Introducción a Scilab

Federico Miyara

1 Generalidades

Scilab es un software libre de cálculo numérico matricial que opera directamente con matrices y vectores, tanto en lo que respecta a las operaciones clásicas de la teoría de matrices (suma, producto, transposición, cálculo de determinantes, inversión de matrices cuadradas, etc.) como a operaciones masivas sobre un conjunto de datos (por ejemplo, la aplicación de una función o procedimiento a todas las componentes de una matriz). En particular resulta interesante para el procesamiento digital de señales ya que con una sola instrucción se puede aplicar un procedimiento iterativo.

Puede descargarse de https://www.scilab.org/ y posee licencia GPL (*General Public License*, Licencia Pública General) de la Free Software Foundation.[1] En http://www.scilab.org/download (figura 1) se pueden descargar los instaladores para Linux, Windows o Mac OS X (figura 2). La versión actual (2025) es la 2025.1.0 y si bien en general las diferentes versiones no son completamente retrocompatibles, la estructura básica y la vasta mayoría de los comandos y funciones del lenguaje se mantienen.

Para su instalación en Windows es necesario seleccionar primero el idioma de la guía de instalación, luego debe aceptarse la licencia. A continuación (figura 3) se debe decidir dónde se instalará. En general la ubicación propuesta por defecto puede aceptarse sin problemas salvo que tengamos una forma muy personal de organizar nuestros programas. Luego es necesario seleccionar los componentes a instalar (figura 4). En general conviene instalarlos a todos para evitar que en el futuro algunas utilidades pudieran no funcionar. Por último, es necesario confirmar la ubicación del ícono del programa en el escritorio así como la asociación con Scilab de los archivos con las extensiones del programa, por ejemplo .sci y .sce (figura 5).

La instalación en otros sistemas operativos tiene las particularidades de cada uno de ellos. En el caso de Linux, por ejemplo se instala a partir de una *appimage* que es una versión portable del programa que se instala como archivo único en cualquier ubicación y luego el programa puede ser ejecutado como cualquier programa en ese sistema operativo. Lo interesante es que la interfaz es idéntica en todos los sistemas operativos, lo cual permite una interoperabilidad perfecta.

Una vez completada la instalación e iniciado el programa aparece la interfaz gráfica de usuario (figura 6). Consta de: 1) Explorador de archivos, donde se muestran los archivos disponibles. 2) Consola, donde se escriben comandos con la finalidad de ejecutarlos inmediatamente. 3) Visor de variables, que muestra algunos datos de las variables (nombre, tamaño, valores si son pocos, tipo de datos y memoria ocupada. 4) Historial de comandos, que almacena cada uno de los comandos escritos en la consola, lo cual permite luego reproducir cualquiera de ellos. 5) Noticias e informaciones sobre el programa.

---

[1] En términos generales esto implica la libertad de usar el software para cualquier propósito, ya sea en forma personal, académica o comercial, para modificarlo, para copiarlo y compartirlo, para compartir las modificaciones realizadas y para publicar dichas modificaciones siempre que se lo haga bajo la misma licencia GPL. Pueden obtenerse más detalles en http://www.gnu.org/copyleft/gpl.html

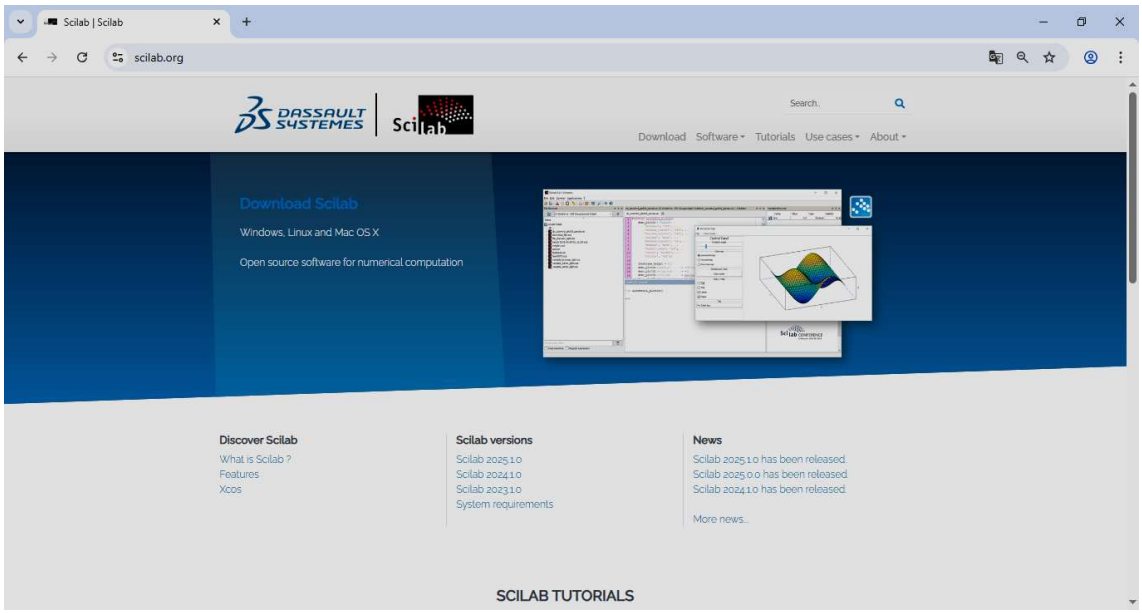

Figura 1. Pagina inicial del sitio de Scilab https://www.scilab.org/

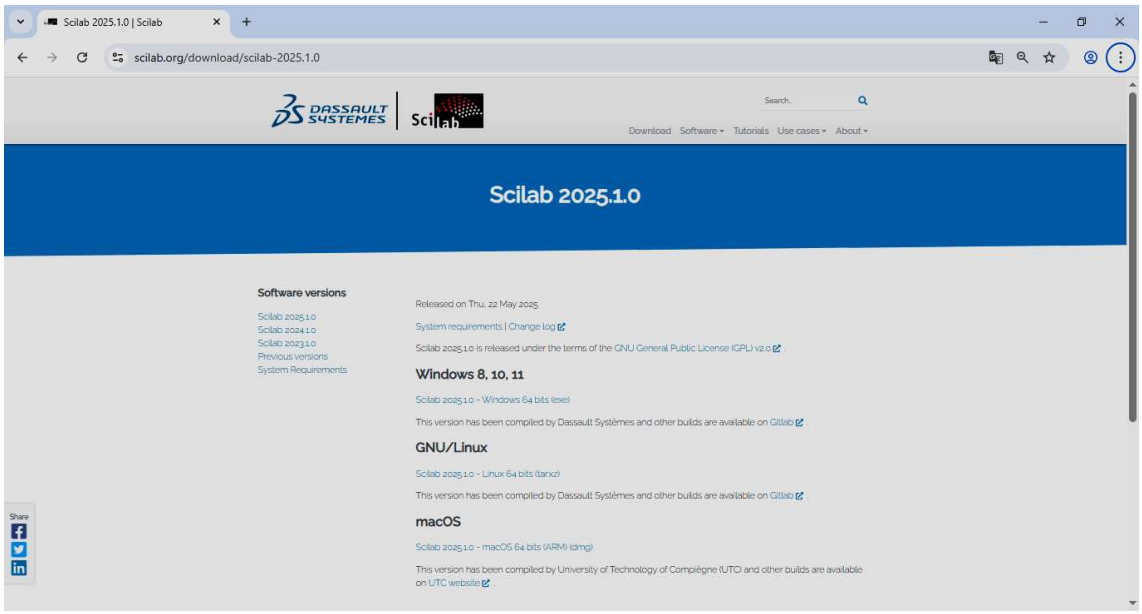

Figura 2. Página de descarga de Scilab para las diversas plataformas

Scilab es un lenguaje de programación *interpretado*, es decir que el programador escribe instrucciones y el software interpreta y ejecuta cada una de ellas. Estas instrucciones o comandos se escriben directamente en la consola en la *línea de comandos* indicada por el símbolo `-->` y se ejecutan pulsando Enter.

Antes de avanzar en detalles específicos de la programación, es útil tener en cuenta el comando `doc` (anteriormente `help`), que puede utilizarse en la consola, seguido por el nombre de cualquier función o instrucción para acceder a la documentación de la misma, que se abre en una ventana aparte. En ella se explican la operación que realiza y los argumentos de entrada y salida, y se brindan ejemplos de uso. Por ejemplo,

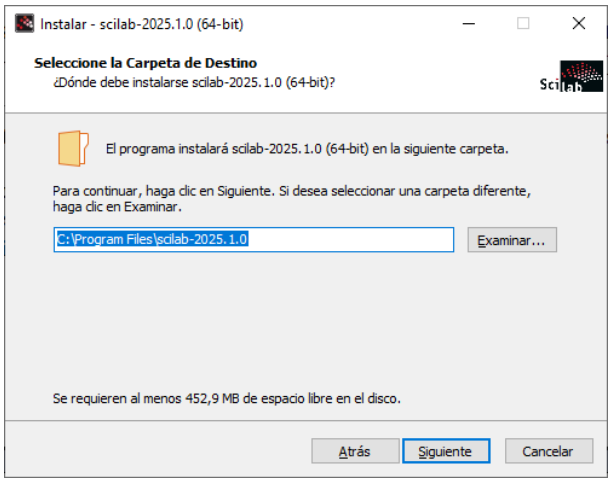

Figura 3. Selección de la ubicación de la instalación

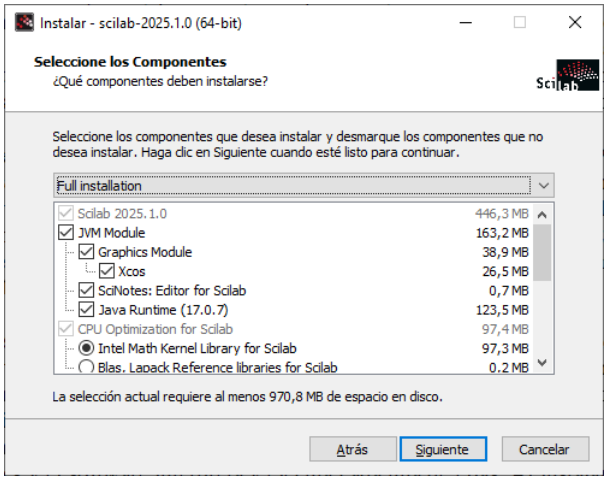

Figura 4. Selección de componentes a instalar

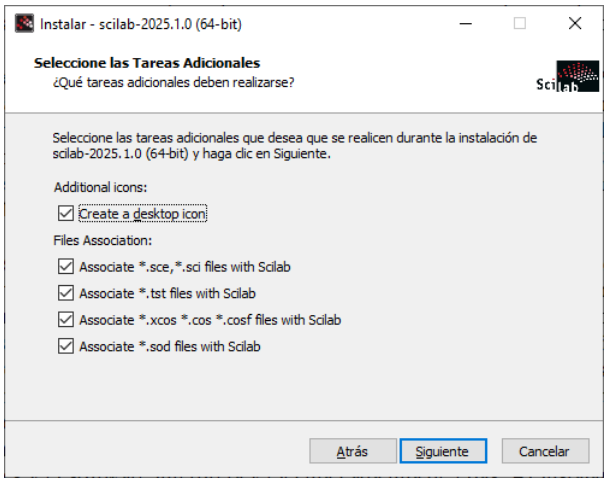

Figura 5. Ícono en el escritorio y asociación con Scilab de archivos según su extensión.

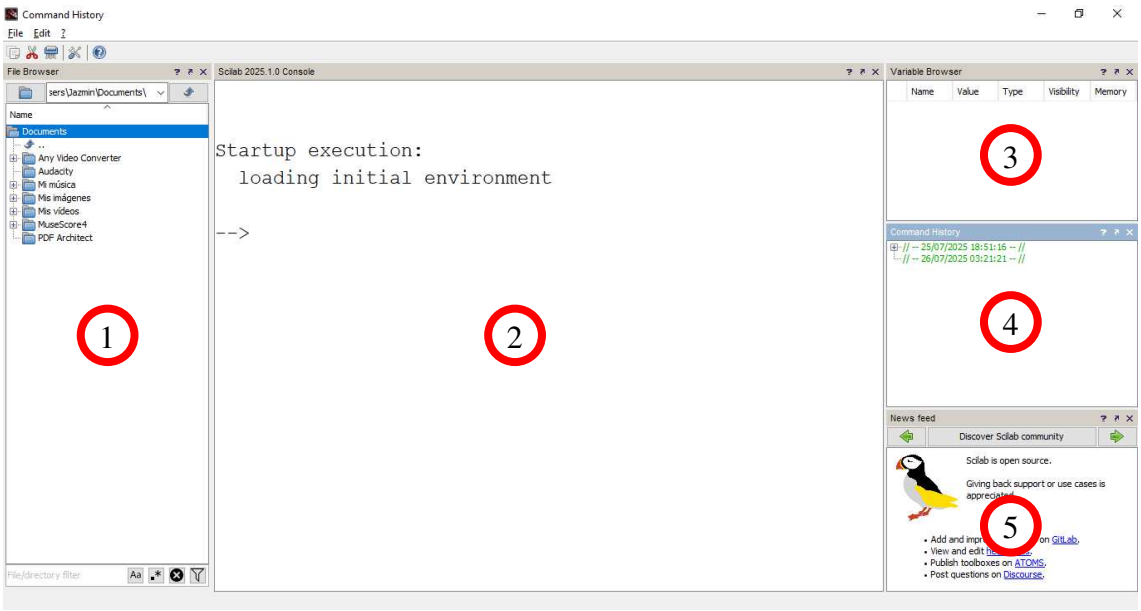

Figura 6. Interfaz gráfica de Scilab. 1) Explorador de archivos. 2) Consola. 3) Visor de variables. 4) Historial de comandos. 5) Noticias e informaciones.

al escribir

```
doc exp
```

y oprimir Enter se obtiene la documentación de la función exp (exponencial de base e), como se muestra en la figura 7. Por otra parte, el comando `apropos` (de *à propos*, "acerca de" en francés) permite realizar búsquedas de una palabra o fragmento dentro de la documentación, y es útil cuando se desea buscar un comando del cual no se conoce o no se recuerda el nombre exacto o completo. El comando `doc` también se comporta como `apropos` si la expresión buscada no corresponde a una función existente.

2 Vectores y matrices

Dado que Scilab trabaja fundamentalmente con vectores y matrices,[2] veamos primero diversas formas de crear matrices. La más simple es utilizar el operador de asignación, "=", encerrando entre corchetes una lista de números separados por comas para delimitar las componentes de una fila y puntos y coma para delimitar las filas:

```
--> a = [1, 2; 3, 4]
```

Al oprimir Enter se obtiene

```
a  =
   1.    2.
   3.    4.
```

[2] Un *vector* es un caso particular de matriz de una sola fila (vector fila) o una sola columna (vector columna). Un *escalar* es el caso particular en el que la matriz tiene una fila y una columna.

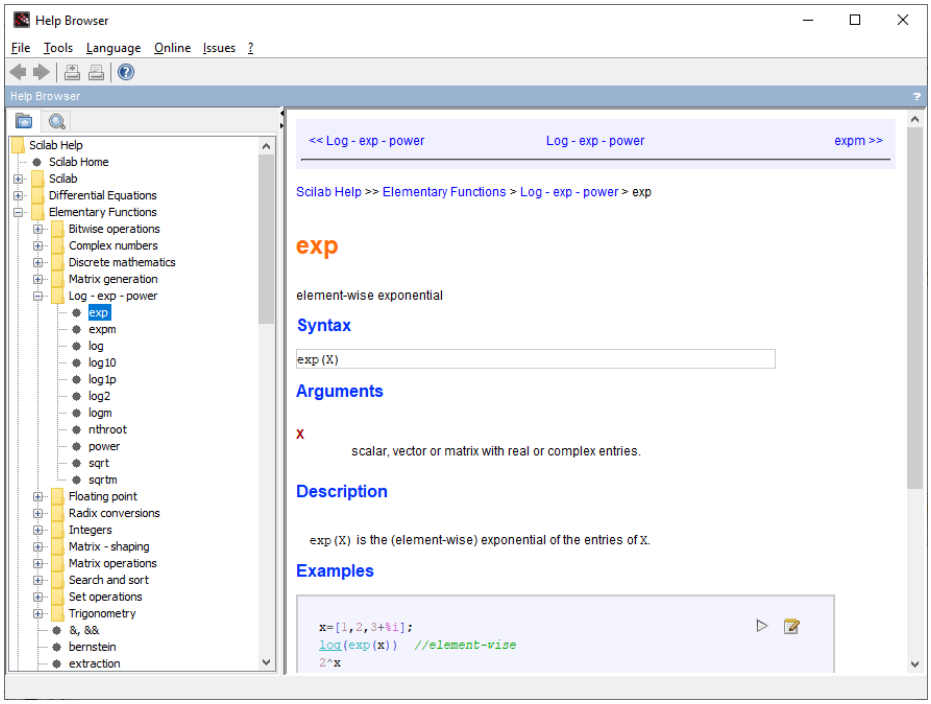

Figura 7. Ejemplo de documentación invocada mediante el comando `doc`, en este caso de la función exp.

Los puntos decimales enfatizan el hecho de que los números son reales de doble precisión, un formato que utiliza 8 bytes (64 bit) de almacenamiento por cada número en formato de punto flotante, normalizado por la norma IEEE 754.

Nótese que por defecto Scilab introduce espacios innecesarios entre la variable y el resultado. Para evitarlos se debe ejecutar la instrucción `mode(0)`.

Las comas pueden reemplazarse por espacios y los puntos y coma por pulsaciones de Enter. Las siguientes dos formas producen idénticos resultados:

```
--> a = [1 2; 3 4]

--> a = [1 2
  > 3 4]
```

Notar que en la segunda forma, al oprimir Enter sin haber completado la sintaxis del comando (es decir, el cierre del corchete), el símbolo "-->" cambia a " >", quedando en espera de que se complete un comando sintácticamente válido.

Opcionalmente, puede agregarse un punto y coma al final del comando:

```
--> a = [1 2; 3 4];
```

En tal caso el comando (en este ejemplo, la asignación de las componentes de la matriz a la variable `a`) se ejecuta, pero el resultado no se muestra en pantalla. Esto es especialmente útil en el caso de matrices de gran tamaño, o en programas o funciones que realizan varios pasos intermedios cuya visualización es irrelevante.

También pueden usarse la coma (espacio) y el punto y coma (o Enter) como operadores de *concatenación*. Si `b` y `c` son vectores fila, entonces

```
--> a = [b, c];
```

es un vector fila que contiene todas las componentes de `b` seguidas por las de `c`. Si `b` y `c` son vectores columna,

```
--> a = [b; c];
```

es un vector columna que contiene todas las componentes de `b` seguidas por las de `c`. Las comas también permiten concatenar matrices de igual número de filas, y los puntos y coma, matrices de igual número de columnas.

3 Scripts y el editor SciNotes

El uso de la consola se restringe normalmente a la ejecución de comandos individuales o unos pocos comandos aislados que no necesitan repetirse. De mayor interés es la realización de programas, consistentes en una sucesión de comandos, ya que pueden reutilizarse con diferentes juegos de datos. Si bien estos programas pueden escribirse en cualquier editor de texto, el entorno de trabajo de Scilab proporciona el editor integrado SciNotes, que puede abrirse desde el menú Aplicaciones (figura 8). Una vez abierto aparece como una ventana flotante (figura 9). Es una buena idea integrarla a la ventana principal del programa. Dado que se trata de una ventana acoplable (*dockable* en inglés), basta arrastrarla desde la barra que se encuentra debajo de la barra de íconos y ubicarla, por ejemplo, arriba de la consola (figura 10). La ventaja de SciNotes por sobre un editor de texto externo es que posee prestaciones tales como resaltado de sintaxis, depuración (*debugging*), ejecución completa o parcial, etc. Los programas escritos en SciNotes, denominados *scripts*, pueden guardarse como archivos de texto con extensión .SCE o, en el caso de funciones (que veremos más adelante), .SCI.

Dado que en lo sucesivo trabajaremos fundamentalmente pensando en programas o funciones, que se escriben en SciNotes, omitiremos el símbolo `-->`, correspondiente únicamente a la consola.

4 Configuración

Existe una serie de opciones de configuración que permiten personalizar la interfaz, Se puede acceder a las mismas haciendo foco en la consola y abriendo el menú *Edit / Preferences*. La ventana que se abre se muestra en la figura 11. Dentro de las opciones

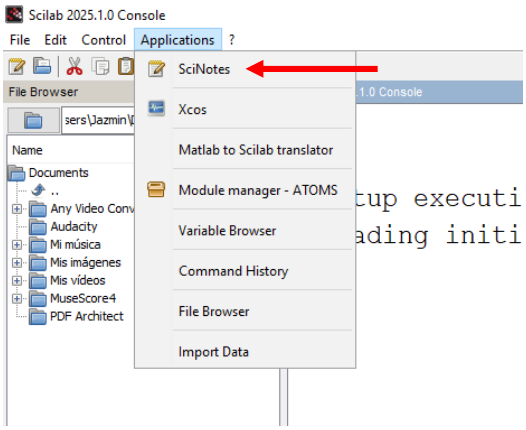

Figura 8. Menú donde se puede abrir el editor de scripts SciNotes.

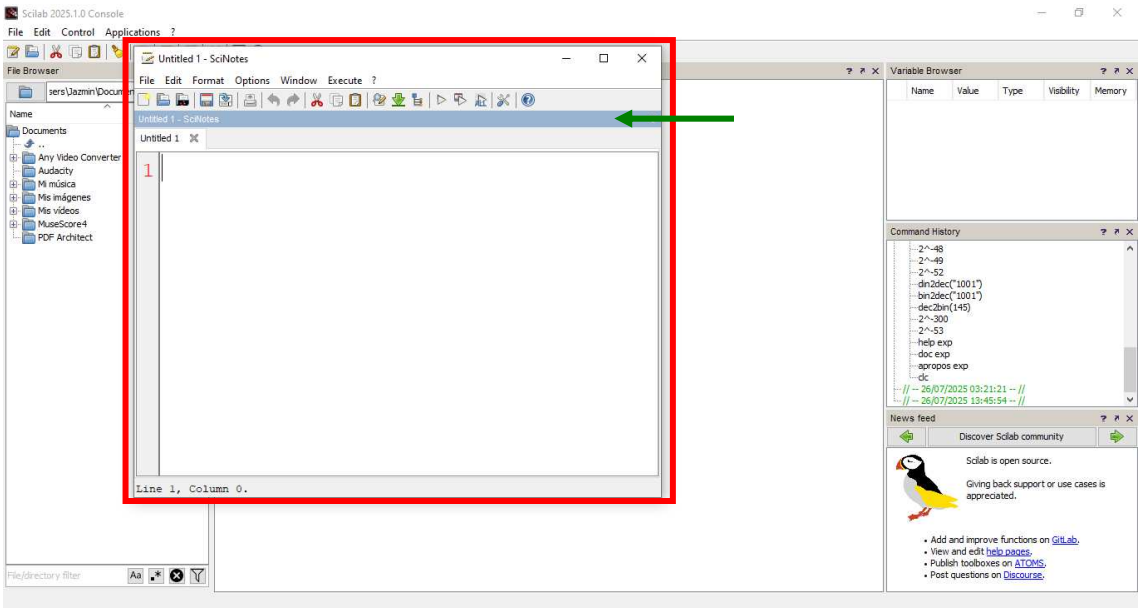

Figura 9. Editor de scripts SciNotes como ventana flotante. La flecha verde indica donde tomar y arrastrar con el mouse para acoplar la ventana a la interfaz principal.

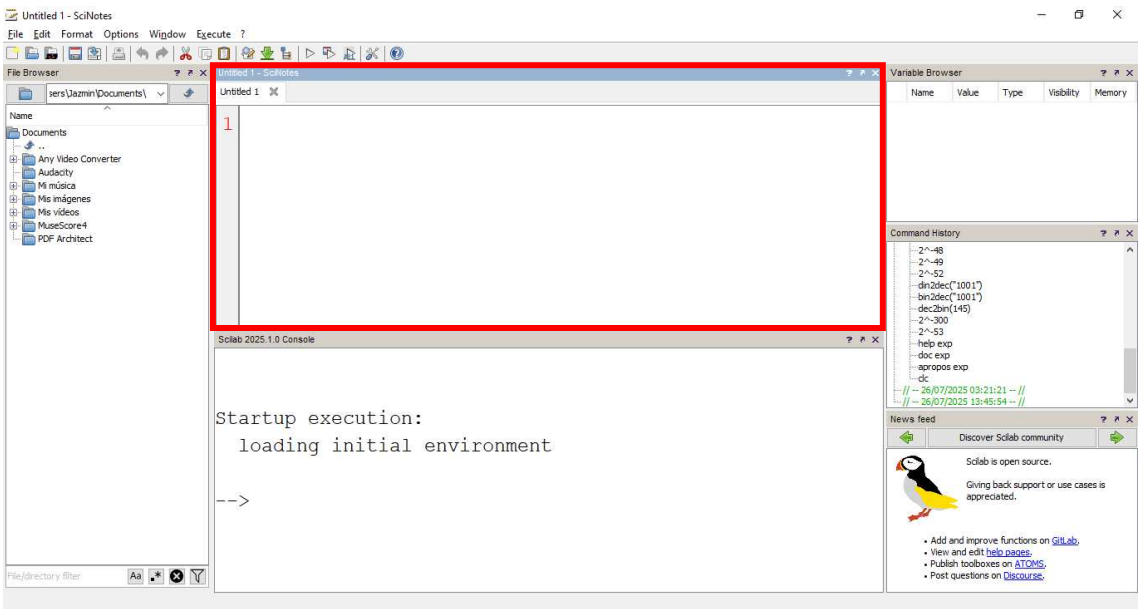

Figura 10. Editor de scripts SciNotes como ventana acoplada a la interfaz de Scilab.

opciones generales se encuentran la selección del idioma (que por defecto es el inglés) y la selección del directorio de trabajo, que puede personalizarse ingresando la ruta deseada en la casilla *Use default directory*. En cuanto al idioma debe tenerse en cuenta que el cambio requiere el reinicio del programa. Por otro lado, los comandos e instrucciones corresponden en general a palabras o siglas en inglés, como `for`, `while`, `if`, `sin`, y la configuración del idioma no las afecta. Sólo afecta la interfaz.

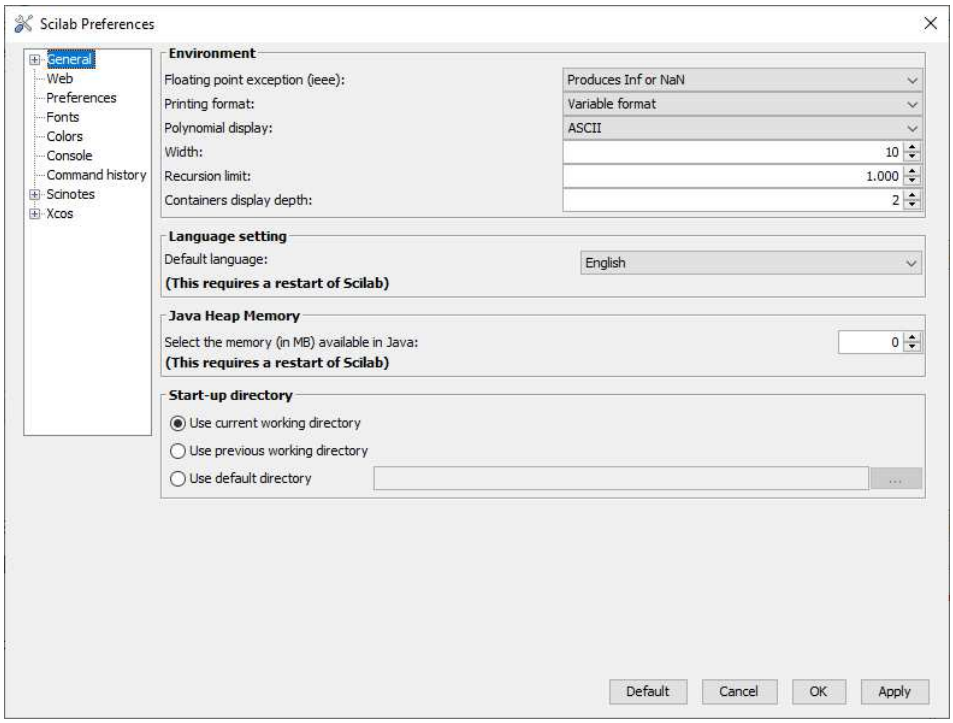

Figura 11. Ventana de preferencias de configuración de Scilab.

Otra opción de interés que se visualiza haciendo clic en el signo + es *Desktop Layout* (distribución del escritorio). Aquí conviene dejar tildado *Save layout on exiting* para conservar la distribución elegida (por ejemplo la que incluye el editor SciNotes acoplado) sin tener que repetir cada vez los mismos pasos.

Finalmente, en las preferencias de SciNotes puede ser conveniente, según la forma de trabajo del usuario, mantener tildado *Restore previous session on start-up*, es decir, restaurar la sesión previa al inicio, lo cual permite seguir trabajando en los scripts en los que se estaba trabajando en la sesión anterior, evitando tener que volver a abrirlos cada vez que se inicia el programa.

5 Comentarios

Para el mantenimiento y actualización de cualquier código de software es importante la inclusión de comentarios descriptivos o explicativos sobre las diferentes instancias de los algoritmos. Su presencia facilita posteriormente la lectura del código y sus posibles mejoras.

En Scilab los comentarios se introducen mediante una doble barra `//`, igual que en el lenguaje C. Cualquier texto que se encuentre a la derecha será ignorado. En la siguiente línea de código

```
Fs = 44100;      // Tasa de muestreo en Hz
```

el comentario indica el significado de la variable.

A veces se utiliza el recurso de aplicar una doble barra al comienzo de una o más instrucciones válidas para evitar su ejecución. Esto es útil para saltear algunos pasos en un algoritmo o para comparar dos formas diferentes de obtener un mismo resultado, o para conservar una versión anterior mientras se depura la nueva. Incluso existe un ítem del menú Formato para comentar o descomentar una selección.

6 Vectores y matrices especiales

Continuando con los métodos de creación de vectores y matrices, el *operador de rango* "`:`" permite crear un vector de números que se van incrementando de a una unidad entre dos valores extremos. Por ejemplo,

```
a = 1:m;
```

es un vector fila con componentes `1, 2, ..., m`, donde `m` es una variable creada previamente mediante asignación de un valor.[3] Con mayor generalidad,

```
a = x1:h:x2;
```

crea un vector de números separados por un incremento `h`, no necesariamente 1, es decir: `x1`, `x1 + h`, `x1 + 2*h`, ..., `x1 + r*h`, donde `r` es tal que `n + r*h` sea el máximo posible menor o igual que `x2` (lo cual significa que `x2` puede no ser alcanzado).

El comando `linspace()` actúa de manera similar, permitiendo obtener un vector fila de `n` componentes linealmente espaciadas entre dos valores `x1` y `x2`:

```
a = linspace(x1, x2, n);
```

Es equivalente a `[x1:(x2-x1)/(n-1):x2]`. El comando `logspace()`, por otra parte, invocado mediante

```
a = logspace(d1, d2, n];
```

crea un vector de `n` componentes equiespaciadas logarítmicamente entre `10^d1` y `10^d2`, donde `^` es el operador de potenciación. Resulta equivalente a `10^linspace(d1, d2, n)`

Es posible crear matrices de `n` filas y `m` columnas formadas por ceros mediante `zeros(n,m)`, y por unos mediante `ones(n,m)`:

```
a = zeros(3,4)
 a = [3x4 double]
   0.   0.   0.   0.
   0.   0.   0.   0.
   0.   0.   0.   0.
```

En muchos casos se requiere generar números aleatorios, por ejemplo en procesos de simulación de errores experimentales o de medición, o para generar ruido aleatorio. Para ello existen las funciones `rand` y `grand`, las cuales permiten obtener números aleatorios con diferentes distribuciones estadísticas. Así

```
x = rand(m, n, "normal");
```

genera una matriz de `m` filas y `n` columnas de números aleatorios distribuidos normalmente con media 0 y desvío estándar 1. La función `grand` ofrece mayor flexibilidad, aunque requiere más argumentos. El código

[3] En todos los casos en que invoquemos una variable, la misma debe haber sido creada previamente, de lo contrario se obtendrá un mensaje de error.

```
x = grand(m, n, "nor", Av, Sd);
```

genera una matriz de `m` filas y `n` columnas de números aleatorios con distribución normal (gaussiana) con media `Av` y desvío estándar `Sd`. Por ejemplo:

```
x = grand(3, 4, "nor", 0, 1)
 x = [3x4 double]
  -0.540479    2.6169952  -0.9412785   0.0081632
  -0.827319    2.8431839   0.7009956   0.0682391
   1.5340618   0.0417409   0.0345522  -1.1478805
```

Análogamente,

```
x = grand(m, n, "unf", min, max);
```

genera una matriz de `m` filas y `n` columnas de números aleatorios con distribución uniforme (equiprobable) entre los límites `min` y `max`. La sintaxis

```
x = grand(n, "prm", v);
```

produce `n` permutaciones aleatorias de las componentes del vector `v`. Si el vector es de la forma `1:m`, esto es muy útil para aleatorizar el orden de presentación de una serie de parámetros de un experimento.

La documentación (`doc grand`, anteriormente `help grand`) brinda detalles de la sintaxis para obtener otras distribuciones.

Dada una matriz, como las creadas con los comandos comentados anteriormente, su tamaño (cantidad de filas y columnas) puede conocerse invocando la función `size(a)`, mientras que la cantidad total de componentes puede obtenerse mediante `length(a)`.

7 Extracción

La obtención del valor de una componente específica de un vector o matriz se denomina *extracción*. El valor de la componente `n` de un vector `x` se invoca escribiendo `x(n)`, y si se desea crear un subvector `y` formado por las componentes `n`, `n+1`, ..., `m` basta escribir

```
y = x(n:m);
```

En el caso de matrices, la componente en la fila `n` y columna `m` se invoca con `x(n,m)`. También se puede invocar la fila `n` o la columna `m` completas escribiendo, respectivamente,

```
y = a(n, :);
y = a(:, m);
```

En cualquiera de los dos casos se puede reemplazar "`:`" por un subintervalo:

```
y = a(n, m1:m2);
y = a(n1:n2, m);
y = a(n1:n2, m1:m2);
```

También se pueden extraer componentes de una matriz mediante un único índice:

```
y = a(n);
```

En este caso se considera la matriz como un único vector columna cuyos elementos se leen primero por filas y luego por columnas. Así, si la matriz tiene `N` filas y `M` columnas, entonces `a((m-1)*N + n)` es equivalente a `a(n,m)`.

En algunos casos es interesante poder acceder o referir directamente al último elemento de una dimensión de un matriz. Si bien podría obtenerse el valor del índice determinando la longitud de cada dimensión mediante la función `size`, existe una variable `$` que representa el último índice. El valor que adopta en cada caso depende de la dimensión. Así, si

```
a = [1 2 3; 4 5 6];
```

entonces `a(1,$)` es la última columna de la fila 1, es decir 3, mientras que `a($,1)` es la última fila de la columna 1, es decir, 4. Utilizando la variante unidimensional del índice, `a($)` es el último elemento de la última columna, es decir, 6.

8 Operadores algebraicos

Los operadores de multiplicación, división y potenciación, `*`, `/`, `^`, actúan matricialmente cuando se aplican a matrices de dimensiones apropiadas. Si son precedidos por un punto, `.*`, `./`, `.^`, lo hacen por componente. En este caso las dimensiones deben coincidir. Por ejemplo, si:

```
a = [1 2; 3 4];
b = [2 5; 6 2];
```

resulta

```
c = a*b
 c = [2x2 double]
   14.   9.
   30.   23.

d = a.*b
 d = [2x2 double]
   2.    10.
   18.   8.
```

Las funciones como la raíz cuadrada (`sqrt`), la exponencial (`exp`), los logaritmos natural, decimal y binario (`log`, `log10`, `log2`), las trigonométricas (`sin`, `cos`, `tan`, `cotg`, `asin`, `acos`, `atan`, `acotg`) y las hiperbólicas (`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`) actúan por componentes.[4]

Una prima o apóstrofo "`'`" a la derecha del símbolo de una matriz actúa como operador de transposición. Así,

```
y = y';
```

[4] Para matrices cuadradas existen algunas de las correspondientes funciones matriciales agregando una m al final (`expm`, `logm`, `logm`, `sinm`, `cosm`, `tanm`, etc.)

convierte el vector columna en vector fila. En el caso de matrices con componentes complejas, además de la transposición las componentes pasan a ser conjugadas. Para obtener una transpuesta sin conjugación debe anteponerse un punto, `.'`.

9 Números complejos

Si bien todos los ejemplos anteriores involucraban números reales, Scilab trabaja en forma nativa con números complejos lo cual es muy conveniente no sólo en problemas de álgebra (por ejemplo las raíces de un polinomio en general son complejas) sino en el análisis espectral mediante transformada de Fourier. Para crear un número complejo basta utilizar el símbolo `%i`, que representa la unidad imaginaria. Así, por ejemplo podemos asignar el número `3 + 4i` a la variable `a` mediante el código

```
a = 3 + 4*%i
```

Una vez ejecutado se obtiene

```
a  =
   3. + 4.i
```

Observar que en la presentación del resultado el símbolo `%i` fue reemplazado por `i`, y que se eliminó el operador de multiplicación `*` (el punto es un punto decimal, no un punto de multiplicación, inexistente en la notación de Scilab). Esto es porque Scilab procura en general presentar los resultados en la forma más cercana a la escritura algebraica convencional.

Hay varias funciones para trabajar con números complejos que pueden ser de utilidad. Son ellas `real()`, `imag()` y `conj()`, que proporcionan la parte real, la parte imaginaria y el complejo conjugado. Asimismo, la función `abs()` permite obtener el módulo de un número complejo (o el valor absoluto de un real). Otra función útil es `phasemag()`, que obtiene la fase en ° y la magnitud en dB de un vector fila de números complejos y es útil para representar la respuesta en frecuencia de un sistema en magnitud y fase. En el ejemplo anterior tenemos

```
b = real(a)
 b =
   3.

c = imag(a)
 c =
   4.

d = conj(a)
 d =
   3. - 4.i

e = abs(a)
 e =
   5.
```

10 Arreglos

Un escalar es un único dato, un vector es un conjunto de *n* datos ordenado unidimensionalmente en forma de fila o columna y una matriz es un conjunto *n*×*m* datos ordenados bidimensionalmente en *n* filas y *m* columnas. Los arreglos (en inglés *array*) son una generalización de los escalares, vectores y matrices en dos posibles sentidos. En primer lugar, son un conjunto ordenado de datos en *k* dimensiones. Por ejemplo, si $k = 3$, sería un conjunto de datos organizado en *p* matrices (a veces llamadas *páginas* o *estratos*) de *n* filas y *m* columnas. Los escalares, vectores y matrices son arreglos de dimensión 0, 1 y 2 respectivamente.

En segundo lugar, los datos no necesariamente son numéricos, pudiendo ser otros tipos de datos que veremos más adelante. De hecho, es posible construir arreglos con cualquier tipo existente encerrando entre corchetes las componentes separadas por comas dentro de cada fila y puntos para pasar a la siguiente fila.

El único detalle a tener en cuenta es que los datos deben ser compatibles. Por ejemplo, es posible un arreglo de matrices siempre y cuando las matrices que están en una misma fila tengan igual número de filas, aunque en realidad en el arreglo final las matrices quedan concatenadas en una única matriz.

Puede generarse un arreglo multidimensional mediante las mismas funciones `ones()`, `zeros()`, `grand()` ya vistas, simplemente incrementando la cantidad de dimensiones. Por ejemplo, la siguiente instrucción crea un arreglo tridimensional de números aleatorios de `m` filas, `n` columnas y `p` matrices:

```
x = grand(m, n, p, "nor", Av, Sd);
```

También se puede crear un arreglo multidimensional asignando un valor a una componente con un índice mayor que 1 en la última dimensión:

```
x(1,1,1,2) = 4;
```

Si dicho índice es `1` y no se ha asignado previamente otro valor, la dimensión se reduce automáticamente a 2, siendo su tamaño `[1, 1]`.

Existen algunas funciones interesantes para el trabajo con arreglos. La primera es `matrix()`, que permite modificar el tamaño e incluso las dimensiones de una matriz o arreglo siempre y cuando el producto de los tamaños de todas las dimensiones sea igual en los arreglos de origen y destino. Así, por ejemplo, si `a` es un arreglo de tamaño 3×4×2, podremos convertirlo en otro arreglo `b` de tamaño 2×3×2×2 ya que ambos tienen 24 componentes. La sintaxis es

```
b = matrix(a, [2 3 2 2])
```

Para la conversión, comienza a leerse por la primera dimensión (filas) luego la segunda (columnas) y así sucesivamente, y lo mismo en lo que respecta a la constitución de la nueva matriz. Si se vuelve a aplicar con las dimensiones originales se recupera `a`. Por ejemplo, supongamos que obtenemos el siguiente arreglo de números aleatorios:

```
a = rand(2,3,2,"uniform")
a = [2x3x2 double]
(:,:,1)
   0.0437334   0.2639556   0.2806498
   0.4818509   0.4148104   0.1280058
```

```
(:,:,2)
   0.7783129   0.1121355   0.1531217
   0.211903    0.6856896   0.6970851
```

Entonces

```
matrix(a, [2 6])
ans = [2x6 double]
   0.0437334   0.2639556   0.2806498   0.7783129   0.1121355   0.1531217
   0.4818509   0.4148104   0.1280058   0.211903    0.6856896   0.6970851
```

La segunda función es `resize_matrix()`, que permite aumentar o reducir el tamaño de las dimensiones de una matriz. Así, partiendo de la misma matriz `a` anterior

```
c = resize_matrix(a, [3 5 2])
```

crea un arreglo donde cada fila original posee una columna adicional rellenada con 0 (*zero-padding*). En cambio,

```
c = resize_matrix(a, [3 3 2])
```

devuelve un arreglo en el que cada fila ha visto recortada su última columna. Esta función es útil, por ejemplo, cuando se quiere aplicar una FFT de longitud N a todas las filas de un arreglo cuyas longitudes difieren, ya sea por exceso o por defecto, de N.[5]

La tercera función es `permute()`, que permite permutar las dimensiones según una permutación de los números de dimensión. Así

```
d = permute(a, [1 3 2])
```

reinterpreta la que originalmnte era la tercera dimensión (páginas) como segunda dimensión (filas) y, recíprocamente, lo que antes eran filas ahora pasarán a ser páginas.

Por último tenemos `repmat()`. Esta función permite replicar un arreglo con una estructura dimensional especificada. Así, por ejemplo, si `a` es un vector columna

```
e = repmat(a, [1 3 2])
```

genera dos páginas en cada una de las cuales aparecen tres columnas iguales a `a`. Es útil, por ejemplo, cuando se requiere multiplicar todas las columnas de un arreglo por una función ventana distribuida en forma de columna.

11 Señales de audio digital

En el caso del procesamiento de señales de audio puede crearse un vector fila (matriz $1\times n$) a partir de un archivo de sonido en formato .WAV mediante el comando `wavread()`. Así, si `archivo` es la ruta completa del archivo entre comillas,

```
[x, Fs, Nbits] = wavread(archivo);
```

[5] Cuando N es mayor que la longitud de la fila, se rellena con ceros. Esto es útil cuando se requiere aumentar la resolución de análisis aunque la señal sea más corta. Y cuando N es menor, es necesario descartar algunos datos.

crea un vector `x` que contiene las muestras de audio normalizadas entre −1 y +1 (es decir que al mínimo valor binario posible se le asigna el valor −1 y al máximo, +1). Los parámetros `Fs` y `Nbits` proporcionan la tasa de muestreo en Hz y el número de bits (8, 16 o 24).

Si el archivo WAV es estereofónico o multicanal (2 o más canales de audio), en lugar de un vector fila se crea una matriz con una fila por cada canal.

También es posible, a la inversa, crear un archivo WAV a partir de un vector fila o una matriz con dos o más filas mediante el comando `wavwrite()`:

```
wavwrite(y, Fs, Nbits, archivo)
```

Las componentes de `y` deben estar comprendidas entre −1 y 1. El valor −1 se convierte en $-2^{Nbits-1}$ y el valor 1 en $2^{Nbits-1}-1$. En el caso más habitual en que `Nbits` = 16, estos valores corresponden respectivamente a −32768 y 32767.

La función `playsnd()` permite reproducir un sonido:

```
playsnd(x, Fs, Niter)
```

Antes de reproducir conviene empezar con el control de volumen bajo para prevenir la emisión accidental de sonidos muy intensos y potencialmente perjudiciales. `Niter` corresponde al número de veces que se reproducirá. Normalmente es 1, pero si no se especifica y luego se reproduce otro sonido, la reproducción del primero se interrumpirá.

12 Polinomios

Scilab puede manipular en forma simbólica polinomios y funciones racionales de una variable. Así, la función `poly()` define un polinomio a partir de sus raíces

```
p = poly([z1, z2], "x", "roots");
```

también, en forma simplificada,

```
p = poly([z1, z2], "x");
```

o bien a partir de sus coeficientes

```
p = poly([a0, a1, a2], "x", "coeff");
```

Se puede operar con polinomios como si fueran números. Un cociente de polinomios proporciona una función racional. Scilab reduce numerador y denominador removiendo factores comunes, siempre y cuando sean exactamente iguales. Una forma cómoda de generar polinomios es crear un monomio elemental

```
x = poly(0, "x");
```

equivalente a la variable x, y luego operar. Por ejemplo,

```
p = 1 - 3*x + 3*x^2 + x^3;
```

obtiene el polinomio $1 - 3x + 3x^2 + x^3$, representado como

```
p  =
  1 -3x +3x² +x³
```

Otra forma es usar una de dos variables polinómicas que ya vienen predefinidas en Scilab, `%s` y `%z`, destinadas en principio a representar la variable *s* de la transformada de Laplace y la variable *z* de la transformada *z*. Por ejemplo:

```
d = 1 + 2*%s + 2*%s^2 + %s^3;
```

Para evaluar un polinomio (o función racional) `p` en un valor del argumento se aplica la función `horner()`.[6] Así,

```
q = horner(p, 5);
```

evalúa el polinomio (o función racional) `p` en `x = 5`. La función `horner` también admite una matriz como segundo argumento, en cuyo caso el polinomio (o función racional) se evalúa en cada componente de la matriz. También es aplicable a un argumento que a su vez es otro polinomio (o función racional).

La función `roots()` obtiene numéricamente las raíces de un polinomio:

```
z = roots(p);
```

El resultado es un vector columna cuyas componentes son las raíces. Desde luego, las raíces pueden ser reales o complejas. En el caso del ejemplo anterior se obtiene:

```
z  =
  -3.8473221 + 0.i
   0.4236611 + 0.283606i
   0.4236611 - 0.283606i
```

La función `coeff()` obtiene un vector fila cuyas componentes son los coeficientes del polinomio en orden creciente de potencias:

```
c = coeff(p);
```

En el caso del ejemplo anterior resulta

```
c  =
   1.  -3.   3.   1.
```

Es posible construir vectores, matrices o, más generalmente, arreglos de polinomios, ya que los polinomios son tipos de datos compatibles entre sí. Así,

```
p = [1 + x, 1 + 2*x + x^2; 1- x, 1 - 2*x + x^2]
```

crea el siguiente vector de polinomios:

---

[6] El nombre de la función honra a William George Horner, quien publicó el algoritmo en 1819, aunque el mismo se ha usado desde varios siglos antes, por ejemplo por el matemático chino del siglo XIII-XIV, Zhu Shijie. El algoritmo consiste en: $a_o + a_1x + ... + a_nx^n = (((a_nx + a_{n-1})x + a_{n-2})x ...)x + a_o$, reduciendo considerablemente el número de multiplicaciones (Wikipedia, 2019).

```
p  =
  1 +x  1 +2x +x²
  1 -x  1 -2x +x²
```

13 Estructuras condicionales y de control

Igual que en todos los lenguajes de programación, Scilab permite escribir estructuras condicionales y de control. La estructura `if` permite realizar o no una serie de operaciones según que cierta condición sea verdadera o falsa. Por ejemplo:

```
if i < 5 then
   a = 1;
elseif i < 10 then
   a = 2;
else
   a = 3;
end
```

La estructura `for` permite repetir una cantidad especificada de veces ciertas instrucciones. Por ejemplo

```
a = [];
for i=1:5
   a = [a, i*ones(1,i)];
end
```

crea el vector `a = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]`. La estructura `while` repite una serie de operaciones mientras una expresión lógica dada resulte verdadera. Las instrucciones

```
i = 6;
a = [];
while (5 < i) & (i <  10)
   a = [a, i];
   i = i + 1;
end
```

crean el vector [6, 7, 8, 9]. Si el valor inicial de `i` hubiera sido ≤ 5 o ≥ 10, el lazo no se habría ejecutado.

Por último, la estructura `select` permite realizar diversas operaciones según el valor de una variable. En la siguiente estructura,

```
select a
case 1
   b = 1;
case 2
   b = 3;
case 3
   b = 0;
end
```

si `a = 2` se asigna a `b` el valor `3`.

14 Sistemas lineales

Se puede definir en Scilab la función de transferencia[7] de un sistema lineal `H` a partir de dos polinomios `p` (numerador) y `q` (denominador) mediante la función `syslin()`:

```
H = syslin("d", p, q);
```

o bien

```
H = syslin("d", p/q);
```

donde `"d"` indica que se trata de una función de transferencia en *tiempo discreto*. Para funciones de transferencia en *tiempo continuo* se utilizaría `"c"`.

Una función de transferencia en tiempo discreto `H` puede aplicarse a una señal discreta `x` mediante la función `flts()`, obteniéndose la correspondiente respuesta del sistema

```
y = flts(x, H);
```

También es posible incluir una condición inicial:

```
y = flts(x, H, [x0; y0]);
```

donde `x0` es un vector fila que contiene `nd` valores previos de `x` e `y0` es un vector fila que contiene `nd` valores previos de `y`, siendo `nd` el grado del denominador de la función de transferencia. Si la señal de entrada comienza recién en la muestra 1, entonces `x0` es un vector con `nd` componentes nulas.

Alternativamente puede utilizarse la función `filter()`, con la siguiente sintaxis:

```
y = filter(A, B, x);
```

donde `A` y `B` son vectores de los coeficientes del numerador y el denominador de la función de transferencia discreta (en transformada *Z*), *en orden decreciente de potencias*.[8] Si se desea especificar condiciones iniciales, puede utilizarse

```
[y, zf] = filter(A, B, x, zi);
```

donde `zi` es un vector de condiciones iniciales. El argumento opcional de salida `zf` contiene las condiciones finales. Las condiciones iniciales y finales son útiles cuando se desea filtrar señales muy largas procesándolas por segmentos. Supongamos, para ilustrar, una señal `x` formada por concatenación de dos señales más cortas `x1` y `x2`:

```
x = [x1, x2];
```

Podemos filtrar la primera parte (en este caso suponiendo condiciones iniciales nulas):

---

[7] La función de transferencia de un sistema dinámico lineal en tiempo continuo es el cociente entre las transformadas de Laplace de la señal de salida y de la señal de entrada. En el caso de un sistema en tiempo discreto, es el cociente entre las respectivas transformadas *Z*. Estas funciones de transferencia son, en general, funciones racionales.

[8] Para más detalles ver la sección siguiente

```
[y1, zf1] = filter(A, B, x1);
```

Para la segunda parte utilizaremos como condiciones iniciales las condiciones finales de la primera:

```
y2 = filter(A, B, x2, zf1);
```

Finalmente, podemos reconstruir la respuesta completa del filtro concatenando las respuestas parciales:

```
y = [y1, y2];
```

Esto garantiza la continuidad de la señal y sus derivadas relevantes.

15 Filtros

Scilab ofrece la función `iir()` para el diseño de filtros de respuesta al impulso infinita (IIR) en tiempo discreto correspondientes a ciertas aproximaciones clásicas, tales como Butterworth y Chebyshev. Admite dos sintaxis:

```
hz = iir(n, ftype, fdesign, frq, delta)
[p, z, g] = iir(n, ftype, fdesign, frq, delta)
```

donde:

`hz` es una función de transferencia discreta, definida por una lista[9] con los siguientes elementos:
- `hz(1)` descripción de la estructura: `!r num den dt !`
- `hz(2)` polinomio numerador
- `hz(3)` polinomio denominador
- `hz(3)` tipo de filtro (`"c"`: continuo; `"d"`: discreto)

`p` vector columna de los polos de la transferencia discreta;

`z` vector columna de los ceros de la transferencia discreta;

`g` ganancia requerida para tener ganancia 1 en la(s) banda(s) de paso;

`n` orden del filtro;

`ftype` es el tipo de plantilla del filtro:

‘`lp`’:	pasabajos	‘`hp`’:	pasaaltos
‘`bp`’:	pasabanda	‘`sp`’:	rechazabanda

`fdesign` es el tipo de aproximación:
- ‘`butt`’: Butterworth (máximamente plano)
- ‘`cheb1`’: Chebychev tipo I (ripple constante en la banda de paso)
- ‘`cheb2`’: Chebychev tipo II (ripple constante en la banda de atenuación)
- ‘`ellip`’: Elíptico o Cauer (ripples constantes en ambas bandas)

`frq` es un vector de 2 componentes: frecuencia inferior de corte y frecuencia superior de corte, ambas normalizadas: $0 \leq F \leq 0{,}5$, donde 0,5 es equivalente a $F_s/2$.
NOTA: Esta normalización es necesaria dado que el filtro discreto no está asociado a ninguna tasa de muestreo en particular. Para un filtro pasabanda de frecuencias inferior y superior f_i y f_s este argumento será `[fi/Fs, fs/Fs];`

[9] Una lista es un tipo de dato de Scilab que permite agrupar una serie de campos heterogéneos. Ver secciones 24 y 27.

`delta` es un vector de 2 componentes, δ_1 y δ_2, aplicables a los filtros de Chebyshev y Cauer, según corresponda, donde δ_1 es el ripple máximo en las bandas de paso y δ_2 es el ripple máximo en las bandas de atenuación. Notar que no están en decibeles, sino en escala lineal entre 0 y 1;

Una vez obtenido el filtro se puede proceder a filtrar la señal mediante `filter()`, utilizando la sintaxis

```
[y, zf] = filter(num, den, x [,zi])
```

donde:

`y` es la salida filtrada;

`zf` es el estado o condición final, consistente en los últimos máx{N, M} valores de la respuesta. Es útil cuando se va a realizar un filtrado por bloques, por ejemplo para evitar problemas de saturación de memoria;

`num` es el numerador del filtro, expresado como coeficientes en orden decreciente de potencia o bien como polinomio. En esta última variante puede utilizarse la variable `hz(2)` obtenida con la primera sintaxis de la función `iir()`;

`den` es el denominador del filtro, expresado como coeficientes en orden decreciente de potencia o bien como polinomio. En esta última variante puede utilizarse la variable `hz(3)` obtenida con la primera sintaxis de la función `iir()`;

`x` es la señal de entrada, expresada como vector fila;

`zi` es el estado o condición inicial, consistente en los máx(N, M) valores de la respuesta previos al primer valor. Puede utilizarse haciéndolo igual a `zf` del filtrado de un bloque inmediatamente anterior.

16 Transformada rápida de Fourier FFT

Puede realizarse el análisis de espectro de una señal mediante el uso de la *transformada discreta de Fourier*, implementada mediante el algoritmo conocido como *Transformada Rápida de Fourier*, FFT. Así, si `x` es un vector,

```
y = fft (x, -1) ;
```

es su transformada discreta de Fourier. El argumento `-1` indica el signo del exponente de las exponenciales que se utilizan en la transformada. Si en su lugar se utiliza `1` se obtiene la transformada rápida inversa de Fourier:

```
x = fft(y, 1);
```

El signo puede omitirse, en cuyo caso el valor por defecto es `-1`, es decir que

```
y = fft(x);
```

también permite calcular la FFT. Análogamente, existe la función `ifft()`, que calcula directamente la transformada discreta inversa de Fourier

```
x = ifft(y);
```

Si se desea restringir el número de puntos a los `N` primeros, por ejemplo, basta poner

```
y = fft(x(1:N), -1);
```

También es posible realizar simultáneamente las transformadas discretas de `M` señales de longitud `N` presentadas como una matriz `X` de `N` filas y `M` columnas donde cada columna es una señal. Para ello puede utilizarse la sintaxis

```
Y = fft(X, -1, 1);
```

El tercer argumento representa la dimensión a lo largo de la cual se realiza la transformada. Como dentro de una columna el índice que varía es el correspondiente a las filas, la dimensión es 1 En el caso en que las señales estuvieran distribuidas como `M` filas de `N` columnas, la dimensión a lo largo de la que varía cada señal sería la 2, es decir,

```
Y = fft(X, -1, 2);
```

17 Gráficas

Es posible graficar una señal mediante el comando `plot2d()`. Aunque tiene muchas opciones, que pueden consultarse mediante `doc plot2d`, se obtiene una gráfica sencilla mediante

```
plot2d(x, y)
```

donde `x` e `y` son vectores de igual longitud que contienen los valores de la abscisa y la ordenada. La función `plot2d()` produce gráficas segmento-lineales uniendo los puntos con segmentos de recta, pero a partir de varias decenas de puntos, la imagen aparece como una curva suave. El siguiente código (donde `%pi` es la constante π):

```
x = 0:0.01:4*%pi;
y = sin(x);
plot2d(x,y)
```

genera 2 ciclos de la función seno y traza la gráfica que se muestra en la figura 12.

Si `y1` e `y2` son vectores columna de igual longitud que `x`, se pueden graficar simultáneamente `y1` e `y2` versus `x` mediante

```
plot2d(x, [y1, y2])
```

En tal caso cada variable se grafica con un color diferente. Si con el `x` del ejemplo ejecutamos el siguiente código

```
y1 = sin(x);
y2 = cos(x);
plot2d(x,[y1', y2'])
```

obtendremos la gráfica de la figura 13.

Alternativamente puede utilizarse la función `plot`, que tiene mayor compatibilidad con Matlab y Octave. A diferencia de la anterior, por defecto presenta la gráfica en un recuadro en lugar de utilizar ejes. Por ejemplo

```
plot(x,[y1', y2'])
```

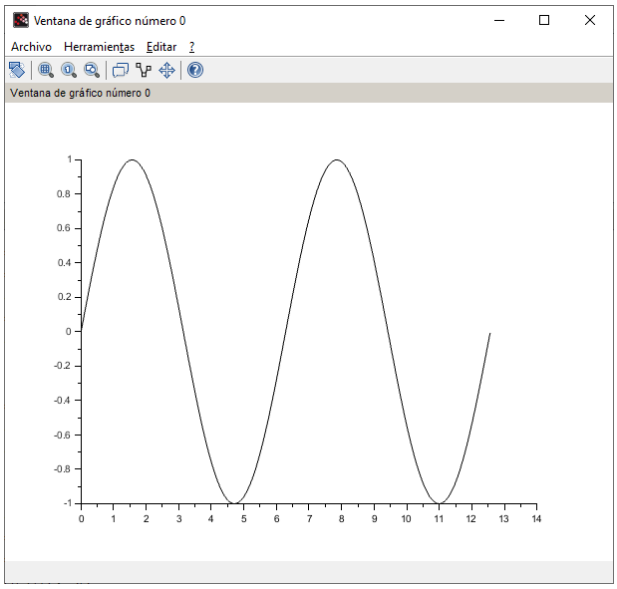

Figura 12. Gráfica de la función seno obtenida mediante `plot2d`.

genera la gráfica de la figura 14.

También se pueden hacer varias gráficas superpuestas, aun con diferentes longitudes de abscisa, simplemente escribiendo varias instrucciones `plot2d` sucesivas:

```
plot2d(x1, y1)
plot2d(x2, y2)
```

ya que una nueva gráfica no borra la anterior. Sin embargo, en este caso se usa el mismo color. El color puede modificarse agregando un tercer argumento que indica el índice de color en el mapa de colores de la figura. Por ejemplo

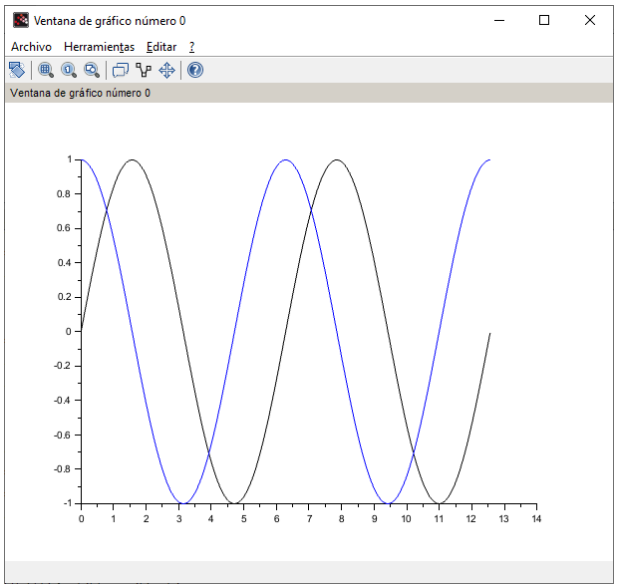

Figura 13. Gráfica de las funciones seno y coseno obtenida mediante `plot2d`. Por defecto se representa en un sistema de ejes.

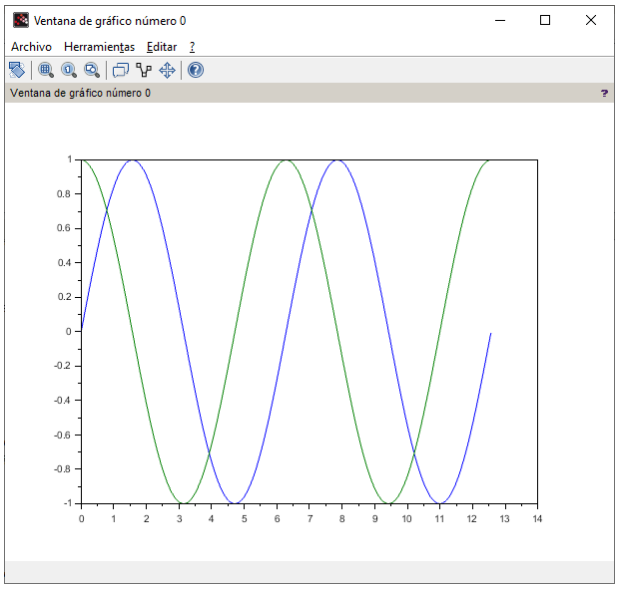

Figura 14. Gráfica de las funciones seno y coseno obtenida mediante `plot`. Por defecto se representa en un recuadro

```
plot2d(x1, y1, 2)
plot2d(x2, y2, 5)
```

grafica `y1` en color azul e `y2` en color rojo. Para conocer el mapa de colores asociado a la figura actual hay una herramienta interactiva, `getcolor()`, que abre una ventana de diálogo donde se muestran los colores del mapa (figura 15). Haciendo clic en cualquiera de ellos se muestra abajo a la izquierda el índice correspondiente junto con la fórmula RGB (valores de rojo, verde y azul) correspondiente .
NOTA: La consola queda congelada hasta que se cierre la ventana haciendo clic en Ok o en la cruz.

En los casos anteriores se utilizó por defecto una escala lineal en ambos ejes. Es posible utilizar escalas logarítmicas en uno o ambos ejes con la siguiente sintaxis:

```
plot2d("ln", x, y)
plot2d("nl", x, y)
plot2d("ll", x, y)
```

donde el primer argumento indica en forma abreviada el tipo de escala de los ejes *x* e *y*, donde `"l"` significa *logarítmico* y `"n"` significa *normal* (lineal). Por ejemplo,

```
plot2d("ln", f, 20*log10(abs(H)))
```

dibuja una respuesta en frecuencia `H` en función de la frecuencia `f` en un diagrama en el cual el eje de frecuencias el logarítmico y el eje de amplitudes es lineal. Notar que en realidad el eje de amplitudes está representando una versión en decibeles, que ya es logarítmica, pero la escala graduada en decibeles siempre se representa linealmente.

Se puede agregar un título a la gráfica, así como indicar qué variable hay en cada eje coordenado, por ejemplo

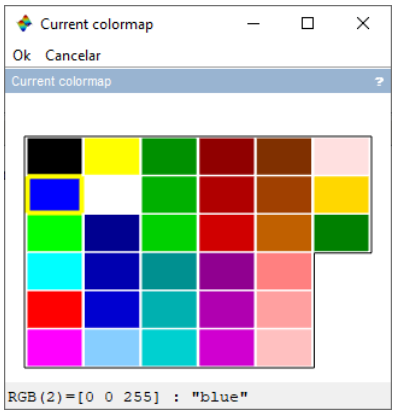

Figura 15. Mapa de colores obtenido mediante la herramienta interactiva, `getcolor()`. Haciendo clic sobre un color se muestra la composición RGB del mismo.

```
title("Espectro del ruido rosa");
xlabel("f   (Hz)");
ylabel("|Y(f)|   (Hz)")
```

En caso de haber varias gráficas en un mismo sistema coordenado, es útil poder brindar una referencia que indique a qué corresponde cada curva. Ello puede hacerse con la función `legend()`. Por ejemplo:

```
legend("y1", "y2")
```

Aunque no incursionaremos en el tema, estos comandos admiten el uso de código $\LaTeX$, que posibilita una presentación muy sofisticada, incluyendo todo tipo de símbolos matemáticos. En caso de interés se puede profundizar en cualquier introducción a $\LaTeX$ (LaTeX, 2022-08-14; Colaboradores de Wikipedia, 2022-07-14).

En proyectos de mediana complejidad pueden ser necesarias diferentes gráficas. Se pueden abrir nuevas ventanas gráficas, para lo cual se utilizan sucesivas instancias de

```
scf(i);
```

donde `scf` es acrónimo en inglés de *set current figure* (*establecer la figura actual*, es decir aquella en la cual se trazará la o las siguientes gráficas) e `i` es el número de identificación de la ventana, cuyos valores posibles son 0, 1, 2, etc. Si no se ha abierto ninguna ventana antes de la primera invocación de `plot2d()` o `plot()` Scilab asigna por defecto la ventana 0.

En algunos casos se necesita borrar el contenido de una figura sin cerrar la ventana que la contiene. Para ello se utiliza

```
clf(i);
```

Si, en cambio, se desea eliminar completamente una ventana gráfica, se utiliza

```
close(i);
```

Para borrar todas las ventanas (lo cual puede ser útil para volver a una situación de origen sin correr el riesgo de trazar una gráfica sobre una anterior), se puede usar el comando

```
close(winsid())
```

donde `winsid()` es un vector que contiene los identificadores numéricos de todas las ventanas abiertas.

Dentro de una misma ventana gráfica pueden incluirse varios gráficos en distintos sistemas coordenados, por ejemplo para mostrar la señal de entrada y la señal de salida de un filtro. Ello se logra mediante el comando `subplot()`. Así,

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

divide la ventana en `m` filas y `n` columnas de gráficas y ubica la próxima gráfica a trazar en la celda `p`-ésima, contando primero por columnas (de izquierda a derecha) y después por filas (de arriba a abajo), donde `p = 1, ..., m*n`. Después de cada comando `subplot()` se procede a graficar como en los casos anteriores.

A modo de ejemplo, el siguiente código

```
x = [0:0.01:2*%pi];
scf(1);
subplot(2, 2, 1)
plot(x, sin(x))
subplot(2, 2, 2)
plot(x, sin(2*x))
subplot(2, 2, 3)
plot(x, sin(3*x))
subplot(2, 2, 4)
plot(x, sin(4*x))
```

grafica una senoide y sus armónicos 2, 3 y 4 en una misma ventana, como se muestra en la figura 16.

18 Gráficas 3D

Aunque no tan habituales como las gráficas 2D, en ocasiones se desea graficar una variable que depende funcionalmente de dos variables independientes,

$$z = g(x, y)$$

por ejemplo cuando una de ellas es una variable principal, como el tiempo o la frecuencia, y la otra es un parámetro que afecta la forma en que la variable dependiente depende de la primera. Hay varios comandos que permiten este tipo de gráficas, como `plot3d()`, `mesh()`, `surf()` y `Sgrayplot()`. Se trata de funciones con una gran versatilidad, pero al mismo tiempo cierta dificultad de uso.

La función `plot3d()` es una extensión de la `plot2d()` que proporciona, en principio, una superficie de un color uniforme donde la curvatura de la superficie se muestra mediante una malla basada en una grilla en los ejes x e y. La función `mesh()`

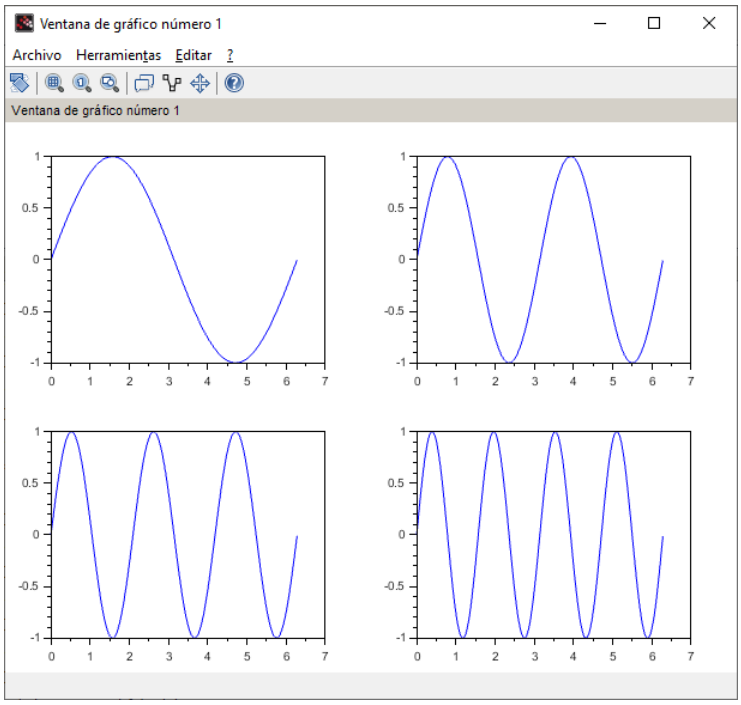

Figura 16. Uso del comando subplot para trazar diferentes gráficas en una misma figura.

proporciona la malla sola, sin ningún color. La función `surf()` dibuja una superficie, y eligiendo adecuadamente algunos parámetros permite una presentación muy elegante. Finalmente, `Sgrayplot()` dibuja la proyección de la superficie sobre el plano *x*-*y* representando con una escala de colores los valores de la variable dependiente.

Veamos un ejemplo de uso de `surf()`. Antes de invocar la función propiamente dicha es necesario establecer un mapa de colores. Un mapa de colores es un conjunto ordenado de colores de tal manera que el primer color representa el mínimo de una variable y el último representa su máximo. Operativamente es una matriz de tres columnas que representan, en números reales entre 0 y 1, las intensidades de los colores rojo, verde y azul (RGB). Cada fila representa un color combinado. Por ejemplo, [1, 0, 0] representa rojo puro de máxima intensidad ( ), [0, 0.5, 0] representa un verde de mediana intensidad ( ) y [1, 1, 0] representa un amarillo intenso ( ). La cantidad de filas es seleccionable y determina la suavidad de la transición entre colores.

Podría construirse un mapa de colores arbitrario, para lo cual basta cualquier matriz de tres columnas con las características mencionadas. Existe, sin embargo una serie de mapas estéticamente logrados que vienen predefinidos en Scilab y pueden encontrarse mediante `doc colormaps`. En la figura 17 se muestran algunos ejemplos. Elegida una combinación, por ejemplo `jet`, hay una función `jet()` cuyo argumento es la cantidad de colores del mapa. Entonces

```
cmap = jet(100);
```

asigna a la variable `cmap` (el nombre de esta variable es arbitrario) un mapa de 100 colores distribuidos en forma gradual siguiendo el patrón de la combinación `jet` (comienza en azul oscuro se va aclarando, pasa por amarillo y luego vira a rojo, hasta llegar a un rojo oscuro).

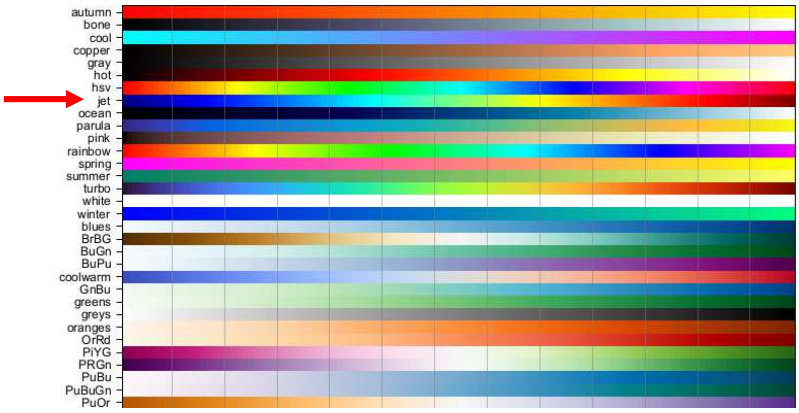

Figura 17. Ejemplos de mapas de colores de Scilab. Se resalta el `jet`.

Una vez creado el mapa de colores, se asigna este mapa a la figura actual, para lo cual se invoca y asigna la propiedad color_map de la figura de la siguiente forma:[10]

```
gcf().color_map = cmap;
```

Luego se grafica mediante

```
surf(x, y, z, "facecol", "interp")
```

El tamaño de `z` debe ser compatible con los de `x` e `y`. Los últimos dos argumentos determinan lo que se denomina una *propiedad global*, dada por un nombre de parámetro, en este caso `"facecol"`, seguido por su valor, en este caso `"interp"` (para más detalles escribir `doc GlobalProperty`). Lo que se está haciendo con esto es especificar que el color que corresponde a un determinado valor de `z` se interpola entre los colores anterior y posterior más cercanos en el mapa.

Por defecto la función `surf()`, además de la superficie coloreada, también superpone una malla. Cuando la grilla determinada por el contenido de `x` e `y` es muy densa, esta malla se vuelve muy invasiva y termina cubriendo los colores, resultando una superficie totalmente negra. Para evitarlo hace falta un último comando que asigna 0 al espesor de la línea:[11]

```
gce().thickness = 0;
```

Para nuestro ejemplo supondremos la función

$$z = 1 - \left(\frac{x-2}{2}\right)^2 - \left(\frac{y-4}{4}\right)^2$$

que corresponde a un paraboloide elíptico invertido. El código siguiente calcula los valores de la función en una retícula de puntos

---

[10] La función `gcf()`, es decir *get current figure*, obtiene el denominado *handle* de la figura, una variable tipo estructura que contiene propiedades configurables de la figura, en este caso `color_map` (mapa de colores).

[11] `gce()`, es decir *get current entity*, obtiene el *handle* de la superficie, que contiene propiedades de la superficie que también pueden configurarse. Una de ellas es *thickness* (epesor).

```
x = 0:0.05:4;
y = 0:0.05:8;
for i=1:length(y)
   z(i,:) = 1 - (x-2).^2/4 - (y(i)-4)^2/16;
end
scf(1);
cmap = jet(100);
gcf().color_map = cmap;
surf(x, y, z, "facecol", "interp")
gce().thickness = 0;
```

En la figura 18 se muestra el resultado obtenido al ejecutarlo.

NOTA: Por alguna razón no especificada, `surf` representa $z(y, x)$ en lugar de $z(x, y)$, por lo cual hay que tener cuidado con el orden de las variables para evitar errores. Por ese motivo en el código anterior se invirtieron.

En este tipo de gráficos con colores representando valores puede ser interesante proporcionar una referencia. Ello puede lograrse mediante

```
colorbar;
```

que agrega una barra de referencia con la escala numérica asociada a los colores.

Esta superficie queda enmarcada en una caja transparente. La orientación de los ejes no siempre es la más estética o la que muestra los detalles en los que estamos interesados. Haciendo clic derecho (botón secundario) en la figura y arrastrando, sin soltar, horizontal o verticalmente se puede rotar la superficie en azimut y elevación para

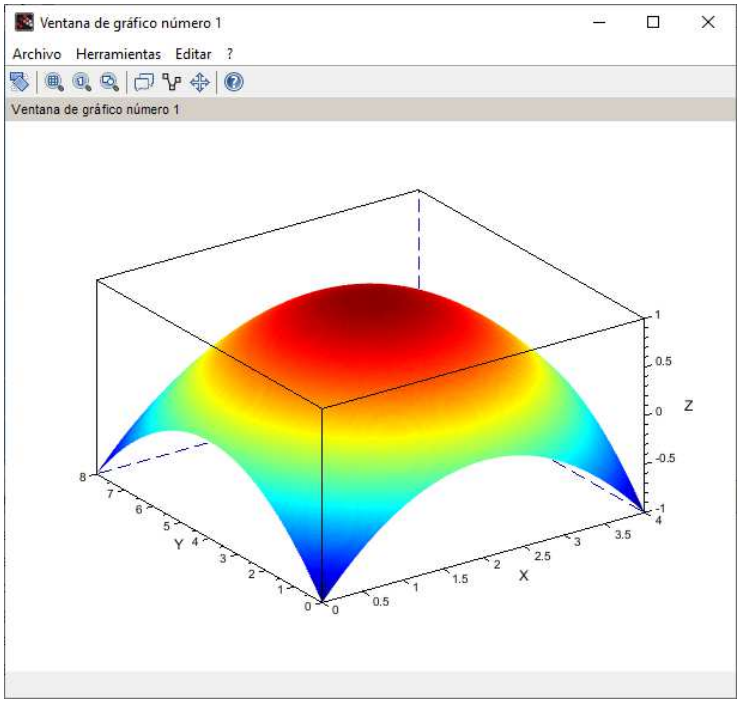

Figura 18. Ejemplo de gráfica 3D: un paraboloide elíptico.

obtener una vista desde una orientación diferente de la obtenida por defecto. Estos ángulos también se pueden especificar mediante, por ejemplo,

```
gca().rotation_angles = [70, 25];
```

El uso de la función `Sgrayplot()` es similar, salvo que, al no dibujar por defecto una grilla, no hace falta especificar un espesor 0. Esta forma de representación es útil para el caso de espectrogramas (representación en función del tiempo y de la frecuencia) y funciones de transferencia relativas a la cabeza (representación en función de la frecuencia y el ángulo de azimut o de elevación).

19 Variables y operadores lógicos

Scilab permite trabajar con variables lógicas o booleanas (*boolean*), las cuales poseen uno de dos valores posibles: `T` (verdadero) y `F` (falso). Hay tres formas básicas de obtener variables booleanas. La primera es la asignación, para lo cual existen dos valores predefinidos `%t` (o `%T`, ambas son válidas) y `%f` (o `%F`). Así, por ejemplo. la instrucción

```
a = %f;
```

asigna a la variable `a` el valor `F`. Notar que en Scilab se utiliza `%f` o `%F` para invocar o asignar el valor falso, pero como resultado se presenta `F` (sin `%`).

La segunda forma es apelando a los denominados *operadores relacionales*, que efectúan distintos tipos de comparaciones entre dos variables. Son ellos la igualdad, representada mediante dos signos igual, == y las desigualdades, <, >, <=, >=, <>. Así,

```
a = 3<5;
```

asigna a la variable `a` el resultado de la comparación `3<5`, que es verdadera (`T`). En cambio

```
a = 3==5;
```

resulta falsa (`F`). Notar que no hace falta colocar paréntesis en el segundo miembro dado que no hay forma posible de confusión, ya que = es un operador de asignación, no de relación.

La tercera forma es aplicar operadores entre valores lógicos. Estos son `&` (*and*, *y*), `|` (*or*, *o*) y ~ (*not*, no). El operador *and* entre dos variables da como resultado `T` si ambas variables son `T` y `F` en cualquier otro caso. El operador *or* arroja un valor `F` si ambas variables son `F` y `T` en cualquier otro caso. Y el operador ~ cambia un valor `F` por uno `T` y viceversa. Por ejemplo:

```
a = (3<4) & (6>8);
```

da como resultado `F` pues `6>8` es `F`. En este caso los paréntesis son necesarios ya que de otra manera la expresión se va calculando de izquierda a derecha, es decir

```
a = ((3<4) & 6)>8;
```

Para interpretar correctamente el segundo miembro, tengamos en cuenta que para los operadores lógicos cualquier real distinto de 0 es interpretado como `T` por lo tanto el resultado del `&` es `T`. Llegado este punto estamos intentando comparar `T` con `8`, lo cual no es posible ya que el operador > sólo está definido para números, no valores lógicos, por lo cual se obtiene un mensaje de error.

Además de los métodos anteriores, existen ciertas funciones cuyos argumentos de salida son variables lógicas y que son útiles para realizar ciertas comprobaciones, por ejemplo `isscalar()`, que aplicada a una matriz produce un resultado `T` si se reduce a un escalar (es decir tiene una fila y una columna, o `isreal()`, que aplicado a una matriz devuelve `T` si todas sus componentes son reales. También pertenecen a esta categoría otras funciones como `isvector()`, `isrow()`, `iscolumn()`, etc.

Igual que en el caso de los números reales o complejos, también son posibles vectores, matrices y, en general, arreglos booleanos. Por ejemplo,

```
a = [1 2 3; 4 5 6] > 4
```

da como resultado

```
 a  =
  F  F  F
  F  T  T
```

20 Cadenas

En muchas instancias es necesario poder utilizar variables que contienen caracteres alfanuméricos, por ejemplo nombres u otros textos. Esto se logra mediante un tipo especial de datos denominado *cadena de caracteres* o simplemente *cadena* (en inglés, *string*). Puede crearse una cadena asignando un texto entre comillas simples o dobles, pero no pueden mezclarse en una misma cadena. Así, por ejemplo,

```
a = "esto es una cadena"
```

asigna a la variable `a` el texto

```
esto es una cadena
```

Son admisibles todos los caracteres del juego ASCII extendido (es decir los que pueden introducirse mediante el teclado normal más los que se introducen mediante la tecla Alt izquierda y el teclado numérico). Para incluir comillas simples o dobles es necesario repetirlas dentro del espacio comprendido entre las comillas que definen la cadena, así por ejemplo

```
a = """esto es una cadena"""
```

asigna la cadena

```
"esto es una cadena"
```

Nótese que las versiones actuales de Scilab visualizan el contenido de una variable de tipo cadena entre comillas, por lo cual la variable anterior se mostrará como

```
 ""esto es una cadena""
```

Pueden crearse también vectores o matrices de cadenas en forma similar a las numéricas, las cuales no necesariamente deben ser de la misma longitud. Por ejemplo

```
a = ["cadena", ""; "cadena larga", "cadena muy larga"]
```

produce el resultado

```
a  =
 "cadena"         ""
 "cadena larga"   "cadena muy larga"
```

Obsérvese que una de las cadenas es la cadena vacía (no contiene ningún carácter).

Es posible invocar las cadenas integrantes de un vector o matriz de cadenas colocando las coordenadas de la componente deseada. Así,

```
a(2,2)
```

devuelve

```
ans  =
 "cadena muy larga"
```

Existen varias funciones y operadores aplicables a las cadenas. La función `strcat()` permite concatenar en una sola cadena las componentes de un vector o matriz de cadenas. Por ejemplo,

```
a = ["esta será ", "una cadena"];
b = strcat(a)
```

produce

```
b  =
 "esta será una cadena"
```

También es posible utilizar el operador + para concatenar dos o más cadenas:

```
a = "esta será " + "una larga " + "cadena"
```

proporciona

```
 a =
  "esta será una larga cadena"
```

Notar que en el caso en que se desee que aparezcan espacios entre las cadenas a concatenar, es necesario incluirlos al final de las mismas.

Otras funciones útiles son `size()`, que devuelve la cantidad de filas y columnas de una matriz de cadenas, y `length()`, que devuelve una matriz de igual tamaño cuyas componentes contienen la cantidad de caracteres de las correspondientes cadenas. Así, si

```
a = ["uno" "dos" "tres"; "cuatro" "cinco" "seis"];
```

entonces

```
b = size(a)
b = [1x2 double]
   2.   3.

c = length(a),
 c = [2x3 double]
   3.   3.   4.
   6.   5.   4.
```

Hay algunas funciones de comprobación. Por ejemplo `isnum`, aplicado a una cadena, devuelve `T` si la cadena representa un número (escrito con cifras):

```
isnum("seis")
 ans =
  F
isnum("6.5")
 ans =
  T
```

Notar que en el primer caso, si bien el *significado* de la cadena alfabética es un número, en realidad como cadena no es un número, mientras que en el segundo caso sí.

La función `isdigit()`, por su parte, devuelve un vector booleano con tantas componentes como caracteres indicando si el carácter homólogo es o no un dígito. Otras: `isletter()`, `isalphanum()`, `isascii()`.

Las cadenas admiten también la búsqueda y extracción de subcadenas mediante las funciones `strindex()` y `part()`. Así, si `cadena` y `subcadena` son una cadena larga y una corta, entonces la sintaxis básica

```
ind = strindex(cadena, subcadena)
```

proporciona las posiciones en la cadena larga del comienzo de cada aparición de la cadena corta. Por ejemplo,

```
ind = strindex("uno, dos, tres, dos, cuatro", "dos")
```

da como resultado

```
 ind  =
   6.   17.
```

En efecto, a partir del carácter ubicado en la posición 6 aparece la cadena `"dos"`, y lo mismo sucede a partir del carácter ubicado en la posición 17. Por otro lado, si `n` es un vector de índices, mediante

```
sub = part(cadena, n)
```

se pueden recuperar los caracteres ubicados en las posiciones indicadas por `n`. Por ejemplo

```
sub = part("uno, dos, tres, dos, cuatro", 6:8)
```

permite corroborar el resultado obtenido anteriormente mediante `strindex`:

```
sub  =
 "dos"
```

Por último, la función `ascii()` permite convertir una cadena en un vector que contiene los códigos ASCII de los caracteres de la cadena y, recíprocamente, convierte un vector de códigos ASCII en una cadena. Así,

```
a = ascii("cadena 1")
```

da como resultado

```
a  =
  99.   97.   100.   101.   110.   97.   32.   49.
```

en tanto que

```
b = ascii(a)
```

devuelve

```
ans  =
  "cadena 1"
```

21 Funciones

Las funciones proporcionadas por Scilab cubren un amplio espectro de aplicaciones. Sin embargo, frecuentemente surge la necesidad de introducir funciones específicas originalmente no implementadas. Es posible definir nuevas funciones del usuario mediante la sintaxis

```
function [y1, ..., yn] = nombre(x1, ..., xm)
   instrucciones
endfunction
```

donde `x1, ..., xm` son las *variables* o *argumentos de entrada* e `y1, ..., yn` son las *variables* o *argumentos de salida*, `nombre` es el nombre de la función e `instrucciones` es una secuencia de instrucciones en las que se definen los valores de `y1, ..., yn`.

La sintaxis anterior puede ingresarse directamente en línea, es decir en la ventana principal de Scilab (consola o espacio de trabajo) o en cualquier editor de texto (por ejemplo SciNotes, el editor interno de Scilab). En el primer caso, una vez definida la función puede invocarse directamente mediante

```
nombre(x1, ..., xm);
```

donde los argumentos de entrada `x1, ..., xm` deben haber sido asignados previamente. En el segundo caso, una vez guardado el archivo con extensión .sci, es necesario cargarlo, para lo cual se usa la expresión

```
exec ruta;
```

donde `ruta` es la ruta completa del archivo .sci. Una vez cargada puede invocarse igual que en el caso anterior.

Si se desea que la función esté permanentemente disponible para Scilab será preciso incluirla en una biblioteca. Por ejemplo, puede usarse la instrucción:

```
genlib("worklib", "SCI/macros/work/", %f, %t)
```

para generar la biblioteca `worklib` (incluyendo su estructura)[12] a partir de las funciones ubicadas como archivos `.sci` en el directorio `SCI/macros/work/`.[13] La función `genlib()` compila cada función en la forma de un archivo `.bin` y crea dos archivos, `names` y `lib`; `names` contiene los nombres de las funciones y `lib` es un archivo binario asociado a la biblioteca.

Para que la biblioteca así generada esté disponible cada vez que se inicia Scilab puede colocarse la instrucción

```
worklib = lib("SCI/macros/work/")
```

en el archivo denominado `.scilab` ubicado en el directorio `SCIHOME`.[14] Este archivo se ejecuta durante la inicialización, declarando dicha biblioteca.

NOTA 1: En Scilab las variables existentes en el espacio de trabajo están disponibles dentro de las funciones invocadas desde el espacio de trabajo. Sin embargo, a partir del momento en que se invoca una de esas variables dentro de una función se realiza una copia y cualquier modificación que experimente tendrá sólo alcance local dentro de la función. Esto significa que la variable original no se modificará. La única excepción es cuando la variable es un argumento de salida de la función.

NOTA 2: Cualquier función previamente definida en línea o cargada mediante `exec` en el espacio de trabajo, estará también disponible dentro de los scripts y las funciones que se definan de ahí en más.

NOTA 3: Dentro de una función de Scilab pueden definirse nuevas funciones. para lo cual basta definirlas antes de utilizarlas. Alternativamente, pueden incluirse funciones a utilizar en una función al final (luego del comando `endfunction`). En ambos casos estas funciones sólo tendrán alcance local, es decir estarán disponibles sólo dentro de la función. Este principio es aplicable recursivamente, es decir que dentro de una función definida dentro de una función también pueden definirse funciones que tendrán alcance en el nivel en el que se definieron o en niveles más internos.

22 Resolución numérica de ecuaciones

La función `fsolve()` permite resolver numéricamente ecuaciones o sistemas de ecuaciones no lineales de la forma

---

[12] Los símbolos `%f` y `%t` son las constantes lógicas *false* (falso) y *true* (verdadero)

[13] SCI es una variable de entorno que contiene una cadena de caracteres que indica el directorio donde se ha instalado Scilab.

[14] La variable de entorno SCIHOME contiene una cadena de caracteres que indica donde se encuentran algunos archivos de inicialización del usuario.

$$F(x) = 0.$$

Para ello debe definirse previamente la función `F`. Como primer ejemplo, supongamos que deseamos resolver

$$x^3 + x = 5$$

Definimos entonces la función F(x) como

$$F(x) = x^3 + x - 5$$

En Scilab esta función quedará definida mediante la sintaxis

```
function y = F(x)
   y = x^3 + x - 5;
end
```

Dado que la función `fsolve()` procede iterativamente, requiere un valor tentativo inicial `x0`. Tomando dicho valor como 0, la sintaxis es

```
x0 = 0;
x = fsolve(x0, F)
```

El resultado es

```
x =
   1.5159802
```

El error entre el valor de la función en `x` y el valor deseado es `F(x)`:

```
error = F(x)
error =
  -8.882D-16
```

Debe tenerse en cuenta que cuanto más cerca se encuentre el valor tentativo de la verdadera raíz más rápida será la convergencia. En el caso de funciones con extremos locales el algoritmo podría fallar si el valor tentativo está más allá de un extremo local.

Para el caso de sistemas de ecuaciones bastará definir `F` como una función multidimensional dependiente de un argumento multidimensional. Por ejemplo, para resolver el sistema

$$x^3 + y = 5$$
$$x + y^3 = 4$$

podemos plantear la función

```
function z = F(u)
   z(1) = u(1)^3 + u(2) - 5;
   z(2) = u(1) + u(2)^3 - 4;
end
```

y luego aplicar la función `fsolve`

```
u0 = [1; 1];
u = fsolve(u0, F)
```

Notar que, por defecto, en la definición de `F` el argumento se toma como vector columna, por lo cual el valor tentativo `u0` debe asignarse como vector columna, de lo contrario se obtiene un mensaje de error.

El resultado del algoritmo anterior es

```
u = [2x1 double]
   1.5396832
   1.3499894
```

La solución en la simbología original es

```
x = u(1)
y = u(2)
```

Podemos, verificar calculando el valor de `F` en `u`:

```
error = F(u)
error = [2x1 double]
  -8.882D-16
   8.882D-16
```

El error es muy pequeño.

23 Funciones estadísticas

Un tipo de análisis de interés es el análisis estadístico de señales o de series de datos. Si bien el tema es demasiado amplio para intentar abarcarlo en forma completa, mencionaremos algunas funciones útiles. La primera es `mean()`, que permite obtener la media aritmética de un vector o matriz. En el caso de matrices, admite la sintaxis

```
xm = mean(x, dim)
```

donde `x` es la matriz y `dim` la dimensión a lo largo de la cual se promedia. Si es 1 se promedia a lo largo de cada columna, dando como resultado un vector fila que contiene tantas medias como columnas en `x`. Si es 2, promedia a lo largo de cada fila, dando un vector columna con tantas medias como filas en `x`. Esto es útil para analizar varias señales de igual tamaño con un solo comando. También están disponibles las funciones `geomean()`, que obtiene la media geométrica, y `median()`, que obtiene la mediana. Las funciones `max()` y `min()` permiten obtener los valores extremos de un vector o matriz. La sintaxis

```
[xmax, i] = max(x, dim)
```

obtiene además el índice `i` del primer máximo. Dado que estas funciones permiten obtener los extremos de una lista de valores (`max(x1, x2, ...,xn)`), para obtener el extremo en cada columna o en cada fila `dim` debe ser `"r"` o `"c"`:

```
[xmax,  i] = max(x, "r")
[xmax,  i] = max(x, "c")
```

donde `"r"` significa que obtiene una fila (*row*) de máximos por columna y `"c"` que obtiene una columna (*column*) de máximos por fila.

Las funciones `stdev()` y `variance()` obtienen el desvío estándar y la varianza de un vector o matriz y también admiten especificar la dimensión a lo largo de la cual se realiza el análisis.

También hay disponibles algunas funciones de estadística descriptiva. la función `histc()` permite obtener el histograma de una serie de datos. Si bien tiene muchas opciones que pueden investigarse mediante `doc histc`, la sintaxis más simple es

```
y = histc(x, nbin)
```

donde `x` es un vector de datos, `nbin`, la cantidad de casilleros (*bins*) entre el máximo y el mínimo de `x`, e `y`, la frecuencia o cantidad de valores en cada casillero. Los casilleros son contiguos y de tamaño uniforme dado por `(max(x) - min(x))/nbin`. El siguiente código obtiene y grafica el histograma de unos números aleatorios con distribución normal:

```
x = rand(1,10000, "n");
nbin = 100;
y = histc(x, nbin);
xbin = min(x) + (0:99)*(max(x) - min(x))/nbin;
plot(xbin,y)
```

En la figura 19 se muestra el histograma correspondiente. Debe tenerse en cuenta que este resultado se dio para una ejecución particular del código anterior. Dado que se trata de números aleatorios,[15] al repetir la ejecución se obtendrían resultados diferentes cada vez.

Otra función de interés es `perctl()`, que permite calcular los percentiles, es decir los valores, dentro del rango de los datos, que superan al *n* % de los valores ordenados en orden creciente. La sintaxis es

```
p = perctl(x, n)
```

donde `x` es un vector de datos, `n` el percentil deseado (o un vector con los percentiles deseados) y `p` una matriz de dos columnas cuya primera columna contiene el o los percentiles deseados y la segunda el o los índices correspondientes a los valores de `x` inmediatos inferiores a los respectivos percentiles.

Por último hay un conjunto de funciones de distribución acumulada correspondientes a diferentes distribuciones clásicas. Por ejemplo, para la distribución normal tenemos `cdfnor()`. Mientras las funciones de densidad de probabilidad responden en general a expresiones analíticas cerradas, las funciones de distribución acumulada, así

[15] En realidad son números pseudoaleatorios obtenidos con un algoritmo determinístico, pero son prácticamente aleatorios.

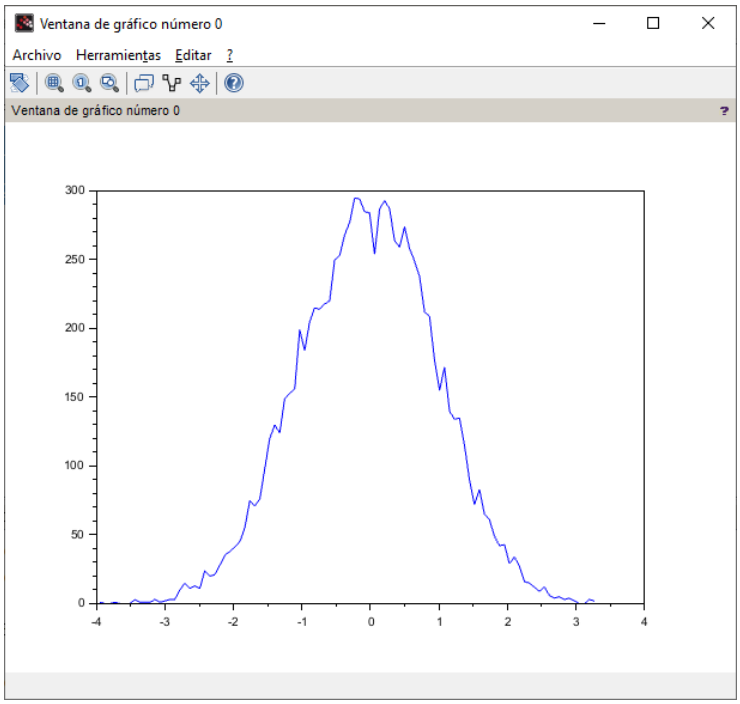

Figura 19. Ejemplo de histograma obtenido mediante `histc()`.

como sus inversas se deben calcular numéricamente. Esta familia de funciones de Scilab tiene la particularidad de que permiten calcular, apelando a diferentes sintaxis, tanto la función de distribución acumulada (la probabilidad `P` de que el valor de la variable aleatoria sea menor que `X`) como su inversa (el valor de `X` para que la probabilidad de que la variable aleatoria sea menor que `X` sea `P`). El primer caso se logra con

```
P = cdfnor("PQ", X, Mean, Std)
```

donde `X` es el valor de la variable aleatoria para el que queremos calcular la probabilidad acumulada `P`, Mean es la media y `Std` es el desvío estándar de la distribución. El segundo caso se logra con

```
X = cdfnor("X", Mean, Std, P, 1-P)
```

NOTA: El último argumento es necesario, aunque carece de sentido ya que es redundante. No incluirlo arroja un mensaje de error.

24 Tipos

Las variables de Scilab, al igual que en muchos otros lenguajes, se clasifican en *tipos* (*types*). El tipo correspondiente a una variable determina el conjunto de valores que puede asumir la variable, la forma de almacenamiento (representación lógica y tamaño requerido en bytes) y los operadores o funciones que se le pueden aplicar.

Cada tipo en Scilab tiene asociado un identificador numérico y un nombre descriptivo según se detalla en la tabla 1, que pueden determinarse mediante las funciones `type()` y `typeof()` respectivamente.

Consideremos, por ejemplo, la siguiente variable:

Tabla 1. Tipos utilizados en Scilab, sus identificadores numéricos y sus nombres obtenidos mediante las funciones `type()` y `typeof()`.

Tipo	Descripción	Nombre según `typeof()`
1	Escalares, vectores o matrices reales o complejos	constant
2	Polinomios o vectores o matrices de polinomios con coeficientes reales o complejos	polynomial
4	Escalares, vectores o matrices lógicas (booleanas)	boolean
5	Matrices ralas (con pocos valores no nulos)	sparse
6	Matrices ralas booleanas	boolean sparse
8	Escalares, vectores y matrices de enteros de 1 byte, 2 byte, 4 byte, 8 byte, y de enteros sin signo de 1 byte, 2 byte, 4 byte, 8 byte	int8, int16, int32, int64 uint8, uint16, uint32, uint64
9	Handles gráficos	handle
10	Objetos de texto (cadenas, vectores o matrices de cadenas)	string
13	Funciones compiladas a partir de código de Scilab	function
14	Bibliotecas de funciones compiladas	library
15	Lista simple	list
16	Lista tipificada (tlist) Funciones racionales Directorios	nombre declarado por el usuario rational dir
17	Lista tipificada matricial (mlist) Arreglos de celdas (cell) Estructuras (struct)	nombre declarado por el usuario ce st
128	Identificador de bloques Xcos, de un resultado `lufact()`, etc.	pointer
129	Lista de tamaño implícito utilizada para indexación (por ejemplo, `:`)	implicitlist
130	Funciones primitivas de Scilab	fptr
0	Componentes indefinidas de una lista simple Argumentos omitidos en un llamado a función Elementos borrados de una lista con null()	void listdelete

```
a = "hola";
```

Entonces el comando

```
type(a)
```

da como resultado el identificador numérico `10`, mientras que

```
typeof(a)
```

da como resultado el nombre de tipo `"string"`.

Obsérvese que cada tipo de variable puede tener asociado más de un nombre. En el caso de las listas tipificadas y las listas tipificadas matriciales, además de varias ya incluidas en Scilab, el usuario puede introducir nuevos nombres descriptivos personalizados.

Hemos visto algunos objetos de Scilab representativos de diferentes tipos simples, como *constant*, *polynomial*, *boolean*, *string*. Analizaremos ahora algunos tipos más.

25 Integer

Un tipo importante es el *integer* (entero). Mientras que el tipo *constant* se almacena en memoria como un número de punto flotante decimal que utiliza 8 byte (o 2 números si es complejo), los tipos enteros pueden utilizar 1 byte, 2 byte, 4 byte u 8 byte, tanto con signo como sin signo. Esto origina 8 subtipos: int8, int16, int32, int64, uint8, uint16, uint32, uint64, siendo los 4 primeros con signo y los 4 últimos sin signo.

Para obtener un entero de tipo uint16, por ejemplo, se aplica la función de igual nombre, `uint16()`, a un número (que por defecto será de punto flotante). La conversión elimina la parte decimal (y, si la hubiera, también la parte imaginaria), y obtiene el resto de dividir por $2^{16} = 65\,536$ (aritmética módulo 2^{16}). Por ejemplo,

```
a = uint16(200000)
```

devuelve

```
a  =
  3392
```

ya que 200 000 / 65 536 es 3 con un resto de 3392.

Otra operación que proporciona enteros y que se verá más adelante, es la lectura de archivos en formato entero.

Una característica de los tipos enteros sin signo es que las operaciones aritméticas se realizan módulo 2^n, donde n es la cantidad de bits. Así, por ejemplo, para uint16, 30 000 + 60 000 no es 90 000 sino 24 464. Esto es porque el máximo representable con 16 bit es 65 535 y cualquier acarreo que la operación implique en el bit más significativo se pierde. Esto sucede para las 4 operaciones, aclarando que la división siempre trunca los decimales.

El caso de los enteros con signo es similar, sólo que antes de proceder a realizar la operación de módulo 2^n se suma 2^{n-1} y luego de realizarla se resta 2^{n-1}.

Si, partiendo de números enteros, se deseara pasar a su representación decimal en punto flotante se puede utilizar la función `double()`. Algunas operaciones, como la potenciación `a^n` en el caso de `n` no entero, convierten internamente el formato numérico a *double*, siendo el resultado *double*.

26 Lista simple (list)

Constituye una lista ordenada de objetos que pueden ser heterogéneos. Para crear una lista se debe utilizar la función `list()`. La instrucción

```
L = list(a1, a2,..., an);
```

crea una lista de `n` objetos `ak`, que pueden ser objetos arbitrarios de Scilab. Por ejemplo

```
L = list("lista", 1, [1, 2; 3, 4], poly([1 2 3], "x"));
```

crea una lista formada por una cadena, una constante, una matriz y un polinomio.

La función `size()` puede utilizarse para determinar la cantidad de elementos en la lista. En este caso, 4. Para el caso de listas, la función `length()` ofrece el mismo resultado.

Los objetos de una lista pueden invocarse de igual forma que los elementos de un vector. Así, `L(1)` da como resultado la cadena `" lista"` y `L(4)` el polinomio

```
                2   3
  -6 +11x -6x  +x
```

Algunos de los objetos podrían no estar especificados:

```
L = list(3, , "una cadena");
```

En este caso `L(2)` no devuelve ningún objeto. El tipo de dato, que puede visualizarse en el visualizador de variables (arriba a la derecha de la pantalla) declara como tipo de variable `"Unknown Datatype"`. Las funciones `type()` y `typeof()` devuelven `0` y `void`. Nada quita que posteriormente pueda asignarse un objeto, por ejemplo:

```
L(2) = poly([1 2 3], "x");
```

También es posible crear una lista vacía, por ejemplo

```
L = list()
```

y en cualquier momento se puede asignar un objeto a cualquier elemento de la lista:

```
L(4) = "una cadena";
```

En este ejemplo `size(L)` da como resultado 4, mientras que si se hubiera aplicado antes de asignar `L(4)` hubiera dado 0. Los otros elementos son declarados como desconocidos.

Se puede reemplazar un elemento de una lista simplemente asignándolo nuevamente. Así, si definimos la lista

```
L = list(1, "hola" ,3);
```

al invocar `L(2)` obtenemos `"hola"`. Podemos reemplazarlo así:

```
L(2) = 4
```

luego de lo cual al invocar `L(2)` obtenemos `4`.

Es posible, por otro lado, insertar un objeto nuevo en una lista haciendo que los que siguen se muevan una posición hacia la derecha para abrir espacio al nuevo. Para ello se usa la función `insert()`. En la lista anterior, por ejemplo, podemos insertar en la posición 2 un elemento mediante:

```
L(2) = insert("nuevo")
```

Con lo cual ahora `L(2)` pasa a contener `"nuevo"`, `L(3)` pasa a contener `4` y `L(4)` pasa a contener `3`.

Es posible, a la inversa, eliminar un elemento de una lista mediante la asignación de la función `null()`:

```
L(2) = null()
```

El elemento desaparece, no es que simplemente deje de estar asignado. En el caso anterior la cadena `"nuevo"` que estaba en la posición 2 desaparece y su lugar es ocupado por el elemento que estaba en la posición 3, de modo que ahora `L(2)` es `4`.

NOTA: En la versión 2025.0.0 la función `insert()` aún no está documentada.

Una aplicación interesante de las listas es su utilización en reemplazo de la lista de índices para extracción de elementos de un arreglo. Por ejemplo, supongamos que `a` es un arreglo de 4 dimensiones. Si definimos una lista `L` como

```
L = list(2, 4, 3, 3);
```

entonces

```
a(L(:))
```

es equivalente a

```
a(2, 4, 3, 3).
```

Así planteado, podría parecer que este enfoque no ofrece ninguna ventaja. La verdadera utilidad se aprecia, por ejemplo, cuando la cantidad de dimensiones es indeterminada o variable, por ejemplo al definir una función del usuario alguno de cuyos argumentos de entrada es un arreglo cuya dimensión es a priori desconocida.

A modo de ejemplo, supongamos que queremos obtener un subarreglo que contenga la última columna de todas las dimensiones de un arreglo `a`. Definimos una lista `L` mediante el siguiente código:

```
siz = size(a);
L = [];
for i= 1:length(siz)
   L($+1) = 1:siz(i);
end
```

Esta lista contiene tantos vectores como dimensiones tiene `a`, y cada uno de estos vectores contiene una cantidad de índices igual a la longitud de la dimensión correspondiente. Por ejemplo, si `a` tiene un tamaño 4×3×2×2, será equivalente a

```
L = list([1 2 3 4], [1 2 3], [1 2], [1 2])
```

Ahora reemplazamos la segunda dimensión (columnas) por la columna con el índice máximo:

```
L(2) = siz(2);
```

En el ejemplo,

```
L = list([1 2 3 4], 3, [1 2], [1 2])
```

Finalmente,

```
b = a(L(:))
```

extrae la tercera columna de todas las dimensiones.

La expresión `L(:)` también puede usarse en reemplazo de uno o más argumentos consecutivos de una función. Por ejemplo, si

```
L = list(2, 3);
```

y `func()` es una función con tres argumentos, entonces `func(1, L(:))` es equivalente a `func(1, 2, 3)`.

27 Lista tipificada (tlist)

Al igual que la lista simple, es una lista ordenada de objetos heterogéneos de Scilab a la cual se la asigna un nombre de tipo y, opcionalmente, un nombre genérico a cada elemento de la lista. Para definir una lista tipificada se usa la función `tlist()`:

```
T = tlist(tipo, a1, ..., an)
```

donde `tipo` es una cadena de caracteres o un vector de cadenas de caracteres y `a1`, ..., `an` son los elementos de la lista. En el primer caso, la cadena es el nombre del tipo asignado a la lista, por ejemplo:

```
T = tlist("altura", 1.50, 1.67, 1.59, 1.35, 1.84)
```

En el segundo caso, `tipo` esta conformado por el nombre de clase o tipo de la lista y los siguientes son los nombres formales de los elementos de la lista, por ejemplo

```
T = tlist(["dim", "h", "a", "b"], 2.70, 5, 8.30)
```

En este caso los elementos propiamente dichos pueden ser invocados de dos formas. La primera, la normal en toda lista, escribiendo `T(k)` donde `k` es el número de elemento. Tener en cuenta que el primer elemento es formal y representa el vector de tipo, por lo que los elementos propiamente dichos se invocan mediante `T(k+1)`.

La segunda forma es considerar los elementos como campos de la estructura, que se invocan anexando el nombre formal del elemento a invocar separado por un punto, por ejemplo, `T.h`. Cualquiera de estas formas permite asignar valores, siendo equivalentes:

```
T(2) = 2.7
T.h =  2.7
```

NOTA 1: Si se sobrescribe `T(1)` con un elemento que no sea una cadena o un vector de cadenas, la lista pierde el carácter de tipificada y pasa a ser una lista simple, aunque luego se asigne una cadena o vector de cadenas, incluso el original.

NOTA 2: No necesariamente deben asignarse todos los campos, o incluso podrían asignarse más campos que los que tienen asignados nombres formales, siendo válidas estas dos instrucciones:

```
T = tlist(["dim", "h", "a", "b"])
T = tlist(["dim", "h", "a", "b"], 2.70, 5, 8.30, 6.7)
```

NOTA 3: Como puede verse, la lista no tiene ningún tipo de protección ni verificación de coherencia, si no se procede con cuidado se puede destruir fácilmente la estructura.

NOTA 4: Las tlist son especialmente interesantes para introducir funciones u operadores sobrecargados (overloaded), es decir, redefinir el comportamiento de una operación o función para un tipo de datos diferente del habitual. No profundizaremos sobre ello por ser una característica demasiado especializada.

Un ejemplo particular interesante de tlist son las *funciones racionales* (cociente de dos polinomios). La forma es

```
R = tlist(["r","num","den","dt"], NUM, DEN, DT)
```

donde `NUM` y `DEN` son los polinomios del numerador y denominador, y `DT` no está asignado (es `[]`). Se asigna en el caso en que la función racional se interprete como un sistema lineal, y en ese caso indica si es en tiempo discreto, `"d"`, o continuo, `"c"`). Nótese que el uso de esta sintaxis no permite realmente crear una función racional, ya que dicho primer argumento es un objeto protegido que no puede redefinirse. Sin embargo, es posible utilizar la función `rlist()`:

```
R  = rlist(NUM, DEN)
```

También puede crearse una función racional simplemente dividiendo dos polinomios. Por ejemplo,

```
NUM = %s - 1;
DEN = %s + 1;
R = NUM/DEN;
```

define la función racional $(s - 1)/(s + 1)$.

Si, como consecuencia de alguna operación, se obtiene una función racional `R`, es posible extraer su numerador o denominador de dos formas: invocando el campo

```
NUM = R.num;
DEN = R.den;
```

o bien invocando el elemento correspondiente de la lista:

```
NUM = R(2);
DEN = R(3);
```

Una vez extraídos, son simples polinomios que admiten todas las operaciones y funciones sobre polinomios, como extracción de coeficientes, `coeff()`, raíces, `roots()`, o cálculo del valor en determinado valor de la variable, `horner()`. Por ejemplo,

```
CNUM = coeff(R.num)
```

28 Lista matricial (mlist)

Es igual a una tlist, sólo que la forma de asignación y extracción difiere en algunos aspectos. Se introduce una mlist mediante la función `mlist()`, por ejemplo:

```
M = mlist(["matriz"; "e1"; "e2"; "e3"], 1, "P(s) ", 1+%s)
```

Primero se indica el nombre del tipo creado, a continuación el nombre de los campos y finalmente el contenido de los campos. Podría no asignarse algún campo.

No es posible invocar las componentes mediante su número de orden, como en las tlist. Así, `M(3)` produce un error.

Una primera forma de extraer, introducir y modificar campos es mediante el agregado del nombre del campo intercalando un punto. Por ejemplo `M.e1` da como resultado 1, mientras que `M.e2` resulta en la cadena `"P(s)"`. Otra forma de invocarlo es expresando el contenido en forma funcional, siendo el argumento el nombre de cualquier campo:

```
M("e1")
```

Matemáticamente, vendría a ser una función cuyo dominio es el conjunto de los nombres de los campos.

Otra forma es utilizar la función `getfield()`, que permite extraer los campos, incluso si, como en el caso de la tlist, el nombre de los campos no se incluye en la definición del tipo. Así,

```
getfield(k, M)
```

obtiene el *k*-ésimo campo, observando que si `k` es 1 se obtiene el vector de cadenas que indica el tipo y los nombres de los campos.

29 Estructura (struct)

Es una lista de nombres de campos y contenidos de los campos. Se crea, normalmente, mediante la instrucción `struct()`:

```
S = struct(campo_1, cont_1, ..., campo_n, cont_n)
```

Los nombres son cadenas simples, los contenidos, cualquier objeto de Scilab, incluyendo escalares, matrices, polinomios, listas, estructuras, etc.). Por ejemplo:

```
S = struct("x", "A", "y", 2.22, "z", struct("a", 1, "b", 0))
```

El contenido de cada campo se invoca agregando un punto y el nombre del campo. Así,

```
S.x
```

devuelve la cadena `"A"`. Este principio se puede extender a las estructuras anidadas. Así

```
S.z.a
```

dará como resultado `1`.

Se puede asignar un objeto a un campo (o editar un campo) después de la creación de la estructura. Por ejemplo

```
S.x = 2
```

De hecho, es posible crear desde cero una estructura previamente inexistente simplemente asignando un campo. Así,

```
T.x = 1
```

crea una estructura `T` que contiene un único campo (siempre y cuando no existiera previamente). Luego se puede ampliar la estructura asignando contenidos a nuevos campos, incluso estructuras anidadas:

```
T.y = 2.2
T.z.a = 1
T.z.b = 0
```

Se pueden obtener los nombres de los campos de una estructura mediante la función `fieldnames()`. Por ejemplo

```
campos = fieldnames(T)
```

devuelve

```
campos  =
  "x"
  "y"
  "z"
```

es decir, un vector de cadenas con los nombres de los campos.

El orden de los campos no es relevante y de hecho no es posible invocar los campos mediante la sintaxis `T(k)`. Al intentar hacerlo, si `k > 1` indica que es un índice inválido. Pero si `k = 1`, es decir, `T(1)`, simplemente devuelve la propia estructura, igual que si se invocara `T`. Esto sucede porque es posible tener un arreglo de estructuras. Por ejemplo, una vez generada una estructura `T` con cualquiera de los métodos anteriores, podríamos agregar una nueva componente con igual estructura simplemente asignando un campo:

```
T(2).x = 7
```

En este caso el resultado es

```
T  =
  2x1 struct with fields:
  ["x", "y", "z"]
```

Ambas “componentes” del arreglo contienen ahora formalmente todos los campos. Si no hacemos nada más, la componente 2 tendrá asignado sólo uno de los campos. Así, si invocamos `T(2).x` obtenemos `7`, pero si invocamos `T(2).y` obtenemos `[]` ya que este nuevo campo no fue asignado para la componente original. Desde luego podemos asignar los campos vacíos:

```
T(2).y = "nivel"
```

Si ahora invocamos `T.y`, es decir sin especificar qué componente del arreglo nos interesa, obtenemos una lista que contiene los dos objetos asociados respectivamente a los campos `T(1).y` y `T(2).y`, en este caso, respectivamente `2.2` y `"nivel"`.

Notar que no necesariamente el tipo de los campos homólogos es el mismo. Sin embargo, el caso más útil se tiene cuando se desea tener muchos conjuntos de datos con la misma estructura, por ejemplo cierta información sobre diferentes personas donde el tipo de datos es siempre el mismo.

También es posible un arreglo bidimensional con la misma estructura. Así, por ejemplo,

```
T(2,3).x = 5
```

asignará el número `5` a la estructura ubicada en la posición `(2, 3)`.

La consola es bastante limitada para presentar estructuras de datos complejas, La función `tree_show()` abre un panel interactivo que permite visuaizar simultáneamente las diferentes componentes de una lista, lista tipificada, o estructura. Así,

```
tree_show(T)
```

mostrará el contenido de cada campo (ver figura 20). En el caso de un arreglo bidimensional de estructuras, esta función muestra el contenido unidimensionalmente, leyendo primero por columnas y luego por filas.

30 Arreglos de celdas (cell array)

Un *arreglo de celdas* (*cell array*) consiste en un arreglo cuyas componentes son un tipo especial de datos, *cell*, que es en realidad un contenedor para cualquier otro tipo de dato. Esto permite compatibilizar tipos de datos heterogéneos, a efectos de integrar una matriz o arreglo multidimensional. La instrucción

```
a = cell(n,m)
```

crea un arreglo de celdas vacío de `n` filas y `m` columnas. Se puede asignar a cada componente un dato de cualquier tipo, por ejemplo una matriz, encerrando entre llaves `{}` los índices de la componente. Así,

```
a{2,3} = [1 2; 3 4]
```

asigna la matriz `[1 2; 3 4]` a la celda en la posición en la fila `2` y columna `3`. Las llaves también se utilizan para invocar o extraer el contenido de una celda (en este caso una matriz). Las componentes de la matriz pueden invocarse como se hace habitualmente para invocar componentes de matrices. Por ejemplo, `a{2,3}(1,2)` da como resultado `3`. Esto se denomina *extracción recursiva*, ya que primero se extrae la

celda de índices `2` y `3` y luego, siendo el contenido de esta celda una matriz, se extrae su componente de índices `1` y `2`.

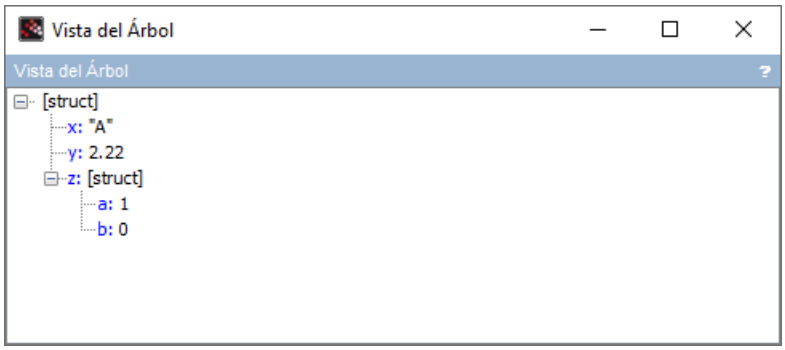

Figura 20. Interfaz de vista de árbol de una variable de tipo struct generada mediante la función `tree_show()`

NOTA: Igual que en matrices, las celdas se pueden recorrer secuencialmente empezando por la primera columna de arriba a abajo, luego la segunda columna de arriba a abajo, etc., invocadas con un solo índice. Por ejemplo, si `a` es un arreglo de celdas de 2 filas y 3 columnas, `a{5}` es lo mismo que `a{1,3}`, ya que se recorre en el siguiente orden:

```
a{1,1}, a{2,1}, a{1,2}, a{2,2}, a{1,3}, a{2,3}
```

También se pueden crear arreglos de celdas directamente asignando las componentes entre llaves:

```
a = {[1 2] [3 4; 5 6]; [7; 8] [9 10]}
```

Este arreglo de celdas tiene esta estructura:

```
a  =
  [1x2 constant]  [2x2 constant]
  [2x1 constant]  [1x2 constant]
```

Aunque Scilab no indica el contenido de cada celda, probablemente debido la naturaleza sumamente heterogénea de las potenciales celdas que podría tener un arreglo genérico, el contenido en este caso es

```
{ [1 2]     [3 4   }
{            5 6]  }
{                  }
{ {[7       [9 10] }
{   8]             }
```

También puede usarse la función `makecell()` para obtener la misma estructura

```
a = makecell([2 2], [1 2], [3 4; 5 6], [7;8], [9 10])
```

donde el primer argumento, `[2 2]`, indica la dimensión del arreglo de celdas y los siguientes el contenido de las celdas, recorrido por la última dimensión (en este caso columna). La siguiente instrucción crea un arreglo de celdas de tipos heterogéneos:

```
a = {[%f, %t], [1, 2; 3, 4]; "hola", poly([0 1 2], "v")}
```

donde la primera fila contiene un vector de valores lógicos y una matriz numérica, mientras que la segunda fila contiene una cadena y una función racional. El resultado es presentado como

```
a  =
  [1x2 boolean]  [2x2 constant  ]
  [1x1 string ]  [1x1 polynomial]
```

Para los arreglos de celdas existe una forma particular de extracción mediante paréntesis en lugar de llaves en la que lo que se obtiene es un subarreglo de celdas. Por ejemplo,

```
b = a(1,:)
b  =
  [1x2 boolean]  [2x2 constant]
```

La función `iscell()` aplicada a una variable permite determinar si es o no un arreglo de celdas, dando como resultado un valor lógico `T` (true) o `F` (false). La función `cell2mat()` convierte un arreglo de celdas del mismo tipo en una matriz, siempre que haya congruencia entre las dimensiones.

31 Archivos

En ocasiones es necesario guardar en un archivo una variable, o, a la inversa, cargar en una variable el contenido de un archivo (o parte de él). Vimos anteriormente el caso particular de las señales de audio digital, para las que existen las funciones dedicadas `wavwrite()` y `wavread()`. En casos más generales será necesario acceder al archivo directamente. Antes de poder realizar operaciones de escritura o lectura de un archivo será necesario, primero, abrirlo mediante la función `mopen()` con la sintaxis

```
[fid, err] = mopen(archivo, modo, swap)
```

donde:

`fid`: Identificador numérico del archivo a utilizar en las operaciones de lectura y escritura asignado automáticamente por orden consecutivo

`err`: Código de error opcional (0 significa que no hubo error)

`archivo`: Ruta y nombre del archivo

`modo`: Cadena que determina el tipo de apertura,
- `"r"`: lectura (read)
- `"rb"`: lectura de datos binarios
- `"rt"`: lectura de texto

`"w"`:	escritura (write) borrando el contenido anterior
`"wb"`:	escritura de datos binarios borrando el contenido anterior
`"wt"`:	escritura de texto borrando el contenido anterior
`"r+"`:	lectura y escritura
`"w+"`:	lectura y escritura borrando el contenido anterior

`swap`: Si es 1 (valor por defecto), cuando el procesador trabaja con formato IEEE little-endian se produce un reordenamiento automático de los bytes para recuperar el orden normal.. Si es 0 no hace nada.

Luego de abrir el archivo se puede realizar la lectura mediante las funciones `mgeti()`, `mget()` o `mgetl()`. La función `mgeti()` lee bytes o palabras numéricas y las entrega como tipo entero (int)

```
x = mgeti(n, tipo, fid)
```

Aquí `n` es la cantidad de ítems a leer, `tipo` indica cómo se interpretan los datos:[16]

`"d"`:	double	(8 byte)
`"f"`:	float	(4 byte)
`"l"`:	long	(8 byte)
`"i"`:	int	(4 byte)
`"s"`:	short	(2 byte)
`"c"`:	character	(1 byte)

Los tipos enteros (`"l"`, `"i"`, `"s"`, `"c"`) pueden ir precedidos de

`"u"`:	sin signo

Todos pueden ir seguidos también de

`"l"`:	*little endian*
`"b"`:	*big endian*

`fid` es el identificador de archivo asignado al abrir el archivo mediante `mopen()`. Si es `−1` se refiere al último archivo abierto, igual que si se omite .

Por ejemplo:

```
x = mgeti(10, "ull", 2)
```

leerá 10 valores enteros sin signo de 8 byte *little endian* del archivo abierto con identificador 2. Para leer todo el archivo bastará utilizar un número mayor o igual que el tamaño del archivo. Si éste no se conoce, puede aplicarse la función `fileinfo()` al nombre de archivo. Esta función obtiene la información del archivo, siendo la primera componente su longitud en bytes. Así, por ejemplo,

```
x = mgeti(fileinfo(archivo)(1), "uc", 2)
```

[16] Los tipos float y double (formato de punto flotante de simple y doble precisión) están definidos en la norma IEEE Std 754.2008

lee todo el archivo en formato entero de 1 byte sin signo. Cada componente de `x` contendrá un byte.

La función `mget()` es similar, pero convierte los números leídos según la interpretación que corresponda a tipo doble (double). Esto es necesario cuando se va a operar algebraicamente con los datos, ya que las operaciones enteras se realizan modulo 2^n, donde *n* es la cantidad de bits.

La función `mgetl()` lee líneas de texto:

```
texto = mgetl(file_desc, m)
```

donde `texto` es la variable que contendrá el texto, `file_desc` puede ser la ruta y nombre del archivo o el identificador `fid` devuelto por `mopen()`, y `m` es la cantidad de líneas a leer. Si `m = -1`, lee todas las líneas.

Las instrucciones anteriores leen el archivo desde el principio. Muchas veces se desea leer a partir de una posición diferente, lo cual se logra aplicando un desplazamiento u *offset* `d` al puntero que señala el primer byte a leer. Ello se logra mediante la función `mseek()`:

```
mseek(d, fid)
```

Debe tenerse en cuenta que un offset igual a 0 corresponde al primer byte del archivo. Una vez leídos `n` bytes, el puntero quedará apuntando al próximo byte no leído, es decir aquél con un offset `d + n`. En cualquier momento se puede determinar el offset del próximo byte a leer mediante la función `mtell()`:

```
mtell()
```

La función `mput()` permite escribir datos en un archivo mediante la siguiente sintaxis:

```
mput(x, tipo, fid)
```

donde `x` es un vector de números de punto flotante (double) o enteros (int8, int 16, etc.), `tipo` indica cómo se interpretan los datos (ver `mgeti()`) y `fid` es el identificador numérico entregado por `mopen`. El argumento `tipo` es opcional, siendo el valor por defecto `"l"` (long). `fid` también es opcional, si se omite se toma como `-1`, correspondiente al último archivo abierto.

La función `file()`, aplicada con la siguiente sintaxis

```
[fid, typ, archivo, mod, swap] = file(fid)
```

permite recuperar información sobre el archivo, entre ella la ruta y nombre completos de un archivo abierto con el idntificador `fid`, entre otros datos.

NOTA 1: Si la función se invoca con la lista de argumentos vacía, `file()`, lista todos los archivos abiertos.

NOTA 2: Por defecto siempre aparecen en la lista los identificadores 0, 5 y 6. No son archivos reales, corresponden a `stderr`, `stdin`, `stdout`, es decir error estándar, entrada estándar y salida estándar.

Una vez que se terminó de utilizar un archivo, tanto para la lectura como para la escritura, es conveniente cerrarlo para evitar modificarlo accidentalmente o bloquearlo para su utilización por otros programas. Se puede cerrar un archivo mediante la función `mclose()`:

```
mclose(fid)
```

donde `fid` es el identificador del archivo específico que se desea cerrar. Si en lugar de un identificador numérico correspondiente a un archivo abierto se utiliza la cadena `"all"`, se cierran todos los archivos.

32 Recursos interactivos

En ocasiones es necesario interactuar con la ejecución de un script, por ejemplo ingresando datos u otra información requerida para la continuidad de un proceso. No siempre esa información se conoce a priori, en cuyo caso podría pasarse como argumento de una función, sino que podría depender de algún evento externo basado en un resultado intermedio. Una primera función es `input()`. Puede solicitarse ingresar un número o una cadena mediante las respectivas sintaxis:

```
x = input(mensaje);
x = input(mensaje, "s");
```

donde `mensaje` es una cadena seleccionada apropiadamente para especificar qué dato se solicita, por ejemplo: `"Ingrese la frecuencia: "` o `"x = "`. Notar que es necesario colocar un espacio final si se quiere separar el mensaje del dato a ingresar (lo cual es recomendable). Luego de escribir el dato se debe oprimir Enter. Si se oprime Enter sin haber ingresado un dato la variable contendrá una matriz vacía. Podría usarse este recurso para validar un dato propuesto por defecto. Luego se deberá constatar si la matriz está vacía viendo si su longitud es 0, y en tal caso asignar el valor por defecto. Por ejemplo:

```
f = input("Frecuencia [1000 Hz]: ");
if length(f)==0
   f = 1000;
end
```

Otra situación de interés es cuando se desea que el script comunique mensajes o resultados. Ello puede lograrse con la función `mprintf()`. A diferencia del comando `disp()`, que muestra el contenido de una variable con un formato estándar,[17] `mprintf()` permite presentar el contenido con un formato personalizado. La sintaxis básica es

```
mprintf(mensaje);
```

[17] En el caso de cadenas, `disp()` las presenta encerradas entre comillas, lo cual es bueno para indicar el tipo de variable pero es inadmisible para comunicar una información.

donde `mensaje` es una cadena. Existen ciertos códigos de control que pueden utilizarse, por ejemplo `\n` permite agregar un salto de línea. Una forma más avanzada es incluir el contenido de una o más variables con especificación del formato. La sintaxis es

```
mprintf(formato, x1, ..., xn);
```

donde `formato` es una cadena que contiene texto y códigos de control que representan las sucesivas variables `x1`, ..., `xn` con sus respectivos formatos. Los códigos de control van siempre precedidos por `%` (si se quiere realmente presentar el carácter `%` debe escribirse `%%`). Algunos ejemplos de códigos de control se muestran en la tabla siguiente:

Código	Formato
`%i`	Número entero
`%f`	Número decimal de punto fijo
`%e`	Número decimal en notación científica
`%x`	Número hexadecimal
`%s`	Cadena

Entre el símbolo `%` y el carácter que define el código puede insertarse una cifra que indica la cantidad mínima de caracteres que ocupará la variable (completando con espacios si ocupa menos) y luego la *precisión*, indicada por un punto y otra cifra. En el caso de números enteros, esta cifra representa la cantidad mínima de cifras mostradas, incluyendo ceros a la izquierda si el número real de cifras es menor. En el caso de decimales, indica la cantidad de cifras decimales. Y en el caso de cadenas, expresa la cantidad de caracteres de la cadena que se mostrarán. Por ejemplo:

```
a = 0.23;
f = 1000;
x = "correcto";
mprintf("a = %.1f\nf = %i Hz\nResultado: %s", a, f, x);
```

devuelve

```
a = 0.2
f = 1000 Hz
Resultado: correcto
```

Obsérvese que para la variable `a` se especificó sólo una cifra decimal. También se pudo agregar la unidad Hz a la presentación de la variable `f`, lo cual sería imposible si sólo se mostrara la variable usando `disp()`.

Escribiendo `help printf_conversion` se puede obtener una lista más completa de todas las opciones de códigos de control.

33 Control de aplicaciones por línea de comandos

En Windows es posible correr aplicaciones externas por línea de comandos a través de la función `dos()`. La sintaxis es:

```
bOK = dos(comandos)
[Y, bOK] = dos(comandos)
[Y, bOK, exitcode] = dos(comandos)
```

donde `comandos` es una cadena que contiene una línea de comandos, `bOK` es una variable booleana cuyo valor es `%T` (verdadero) si la ejecución no produjo errores, `Y` es una columna de cadenas que representan la salida estándar (incluyendo errores) del intérprete de línea de comandos cmd.exe, y `exitcode` es el código devuelto por cmd.exe al terminar la ejecución. La función inicia una ventana del intérprete de comandos y ejecuta la línea de comandos, y una vez completada la ejecución cierra la ventana.

Debe notarse que la función no permite ejecutar más de una línea de comandos, pero es posible utilizar el operador `&&` para concatenar en una sola línea varios comandos.

Por ejemplo, podríamos utilizar el códec (codificador/decodificador) `flac.exe` para decodificar un archivo de sonido comprimido con formato flac y convertirlo en wav para luego abrirlo mediante `wavread()`. Para ello, si el ejecutable `flac.exe`[18] se encuentra instalado en el directorio `d:\flac` y queremos convertir el archivo `sonido.flac` que se encuentra en `e:\sonidos\`, entonces el código

```
dos("d: && cd d:\flac && flac –d e:\sonidos\sonido.flac")
```

decodifica el archivo y lo guarda como `sonido.wav` en el mismo directorio. El primer comando cambia a la unidad `d:`, el segundo cambia al directorio que contiene el códec y el tercero aplica el códec al archivo deseado.

El uso de esta función desde ya requiere investigar las opciones que ofrece la función que queremos ejecutar, así como su sintaxis. Esto puede hacerse desde una ventana de línea de comandos ingresando `help función` o `función /?`, donde `función` es el nombre de la función o comando a ejecutar.

34 Creación de interfaces gráficas de usuario (GUI)

En Scilab hay una serie de elementos asociados al concepto de figura que permiten agregar funcionalidad de interfaz gráfica a un script o función. Si bien el resultado no es un programa autónomo ya que su funcionamiento requiere tener Scilab instalado y en ejecución, sí permite un control gráfico con características propias diseñadas por el usuario.

Las funciones básicas que permiten diseñar una interfaz gráfica a partir de una figura son `uimenu()` y `uicontrol()`. La función `uimenu()` permite agregar menús en la barra de menús. La función `uicontrol()` permite agregar a una figura controles de diferentes tipos o estilos, por ejemplo botones que desencadenan acciones, barras deslizantes, menús emergentes.

El diseño de una interfaz gráfica comienza por la creación de una ventana gráfica o figura. Al crear una figura mediante `scf()`, Scilab incluye por defecto ciertos elementos, por ejemplo, algunos menús y una barra de herramientas. Al crear nuestra pro-

[18] El códec de FLAC puede encontrarse en https://xiph.org/flac/

pia GUI será conveniente que la figura esté completamente vacía para evitar que contenga menús o herramientas propios de Scilab. Ello puede lograrse mediante el comando `figure()`, cuya sintaxis general incluye una serie de pares de argumentos donde el primer argumento de cada par es el nombre de una propiedad y el segundo el valor de la propiedad.[19] En nuestro caso podemos, por ejemplo, escribir

```
fig = figure("figure_name", "Mi GUI", ..
             "figure_id", 1, ..
             "menubar", "none", ..
             "toolbar_visible", "off", ..
             "dockable", "off", ..
             "infobar_visible", "off",..
             "default_axes", "off", ..
             "Position", [150 150 450 410], ..
             "BackgroundColor", [0.9 0.9 0.9], ..
             "resize", "off", ..
             "Tag", "Mi_GUI_figure");
```

Para facilitar la lectura se han separado los pares *nombre de propiedad* / *valor de la propiedad* en líneas diferentes (con puntos suspensivos), aunque ello no es imprescindible. La primera línea establece el nombre de la figura, que aparecerá en la barra superior (en este caso, `"Mi GUI"`). La segunda asigna el identificador numérico de la figura. Luego se eliminan la barra de menús, la barra de herramientas, la barra de acoplamiento (la cual permite, en caso de estar habilitada, integrar la ventana a la interfaz principal de Scilab) y la barra de información ubicada al pie de la ventana. Luego se quitan los ejes por defecto, se establece la posición y el color de fondo (un vector con valores entre 0 y 1 para los colores rojo, verde y azul) y se quita la posibilidad de redimensionar la ventana arrastrando con el mouse sus bordes (lo que podría llegar a ocultar botones u otros elementos). Por último se asigna una etiqueta a la figura. El resultado es una ventana vacía, como la de la figura 21.

La variable `fig` devuelta por la función es una estructura de datos asociada a la figura que se denomina *handle* (*mango*). Cada objeto gráfico posee un *handle* que contiene los valores de diversas propiedades del objeto.

Ahora podemos empezar a agregar menús. Para ello utilizaremos el comando `uimenu()`, cuyos argumentos siguen la misma lógica de pares *nombre de propiedad* / *valor de la propiedad*.[20] La siguiente línea crea un menú llamado `"Menú1"`:

```
menu1 = uimenu("Parent", fig, "Label", "Menú1");
```

Los dos primeros argumentos establecen que el menú aparecerá en la figura cuyo *handle* es `fig`.[21] Los otros dos indican que la etiqueta asignada al menú (la que aparecerá en la barra de menús y sobre la que se hará clic para abrir el menú) es `"Menú1"`.

---

[19] El listado de los nombres de las propiedades disponibles y su interpretación puede consultarse mediante `doc figure_properties`.

[20] El listado completo de las propiedades disponibles para `uimenu()` pueden consultarse mediante `doc uimenu_properties`

[21] Los objetos gráficos forman parte de una jerarquía donde hay un "progenitor" (en inglés "Parent") y "descendientes" (en inglés "children"). En este caso los primeros argumentos indican que la figura cuyo *handle* es `fig` es progenitora del menú cuyo *handle* es `menu1`.

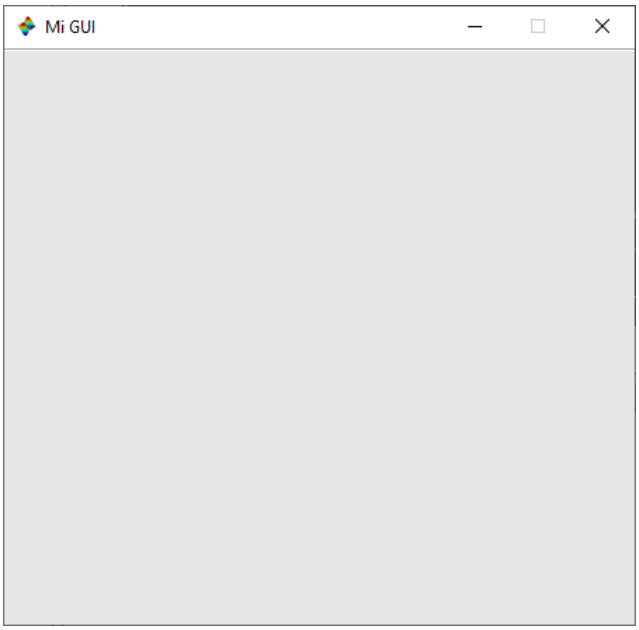

Figura 21. Ventana vacía inicial a partir de la cual se construirá la interfaz gráfica titulada Mi GUI.

Hasta el momento el menú está vacío. Para agregar un ítem bastará crear un menú dentro de éste:

```
menu1_close = uimenu("Parent", menu1, ..
                     "Label", "Close", ..
                     "callback", "close_GUI();");
```

Los primeros dos argumentos establecen que `menu1_close` aparecerá en el menú cuyo *handle* es `menu1`. Los dos siguientes indican que el ítem del menú correspondiente tendrá la etiqueta `"Close"` (ver figura 22). Los últimos dos indican que al hacer clic en ese ítem del menú se ejecutará la cadena `"close_GUI();"`, que en este caso invoca a la función `close_GUI()`, que deberá estar previamente definida, por ejemplo:

```
function close_GUI()
   close(fig.figure_id);
endfunction
```

o, también,

```
function close_GUI()
   delete(get("Mi_GUI_figure"));
endfunction
```

En la primera variante, dentro del *handle* de la figura se invoca el campo que contiene su identificador numérico (denominado `figure_id`). En la segunda variante se aplica la función `get` a la etiqueta de la figura, que devuelve el *handle* de la figura correspondiente. Al borrar el *handle* se borra la figura, lo cual equivale a cerrar la figura mediante el comando `close`.

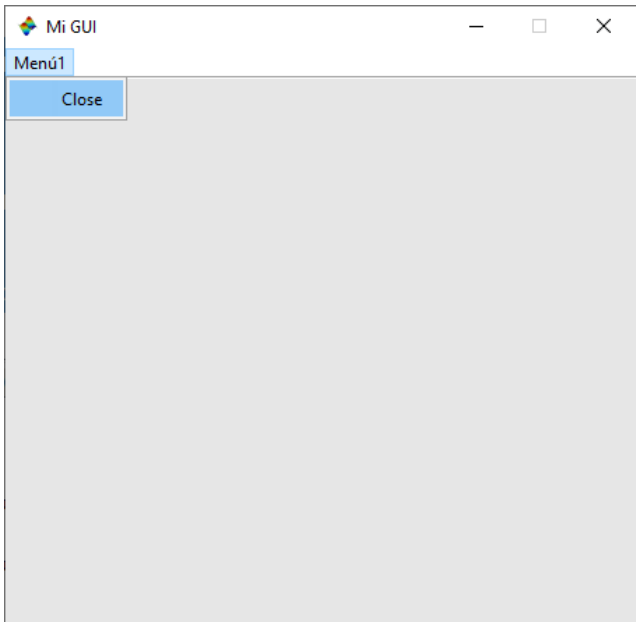

Figura 22. Incorporación de un menú con un ítem a una GUI. El menú ha sido abierto haciendo clic en él

En forma similar pueden agregarse otros ítems al menú `"Menú1"`, así como otros menús a la barra de menús, tales como `"Menú2"`, `"Menú3"`, etc.

Se pueden agregar elementos de control a la interfaz mediante `uicontrol()`. Estos elementos pueden ser botones, barras deslizantes, casillas de texto, listas seleccionables, menús emergentes, etc. La sintaxis general es

```
uicontrol(progenitor, nom_propiedad1, val_propiedad1, ...)
```

El argumento `progenitor` es el *handle* de la figura en la cual se desea colocar el control. Va seguido por pares *nombre de propiedad* / *valor de la propiedad*.[22] El siguiente código inserta un botón que permite reproducir un sonido previamente generado y asignado a la variable `y`:

```
play_button = uicontrol(fig, ..
                        "Position", [20 85 100 25], ..
                        "Style", "pushbutton", ..
                        "FontSize", 11, ..
                        "String", "Play", ..
                        "callback", "playsnd(y, Fs);");
```

La primera línea especifica que el control a generar residirá en la figura cuyo *handle* es `fig`. La segunda establece la posición y tamaño como un vector de 4 componentes. Las dos primeras son la abscisa y la ordenada del extremo inferior izquierdo del botón (el origen está en el extremo inferior izquierdo de la ventana) expresadas en píxeles. Las

[22] El listado de las propiedades disponibles para `uicontrol()` y sus descripciones pueden consultarse mediante `doc uicontrol_properties`. Sin embargo, una de las propiedades más básicas, `"Style"` y sus posibles valores se encuentran descritos en la documentación de la propia función, accesible mediante `doc uicontrol`.

dos últimas dos componentes indican el ancho y la altura del botón. En general estos datos geométricos se obtendrán por prueba y error. La tercera línea establece que el *estilo* del control es un botón a presionar (haciendo clic con el mouse sobre él). Luego siguen el tamaño de la tipografía a utilizar y el texto del botón (en este caso `"Play"`). La última línea establece que la acción a ejecutar al hacer clic sobre el botón será la reproducción de un sonido que debe estar previamente almacenado en la variable `y` con la tasa de muestreo `Fs`, aunque podría ejecutarse cualquier instrucción, script o función del usuario. En la figura 23 se muestra el resultado.

En forma similar pueden añadirse otros estilos de controles que pueden consultarse en la documentación de la función `uicontrol()`, así como gráficas y otros elementos.

35 Depuración

Existen algunas herramientas de interés para realizar la depuración (debugging) de los programas (tanto scripts como funciones) escritos en Scilab. La primera es la función `warning()`, con diferentes opciones para las advertencias. Así, `warning stop`, o `warning("stop")`, detiene la ejecución de un script cuando se produce un mensaje de advertencia, permitiendo detectar en qué instrucción se produjo.

También puede insertarse la instrucción `pause`, que interrumpe la ejecución de una función sin perder las variables internas de la función, lo cual permite examinarlas y así detectar si hay algún problema. Cuando esto sucede, la línea de comandos está precedida por `-1->` en lugar de `-->`. Puede volverse al espacio de trabajo usando el comando `return`. También puede utilizarse `abort`, pero en este caso se interrumpe la ejecución de la función y se pierden las variables internas.

Es posible ejecutar un script completo mediante el ícono de ejecución (un triángulo) en la barra de herramientas, o parcialmente, para lo cual basta seleccionar el grupo de instrucciones deseado, hacer clic con el botón secundario (derecho) del mouse y seleccionar la opción de evaluar la selección.

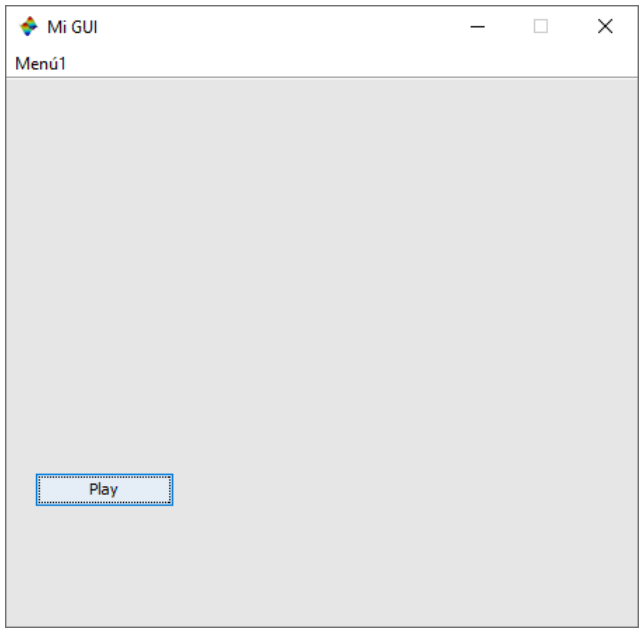

Figura 23. Incorporación de un botón a una GUI.

Cuando se requiere evaluar la eficiencia computacional de un algoritmo, se pueden utilizar las instrucciones `tic` y `toc` (los nombres son onomatopeyas en inglés del sonido de un reloj mecánico). Si la primera se ubica antes de un grupo de instrucciones y la segunda al final, al ejecutar las instrucciones se obtiene el tiempo en segundos insumido en la ejecución. Esto es útil para comparar diferentes versiones de un mismo algoritmo. En general se observa que los procesos vectorizados (es decir, que aprovechan las operaciones que actúan directamente sobre vectores o arreglos) son más eficientes que los que utilizan métodos iterativos basados en lazos (tales como los lazos `for` o `while`).

36 *Discourse*: foro de discusión

Es inevitable que en algún momento el usuario, especialmente el principiante pero en ocasiones también el experto, se enfrente a algún problema que no puede resolver en forma simple. En algunos casos se debe al desconocimiento de alguna función específica pero en otros se origina en una falla del programa (*bug*) o en alguna prestación aún no implementada. En tales situaciones es útil recurrir al foro de la comunidad de Scilab, llamado *Discourse*. Se puede acceder al mismo desde la pestaña About / Community de la página principal del sitio de Scilab o directamente en https//:scilab.discourse.group. Una vez allí (figura 24) es posible hacer búsquedas simples haciendo clic en el ícono en forma de lupa (figura 24, 1) ingresando unos pocos términos (en inglés). Alternativamente, haciendo clic en el ícono dentro de la casilla de búsqueda que se abre se puede acceder a una búsqueda avanzada (*Advanced filters*). En cualquiera de estos casos se accede a consultas realizadas anteriormente sobre tópicos similares, donde es bastante probable encontrar repuestas útiles.

La alternativa más completa es poder participar del foro enviando preguntas propias. Para ello es preciso registrarse (figura 24, 2) haciendo clic en el ícono *Sign up*. Se abre un formulario que solicita dirección de Email, nombre de usuario, contraseña y, opcionalmente, nombre. Al oprimir el botón *Sign up* se completa el proceso. Luego se recibirá un correo electrónico de confirmación. A continuación es posible iniciar

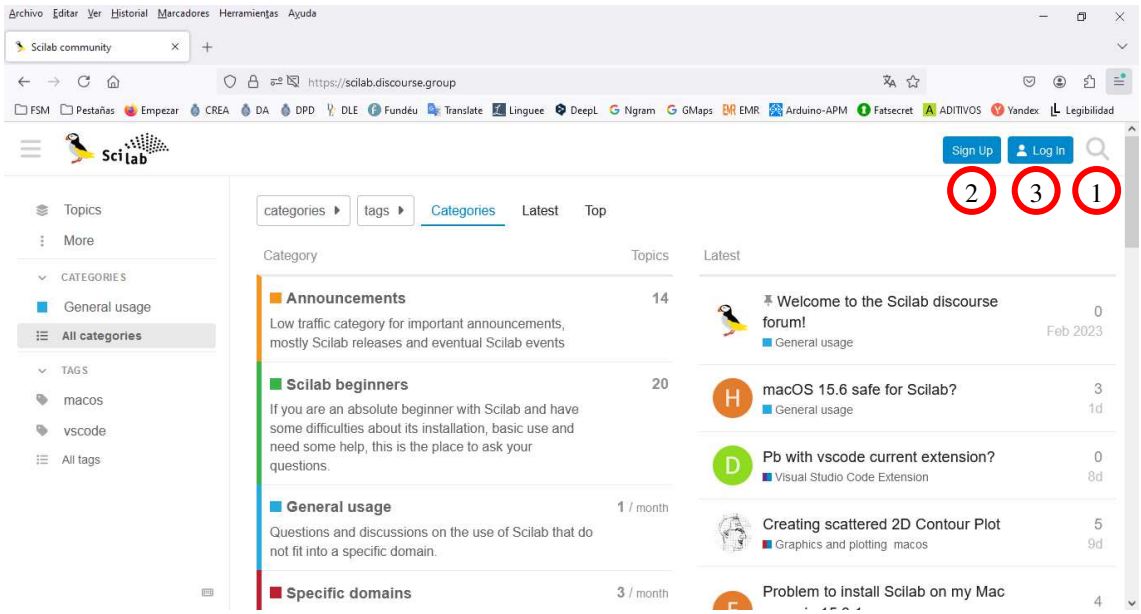

Figura 24. Página inicial del foro *Discourse* de Scilab. 1, ícono de búsquedas. 2, botón para registrarse. 3, botón para iniciar sesión.

sesión (figura 24, 3), lo cual permite, usando el botón *New topic*, ingresar un mensaje en el formulario que se abre (figura 25). Aunque el resultado dependerá de la voluntad y los conocimientos de los miembros de la comunidad, en general se recibirán algunas respuestas que permiten resolver el problema.

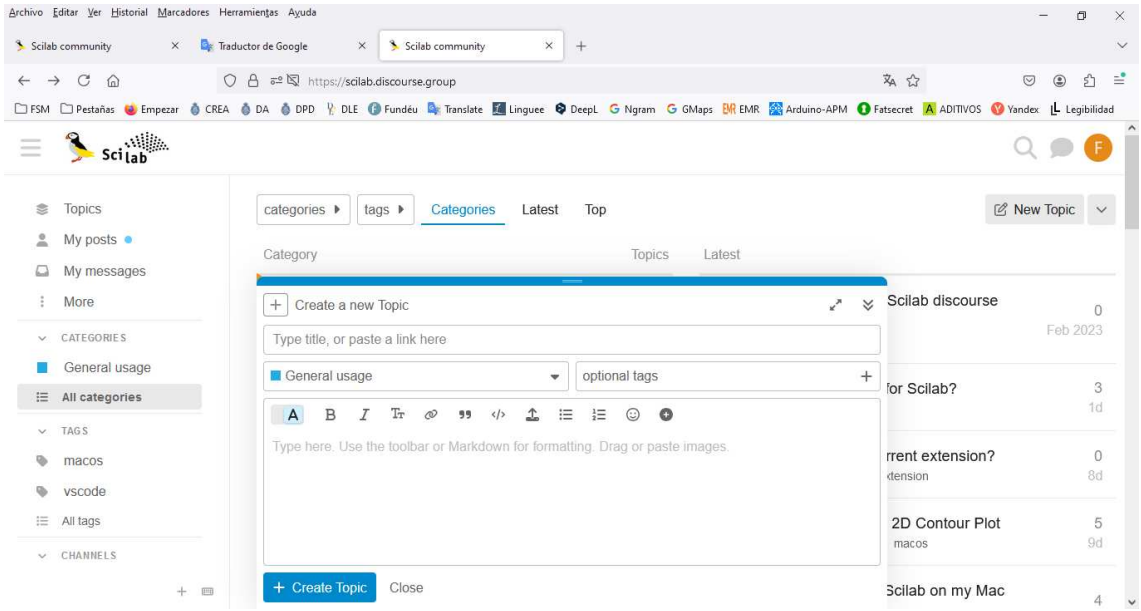

Figura 25. Formulario de *Discourse* de Scilab para enviar un mensaje al foro.

Referencias

IEEE Std 754-2008 *IEEE Standard for Floating-Point Arithmetic*

Scilab Home Page. https://www.scilab.org/

Scilab Online help. https://help.scilab.org/docs/2025.0.0/en_US/index.html

Colaboradores de Wikipedia (2019). *Algoritmo de Horner* [en línea]. Wikipedia, La enciclopedia libre, [fecha de consulta: 22 de enero de 2020]. Disponible en: <https://es.wikipedia.org/w/index.php?title=Algoritmo_de_Horner&oldid=119488949>.

Colaboradores de Wikipedia (2022-06-28) *La*TeX [en línea]. Wikipedia, La enciclopedia libre, [fecha de consulta: 12 de agosto de 2022]. Disponible en: https://es.wikipedia.org/wiki/LaTeX

LaTeX. (2022-07-14). Wikibooks, The Free Textbook Project. Fecha de consulta 19:37, August 12, 2022 de https://en.wikibooks.org/w/index.php?title=LaTeX&oldid=4084298. También disponible en PDF en https://upload.wikimedia.org/wikipedia/commons/2/2d/LaTeX.pdf