serie
techné

Introducción a la computación numérica usando la herramienta Scilab

Un texto para estudiantes de matemáticas aplicadas a ciencias e ingeniería

javier ignacio
carrero mantilla

UNIVERSIDAD NACIONAL DE COLOMBIA

VICERRECTORÍA DE INVESTIGACIÓN
DIRECCIÓN DE INVESTIGACIÓN SEDE BOGOTÁ
EDITORIAL

Introducción a la computación numérica usando la herramienta Scilab

Un texto para estudiantes de matemáticas aplicadas a ciencias e ingeniería

Introducción a la computación numérica usando la herramienta Scilab

Un texto para estudiantes de matemáticas aplicadas a ciencias e ingeniería

javier ignacio
carrero mantilla

UNIVERSIDAD NACIONAL DE COLOMBIA
VICERRECTORÍA DE INVESTIGACIÓN
DIRECCIÓN DE INVESTIGACIÓN SEDE BOGOTÁ
EDITORIAL

Bogotá, D.C., febrero de 2014

© Universidad Nacional de Colombia
© Dirección de Investigación Sede Bogotá
© Editorial Universidad Nacional de Colombia
© Javier Ignacio Carrero Mantilla

Editorial
Universidad Nacional de Colombia
Oscar F. Castellanos Domínguez
Director

Primera edición, 2013
ISBN: 978-958-761-750-4 (papel)
ISBN: 978-958-761-752-8 (IPD)
ISBN: 978-958-761-751-1 (digital)

Diseño Serie Techné
Marco Aurelio Cárdenas

Prohibida la reproducción total o parcial
por cualquier medio sin la autorización escrita
del titular de los derechos patrimoniales

Catalogación en la publicación Universidad Nacional de Colombia

Carrero Mantilla, Javier Ignacio, 1973-
Introducción a la computación numérica usando la herramienta Scilab: un texto para estudiantes de matemáticas aplicadas a ciencias e ingeniería / Javier I. Carrero Mantilla.– Bogotá: Universidad Nacional de Colombia. Vicerrectoría de Investigación, 2014.

154 páginas: ilustraciones. - (Serie Techné)

Incluye referencias bibliográficas

ISBN : 978-958-761-750-4 (papel) - ISBN : 978-958-761-752-8 (IPD) – ISBN : 978-958-761-751-1 (digital)

1. Análisis numérico - Procesamiento de datos 2. Programación (Matemáticas) 3. Scilab I. Título II. Serie

CDD-21 519.40285 / 2014

Contenido

Introdución 11

Capítulo 1 Instalación y uso de la consola 13

Capítulo 2 Operaciones de archivo 25

Capítulo 3 Tipos de datos 33
- 3.1 Tipos básicos de variables 33
- 3.2 Matrices 38
- 3.3 Manipulación de matrices 43
- 3.4 Polinomios 51
- 3.5 Listas y células 52

Capítulo 4 Programación 57
- 4.1 Funciones y *scripts* 57
- 4.2 Funciones definidas por el usuario 63
- 4.3 Manejo de la memoria 67
- 4.4 Manipulación de entradas y salidas 70
- 4.5 Estructuras de control 73
- 4.6 Depuración 80

Capítulo 5 Gráficas 87
- 5.1 De números a gráficas 87
- 5.2 Programación de gráficas 93

Capítulo 6 Herramientas numéricas 99
- 6.1 Cálculo 99
- 6.2 Ecuaciones diferenciales ordinarias 102
- 6.3 Ajuste de datos a una curva 106

Capítulo 7 Programación orientada a objetos 115
- 7.1 Listas tipificadas 115
- 7.2 Sobrecarga de funciones y operadores 119
- 7.3 Estructuras 123

Apéndice A Proyectos 127

Apéndice B Extensiones 131

Apéndice C Interfaces 135

Apéndice D Xcos 141

Solucionario para las guías de autoaprendizaje 145

Bibliografía 151

Lista de tablas

Tabla 1.1	Algunos operadores comunes	19
Tabla 1.2	Algunas funciones comunes	20
Tabla 2.1	Órdenes para escribir variables	26
Tabla 3.1	Operadores lógicos	34
Tabla 3.2	Ejemplos de formato numérico	35
Tabla 3.3	Constantes incorporadas	38
Tabla 3.4	Operadores para matrices	40
Tabla 5.1	Algunas órdenes relacionadas con gráficas	88
Tabla 6.1	Datos para regresión lineal multivariada	107
Tabla 6.2	Datos para regresión no lineal	110
Tabla 6.3	Datos para regresión no lineal (guía de autoaprendizaje)	112
Tabla C.1	Órdenes para manipular menús	135

Lista de figuras

Figura 1.1	Interfaz de usuario de Scilab	14
Figura 1.2	Cuadro de diálogo para las preferencias	15
Figura 1.3	Función de autocompletado en la consola	16
Figura 1.4	Command History (historial de órdenes)	18
Figura 1.5	Ventana del sistema de ayuda de Scilab	19
Figura 1.6	Explorador de variables	21
Figura 1.7	Editor de variables (Variable Editor).	21
Figura 2.1	File Browser (navegador de ficheros).	26
Figura 2.2	Matriz guardada como archivo de texto	28
Figura 4.1	SciNotes: el editor de archivos de Scilab	58
Figura 4.2	Espacios de memoria	67

Figura 4.3	Órdenes para romper un ciclo de repetición	78
Figura 5.1	Una gráfica simple, $f(t) = \exp(-0.5t)\sin(5t)$	88
Figura 5.2	Gráfica simple con formato	90
Figura 5.3	Gráfica con cuatro *plots*	91
Figura 5.4	Gráfica tridimensional de la función del cenicero.	92
Figura 5.5	Gráfica de contorno de la función del cenicero.	93
Figura 5.6	Figura *Homer*.	97
Figura 6.1	Solución de $dy/dt = (1+2t)\sqrt{y}$.	103
Figura 6.2	Solución de un sistema de dos ecuaciones diferenciales	105
Figura 6.3	Ajuste de datos	112
Figura B.1	Administrador de módulos ATOMS	133
Figura C.1	Ventana con dos botones de menú definidos por el usuario	136
Figura C.2	Ventana con entrada de menú definida por el usuario	137
Figura C.3	Interfaz para calcular las raíces de la ecuación $ax^2+bx+c=0$	138
Figura D.1	Ventana de Xcos acompañada del Palette Browser	141
Figura D.2	Resultado de la simulación con Xcos	143

Introducción

Este libro nació de la experiencia del autor dictando cursos de métodos numéricos. En ellos, la solución de problemas usando lenguajes de alto nivel, tales como C++ o FORTAN, lo que se podría llamar "esquema ingeniero-programador", ha llegado a ser muy cuestionable por dos razones: tiene una relación costo/beneficio muy elevada y es muy difícil de adaptar y reciclar. Este esquema "ingeniero-programador" tiende a ser reemplazado por el esquema "ingeniero-usuario" en que las soluciones se obtienen usando software numérico especializado.

Sin embargo, es común que los "ingenieros-usuarios" –mejor dicho, los "estudiantes-usuarios"– pretendan que el software resuelva cualquier problema por arte de magia. En realidad, los programas de computador son similares a una calculadora de grandes capacidades, y manejarlos requiere ciertas competencias básicas. Por ejemplo: traducir fórmulas matemáticas a funciones computacionales, guardar resultados en archivos, o hacer gráficas. Por carecer de estas competencias, los usuarios caen en una situación frustrante: saben *qué quieren* hacer pero no *cómo*, y desisten de la solución de problemas numéricos o gastan un tiempo exagerado.

Esto indica la necesidad de formar al estudiante no solamente en la teoría matemática, sino también en la práctica computacional, usando herramientas actualizadas. La plataforma de computación numérica Scilab es muy apropiada para este propósito por ser de descarga gratuita y de código fuente abierto. Scilab fue creado en el Instituto Nacional para la Investigación en Control y Ciencia Computacional (Inria, por sus siglas en francés), y ha sido actualizado y revisado con frecuencia desde que se hizo disponible al público en 1994.

Puede ser cuestionable basar el aprendizaje en una herramienta particular, pero la interfaz de usuario y el lenguaje de programación de Scilab siguen convenciones comunes a la mayoría de programas. El conocimiento de Scilab le servirá al lector como base para dominar casi cualquier otro software o lenguaje de programación con el que se encuentre. Además, usar una sola herramienta es más didáctico que hacer saltar al lector entre varios programas distintos.

Este libro busca preparar al lector para crear sus propias soluciones. De esta forma, al terminar el libro será capaz de hacer lo siguiente:

- Evaluar fórmulas para obtener resultados numéricos.
- Importar y exportar datos.
- Crear nuevas funciones.
- Resolver problemas numéricos comunes.

- Dibujar gráficas.
- Hacer estructuras de datos sencillas.

Para cumplir con esos objetivos es necesario, pero no suficiente, leer el libro: hay que practicar, reproducir los ejemplos, y seguir las guías de autoaprendizaje al final de cada capítulo. En otras palabras este libro no es un manual, ni una lista de todas las funciones disponibles, ni un repertorio de "trucos".

La obra está dividida en siete capítulos y tres apéndices, y se recomienda abarcarla así

- Los capítulos 1 a 4 deben leerse en secuencia porque cada uno depende del anterior. Sus temas van desde la instalación y el uso de la consola hasta la programación.
- Los capítulos 5 (gráficas), 6 (herramientas numéricas), y 7 (programación orientada a objetos) se pueden leer en cualquier orden, después de terminar los cuatro primeros.
- En el apéndice A hay cinco proyectos. Cada uno se puede resolver usando las herramientas descritas en los capítulos 1 a 4.
- Los apéndices B y C pueden dejarse como referencia futura si el lector está corto de tiempo. Los temas que tratan, extensiones e interfases, requieren conocer bien el funcionamiento del programa.
- El apéndice D es de interés para aplicaciones de control. Es una descripción breve del sistema de simulación y control Xcos.

Cada tema se explica en sus puntos básicos y se evita repetir información que está en la ayuda, por eso en varios puntos del texto se sugiere consultarla. La exposición de los temas se mezcla con ejemplos y al final de cada capítulo se incluye una "Guía de autoaprendizaje" con actividades de repaso de lo aprendido. Dentro del texto `este tipo de letra` se usa para las órdenes y código de programación, y cuando un renglón empieza con `-->` se entiende que se trata de órdenes que el lector tiene que escribir la consola (se explica en el capítulo 1). Los términos propios de Scilab, por ejemplo nombres de funciones y tipos de datos se han dejado en inglés porque así están en el programa y en el sistema de ayuda.

Este libro está concebido para ser distribuido en forma electrónica porque es más cómodo tenerlo en la pantalla del computador que impreso en papel. Los ejemplos están dentro del texto porque así es más fácil copiarlos y pegarlos. En algunos sistemas puede aparecer un pequeño inconveniente: que al copiar porciones de texto que contengan comillas (por ejemplo `''estaInstruccion''`) o signos tales como `+`, `-`, `^`, `'` se genere un mensaje de error debido a falta de compatibilidad de caracteres ASCII. Si le ocurre, por favor escriba las porciones de texto problemáticas en lugar de pegarlas.

Capítulo 1

INSTALACIÓN Y USO DE LA CONSOLA

Al finalizar este capítulo usted deberá ser capaz de:

- Instalar y configurar Scilab.
- Dar órdenes básicas en la consola.
- Buscar ayuda.
- Usar Scilab como una calculadora.
- Crear y borrar variables.

Scilab está disponible para los sistemas operativos Windows, MacOSX, y Linux. Por ser gratuito y abierto, la versión más reciente siempre se puede descargar desde el sitio[1] de Scilab, sin restricciones. No hacen falta claves de activación o cosas similares para instalar el programa, pero hay algunos detalles propios de cada sistema operativo

- Para Windows existe un programa instalador, pero la instalación no es acumulativa. Por ejemplo, si se tiene la versión 5.3, al instalar la versión 5.4 quedan ambas coexistiendo. Por omisión, el instalador trata de descargar software adicional de Intel: CPU Optimization for Scilab y FFTW Interface for Scilab. Esos componentes no son indispensables, pero se supone que mejoran el rendimiento en operaciones especializadas, tales como las transformadas rápidas de Fourier. Si usted no está conectado a internet o simplemente no quiere instalar esos dos componentes, los puede evitar seleccionando las otras casillas, que corresponden a las bibliotecas de referencia. Scilab aparece en el menú principal del botón de Inicio como "Scilab 5.x.y" donde x, y son los números de la versión. El programa principal aparece como "Scilab 5.x.y" en el submenú. El icono Scilab Console arranca el programa sin ninguna interfaz, pero esa opción no se trata aquí.
- En Linux, el archivo que se descarga no es un instalador, sino una carpeta comprimida donde está el programa compilado. Por lo tanto, primero hay que descomprimirla en la ubicación donde se quiera tenerla y luego se inicia

---

[1] www.scilab.org, buscar por "download".

el archivo ejecutable `/bin/scilab`. Varias distribuciones Linux, entre ellas Ubuntu, tienen disponible a Scilab en sus *repositorios* de software. En esos casos, la instalación es automática: basta con seleccionar el programa en la lista de disponibles. Pero hay un pequeño inconveniente: como los distribuidores demoran algo de tiempo verificando la compatibilidad, la versión disponible en el repositorio puede no ser la más reciente.

- Para MacOSX la instalación está contenida en un archivo dmg, así que debe seguirse desde el procedimiento de instalación desde el directorio Applications (o Aplicaciones).

Aunque la instalación ofrece varios idiomas para la interfaz de usuario, se recomienda instalarlo en inglés porque el sistema de ayuda solo está en ese idioma, lo mismo que las palabras clave. En el apéndice B se explica cómo extender modularmente una instalación. Además, se incluye un ambiente gráfico separado para crear modelos de sistemas de control y simulación llamado Xcos (ver apéndice D). Para los interesados en modificar o adaptar el programa a fondo, Inria permite descargar el código fuente completo de la aplicación para compilarla o modificarla, pero hay que tener en cuenta que es un programa con miles de líneas de código e instrucciones.

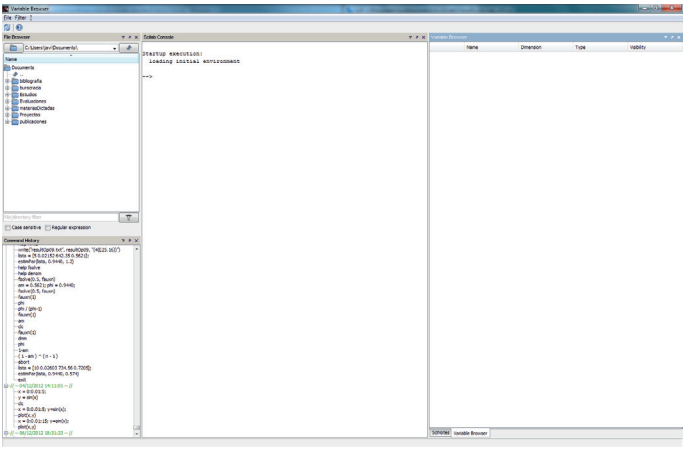


Figura 1.1. Interfaz de usuario de Scilab, versión 5.4. La consola está en el centro. La disposición puede variar según el sistema

La interfaz de Scilab incluye estos componentes (ver figura 1.1):

- Consola (Console).
- Navegador de ficheros (File Browser).
- Navegador de variables (Variable Browser).

- Historial de órdenes (Command History).
- Editor de variables (Variable Editor).
- Editor de archivos (Scinotes).
- Navegador del sistema de ayuda (Help Browser).

Los componentes se activan en el menú `Applications`. El menú de la ventana principal cambia según el componente que esté activo (figura 1.1, en inglés), y el usuario puede poner cualquier componente en la ventana principal o en una ventana aparte. Para eso se arrastra la barra de color, en cada (sub)ventana, hasta la posición deseada. Además, todas las opciones de configuración están centralizadas en una solo cuadro de diálogo (figura 1.2), que se activa en el menú `Edit/references`. Por todo esto, pueden existir algunas diferencias menores entre lo que usted tiene en pantalla y las ilustraciones de este libro.

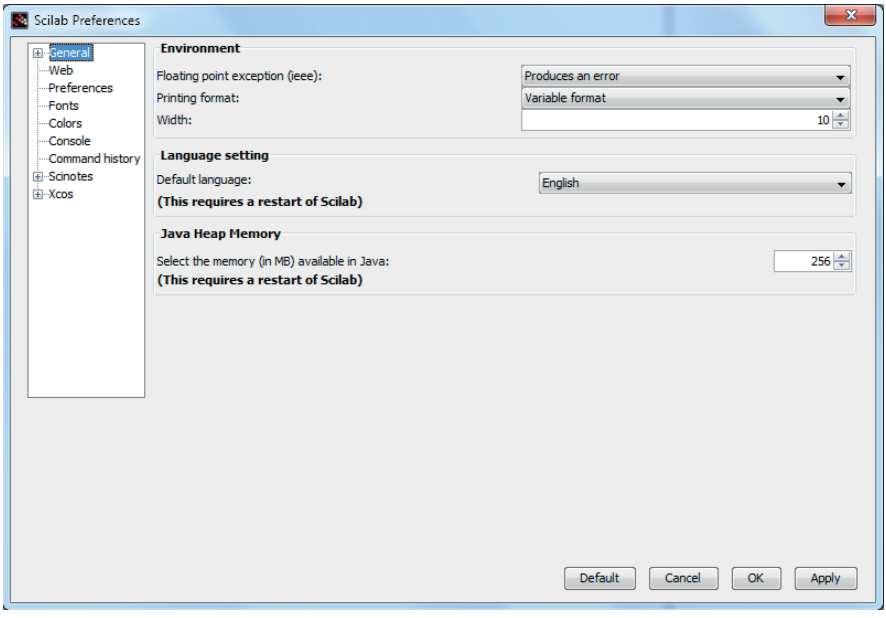


Figura 1.2. Cuadro de diálogo para las preferencias

El componente más importante es la consola, que aparece usualmente en el centro de la ventana principal, figuras 1.1 y 1.3. También se le llama "intérprete de órdenes" porque el núcleo de Scilab es un intérprete de funciones matemáticas básicas y de álgebra lineal. Como el programa acepta cientos de órdenes distintas, en el menú de la consola aparecen solo las más importantes, por cuestión de espacio. En realidad, se espera que el usuario escriba las órdenes dentro de la consola en la parte de la pantalla que sigue al `-->`, llamada *prompt*.

- Cuando aparece `-->` en el texto, se entiende que se trata de una instrucción que hay que escribir en la consola.

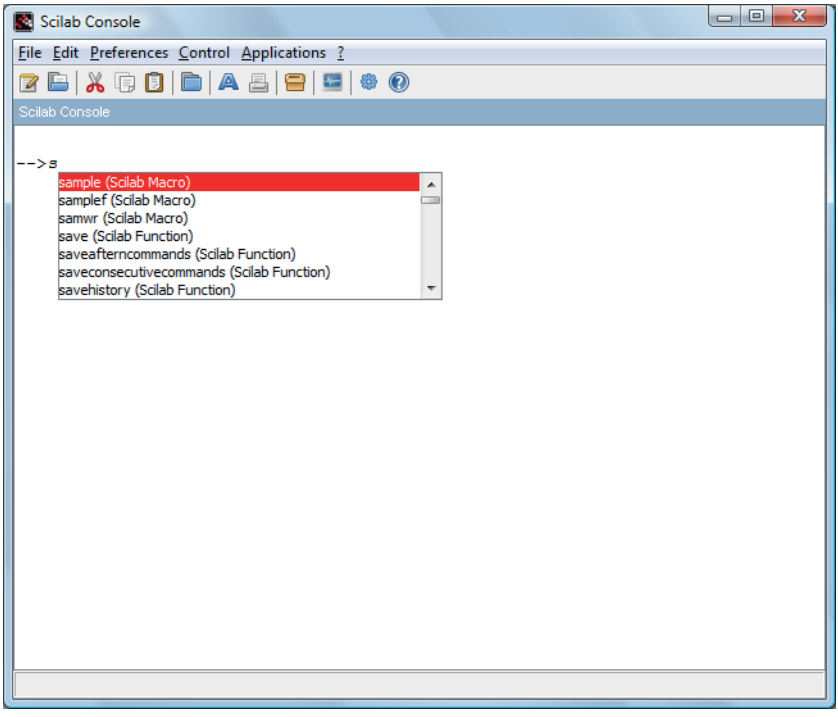


Figura 1.3. Función de autocompletado en la consola

En el *prompt* hay una ayuda para completar órdenes, que se describe más adelante. Cada orden se ejecuta pulsando la tecla enter, por ejemplo

```
-->orden+enter
```

- De aquí en adelante, se omite `+enter` en el texto. Se sobreentiende, en las explicaciones, que al final hay que pulsar la tecla *enter* en la consola.

Las órdenes también se pueden seleccionar en el menú usando el ratón. La consola permite copiar y pegar texto en las formas usuales: las combinaciones de teclas `ctrl-c` y `ctrl-v`, el menú del botón secundario del ratón y las opciones del menú `Edit`.

El resultado de las órdenes se muestra en la pantalla. Eso es llamado el "eco" de la orden. Además, queda registrado en una variable llamada `ans`, por *answer* –es decir, "respuesta" en inglés–. Por ejemplo, la función `rand()` produce un número seudoaleatorio con un valor entre 0 y 1. Se invoca con la orden:

```
-->rand()
ans = 0.6283918
-->ans
ans = 0.6283918
```

Si al final de la orden se agrega un ; el eco se suprime, pero igual el resultado queda registrado en `ans`. Para el mismo ejemplo:

```
-->rand();
-->ans
ans = 0.7560439
```

En este caso, el resultado cambia porque `rand()` produce un número seudoaleatorio[2] distinto cada vez que se invoca. Las órdenes que controlan el funcionamiento del programa, por ejemplo `clc`, no producen un resultado, así que no tienen eco. Hay algunas otras observaciones importantes sobre las órdenes:

- La mayoría se escribe en minúsculas.
- Son sensibles al tipo de letra, mayúscula o minúscula. `estaOrden` no se puede escribir como `estaorden`, porque el programa no la reconoce.
- Da lo mismo escribir una orden o seleccionarla del menú. Si se selecciona del menú igual aparece en el *prompt*.
- No hay opción de deshacer una orden. Es decir, no hay botones undo y *redo*.
- Para recuperar en el *prompt* una orden que ya se escribió se usan las teclas flecha arriba (↑) y flecha abajo (↓).
- La orden `clc` (atajo: tecla `F2`) borra la pantalla, pero el programa sigue "recordando" la secuencia de órdenes.
- La orden `exit` finaliza la ejecución de Scilab. Advertencia: todos los resultados que no se hayan guardado en disco se pierden.
- La opción `Quit` del menú `File` de la consola equivale a `exit`.

Las órdenes que se escriben quedan guardadas automáticamente en la lista que se muestra en el componente Command History, figura 1.4. Para llevar una orden al *prompt* desde la lista se hace doble clic sobre ella. También se pueden utilizar las funciones `savehistory` y `loadhistory` para guardar y recuperar las órdenes introducidas en una sesión. Pero volver a ejecutar las órdenes no garantiza que se recuperará el estado del programa, sobre todo si se han borrado variables.

---

2 En general, un computador no puede producir números de verdad aleatorios, pero existen técnicas matemáticas que permiten producir en forma algorítmica secuencias de números que no tienen relación unos con otros y que se pueden considerar, por lo tanto, seudoaleatorios.

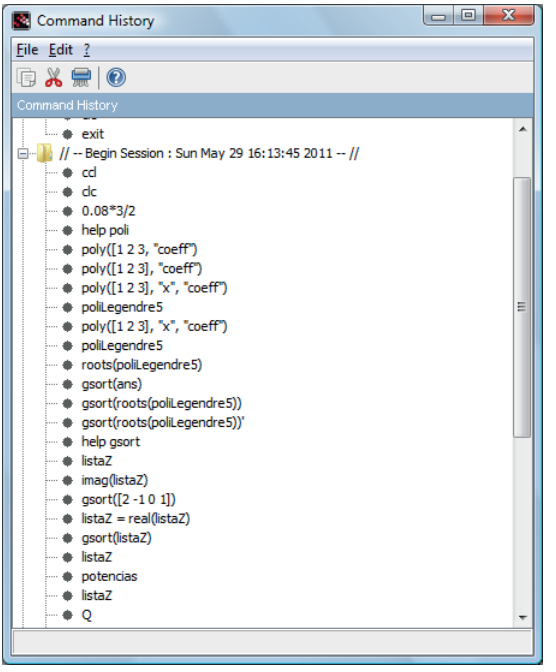


Figura 1.4. Command History (historial de órdenes)

Hay tres formas de activar la ventana del sistema de ayuda (figura 1.5):

- La opción `Scilab Help` del menú ? en el extremo derecho, o el icono ?
- La tecla `F1`.
- Desde el *prompt*, con la orden `-->help`
- Para una orden específica: `-->help nombre_de_la_orden`.

Además de la ayuda, la consola tiene una ayuda automática de autocompletar, que se activa con la tecla `tab` (tabulador). Si se escribe el comienzo de una orden o función al pulsar `tab` aparece una caja con opciones que se pueden seleccionar usando el ratón o las teclas de flecha arriba/flecha abajo. Por ejemplo, si se está buscando una orden que empieza por `s` se escribe `s+<tab>` y aparece la caja que se muestra en la figura 1.3.

Hasta este punto, se ha descrito la lógica del funcionamiento de la consola, pero sin explicar muy bien para qué sirve. Scilab es en esencia una calculadora y la consola es el equivalente a la pantalla, es el sitio donde se muestran las operaciones y los resultados. Una operación válida, escrita con números y operadores en el *prompt*, se interpreta y el resultado queda en la variable especial `ans`:

```
-->5+3-2
ans = 6.
```

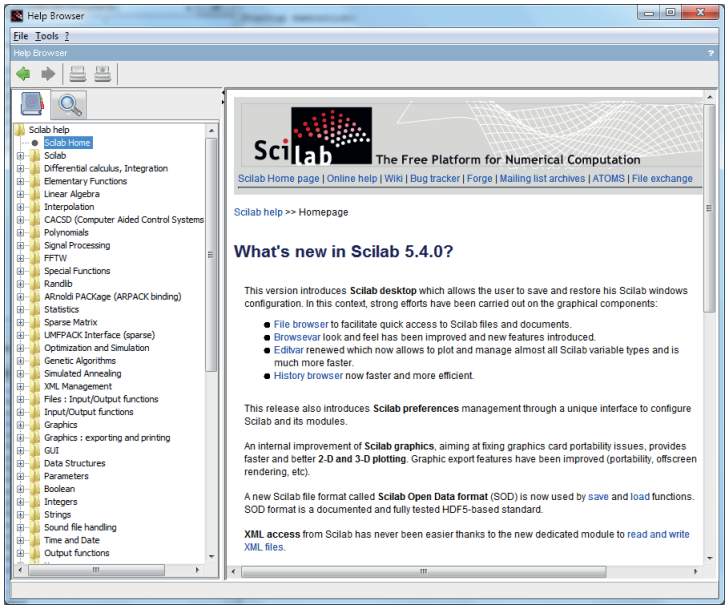


Figura 1.5. Ventana del sistema de ayuda de Scilab

Tabla 1.1. Algunos operadores comunes

Operador	Operación
+	suma
-	resta
*	multiplicación
/	división
^	potencia

Se puede invocar `ans` para recuperar el resultado:

```
-->ans
ans = 6.
```

pero el valor cambiará en cada orden nueva que se ejecute. Están disponibles los operadores comunes de cualquier calculadora o lenguaje de programación (ver tabla 1.1): suma, resta, multiplicación, división, etcétera. Pero hay una ventaja: se pueden aplicar a cualquier tipo de variable que acepte el programa, siempre que la aplicación tenga sentido matemático. El orden de precedencia es el usual: $*$ y $/$ se evalúan antes de $+$ y $-$, y así sucesivamente. Pero es muy recomendable agrupar las expresiones usando paréntesis, es decir, `()`. Para explicar por qué se muestra cómo cambia el resultado de la evaluación de una expresión sencilla al usar tres formas distintas de agrupación:

```
-->1+2*5
ans = 11.
-->(1+2)*5
ans = 15.
-->1+(2*5)
ans = 11.
```

Así que en caso de duda lo mejor es poner todos los paréntesis necesarios. También hay funciones predefinidas que se combinan con los operadores. Por ejemplo, la función exponencial se invoca como `exp(x)`, donde `x` es el argumento:

```
-->exp(1)
ans = 2.7182818
-->5 + ( ( exp(2) / 7 ) ^ 2 )
ans = 6.114248
```

En la tabla 1.2 se muestran las funciones más comunes, pero la lista completa está en la sección `Elementary functions` del sistema de ayuda. Hay funciones que requieren más de un argumento, pero por simplicidad se deja ese tema para la sección 4.

Tabla 1.2. Algunas funciones comunes

Función	Significado
`exp(x)`	exponencial
`log(x)`	logaritmo (natural)
`log10(x)`	logaritmo (base 10)
`sin(x)`	seno
`cos(x)`	coseno
`tan(x)`	tangente
`sqrt(x)`	raíz cuadrada

La característica clave de Scilab es que permite variables definidas por el usuario. El explorador de variables, figura 1.6, muestra las variables presentes en la memoria. El editor de variables, figura 1.7, es una interfaz gráfica similar a una hoja de cálculo para crear, borrar y manipular variables. Además del menú `Applications`, el editor también se activa al hacer doble clic en el nombre de una variable en el explorador. El editor de variables y la consola pueden producir los mismos resultados, pero la consola es preferible por su flexibilidad.

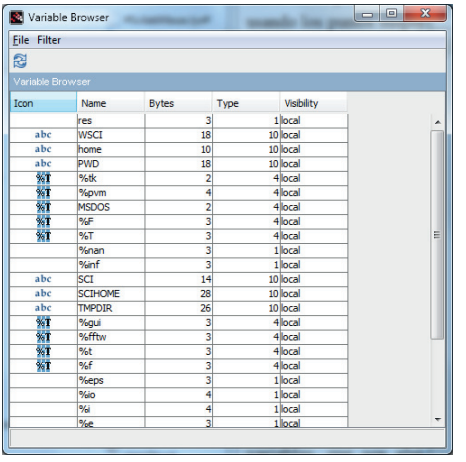


Figura 1.6. Explorador de variables

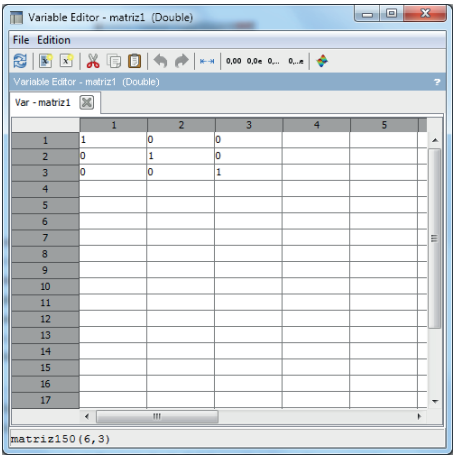


Figura 1.7. Editor de variables (Variable Editor).

En la consola se definen variables con la sintaxis.

```
nombreVariable = valorVariable
```

después el valor se obtiene invocando `valorVariable`. Por ejemplo:

```
-->a=5;
-->b=6;
-->a
a = 5.
-->b
```

```
b = 6.
-->a+b
ans = 11.
```

El ; sirve también para separar operaciones y asignaciones dentro de una sola línea del *prompt*. El ejemplo anterior se puede escribir como

```
-->a=5; b=6; c=a+b;
-->c
c = 11.
```

Los nombres de variables solo pueden tener caracteres ASCII. Por eso, letras como la ñ o la ç no se aceptan. Pero se pueden diferenciar por mayúsculas y minúsculas, `estenombre` y `EsteNombre` se consideran distintos.

Scilab tiene la capacidad, poco conocida, de hacer asignaciones múltiples usando la sintaxis (van Dijk y Spiel, 2000):

```
[nombreVar1, nombreVar2, ... nombreVarN] = (valor1, valor2,
..., valorN)
```

lo que permite ahorrar órdenes separadas:

```
-->[uno, deux, three] = (1, 2, 3)
three = 3.
deux = 2.
uno = 1.
```

Las órdenes largas se pueden escribir en más de un renglón de la consola usando los puntos suspensivos (en realidad, tres puntos seguidos). La consola aplaza la ejecución hasta la siguiente vez que se pulse `enter`:

```
-->res = 2 + 5 - ( ( 5 / 8 ) + 3 - ...
-->10 )
res = 13.375
```

Incluso se pueden usar varios ... para continuar la expresión en más de dos renglones:

```
-->res = 2 + 5 - ( ( 5 / 8 ) + 3 - ...
-->10 ) + 5.238 - 4 / ( 10 + 5 ) - ...
-->5.55 - 3.2
res = 9.5963333
```

La memoria disponible para el usuario está representada en un *stack*[3] cuyo tamaño se asigna en forma automática al arrancar. Es poco probable que se necesite

---

3 En español se diría "pila", pero se prefiere el término en inglés *stack* porque así aparece en el sistema de ayuda.

manipular el tamaño del *stack*, excepto en aplicaciones muy especializadas que requieran, por ejemplo, matrices gigantescas de millones de elementos. En esos casos, se pueden usar las funciones `stacksize`, y `gstacksize`. Con la orden `who`, se muestra en pantalla la lista de todas las variables que están en la memoria. Las variables del usuario se muestran con `who_user`:

```
-->who_user
User variables are: uno deux three WSCI
Using 30 elements out of 4990699
```

Aquí `WSCI` representa el directorio donde Scilab guarda y lee archivos por omisión. La orden `clear`, invocada sin argumentos borra TODAS las variables, así que debe ser usada con precaución:

```
-->clear
-->deux
!--error 4 Undefined variable: deux
-->one
!--error 4 Undefined variable: one
-->three
!--error 4 Undefined variable: three
```

Pero cuando se invoca en la forma
`clear(‘‘nombreVariable’’)`
solo borra la variable identificada entre comillas en el argumento:

```
-->[uno, deux, three] = (1, 2, 3);
-->three three = 3.
-->clear("three")
-->three
!--error 4 Undefined variable: three
-->deux
deux = 2.
```

Para borrar varias variables se ponen como argumento sus nombres separados con comas. Las variables se pueden proteger contra el borrado usando `predef(n)` donde `n` representa cuántas de las variables más recientes del *stack* serán protegidas. Esta orden también se puede aplicar a todas las variables, tanto para proteger, `predef(‘‘all’’)`, como para retirar la protección, `predef(‘‘clear’’)`.

GUÍA DE AUTOAPRENDIZAJE

1. Explore el sistema de ayuda. Por ejemplo, busque las entradas de funciones comunes.

2. Defina tres variables con los nombres `a`, `b` y `c`.

3. Suponiendo que las variables del ejemplo anterior representan los términos de la ecuación $ax^2 + bx + c = 0$ obtenga las raíces con la fórmula cuadrática $x = -b/2a \pm \sqrt{b^2 - 4ac}/2a$.

4. Borre las variables y luego vuélvalas a crear con valores distintos. Repita el cálculo de las raíces.

Capítulo 2

Operaciones de archivo

Al finalizar este capítulo usted deberá ser capaz de

- Guardar y cargar los contenidos de la memoria en un archivo binario.
- Cambiar el directorio de trabajo.
- Guardar variables específicas en archivos de texto.
- Importar datos desde archivos de texto y hojas de cálculo.

Como en todo sistema de computación numérica la memoria de Scilab es una fracción de la memoria disponible del computador. Pero hay un detalle de importancia clave: la memoria de Scilab no es persistente. Las calculadoras tienen una memoria persistente porque su estado –es decir, las variables de su memoria– no se borra al apagarlas. En cambio, al cerrar Scilab el contenido del *stack* se pierde, a menos que el usuario lo haya guardado en un archivo. Por eso es necesaria una sección dedicada al tema de escritura de archivos.

Para empezar, las operaciones de lectura y escritura se hacen, por omisión, en una ubicación del sistema de archivos llamada aquí "directorio de trabajo". El File Browser, figura 2.1, muestra el directorio de trabajo y permite abrir archivos en él. Además, el menú `File` incluye las opciones

`Change current directory`

y

`Display current directory`

para mostrar y cambiar el directorio de trabajo usando una ventana de diálogo. También se puede escribir a mano la ubicación, usando las órdenes `pwd` y `cd`:

```
-->pwd()
ans =
C:\Program Files\scilab-5.3.2
-->cd("C:\Documents and Settings\Javi\
Mis documentos\publicaciones\ManualScilab")
ans =
C:\Documents and Settings\Javi\
Mis documentos\publicaciones\ManualScilab
```

Pero la forma de escribir la ubicación del directorio de trabajo depende del sistema operativo. Por eso quizás es más recomendable usar la ventana de diálogo o el File Browser.

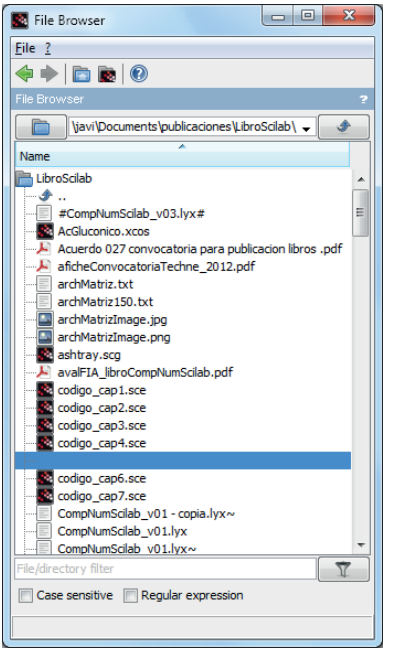


Figura 2.1. File Browser (navegador de ficheros).

Las "variables" son los objetos que existen en la memoria de Scilab: números, polinomios, matrices, incluso las funciones mismas. Hay dos formas principales de leer y escribir variables en una unidad de almacenamiento (tabla 2.1):

- En forma binaria. Las variables se guardan en disco tal y como aparecen en memoria, es decir, como cadenas de bits. Los datos binarios solo pueden ser interpretados por Scilab mismo, pero tienden a ocupar menos espacio que en forma de texto.
- Como texto. Las variables pueden ser leídas por humanos o interpretadas por otros programas. Esto permite intercambiar datos con otros software, digamos hojas de cálculo o bases de datos. Sin embargo, las funciones cargadas en la memoria –es decir, "compiladas"– no se pueden guardar como texto.

Tabla 2.1. Algunas órdenes para escribir variables en el disco duro, en forma binaria y como texto

Acción	Binario	Texto
Escritura	`save`	`fprintfMat write`
Lectura	`load`	`fscanfMat read`

En el menú `File` la opción `Save environment` escribe todo el contenido de la memoria en un archivo de formato binario. La opción `Load environment` carga todos los contenidos de un archivo binario en la memoria, con el riesgo implícito de sobrescribir variables sin previo aviso. Las órdenes `save` y `load` sirven para hacer lo mismo desde el *prompt* cuando se invocan con la sintaxis[1].

```
save("nombreArchivo.sod")
```

y

```
load("nombreArchivo.sod")
```

Las variables también se pueden invocar con argumentos adicionales para leer o escribir variables específicas, como se muestra en este ejemplo[2]:

```
-->matriz1 = eye(3,3)
mat3 =
1. 0. 0.
0. 1. 0.
0. 0. 1.
-->matriz2 = rand(2,2)
matriz2  =
  0.2113249    0.0002211
  0.7560439    0.3303271
-->save("guardado.sod", "matriz1", "matriz2")
-->matriz1 = [1 4 5]
matriz1 =
! 1. 4. 5. !
-->load("guardado.sod", "matriz1", "matriz2")
-->matriz1
matriz1 =
1. 0. 0.
0. 1. 0.
0. 0. 1.
```

Al cargar el archivo con los valores originales se sobrescribió el contenido de la variable `matriz1`. Ahora se muestra el guardado de todas las variables

```
-->save("todos.sod")
-->clear
-->matriz1
!--error 4  Undefined variable: matriz1
```

---

1 La extensión `.sod` representa el nuevo formato binario de Scilab, basado en el estándar HDF5. Se recomienda seguirlo porque el formato anterior (sav) está depreciado y será eliminado en la versión 6. Para más información lea la entrada de `save` en el sistema de ayuda.

2 Los números generados por `rand` son seudoaleatorios, así que los valores que usted obtenga pueden ser distintos. También aquí se anticipan algunas cosas sobre matrices que se tratan en la sección 3.2. La orden `eye` crea una matriz identidad, y los paréntesis cuadrados `[]` sirven para agrupar números en una matriz.

```
-->matriz2
!--error 4  Undefined variable: matriz2
-->load("todos.sod")
-->matriz1
matriz1  =
1.    0.    0.
0.    1.    0.
0.    0.    1.
-->matriz2
matriz2  =
0.6653811    0.8497452
0.6283918    0.6857310
```

Después de borrar las variables de la memoria se recuperan usando `load`.

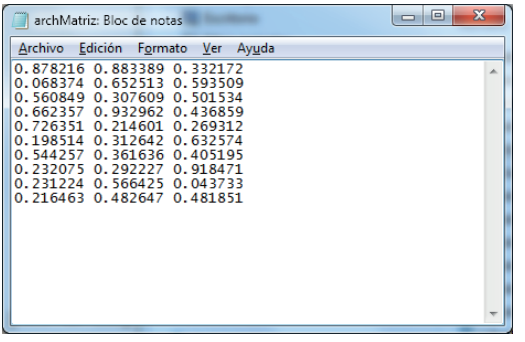


Figura 2.2. Matriz guardada como archivo de texto, mostrada en el bloc de notas de Windows

De otro lado, hay varias órdenes disponibles para guardar variables como texto (ver tabla 2.1). `fprintfMat` escribe una matriz en un archivo de texto con la sintaxis

```
fprintfMat(‘‘nombreArchivo’’, matriz, [format, text]),
```

donde los argumentos `format` y `text` (formato y texto de encabezado) son opcionales. Por ejemplo

```
-->ejMatriz = rand(10,3);
-->fprintfMat(’’archMatriz.txt’’, ejMatriz)
```

produce un archivo de texto con los valores de `ejMatriz`, que se puede leer en cualquier editor de texto (figura 2.2) o importar a una hoja de cálculo. La contraparte para leer matrices desde archivos de texto es la orden `fscanfMat`, que tiene la sintaxis

```
matriz = fscanfMat(nombreArch)
```

Continuando el ejemplo anterior, se muestra cómo recuperar la matriz desde el archivo

```
-->clear
-->ejMatriz
!--error 4 Undefined variable: ejMatriz
-->ejMatriz = fscanfMat("archMatriz.txt")
ejMatriz =
0.878216 0.883389 0.332172
0.068374 0.652513 0.593509
0.560849 0.307609 0.501534
0.662357 0.932962 0.436859
0.726351 0.214601 0.269312
0.198514 0.312642 0.632574
0.544257 0.361636 0.405195
0.232075 0.292227 0.918471
0.231224 0.566425 0.043733
0.216463 0.482647 0.481851
```

Incluso se puede obtener el encabezado no numérico del archivo diferenciando líneas de texto de líneas de números al hacer
`[nombreMatriz, encabezado] = fscanfMat(nombreArch)`.

La orden `write` escribe una matriz fila a fila en un archivo de texto. La sintaxis es

```
write(‘‘NombreArchivo’’, matriz, [formato])
```

El argumento `formato` aparece entre paréntesis cuadrados porque es opcional, pero en realidad es indispensable cuando `matriz` tiene muchas columnas. Para ilustrar esto se muestra aquí la escritura de una matriz de 150 filas y 10 columnas, pero usando `write` sin formato:

```
-->matriz150=rand(150,10);
-->write("archMatriz150.txt", matriz150)
```

Al abrir el archivo `archMatriz150.txt` se nota de inmediato que es inservible, pues quedó dividido en 600 filas desiguales, cuando debería tener 150 filas de 10 números cada una. Esto se debe a que el espacio asignado por omisión es insuficiente para más de 6 columnas y el programa lo resuelve partiendo las filas. Para resolver esto hay que dar formato de las filas así:

formato = “(nCol1(espec1), nCol2(espec2),...,nColN(especn))”

(sin los puntos suspensivos) con las convenciones del lenguaje FORTRAN: la suma de `nCol1`, `nCol2`,... debe ser igual al número de columnas de la matriz, `espc` es la especificación,

`espc = TIPOm.n`

donde

- `m`: número de lugares reservados para el número.
- `n`: número de cifras decimales.
- `TIPO` es una letra mayúscula que identifica el tipo de formato numérico:
  - `F`: punto flotante, sin exponencial. Produce, por ejemplo: .7981248387
  - `E`: notación exponencial. Produce, por ejemplo: .75278E+05
  - `I`: números enteros (si solo hay enteros en la matriz). Produce, por ejemplo: 412. No necesita la especificación del número de decimales.

Para volver al ejemplo, borre del disco duro el archivo `archMatriz150.txt`, y escríbalo de nuevo formateado con la instrucción

```
-->write("archMatriz150.txt", matriz150,"(10(F25.15))")
```

así se produce un archivo ordenado, cada fila tiene diez números con el mismo formato (verifíquelo abriéndolo en un editor de texto). También es posible usar combinaciones de formatos en una misma fila. Para el mismo ejemplo, borre del disco duro `archMatriz150.txt` y vuélvalo a escribir como

```
-->write("archMatriz150.txt", matriz150,
"(2(F22.10), 8(F25.15))")
```

al abrir el archivo en un editor de texto verá la diferencia en el espacio ocupado y el número de cifras decimales en las dos primeras columnas.

La contraparte de `write` es `read`, que carga una matriz a la vez:

```
[nombreVariable]=read(‘‘nombreArchivo’’, m, n, [formato])
```

Los argumentos `nombreVariable` y `formato` son opcionales. `m` y `n` son respectivamente los números de filas y columnas que se van a leer. Si `m=-1` se leen todas las filas. La especificación de `formato` es la misma que se explicó para `write`. Para ilustrar el uso de `read` se propone construir una nueva matriz leyendo las primeras cinco filas y tres columnas de la matriz guardada en el archivo `archMatriz150.txt` en el ejemplo anterior. Para asegurar que se lee del archivo, primero se borra la memoria

```
-->clear
-->nuevaMatriz = read("archMatriz150.txt", 5, 3)
nuevaMatriz =
0.0367152 0.8054929 0.9594440
```

```
0.3313874 0.0042473 0.7274638
0.0632192 0.7690750 0.6779780
0.1558463 0.9471904 0.0082783
0.2055220 0.9119136 0.9409017
```

Además de las funciones discutidas aquí (tabla 2.1) Scilab tiene docenas de funciones especializadas de lectura y escritura. Las secciones de la ayuda relacionadas con ese tema son

- `Files: Input/Output functions:` todas las órdenes relacionadas con leer o escribir variables.
- `Matlab binary files I/O:` órdenes compatibles con el formato MAT del software Matlab.
- `Spreadsheet:` órdenes especializadas para leer/escribir archivos en formatos de hoja de cálculo, por ejemplo Excel u OpenOffice Calc.

Una tarea muy común es intercambiar datos entre Scilab y una hoja de cálculo. Para esto se pueden leer datos usando las funciones `xls_open`, `xls_read` y `readxls` si el el archivo de hoja de cálculo es del tipo xls (Excel 97-2003). Pero no hay funciones específicas para leer archivos con el formato de Excel 2007 y Excel 2010, aunque sí para archivos xml (ver `xmlRead`). Pero quizás la forma más segura de intercambiar datos es usar archivos de texto como intermediarios. En ese caso la única fuente de incompatibilidad puede ser el formato del separador decimal, Scilab solamente acepta el punto, mientras que las hojas de cálculo pueden usar punto o coma. De ser necesario, el usuario debe ajustar el programa de hoja de cálculo.

GUÍA DE AUTOAPRENDIZAJE

1. En su disco duro cree una carpeta (es decir un directorio) llamado `aprenderScilab` y luego conviértalo en su directorio de trabajo.
2. Genere una matriz de números al azar de 1000 filas y 10 columnas y guárdela en formato binario y como texto, usando solo 4 cifras decimales.
3. Reinicie Scilab y cargue la matriz desde el archivo binario y desde el archivo de texto.
4. Dentro de un programa de hoja de cálculo importe la matriz desde el archivo de texto que creó. *Advertencia*: puede haber una inconsistencia si el programa usa coma en lugar de punto como separador decimal.
5. Haga la operación contraria: dentro de la hoja de cálculo genere datos numéricos, expórtelos como texto y luego impórtelos a Scilab.
6. Repita lo anterior desde un archivo xls en lugar de un archivo de texto.

Capítulo 3

TIPOS DE DATOS

Al finalizar este capítulo usted deberá ser capaz de

- Crear variables de los tipos lógico, numérico y de texto.
- Crear y concatenar matrices.
- Acceder a los elementos de una matriz.
- Hacer operaciones básicas de álgebra lineal, incluyendo la solución de un sistema de ecuaciones lineales.

3.1 Tipos básicos de variables

Los tipos de datos no requieren declaración o dimensionamiento (Gómez, 1999). Las variables[1] más simples son las booleanas, es decir los valores lógicos falso y verdadero. Por sus nombres en inglés (*true* y *false*) se representan con las constantes `%t` para verdadero y `%f` para falso. Los operadores lógicos en la tabla 3.1 sirven para hacer operaciones lógicas como estas:

```
-->%t &%t
ans = T
-->%t &%f
ans = F
-->%t <>%f
ans = T
-->~%t
ans = F
-->%t ==%t
ans = T
-->%t ==%f
ans = F
-->~ans
ans = T
```

---

[1] La lista completa de tipos de variables está disponible en las entradas `type` y `typeof` de la ayuda.

Tabla 3.1. Operadores lógicos

Operador	Explicación
&	"y" lógico, conjunción (and)
|	"o" lógico (or)
<>	"o" lógico (or)
~	negación (not)
==	igualdad
<	menor que
<=	menor o igual que
>	mayor que
>=	mayor o igual que

Los operadores lógicos también se aplican entre números, por ejemplo:

```
-->1<2
ans = T
-->1>2
ans = F
-->1==2
ans = F
-->1<>2
ans = T
-->1==1
ans = T
-->1<=2
ans = T
-->1>=2
ans = F
-->(1<2)&(3<4)
ans = T
-->(1<2)|(3<4)
ans = T
-->(1<2)&(3>=4)
ans = F
-->(1<2)|(3>=4)
ans = T
```

Curiosamente, las constantes `%t` y `%f` son aceptables dentro de expresiones algebraicas. En ese caso `%t` equivale a 1 y `%f` a 0, por ejemplo:

```
-->1 +%t
ans = 2.
-->1 +%f
ans = 1.
-->5 / ( 1 +%t )
ans = 2.5
```

pero este uso no es muy común.

Tabla 3.2. Ejemplos de formato numérico en la consola con la constante $\pi = 3.14159\ldots$

Formato	Presentación de `%pi`
`format("v",4)`	`3.1`
`format("v",15)`	`3.14159265359`
`format("e",4)`	`3.1D+00`
`format("e",15)`	`3.14159265D+00`

Nota: el primer argumento en `format` es el tipo de formato, "v" para exponente. El segundo argumento es la cantidad máxima de caracteres que ocupará el número. Por omisión es 10.

El *prompt* en la consola se puede usar como una calculadora cualquiera con variables numéricas reales y complejas, aunque hay algunas peculiaridades:

- El separador decimal es siempre el punto. No se aceptan comas, para evitar confusión con la separación de números al construir matrices. Como ya se escribió, esto mismo aplica para la escritura y lectura de archivos de texto. Ese es un detalle importante al intercambiar datos con programas que usen la coma como separador decimal.
- Al escribir números en notación científica se puede usar la letra `E` o la letra `D` para identificar el exponente. Por ejemplo, el número 1538 se puede escribir como `1.538E3` o `1.538D3`.
- El formato de los números en la consola se puede cambiar con la orden `format` (ver la tabla 3.2).
- Al intentar una operación no definida –por ejemplo, al intentar una división por 0–, ocurre un error de punto flotante. El comportamiento, por omisión, es detener la ejecución y mostrar un mensaje de error. Pero se puede cambiar con la orden `ieee(modo)`. Con `ieee(1)` o `ieee(2)` se asigna un valor `%inf` (infinito) o `%NaN` (*not a number*, un "no número"). El modo 0, es decir `ieee(0)`, es el comportamiento por omisión.

Por omisión, los resultados son numéricos[2] pero, a diferencia de FORTRAN o C++, el lenguaje de Scilab no está fuertemente tipificado. No hace falta declarar el tipo de cada variable y los resultados tienen "tipificación dinámica", es decir, su tipo puede ser distinto al de los argumentos. Por ejemplo, el cociente de dos números enteros da un resultado real:

---

[2] Hay unas pocas funciones simbólicas (entrada `Symbolic` en la ayuda) y para para manipular polinomios (sección 3.4). Por simplicidad, se omite la construcción de interfases entre Scilab y sistemas de manipulación simbólica tales como Maxima o Maple.

```
--> 1 + ( 5 / 3 )
ans = 2.6666667
```

El programa produce las cifras decimales. El logaritmo de un número real negativo da como resultado un número complejo

```
-->log(-10)
ans = 2.3025851 + 3.1415927i
```

El programa produce la parte imaginaria (la constante `%i` representa el valor imaginario $i = \sqrt{-1}$). Y operaciones entre números reales y complejos dan un resultado complejo:

```
-->5 + ( 3 / ( 1 + 2*%i ) )
ans = 5.6 - 1.2i
```

Es necesario tener en cuenta que las operaciones con variables reales implican errores de redondeo. Por ejemplo, $\sin(\pi)$ no da exactamente 0, sino 1.225×10^{-16}:

```
-->%pi
%pi =
3.1415927
-->sin(%pi)
ans = 1.225D-16
```

(la notación `1.225D-16` se lee 1.225×10^{-16}), pero el resultado es correcto porque $\sin(\pi) \approx 0$. El valor obtenido es incluso menor a la constante `%eps` (la cantidad más grande tal que que sumada a 1 no cambia al resultado).

```
-->%eps
%eps =
2.220D-16
-->1+%eps
ans = 1.
```

En la mayoría de los casos, el usuario se puede desentender de las minucias del redondeo. Pero los interesados pueden encontrar una descripción detallada de la aritmética de punto flotante de Scilab en las referencias (Baudin, 2010; van Dijk y Spiel, 2000).

También existen variables enteras –es decir, números que literalmente no tienen parte decimal, en oposición a números reales cuya parte decimal es cero (Baudin, 2010; van Dijk y Spiel, 2000). Al combinar este tipo de variables, la tipificación dinámica deja de funcionar. Esto se muestra en un ejemplo simple, la operación $5/2$. Con variables reales (la forma por omisión) se obtiene:

```
-->a = 5; b = 2
b = 2.
-->a / b
ans = 2.5
```

Aquí a y b son números reales sin parte decimal. Por eso b se muestra como 2. (con punto) y el resultado de la operación también es real, con una parte decimal. Al repetir el ejemplo usando la función uint8(x) para producir variables enteras de ocho bits[3] sin signo, se obtiene esto:

```
-->c = uint8(5); d = uint8(2)
d = 2
-->c / d
ans = 2
```

aquí c y d son enteros y no tienen parte decimal. Por eso d aparece como 2 (sin punto) y el resultado[4] de la operación, 2, es también un entero.

Las cadenas de caracteres se delimitan con comillas[5] dobles y se pueden concatenar usando +

```
-->a = "con"
a = con
-->b = "Cate";
-->c = "Nado";
-->a+b+c
ans = conCateNado
```

En la sección Strings de la ayuda del programa se describen todas las funciones relacionadas con cadenas de caracteres.

Cada tipo de variable tiene dos identificaciones: un número entero y una cadena de caracteres. Por ejemplo, el tipo "matriz de números reales y complejos" se identifica con 1 y constant. Las funciones type y typeof sirven para identificar el tipo de una variable: type(x) devuelve el número identificador de x y typeof, la cadena de caracteres. Por ejemplo:

```
-->miVar = [2 0; 9 7];
-->type(miVar)
ans = 1.
-->typeof(miVar)
ans = constant
```

---

3 Un bit tiene un solo valor booleano, 0 o 1. Por lo tanto, en una palabra de 8 bits 5 se representa como 00000101 y 2 como 00000010. Para obtener enteros de 16 y 32 bits sin signo existen las funciones uint16(x) y uint32(x), y con signo, int16(x) e int32(x).

4 El residuo de la división entera se obtiene con la función módulo.

5 Las comillas sencillas también sirven para delimitar texto, pero no se recomienda usarlas, por el riesgo de confundirlas con el operador de transposición matricial.

Tabla 3.3. Constantes incorporadas.

Constante	Significado
`%t`	verdadero
`%f`	falso
`%i`	$\sqrt{-1}$
`%e`	$e = 2.7182818...$
`%pi`	$\pi = 3.1415927...$
`%eps`	épsilon de máquina
`%inf`	infinito
`%nan`	NotANumber

Nota: La constante `%eps` es tal que `1+%eps=1`. `%inf` y `%nan` aparecen cuando el resultado de una operación es demasiado grande o produce un error de desbordamiento numérico.

La lista completa de identificadores está en las entradas de `type` y `typeof` en el sistema de ayuda. Para el tipo entero también existe la función `inttype(x)`, que sirve para identificar el subtipo de entero, es decir, si es de 1 byte, 2 bytes, etcétera.

Las constantes predefinidas son números reales o cadenas de caracteres (Gómez, 1999). Se identifican con el prefijo `%`. Por ejemplo, el número π corresponde a `%pi`, y así para todas las demás (ver tabla 3.3). Las constantes se pueden utilizar como cualquier otro número, como se muestra a continuación:

```
-->1+%pi
ans = 4.1415927
-->%e
%e =
2.7182818
-->2+%e
ans = 4.7182818
-->log(%e)
ans = 1.
```

Nota: para evitar confusión con las constantes predefinidas `%` no debe usarse en nombres de variables.

3.2 Matrices

Una matriz es un arreglo rectangular de variables, en principio números reales o complejos. Scilab también acepta matrices de valores lógicos (booleanos) y de cadenas de caracteres (Gómez, 1999). La combinación de variables de distinto tipo en una matriz es posible con valores lógicos y números. En este caso, los valores lógicos se interpretarán como los números 0 y 1, pero otras combinaciones –por ejemplo, valores lógicos y cadenas de caracteres– no se aceptan. La forma de acceso

a los elementos de la matriz es igual en todos los casos, así que en este capítulo solamente se usan como ejemplos matrices de números.

Una matriz se define agrupando el conjunto completo de valores dentro de paréntesis `[]`. Las filas se separan con punto y coma; los elementos dentro de una fila se separan con espacios o comas (Baudin, 2010; Mora, 2005; Rodríguez, 2007). Por ejemplo, para asignar la matriz

$$\begin{bmatrix} 9 & 4 & 8 \\ 7 & 2 & 11 \end{bmatrix}$$

a una variable llamada `matriz` se hace esto:

```
-->matriz = [9 4 8; 7 2 11]
matriz =
9. 4. 8.
7. 2. 11.
```

Las matrices también se pueden construir concatenando matrices más pequeñas. Para el mismo ejemplo anterior se pueden construir primero las filas y luego unirlas:

```
-->fila1 = [9 4 8];
-->fila2 = [7 2 11];
-->matriz = [fila1; fila2]
matriz =
9. 4. 8.
7. 2. 11.
```

O bien se puede construir una matriz a partir de columnas:

```
-->col1 = [2; 2; 8]
col1 =
2.
2.
8.
-->col2 = [5; 1; 2];
-->col3 = [2; 7; 4];
-->[col1 col2 col3]
ans =
2. 5. 2.
2. 1. 7.
8. 2. 4.
```

Nota: una matriz de una sola fila es llamada a veces "vector fila" y una matriz de una sola columna, "vector columna".

Tabla 3.4. Operadores para matrices

Operador	Significado
`[]`	concatenación
`()`	extracción/inserción de elementos
`'`	transposición
`,`	separador de elementos
`;`	separador de filas
`$`	último elemento
`+`	suma
`-`	resta
`*`	producto
`.*`	producto (elemento a elemento)
`.*.`	producto de Kronecker
`^`	exponente
`.^`	exponente (elemento a elemento)
`/`	división
`./`	división (elemento a elemento)
`\`	división "izquierda"

Por último, varias matrices se pueden unir en una sola concatenándolas dentro de `[]` (ver tabla 3.4):

```
-->sm1 = [3 2; 7 1; 4 5];
-->sm2 = [9; 3; 5];
-->[sm1 sm2]
ans =
3. 2. 9.
7. 1. 3.
4. 5. 5.
```

Definir matrices a mano no es práctico cuando tienen cientos o miles de elementos. Por eso existen funciones que producen tipos especiales de matrices, entre otras:

- `zeros(m,n)`: crea una matriz de `m` filas y `n` columnas cuyos elementos son todos 0.
- `ones(m,n)`: lo mismo que `zeros`, pero los elementos son todos 1.
- `eye(m,n)`: genera una matriz identidad, en la que todos los elementos son 0, excepto los de la diagonal principal, que son iguales a 1.
- `rand(m,n)`: los elementos son todos números seudoaleatorios. Como se había explicado antes, el resultado cambia cada vez que se invoca.

Por ejemplo, especificando tres filas y tres columnas se obtiene:

```
-->zeros(3,3)
ans =
0. 0. 0.
0. 0. 0.
0. 0. 0.
-->ones(3,3)
ans =
1. 1. 1.
1. 1. 1.
1. 1. 1.
-->eye(3,3)
ans =
1. 0. 0.
0. 1. 0.
0. 0. 1.
-->rand(3,3)
ans =
0.2113249 0.3303271 0.8497452
0.7560439 0.6653811 0.6857310
0.0002211 0.6283918 0.8782165
```

También es posible crear matrices vacías con la construcción `[]` y luego llenarlas usando la instrucción:

```
nombreMatriz(i,j) = valor
```

donde `i,j` representan respectivamente la fila y la columna donde se pondrá el valor respectivo:

```
-->matVacia = []
matVacia = []
-->matVacia(2,2) = 8
matVacia =
0. 0.
0. 8.
-->matVacia(1,3) = 5
matVacia =
0. 0. 5.
0. 8. 0.
```

De hecho, el programa asignará valores 0 a cada uno de los elementos no especificados de la matriz.

La mayoría de las matrices son entes bidimensionales organizados en filas y columnas, pero también existen hipermatrices[6] de tres o más dimensiones. Una

---

6 Las hipermatrices son en realidad listas matriciales. Ver la sección 7.1.

hipermatriz se puede generar con la orden `hypermat`, o añadiendo argumentos a cualquier función que genere una matriz. Esto último es quizás más conveniente. Por ejemplo una matriz de ceros de 3x3x3 es el resultado de esta instrucción:

```
-->ejHipMat = zeros(3,3,3)
ejHipMat =
(:,:,1)
0. 0. 0.
0. 0. 0.
0. 0. 0.
(:,:,2)
0. 0. 0.
0. 0. 0.
0. 0. 0.
(:,:,3)
0. 0. 0.
0. 0. 0.
0. 0. 0.
```

El acceso y la concatenación son análogos a los de las matrices bidimensionales:

```
-->ejHipMat(:,:,2) = [1 2 5; 0 7 3; 6 6 8];
-->ejHipMat(1,1,1) = 777;
-->ejHipMat
ejHipMat =
(:,:,1)
777. 0. 0.
0.   0. 0.
0.   0. 0.
(:,:,2)
1. 2. 5.
0. 7. 3.
6. 6. 8.
(:,:,3)
0. 0. 0.
0. 0. 0.
0. 0. 0.
-->ejHipMat(1,:,1)
ans =
777. 0. 0.
```

solo que en lugar de dos índices (filas y columnas) tienen tres o más.

Otro tipo especial es la "matriz dispersa" (en inglés *sparse*). cuyos elementos son casi todos iguales a cero (Gómez, 1999). Por ejemplo, una matriz de 1000x1000 con solo 45 elementos *distintos* de 0 es dispersa: tiene un millón de elementos pero solo 45 no son nulos. Este tipo de matrices se pueden definir en la forma usual, ya que al fin y al cabo 0 es un número como cualquier otro. Pero esto tiene dos inconvenientes:

- A diferencia de la memoria humana, en la memoria del computador el 0 ocupa el mismo espacio que cualquier otro número real.
- Al aplicar álgebra lineal, –por ejemplo, al multiplicar dos matrices– buena parte de las operaciones serían multiplicaciones por 0, pero requieren el mismo tiempo de ejecución que una multiplicación no nula.

Para evitar estos dos inconvenientes, las variables tipo matriz dispersa solo contienen los valores *distintos* de cero y sus posiciones. Hay un conjunto de funciones especiales para manipularlas, que minimizan el tiempo de ejecución (ver la sección `Sparse matrix` en la ayuda). Por ejemplo, se muestra la creación de una matriz dispersa de cien elementos, de los cuales solo cuatro no son nulos, usando la función `spzeros`:

```
-->matrizDisp = spzeros(10,10);
-->matrizDisp(1,1) = 2; matrizDisp(2,3) = 4;
-->matrizDisp(3,3) = 1; matrizDisp(9,8) = 3;
-->matrizDisp
matrizDisp = ( 10, 10) sparse matrix
( 1, 1) 2.
( 2, 3) 4.
( 3, 3) 1.
( 9, 8) 3.
```

Aunque solo se guardan los elementos no nulos, el acceso se hace en la misma forma que en las matrices convencionales (ver la siguiente sección). Por ejemplo, para obtener el valor en la columna 3 de la fila 2 se escribe:

```
-->matrizDisp(2,3)
ans =
( 1, 1) sparse matrix
( 1, 1) 4.
```

Y para usarlo se invocar en la misma forma:

```
-->2 * matrizDisp(2,3)
ans = 8.
```

No hay diferencias mayores entre la manipulación de una matriz dispersa y las de una matriz convencional, aunque hay una excepción notoria: para resolver un sistema lineal definido con matrices dispersas se debe usar `lusolve` , en lugar del operador \ (este último se explica en la siguiente sección).

3.3 Manipulación de matrices

La numeración de las matrices empieza en 1. Es decir, la primera fila es la 1 la primera columna también se identifica con 1, y el primer elemento es el (1) o (1,1). Para acceder a los elementos de una matriz se emplea la sintaxis:

```
nombreMatriz(fila, columna)
```

Por ejemplo, para obtener el valor del elemento 3 en la fila 2 se escribe:

```
nombreMatriz(2, 3)
```

Y así sucesivamente:

```
-->matriz
matriz =
9. 4. 8.
7. 2. 11.
-->matriz(2,3)
ans = 11.
-->matriz(1,2)
ans  = 4.
```

El operador : (en inglés en la ayuda es *colon*. Ver también la tabla 3.4) hace tres operaciones clave para manipular y crear matrices. Primero, genera listas de números cuando se invoca en la forma:

```
elemInicial:avance:elemFinal
```

Por ejemplo:

```
-->1:10
ans =
1. 2. 3. 4. 5. 6. 7. 8. 9. 10.
-->10:-1:0
ans =
10. 9. 8. 7. 6. 5. 4. 3. 2. 1. 0.
-->0:2:20
ans =
0. 2. 4. 6. 8. 10. 12. 14. 16. 18. 20.
```

Segundo, es un comodín para seleccionar filas y columnas completas. La fila i se obtiene así:

```
nombreMatriz(i,:)
```

Y la columna j:

```
nombreMatriz(:,j)
```

Por ejemplo:

```
-->matriz = [1 2 3; 4 5 6; 7, 8, 9]
matriz =
1. 2. 3.
4. 5. 6.
7. 8. 9. -
->matriz(2,:)
ans = 4. 5. 6.
-->matriz(:,1)
ans =
1.
4.
7.
```

Y tercero, : selecciona porciones de matrices:

```
-->matriz(1:2, 2:3)
ans =
2. 3.
5. 6.
```

Si la selección tiene filas y columnas no contiguas se usa una lista –es decir, un vector– con sus números:

```
-->misCols = [1 3]
misCols = 1. 3.
-->matriz(:, misCols)
ans =
1. 3.
4. 6.
7. 9.
```

El operador : también lista todos los elementos de la matriz en un solo vector:

```
-->matriz(:)
ans  =
1.
4.
7.
2.
5.
8.
3.
6.
9.
```

El orden del resultado corresponde a la numeración de los elementos de la matriz en la memoria. Como están organizados por columnas, el elemento 1 es el (1,1), el 2 es el (2,1), y así sucesivamente. Por eso mismo la sintaxis:

```
nombreMatriz(n)
```

permite acceder al n-ésimo elemento dentro de ese esquema de numeración, incluso para hipermatrices. Para el mismo ejemplo anterior:

```
-->matriz(1)
ans = 1.
-->matriz(4)
ans = 2.
-->matriz(9)
ans = 9.
```

y el último elemento es el 9 porque la matriz tiene tres filas por tres columnas.

De otro lado, el operador $ da acceso al último elemento, fila o columna (Baudin, 2010; van Dijk y Spiel, 2000). Para continuar con el ejemplo anterior:

```
-->matriz($)
ans = 9.
-->matriz($,$)
ans = 9.
-->matriz(1,$)
ans = 3.
-->matriz(3,$)
ans = 9.
-->matriz($,2)
ans = 8.
```

El tamaño de una matriz se puede averiguar con las funciones `length` (número de elementos) y `size` (número de filas y número de columnas):

```
-->length(matriz)
ans = 9.
-->[numFilas, numCols] = size(matriz)
numCols = 3.
numFilas = 3.
```

El operador ' traspone la matriz, es decir, convierte el elemento (i,j) en el elemento (j,i):

```
-->matriz
matriz =
```

```
1. 2. 3.
4. 5. 6.
7. 8. 9.
-->matriz'
ans =
1. 4. 7.
2. 5. 8.
3. 6. 9.
```

y las funciones `flipdim` `permute`, y `resize_matrix` sirven para manipular el orden y la forma de las matrices (consultar la ayuda).

El álgebra matricial se hace con los mismos operadores que se aplican a los escalares, pero las operaciones deben tener sentido matemático. Por eso, la suma y resta de matrices con + y - produce resultados solo si los argumentos tienen las mismas dimensiones:

```
-->A = [6 7 2; 5 1 2; 8 2 3]
A =
6. 7. 2.
5. 1. 2.
8. 2. 3.
-->B = [1 4 7 ; 5 2 9; 3 1 0]
B =
1. 4. 7.
5. 2. 9.
3. 1. 0.
-->C = [10 7; 6 7]
C =
10. 7.
6. 7.
-->A + B
ans =
7. 11. 9.
10. 3. 11.
11. 3. 3.
-->A + C
!--error 8 Inconsistent addition.
```

En el último caso, `A+C` no tiene sentido porque `A` tiene 3×3 elementos y `C` 2×2. La multiplicación también requiere que las dimensiones de los operandos sean iguales:

```
-->A * C
!--error 10  Inconsistent multiplication.
-->A * B
ans =
47. 40. 105.
16. 24. 44.
27. 39. 74.
```

(El elemento `(i,j)` es el producto escalar de la fila `i` de `A` por la columna `j` de `B`).

El punto sirve para modificar los operadores en este sentido:

> *Cuando un operador está precedido por un punto, la operación se hará elemento a elemento.*

Por eso existen varios tipos de multiplicación y división (ver tabla 3.4). Por ejemplo, si se escribe `A .* B`, el elemento `(i,j)` del resultado será `A(i,j) * B(i,j)`, en lugar del producto escalar `(fila i)*(columna j)`:

```
-->A .* B
ans =
6. 28. 14.
25. 2. 18.
24. 2. 0.
```

Lo mismo aplica para la división:

```
-->1 / [2 3 5]
ans =
0.0526316
0.0789474
0.1315789
-->1 ./ [2 3 5]
ans = 0.5 0.3333333 0.2
```

En el primer caso se calculó el vector tal que multiplicado por [2 3 5] produce 1, mientras que en el segundo caso se calculó el vector cuyos elementos son 1/2, 1/3 y 1/5. El comportamiento "puntual" se aplica también en otros operadores. Por ejemplo, si $\mathbf{A}$ es una matriz cuadrada entonces $\mathbf{A}^2 = \mathbf{AA}$, $\mathbf{A}^3 = \mathbf{AAA}$, etcétera. Es decir, da lo mismo multiplicar repetidamente la matriz que usar la potencia.[7]

```
-->A^3
ans =
1043. 726. 376.
650. 433. 236.
1044. 696. 379.
-->A*A*A
ans =
1043. 726. 376.
650. 433. 236.
1044. 696. 379.
```

Pero al usar `.^n`, el elemento `(i,j)` del resultado es simplemente $A(i,j)^n$, como se muestra aquí:

---

7 En la entrada `power` de la ayuda se da una descripción detallada del funcionamiento de `^`.

```
-->A .^ 3
ans =
216. 343. 8.
125. 1. 8.
512. 8. 27.
```

El punto cambia por completo el significado de una operación y sobra decir que debe ser usado con mucha precaución.

La inversa de una matriz $\mathbf{A}$, llamada $\mathbf{A}^{-1}$, es la matriz tal que $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ donde $\mathbf{I}$ es una matriz identidad de las mismas dimensiones de $\mathbf{A}$. La inversa se puede obtener de dos formas, con la función `inv` o elevando a la potencia -1:

```
-->inv(A)
ans =
- 0.2 - 3.4 2.4
0.2 0.4 - 0.4
0.4 8.8 - 5.8
-->A^(-1)
ans =
- 0.2 - 3.4 2.4
0.2 0.4 - 0.4
0.4 8.8 - 5.8
-->A * inv(A)
ans =
1. 0. - 1.776D-15
0. 1. 0.
0. 0. 1.
```

(El valor del elemento (1,3) es prácticamente 0). Con la inversa se puede resolver un sistema de ecuaciones lineales de la forma $\mathbf{AX} = \mathbf{B}$ haciendo $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$. Por ejemplo, el conjunto de ecuaciones

$$\begin{aligned} 5x + 3y - 2z &= 10 \\ 3x - 2y &= 5 \\ x + y - 4z &= 8 \end{aligned} \tag{3.1}$$

se escribe como

$$\begin{bmatrix} 5 & 3 & -2 \\ 3 & -2 & 0 \\ 1 & 1 & -4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ 8 \end{bmatrix} \tag{3.2}$$

lo que equivale a la forma $\mathbf{AX} = \mathbf{B}$ con

$$\mathbf{A} = \begin{bmatrix} 5 & 3 & -2 \\ 3 & -2 & 0 \\ 1 & 1 & -4 \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} 10 \\ 5 \\ 8 \end{bmatrix},$$

y

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

En el *prompt* la solución se obtiene así:

```
-->A = [5 3 -2; 3 -2 0; 1 1 -4]
A =
5. 3. - 2.
3. - 2. 0.
1. 1. - 4.
-->B = [10 5 8]'
B =
10.
5.
8.
-->X = inv(A) * B
X =
1.4848485
- 0.2727273
- 1.6969697
```

Le queda como tarea al lector verificar la respuesta haciendo la multiplicación $\mathbf{AX}$. El resultado debe ser igual a $\mathbf{B}$. El operador \ o "división izquierda" (en inglés *backslash*) sirve para resolver sistemas lineales sin pasar por la inversión matricial. Para el mismo ejemplo anterior:

```
-->A \ B
ans =
1.4848485
- 0.2727273
- 1.6969697
```

y por supuesto da el mismo resultado.

Como comentario final, en este capítulo se ha hecho apenas una introducción a las capacidades de álgebra lineal de Scilab. Para los lectores interesados en profundizar en el tema y conocer más funciones se recomienda leer la sección `Linear Algebra`, de la ayuda y las referencias (Baudin, 2010; Rietsch, 2010).

3.4 POLINOMIOS

Un polinomio de grado n es una suma de términos de la forma

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{3.3}$$

donde los a_i son llamados coeficientes y x es la variable del polinomio. En Scilab los polinomios son considerados un tipo de variable aparte y se crean con la orden

```
poly(vectorValores, nombreVar, tipoCreacion)
```

donde:

- `vectorValores` contiene los valores de los coeficientes en este orden $[a_0, a_1, \ldots, a_n]$, o de las raíces del polinomio (ver `tipoCreacion`).
- `nombreVar` es la letra que representará a la variable del polinomio. Por ejemplo, ‘‘x’’, ‘‘s’’, o lo que prefiera el usuario.
- `tipoCreacion` especifica si el polinomio se creará a partir de sus coeficientes, asignando la cadena de caracteres ‘‘`coeff`’’ o de las raíces con ‘‘`roots`’’.

Por ejemplo, el polinomio $2+3x+5x^2+8x^3$ se define a partir de sus coeficientes de esta forma:

```
-->poli01 = poly([2 3 5 8], "x", "coeff")
                  2    3
ans = 2 + 3x + 5x + 8x
```

Y un polinomio cuyas raíces sean 2, 3 y 5 se crea así:

```
-->poli02 = poly([2 3 5], "x", "roots")
poli02 =
                 2   3
-30 + 31x - 10x + x
```

Para evaluar un polinomio se usa la función

```
horner(P, x)
```

que calcula el resultado de reemplazar el valor del argumento `x` en el polinomio de nombre `P`. Para seguir con el ejemplo anterior, digamos que se quiere obtener el valor de $2+3x+5x^2+8x^3$ cuando $x = 0.5$. En este caso se haría:

```
-->horner(poli01, 0.5)
ans = 5.75
```

Las raíces de un polinomio son los valores de x tales que $P_n(x) = 0$. En total, P_n tiene n raíces, que pueden ser reales o complejas. Las raíces se calculan con la función `roots`:

```
-->raices01 = roots(poli01)
raices01 =
-0.6447606
0.0098803 + 0.62261i
0.0098803 - 0.62261i
-->roots(poli02)
ans = 5. 3. 2.
```

Como se ve en el resultado, las raíces complejas siempre aparecen en pares conjugados, si $a+bi$ es una raíz entonces $a-bi$ también lo es.

Los polinomios funcionan como variables simbólicas en operaciones de suma, resta, multiplicación y división. Esto se muestra con los mismos polinomios de los ejemplos anteriores:

```
-->poli01 + poli02
                    2    3
ans = -28 + 34x - 5x + 9x
-->poli01 * poli02
                       2      3      4     5    6
ans =  - 60 - 28x - 77x - 113x + 201x - 75x + 8x
-->poli01 / poli02
ans =
           2    3
2 + 3x + 5x + 8x
------------------
                2   3
- 30 + 31x - 10x + x
```

En el último caso, el resultado queda en la forma de un cociente (un *rational object*). Incluso se puede obtener la derivada simbólica de un polinomio usando `derivat`:

```
-->derivat(poli01)
                    2
ans = 3 + 10x + 24x
```

La lista completa de funciones relacionadas con polinomios está en la sección `Polinomials` de la ayuda del programa.

3.5 Listas y células

Una lista es una secuencia o colección de objetos, no necesariamente del mismo tipo. Las listas se pueden manipular más o menos como las matrices, pero no tienen

definida un álgebra propia porque son meramente estructuras de datos. Hay tres tipos de listas: ordinarias, tipificadas y orientadas a matrices (o matriciales); pero estos dos últimos tipos se tratan en la sección 7.

Una lista ordinaria se construye con la sintaxis

```
nombreLista = list(obj1, obj2, ...,objN)
```

donde `obj` representa cada uno de los objetos que se incluirá. Por ejemplo:

```
-->miLista = list(1, "one", [3 0; 7 8], 2+3*%i);
-->miLista(1)
   ans = 1.
-->miLista(2)
   ans = one
-->miLista(3)
   ans =
     3. 0.
     7. 8.
-->miLista($)
   ans = 2. + 3.i
```

Siendo las listas entidades unidimensionales, el acceso a sus elementos se hace en la misma forma que en un vector fila o un vector columna, como se ve en el ejemplo anterior. Incluso, una lista puede ser un objeto dentro de otra lista.

Sin embargo hay algo curioso: a diferencia de las matrices, en una lista se puede añadir un elemento 0:

```
-->miLista(0) = 3;
-->length(miLista)
ans = 5.
```

Pero esta adición solo corre los elementos hacia atrás. Es decir, de todas formas los índices empiezan en 1

```
-->miLista(0)
!--error 21 Invalid index.
-->miLista(1)
ans = 3.
```

Igual que en las matrices, se emplea $ para añadir un elemento al final:

```
-->miLista($+1) = 888;
-->length(miLista)
ans = 6.
```

Por su parte, la eliminación de un elemento se hace con la orden `null`:

```
-->miLista(5)
ans = 2. + 3.i
-->miLista(5) = null();
-->miLista(5)
ans = 888.
-->length(miLista)
ans = 5.
```

En resumen, los cambios de numeración de una lista debidos a la inserción o remoción de elementos se propagan hacia atrás. Las listas tipificadas y matriciales tienen más posibilidades, pero se tratan en la sección 7, dedicada a la programación orientada a objetos.

Una célula (de aquí en adelante `cell`) es una matriz de variables de tipos diversos. En otras palabras, es una especie de `list` multidimensional. Para crear una variable `cell` se emplea la función

```
cell(i,j,k,...)
```

donde `i, j, k ...` son los números de filas, columnas, etcétera. Por ejemplo, para crear una `cell` vacía de 2x2 se hace:

```
-->ejemCell = cell(2,2)
ejemCell =
!{} {} !
!      !
!{} {} !
```

Cada variable `cell` tiene dos propiedades, `dims` y `entries`, que representan respectivamente sus dimensiones y sus entradas. Para acceder a una entrada hay que usar la sintaxis:

```
nombreCell(i,j).entries
```

Esto se muestra llenando `ejemCell`:

```
-->ejemCell(1,1).entries = [0 3; 2 5];
-->ejemCell(1,2).entries = rand(3,3);
-->ejemCell(2,2).entries = 888;
-->ejemCell
ejemCell =
!{2x2 constant} {3x3 constant} !
! !
!{} 888 !
-->ejemCell(1,1).entries
ans =
0. 3.
```

```
2. 5.
-->ejemCell.dims
ans  =     2  2
```

Pero si una entrada de la `cell` es una matriz, hace falta especificar los índices en `entries` para acceder a los elementos de dicha matriz:

```
-->ejemCell(1,1).entries(2,1)
ans = 2.
```

Un objeto `cell` permite entradas de cualquier tipo, no solamente números y matrices. Eso se muestra aquí añadiendo una cadena de caracteres y un polinomio:

```
-->ejemCell(2,1).entries = "ent21";
-->ejemCell(2,2).entries = poly([0], "x");
-->ejemCell
ejemCell =
!{2x2 constant} {3x3 constant} !
!                              !
!"ent21" x                     !
-->ejemCell(2,2).entries
ans = x
```

De todas formas, la sintaxis se conserva.

GUÍA DE AUTOAPRENDIZAJE

1. Defina una matriz de 5x3 y dos vectores columna, para luego concatenarlos en una matriz de 5x5. Use valores arbitrarios.

2. Del resultado anterior extraiga la columna 1 y la fila 3.

3. Con valores arbitrarios, defina dos matrices, $\mathbf{A}$ y $\mathbf{B}$ de 3x3 y 4x4 respectivamente. Luego concatene una matriz de 4x8 a partir de $\mathbf{A}$, $\mathbf{B}$ y los vectores fila y columna que hagan falta.

4. Repita lo anterior, pero concatenando una matriz de 8x4.

5. Construya una matriz tridiagonal que tenga los valores 5 8 10 6 3 en la diagonal principal, 7 0 4 1 en la diagonal superior y 11 8 9 6 en la diagonal inferior. Construya las diagonales como vectores fila y luego use la función `diag` para construir la matriz tridiagonal (consulte la ayuda).

6. Defina los polinomios $w = 8s + 2s^2 + 2s^4 - s^5$ y $z = 10 + 5s^2 + 2s^3 + 9s^4 + 6s^5$. Luego obtenga $w+z$, $w-z$, wz, y w/z.

7. Defina el polinomio cuyas raíces son 5 -9 3 4. Luego verifique calculándolas.

8. Cree una lista con los elementos que definen el polinomio del punto anterior: el grado, el conjunto de coeficientes y la letra usada para la variable. Use las funciones descritas en la sección `polynomial` de la ayuda para obtener los valores.

9. Repita lo anterior con un `cell`.

Capítulo 4

PROGRAMACIÓN

Al finalizar este capítulo usted deberá ser capaz de:

- Ejecutar un *script*.
- Usar las funciones predefinidas.
- Crear funciones nuevas.
- Depurar errores.

4.1 Funciones y *scripts*

Un *script*[1] es un archivo de texto que contiene una serie de órdenes que se cumplen al ejecutarlo. De otro, lado una función es una entidad que recibe unos valores de entrada, llamados "argumentos", los procesa y devuelve unos valores de salida, llamados "resultados" (Baudin, 2010; Campbell, Chancelier y Nikoukhah, 2006; Mora, 2005). En esta sección se explica la forma de definir y ejecutar un *script*, y luego se muestra cómo usar las funciones predefinidas en el programa. Sin embargo, creación de nuevas funciones se deja para la siguiente sección.

Tanto los *scripts* como las funciones definidas por el usuario se escriben en archivos de texto simple con nombres que deben terminar con las extensiones .sce o .sci. La única regla realmente fija sobre el contenido de los archivos es que el texto después de // es un comentario, lo cual quiere decir que el programa no lo lee. Por lo demás, dentro de un mismo archivo se pueden mezclar varias funciones, o funciones y *scripts*, sin importar la extensión. Aunque los archivos sce y sci se pueden manipular en cualquier editor de texto, es preferible usar el editor propio de Scilab, llamado SciNotes (ver figura 4.1). Este editor formatea con colores y tipos de letra predefinidos las variables, los comentarios y las funciones. Además, permite ejecutar órdenes escritas en él dentro de Scilab.

---

[1] Aunque en algunos libros de informática se traduce la palabra inglesa *script* como "guión" (como el guión de una película, por ejemplo) aquí se ha preferido mantener el término original en inglés.

Como ejemplo se propone un *script* con órdenes para crear y manipular tres matrices:

```
M1 = [2 3 5; 5 67 15; 14 12 11];
M2 = [9 8 1; 5 4 3; 12 3 6] ;
M3 = M1 + M2;
M4 = inv(M3) + (M2*M1)
```

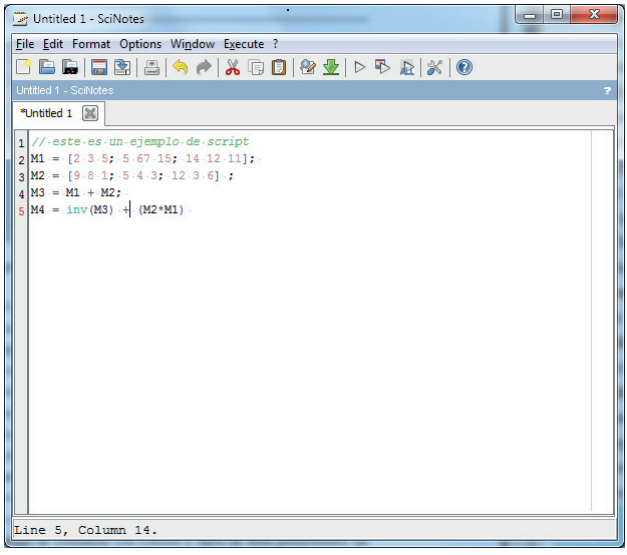


Figura 4.1. SciNotes: el editor de archivos de Scilab

Se guarda en un archivo llamado `ejemploScript.sce`. Este *script* se puede ejecutar de tres formas distintas:

- Desde SciNotes, usando el menú `Execute` o el icono respectivo (ver figura 4.1).

- Desde el *prompt*, usando

  ```
  -->exec("ejemploScript.sce")
  ```

  Aquí se supone que el archivo está en el directorio de trabajo. De lo contrario, hay que añadir la ruta de acceso, algo como:

  ```
  exec("rutaAcceso\ejemploScript.sce")
  ```

- Desde el menú `File/Execute` de la consola.

Las tres formas son equivalentes, el resultado del *script* es el mismo y queda en la memoria como M4. Hay siete modos de mostrar el eco en la ejecución de un *script*. Se pueden mostrar las órdenes, los resultados o ambos usando el parámetro opcional mode:

```
-->exec("ejemploScript.sce", mode)
```

Por ejemplo:

```
-->exec("ejemploScript.sce",-1);
-->exec("ejemploScript.sce", 0);
M4 = 7
2.274861 574.97155 175.93312
72.087416 319.00909 117.95952
122.50249 309.03549 171.19683
-->exec("EjemploScript.sce", 1);
```

En este modo, M4 aparece en la pantalla porque está en la única línea sin ; (es la que produce M4). La lista completa de los modos está en la ayuda.

Como se explicó al comienzo, una función es una especie de "caja negra", que procesa unos argumentos (la entrada) y devuelve unos resultados (la salida). En la forma más sencilla tiene un solo argumento y un solo resultado. Por ejemplo, la función logaritmo:

```
-->log(10)
ans = 2.3025851
```

En el caso general, con M argumentos y N resultados la sintaxis de una función es:

```
[result1, result2, ..., resultN] = nombreFuncion(arg1, arg2,
..., argM)
```

Si hay varios resultados, pero la función se invoca con uno solo, se obtiene el primero de la lista, es decir, result1. Los argumentos se deben escribir en el orden en que están en la ayuda de la función, pero también es posible introducirlos con su nombre en la forma

```
nombreArgumento = valorArgumento
```

También hay funciones definidas sin argumentos, sin resultados, o con varios argumentos y resultados. Incluso con argumentos opcionales y resultados opcionales. Debido a lo confuso que puede ser el manejo de los argumentos es mejor explicarlo usando como ejemplos estas funciones:

- rand: generación, una matriz de números seudoaleatorios.
- gsort: ordena un vector o una matriz de mayor a menor o viceversa.

- `integrate`: cálculo de la integral definida de una función.
- `derivative`: estimación numérica de la derivada de una función.

La función `rand` se puede definir sin argumentos, pero también admite la sintaxis[2] `rand(n,m)`, donde los parámetros opcionales `n` y `m` representan los números de filas y columnas en el resultado:

```
-->rand
ans =
0.2113249
-->rand(2,2)
ans =
0.7560439 0.3303271
0.0002211 0.6653811
```

Luego, si no se especifican valores, se asigna por omisión `m=1` y `n=1`.

La función `gsort` es más complicada. Se puede invocar de tres formas:

```
[B [,k]]=gsort(A),
[B [,k]]=gsort(A,option),
[B [,k]]=gsort(A,option,direction)
```

El argumento `A` es el único obligatorio, es la matriz que hay que ordenar. El resultado `B` es el mismo argumento `A`, pero ordenado. El resultado opcional `k` es un vector con las posiciones en el nuevo orden. Primero se muestra la forma más simple de usar esta función, que por omisión ordena en forma descendente todos los elementos en columnas:

```
-->matAzar = rand(3,4)
 matAzar  =
0.2113249    0.3303271    0.8497452    0.0683740
0.7560439    0.6653811    0.6857310    0.5608486
0.0002211    0.6283918    0.8782165    0.6623569
-->matOrd = gsort(matAzar)
matOrd  =
0.8782165    0.6857310    0.6283918    0.2113249
0.8497452    0.6653811    0.5608486    0.0683740
0.7560439    0.6623569    0.3303271    0.0002211
-->[matOrd, listaPos] = gsort(matAzar)
listaPos  =
9.    8.     6.     1.
7.    5.     11.    10.
2.    12.    4.     3.
```

---

2 De hecho, la función `rand` se puede definir de otras formas para obtener la "semilla" del generador de números aleatorios y las características del generador, consulte la ayuda.

```
matOrd  =
0.8782165    0.6857310    0.6283918    0.2113249
0.8497452    0.6653811    0.5608486    0.0683740
0.7560439    0.6623569    0.3303271    0.0002211
```

(Recuerde que los elementos se numeran por columna, ver sección 3.3). Pero usando los argumentos opcionales `option` y `direction` es posible establecer otras formas de ordenamiento en

```
gsort(A,option,direction)
```

(Como ambos argumentos son cadenas de caracteres hay que incluir las comillas):

- Con `option` igual a ‘‘`g`’’ se ordena todo por columnas; con ‘‘`r`’’ se ordena cada columna por separado; con ‘‘`c`’’, cada fila por separado y con ‘‘`lr`’’ y ‘‘`lc`’’ se hace ordenamiento lexicográfico.
- Si `direction` es igual a ‘‘`i`’’, el orden es ascendente, y si es igual a ‘‘`d`’’, es descendente.

Para el ejemplo, el ordenamiento ascendente en columnas se hace así:

```
-->matAsc = gsort(matAzar, "g", "i")
matAsc =
0.0002211 0.3303271 0.6623569 0.7560439
0.0683740 0.5608486 0.6653811 0.8497452
0.2113249 0.6283918 0.6857310 0.8782165
```

Y también se pueden especificar todos los argumentos con sus nombres:

```
matAsc = gsort(matAzar, option = "g", direction = "i")
```

(Por favor compruebe que el resultado es el mismo) o solamente algunos:

```
-->matFilaAsc = gsort(matAzar, option = "r", "i")
matFilaAsc =
0.0002211 0.3303271 0.6857310 0.0683740
0.2113249 0.6283918 0.8497452 0.5608486
0.7560439 0.6653811 0.8782165 0.6623569
-->matFilaAsc = gsort(matAzar, "c", direction = "i")
matFilaAsc =
0.0683740 0.2113249 0.3303271 0.8497452
0.5608486 0.6653811 0.6857310 0.7560439
0.0002211 0.6283918 0.6623569 0.8782165
```

Pero no se puede especificar por separado el argumento opcional `direction`. Al intentar especificar el orden ascendente solamente con `direction` = ‘‘`i`’’ se obtiene:

```
-->matAsc = gsort(matAzar, direction = "i")
!--error 999  gsort: Wrong value for input argument #2:
''r'', ''c'', ''g'', ''lr'' or ''lc'' expected.
```

El problema es que la especificación del tercer argumento requiere la del segundo. El programa solamente acepta un valor para `direction` si antes se ha dado un valor para `option`.

La función `integrate` tiene el mismo comportamiento, aunque en la ayuda aparece con argumentos opcionales. Su sintaxis es

```
integrate(expr,v,x0,x1 [,atol [,rtol]])
```

donde `expr` es la función a integrar, escrita entre comillas, `v` es la variable de integración y `x0`, `x1` son, respectivamente, los límites superior e inferior de la integral. Los dos argumentos restantes, `atol` y `rtol`, son parámetros numéricos opcionales y por eso aparecen entre `[]`. Primero se muestra aquí el uso de `integrate` para obtener $\int_{0.5}^{1.0} \cos(x)\,dx$ en la forma más simple:

```
-->integrate("cos(x)","x", 0.5, 1.0)
ans = 0.3620454
```

Pero los argumentos opcionales deben ir en el orden `atol, rtol`. Para especificar solamente `atol=1.5E-8`, se escribiría así

```
-->integrate("cos(x)","x", 0.5, 1.0, 1.5E-8)
ans = 0.3620454
```

Pero para especificar `rtol=1E-10` hay darle algún valor a `atol`:

```
-->integrate("cos(x)","x", 0.5, 1.0, 1.5E-8, 1E-10)
ans = 0.3620454
```

Los parámetros `atol` y `rtol` controlan el error máximo admisible (consulte la ayuda).

La sintaxis de `derivative` es un poco distinta:

```
[J [,H]] = derivative(F,x [,h ,order ,H_form ,Q]),
```

El resultado `J` es la derivada y `H` es opcional, para funciones multivariadas `J` es el Jacobiano. El argumento `F` es la función a derivar y `x` es el valor o valores en los que se calcula `J` o `H`. Los argumentos restantes son opcionales. El ejemplo que se usa aquí es la derivada de $\sin(x)$ evaluada en $x = 0.5$, pero `F` debe ser una función definida[3] por el usuario. Primero se emplea la orden `deff` para definir una nueva función llamada "seno":

---

[3] En la siguiente sección se explica en detalle cómo se definen funciones.

```
-->deff("[salida] = seno(x)", "salida = sin(x)")
-->seno(0.5)
ans  =    0.4794255
```

y luego se aplica `derivative` en $x = 0.5$:

```
-->derivative(seno, 0.5)
ans  =    0.8775826
```

En este caso, los argumentos opcionales sí se pueden especificar en desorden. Por ejemplo, para obtener la segunda derivada de seno usando un tamaño de paso $h = 0.01$:

```
-->derivative(seno, 0.5, h=0.01, order=2)
ans = 0.8775679
-->derivative(seno, 0.5, order=2, h=0.01)
ans = 0.8775679
```

Cualquier orden es válido *siempre que se identifiquen los argumentos*. Hay otro detalle de importancia en este ejemplo: una función puede ser argumento de otra, y así se pueden "encapsular" métodos numéricos.

Scilab tiene cientos de funciones y mostrar aquí la lista completa es imposible por razones de espacio. Sin embargo, el sistema de ayuda incluye toda la información de uso necesaria (argumentos, resultados, etcétera). También en la referencia (Rietsch, 2010) se encuentran las funciones agrupadas por tema.

4.2 Funciones definidas por el usuario

Toda función en la memoria de Scilab es una variable más. Por lo tanto, se puede guardar y cargar en forma binaria, como se muestra en la sección 2. Incluso se puede copiar con otro nombre, igual que una variable numérica. Eso permite duplicar funciones, por ejemplo:

```
-->seno=sin;
-->seno(%pi/2)
ans = 1.
```

Pero con funciones que no requieren argumentos, el uso de = para copiar la función se puede confundir con la asignación de un resultado. Por ejemplo, la función `clock` proporciona un vector de valores de fecha y hora, en este caso:

```
-->miReloj = clock;
-->fechaHora = clock()
fechaHora = 2011. 11. 10. 15. 27. 13.999998
```

y un tiempo después se obtiene:

```
-->fechaHora
fechaHora = 2011. 11. 10. 15. 27. 13.999998
-->miReloj
ans = 2011. 11. 10. 15. 29. 12.000002
```

Resulta que `fechaHora` no cambia porque es un vector de números. Pero `miReloj` es una copia de la función `clock`, y da fecha y hora en el momento en que se invoca. En este caso peculiar no es necesario incluir los paréntesis () para invocar la función.

Existen dos maneras de crear nuevas funciones (Baudin, 2010; Campbell, Chancelier y Nikoukhah, 2006; Mora, 2005):

- Escribiendo las órdenes en un archivo de texto. Al "ejecutar" el archivo usando `exec` o su equivalente, la función queda en la memoria en forma binaria. Por eso se puede decir que se compila el archivo.
- Con la función `deff`, invocada en el *prompt* o desde un *script*. Pero solo es recomendable para funciones muy sencillas.

Como ya se había mencionado, la definición de funciones en archivos de texto (preferiblemente identificados con la extensión `.sci`) es flexible:

- No es necesario que el nombre de una función coincida con el del archivo desde el cual se compila.
- En un mismo archivo se pueden escribir varias funciones o *scripts*.
- Los comentarios se inician con dos barras diagonales (//).
- No es necesario terminar las órdenes con `;` de todas formas el usuario solamente verá como salida el resultado de la función.

La sintaxis más simple posible es esta:

```
function resultado = mombreFuncion(argumento)
órdenes
endfunction
```

pero si se tienen `M` argumentos y `N` resultados hay que especificarlos

```
function [res1, res2, ..., resN] =
         nombreFuncion(arg1, arg2, ..., argM)
órdenes
endfunction
```

e incluso pueden definirse funciones sin argumentos o resultados.

Para ilustrar la creación de una función se propone este ejemplo: crear dos funciones, una para evaluar el valor del polinomio $5x^3 - 3.5x^2 + 4.8x + 7.2$ y otra para su derivada, $15x^2 - 7x + 4.8$. Primero abra el editor SciNotes y cree un archivo llamado `ejemplosFun.sci` en el directorio de trabajo. Luego escriba estas dos funciones:

```
function valor = funcion1( x )
// evaluación de un polinomio en un punto x
valor = ( 5 * ( x ^ 3 ) )-( 3.5 * ( x ^ 2 ) )+( 4.8 * x )+7.2
endfunction
function [valor, deriv] = funcion2( x )
// evaluación de un polinomio y su primera derivada
en un punto x
valor = ( 5 * ( x ^ 3 ) ) - ( 3.5 * ( x ^ 2 ) )
+ ( 4.8 * x ) + 7.2
deriv = ( 15 * ( x ^ 2 ) ) - ( 7.0 * x ) + 4.8
endfunction
```

Ejecute el archivo y después invoque las funciones desde el *prompt* de esta forma

```
-->funcion1(4)
ans = 290.4
-->funcion2(4)
ans = 290.4
-->[v, d] = funcion2(4)
d = 216.8
v = 290.4
-->derivative(funcion1, 4)
ans = 216.8
```

Tener las funciones en forma binaria en la memoria no permite ver las órdenes, es decir, su código fuente. Además, los cambios hechos al archivo donde está el código fuente *no* se traducen automáticamente a la representación binaria de la función en la memoria. Después de cambiar el código fuente hay que ejecutar (compilar) de nuevo la función para que los cambios queden en la memoria. Como ejercicio, modifique el ejemplo anterior para que evalúe el polinomio $x^3 - 8x^2 + x + 1$ y su derivada. Además, incluya la capacidad de obtener resultados para múltiples valores agrupados en vectores. Por ejemplo, $x = \begin{bmatrix} 0.5 & 1 & 1.5 & \dots & 10 \end{bmatrix}$. Recuerde que no basta con cambiar las líneas en el archivo .sci, hay que ejecutarlo de nuevo.

En principio, las funciones están en la memoria y el usuario debe proveer los argumentos, pero también existe una forma sencilla de preguntarle valores al usuario, la instrucción

```
variable = input(‘‘mensaje’’).
```

En este ejemplo se muestra cómo usarla para un programa que decide si un número es par o impar:

```
function esPar
    x = input(‘‘Escriba un número entero: ’’)
    if ( modulo(x,2) == 0 ) then
        disp(‘‘Felicitaciones, es par’’)
    else
        disp(‘‘El número es impar’’)
    end
endfunction
```

(se deja como tarea probarla). Esta función es peculiar: no tiene argumentos ni resultados; se introducen y muestran en el *prompt*.

La función `deff` sirve para crear nuevas funciones, incluso dentro de la consola. Cada renglón de código de la nueva función es un argumento de `deff`, es decir que

```
deff(‘‘[res1, res2,...]=nombreFuncion(arg1, arg2,....)’’,
                    matrizRenglones [,opt])
```

crea `nombreFuncion`. El argumento `matrizRenglones` es un vector fila con la forma

```
[’’renglon1’’, ‘‘renglon2’’, ..., ‘‘renglonN’’]
```

y `opt` es un argumento opcional para generar perfiles en la compilación (consultar la ayuda). Esto produce el mismo objeto-función que la ejecución de

```
function [res1, res2,...] = mombreFuncion(arg1, arg2,....)
renglon1
renglon2
...
renglonN
endfunction
```

Para el mismo ejemplo anterior (polinomio $5x^3 - 3.5x^2 + 4.8x + 7.2$ y su derivada) `funcion2` se compila con `deff` de esta forma:

```
-->deff("[valor, deriv] = funcion2(x)", ...
["valor = (5*(x^3)) - (3.5*(x^2)) + (4.8*x) + 7.2", ...
-->"deriv = (15*(x^2)) - (7.0*x) + 4.8"])
-->[v,d] = funcion2(4)
d = 216.8
v = 290.4
```

Al ser una función `deff` también se puede invocar desde un *script*. Esto puede servir para definir funciones en tiempo de ejecución, pero no representa ventaja alguna respecto a la programación directa.

Por último, la función `execstr` ejecuta líneas "sueltas" de código. Por ejemplo:

```
-->execstr("a=5+4")
-->a
a = 9.
```

y se puede usar para propósitos similares a los de `deff`.

4.3 Manejo de la memoria

Al ejecutar un *script* las variables creadas, borradas o sobrescritas en él también se crean, borran o sobrescriben en la memoria. Pero si hay funciones, los argumentos pueden ser variables que están en memoria, dentro de cada función se pueden crear variables y también puede haber funciones invocadas por otras funciones. Esta sección explica las sutilezas de la manipulación de variables cuando hay funciones involucradas.

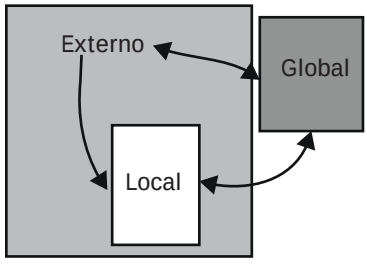


Figura 4.2. Espacios de memoria. La memoria local tiene acceso a la externa, pero no lo contrario. Las variables globales pueden ser leídas o escritas desde donde estén definidas

Para explicar el manejo de memoria, aquí se llamará "locales" a las variables definidas dentro de una función y "externas" a las definidas por fuera de ella (ver figura 4.2). Cada función crea su propio espacio de memoria, independiente del entorno que la está definiendo, con estas consecuencias:

- A no ser que se especifique lo contrario, las variables locales desaparecen al terminar la ejecución de la función.
- Una función tiene acceso a todas las variables del entorno que la define. Dentro de una función se tiene acceso a las variables externas. Estas pueden ser del espacio de memoria principal, el *prompt*, o de otra función. Esto último corresponde al caso de una función invocada dentro de otra.
- Una variable local puede tener el mismo nombre de otra variable externa, y a pesar de eso serán entidades independientes, que pueden tener valores distintos.

Por ejemplo,la siguiente función requiere una variable externa:

```
function valor = funcion3( vin )
// Requiere que la variable vext esté definida
// externamente
valor = vin * vext
endfunction
```

Primero se carga en Scilab y se invoca sin haber definido `vext`,

```
-->funcion3(4)
!--error 4 Undefined variable: vext
at line 3 of function funcion3 called by : funcion3(4)
```

luego se invoca con `vext` ya definido:

```
-->vext = 5;
-->funcion3(4)
ans = 20.
-->vext = 8;
-->funcion3(4)
ans = 32.
```

y, como era de esperarse, el resultado varía de acuerdo al valor de `vext`.

Los argumentos se pasan por valor. De esta forma, si una variable se usa como argumento, no cambia. Por ejemplo, la función siguiente modifica (internamente) el argumento:

```
function valor = funcion4( vin )
vin = vin ^ 2
valor = vin + 4
endfunction
```

Pero esto no afecta a una variable externa que se usa como argumento:

```
-->y = 5;
-->funcion4(y)
ans =
29.
-->y
y = 5
```

De otro lado, si a la función le falta un argumento "obligatorio", lo busca en el entorno de definición. Si hay una variable con el mismo nombre del argumento faltante, toma su valor (ver figura 4.2). Por ejemplo, considere esta función:

```
function salida = funcion5(x, y )
salida = ( x ^ 2 ) + ( y ^ 2 )
endfunction
```

Primero se limpia la memoria con `clear` y luego se recarga la función (recuerde que el nombre de archivo que usted usa puede ser distinto):

```
-->clear
-->exec(‘‘rutaUbicacionArchivo\ejemplosFun.sci’’, -1)
```

Ahora se va, invocar con todos los argumentos, y luego con un argumento faltante:

```
-->funcion5(2,3)
ans = 13.
-->funcion5(2)
!--error 4 Undefined variable: y
at line 2 of function funcion5 called by : funcion5(2)
-->y = 4;
-->funcion5(2)
ans = 20.
-->funcion5(2,4)
ans = 20.
```

Como se ve, da lo mismo tener `y` en la memoria que escribirlo como argumento.

En este punto, muchos lectores se preguntarán por qué las variables externas se llamaron así, en lugar de “globales”, que parece más lógico. La respuesta es que en Scilab existen variables globales, definidas con la especificación `global`. Pero una variable global *no* está automáticamente disponible para cualquier función en cualquier momento. En lugar de eso, residen en un espacio de memoria común. Son accesibles desde las funciones donde hayan sido declaradas y desde el *prompt* de Scilab por igual (ver figura 4.2). Por eso los cambios hechos a una variable global dentro de una función son persistentes. Para explicar eso se propone esta función:

```
function mostrar = ejemploGlobal(suma)
global("x")
x = x + suma
mostrar = x
endfunction
```

que añade el valor del argumento `suma` a una variable global `x`. Pero el resultado solamente está disponible en el espacio principal de memoria si allí también se define `x` como global:

```
-->ejemploGlobal(4)
ans = 4.
-->x
```

```
!--error 4 Undefined variable: x
-->global("x")
-->x
x = 4.
-->ejemploGlobal(6)
ans = 10.
-->x
x = 10.
```

Las modificaciones hechas en el espacio principal de memoria se propagan a la función:

```
-->x = 20
x = 20.
-->ejemploGlobal(4)
ans = 24.
```

Para compartir variables, la especificación global debe existir tanto en la función como en el espacio principal de memoria. Además, una variable global definida dentro de una función puede sobrevivir a una limpieza de memoria:

```
-->clear
-->ejemploGlobal(3)
!--error 4 Undefined variable: ejemploGlobal
-->exec(''rutaArchivo\archivoEejemplos.sci'', -1);
-->ejemploGlobal(3)
ans = 27.
```

Por eso existen órdenes específicas para manejar el espacio de memoria global:

- `clearglobal` permite eliminar variables globales en la misma forma que lo hace `clear`
- `isglobal` verifica si una variable es global

Como comentario final, es el programador quien debe controlar la naturaleza de las variables. Usar variables globales es tentador, y desde el punto de vista lógico es correcto, pero su persistencia puede generar problemas difíciles de diagnosticar.

4.4 Manipulación de entradas y salidas

Muchas funciones amigables con el usuario usan valores por omisión. Es decir, si el usuario no define el valor del argumento, automáticamente se le asigna uno. Para lograr esto, el código debe establecer si el usuario ha definido argumento o no. Para esto se emplea la función

```
exists(''nombreArgumento'')
```

que devuelve un valor 1, si `nombreArgumento` está definido, y 0, de lo contrario. Para ilustrarlo, en este ejemplo se va a crear una función que sirve para calcular la derivada de otra mediante esta aproximación:

$$\frac{f(x+\xi\,|x|)-f(x)}{\xi\,|x|} \tag{4.1}$$

donde $f(x)$ es la función cuya derivada se va a estimar. El "tamaño de paso" ξ tiene un valor por omisión de 10^{-6}. Esto se implementa en el siguiente código, donde el argumento opcional `tolx` representa a ξ:

```
function salida = derivNum(f, x, tolx )
//-Sirve para calcular numéricamente (hacia adelante)
// la primera derivada de una función f en el punto x
//-La tolerancia por omisión es de 1.E-6
if ( exists(''tolx'') == 0 ) then
  tolx = 1.E-6
end,
v1 = f(x)
dx = abs(x) * tolx
xp = x + dx
v2 = f(xp)
salida = ( v2 - v1 ) / dx
endfunction
```

(La estructura de decisión `if` se explica en la siguiente sección, pero fue necesario ponerla aquí por anticipado). Para probar el código se utiliza `funcion4`, definida atrás:

```
-->derivNum(funcion4, 2)
ans = 4.000002
-->derivNum(funcion4, 2, 1.E-6)
ans = 4.000002
-->derivNum(funcion4, 2, 1.E-3)
ans = 4.002
-->derivNum(funcion4, 2, 1.E-8)
ans = 4.
```

Debido al argumento opcional, la función `derivNum` es "flexible": puede invocarse con dos o con tres argumentos.

Las órdenes `varargin` y `varargout` son una forma mucho más elaborada de crear una función con un número variable de argumentos y resultados. Al usar `varargin` en el último lugar de la lista de argumentos, se indica que hay un número variable de estos. Por ejemplo, la función[4]

---

[4] La orden `disp` muestra el resultado en la pantalla. La estructura de control `for` se explica en la siguiente sección.

```
function replicante(varargin)
for i=1:length(varargin)
   disp(varargin(i))
end
endfunction
```

simplemente muestra en la pantalla cualquier cantidad de argumentos con los que se invoque, como se muestra a continuación:

```
-->replicante(4, 8, "nada")
4.
8.
nada
-->replicante(2, 3)
2.
3.
```

De otro lado, se puede crear un número variable de resultados usando `varargout` como salida de la función:

```
varargout = nombreFuncion(argumentos)
```

Por ejemplo, la siguiente función crea tantos resultados como se le indique en el argumento de entrada:

```
function varargout = funcSalidaVar(n)
listaSalida = list()
for i=1:n
   listaSalida(i) = i
end
varargout = listaSalida
endfunction
```

y se puede usar con este tipo de asignación:

```
-->[a, b] = funcSalidaVar(2)
b = 2.
a = 1.
-->[a, b, c] = funcSalidaVar(3)
c = 3.
b = 2.
a = 1.
```

También sirve para obtener resultados adicionales usando la sintaxis

```
[res1, res2, varargout] = nombreFuncion(argumentos).
```

Por último, dentro de una función, `argn` produce el número de entradas y salidas. Los argumentos son las entradas, es decir, el "lado derecho" o `rhs`. Los resultados son el "lado izquierdo" o `lhs`. La sintaxis es `argn(n)`: con `n=0` se obtiene `[lhs rhs]`; con `n=1`, solo `lhs`, y con `n=2` solo `rhs`. En este ejemplo se usa para completar una conjunto de tres argumentos:

```
function salida = verificaArgum(one, deux, trois)
[numRes, numArg] = argn(0)
if ( numArg == 0 ) then
  error("Escriba al menos un argumento")
end
if ( numArg == 1 ) then
  deux = 2
  trois = 3
end
if ( numArg == 2 ) then
  trois = 3
end
salida = one + deux + trois
endfunction
```

de forma que si el único argumento es x, el resultado es $x+2+3$; si se especifican x y y, es $x+y+3$, y en el caso de tener como entradas x, y y z, es $x+y+z$.

4.5 Estructuras de control

Las estructuras de control manipulan el flujo de datos en un *script* o en una función. Las hay de decisión, tales como `if-then-else` y `select-case`, y de repetición: `for`, `while` (Baudin, 2010; Mora, 2005). La más sencilla es `if-then-else`, que se puede describir con la frase

> Si (`if`) se cumple una condición entonces (`then`) haga esto; si no (`else`), haga aquello.

La condición se representa con un valor lógico y se considera cumplida si es verdadero (`%t`, ver sección 3.1). En resumen, la sintaxis del `if` es:

```
if (condicion) then
  //instrucciones si condicion tiene un valor
  //verdadero (%t)
else
    //instrucciones si condicion tiene un
    // valor falso (%f)
end,
```

donde `condicion` es una expresión lógica, y la parte del `else` es opcional. Por ejemplo, esta función:

```
function salida = ejemIf(a, b)
if ( ( a > b ) & ( b <= 1.5 ) ) then
  c = a + b - ( b ^ 2 )
else
    c = a - b
end,
salida = c
endfunction
```

produce estos resultados:

```
-->ejemIf(3, 1)
ans = 3.
-->ejemIf(3, 1.5)
ans = 2.25
-->ejemIf(2, 4)
ans = - 2.
```

La orden `elseif` permite anidar instrucciones `if` unas dentro de otras:

```
if condExt then
  //instrucciones si condExt ==%t
elseif  condInt then
    //instrucciones si condInt ==%t
else
    //instrucciones si condInt ==%f
end,
```

pero da lo mismo escribir un `if` dentro de otro, es decir, "anidados".

La estructura `select` ejecuta distintas instrucciones según el valor que tome una variable. Es muy recomendable para casos complicados. Por ejemplo, para una serie de decisiones que de otra forma requeriría varios `if` anidados. Su sintaxis es:

```
select expresion,
case valor_1 then
    instrucciones_1
case valor_2 then
    instrucciones_2
...
case valor_n then
    instrucciones_n
else
    otras_instrucciones
end
```

`expresion` y `valor_i` deben ser ambos números enteros o cadenas de caracteres. La `instrucción_i` se ejecuta si `expresion == valor_i` es verdadero, es decir,

si la expresión es igual al valor previsto. Pero, en forma opcional, si `expresion` no es igual a ninguno de los valores propuestos se ejecuta `otras_instrucciones`. Cada `then` debe estar en la misma línea de su respectivo `case`. Por ejemplo:

```
function valor = ejemSelect( vin )
select vin
case 1 then
    valor = vin + 1
case 2 then
    valor = (vin ^ 2 ) + 2
else
    valor = 25
end,
endfunction
```

funciona así:

```
-->ejemSelect(1)
ans = 2.
-->ejemSelect(2)
ans = 6.
-->ejemSelect(4)
ans = 25.
```

Nota: se supone que el argumento `vin` es un entero.

La estructura de repetición `for` ejecuta varias veces un conjunto de instrucciones de acuerdo a un vector de índices. Cada `for` debe estar cerrado por un `end`, y las instrucciones están entre ellos dos:

```
for indice = vector
   // aquí van las instrucciones a ejecutar
end,
```

Usualmente, el índice debe avanzar una cantidad fija, de forma que se puede usar el operador : para generar el vector. Por omisión, cuando se usa sintaxis

```
valorInicial:valorFinal
```

el avance es 1 (ver sección 3.3), tal como se muestra aquí:

```
val = 0;
for i=1:7
   val = val + i;
end
```

Esto produce

```
val = 28.
```

Pero también se pueden especificar avances distintos de 1 en la forma

```
valorInicial:avance:valorFinal
```

El avance puede ser incluso negativo, para producir valores "en reversa":

```
val = 100;
for i=8:-1:1
   val = val - i;
end,
```

da

```
val = 64
```

Incluso se pueden usar valores arbitrarios:

```
val=0;
lista = 1:1:100;
for i=[1 10 20 50]
   val = val + lista(i);
end
```

produce

```
val = 81.
```

Advertencia: nunca debe alterarse el valor del índice en las instrucciones que van dentro del `for`.

La estructura `while` se utiliza para repetir unas instrucciones mientras sea cierta una condición:

```
while (condicion)
     //instrucciones a repetir
end,
```

En este ejemplo se usa para producir la sucesión de Fibonacci[5] con esta función:

```
function listnum = fibonacci(n)
//-------------------------------------------//
//-Devuelve un vector con los primeros n      //
// términos de la sucesión de Fibonacci       //
//-------------------------------------------//
if (n<=0) then
  disp(‘‘Entrada incorrecta, n debe ser mayor que 0’’)
  abort
end,
select n
  case 1 then
  listnum = [1]
case 2 then
  listnum = [1 2]
else
  listnum = [1 2]
  i = 2
  while ( i < n )
    i = i + 1
    //listnum se agranda automáticamente
    //cuando i es mayor que su longitud
    listnum(i) = listnum(i-1) + listnum(i-2)
  end,
end,
endfunction
```

que produce estos resultados:

```
-->fibonacci(0)
Entrada incorrecta, n debe ser mayor que 0
-->fibonacci(1)
ans = 1.
-->fibonacci(2)
ans = 1. 2.
-->fibonacci(10)
ans = 1. 2. 3. 5. 8. 13. 21. 34. 55. 89.
```

La estructura `while` sirve para repetir instrucciones en casos en los que no se sabe de antemano cuántas repeticiones serán necesarias para llegar a un resultado. Pero dentro de las instrucciones se debe especificar alguna forma de hacer que `condicion` tome un valor falso (es decir `%f`). De lo contrario, el ciclo de repetición no para nunca y se convierte en un “bucle infinito”.

---

5 En la sucesión de Fibonacci cada número es la suma de sus dos antecesores. Los dos primeros términos son 1 y 2, el tercero es 3=1+2, el cuarto es 5=3+2, etcétera. “Fibonacci” es en realidad el apodo del matemático italiano Leonardo de Pisa (1170-1250).

La instrucción `break` "rompe" un ciclo `for` o `while`, al interrumpir la ejecución "saltando" a la instrucción después del `end` que cierra el `for` o el `while`, como se muestra en la figura 4.3. En el siguiente ejemplo, se usa como salvaguarda para interrumpir un ciclo `while` si se han hecho más de 1500 iteraciones:

```
contador = 0
numMaxCiclos = 1500
while (seguir)
  contador = contador + 1
  //instrucciones varias
  //protección contra un ciclo sin fin
  if ( contador > numMaxCiclos ) then
     break
  end,
end,
```

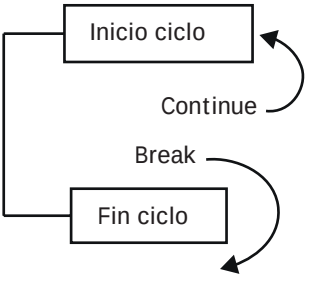


Figura 4.3. Órdenes para romper un ciclo de repetición

Nota: si `break` se define dentro de varios ciclos anidados entonces interrumpe solamente el ciclo más interno. En oposición a `break` existe la orden `continue` que, en lugar de interrumpir el ciclo, regresa la evaluación al punto inicial (ver figura 4.3). Se puede utilizar para evitar ejecutar ciertas porciones del código dentro de la estructura de repetición. Por ejemplo, se tiene un vector con los números 10 20 30 40 50 60 70 80 90 y se quiere sumar 5 a cada uno de ellos, excepto al número 30. Para esto se usa un `for` con una instrucción `continue`:

```
listaValores = [10 20 30 40 50 60 70 80 90];
numVal = length(listaValores);
for i=1:numVal
   if ( i == 3 ) then
     continue;
   else
     listaValores(i) = listaValores(i) + 5;
   end,
end,
disp(listaValores)
```

lo que produce este resultado:

```
15. 25. 30. 45. 55. 65. 75. 85. 95.
```

De hecho, `continue` también se puede utilizar dentro de un `while`. Estas instrucciones:

```
listaValores = [10 20 30 40 50 60 70 80 90];
numVal = length(listaValores);
i = 0
while ( i < numVal )
     i = i + 1
     if ( i == 3 ) then
       continue;
     else
         listaValores(i) = listaValores(i) + 5;
     end,
end,
disp(listaValores)
```

producen el mismo conjunto de valores.

Además de `break` existen otras órdenes de interrupción:

- `return` interrumpe la ejecución de una función. Omite cualquier instrucción después de él, pero no borra los resultados de la función. También se puede usar con la función en pausa (ver la sección 4.6).
- `abort` interrumpe la ejecución de la función. Pero borra las variables de la función, así que los resultados (si los hay) se pierden. También se puede usar con la función en pausa (ver la sección 4.6).
- `quit` interrumpe completamente la ejecución de Scilab y borra la memoria. Se debe usar con precaución extrema. Incluso se puede llamar desde dentro de una función.
- `halt` interrumpe la ejecución hasta que el usuario presiona una tecla de nuevo. Pero no proporciona acceso a las variables locales de la función.
- `pause` se utiliza para depurar la función. Detiene la ejecución, pero no borra las variables (ver la sección 4.6).

Por ejemplo, `abort` se puede usar en la función `fibonacci` para detener la función en el décimo término:

```
....
listnum(i) = listnum(i-1) + listnum(i-2)
if ( i == 10 ) then
abort
end
...
```

En ese caso no hay resultado:

```
-->borrar = fibonacci(10);
-->borrar
!--error 4
Undefined variable: borrar
```

Se hace lo mismo con `return` y se repite la prueba:

```
-->borrar = fibonacci(10);
-->borrar
borrar = 1. 2. 3. 5. 8. 13. 21. 34. 55. 89.
```

En este caso sí aparecen los valores que ya se habían calculado antes del `return`. Se deja como ejercicio para el lector probar el uso de `halt` y `quit`. Recuerde que `quit` termina la ejecución de todo.

4.6 Depuración

La depuración de una función o un *script*, es la identificación y corrección de errores. A veces es posible identificar un error leyendo el código, pero en la mayoría de los casos el error no es evidente. En esta sección se muestra cómo buscar errores deteniendo una función para acceder a las variables locales.

La forma más fácil de pausar la ejecución es la combinación de teclas `ctrl-C` porque el *prompt* deja de estar disponible cuando se ejecuta una función o un *script*. Dentro de una pausa, la orden `abort` cancela la ejecución de la función y `return` la continúa.

Aquí se muestra el resultado de detener la ejecución de la función `fibonacci`:

```
-->borrar=fibonacci(25000);
-1->i
i = 19066.
-1->return
-->borrar
borrar =
column 1 to 11
1. 2. 3. 5. 8. 13. 21. 34. 55. 89. 144.
```

(No se muestra la respuesta completa por cuestión de espacio). En el *prompt*, `-1->` indica que se está dentro del espacio de memoria de `fibonacci`. Eso da acceso a sus variables locales, tales como `i` y `n`. En general, cuando hay una pausa en el *prompt* aparece `-m->`, donde `m` es el (sub)nivel[6] de ejecución. El número 1 en `-1->`

---

6 La orden `whereami` muestra el nivel de pausa. Consulte la ayuda.

indica un subproceso del proceso principal, que es Scilab mismo. Al interrumpir un subproceso dentro de otro (haciendo `ctrl-C` dos veces seguidas) aparece `-2->`, y así sucesivamente.

El inconveniente principal de `ctrl-C` es que no se puede controlar en qué punto del código se pausa la ejecución. Al reproducir el ejemplo es poco probable obtener el mismo valor de i. En cambio, la orden `pause` pausa la ejecución en un punto definido. Por ejemplo, para detener la función `fibonacci` en el término 50 se usaría de esta forma:

```
function listnum = fibonacci(n)
//-------------------------------------------//
//-Devuelve un vector con los primeros n      //
// términos de la sucesión de Fibonacci       //
//-------------------------------------------//
if (n<=0) then
  disp(Éntrada incorrecta, n debe ser mayor que 0')
  abort
end,
select n
  case 1 then
  listnum = [1]
case 2 then
  listnum = [1 2]
else
  listnum = [1 2]
  i = 2
  while ( i < n )
    i = i + 1
    //listnum se agranda automáticamente
    //cuando i es mayor que su longitud
    listnum(i) = listnum(i-1) + listnum(i-2)
    if ( i == 50 ) then
      pause
    end,
  end,
end,
endfunction
```

para producir este resultado:

```
->fibonacci(120)
-1->i
i = 50.
-1->listnum(50)
ans = 2.037D+10
-1->abort
```

(Se usa `abort` para cancelar la ejecución).

Al igual que `return`, la orden `resume` da acceso a las variables locales creadas en una función. En la forma

```
[nom1, nom2, ..., nomN] = resume(var1, var2, ...,varN)
```

detiene la ejecución de la función y devuelve las variables bajo los nombres

```
nom1, nom2, ..., nomN
```

en el espacio de memoria desde el que se ha definido la función, usualmente el *prompt*. Como ejemplo se propone una función que aproxima $\exp(x)$ según la serie:

$$1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots+\frac{x^n}{n!} \tag{4.2}$$

La función suma los primeros n términos de la serie a medida que se generan:

```
function salida = altExp(x, numTerm)
//---------------------------------------------//
//-Aproxima la función exponencial usando los   //
// primeros n términos en la serie de potencias //
//---------------------------------------------//
sumatoria = 1
numerador = 1
denominador = 1
i = 0
while ( i <= numTerm )
    i = i + 1
    numerador = numerador * x
    denominador = denominador * i
    termino = numerador / denominador
    sumatoria = sumatoria + termino
end
salida = sumatoria
endfunction
```

Sugerencia: pruébelo calculando la aproximación de $\exp(0.5)$ con diversos valores de n y comparándolo con `exp(0.5)`. Digamos que por propósitos de verificación se quieren obtener los valores de x^n, $n!$, y $x^n/n!$ cuando $n=6$. En la ejecución normal no se puede

```
-->altExp(0.5, 6)
ans = 1.6487212
-->numerador
!--error 4
Undefined variable: numerador
```

pero esos valores quedan disponibles usando `resume` en la penúltima línea de `altExp`:

```
...
salida = sumatoria
[v1, v2, v3, v4] = resume(i, numerador, denominador,
termino)
endfunction
```

lo que produce este resultado:

```
-->altExp(0.5, 6)
ans = 1.6487212
-->v1
v1 = 7.
-->v2
v2 = 0.0078125
-->v3
v3 = 5040.
-->v4
v4 = 0.0000016
```

Como se muestra, se pueden asignar nombres distintos a los especificados dentro de la función. Para el lector se deja como ejercicio usar `resume` para detener obtención del cuarto término de la sumatoria en `altExp`.

Tal como se ha mostrado en algunos de los ejemplos de este libro, Scilab tiene la capacidad de identificar errores al evaluar funciones. Por ejemplo, la división por cero o la invocación de variables no definidas. Cada tipo de error se identifica con un código. Por ejemplo, el 4 es "Variable no definida"; el 2, "Factor no válido", y así sucesivamente (para obtener la lista completa busque en la ayuda `error_table`). Si el error se produce dentro de una función, la ejecución se cancela y se pierden los resultados intermedios. Pero eso se puede evitar, definiendo la acción a seguir ante la ocurrencia de un tipo de error específico, con la orden `errcatch`. La sintaxis es:

```
errcatch(n [,action] [,option])
```

donde:

- `n` es el código del error. Si `n<0` entonces la acción corresponde a cualquiera de los códigos.
- `action` es un parámetro opcional con dos valores posibles:
  - `action = ''continue''`, continua la ejecución.
  - `action = ''pause''`, detiene la ejecución.
- `option` es un parámetro opcional con dos valores posibles:
  - `option = ''kill''`, regresa al nivel 0 del *prompt* (es decir `-->`).

- `option = ‘‘stop’’`, detiene Scilab del todo.

Las órdenes relacionadas `iserror`, `lasterror`, y `errclear` permiten respectivamente, establecer si ha ocurrido un error obtener el último mensaje de error y "limpiar" la acción de `errcatch` (consulte la ayuda). También existe un mecanismo para controlar los daños que provoca un error: la estructura `try-catch`. Su propósito es hacer que la función siga un curso predefinido si se produce un error al ejecutarla. La sintaxis es:

```
try
//instrucciones que pueden generar un error
catch
//instrucciones que se siguen si aparece un error
end
```

Se intenta (`try`) un conjunto de instrucciones y si se atrapa (`catch`) un error se ejecutan unas instrucciones alternas. Como ejemplo se propone una función que calcula.

$$f(x) = \frac{1}{1-x}. \tag{4.3}$$

Esta función tiene una peculiaridad: si $x = 1$ el resultado se indetermina porque se está intentando una división por 0:

```
-->x = 1; 1 / ( 1 - x )
!--error 27 Division by zero...
```

Sin embargo, también es cierto que

$$\lim_{x \to 1} \left[\frac{1}{1-x} \right] \to \infty, \tag{4.4}$$

así que se puede asignar el valor `%inf` al resultado cuando el argumento es 1 escribiendo la función de esta forma[7].

```
function salida = ejemTryCatch(x)
try
   salida = 1 / ( 1 - x )
catch
     disp("Resultado sin sentido numérico")
     salida =%inf
end
endfunction
```

---

7 Recuerde que el comportamiento por omisión cuando aparece un error numérico se controla con la orden `ieee(modo)`.

Esto siempre asigna un resultado, incluso cuando $x = 1$:

```
-->res = ejemTryCatch(0.5)
res = 2.
-->res = ejemTryCatch(1)
Resultado sin sentido numerico
res = Inf
```

GUÍA DE AUTOAPRENDIZAJE

1. La sección áurea se define como $\left(1+\sqrt{5}\right)/2 = 1.618034\ldots$ Se ha encontrado que la división de los términos sucesivos de la serie de Fibonacci se acerca a ese valor. por ejemplo, $55/34 = 1.6176471\ldots$ Modifique la función `fibonacci` para que proporcione la aproximación a la sección áurea como segundo resultado.

2. Haga una función similar a `altExp` para calcular el logaritmo natural de números entre 0 y 1, basada en la serie:

$$\ln(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}(x-1)^n.$$

   Sugerencia: verifique sus resultados contra los de la función `log`.

3. Las raíces de un polinomio de grado 2 son los x tales que $ax^2 + bx + c = 0$. Para hallar las raíces se propone reescribir esta ecuación como

$$x = -\frac{1}{b}\left(c + ax^2\right)$$

   y resolverla en forma iterativa. Un estimado inicial de x se mejora haciendo $x_{i+1} = -(1/b)\left(c + ax_i^2\right)$ hasta que la diferencia entre x_i y x_{i+1} sea despreciable (i es el índice de la iteración). Implemente esta solución en una función y pruébela con varios conjuntos de valores. Incluya esto:

   a) Un criterio ε para detener la búsqueda cuando $|x_{i+1} - x_i| < \varepsilon$ como un parámetro opcional, con un valor por omisión de 10^{-6}

   b) Una salvaguarda para detener la búsqueda después de mil iteraciones.

4. Escriba una función que calcule la suma

$$1 + \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \cdots + \frac{1}{x^i} + \cdots + \frac{1}{x^n}$$

   usando n como argumento. Incluya lo siguiente:

   a) Una estructura `try-catch` para identificar el error que se produce cuando el divisor es muy pequeño

b) Una salvaguarda para detener la suma y reportar el resultado si se encuentra que $1/x^i$ es prácticamente 0.

Capítulo 5

GRÁFICAS

Al finalizar este capítulo usted deberá ser capaz de:

- Trazar gráficas 2D y 3D a partir de una función matemática o de un conjunto de datos.
- Exportar imágenes de gráficas a otras aplicaciones.
- Modificar una gráfica mediante cuadros de diálogo o código.

5.1 De números a gráficas

Scilab solamente tiene la capacidad de trazar figuras geométricas tales como puntos o líneas a partir de valores generados por el usuario. En lugar de "pedirle" al programa que dibuje la gráfica de una función hay que generar los valores primero y luego trazarla. Por ejemplo, la gráfica de la función

$$f(t) = \exp(-0.5t)\sin(5t), \tag{5.1}$$

en el intervalo $0 \leq t \leq 20$ requiere tres pasos:

- Definir un conjunto de valores de t, es decir, de la "variable independiente" en el intervalo $[0, 20]$
- Generar un conjunto de valores de $f(t)$, en el que cada elemento corresponda a un valor de t.
- Trazar los resultados usando la orden `plot`.

Los pasos se traducen a estas instrucciones:

```
//valores de la variable independiente t
tini = 0;
dlt  = 0.025;
tfin = 20;
lstTiempo = tini:dlt:tfin;
//valores de f(t)
dmpr = 0.5;
mult = 5;
lstAmpl = exp(-dmpr * lstTiempo)  .* sin(mult * lstTiempo);
//trazado del grafico
plot(lstTiempo, lstAmpl)
```

que producen un gráfico uniendo los puntos correspondientes a $t = 0$, $t = 0.025$, hasta $t = 20$, como se muestra en la figura 5.1.

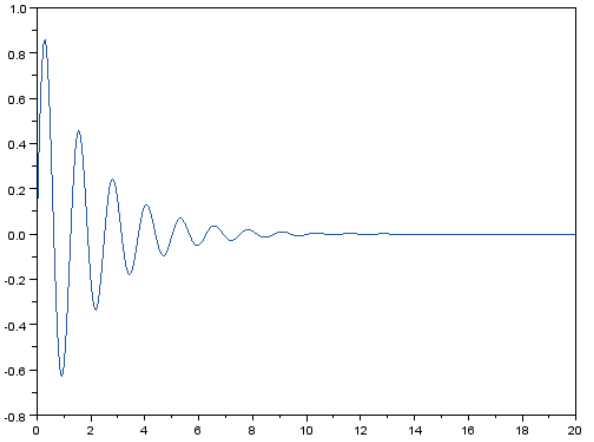


Figura 5.1. Una gráfica simple, $f(t) = \exp(-0.5t)\sin(5t)$

Los gráficos se trazan en ventanas separadas de la consola que toman el título ‘‘`Graphic window number n`’’, donde `n` es el número de la ventana. La numeración de las ventanas arranca en 0, pero también se puede crear o activar la ventana n-ésima usando la orden `scf(n)` (ver tabla 5.1). Toda orden de gráficos se ejecuta en la ventana que esté activa en el momento de invocarla, pero si no existe ninguna, `plot` automáticamente crea y activa una nueva ventana 0 y traza en ella las series de datos que se hayan proporcionado como argumentos.

Tabla 5.1. Algunas órdenes relacionadas con gráficas

Orden	Significado
`clf`	*Clear Figure*. Borra la figura activa
`gca()`	*Get Current Axis*. Captura el control de los ejes
`gce()`	*Get Current Entity*. Captura el control de la entidad gráfica
`gcf()`	*Get Current Figure*. Captura el control de la figura
`get()`	Captura el control de objetos gráficos
`scf(n)`	*Set Current Figure*. Activa una nueva ventana gráfica

El contenido de una ventana de gráficos es *persistente*, es decir, el trazado no borra las gráficas ya presentes. Para el mismo ejemplo anterior se puede usar esta persistencia para añadir *plots* (es decir, trazados) de los límites superior e inferior usando las siguientes órdenes:

```
lstSup = exp(-dmpr * lstTiempo);
lstInf = -exp(-dmpr * lstTiempo);
plot(lstTiempo, lstInf);
plot(lstTiempo, lstSup);
```

Para borrar el contenido de la ventana de gráficos activa se usa la orden `clf`, y con `clf(n)` se borra el contenido de la ventana `n`.

La mayoría de las gráficas se pueden hacer con la orden `plot`, pero esta tiene varias sintaxis posibles. La más sencilla es la que se mostró:

```
plot(valores_x, valores_y)
```

Pero también se pueden trazar en simultánea varios *plots* agrupando los datos del eje y en un solo argumento. Por ejemplo:

```
-->plot(lstTiempo, [lstAmpl; lstSup; lstInf])
```

Los números de filas o columnas deben ser consistentes. Si el gráfico tiene tres series de 801 puntos y los argumentos se organizan en columnas, el de x debe tener 801 filas y el de y 3 columnas y 801 filas. Lo mismo vale si los datos se organizan en filas. Por último, al trazar varios *plots* en simultánea cada uno tendrá un color distinto.

Otra forma de usar `plot` es trazar en simultánea varios *plots* especificando las series de datos por separado, es decir:

```
plot(x1, y1, <lineSpec1>, x2, y2, <lineSpec2>, ...., xN, yN,
<lineSpecN>)
```

donde `xi`, `yi` son las variables del i-ésimo *plot*. `lineSpec` es una cadena de caracteres (opcional) que controla el aspecto y tiene tres componentes: línea, color, y marcador[1], que deben escribirse pegados, es decir, ‘‘`lineaColorMarcador`’’:

- `linea` toma por omisión el valor - para unir los puntos con líneas sólidas. Pero se pueden especificar trazos también.
- En `color`, cada tono corresponde a una letra. `r` corresponde al color rojo (*red* en inglés), g al verde (*green*) y así.
- `marcador` se especifica con un carácter predefinido. Por ejemplo, o para círculo, * para un asterisco. No hay un valor predefinido, si se omite solo se traza la línea.

---

1 La lista completa de valores aceptables está en la entrada `lineSpec` de la ayuda.

Volviendo al ejemplo, al borrar la gráfica y redibujarla con las siguientes órdenes:

```
-->clf;
-->plot(lstTiempo, lstAmpl, "-ko", lstTiempo, lstSup, "--r",
        lstTiempo, lstInf, "--g");
```

se obtiene la misma gráfica de amplitud, pero con líneas rojas y verdes para los límites superior e inferior, y los puntos aparecen marcados con círculos (figura 5.2). La manipulación de las propiedades de ejes, líneas, títulos, etcétera, se hace con un cuadro de diálogo que se invoca en la entrada `Edit/Figure properties` del menú de la ventana del gráfico. Pero también se puede hacer mediante código usando órdenes específicas[2] que se explican en la siguiente sección.

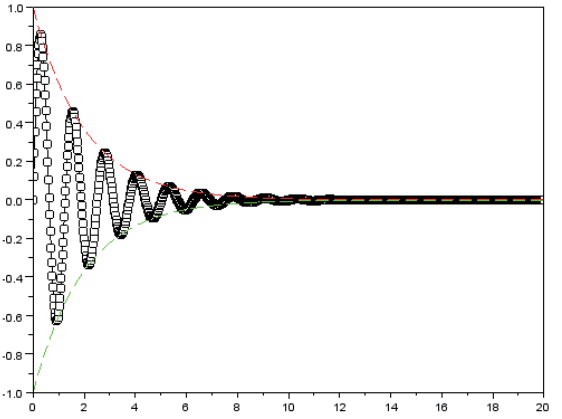


Figura 5.2. Gráfica simple con formato

Para trazar varios *plots* en una sola ventana hay que definir el número de divisiones antes de hacer el trazado, con la orden

```
subplot(m,n,p)
```

donde `m` y `n` son respectivamente los números de filas y columnas en que se divide la ventana y `p` el número de la división donde se hará el siguiente *plot*, se asigna fila por fila. Por ejemplo, para el mismo caso anterior, el código

```
subplot(2,2,1); plot(lstTiempo, lstInf);
subplot(2,2,2); plot(lstTiempo, lstSup);
subplot(2,2,3); plot(lstTiempo, lstAmpl);
subplot(2,2,4); plot(lstTiempo, [lstInf; lstSup; lstAmpl]);
```

produce la figura 5.3 dividiendo la ventana en cuatro secciones.

---

2 Ver la entrada `Graphics/Annotation` en la ayuda.

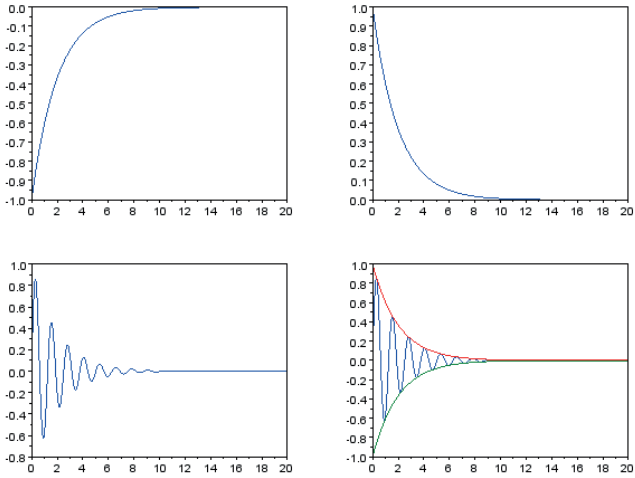


Figura 5.3. Gráfica con cuatro *plots*

El trazado de una gráfica tridimensional implica tener una función o una relación de la forma $z = f(x,y)$, donde x y y son variables independientes. Hay que definir primero los valores de x y de y, y luego para cada *combinación* de valores (x,y) calcular un valor correspondiente de z. La gráfica se traza con la instrucción

```
plot3d(valoresX, valoresY, valoresZ)
```

Como caso de estudio se muestra la creación de una gráfica de la "función del cenicero"

$$z(x,y) = x^6\,[2+\sin(1/x)] + y^6\,[2+\sin(1/y)] \tag{5.2}$$

en el intervalo $-1 \le x \le 1$, $-1 \le y \le 1$. El primer paso es definir la función:

```
function salida = ashtray(x,y)
  parte1 = ( x ^ 6 ) * ( 2.0 + sin( 1 / x ) )
  parte2 = ( y ^ 6 ) * ( 2.0 + sin( 1 / y ) )
  salida = parte1 + parte2
endfunction
```

y los valores[3] de x y y en los que se evaluará la función:

```
lst_x = -1:0.07:1;
lst_y = -1:0.07:1;
nx = length(lst_x);
ny = length(lst_y);
```

---

3 En este caso se ha usado una separación entre valores de 0.07 para evitar el valor $(0,0)$, que produce una indeterminación.

Luego se calculan los valores de z y se traza el resultado:

```
val_z = zeros(nx,ny)
for i=1:nx
   x = lst_x(i)
   for j=1:ny
      y = lst_y(j)
      val_z(i,j) = ashtray(x,y)
   end
end
plot3d(lst_x, lst_y, val_z)
```

Con esto se obtiene la figura 5.4. Cada valor z depende de un par (x,y), por eso el número de elementos en el argumento `valoresZ` es igual al producto de los números de elementos en `valoresX` y `valoresY`. Por ejemplo, si se especifican 100 valores de x y 200 valores de y entonces habrá que generar 20 000 valores para z, uno para cada combinación de x y y.

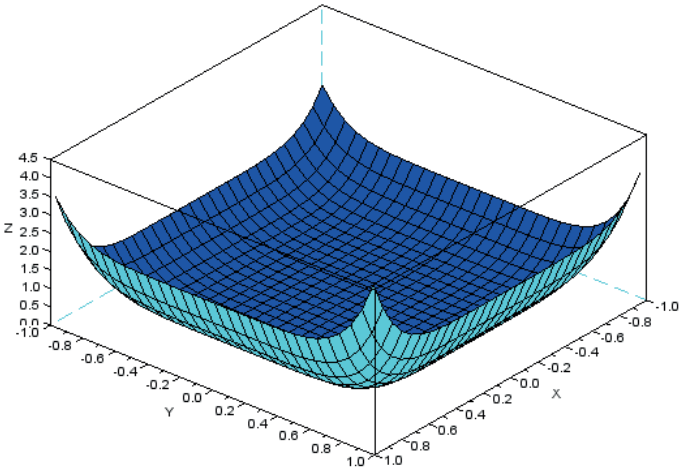


Figura 5.4. Gráfica tridimensional de la función del cenicero.

Las gráficas de línea de contorno son un complemento de las gráficas tridimensionales en coordenadas x-y. En ellas se mantiene $z = f(x,y)$, pero cada línea representa un valor constante de z. Estas gráficas se trazan con la orden `contour`. Los argumentos son las series de valores x, y y z, y un vector con los valores de z en los que se trazarán las líneas de contorno. Para los mismos datos del ejemplo de la función del cenicero la instrucción

```
-->contour(lst_x, lst_y, val_z, [0 0.5 1 1.5 2])
```

produce la figura 5.5, correspondiente a los niveles $z = 0$, $z = 0.5, \ldots, z = 2$.

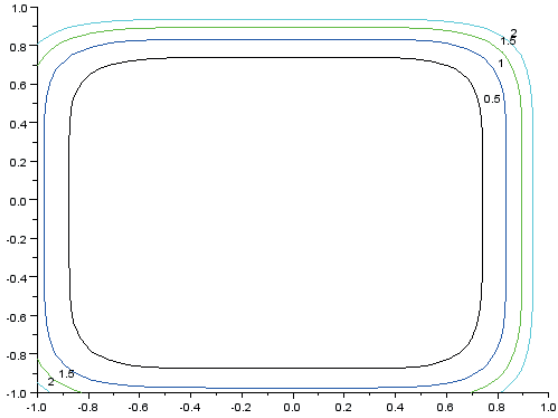


Figura 5.5. Gráfica de contorno de la función del cenicero.

Hasta aquí se ha descrito cómo trazar gráficos, pero falta explicar cómo guardarlos y recuperarlos. Para empezar, cada ventana de gráficos es una entidad independiente que se puede guardar o leer en un archivo (extensión .scg) usando las entradas `File/Load` y `File/Save` del menú de la ventana. Pero esos archivos contienen solamente la información interna de la gráfica. No es posible recuperar de ellos ni las fórmulas ni los valores de las series de datos. Un archivo scg no se puede usar como medio de exportación de las gráficas porque su formato es propio de Scilab. Pero sí se puede copiar la imagen (no la gráfica) al portapapeles del sistema operativo, o exportarla a archivos de formato normalizado, tales como png o bmp usando la ventana de diálogo `File/Export to ...`, o imprimirla (`File/Print`).

5.2 Programación de gráficas

Es fácil manipular cualquier gráfica usando la ventana de diálogo de modificación de propiedades de la figura, e incluso se puede guardar o exportar. Pero hay que dibujarla de nuevo cada vez que se modifiquen las series de datos porque no está enlazada a los datos que la generan. Eso implica que el formato de las gráficas solo se puede mantener si se incluye en el código. Para las líneas eso puede ser sencillo en la orden `plot` y, pero no para ejes, títulos y demás.

Los elementos de una gráfica, ejes, líneas y demás están concebidos como "objetos" que poseen "propiedades". Al trazar una gráfica se crea un conjunto de nuevos

objetos dentro de la memoria, y sus propiedades[4] se "inicializan" con valores por omisión. Por ejemplo, la ventana de diálogo `Figure properties` cambia los límites de los ejes o los colores de las líneas manipulando las propiedades de los objetos que los representan en la memoria, sin que el usuario lo note. Pero modificar o consultar las propiedades de un objeto mediante órdenes implica asignar un manejador, llamado *handler* en la terminología de Scilab, que le da nombre al objeto (ver tabla 5.1). Existen dos formas de hacerlo.

- Con la sintaxis

  ```
  nombreHandler = gxx
  ```

  donde `gxx` depende del tipo de objeto –`gcf` para la figura, `gce` para la última entidad gráfica creada y `gca` para los ejes–.

- Usando la orden

  ```
  nombreHandler=get(''elemGrafico'')
  ```

  En este caso, `elemGrafico` define el objeto que se va a "capturar". Se usa `current_figure` para la figura, `current_axes` para los ejes[5] y `current_entity` para la última entidad gráfica creada.

Continuando con la gráfica de movimiento amortiguado descrita en la sección anterior, el siguiente código le asigna un *handler* llamado `Homer` al objeto figura, `lineasHomer` al objeto líneas (el último en ser creado) y `ejesHomer` al objeto ejes:

```
plot(lstTiempo, [lstAmpl; lstSup; lstInf])
Homer = gcf();
lineasHomer = gce();
ejesHomer = gca();
```

Aunque también se puede escribir de esta forma:

```
Homer = get("current_figure")
ejesHomer = get("current_axes")
lineasHomer = get("current_entity")
```

En ambos casos la asignación es la misma, y `Homer` queda como una "instancia" del tipo de entidad gráfica `figure` con sus propiedades correspondientes. Todo objeto gráfico funciona como una variable cualquiera con el nombre asignado a su *handler*, para verificarlo escriba `Homer` en el *prompt*:

---

4 De hecho, una propiedad puede ser en sí misma un objeto.

5 Los ejes son llamados *axis* o *axes*.

```
-->Homer
Homer =
Handle of type "Figure" with properties:
========================================
children: "Axes"
figure_position = [200,200]
figure_size = [628,541]
axes_size = [610,407]
auto_resize = "on"
...
```

(Se omitió el resto). De la misma forma, `ejesHomer` es una instancia del tipo `Axes` que representa los ejes de una figura

```
-->ejesHomer
ejesHomer =
Handle of type "Axes" with properties:
======================================
parent: Figure
children: "Compound"
visible = "on"
axes_visible = ["on","on","on"]
axes_reverse = ["off","off","off"]
grid = [-1,-1]
grid_position = "background"
x_location = "bottom"
y_location = "left"
title: "Label"
x_label: "Label"
y_label: "Label"
...
```

(Se omitió el resto. Ver también la sección `figure_properties` en el sistema de ayuda). Los nombres de las propiedades se explican por sí mismos: `x_label` es la etiqueta del eje horizontal, `y_label` es la etiqueta del eje vertical, `axes_bounds` representa los límites numéricos de los ejes, etcétera.

Las líneas de una figura son instancias del tipo `polyline`, pero si hay varias trazadas en un mismo *plot* entonces se agrupan como una entidad `Compound`, como se ve en el ejemplo:

```
-->lineasHomer
lineasHomer =
Handle of type "Compound" with properties:
==========================================
parent: Axes
children: ["Polyline";"Polyline";"Polyline"]
visible = "on"
user_data = []
```

Aquí aparece una relación de "paternidad" entre objetos: cada línea es un objeto "hijo" del `Compound`, es decir, una propiedad `children` en la terminología de Scilab. Pero a su vez este `Compound` de líneas tiene un objeto "padre" (es decir, una propiedad `parent`): el objeto eje de la figura (es decir `Axes`). Las líneas se redibujan cada vez que cambian los límites de los ejes, el tipo de escala o cualquier otra propiedad. En resumen, todo cambio en un objeto afecta a sus propiedades `children`, si las hay; y si un objeto tiene una propiedad `parent` dependerá de sus cambios.

Habiendo "capturado" al objeto con un *handler*, el acceso a propiedades específicas se hace con la sintaxis

```
nombreHandler.propiedad,
```

Por ejemplo, con las siguientes órdenes se obtiene el tamaño, medido en pixeles, de `Homer`, que luego se fija en 400x300:

```
-->Homer.figure_size
ans = 628. 594.
-->Homer.figure_size = [400 300]
```

Sin embargo, también existen propiedades que son objetos en sí mismas. Por ejemplo, `x_label` tiene, entre otras, las propiedades `text`, que define el texto que aparece en la etiqueta, y `font_style`, que define el tipo de letra. En esos casos hay que volver a aplicar el sufijo `.propiedad` para tener acceso. Por ejemplo, las siguientes órdenes marcan[6] los ejes horizontal y vertical con las palabras "t" y "amplitud":

```
ejesHomer.x_label.text = "t"
ejesHomer.y_label.text = "amplitud"
```

Para acceder a las propiedades de objetos hijos hay que definir *handlers* para ellos. Si hay varios, se diferencian añadiendo el prefijo `.children(i)`, donde `i` es el lugar en el conjunto de objetos. Para volver al ejemplo, se asigna un *handler* a cada uno de los tres objetos `children` que posee `lineasHomer`:

```
bart = lineasHomer.children(1);
lisa = lineasHomer.children(2);
maggie = lineasHomer.children(3);
```

y luego se cambia el estilo de las líneas y su grosor con los valores de sus propiedades

---

6 Para representar fórmulas matemáticas, las propiedades `text` pueden ser escritas en LaTeX o MathML. De hecho, también se pueden representar resultados en esos formatos usando la instrucción `prettyprint` (Consulte la ayuda).

```
bart.thickness = 2;
bart.line_style = 3;
lisa.thickness = 2;
lisa.line_style = 3;
```

(Ver la entrada `polyline_properties` en el sistema de ayuda). El resultado está en la figura 5.6.

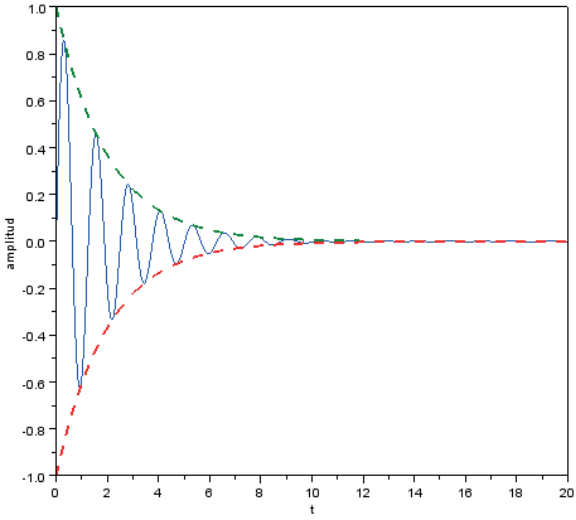


Figura 5.6. Figura *Homer*.

Los objetos gráficos de Scilab tienen una estructura bastante compleja, con docenas de propiedades, y resulta imposible mostrarlas todas aquí. Por eso solo se ha mostrado una pequeña parte de esas estructuras y la forma en que están diseñadas, lu cual deja al lector la tarea de consultar los archivos de ayuda del programa para cualquier otro caso específico que se presente.

GUÍA DE AUTOAPRENDIZAJE

1. En una sola figura grafique las funciones de Bessel $J_1(x)$, $J_2(x)$, $\ldots$, $J_5(x)$ en el intervalo $0 \leq x \leq 10$. Use la función `besselj`.

2. Copie la gráfica anterior en un procesador de texto. Use el portapapeles y la exportación como imagen png.

3. Modifique la figura *Homer* para que el eje t corte el eje vertical en el valor de amplitud 0.

4. Modifique la gráfica de las funciones de Bessel mediante código, para asignarle títulos a los ejes. Cambie la posición del eje x para que corte el eje vertical en el valor 0.

5. Grafique la función $z = xy^2 + x - 10y + 8$ en el intervalo $-5 \leq x \leq 5$, $-5 \leq y \leq 5$ en 3D y en contornos. Presente las dos figuras en una sola ventana.

Capítulo 6

HERRAMIENTAS NUMÉRICAS

Al finalizar este capítulo usted deberá ser capaz de:

- Calcular integrales y derivadas.
- Resolver sistemas de ecuaciones diferenciales.
- Obtener parámetros para representar un conjunto de datos con una función.

6.1 CÁLCULO

La raíz de una función f –es decir, el valor x_r tal que $f(x_r) = 0$– se obtiene con la función[1] `fsolve`:

```
xr = fsolve(x0, fun)
```

donde `x0` es un estimado inicial del valor `xr` de la raíz y `fun` representa a $f(x)$. Por ejemplo, para obtener la raíz de

$$f(x) = e^{-x} - x^2 \tag{6.1}$$

primero se define en la memoria

```
function salida = fun(x)
  salida = exp(-x) - (x^2)
endfunction
```

y luego se tantea un estimado inicial de la raíz, es decir, un valor que lleve a `fun` a producir un resultado razonablemente cercano a 0. Como

```
-->fun(0.5)
ans =
0.3565307
```

se elige `x0=0.5` como valor de arranque en `fsolve`:

---

[1] `fsolve` tiene argumentos opcionales, pero se recomienda usarlos solo si se entiende por completo el método numérico que utiliza (consultar la ayuda).

```
-->fsolve(0.5, fun)
ans =
0.7034674
```

Es decir que $x_r = 0.7034674$ (compruébelo evaluando `fun(0.70347)`).

Sin embargo, `fsolve` también permite obtener raíces de *sistemas* de ecuaciones no lineales. Dado un conjunto de funciones $f_1(\mathbf{X})$, $f_2(\mathbf{X}), \ldots, f_n(\mathbf{X})$, donde $\mathbf{X} = [x_1, x_2, \ldots, x_n]$ se obtiene un conjunto de valores $\mathbf{X}_r = [x_1, x_2, \ldots, x_n]_r$ tales que se cumple:

$$\begin{aligned} f_1(\mathbf{X}_r) &= 0 \\ f_2(\mathbf{X}_r) &= 0 \\ &\vdots \\ f_n(\mathbf{X}_r) &= 0 \end{aligned} \tag{6.2}$$

En este caso, la sintaxis de `fsolve` permanece igual, pero ahora los argumentos y resultados son vectores de n componentes. Por ejemplo, para resolver el sistema de 2x2

$$\begin{aligned} x - 10x^2 + y^2 + 7 &= 0 \\ xy^2 + x^3 - 10y + 5 &= 0 \end{aligned} \tag{6.3}$$

el primer paso es definir una función de $\mathbf{X}$ que produzca un vector de 2 componentes:

```
function salida = fun2var(argX)
  x = argX(1)
  y = argX(2)
  s1 = x - ( 10 * ( x ^ 2 ) ) + ( y ^ 2 ) + 7
  s2 = ( x * ( y ^ 2 ) ) + ( x ^ 3 )  - ( 10 * y ) + 5
  salida = [s1; s2]
endfunction
```

y se introduce como argumento en `fsolve`. En este caso, el estimado inicial es $\mathbf{X} = [0.5, 0.5]'$:

```
-->fsolve([0.5; 0.5], fun2var)
ans =
0.9100017
0.6091212
```

Igual que en el caso de una variable, queda al lector verificar que en efecto la respuesta produce 0 en ambas ecuaciones. Es necesario advertir que tanto en el caso de una sola ecuación como en sistemas de varias ecuaciones puede haber múltiples

soluciones (raíces). El resultado de `fsolve` puede ser una u otra dependiendo del estimado inicial.

La función `integrate` calcula el valor numérico de la integral definida de una función $f(x)$ entre dos valores límite, es decir:

$$\int_a^b f(x)\,dx, \tag{6.4}$$

La sintaxis es

```
integrate(fx, varInt, a, b)
```

donde `fx` es una cadena de caracteres que representa a $f(x)$, `varInt` es una cadena de caracteres que identifica la variable de integración, y `a`, `b` son respectivamente, los límites inferior y superior. Por ejemplo, para obtener el valor de

$$\int_1^2 \left(x^2+3\right) dx \tag{6.5}$$

se usa `integrate` de esta forma:

```
-->integrate("x^2+3", "x", 1, 2)
ans = 5.3333333
```

El segundo argumento, que identifica la variable de integración, es necesario porque si aparecen varios símbolos `integrate`, no se tiene forma de deducir cuál de ellos es la variable. Por ejemplo, al integrar $f(x) = x^2 + a$ con $a = 3$ de la siguiente manera:

```
-->a = 3;
-->integrate("x^2+a", "x", 1, 2)
ans = 5.3333333
```

la variable de integración podría ser `x` o `a`, pero el segundo argumento resuelve esta ambigüedad especificando que es `x`.

De otro lado, para integrar una función computacional (en lugar de una expresión) se emplea la función

```
intg(a, b, fun)
```

que calcula la integral definida de `fun` en el intervalo `[a, b]`. Para el mismo ejemplo ($f(x) = x^2 + 3$), el argumento es esta función:

```
function salida = ejInt(x)
 salida = ( x ^ 2 ) + 3
endfunction
```

Y en el *prompt* el resultado se obtiene así:

```
-->intg(1, 2, ejInt)
ans = 5.3333333
```

por supuesto, después de cargar `ejInt` en la memoria.

La derivada de una función $f(x)$ en un punto x cualquiera se calcula con la función

```
derivative(fun, x)
```

donde `fun` representa a $f(x)$ como una función computacional. Por ejemplo, para obtener el valor de la derivada de $f(x) = x^2 + 3$ en $x = 0.5$ se emplea de nuevo `ejInt`:

```
-->derivative(ejInt, 0.5)
ans = 1.
```

(El resultado coincide con la respuesta exacta, 1). De hecho, `derivative` también sirve para calcular el jacobiano de una función de múltiples variables. En este caso `x` es un vector y `fun` una función multivariada. Hay otras funciones para obtener derivadas: `numdiff` utiliza el método de diferencias finitas y `derivat` sirve para obtener derivadas simbólicas de polinomios.

6.2 Ecuaciones diferenciales ordinarias

La función `ode` integra numéricamente la ecuación diferencial

$$\frac{dy}{dt} = f(t, y) \tag{6.6}$$

sujeta a la condición de frontera

$$y(t_0) = y_0. \tag{6.7}$$

Las soluciones obtenidas con `ode` son "numéricas", series de valores de y correspondientes a los valores de t, que son el argumento de entrada. La sintaxis de `ode` es:

```
y=ode(y0,t0,lista_t,fun)
```

donde:

- `y0` representa el valor inicial de la variable dependiente, es decir, y_0.
- `t0` es el valor inicial de t.

- `lista_t` es la lista de valores de t para los cuales se encontrará la respuesta. El primer elemento es t_0.
- `fun` es la función que representa a $f(t,y)$. Esta función debe tener dos argumentos, t y y, en ese orden.

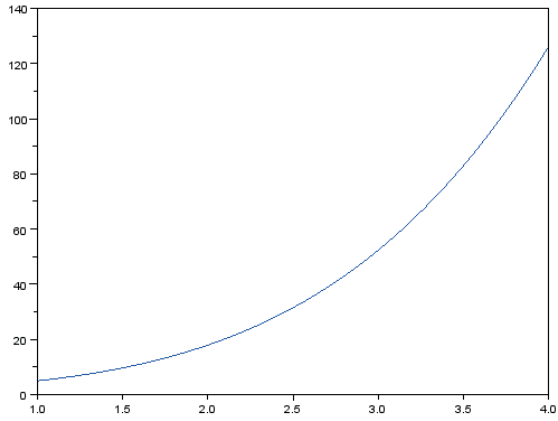


Figura 6.1. Solución y vs. t de la ecuación $dy/dt = (1+2t)\sqrt{y}$ con la función `ode`.

Por ejemplo, se quiere resolver la ecuación diferencial

$$\frac{dy}{dt} = (1+2t)\sqrt{y} \tag{6.8}$$

sujeta a la condición de frontera

$$y(1) = 5,$$

es decir, $t_0 = 1$ y $y_0 = 5$, en el intervalo $1 \leq t \leq 4$.

El primer paso es definir la "función del lado derecho", es decir, $f(t,y)$:

```
function salida = ejOde(t, y)
  salida = ( 1 + ( 2 * t ) ) * sqrt( y )
endfunction
```

y cargarla en memoria. Luego, para producir la solución de la figura 6.1 se definen los valores de t. En este caso, con un $\Delta t = 0.01$, son 1, 1.01, 1.02,..., 3.98, 3.99, y 4.00 y se invoca `ode` con los parámetros y_0 y t_0.

```
lista_t = 1:0.01:4;
lista_y = ode(5,1,lista_t,ej0de);
plot(lista_t, lista_y);
```

El resultado `lista_y` tiene el mismo número de elementos que `lista_t`, y corresponden uno a uno:

```
-->[lista_t' lista_y']
ans =
1. 5.
1.01 5.0675328
1.02 5.1359714
1.03 5.2053247
...
3.97 123.2437
3.98 124.23929
3.99 125.24112
4. 126.24922
```

El número de elementos en la respuesta depende del valor de Δt que se haya tomado. Aunque la elección de Δt es arbitraria, en general la exactitud de la respuesta mejora al reducir su valor.

Aunque por simplicidad se trató aquí primero el caso particular de una sola ecuación ode sirve también para resolver *sistemas* de ecuaciones diferenciales ordinarias, es decir, conjuntos de ecuaciones de la forma

$$\begin{aligned} dy_1/dt &= f_1(t, y_1, y_2, \ldots, y_n) \\ dy_2/dt &= f_2(t, y_1, y_2, \ldots, y_n) \\ &\vdots \\ dy_n/dt &= f_n(t, y_1, y_2, \ldots, y_n) \end{aligned} \tag{6.9}$$

La sintaxis de ode sigue siendo la misma, pero en el caso multivariable:

- y0 representa el *conjunto* de valores iniciales de las variables y_i. Es decir, y0 ahora es un vector $[y_1, y_2, \ldots, y_n]'_{t=t_0}$,
- `fun` es una representación de $f(t, \mathbf{Y})$ donde $\mathbf{Y} = [y_1, y_2, \ldots, y_n]'$.
- El resultado de la función es $[dy_1/dt, dy_2/dt, \ldots, dy_n/dt]'$.
- El resultado de ode es una matriz con n filas.

Por ejemplo, se muestra cómo usar ode para resolver el sistema de ecuaciones

$$\frac{dy_1}{dt} = \frac{y_1 y_2 + 5 y_1 \sin(t)}{y_1 + y_2} \tag{6.10}$$

$$\frac{dy_2}{dt} = \frac{y_2 \cos(t)}{y_1 + 4} \tag{6.11}$$

en el intervalo $0 \leq t \leq 5$, sujeto a la condición inicial $y_1(0) = 2$, $y_2(0) = 3$. La variable dependiente $\mathbf{Y}$ tiene dos componentes ($n = 2$):

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Los argumentos de la función del lado derecho (solo es necesaria una) son t y $\mathbf{Y}$:

```
function salida = ej2Ode(t, lstY)
  y1 = lstY(1)
  y2 = lstY(2)
  elem1 = ( ( y1 * y2 ) + ( 5 * y1 * sin(t) ) )
/ ( y1 + y2 )
  elem2 = ( y2 * cos(t) ) / ( y1 + 4 )
  salida = [elem1; elem2]
endfunction
```

Esta función produce los valores de los lados derechos de las ecuaciones diferenciales, pero agrupados en un solo vector:

```
-->ej2Ode(0, [2;3])
ans =
1.2
0.5
```

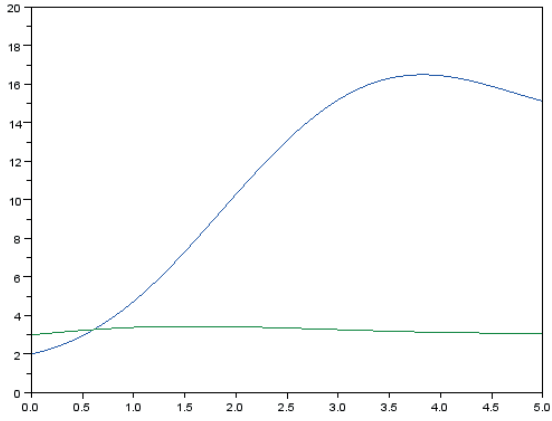


Figura 6.2. Solución de un sistema de dos ecuaciones diferenciales ordinarias con la función ode.

Con $\Delta t = 0.1$ el código

```
Y0 = [2; 3];
lista_t = 0:0.1:5;
listaY = ode(Y0, 0, lista_t, ej2Ode);
plot(lista_t, listaY)
```

produce el resultado de la figura 6.2. La aplicación a sistemas de más de dos ecuaciones es bastante sencilla: simplemente se añaden los elementos que haga falta a la condición inicial y a la función del lado derecho.

6.3 Ajuste de datos a una curva

La forma de ajuste de datos más común es la regresión lineal:

> Dadas dos variables, x, y tales que $y = y(x)$ para un conjunto de datos organizado en pares $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, se calculan a y b tales que la función $y = ax + b$ produzca el menor error posible al predecir los valores de y.

Pero en realidad ese problema es un caso particular del problema general de regresión lineal multivariada, definido así:

> Encontrar el conjunto de valores $a_0, a_1, a_2, \ldots, a_n$, que producen el mejor ajuste en la ecuación
>
> $$y = a_0 x_0 + a_1 x_1 + a_2 x_2 + \cdots + a_m x_m \tag{6.12}$$
>
> a partir de n conjuntos de datos $\left(x_0, x_1, \ldots, x_m; y\right)_1, (x_0, x_1, \ldots, x_m; y)_2, \ldots, (x_0, x_1, \ldots, x_m; y)_n$.

En forma matricial, este problema se escribe como

$$\mathbf{XA} = \mathbf{Y} \tag{6.13}$$

donde

$$\mathbf{X} = \begin{bmatrix} x_{0,1} & x_{1,1} & \cdots & x_{m,1} \\ x_{0,2} & x_{1,2} & \cdots & x_{m,2} \\ \vdots & \vdots & & \vdots \\ x_{0,n} & x_{1,n} & \cdots & x_{m,n} \end{bmatrix},$$

$$\mathbf{A} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix},$$

y

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

En la ecuación 6.13 la incógnita no es $\mathbf{X}$, sino $\mathbf{A}$, y se puede obtener directamente si $m+1=n$. Pero esto es poco probable ya que casi siempre el número n de medidas es mucho mayor que m. En el caso general ($n \neq m$) se busca el conjunto de valores a_i tales que la suma de los residuos, definida con

$$S_r = \sum_{i=1}^{n} [y_i - (a_0x_0 + a_1x_1 + a_2x_2 + \cdots + a_mx_m)_i]^2 \tag{6.14}$$

sea lo más pequeña posible, es decir, el $\mathbf{A}$ tal que la diferencia entre los valores tabulados y calculados de y sea mínima.

Las funciones `reglin` y `lsq` calculan $\mathbf{A}$ a partir de $\mathbf{X}$ y $\mathbf{Y}$ usando el planteamiento de mínimos cuadrados de la ecuación 6.14. Como ejemplo se obtienen los valores a_0, a_1 y a_2 que mejor ajusten los datos de la tabla 6.1 a la expresión

$$y = a_0 + a_1x_1 + a_2x_2. \tag{6.15}$$

Tabla 6.1. Datos para regresión lineal multivariada

y	x_1	x_2
25.5	3.0	19.0
26.0	3.5	17.5
26.8	4.0	11.5
27.2	2.0	3.5
29.2	9.0	8.5
30.0	10.5	4.0
31.5	12.0	1.0

En este caso, $m=2$, $n=7$, y además se está forzando $x_0=1$ para todos los elementos. Por lo tanto,[2]

$$\mathbf{X} = \begin{bmatrix} 1 & 3.0 & 19.0 \\ 1 & 3.5 & 17.5 \\ \vdots & \vdots & \vdots \\ 1 & 12.0 & 1.0 \end{bmatrix},$$

[2] La descripción que aparece en la ayuda de `lsq` es un poco distinta, el sistema se define como `AX=B`, aunque el problema es el mismo.

y

$$\mathbf{Y} = \begin{bmatrix} 25.5 \\ 26.0 \\ \vdots \\ 31.5 \end{bmatrix}.$$

Usando

```
A = lsq(X,Y)
```

se resuelve el problema a partir de los conjuntos de valores de x_1, x_2 y y.

```
lista_y  = [ 25.5 26.0 26.8 27.2 29.2 30.0 31.5];
lista_x0 = ones(1,7);
lista_x1=[3.0 3.5 4.0 2.0 9.0 10.5 12.0];
lista_x2 = [19.0 17.5 11.5 3.5 8.5 4.0 1.0];
X = [lista_x0' lista_x1' lista_x2'];
Y = lista_y';
A = lsq(X, Y);
a0 = A(1);
a1 = A(2);
a2 = A(3);
y_calc = a0 + ( a1 * lista_x1 )  + ( a2 * lista_x2 );
-->[a0 a1 a2]
ans  =
26.880958    0.3764828  - 0.1312608
```

La comparación de los valores de y originales con los valores calculados muestra que el ajuste es bastante bueno:

```
-->[lista_y' y_calc']
ans =
25.5    25.516452
26.     25.901584
26.8    26.87739
27.2    27.174511
29.2    29.153587
30.     30.308985
31.5    31.267491
```

Le queda al lector la tarea de resolver el problema cuando la forma de ajuste de la función es

$$y = a_1x_1 + a_2x_2.$$

eliminando la definición de x_0.

La función `reglin` resuelve el mismo problema de regresión lineal, pero planteado en la forma

$$y = a_1x_1 + a_2x_2 + \cdots + a_mx_m + b,$$

Entonces

```
[aa, b, sig] = reglin(X', Y')
```

produce el conjunto $[a_1, a_2, \ldots, a_m]$ (resultado `aa`), el término b (resultado `b`) y la desviación estándar $\sqrt{S_r/m}$ (resultado `sig`, ver ecuación 6.14). Los argumentos se agrupan por filas. Por eso en la descripción de la sintaxis de `reglin` se escriben transpuestos (`X', Y'`). Para el ejemplo anterior:

```
X = [lista_x1; lista_x2];
Y = lista_y;
[aa, bb, sig] = reglin(X,Y)
```

el resultado es el mismo que se obtuvo con `lsq`:

```
-->[a0 a1 a2]
ans =
26.880958    0.3764828  - 0.1312608
-->[bb aa]
ans =
26.880958    0.3764828  - 0.1312608
```

La regresión de datos es "no lineal" cuando la dependencia de y respecto a las variables x_i no se puede escribir en términos de una sumatoria de términos $a_i x_i$. Es decir que

$$y = f(x_0, x_1, x_2, \ldots, x_m; a_0, a_1, \ldots, a_p) \tag{6.16}$$

no puede ser transformada en la forma de la ecuación 6.12. Por ejemplo, la ecuación

$$y = a_0 + \frac{a_1}{x + a_2}, \tag{6.17}$$

representa una dependencia no lineal de y respecto a x porque no se puede transformar $y(x)$ en una expresión tal como $y = b_0 + b_1 x$ o algo equivalente. Al igual que en el caso lineal, se define la suma de residuos como

$$\begin{aligned} S_r &= \sum_{i=1}^{n} g_i^2 \\ &= \sum_{i=1}^{n} (y_i - f_i)^2, \end{aligned} \tag{6.18}$$

donde

$$g_i = y_i - f_i$$

para $i = 1, 2, \ldots, n$. Así, dado que los valores de x y y están fijos, S_r es función de los p parámetros a_i, de los que depende el ajuste. El problema se reduce a encontrar el conjunto de valores de los a_i que produce el menor valor de S_r. La función

`leastsq` resuelve problemas de regresión no lineal, pero su argumento principal *no* es el conjunto de valores x, y. En lugar de eso requiere una función $R^p \to R^n$, es decir, con p entradas, que representan los a_i, y n salidas, que representan los g_i (las entradas están agrupadas en un solo vector, lo mismo que las salidas). La instrucción

```
[fopt, aopt] = leastsq(funGr, parIni)
```

produce el vector `aopt` de valores de a_i que producen el valor mínimo, `fopt`, de S_r. `funGr` es la función $R^p \to R^n$, y `parIni` es un estimado inicial del vector `aopt`.

Tabla 6.2. Datos para regresión no lineal

x	y
300	-1.9797772
325	-0.9475234
350	-0.0876297
375	0.6387438
400	1.2608445
425	1.800554
450	2.2745971

Por ejemplo, para encontrar el conjunto de valores a_0, a_1, a_2 que produzca el mejor ajuste de los datos de la tabla 6.2 a la ecuación 6.17 se define primero la función $R^p \to R^n$ bajo el nombre `funSr` usando los datos tabulados:

```
lista_x = [300 325 350 375 400 425 450];
lista_y = [-1.9797772 -0.9475234 -0.0876297 0.6387438
        1.2608445 1.800554 2.2745971];
function salida = funSr(lst_a)
  a0 = lst_a(1)
  a1 = lst_a(2)
  a2 = lst_a(3)
  y_calc = a0 + ( a1 ./ ( lista_x + a2 ) )
  salida = lista_y' - y_calc'
endfunction
```

De esta forma, se tienen $p = 3$ parámetros de entrada y $n = 7$ de salida, aunque agrupados cada uno en una sola entidad. Por ejemplo, la evaluación de `funSr` con $a_0 = 10$, $a_1 = -3000$, y $a_2 = -100$ produce siete valores:

```
-->parIni = [10 -3000 -100]';
-->funSr(parIni)
ans =
3.0202228
```

```
2.3858099
1.9123703
1.5478347
1.2608445
1.0313232
0.8460257
```

Con la función `funSr` ya definida ($p = 3$, $n = 7$), la instrucción

```
[fopt, aopt] = leastsq(funSr, parIni)
```

produce el conjunto óptimo `aopt` de parámetros y el valor `fopt` de la suma de residuos:

```
-->parIni = [10 -3000 -100]';
-->[fopt, aopt] = leastsq(funSr, parIni)
aopt =
9.3461365
-2825.0116
-50.564345
fopt =
0.0000029
```

Es decir, el resultado es $a_0 = 9.3461365$, $a_1 = -2825.0116$, y $a_2 = -50.564345$, relativamente cerca a los estimados iniciales. Para verificar el resultado, se calcula y según la ecuación 6.17:

```
-->a0 = aopt(1);
-->a1 = aopt(2);
-->a2 = aopt(3);
-->lyc = a0 + ( a1 ./ ( lista_x + a2 ) );
-->[lista_y' lyc']
ans  =
-1.9797772  -1.9794761
-0.9475234  -0.9477577
-0.0876297  -0.0883165
 0.6387438   0.6386731
 1.2608445   1.2616392
 1.800554    1.801418
 2.2745971   2.2736292
```

(Ver también la figura 6.3). Es necesario advertir que el resultado de `leastsq` depende de la elección del estimado inicial. Por eso es posible obtener distintos conjuntos de parámetros a partir de una misma tabla. Incluso se puede resolver el problema haciendo que `funSr` produzca como único resultado S_r –es decir, $n = 1$– ya que el algoritmo de solución no impone restricciones sobre el número de resultados asociados a la función objetivo.

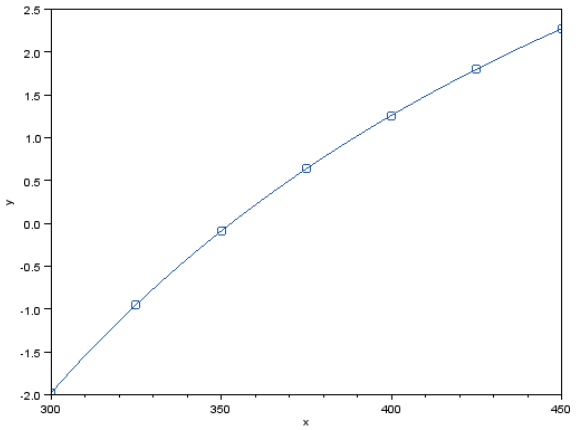


Figura 6.3. Ajuste de datos para la tabla 6.2. Los círculos representan los datos tabulados y la línea, el ajuste con $y = 9.346 - 2825/(x - 50.56)$

Tabla 6.3. Datos para regresión no lineal (guía de autoaprendizaje)

x	y
300	0.1381
320	0.3203
340	0.6613
360	1.2425
380	2.1614
400	3.5284

Para los interesados en la aplicación de otras técnicas de optimización para ajuste no lineal de datos Baudin y Couvert (2010); Urroz (2001) y la sección `Optimization and Simulation` de la ayuda.

GUÍA DE AUTOAPRENDIZAJE

1. Obtenga el valor de la integral

$$\int_0^2 \frac{x}{x^2+5} dx$$

2. Calcule la derivada de

$$f(x) = \frac{x^3}{\sin(x)} - 8x^2$$

en $x = 1.5$

3. Integre las ecuaciones diferenciales

$$\begin{aligned}\frac{dy_1}{dt} &= a_1 y_1 (1 - a_2 y_2)\\ \frac{dy_2}{dt} &= a_3 \frac{y_1 y_3}{1 + y_3} - a_4 y_2\\ \frac{dy_3}{dt} &= 1 - a_5 \left(\frac{y_1 y_2}{1 + y_2}\right)\end{aligned}$$

para obtener los valores de y_1, y_2 y y_3 en el intervalo $0 \leq t \leq 5$, dado que en $t = 0$ los valores de y_i son $y_1 = 1$, $y_2 = y_3 = 0$. Los parámetros son $a_1 = 0.50$, $a_2 = 4.0$, $a_3 = 20$, $a_4 = 30$, y $a_5 = 1.5$.

4. Repita el ejemplo de regresión no lineal (tabla 6.2) definiendo directamente S_r como salida de la función argumento de `leastsq`.

5. Para el conjunto de datos de la tabla 6.3 obtenga los parámetros a_0 y a_1 que den el ajuste óptimo según la función

$$\ln(y) = a_0 + \frac{a_1}{x}.$$

6. Lo mismo que el anterior punto, pero con la función

$$\ln(y) = a_0 + \frac{a_1}{x + a_2}.$$

Capítulo 7

PROGRAMACIÓN ORIENTADA A OBJETOS

Al finalizar este capítulo usted deberá ser capaz de:

- Crear y manipular listas tipificadas.
- Agregar nuevas acciones a los operadores.
- Crear estructuras.

7.1 Listas tipificadas

Aunque el lenguaje de programación de Scilab no permite Programación Orientada a Objetos (*POO*) sí es posible emular[1] algunas construcciones de *POO* usando técnicas que se explican en esta sección. Las listas descritas en la sección 3.5 son los cuasiobjetos más sencillos, contenedores de datos de distintos tipos a los que se accede como si fueran matrices. Las listas tipificadas añaden la posibilidad de acceder a sus campos por nombre propio. Para esto, el primer elemento es un vector fila que contiene (en ese orden) el nombre del tipo de lista y luego los nombres de los campos. Al definir una lista tipificada con la sintaxis

```
variable = tlist(["nombreTipo", "nombreCampo1",
"nombreCampo2", ...], valorCampo1, valorCampo2, ....)
```

se puede acceder a `valorCampoX` como

```
variable.nombreCampoX,
```

o

```
variable(‘‘nombreCampoX’’).
```

Esto se muestra en este ejemplo:

---

[1] Existe una propuesta para incluir todas las características de POO en el lenguaje Scilab (ver http://wiki.scilab.org/Emulate%20Object%20Oriented%20in%20Scilab).

```
-->ejPersona = tlist(["tMedidas", "peso",
"estatura"], 75, 170);
-->ejPersona(1)
ans = !tMedidas peso estatura !
-->ejPersona(2)
ans = 75.
-->ejPersona(3)
ans = 170.
-->ejPersona.peso
ans = 75.
-->ejPersona.estatura
ans = 170.
-->ejPersona("peso")
ans = 75.
-->ejPersona("estatura")
ans = 170.
```

Aquí se asignaron valores a los campos `peso` y `estatura` al crear la lista, pero esa acción es opcional. Una lista tipificada se puede crear sin valores definidos para los campos. La función `fieldnames` sirve para establecer los nombres de los campos de una lista tipificada:

```
-->fieldnames(ejPersona)
ans =
!peso     !
!         !
!estatura !
```

La principal ventaja de una lista tipificada es evitar confusiones cuando se usa como argumento de una función. Para continuar con el ejemplo anterior, digamos que se requiere una función que calcule el índice de masa corporal (imc), definido como el cociente peso/estatura. Como el imc depende de dos parámetros, se podría programar así:

```
function salida = imc(p, e)
   salida = p / e
endfunction
```

Pero en este caso se podrían confundir fácilmente el peso y la estatura al invocar la función. En cambio, con una lista tipificada de la forma:

```
function salida = imc(unaPersona)
   salida = unaPersona.peso/unaPersona.estatura
endfunction
```

no es posible la confusión, porque solamente habrá un argumento

```
-->imc(ejPersona)
ans = 0.4411765
```

Hay otra ventaja: el comportamiento será igual si se añaden nuevos campos a la lista. Para el mismo ejemplo, digamos que se incorporan dos nuevos campos, presiones sanguíneas sistólica y diastólica, a la lista:

```
-->ejPersona(1)($+1) = "presionSist"
-->ejPersona(1)($+1) = "presionDiast"
-->ejPersona.presionSist = 120
-->ejPersona.presionDiast = 80
-->imc(ejPersona)
ans = 0.4411765
```

Como se ve, la función `imc` sigue funcionando igual, a pesar de que la estructura del argumento ha cambiado.

Las listas matriciales se crean con la orden `mlist` y su comportamiento es muy similar al de las listas tipificadas. Para ilustrar esto se repite el ejemplo `ejPersona`

```
->ejPersona = mlist(["tMedidas", "peso",  "estatura",
"presionSist", "presionDiast"], 75, 170, 120, 80);
-->ejPersona.presionDiast
ans = 80.
```

Pero el acceso a los campos *no* se puede hacer numéricamente, no se permite usar `nombreLista(i)`, pero sí el nombre del campo:

```
-->ejPersona(2)
!--error 144  Undefined operation for the  given operands.
check or define function%l_e for overloading.
-->ejPersona.estatura
ans  =       170.
-->ejPersona.estatura = 180
ejPersona  =
ejPersona(1)   !tMedidas  peso  estatura   presionSist
presionDiast  !
ejPersona(2)       75.
ejPersona(3)       180.
ejPersona(4)       120.
ejPersona(5)       80.
```

Curiosamente, el eco muestra los números de los campos en lugar de los nombres –el primer campo es la lista de los nombres–. Para obtener acceso al i-ésimo campo de una variable `mlist` existen las funciones

```
getfield(i, listaMatr)
```

y `setfield(i, valorCampo, listaMatr)`

cuyo funcionamiento se muestra con el mismo ejemplo:

```
-->getfield(1, ejPersona)
ans = !tMedidas peso estatura presionSist  presionDiast !
-->getfield(2, ejPersona)
ans = 75.
-->getfield(3, ejPersona)
ans = 170.
-->setfield(2, 90, ejPersona)
-->ejPersona.peso ans = 90.
-->setfield(3, 155, ejPersona)
-->ejPersona.estatura
ans = 155.
```

Como se ve, las listas matriciales internamente siguen teniendo la forma de una lista.

Las listas tipificadas y matriciales se pueden definir con una función, en lugar de repetir llamadas a `tlist` y `mlist`. En *POO* esto podría denominarse "constructor de las estructuras de datos representadas en las listas". Para continuar con el ejemplo:

```
function nuevaPersona = crearPersona()
    nuevaPersona = mlist(["tMedidas", "peso",  "estatura",
      "presionSist", "presionDiast"])
endfunction
```

sirve para generar dos variables con la estructura `tMedidas`, llamadas `alberto` y `brenda`:

```
-->alberto = crearPersona();
-->alberto.peso = 60;
-->alberto.estatura = 165;
-->alberto.presionSist = 110;
-->alberto.presionDiast = 75;
-->brenda = crearPersona();
-->brenda.peso = 55;
-->brenda.estatura = 161;
-->brenda.presionSist = 130;
-->brenda.presionDiast = 82;
```

Tal como se ve, la función `crearPersona` crea el `mlist` "vacío" y por eso hace falta especificar los valores de los campos.

7.2 Sobrecarga de funciones y operadores

Una función se sobrecarga definiendo para ella un nuevo tipo de comportamiento asociado a un tipo de datos creado por el usuario, con la sintaxis

```
%nombreTipo_nombreFuncion(unaVariableTipo).
```

Siguiendo con el tipo `tMedidas`, al tratar de obtener la raíz cuadrada de una variable de este tipo se obtiene algo así:

```
-->sqrt(alberto)
!--error 246 Function not defined for given
argument type(s),
check arguments or define function%tMedidas_sqrt
for overloading.
```

Pero el comportamiento de `sqrt` se puede definir fácilmente. Digamos que `sqrt` debe producir la raíz cuadrada del peso, pero dividida por mil

```
function salida =%tMedidas_sqrt(unaP)
salida = sqrt(unaP.peso) / 1000
endfunction
```

lo que implica, con un peso de 60:

```
-->sqrt(alberto)
ans = 0.0077460
```

Y de la misma forma se puede definir cualquier comportamiento para esta función.

La "sobrecarga de operadores" es un concepto *POO* que se explica con el operador suma (+). Su significado es obvio para operandos escalares: `A+B` es "la adición de la cantidad `B` a la cantidad `A`" –por ejemplo, 5+4 produce 9 y así–. Sin embargo, Scilab también acepta + en casos en que `A` o `B` sean matrices. Si `A` es un escalar y `B` es una matriz entonces `A+B` es "el resultado de sumar la cantidad `A` a cada elemento de la matriz `B`". Si `A` y `B` son matrices de iguales dimensiones, `A+B` es "la matriz resultante de sumar a cada elemento de `A` el respectivo elemento de `B`". El operador + está sobrecargado porque tiene varios significados de acuerdo al tipo de operandos que se le suministren. En su operación interna, Scilab identifica los tipos de operandos y decide la función a usar a partir de esa información. Pero el comportamiento asociado a datos definidos por el usuario debe ser definido también por el usuario mismo, como se describe a continuación.

Los nombres de las funciones que definen la sobrecarga de operadores empiezan por el carácter especial `%` y deben formarse según estas reglas (Rietsch, 2010):

- Para operadores unarios (es decir, que requieren un solo operando, tal como ~) el nombre debe ser:
  `%tipoOperando_codigoOperador`

- Para operadores binarios (es decir, que requieren dos operadores, tal como * o +) el nombre debe ser:
  `%tipoOperando1_codigoOperador_tipoOperando2.`

donde:

- `tipoOperando` es la identificación del operando (debe tener máximo ocho caracteres). Si es definido por el usuario, es simplemente el nombre mismo. Si es un tipo de variable predefinido, se identifica con código especificado en la entrada `overloading` del sistema de ayuda. Por ejemplo, las matrices de números se identifican con `s`.
- `codigoOperador` es un grupo de letras que identifica el operador que se está sobrecargando. También se encuentra en la entrada `overloading` del sistema de ayuda. Por ejemplo, el código del operador + es la letra `a`.

Para nuestro ejemplo de trabajo se tiene una estructura que incluye varios datos fisiológicos de una persona. Digamos que se define el resultado del operador - (identificado con el código `s`) como la multiplicación por -1 de cada uno de los campos

```
function unaP =%tMedidas_s(unaP)
  unaP.peso = unaP.peso * (-1)
  unaP.estatura = unaP.estatura * (-1)
  unaP.presionSist = unaP.presionSist * (-1)
  unaP.presionDiast = unaP.presionDiast * (-1)
endfunction
```

Al aplicar -, se produce un objeto (lista) con los mismos valores, pero negativos:

```
-->alberto
alberto = 60. 165. 110. 75.
-->ceci = -alberto
ceci =
- 60. - 165. - 110. - 75.
```

De la misma forma, se define el resultado de la operación + entre dos estructuras del tipo `tMedidas`. Es una nueva estructura cuyos campos son cada uno la suma de los campos respectivos de los dos operandos. El código del + es `a` y se incluye en el nombre de la función:

```
function nP =%tMedidas_a_tMedidas(per1, per2)
  nP = crearPersona()
  nP.peso = per1.peso + per2.peso
  nP.estatura = per1.estatura + per2.estatura
  nP.presionSist = per1.presionSist  + per2.presionSist
  nP.presionDiast = per1.presionDiast  + per2.presionDiast
endfunction
```

esto produce:

```
-->alberto
alberto  =
peso: 60
estatura: 165
presionSist: 110
presionDiast: 75
-->brenda
brenda  =
peso: 55
estatura: 160
presionSist: 130
presionDiast: 82
-->ceci = alberto + brenda
ceci  =
peso: 115
estatura: 325
presionSist: 240
presionDiast: 157
```

La sobrecarga también sirve para modificar la forma de acceso a una lista. El nombre de la función de sobrecarga debe incluir el prefijo % y un sufijo que varía según el tipo de sobrecarga. Hay tres formas (`nombreTipo` es el identificador del tipo de lista y `talCual` es el nombre de la lista dentro del código de la función):

- `function%nombreTipo_p(talCual)`

  para el eco en pantalla.

- `function b =%nombreTipo_e(i1,...,in, talCual)`

  para la extracción de elementos, es decir, para obtener un `b` cuando se escribe

  `b = varTipoLista(i1,...,in)`

- `function talCual=%tipob_i_nombreTipo(i1,...,in,b,talCual)`
  para la inserción de elementos, es decir, para adjudicar una variable `b` a un campo escribiendo:

  `varTipoLista(i1,...,in) = b.`

  En esta sintaxis `tipob` es una letra o conjunto de letras que identifica al tipo de variable de `b`. La lista completa está en la entrada `overloading` en la ayuda.

Siguiendo los ejemplos anteriores, digamos que se quiere mostrar el contenido de las listas del tipo `tMedidas` como un vector de la forma

```
[peso estatura presionSist presionDiast]
```

para esto se define la función de sobrecarga

```
function%tMedidas_p(unaP)
  disp([unaP.peso unaP.estatura unaP.presionSist
unaP.presionDiast])
endfunction
```

que produce este resultado:

```
-->alberto
alberto =
60. 165. 110. 75.
```

en lugar de la impresión usual. Lo mismo aplica para cualquier variable del mismo tipo (pruebe con `brenda`). Las siguientes sobrecarga de `tMedidas`:

```
function salida =%tMedidas_e(indx, unaP)
if ( indx <= 2 ) then
  salida = [unaP.peso unaP.estatura]
else
    salida = [unaP.presionSist unaP.presionDiast]
end
endfunction
```

muestra el peso y la estatura o las presiones sistólica y diastólica

```
-->alberto(1)
ans = 60. 165.
-->alberto(3)
ans = 110. 75.
```

Es decir, los valores salen en pares, en lugar de los valores de los campos a partir de sus índices en la lista.

La sobrecarga de la inserción sirve para procesar los valores que se asignan a una lista especificando el número del campo. Aquí se muestra el incremento del peso en diez unidades, la división de la estatura por cien y la suma y resta de uno a las presiones sistólica y diastólica:

```
function unaP =%s_i_tMedidas(indx, entrada, unaP)
select (indx)
  case (1) unaP.peso = entrada + 10
  case (2) unaP.estatura = entrada / 100
  case (3) unaP.presionSist = entrada + 1
  case (4) unaP.presionDiast = entrada - 1
end
endfunction
```

(s es el identificador de matrices, pero también aplica para números porque son el caso particular 1x1). El resultado es:

```
-->alberto
alberto = 60. 165. 110. 75.
-->alberto(1) = 200; alberto(2) = 180;
-->alberto(3) = 120; alberto(4) = 70;
-->alberto
alberto = 210. 1.8 121. 69.
-->alberto.peso = 500
alberto = 500. 1.8 121. 69.
```

Pero, tal como se ve, la función de sobrecarga solo se invoca cuando se hace la asignación con el número del campo.

7.3 Estructuras

La función `struct` construye estructuras que no están basadas en ningún tipo de lista. Esto las hace mucho más cercanas a un objeto en el sentido estricto de la palabra que las listas tipificadas y matriciales, e incluso su sintaxis es más sencilla:

```
struct("nombreCampo1", valorCampo1, "nombreCampo2",
valorCampo2, ...)
```

como se puede ver repitiendo el ejemplo

```
ejPersona = struct("peso", 70, "estatura", 175,
"presionSist", 120, "presionDiast", 80)
```

Una vez creada una estructura, el acceso a los campos es bastante intuitivo:

```
-->ejPersona.peso
ans = 70.
-->ejPersona.estatura
ans = 175.
-->ejPersona.estatura = 85
ejPersona  =
peso: 70
estatura: 85
presionSist: 120
presionDiast: 80
```

pero, a diferencia de las listas tipificadas, no hay un nombre que permita distinguir un tipo de estructura de otra. De hecho, no hace falta `struct` mismo para definir una estructura:

```
-->placa.letras = "ABC";
-->placa.numeros = 123;
-->placa
placa =
letras: "ABC"
numeros: 123
-->typeof(placa)
ans = st
```

En este caso, `st` es la combinación de letras que identifica a una estructura. La función `fieldnames` muestra los nombres de los campos, e `isfield` establece si un nombre se está utilizando para identificar un campo de una variable `struct` (busque la entrada `overloading` en la ayuda).

La sobrecarga de operadores aplica también para estructuras. Por ejemplo, al definir

```
alberto = struct("peso", 60, "estatura", 165,
          "presionSist", 110, "presionDiast",  75);
brenda = struct("peso", 55, "estatura", 160,
          "presionSist", 130, "presionDiast",  82);
```

con la función de sobrecarga

```
function nP =%st_a_st(per1, per2)
  nP.peso = per1.peso + per2.peso
  nP.estatura = per1.estatura + per2.estatura
  nP.presionSist = per1.presionSist  + per2.presionSist
  nP.presionDiast = per1.presionDiast  + per2.presionDiast
endfunction
```

(`st` es el código de structure) se obtiene

```
-->alberto + brenda
ans  =
peso: 115
estatura: 325
presionSist: 240
presionDiast: 157
```

Sin embargo, hay un "pero": como las estructuras no tienen nombre propio que las identifique, la sobrecarga aplicará a cualquier estructura. El operador fallará si se aplica a alguna que no sea del tipo para el cual se definió. Por último, este tipo de sobrecarga también se puede aplicar a los tipos de datos predefinidos. Se puede modificar el comportamiento de los operadores `+ - * /` cuando se usan con números o matrices, pero es dudoso que se obtenga algún beneficio de esto.

GUÍA DE AUTOAPRENDIZAJE

En un experimento con un balde y un cronómetro se recolectarán datos de tiempo en función de volumen de agua. Cada observación incluirá el volumen (en litros) y el tiempo (en minutos y segundos).

1. Escriba una función para crear listas tipificadas que contengan la información de cada observación: volumen, minutos y segundos, y flujo. Los valores de los campos deben ser argumentos opcionales, y si faltan, la lista se creará vacía.

2. Use sobrecarga de operadores para que al asignar los datos en una lista (por el número del campo) automáticamente se calcule el flujo como el cociente entre el volumen y el tiempo en segundos. Recuerde que el número de minutos se debe multiplicar por sesenta y se le suma el número de segundos.

3. Haga que al invocar una lista tipificada se presente la información así: `[flujo volumen tiempo]`, con el tiempo en segundos.

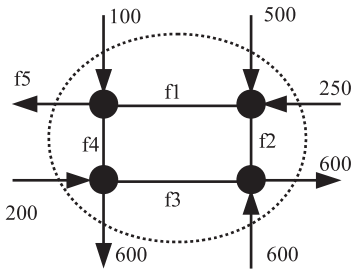
4. Sobrecargue el operador + para que al aplicarse a dos listas tipificadas calcule el flujo promedio.

Apéndice A

Proyectos

Acueducto

El diseño de un sistema de acueducto y alcantarillado tiene en cuenta el flujo, en m^3/h en varios sectores de la ciudad.



En el dibujo están los datos conocidos de flujo en m^3/h en varios tramos. Los círculos representan mezcladores de flujo. En cada elemento, la suma de los flujos que entran debe ser igual a la suma de los flujos que salen. Pero falta asignar los diámetros de las tuberías en los tramos f1 ... f5. Esto depende de los flujos que circulan por ellos. Para calcularlos, plantee un sistema (o sistemas) de ecuaciones lineales.

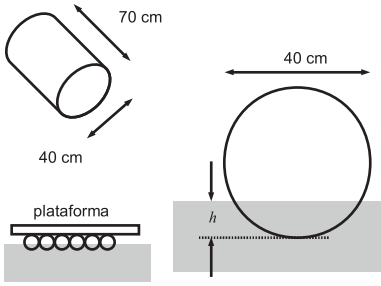
Flotación

Se va a montar una plataforma flotante usando seis barriles sellados como flotadores. Cada barril tiene una masa de 10 kg, con unas dimensiones de 40 cm de ancho por 70 cm de alto. A usted se le pide hacer tres cosas:

- Calcular la masa máxima que puede tener la plataforma sin hundirse.
- Preparar una tabla de valores de nivel sumergido h (en cm) en función de la masa m (en kg) de la plataforma.
- Hacer una gráfica de h en función de m.

Según el principio de Arquímedes, el peso de la plataforma debe ser igual al peso del agua desplazada por los barriles. Esto implica que las dos fuerzas, el peso y la flotación, se igualan. Por lo tanto, $mg = \rho g V_h$, donde V_h es el volumen de agua desplazada y la densidad es $\rho = 1000\,\mathrm{kg/m^3}$. Al fijar m se puede calcular h,

pero primero hay que obtener la fórmula para calcular V_h. Ayuda: para simplificar el problema sume la masa de los barriles a la masa de la plataforma. Repita el problema, pero con esferas en lugar de barriles.



Acciones

Esta tabla muestra el precio diario de la acción de la corporación Acme durante dos semanas:

Día	L	M	M	J	V
Precio ($)	1260	1271	1264	1250	1242

Día	L	M	M	J
Precio ($)	1249	1251	1252	1248

Use esa información para predecir el precio de la acción en la apertura del lunes de la tercera semana usando un modelo polinómico de la forma

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

donde y es el precio y x el tiempo transcurrido desde el lunes de la primera semana. Pruebe distintos valores de n y verifique el ajuste gráficamente con la suma de las diferencias entre y calculado y tabulado elevadas al cuadrado. Nota: la bolsa está cerrada sábados y domingos, y el viernes de la segunda semana fue festivo.

Lotka-Volterra

En una mezcla de las especies A, B, X y Y ocurren estas reacciones químicas:

$$\begin{aligned} \mathrm{A} + \mathrm{X} &\rightarrow 2\mathrm{X} \\ \mathrm{X} + \mathrm{Y} &\rightarrow 2\mathrm{Y} \\ \mathrm{Y} &\rightarrow \mathrm{B} \end{aligned}$$

La variación en el tiempo de las concentraciones se describe con este conjunto de ecuaciones diferenciales:

$$\frac{d\,[\mathrm{A}]}{dt} = -k_1\,[\mathrm{A}]\,[\mathrm{X}]$$
$$\frac{d\,[\mathrm{B}]}{dt} = k_3\,[\mathrm{Y}]$$
$$\frac{d\,[\mathrm{X}]}{dt} = k_1\,[\mathrm{A}]\,[\mathrm{X}] - k_2\,[\mathrm{X}]\,[\mathrm{Y}]$$
$$\frac{d\,[\mathrm{Y}]}{dt} = k_2\,[\mathrm{X}]\,[\mathrm{Y}] - k_3\,[\mathrm{Y}]$$

donde $[\mathrm{i}]$ representa la concentración de la especie i y t el tiempo. Integre esas ecuaciones para $0 \leq t \leq 300$ con estas constantes: $k_1 = 0.1$, $k_2 = 0.12$ y $k_3 = 0.05$ y grafique los resultados de concentración en función del tiempo. Las concentraciones iniciales son $[\mathrm{A}]_0 = 1$, $[\mathrm{X}]_0 = 1$, $[\mathrm{Y}]_0 = 1$. Grafique también $[\mathrm{Y}]$ como función de $[\mathrm{X}]$.

Analice lo que sucede cuando las concentraciones iniciales de X o Y son mayores que las de A repitiendo el procedimiento con otros valores iniciales de concentración.

CONTAMINACIÓN

Para calcular la cantidad de CO_2 emitido por una caldera industrial, se mide en la chimenea la temperatura, la velocidad de los gases y la concentración de CO_2 en % en volumen. La chimenea es un tubo de 50 cm de diámetro. Los datos se tomaron a intervalos de 1 a 2 horas con una presión atmosférica estimada de 0.8 bar:

Tiempo (hora:minuto)	0:52	2:03	2:58	4:10	5:04	7:20	8:05	10:07	13:20	15:07	17:00	18:57	20:01	22:02	23:56
$T\,(^\circ\mathrm{C})$	510	481	457	490	514	540	550	520	501	498	494	480	499	492	497
$v\,(\mathrm{m/s})$	9.1	8.2	6.3	7.0	8.5	9.0	9.3	9.7	9.1	6.5	8.2	8.7	9.0	9.1	9.5
% CO_2	10.2	7.8	9.2	8.8	11.1	12.3	14.1	14.2	13.2	12.9	14.7	16.0	15.1	14.6	14.5

Estime la cantidad de CO_2, en kg, que se ha emitido en ese día integrando la tasa de emisión de dióxido de carbono, es decir:

$$m_{\mathrm{CO_2}} = \int_{t_0}^{t_f} \dot{m}_{\mathrm{CO_2}}\,dt$$

donde $t_0 = 0$ y t_f es el tiempo final de la toma de datos. Hágalo de dos formas para comparar el resultado:

- Sumando las áreas bajo la curva. Para esto suponga que el valor de $\dot{m}$ en cada intervalo de tiempo es constante e igual al dato final. Por ejemplo, el valor de $\dot{m}$ calculado en t =0:52 es válido desde 0 hasta 52 min, el valor en t =2:03 es válido desde 52 hasta 2:03 y así sucesivamente.
- Usando la función `intsplin` (consultar la ayuda).

$\dot{m}$ es proporcional al flujo volumétrico que sale de la chimenea. Este se calcula como

$$\dot{V} = Av$$

donde A es el área transversal del tubo. Dadas las condiciones de operación, se puede aplicar la ecuación del gas ideal

$$PV = nRT$$

donde n es el número de moles y T la temperatura en Kelvin ($T\,(\mathrm{K}) = T\,(^\circ\mathrm{C}) + 273.15$). Con esto se obtiene el flujo molar $\dot{n}$:

$$P\dot{V} = \dot{n}RT$$

(en mol/segundo). Sin embargo, $\dot{n}$ es el flujo total. Por lo tanto falta multiplicarlo por la fracción volumétrica para obtener el flujo de CO_2. Ese resultado se convierte a masa/tiempo usando el peso molar W:

$$\dot{m}_{CO_2} = y_{CO_2}\dot{n}W$$

Las constantes son $R = 8.31451 \times 10^{-5}\mathrm{bar} \cdot \mathrm{m}^3/\mathrm{mol} \cdot \mathrm{K}$ y $W = 44 \times 10^{-3}\mathrm{kg/mol}$.

Apéndice B

Extensiones

Las funciones básicas –tales como logaritmo, seno o coseno– están siempre disponibles en el *prompt* porque se cargan en la memoria cada vez que arranca el programa. Eso mismo se puede hacer con una función definida por el usuario, pero hay que cargarla desde un subdirectorio *diferente* al predefinido para las funciones básicas (Baudin, 2010; Gómez, 1999). Además, hay que ubicar las funciones dentro de "bibliotecas" y manipular un archivo interno. Por lo complicado, el proceso se explica paso a paso para dos funciones hipotéticas llamadas `miFunEsp1` y `miFunEsp2`:

1. El código de las funciones debe escribirse en archivos de extensión `.sci`. Aunque cada archivo puede contener varias funciones, solo la primera estará disponible públicamente. Se recomienda que el nombre de la función principal sea el mismo del archivo para facilitar la identificación. En este ejemplo se tendrían dos archivos, `miFunEsp1.sci` y `miFunEsp2.sci`, en un directorio llamado (digamos) `dirBiblioEsp`.

2. Las funciones se compilan usando la orden

   ```
   genlib(nombreBiblioteca, rutaBiblioteca)
   ```

   donde los argumentos son cadenas de caracteres que representan el nombre y la ruta de acceso. En este caso, la instrucción

   ```
   genlib(''especial'', ''rutaDirectorio\dirBiblioEsp'')
   ```

   genera las funciones compiladas en dos archivos, llamados `miFunEsp1.bin` y `miFunEsp2.bin`, y dos archivos con información adicional, llamados `lib` y `names`, todo en el directorio `dirBiblioEsp`.

3. Las órdenes de carga en el arranque se escriben en el archivo `SCIHOME\.scilab`. La ubicación `SCIHOME` varía de una instalación a otra y el archivo `.scilab` puede existir o no. Aquí se dan instrucciones[1] para localizarlos o crearlos:

   a) En el *prompt* escriba `SCIHOME`. Esta orden devuelve la ruta del directorio de información del usuario. Por ejemplo, en Windows debe ser algo como

   C:\Users\javier\AppData\Roaming\Scilab\scilab-5.4.0

---

[1] Los archivos de arranque `.scilab` y `scilab.ini` se explican en la entrada `startup` de la ayuda.

b) Si dentro de ese directorio no existe un archivo llamado `.scilab` debe crearlo con un editor de texto. Recuerde que `.scilab` es literalmente el nombre del archivo.

c) Dentro del archivo `.scilab` escriba esta línea:

```
nombreBiblioteca = lib(‘‘rutaDirectorio
\dirBiblioEsp’’)
```

Como el programa lee `.scilab` como un *script* cada vez que arranca la orden `lib` cargará automáticamente la biblioteca. Incluso `lib` también se puede usar desde el *prompt* para cargar librerías y probarlas.

Después de haber hecho esto, las funciones `miFunEsp1` y `miFunEsp2` deben estar disponibles en el *prompt* en el siguiente arranque.

Es posible incorporar funciones escritas, los lenguajes de programación FORTRAN o C, llamadas "primitivas", usando un compilador externo (Gómez, 1999). El primer paso es compilar el código de la función como un archivo-objeto, usualmente de extensión `.o`. Después el archivo-objeto se enlaza a Scilab usando la orden

```
link(‘‘nombreFuncion.o’’, ‘‘nombreFuncion’’)
```

Con esto ya se tendrá disponible `nombreFuncion` en el *prompt*. Por omisión, la orden `link` procesa objetos programados en el lenguaje FORTRAN, pero se pueden usar archivos-objeto creados con un compilador C mediante el argumento opcional `flag` (consulte la ayuda).

Es necesario aclarar que el propósito de la compilación externa es mejorar la velocidad de ejecución evitando la interpretación de código. Cuando se aplica, permite por ejemplo usar funciones externas como argumentos en rutinas complejas de optimización o búsqueda de raíces. No hace falta usarla en casos ordinarios en las que la velocidad de ejecución no es clave.

En el sitio web del consorcio Scilab[2] están disponibles módulos que agrupan conjuntos de funciones adicionales sobre temas específicos, tales como estadística, procesamiento de imágenes y optimización, entre otros. El administrador de módulos ATOMS (ver figura B.1), disponible en el menú `Applications` de la consola, muestra los módulos organizados por categorías y los descarga e incorpora a Scilab. El botón `install` instala un módulo ya seleccionado, incluido el sistema de ayuda. Los botones `remove` y `update` sirven respectivamente para quitar o actualizar un módulo ya instalado. También hay funciones para manipular módulos mediante código. Incluso, un usuario puede crear un módulo y enviarlo para aprobación al consorcio Scilab. Pero las instrucciones disponibles para programar y probar un módulo localmente son bastante escasas, así que se ha preferido no abordar el tema en este libro. Por último, la búsqueda o instalación de módulos pueden ser lentas porque ATOMS descarga la lista completa del servidor.

---

2 Es posible crear repositorios de módulos en servidores distintos a los del consorcio.

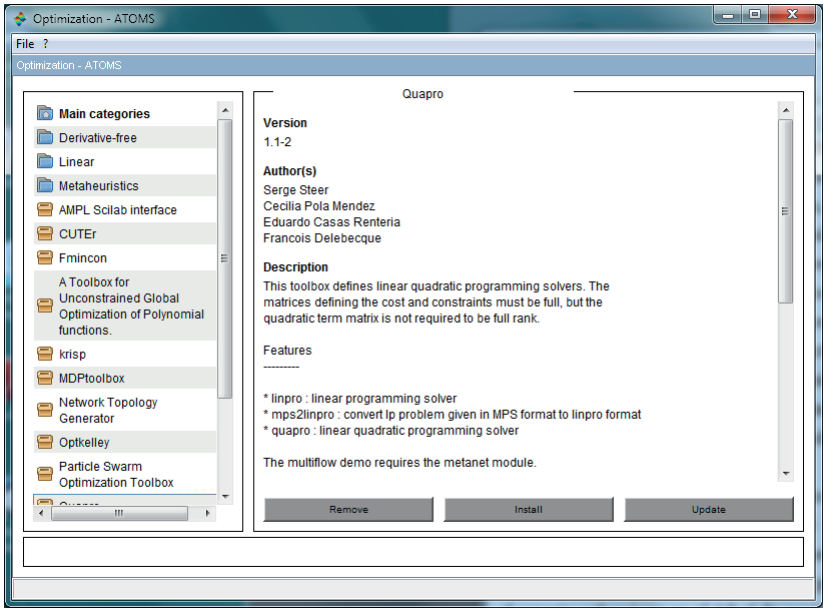


Figura B.1. Administrador de módulos ATOMS

La "inmigración" de usuarios con experiencia previa en Matlab es fácil porque los lenguajes de programación de Matlab y Scilab son muy parecidos, e incluso hay una guía especializada (Rietsch, 2010). En algunos casos afortunados es posible ejecutar un código creado originalmente para Matlab sin modificaciones. Pero hay un traductor incorporado Matlab→Scilab disponible en la entrada `Applications/Matlab to Scilab translator` del menú de la consola.

Esta aplicación utiliza un conjunto de "funciones de compatibilidad" para emular el comportamiento numérico de Matlab (ver la entrada `Compatibility Functions` en la ayuda). Aun así, es recomendable verificar siempre los resultados obtenidos con el código traducido.

Apéndice C

INTERFACES

Cada elemento del menú corresponde a una orden, y por lo tanto es posible enlazar órdenes a nuevas entradas del menú de la consola. Las órdenes pueden ser o no definidas por el usuario y hay varias instrucciones para manipular los menús en la tabla C.1.

Tabla C.1. Órdenes para manipular menús

Orden	Significado
`addmenu`	define un nuevo menú
`setmenu`	activa un menú
`unsetmenu`	desactiva un menú
`delmenu`	borra un menú ya definido

Para añadir un menú a la ventana se utiliza

```
addmenu(nomMenu [,entrMenu] [,accion])
```

donde:

- `nomMenu` es el texto que identifica al menú en la parte superior de la ventana. Puede tener una acción asociada a él.
- `entrMenu` es un vector con los nombres de las entradas.
- `accion` es una lista de dos elementos definida como

  `list(flag, ‘‘nomManejador’’)`.
- `flag` identifica el tipo de acciones. Es 0 para órdenes de Scilab, 1 para funciones compiladas externamente y 2 para funciones interpretadas. El valor por omisión es 0.
- `nomManejador` define las acciones que ocurren al elegir una entrada

Para generar un menú en una ventana específica se añade un argumento inicial

```
addmenu(numVen, nomBoton [,entrMenu] [,accion])
```

donde `numVen` (un entero) es el número de la ventana.

El caso más simple es un “botón”, es decir, un menú sin entradas adicionales. Por ejemplo dos botones para detener y abortar la ejecución de un programa se crean así:

```
addmenu("Pausa", list(0,"pause"));
addmenu("Abortar", list(0,"abort"));
```

No hay entradas, por lo tanto se omite `entrMenu`. Las acciones están definidas por órdenes de Scilab, así que `flag==0`. Los botones aparecen en la parte superior de la consola (ver figura C.1).

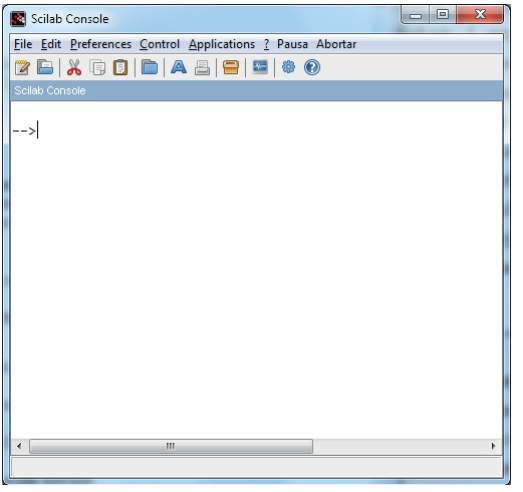


Figura C.1. Ventana con dos botones de menú definidos por el usuario

Para eliminar un menú se utiliza

```
delmenu(nomBoton)
```

o bien

```
delmenu(numVen, nomBoton)
```

Puede probarlo borrando los menús (botones) del ejemplo anterior.

Ahora se muestra un ejemplo más complicado, un menú llamado `Prueba` con tres entradas: la primera sirve para mostrar un mensaje, "Hola mundo"; la segunda, para mostrar la fecha y la hora, y la tercera estará deshabilitada. El primer paso es definir una función que sirve como manejador:

```
function manejaItems(numItem)
  select(numItem)
  case(1) then
    disp("Hola mundo")
  case(2) then
```

```
    disp(clock())
  end
endfunction
```

y luego se construye el menú:

```
addmenu("Prueba", ["Hola","Reloj","Nada"],
list(2, "manejaItems")) unsetmenu("Prueba",3)
```

En este caso, `flag==2` porque el manejador es una función, y la instrucción `unsetmenu` deshabilita el tercer elemento, para el que no hay acción definida (ver figura C.2).

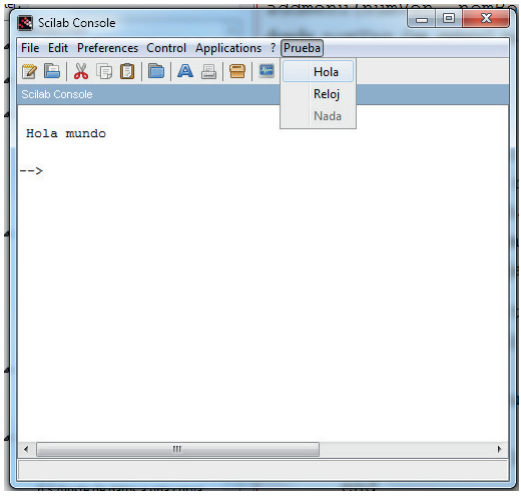


Figura C.2. Ventana con entrada de menú definida por el usuario

Crear una ventana de diálogo requiere dibujarla, definir sus elementos de control y, por último, especificar el comportamiento de la interfaz. La orden `figure` dibuja la figura. Por ejemplo, la interfaz para calcular las raíces x_1 y x_2 de la ecuación $ax^2+bx+c=0$ en la figura C.3 se crea con esta instrucción:

```
figEvalPol = figure("position",  [50 50 400 400], ...
             "figure_name",  "Ecuación cuadrática", ...
             "Tag", "fEP");
```

Las propiedades se definen en los argumentos de `figure` siguiendo este orden

```
‘‘nombrePropiedad’’, valorPropiedad
```

pero las asignaciones pueden ir en cualquier secuencia. `figure_name` es el título de la ventana y `Tag` es un identificador. `position` controla la posición y el tamaño

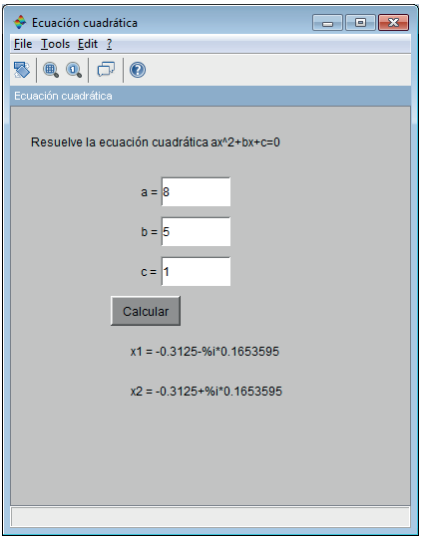


Figura C.3. Interfaz para calcular las raíces de la ecuación $ax^2+bx+c=0$

de la ventana. Se asigna como `[x y ancho alto]`, medidos en pixeles desde la esquina superior izquierda.

Los elementos de control –tales como botones, cuadros de texto, editores de texto, etcétera[1]– se crean con la orden `uicontrol` de esta forma:

```
nombreControl = uicontrol(parent,
     ‘‘nombrePropiedad1’’, valorPropiedad1, ...
     ‘‘nombrePropiedad2’’, valorPropiedad2, ...)
```

El argumento `parent` es la figura de la cual depende el elemento que se está creando. En el ejemplo, los cuadros de texto y de edición y el botón “Calcular” dependen de `figEvalPol`:

```
textExpl = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "Resuelve la ecuación  cuadrática", ...
        "fontsize", 12, ...
        "position", [20 340 200 50]);
textECuad = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "ax^2+bx+c=0", ...
```

---

1 Hay nueve tipos de elementos de control. Se describen en la entrada `uicontrol` de la ayuda.

```
        "fontsize", 12, ...
        "position", [200 340 200 50]);
text_a = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "a = ", ...
        "position", [130 300 30 30], ...
        "fontsize", 12);
edit_a = uicontrol(figEvalPol, ...
        "style", "edit", ...
        "backgroundcolor", [1 1 1], ...
        "position", [150 300 70 30], ...
        "fontsize", 12);
text_b = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "b = ", ...
        "position", [130 260 30 30], ...
        "fontsize", 12);
edit_b = uicontrol(figEvalPol, ...
        "style", "edit", ...
        "backgroundcolor", [1 1 1], ...
        "position", [150 260 70 30], ...
        "fontsize", 12);
text_c = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "c = ", ...
        "position", [130 220 30 30], ...
        "fontsize", 12);
edit_c = uicontrol(figEvalPol, ...
        "style", "edit", ...
        "backgroundcolor", [1 1 1], ...
        "position", [150 220 70 30], ...
        "fontsize", 12);
btCalc = uicontrol(figEvalPol, ...
        "style", "pushbutton", ...
        "string", "Calcular", ...
        "position", [100 180 70 30], ...
        "callback", "calcRaices();", ...
        "fontsize", 12);
text_x1 = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "x1 = ", ...
        "position", [120 140 250 30], ...
        "fontsize", 12);
text_x2 = uicontrol(figEvalPol, ...
        "style", "text", ...
        "string", "x2 = ", ...
        "position", [120 100 250 30], ...
        "fontsize", 12);
```

La propiedad `style` define el tipo de elemento; `string`, las cadenas de texto y `fontsize`, el tamaño de la fuente. `callback` se explica a continuación.

El comportamiento de la interfaz depende de la reacción de los elementos de control a las acciones del usuario. Esta se define en la propiedad `callback`, cuyo valor es una cadena de caracteres con la instrucción que se ejecuta al activar el elemento. `callback` es opcional: si un elemento carece de ella, simplemente no hace nada y su función puede ser solo decorativa o informativa. En el ejemplo, solo el botón "Calcular" tiene `callback`, que invoca esta función:

```
function calcRaices()
  //lectura de valores
  a = eval(get(edit_a, "string"))
  b = eval(get(edit_b, "string"))
  c = eval(get(edit_c, "string"))
  //cálculo de las raíces
  aux = -b / ( 2 * a )
  raiz = sqrt( ( aux ^ 2 ) - ( c / a ) )
  x1 = aux - raiz
  x2 = aux + raiz
  //se muestran las raíces en el prompt
  disp("Raices")
  disp(x1)
  disp(x2)
  //se muestran las raíces en la ventana
  //hay que hacer la conversión de número a texto
  set(text_x1, "string", "x1 = "+string(x1))
  set(text_x2, "string", "x2 = "+string(x2))
endfunction
```

`calcRaices` contiene tres acciones:

- Leer a, b y c desde los elementos de edición. Las cadenas de caracteres en la propiedad `string` se obtienen con la función `get`. Se convierten a valores numéricos con `eval`.
- Calcular las dos raíces usando a, b, y c.
- Mostrar el resultado en el *prompt* y la interfaz misma. La manipulación directa de la propiedad `string` no funciona, hay que fijar su valor con la función `set`. En este caso, primero hay que convertir el número a una cadena de caracteres usando `string()` y luego concatenarla con "`xi = `" para generar la salida.

Como resultado, cada vez que el usuario pulsa el botón "Calcular" las dos raíces se "imprimen" tanto en el *prompt* como en los elementos de texto en la interfaz.

Construir una interfaz usando código requiere muchos ensayos y errores para alinear los elementos y fijar sus tamaños. Sin embargo, existe un módulo llamado GUI Builder, disponible en la categoría GUI de ATOMS, que permite dibujar interfases directamente.

Apéndice D

Xcos

Xcos (antes llamado Scicos) es un programa complementario para crear y simular modelos definidos por ecuaciones algebraicas y diferenciales. Aunque Xcos tiene sus propias ventanas y menús, depende de la instalación de Scilab. Se activa desde el menú principal de la consola, en la entrada `Applications`, o con su respectivo icono.

La clave de Xcos es que en lugar de escribir las ecuaciones en código se "dibujan" con bloques conectados entre sí por líneas. Cada tipo de bloque representa una operación matemática o lógica distinta. Están disponibles en una ventana llamada `Palette browser` (ver figura D.1), que se puede activar desde la entrada `View` del menú de cualquier ventana Xcos.

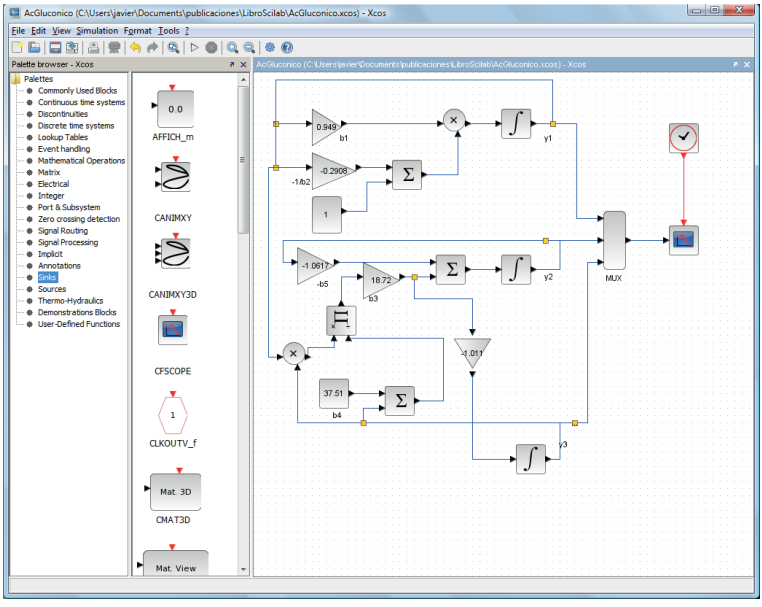


Figura D.1. Ventana de Xcos acompañada del Palette Browser

Por ejemplo, un modelo de la producción de ácido glucónico por fermentación usando la bacteria *Pseudomonas ovalis* tiene tres ecuaciones diferenciales ordinarias (Constantinides y Mostoufi, 1999):

$$
\begin{aligned}
\frac{dy_1}{dt} &= b_1 y_1 \left(1 - \frac{y_1}{b_2}\right) \\
\frac{dy_2}{dt} &= b_3 \frac{y_1 y_3}{b_4 + y_3} - b_5 y_2 \\
\frac{dy_3}{dt} &= -1.011 b_3 \left(\frac{y_1 y_3}{b_4 + y_3}\right)
\end{aligned}
\tag{D.1}
$$

donde t es el tiempo, medido en horas, y_i representa la concentración del microorganismo o sustancia i y las constantes son $b_1 = 0.949$, $b_2 = 3.439$, $b_3 = 18.72$, $b_4 = 37.51$ y $b_5 = 1.0617$. En este caso podría escribirse una función para los lados derechos de la ecuación D.1 e integrarla con la función `ode`, tal como se explica en la sección 6.2. Pero en lugar de eso, en Xcos se hace un diagrama que representa el sistema de ecuaciones.

En la figura D.1:

- Las líneas "transportan" los valores generados entre bloques.
- Los bloques triangulares `GAINBLK_f` representan "ganancia". Multiplican el valor de entrada por una cantidad predeterminada.
- Los bloques $\sum$ y $\prod$ suman y multiplican valores.
- Los tres bloques $\int$ son integradores. En este modelo, generan los valores de y_1, y_2 y y_3 resolviendo dy_1/dt, dy_2/dt, y dy_3/dt (de ahí su nombre). En cada uno, la línea entrante representa el lado derecho de la ecuación diferencial y la saliente, el valor de la variable respectiva.
- Los lados derechos de las ecuaciones diferenciales se construyen conectando varios bloques. Por ejemplo, para dy_2/dt se conectan los bloques que producen $-b_5 y_2$, $y_1 y_3$ y $b_4 + y_3$. Cada constante b_i está representada por un bloque de ganancia.

Las condiciones iniciales –es decir, los valores de y cuando $t = 0$– se especifican en cada bloque integrador. En este caso, $y_1(t = 0) = 0.5$ y $y_2(t = 0) = y_3(t = 0) = 0$). La duración de la simulación –es decir, el valor máximo de t– se establece en la entrada `Simulation/Setup` del menú de la ventana. Los resultados se "recogen" en el bloque `MUX` en el extremo derecho del diagrama, que los alimenta al bloque `CSCOPE` donde se grafican en función del tiempo. La simulación se inicia con el botón `Simulation/Start` y produce el resultado de la figura D.2.

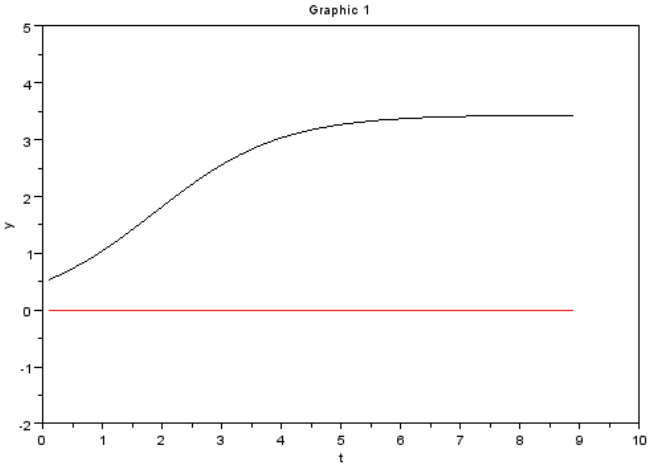


Figura D.2. Resultado de la simulación con Xcos

Xcos es más una aplicación independiente de control y simulación que un componente de Scilab. Incluye bloques específicos para simular sistemas mecánicos, hidráulicos, eléctricos, etcétera. Una descripción completa de Xcos requeriría un libro aparte (para los lectores interesados, se recomiendan las referencias Campbell, Chancelier y Nikoukhah (2006); Rodríguez (2007)).

Solucionario para las guías de autoaprendizaje

Capítulo 1

2-3) Ecuación cuadrática:

```
-->a = 1; b = 2; c = 1;
-->aux = -b / ( 2 * a );
-->raiz = sqrt( ( aux ^ 2 ) - ( c / a ) );
-->x1 = aux - raiz
ans
-1
-->x2 = aux + raiz
ans
-1
```

Capítulo 2

2) Escribir y leer una matriz generada:

```
matrizAzar = rand(1000, 10);
fprintfMat("ejemploCap2.txt", matrizAzar, "%10.4e");
clear("matrizAzar");
matrizLeida = fscanfMat("ejemploCap2.txt");
```

4) Leer una matriz guardada desde una hoja de cálculo en formato texto (se supone que el separador decimal es la coma):

```
matrizExcel = fscanfMat("ejemImportExcel.txt")
```

6) Leer la primera hoja de un archivo xls (Excel):

```
[undLogica, letrasHojas, nomHojas, posHojas] = ...
              xls_open("ejemImportExcel.xls")
[matrizExcel, textInd] = xls_read(undLogica, posHojas(1))
mclose(undLogica)
```

Capítulo 3

3) Concatenación de matrices de 3x3 y 4x4, con valores al azar:

```
A = round(10*rand(3,3));
```

```
B = round(10*rand(4,4));
c8 = round(10*rand(4,1));
f4 = round(10*rand(1,3));
res = [[A; f4] B c8];
```

5) Generación de una matriz tridiagonal con valores predefinidos:

```
d1 = diag([5 8 10 6 3]);
d2 = diag([7 0 4 1],1);
d3 = diag([11 8 9 6],-1);
res = d1 + d2 + d3
```

7, 8 ,9) Creación de un polinomio a partir de las raíces, y de una lista y una célula con los elementos del polinomio:

```
poli1 = poly([5 -9 3 4], "x", "roots") ;
lstElemPol = list(degree(poli1), coeff(poli1),
varn(poli1) ) ;
cellElemPol = cell(3);
cellElemPol(1).entries = degree(poli1);
cellElemPol(2).entries = coeff(poli1);
cellElemPol(3).entries = varn(poli1);
```

CAPÍTULO 4

2) Función alterna para calcular el logaritmo natural:

```
function salida = altLog(x, resolucion)
  if ( exists("resolucion") == 0 ) then
    resolucion = 1.0E-6
  end
  base = x - 1.0
  xm1p = 1
  seguir =%t
  potn = -1
  suma = 0.0
  n = 0
  while (seguir ==%t)
       n = n + 1
       xm1p = xm1p * base
       potn = potn * (-1)
       termino = potn * xm1p / n
       suma = suma + termino
       if ( abs(termino) <= resolucion ) then
         seguir =%f
       end
  end
  salida = suma
endfunction
```

3) Función alterna para calcular una raíz de una ecuación cuadrática:

```
function salida = iterCuad(vecParam, xini, crit,  nMaxIter)
  if ( exists("crit") == 0 ) then
    crit = 1.0E-6
  end
  if ( exists("nMaxIter") == 0 ) then
    nMaxIter = 1000
  end
  [a, b, c] = (vecParam(1), vecParam(2), vecParam(3))
  n = 0
  seguir =%t
  x = xini
  while (seguir ==%t)
       xviejo = x
       x = -( c + ( a * ( x ^ 2 ) ) ) / b
       if ( abs( x - xviejo ) <= crit ) then
         seguir =%f
       end
       n = n + 1
       if ( n > nMaxIter ) then
         seguir =%f
         disp("excedio nMaxIter")
       end
  end
  salida = x
endfunction
```

4) Cálculo de la serie $1+(1/x)+\left(1/x^2\right)+\cdots$ con salvaguardas:

```
function salida = serieRara(x, n, limCero)
  if ( exists("limCero") == 0 ) then
    limCero =%eps
  end
  try
     termino = 1.0
     divisor = 1.0
     suma = 1.0
     for i=1:n
        divisor = divisor * x
        termino = 1.0 / divisor
        suma = suma + termino
        if ( abs(termino) <= limCero ) then
          disp("detenida en i = "+string(i))
          disp("termino = "+string(termino))
          break
        end
      end
      salida = suma
  catch
```

```
        //error 27: division por cero
        salida =%inf
        disp("Division por cero en serieRara")
        errcatch(27, action="pause", option="kill")
  end
endfunction
```

CAPÍTULO 5

1, 3, 4) Gráfica de las funciones de Bessel. *Nota: se usó código LaTeX, entre $ para los ejes, pero se puede usar también texto: normal.*

```
listX = 0:0.01:10;
np = length(listX);
n0 = 5
resBessel = zeros(5,np)
for i=1:n0
   resBessel(i,:) = besselj(i,listX);
end
plot(listX,resBessel)
figBessel = get("current_figure");
ejesBessel = get("current_axes");
lineasBessel = get("current_entity");
ejesBessel.x_location = "middle"
ejesBessel.x_label.text = "$x$"
ejesBessel.y_label.text = "$B\left( x \right)$"
```

5) Gráfica tridimensional y gráfica de contorno:

```
function salida = f2var(x,y)
  salida = ( x * ( y ^ 2 ) ) + x - ( 10 * y ) +8
endfunction
listaX = -5:0.1:5;
listaY = -5:0.1:5;
nx = length(listaX);
ny = length(listaY);
matrZ = zeros(nx,ny);
for ix=1:nx
   x = listaX(ix)
   for iy=1:ny
      y = listaY(iy)
      matrZ(ix,iy) = f2var(x,y)
   end
end
subplot(2,1,1)
plot3d(listaX,listaY,matrZ)
subplot(2,1,2)
contour(listaX,listaY,matrZ,[-150:10:150])
```

CAPÍTULO 6

3) Integración de ecuaciones diferenciales:

```
[a1, a2, a3, a4, a5] = (0.50, 4.0, 20, 30, 1.5)
function salida = ladoIzq(t, ly)
  y1 = ly(1)
  y2 = ly(2)
  y3 = ly(3)
  salida = ly * 0
  salida(1) = a1 * y1 * ( 1.0 - ( a2 * y2 ) )
  salida(2) = ( a3 * y1 * y3 / ( 1 + y3 ) ) - ( a4 * y2 )
  salida(3) = 1 - ( a5 * ( y1 * y2 / ( 1 + y2 ) ) )
endfunction
yini = [1 0 0]';
list_t = 0:0.01:5;
list_y = ode(yini, 0, list_t, ladoIzq)
```

5, 6) Ajuste de datos. Los resultados a_0 y a_1 del caso lineal se usan como estimados iniciales para el caso no lineal:

```
//regresión lineal
lista_x = [300 320 340 360 380 400];
lista_y = [0.1381 0.3203 0.6613 1.2425 2.1614 3.5284];
nval = length(lista_x);
lista_1x = ( 1.0 ./ lista_x )';
lista_unos = ones(nval,1);
X = [lista_unos, lista_1x];
Y = log( lista_y )';
A = lsq(X,Y);
a0 = A(1);
a1 = A(2);
//verificacion
lista_ycalc = exp( a0 + ( a1 ./ lista_x ) ) ;
disp( [lista_x' lista_y' lista_ycalc'] )
//forma no lineal
function salida = fAjNL(entrada)
  a0 = entrada(1)
  a1 = entrada(2)
  a2 = entrada(3)
  aux = a0 + ( a1 ./ ( lista_x + a2 ) )
  lista_g = lista_y - exp( aux )
  salida = lista_g'
endfunction
aIni = [11.0 -3887.87 100]';
[valMin, aOpt] = leastsq(fAjNL, aIni)
//verificacion
a0 = aOpt(1);
a1 = aOpt(2);
a2 = aOpt(3);
```

```
lista_ycalc = exp( a0 + ( a1 ./ ( lista_x + a2 ) ) ) ;
disp( [lista_x' lista_y' lista_ycalc'] )
```

CAPÍTULO 7

1-4) Lista tipificada y sobrecarga de operadores:

```
function nuevaObsv = crearObsv(v, m, s)
    if ( exists("s") == 1 ) then
        f = v / ( ( m * 60 ) + s )
        nuevaObsv = mlist(["tObsv", "volumen", "minutos", ...
                           "segundos", "flujo"], v, m, s, f)
    else
        nuevaObsv = mlist(["tObsv", "volumen", "minutos", ...
                           "segundos", "flujo"])
    end
endfunction
function%tObsv_p(unaObsv)
   tiempo = ( unaObsv.minutos * 60 ) + unaObsv.segundos
   unaObsv.flujo = unaObsv.volumen / tiempo
   disp([unaObsv.flujo unaObsv.volumen tiempo])
endfunction
function unaObsv =%s_i_tObsv(indx, entrada, unaObsv)
    select (indx)
       case (1) unaObsv.volumen  = entrada
       case (2) unaObsv.minutos  = entrada
       case (3) unaObsv.segundos = entrada
    end
    tiempo = ( unaObsv.minutos * 60 ) + unaObsv.segundos
    unaObsv.flujo = unaObsv.volumen / tiempo
endfunction
function nObsv =%tObsv_a_tObsv(obsv1, obsv2)
      nObsv.volumen = obsv1.volumen + obsv2.volumen
      nObsv.minutos = obsv1.minutos + obsv2.minutos
      nObsv.segundos = obsv1.segundos + obsv2.segundos
      nObsv.flujo = 0.5 * ( obsv1.flujo + obsv2.flujo )
endfunction
```

Bibliografía

Baudin, M. (2010). *Introduction to Scilab*. Rocquencourt: Scilab Consortium. Recuperado de http://www.scilab.org/content/download/1754/19024/file/introscilab.pdf

Baudin, M. (2010). *Scilab is not naive*. Rocquencourt: Scilab Consortium. Recuperado de http://www.scilab.org/content/download/1106/10848/file/scilabisnotnaive.pdf

Baudin, M. y Couvert, V. (2010). *Optimization in Scilab*. Rocquencourt: Scilab Consortium. Recuperado de http://www.scilab.org/content/download/1390/13697/file/optimization_in_scilab.pdf

Campbell, S. L., Chancelier, J. P. y Nikoukhah, R. (2006). *Modeling and simulation in Scilab/Scicos*. Nueva York: Springer.

Constantinides, A. y Mostoufi, N. (1999). *Numerical methods for chemical engineers with Matlab applications*. Upper Saddle River, Nueva Jersey: Prentice-Hall

Gómez, C. (ed.) (1999). *Engineering and scientific computing with Scilab*. Birkhäuser Boston

Van Dijk, L. E. y Spiel, C. L. (2000). *Scilab bag of tricks: The Scilab-2.5 IAQ (Infrequently Asked Questions)*. Recuperado de http://kiwi.emse.fr/SCILAB/sci-bot/sci-bot.pdf

Mora, H. M. (2005). *Introducción a Scilab*. Recuperado de http://wiki.scilab.org/Tutorials?action=AttachFile&do=get&target=Scilab4Matlab.pdf

Rietsch, E. (2010). *An introduction to Scilab from a Matlab user's point of view*. Rocquencourt: Scilab Consortium.

Rodríguez, C. (2007). *Manual de simulación para sistemas de control usando el software Matlab y Scilab*. Universidad Veracruzana (México), Facultad de Ingeniería Mecánica Eléctrica. Recuperado de http://es.scribd.com/doc/43608860/Manual-de-Simulacion

Urroz, G. E. (2001). *Optimization techniques with Scilab*. InfoClearing House. Recuperado de http://www.infoclearinghouse.com

Introducción a la computación numérica
usando la herramienta Scilab

Otros títulos de la Serie Techné

Stable computations by discrete mollification (2014)
Carlos Daniel Acosta Medina
Carlos Enrique Mejía Salazar

La custodia de los hijos en las parejas separadas (2014)
Yolanda López Díaz

Repelencia al agua en Andisoles de Antioquia (2011)
Daniel Francisco Jaramillo Jaramillo

Gestión empresarial en Colombia: un aporte desde la administración (2011)
Gregorio Calderón
Julia Naranjo
Claudia Álvarez

Desarrollo rural: superando el desarrollo agrícola (2011)
Fabio Alberto Pachón Ariza

javier ignacio
carrero mantilla

Javier Ignacio Carrero Mantilla, ingeniero químico, egresado de la Universidad del Valle (Cali, Colombia); obtuvo la maestría y el doctorado en Ingeniería en la misma institución. Desde 2006 es profesor del Departamento de Ingeniería Química en la Universidad Nacional de Colombia, sede Manizales. Allí ha dictado cursos de métodos numéricos aplicados a ingeniería química y matemáticas avanzadas.

Este libro enseña conceptos básicos de computación numérica: constantes, expresiones, funciones, métodos numéricos, trazado de gráficas, etcétera, en un enfoque práctico. Cada capítulo se basa en objetivos de aprendizaje que se alcanzan mediante explicaciones aplicadas a un entorno computacional (Scilab). Estos objetivos son también la base de las guías de autoaprendizaje incluidas al final de los capítulos. Para mantener la consistencia en el texto se usó un solo *software* en toda la obra, pero los conceptos aplicados son generales y el lector los puede aplicar a cualquier otro *software* de su elección. El uso de este libro es un complemento al aprendizaje de métodos numéricos o matemáticas aplicadas para estudiantes de ingeniería y ciencias.

editorial UNAM