EBook Gratis

APRENDIZAJE MATLAB Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#matlab

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: matlab-language

It is an unofficial and free MATLAB Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official MATLAB Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con MATLAB Language

Versiones

Versión	Lanzamiento	Fecha de lanzamiento
1.0		1984-01-01
2		1986-01-01
3		1987-01-01
3.5		1990-01-01
4		1992-01-01
4.2c		1994-01-01
5.0	Volumen 8	1996-12-01
5.1	Volumen 9	1997-05-01
5.1.1	R9.1	1997-05-02
5.2	R10	1998-03-01
5.2.1	R10.1	1998-03-02
5.3	R11	1999-01-01
5.3.1	R11.1	1999-11-01
6.0	R12	2000-11-01
6.1	R12.1	2001-06-01
6.5	R13	2002-06-01
6.5.1	R13SP2	2003-01-01
6.5.2	R13SP2	2003-01-02
7	R14	2006-06-01
7.0.4	R14SP1	2004-10-01
7.1	R14SP3	2005-08-01

Versión	Lanzamiento	Fecha de lanzamiento
7.2	R2006a	2006-03-01
7.3	R2006b	2006-09-01
7.4	R2007a	2007-03-01
7.5	R2007b	2007-09-01
7.6	R2008a	2008-03-01
7.7	R2008b	2008-09-01
7.8	R2009a	2009-03-01
7.9	R2009b	2009-09-01
7.10	R2010a	2010-03-01
7.11	R2010b	2010-09-01
7.12	R2011a	2011-03-01
7.13	R2011b	2011-09-01
7.14	R2012a	2012-03-01
8.0	R2012b	2012-09-01
8.1	R2013a	2013-03-01
8.2	R2013b	2013-09-01
8.3	R2014a	2014-03-01
8.4	R2014b	2014-09-01
8.5	R2015a	2015-03-01
8.6	R2015b	2015-09-01
9.0	R2016a	2016-03-01
9.1	R2016b	2016-09-14
9.2	R2017a	2017-03-08

Vea también: Historial de lanzamientos de MATLAB en Wikipedia .

Examples

Hola Mundo

Abra un nuevo documento en blanco en el editor de MATLAB (en las versiones recientes de MATLAB, haga clic en la pestaña Inicio de la barra de herramientas y haga clic en Nueva secuencia de comandos). El atajo de teclado predeterminado para crear un nuevo script es `Ctrl-n` .

Alternativamente, al escribir `edit myscriptname.m` se abrirá el archivo `myscriptname.m` para editarlo, u ofrecerá crear el archivo si no existe en la ruta de MATLAB.

En el editor, escriba lo siguiente:

```
disp('Hello, World!');
```

Seleccione la pestaña Editor de la barra de herramientas y haga clic en Guardar como. Guarde el documento en un archivo en el directorio actual llamado `helloworld.m` . Al guardar un archivo sin título aparecerá un cuadro de diálogo para nombrar el archivo.

En la ventana de comandos de MATLAB, escriba lo siguiente:

```
>> helloworld
```

Debería ver la siguiente respuesta en la ventana de comandos de MATLAB:

```
Hello, World!
```

Vemos que, en la ventana de comandos, podemos escribir los nombres de las funciones o los archivos de script que hemos escrito, o que se incluyen con MATLAB, para ejecutarlos.

Aquí, hemos ejecutado el script 'helloworld'. Observe que escribir la extensión (`.m`) no es necesario. Las instrucciones contenidas en el archivo de comandos son ejecutadas por MATLAB, aquí se imprime '¡Hola, mundo!' utilizando la función `disp` .

Los archivos de script se pueden escribir de esta manera para guardar una serie de comandos para su uso (re) posterior.

Matrices y matrices

En MATLAB, el tipo de datos más básico es la matriz numérica. Puede ser un escalar, un vector 1-D, una matriz 2-D o una matriz multidimensional ND.

```
% a 1-by-1 scalar value
x = 1;
```

Para crear un vector de fila, ingrese los elementos entre corchetes, separados por espacios o

comas:

```
% a 1-by-4 row vector
v = [1, 2, 3, 4];
v = [1 2 3 4];
```

Para crear un vector de columna, separe los elementos con punto y coma:

```
% a 4-by-1 column vector
v = [1; 2; 3; 4];
```

Para crear una matriz, ingresamos las filas como antes separadas por puntos y comas:

```
% a 2 row-by-4 column matrix
M = [1 2 3 4; 5 6 7 8];

% a 4 row-by-2 column matrix
M = [1 2; ...
     4 5; ...
     6 7; ...
     8 9];
```

Tenga en cuenta que no puede crear una matriz con un tamaño de fila / columna desigual. Todas las filas deben tener la misma longitud y todas las columnas deben tener la misma longitud:

```
% an unequal row / column matrix
M = [1 2 3 ; 4 5 6 7]; % This is not valid and will return an error

% another unequal row / column matrix
M = [1 2 3; ...
     4   5; ...
     6 7 8; ...
     9   10];     % This is not valid and will return an error
```

Para transponer un vector o una matriz, usamos el `.'` -operador, o el `'` operador para tomar su conjugado hermitiano, que es el conjugado complejo de su transposición. Para matrices reales, estas dos son las mismas:

```
% create a row vector and transpose it into a column vector
v = [1 2 3 4].';              % v is equal to [1; 2; 3; 4];

% create a 2-by-4 matrix and transpose it to get a 4-by-2 matrix
M = [1 2 3 4; 5 6 7 8].';     % M is equal to [1 5; 2 6; 3 7; 4 8]

% transpose a vector or matrix stored as a variable
A = [1 2; 3 4];
B = A.';                      % B is equal to [1 3; 2 4]
```

Para matrices de más de dos dimensiones, no hay sintaxis de lenguaje directo para ingresarlas literalmente. En su lugar, debemos usar funciones para construirlas (como `ones` , `zeros` , `rand`) o mediante la manipulación de otras matrices (usando funciones como `cat` , `reshape` , `permute`). Algunos ejemplos:

```
% a 5-by-2-by-4-by-3 array (4-dimensions)
arr = ones(5, 2, 4, 3);

% a 2-by-3-by-2 array (3-dimensions)
arr = cat(3, [1 2 3; 4 5 6], [7 8 9; 0 1 2]);

% a 5-by-4-by-3-by-2 (4-dimensions)
arr = reshape(1:120, [5 4 3 2]);
```

Matrices y matrices de indexación.

MATLAB permite varios métodos para indexar (acceder) elementos de matrices y matrices:

- **Indización de subíndices** : donde se especifica la posición de los elementos que desea en cada dimensión de la matriz por separado.
- **Indexación lineal** : donde la matriz se trata como un vector, sin importar sus dimensiones. Eso significa que usted especifica cada posición en la matriz con un solo número.
- **Indización lógica** : donde se utiliza una matriz lógica (y una matriz de valores `true` y `false`) con las dimensiones idénticas de la matriz que está intentando indexar como una máscara para especificar qué valor devolver.

Estos tres métodos ahora se explican con más detalle utilizando la siguiente matriz `M` 3 por 3 como ejemplo:

```
>> M = magic(3)

ans =

       8    1    6
       3    5    7
       4    9    2
```

Indexación de subíndices

El método más directo para acceder a un elemento es especificar su índice fila-columna. Por ejemplo, accediendo al elemento en la segunda fila y tercera columna:

```
>> M(2, 3)

ans =

      7
```

El número de subíndices proporcionados coincide exactamente con el número de dimensiones que `M` tiene (dos en este ejemplo).

Tenga en cuenta que el orden de los subíndices es el mismo que el de la convención matemática: el índice de fila es el primero. Además, los índices de MATLAB **comienzan con** `1` y **no con** `0` como la mayoría de los lenguajes de programación.

Puede indexar varios elementos a la vez pasando un vector para cada coordenada en lugar de un

solo número. Por ejemplo, para obtener la segunda fila completa, podemos especificar que queremos las columnas primera, segunda y tercera:

```
>> M(2, [1,2,3])

ans =

       3    5    7
```

En MATLAB, el vector `[1,2,3]` se crea más fácilmente usando el operador de dos puntos, es decir, `1:3` . Puedes usar esto también en la indexación. Para seleccionar una fila (o columna) completa, MATLAB proporciona un acceso directo permitiéndole simplemente especificar `:` Por ejemplo, el siguiente código también devolverá la segunda fila completa

```
>> M(2, :)

ans =

       3    5    7
```

MATLAB también proporciona un acceso directo para especificar el último elemento de una dimensión en la forma de la palabra clave `end` . La palabra clave `end` funcionará exactamente como si fuera el número del último elemento en esa dimensión. Entonces, si desea que todas las columnas de la columna `2` a la última columna, puede usar escriba lo siguiente:

```
>> M(2, 2:end)

ans =

       5    7
```

La indexación de subíndices puede ser restrictiva ya que no permitirá extraer valores individuales de diferentes columnas y filas; se extraerá la combinación de todas las filas y columnas.

```
>> M([2,3], [1,3])
ans =

       3    7
       4    2
```

Por ejemplo, la indexación de subíndices no puede extraer solo los elementos `M(2,1)` o `M(3,3)` . Para ello debemos tener en cuenta la indexación lineal.

Indexación lineal

MATLAB le permite tratar las matrices n-dimensionales como matrices unidimensionales cuando indexa usando solo una dimensión. Puedes acceder directamente al primer elemento:

```
>> M(1)

ans =
```

```
       8
```

Tenga en cuenta que las matrices se almacenan en orden mayor de columnas en MATLAB, lo que significa que usted accede a los elementos bajando primero las columnas. Entonces, `M(2)` es el segundo elemento de la primera columna, que es `3` y `M(4)` será el primer elemento de la segunda columna, es decir,

```
>> M(4)

ans =

        1
```

Existen funciones integradas en MATLAB para convertir los índices de subíndices en índices lineales, y viceversa: `sub2ind` e `ind2sub` respectivamente. Puede convertir manualmente los subíndices (`r` , `c`) en un índice lineal mediante

```
idx = r + (c-1)*size(M,1)
```

Para entender esto, si estamos en la primera columna, entonces el índice lineal será simplemente el índice de la fila. La fórmula anterior es válida para esto porque para `c == 1` , `(c-1) == 0` . En las siguientes columnas, el índice lineal es el número de fila más todas las filas de las columnas anteriores.

Tenga en cuenta que la palabra clave `end` todavía se aplica y ahora se refiere al último elemento de la matriz, es decir, `M(end) == M(end, end) == 2` .

También puede indexar múltiples elementos utilizando la indexación lineal. Tenga en cuenta que si lo hace, la matriz devuelta tendrá la misma forma que la matriz de vectores de índice.

`M(2:4)` devuelve un vector de fila porque `2:4` representa el vector de fila `[2,3,4]` :

```
>> M(2:4)

ans =

        3    4    1
```

Como otro ejemplo, `M([1,2;3,4])` devuelve una matriz de 2 por 2 porque `[1,2;3,4]` es una matriz de 2 por 2. Vea el siguiente código para convencerse:

```
>> M([1,2;3,4])

ans =

       8        3
       4        1
```

Tenga en cuenta que la indexación con `:` alone *siempre* devolverá un vector de columna:

```
>> M(:)

ans =

        8
        3
        4
        1
        5
        9
        6
        7
        2
```

Este ejemplo también ilustra el orden en que MATLAB devuelve elementos cuando se utiliza la indexación lineal.

Indexación lógica

El tercer método de indexación es usar una matriz lógica, es decir, una matriz que contenga solo valores `true` o `false` , como una máscara para filtrar los elementos que no desea. Por ejemplo, si queremos encontrar todos los elementos de `M` que son mayores que `5` podemos usar la matriz lógica

```
>> M > 5

ans =

    1    0    1
    0    0    1
    0    1    0
```

para indexar `M` y devolver solo los valores que son mayores que `5` como sigue:

```
>> M(M > 5)

ans =

        8
        9
        6
        7
```

Si desea que estos números permanezcan en su lugar (es decir, mantenga la forma de la matriz), puede asignar el complemento lógico

```
>> M(~(M > 5)) = NaN

ans =

    8      NaN    6
    NaN    NaN    7
    NaN    9      Nan
```

Podemos reducir los bloques de código complicados que contienen `if` `for` declaraciones mediante la indexación lógica.

Tome lo no vectorizado (ya reducido a un solo bucle mediante el uso de índices lineales):

```
for elem = 1:numel(M)
  if M(elem) > 5
    M(elem) = M(elem) - 2;
  end
end
```

Esto se puede reducir al siguiente código utilizando la indexación lógica:

```
idx = M > 5;
M(idx) = M(idx) - 2;
```

O incluso más corto:

```
M(M > 5) = M(M > 5) - 2;
```

Más sobre indexación.

Matrices de dimensiones superiores

Todos los métodos mencionados anteriormente se generalizan en n-dimensiones. Si usamos la matriz tridimensional `M3 = rand(3,3,3)` como ejemplo, entonces puede acceder a todas las filas y columnas del segundo segmento de la tercera dimensión escribiendo

```
>> M(:,:,2)
```

Puede acceder al primer elemento de la segunda porción utilizando la indexación lineal. La indexación lineal solo se moverá a la segunda división después de todas las filas y todas las columnas de la primera división. Así que el índice lineal para ese elemento es

```
>> M(size(M,1)*size(M,2)+1)
```

De hecho, en MATLAB, *cada* matriz es n-dimensional: resulta que el tamaño de la mayoría de las otras n dimensiones es uno. Entonces, si `a = 2` entonces `a(1) == 2` (como es de esperar), *pero también* `a(1, 1) == 2` , al igual que `a(1, 1, 1) == 2` , `a(1, 1, 1, ..., 1) == 2` y así sucesivamente. Estas dimensiones "adicionales" (de tamaño `1`), se denominan *dimensiones singleton* . El comando `squeeze` los eliminará, y uno puede usar `permute` para intercambiar el orden de las dimensiones (e introducir cotas singleton si es necesario).

Una matriz n-dimensional también se puede indexar utilizando un m subíndices (donde m <= n). La regla es que los primeros subíndices m-1 se comportan normalmente, mientras que el último (m'th) hace referencia a las dimensiones restantes (n-m + 1), al igual que un índice lineal haría referencia a una dimensión (n-m + 1) formación. Aquí hay un ejemplo:

```
>> M = reshape(1:24,[2,3,4]);
>> M(1,1)
ans =
     1
>> M(1,10)
ans =
    19
>> M(:,:)
ans =
     1     3     5     7     9    11    13    15    17    19    21    23
     2     4     6     8    10    12    14    16    18    20    22    24
```

Volviendo rangos de elementos.

Con la indexación de subíndices, si especifica más de un elemento en más de una dimensión, MATLAB devuelve cada posible par de coordenadas. Por ejemplo, si prueba M ([1,2], [1,3]), MATLAB devolverá `M(1,1)` y `M(2,3)` pero **también** devolverá `M(1,3)` y `M(2,1)` . Esto puede parecer poco intuitivo cuando busca los elementos para una lista de pares de coordenadas, pero considere el ejemplo de una matriz más grande, `A = rand(20)` (la nota `A` es ahora `20` -by `20`), donde desea obtener la cuadrante superior derecho. En este caso, en lugar de tener que especificar cada par de coordenadas en ese cuadrante (y este sería `100` pares), simplemente especifique las `10` filas y las `10` columnas que desea, así que `A(1:10, 11:end)` . *Cortar* una matriz como esta es mucho más común que requerir una lista de pares de coordenadas.

En el caso de que desee obtener una lista de pares de coordenadas, la solución más sencilla es convertirla a la indexación lineal. Considere el problema donde tiene un vector de índices de columna que desea que se devuelvan, donde cada fila del vector contiene el número de columna que desea que se devuelva para la fila *correspondiente* de la matriz. Por ejemplo

```
colIdx = [3;2;1]
```

Entonces, en este caso, realmente quieres recuperar los elementos en `(1,3)` , `(2,2)` y `(3,1)` . Así que usando la indexación lineal:

```
>> colIdx = [3;2;1];
>> rowIdx = 1:length(colIdx);
>> idx = sub2ind(size(M), rowIdx, colIdx);
>> M(idx)

ans =

        6    5    4
```

Devolviendo un elemento varias veces

Con el subíndice y la indexación lineal, también puede devolver un elemento varias veces al repetir su índice, de modo que

```
>> M([1,1,1,2,2,2])

ans =
```

```
        8     8     8     3     3     3
```

Puede usar esto para duplicar filas y columnas completas, por ejemplo, para repetir la primera fila y la última columna

```
>> M([1, 1:end], [1:end, end])

ans =

        8     1     6     6
        8     1     6     6
        3     5     7     7
        4     9     2     2
```

Para más información, ver aquí .

Ayudandote

MATLAB viene con muchos scripts y funciones incorporados que van desde la simple multiplicación de herramientas de reconocimiento de imágenes. Para obtener información sobre una función que desea utilizar, escriba: `help functionname` en la línea de comandos. Tomemos como ejemplo la función de `help` .

La información sobre cómo usarla se puede obtener escribiendo:

```
>> help help
```

en la ventana de comandos. Esto devolverá información del uso de la `help` de la función. Si la información que está buscando aún no está clara, puede probar la página de **documentación** de la función. Simplemente escriba:

```
>> doc help
```

en la ventana de comandos. Esto abrirá la documentación navegable en la página para obtener `help` función `help` proporciona toda la información que necesita para comprender cómo funciona la "ayuda".

Este procedimiento funciona para todas las funciones y símbolos incorporados.

Al desarrollar sus propias funciones, puede dejar que tengan su propia sección de ayuda agregando comentarios en la parte superior del archivo de funciones o justo después de la declaración de la función.

Ejemplo para una función simple `multiplyby2` guardada en el archivo `multiplyby2.m`

```
function [prod]=multiplyby2(num)
% function MULTIPLYBY2 accepts a numeric matrix NUM and returns output PROD
% such that all numbers are multiplied by 2

    prod=num*2;
end
```

o

```
% function MULTIPLYBY2 accepts a numeric matrix NUM and returns output PROD
% such that all numbers are multiplied by 2

function [prod]=multiplyby2(num)
    prod=num*2;
end
```

Esto es muy útil cuando recupera su código semanas / meses / años después de haberlo escrito.

La `help` y la función `doc` proporcionan mucha información, aprender a usar esas funciones lo ayudará a progresar rápidamente y usar MATLAB de manera eficiente.

Lectura de entrada y escritura de salida

Al igual que todo lenguaje de programación, Matlab está diseñado para leer y escribir en una gran variedad de formatos. La biblioteca nativa admite una gran cantidad de formatos de texto, imagen, video, audio y datos con más formatos incluidos en cada actualización de versión. Consulte aquí para ver la lista completa de formatos de archivo admitidos y qué función usar para importarlos.

Antes de intentar cargar en su archivo, debe preguntarse en qué desea que se conviertan los datos y cómo espera que la computadora organice los datos para usted. Digamos que tienes un archivo txt / csv en el siguiente formato:

```
Fruit,TotalUnits,UnitsLeftAfterSale,SellingPricePerUnit
Apples,200,67,$0.14
Bananas,300,172,$0.11
Pineapple,50,12,$1.74
```

Podemos ver que la primera columna está en el formato de Cadenas, mientras que la segunda, la tercera es numérica, la última columna está en la forma de moneda. Digamos que queremos encontrar la cantidad de ingresos que obtuvimos hoy con Matlab y primero queremos cargar en este archivo txt / csv. Después de comprobar el enlace, podemos ver que la cadena y el tipo numérico de archivos txt son manejados por `textscan` . Así podríamos intentar:

```
fileID = fopen('dir/test.txt'); %Load file from dir
C = textscan(fileID,'%s %f %f %s','Delimiter',',','HeaderLines',1); %Parse in the txt/csv
```

donde `%s` sugiere que el elemento es un tipo de cadena, `%f` sugiere que el elemento es un tipo flotante, y que el archivo está delimitado por ",". La opción HeaderLines le pide a Matlab que se salte las primeras N líneas mientras que la 1 inmediatamente después significa que se salte la primera línea (la línea del encabezado).

Ahora C es la información que hemos cargado, que tiene la forma de una matriz de celdas de 4 celdas, cada una de las cuales contiene la columna de datos en el archivo txt / csv.

Entonces, primero queremos calcular cuántas frutas vendimos hoy al restar la tercera columna de la segunda columna, esto se puede hacer de la siguiente manera:

```
sold = C{2} - C{3}; %C{2} gives the elements inside the second cell (or the second column)
```

Ahora queremos multiplicar este vector por el Precio por unidad, así que primero debemos convertir esa columna de Cadenas en una columna de Números, luego convertirla en una Matriz Numérica usando el `cell2mat` de Matlab. `cell2mat` primero que debemos hacer es eliminar. fuera del signo "$", hay muchas maneras de hacer esto. La forma más directa es usando un regex simple:

```
D = cellfun(@(x)(str2num(regexprep(x, '\$',''))), C{4}, 'UniformOutput', false);%cellfun
allows us to avoid looping through each element in the cell.
```

O puedes usar un bucle:

```
for t=1:size(C{4},1)
   D{t} = str2num(regexprep(C{4}{t}, '\$',''));
end

E = cell2mat(D)% converts the cell array into a Matrix
```

La función `str2num` convierte la cadena con los signos "$" en tipos numéricos y `cell2mat` convierte la celda de elementos numéricos en una matriz de números.

Ahora podemos multiplicar las unidades vendidas por el costo por unidad:

```
revenue = sold .* E; %element-wise product is denoted by .* in Matlab

totalrevenue = sum(revenue);
```

Arrays celulares

Los elementos de la misma clase a menudo se pueden concatenar en arreglos (con algunas raras excepciones, por ejemplo, manejadores de funciones). Los escalares numéricos, por defecto de clase `double` , pueden almacenarse en una matriz.

```
>> A = [1, -2, 3.14, 4/5, 5^6; pi, inf, 7/0, nan, log(0)]
A =
   1.0e+04 *
    0.0001   -0.0002    0.0003    0.0001    1.5625
    0.0003       Inf       Inf       NaN      -Inf
```

Los caracteres, que son de clase `char` en MATLAB, también pueden almacenarse en una matriz usando una sintaxis similar. Dicha matriz es similar a una cadena en muchos otros lenguajes de programación.

```
>> s = ['MATLAB ','is ','fun']
s =
MATLAB is fun
```

Tenga en cuenta que a pesar de que ambos utilizan corchetes `[` y `]` , las clases de resultados son

diferentes. Por lo tanto, las operaciones que se pueden hacer en ellos también son diferentes.

```
>> whos
  Name      Size            Bytes  Class     Attributes

  A         2x5                80  double
  s         1x13               26  char
```

De hecho, la matriz `s` no es una matriz de las cadenas `'MATLAB '` , `'is '` y `'fun'` , es solo una cadena, una matriz de 13 caracteres. Obtendría los mismos resultados si estuviese definido por alguno de los siguientes:

```
>> s = ['MAT','LAB ','is f','u','n'];
>> s = ['M','A','T','L','A','B,' ','i','s',' ','f','u','n'];
```

Un vector MATLAB regular no le permite almacenar una mezcla de variables de diferentes clases o unas pocas cadenas diferentes. Aquí es donde la matriz de `cell` es útil. Esta es una matriz de celdas que cada una puede contener algún objeto MATLAB, cuya clase puede ser diferente en cada celda si es necesario. Use llaves `{` y `}` alrededor de los elementos para almacenar en una matriz de celdas.

```
>> C = {A; s}
C =
    [2x5 double]
    'MATLAB is fun'
>> whos C
  Name      Size            Bytes  Class    Attributes

  C         2x1               330  cell
```

Los objetos MATLAB estándar de cualquier clase se pueden almacenar juntos en una matriz de celdas. Tenga en cuenta que las matrices de celdas requieren más memoria para almacenar sus contenidos.

El acceso al contenido de una celda se realiza utilizando llaves `{` y `}` .

```
>> C{1}
ans =
   1.0e+04 *
    0.0001   -0.0002    0.0003    0.0001    1.5625
    0.0003       Inf       Inf       NaN      -Inf
```

Tenga en cuenta que `C(1)` es diferente de `C{1}` . Mientras que el último devuelve el contenido de la celda (y tiene una clase `double` en el ejemplo), el primero devuelve una matriz de celdas que es una sub-matriz de `C` De manera similar, si `D` fuera una matriz de 10 por 5 celdas, entonces `D(4:8,1:3)` devolvería una sub-matriz de `D` cuyo tamaño es 5 por 3 y cuya clase es `cell` . Y la sintaxis `C{1:2}` no tiene un solo objeto devuelto, sino que devuelve 2 objetos diferentes (similar a una función MATLAB con múltiples valores de retorno):

```
>> [x,y] = C{1:2}
x =
```

```
                         1                         -2                      3.14
0.8                    15625
          3.14159265358979                        Inf                       Inf
NaN                     -Inf
y =
MATLAB is fun
```

Scripts y Funciones

El código MATLAB se puede guardar en archivos m para reutilizarlo. Los archivos m tienen la extensión `.m` que se asocia automáticamente con MATLAB. Un archivo-M puede contener una secuencia de comandos o funciones.

Guiones

Los scripts son simplemente archivos de programa que ejecutan una serie de comandos MATLAB en un orden predefinido.

Los scripts no aceptan entradas, ni los scripts devuelven resultados. Funcionalmente, los scripts son equivalentes a escribir comandos directamente en la ventana de comandos de MATLAB y poder reproducirlos.

Un ejemplo de un script:

```
length = 10;
width = 3;
area = length * width;
```

Este script definirá la `length` , el `width` y el `area` en el espacio de trabajo actual con el valor `10` , `3` y `30` respectivamente.

Como se indicó anteriormente, el script anterior es funcionalmente equivalente a escribir los mismos comandos directamente en la ventana de comandos.

```
>> length = 10;
>> width = 3;
>> area = length * width;
```

Funciones

Las funciones, en comparación con los scripts, son mucho más flexibles y extensibles. A diferencia de los scripts, las funciones pueden aceptar entrada y devolver la salida a la persona que llama. Una función tiene su propio espacio de trabajo, lo que significa que las operaciones internas de las funciones no cambiarán las variables del llamante.

Todas las funciones se definen con el mismo formato de encabezado:

```
function [output] = myFunctionName(input)
```

La palabra clave de `function` comienza cada encabezado de función. La lista de salidas sigue. La

lista de resultados también puede ser una lista de variables separadas por comas para devolver.

```
function [a, b, c] = myFunctionName(input)
```

El siguiente es el nombre de la función que se utilizará para llamar. Este es generalmente el mismo nombre que el nombre de archivo. Por ejemplo, guardaríamos esta función como `myFunctionName.m` .

Después del nombre de la función está la lista de entradas. Al igual que las salidas, esta también puede ser una lista separada por comas.

```
function [a, b, c] = myFunctionName(x, y, z)
```

Podemos reescribir el script de ejemplo de antes como una función reutilizable como la siguiente:

```
function [area] = calcRecArea(length, width)
   area = length * width;
end
```

Podemos llamar a funciones desde otras funciones, o incluso desde archivos de script. Este es un ejemplo de nuestra función anterior que se utiliza en un archivo de script.

```
l = 100;
w = 20;
a = calcRecArea(l, w);
```

Como antes, creamos `l` , `w` y `a` en el espacio de trabajo con los valores de `100` , `20` y `2000` respectivamente.

Tipos de datos

Hay 16 tipos de datos fundamentales , o clases, en MATLAB. Cada una de estas clases tiene la forma de una matriz o matriz. Con la excepción de los manejadores de funciones, esta matriz o matriz tiene un tamaño mínimo de 0 por 0 y puede crecer a una matriz n-dimensional de cualquier tamaño. Un controlador de función siempre es escalar (1 por 1).

El momento importante en MATLAB es que no es necesario utilizar ninguna declaración de tipo o declaración de dimensión de forma predeterminada. Cuando define una nueva variable, MATLAB la crea automáticamente y asigna el espacio de memoria apropiado.

Ejemplo:

```
a = 123;
b = [1 2 3];
c = '123';

>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
```

```
  b         1x3                24  double
  c         1x3                 6  char
```

Si la variable ya existe, MATLAB reemplaza los datos originales por uno nuevo y asigna nuevo espacio de almacenamiento si es necesario.

Tipos de datos fundamentales

Los tipos de datos fundamentales son: numérico, `logical` , `char` , `cell` , `struct` , `table` y `function_handle` .

Tipos de datos numéricos :

- Números de punto flotante (por *defecto*)

  MATLAB representa números de punto flotante en formato de precisión doble o de precisión simple. El valor predeterminado es la precisión doble, pero puede hacer que cualquier número tenga una sola precisión con una función de conversión simple:

```
a = 1.23;
b = single(a);

>> whos
  Name      Size             Bytes  Class     Attributes

  a         1x1                  8  double
  b         1x1                  4  single
```

- Enteros

  MATLAB tiene cuatro clases enteras con signo y cuatro sin signo. Los tipos con signo le permiten trabajar con enteros negativos y positivos, pero no pueden representar un rango de números tan amplio como los tipos sin signo porque se utiliza un bit para designar un signo positivo o negativo para el número. Los tipos sin firmar le ofrecen una gama más amplia de números, pero estos números solo pueden ser cero o positivos.

  MATLAB admite el almacenamiento de 1, 2, 4 y 8 bytes para datos enteros. Puede ahorrar memoria y tiempo de ejecución para sus programas si utiliza el tipo de entero más pequeño que acomoda sus datos. Por ejemplo, no necesita un entero de 32 bits para almacenar el valor 100.

```
a = int32(100);
b = int8(100);

>> whos
  Name      Size             Bytes  Class    Attributes

  a         1x1                  4  int32
  b         1x1                  1  int8
```

  Para almacenar datos como un entero, debe convertir el doble al tipo de entero deseado. Si el número que se está convirtiendo en un entero tiene una parte fraccionaria, MATLAB

redondea al entero más cercano. Si la parte fraccionaria es exactamente `0.5` , entonces, de los dos enteros igualmente cercanos, MATLAB elige aquel para el cual el valor absoluto es mayor en magnitud.

```
a  = int16(456);
```

- `char`

  Las matrices de caracteres proporcionan almacenamiento para datos de texto en MATLAB. De acuerdo con la terminología de programación tradicional, una matriz (secuencia) de caracteres se define como una cadena. No hay un tipo de cadena explícito en las versiones comerciales de MATLAB.

- lógico: los valores lógicos de 1 o 0, representan verdadero y falso respectivamente. Se usa para condiciones relacionales e indexación de matrices. Porque es VERDADERO o FALSO tiene un tamaño de 1 byte.

  ```
  a = logical(1);
  ```

- estructura. Una matriz de estructura es un tipo de datos que agrupa variables de diferentes tipos de datos utilizando contenedores de datos llamados *campos* . Cada campo puede contener cualquier tipo de datos. Acceda a los datos en una estructura usando la notación de puntos del formulario structName.fieldName.

  ```
  field1 = 'first';
  field2 = 'second';
  value1 = [1 2 3 4 5];
  value2 = 'sometext';
  s = struct(field1,value1,field2,value2);
  ```

  Para acceder a value1, cada una de las siguientes sintaxis son equivalentes

  ```
  s.first or s.(field1) or s.('first')
  ```

  Podemos acceder explícitamente a un campo que sabemos que existirá con el primer método, o bien pasar una cadena o crear una cadena para acceder al campo en el segundo ejemplo. El tercer ejemplo está demostrando que la notación de puntos entre parásitos toma una cadena, que es la misma almacenada en la variable field1.

- las variables de la tabla pueden ser de diferentes tamaños y tipos de datos, pero todas las variables deben tener el mismo número de filas.

  ```
  Age = [15 25 54]';
  Height = [176 190 165]';
  Name = {'Mike', 'Pete', 'Steeve'}';
  T = table(Name,Age, Height);
  ```

- célula. Es muy útil el tipo de datos MATLAB: la matriz de celdas es una matriz en la que cada elemento puede ser de diferente tipo y tamaño de datos. Es un instrumento muy fuerte

para manipular los datos como desee.

```
a = { [1 2 3], 56, 'art'};
```

o

```
a = cell(3);
```

- manijas de función almacena un puntero a una función (por ejemplo, a función anónima). Le permite pasar una función a otra función o llamar a funciones locales desde fuera de la función principal.

Hay muchos instrumentos para trabajar con cada tipo de datos y también funciones `str2double` conversión de tipos de datos (`str2double` , `table2cell`).

Tipos de datos adicionales

Hay varios tipos de datos adicionales que son útiles en algunos casos específicos. Son:

- Fecha y hora: matrices para representar fechas, hora y duración. `datetime('now')` devuelve `21-Jul-2016 16:30:16` .
- Arreglos categóricos: es el tipo de datos para almacenar datos con valores de un conjunto de categorías discretas. Útil para almacenar datos no numéricos (memoria efectiva). Se puede utilizar en una tabla para seleccionar grupos de filas.

```
a = categorical({'a' 'b' 'c'});
```

- Los contenedores de mapas son una estructura de datos que tiene una capacidad única para indexar no solo a través de los valores numéricos escalares sino también del vector de caracteres. Los índices en los elementos de un mapa se llaman claves. Estas claves, junto con los valores de datos asociados con ellas, se almacenan dentro del Mapa.
- Las series temporales son vectores de datos muestreados a lo largo del tiempo, en orden, a menudo a intervalos regulares. Es útil para almacenar los datos conectados con pasos de tiempo y tiene muchos métodos útiles para trabajar.

Funciones anónimas y manejadores de funciones.

Lo esencial

Las funciones anónimas son una herramienta poderosa del lenguaje MATLAB. Son funciones que existen localmente, es decir: en el espacio de trabajo actual. Sin embargo, no existen en la ruta MATLAB como lo haría una función normal, por ejemplo, en un archivo-m. Por eso se los llama anónimos, aunque pueden tener un nombre como una variable en el área de trabajo.

El operador `@`

Use el operador `@` para crear funciones anónimas y manejadores de funciones. Por ejemplo, para crear un controlador para la función `sin` (sine) y usarlo como `f` :

```
>> f = @sin
f =
    @sin
```

Ahora `f` es una manija de la función de `sin` . Al igual que (en la vida real) una manija de puerta es una forma de usar una puerta, una manija de función es una forma de usar una puerta. Para usar `f` , los argumentos se le pasan como si fuera la función `sin` :

```
>> f(pi/2)
ans =
     1
```

`f` acepta cualquier argumento de entrada que acepte la función `sin` . Si `sin` sería una función que acepta cero argumentos de entrada (que no lo hace, pero otros sí lo hacen, por ejemplo, la función de `peaks`), `f()` se usaría para llamarla sin argumentos de entrada.

Funciones anónimas personalizadas

Funciones anónimas de una variable.

Obviamente, no es útil crear un identificador para una función existente, como el `sin` en el ejemplo anterior. Es un poco redundante en ese ejemplo. Sin embargo, es útil crear funciones anónimas que hagan cosas personalizadas que de lo contrario tendrían que repetirse varias veces o crear una función separada para. Como ejemplo de una función anónima personalizada que acepta una variable como entrada, sume el seno y el coseno al cuadrado de una señal:

```
>> f = @(x) sin(x)+cos(x).^2
f =
    @(x)sin(x)+cos(x).^2
```

Ahora `f` acepta un argumento de entrada llamado `x` . Esto se especificó utilizando paréntesis `(...)` directamente después del operador `@` . `f` ahora es una función anónima de `x` : `f(x)` . Se usa pasando un valor de `x` a `f` :

```
>> f(pi)
ans =
    1.0000
```

Un vector de valores o una variable también se puede pasar a `f` , siempre que se utilicen de manera válida dentro de `f` :

```
>> f(1:3) % pass a vector to f
```

```
ans =
    1.1334    1.0825    1.1212
>> n = 5:7;
>> f(n) % pass n to f
ans =
   -0.8785    0.6425    1.2254
```

Funciones anónimas de más de una variable.

De la misma manera, se pueden crear funciones anónimas para aceptar más de una variable. Un ejemplo de una función anónima que acepta tres variables:

```
>> f = @(x,y,z) x.^2 + y.^2 - z.^2
f =
    @(x,y,z)x.^2+y.^2-z.^2
>> f(2,3,4)
ans =
    -3
```

Parametrización de funciones anónimas

Las variables en el área de trabajo se pueden usar dentro de la definición de funciones anónimas. Esto se llama parametrización. Por ejemplo, para usar una constante `c = 2` en una función anónima:

```
>> c = 2;
>> f = @(x) c*x
f =
    @(x)c*x
>> f(3)
ans =
     6
```

`f(3)` usó la variable `c` como parámetro para multiplicar con la `x` proporcionada. Tenga en cuenta que si el valor de `c` se establece en algo diferente en este punto, entonces se llama a `f(3)` , el resultado **no** sería diferente. El valor de `c` es el valor *en el momento de la creación* de la función anónima:

```
>> c = 2;
>> f = @(x) c*x;
>> f(3)
ans =
     6
>> c = 3;
>> f(3)
ans =
     6
```

Los argumentos de entrada a una función anónima no se refieren a las variables del área de trabajo

Tenga en cuenta que usar el nombre de las variables en el área de trabajo como uno de los argumentos de entrada de una función anónima (es decir, usar `@(...)`) **no** usará los valores de esas variables. En su lugar, se tratan como variables diferentes dentro del alcance de la función anónima, es decir: la función anónima tiene su espacio de trabajo privado donde las variables de entrada nunca se refieren a las variables del espacio de trabajo principal. El espacio de trabajo principal y el espacio de trabajo de la función anónima no conocen los contenidos de los demás. Un ejemplo para ilustrar esto:

```
>> x = 3 % x in main workspace
x =
     3
>> f = @(x) x+1; % here x refers to a private x variable
>> f(5)
ans =
     6
>> x
x =
     3
```

El valor de `x` del espacio de trabajo principal no se usa dentro de `f` . Además, en el espacio de trabajo principal `x` quedó sin tocar. Dentro del alcance de `f` , los nombres de las variables entre paréntesis después del operador `@` son independientes de las variables principales del área de trabajo.

Las funciones anónimas se almacenan en variables.

Una función anónima (o, más precisamente, el controlador de función que apunta a una función anónima) se almacena como cualquier otro valor en el espacio de trabajo actual: en una variable (como hicimos anteriormente), en una matriz de celdas (`{@(x)x.^2,@(x)x+1}`), o incluso en una propiedad (como `h.ButtonDownFcn` para gráficos interactivos). Esto significa que la función anónima puede tratarse como cualquier otro valor. Al almacenarlo en una variable, tiene un nombre en el espacio de trabajo actual y se puede cambiar y borrar al igual que las variables que contienen números.

En otras palabras: un manejador de función (ya sea en la forma `@sin` o para una función anónima) es simplemente un valor que puede almacenarse en una variable, como lo puede ser una matriz numérica.

Uso avanzado

Pasando los mangos de funciones a otras funciones.

Como los manejadores de función se tratan como variables, se pueden pasar a funciones que aceptan los manejadores de función como argumentos de entrada.

Un ejemplo: una función se crea en un archivo-m que acepta un identificador de función y un número escalar. A continuación, llama al controlador de función pasándole `3` y luego agrega el

número escalar al resultado. Se devuelve el resultado.

Contenido de `funHandleDemo.m` :

```
function y = funHandleDemo(fun,x)
y = fun(3);
y = y + x;
```

Guárdelo en algún lugar de la ruta, por ejemplo, en la carpeta actual de MATLAB. Ahora `funHandleDemo` se puede usar de la siguiente manera, por ejemplo:

```
>> f = @(x) x^2; % an anonymous function
>> y = funHandleDemo(f,10) % pass f and a scalar to funHandleDemo
y =
    19
```

El identificador de otra función existente se puede pasar a `funHandleDemo` :

```
>> y = funHandleDemo(@sin,-5)
y =
   -4.8589
```

Observe cómo `@sin` era una forma rápida de acceder al `sin` función sin primero almacenarlo en una variable mediante `f = @sin` .

Usando `bsxfun` , `cellfun` y funciones similares con funciones anónimas

MATLAB tiene algunas funciones integradas que aceptan funciones anónimas como entrada. Esta es una forma de realizar muchos cálculos con un número mínimo de líneas de código. Por ejemplo, `bsxfun` , que realiza operaciones binarias elemento por elemento, es decir: aplica una función en dos vectores o matrices en una forma elemento por elemento. Normalmente, esto requeriría el uso de `for` loops, que a menudo requiere una preasignación para la velocidad. Usando `bsxfun` este proceso se acelera. El siguiente ejemplo ilustra esto usando `tic` y `toc` , dos funciones que se pueden usar para medir el tiempo que toma el código. Calcula la diferencia de cada elemento de la matriz a partir de la media de la columna de la matriz.

```
A = rand(50); % 50-by-50 matrix of random values between 0 and 1

% method 1: slow and lots of lines of code
tic
meanA = mean(A); % mean of every matrix column: a row vector
% pre-allocate result for speed, remove this for even worse performance
result = zeros(size(A));
for j = 1:size(A,1)
    result(j,:) = A(j,:) - meanA;
end
toc
clear result % make sure method 2 creates its own result

% method 2: fast and only one line of code
```

```
tic
result = bsxfun(@minus,A,mean(A));
toc
```

Al ejecutar el ejemplo anterior se obtienen dos resultados:

```
Elapsed time is 0.015153 seconds.
Elapsed time is 0.007884 seconds.
```

Estas líneas provienen de las funciones `toc` , que imprimen el tiempo transcurrido desde la última llamada a la función `tic` .

La llamada `bsxfun` aplica la función en el primer argumento de entrada a los otros dos argumentos de entrada. `@minus` es un nombre largo para la misma operación que haría el signo menos. Una función anónima diferente o manejar (`@`) para cualquier otra función que podría haber sido especificado, siempre y cuando se acepta `A` y `mean(A)` como insumos para generar un resultado significativo.

Especialmente para grandes cantidades de datos en matrices grandes, `bsxfun` puede acelerar mucho las cosas. También hace que el código se vea más limpio, aunque podría ser más difícil de interpretar para las personas que no conocen MATLAB o `bsxfun` . (Tenga en cuenta que en MATLAB R2016a y posteriores, muchas de las operaciones que utilizaron previamente `bsxfun` ya no las necesitan; `A-mean(A)` funciona directamente y en algunos casos puede ser incluso más rápida).

Lea Empezando con MATLAB Language en línea:
https://riptutorial.com/es/matlab/topic/235/empezando-con-matlab-language

Capítulo 2: Aplicaciones financieras

Examples

Caminata aleatoria

El siguiente es un ejemplo que muestra 5 recorridos aleatorios unidimensionales de 200 pasos:

```
y = cumsum(rand(200,5) - 0.5);

plot(y)
legend('1', '2', '3', '4', '5')
title('random walks')
```

En el código anterior, `y` es una matriz de 5 columnas, cada una de 200 de longitud. Como se omite `x` , el valor predeterminado es los números de fila de `y` (equivalente a usar `x=1:200` como el eje x). De esta manera, la función de `plot` traza múltiples vectores y contra el mismo vector x, cada uno usando un color diferente automáticamente.

Movimiento Browniano Geométrico Univariable

La dinámica del Movimiento Browniano Geométrico (GBM) se describe mediante la siguiente ecuación diferencial estocástica (SDE):

$$dS_t = \mu S_t\,dt + \sigma S_t\,dW_t$$

Puedo usar la solución **exacta** para el SDE

$$S_t = S_0 \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right)$$

para generar rutas que siguen un GBM.

Dados parámetros diarios para una simulación de un año.

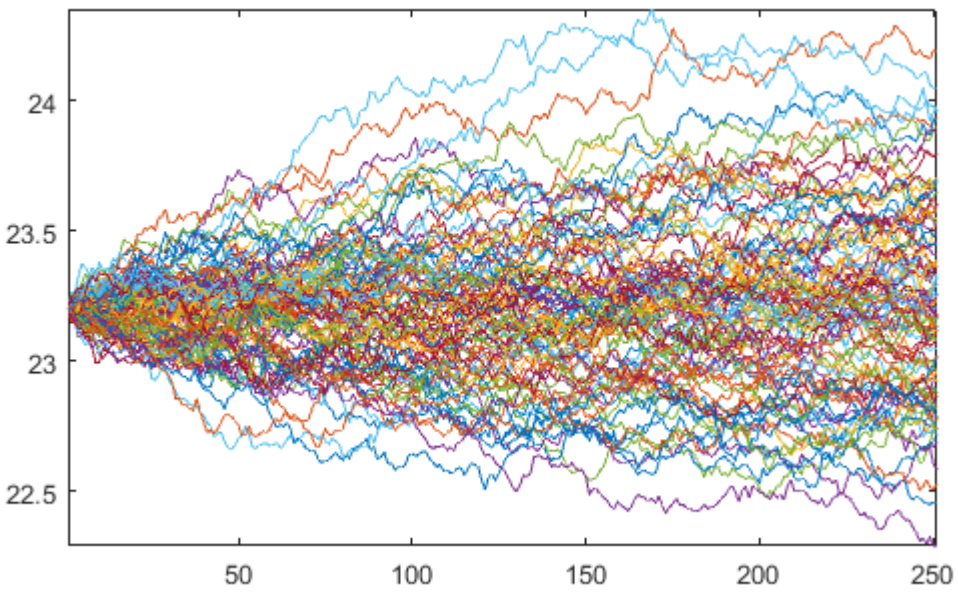
```
mu     = 0.08/250;
sigma  = 0.25/sqrt(250);
dt     = 1/250;
npaths = 100;
nsteps = 250;
S0     = 23.2;
```

podemos obtener el Brownian Motion (BM) `W` partir de 0 y utilizarlo para obtener el GBM a partir de `S0`

```
% BM
epsilon = randn(nsteps, npaths);
W       = [zeros(1,npaths); sqrt(dt)*cumsum(epsilon)];

% GBM
t = (0:nsteps)'*dt;
Y = bsxfun(@plus, (mu-0.5*sigma.^2)*t, sigma*W);
Y = S0*exp(Y);
```

Lo que produce los caminos.

```
plot(Y)
```



Lea Aplicaciones financieras en línea: https://riptutorial.com/es/matlab/topic/1197/aplicaciones-

financieras

Capítulo 3: Características no documentadas

Observaciones

- El uso de características no documentadas se considera una práctica arriesgada [1] , ya que estas características pueden cambiar sin previo aviso o simplemente funcionan de manera diferente en diferentes versiones de MATLAB. Por esta razón, se recomienda emplear técnicas de programación defensiva , como encerrar piezas de código no documentadas dentro de bloques `try/catch` con fallbacks documentados.

Examples

Funciones auxiliares compatibles con C ++

El uso de **Matlab Coder a** veces niega el uso de algunas funciones muy comunes, si no son compatibles con C ++. Relativamente a menudo existen **funciones de ayuda indocumentadas** , que se pueden usar como reemplazos.

Aquí hay una lista completa de las funciones compatibles. .

Y siguiendo una colección de alternativas, para funciones no soportadas:

Las funciones `sprintf` y `sprintfc` son bastante similares, la primera devuelve una *matriz de caracteres* , la última una *cadena de celdas* :

```
str = sprintf('%i',x)   % returns '5' for x = 5
str = sprintfc('%i',x)  % returns {'5'} for x = 5
```

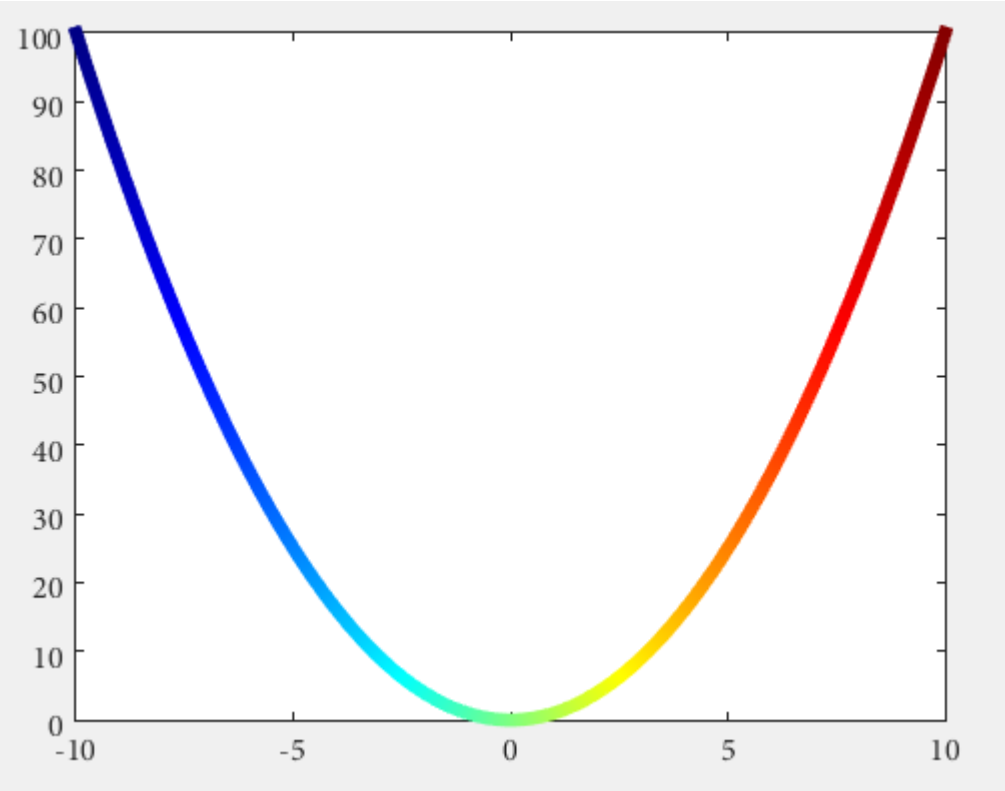
Sin embargo, `sprintfc` es compatible con C ++ soportado por Matlab Coder, y `sprintf` no lo es.

Gráficos lineales 2D codificados por color con datos de color en la tercera dimensión

En las versiones de MATLAB anteriores a **R2014b** , utilizando el antiguo motor gráfico HG1, no era obvio cómo crear gráficos de líneas 2D codificados por colores . Con el lanzamiento del nuevo motor gráfico HG2 surgió una nueva **característica no documentada presentada por Yair Altman** :

```
n = 100;
x = linspace(-10,10,n); y = x.^2;
p = plot(x,y,'r', 'LineWidth',5);

% modified jet-colormap
cd = [uint8(jet(n)*255) uint8(ones(n,1))].';

drawnow
```

```
set(p.Edge, 'ColorBinding','interpolated', 'ColorData',cd)
```



Marcadores semitransparentes en trazos de línea y dispersión.

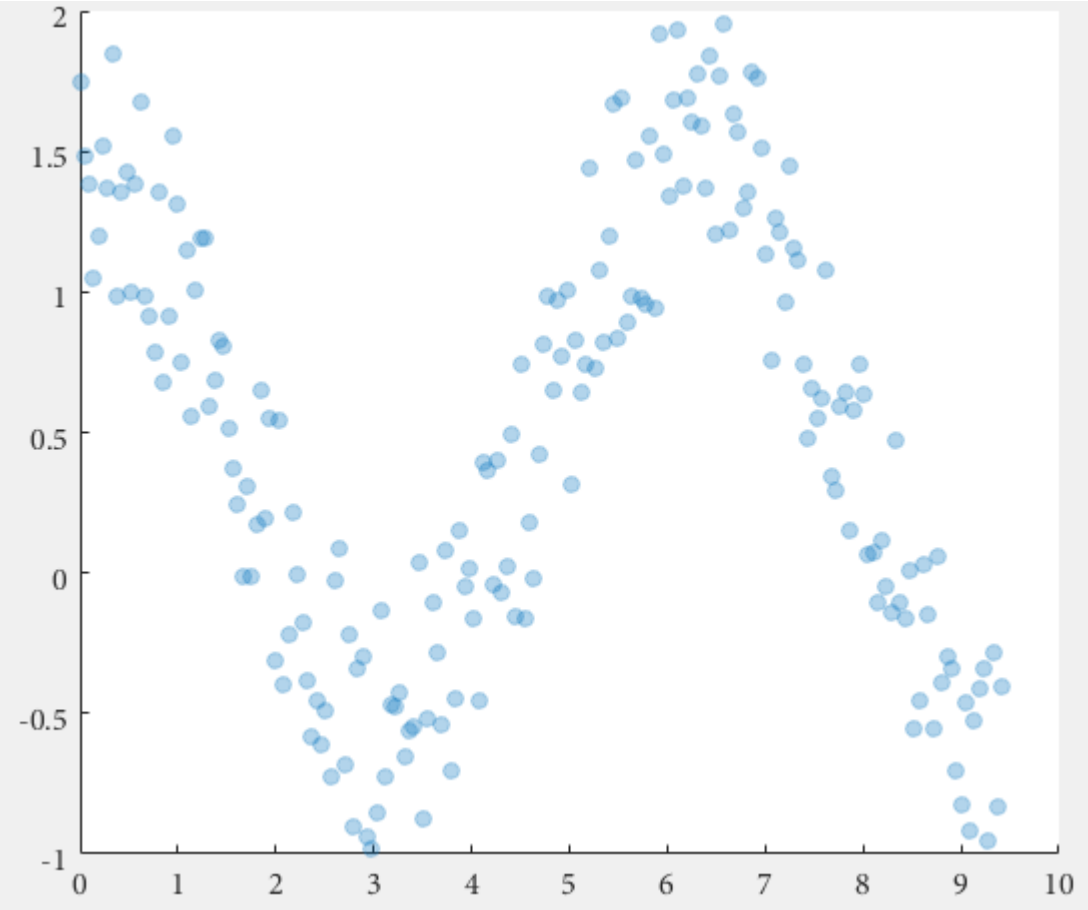
Desde **Matlab R2014b** es fácilmente posible lograr marcadores semitransparentes para trazados de líneas y dispersión utilizando **características no documentadas introducidas por Yair Altman** .

La idea básica es obtener el controlador oculto de los marcadores y aplicar un valor **<1** para el último valor en `EdgeColorData` para lograr la transparencia deseada.

Aquí vamos por la `scatter` :

```
%// example data
x = linspace(0,3*pi,200);
y = cos(x) + rand(1,200);

%// plot scatter, get handle
h = scatter(x,y);
drawnow; %// important

%// get marker handle
hMarkers = h.MarkerHandle;

%// get current edge and face color
edgeColor = hMarkers.EdgeColorData
faceColor = hMarkers.FaceColorData

%// set face color to the same as edge color
faceColor = edgeColor;

%// opacity
```

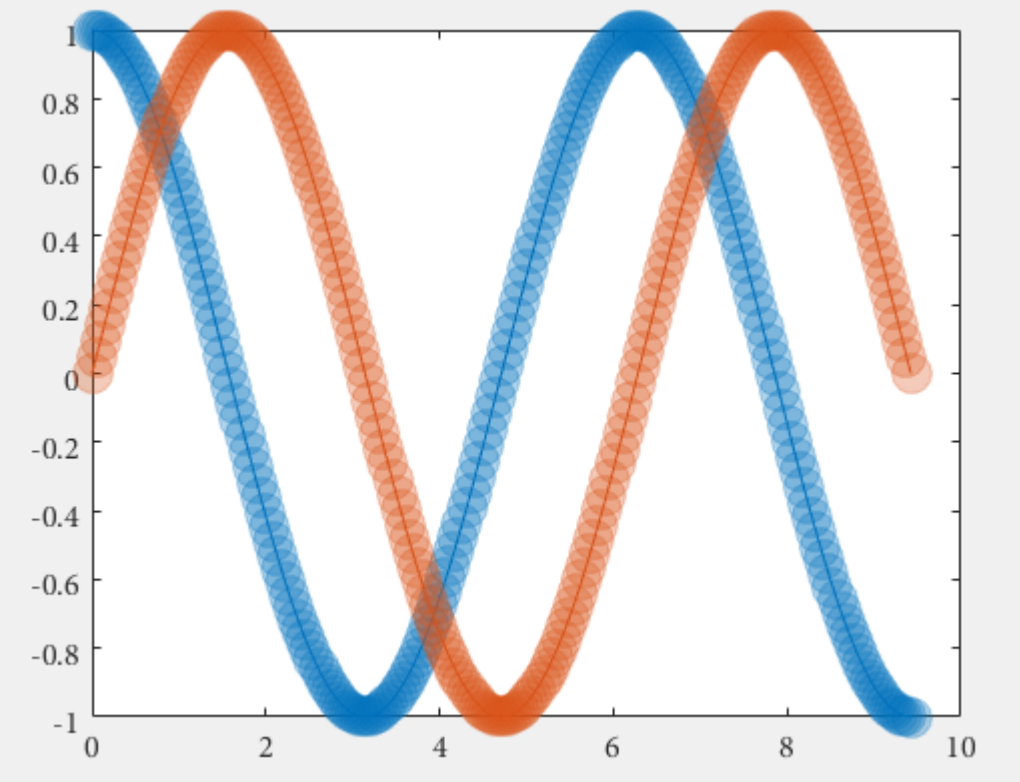
```
opa = 0.3;

%// set marker edge and face color
hMarkers.EdgeColorData = uint8( [edgeColor(1:3); 255*opa] );
hMarkers.FaceColorData = uint8( [faceColor(1:3); 255*opa] );
```



y para una línea de `plot`

```
%// example data
x = linspace(0,3*pi,200);
y1 = cos(x);
y2 = sin(x);

%// plot scatter, get handle
h1 = plot(x,y1,'o-','MarkerSize',15); hold on
h2 = plot(x,y2,'o-','MarkerSize',15);
drawnow; %// important

%// get marker handle
h1Markers = h1.MarkerHandle;
h2Markers = h2.MarkerHandle;

%// get current edge and face color
edgeColor1 = h1Markers.EdgeColorData;
edgeColor2 = h2Markers.EdgeColorData;

%// set face color to the same as edge color
faceColor1 = edgeColor1;
faceColor2 = edgeColor2;

%// opacity
```

```
opa = 0.3;

%// set marker edge and face color
h1Markers.EdgeColorData = uint8( [edgeColor1(1:3); 255*opa] );
h1Markers.FaceColorData = uint8( [faceColor1(1:3); 255*opa] );
h2Markers.EdgeColorData = uint8( [edgeColor2(1:3); 255*opa] );
h2Markers.FaceColorData = uint8( [faceColor2(1:3); 255*opa] );
```



Los identificadores de marcador, que se utilizan para la manipulación, se crean con la figura. El comando **`drawnow`** garantiza la creación de la figura antes de que se llame a los comandos subsiguientes y evita errores en caso de demoras.

Gráficos de contorno - Personalizar las etiquetas de texto

Al mostrar etiquetas en los contornos, Matlab no le permite controlar el formato de los números, por ejemplo, para cambiar a notación científica.

Los objetos de texto individuales son objetos de texto normales, pero la forma en que los obtiene no está documentada. Puede acceder a ellos desde la propiedad `TextPrims` del `TextPrims` de contorno.

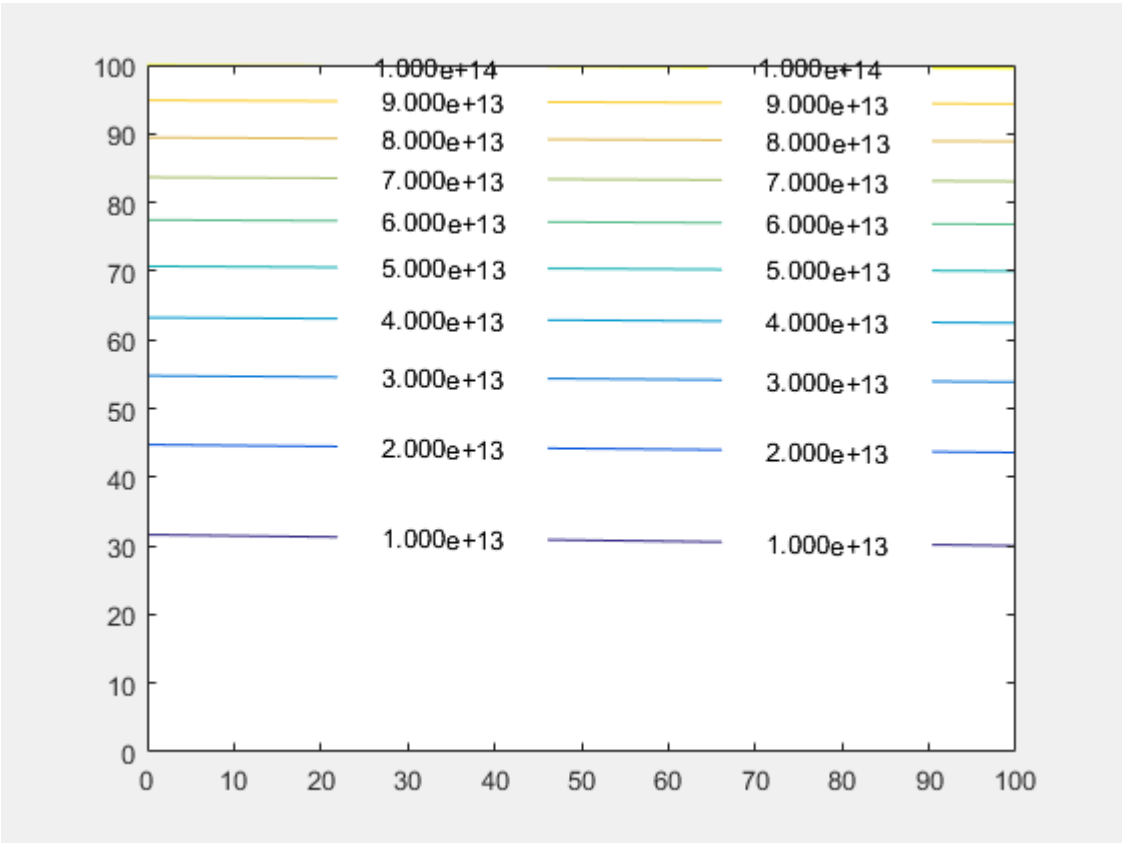
```
figure
[X,Y]=meshgrid(0:100,0:100);
Z=(X+Y.^2)*1e10;
[C,h]=contour(X,Y,Z);
h.ShowText='on';
drawnow();
txt = get(h,'TextPrims');
v = str2double(get(txt,'String'));
for ii=1:length(v)
  set(txt(ii),'String',sprintf('%0.3e',v(ii)))
```

```
  end
```

Nota : que debe agregar un comando de `drawnow` para forzar a Matlab a dibujar los contornos, el número y la ubicación de los objetos txt solo se determinan cuando los contornos se dibujan realmente, por lo que los objetos de texto solo se crean en ese momento.

El hecho de que los objetos txt se crean cuando se dibujan los contornos significa que se recalculan cada vez que la trama se vuelve a dibujar (por ejemplo, el tamaño de la figura). Para gestionar esto, necesita escuchar el `undocumented event MarkedClean` :

```
function customiseContour
  figure
  [X,Y]=meshgrid(0:100,0:100);
  Z=(X+Y.^2)*1e10;
  [C,h]=contour(X,Y,Z);
  h.ShowText='on';
  % add a listener and call your new format function
  addlistener(h,'MarkedClean',@(a,b)ReFormatText(a))
end
function ReFormatText(h)
  % get all the text items from the contour
  t = get(h,'TextPrims');
  for ii=1:length(t)
    % get the current value (Matlab changes this back when it
    %   redraws the plot)
    v = str2double(get(t(ii),'String'));
    % Update with the format you want - scientific for example
    set(t(ii),'String',sprintf('%0.3e',v));
  end
end
```



Ejemplo probado utilizando Matlab r2015b en Windows

Añadir / agregar entradas a una leyenda existente

Las leyendas existentes pueden ser difíciles de manejar. Por ejemplo, si su parcela tiene dos líneas, pero solo una de ellas tiene una entrada de leyenda y debería seguir siendo así, agregar una tercera línea con una entrada de leyenda puede ser difícil. Ejemplo:

```
figure
hold on
fplot(@sin)
fplot(@cos)
legend sin  % Add only a legend entry for sin
hTan = fplot(@tan);  % Make sure to get the handle, hTan, to the graphics object you want to
add to the legend
```

Ahora, para agregar una entrada de leyenda para `tan` , pero no para `cos` , cualquiera de las siguientes líneas no funcionará; todos ellos fallan de alguna manera

```
legend tangent  % Replaces current legend -> fail
legend -DynamicLegend  % Undocumented feature, adds 'cos', which shouldn't be added -> fail
legend sine tangent  % Sets cos DisplayName to 'tangent' -> fail
legend sine '' tangent  % Sets cos DisplayName, albeit empty -> fail
legend(f)
```

Afortunadamente, una propiedad de leyenda no documentada llamada `PlotChildren` un seguimiento de los hijos de la figura principal [1] . Entonces, el camino a seguir es establecer explícitamente los hijos de la leyenda a través de su propiedad `PlotChildren` siguiente manera:

```
hTan.DisplayName = 'tangent';  % Set the graphics object's display name
l = legend;
l.PlotChildren(end + 1) = hTan;  % Append the graphics handle to legend's plot children
```

La leyenda se actualiza automáticamente si se agrega o elimina un objeto de su propiedad `PlotChildren` .

[1] En efecto: figura. Puede agregar el elemento secundario de cualquier figura con la propiedad `DisplayName` a cualquier leyenda en la figura, por ejemplo, desde una trama secundaria diferente. Esto se debe a que una leyenda en sí misma es básicamente un objeto de ejes.

Probado en MATLAB R2016b

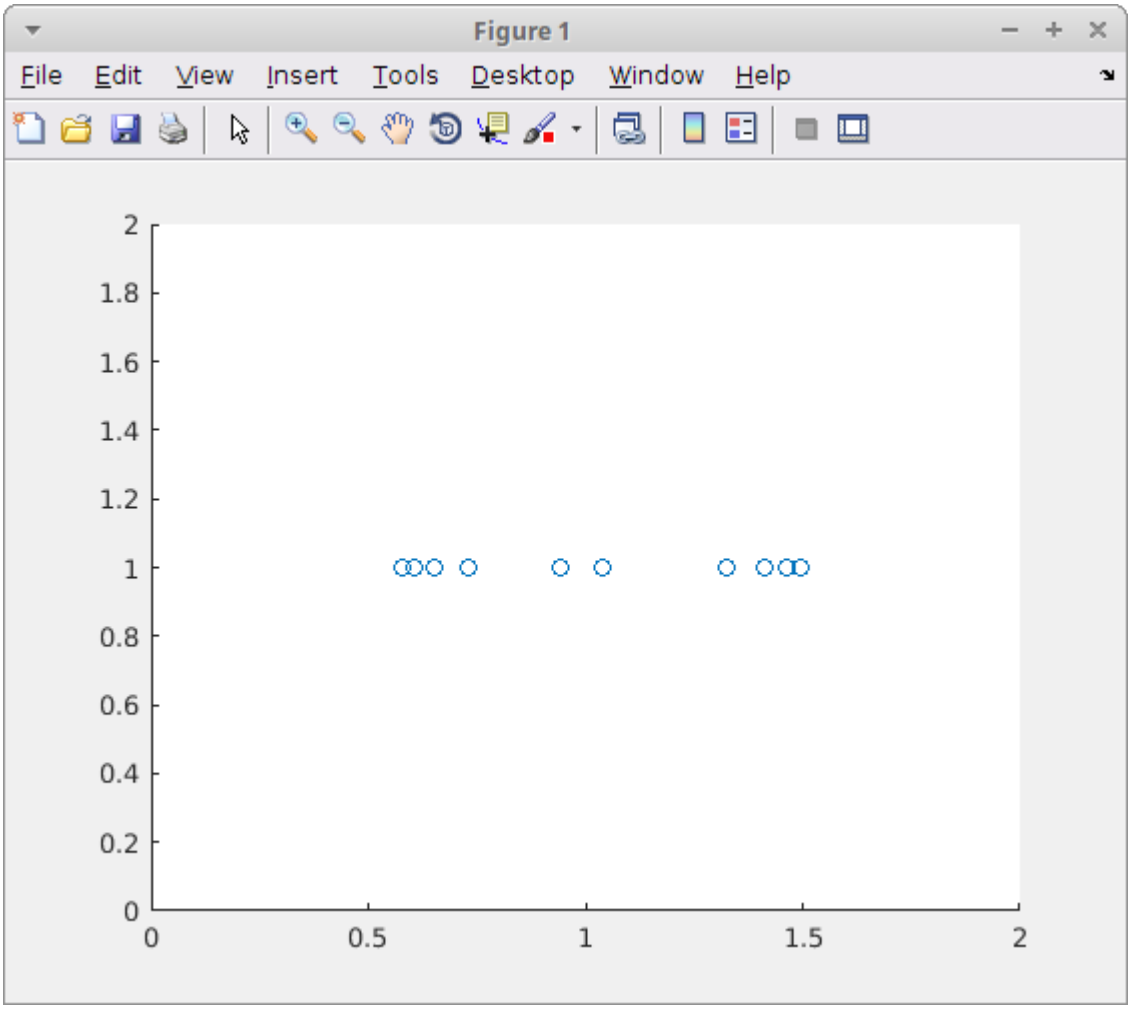
Dispersión de la trama de dispersión

La función de `scatter` tiene dos propiedades no documentadas `'jitter'` y `'jitterAmount'` que permiten jitter los datos solo en el eje x. Esto se remonta a Matlab 7.1 (2005), y posiblemente antes.

Para habilitar esta función, establezca la propiedad `'jitter'` en `'on'` y configure la propiedad `'jitterAmount'` en el valor absoluto deseado (el valor predeterminado es `0.2`).

Esto es muy útil cuando queremos visualizar datos superpuestos, por ejemplo:

```
scatter(ones(1,10), ones(1,10), 'jitter', 'on', 'jitterAmount', 0.5);
```



Leer más en Undocumented Matlab

Lea Características no documentadas en línea:
https://riptutorial.com/es/matlab/topic/2383/caracteristicas-no-documentadas

Capítulo 4: Condiciones

Sintaxis

- si *expresión* ... fin
- si *expresion* ... lo demas ... fin
- si *expresión* ... elseif *expresión* ... fin
- if *expresión* ... elseif *expresión* ... else ... fin

Parámetros

Parámetro	Descripción
expresión	Una expresión que tiene un significado lógico.

Examples

Condición de si

Las condiciones son una parte fundamental de casi cualquier parte del código. Se utilizan para ejecutar algunas partes del código solo en algunas situaciones, pero no en otras. Veamos la sintaxis básica:

```
a = 5;
if a > 10    % this condition is not fulfilled, so nothing will happen
    disp('OK')
end

if a < 10    % this condition is fulfilled, so the statements between the if...end are
executed
    disp('Not OK')
end
```

Salida:

```
Not OK
```

En este ejemplo, vemos que el `if` consta de 2 partes: la condición y el código que se ejecutará si la condición es verdadera. El código es todo lo escrito después de la condición y antes del `end` de ese `if` . La primera condición no se cumplió y, por lo tanto, el código que contiene no se ejecutó.

Aquí hay otro ejemplo:

```
a = 5;
if a ~= a+1        % "~=" means "not equal to"
    disp('It''s true!') % we use two apostrophes to tell MATLAB that the ' is part of the
```

```
string
end
```

La condición anterior siempre será verdadera y mostrará la salida ¡ `It's true!` .

También podemos escribir:

```
a = 5;
if a == a+1    % "==" means "is equal to", it is NOT the assignment ("=") operator
    disp('Equal')
end
```

Esta vez la condición siempre es falsa, por lo que nunca obtendremos la salida `Equal` .

Sin embargo, las condiciones que siempre son verdaderas o falsas no son muy útiles, porque si siempre son falsas, simplemente podemos eliminar esta parte del código, y si siempre son verdaderas, entonces la condición no es necesaria.

Condición IF-ELSE

En algunos casos, queremos ejecutar un código alternativo si la condición es falsa, para esto utilizamos la `else` parte opcional:

```
a = 20;
if a < 10
    disp('a smaller than 10')
else
    disp('a bigger than 10')
end
```

Aquí vemos que debido `a` que no es menor que `10` la segunda parte del código, después de `else` se ejecuta `else` , obtenemos `a bigger than 10` resultado `a bigger than 10` . Ahora veamos otro intento:

```
a = 10;
if a > 10
    disp('a bigger than 10')
else
    disp('a smaller than 10')
end
```

En este ejemplo, se muestra que no verificamos si `a` es realmente menor que 10, y recibimos un mensaje incorrecto porque la condición solo verifica la expresión tal como es, y CUALQUIER caso que no sea verdadero (`a = 10`) causará que Segunda parte a ser ejecutada.

Este tipo de error es un error muy común tanto para los principiantes como para los programadores experimentados, especialmente cuando las condiciones se vuelven complejas y siempre se debe tener en cuenta.

Condición IF-ELSEIF

Usando `else` podemos realizar alguna tarea cuando la condición no se cumple. Pero qué pasa si queremos verificar una segunda condición en caso de que la primera fuera falsa. Podemos hacerlo de esta manera:

```
a = 9;
if mod(a,2)==0   % MOD - modulo operation, return the remainder after division of 'a' by 2
    disp('a is even')
else
    if mod(a,3)==0
        disp('3 is a divisor of a')
    end
end

OUTPUT:
3 is a divisor of a
```

Esto también se conoce como "condición anidada" , pero aquí tenemos un caso específico que puede mejorar la legibilidad del código y reducir la posibilidad de error y error: podemos escribir:

```
a = 9;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
end

OUTPUT:
3 is a divisor of a
```

utilizando el `elseif` podemos verificar otra expresión dentro del mismo bloque de condición, y esto no está limitado a un intento:

```
a = 25;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
end

OUTPUT:
5 is a divisor of a
```

Un cuidado especial debe ser tomado al elegir utilizar `elseif` en una fila, ya que **sólo uno** de ellos se ejecutará de todo el `if` a `end` de bloque. Así, en nuestro ejemplo, si queremos visualizar todos los divisores de `a` (de las que comprobar explícitamente) el ejemplo anterior no será bueno:

```
a = 15;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
```

```
end

OUTPUT:
3 is a divisor of a
```

no solo 3, sino también 5 es un divisor de 15, pero la parte que verifica la división por 5 no se alcanza si alguna de las expresiones anteriores era verdadera.

Finalmente, podemos agregar uno `else` (y solo **uno**) después de todas las condiciones de `elseif` para ejecutar un código cuando no se cumple ninguna de las condiciones anteriores:

```
a = 11;
if mod(a,2)==0
    disp('a is even')
elseif mod(a,3)==0
    disp('3 is a divisor of a')
elseif mod(a,5)==0
    disp('5 is a divisor of a')
else
    disp('2, 3 and 5 are not divisors of a')
end

OUTPUT:
2, 3 and 5 are not divisors of a
```

Condiciones anidadas

Cuando usamos una condición dentro de otra condición, decimos que las condiciones están "anidadas". La opción `elseif` proporciona un caso especial de condiciones anidadas, pero existen muchas otras formas de utilizar las condiciones anidadas. Examinemos el siguiente código:

```
a = 2;
if mod(a,2)==0    % MOD - modulo operation, return the remainder after division of 'a' by 2
    disp('a is even')
    if mod(a,3)==0
        disp('3 is a divisor of a')
        if mod(a,5)==0
            disp('5 is a divisor of a')
        end
    end
else
    disp('a is odd')
end
```

Para `a=2` , la salida será `a is even` , lo que es correcto. Para `a=3` , la salida será `a is odd` , lo que también es correcto, pero no se comprueba si 3 es un divisor de `a` . Esto se debe a que las condiciones están anidadas, por lo que **solo** si la primera es `true` , entonces nos movemos hacia la interna, y si `a` es impar, ninguna de las condiciones internas se verifica. Esto es algo opuesto al uso de `elseif` donde solo si la primera condición es `false` que la que verificamos en la siguiente. ¿Qué hay de comprobar la división por 5? solo un número que tenga 6 como divisor (tanto 2 como 3) se verificará para la división entre 5, y podemos probar y ver que para `a=30` la salida es:

```
a is even
```

```
3 is a divisor of a
5 is a divisor of a
```

También debemos notar dos cosas:

1. La posición de la `end` en el lugar correcto para cada `if` es crucial para el conjunto de condiciones para trabajar como se esperaba, por lo que la sangría es más que una buena recomendación aquí.
2. La posición de la declaración `else` también es crucial, porque necesitamos saber en qué `if` (y puede haber varios) queremos hacer algo en caso de que la expresión sea `false` .

Veamos otro ejemplo:

```
for a = 5:10    % the FOR loop execute all the code within it for every a from 5 to 10
    ch = num2str(a);    % NUM2STR converts the integer a to a character
    if mod(a,2)==0
        if mod(a,3)==0
            disp(['3 is a divisor of ' ch])
        elseif  mod(a,4)==0
            disp(['4 is a divisor of ' ch])
        else
            disp([ch ' is even'])
        end
    elseif mod(a,3)==0
        disp(['3 is a divisor of ' ch])

    else
        disp([ch ' is odd'])
    end
end
```

Y la salida será:

```
5 is odd
3 is a divisor of 6
7 is odd
4 is a divisor of 8
3 is a divisor of 9
10 is even
```

vemos que solo tenemos 6 líneas para 6 números, porque las condiciones están anidadas de manera que se garantiza una sola impresión por número y también (aunque no se pueden ver directamente desde la salida) no se realizan comprobaciones adicionales, por lo que si un número ni siquiera tiene ningún punto para verificar si 4 es uno de sus divisores.

Lea Condiciones en línea: https://riptutorial.com/es/matlab/topic/3806/condiciones

Capítulo 5: Controlando la coloración de la subparcela en Matlab

Introducción

Ya que estuve luchando con esto más de una vez, y la web no tiene muy claro qué hacer, decidí tomar lo que hay por ahí, agregando algo propio para explicar cómo crear subparcelas que tienen una barra de colores y Se escalan de acuerdo a ello.

He probado esto usando el último Matlab pero estoy bastante seguro de que funcionará en versiones anteriores.

Observaciones

Lo único que necesita resolver por sí mismo es la posición de la barra de colores (si desea mostrarla). esto dependerá de la cantidad de gráficos que tengas y de la orientación de la barra.

La posición y el tamaño se definen utilizando 4 parámetros: x_start, y_start, x_width, y_width. La gráfica se suele escalar a unidades normalizadas de modo que la esquina inferior izquierda se corresponda con (0,0) y la superior derecha con (1,1).

Examples

Cómo está hecho

Este es un código simple que crea 6 subplots 3d y al final sincroniza el color que se muestra en cada uno de ellos.

```
c_fin = [0,0];
[X,Y] = meshgrid(1:0.1:10,1:0.1:10);

figure; hold on;
for i = 1 : 6
    Z(:,:,i) = i * (sin(X) + cos(Y));

    ax(i) = subplot(3,2,i); hold on; grid on;
    surf(X, Y, Z(:,:,i));
    view(-26,30);
    colormap('jet');
    ca = caxis;
    c_fin = [min(c_fin(1),ca(1)), max(c_fin(2),ca(2))];
end

%%you can stop here to see how it looks before we color-manipulate

c = colorbar('eastoutside');
c.Label.String = 'Units';
set(c, 'Position', [0.9, 0.11, 0.03, 0.815]); %%you may want to play with these values
```

```
pause(2); %%need this to allow the last image to resize itself before changing its axes
for i = 1 : 6
    pos=get(ax(i), 'Position');
    axes(ax(i));
    set(ax(i), 'Position', [pos(1) pos(2) 0.85*pos(3) pos(4)]);
    set(ax(i),'Clim', c_fin); %%this is where the magic happens
end
```

Lea Controlando la coloración de la subparcela en Matlab en línea:
https://riptutorial.com/es/matlab/topic/10913/controlando-la-coloracion-de-la-subparcela-en-matlab

Capítulo 6: Depuración

Sintaxis

- dbstop en el archivo en la ubicación si la expresión

Parámetros

Parámetro	Detalles
expediente	Nombre del archivo `.m` (sin extensión), por ejemplo, `fit` . Este parámetro es *(Requerido) a* menos que se establezcan tipos especiales de punto de interrupción condicional, como `dbstop if error` o `dbstop if naninf` .
ubicación	Número de línea donde se debe colocar el punto de interrupción. Si la línea especificada no contiene código ejecutable, el punto de interrupción se colocará en la primera línea válida **después de** la especificada.
expresión	Cualquier expresión o combinación de las mismas que se evalúe como un valor booleano. Ejemplos: `ind == 1` , `nargin < 4 && isdir('Q:\')` .

Examples

Trabajando con Breakpoints

Definición

> *En el desarrollo de software, un **punto de interrupción** es un lugar para detener o pausar intencionalmente un programa, implementado para fines de depuración.*
>
> *Más generalmente, un punto de interrupción es un medio para adquirir conocimiento sobre un programa durante su ejecución. Durante la interrupción, el programador inspecciona el entorno de prueba (registros de propósito general, memoria, registros, archivos, etc.) para averiguar si el programa está funcionando como se esperaba. En la práctica, un punto de interrupción consiste en una o más condiciones que determinan cuándo se debe interrumpir la ejecución de un programa.*
>
> -Wikipedia

Puntos de interrupción en MATLAB

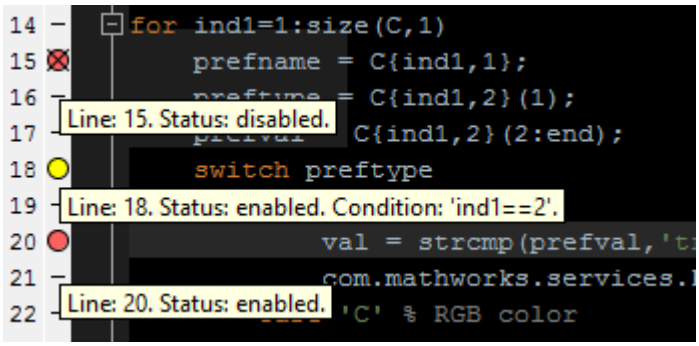
Motivación

En MATLAB, cuando la ejecución se detiene en un punto de interrupción, las variables existentes en el área de trabajo actual (también conocido como *alcance*) o cualquiera de las áreas de trabajo que llaman, se pueden inspeccionar (y generalmente también se pueden modificar).

Tipos de puntos de ruptura

MATLAB permite a los usuarios colocar dos tipos de puntos de interrupción en archivos `.m` :

- *Puntos de* interrupción estándar (o "no *restringidos* ") (mostrados en rojo): pausa la ejecución cada vez que se alcanza la línea marcada.
- Puntos de interrupción "condicionales" (mostrados en amarillo): pausa la ejecución cada vez que se alcanza la línea marcada Y la condición definida en el punto de interrupción se evalúa como `true` .



Colocando puntos de ruptura

Ambos tipos de puntos de interrupción se pueden crear de varias maneras:

- Usando la GUI del editor de MATLAB, haga clic derecho en la línea horizontal al lado del número de línea.

- Usando el comando `dbstop` :

```
% Create an unrestricted breakpoint:
dbstop in file at location
% Create a conditional breakpoint:
dbstop in file at location if expression

% Examples and special cases:
dbstop in fit at 99 % Standard unrestricted breakpoint.

dbstop in fit at 99 if nargin==3 % Standard conditional breakpoint.

dbstop if error % This special type of breakpoint is not limited to a specific file, and
                % will trigger *whenever* an error is encountered in "debuggable" code.

dbstop in file % This will create an unrestricted breakpoint on the first executable line
               % of "file".
```

```
dbstop if naninf % This special breakpoint will trigger whenever a computation result
                 % contains either a NaN (indicates a division by 0) or an Inf
```

- Uso de métodos abreviados de teclado: la tecla predeterminada para crear un punto de interrupción estándar en Windows es `F12` ; la clave por defecto para los puntos de interrupción condicionales *no* está *definido.*

Deshabilitar y volver a habilitar los puntos de interrupción

Deshabilite un punto de interrupción para ignorarlo temporalmente: los puntos de interrupción deshabilitados no pausan la ejecución. La desactivación de un punto de interrupción se puede hacer de varias maneras:

- Haga clic con el botón derecho en el círculo rojo / amarillo del punto de interrupción> Deshabilitar el punto de interrupción.
- Clic izquierdo en un punto de quiebre condicional (amarillo).
- En la pestaña Editor> Puntos de interrupción> Habilitar \ Deshabilitar.

Eliminar puntos de interrupción

Todos los puntos de interrupción permanecen en un archivo hasta que se eliminen, ya sea de forma manual o automática. Los puntos de interrupción se borran *automáticamente* al finalizar la sesión de MATLAB (es decir, al finalizar el programa). La eliminación manual de los puntos de interrupción se realiza de una de las siguientes maneras:

- Usando el comando `dbclear` :

```
dbclear all
dbclear in file
dbclear in file at location
dbclear if condition
```

- Haga clic con el botón izquierdo en un icono de punto de interrupción estándar o un icono de punto de interrupción condicional desactivado.
- Haga clic derecho en cualquier punto de interrupción> Borrar punto de interrupción.
- En la pestaña Editor> Puntos de interrupción> Borrar todo.
- En versiones anteriores a R2015b de MATLAB, usando el comando `clear` .

Reanudar la ejecución

Cuando la ejecución se detiene en un punto de interrupción, hay dos formas de continuar ejecutando el programa:

- Ejecute la línea actual y vuelva a hacer una pausa antes de la línea siguiente.

`F10` [1] en el Editor, `dbstep` en la ventana de comandos, "Paso" en Ribbon> Editor> DEBUG.

- Ejecutar hasta el siguiente punto de interrupción (si no hay más puntos de interrupción, la ejecución continúa hasta el final del programa).

  `F12` [1] en el Editor, `dbcont` en la ventana de comandos, "Continuar" en Ribbon> Editor> DEBUG.

---

[1] - por defecto en Windows.

Depuración de código Java invocado por MATLAB

Visión general

Para depurar las clases Java que se llaman durante la ejecución de MATLAB, es necesario realizar dos pasos:

1. Ejecute MATLAB en modo de depuración JVM.
2. Adjunte un depurador de Java al proceso MATLAB.

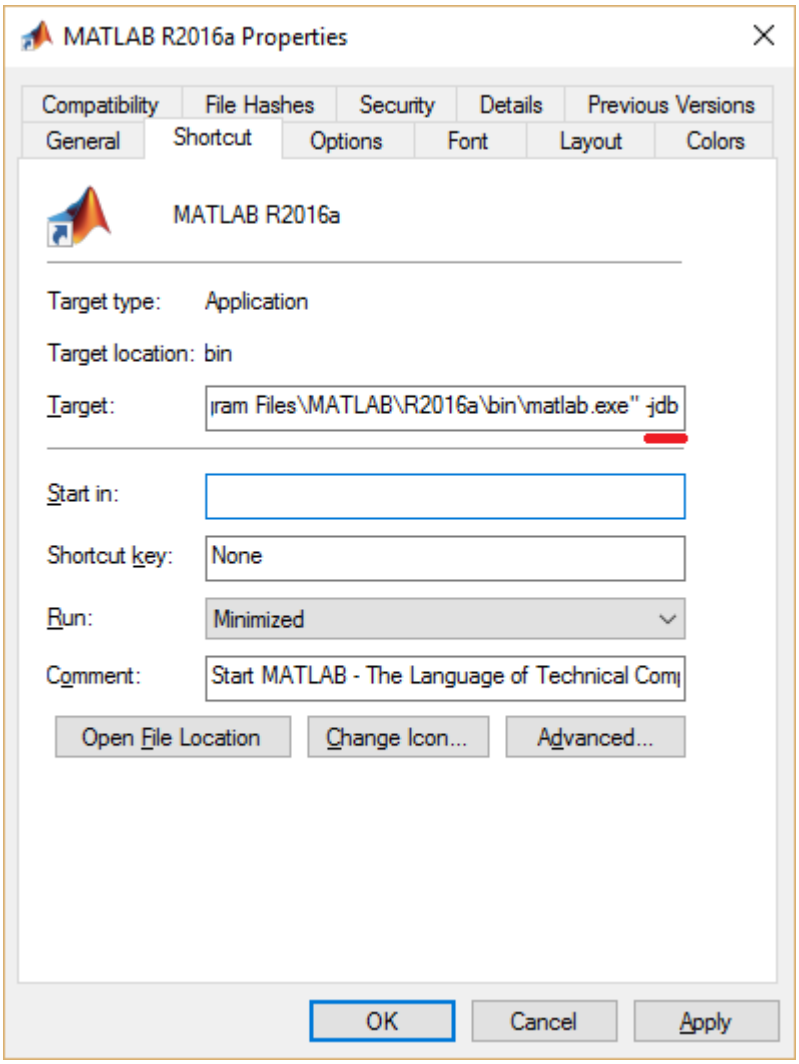
Cuando MATLAB se inicia en el modo de depuración JVM, aparece el siguiente mensaje en la ventana de comandos:

```
JVM is being started with debugging enabled.
Use "jdb -connect com.sun.jdi.SocketAttach:port=4444" to attach debugger.
```

Final de MATLAB

Windows:

Cree un acceso directo al ejecutable de MATLAB (`matlab.exe`) y agregue la bandera `-jdb` al final como se muestra a continuación:



Al ejecutar MATLAB utilizando este acceso directo, se habilitará la depuración de JVM.

Alternativamente, el archivo `java.opts` se puede crear / actualizar. Este archivo se almacena en "matlab-root \ bin \ arch", donde "matlab-root" es el directorio de instalación de MATLAB y "arch" es la arquitectura (por ejemplo, "win32").

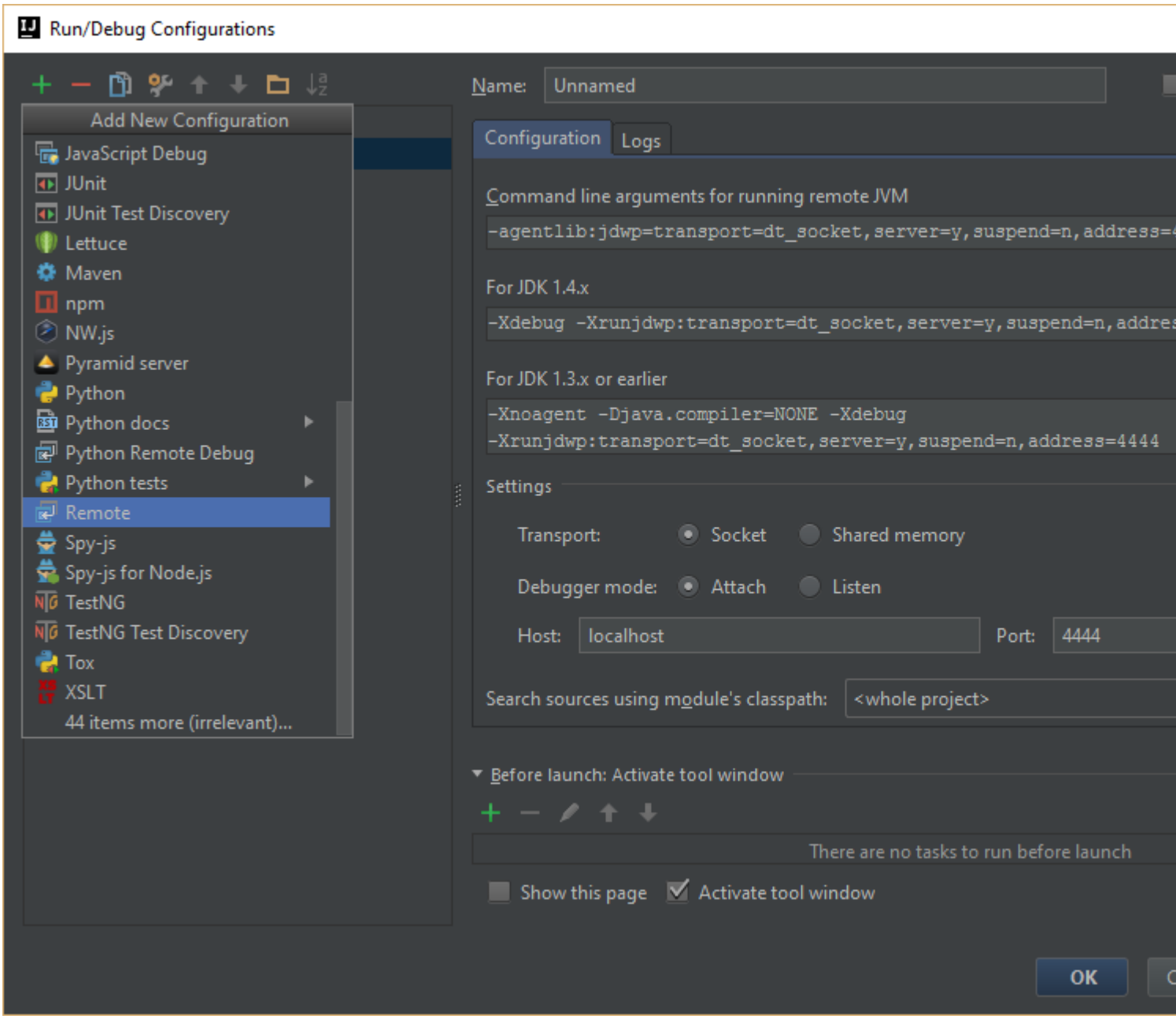
Se debe agregar lo siguiente en el archivo:

```
-Xdebug
-Xrunjdwp:transport=dt_socket,address=1044,server=y,suspend=n
```

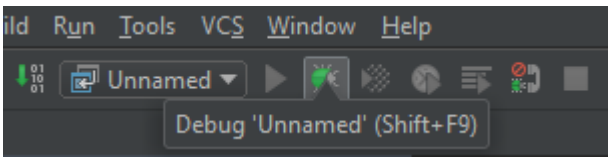
Final del depurador

IntelliJ IDEA

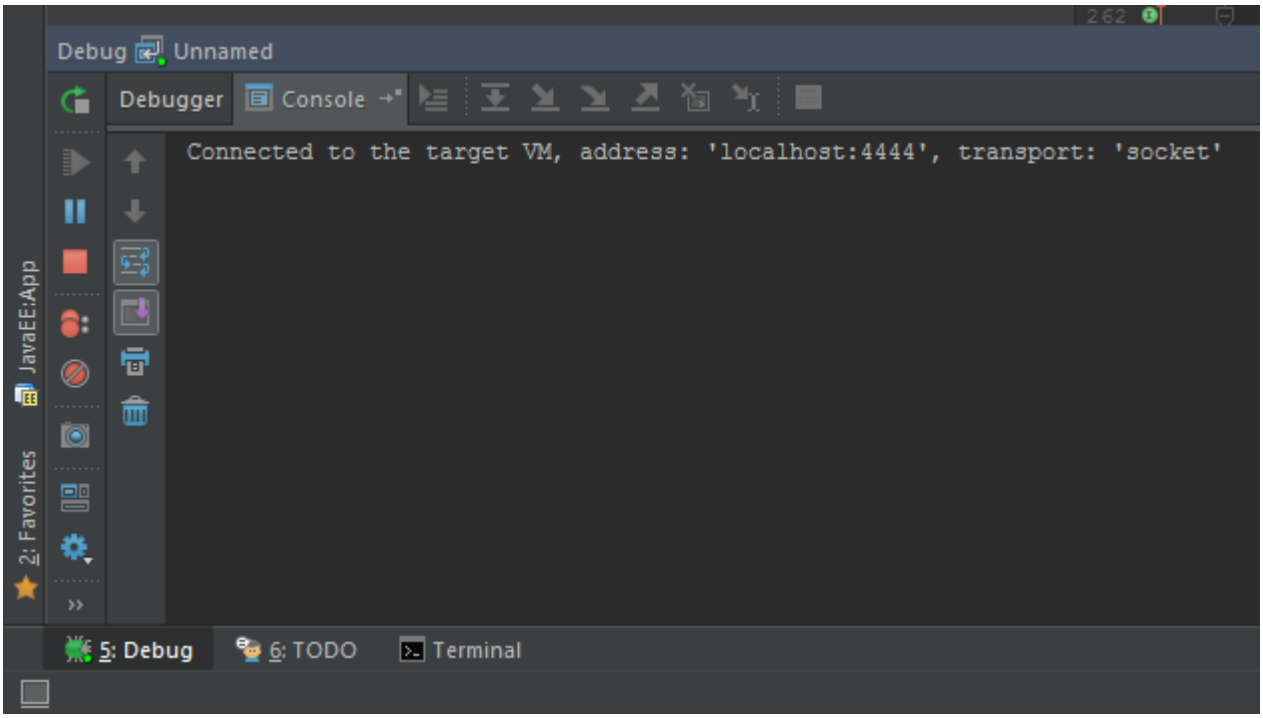
Adjuntar este depurador requiere la creación de una configuración de "depuración remota" con el puerto expuesto por MATLAB:



Entonces se inicia el depurador:



Si todo funciona como se esperaba, aparecerá el siguiente mensaje en la consola:



Lea Depuración en línea: https://riptutorial.com/es/matlab/topic/1045/depuracion

Capítulo 7: Dibujo

Examples

Círculos

La opción más fácil para dibujar un círculo es, obviamente, la función de `rectangle` .

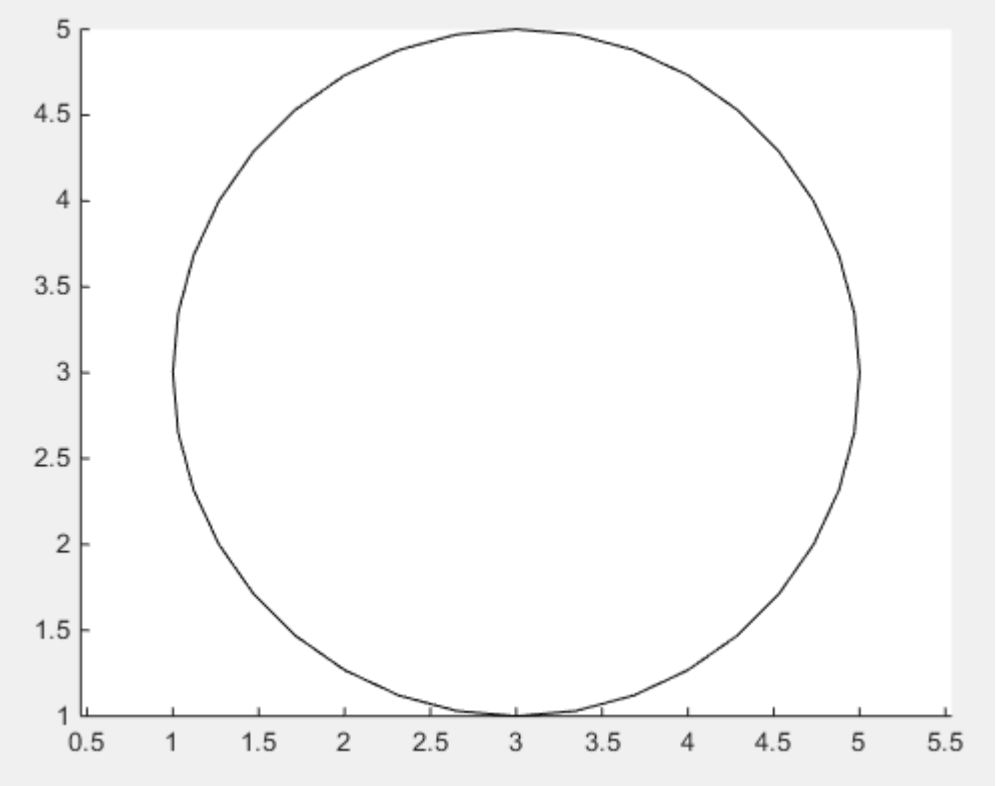
```
%// radius
r = 2;

%// center
c = [3 3];

pos = [c-r 2*r 2*r];
rectangle('Position',pos,'Curvature',[1 1])
axis equal
```

¡Pero la curvatura del rectángulo se debe establecer en **1** !

El vector de `position` define el rectángulo, los dos primeros valores `x` e `y` son la esquina inferior izquierda del rectángulo. Los dos últimos valores definen el ancho y la altura del rectángulo.

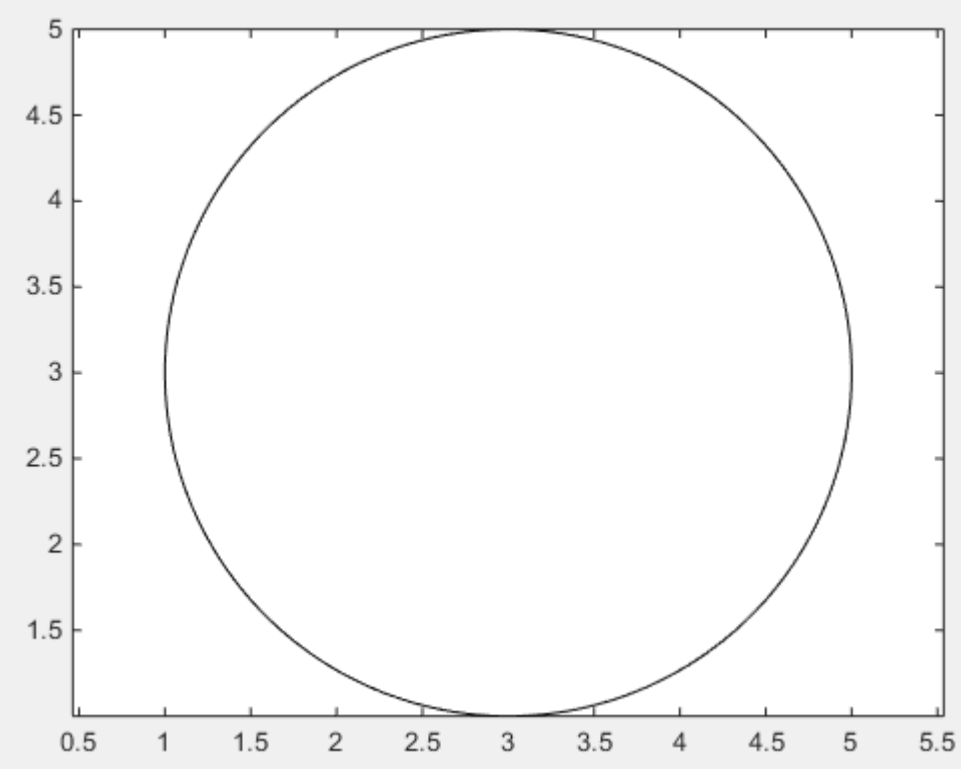
```
pos = [ [x y] width height ]
```

La *esquina* inferior izquierda del círculo, sí, este círculo tiene esquinas, aunque imaginarias, es el **centro** `c = [3 3]` **menos el radio** `r = 2` que es `[xy] = [1 1]` . **El ancho** y la **altura** son iguales al **diámetro** del círculo, entonces el `width = 2*r; height = width;`



En caso de que la suavidad de la solución anterior no sea suficiente, no hay manera de evitar el uso de la forma obvia de dibujar un círculo real mediante el uso de **funciones trigonométricas** .

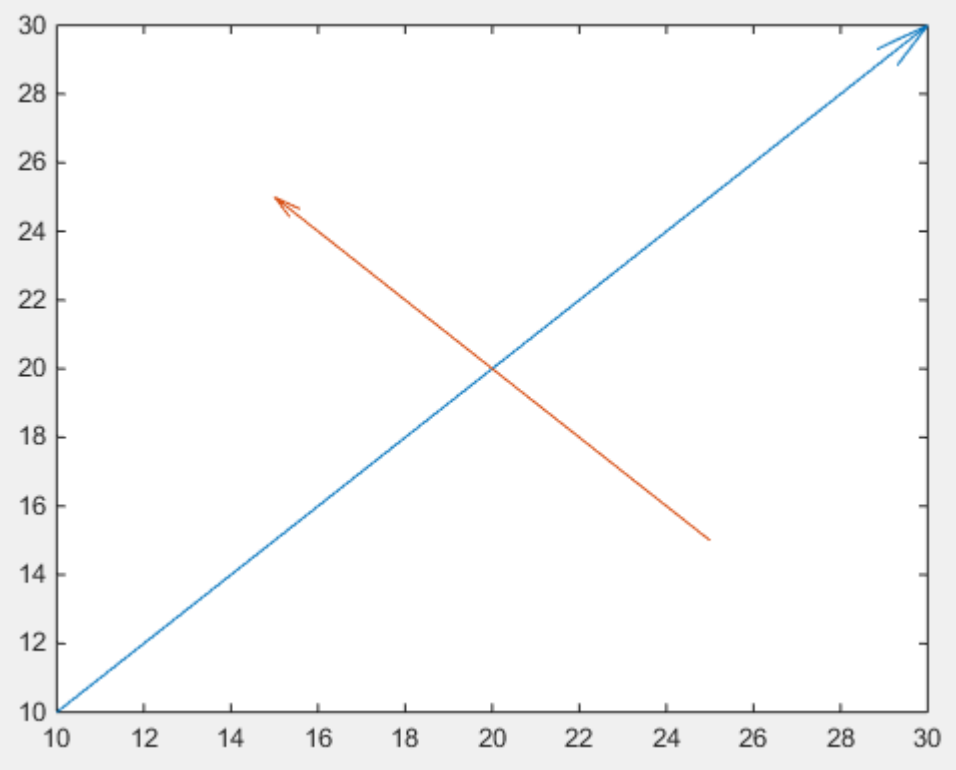
```
%// number of points
n = 1000;

%// running variable
t = linspace(0,2*pi,n);

x = c(1) + r*sin(t);
y = c(2) + r*cos(t);

%// draw line
line(x,y)

%// or draw polygon if you want to fill it with color
%// fill(x,y,[1,1,1])
axis equal
```



Flechas

En primer lugar, uno puede usar el `quiver` , en el que no tiene que lidiar con unidades de figuras no normalizadas mediante el uso de `annotation`

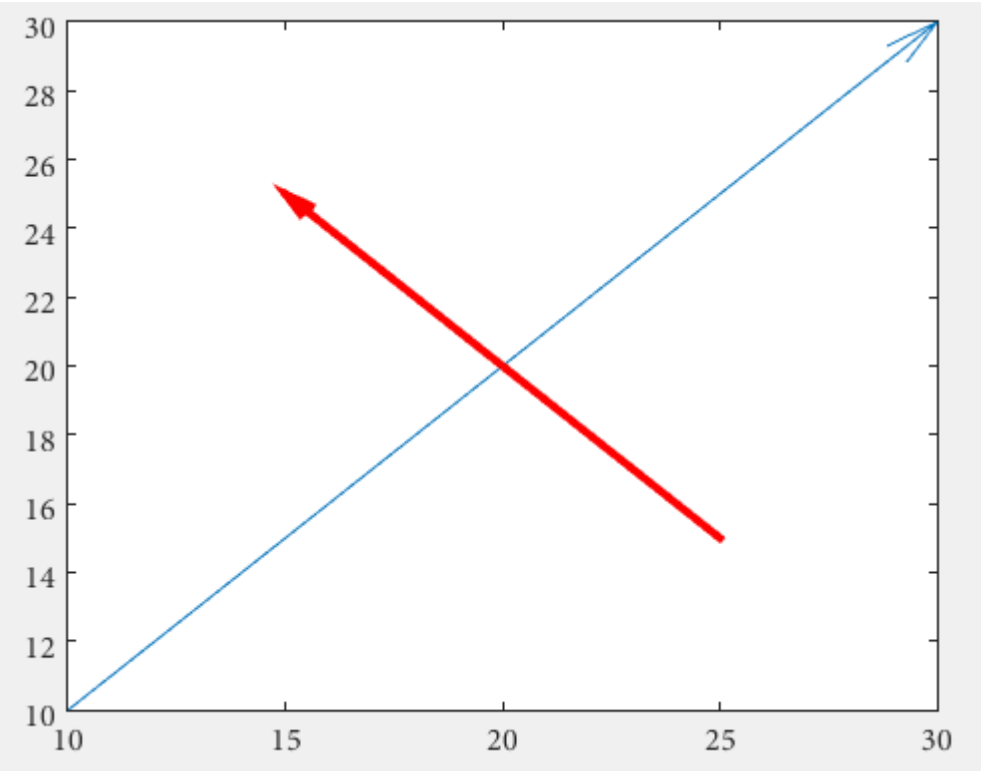
```
drawArrow = @(x,y) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0 )

x1 = [10 30];
y1 = [10 30];

drawArrow(x1,y1); hold on

x2 = [25 15];
y2 = [15 25];

drawArrow(x2,y2)
```



Importante es el quinto argumento de `quiver` : 0 que deshabilita una escala por defecto, ya que esta función se usa generalmente para trazar campos vectoriales. (o use el par de valores de propiedad `'AutoScale','off'`)

También se pueden agregar características adicionales:

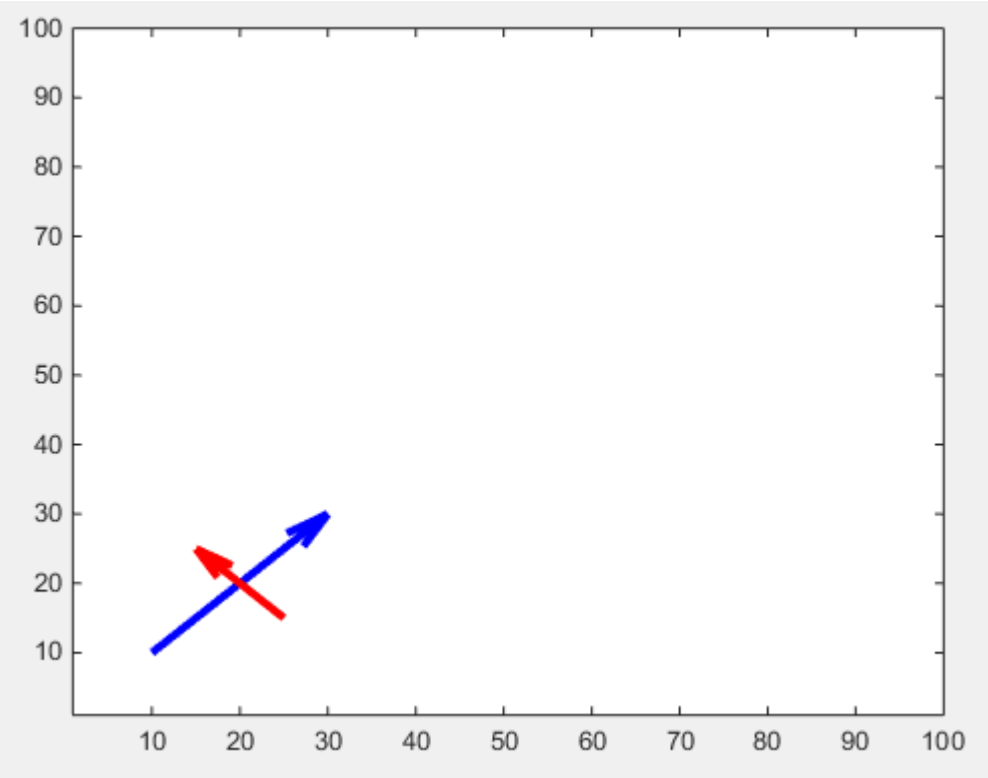
```
drawArrow = @(x,y,varargin) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0, varargin{:} )
drawArrow(x1,y1); hold on
drawArrow(x2,y2,'linewidth',3,'color','r')
```



Si se desean diferentes puntas de flecha, es necesario usar anotaciones (esta respuesta puede ser útil. ¿Cómo cambio el estilo de la punta de flecha en el diagrama de carcaj?).

El tamaño de la punta de flecha se puede ajustar con la propiedad `'MaxHeadSize'` . No es consistente por desgracia. Los límites de los ejes deben establecerse después.

```
x1 = [10 30];
y1 = [10 30];
drawArrow(x1,y1,{'MaxHeadSize',0.8,'Color','b','LineWidth',3}); hold on

x2 = [25 15];
y2 = [15 25];
drawArrow(x2,y2,{'MaxHeadSize',10,'Color','r','LineWidth',3}); hold on

xlim([1, 100])
ylim([1, 100])
```

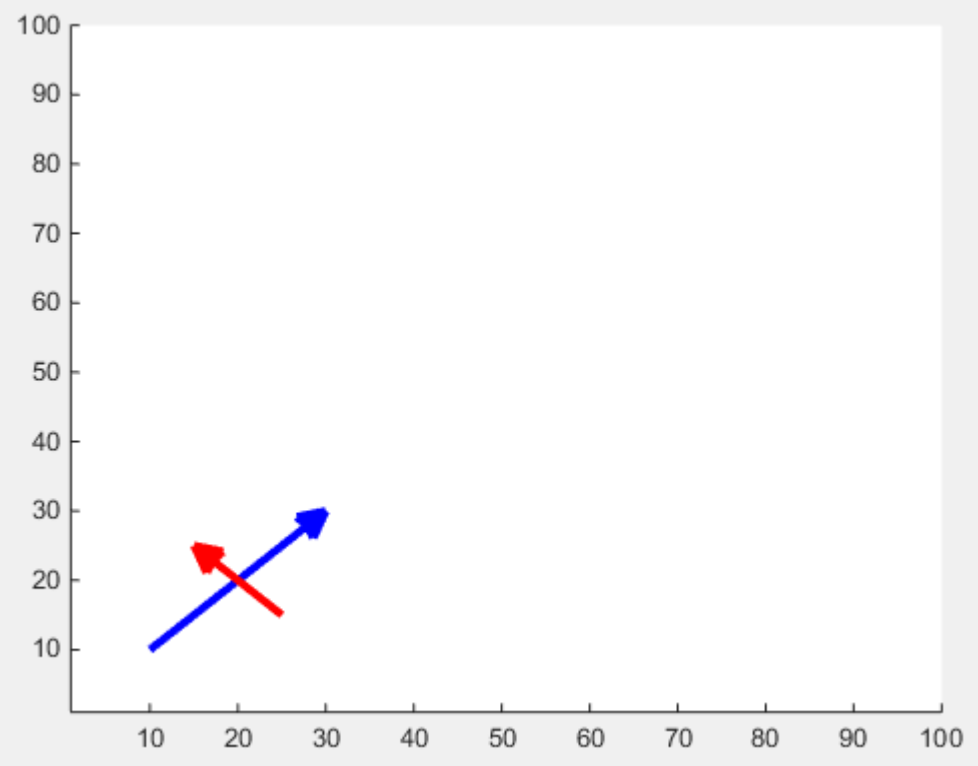


Hay otro tweak para cabezas de flecha ajustables:

```
function [ h ] = drawArrow( x,y,xlimits,ylimits,props )

xlim(xlimits)
ylim(ylimits)

h = annotation('arrow');
set(h,'parent', gca, ...
    'position', [x(1),y(1),x(2)-x(1),y(2)-y(1)], ...
    'HeadLength', 10, 'HeadWidth', 10, 'HeadStyle', 'cback1', ...
    props{:} );

end
```

que puede llamar desde su script de la siguiente manera:

```
drawArrow(x1,y1,[1, 100],[1, 100],{'Color','b','LineWidth',3}); hold on
drawArrow(x2,y2,[1, 100],[1, 100],{'Color','r','LineWidth',3}); hold on
```



Elipse

Para trazar una elipse puedes usar su ecuación . Una elipse tiene un eje mayor y otro menor. También queremos poder trazar la elipse en diferentes puntos centrales. Por lo tanto escribimos una función cuyas entradas y salidas son:

```
Inputs:
    r1,r2: major and minor axis respectively
    C: center of the ellipse (cx,cy)
Output:
    [x,y]: points on the circumference of the ellipse
```

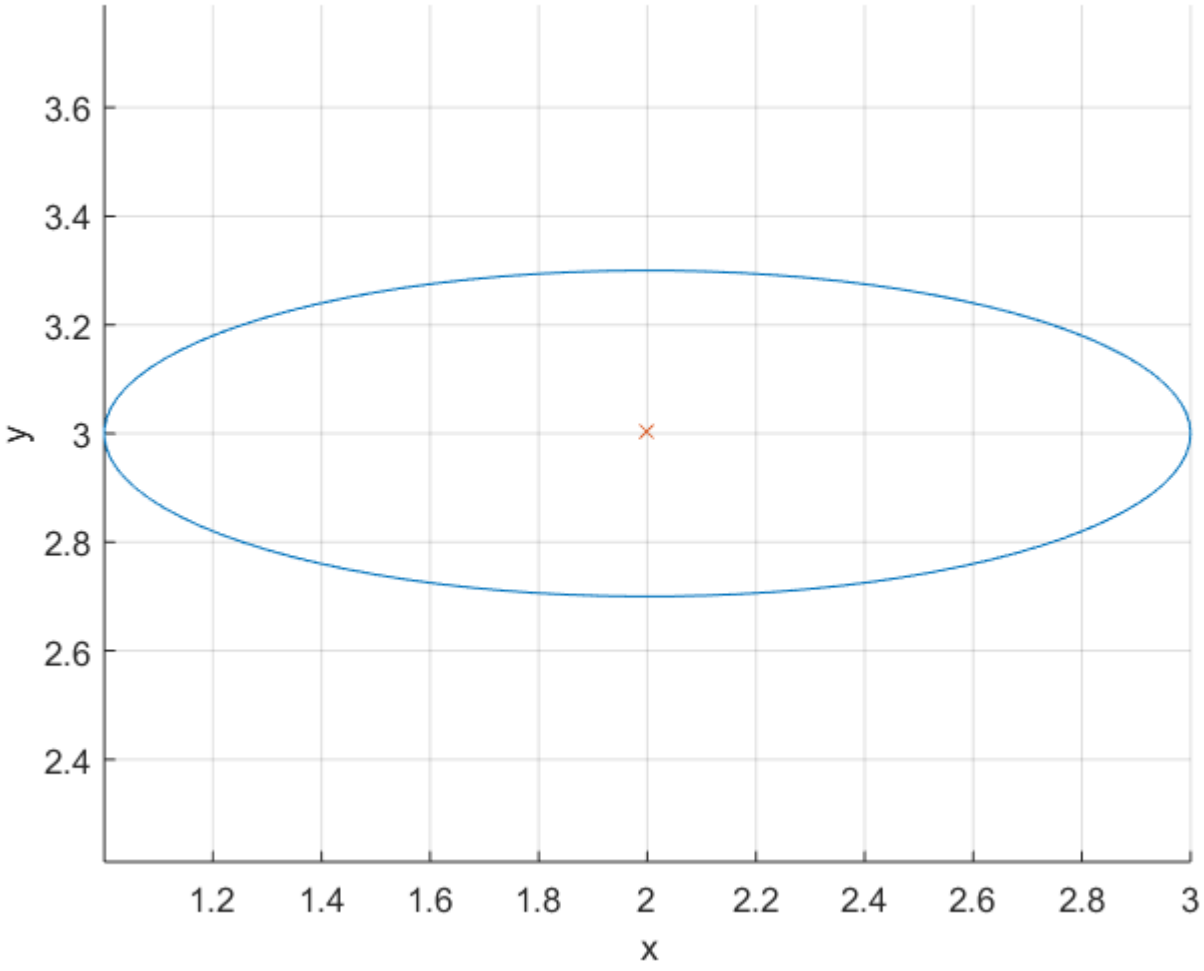
Puede usar la siguiente función para obtener los puntos en una elipse y luego trazar esos puntos.

```
function [x,y] = getEllipse(r1,r2,C)
beta = linspace(0,2*pi,100);
x = r1*cos(beta) - r2*sin(beta);
y = r1*cos(beta) + r2*sin(beta);
x = x + C(1,1);
y = y + C(1,2);
end
```

Ejemplo:

```
[x,y] = getEllipse(1,0.3,[2 3]);
```

```
plot(x,y);
```

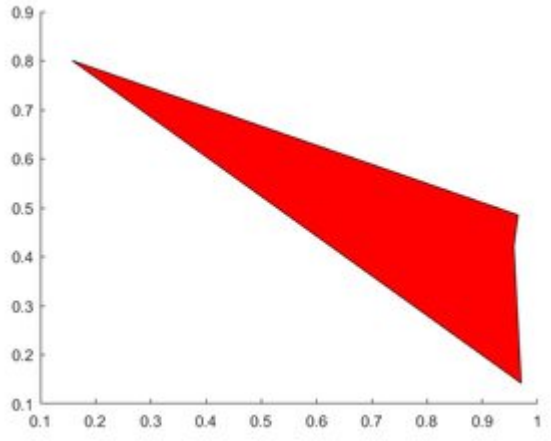


Polígono (s)

Cree vectores para contener las ubicaciones x e y de los vértices, alimente estos en el `patch` .

Polígono único

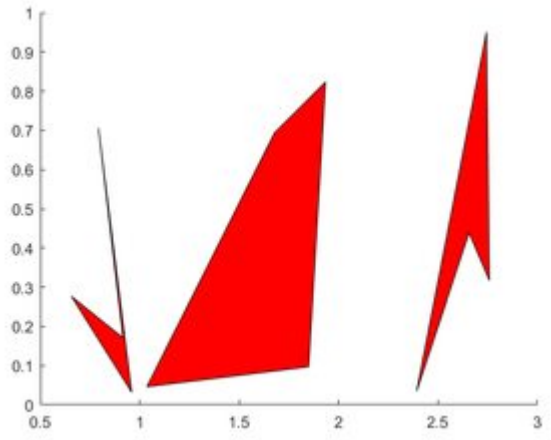
```
X=rand(1,4); Y=rand(1,4);
h=patch(X,Y,'red');
```



Poligonos multiples

Los vértices de cada polígono ocupan una columna de cada uno de `X` , `Y`

```
X=rand(4,3); Y=rand(4,3);
for i=2:3
  X(:,i)=X(:,i)+(i-1); % create horizontal offsets for visibility
end

h=patch(X,Y,'red');
```



Parcela pseudo 4D

Una matriz `(mxn)` puede estar representada por una superficie usando surf ;

El color de la superficie se establece automáticamente como función de los valores en la matriz `(mxn)` . Si no se especifica el mapa de colores , se aplica el predeterminado.

Se puede agregar una barra de colores para mostrar el mapa de colores actual e indicar la asignación de valores de datos en el mapa de colores.

En el siguiente ejemplo, la matriz `z (mxn)` es generada por la función:

```
z=x.*y.*sin(x).*cos(y);
```

en el intervalo `[-pi,pi]` . Los valores de `x` e `y` se pueden generar mediante la función meshgrid y la superficie se representa de la siguiente manera:

```
% Create a Figure
figure
% Generate the `x` and `y` values in the interval `[-pi,pi]`
[x,y] = meshgrid([-pi:.2:pi],[-pi:.2:pi]);
% Evaluate the function over the selected interval
z=x.*y.*sin(x).*cos(y);
% Use surf to plot the surface
S=surf(x,y,z);
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

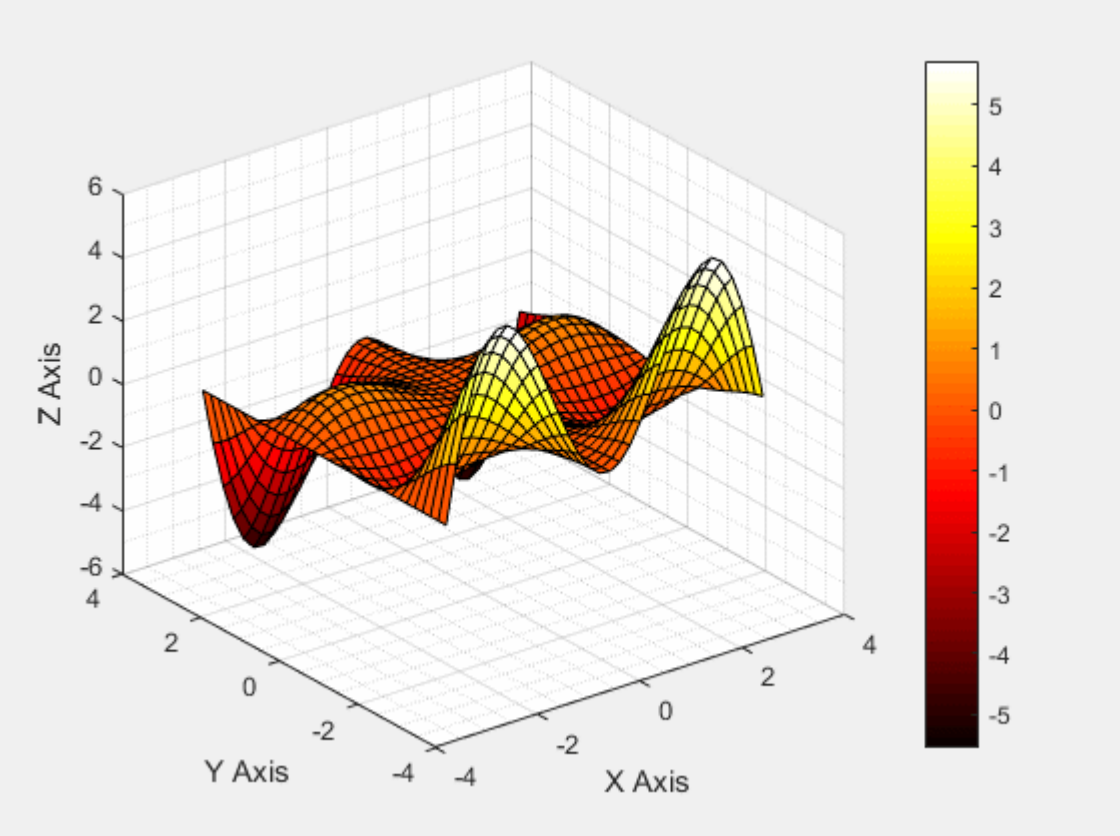


Figura 1

Ahora podría darse el caso de que información adicional esté vinculada a los valores de la matriz `z` y se almacene en otra matriz `(mxn)`

Es posible agregar esta información adicional en el gráfico modificando la forma en que se colorea la superficie.

Esto permitirá tener un poco de gráfico 4D: a la representación 3D de la superficie generada por la primera matriz `(mxn)` , la cuarta dimensión estará representada por los datos contenidos en la

segunda matriz `(mxn)` .

Es posible crear dicha trama llamando a `surf` con 4 entradas:

```
surf(x,y,z,C)
```

donde el parámetro `C` es la segunda matriz (que debe ser del mismo tamaño de `z`) y se utiliza para definir el color de la superficie.

En el siguiente ejemplo, la matriz `C` es generada por la función:

```
C=10*sin(0.5*(x.^2.+y.^2))*33;
```

sobre el intervalo `[-pi,pi]`

La superficie generada por `C` es

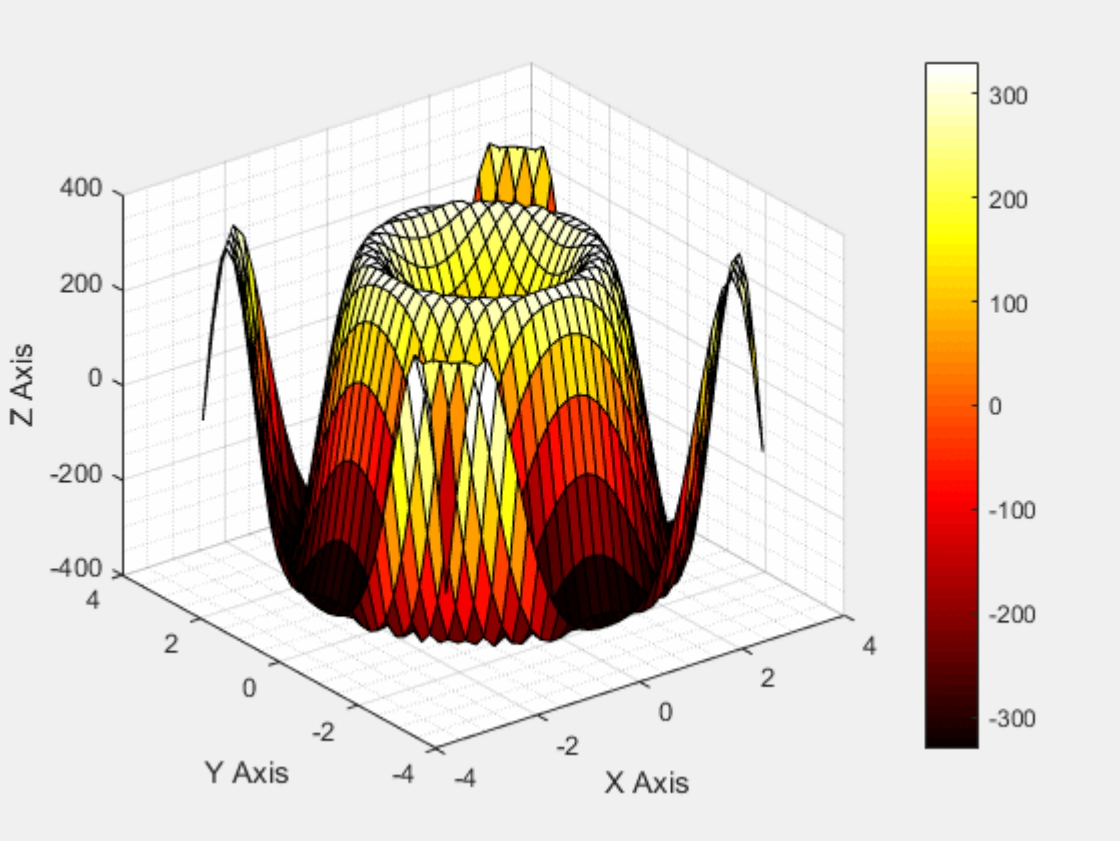


Figura 2

Ahora podemos llamar `surf` con cuatro entradas:

```
figure
surf(x,y,z,C)
% shading interp
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
```

```
colorbar
```

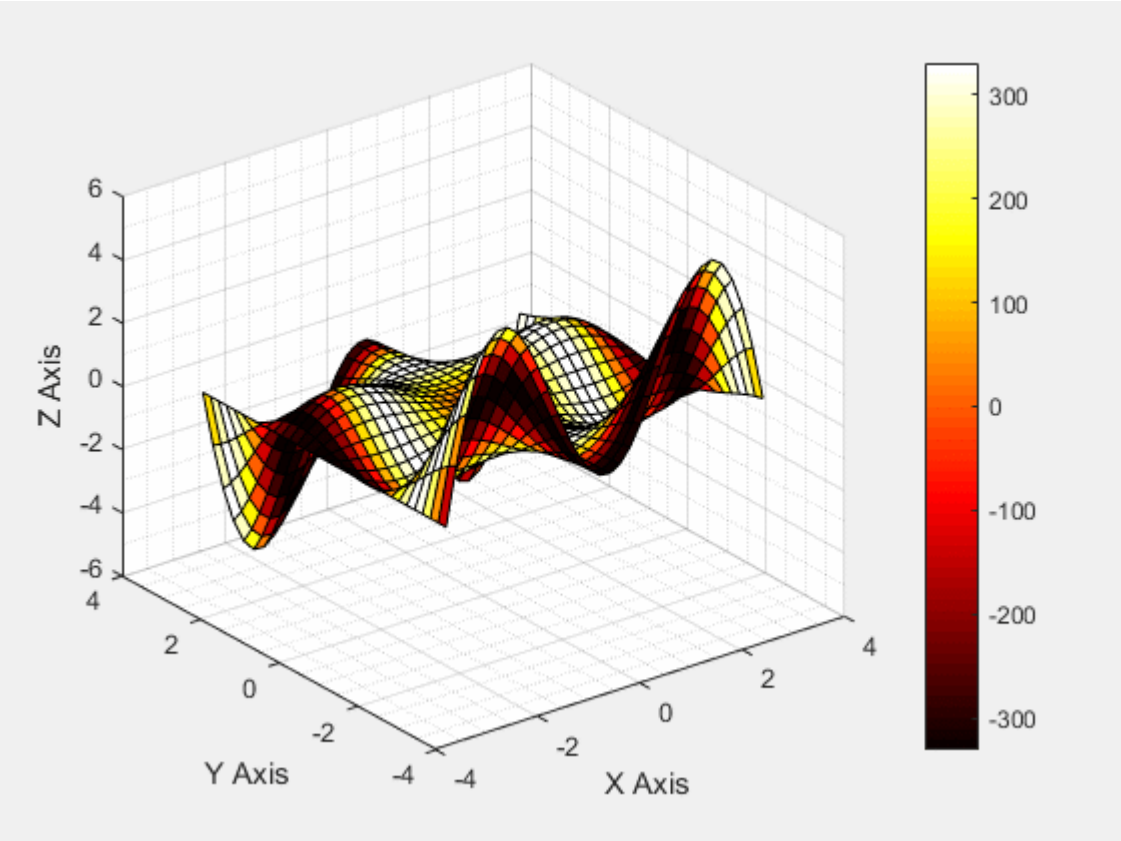


figura 3

Comparando la Figura 1 y la Figura 3, podemos notar que:

- la forma de la superficie corresponde a los valores `z` (la primera matriz `(mxn)`)
- el color de la superficie (y su rango, dado por la barra de colores) corresponde a los valores `C` (la primera matriz `(mxn)`)

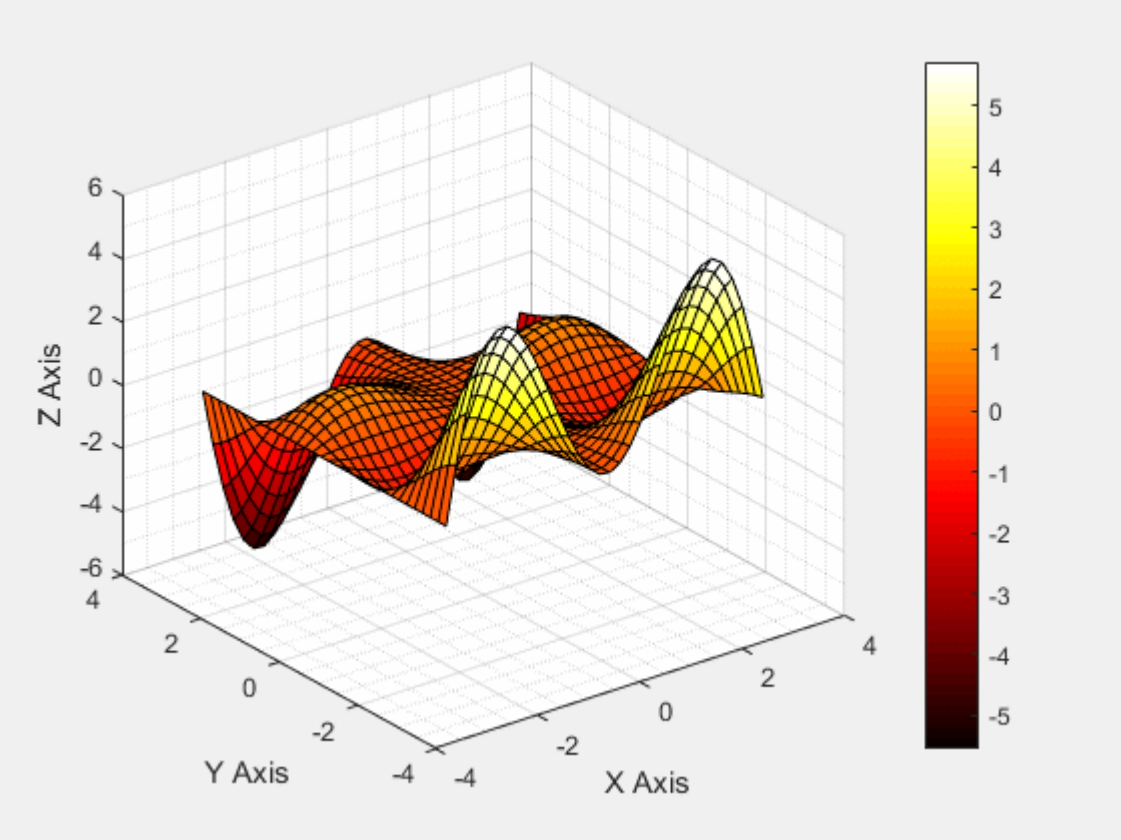
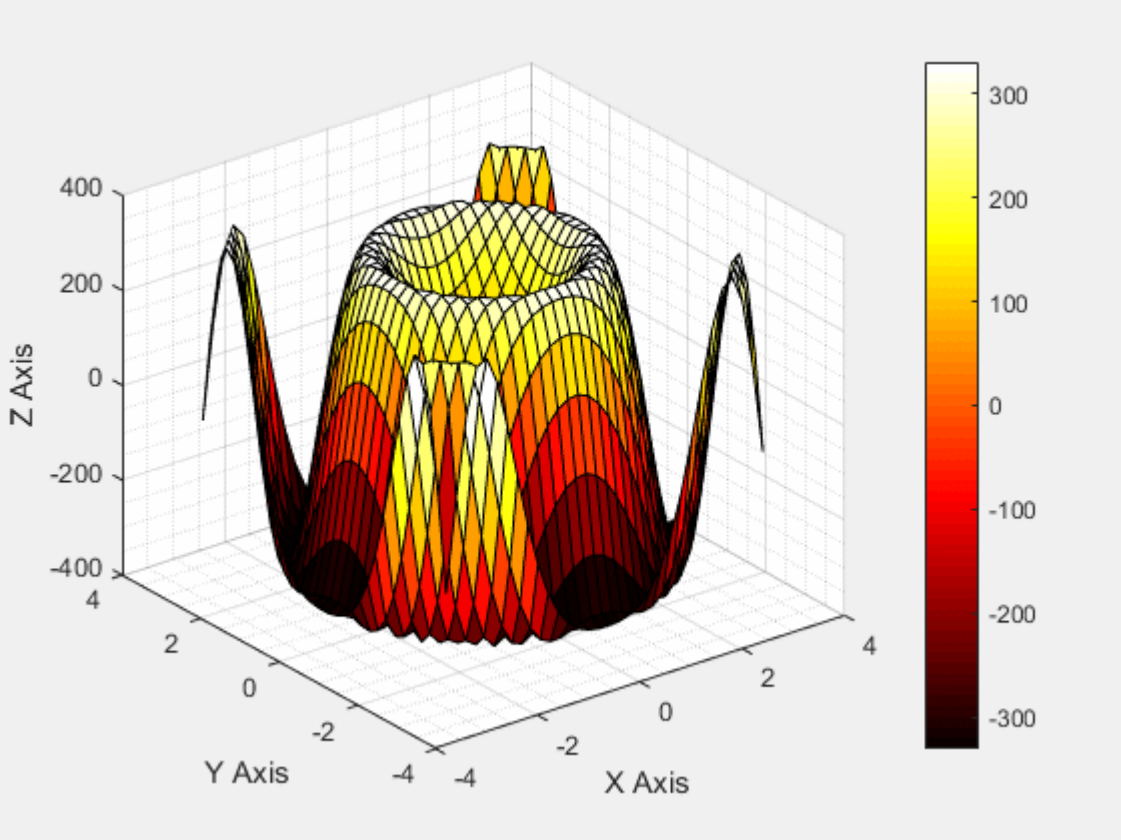


Figura 4

Por supuesto, es posible intercambiar `z` y `C` en la gráfica para tener la forma de la superficie dada por la matriz `C` y el color dado por la matriz `z` :

```
figure
surf(x,y,C,z)
% shading interp
xlabel('X Axis');
ylabel('Y Axis');
zlabel('Z Axis');
grid minor
colormap('hot')
colorbar
```

y comparar la Figura 2 con la Figura 4



Dibujo rapido

Hay tres formas principales de hacer una trama o animaciones secuenciales: `plot(x,y)` , `set(h , 'XData' , y, 'YData' , y)` y `animatedline` . Si desea que su animación sea fluida, necesita un dibujo eficiente y los tres métodos no son equivalentes.

```
% Plot a sin with increasing phase shift in 500 steps
x = linspace(0 , 2*pi , 100);

figure
tic
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);
    plot(x,y)
    drawnow
end
toc
```

Obtengo `5.278172 seconds` . La función de trazado básicamente elimina y recrea el objeto de línea cada vez. Una forma más eficiente de actualizar un gráfico es usar las propiedades `XData` y `YData` del objeto `Line` .

```
tic
h = [];   % Handle of line object
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);

    if isempty(h)
        % If Line still does not exist, create it
        h = plot(x,y);
```

```
    else
        % If Line exists, update it
        set(h , 'YData' , y)
    end
    drawnow
end
toc
```

Ahora tengo `2.741996 seconds` , ¡mucho mejor!

`animatedline` es una función relativamente nueva, introducida en 2014b. Veamos cómo le va:

```
tic
h = animatedline;
for thetha = linspace(0 , 10*pi , 500)
    y = sin(x + thetha);
    clearpoints(h)
    addpoints(h , x , y)
    drawnow
end
toc
```

`3.360569 seconds` , no tan bueno como actualizar una gráfica existente, pero aún mejor que la `plot(x,y)` .

Por supuesto, si tiene que trazar una sola línea, como en este ejemplo, los tres métodos son casi equivalentes y dan animaciones suaves. Pero si tiene trazados más complejos, la actualización de los objetos de `Line` existentes hará una diferencia.

Lea Dibujo en línea: https://riptutorial.com/es/matlab/topic/3978/dibujo

Capítulo 8: Errores comunes y errores.

Examples

No nombre una variable con un nombre de función existente

Ya existe una función `sum()` . Como resultado, si nombramos una variable con el mismo nombre

```
sum = 1+3;
```

y si intentamos usar la función mientras la variable aún existe en el área de trabajo

```
A = rand(2);
sum(A,1)
```

Obtendremos el **error** críptico:

```
Subscript indices must either be real positive integers or logicals.
```

`clear()` la variable primero y luego usar la función

```
clear sum

sum(A,1)
ans =
       1.0826       1.0279
```

¿Cómo podemos verificar si ya existe una función para evitar este conflicto?

Utilice `which()` con la bandera `-all` :

```
which sum -all
sum is a variable.
built-in (C:\Program Files\MATLAB\R2016a\toolbox\matlab\datafun\@double\sum)    % Shadowed
double method
...
```

Esta salida nos dice que la `sum` es primero una variable y que los siguientes métodos (funciones) están ensombrecidos, es decir, MATLAB primero intentará aplicar nuestra sintaxis a la variable, en lugar de usar el método.

Lo que ves NO es lo que obtienes: char vs cellstring en la ventana de comandos

Este es un ejemplo básico dirigido a nuevos usuarios. No se enfoca en explicar la diferencia entre `char` y `cellstring` .

Puede suceder que quieras deshacerte de `'` en tus cuerdas, aunque nunca las hayas agregado. De hecho, esos son *artefactos* que la **ventana de comando** usa para distinguir entre algunos tipos.

Una cadena se imprimirá

```
s = 'dsadasd'
s =
dsadasd
```

Se imprimirá una cadena de celdas .

```
c = {'dsadasd'};
c =
    'dsadasd'
```

Observe cómo las **comillas simples** y la **sangría** son artefactos para notificarnos que `c` es una `cellstring` lugar de un `char` . La cadena está de hecho contenida en la celda, es decir

```
c{1}
ans =
dsadasd
```

Los operadores de transposición.

- `.'` Es la forma correcta de **transponer** un vector o matriz en MATLAB.
- `'` es la forma correcta de tomar el **complejo transpuesto de conjugado** (también conocido como conjugado de Hermitian) de un vector o matriz en MATLAB.

Tenga en cuenta que para la transposición `.'` , hay un **punto** delante del apóstrofe. Esto está en consonancia con la sintaxis de las demás operaciones de elementos de MATLAB: `*` multiplica *matrices,*. `.*` Multiplica *elementos de matrices* juntos. Los dos comandos son muy similares, pero conceptualmente muy distintos. Al igual que otros comandos de MATLAB, estos operadores son "azúcar sintáctica" que se convierte en una llamada de función "adecuada" en tiempo de ejecución. Así como `==` convierte en una evaluación de la función eq , piensa en `.'` como la taquigrafía para la `transpose` . Si solo escribiera `'` (sin el punto), de hecho está utilizando el comando `ctranspose` , que calcula la transposición compleja del conjugado , que también se conoce como el conjugado de Hermitian , a menudo utilizado en la física. Mientras el vector o matriz transpuesta tenga un valor real, los dos operadores producen el mismo resultado. Pero tan pronto como tratemos con números complejos , inevitablemente tendremos problemas si no usamos la taquigrafía "correcta". Lo que es "correcto" depende de su aplicación.

Considere el siguiente ejemplo de una matriz `C` contiene números complejos:

```
>> C = [1i, 2; 3*1i, 4]
C =
   0.0000 + 1.0000i   2.0000 + 0.0000i
   0.0000 + 3.0000i   4.0000 + 0.0000i
```

Tomemos la *transposición* usando la taquigrafía `.'` (con el periodo). La salida es la esperada, la forma transpuesta de `C`

```
>> C.'
ans =
   0.0000 + 1.0000i   0.0000 + 3.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i
```

Ahora, vamos a usar `'` (sin el punto). Vemos que, además de la transposición, los valores complejos también se han transformado en sus *conjugados complejos* .

```
>> C'
ans =
   0.0000 - 1.0000i   0.0000 - 3.0000i
   2.0000 + 0.0000i   4.0000 + 0.0000i
```

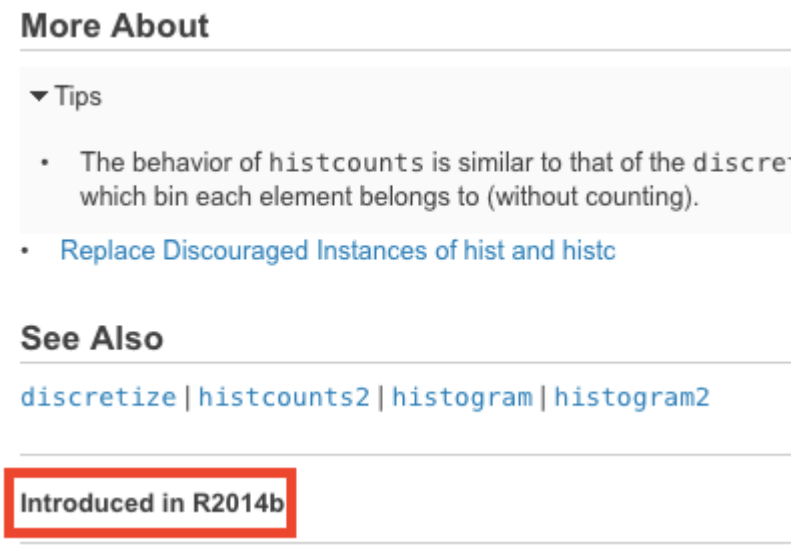
En resumen, si pretende calcular el conjugado de Hermitian, el complejo de transposición de conjugado, use `'` (sin el punto). Si solo desea calcular la transposición sin conjugar complejos los valores, utilice `.'` (con el periodo).

Función indefinida o método X para argumentos de entrada de tipo Y

Esta es la manera larga de MATLAB de decir que no puede encontrar la función a la que intenta llamar. Hay varias razones por las que puedes obtener este error:

Esa función fue introducida *después de* su versión actual de MATLAB

La documentación en línea de MATLAB proporciona una característica muy agradable que le permite determinar en qué versión se introdujo una función determinada. Se encuentra en la parte inferior izquierda de cada página de la documentación:

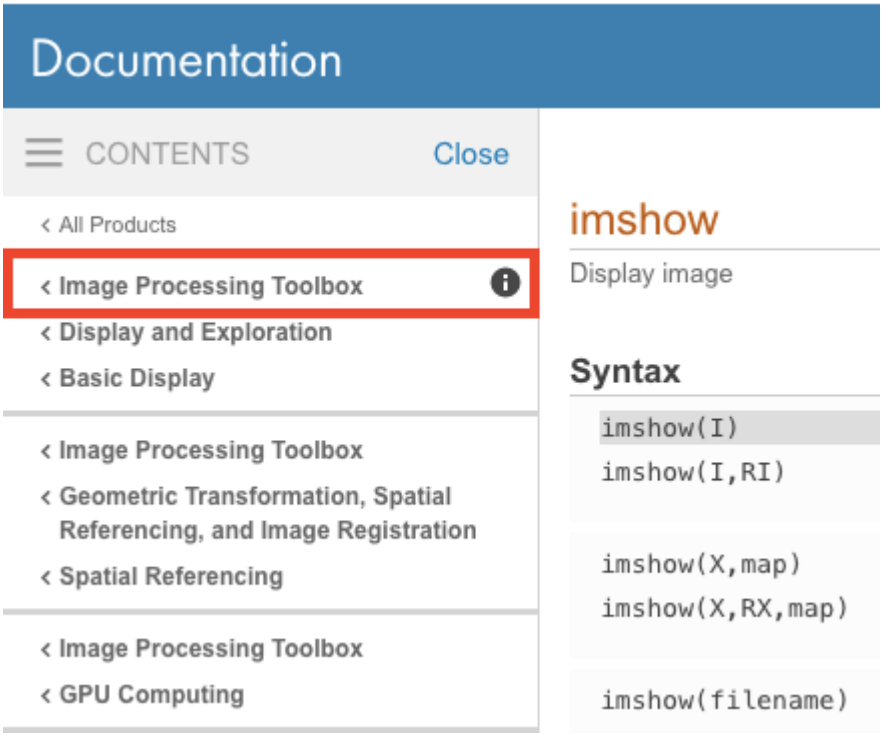


Compare esta versión con su propia versión actual (`ver`) para determinar si esta función está disponible en su versión particular. Si no es así, intente buscar las versiones archivadas de la documentación para encontrar una alternativa adecuada para su versión.

¡No tienes esa caja de herramientas!

La instalación base de MATLAB tiene un gran número de funciones; sin embargo, las funcionalidades más especializadas se empaquetan en cajas de herramientas y se venden por separado por Mathworks. La documentación de *todas las* cajas de herramientas está visible tanto si tiene la caja de herramientas como si no, así que asegúrese de verificar y ver si tiene la caja de herramientas adecuada.

Para verificar a qué caja de herramientas pertenece una función determinada, consulte la parte superior izquierda de la documentación en línea para ver si se menciona una caja de herramientas específica.



Luego, puede determinar qué cajas de herramientas ha instalado su versión de MATLAB emitiendo el comando `ver` que imprimirá una lista de todas las cajas de herramientas instaladas.

Si no tiene esa caja de herramientas instalada y desea usar la función, deberá comprar una licencia para esa caja de herramientas en particular de The Mathworks.

MATLAB no puede localizar la función

Si MATLAB aún no puede encontrar su función, entonces debe ser una función definida por el usuario. Es posible que se encuentre en otro directorio y ese directorio se debe agregar a la ruta de búsqueda para que se ejecute su código. Puede verificar si MATLAB puede localizar su

función usando `which` que debería devolver la ruta al archivo fuente.

Tenga en cuenta la inexactitud del punto flotante

Los números de punto flotante no pueden representar todos los números reales. Esto se conoce como inexactitud de punto flotante.

Hay infinitos números de puntos flotantes y pueden ser infinitamente largos (por ejemplo, `π`), por lo que ser capaz de representarlos perfectamente requeriría una cantidad infinita de memoria. Al ver que esto era un problema, se diseñó una representación especial para el almacenamiento de "número real" en computadora, el estándar IEEE 754 . En resumen, describe cómo las computadoras almacenan este tipo de números, con un exponente y una mantisa, como,

```
floatnum = sign * 2^exponent * mantissa
```

Con una cantidad limitada de bits para cada uno de estos, solo se puede lograr una precisión finita. Cuanto menor sea el número, menor será el espacio entre los posibles números (y viceversa). Puedes probar tus números reales en esta demostración en línea .

Tenga en cuenta este comportamiento e intente evitar todas las comparaciones de puntos flotantes y su uso como condiciones de detención en los bucles. Vea a continuación dos ejemplos:

Ejemplos: comparación de punto flotante MAL:

```
>> 0.1 + 0.1 + 0.1  == 0.3

ans =

  logical

   0
```

Es una mala práctica usar la comparación de punto flotante como se muestra en el ejemplo anterior. Puede superarlo tomando el valor absoluto de su diferencia y comparándolo con un nivel de tolerancia (pequeño).

A continuación se muestra otro ejemplo, donde se usa un número de punto flotante como condición de detención en un bucle while: **

```
k = 0.1;
while k <= 0.3
  disp(num2str(k));
  k = k + 0.1;
end

% --- Output: ---
0.1
0.2
```

Se pierde el último bucle esperado (`0.3 <= 0.3`).

Ejemplo: comparación de punto flotante a la DERECHA:

```
x = 0.1 + 0.1 + 0.1;
y = 0.3;
tolerance = 1e-10; % A "good enough" tolerance for this case.

if ( abs( x - y ) <= tolerance )
  disp('x == y');
else
  disp('x ~= y');
end

% --- Output: ---
x == y
```

Varias cosas a tener en cuenta:

- Como era de esperar, ahora `x` e `y` se tratan como equivalentes.
- En el ejemplo anterior, la elección de la tolerancia se hizo de forma arbitraria. Por lo tanto, el valor elegido puede no ser adecuado para todos los casos (especialmente cuando se trabaja con números mucho más pequeños). La elección *inteligente* del límite se puede hacer utilizando la función `eps` , es decir, `N*eps(max(x,y))` , donde `N` es un número específico del problema. Una opción razonable para `N` , que también es lo suficientemente permisiva, es `1E2` (aunque, en el problema anterior, `N=1` también sería suficiente).

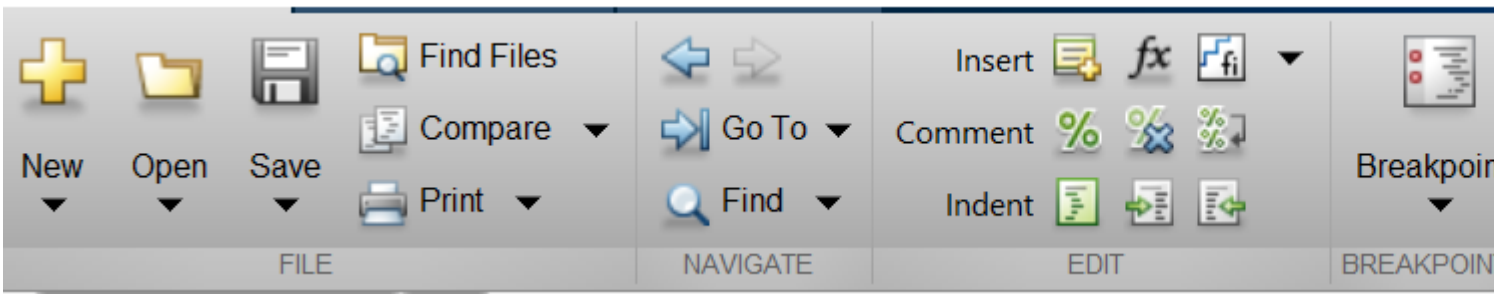
Otras lecturas:

Vea estas preguntas para obtener más información sobre la inexactitud de punto flotante:

- ¿Por qué 24.0000 no es igual a 24.0000 en MATLAB?
- ¿Es rota la matemática de punto flotante?

los argumentos de entrada no son suficientes

A menudo, los desarrolladores principiantes de MATLAB utilizarán el editor de MATLAB para escribir y editar código, en particular funciones personalizadas con entradas y salidas. Hay un botón *Ejecutar* en la parte superior que está disponible en las versiones recientes de MATLAB:



Una vez que el desarrollador termina con el código, a menudo se ven tentados a presionar el botón *Ejecutar* . Para algunas funciones esto funcionará bien, pero para otras recibirán un error de `Not enough input arguments` y quedarán desconcertados acerca de por qué ocurre el error.

La razón por la que este error puede no ocurrir es porque escribió un script de MATLAB o una función que no admite argumentos de entrada. El uso del botón *Ejecutar* ejecutará un script de prueba o ejecutará una función suponiendo que no hay argumentos de entrada. Si su función requiere argumentos de entrada, se producirá el error `Not enough input arguments` escribir una función que espera que las entradas entren dentro de la función. Por lo tanto, no puede esperar que la función se ejecute simplemente presionando el botón *Ejecutar* .

Para demostrar este problema, supongamos que tenemos una función `mult` que simplemente multiplica dos matrices juntas:

```
function C = mult(A, B)
    C = A * B;
end
```

En las versiones recientes de MATLAB, si escribió esta función y presionó el botón *Ejecutar* , le dará el error que esperamos:

```
>> mult
Not enough input arguments.

Error in mult (line 2)
    C = A * B;
```

Hay dos formas de resolver este problema:

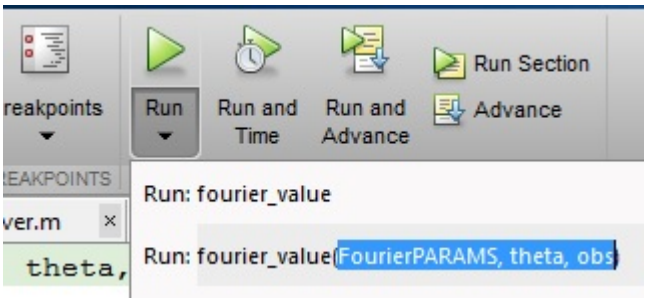
Método # 1 - A través del símbolo del sistema

Simplemente cree las entradas que necesita en la solicitud de comando, luego ejecute la función usando esas entradas que ha creado:

```
A = rand(5,5);
B = rand(5,5);
C = mult(A,B);
```

Método # 2 - Interactivamente a través del Editor

Debajo del botón *Ejecutar* , hay una flecha negra oscura. Si hace clic en esa flecha, puede especificar las variables que desea obtener del área de trabajo de MATLAB escribiendo la forma en que desea llamar a la función exactamente como lo ha visto en el método # 1. Asegúrese de que las variables que está especificando dentro de la función existen en el espacio de trabajo de MATLAB:



Cuidado con los cambios de tamaño de matriz

Algunas operaciones comunes en MATLAB, como la **diferenciación** o la **integración** , producen resultados que tienen una cantidad de elementos diferente a la que tienen los datos de entrada. Este hecho se puede pasar por alto fácilmente, lo que generalmente causaría errores, como las `Matrix dimensions must agree` . Considere el siguiente ejemplo:

```
t = 0:0.1:10;         % Declaring a time vector
y = sin(t);           % Declaring a function

dy_dt = diff(y);      % calculates dy/dt for y = sin(t)
```

Digamos que queremos trazar estos resultados. Echamos un vistazo a los tamaños de matriz y vemos:

```
size(y) is 1x101
size(t) is 1x101
```

Pero:

```
size(dy_dt) is 1x100
```

La matriz es un elemento más corto!

Ahora imagine que tiene datos de medición de posiciones a lo largo del tiempo y desea calcular el *tirón (t)* , obtendrá una matriz de 3 elementos menos que la matriz de tiempo (porque el tirón es la posición diferenciada 3 veces).

```
vel = diff(y);       % calculates velocity vel=dy/dt for y = sin(t)  size(vel)=1x100
acc = diff(vel);     % calculates acceleration acc=d(vel)/dt         size(acc)=1x99
jerk = diff(acc);    % calculates jerk jerk=d(acc)/dt                size(jerk)=1x98
```

Y luego operaciones como:

```
x = jerk .* t;          % multiplies jerk and t element wise
```

Devuelve errores, porque las dimensiones de la matriz no concuerdan.

Para calcular operaciones como las anteriores, debe ajustar el tamaño del arreglo más grande para que se ajuste al tamaño más pequeño. También puede ejecutar una regresión (`polyfit`) con

sus datos para obtener un polinomio para sus datos.

Errores de desajuste en la dimensión

Los errores de **desajuste de dimensión** suelen aparecer cuando:

- No prestar atención a la forma de las variables devueltas de las llamadas de función / método. En muchas funciones integradas de MATLAB, las matrices se convierten en vectores para acelerar los cálculos, y la variable devuelta podría ser un vector en lugar de la matriz que esperábamos. Este es también un escenario común cuando se trata de enmascaramiento lógico .
- Usar tamaños de matriz incompatibles al invocar la expansión de matriz implícita .

El uso de "i" o "j" como unidad imaginaria, índices de bucle o variable común.

Recomendación

Debido a que los símbolos `i` y `j` pueden representar cosas significativamente diferentes en MATLAB, su uso como índices de bucle ha dividido la comunidad de usuarios de MATLAB desde hace años. Si bien algunas razones históricas de rendimiento podrían ayudar a que el equilibrio se incline hacia un lado, este ya no es el caso y ahora la elección depende exclusivamente de usted y de las prácticas de codificación que elija seguir.

Las recomendaciones oficiales actuales de Mathworks son:

- Como `i` es una función, se puede anular y utilizar como variable. Sin embargo, es mejor evitar el uso de `i` y `j` para nombres de variables si pretende usarlos en aritmética compleja.
- Para la velocidad y la robustez mejorada en aritmética compleja, use `1i` y `1j` lugar de `i` y `j` .

Defecto

En MATLAB, de forma predeterminada, las letras `i` y `j` son nombres de `function` , que se refieren a la unidad imaginaria en el dominio complejo.

Entonces, por defecto `i = j = sqrt(-1)` .

```
>> i
ans =
   0.0000 + 1.0000i
>> j
ans =
   0.0000 + 1.0000i
```

y como es de esperar:

```
>> i^2
ans =
    -1
```

Usándolos como una variable (para índices de bucle u otra variable)

MATLAB permite usar el nombre de la función incorporada como una variable estándar. En este caso, el símbolo utilizado no apuntará más a la función incorporada sino a su propia variable definida por el usuario. Esta práctica, sin embargo, generalmente no se recomienda, ya que puede generar confusión, dificultad de depuración y mantenimiento (*consulte otro ejemplo de nombre-no-nombre-una-variable-con-un-existente-función-nombre*).

Si es ultra pedante con respecto a las convenciones y las mejores prácticas, evitará usarlas como índices de bucle en este idioma. Sin embargo, el compilador lo permite y es perfectamente funcional, por lo que también puede optar por mantener los viejos hábitos y utilizarlos como iteradores de bucle.

```
>> A = nan(2,3);
>> for i=1:2        % perfectly legal loop construction
       for j = 1:3
        A(i, j) = 10 * i + j;
       end
   end
```

Tenga en cuenta que los índices de bucle no quedan fuera del alcance al final del bucle, por lo que mantienen su nuevo valor.

```
>> [ i ; j ]
ans =
     2
     3
```

En el caso de que los utilice como variable, asegúrese de **que estén inicializados** antes de que se utilicen. En el bucle anterior, MATLAB los inicializa automáticamente cuando prepara el bucle, pero si no se inicializa correctamente, puede ver rápidamente que puede introducir inadvertidamente números `complex` en su resultado.

Si más adelante, necesita deshacer el sombreado de la función incorporada (= por ejemplo, desea que `i` y `j` representen la unidad imaginaria nuevamente), puede `clear` las variables:

```
>> clear i j
```

Ahora entiende la reserva de Mathworks acerca de usarlos como índices de bucle *si pretende usarlos en aritmética compleja* . Su código estaría plagado de inicializaciones variables y comandos `clear` , la mejor manera de confundir al programador más serio (¡ *sí, allí! ...*) y los accidentes del programa que están a punto de suceder.

Si no se espera una aritmética compleja, el uso de `i` y `j` es perfectamente funcional y no hay penalización de rendimiento.

Usándolos como unidad imaginaria:

Si su código tiene que lidiar con números `complex` , entonces `i` y `j` ciertamente serán útiles. Sin embargo, por motivos de desambiguación e incluso para actuaciones, se recomienda utilizar la forma completa en lugar de la sintaxis abreviada. La forma completa es `1i` (o `1j`).

```
>> [ i ; j ; 1i ; 1j]
ans =
   0.0000 + 1.0000i
   0.0000 + 1.0000i
   0.0000 + 1.0000i
   0.0000 + 1.0000i
```

Ellos representan el mismo valor `sqrt(-1)` , pero la forma posterior:

- Es más explícito, de forma semántica.
- es más fácil de mantener (alguien que mire su código más adelante no tendrá que leerlo para saber si `i` o `j` era una variable o la unidad imaginaria).
- Es más rápido (fuente: Mathworks).

Tenga en cuenta que la sintaxis completa `1i` es válida con cualquier número que preceda al símbolo:

```
>> a = 3 + 7.8j
a =
   3.0000 + 7.8000i
```

Esta es la única función que puede mantener con un número sin un operador entre ellos.

Escollos

Si bien su uso como *unidad* **O** *variable imaginaria* es perfectamente legal, aquí hay solo un pequeño ejemplo de lo confuso que podría resultar si se mezclan ambos usos:

Vamos a anular `i` y convertirlo en una variable:

```
>> i=3
i =
     3
```

Ahora `i` es una *variable* (que contiene el valor `3`), pero solo anulamos la notación *abreviada* de la unidad imaginaria, la forma completa aún se interpreta correctamente:

```
>> 3i
```

```
ans =
   0.0000 + 3.0000i
```

Lo que ahora nos permite construir las formulaciones más oscuras. Te dejo evaluar la legibilidad de todos los siguientes constructos:

```
>> [ i ; 3i ; 3*i ; i+3i ; i+3*i ]
ans =
   3.0000 + 0.0000i
   0.0000 + 3.0000i
   9.0000 + 0.0000i
   3.0000 + 3.0000i
  12.0000 + 0.0000i
```

Como puede ver, cada valor en la matriz anterior devuelve un resultado diferente. Si bien cada resultado es válido (siempre que esa haya sido la intención inicial), la mayoría de ustedes admitirán que sería una pesadilla adecuada leer un código plagado de tales construcciones.

Usando `length` para arrays multidimensionales

Un error común que tienen los codificadores de MATLAB es usar la función de `length` para las matrices (a diferencia de los **vectores** , para los cuales está destinada). La función de `length` , como se menciona en su documentación , " *devuelve la longitud de la dimensión de matriz más grande* " de la entrada.

Para los vectores, el valor de retorno de la `length` tiene dos significados diferentes:

1. El número total de elementos en el vector.
2. La dimensión más grande del vector.

A diferencia de los vectores, los valores anteriores no serían iguales para las matrices de más de una dimensión no individual (es decir, cuyo tamaño es mayor que `1`). Es por esto que el uso de la `length` para matrices es ambiguo. En su lugar, se recomienda utilizar una de las siguientes funciones, incluso cuando se trabaja con vectores, para que la intención del código quede perfectamente clara:

1. `size(A)` : devuelve un vector de fila cuyos elementos contienen la cantidad de elementos a lo largo de la dimensión correspondiente de `A`
2. `numel(A)` - devuelve el número de elementos en `A` Equivalente a `prod(size(A))` .
3. `ndims(A)` : devuelve el número de dimensiones en la matriz `A` Equivalente a `numel(size(A))` .

Esto es especialmente importante cuando se escriben funciones de biblioteca vectorizadas "a prueba de futuro", cuyas entradas no se conocen de antemano y pueden tener varios tamaños y formas.

Lea Errores comunes y errores. en línea: https://riptutorial.com/es/matlab/topic/973/errores-comunes-y-errores-

Capítulo 9: Establecer operaciones

Sintaxis

1. C = unión (A, B);
2. C = intersección (A, B);
3. C = setdiff (A, B);
4. a = miembro (A, x);

Parámetros

Parámetro	Detalles
A, B	Conjuntos, posiblemente matrices o vectores.
X	posible elemento de un conjunto

Examples

Conjunto de operaciones elementales

Es posible realizar operaciones de conjuntos elementales con Matlab. Supongamos que hemos dado dos vectores o matrices

```
A = randi([0 10],1,5);
B = randi([-1 9], 1,5);
```

y queremos encontrar todos los elementos que están en `A` y en `B` Para esto podemos usar

```
C = intersect(A,B);
```

`C` incluirá todos los números que son parte de `A` y parte de `B` Si también queremos encontrar la posición de estos elementos que llamamos

```
[C,pos] = intersect(A,B);
```

`pos` es la posición de estos elementos tal que `C == A(pos)` .

Otra operación básica es la unión de dos conjuntos.

```
D = union(A,B);
```

Herby contiene `D` todos los elementos de `A` y `B`

Tenga en cuenta que `A` y `B` se tratan como conjuntos, lo que significa que no importa la frecuencia con la que un elemento forme parte de `A` o `B` Para aclarar esto se puede verificar `D == union(D,C)` .

Si queremos obtener los datos que están en 'A' pero no en 'B' podemos usar la siguiente función

```
E = setdiff(A,B);
```

Queremos señalar nuevamente que se trata de conjuntos tales que la siguiente declaración contiene `D == union(E,B)` .

Supongamos que queremos comprobar si

```
x = randi([-10 10],1,1);
```

Es un elemento de `A` o `B` que podemos ejecutar el comando.

```
a = ismember(A,x);
b = ismember(B,x);
```

Si `a==1` entonces `x` es elemento de `A` `x` es ningún elemento es `a==0` . Lo mismo ocurre con `B` Si `a==1 && b==1` `x` también es un elemento de `C` Si `a == 1 || b == 1` `x` es elemento de `D` y si `a == 1 || b == 0` también es elemento de `E`

Lea Establecer operaciones en línea: https://riptutorial.com/es/matlab/topic/3242/establecer-operaciones

Capítulo 10: Funciones

Examples

Ejemplo de base

La siguiente secuencia de comandos de MATLAB muestra cómo definir y llamar a una función básica:

miFun.m :

```
function [out1] = myFun(arg0, arg1)
    out1 = arg0 + arg1;
end
```

terminal :

```
>> res = myFun(10, 20)

res =

    30
```

Salidas multiples

La siguiente secuencia de comandos de MATLAB muestra cómo devolver múltiples salidas en una sola función:

miFun.m :

```
function [out1, out2, out3] = myFun(arg0, arg1)
    out1 = arg0 + arg1;
    out2 = arg0 * arg1;
    out3 = arg0 - arg1;
end
```

terminal :

```
>> [res1, res2, res3] = myFun(10, 20)

res1 =

        30

res2 =

        200

res3 =
        -10
```

Sin embargo, MATLAB devolverá solo el primer valor cuando se asigne a una sola variable

```
    >> res = myFun(10, 20)

    res =

            30
```

El siguiente ejemplo muestra cómo obtener una salida específica

```
    >> [~, res] = myFun(10, 20)

    res =

            200
```

nargin

En el cuerpo de una función, `nargin` y `nargout` indican respectivamente el número real de entrada y salida suministrados en la llamada.

Por ejemplo, podemos controlar la ejecución de una función en función del número de entradas proporcionadas.

myVector.m :

```
function [res] = myVector(a, b, c)
    % Roughly emulates the colon operator

    switch nargin
        case 1
            res = [0:a];
        case 2
            res = [a:b];
        case 3
            res = [a:b:c];
        otherwise
            error('Wrong number of params');
    end
end
```

terminal:

```
>> myVector(10)

ans =

    0    1    2    3    4    5    6    7    8    9   10

>> myVector(10, 20)

ans =

   10   11   12   13   14   15   16   17   18   19   20
```

```
>> myVector(10, 2, 20)

ans =

   10   12   14   16   18   20
```

De manera similar, podemos controlar la ejecución de una función en función del número de parámetros de salida.

myIntegerDivision

```
function [qt, rm] = myIntegerDivision(a, b)
    qt = floor(a / b);

    if nargout == 2
        rm = rem(a, b);
    end
end
```

terminal :

```
>> q = myIntegerDivision(10, 7)

q = 1

>> [q, r] = myIntegerDivision(10, 7)

q = 1
r = 3
```

Lea Funciones en línea: https://riptutorial.com/es/matlab/topic/5659/funciones

Capítulo 11: Funciones de documentacion

Observaciones

- El texto de ayuda puede ubicarse antes o después de la línea de `function` , siempre que no haya código entre la línea de función y el inicio del texto de ayuda.
- El uso de mayúsculas en el nombre de la función solo muestra el nombre en negrita y no es obligatorio.
- Si una línea está precedida por `See also` , cualquier nombre en la línea que coincida con el nombre de una clase o función en la ruta de búsqueda se vinculará automáticamente a la documentación de esa clase / función.
    - Las funciones globales se pueden consultar aquí precediendo su nombre con una `\` . De lo contrario, los nombres primero intentarán resolver las funciones miembro.
- Se permiten los hipervínculos del formulario `<a href="matlab:web('url')">Name</a>` .

Examples

Documentación de función simple

```
function output = mymult(a, b)
% MYMULT Multiply two numbers.
%   output = MYMULT(a, b) multiplies a and b.
%
%   See also fft, foo, sin.
%
%   For more information, see <a href="matlab:web('https://google.com')">Google</a>.
    output = a * b;
end
```

`help mymult` luego proporciona:

> **mymult** Multiplica dos números.
>
> output = **mymult** (a, b) multiplica a y b.
>
> Véase también fft, foo, pecado.
>
> Para obtener más información, consulte Google .

`fft` y `sin` enlazan automáticamente a su respectivo texto de ayuda, y Google es un enlace a google.com . `foo` no enlazará con ninguna documentación en este caso, siempre que no haya una función / clase documentada con el nombre de `foo` en la ruta de búsqueda.

Documentación de la función local

En este ejemplo, se puede acceder a la documentación para la función local `baz` (definida en `foo.m`) mediante el enlace resultante en `help foo` , o directamente a través de `help foo>baz` .

```
function bar = foo
%This is documentation for FOO.
% See also foo>baz

% This wont be printed, because there is a line without % on it.
end

function baz
% This is documentation for BAZ.
end
```

Obtención de una firma de función.

A menudo es útil que MATLAB imprima la primera línea de una función, ya que generalmente contiene la firma de la función, incluidas las entradas y salidas:

```
dbtype <functionName> 1
```

Ejemplo:

```
>> dbtype fit 1

1  function [fitobj,goodness,output,warnstr,errstr,convmsg] =
fit(xdatain,ydatain,fittypeobj,varargin)
```

Documentar una función con un script de ejemplo

Para documentar una función, a menudo es útil tener un script de ejemplo que use su función. La función de publicación en Matlab se puede usar para generar un archivo de ayuda con imágenes, códigos, enlaces, etc. incrustados. La sintaxis para documentar su código se puede encontrar aquí .

La función Esta función utiliza un FFT "corregido" en Matlab.

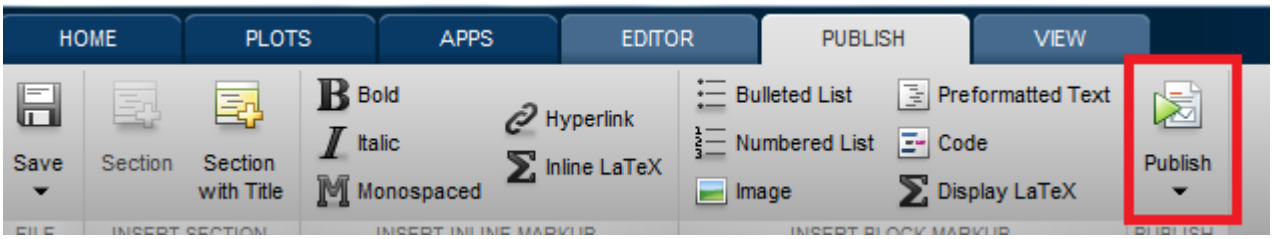
```
function out_sig = myfft(in_sig)

out_sig = fftshift(fft(ifftshift(in_sig)));

end
```

La secuencia de comandos de ejemplo Esta es una secuencia de comandos independiente que explica las entradas, salidas y da un ejemplo que explica por qué es necesaria la corrección. Gracias a Wu, Kan, el autor original de esta función.

```
%% myfft
% This function uses the "proper" fft in matlab. Note that the fft needs to
% be multiplied by dt to have physical significance.
% For a full description of why the FFT should be taken like this, refer
% to: Why_use_fftshift(fft(fftshift(x)))__in_Matlab.pdf included in the
% help folder of this code. Additional information can be found:
% <https://www.mathworks.com/matlabcentral/fileexchange/25473-why-use-fftshift-fft-fftshift-x-
---in-matlab-instead-of-fft-x-->
```

```
%
%% Inputs
% *in_sig* - 1D signal
%
%% Outputs
% *out_sig* - corrected FFT of *in_sig*
%
%% Examples
% Generate a signal with an analytical solution. The analytical solution is
% then compared to the fft then to myfft. This example is a modified
% version given by Wu, Kan given in the link aboce.
%%
% Set parameters
fs = 500;           %sampling frequency
dt = 1/fs;          %time step
T=1;                %total time window
t = -T/2:dt:T/2-dt; %time grids
df = 1/T;           %freq step
Fmax = 1/2/dt;      %freq window
f=-Fmax:df:Fmax-df; %freq grids, not used in our examples, could be used by plot(f, X)
%%
% Generate Gaussian curve
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);    %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                   %Create Gaussian Curve
%%
% Generate Analytical solution
Xan = A*sqrt(2*pi)*exp(-2*pi^2*f.^2*A^2); %X(f), real part of the analytical Fourier transform
of x(t)

%%
% Take FFT and corrected FFT then compare
Xfft = dt *fftshift(fft(x));    %FFT
Xfinal = dt * myfft(x);         %Corrected FFT
hold on
plot(f,Xan);
plot(f,real(Xfft));
plot(f,real(Xfinal),'ro');
title('Comparison of Corrected and Uncorrected FFT');
legend('Analytical Solution','Uncorrected FFT','Corrected FFT');
xlabel('Frequency'); ylabel('Amplitude');
DT = max(f) - min(f);
xlim([-DT/4,DT/4]);
```

La salida La opción de publicación se puede encontrar en la pestaña "Publicar", resaltada en la imagen Documentación de la función simple a continuación.



Matlab ejecutará el script y guardará las imágenes que se muestran, así como el texto generado por la línea de comandos. La salida se puede guardar en muchos tipos diferentes de formatos, incluidos HTML, Latex y PDF.

La salida de la secuencia de comandos de ejemplo proporcionada anteriormente se puede ver en

la imagen de abajo.

myfft

This function uses the "proper" fft in matlab. Note that the fft needs to be multiplied by dt to have physical significance. For a full description of why the FFT should be taken like this, refer to: Why_use_fftshift(fft(fftshift(x)))__in_Matlab.pdf included in the help folder of this code. Additional information can be found: https://www.mathworks.com/matlabcentral/fileexchange/25473-why-use-fftshift-fft-fftshift-x----in-matlab-instead-of-fft-x--

Contents

Inputs

in_sig - 1D signal

Outputs

out_sig - corrected FFT of **in_sig**

Examples

Generate a signal with an analytical solution. The analytical solution is then compared to the fft then to myfft. This example is a modified version given by Wu, Kan given in the link aboce.

Set parameters

```
fs = 500;            %sampling frequency
dt = 1/fs;           %time step
T=1;                 %total time window
t = -T/2:dt:T/2-dt; %time grids
df = 1/T;            %freq step
Fmax = 1/2/dt;       %freq window
f=-Fmax:df:Fmax-df; %freq grids, not used in our examples, could be used by plot(f, X)
```

Generate Gaussian curve

```
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);    %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                   %Create Gaussian Curve
```

Generate Analytical solution

```
Xan = A*sqrt(2*pi)*exp(-2*pi^2*f.^2*A^2); %X(f), real part of the analytical Fourier transform of x(t)
```

Take FFT and corrected FFT then compare

```
Xfft = dt *fftshift(fft(x));     %FFT
Xfinal = dt * myfft(x);          %Corrected FFT
hold on
plot(f,Xan);
plot(f,real(Xfft));
plot(f,real(Xfinal),'ro');
title('Comparison of Corrected and Uncorrected FFT');
legend('Analytical Solution','Uncorrected FFT','Corrected FFT');
xlabel('Frequency'); ylabel('Amplitude');
DT = max(f) - min(f);
xlim([-DT/4,DT/4]);
```

https://riptutorial.com/es/matlab/topic/1253/funciones-de-documentacion

Capítulo 12: Gráficos: Transformaciones 2D y 3D

Examples

Transformaciones 2D

En este ejemplo, vamos a tomar una línea en forma de sqaure trazada usando una `line` y realizar transformaciones en ella. Luego usaremos las mismas transformaciones, pero en un orden diferente y veremos cómo influye en los resultados.

Primero abrimos una figura y configuramos algunos parámetros iniciales (coordenadas de punto cuadrado y parámetros de transformación)

```
%Open figure and create axis
Figureh=figure('NumberTitle','off','Name','Transformation Example',...
    'Position',[200 200 700 700]); %bg is set to red so we know that we can only see the axes
Axesh=axes('XLim',[-8 8],'YLim',[-8,8]);

 %Initializing Variables
    square=[-0.5 -0.5;-0.5 0.5;0.5 0.5;0.5 -0.5]; %represented by its vertices
    Sx=0.5;
    Sy=2;
    Tx=2;
    Ty=2;
    teta=pi/4;
```

A continuación construimos las matrices de transformación (escala, rotación y traducción):

```
%Generate Transformation Matrix
S=makehgtform('scale',[Sx Sy 1]);
R=makehgtform('zrotate',teta);
T=makehgtform('translate',[Tx Ty 0]);
```

A continuación trazamos el suare azul:

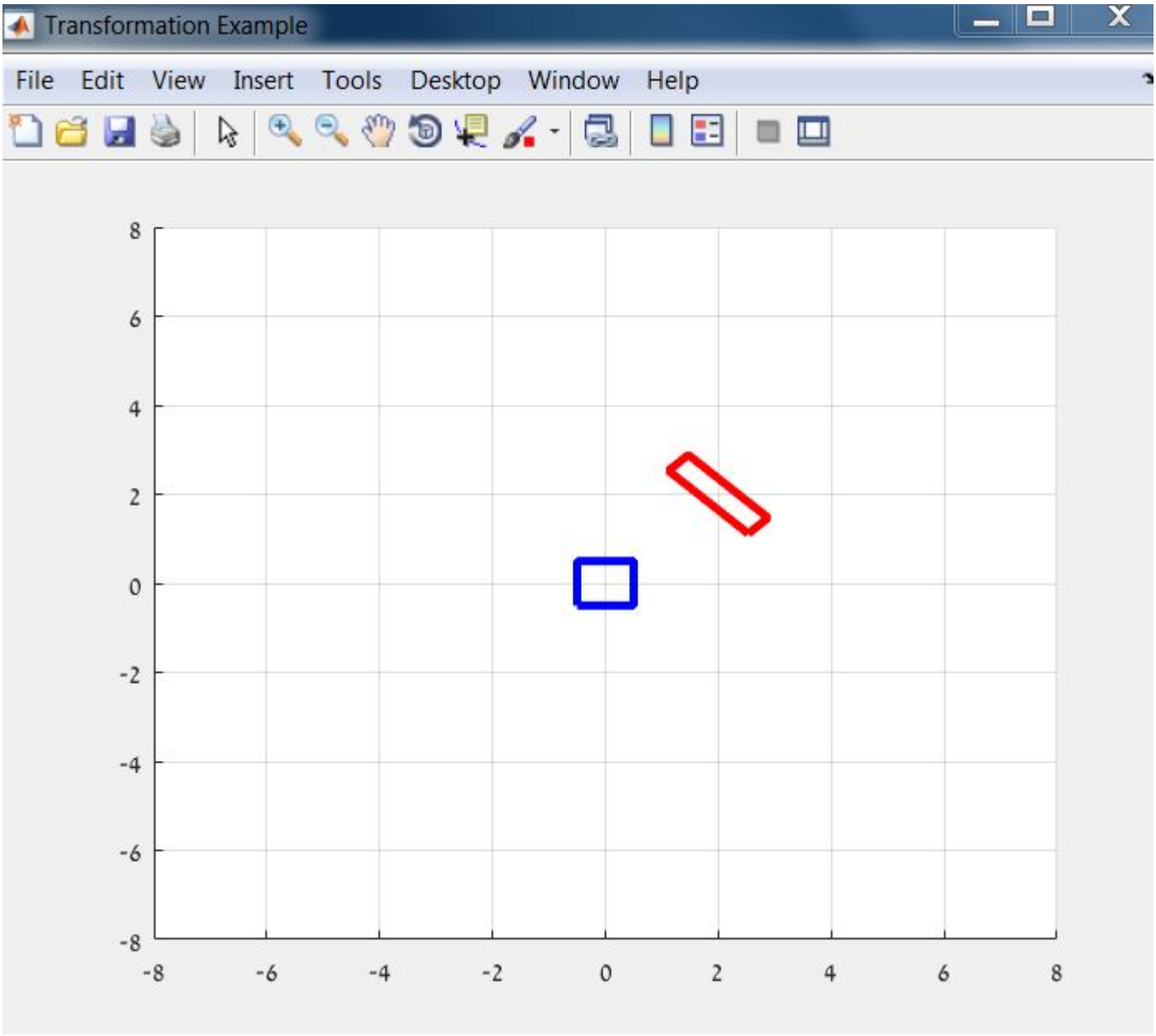
```
%% Plotting the original Blue Square
OriginalSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','b','LineWidth',3);

grid on;    % Applying grid on the figure
hold all;   % Holding all Following  graphs to current axes
```

A continuación, lo trazaremos de nuevo en un color diferente (rojo) y aplicaremos las transformaciones:

```
%% Plotting the Red Square
%Calculate rectangle vertices
HrectTRS=T*R*S;
RedSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','r','LineWidth',3);
```

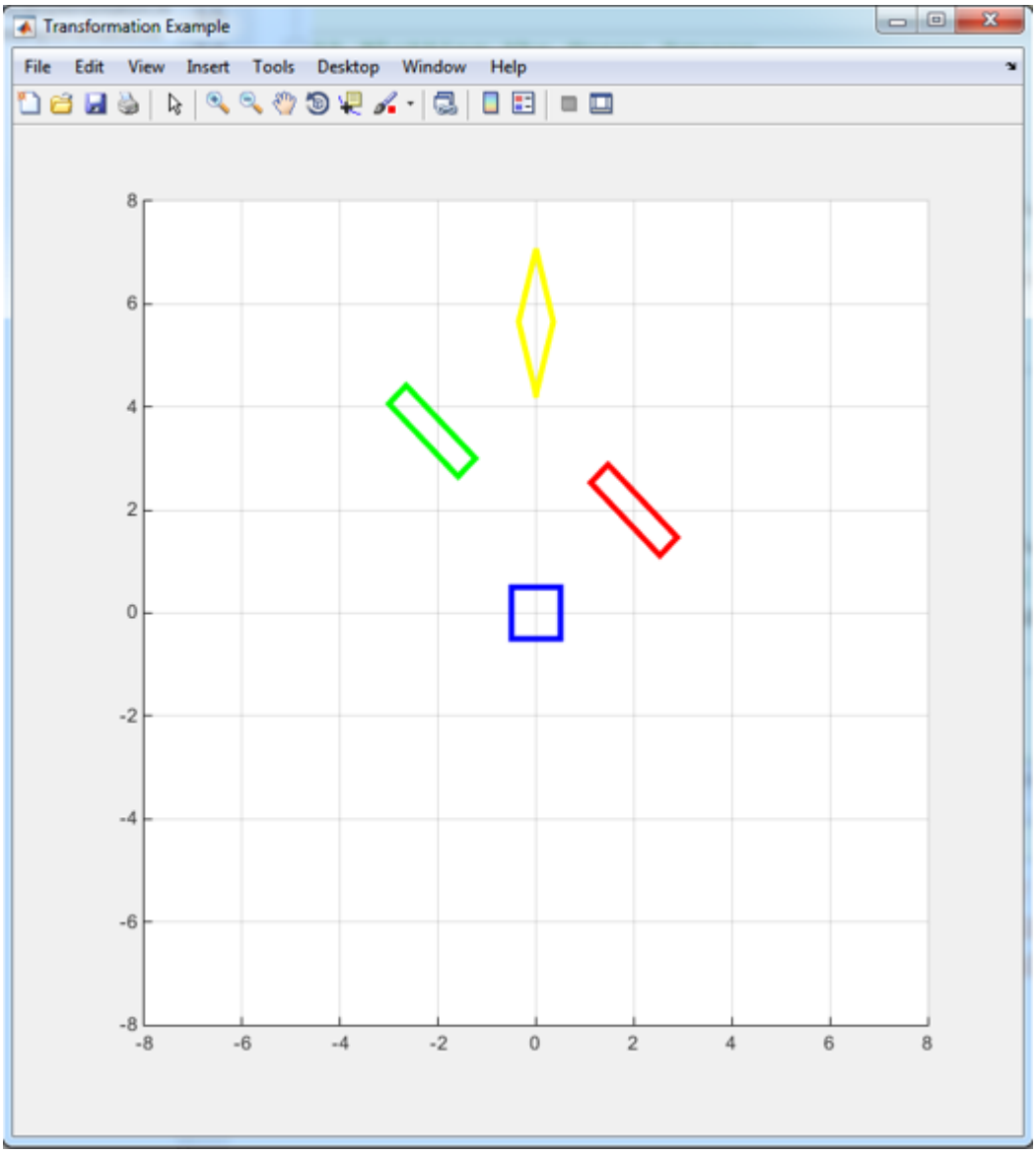
```
%transformation of the axes
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectTRS);
%seting the line to be a child of transformed axes
set(RedSQ,'Parent',AxesTransformation);
```

El resultado debería verse así:



Ahora veamos qué sucede cuando cambiamos el orden de transformación:

```
%% Plotting the Green Square
HrectRST=R*S*T;
GreenSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','g','LineWidth',3);
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectRST);
set(GreenSQ,'Parent',AxesTransformation);

%% Plotting the Yellow Square
HrectSRT=S*R*T;
YellowSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','y','LineWidth',3);
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectSRT);
set(YellowSQ,'Parent',AxesTransformation);
```



Lea Gráficos: Transformaciones 2D y 3D en línea:
https://riptutorial.com/es/matlab/topic/7418/graficos--transformaciones-2d-y-3d

Capítulo 13: Gráficos: trazos de líneas 2D

Sintaxis

- parcela (Y)
- parcela (Y, LineSpec)
- parcela (X, Y)
- plot (X, Y, LineSpec)
- gráfico (X1, Y1, X2, Y2, ..., Xn, Yn)
- gráfico (X1, Y1, LineSpec1, X2, Y2, LineSpec2, ..., Xn, Yn, LineSpecn)
- parcela (___, nombre, valor)
- h = parcela (___)

Parámetros

Parámetro	Detalles
X	valores de x
Y	valores de y
LineSpec	Estilo de línea, símbolo de marcador y color, especificados como una cadena
Nombre, Valor	Pares opcionales de argumentos nombre-valor para personalizar las propiedades de línea
h	manejar para alinear objeto gráfico

Observaciones

http://www.mathworks.com/help/matlab/ref/plot.html

Examples

Múltiples líneas en una sola parcela.

En este ejemplo, vamos a trazar varias líneas en un solo eje. Además, elegimos una apariencia

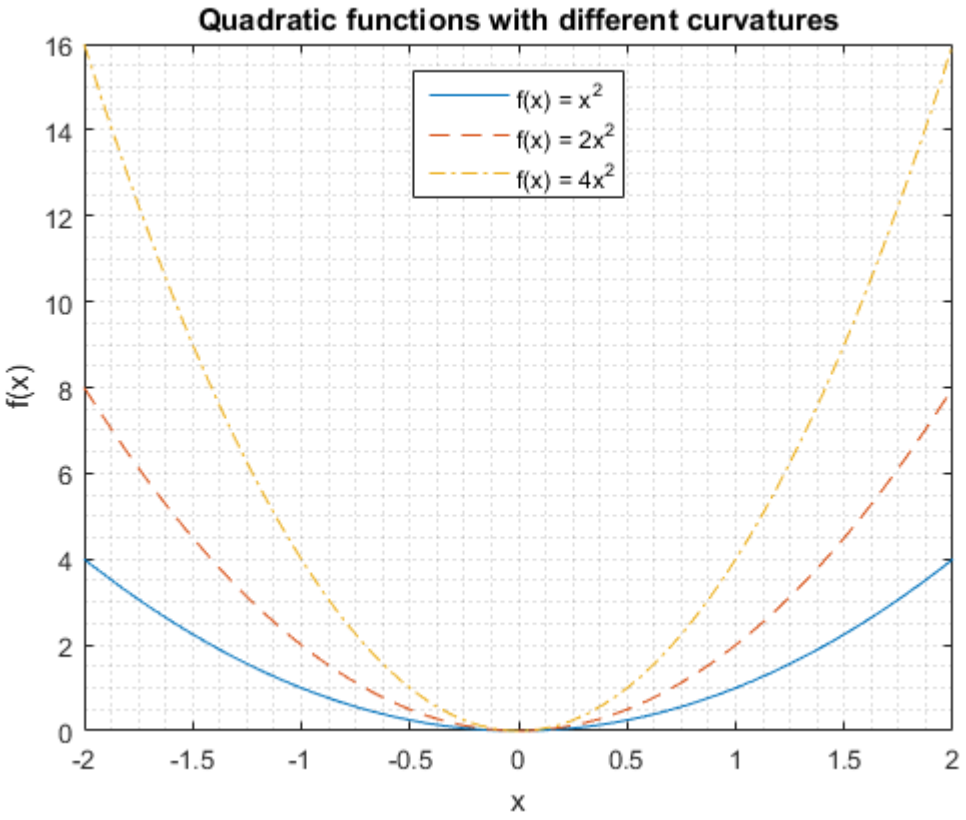
diferente para las líneas y creamos una leyenda.

```
% create sample data
x = linspace(-2,2,100);                 % 100 linearly spaced points from -2 to 2
y1 = x.^2;
y2 = 2*x.^2;
y3 = 4*x.^2;

% create plot
figure;                                 % open new figure
plot(x,y1, x,y2,'--', x,y3,'-.');       % plot lines
grid minor;                             % add minor grid
title('Quadratic functions with different curvatures');
xlabel('x');
ylabel('f(x)');
legend('f(x) = x^2', 'f(x) = 2x^2', 'f(x) = 4x^2', 'Location','North');
```

En el ejemplo anterior, trazamos las líneas con un solo `plot` comando. Una alternativa es usar comandos separados para cada línea. Necesitamos *mantener* el contenido de una figura con el último `hold on` antes de agregar la segunda línea. De lo contrario, las líneas trazadas previamente desaparecerán de la figura. Para crear la misma trama que la anterior, podemos usar los siguientes comandos:

```
figure; hold on;
plot(x,y1);
plot(x,y2,'--');
plot(x,y3,'-.');
```

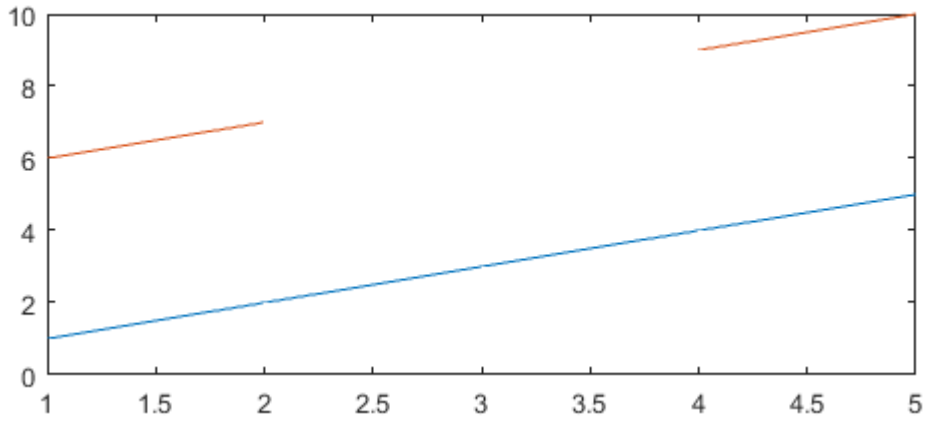
La figura resultante se ve así en ambos casos:



Línea de división con NaNs

Intercala tus valores `y` o `x` con `NaNs`

```
x = [1:5; 6:10]';

x(3,2) = NaN
x =
     1     6
     2     7
     3   NaN
     4     9
     5    10

plot(x)
```

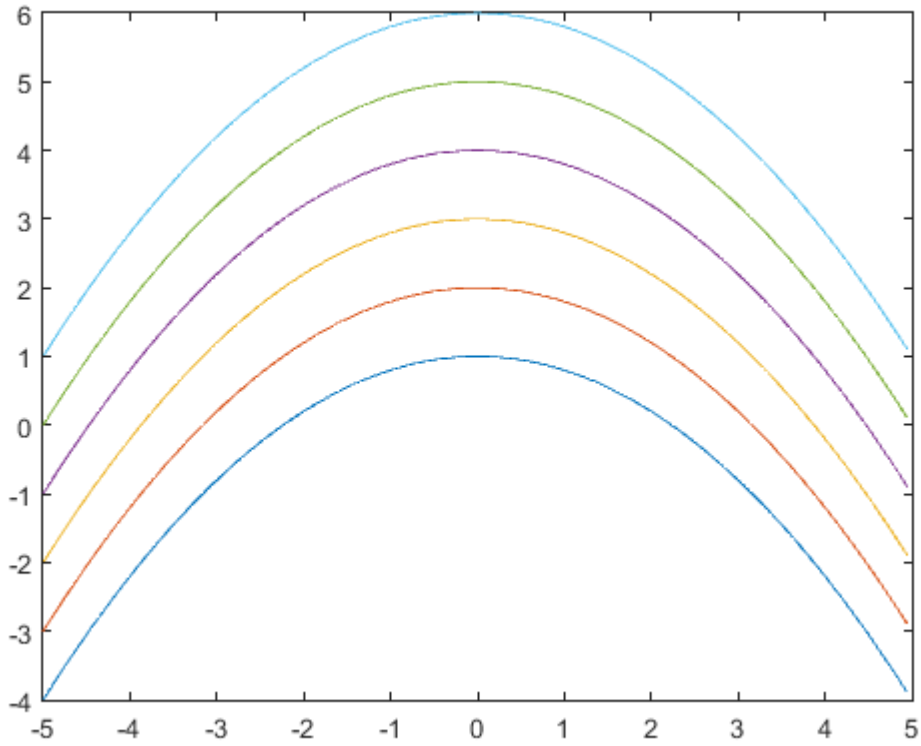


Color personalizado y órdenes de estilo de línea

En MATLAB, podemos establecer nuevos pedidos personalizados *predeterminados* , como un orden de color y un orden de estilo de línea. Eso significa que los nuevos pedidos se aplicarán a cualquier figura que se cree después de que se hayan aplicado estas configuraciones. La nueva configuración permanece hasta que se cierra la sesión de MATLAB o se realiza una nueva configuración.

Color predeterminado y orden de estilo de línea

Por defecto, MATLAB usa un par de colores diferentes y solo un estilo de línea sólida. Por lo tanto, si se llama a `plot` para dibujar múltiples líneas, MATLAB alterna a través de un orden de colores para dibujar líneas en diferentes colores.



Podemos obtener el orden de color predeterminado llamando a `get` con un identificador global `0` seguido de este atributo `DefaultAxesColorOrder` :

```
>> get(0, 'DefaultAxesColorOrder')
ans =
         0    0.4470    0.7410
    0.8500    0.3250    0.0980
    0.9290    0.6940    0.1250
    0.4940    0.1840    0.5560
    0.4660    0.6740    0.1880
    0.3010    0.7450    0.9330
    0.6350    0.0780    0.1840
```

Color personalizado y orden de estilo de línea

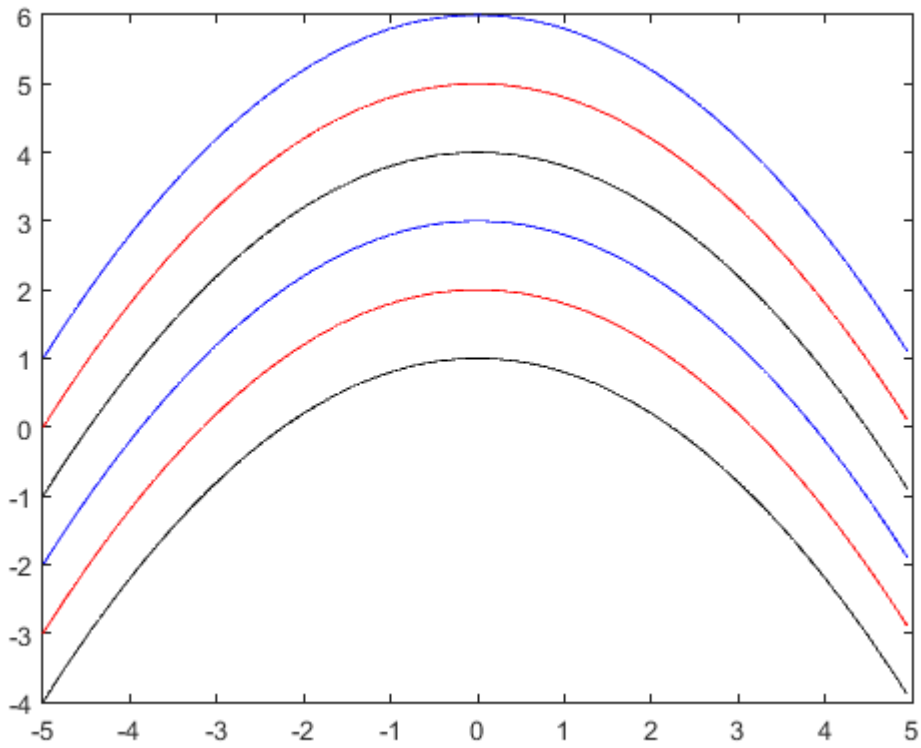
Una vez que hayamos decidido establecer un orden de color personalizado y un orden de estilo de línea, MATLAB debe alternar entre ambos. El primer cambio que aplica MATLAB es un color. Cuando se agotan todos los colores, MATLAB aplica el siguiente estilo de línea de un orden de estilo de línea definido y establece un índice de color en 1. Eso significa que MATLAB comenzará a alternar a través de todos los colores nuevamente, pero utilizando el siguiente estilo de línea en su orden. Cuando se agotan todos los estilos de línea y los colores, obviamente MATLAB comienza a cambiar desde el principio utilizando el primer color y el primer estilo de línea.

Para este ejemplo, he definido un vector de entrada y una función anónima para facilitar el trazado de las figuras:

```
F = @(a,x) bsxfun(@plus, -0.2*x(:).^2, a);
x = (-5:5/100:5-5/100)';
```

Para establecer un nuevo color o un nuevo orden de estilo de línea, llamamos a la función `set` con un identificador global `0` seguido de un atributo `DefaultAxesXXXXXXX` ; `XXXXXXX` puede ser `ColorOrder` o `LineStyleOrder` . El siguiente comando establece un nuevo orden de color en negro, rojo y azul, respectivamente:

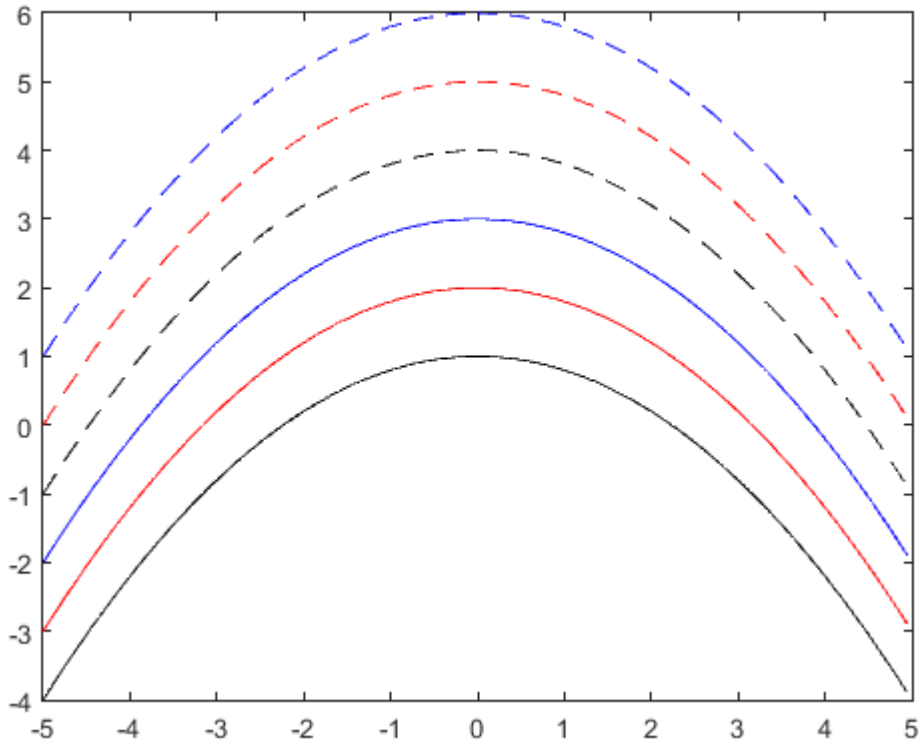
```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
plot(x, F([1 2 3 4 5 6],x));
```



Como puede ver, MATLAB se alterna solo a través de los colores porque el orden del estilo de línea se establece en una línea sólida de forma predeterminada. Cuando se agota un conjunto de colores, MATLAB comienza desde el primer color en el orden de los colores.

Los siguientes comandos establecen órdenes de color y estilo de línea:

```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
set(0, 'DefaultAxesLineStyleOrder', {'-' '--'});
plot(x, F([1 2 3 4 5 6],x));
```



Ahora, MATLAB se alterna a través de diferentes colores y diferentes estilos de línea utilizando el color como atributo más frecuente.

Lea Gráficos: trazos de líneas 2D en línea: https://riptutorial.com/es/matlab/topic/426/graficos--trazos-de-lineas-2d

Capítulo 14: Inicializando matrices o matrices

Introducción

Matlab tiene tres funciones importantes para crear matrices y establecer sus elementos en ceros, unos o la matriz de identidad. (La matriz de identidad tiene unos en la diagonal principal y ceros en otros lugares).

Sintaxis

- Z = ceros (sz, datatype, arraytype)
- X = unos (sz, tipo de datos)
- I = ojo (sz, tipo de datos)

Parámetros

Parámetro	Detalles
sz	n (para una matriz nxn)
sz	n, m (para una matriz nxm)
sz	m, n, ..., k (para una matriz m-by-n-by -...- by-k)
tipo de datos	'double' (predeterminado), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64' o 'uint64'
arraytype	'repartido'
arraytype	'codistribuido'
arraytype	'gpuArray'

Observaciones

Estas funciones crearán una matriz de dobles, por defecto.

Examples

Creando una matriz de 0s.

```
z1 = zeros(5); % Create a 5-by-5 matrix of zeroes
z2 = zeros(2,3); % Create a 2-by-3 matrix
```

Creando una matriz de 1s.

```
o1 = ones(5); % Create a 5-by-5 matrix of ones
o2 = ones(1,3); % Create a 1-by-3 matrix / vector of size 3
```

Creando una matriz de identidad.

```
i1 = eye(3); % Create a 3-by-3 identity matrix
i2 = eye(5,6); % Create a 5-by-6 identity matrix
```

Lea Inicializando matrices o matrices en línea:
https://riptutorial.com/es/matlab/topic/8049/inicializando-matrices-o-matrices

Capítulo 15: Integración

Examples

Integral, integral2, integral3

1 dimensional

Integrar una función unidimensional.

```
f = @(x) sin(x).^3 + 1;
```

dentro del rango

```
xmin = 2;
xmax = 8;
```

uno puede llamar a la función

```
q = integral(f,xmin,xmax);
```

También es posible establecer límites para errores relativos y absolutos.

```
q = integral(f,xmin,xmax, 'RelTol',10e-6, 'AbsTol',10-4);
```

2 dimensiones

Si uno quiere integrar una función bidimensional.

```
f = @(x,y) sin(x).^y ;
```

dentro del rango

```
xmin = 2;
xmax = 8;
ymin = 1;
ymax = 4;
```

uno llama a la función

```
q = integral2(f,xmin,xmax,ymin,ymax);
```

Como en el otro caso es posible limitar las tolerancias.

```
q = integral2(f,xmin,xmax,ymin,ymax, 'RelTol',10e-6, 'AbsTol',10-4);
```

3 dimensiones

Integrando una función tridimensional.

```
f = @(x,y,z) sin(x).^y - cos(z) ;
```

dentro del rango

```
xmin = 2;
xmax = 8;
ymin = 1;
ymax = 4;
zmin = 6;
zmax = 13;
```

se realiza llamando

```
q = integral3(f,xmin,xmax,ymin,ymax, zmin, zmax);
```

De nuevo es posible limitar las tolerancias.

```
q = integral3(f,xmin,xmax,ymin,ymax, zmin, zmax, 'RelTol',10e-6, 'AbsTol',10-4);
```

Lea Integración en línea: https://riptutorial.com/es/matlab/topic/7022/integracion

Capítulo 16: Interfaces de usuario MATLAB

Examples

Transferencia de datos alrededor de la interfaz de usuario

La mayoría de las interfaces de usuario avanzadas requieren que el usuario pueda pasar información entre las diversas funciones que conforman una interfaz de usuario. MATLAB tiene varios métodos diferentes para hacerlo.

`guidata`

Propia de MATLAB entorno de desarrollo de interfaz gráfica de usuario (GUIDE) prefiere utilizar una `struct` con nombre `handles` para pasar datos entre las devoluciones de llamada. Esta `struct` contiene todos los controladores de gráficos para los diversos componentes de la interfaz de usuario, así como los datos especificados por el usuario. Si no está utilizando una devolución de llamada creada por GUIDE que pasa automáticamente a los `handles` , puede recuperar el valor actual utilizando `guidata`

```
% hObject is a graphics handle to any UI component in your GUI
handles = guidata(hObject);
```

Si desea modificar un valor almacenado en esta estructura de datos, puede modificarlo pero luego debe almacenarlo nuevamente dentro del `hObject` para que los cambios sean visibles por otras devoluciones de llamada. Puede almacenarlo especificando un segundo argumento de entrada a `guidata` .

```
% Update the value
handles.myValue = 2;

% Save changes
guidata(hObject, handles)
```

El valor de `hObject` no importa siempre que sea un componente UI *dentro de la misma* `figure` porque, en última instancia, los datos se almacenan dentro de la figura que contiene `hObject` .

Mejor para:

- Almacenar la estructura de los `handles` , en la que puede almacenar todos los manejadores de sus componentes GUI.
- Almacenar "pequeñas" otras variables a las que debe acceder la mayoría de las devoluciones de llamada.

No recomendado para :

- Almacenar variables grandes a las que no deben acceder todas las devoluciones de

llamada y subfunciones (use `setappdata` / `getappdata` para estas).

`setappdata` / `getappdata`

De manera similar al enfoque de `guidata` , puede usar `setappdata` y `getappdata` para almacenar y recuperar valores desde un controlador de gráficos. La ventaja de usar estos métodos es que puede recuperar *solo el valor que desea en* lugar de una `struct` completa que contenga *todos los* datos almacenados. Es similar a un almacén de clave / valor.

Para almacenar datos dentro de un objeto gráfico

```
% Create some data you would like to store
myvalue = 2

% Store it using the key 'mykey'
setappdata(hObject, 'mykey', myvalue)
```

Y para recuperar ese mismo valor desde una devolución de llamada diferente

```
value = getappdata(hObject, 'mykey');
```

Nota: Si no se almacenó ningún valor antes de llamar a `getappdata` , devolverá una matriz vacía (`[]`).

Al igual que con `guidata` , los datos se almacenan en la figura que contiene `hObject` .

Mejor para:

- Almacenar variables grandes a las que no deben acceder todas las devoluciones de llamada y subfunciones.

`UserData`

Cada controlador de gráficos tiene una propiedad especial, `UserData` que puede contener cualquier información que desee. Podría contener una matriz de celdas, una `struct` o incluso un escalar. Puede aprovechar esta propiedad y almacenar cualquier dato que desee asociar con un controlador de gráficos dado en este campo. Puede guardar y recuperar el valor utilizando los métodos estándar de `get` / `set` para objetos gráficos o notación de puntos si está utilizando R2014b o más reciente.

```
% Create some data to store
mydata = {1, 2, 3};

% Store it within the UserData property
set(hObject, 'UserData', mydata)

% Of if you're using R2014b or newer:
```

```
% hObject.UserData = mydata;
```

Luego, desde dentro de otra devolución de llamada, puede recuperar estos datos:

```
their_data = get(hObject, 'UserData');

% Or if you're using R2014b or newer:
% their_data = hObject.UserData;
```

Mejor para:

- Almacenamiento de variables con un alcance limitado (variables que probablemente solo serán utilizadas por el objeto en el que están almacenadas u objetos que tengan una relación directa con él).

Funciones anidadas

En MATLAB, una función anidada puede leer y modificar cualquier variable definida en la función principal. De esta manera, si especifica que una devolución de llamada sea una función anidada, puede recuperar y modificar cualquier dato almacenado en la función principal.

```
function mygui()
    hButton = uicontrol('String', 'Click Me', 'Callback', @callback);

    % Create a counter to keep track of the number of times the button is clicked
    nClicks = 0;

    % Callback function is nested and can therefore read and modify nClicks
    function callback(source, event)
        % Increment the number of clicks
        nClicks = nClicks + 1;

        % Print the number of clicks so far
        fprintf('Number of clicks: %d\n', nClicks);
    end
end
```

Mejor para:

- GUIs pequeñas y simples. (para la creación rápida de prototipos, para no tener que implementar los `guidata` y / o `set/getappdata`).

No recomendado para :

- GUIs medianas, grandes o complejas.
- GUI creado con `GUIDE` .

Argumentos de entrada explícitos

Si necesita enviar datos a una función de devolución de llamada y no necesita modificar los datos dentro de la devolución de llamada, siempre puede considerar pasar los datos a la devolución de llamada utilizando una definición de devolución de llamada cuidadosamente diseñada.

Podría usar una función anónima que agregue entradas

```
% Create some data to send to mycallback
data = [1, 2, 3];

% Pass data as a third input to mycallback
set(hObject, 'Callback', @(source, event)mycallback(source, event, data))
```

O puede usar la sintaxis de la matriz de celdas para especificar una devolución de llamada, especificando de nuevo entradas adicionales.

```
set(hObject, 'Callback', {@mycallback, data})
```

Mejor para:

- Cuando la devolución de llamada necesita `data` para realizar algunas operaciones pero la variable de `data` no necesita modificarse y guardarse en un nuevo estado.

Hacer un botón en su interfaz de usuario que detiene la ejecución de devolución de llamada

A veces nos gustaría pausar la ejecución del código para inspeccionar el estado de la aplicación (ver Depuración). Al ejecutar el código a través del editor MATLAB, esto se puede hacer usando el botón "Pausa" en la interfaz de usuario o presionando `Ctrl` + `c` (en Windows). Sin embargo, cuando se inició un cálculo desde una GUI (a través de la devolución de llamada de algún `uicontrol`), este método ya no funciona, y la devolución de llamada debe *interrumpirse a* través de otra devolución de llamada. A continuación se muestra una demostración de este principio:

```
function interruptibleUI
dbclear in interruptibleUI % reset breakpoints in this file
figure('Position',[400,500,329,160]);

uicontrol('Style', 'pushbutton',...
          'String', 'Compute',...
          'Position', [24 55 131 63],...
          'Callback', @longComputation,...
          'Interruptible','on'); % 'on' by default anyway

uicontrol('Style', 'pushbutton',...
          'String', 'Pause #1',...
          'Position', [180 87 131 63],...
          'Callback', @interrupt1);
```

```
uicontrol('Style', 'pushbutton',...
          'String', 'Pause #2',...
          'Position', [180 12 131 63],...
          'Callback', @interrupt2);

end

function longComputation(src,event)
  superSecretVar = rand(1);
  pause(15);
  print('done!'); % we'll use this to determine if code kept running "in the background".
end

function interrupt1(src,event) % depending on where you want to stop
  dbstop in interruptibleUI at 27 % will stop after print('done!');
  dbstop in interruptibleUI at 32 % will stop after **this** line.
end

function interrupt2(src,event) % method 2
  keyboard;
  dbup; % this will need to be executed manually once the code stops on the previous line.
end
```

Para asegurarse de que comprende este ejemplo, haga lo siguiente:

1. Pegue el código anterior en un nuevo archivo llamado y guárdelo como `interruptibleUI.m` , de modo que el código comience en la **primera línea** del archivo (esto es importante para que funcione el primer método).
2. Ejecutar el script.
3. Haga clic en `Calcular` y poco después haga clic en `Pausa # 1` o en `Pausa # 2` .
4. Asegúrese de que puede encontrar el valor de `superSecretVar` .

Pasar datos utilizando la estructura de "manejadores"

Este es un ejemplo de una GUI básica con dos botones que cambian un valor almacenado en la estructura de los `handles` la GUI.

```
function gui_passing_data()
    % A basic GUI with two buttons to show a simple use of the 'handles'
    % structure in GUI building

    %  Create a new figure.
    f = figure();

    % Retrieve the handles structure
    handles = guidata(f);

    % Store the figure handle
    handles.figure = f;

    % Create an edit box and two buttons (plus and minus),
    % and store their handles for future use
    handles.hedit  = uicontrol('Style','edit','Position',[10,200,60,20] , 'Enable',
'Inactive');

    handles.hbutton_plus  = uicontrol('Style','pushbutton','String','+',...
```

```
                'Position',[80,200,60,20] , 'Callback' , @ButtonPress);

    handles.hbutton_minus  = uicontrol('Style','pushbutton','String','-',...
                'Position',[150,200,60,20] , 'Callback' , @ButtonPress);

    % Define an initial value, store it in the handles structure and show
    % it in the Edit box
    handles.value = 1;
    set(handles.hedit , 'String' , num2str(handles.value))

    % Store handles
    guidata(f, handles);



function  ButtonPress(hObject, eventdata)
    % A button was pressed
    % Retrieve the handles
    handles = guidata(hObject);

    % Determine which button was pressed; hObject is the calling object
    switch(get(hObject , 'String'))
        case '+'
            % Add 1 to the value
            handles.value = handles.value + 1;
            set(handles.hedit , 'String', num2str(handles.value))
        case '-'
            % Substract 1 from the value
            handles.value = handles.value - 1;
    end

    % Display the new value
    set(handles.hedit , 'String', num2str(handles.value))

    % Store handles
    guidata(hObject, handles);
```

Para probar el ejemplo, guárdelo en un archivo llamado `gui_passing_data.m` y `gui_passing_data.m` con F5. Tenga en cuenta que en un caso tan simple, ni siquiera necesitaría almacenar el valor en la estructura de los manejadores porque podría acceder directamente desde la propiedad `String` del cuadro de edición.

Problemas de rendimiento al pasar datos por la interfaz de usuario

Dos técnicas principales permiten pasar datos entre funciones GUI y devoluciones de llamada: setappdata / getappdata y guidata (lea más sobre esto). Lo primero debería usarse para variables más grandes, ya que es más eficiente en el tiempo. El siguiente ejemplo prueba la eficiencia de los dos métodos.

Se crea una GUI con un botón simple y se almacena una gran variable (10000x10000 doble) con guidata y con setappdata. El botón vuelve a cargar y almacena la variable utilizando los dos métodos mientras cronometra su ejecución. El tiempo de ejecución y la mejora del porcentaje con setappdata se muestran en la ventana de comandos.

```
function gui_passing_data_performance()
```

```
    % A basic GUI with a button to show performance difference between
    % guidata and setappdata

    %  Create a new figure.
    f = figure('Units' , 'normalized');

    % Retrieve the handles structure
    handles = guidata(f);

    % Store the figure handle
    handles.figure = f;

    handles.hbutton =
uicontrol('Style','pushbutton','String','Calculate','units','normalized',...
               'Position',[0.4 , 0.45 , 0.2 , 0.1] , 'Callback' , @ButtonPress);

    % Create an uninteresting large array
    data = zeros(10000);

    % Store it in appdata
    setappdata(handles.figure , 'data' , data);

    % Store it in handles
    handles.data = data;

    % Save handles
    guidata(f, handles);



function  ButtonPress(hObject, eventdata)

    % Calculate the time difference when using guidata and appdata
    t_handles = timeit(@use_handles);
    t_appdata = timeit(@use_appdata);

    % Absolute and percentage difference
    t_diff = t_handles - t_appdata;
    t_perc = round(t_diff / t_handles * 100);

    disp(['Difference: ' num2str(t_diff) ' ms / ' num2str(t_perc) ' %'])




function  use_appdata()

    % Retrieve the data from appdata
    data = getappdata(gcf , 'data');

    % Do something with data %

    % Store the value again
    setappdata(gcf , 'data' , data);


function use_handles()

    % Retrieve the data from handles
    handles = guidata(gcf);
    data = handles.data;
```

```
    % Do something with data %

    % Store it back in the handles
    handles.data = data;
    guidata(gcf, handles);
```

En mi Xeon W3530@2.80 GHz obtengo la `Difference: 0.00018957 ms / 73 %` , por lo tanto, al usar getappdata / setappdata obtengo una mejora en el rendimiento del 73%. Tenga en cuenta que el resultado no cambia si se usa una variable doble 10x10, sin embargo, el resultado cambiará si los `handles` contienen muchos campos con datos grandes.

Lea Interfaces de usuario MATLAB en línea:
https://riptutorial.com/es/matlab/topic/2883/interfaces-de-usuario-matlab

Capítulo 17: Interpolación con MATLAB

Sintaxis

1. zy = interp1 (x, y);
2. zy = interp1 (x, y, 'método');
3. zy = interp1 (x, y, 'método', 'extrapolación');
4. zy = interp1 (x, y, zx);
5. zy = interp1 (x, y, zx, 'método');
6. zy = interp1 (x, y, zx, 'método', 'extrapolación');

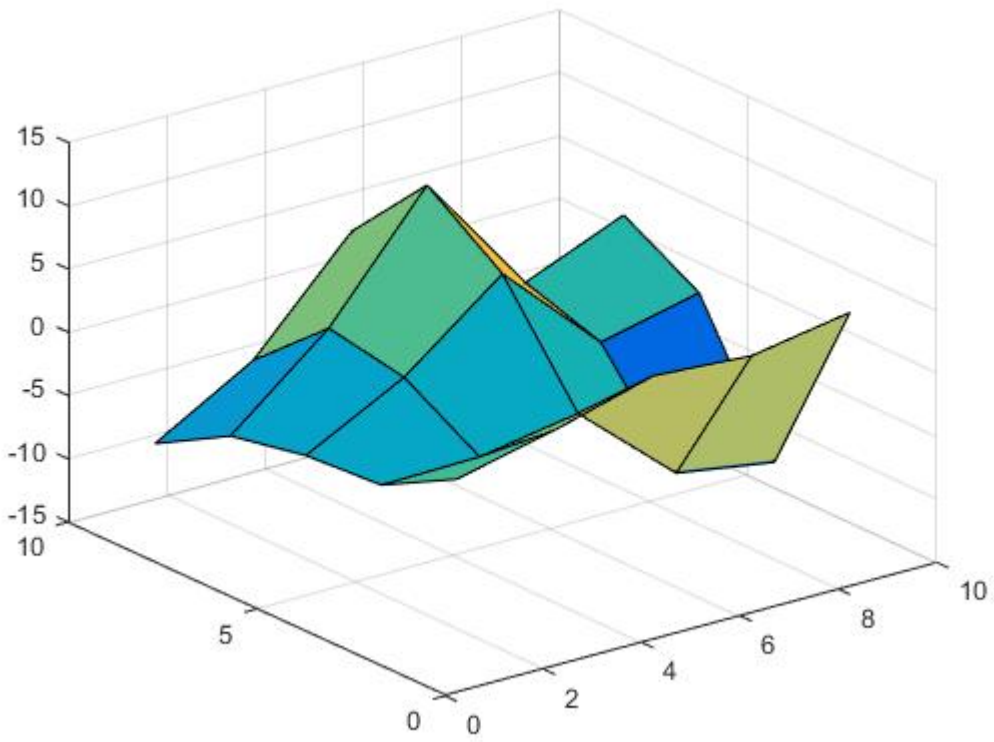
Examples

Interpolación a trozos 2 dimensiones.

Inicializamos los datos:

```
[X,Y] = meshgrid(1:2:10);
Z = X.*cos(Y) - Y.*sin(X);
```

La superficie se parece a la siguiente.

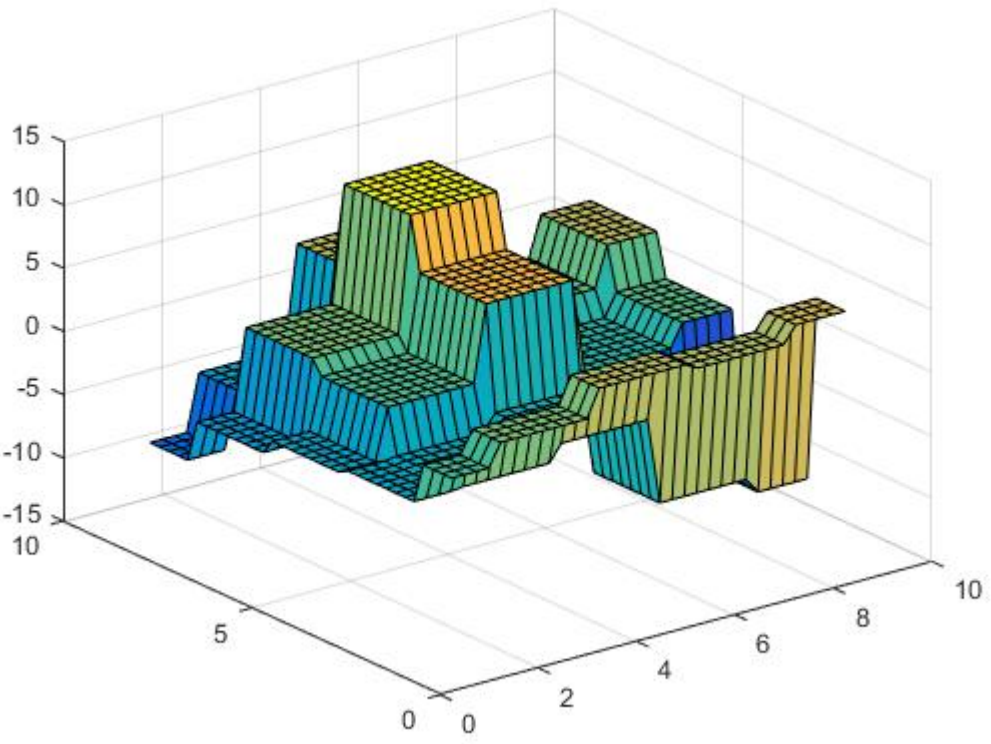


Ahora establecemos los puntos donde queremos interpolar:

```
[Vx,Vy] = meshgrid(1:0.25:10);
```

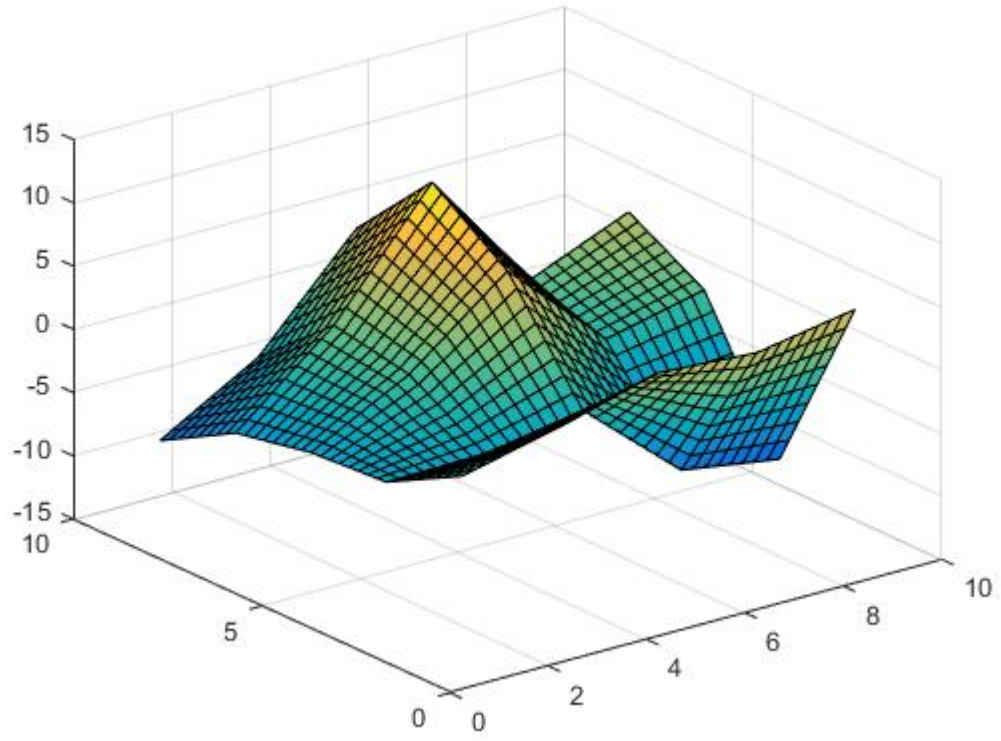
Ahora podemos realizar la interpolación más cercana,

```
Vz = interp2(X,Y,Z,Vx,Vy,'nearest');
```



Interpolación linear,

```
Vz = interp2(X,Y,Z,Vx,Vy,'linear');
```

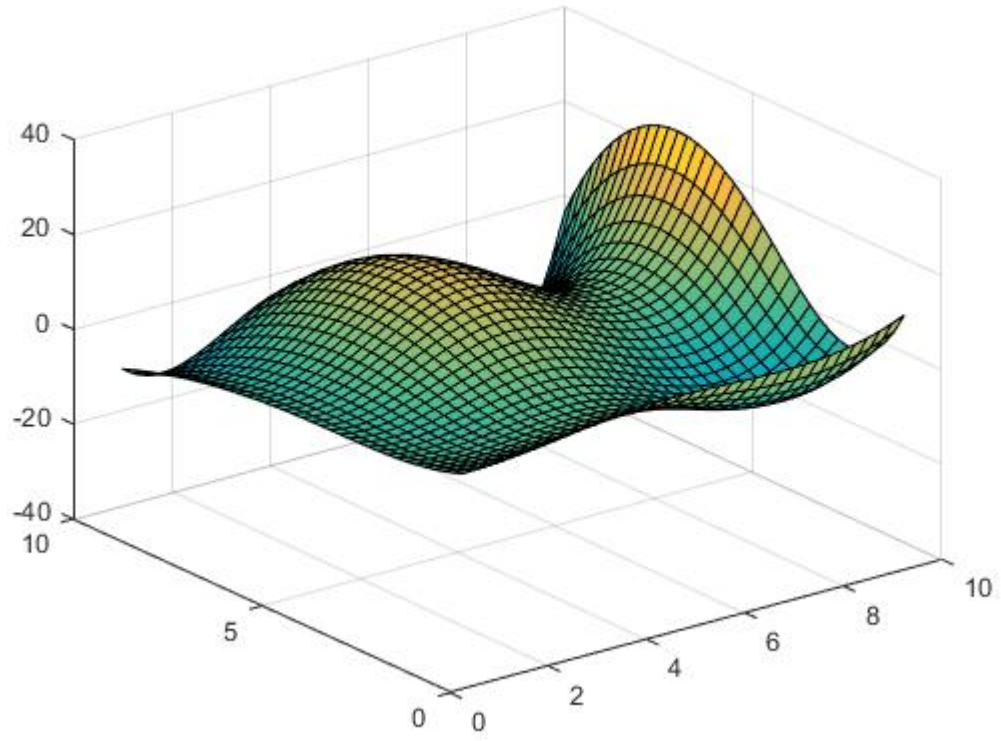


interpolación cúbica

```
Vz = interp2(X,Y,Z,Vx,Vy,'cubic');
```

o interpolación spline:

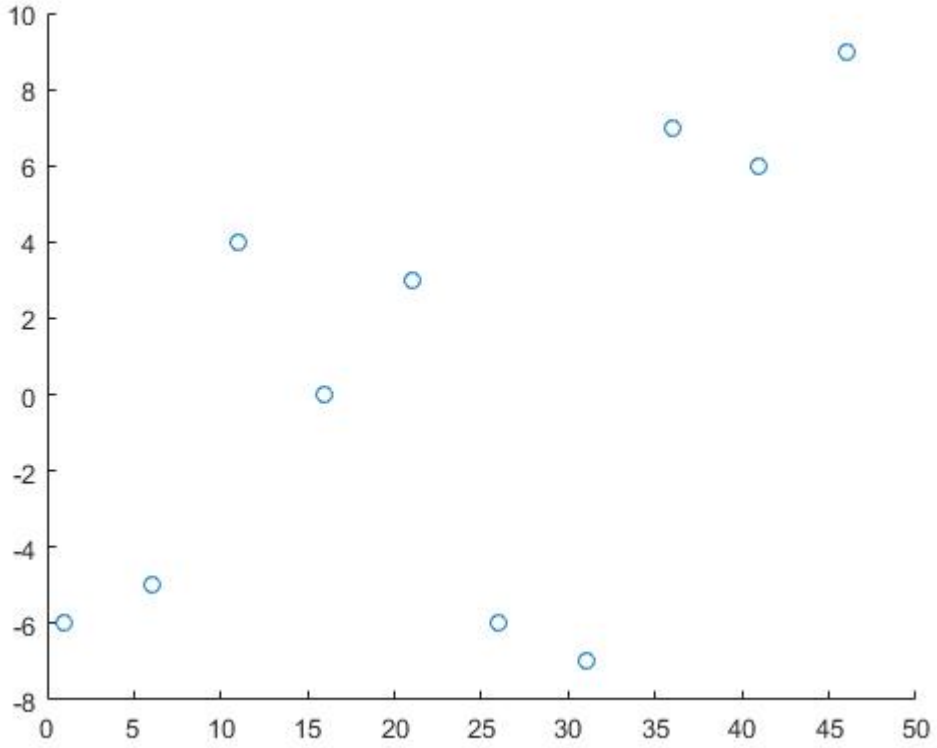
```
Vz = interp2(X,Y,Z,Vx,Vy,'spline');
```



Interpolación por partes 1 dimensional

Utilizaremos los siguientes datos:

```
x = 1:5:50;
y = randi([-10 10],1,10);
```

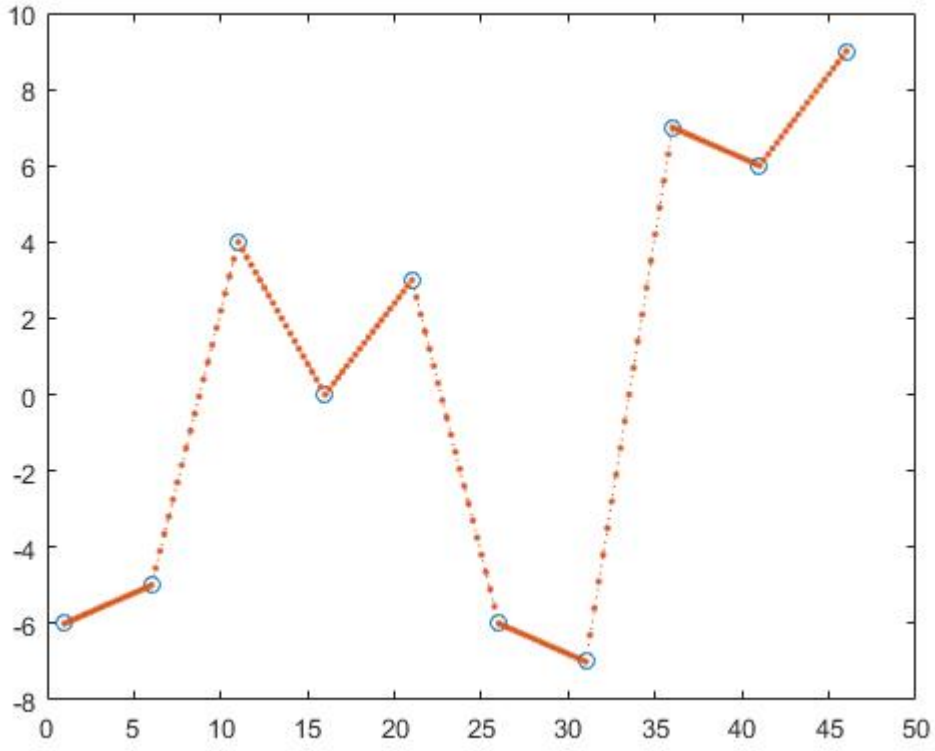


De este modo, `x` e `y` son las coordenadas de los puntos de datos y `z` son los puntos sobre los que necesitamos información.

```
z = 0:0.25:50;
```

Una forma de encontrar los valores de y de z es la interpolación lineal por partes.

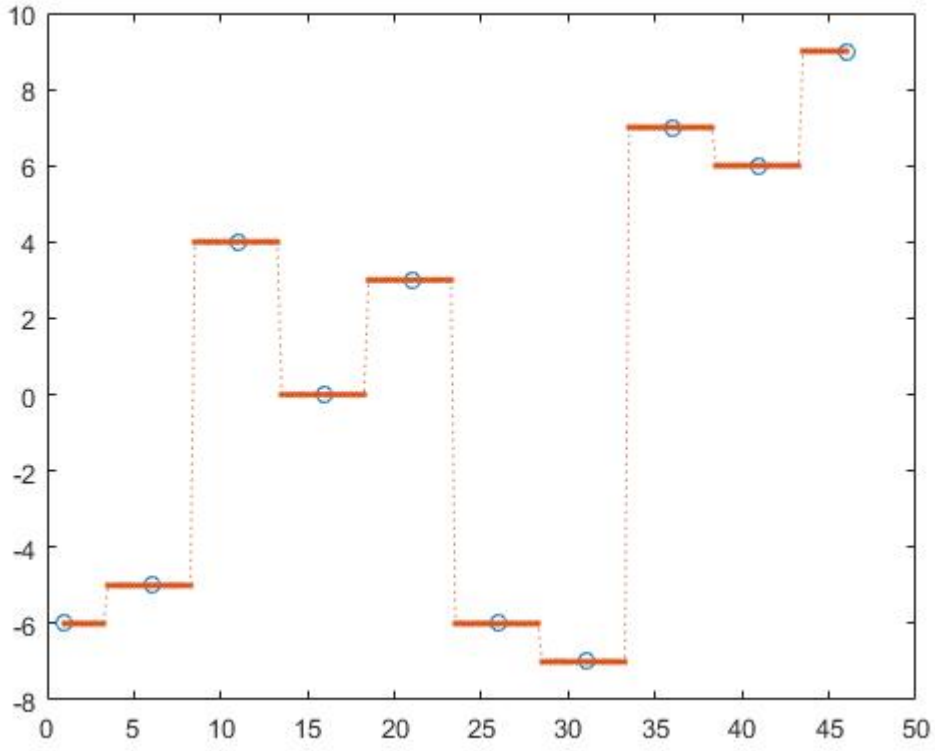
```
z_y = interp1(x,y,z,'linear');
```



De este modo, uno calcula la línea entre dos puntos adyacentes y obtiene `z_y` suponiendo que el punto sería un elemento de esas líneas.

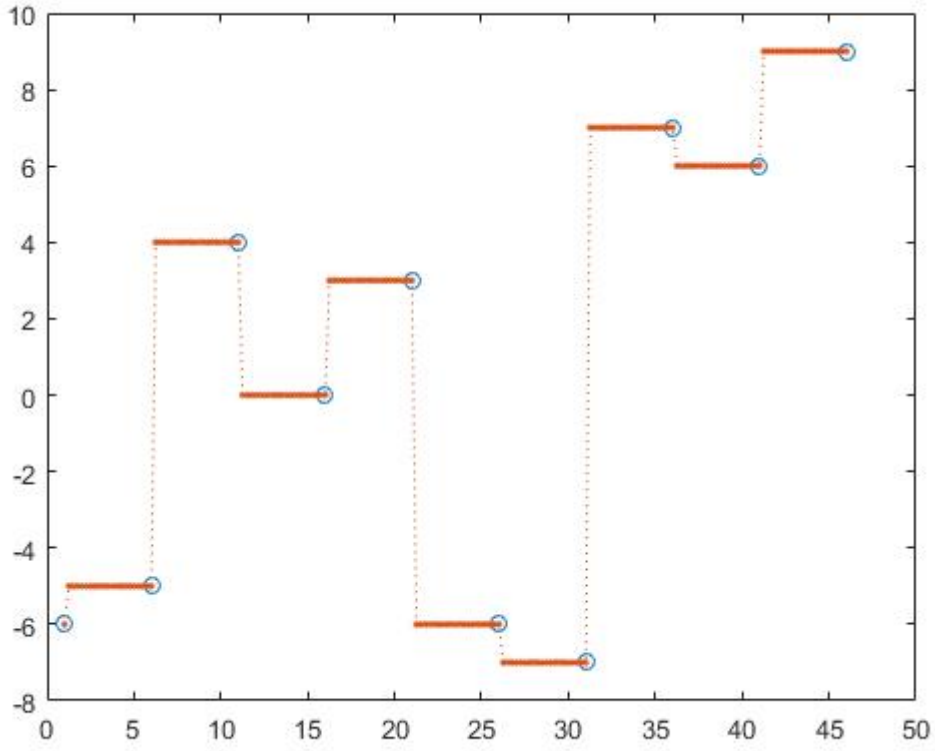
`interp1` proporciona otras opciones como la interpolación más cercana,

```
z_y = interp1(x,y,z, 'nearest');
```



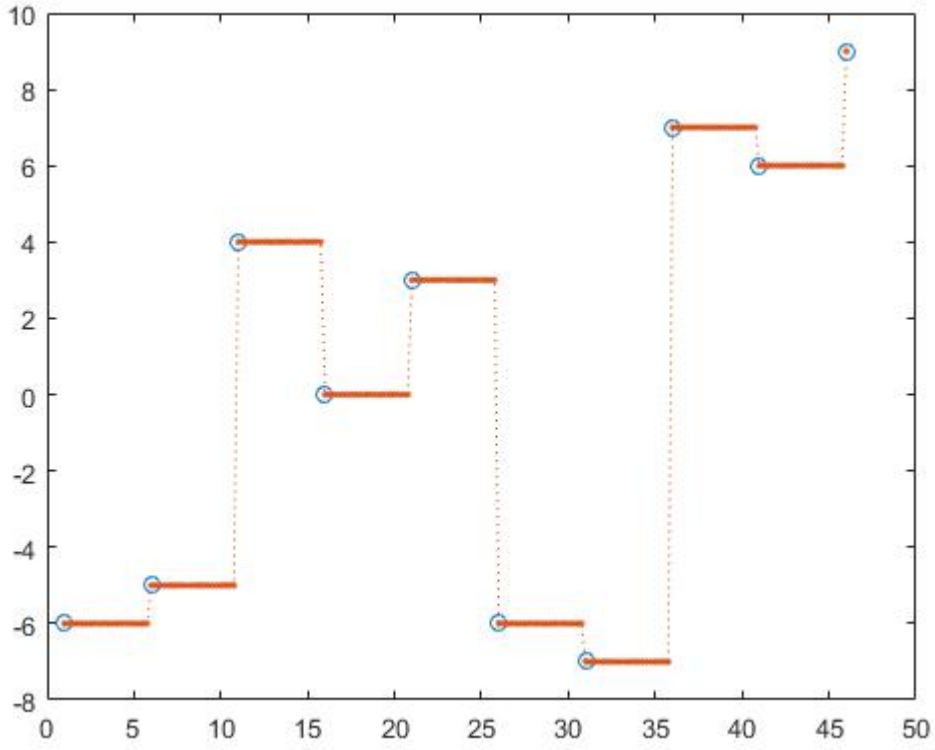
siguiente interpolación,

```
z_y = interp1(x,y,z, 'next');
```



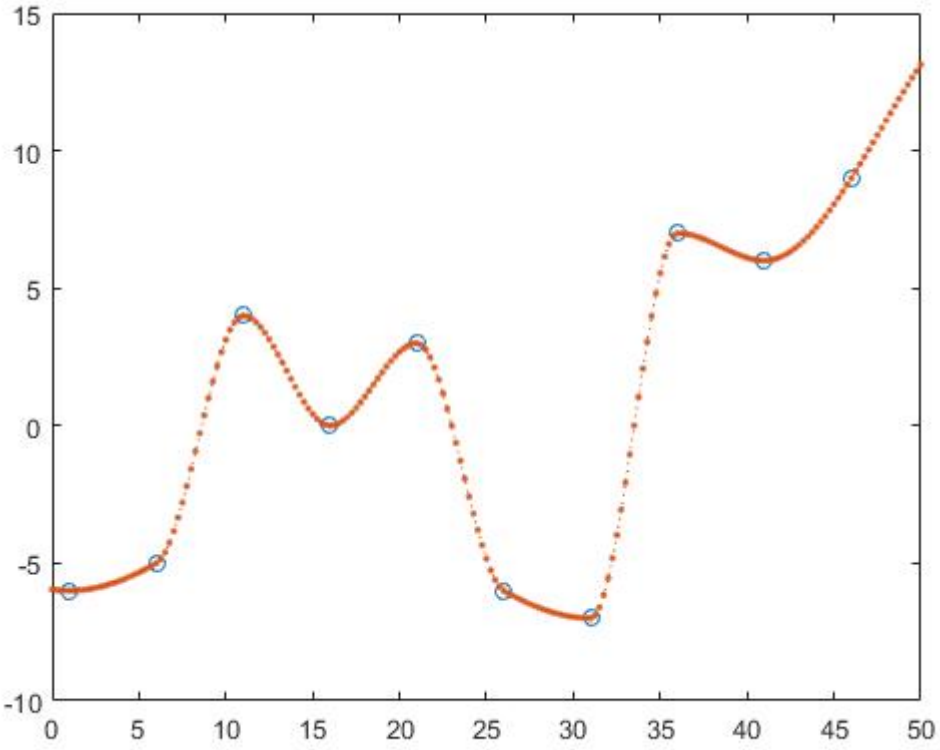
interpolación previa,

```
z_y = interp1(x,y,z, 'previous');
```

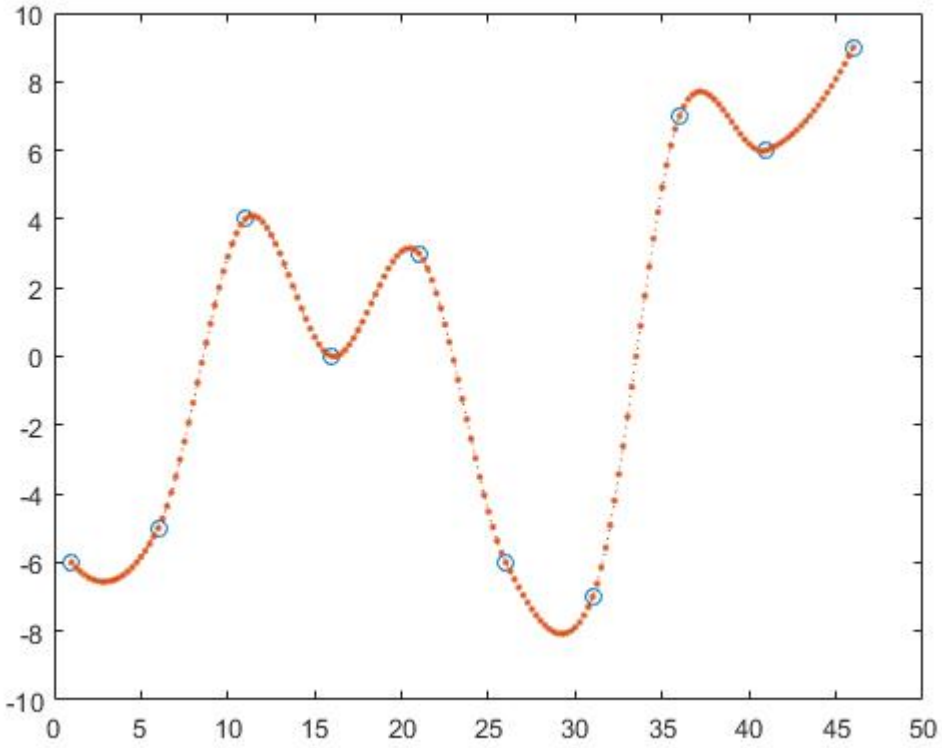


Interpolación cúbica por trozos que conserva la forma,

```
z_y = interp1(x,y,z, 'pchip');
```

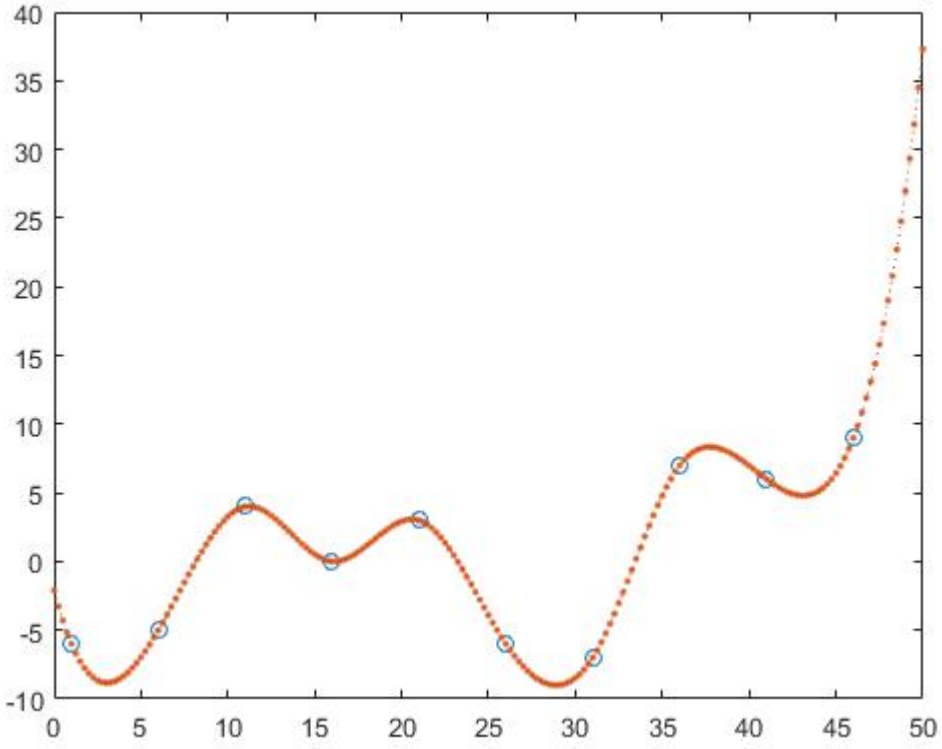


convolución cúbica, z_y = interp1 (x, y, z, 'v5cubic');



y la interpolación spline

```
z_y = interp1(x,y,z, 'spline');
```



De este modo, la interpolación más cercana, la siguiente y la anterior son interpolaciones constantes a trozos.

Interpolación polinómica

Inicializamos los datos que queremos interpolar:

```
x = 0:0.5:10;
y = sin(x/2);
```

Esto significa que la función subyacente para los datos en el intervalo [0,10] es sinusoidal. Ahora se están calculando los coeficientes de los polinomios aproximados:

```
p1 = polyfit(x,y,1);
p2 = polyfit(x,y,2);
p3 = polyfit(x,y,3);
p5 = polyfit(x,y,5);
p10 = polyfit(x,y,10);
```

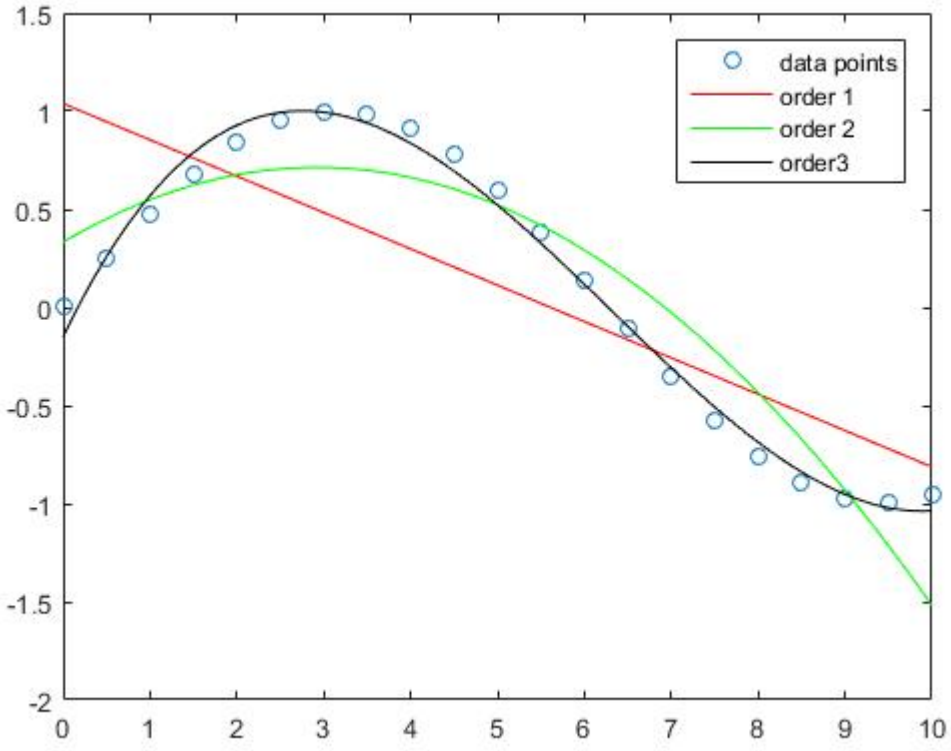
De este modo, `x` es el valor de x e `y` el valor de y de nuestros puntos de datos y el tercer número es el orden / grado del polinomio. Ahora configuramos la cuadrícula en la que queremos calcular nuestra función de interpolación en:

```
zx = 0:0.1:10;
```

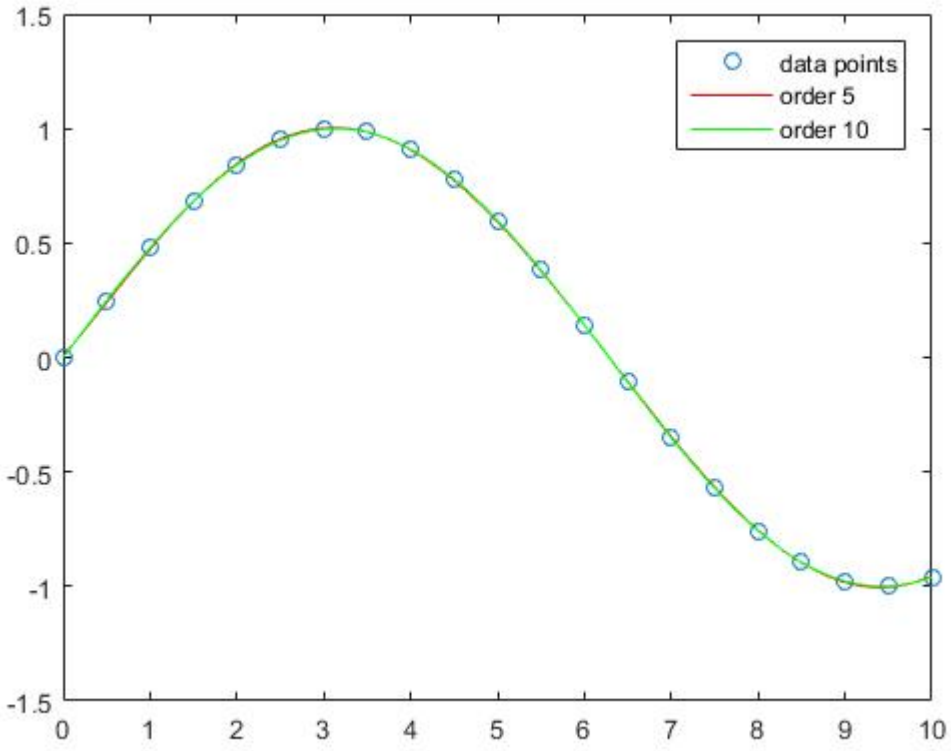
y calcular los valores de y:

```
zy1 = polyval(p1,zx);
zy2 = polyval(p2,zx);
zy3 = polyval(p3,zx);
zy5 = polyval(p5,zx);
zy10 = polyval(p10,zx);
```

Se puede ver que el error de aproximación para la muestra se reduce cuando aumenta el grado del polinomio.



Si bien la aproximación de la línea recta en este ejemplo tiene errores mayores, el polinomio de orden 3 se aproxima a la función sinusal en este intervalo relativamente bueno.



La interpolación con los polinomios orden 5 y orden 10 casi no tiene error de aproximación.

Sin embargo, si consideramos el rendimiento fuera de la muestra, se ve que los pedidos

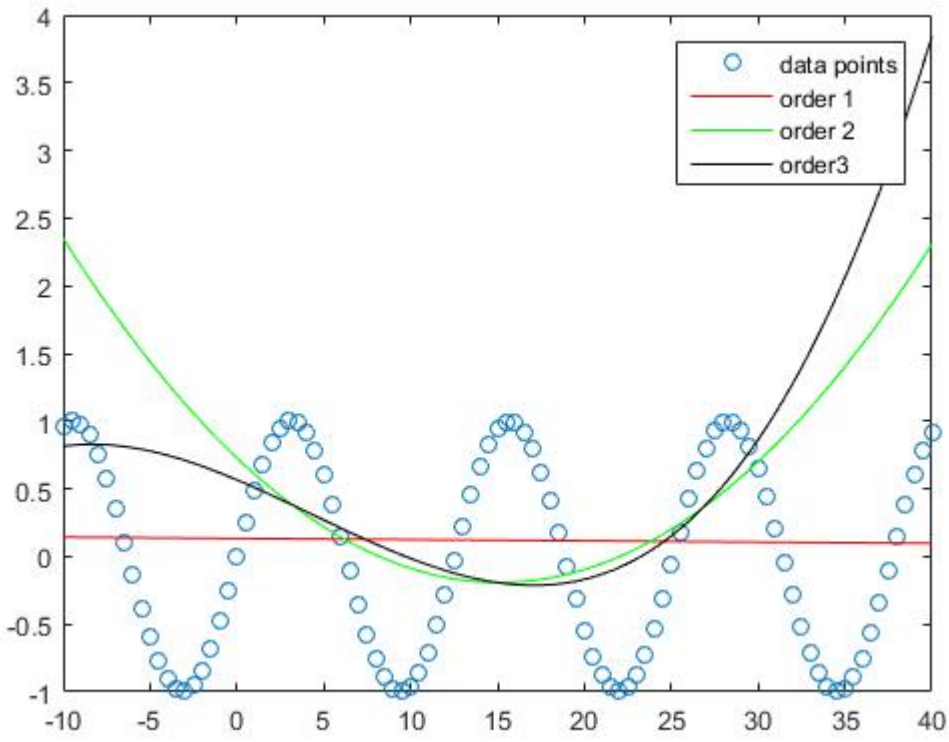
demasiado altos tienden a sobreponerse y, por lo tanto, obtener un mal rendimiento fuera de la muestra. Nosotros fijamos

```
zx = -10:0.1:40;
p10 = polyfit(X,Y,10);
p20 = polyfit(X,Y,20);
```

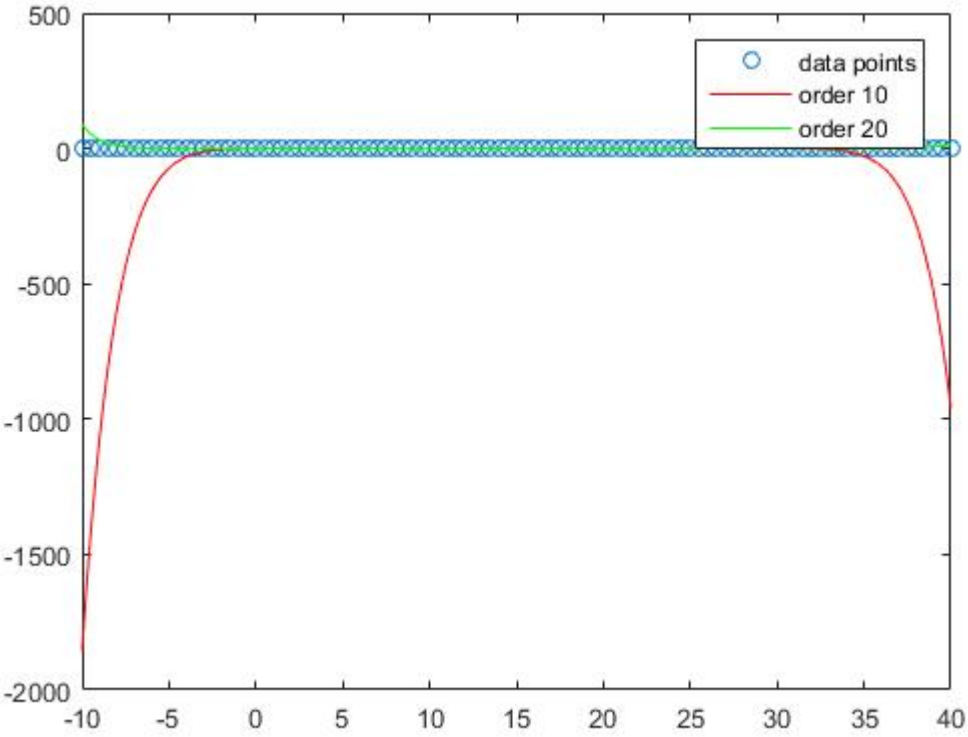
y

```
zy10 = polyval(p10,zx);
zy20 = polyval(p20,zx);
```

Si echamos un vistazo a la trama, vemos que el rendimiento fuera de la muestra es mejor para el orden 1



y sigue empeorando con el aumento de grado.



Lea Interpolación con MATLAB en línea: https://riptutorial.com/es/matlab/topic/6997/interpolacion-con-matlab

Capítulo 18: Introducción a la API de MEX

Examples

Verifique el número de entradas / salidas en un archivo MEX de C ++

En este ejemplo, escribiremos un programa básico que verifique el número de entradas y salidas pasadas a una función MEX.

Como punto de partida, necesitamos crear un archivo C ++ que implemente la "puerta de enlace MEX". Esta es la función que se ejecuta cuando se llama el archivo desde MATLAB.

testinputs.cpp

```
// MathWorks provided header file
#include "mex.h"

// gateway function
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // This function will error if number of inputs its not 3 or 4
    // This function will error if number of outputs is more than 1

    // Check inputs:
    if (nrhs < 3 || nrhs > 4) {
        mexErrMsgIdAndTxt("Testinputs:ErrorIdIn",
            "Invalid number of inputs to MEX file.");
    }

    // Check outputs:
    if (nlhs > 1) {
        mexErrMsgIdAndTxt("Testinputs:ErrorIdOut",
                "Invalid number of outputs to MEX file.");
    }
}
```

Primero, incluimos el encabezado `mex.h` que contiene definiciones de todas las funciones y tipos de datos necesarios para trabajar con la API de MEX. Luego implementamos la función `mexFunction` como se muestra, donde su firma no debe cambiar, independientemente de las entradas / salidas realmente utilizadas. Los parámetros de la función son los siguientes:

- `nlhs` : Número de salidas solicitadas.
- `*plhs[]` : Array que contiene todas las salidas en formato de API MEX.
- `nrhs` : Número de entradas pasadas.
- `*prhs[]` : Array que contiene todas las entradas en formato MEX API.

A continuación, verificamos el número de argumentos de entradas / salidas, y si la validación falla, se produce un error utilizando la función `mexErrMsgIdAndTxt` (se espera un nombre `somename:iD` identificador de formato `somename:iD` , una simple "ID" no funcionará).

Una vez que el archivo se compila como `mex testinputs.cpp` , se puede llamar a la función en MATLAB como:

```
>> testinputs(2,3)
Error using testinputs. Invalid number of inputs to MEX file.

>> testinputs(2,3,5)

>> [~,~] = testinputs(2,3,3)
Error using testinputs. Invalid number of outputs to MEX file.
```

Ingrese una cadena, modifíquela en C y envíela

En este ejemplo, ilustramos la manipulación de cadenas en MATLAB MEX. Crearemos una función MEX que acepte una cadena como entrada de MATLAB, copiaremos los datos en la cadena C, la modificaremos y la convertiremos de nuevo a `mxArray` devuelta al lado de MATLAB.

El objetivo principal de este ejemplo es mostrar cómo se pueden convertir las cadenas a C / C ++ desde MATLAB y viceversa.

stringIO.cpp

```
#include "mex.h"
#include <cstring>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // check number of arguments
    if (nrhs != 1 || nlhs > 1) {
        mexErrMsgIdAndTxt("StringIO:WrongNumArgs", "Wrong number of arguments.");
    }

    // check if input is a string
    if (mxIsChar(prhs[0])) {
        mexErrMsgIdAndTxt("StringIO:TypeError", "Input is not a string");
    }

    // copy characters data from mxArray to a C-style string (null-terminated)
    char *str = mxArrayToString(prhs[0]);

    // manipulate the string in some way
    if (strcmp("theOneString", str) == 0) {
        str[0] = 'T';  // capitalize first letter
    } else {
        str[0] = ' ';  // do something else?
    }

    // return the new modified string
    plhs[0] = mxCreateString(str);

    // free allocated memory
    mxFree(str);
}
```

Las funciones relevantes en este ejemplo son:

- `mxIsChar` para probar si un `mxArray` es de tipo `mxCHAR` .
- `mxArrayToString` para copiar los datos de una cadena `mxArray` a un `char *` buffer.
- `mxCreateString` para crear una cadena `mxArray` partir de un `char*` .

Como nota al margen, si solo desea leer la cadena y no modificarla, recuerde declararla como `const char*` por su velocidad y robustez.

Finalmente, una vez compilados lo podemos llamar desde MATLAB como:

```
>> mex stringIO.cpp

>> strOut = stringIO('theOneString')
strOut =
TheOneString

>> strOut = stringIO('somethingelse')
strOut=
omethingelse
```

Pasa una matriz 3D de MATLAB a C

En este ejemplo, ilustramos cómo tomar una matriz 3D doble de tipo real de MATLAB y pasarla a una matriz C `double*` .

Los principales objetivos de este ejemplo son mostrar cómo obtener datos de matrices MATLAB MEX y resaltar algunos pequeños detalles en el almacenamiento y manejo de matrices.

matrixIn.cpp

```
#include "mex.h"

void mexFunction(int  nlhs , mxArray *plhs[],
        int nrhs, mxArray const *prhs[]){
   // check amount of inputs
   if (nrhs!=1) {
        mexErrMsgIdAndTxt("matrixIn:InvalidInput", "Invalid number of inputs to MEX file.");
    }

   // check type of input
   if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])){
        mexErrMsgIdAndTxt("matrixIn:InvalidType",  "Input matrix must be a double, non-complex
array.");
   }

   // extract the data
   double const * const matrixAux= static_cast<double const *>(mxGetData(prhs[0]));

   // Get matrix size
   const mwSize *sizeInputMatrix= mxGetDimensions(prhs[0]);

   // allocate array in C. Note: its 1D array, not 3D even if our input is 3D
   double*  matrixInC= (double*)malloc(sizeInputMatrix[0] *sizeInputMatrix[1]
```

```
*sizeInputMatrix[2]* sizeof(double));


   // MATLAB is column major, not row major (as C). We need to reorder the numbers
   // Basically permutes dimensions

   // NOTE: the ordering of the loops is optimized for fastest memory access!
   // This improves the speed in about 300%

    const int size0 = sizeInputMatrix[0]; // Const makes compiler optimization kick in
    const int size1 = sizeInputMatrix[1];
    const int size2 = sizeInputMatrix[2];

    for (int j = 0; j < size2; j++)
    {
        int jOffset = j*size0*size1; // this saves re-computation time
        for (int k = 0; k < size0; k++)
        {
            int kOffset = k*size1; // this saves re-computation time
            for (int i = 0; i < size1; i++)
            {
                int iOffset = i*size0;
                matrixInC[i + jOffset + kOffset] = matrixAux[iOffset + jOffset + k];
            }
        }
    }

    // we are done!

    // Use your C matrix here

    // free memory
    free(matrixInC);
    return;
}
```

Los conceptos relevantes a tener en cuenta:

- Las matrices de MATLAB son todas 1D en la memoria, sin importar cuántas dimensiones tengan cuando se usan en MATLAB. Esto también es cierto para la mayoría (si no todas) de la representación de matriz principal en las bibliotecas C / C ++, ya que permite la optimización y la ejecución más rápida.
- Debe copiar explícitamente matrices de MATLAB a C en un bucle.
- Las matrices de MATLAB se almacenan en el orden mayor de la columna, como en Fortran, pero C / C ++ y la mayoría de los lenguajes modernos son la fila principal. Es importante permutar la matriz de entrada, de lo contrario los datos se verán completamente diferentes.

La función relevante en este ejemplo es:

- `mxIsDouble` comprueba si la entrada es de tipo `double` .
- `mxIsComplex` comprueba si la entrada es real o imaginaria.
- `mxGetData` devuelve un puntero a los datos reales en la matriz de entrada. `NULL` si no hay datos reales.
- `mxGetDimensions`

devuelve un puntero a una matriz `mwSize` , con el tamaño de la dimensión en cada índice.

Pasando una estructura por nombre de campo

Este ejemplo ilustra cómo leer las entradas de estructuras de varios tipos de MATLAB y pasarlas a C variables de tipo equivalente.

Si bien es posible y fácil averiguar a partir del ejemplo cómo cargar campos por números, aquí se logra comparando los nombres de los campos con cadenas. Por lo tanto, los campos de estructura pueden direccionarse por sus nombres de campo y las variables en ella pueden leerse por C.

structIn.c

```
#include "mex.h"
#include <string.h> // strcmp


void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  // helpers
  double* double_ptr;
  unsigned int i; // loop variable

  // to be read variables
  bool optimal;
  int randomseed;
  unsigned int desiredNodes;

  if (!mxIsStruct(prhs[0])) {
    mexErrMsgTxt("First argument has to be a parameter struct!");
  }
  for (i=0; i<mxGetNumberOfFields(prhs[0]); i++) {
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"randomseed")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      randomseed = *mxGetPr(p);
    }
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"optimal")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      optimal = (bool)*mxGetPr(p);
    }
    if (0==strcmp(mxGetFieldNameByNumber(prhs[0],i),"numNodes")) {
      mxArray *p = mxGetFieldByNumber(prhs[0],0,i);
      desiredNodes = *mxGetPr(p);
    }
  }
}
```

El bucle sobre `i` ejecuta en todos los campos de la estructura dada, mientras que las partes `if(0==strcmp)` comparan el nombre del campo matlab con la cadena dada. Si es una coincidencia, el valor correspondiente se extrae a una variable C.

Lea Introducción a la API de MEX en línea:

https://riptutorial.com/es/matlab/topic/680/introduccion-a-la-api-de-mex

Capítulo 19: Leyendo archivos grandes

Examples

textoscan

Suponga que ha formateado datos en un archivo de texto grande o una cadena, por ejemplo

```
Data,2015-09-16,15:41:52;781,780.000000,0.0034,2.2345
Data,2015-09-16,15:41:52;791,790.000000,0.1255,96.5948
Data,2015-09-16,15:41:52;801,800.000000,1.5123,0.0043
```

Uno puede usar `textscan` para leer esto bastante rápido. Para hacerlo, obtenga un identificador de archivo de texto con `fopen` :

```
fid = fopen('path/to/myfile');
```

Supongamos que para los datos de este ejemplo, queremos ignorar la primera columna "Datos", leer la fecha y la hora como cadenas y leer el resto de las columnas como dobles, es decir

```
 Data   ,  2015-09-16  , 15:41:52;801  , 800.000000  , 1.5123  ,  0.0043
ignore      string         string         double      double     double
```

Para hacer esto, llama:

```
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

El asterisco en `%*s` significa "ignorar esta columna". `%s` significa "interpretar como una cadena". `%f` significa "interpretar como dobles (flotadores)". Finalmente, `'Delimiter',','` establece que todas las comas deben interpretarse como el delimitador entre cada columna.

Para resumir:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

`data` ahora contienen una matriz de celdas con cada columna en una celda.

Cadenas de fecha y hora para matriz numérica rápido

La conversión de cadenas de fecha y hora en matrices numéricas se puede hacer con el `datenum` , aunque puede llevar la mitad del tiempo de lectura de un archivo de datos de gran tamaño.

Considere los datos en el ejemplo **Textscan** . De nuevo, al usar los textos, se puede interpretar la fecha y la hora como números enteros, se pueden convertir rápidamente en una matriz numérica.

Es decir, una línea en los datos de ejemplo se interpretaría como:

```
 Data , 2015  - 09  -  16  ,  15  :  41  :  52  ;  801 , 800.000000 , 1.5123 , 0.0043
ignore double double double double double double double    double     double   double
```

que se leerá como:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %f %f %f %f %f %f %f %f %f','Delimiter',',-:;');
fclose(fid);
```

Ahora:

```
y = data{1};          % year
m = data{2};          % month
d = data{3};          % day
H = data{4};          % hours
M = data{5};          % minutes
S = data{6};          % seconds
F = data{7};          % milliseconds

% Translation from month to days
ms = [0,31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334];

n = length(y);        % Number of elements
Time = zeros(n,1);    % Declare numeric time array

% Algorithm for calculating numeric time array
for k = 1:n
    Time(k) = y(k)*365 + ms(m(k)) + d(k) + floor(y(k)/4)...
              - floor(y(k)/100) + floor(y(k)/400) + (mod(y(k),4)~=0)...
              - (mod(y(k),100)~=0) + (mod(y(k),400)~=0)...
              + (H(k)*3600 + M(k)*60 + S(k) + F(k)/1000)/86400 + 1;
end
```

El uso del `datenum` en 566,678 elementos requirió 6,626570 segundos, mientras que el método anterior requirió 0,048334 segundos, es decir, el 0,73% del tiempo para el `datenum` o ~ 137 veces más rápido.

Lea Leyendo archivos grandes en línea: https://riptutorial.com/es/matlab/topic/9023/leyendo-archivos-grandes

Capítulo 20: Matrices de descomposiciones

Sintaxis

1. R = chol (A);
2. [L, U] = lu (A);
3. R = qr (A);
4. T = schur (A);
5. [U, S, V] = svd (A);

Examples

Descomposición de Cholesky

La descomposición de Cholesky es un método para descomponer una matriz definitiva, positiva y hermítica en una matriz triangular superior y su transposición. Puede usarse para resolver sistemas de ecuaciones lineales y es aproximadamente el doble de rápido que la descomposición de LU.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = chol(A);
```

Esto devuelve la matriz triangular superior. El inferior se obtiene por transposición.

```
L = R';
```

Finalmente podemos comprobar si la descomposición fue correcta.

```
b = (A == L*R);
```

Descomposición QR

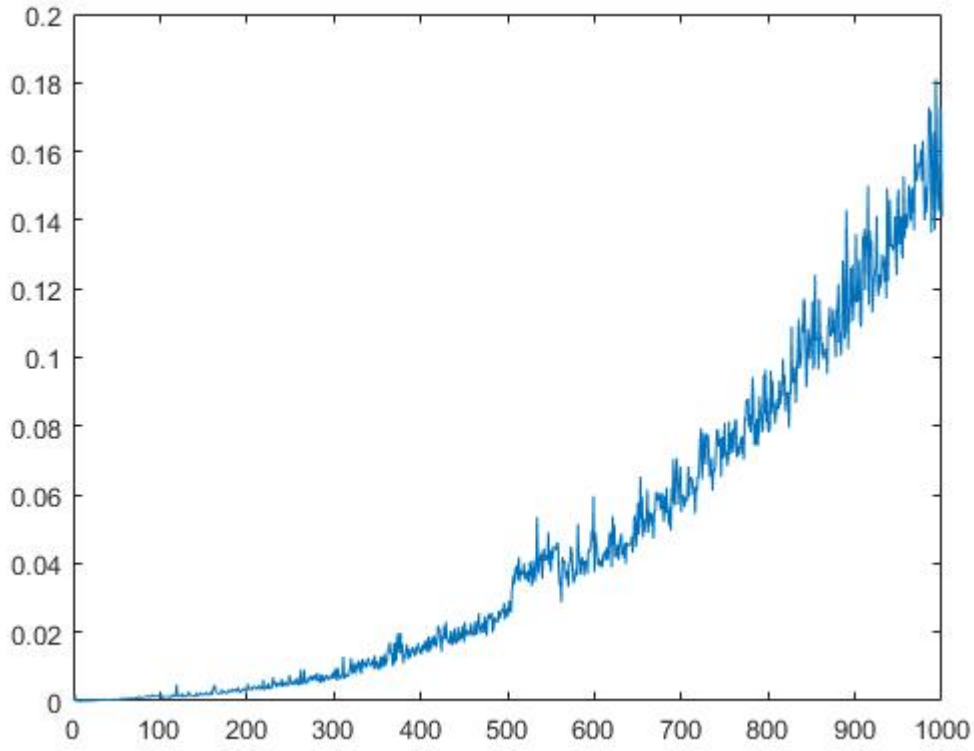
Este método descompondrá una matriz en una matriz triangular superior y ortogonal.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = qr(A);
```

Esto devolverá la matriz triangular superior, mientras que lo siguiente devolverá ambas matrices.

```
[Q,R] = qr(A);
```

La siguiente gráfica mostrará el tiempo de ejecución de `qr` dependiente de la raíz cuadrada de los elementos de la matriz.



Descomposición de LU

De este modo, una matriz se descompondrá en una matriz trangular superior y una matriz triangular inferior. A menudo se utilizará para aumentar el rendimiento y la estabilidad (si se hace con permutación) de la eliminación de Gauss.

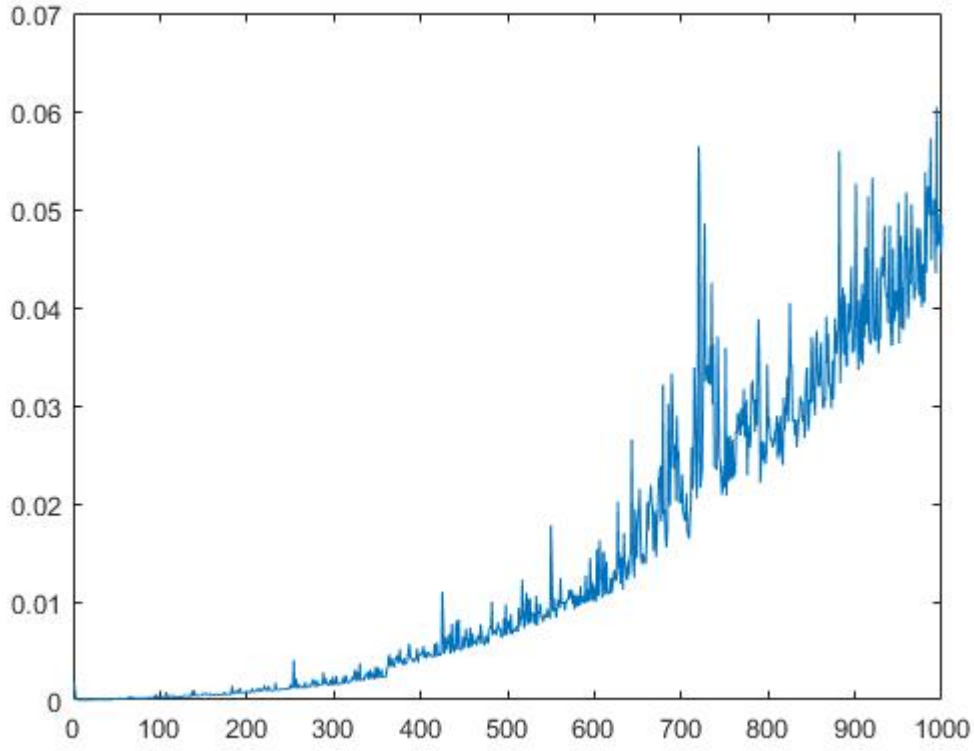
Sin embargo, muy a menudo este método no funciona o funciona mal, ya que no es estable. Por ejemplo

```
A = [8 1 6
    3 5 7
    4 9 2];
[L,U] = lu(A);
```

Es suficiente agregar una matriz de permutación tal que PA = LU:

```
[L,U,P]=lu(A);
```

A continuación, trazaremos el tiempo de ejecución de `lu 'dependiente de la raíz cuadrada de los elementos de la matriz.

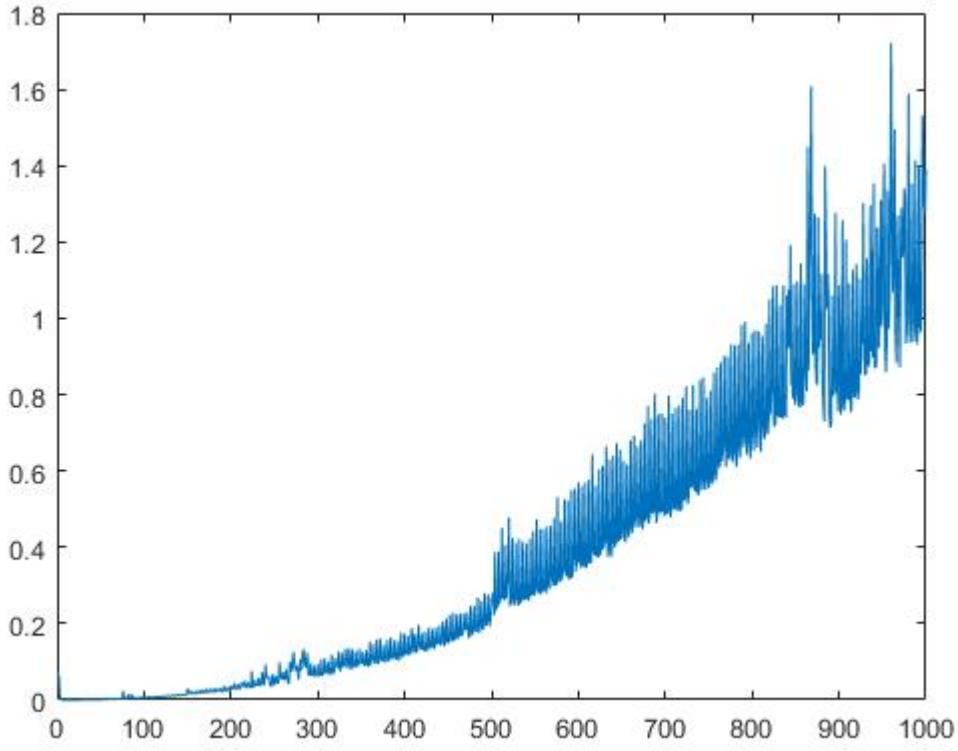


Descomposición de Schur

Si A es una matriz compleja y cuadrática, existe una Q unitaria tal que Q * AQ = T = D + N, siendo D la matriz diagonal que consta de los valores propios y N es estrictamente tridiagonal superior.

```
A = [3 6 1
    23 13 1
    0 3 4];
T = schur(A);
```

También mostramos el tiempo de ejecución de `schur` dependiendo de la raíz cuadrada de los elementos de la matriz:



Valor singular de descomposición

Dada una m veces n matriz A con n mayor que m. La descomposición del valor singular.

```
[U,S,V] = svd(A);
```

calcula las matrices U, S, V.

La matriz U está formada por los vectores propios singulares izquierdos que son los vectores propios de `A*A.'` mientras que V consiste en los valores propios e individuales singulares que son los vectores propios de `A.'*A` La matriz `S` tiene las raíces cuadradas de los valores propios de `A*A.'` y `A.'*A` en su diagonal.

Si m es mayor que n se puede usar

```
[U,S,V] = svd(A,'econ');
```

Para realizar una descomposición del valor singular de tamaño económico.

Lea Matrices de descomposiciones en línea: https://riptutorial.com/es/matlab/topic/6163/matrices-de-descomposiciones

Capítulo 21: Mejores Prácticas de MATLAB

Observaciones

Este tema muestra las mejores prácticas que la comunidad ha aprendido con el tiempo.

Examples

Mantener las líneas cortas

Use el carácter de continuación (puntos suspensivos) `...` para continuar con una declaración larga.

Ejemplo:

```
MyFunc( parameter1,parameter2,parameter3,parameter4, parameter5, parameter6,parameter7,
parameter8, parameter9)
```

puede ser reemplazado por:

```
MyFunc( parameter1, ...
        parameter2, ...
        parameter3, ...
        parameter4, ...
        parameter5, ...
        parameter6, ...
        parameter7, ...
        parameter8, ...
        parameter9)
```

Indentar correctamente el código

La indentación adecuada proporciona no solo el aspecto estético sino que también aumenta la legibilidad del código.

Por ejemplo, considere el siguiente código:

```
%no need to understand the code, just give it a look
n = 2;
bf = false;
while n>1
for ii = 1:n
for jj = 1:n
if ii+jj>30
bf = true;
break
end
end
if bf
break
```

```
end
end
if bf
break
end
n = n + 1;
end
```

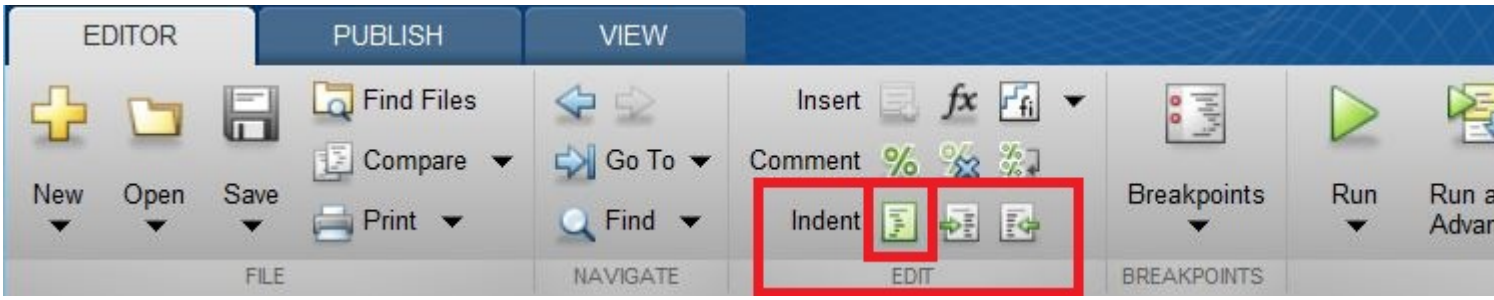
Como puede ver, debe mirar detenidamente para ver qué bucle y `if` declaraciones terminan donde.
Con sangría inteligente, obtendrá este look:

```
n = 2;
bf = false;
while n>1
    for ii = 1:n
        for jj = 1:n
            if ii+jj>30
                bf = true;
                break
            end
        end
        if bf
            break
        end
    end
    if bf
        break
    end
    n = n + 1;
end
```

Esto indica claramente el inicio y el final de los bucles / `if` .

Puedes hacer sangría inteligente por:
• seleccionando todo tu código (`Ctrl` + `A`)

• y luego presionando `Ctrl` + `I` o haciendo clic  desde la barra de edición.



Usar afirmar

Matlab permite que algunos errores muy triviales pasen silenciosamente, lo que podría causar un error que se generará mucho más tarde en la ejecución, lo que dificulta la depuración. Si **asumes** algo sobre tus variables, **valídalo** .

```
function out1 = get_cell_value_at_index(scalar1,cell2)
```

```
assert(isscalar(scalar1),'1st input must be a scalar')
assert(iscell(cell2),'2nd input must be a cell array')

assert(numel(cell2) >= scalar1),'2nd input must have more elements than the value of the 1st
input')
assert(~isempty(cell2{scalar1}),'2nd input at location is empty')

out1 = cell2{scalar1};
```

Evitar bucles

La mayoría de las veces, los bucles son computacionalmente costosos con Matlab. Su código será órdenes de magnitudes más rápido si utiliza la vectorización. También a menudo hace que su código sea más modular, fácilmente modificable y más fácil de depurar. El principal inconveniente es que debe tomar tiempo para planificar las estructuras de datos, y es más fácil encontrar errores de dimensión.

Ejemplos

No escribas

```
for t=0:0.1:2*pi
    R(end+1)=cos(t);
end
```

pero

```
t=0:0.1:2*pi;
R=cos(t)
```

No escribas

```
for i=1:n
    for j=1:m
        c(i,j)=a(i)+2*b(j);
    end
end
```

Pero algo similar a

```
c=repmat(a.',1,m)+2*repmat(b,n,1)
```

Para más detalles, ver vectorización.

Crear un nombre único para el archivo temporal

Al codificar un script o una función, puede darse el caso de que se necesite uno o más de un archivo temporal para, por ejemplo, almacenar algunos datos.

Para evitar sobrescribir un archivo existente o para sombrear una función MATLAB, la función tempname se puede usar para generar un **nombre único** para un archivo temporal en la carpeta temporal del sistema.

```
my_temp_file=tempname
```

El nombre del archivo se genera sin la extensión; se puede agregar concatenando la extensión deseada al nombre generado por `tempname`

```
my_temp_file_with_ext=[tempname '.txt']
```

La ubicación de la carpeta temporal del sistema se puede recuperar llamando a la función tempdir .

Si, durante la ejecución de la función / script, el archivo temporal ya no es necesario, se puede eliminar mediante la función eliminar

Desde `delete` no pide confirmación, que podría ser útil para establecer `on` la opción de mover el archivo que desea borrar en el `recycle` carpeta.

Esto se puede hacer usando la función reciclar de esta manera:

```
recycle('on')
```

En el siguiente ejemplo, se propone un posible uso de las funciones `tempname` , `delete` y `recycle` .

```
%
% Create some example data
%
theta=0:.1:2*pi;
x=cos(theta);
y=sin(theta);
%
% Generate the temporary filename
%
my_temp_file=[tempname '.mat'];
%
% Split the filename (path, name, extension) and display them in a message box
[tmp_file_path,tmp_file_name, tmp_file_ext]=fileparts(my_temp_file)
uiwait(msgbox(sprintf('Path= %s\nName= %s\nExt= %s', ...
              tmp_file_path,tmp_file_name,tmp_file_ext),'TEMPORARY FILE'))
%
% Save the varaibles in a temporary file
%
save(my_temp_file,'x','y','theta')
%
% Load the varaibles from the temporary file
%
load(my_temp_file)
%
% Set the reclycle option on
%
recycle('on')
%
```

```
% Delete the temporary file
%
delete(my_temp_file)
```

Advertencia

El nombre de archivo temporal se genera utilizando el método `java.util.UUID.randomUUID` (randomUUID).

Si MATLAB se ejecuta sin JVM, el nombre de archivo temporal se genera usando `matlab.internal.timing.timing` basado en el contador y el tiempo de la CPU. En este caso, no se garantiza que el nombre de archivo temporal sea único.

Use validateattributes

La función validateattributes se puede usar para validar una matriz contra un conjunto de especificaciones

Por lo tanto, se puede utilizar para validar la entrada proporcionada a una función.

En el siguiente ejemplo, la función `test_validateattributes` requiere tres entradas

```
function test_validateattributes(input_1,input_2,input_3)
```

Las especificaciones de entrada son:

- array_1:
    - clase: doble
    - tamaño: [3,2]
    - Valores: los elementos no deben ser NaN.
- char_array:
    - clase: char
    - valor: la cadena no debe estar vacía
- array_3
    - clase: doble
    - tamaño: [5 1]
    - Valores: los elementos deben ser reales.

Para validar las tres entradas, la función `validateattributes` se puede llamar con la siguiente sintaxis:

```
validateattributes(A,classes,attributes,funcName,varName,argIndex)
```

dónde:

- `A` es la matriz a ser anulada.

- `classes` : es el `type` de la matriz (por ejemplo, `single` , `double` , `logical`)
- `attributes` : son los atributos que la matriz de entrada debe coincidir (p `[3,2], size` ej., `[3,2], size` para especificar el tamaño de la matriz, `nonnan` para especificar que la matriz no tendrá valores de NaN)
- `funcName` : es el nombre de la función en la que se produce la validación. Este argumento se utiliza en la generación del mensaje de error (si existe)
- `varName` : es el nombre de la matriz bajo validación. Este argumento se utiliza en la generación del mensaje de error (si existe)
- `argIndex` : es la posición de la matriz de entrada en la lista de entrada. Este argumento se utiliza en la generación del mensaje de error (si existe)

En caso de que una o más de una entrada no coincida con la especificación, se genera un mensaje de error.

En caso de que haya más de una entrada no válida, la validación se detiene cuando se encuentra la primera falta de coincidencia.

Esta es la `function test_validateattributes` en la que se ha implementado la validación de entrada.

Dado que la función requiere tres entradas, se realiza una primera comprobación del número de entradas proporcionadas utilizando el criterio de función .

```
function test_validateattributes(array_1,char_array_1,array_3)
%
% Check for the number of expected input: if the number of input is less
% than the require, the function exits with an error message
%
if(nargin ~= 3)
   error('Error: TEST_VALIDATEATTRIBUTES requires 3 input, found %d',nargin)
end
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the first input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #1 name (only used in the generation of the error message)
%
input_1_name='array_1';
%
% INPUT #1 position (only used in the generation of the error message)
%
input_1_position=1;
%
% Expected CLASS of the first input MUST BE "double"
%
input_1_class={'double'};
%
% Expected ATTRIBUTES of the first input
%   SIZE: MUST BE [3,2]
%
input_1_size_attribute='size';
input_1_size=[3,2];
%
%   VALUE CHECK: the element MUST BE NOT NaN
```

```
%
input_1_value_type='nonnan';
%
% Build the INPUT 1 attributes
%
input_1_attributes={input_1_size_attribute,input_1_size, ...
                    input_1_value_type};
%
% CHECK THE VALIDITY OF THE FIRST INPUT
%
validateattributes(array_1, ...
                   input_1_class,input_1_attributes,'', ...
                   input_1_name,input_1_position);

%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the second input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #1 name (only used in the generation of the error message)
%
input_2_name='char_array_1';
%
% INPUT #2 position (only used in the generation of the error message)
%
input_2_position=2;
%
% Expected CLASS of the first input MUST BE "string"
%
input_2_class={'char'};
%
%   VALUE CHECK: the element must be not NaN
%
input_2_size_attribute='nonempty';
%
% Build the INPUT 2 attributes
%
input_2_attributes={input_2_size_attribute};
%
% CHECK THE VALIDITY OF THE SECOND INPUT
%
validateattributes(char_array_1, ...
                   input_2_class,input_2_attributes,'', ...
                   input_2_name,input_2_position);
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the expected characteristics of the third input %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% INPUT #3 name (only used in the generation of the error message)
%
input_3_name='array_3';
%
% INPUT #3 position (only used in the generation of the error message)
%
input_3_position=3;
%
% Expected CLASS of the first input MUST BE "double"
%
input_3_class={'double'};
%
```

```
% Expected ATTRIBUTES of the first input
%   SIZE: must be [5]
input_3_size_attribute='size';
input_3_size=[5 1];
%   VALUE CHECK: the elements must be real
input_3_value_type='real';
%
% Build the INPUT 3 attributes
%
input_3_attributes={input_3_size_attribute,input_3_size, ...
                    input_3_value_type};
%
% CHECK THE VALIDITY OF THE FIRST INPUT
%
validateattributes(array_3, ...
                   input_3_class,input_3_attributes,'', ...
                   input_3_name,input_3_position);

disp('All the three input are OK')
```

El siguiente script se puede utilizar para probar la implementación del procedimiento de validación.

Genera las tres entradas requeridas y, al azar, las hace no válidas.

```
%
% Generate the first input
%
n_rows=randi([2 3],1);
n_cols=2;
input_1=randi([20 30],n_rows,n_cols);
%
% Generate the second input
%
if(rand > 0.5)
   input_2='This is a string';
else
   input_2='';
end
%
% Generate the third input
%
input_3=acos(rand(5,1)*1.2);
%
% Call the test_validateattributes function with the above generated input
%
input_1
input_2
input_3
%
test_validateattributes(input_1,input_2,input_3)
```

Estos son algunos ejemplos de entradas incorrectas detectadas por la función `validateattributes` :

Entrada incorrecta

```
input_1 =
```

```
    23    22
    26    28

input_2 =

     ''

input_3 =

   0.0000 + 0.4455i
   1.2420 + 0.0000i
   0.4063 + 0.0000i
   1.3424 + 0.0000i
   1.2186 + 0.0000i

Error using test_validateattributes (line 44)
Expected input number 1, array_1, to be of size 3x2 when it is actually
size 2x2.
```

Entrada incorrecta

```
input_1 =

    22    24
    21    25
    26    27

input_2 =

This is a string

input_3 =

   1.1371 + 0.0000i
   0.6528 + 0.0000i
   1.0479 + 0.0000i
   0.0000 + 0.1435i
   0.0316 + 0.0000i

Error using test_validateattributes (line 109)
Expected input number 3, array_3, to be real.
```

Entrada valida

```
input_1 =

    20    25
    25    28
    24    23

input_2 =

This is a string

input_3 =

    0.9696
    1.5279
    1.3581
```

```
    0.5234
    0.9665

All the three input are OK
```

Operador de comentarios de bloque

Es una buena práctica agregar comentarios que describan el código. Es útil para otros e incluso para el codificador cuando se devuelve más tarde. Se puede comentar una sola línea usando el símbolo `%` o usando las `Ctrl+R` acceso rápido `Ctrl+R` Para descomentar una línea comentada previamente, elimine el símbolo `%` o use la tecla de `Crtl+T` directo `Crtl+T`

Mientras se puede comentar un bloque de código al agregar un símbolo `%` al comienzo de cada línea, las versiones más recientes de MATLAB (después de 2015a) le permiten usar el **Operador de comentarios de bloque** `%{ code %}` . Este operador aumenta la legibilidad del código. Puede usarse tanto para comentar el código como para la documentación de ayuda de la función. El bloque se puede **plegar** y **desplegar** para aumentar la legibilidad del código.

```
1  function y = myFunction(x)
2  %{
3  myFunction  Binary Singleton Expansion Function
4  y = myFunction(x) applies the element-by-element binary operation
5  specified by the function handle FUNC to arrays A and B, with impli
6  expansion enabled.
7  %}
8
9  %%  Compute z(x, y) = x.*sin(y) on a grid:
10 % x = 1:10;
11 y = x.';
12
13 %{
14 z = zeros(numel(x),numel(y));
15 for ii=1:numel(x)
16     for jj=1:numel(y)
17         z(ii,jj) = x(ii)*sin(y(jj));
18     end
19 end
20 %}
21
22 z = bsxfun(@(x, y) x.*sin(y), x, y);
23 y = y + z;
24
25 end
```

Como se puede ver, los operadores `%{` y `%}` deben aparecer solos en las líneas. No incluya ningún otro texto en estas líneas.

```
function y = myFunction(x)
%{
myFunction  Binary Singleton Expansion Function
y = myFunction(x) applies the element-by-element binary operation
specified by the function handle FUNC to arrays A and B, with implicit
expansion enabled.
%}

%%  Compute z(x, y) = x.*sin(y) on a grid:
% x = 1:10;
y = x.';

%{
z = zeros(numel(x),numel(y));
for ii=1:numel(x)
    for jj=1:numel(y)
        z(ii,jj) = x(ii)*sin(y(jj));
    end
end
%}

z = bsxfun(@(x, y) x.*sin(y), x, y);
y = y + z;

end
```

Lea Mejores Prácticas de MATLAB en línea: https://riptutorial.com/es/matlab/topic/2887/mejores-practicas-de-matlab

Capítulo 22: Multihilo

Examples

Usando Parfor para paralelizar un bucle

Puede usar `parfor` para ejecutar las iteraciones de un bucle en paralelo:

Ejemplo:

```
poolobj = parpool(2);       % Open a parallel pool with 2 workers

s = 0;                      % Performing some parallel Computations
parfor i=0:9
    s = s + 1;
end
disp(s)                     % Outputs '10'

delete(poolobj);            % Close the parallel pool
```

Nota: `parfor` no puede ser anidado directamente. Para el anidamiento de `parfor` use una función en parr `parfor` y agregue el segundo `parfor` en esa función.

Ejemplo:

```
parfor i = 1:n
[op] = fun_name(ip);
end

function [op] = fun_name(ip)
parfor j = 1:length(ip)
% Some Computation
end
```

Cuando usar parfor

Básicamente, se recomienda `parfor` en dos casos: muchas iteraciones en su bucle (es decir, como `1e10`), o si cada iteración toma mucho tiempo (por ejemplo, `eig(magic(1e4))`). En el segundo caso es posible que desee considerar el uso de `spmd` . La razón `parfor` es más lento que una `for` lazo para distancias cortas o iteraciones rápidas es la sobrecarga necesaria para gestionar correctamente todos los trabajadores, en lugar de sólo hacer el cálculo.

Además, muchas funciones tienen un subprocesamiento múltiple implícito incorporado , lo que hace que un bucle `parfor` no sea más eficiente, cuando se usan estas funciones, que un bucle serie `for` bucle, ya que todos los núcleos ya están en uso. `parfor` realmente será un detrimento en este caso, ya que tiene la sobrecarga de asignación, mientras que es tan paralelo como la función que está tratando de usar.

Considere el siguiente ejemplo para ver el comportamiento de `for` en oposición al de `parfor` .

Primero abre la piscina paralela si aún no lo has hecho:

```
gcp; % Opens a parallel pool using your current settings
```

Luego ejecuta un par de bucles grandes:

```
n = 1000; % Iteration number
EigenValues = cell(n,1); % Prepare to store the data
Time = zeros(n,1);
for ii = 1:n
tic
    EigenValues{ii,1} = eig(magic(1e3)); % Might want to lower the magic if it takes too long
Time(ii,1) = toc; % Collect time after each iteration
end

figure; % Create a plot of results
plot(1:n,t)
title 'Time per iteration'
ylabel 'Time [s]'
xlabel 'Iteration number[-]';
```

Luego haz lo mismo con `parfor` lugar de `for` . Notarás que el tiempo promedio por iteración aumenta. Sin embargo, `parfor` cuenta que el `parfor` usó todos los trabajadores disponibles, por lo tanto, el tiempo total (`sum(Time)`) debe dividirse por el número de núcleos en su computadora.

Entonces, mientras que el tiempo para hacer cada iteración por separado aumenta usando `parfor` con respecto a usar `for` , el tiempo total disminuye considerablemente.

Ejecutar comandos en paralelo utilizando una declaración "Programa único, datos múltiples" (SPMD)

A diferencia de un bucle for paralelo (`parfor`), que toma las iteraciones de un bucle y las distribuye entre varios subprocesos, un solo programa, una declaración de datos múltiples (`spmd`) toma una serie de comandos y los distribuye a **todos** los subprocesos, de modo que cada uno El hilo ejecuta el comando y almacena los resultados. Considera esto:

```
poolobj = parpool(2);    % open a parallel pool with two workers

spmd
    q = rand(3);         % each thread generates a unique 3x3 array of random numbers
end

q{1}             % q is called like a cell vector
q{2}             % the values stored in each thread may be accessed by their index

delete(poolobj)  % if the pool is closed, then the data in q will no longer be accessible
```

Es importante señalar que cada hilo se puede acceder durante el `spmd` bloque por su índice de hilo (también llamada índice de laboratorio, o `labindex`):

```
poolobj = parpool(2);        % open a parallel pool with two workers
```

```
spmd
    q = rand(labindex + 1);  % each thread generates a unique array of random numbers
end

size(q{1})                   % the size of q{1} is 2x2
size(q{2})                   % the size of q{2} is 3x3

delete(poolobj)              % q is no longer accessible
```

En ambos ejemplos, `q` es un objeto compuesto , que puede inicializarse con el comando `q = Composite()` . Es importante tener en cuenta que los objetos compuestos solo son accesibles mientras se está ejecutando el grupo.

Usando el comando batch para hacer varios cálculos en paralelo

Para usar subprocesos múltiples en MATLAB, se puede usar el comando `batch` . Tenga en cuenta que debe tener instalada la caja de herramientas de computación paralela.

Para un script que consume mucho tiempo, por ejemplo,

```
for ii=1:1e8
   A(ii)=sin(ii*2*pi/1e8);
end
```

para ejecutarlo en modo batch deberías usar lo siguiente:

```
job=batch("da")
```

que permite que MATLAB se ejecute en modo por lotes y permite utilizar MATLAB mientras tanto para hacer otras cosas, como agregar más procesos por lotes.

Para recuperar los resultados después de finalizar el trabajo y cargar la matriz `A` en el espacio de trabajo:

```
load(job, 'A')
```

Finalmente, abra la "interfaz de trabajo del monitor" desde *Inicio* → *Entorno* → *Paralelo* → *Supervisar trabajos* y eliminar el trabajo a través de:

```
delete(job)
```

Para cargar una función para el procesamiento por lotes, simplemente use esta declaración donde `fcn` es el nombre de la función, `N` es el número de matrices de salida y `x1` , `...` , `xn` son matrices de entrada:

```
 j=batch(fcn, N, {x1, x2, ..., xn})
```

Lea Multihilo en línea: https://riptutorial.com/es/matlab/topic/4378/multihilo

Capítulo 23: Para bucles

Observaciones

Iterar sobre el vector de columna

Una fuente común de errores está intentando hacer un bucle sobre los elementos de un vector de columna. Un vector de columna se trata como una matriz con una columna. (En realidad no hay distinción en Matlab). El bucle `for` ejecuta una vez con la variable de bucle establecida en la columna.

```
% Prints once: [3, 1]
my_vector = [1; 2; 3];
for i = my_vector
    display(size(i))
end
```

Alterar la variable de iteración.

La modificación de la variable de iteración cambia su valor para la iteración actual, pero no tiene impacto en su valor en iteraciones posteriores.

```
% Prints 1, 2, 3, 4, 5
for i = 1:5
    display(i)
    i = 5; % Fail at trying to terminate the loop
end
```

Caso especial de `a:b` en el lado derecho.

El ejemplo básico trata `1:n` como una instancia normal de crear un vector de fila y luego iterar sobre él. Por razones de rendimiento, Matlab en realidad trata cualquier `a:b` o `a:c:b` especialmente al no crear el vector de fila por completo, sino a cada elemento de uno en uno.

Esto se puede detectar alterando ligeramente la sintaxis.

```
% Loops forever
for i = 1:1e50
end
```

```
% Crashes immediately
for i = [1:1e50]
end
```

Examples

Lazo 1 a n

El caso más simple es simplemente realizar una tarea para un número fijo de veces conocido. Digamos que queremos mostrar los números entre 1 a n, podemos escribir:

```
n = 5;
for k = 1:n
    display(k)
end
```

El bucle se ejecute la instrucción (s) interior, todo entre el `for` y la `end` , por `n` veces (5 en este ejemplo):

```
1
2
3
4
5
```

Aquí hay otro ejemplo:

```
n = 5;
for k = 1:n
    disp(n-k+1:-1:1) % DISP uses more "clean" way to print on the screen
end
```

esta vez usamos tanto `n` como `k` en el bucle para crear una pantalla "anidada":

```
 5     4     3     2     1

 4     3     2     1

 3     2     1

 2     1

 1
```

Iterar sobre elementos de vector

El lado derecho de la asignación en un bucle `for` puede ser cualquier vector de fila. El lado izquierdo de la asignación puede ser cualquier nombre de variable válido. El bucle `for` asigna un elemento diferente de este vector a la variable de cada ejecución.

```
other_row_vector = [4, 3, 5, 1, 2];
for any_name = other_row_vector
    display(any_name)
end
```

La salida mostraría

```
4
3
5
1
2
```

(La versión `1:n` es un caso normal de esto, porque en Matlab `1:n` es solo una sintaxis para construir un vector de fila de `[1, 2, ..., n]` .)

Por lo tanto, los dos bloques de código siguientes son idénticos:

```
A = [1 2 3 4 5];
for x = A
  disp(x);
end
```

y

```
for x = 1:5
  disp(x);
end
```

Y los siguientes también son idénticos:

```
A = [1 3 5 7 9];
for x = A
  disp(x);
end
```

y

```
for x = 1:2:9
  disp(x);
end
```

Cualquier vector de fila servirá. No tienen que ser números.

```
my_characters = 'abcde';
for my_char = my_characters
    disp(my_char)
end
```

dará salida

```
a
b
c
d
e
```

Iterar sobre columnas de matriz.

Si el lado derecho de la asignación es una matriz, en cada iteración se asignan columnas subsiguientes a esta variable.

```
some_matrix = [1, 2, 3; 4, 5, 6]; % 2 by 3 matrix
for some_column = some_matrix
    display(some_column)
end
```

(La versión de vector de fila es un caso normal de esto, porque en Matlab un vector de fila es solo una matriz cuyas columnas son de tamaño 1.)

La salida mostraría

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

es decir, cada columna de la matriz iterada mostrada, cada columna impresa en cada llamada de `display` .

Loop sobre índices

```
my_vector = [0, 2, 1, 3, 9];
for i = 1:numel(my_vector)
    my_vector(i) = my_vector(i) + 1;
end
```

La mayoría de las cosas simples hechas con `for` bucles se puede hacer más rápido y más fácil por las operaciones vectorizadas. Por ejemplo, el bucle anterior se puede reemplazar por `my_vector = my_vector + 1` .

Bucles anidados

Los bucles se pueden anidar para realizar una tarea iterada dentro de otra tarea iterada. Considere los siguientes bucles:

```
ch = 'abc';
m = 3;
for c = ch
    for k = 1:m
        disp([c num2str(k)]) % NUM2STR converts the number stored in k to a charachter,
                             % so it can be concataneted with the letter in c
    end
end
```

usamos 2 iteradores para mostrar todas las combinaciones de elementos de `abc` y `1:m` , que

producen:

```
a1
a2
a3
b1
b2
b3
c1
c2
c3
```

También podemos usar bucles anidados para combinar entre las tareas que se realizarán cada vez y las tareas que se realizarán una vez en varias iteraciones:

```
N = 10;
n = 3;
a1 = 0; % the first element in Fibonacci series
a2 = 1; % the secound element in Fibonacci series
for j = 1:N
    for k = 1:n
        an = a1 + a2; % compute the next element in Fibonacci series
        a1 = a2;      % save the previous element for the next iteration
        a2 = an;      % save ht new element for the next iteration
    end
    disp(an) % display every n'th element
end
```

Aquí queremos calcular toda la serie de Fibonacci , pero para mostrar sólo el `n` -ésimo elemento cada vez, por lo que tenemos

```
   3
   13
   55
   233
   987
   4181
   17711
   75025
   317811
   1346269
```

Otra cosa que podemos hacer es usar el primer iterador (externo) dentro del bucle interno. Aquí hay otro ejemplo:

```
N = 12;
gap = [1 2 3 4 6];
for j = gap
    for k = 1:j:N
        fprintf('%d ',k) % FPRINTF prints the number k proceeding to the next the line
    end
    fprintf('\n')        % go to the next line
end
```

Esta vez usamos el bucle anidado para formatear la salida y frenar la línea solo cuando se

introdujo un nuevo espacio (`j`) entre los elementos. Recorremos el ancho de la brecha en el bucle externo y lo usamos dentro del bucle interno para iterar a través del vector:

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

Aviso: Extraños bucles anidados del mismo contador.

Esto no es algo que verá en otros entornos de programación. Lo encontré hace algunos años y no podía entender por qué estaba sucediendo, pero después de trabajar con MATLAB por un tiempo pude resolverlo. Mira el fragmento de código a continuación:

```
for x = 1:10
    for x = 1:10
        fprintf('%d,', x);
    end
    fprintf('\n');
end
```

no esperaría que esto funcione correctamente pero lo hace, produciendo el siguiente resultado:

```
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
1,2,3,4,5,6,7,8,9,10,
```

La razón es que, como con todo lo demás en MATLAB, el contador `x` también es una matriz, un vector para ser preciso. Como tal, `x` es solo una referencia a una 'matriz' (una estructura de memoria coherente y consecutiva) a la que se hace referencia apropiada con cada bucle consecuente (anidado o no). El hecho de que el bucle anidado use el mismo identificador no hace ninguna diferencia en la forma en que se hace referencia a los valores de esa matriz. El único problema es que dentro del bucle anidado, la `x` externa está oculta por la `x` anidada (local) y, por lo tanto, no se puede hacer referencia a ella. Sin embargo, la funcionalidad de la estructura de bucle anidado permanece intacta.

Lea Para bucles en línea: https://riptutorial.com/es/matlab/topic/1927/para-bucles

Capítulo 24: Procesamiento de imágenes

Examples

Imagen básica de E / S

```
>> img = imread('football.jpg');
```

Use `imread` para leer archivos de imagen en una matriz en MATLAB.
Una vez que haya `imread` una imagen, se almacena como una matriz ND en la memoria:

```
>> size(img)
ans =
 256   320     3
```

La imagen `'football.jpg'` tiene 256 filas y 320 columnas y tiene 3 canales de color: rojo, verde y azul.

Ahora puedes reflejarlo:

```
>> mirrored = img(:, end:-1:1, :);  %// like mirroring any ND-array in Matlab
```

Y finalmente, escríbalo como una imagen usando `imwrite` :

```
>> imwrite(mirrored, 'mirrored_football.jpg');
```

Recuperar imágenes de internet

Mientras tenga una conexión a Internet, puede leer imágenes desde un hipervínculo.

```
I=imread('https://cdn.sstatic.net/Sites/stackoverflow/company/img/logos/so/so-logo.png');
```

Filtrado utilizando un FFT 2D

Al igual que para las señales 1D, es posible filtrar imágenes aplicando una transformación de Fourier, multiplicando con un filtro en el dominio de la frecuencia y transformando nuevamente en el dominio del espacio. Aquí es cómo puede aplicar filtros de paso alto o bajo a una imagen con Matlab:

Deje que la `image` sea la original, sin filtrar. A continuación se muestra cómo calcular su FFT 2D:

```
ft = fftshift(fft2(image));
```

Ahora, para excluir una parte del espectro, es necesario establecer sus valores de píxel en 0. La frecuencia espacial contenida en la imagen original se asigna desde el centro a los bordes

(después de usar `fftshift`). Para excluir las bajas frecuencias, estableceremos el área circular central en 0.

Aquí se explica cómo generar una máscara binaria en forma de disco con radio `D` mediante la función incorporada:

```
[x y ~] = size(ft);
D = 20;
mask = fspecial('disk', D) == 0;
mask = imresize(padarray(mask, [floor((x/2)-D) floor((y/2)-D)], 1, 'both'), [x y]);
```

El enmascaramiento de la imagen del dominio de la frecuencia se puede hacer multiplicando la FFT puntualmente con la máscara binaria obtenida anteriormente:

```
masked_ft = ft .* mask;
```

Ahora, vamos a calcular la FFT inversa:

```
filtered_image = ifft2(ifftshift(masked_ft), 'symmetric');
```

Las frecuencias altas de una imagen son los bordes afilados, por lo que este filtro de paso alto se puede usar para enfocar imágenes.

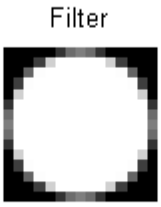
Filtrado de imágenes

Digamos que tiene una imagen `rgbImg` , por ejemplo, lea usando `imread` .

```
>> rgbImg = imread('pears.png');
>> figure, imshow(rgbImg), title('Original Image')
```

Use `fspecial` para crear un filtro 2D:

```
>> h = fspecial('disk', 7);
>> figure, imshow(h, []), title('Filter')
```



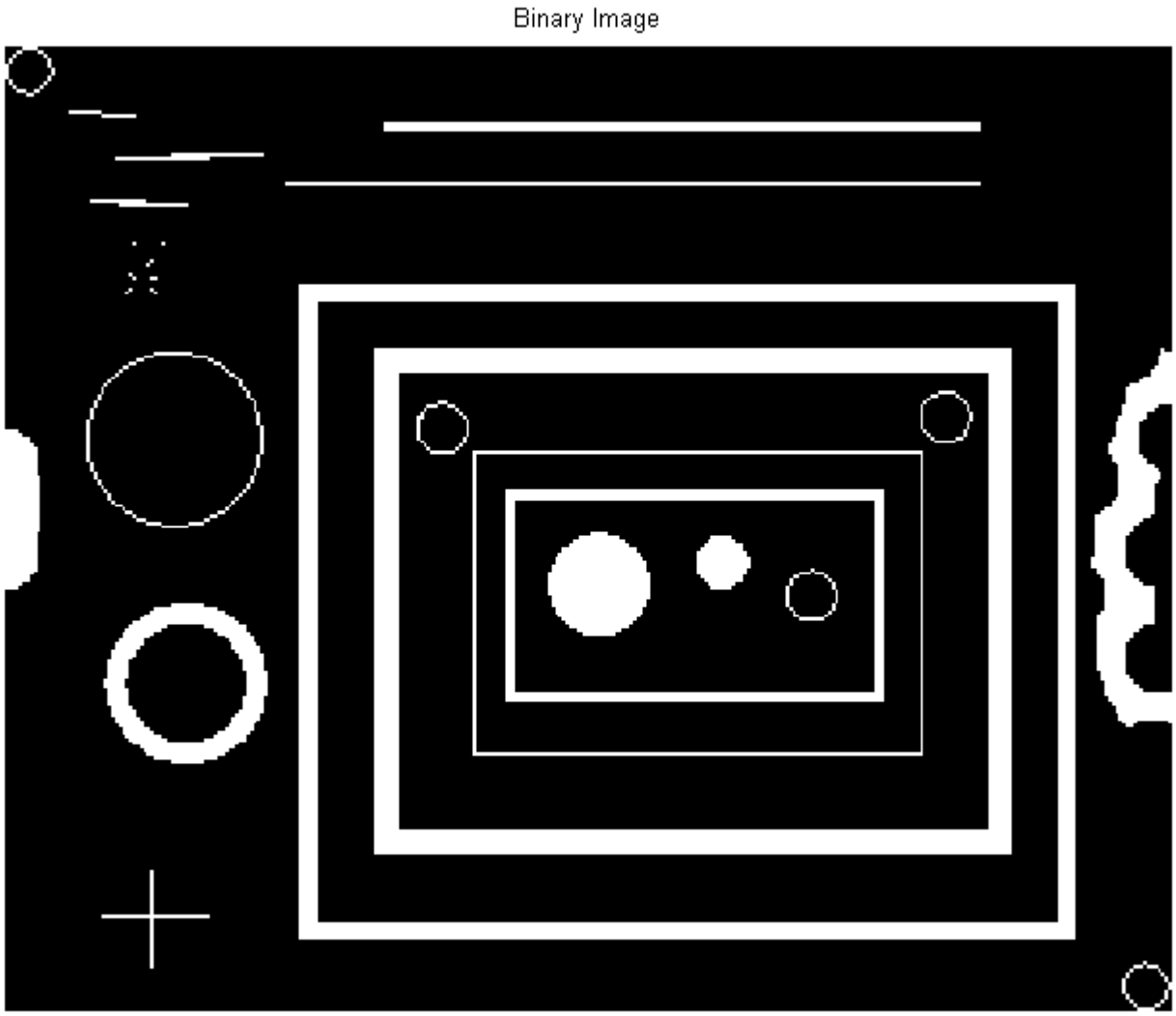
Use `imfilter` para aplicar el filtro en la imagen:

```
>> filteredRgbImg = imfilter(rgbImg, h);
>> figure, imshow(filteredRgbImg), title('Filtered Image')
```

Medición de las propiedades de las regiones conectadas

Comenzando con una imagen binaria, `bwImg` , que contiene varios objetos conectados.

```
>> bwImg = imread('blobs.png');
>> figure, imshow(bwImg), title('Binary Image')
```



Para medir las propiedades (por ejemplo, área, centroide, etc.) de cada objeto en la imagen, use `regionprops` :

```
>> stats = regionprops(bwImg, 'Area', 'Centroid');
```

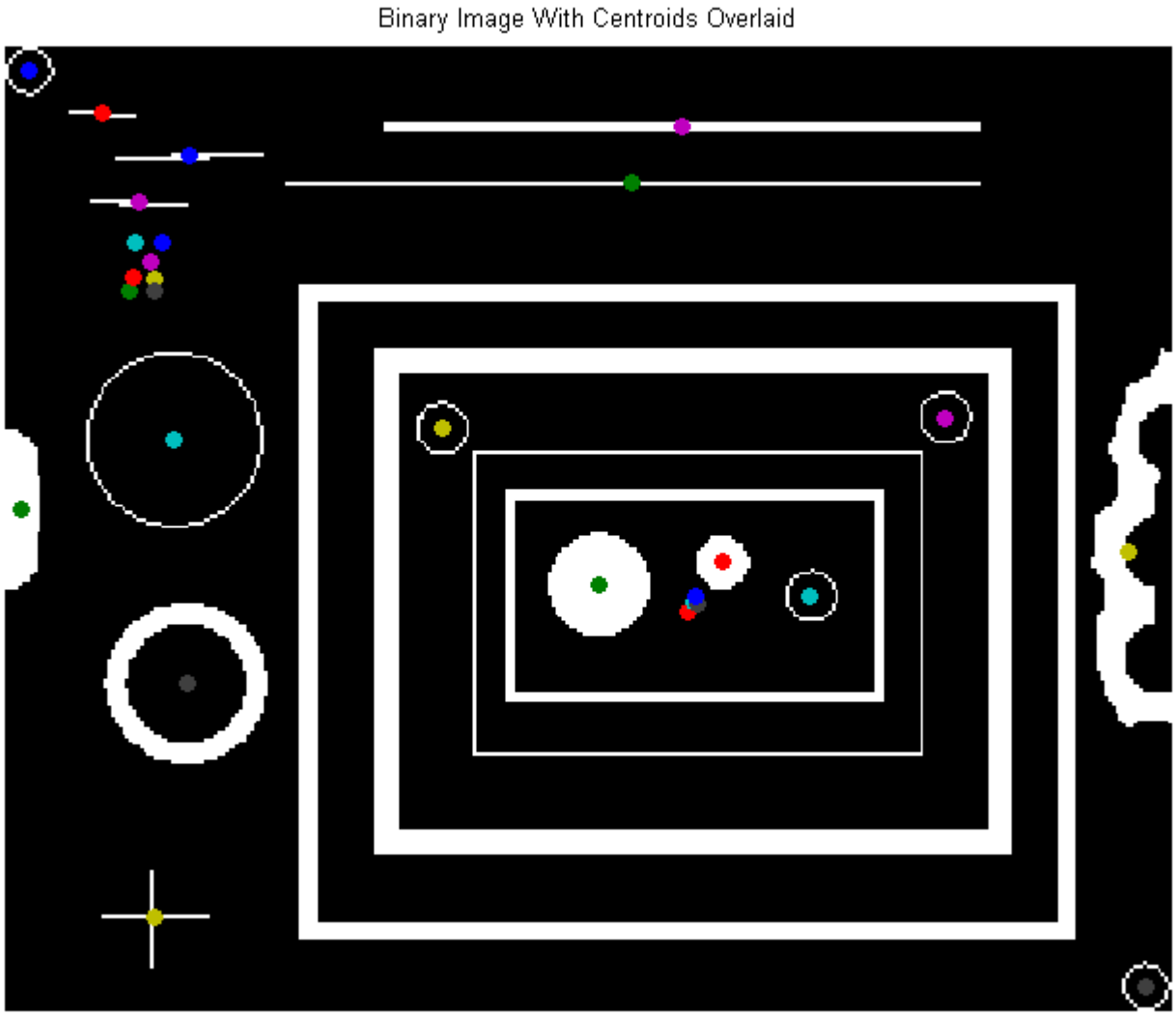
`stats` es una matriz de estructura que contiene una estructura para cada objeto en la imagen. Acceder a una propiedad medida de un objeto es simple. Por ejemplo, para mostrar el área del primer objeto, simplemente,

```
>> stats(1).Area

ans =

    35
```

Visualice los centroides del objeto superponiéndolos en la imagen original.

```
>> figure, imshow(bwImg), title('Binary Image With Centroids Overlaid')
>> hold on
>> for i = 1:size(stats)
scatter(stats(i).Centroid(1), stats(i).Centroid(2), 'filled');
```

```
end
```



Lea Procesamiento de imágenes en línea:
https://riptutorial.com/es/matlab/topic/3274/procesamiento-de-imagenes

Capítulo 25: Programación orientada a objetos

Examples

Definiendo una clase

Una clase se puede definir usando `classdef` en un archivo `.m` con el mismo nombre que la clase. El archivo puede contener la `classdef` ... `end` del bloque y funciones locales para su uso dentro de los métodos de clase.

La definición de clase MATLAB más general tiene la siguiente estructura:

```
classdef (ClassAttribute = expression, ...) ClassName < ParentClass1 & ParentClass2 & ...

   properties (PropertyAttributes)
      PropertyName
   end

   methods (MethodAttributes)
      function obj = methodName(obj,arg2,...)
         ...
      end
   end

   events (EventAttributes)
      EventName
   end

   enumeration
      EnumName
   end

end
```

Documentación de MATLAB: atributos de clase , atributos de propiedad , atributos de método , atributos de evento , restricciones de clase de enumeración .

Ejemplo de clase:

Una clase llamada `Car` puede definirse en el archivo `Car.m` como

```
classdef Car < handle % handle class so properties persist
    properties
        make
        model
        mileage = 0;
    end

    methods
        function obj = Car(make, model)
```

```
            obj.make = make;
            obj.model = model;
        end
        function drive(obj, milesDriven)
            obj.mileage = obj.mileage + milesDriven;
        end
    end
end
```

Tenga en cuenta que el constructor es un método con el mismo nombre que la clase. <Un constructor es un método especial de una clase o estructura en la programación orientada a objetos que inicializa un objeto de ese tipo. Un constructor es un método de instancia que generalmente tiene el mismo nombre que la clase y se puede usar para establecer los valores de los miembros de un objeto, ya sea por defecto o por valores definidos por el usuario.>

Se puede crear una instancia de esta clase llamando al constructor;

```
>> myCar = Car('Ford', 'Mustang'); //creating an instance of car class
```

Llamar al método de `drive` incrementará el kilometraje.

```
>> myCar.mileage

    ans =
             0

>> myCar.drive(450);

>> myCar.mileage

   ans =
             450
```

Clases de valor vs manejo

Las clases en MATLAB se dividen en dos categorías principales: clases de valor y clases de manejo. La principal diferencia es que al copiar una instancia de una clase de valor, los datos subyacentes se copian en la nueva instancia, mientras que para las clases de manejo la nueva instancia apunta a los datos originales y el cambio de valores en la nueva instancia los cambia en el original. Una clase se puede definir como un identificador heredando de la clase de `handle` .

```
classdef valueClass
    properties
        data
    end
end
```

y

```
classdef handleClass < handle
    properties
        data
```

```
    end
end
```

entonces

```
>> v1 = valueClass;
>> v1.data = 5;
>> v2 = v1;
>> v2.data = 7;
>> v1.data
ans =
     5

>> h1 = handleClass;
>> h1.data = 5;
>> h2 = h1;
>> h2.data = 7;
>> h1.data
ans =
     7
```

Herencia de clases y clases abstractas.

Descargo de responsabilidad: los ejemplos que se presentan aquí solo tienen el propósito de mostrar el uso de las clases abstractas y la herencia y no necesariamente tienen un uso práctico. Además, no hay ninguna cosa tan polimórfica en MATLAB y, por lo tanto, el uso de clases abstractas es limitado. Este ejemplo es para mostrar quién debe crear una clase, heredar de otra clase y aplicar una clase abstracta para definir una interfaz común.

El uso de clases abstractas es bastante limitado en MATLAB, pero aún puede ser útil en un par de ocasiones.

Digamos que queremos un registrador de mensajes. Podríamos crear una clase similar a la siguiente:

```
classdef ScreenLogger
    properties(Access=protected)
        scrh;
    end

    methods
        function obj = ScreenLogger(screenhandler)
            obj.scrh = screenhandler;
        end

        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                fprintf(obj.scrh, '%s\n', sprintf(varargin{:}));
            end
        end
    end
end
```

Propiedades y metodos

En resumen, las propiedades mantienen el estado de un objeto, mientras que los métodos son como una interfaz y definen acciones en los objetos.

La propiedad `scrh` está protegida. Es por esto que debe ser inicializado en un constructor. Hay otros métodos (captadores) para acceder a esta propiedad, pero no es posible en este ejemplo. Se puede acceder a las propiedades y los métodos a través de una variable que contiene una referencia a un objeto utilizando la notación de puntos seguida de un nombre de un método o una propiedad:

```
mylogger = ScreenLogger(1);                          % OK
mylogger.LogMessage('My %s %d message', 'very', 1); % OK
mylogger.scrh = 2;                                   % ERROR!!! Access denied
```

Las propiedades y los métodos pueden ser públicos, privados o protegidos. En este caso, protegido significa que podré acceder a `scrh` desde una clase heredada pero no desde afuera. Por defecto, todas las propiedades y métodos son públicos. Por lo tanto, `LogMessage()` se puede usar libremente fuera de la definición de clase. Además, `LogMessage` define una interfaz que significa que esto es lo que debemos llamar cuando queremos que un objeto registre nuestros mensajes personalizados.

Solicitud

Digamos que tengo un script donde utilizo mi registrador:

```
clc;
% ... a code
logger = ScreenLogger(1);
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
```

Si tengo varios lugares donde utilizo el mismo registrador y luego quiero cambiarlo por algo más sofisticado, como escribir un mensaje en un archivo, tendría que crear otro objeto:

```
classdef DeepLogger
    properties(SetAccess=protected)
        FileName
    end
    methods
        function obj = DeepLogger(filename)
            obj.FileName = filename;
        end

        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
```

```
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

y solo cambia una línea de un código en esto:

```
clc;
% ... a code
logger = DeepLogger('mymessages.log');
```

El método anterior simplemente abrirá un archivo, agregará un mensaje al final del archivo y lo cerrará. En este momento, para ser coherente con mi interfaz, debo recordar que el nombre de un método es `LogMessage()` pero que podría ser cualquier otra cosa. MATLAB puede forzar al desarrollador a mantener el mismo nombre usando clases abstractas. Digamos que definimos una interfaz común para cualquier registrador:

```
classdef MessageLogger
    methods(Abstract=true)
        LogMessage(obj, varargin);
    end
end
```

Ahora, si tanto `ScreenLogger` como `DeepLogger` heredan de esta clase, MATLAB generará un error si `LogMessage()` no está definido. Las clases abstractas ayudan a construir clases similares que pueden usar la misma interfaz.

Por el bien de este ejemplo, haré un cambio ligeramente diferente. Voy a asumir que DeepLogger hará el mensaje de registro en una pantalla y en un archivo al mismo tiempo. Como `ScreenLogger` ya registra los mensajes en la pantalla, heredaré `DeepLogger` de `ScreenLoggger` para evitar repeticiones. `ScreenLogger` no cambia en absoluto aparte de la primera línea:

```
classdef ScreenLogger < MessageLogger
// the rest of previous code
```

Sin embargo, `DeepLogger` necesita más cambios en el método `LogMessage` :

```
classdef DeepLogger < MessageLogger & ScreenLogger
    properties(SetAccess=protected)
        FileName
        Path
    end
    methods
        function obj = DeepLogger(screenhandler, filename)
            [path,filen,ext] = fileparts(filename);
            obj.FileName = [filen ext];
            pbj.Path     = pathn;
            obj = obj@ScreenLogger(screenhandler);
        end
        function LogMessage(obj, varargin)
```

```
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                LogMessage@ScreenLogger(obj, varargin{:});
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

En primer lugar, simplemente inicializo las propiedades en el constructor. En segundo lugar, debido a que esta clase se hereda de `ScreenLogger` , también tengo que inicializar este objeto original. Esta línea es aún más importante porque el constructor `ScreenLogger` requiere un parámetro para inicializar su propio objeto. Esta línea:

```
obj = obj@ScreenLogger(screenhandler);
```

simplemente dice "llame al constructor de ScreenLogger e initalize con un controlador de pantalla". Vale la pena señalar aquí que he definido `scrh` como protegido. Por lo tanto, también podría acceder a esta propiedad desde `DeepLogger` . Si la propiedad fue definida como privada. La única forma de inicializarlo sería utilizando un consejero.

Otro cambio está en los `methods` sección. De nuevo, para evitar la repetición, llamo a `LogMessage()` desde una clase primaria para registrar un mensaje en una pantalla. Si tuviera que cambiar algo para realizar mejoras en el registro de pantalla, ahora tengo que hacerlo en un solo lugar. El código de resto es el mismo que forma parte de `DeepLogger` .

Debido a que esta clase también se hereda de una clase abstracta `MessageLogger` , tuve que asegurarme de que `LogMessage()` dentro de `DeepLogger` también esté definido. Heredar de `MessageLogger` es un poco complicado aquí. Creo que es obligatorio redefinir `LogMessage` , supongo.

En términos del código donde se aplica un registrador, gracias a una interfaz común en las clases, puedo estar seguro de que el cambio de esta línea en todo el código no generará ningún problema. Los mismos mensajes se registrarán en la pantalla que antes, pero además el código escribirá dichos mensajes en un archivo.

```
clc;
% ... a code
logger = DeepLogger(1, 'mylogfile.log');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
% ... a code
logger.LogMessage('something');
```

Espero que estos ejemplos expliquen el uso de las clases, el uso de la herencia y el uso de las clases abstractas.

PD. La solución para el problema anterior es una de muchas. Otra solución, menos compleja, sería hacer que `ScreenLoger` sea un componente de otro registrador como `FileLogger` etc. `ScreenLogger` se mantendría en una de las propiedades. Su `LogMessage` simplemente llamaría `LogMessage` del `ScreenLogger` y mostraría el texto en una pantalla. He elegido un enfoque más complejo para mostrar cómo funcionan las clases en MATLAB. El código de ejemplo a continuación:

```
classdef DeepLogger < MessageLogger
    properties(SetAccess=protected)
        FileName
        Path
        ScrLogger
    end
    methods
        function obj = DeepLogger(screenhandler, filename)
            [path,filen,ext] = fileparts(filename);
            obj.FileName     = [filen ext];
            obj.Path         = pathn;
            obj.ScrLogger    = ScreenLogger(screenhandler);
        end
        function LogMessage(obj, varargin)
            if ~isempty(varargin)
                varargin{1} = num2str(varargin{1});
                obj.LogMessage(obj.ScrLogger, varargin{:}); % <-------- thechange here
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

Constructores

Un constructor es un método especial en una clase que se llama cuando se crea una instancia de un objeto. Es una función MATLAB normal que acepta parámetros de entrada, pero también debe seguir ciertas reglas .

No se requieren constructores ya que MATLAB crea uno predeterminado. En la práctica, sin embargo, este es un lugar para definir un estado de un objeto. Por ejemplo, las propiedades se pueden restringir especificando atributos . Luego, un constructor puede inicializar dichas propiedades por defecto o valores definidos por el usuario que, de hecho, pueden enviarse por los parámetros de entrada de un constructor.

Llamando a un constructor de una clase simple.

Esta es una simple clase de `Person` .

```
classdef Person
    properties
        name
        surname
        address
    end
```

```
    methods
        function obj = Person(name,surname,address)
            obj.name = name;
            obj.surname = surname;
            obj.address = address;
        end
    end
end
```

El nombre de un constructor es el mismo que el nombre de una clase. En consecuencia, los constructores son llamados por el nombre de su clase. Una clase de `Person` puede ser creada de la siguiente manera:

```
>> p = Person('John','Smith','London')
p =
  Person with properties:

       name: 'John'
    surname: 'Smith'
    address: 'London'
```

Llamando a un constructor de una clase infantil

Las clases se pueden heredar de las clases primarias si comparten propiedades o métodos comunes. Cuando una clase se hereda de otra, es probable que se deba llamar a un constructor de una clase primaria.

Un `Member` clase hereda de una `Person` clase porque el `Member` usa las mismas propiedades que la persona de la clase pero también agrega el `payment` a su definición.

```
classdef Member < Person
    properties
        payment
    end

    methods
        function obj = Member(name,surname,address,payment)
            obj = obj@Person(name,surname,address);
            obj.payment = payment;
        end
    end
end
```

De manera similar a la clase `Person` , el `Member` se crea llamando a su constructor:

```
>> m = Member('Adam','Woodcock','Manchester',20)
m =
  Member with properties:

    payment: 20
       name: 'Adam'
    surname: 'Woodcock'
    address: 'Manchester'
```

Un constructor de `Person` requiere tres parámetros de entrada. `Member` debe respetar este hecho y, por lo tanto, llamar a un constructor de la clase `Person` con tres parámetros. Se cumple por la línea:

```
obj = obj@Person(name,surname,address);
```

El ejemplo anterior muestra el caso cuando una clase secundaria necesita información para su clase principal. Por esta razón, un constructor de `Member` requiere cuatro parámetros: tres para su clase principal y uno para sí mismo.

Lea Programación orientada a objetos en línea:
https://riptutorial.com/es/matlab/topic/1028/programacion-orientada-a-objetos

Capítulo 26: Rendimiento y Benchmarking

Observaciones

- El código de perfil es una forma de evitar la temida práctica de la " optimización prematura ", al enfocar al desarrollador en aquellas partes del código que *realmente* justifican los esfuerzos de optimización.
- Artículo de documentación de MATLAB titulado " Medir el rendimiento de su programa ".

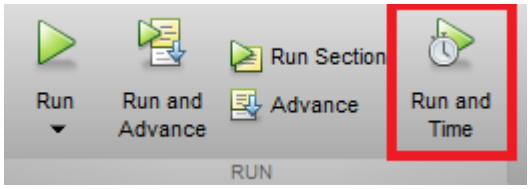
Examples

Identificar cuellos de botella de rendimiento utilizando el Perfilador

El perfilador de MATLAB es una herramienta para la creación de perfiles de software de código MATLAB. Usando el Perfilador, es posible obtener una representación visual del tiempo de ejecución y del consumo de memoria.

Ejecutar el Perfilador se puede hacer de dos maneras:

- Al hacer clic en el botón "Ejecutar y tiempo" en la GUI de MATLAB mientras se abre algún archivo `.m` en el editor (agregado en **R2012b**).

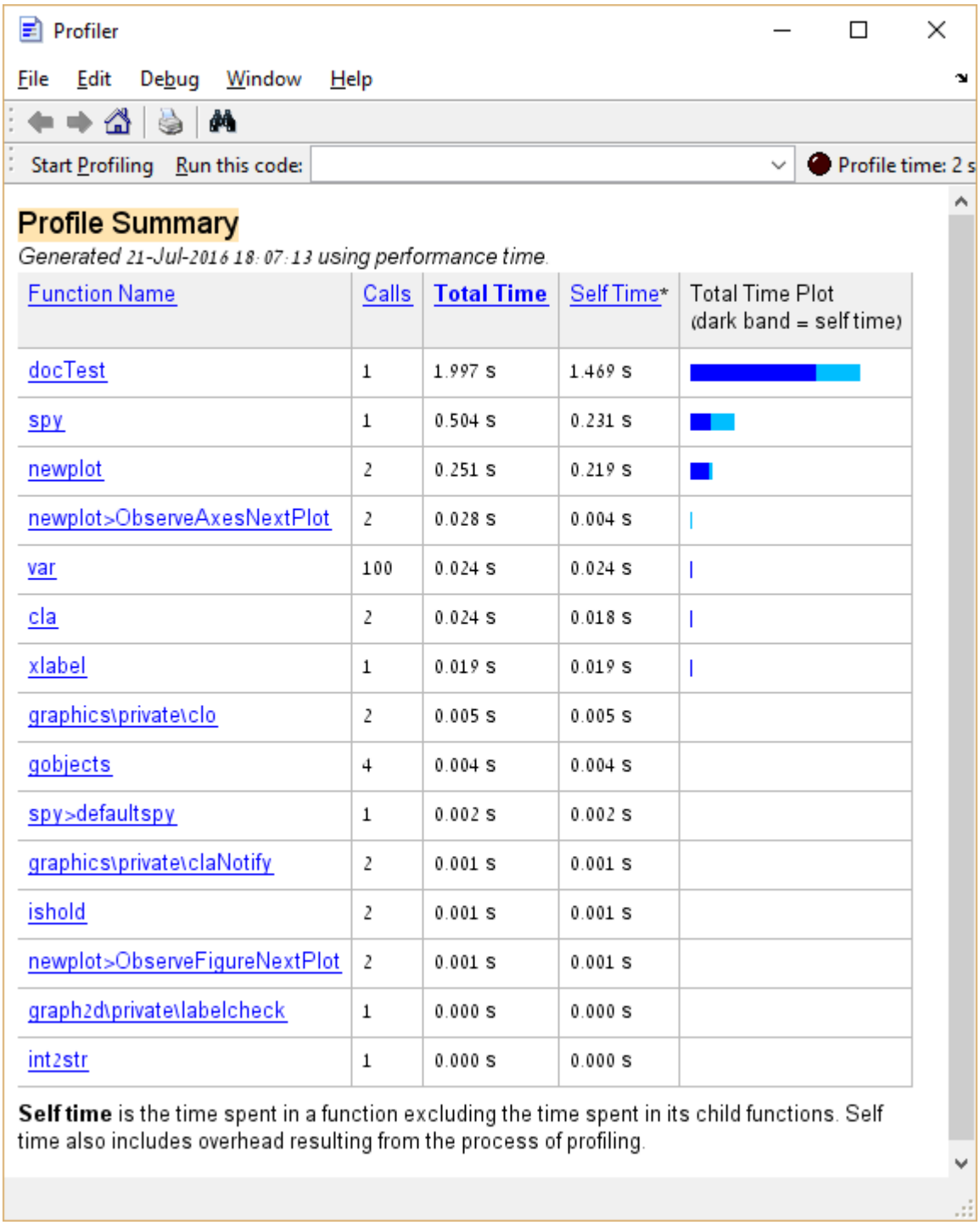


- Programáticamente, utilizando:

```
profile on
<some code we want to test>
profile off
```

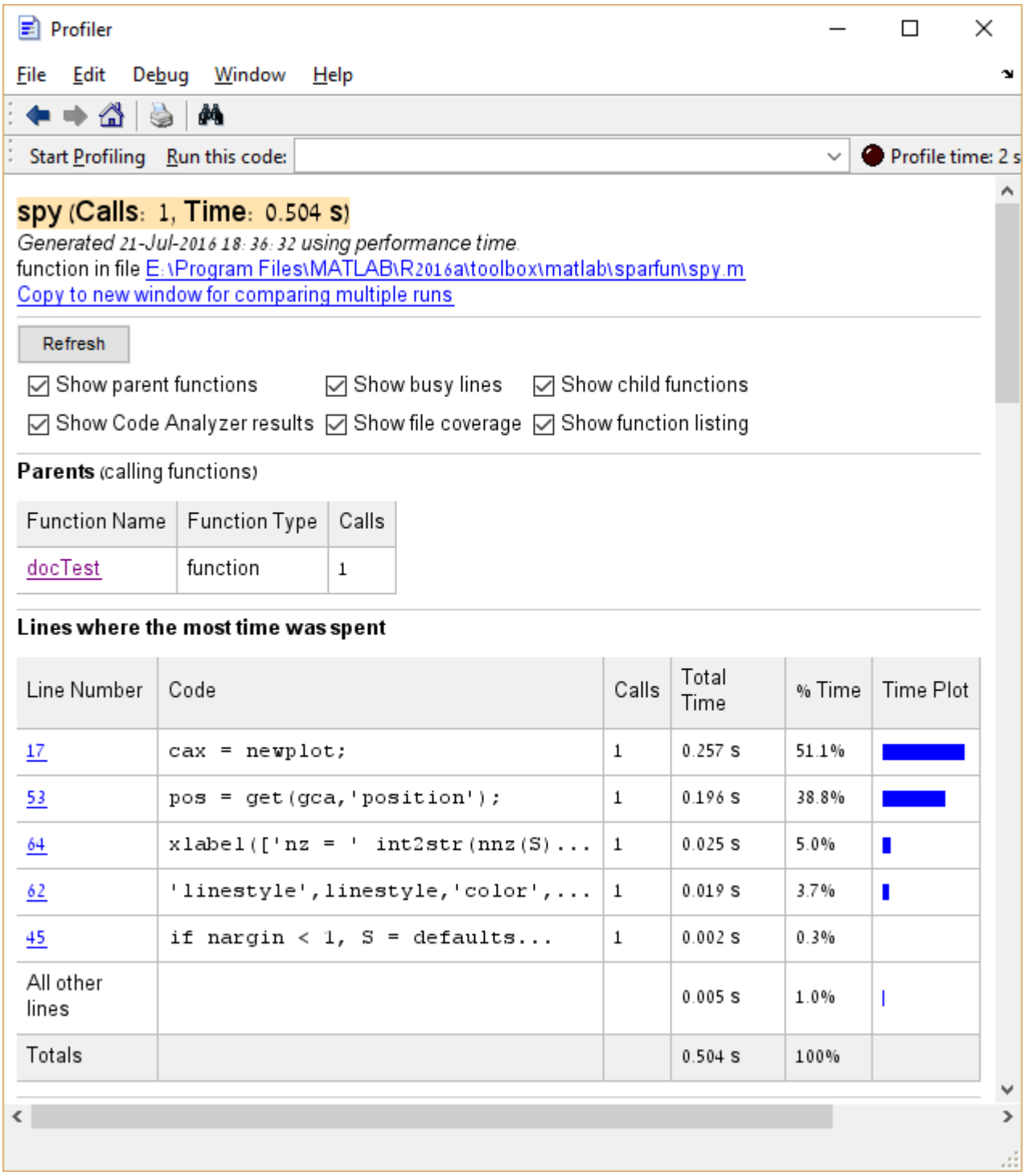
A continuación se muestra un código de ejemplo y el resultado de su perfil:

```
function docTest

for ind1 = 1:100
  [~] = var(...
             sum(...
                 randn(1000)));
end

spy
```

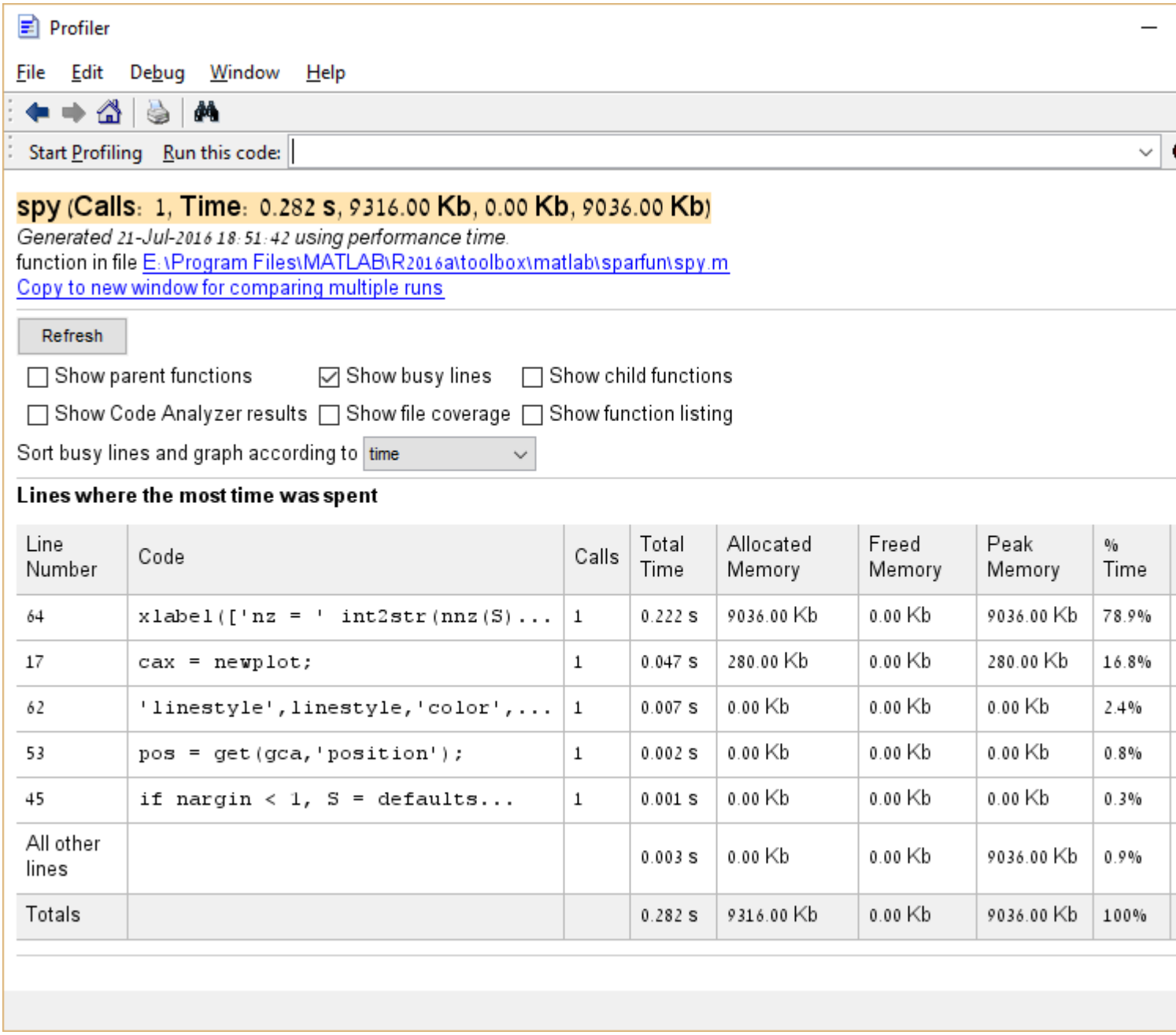


De lo anterior, aprendemos que la función de `spy` ocupa aproximadamente el 25% del tiempo total de ejecución. En el caso de "código real", una función que tome un porcentaje tan grande de tiempo de ejecución sería un buen candidato para la optimización, en lugar de funciones análogas a `var` y `cla` cuya optimización debería evitarse.

Además, es posible hacer clic en las entradas en la columna *Nombre de* la *función* para ver un desglose detallado del tiempo de ejecución de esa entrada. Aquí está el ejemplo de hacer clic `spy` :



También es posible perfilar el consumo de memoria ejecutando el `profile('-memory')` antes de ejecutar el Perfilador.



Line Number	Code	Calls	Total Time	Allocated Memory	Freed Memory	Peak Memory	% Time
64	`xlabel(['nz = ' int2str(nnz(S)...`	1	0.222 s	9036.00 Kb	0.00 Kb	9036.00 Kb	78.9%
17	`cax = newplot;`	1	0.047 s	280.00 Kb	0.00 Kb	280.00 Kb	16.8%
62	`'linestyle',linestyle,'color',...`	1	0.007 s	0.00 Kb	0.00 Kb	0.00 Kb	2.4%
53	`pos = get(gca,'position');`	1	0.002 s	0.00 Kb	0.00 Kb	0.00 Kb	0.8%
45	`if nargin < 1, S = defaults...`	1	0.001 s	0.00 Kb	0.00 Kb	0.00 Kb	0.3%
All other lines			0.003 s	0.00 Kb	0.00 Kb	9036.00 Kb	0.9%
Totals			0.282 s	9316.00 Kb	0.00 Kb	9036.00 Kb	100%

Comparando el tiempo de ejecución de múltiples funciones

La combinación ampliamente utilizada de `tic` y `toc` puede proporcionar una idea aproximada del tiempo de ejecución de una función o fragmentos de código.

Para comparar varias funciones no debe utilizarse. ¿Por qué? Es casi imposible proporcionar las *mismas condiciones* para que todos los fragmentos de código se comparen dentro de un script utilizando la solución anterior. Tal vez las funciones comparten el mismo espacio de funciones y variables comunes, por lo que más adelante las funciones denominadas fragmentos de código ya aprovechan las variables y funciones previamente iniciadas. Además, no hay ninguna idea de si el compilador JIT manejaría estos fragmentos posteriormente llamados igualmente.

La función dedicada para los puntos de referencia es el `timeit` . El siguiente ejemplo ilustra su uso.

Están la matriz `A` y la matriz `B` Se debe determinar qué fila de `B` es la más similar a `A` contando el número de elementos diferentes.

```
function t = bench()
    A = [0 1 1 1 0 0];
    B = perms(A);

    % functions to compare
    fcns = {
        @() compare1(A,B);
        @() compare2(A,B);
        @() compare3(A,B);
        @() compare4(A,B);
    };

    % timeit
    t = cellfun(@timeit, fcns);
end

function Z = compare1(A,B)
    Z = sum(  bsxfun(@eq,  A,B) , 2);
end
function Z = compare2(A,B)
    Z = sum(bsxfun(@xor, A, B),2);
end
function Z = compare3(A,B)
    A = logical(A);
    Z = sum(B(:,~A),2) + sum(~B(:,A),2);
end
function Z = compare4(A,B)
     Z = pdist2( A, B, 'hamming', 'Smallest', 1 );
end
```

Esta forma de referencia fue vista por primera vez en esta respuesta .

¡Está bien ser 'soltero'!

Visión general:

El tipo de datos predeterminado para matrices numéricas en MATLAB es `double` . `double` es una representación de números en coma flotante , y este formato toma 8 bytes (o 64 bits) por valor. En **algunos** casos, donde, por ejemplo, tratar solo con números enteros o cuando la inestabilidad numérica no es un problema inminente, tal profundidad de bit alta puede no ser necesaria. Por este motivo, se recomienda considerar los beneficios de la precisión `single` (u otros tipos apropiados):

- Tiempo de ejecución más rápido (especialmente notable en las GPU).
- La mitad del consumo de memoria: puede tener éxito cuando el `double` falla debido a un error de falta de memoria; Más compacto cuando se almacena como archivos.

La conversión de una variable de cualquier tipo de datos compatible a uno `single` se realiza mediante:

```
sing_var = single(var);
```

Algunas funciones de uso común (como `zeros` , `eye` , `ones` , etc.) que emiten valores `double` de forma predeterminada, permiten especificar el tipo / clase de la salida.

Conversión de variables en una secuencia de comandos a una precisión / tipo / clase no predeterminada:

A partir de julio de 2016, no existe una forma documentada de cambiar el tipo de datos MATLAB *predeterminado* de `double` .

En MATLAB, las nuevas variables generalmente imitan los tipos de datos de las variables utilizadas al crearlas. Para ilustrar esto, considere el siguiente ejemplo:

```
A = magic(3);
B = diag(A);
C = 20*B;
>> whos C
  Name      Size            Bytes  Class     Attributes
  C         3x1                24  double
```

```
A = single(magic(3)); % A is converted to "single"
B = diag(A);
C = B*double(20);     % The stricter type, which in this case is "single", prevails
D = single(size(C));  % It is generally advised to cast to the desired type explicitly.
>> whos C
  Name      Size            Bytes  Class     Attributes
  C         3x1                12  single
```

Por lo tanto, puede parecer suficiente convertir / convertir varias variables iniciales para que el cambio esté permeado en todo el código, sin embargo, esto no es **recomendable** (consulte *Advertencias y errores a* continuación).

Advertencias y escollos:

1. Las conversiones repetidas se **desaconsejan** debido a la introducción de ruido numérico (cuando se convierte de `single` a `double`) o la pérdida de información (cuando se convierte de `double` a `single` , o entre ciertos tipos de enteros), por ejemplo:

   ```
   double(single(1.2)) == double(1.2)
   ans =
        0
   ```

   Esto puede ser mitigado un poco usando `typecast` . Consulte también Tenga en cuenta la

inexactitud del punto flotante .

2. Se **desaconseja** confiar únicamente en la tipificación de datos implícita (es decir, lo que MATLAB adivina sobre el tipo de salida de un cálculo) se **desaconseja** debido a varios efectos no deseados que pueden surgir:

   - *Pérdida de información* : cuando se espera un resultado `double` , pero una combinación descuidada de operandos `single` y `double` produce `single` precisión `single` .
   - *Consumo de memoria inesperadamente alto* : cuando se espera un `single` resultado pero un cálculo descuidado da como resultado una salida `double` .
   - *Gastos generales innecesarios cuando se trabaja con GPU* : cuando se mezclan los tipos `gpuArray` (es decir, las variables almacenadas en VRAM) con variables que no son `gpuArray` (es decir, las que *normalmente se* almacenan en la RAM), los datos deberán transferirse de una forma u otra antes de poder realizar el cálculo. Esta operación lleva tiempo y puede ser muy notable en los cálculos repetitivos.
   - *Errores al mezclar tipos de punto flotante con tipos enteros* : las funciones como `mtimes` (`*`) no están definidas para entradas mixtas de tipos enteros y de punto flotante, y aparecerán errores. Las funciones como `times` (`.*`) No están definidas en absoluto para entradas de tipo entero, y volverán a generar un error.

     ```
     >> ones(3,3,'int32')*ones(3,3,'int32')
     Error using  *
     MTIMES is not fully supported for integer classes. At least one input must be
     scalar.

     >> ones(3,3,'int32').*ones(3,3,'double')
     Error using  .*
     Integers can only be combined with integers of the same class, or scalar doubles.
     ```

   Para una mejor legibilidad del código y un menor riesgo de tipos no deseados, se **recomienda** un enfoque defensivo, donde las variables se *convierten explícitamente* al tipo deseado.

Ver también:

- Documentación MATLAB: Números de punto flotante .
- Artículo técnico de Mathworks: Mejores prácticas para convertir código MATLAB a punto fijo .

reorganizar una matriz ND puede mejorar el rendimiento general

En algunos casos, debemos aplicar funciones a un conjunto de matrices ND. Veamos este sencillo ejemplo.

```
A(:,:,1) = [1 2; 4 5];
A(:,:,2) = [11 22; 44 55];
```

```
B(:,:,1) = [7 8; 1 2];
B(:,:,2) = [77 88; 11 22];

A =

ans(:,:,1) =

   1   2
   4   5

ans(:,:,2) =

   11   22
   44   55

>> B
B =

ans(:,:,1) =

   7   8
   1   2

ans(:,:,2) =

   77   88
   11   22
```

Ambas matrices son 3D, digamos que tenemos que calcular lo siguiente:

```
result= zeros(2,2);
...
for k = 1:2
   result(i,j) = result(i,j) + abs( A(i,j,k) - B(i,j,k) );
...

if k is very large, this for-loop can be a bottleneck since MATLAB order the data in a column
major fashion. So a better way to compute "result" could be:

% trying to exploit the column major ordering
Aprime = reshape(permute(A,[3,1,2]), [2,4]);
Bprime = reshape(permute(B,[3,1,2]), [2,4]);


>> Aprime
Aprime =

    1    4    2    5
   11   44   22   55

>> Bprime
Bprime =

    7    1    8    2
   77   11   88   22
```

Ahora reemplazamos el bucle anterior por lo siguiente:

```
result= zeros(2,2);
```

```
....
temp = abs(Aprime - Bprime);
for k = 1:2
    result(i,j) = result(i,j) + temp(k, i+2*(j-1));
...
```

Reorganizamos los datos para poder explotar la memoria caché. La permutación y la reforma pueden ser costosas, pero cuando se trabaja con grandes matrices ND, el costo computacional relacionado con estas operaciones es mucho más bajo que trabajar con matrices no organizadas.

La importancia de la preasignación.

Las matrices en MATLAB se mantienen como bloques continuos en memoria, asignados y liberados automáticamente por MATLAB. MATLAB oculta las operaciones de administración de memoria, como el cambio de tamaño de una matriz con una sintaxis fácil de usar:

```
a = 1:4

a =

     1     2     3     4

a(5) = 10  % or alternatively a = [a, 10]

a =

     1     2     3     4    10
```

Es importante entender que lo anterior no es una operación trivial, `a(5) = 10` hará que MATLAB asigne un nuevo bloque de memoria de tamaño 5, copie los primeros 4 números y establezca el 5'th a 10. Es una `O(numel(a))` , y no `O(1)` .

Considera lo siguiente:

```
clear all
n=12345678;
a=0;
tic
for i = 2:n
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 3.004213 seconds.
```

`a` se reasigna `n` veces en este bucle (excluyendo algunas optimizaciones realizadas por MATLAB)! Tenga en cuenta que MATLAB nos da una advertencia:

> "La variable 'a' parece cambiar de tamaño en cada iteración de bucle. Considere la posibilidad de preasignar para la velocidad".

¿Qué pasa cuando nos preasignamos?

```
a=zeros(1,n);
tic
for i = 2:n
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 0.410531 seconds.
```

Podemos ver que el tiempo de ejecución se reduce en un orden de magnitud.

Métodos de preasignación:

MATLAB proporciona varias funciones para la asignación de vectores y matrices, dependiendo de los requisitos específicos del usuario. Estos incluyen: `zeros` , `ones` , `nan` , `eye` , `true` , etc.

```
a = zeros(3)       % Allocates a 3-by-3 matrix initialized to 0
a =

     0     0     0
     0     0     0
     0     0     0

a = zeros(3, 2)     % Allocates a 3-by-2 matrix initialized to 0
a =

     0     0
     0     0
     0     0

a = ones(2, 3, 2)      % Allocates a 3 dimensional array (2-by-3-by-2) initialized to 1
a(:,:,1) =

     1     1     1
     1     1     1


a(:,:,2) =

     1     1     1
     1     1     1

a = ones(1, 3) * 7  % Allocates a row vector of length 3 initialized to 7
a =

     7     7     7
```

También se puede especificar un tipo de datos:

```
a = zeros(2, 1, 'uint8');  % allocates an array of type uint8
```

También es fácil clonar el tamaño de una matriz existente:

```
a = ones(3, 4);       % a is a 3-by-4 matrix of 1's
b = zeros(size(a));  % b is a 3-by-4 matrix of 0's
```

Y clonar el tipo:

```
a = ones(3, 4, 'single');       % a is a 3-by-4 matrix of type single
b = zeros(2, 'like', a);        % b is a 2-by-2 matrix of type single
```

tenga en cuenta que 'me gusta' también clona la *complejidad* y la *escasez* .

La preasignación se logra implícitamente mediante cualquier función que devuelva una matriz del tamaño final requerido, como `rand` , `gallery` , `kron` , `bsxfun` , `colon` y muchos otros. Por ejemplo, una forma común de asignar vectores con elementos que varían linealmente es mediante el uso del operador de dos puntos (con la variante [1] o 2 del operando [3]):

```
a = 1:3
a =

     1     2     3

a = 2:-3:-4
a =

     2    -1    -4
```

Los arreglos de celdas pueden asignarse usando la función `cell()` de la misma manera que los `zeros()` .

```
a = cell(2,3)
a =

    []    []    []
    []    []    []
```

Tenga en cuenta que las matrices de celdas funcionan al mantener los punteros en las ubicaciones en la memoria del contenido de la celda. Por lo tanto, todas las sugerencias de preasignación se aplican también a los elementos de la matriz de celdas individuales.

Otras lecturas:

- Documentación oficial de MATLAB sobre " **Memoria de preasignación** ".
- Documentación oficial de MATLAB sobre " **Cómo MATLAB asigna memoria** ".
- Rendimiento de preasignación en **matlab sin documentar** .
- Entendiendo la ubicación previa de la matriz en **Loren sobre el arte de MATLAB**

Lea Rendimiento y Benchmarking en línea:
https://riptutorial.com/es/matlab/topic/1141/rendimiento-y-benchmarking

Capítulo 27: Solucionadores de Ecuaciones Diferenciales Ordinarias (EDO)

Examples

Ejemplo para odeset

Primero inicializamos nuestro problema de valor inicial que queremos resolver.

```
odefun = @(t,y) cos(y).^2*sin(t);
tspan = [0 16*pi];
y0=1;
```

Luego usamos la función ode45 sin ninguna opción específica para resolver este problema. Para compararlo después trazamos la trayectoria.

```
[t,y] = ode45(odefun, tspan, y0);
plot(t,y,'-o');
```

Ahora establecemos un pariente estrecho y un límite estrecho absoluto de tolerancia para nuestro problema.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Puse un límite estricto relativo y estrecho de tolerancia absoluto.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Establecemos estrecho límite relativo y estricto absoluto de tolerancia. Como en los ejemplos anteriores con límites estrechos de tolerancia, se observa que la trayectoria es completamente diferente de la primera gráfica sin ninguna opción específica.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Puse un límite estricto relativo y estricto de tolerancia. La comparación del resultado con la otra gráfica subrayará los errores cometidos al calcular con límites de tolerancia estrechos.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

Lo siguiente debe demostrar el compromiso entre la precisión y el tiempo de ejecución.

```
tic;
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
time1 = toc;
plot(t,y,'-o');
```

Para comparación, ajustamos el límite de tolerancia para el error absoluto y relativo. Ahora podemos ver que, sin una gran ganancia de precisión, tomará mucho más tiempo resolver nuestro problema de valor inicial.

```
tic;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
[t,y] = ode45(odefun, tspan, y0, options);
time2 = toc;
plot(t,y,'-o');
```

Lea Solucionadores de Ecuaciones Diferenciales Ordinarias (EDO) en línea: https://riptutorial.com/es/matlab/topic/6079/solucionadores-de-ecuaciones-diferenciales-ordinarias--edo-

Capítulo 28: Transformadas de Fourier y Transformadas de Fourier inversas

Sintaxis

1. Y = fft (X)% calcula la FFT de Vector o Matriz X utilizando una Longitud de transformación predeterminada de 256 (que se confirmará para la versión)
2. Y = fft (X, n)% calcula la FFT de X utilizando n como Longitud de transformación, n debe ser un número basado en 2 potencias. Si la longitud de X es menor que n, entonces Matlab rellenará automáticamente X con ceros de manera que la longitud (X) = n
3. Y = fft (X, n, dim)% calcula la FFT de X usando n como Longitud de transformación a lo largo de la dimensión dim (puede ser 1 o 2 para horizontal o vertical respectivamente)
4. Y = fft2 (X)% Calcule la FFT 2D de X
5. Y = fftn (X, dim)% Calcule la FFT dimidimensional de X, con respecto al vector de dimensiones dim.
6. y = ifft (X)% calcula la Inversa de FFT de X (que es una matriz / vector de números) utilizando la longitud de transformación predeterminada de 256
7. y = ifft (X, n)% calcula el IFFT de X utilizando n como Longitud de transformación
8. y = ifft (X, n, dim)% calcula el IFFT de X utilizando n como Longitud de transformación sobre dimensión dim (puede ser 1 o 2 para horizontal o vertical respectivamente)
9. y = ifft (X, n, dim, 'simétrico')% La opción Simétrico hace que ifft trate X como conjugado simétrico a lo largo de la dimensión activa. Esta opción es útil cuando X no es exactamente simétrico conjugado, simplemente debido a un error de redondeo.
10. y = ifft2 (X)% Calcule los pies 2D inversos de X
11. y = ifftn (X, dim)% Calcule el fft dim-dimensional inverso de X.

Parámetros

Parámetro	Descripción
X	Esta es su señal de entrada de dominio de tiempo, debería ser un vector de números.
norte	este es el parámetro NFFT conocido como Longitud de transformación, piense en él como en la resolución de su resultado FFT, DEBE ser un número que sea una potencia de 2 (es decir, 64,128,256 ... 2 ^ N)

Parámetro	Descripción
oscuro	esta es la dimensión en la que desea calcular FFT, use 1 si desea calcular su FFT en la dirección horizontal y 2 si desea calcular su FFT en la dirección vertical. Tenga en cuenta que este parámetro generalmente se deja en blanco, ya que la función es Capaz de detectar la dirección de tu vector.

Observaciones

Matlab FFT es un proceso muy paralelizado capaz de manejar grandes cantidades de datos. También puede utilizar la GPU con gran ventaja.

```
ifft(fft(X)) = X
```

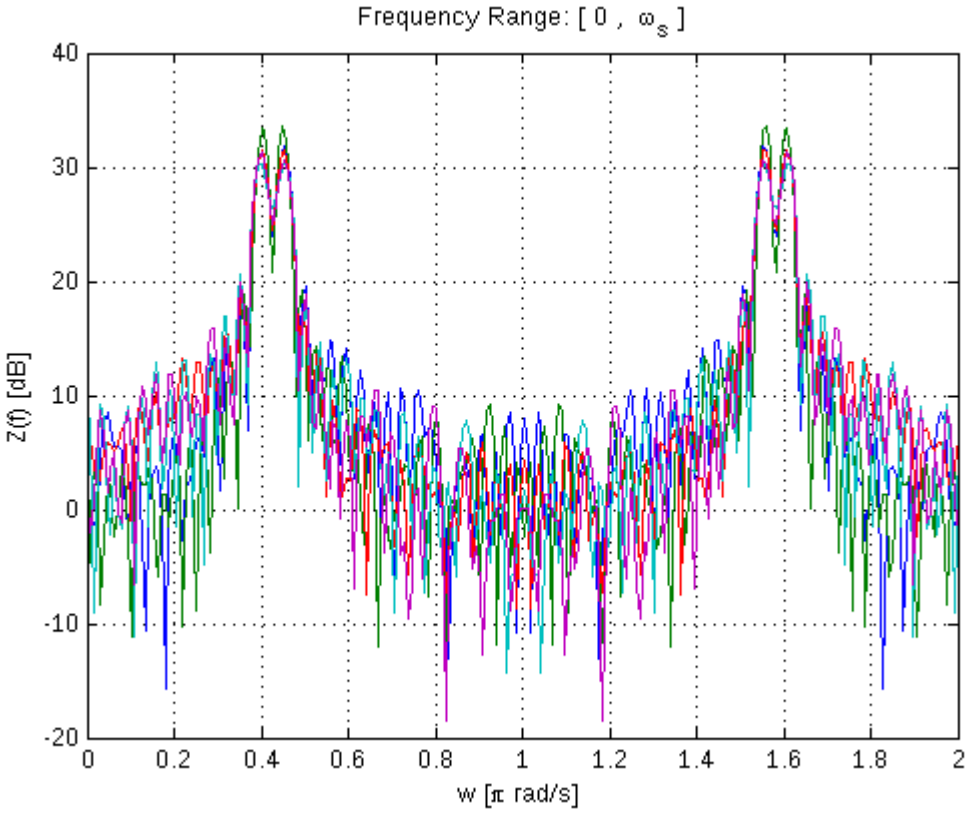
La declaración anterior es verdadera si se omiten los errores de redondeo.

Examples

Implementar una transformada de Fourier simple en Matlab

La Transformada de Fourier es probablemente la primera lección en Procesamiento de Señal Digital, su aplicación está en todas partes y es una herramienta poderosa cuando se trata de analizar datos (en todos los sectores) o señales. Matlab tiene un conjunto de poderosas cajas de herramientas para la Transformada de Fourier. En este ejemplo, usaremos la Transformada de Fourier para analizar una señal de onda sinusoidal básica y generar lo que a veces se conoce como Periodograma utilizando FFT:

```
%Signal Generation
A1=10;                % Amplitude 1
A2=10;                % Amplitude 2
w1=2*pi*0.2;          % Angular frequency 1
w2=2*pi*0.225;        % Angular frequency 2
Ts=1;                 % Sampling time
N=64;                 % Number of process samples to be generated
K=5;                  % Number of independent process realizations
sgm=1;                % Standard deviation of the noise
n=repmat([0:N-1].',1,K);             % Generate resolution
phi1=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 1
phi2=repmat(rand(1,K)*2*pi,N,1);     % Random phase matrix 2
x=A1*sin(w1*n*Ts+phi1)+A2*sin(w2*n*Ts+phi2)+sgm*randn(N,K);   % Resulting Signal

NFFT=256;            % FFT length
F=fft(x,NFFT);       % Fast Fourier Transform Result
Z=1/N*abs(F).^2;     % Convert FFT result into a Periodogram
```



Tenga en cuenta que la Transformada Discreta de Fourier se implementa mediante la Transformada Rápida de Fourier (fft) en Matlab, ambos producirán el mismo resultado, pero FFT es una implementación rápida de DFT.

```
figure
w=linspace(0,2,NFFT);
plot(w,10*log10(Z)),grid;
xlabel('w [\pi rad/s]')
ylabel('Z(f) [dB]')
title('Frequency Range: [ 0 ,  \omega_s ]')
```

Transformadas de Fourier inversas

Uno de los principales beneficios de la Transformada de Fourier es su capacidad de revertir al Dominio del Tiempo sin perder información. Consideremos la misma señal que usamos en el ejemplo anterior:

```
A1=10;                  % Amplitude 1
A2=10;                  % Amplitude 2
w1=2*pi*0.2;            % Angular frequency 1
w2=2*pi*0.225;          % Angular frequency 2
Ts=1;                   % Sampling time
N=64;                   % Number of process samples to be generated
K=1;                    % Number of independent process realizations
sgm=1;                  % Standard deviation of the noise
n=repmat([0:N-1].',1,K);              % Generate resolution
phi1=repmat(rand(1,K)*2*pi,N,1);      % Random phase matrix 1
phi2=repmat(rand(1,K)*2*pi,N,1);      % Random phase matrix 2
x=A1*sin(w1*n*Ts+phi1)+A2*sin(w2*n*Ts+phi2)+sgm*randn(N,K);   % Resulting Signal
```

```
NFFT=256;            % FFT length
F=fft(x,NFFT);       % FFT result of time domain signal
```

Si abrimos `F` en Matlab, encontraremos que es una matriz de números complejos, una parte real y una parte imaginaria. Por definición, para recuperar la señal del dominio del tiempo original, necesitamos tanto el real (que representa la variación de la magnitud) como el imaginario (que representa la variación de la fase), para regresar al dominio del tiempo, uno puede querer simplemente:

```
TD = ifft(F,NFFT);   %Returns the Inverse of F in Time Domain
```

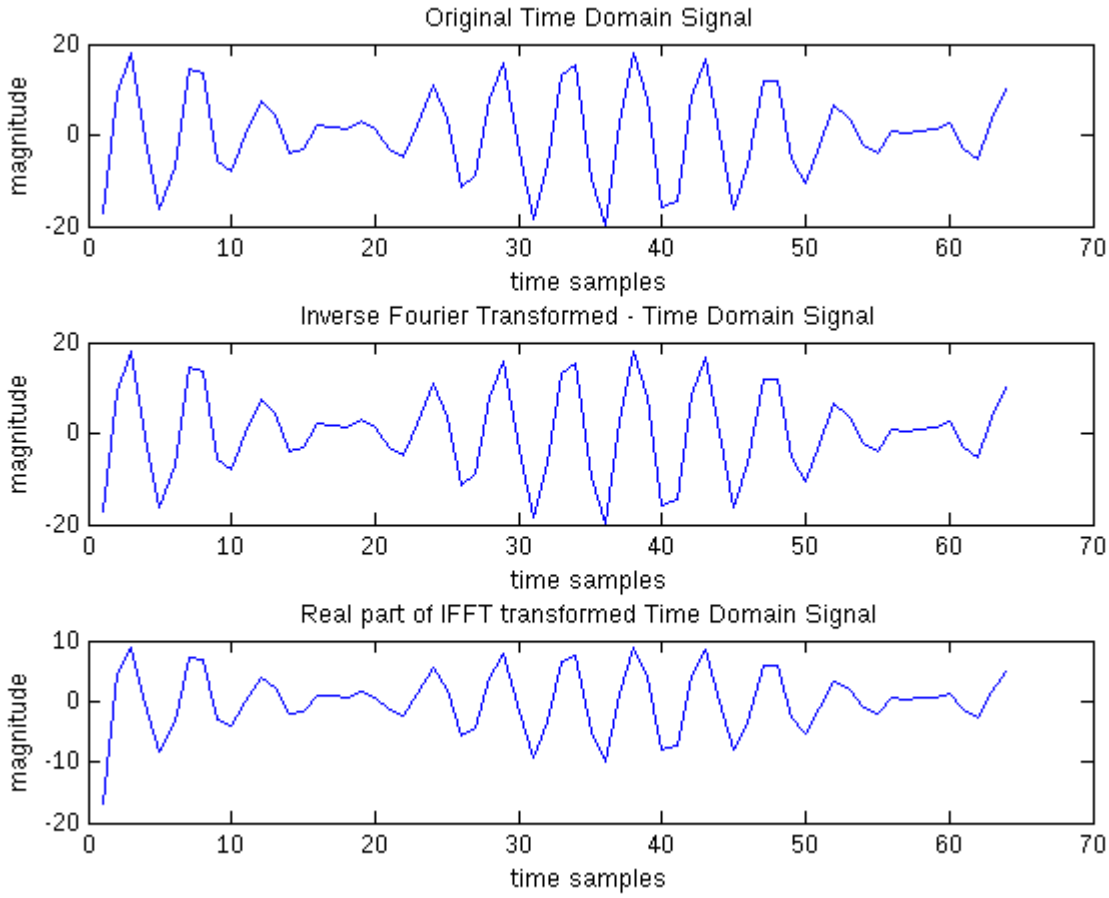
Tenga en cuenta que el TD devuelto sería la longitud 256 porque establecimos NFFT en 256, sin embargo, la longitud de x es solo 64, por lo que Matlab rellenará los ceros hasta el final de la transformación TD. Entonces, por ejemplo, si NFFT fue 1024 y la longitud fue 64, entonces el TD devuelto será 64 + 960 ceros. También tenga en cuenta que debido al redondeo de punto flotante, puede obtener algo como 3.1 * 10e-20 pero para propósitos generales: para cualquier X, ifft (fft (X)) es igual a X dentro del error de redondeo.

Digamos por un momento que después de la transformación, hicimos algo y solo nos queda la parte REAL de la FFT:

```
R = real(F);         %Give the Real Part of the FFT
TDR = ifft(R,NFFT);  %Give the Time Domain of the Real Part of the FFT
```

Esto significa que estamos perdiendo la parte imaginaria de nuestra FFT y, por lo tanto, estamos perdiendo información en este proceso inverso. Para conservar el original sin perder información, siempre debe mantener la parte imaginaria de la FFT utilizando `imag` y aplicar sus funciones a ambas o a la parte real.

```
figure
subplot(3,1,1)
plot(x);xlabel('time samples');ylabel('magnitude');title('Original Time Domain Signal')
subplot(3,1,2)
plot(TD(1:64));xlabel('time samples');ylabel('magnitude');title('Inverse Fourier Transformed -
Time Domain Signal')
subplot(3,1,3)
plot(TDR(1:64));xlabel('time samples');ylabel('magnitude');title('Real part of IFFT
transformed Time Domain Signal')
```



Imágenes y FT multidimensionales.

En la imagen médica, la espectroscopia, el procesamiento de imágenes, la criptografía y otras áreas de la ciencia y la ingeniería, es a menudo el caso de que se desea calcular las transformadas de Fourier multidimensionales de las imágenes. Esto es bastante sencillo en Matlab: las imágenes (multidimensionales) son solo matrices n-dimensionales, después de todo, y las transformadas de Fourier son operadores lineales: una transformada de Fourier a lo largo de otras dimensiones. Matlab proporciona `fft2` y `ifft2` para hacer esto en 2-d, o `fftn` en n-dimensiones.

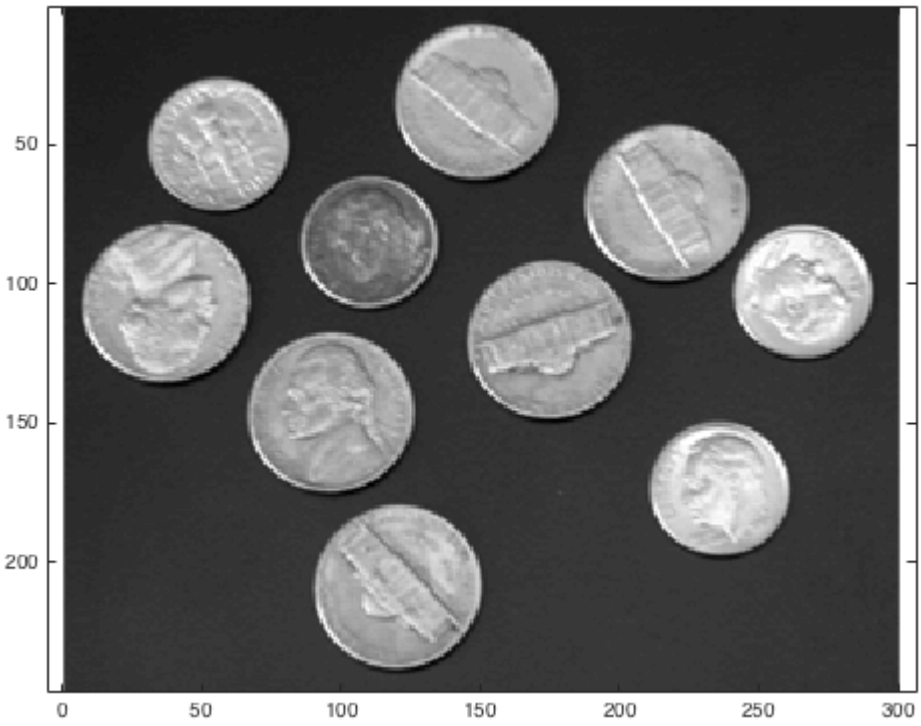
Una de las posibles fallas es que la transformada de Fourier de las imágenes generalmente se muestra "centrada en el orden", es decir, con el origen del espacio k en el centro de la imagen. Matlab proporciona el comando `fftshift` para intercambiar la ubicación de los componentes de CC de la transformada de Fourier de manera adecuada. Esta notación de ordenación hace que sea mucho más fácil realizar técnicas comunes de procesamiento de imágenes, una de las cuales se ilustra a continuación.

Relleno cero

Una forma "rápida y sucia" de interpolar una imagen pequeña a un tamaño más grande es transformarla con Fourier, rellenar con ceros la transformación de Fourier y luego tomar la transformación inversa. Esto se interpola de manera efectiva entre cada píxel con una función de

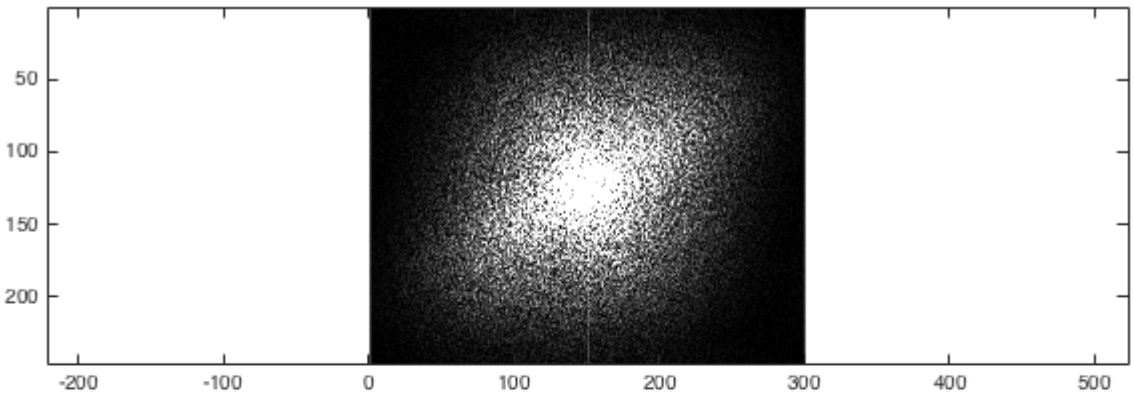
base de forma sincera, y se usa comúnmente para ampliar los datos de imágenes médicas de baja resolución. Empecemos cargando un ejemplo de imagen incorporado.

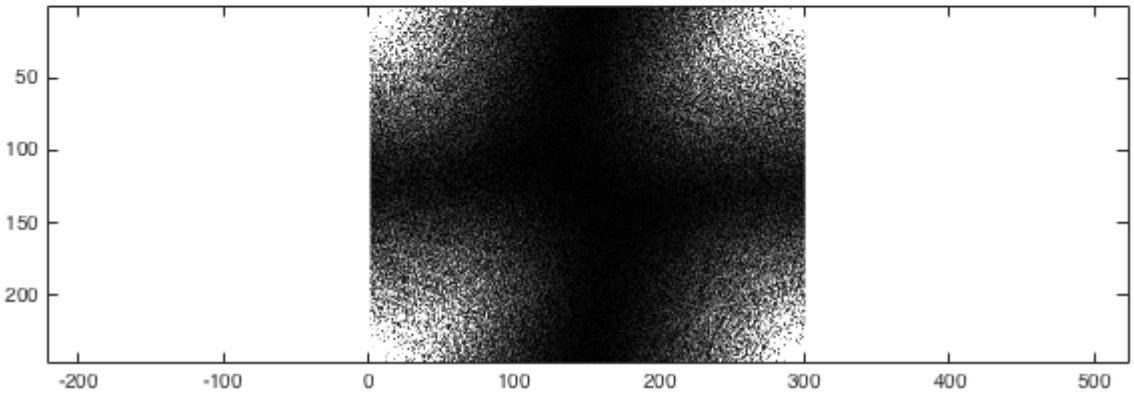
```
%Load example image
I=imread('coins.png'); %Load example data -- coins.png is builtin to Matlab
I=double(I); %Convert to double precision -- imread returns integers
imageSize = size(I); % I is a 246 x 300 2D image

%Display it
imagesc(I); colormap gray; axis equal;
%imagesc displays images scaled to maximum intensity
```



Ahora podemos obtener la transformada de Fourier de I. Para ilustrar lo que hace `fftshift` , comparemos los dos métodos:

```
% Fourier transform
%Obtain the centric- and non-centric ordered Fourier transform of I
k=fftshift(fft2(fftshift(I)));
kwrong=fft2(I);

%Just for the sake of comparison, show the magnitude of both transforms:
figure; subplot(2,1,1);
imagesc(abs(k),[0 1e4]); colormap gray; axis equal;
subplot(2,1,2);
imagesc(abs(kwrong),[0 1e4]); colormap gray; axis equal;
%(The second argument to imagesc sets the colour axis to make the difference clear).
```

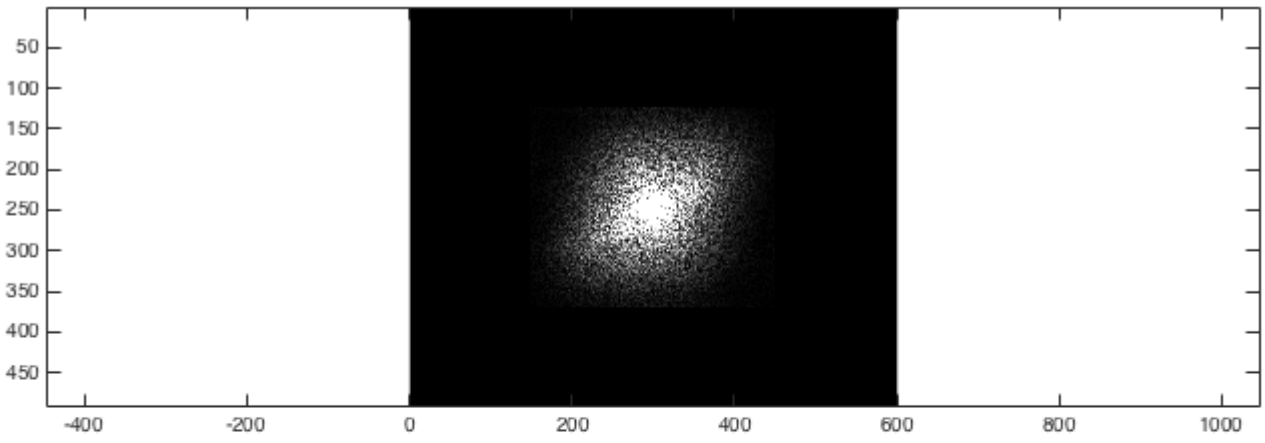


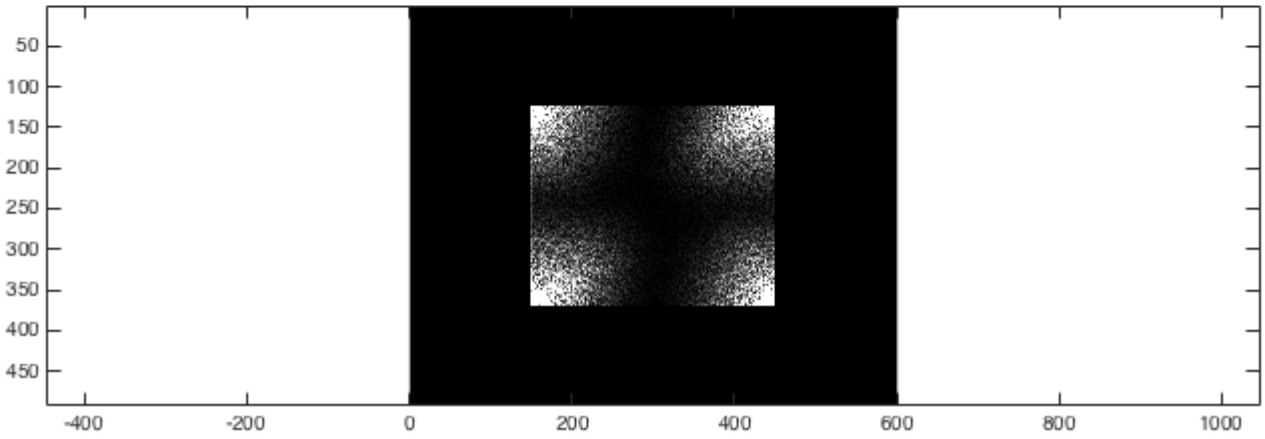


Ahora hemos obtenido el FT 2D de una imagen de ejemplo. Para rellenarlo con cero, queremos tomar cada espacio k, rellenar los bordes con ceros y luego tomar la transformada posterior:

```
%Zero fill
kzf = zeros(imageSize .* 2); %Generate a  492x600 empty array to put the result in
kzf(end/4:3*end/4-1,end/4:3*end/4-1) = k; %Put k in the middle
kzfwrong = zeros(imageSize .* 2); %Generate a  492x600 empty array to put the result in
kzfwrong(end/4:3*end/4-1,end/4:3*end/4-1) = kwrong; %Put k in the middle

%Show the differences again
%Just for the sake of comparison, show the magnitude of both transforms:
figure; subplot(2,1,1);
imagesc(abs(kzf),[0 1e4]); colormap gray; axis equal;
subplot(2,1,2);
imagesc(abs(kzfwrong),[0 1e4]); colormap gray; axis equal;
%(The second argument to imagesc sets the colour axis to make the difference clear).
```

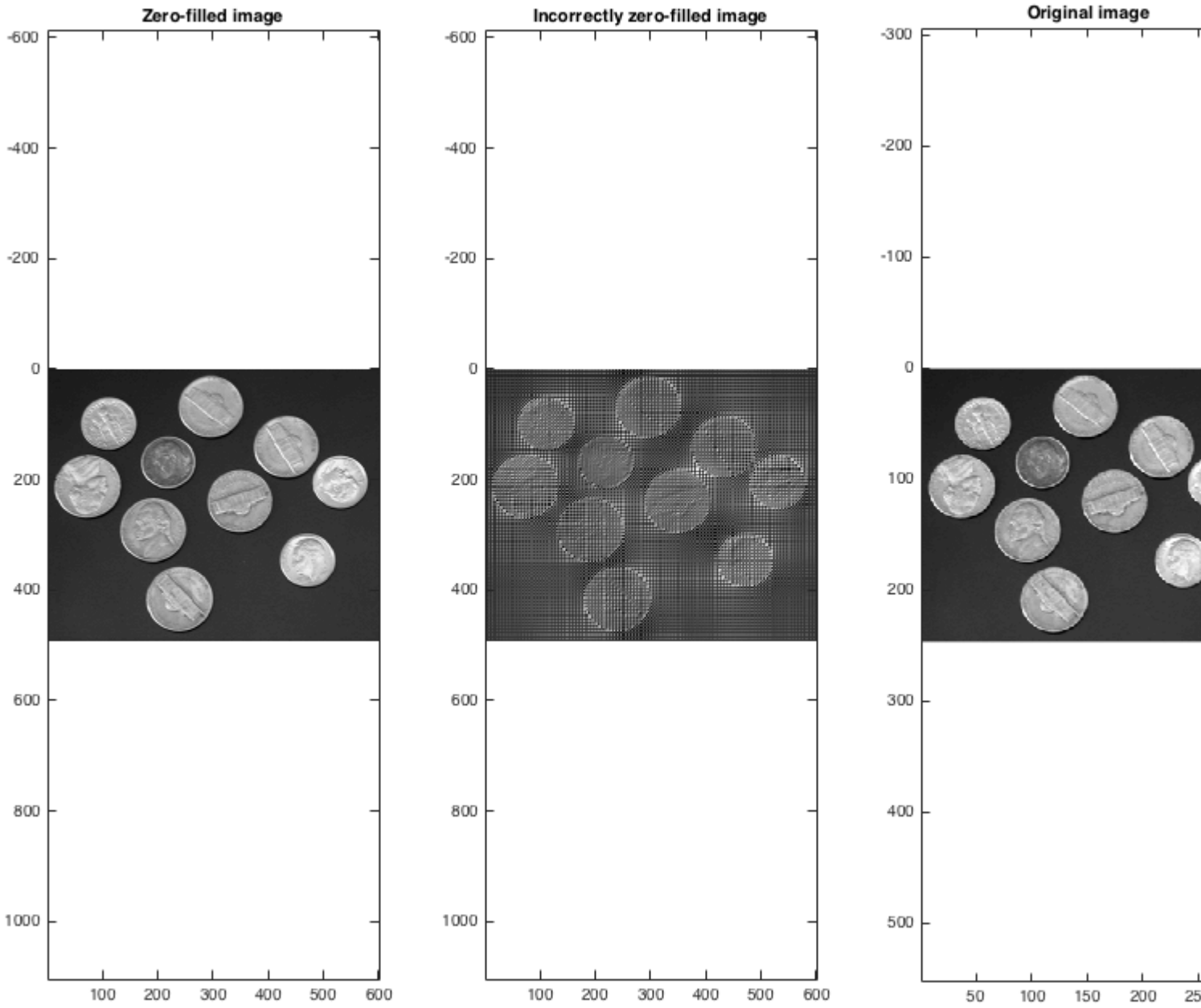
En este punto, el resultado es bastante poco destacable:





Una vez que tomamos las transformaciones inversas, podemos ver que (¡correctamente!) Los datos de relleno cero proporcionan un método sensato para la interpolación:

```
% Take the back transform and view
Izf = fftshift(ifft2(ifftshift(kzf)));
Izfwrong = ifft2(kzfwrong);

figure; subplot(1,3,1);
imagesc(abs(Izf)); colormap gray; axis equal;
title('Zero-filled image');
subplot(1,3,2);
imagesc(abs(Izfwrong)); colormap gray; axis equal;
title('Incorrectly zero-filled image');
subplot(1,3,3);
imagesc(I); colormap gray; axis equal;
title('Original image');
set(gcf,'color','w');
```



Tenga en cuenta que el tamaño de imagen con relleno cero es el doble del original. Uno puede rellenar a cero por más de un factor de dos en cada dimensión, aunque obviamente hacerlo no aumenta arbitrariamente el tamaño de una imagen.

Consejos, trucos, 3D y más allá.

El ejemplo anterior es válido para imágenes en 3D (como se genera a menudo por técnicas de imagen médica o microscopía confocal, por ejemplo), pero requiere que `fft2` sea reemplazado por `fftn(I, 3)` , por ejemplo. Debido a la naturaleza algo incómoda de escribir `fftshift(fft(fftshift(...` varias veces, es bastante común definir funciones como `fft2c` localmente para proporcionar una sintaxis más fácil a nivel local, como:

```
function y = fft2c(x)

y = fftshift(fft2(fftshift(x)));
```

Tenga en cuenta que la FFT es rápida, pero las transformaciones de Fourier multidimensionales y grandes aún tomarán tiempo en una computadora moderna. Además, es intrínsecamente complejo: la magnitud del espacio k se mostró arriba, pero la fase es absolutamente vital; Las traducciones en el dominio de imagen son equivalentes a una rampa de fase en el dominio de Fourier. Hay varias operaciones mucho más complejas que uno podría desear hacer en el dominio de Fourier, como filtrar frecuencias espaciales altas o bajas (al multiplicarlo con un filtro), o enmascarar puntos discretos correspondientes al ruido. En consecuencia, hay una gran cantidad de código generado por la comunidad para manejar las operaciones comunes de Fourier disponibles en el sitio principal del repositorio comunitario de Matlab, el Intercambio de archivos .

Lea Transformadas de Fourier y Transformadas de Fourier inversas en línea: https://riptutorial.com/es/matlab/topic/2181/transformadas-de-fourier-y-transformadas-de-fourier-inversas

Capítulo 29: Trucos utiles

Examples

Funciones útiles que operan en celdas y matrices.

Este sencillo ejemplo proporciona una explicación de algunas funciones que me parecieron extremadamente útiles desde que comencé a usar MATLAB: `cellfun` , `arrayfun` . La idea es tomar una variable de clase de matriz o celda, recorrer todos sus elementos y aplicar una función dedicada a cada elemento. Una función aplicada puede ser anónima, que suele ser un caso, o cualquier función regular definida en un archivo * .m.

Comencemos con un problema simple y digamos que necesitamos encontrar una lista de archivos * .mat en la carpeta. Para este ejemplo, primero vamos a crear algunos archivos * .mat en una carpeta actual:

```
for n=1:10; save(sprintf('mymatfile%d.mat',n)); end
```

Después de ejecutar el código, debe haber 10 archivos nuevos con la extensión * .mat. Si ejecutamos un comando para listar todos los archivos * .mat, como:

```
mydir = dir('*.mat');
```

debemos obtener una serie de elementos de una estructura dir; MATLAB debería dar un resultado similar a este:

```
10x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Como puedes ver, cada elemento de esta matriz es una estructura con un par de campos. Toda la información es realmente importante con respecto a cada archivo, pero en el 99% estoy bastante interesado en los nombres de archivo y nada más. Para extraer información de una matriz de estructura, solía crear una función local que implicaría crear variables temporales de un tamaño correcto, para bucles, extraer un nombre de cada elemento y guardarla en la variable creada. Una forma mucho más fácil de lograr exactamente el mismo resultado es usar una de las funciones mencionadas anteriormente:

```
mydirlist = arrayfun(@(x) x.name, dir('*.mat'), 'UniformOutput', false)
mydirlist =
    'mymatfile1.mat'
    'mymatfile10.mat'
    'mymatfile2.mat'
    'mymatfile3.mat'
```

```
    'mymatfile4.mat'
    'mymatfile5.mat'
    'mymatfile6.mat'
    'mymatfile7.mat'
    'mymatfile8.mat'
    'mymatfile9.mat'
```

¿Cómo funciona esta función? Por lo general, toma dos parámetros: un controlador de función como primer parámetro y una matriz. Entonces, una función operará en cada elemento de una matriz dada. Los parámetros tercero y cuarto son opcionales pero importantes. Si sabemos que una salida no será regular, debe guardarse en la celda. Esto debe ser señalado estableciendo `false` a `UniformOutput` . De forma predeterminada, esta función intenta devolver una salida regular, como un vector de números. Por ejemplo, extraigamos información sobre la cantidad de espacio en disco que toma cada archivo en bytes:

```
mydirbytes = arrayfun(@(x) x.bytes, dir('*.mat'))
mydirbytes =
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
```

o kilobytes:

```
mydirbytes = arrayfun(@(x) x.bytes/1024, dir('*.mat'))
mydirbytes =
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
   33.7500
```

Esta vez la salida es un vector regular de doble. `UniformOutput` se estableció en `true` de forma predeterminada.

`cellfun` es una función similar. La diferencia entre esta función y `arrayfun` es que `cellfun` opera en variables de clase de celda. Si deseamos extraer solo los nombres dados una lista de nombres de archivos en una celda 'mydirlist', solo tendríamos que ejecutar esta función de la siguiente manera:

```
mydirnames = cellfun(@(x) x(1:end-4), mydirlist, 'UniformOutput', false)
mydirnames =
    'mymatfile1'
```

```
    'mymatfile10'
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

Nuevamente, como una salida no es un vector regular de números, una salida debe guardarse en una variable de celda.

En el siguiente ejemplo, combino dos funciones en una y devuelvo solo una lista de nombres de archivos sin una extensión:

```
cellfun(@(x) x(1:end-4), arrayfun(@(x) x.name, dir('*.mat'), 'UniformOutput', false),
'UniformOutput', false)
ans =
    'mymatfile1'
    'mymatfile10'
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

Es una locura pero es muy posible porque `arrayfun` devuelve una celda que es la entrada esperada de `cellfun` ; una nota al margen de esto es que podemos forzar a cualquiera de esas funciones a devolver resultados en una variable de celda estableciendo `UniformOutput` en falso, explícitamente. Siempre podemos obtener resultados en una celda. Es posible que no podamos obtener resultados en un vector regular.

Hay una función más similar que opera en los campos de una estructura: `structfun` . No lo he encontrado particularmente tan útil como los otros dos, pero brillaría en algunas situaciones. Si, por ejemplo, alguien quisiera saber qué campos son numéricos o no numéricos, el siguiente código puede dar la respuesta:

```
structfun(@(x) ischar(x), mydir(1))
```

El primer y el segundo campo de una estructura dir son de tipo char. Por lo tanto, la salida es:

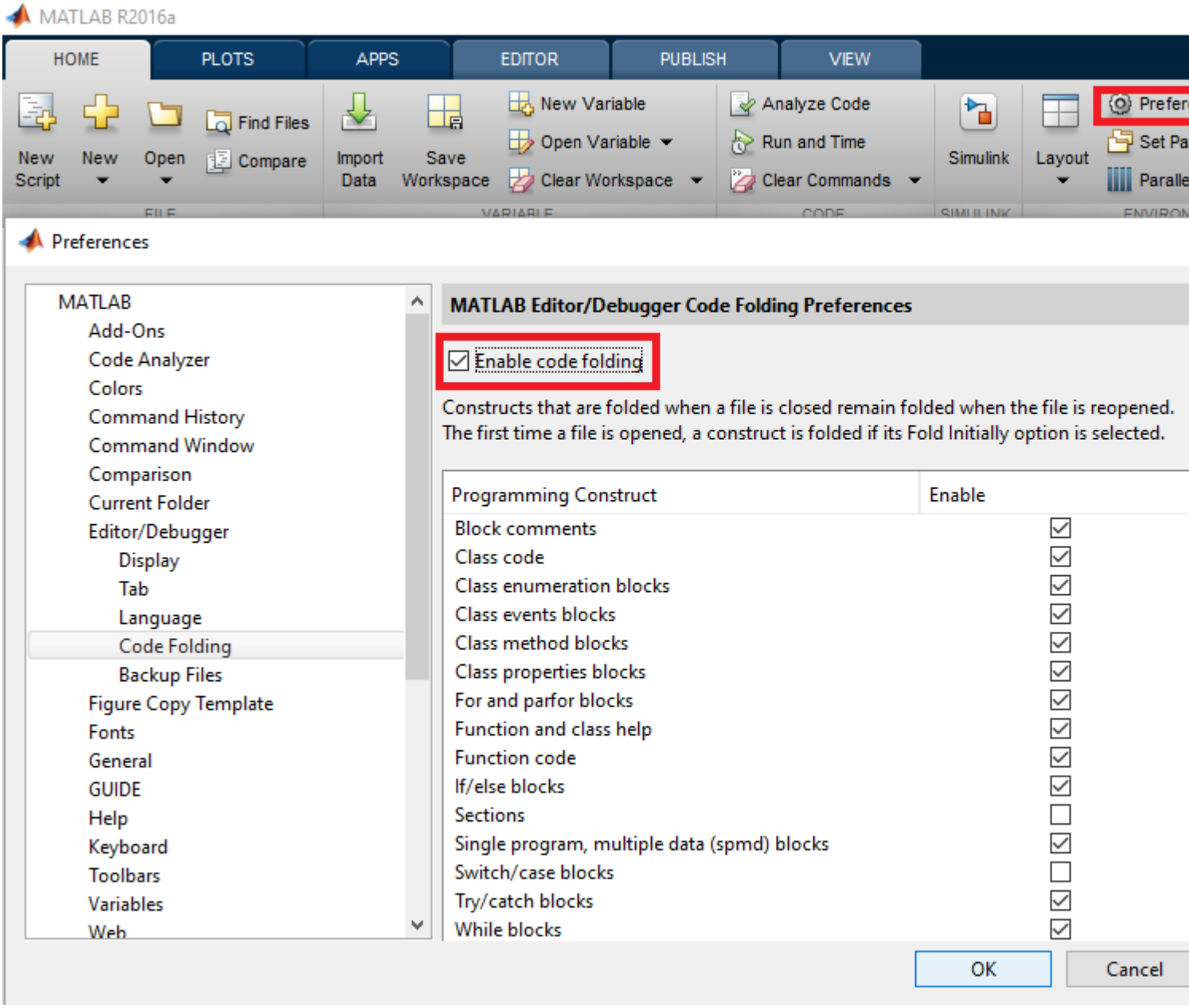
```
 1
 1
 0
 0
 0
```

Además, la salida es un vector lógico de `true` / `false` . En consecuencia, es regular y se puede guardar en un vector; No hay necesidad de usar una clase celular.

Preferencias de plegado de código

Es posible cambiar la preferencia de plegado de código para satisfacer sus necesidades. Por lo tanto, el plegado de código se puede configurar para habilitar / no poder para construcciones específicas (por ejemplo, `if block` , `for loop` , `Sections` ...).

Para cambiar las preferencias de plegado, vaya a Preferencias -> Plegado de código:



A continuación, puede elegir qué parte del código se puede plegar.

Alguna información:

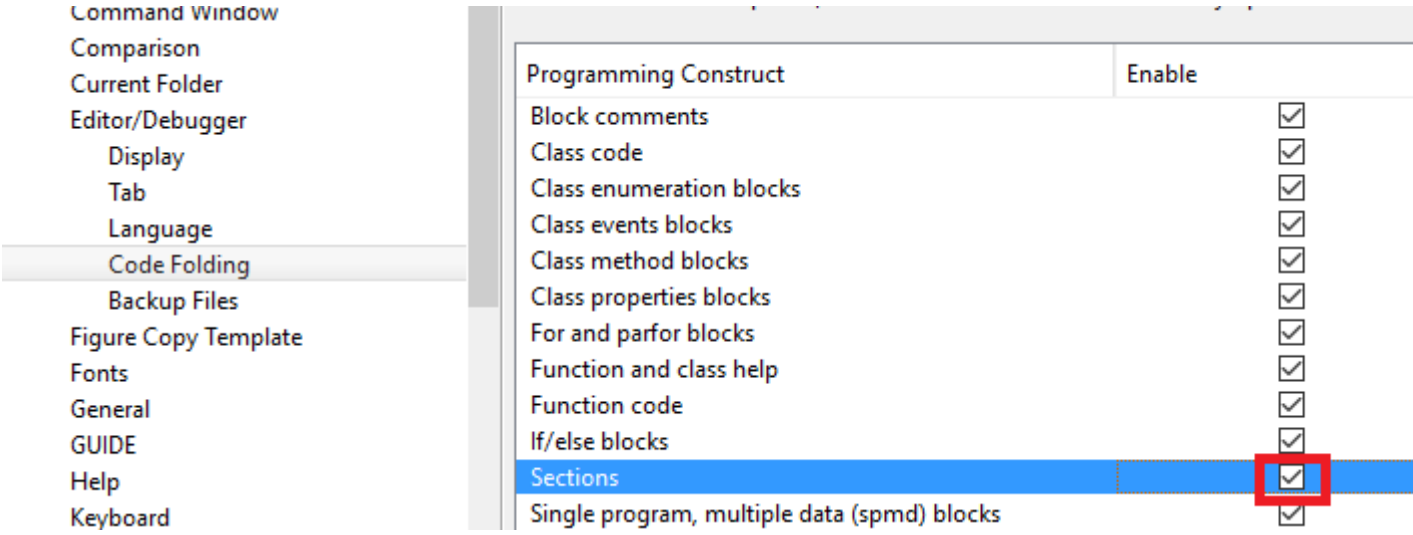
- Tenga en cuenta que también puede expandir o contraer todo el código de un archivo colocando el cursor en cualquier lugar dentro del archivo, haga clic con el botón derecho y luego seleccione Plegado de código> Expandir todo o Plegado de código> Plegado todo en el menú contextual.
- Tenga en cuenta que el plegado es persistente, en el sentido de que parte del código que se

ha expandido / contraído mantendrá su estado después de que Matlab o el archivo-m se haya cerrado y se vuelva a abrir.

Ejemplo: Para habilitar el plegado de secciones:

Una opción interesante es permitir plegar secciones. Las secciones están delimitadas por dos signos de porcentaje (`%%`).

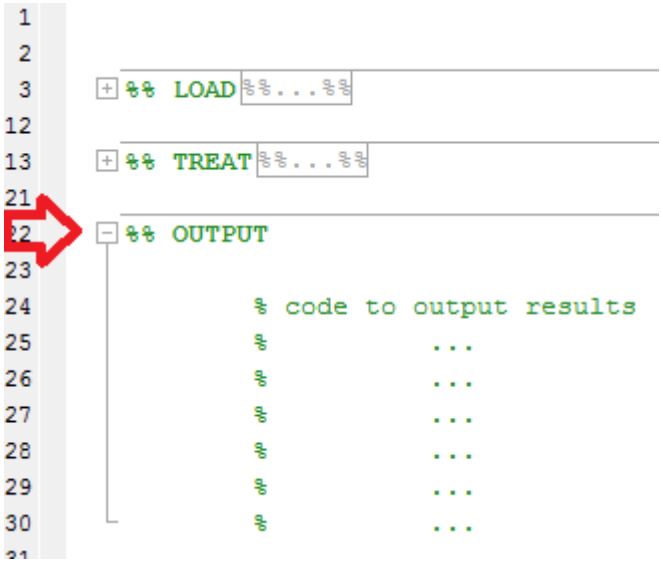
Ejemplo: Para habilitarlo marque la casilla "Secciones":



Entonces, en lugar de ver un código fuente largo similar a:

```
%% LOAD

        %code to load data
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...

%% TREAT
        % code to run the model
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...

%% OUTPUT

        % code to output results
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
```

Podrá plegar secciones para tener una visión general de su código:



Extraer datos de figuras

En algunas ocasiones, tuve una figura interesante que guardé, pero perdí un acceso a sus datos. Este ejemplo muestra un truco de cómo extraer información de una figura.

Las funciones clave son findobj y get . findobj devuelve un controlador a un objeto dado los

atributos o las propiedades del objeto, como `Type` o `Color` , etc. Una vez que se ha encontrado un objeto de línea, obtener puede devolver cualquier valor en poder de las propiedades. Resulta que los objetos de `Line` contienen todos los datos en las siguientes propiedades: `XData` , `YData` y `ZData` ; el último suele ser 0 a menos que una figura contenga un gráfico 3D.

El siguiente código crea una figura de ejemplo que muestra dos líneas, una función sin, un umbral y una leyenda

```
t = (0:1/10:1-1/10)';
y = sin(2*pi*t);
plot(t,y);
hold on;
plot([0 0.9],[0 0], 'k-');
hold off;
legend({'sin' 'threshold'});
```

El primer uso de findobj devuelve dos manejadores a ambas líneas:

```
findobj(gcf, 'Type', 'Line')
ans =
  2x1 Line array:

  Line    (threshold)
  Line    (sin)
```

Para acotar el resultado, findobj también se puede utilizar combinación de operadores lógicos `-and` , `-or` y nombres de propiedades. Por ejemplo, puedo encontrar un objeto de línea cuyo `DiplayName` es `sin` y leer su `XData` y `YData` .

```
lineh = findobj(gcf, 'Type', 'Line', '-and', 'DisplayName', 'sin');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

y compruebe si los datos son iguales.

```
isequal(t(:),xdata(:))
ans =
     1
isequal(y(:),ydata(:))
ans =
     1
```

Del mismo modo, puedo restringir mis resultados al excluir la línea negra (umbral):

```
lineh = findobj(gcf, 'Type', 'Line', '-not', 'Color', 'k');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

y la última verificación confirma que los datos extraídos de esta figura son los mismos:

```
isequal(t(:),xdata(:))
ans =
```

```
     1
isequal(y(:),ydata(:))
ans =
     1
```

Programación funcional utilizando funciones anónimas

Se pueden utilizar funciones anónimas para la programación funcional. El principal problema a resolver es que no existe una forma nativa para anclar una recursión, pero esto todavía se puede implementar en una sola línea:

```
if_ = @(bool, tf) tf{2-bool}();
```

Esta función acepta un valor booleano y una matriz de celdas de dos funciones. La primera de esas funciones se evalúa si el valor booleano se evalúa como verdadero, y la segunda si el valor booleano se evalúa como falso. Podemos escribir fácilmente la función factorial ahora:

```
fac = @(n,f) if_(n>1, {@()n*f(n-1,f), @()1});
```

El problema aquí es que no podemos invocar directamente una llamada recursiva, ya que la función aún no está asignada a una variable cuando se evalúa el lado derecho. Sin embargo podemos completar este paso escribiendo

```
factorial_ = @(n)fac(n,fac);
```

Ahora `@(n)fac(n,fac)` evalúa recursivamente la función factorial. Otra forma de hacer esto en la programación funcional utilizando un combinador y, que también se puede implementar fácilmente:

```
y_ = @(f)@(n)f(n,f);
```

Con esta herramienta, la función factorial es aún más corta:

```
factorial_ = y_(fac);
```

O directamente:

```
factorial_ = y_(@(n,f) if_(n>1, {@()n*f(n-1,f), @()1}));
```

Guarda varias figuras en el mismo archivo .fig

Al colocar varios manejadores de figuras en una matriz de gráficos, se pueden guardar varias figuras en el mismo archivo .fig

```
h(1) = figure;
scatter(rand(1,100),rand(1,100));
```

```
h(2) = figure;
scatter(rand(1,100),rand(1,100));

h(3) = figure;
scatter(rand(1,100),rand(1,100));

savefig(h,'ThreeRandomScatterplots.fig');
close(h);
```

Esto crea 3 diagramas de dispersión de datos aleatorios, cada parte de la matriz gráfica h. Luego, la matriz de gráficos se puede guardar usando savefig como con una figura normal, pero con el controlador de la matriz de gráficos como un argumento adicional.

Una nota al margen interesante es que las figuras tienden a permanecer dispuestas de la misma manera que se guardaron al abrirlas.

Bloques de comentarios

Si desea comentar parte de su código, los bloques de comentarios pueden ser útiles. El bloque de comentarios comienza con `%{` en una línea nueva y termina con `%}` en otra línea nueva:

```
a = 10;
b = 3;
%{
c = a*b;
d = a-b;
%}
```

Esto le permite plegar las secciones que comentó para que el código sea más limpio y compacto.

Estos bloques también son útiles para activar / desactivar partes de su código. Todo lo que tienes que hacer para descomentar el bloqueo es agregar otro `%` antes de que se establezca:

```
a = 10;
b = 3;
%%{ <-- another % over here
c = a*b;
d = a-b;
%}
```

A veces desea comentar una sección del código, pero sin afectar su sangría:

```
for k = 1:a
    b = b*k;
    c = c-b;
    d = d*c;
    disp(b)
end
```

Por lo general, cuando marca un bloque de código y presiona Ctrl + r para comentarlo (al agregar el `%` automáticamente a todas las líneas, luego cuando presiona Ctrl + i más tarde para la sangría automática, el bloque de código se mueve de su jerárquico correcto). lugar, y se movió

demasiado a la derecha:

```
for k = 1:a
    b = b*k;
    %     c = c-b;
    %     d = d*c;
    disp(b)
end
```

Una forma de resolver esto es usar bloques de comentarios, para que la parte interna del bloque permanezca correctamente sangrada:

```
for k = 1:a
    b = b*k;
    %{
    c = c-b;
    d = d*c;
    %}
    disp(b)
end
```

Lea Trucos utiles en línea: https://riptutorial.com/es/matlab/topic/4179/trucos-utiles

Capítulo 30: Usando funciones con salida lógica.

Examples

All and Any con arreglos vacíos

Se debe tener especial cuidado cuando existe la posibilidad de que una matriz se convierta en una matriz vacía cuando se trata de operadores lógicos. A menudo se espera que si `all(A)` es verdadero, entonces `any(A)` debe ser verdadero y si `any(A)` es falso, `all(A)` también debe ser falso. Ese no es el caso en MATLAB con matrices vacías.

```
>> any([])
ans =
     0
>> all([])
ans =
     1
```

Entonces, si por ejemplo está comparando todos los elementos de una matriz con un cierto umbral, debe tener en cuenta el caso en el que la matriz está vacía:

```
>> A=1:10;
>> all(A>5)
ans =
     0
>> A=1:0;
>> all(A>5)
ans =
     1
```

Use la función incorporada `isempty` para verificar las matrices vacías:

```
a = [];
isempty(a)
ans =
1
```

Lea Usando funciones con salida lógica. en línea:
https://riptutorial.com/es/matlab/topic/5608/usando-funciones-con-salida-logica-

Capítulo 31: Usando puertos seriales

Introducción

Los puertos serie son una interfaz común para comunicarse con sensores externos o sistemas integrados como Arduinos. Las comunicaciones seriales modernas a menudo se implementan a través de conexiones USB utilizando adaptadores serie USB. MATLAB proporciona funciones integradas para comunicaciones en serie, incluidos los protocolos RS-232 y RS-485. Estas funciones se pueden utilizar para puertos serie de hardware o conexiones serie-USB "virtuales". Los ejemplos aquí ilustran las comunicaciones seriales en MATLAB.

Parámetros

Parámetro puerto serie	Que hace
`BaudRate`	Establece la velocidad en baudios. Hoy en día, el más común es 57600, pero también se ven con frecuencia 4800, 9600 y 115200
`InputBufferSize`	El número de bytes guardados en la memoria. Matlab tiene un FIFO, lo que significa que los nuevos bytes serán descartados. El valor predeterminado es 512 bytes, pero se puede establecer fácilmente en 20 MB sin problemas. Solo hay unos pocos casos de borde en los que el usuario desearía que esto fuera pequeño.
`BytesAvailable`	El número de bytes que esperan ser leídos.
`ValuesSent`	El número de bytes enviados desde que se abrió el puerto.
`ValuesReceived`	El número de bytes leídos desde que se abrió el puerto
`BytesAvailableFcn`	Especifique la función de devolución de llamada que se ejecutará cuando un número especificado de bytes esté disponible en el búfer de entrada o se lea un terminador
`BytesAvailableFcnCount`	Especifique el número de bytes que deben estar disponibles en el búfer de entrada para generar un evento de `bytes-available`
`BytesAvailableFcnMode`	Especifique si el evento de `bytes-available` se genera después de que un número específico de bytes esté disponible en el búfer de entrada, o después de que se lea un terminador

Examples

Creando un puerto serial en Mac / Linux / Windows

```
% Define serial port with a baud rate of 115200
rate = 115200;
if ispc
    s = serial('COM1', 'BaudRate',rate);
elseif ismac
    % Note that on OSX the serial device is uniquely enumerated. You will
    % have to look at /dev/tty.* to discover the exact signature of your
    % serial device
    s = serial('/dev/tty.usbserial-A104VFT7', 'BaudRate',rate);
elseif isunix
    s = serial('/dev/ttyusb0', 'BaudRate',rate);
end

% Set the input buffer size to 1,000,000 bytes (default: 512 bytes).
s.InputBufferSize = 1000000;

% Open serial port
fopen(s);
```

Leyendo desde el puerto serie

Suponiendo que ha creado la serie de objetos puerto `s` como en este ejemplo, entonces

```
% Read one byte
data = fread(s, 1);

% Read all the bytes, version 1
data = fread(s);

% Read all the bytes, version 2
data = fread(s, s.BytesAvailable);

% Close the serial port
fclose(s);
```

Cerrar un puerto serie incluso si se pierde, se elimina o se sobrescribe

Suponiendo que ha creado la serie de objetos puerto `s` como en este ejemplo, entonces para cerrarlo

```
fclose(s)
```

Sin embargo, a veces puede perder el puerto accidentalmente (por ejemplo, borrar, sobrescribir, cambiar el alcance, etc.), y `fclose(s)` ya no funcionará. La solucion es facil

```
fclose(instrfindall)
```

Más información en `instrfindall()` .

Escribiendo al puerto serie

Suponiendo que ha creado la serie de objetos puerto `s` como en este ejemplo, entonces

```
% Write one byte
fwrite(s, 255);

% Write one 16-bit signed integer
fwrite(s, 32767, 'int16');

% Write an array of unsigned 8-bit integers
fwrite(s,[48 49 50],'uchar');

% Close the serial port
fclose(s);
```

Elegir su modo de comunicación

Matlab admite *la* comunicación *síncrona* y *asíncrona* con un puerto serie. Es importante elegir el modo de comunicación correcto. La elección dependerá de:

- Cómo se comporta el instrumento con el que te estás comunicando.
- qué otras funciones tendrá que hacer su programa principal (o GUI) además de administrar el puerto serie.

Definiré 3 casos diferentes para ilustrar, desde el más simple hasta el más exigente. Para los 3 ejemplos, el instrumento al que me estoy conectando es una placa de circuito con un inclinómetro, que puede funcionar en los 3 modos que describiré a continuación.

Modo 1: Sincrónico (Maestro / Esclavo)

Este modo es el más sencillo. Corresponde al caso en el que la PC es el *maestro* y el instrumento es el *esclavo* . El instrumento no envía nada al puerto serie por sí mismo, solo **responde** una respuesta después de que el maestro (la PC, su programa) le haga una pregunta o una orden. Por ejemplo:

- La PC envía un comando: "Dame una medida ahora"
- El instrumento recibe el comando, toma la medida y luego devuelve el valor de la medida a la línea serie: "El valor del inclinómetro es XXX".

O

- La PC envía un comando: "Cambiar del modo X al modo Y"
- El instrumento recibe el comando, lo ejecuta y luego envía un mensaje de confirmación a la línea serie: " *Comando ejecutado* " (o " *Comando NO ejecutado* "). Esto se denomina comúnmente respuesta ACK / NACK (para "Confirmar (d)" / "NO reconocido").

Resumen: en este modo, el instrumento (el *Esclavo*) solo envía datos a la línea serie **inmediatamente después de** que el PC (el *Maestro*) le haya pedido

Modo 2: asíncrono

Ahora supongamos que inicié mi instrumento, pero es más que un simple sensor. Monitorea constantemente su propia inclinación y mientras sea vertical (dentro de una tolerancia, digamos +/- 15 grados), permanece en silencio. Si el dispositivo está inclinado más de 15 grados y se acerca a la horizontal, envía un mensaje de alarma a la línea serie, seguido inmediatamente por una lectura de la inclinación. Mientras la inclinación esté por encima del umbral, continúa enviando una lectura de inclinación cada 5 s.

Si su programa principal (o GUI) está constantemente "esperando" el mensaje que llega a la línea serie, puede hacerlo bien ... pero no puede hacer nada más mientras tanto. Si el programa principal es una GUI, es muy frustrante tener una GUI aparentemente "congelada" porque no aceptará ninguna entrada del usuario. Esencialmente, se convirtió en el *Esclavo* y el instrumento es el *Maestro* . A menos que tenga una forma elegante de controlar su GUI desde el instrumento, esto es algo que debe evitar. Afortunadamente, el modo de comunicación *asíncrono* le permitirá:

- define una función separada que le dice a tu programa qué hacer cuando se recibe un mensaje
- mantenga esta función en una esquina, solo se llamará y ejecutará **cuando llegue un mensaje** a la línea serie. El resto del tiempo la GUI puede ejecutar cualquier otro código que tenga que ejecutar.

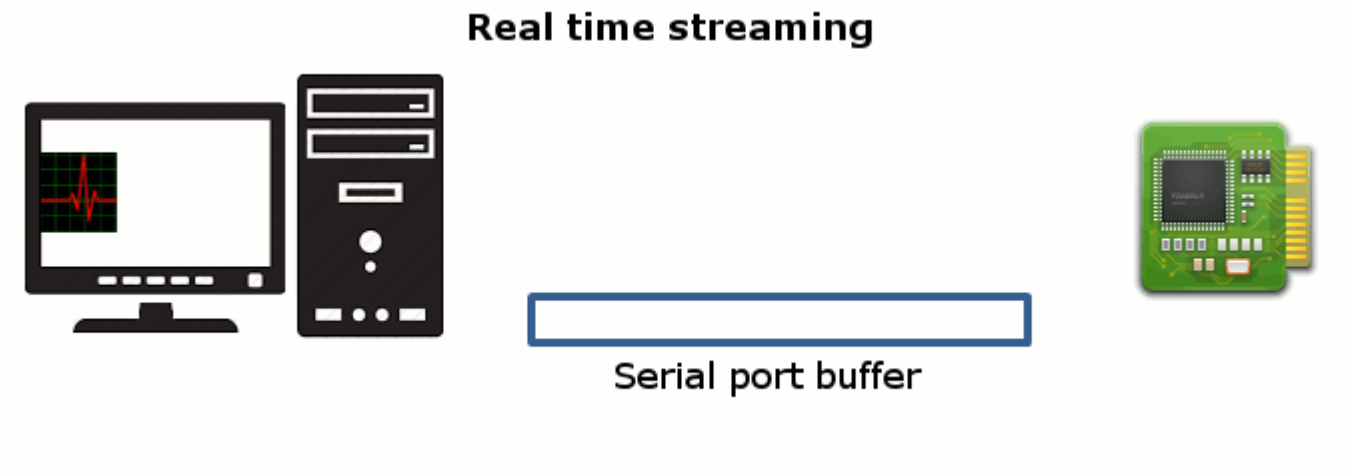
Resumen: en este modo, el instrumento puede enviar un mensaje a la línea serie en cualquier momento (pero no necesariamente *todo* el tiempo). La PC no *espera* permanentemente a que se procese un mensaje. Se permite ejecutar cualquier otro código. Solo cuando llega un mensaje, activa una función que luego leerá y procesará este mensaje.

Modo 3: Streaming (*tiempo real*)

Ahora desatemos todo el poder de mi instrumento. Lo puse en un modo en el que enviará constantemente mediciones a la línea serie. Mi programa desea recibir estos paquetes y mostrarlos en una curva o en una pantalla digital. Si solo envía un valor cada 5 segundos como antes, no hay problema, mantenga el modo anterior. Pero mi instrumento a toda potencia envía un punto de datos a la línea serie a 1000Hz, es decir, envía un nuevo valor cada milisegundo. Si me quedo en el *modo asíncrono* descrito anteriormente, existe un alto riesgo (en realidad, una certeza garantizada) de que la función especial que definimos para procesar cada nuevo paquete tomará más de 1 ms en ejecutarse (si desea trazar o mostrar el valor, las funciones gráficas son bastante lentas, ni siquiera considerando el filtrado o FFT de la señal). Significa que la función comenzará a ejecutarse, pero antes de que finalice, un nuevo paquete llegará y activará la función nuevamente. La segunda función se coloca en una cola para su ejecución, y solo se iniciará cuando se complete la primera ... pero para este momento llegaron algunos paquetes nuevos y cada uno colocó una función para ejecutar en la cola. Puede prever rápidamente el resultado: Cuando estoy trazando los puntos 5, ya tengo cientos esperando para ser trazados también ... el gui se ralentiza, eventualmente se congela, la pila crece, los búferes se llenan, hasta que algo cede. Finalmente, te quedas con un programa completamente congelado o simplemente uno bloqueado.

Para superar esto, desconectaremos aún más el enlace de sincronización entre la PC y el instrumento. Dejaremos que el instrumento envíe datos a su propio ritmo, sin activar inmediatamente una función a cada llegada de paquete. El búfer del puerto serie solo acumulará los paquetes recibidos. La PC solo recopilará los datos en el búfer a un ritmo que pueda administrar (un intervalo regular, configurado en el lado de la PC), hará algo con ella (mientras el búfer se llena por el instrumento) y luego recopila un nuevo lote de Datos del búfer ... y así sucesivamente.

Resumen: en este modo, el instrumento envía datos continuamente, que son recopilados por el búfer del puerto serie. A intervalos regulares, la PC recopila datos del búfer y hace algo con él. No hay un enlace de sincronización duro entre la PC y el instrumento. Ambos ejecutan sus tareas en su propio tiempo.



Procesando automáticamente los datos recibidos de un puerto serie

Algunos dispositivos conectados a través de un puerto serie envían datos a su programa a una velocidad constante (transmisión de datos) o envían datos a intervalos impredecibles. Puede configurar el puerto serie para ejecutar una función automáticamente para manejar datos cada vez que llega. Esto se denomina "función de devolución de llamada" para el objeto de puerto serie.

Hay dos propiedades del puerto serie que deben configurarse para usar esta función: el nombre de la función que desea para la devolución de llamada (`BytesAvailableFcn`) y la condición que debe activar la ejecución de la función de devolución de llamada (`BytesAvailableFcnMode`).

Hay dos formas de activar una función de devolución de llamada:

1. Cuando se ha recibido un cierto número de bytes en el puerto serie (normalmente se utiliza para datos binarios)
2. Cuando se recibe un determinado carácter en el puerto serie (normalmente se utiliza para texto o datos ASCII)

Las funciones de devolución de llamada tienen dos argumentos de entrada requeridos, llamados `obj` y `event` . `obj` es el puerto serie. Por ejemplo, si desea imprimir los datos recibidos del puerto serie, defina una función para imprimir los datos llamados `newdata` :

```
function newdata(obj,event)
    [d,c] = fread(obj);  % get the data from the serial port
    % Note: for ASCII data, use fscanf(obj) to return characters instead of binary values
    fprintf(1,'Received %d bytes\n',c);
    disp(d)
end
```

Por ejemplo, para ejecutar la función `newdata` cada vez que se reciben 64 bytes de datos, configure el puerto serie de esta manera:

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcnCount = 64;
```

```
s.BytesAvailableFcn = @newdata;
```

Con el texto o los datos ASCII, los datos se suelen dividir en líneas con un "carácter de terminación", al igual que el texto en una página. Para ejecutar la función `newdata` cada vez que se recibe el carácter de retorno de carro, configure el puerto serie de esta manera:

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'terminator';
s.Terminator = 'CR';  % the carriage return, ASCII code 13
s.BytesAvailableFcn = @newdata;
```

Lea Usando puertos seriales en línea: https://riptutorial.com/es/matlab/topic/1176/usando-puertos-seriales

Capítulo 32: Uso de la función "úmarray () `

Introducción

`accumarray` permite agregar elementos de una matriz de varias maneras, potencialmente aplicando alguna función a los elementos en el proceso. `accumarray` puede considerar la `accumarray` como un reductor ligero (ver también: Introducción a MapReduce).

Este tema contendrá escenarios comunes en los que la `accumarray` es especialmente útil.

Sintaxis

- acumulación (subscriptArray, valuesArray)
- acumulación (subscriptArray, valuesArray, sizeOfOutput)
- acumulación (subscriptArray, valuesArray, sizeOfOutput, funcHandle)
- acumulación (subscriptArray, valuesArray, sizeOfOutput, funcHandle, fillVal)
- acumulación (subscriptArray, valuesArray, sizeOfOutput, funcHandle, fillVal, isSparse)

Parámetros

Parámetro	Detalles
`subscriptArray`	Matriz de subíndices, especificada como un vector de índices, matriz de índices o matriz de celdas de vectores de índice.
`valuesArray`	Datos, especificados como un vector o un escalar.
`sizeOfOutput`	Tamaño de la matriz de salida, especificado como un vector de enteros positivos.
`funcHandle`	Función que se aplicará a cada conjunto de elementos durante la agregación, especificada como un identificador de función o `[]` .
`fillVal`	Valor de relleno, para cuando `subs` no hace referencia a cada elemento en la salida.
`isSparse`	¿Debería la salida ser una matriz dispersa?

Observaciones

- Introducido en MATLAB v7.0.

Referencias :

1. " `accumarray` ", *por Loren Shure* , 20 de febrero de 2008 .
2. `accumarray` en la documentación oficial de MATLAB.

Examples

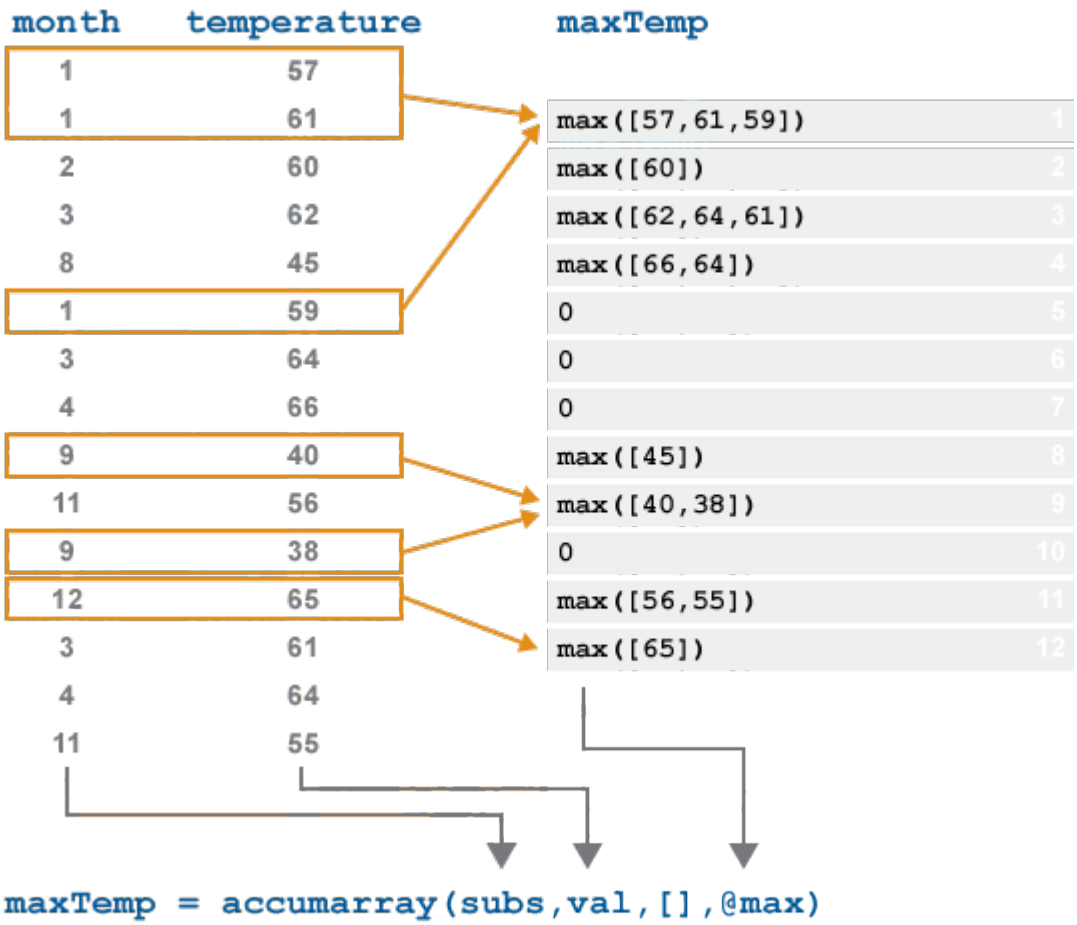
Encontrar el valor máximo entre los elementos agrupados por otro vector.

Este es un ejemplo oficial de MATLAB.

Considere el siguiente código:

```
month = [1;1;2;3;8;1;3;4;9;11;9;12;3;4;11];
temperature = [57;61;60;62;45;59;64;66;40;56;38;65;61;64;55];
maxTemp = accumarray(month,temperature,[],@max);
```

La siguiente imagen muestra el proceso de cálculo realizado por el `accumarray` en este caso:



En este ejemplo, todos los valores que tienen el mismo `month` se recopilan primero, y luego se aplica la función especificada por la 4ª entrada de `accumarray` (en este caso, `@max`) a cada uno de estos conjuntos.

Aplicar filtro a los parches de imagen y establecer cada píxel como la media del resultado de cada parche

Muchos de los algoritmos modernos de procesamiento de imágenes utilizan parches, que son su elemento básico para trabajar.
Por ejemplo, uno podría eliminar parches (ver algoritmo BM3D).

Sin embargo, al crear la imagen a partir de los parches procesados tenemos muchos resultados para el mismo píxel.
Una forma de lidiar con esto es tomando el promedio (Promedio empírico) de todos los valores del mismo píxel.

El siguiente código muestra cómo dividir una imagen en parches y reconstruir la imagen a partir de parches usando el promedio utilizando `[accumarray()][1]` :

```
numRows = 5;
numCols = 5;

numRowsPatch = 3;
numColsPatch = 3;

% The Image
mI = rand([numRows, numCols]);

% Decomposing into Patches - Each pixel is part of many patches (Neglecting
% boundariwes, each pixel is part of (numRowsPatch * numColsPatch) patches).
mY = ImageToColumnsSliding(mI, [numRowsPatch, numColsPatch]);

% Here one would apply some operation which work on patches

% Creating image of the index of each pixel
mPxIdx = reshape(1:(numRows * numCols), [numRows, numCols]);

% Creating patches of the same indices
mSubsAccu = ImageToColumnsSliding(mPxIdx, [numRowsPatch, numColsPatch]);

% Reconstruct the image - Option A
mO = accumarray(mSubsAccu(:), mY(:)) ./ accumarray(mSubsAccu(:), 1);

% Reconstruct the image - Option B
mO = accumarray(mSubsAccu, mY(:), [(numRows * numCols), 1], @(x) mean(x));

% Rehsape the Vector into the Image
mO = reshape(mO, [numRows, numCols]);
```

Lea Uso de la función "úmarray () ` en línea: https://riptutorial.com/es/matlab/topic/9321/uso-de-la-funcion--umarray-----

Capítulo 33: Utilidades de programación

Examples

Temporizador simple en MATLAB

El siguiente es un temporizador que se dispara en un intervalo fijo. Su tiempo de espera está definido por `Period` e invoca una devolución de llamada definida por `Timerfcn` en el tiempo de espera.

```
t = timer;
t.TasksToExecute = Inf;
t.Period = 0.01; % timeout value in seconds
t.TimerFcn = @(myTimerObj, thisEvent)disp('hello'); % timer callback function
t.ExecutionMode = 'fixedRate';
start(t)
pause(inf);
```

Lea Utilidades de programación en línea: https://riptutorial.com/es/matlab/topic/1655/utilidades-de-programacion

Capítulo 34: Vectorización

Examples

Operaciones de elementos sabios

MATLAB apoya (y alienta) operaciones vectorizadas en vectores y matrices.
Por ejemplo, supongamos que tenemos `A` y `B` , dos matrices `n` by- `m` y queremos que `C` sea el producto de los elementos correspondientes (es decir, `C(i,j) = A(i,j)*B(i,j)`).

La forma no vectorizada, utilizando bucles anidados es la siguiente:

```
C = zeros(n,m);
for ii=1:n
    for jj=1:m
        C(ii,jj) = A(ii,jj)*B(ii,jj);
    end
end
```

Sin embargo, la forma vectorizada de hacerlo es mediante el uso de un operador de elementos `.*` :

```
C = A.*B;
```

- Para obtener más información sobre la multiplicación de elementos en MATLAB, consulte la documentación de `times` .
- Para obtener más información sobre la diferencia entre las operaciones de matriz y matriz, consulte Operaciones de matriz frente a Matriz en la documentación de MATLAB.

Suma, media, prod & co

Dado un vector aleatorio

```
v = rand(10,1);
```

Si desea la suma de sus elementos, **NO** use un bucle

```
s = 0;
for ii = 1:10
    s = s + v(ii);
end
```

pero usa la capacidad vectorizada de la función `sum()`

```
s = sum(v);
```

Las funciones como `sum()` , `mean()` , `prod()` y otras, tienen la capacidad de operar directamente a lo largo de filas, columnas u otras dimensiones.

Por ejemplo, dada una matriz aleatoria.

```
A = rand(10,10);
```

el promedio para cada **columna** es

```
m = mean(A,1);
```

el promedio para cada **fila** es

```
m = mean(A,2)
```

Todas las funciones anteriores solo funcionan en una dimensión, pero ¿qué sucede si desea sumar toda la matriz? Podrías usar:

```
s = sum(sum(A))
```

Pero, ¿qué pasa si tengo una matriz ND? aplicando `sum` en `sum` en `sum` ... no parece la mejor opción, en su lugar use el operador `:` para vectorizar su matriz:

```
s = sum(A(:))
```

y esto resultará en un número que es la suma de toda tu matriz, no importa cuántas dimensiones tenga.

Uso de bsxfun

Muy a menudo, la razón por la que el código se ha escrito en un bucle `for` es para calcular valores de valores "cercanos". La función `bsxfun` se puede usar para hacer esto de una manera más sucinta.

Por ejemplo, suponga que desea realizar una operación de columnas en la matriz `B` , restando de ella la media de cada columna:

```
B = round(randn(5)*10);                 % Generate random data
A = zeros(size(B));                     % Preallocate array
for col = 1:size(B,2);                   % Loop over columns
    A(:,col) = B(:,col) - mean(B(:,col));   % Subtract means
end
```

Este método es ineficiente si `B` es grande, a menudo debido a que MATLAB tiene que mover el contenido de las variables en la memoria. Al usar `bsxfun` , uno puede hacer el mismo trabajo de forma clara y sencilla en una sola línea:

```
A = bsxfun(@minus, B, mean(B));
```

Aquí, `@minus` es un controlador de función para el operador `minus` (`-`) y se aplicará entre los elementos de las dos matrices `B` y la `mean(B)` . También son posibles otros manejadores de función, incluso los definidos por el usuario.

A continuación, suponga que desea agregar el vector de fila `v` a cada fila de la matriz `A` :

```
v = [1,  2,  3];

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

El enfoque ingenuo es usar un bucle (***no*** *hacer esto*):

```
B = zeros(3);
for row = 1:3
    B(row,:) = A(row,:) + v;
end
```

Otra opción sería replicar `v` con `repmat` (***tampoco*** *haga esto*):

```
>> v = repmat(v,3,1)
v =
     1     2     3
     1     2     3
     1     2     3

>> B = A + v;
```

En su lugar, use `bsxfun` para esta tarea:

```
>> B = bsxfun(@plus, A, v);
B =
     9     3     9
     4     7    10
     5    11     5
```

Sintaxis

```
bsxfun(@fun, A, B)
```

donde `@fun` es una de las funciones admitidas y las dos matrices `A` y `B` respetan las dos condiciones a continuación.

El nombre `bsxfun` ayuda a entender cómo funciona la función y que representa **B** ción **FUN** ciertas piezas con **S** ingleton e **X** pansión. En otras palabras, si:

1. Dos matrices comparten las mismas dimensiones excepto una

2. y la dimensión discordante es un singleton (es decir, tiene un tamaño de `1`) en cualquiera de las dos matrices

luego la matriz con la dimensión singleton se expandirá para coincidir con la dimensión de la otra matriz. Después de la expansión, una función binaria se aplica elementwise en los dos arreglos.

Por ejemplo, sea `A` una matriz `M` -by- `N` -by `K` y `B` es una matriz `M` -by- `N` En primer lugar, sus dos primeras dimensiones tienen tamaños correspondientes. En segundo lugar, `A` tiene `K` capas, mientras que `B` tiene implícitamente solo `1` , por lo que es un singleton. Se cumplen todas las **condiciones** y `B` se replicará para coincidir con la tercera dimensión de `A`

En otros idiomas, esto se conoce comúnmente como *transmisión* y ocurre automáticamente en Python (numpy) y Octave.

La función, `@fun` , debe ser una función binaria, lo que significa que debe tener exactamente dos entradas.

Observaciones

Internamente, `bsxfun` no replica la matriz y ejecuta un bucle eficiente.

Enmascaramiento logico

MATLAB admite el uso de enmascaramiento lógico para realizar la selección en una matriz sin el uso de bucles for o enunciados if.

Una máscara lógica se define como una matriz compuesta de solo `1` y `0` .

Por ejemplo:

```
mask = [1 0 0; 0 1 0; 0 0 1];
```

Es una matriz lógica que representa la matriz de identidad.

Podemos generar una máscara lógica utilizando un predicado para consultar una matriz.

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

Primero creamos una matriz de 3x3, `A` , que contiene los números del 1 al 9. Luego buscamos en `A` valores que son mayores que 4 y almacenamos el resultado en una nueva matriz llamada `B`

`B` es una matriz lógica de la forma:

```
B = [0 0 0
     0 1 1
     1 1 1]
```

O `1` cuando el predicado `A > 4` era verdadero. Y `0` cuando era falso.

Podemos usar matrices lógicas para acceder a elementos de una matriz. Si se utiliza una matriz lógica para seleccionar elementos, se seleccionarán los índices en los que aparece un `1` en la matriz lógica en la matriz que se está seleccionando.

Usando la misma `B` de arriba, podríamos hacer lo siguiente:

```
C = [0 0 0; 0 0 0; 0 0 0];
C(B) = 5;
```

Esto seleccionaría todos los elementos de `C` donde `B` tiene un `1` en ese índice. Los índices en `C` se ajustan a `5` .

Nuestra `C` ahora se ve como:

```
C = [0 0 0
     0 5 5
     5 5 5]
```

Podemos reducir los bloques de código complicados que contienen `if` y `for` uso de máscaras lógicas.

Toma el código no vectorizado:

```
A = [1 3 5; 7 9 11; 11 9 7];
for j = 1:length(A)
  if A(j) > 5
    A(j) = A(j) - 2;
  end
end
```

Esto se puede acortar utilizando el enmascaramiento lógico al siguiente código:

```
A = [1 3 5; 7 9 11; 11 9 7];
B = A > 5;
A(B) = A(B) - 2;
```

O incluso más corto:

```
A = [1 3 5; 7 9 11; 11 9 7];
A(A > 5) = A(A > 5) - 2;
```

Expansión de matriz implícita (difusión) [R2016b]

MATLAB R2016b presentó una generalización de su mecanismo de expansión escalar [1] , [2] , para admitir también ciertas operaciones de elementos entre *matrices* de diferentes tamaños, siempre que su dimensión sea compatible. Los operadores que soportan la expansión implícita son [1] :

- **Operadores aritméticos elemento a elemento:** `+` , `-` , `.*` , `.^` , `./` , `.\` .
- **Operadores relacionales:** `<` , `<=` , `>` , `>=` , `==` , `~=` .
- **Operadores lógicos:** `&` , `|` , `xor` .

- **Funciones de bit-bit:** `bitand` , `bitor` , `bitxor` .
- **Funciones matemáticas elementales:** `max` , `min` , `mod` , `rem` , `hypot` , `atan2` , `atan2d` .

Las operaciones binarias mencionadas anteriormente están permitidas entre matrices, siempre que tengan "tamaños compatibles". Los tamaños se consideran "compatibles" cuando cada dimensión en una matriz es exactamente igual a la misma dimensión en la otra matriz, o es igual a `1` . Tenga en cuenta que MATLAB omite las dimensiones finales de singleton (es decir, de tamaño `1`), aunque en teoría hay una cantidad infinita de ellas. En otras palabras, las dimensiones que aparecen en una matriz y no aparecen en la otra, se ajustan implícitamente a la expansión automática.

Por ejemplo, en las versiones de MATLAB **anteriores a R2016b,** esto sucedería:

```
>> magic(3) + (1:3)
Error using  +
Matrix dimensions must agree.
```

Considerando que a **partir de R2016b** la operación anterior tendrá éxito:

```
>> magic(3) + (1:3)
ans =

     9     3     9
     4     7    10
     5    11     5
```

Ejemplos de tamaños compatibles:

Descripción	1er tamaño de la matriz	2do tamaño de la matriz	Tamaño del resultado
Vector y escalar	`[3x1]`	`[1x1]`	`[3x1]`
Vectores de fila y columna	`[1x3]`	`[2x1]`	`[2x3]`
Vector y matriz 2D	`[1x3]`	`[5x3]`	`[5x3]`
Arreglos ND y KD	`[1x3x3]`	`[5x3x1x4x2]`	`[5x3x3x4x2]`

Ejemplos de tamaños incompatibles:

Descripción	1er tamaño de la matriz	2do tamaño de la matriz	Posible solución
Vectores donde una dimensión es un múltiplo de la misma dimensión en la otra matriz.	`[1x2]`	`[1x8]`	`transpose`
Arreglos con dimensiones que son múltiplos entre sí.	`[2x2]`	`[8x8]`	`repmat` , `reshape`
Arrays de ND que tienen la cantidad correcta de dimensiones singleton pero están en el orden incorrecto (# 1).	`[2x3x4]`	`[2x4x3]`	`permute`
Arrays ND que tienen la cantidad correcta de dimensiones singleton pero están en el orden incorrecto (# 2).	`[2x3x4x5]`	`[5x2]`	`permute`

IMPORTANTE:
Código confiar en esta convención **no** es compatible hacia atrás con *alguna* versión anterior de MATLAB. Por lo tanto, la invocación explícita de `bsxfun` [1] , [2] (que logra el mismo efecto) debe usarse si el código necesita ejecutarse en versiones anteriores de MATLAB. Si no existe tal preocupación, las notas de la versión de MATLAB R2016 animan a los usuarios a cambiar de `bsxfun` :

> *En comparación con el uso de `bsxfun` , la expansión implícita ofrece una velocidad de ejecución más rápida, un mejor uso de la memoria y una mejor legibilidad del código.*

Lectura relacionada:

- Documentación de MATLAB sobre " *Tamaños de matriz compatibles para operaciones básicas* ".
- NumPy Broadcasting [1] , [2] .
- Una comparación entre la velocidad de computación usando `bsxfun` contra la expansión de matriz implícita .

Obtener el valor de una función de dos o más argumentos.

En muchas aplicaciones es necesario calcular la función de dos o más argumentos.

Tradicionalmente, utilizamos `for` -loops. Por ejemplo, si necesitamos calcular `f = exp(-x^2-y^2)` (no use esto si necesita **simulaciones rápidas**):

```
% code1
x = -1.2:0.2:1.4;
y = -2:0.25:3;
for nx=1:lenght(x)
   for ny=1:lenght(y)
      f(nx,ny) = exp(-x(nx)^2-y(ny)^2);
   end
```

```
end
```

Pero la versión vectorizada es más elegante y más rápida:

```
% code2
[x,y] = ndgrid(-1.2:0.2:1.4, -2:0.25:3);
f = exp(-x.^2-y.^2);
```

de lo que podemos visualizarlo:

```
surf(x,y,f)
```

Nota 1 - Cuadrículas: por lo general, el almacenamiento de matriz se organiza *fila por fila* . Pero en el MATLAB, es el almacenamiento *columna por columna* como en FORTRAN. Por lo tanto, hay dos funciones simulares `ndgrid` y `meshgrid` en MATLAB para implementar los dos modelos mencionados anteriormente. Para visualizar la función en el caso de `meshgrid` , podemos utilizar:

```
surf(y,x,f)
```

Nota 2 - El consumo de memoria: Que tamaño de `x` o `y` es 1000. Por lo tanto, necesitamos almacenar `1000*1000+2*1000 ~ 1e6` elementos para ***Code1*** no vectorizado. Pero necesitamos `3*(1000*1000) = 3e6` elementos en el caso de ***código*** vectorizado2. En el caso 3D (si `z` tiene el mismo tamaño que `x` o `y`), el consumo de memoria aumenta dramáticamente: `4*(1000*1000*1000)` (~ 32GB para dobles) en el caso del ***código*** vectorizado 2 vs `~1000*1000*1000` (solo ~ 8GB) en el caso de ***code1*** . Por lo tanto, tenemos que elegir entre la memoria o la velocidad.

Lea Vectorización en línea: https://riptutorial.com/es/matlab/topic/750/vectorizacion

Creditos

S. No	Capítulos	Contributors
1	Empezando con MATLAB Language	adjpayot, Amro, chrisb2244, Christopher Creutzig, Community, Dan, Dev-iL, DVarga, EBH, Erik, excaza, flawr, Franck Dernoncourt, fyrepenguin, GameOfThrows, H. Pauwelyn, honi, Landak, Lior, Matt, Mikhail_Sam, Mohsen Nosratinia, Sam Roberts, Shai, Tyler
2	Aplicaciones financieras	Amro, Franck Dernoncourt, Mikhail_Sam, Oleg, Royi, StefanM
3	Características no documentadas	Amro, codeaviator, Dev-iL, Erik, matlabgui, thewaywewalk
4	Condiciones	ammportal, EBH
5	Controlando la coloración de la subparcela en Matlab	Noa Regev
6	Depuración	Dev-iL, DVarga, jenszvs, Justin
7	Dibujo	il_raffa, nitsua60, NKN, thewaywewalk, Trilarion, Zep
8	Errores comunes y errores.	Amro, Ander Biguri, Dev-iL, EBH, edwinksl, erfan, Franck Dernoncourt, Hoki, Landak, Malick, Matt, Mikhail_Sam, Mohsen Nosratinia, nahomyaja, NKN, Oleg, R. Joiny, rafa, rayryeng, Rody Oldenhuis, S. Radev, Sardar Usama, strpeter, Suever, Thierry Dalon, Tim, Umar
9	Establecer operaciones	Shai, StefanM
10	Funciones	Batsu
11	Funciones de documentacion	alexforrence, Ander Biguri, ceiltechbladhm, Dev-iL, Eric, excaza
12	Gráficos: Transformaciones 2D y 3D	itzik Ben Shabat, Royi
13	Gráficos: trazos de	Amro, Celdor, EBH, Erik, Matt, Oleg, StefanM

	líneas 2D	
14	Inicializando matrices o matrices	Parag S. Chandakkar, rajah9, Théo P.
15	Integración	StefanM
16	Interfaces de usuario MATLAB	Dev-iL, Hoki, Royi, Suever, Zep
17	Interpolación con MATLAB	Royi, StefanM
18	Introducción a la API de MEX	Amro, Ander Biguri, Kamiccolo, Matt, mike
19	Leyendo archivos grandes	JCKaz
20	Matrices de descomposiciones	StefanM
21	Mejores Prácticas de MATLAB	il_raffa, Malick, MayeulC, McLemon, mnoronha, NKN, rayryeng, Sardar Usama, Thierry Dalon
22	Multihilo	Adriaan, daren shan, Franck Dernoncourt, Hardik_Jain, Jim, Peter Mortensen
23	Para bucles	agent_C.Hdj, drhagen, EBH, Tyler
24	Procesamiento de imágenes	A.Youssouf, Ander Biguri, Cape Code, girish_m, Royi, Shai, Trilarion
25	Programación orientada a objetos	alexforrence, Celdor, daren shan, Dev-iL, jenszvs, Mohsen Nosratinia, Trogdor
26	Rendimiento y Benchmarking	Celdor, daleonpz, Dev-iL, il_raffa, Oleg, pseudoDust, thewaywewalk
27	Solucionadores de Ecuaciones Diferenciales Ordinarias (EDO)	Royi, StefanM
28	Transformadas de Fourier y Transformadas de Fourier inversas	GameOfThrows, Landak
29	Trucos utiles	Celdor, EBH, Erik, fyrepenguin, Malick

30	Usando funciones con salida lógica.	Landak, Mikhail_Sam, Mohsen Nosratinia, S. Radev, Trilarion
31	Usando puertos seriales	Abdul Rehman, Ander Biguri, Hoki, Kenn Sebesta, mhopeng, Oleg
32	Uso de la función "úmarray () `	Dev-iL, Royi
33	Utilidades de programación	Celdor, The Vivandiere
34	Vectorización	Alexander Korovin, Ander Biguri, Dan, Dev-iL, EBH, Landak, Matt, Oleg, Shai, Tyler