_comando.sh lsfd
./chequear_comando.sh: line 2: type: lsfd: not found
El comando lsfd NO existe...
fed@fedelaptopLenovo:~/tmp$
```


Para profundizar:

- *Using Linux* J. Tackett Jr. and S. Burnett Fifth Ed. QUE.
- <https://thales.cica.es/rd/glinex/practicas-glinex05/manuales/bash/practica.pdf>
- <http://www.tldp.org/LDP/Bash-Beginners-Guide/html/>
- *Unix Programación avanzada* F. M. Márquez 3ra ed. Alfaomega RA-MA Editorial.