MATLAB®

Notes for Professionals

100+ pages

of professional hints and tricks

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/MATLABBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with MATLAB Language

Version	Release	Release Date
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	Volume 8	1996-12-01
5.1	Volume 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
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

See also: MATLAB release history on Wikipedia.

Section 1.1: Indexing matrices and arrays

MATLAB allows for several methods to index (access) elements of matrices and arrays:

- **Subscript indexing** - where you specify the position of the elements you want in each dimension of the matrix separately.
- **Linear indexing** - where the matrix is treated as a vector, no matter its dimensions. That means, you specify each position in the matrix with a single number.
- **Logical indexing** - where you use a logical matrix (and matrix of `true` and `false` values) with the identical dimensions of the matrix you are trying to index as a mask to specify which value to return.

These three methods are now explained in more detail using the following 3-by-3 matrix `M` as an example:

```
>> M = magic(3)

ans =

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

Subscript indexing

The most straight-forward method for accessing an element, is to specify its row-column index. For example, accessing the element on the second row and third column:

```
>> M(2, 3)

ans =

     7
```

The number of subscripts provided exactly matches the number of dimensions `M` has (two in this example).

Note that the order of subscripts is the same as the mathematical convention: row index is the first. Moreover, MATLAB indices **starts with `1`** and **not `0`** like most programming languages.

You can index multiple elements at once by passing a vector for each coordinate instead of a single number. For example to get the entire second row, we can specify that we want the first, second and third columns:

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

ans =

     3     5     7
```

In MATLAB, the vector `[1,2,3]` is more easily created using the colon operator, i.e. `1:3`. You can use this in indexing as well. To select an entire row (or column), MATLAB provides a shortcut by allowing you just specify `:`. For example, the following code will also return the entire second row

```
>> M(2, :)

ans =

       3     5     7
```

MATLAB also provides a shortcut for specifying the last element of a dimension in the form of the `end` keyword. The `end` keyword will work exactly as if it was the number of the last element in that dimension. So if you want all the columns from column 2 to the last column, you can use write the following:

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

ans =

       5     7
```

Subscript indexing can be restrictive as it will not allow to extract single values from different columns and rows; it will extract the combination of all rows and columns.

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

       3     7
       4     2
```

For example subscript indexing cannot extract only the elements `M(2,1)` or `M(3,3)`. To do this we must consider linear indexing.

Linear indexing

MATLAB allows you to treat n-dimensional arrays as one-dimensional arrays when you index using only one dimension. You can directly access the first element:

```
>> M(1)

ans =

       8
```

Note that arrays are stored in column-major order in MATLAB which means that you access the elements by first going down the columns. So `M(2)` is the second element of the first column which is `3` and `M(4)` will be the first element of the second column i.e.

```
>> M(4)

ans =

        1
```

There exist built-in functions in MATLAB to convert subscript indices to linear indices, and vice versa: `sub2ind` and `ind2sub` respectively. You can manually convert the subscripts (`r`,`c`) to a linear index by

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

To understand this, if we are in the first column then the linear index will simply be the row index. The formula above holds true for this because for `c == 1`, `(c-1) == 0`. In the next columns, the linear index is the row number

plus all the rows of the previous columns.

Note that the `end` keyword still applies and now refers to the very last element of the array i.e. `M(end) == M(end, end) == 2`.

You can also index multiple elements using linear indexing. Note that if you do that, the returned matrix will have the same shape as the matrix of index vectors.

`M(2:4)` returns a row vector because `2:4` represents the row vector `[2,3,4]`:

```
>> M(2:4)

ans =

     3     4     1
```

As another example, `M([1,2;3,4])` returns a 2-by-2 matrix because `[1,2;3,4]` is a 2-by-2 matrix as well. See the below code to convince yourself:

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

ans =

     8     3
     4     1
```

Note that indexing with `:` alone will *always* return a column vector:

```
>> M(:)

ans =

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

This example also illustrates the order in which MATLAB returns elements when using linear indexing.

Logical indexing

The third method of indexing is to use a logical matrix, i.e. a matrix containing only `true` or `false` values, as a mask to filter out the elements you don't want. For example, if we want to find all the elements of `M` that are greater than `5` we can use the logical matrix

```
>> M > 5

ans =

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

to index `M` and return only the values that are greater than `5` as follows:

```
>> M(M > 5)

ans =

        8
        9
        6
        7
```

If you wanted these number to stay in place (i.e. keep the shape of the matrix), then you could assign to the logic compliment

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

ans =

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

We can reduce complicated code blocks containing `if` and `for` statements by using logical indexing.

Take the non-vectorized (already shortened to a single loop by using linear indexing):

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

This can be shortened to the following code using logical indexing:

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

Or even shorter:

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

More on indexing

Higher dimension matrices

All the methods mentioned above generalize into n-dimensions. If we use the three-dimensional matrix `M3 = rand(3,3,3)` as an example, then you can access all the rows and columns of the second slice of the third dimension by writing

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

You can access the first element of the second slice using linear indexing. Linear indexing will only move on to the second slice after all the rows and all the columns of the first slice. So the linear index for that element is

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

In fact, in MATLAB, *every* matrix is n-dimensional: it just happens to be that the size of most of the other n-dimensions are one. So, if `a = 2` then `a(1) == 2` (as one would expect), *but also* `a(1, 1) == 2`, as does `a(1, 1, 1) == 2`, `a(1, 1, 1, ..., 1) == 2` and so on. These "extra" dimensions (of size 1), are referred to as *singleton dimensions*. The command `squeeze` will remove them, and one can use `permute` to swap the order of dimensions around (and introduce singleton dimensions if required).

An n-dimensional matrix can also be indexed using an m subscripts (where m<=n). The rule is that the first m-1 subscripts behave ordinarily, while the last (m'th) subscript references the remaining (n-m+1) dimensions, just as a linear index would reference an (n-m+1) dimensional array. Here is an example:

```
>> 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
```

Returning ranges of elements

With subscript indexing, if you specify more than one element in more than one dimension, MATLAB returns each possible pair of coordinates. For example, if you try M([1,2],[1,3]) MATLAB will return `M(1,1)` and `M(2,3)` but it will **also** return `M(1,3)` and `M(2,1)`. This can seem unintuitive when you are looking for the elements for a list of coordinate pairs but consider the example of a larger matrix, `A = rand(20)` (note `A` is now `20`-by-`20`), where you want to get the top right hand quadrant. In this case instead of having to specify every coordinate pair in that quadrant (and this this case that would be `100` pairs), you just specify the `10` rows and the `10` columns you want so `A(1:10, 11:end)`. *Slicing* a matrix like this is far more common than requiring a list of coordinate pairs.

In the event that you do want to get a list of coordinate pairs, the simplest solution is to convert to linear indexing. Consider the problem where you have a vector of column indices you want returned, where each row of the vector contains the column number you want returned for the *corresponding* row of the matrix. For example

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

So in this case you actually want to get back the elements at `(1,3)`, `(2,2)` and `(3,1)`. So using linear indexing:

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

ans =

        6     5     4
```

Returning an element multiple times

With subscript and linear indexing you can also return an element multiple times by repeating it's index so

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

ans =
```

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

You can use this to duplicate entire rows and column for example to repeat the first row and last column

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

ans =

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

For more information, see here.

Section 1.2: Anonymous functions and function handles

Basics

Anonymous functions are a powerful tool of the MATLAB language. They are functions that exist locally, that is: in the current workspace. However, they do not exist on the MATLAB path like a regular function would, e.g. in an m-file. That is why they are called anonymous, although they can have a name like a variable in the workspace.

The `@` operator

Use the `@` operator to create anonymous functions and function handles. For example, to create a handle to the `sin` function (sine) and use it as `f`:

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

Now `f` is a handle to the `sin` function. Just like (in real life) a door handle is a way to use a door, a function handle is a way to use a function. To use `f`, arguments are passed to it as if it were the `sin` function:

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

`f` accepts any input arguments the `sin` function accepts. If `sin` would be a function that accepts zero input arguments (which it does not, but others do, e.g. the `peaks` function), `f()` would be used to call it without input arguments.

Custom anonymous functions
Anonymous functions of one variable

It is not obviously useful to create a handle to an existing function, like `sin` in the example above. It is kind of redundant in that example. However, it is useful to create anonymous functions that do custom things that otherwise would need to be repeated multiple times or created a separate function for. As an example of a custom anonymous function that accepts one variable as its input, sum the sine and cosine squared of a signal:

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

Now `f` accepts one input argument called `x`. This was specified using parentheses `(...)` directly after the `@` operator. `f` now is an anonymous function of `x`: `f(x)`. It is used by passing a value of `x` to `f`:

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

A vector of values or a variable can also be passed to `f`, as long as they are used in a valid way within `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
```

Anonymous functions of more than one variable

In the same fashion anonymous functions can be created to accept more than one variable. An example of an anonymous function that accepts three 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
```

Parameterizing anonymous functions

Variables in the workspace can be used within the definition of anonymous functions. This is called parameterizing. For example, to use a constant `c = 2` in an anonymous function:

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

`f(3)` used the variable `c` as a parameter to multiply with the provided `x`. Note that if the value of `c` is set to something different at this point, then `f(3)` is called, the result would **not** be different. The value of `c` is the value *at the time of creation* of the anonymous function:

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

Input arguments to an anonymous function do not refer to workspace variables

Note that using the name of variables in the workspace as one of the input arguments of an anonymous function

(i.e., using `@(...)`) will **not** use those variables' values. Instead, they are treated as different variables within the scope of the anonymous function, that is: the anonymous function has its private workspace where the input variables never refer to the variables from the main workspace. The main workspace and the anonymous function's workspace do not know about each other's contents. An example to illustrate this:

```
>> 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
```

The value of `x` from the main workspace is not used within `f`. Also, in the main workspace `x` was left untouched. Within the scope of `f`, the variable names between parentheses after the `@` operator are independent from the main workspace variables.

Anonymous functions are stored in variables

An anonymous function (or, more precisely, the function handle pointing at an anonymous function) is stored like any other value in the current workspace: In a variable (as we did above), in a cell array (`{@(x)x.^2,@(x)x+1}`), or even in a property (like `h.ButtonDownFcn` for interactive graphics). This means the anonymous function can be treated like any other value. When storing it in a variable, it has a name in the current workspace and can be changed and cleared just like variables holding numbers.

Put differently: A function handle (whether in the `@sin` form or for an anonymous function) is simply a value that can be stored in a variable, just like a numerical matrix can be.

Advanced use
Passing function handles to other functions

Since function handles are treated like variables, they can be passed to functions that accept function handles as input arguments.

An example: A function is created in an m-file that accepts a function handle and a scalar number. It then calls the function handle by passing `3` to it and then adds the scalar number to the result. The result is returned.

Contents of `funHandleDemo.m`:

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

Save it somewhere on the path, e.g. in MATLAB's current folder. Now `funHandleDemo` can be used as follows, for example:

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

The handle of another existing function can be passed to `funHandleDemo`:

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

Notice how `@sin` was a quick way to access the `sin` function without first storing it in a variable using `f = @sin`.

Using `bsxfun`, `cellfun` and similar functions with anonymous functions

MATLAB has some built-in functions that accept anonymous functions as an input. This is a way to perform many calculations with a minimal number of lines of code. For example `bsxfun`, which performs element-by-element binary operations, that is: it applies a function on two vectors or matrices in an element-by-element fashion. Normally, this would require use of `for`-loops, which often requires preallocation for speed. Using `bsxfun` this process is sped up. The following example illustrates this using `tic` and `toc`, two functions that can be used to time how long code takes. It calculates the difference of every matrix element from the matrix column mean.

```
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
```

Running the example above results in two outputs:

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

These lines come from the `toc` functions, which print the elapsed time since the last call to the `tic` function.

The `bsxfun` call applies the function in the first input argument to the other two input arguments. `@minus` is a long name for the same operation as the minus sign would do. A different anonymous function or handle (`@`) to any other function could have been specified, as long as it accepts `A` and `mean(A)` as inputs to generate a meaningful result.

Especially for large amounts of data in large matrices, `bsxfun` can speed up things a lot. It also makes code look cleaner, although it might be more difficult to interpret for people who don't know MATLAB or `bsxfun`. (Note that in MATLAB R2016a and later, many operations that previously used `bsxfun` no longer need them; `A-mean(A)` works directly and can in some cases be even faster.)

Section 1.3: Matrices and Arrays

In MATLAB, the most basic data type is the numeric array. It can be a scalar, a 1-D vector, a 2-D matrix, or an N-D multidimensional array.

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

To create a row vector, enter the elements inside brackets, separated by spaces or commas:

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

To create a column vector, separate the elements with semicolons:

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

To create a matrix, we enter the rows as before separated by semicolons:

```
% 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];
```

Notice you cannot create a matrix with unequal row / column size. All rows must be the same length, and all columns must be the same length:

```
% 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
```

To transpose a vector or a matrix, we use the `.'`-operator, or the `'` operator to take its Hermitian conjugate, which is the complex conjugate of its transpose. For real matrices, these two are the same:

```
% 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]
```

For arrays of more than two-dimensions, there is no direct language syntax to enter them literally. Instead we must use functions to construct them (such as `ones`, `zeros`, `rand`) or by manipulating other arrays (using functions such as `cat`, `reshape`, `permute`). Some examples:

```
% 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]);
```

Section 1.4: Cell arrays

Elements of the same class can often be concatenated into arrays (with a few rare exceptions, e.g. function handles). Numeric scalars, by default of class `double`, can be stored in a matrix.

```
>> 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
```

Characters, which are of class `char` in MATLAB, can also be stored in array using similar syntax. Such an array is similar to a string in many other programming languages.

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

Note that despite both of them are using brackets `[` and `]`, the result classes are different. Therefore the operations that can be done on them are also different.

```
>> whos
  Name      Size             Bytes  Class     Attributes

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

In fact, the array `s` is not an array of the strings `'MATLAB '`,`'is '`, and `'fun'`, it is just one string - an array of 13 characters. You would get the same results if it were defined by any of the following:

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

A regular MATLAB vector does not let you store a mix of variables of different classes, or a few different strings. This is where the `cell` array comes in handy. This is an array of cells that each can contain some MATLAB object, whose class can be different in every cell if needed. Use curly braces `{` and `}` around the elements to store in a cell array.

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

  C         2x1                330  cell
```

Standard MATLAB objects of any classes can be stored together in a cell array. Note that cell arrays require more memory to store their contents.

Accessing the contents of a cell is done using curly braces `{` and `}`.

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

Note that `C(1)` is different from `C{1}`. Whereas the latter returns the cell's content (and has class `double` in out example), the former returns a cell array which is a sub-array of `C`. Similarly, if `D` were an 10 by 5 cell array, then `D(4:8,1:3)` would return a sub-array of `D` whose size is 5 by 3 and whose class is `cell`. And the syntax `C{1:2}` does not have a single returned object, but rather it returns 2 different objects (similar to a MATLAB function with multiple return values):

```
>> [x,y] = C{1:2}
x =
                         1                        -2                      3.14
  0.8                     15625
         3.14159265358979                       Inf                       Inf
  NaN                     -Inf
y =
MATLAB is fun
```

Section 1.5: Hello World

Open a new blank document in the MATLAB Editor (in recent versions of MATLAB, do this by selecting the Home tab of the toolstrip, and clicking on New Script). The default keyboard shortcut to create a new script is `Ctrl-n`.

Alternatively, typing `edit myscriptname.m` will open the file `myscriptname.m` for editing, or offer to create the file if it does not exist on the MATLAB path.

In the editor, type the following:

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

Select the Editor tab of the toolstrip, and click Save As. Save the document to a file in the current directory called `helloworld.m`. Saving an untitled file will bring up a dialog box to name the file.

In the MATLAB Command Window, type the following:

```
>> helloworld
```

You should see the following response in the MATLAB Command Window:

```
Hello, World!
```

We see that in the Command Window, we are able to type the names of functions or script files that we have written, or that are shipped with MATLAB, to run them.

Here, we have run the 'helloworld' script. Notice that typing the extension (`.m`) is unnecessary. The instructions held in the script file are executed by MATLAB, here printing 'Hello, World!' using the `disp` function.

Script files can be written in this way to save a series of commands for later (re)use.

Section 1.6: Scripts and Functions

MATLAB code can be saved in m-files to be reused. m-files have the `.m` extension which is automatically associated

with MATLAB. An m-file can contain either a script or functions.

Scripts

Scripts are simply program files that execute a series of MATLAB commands in a predefined order.

Scripts do not accept input, nor do scripts return output. Functionally, scripts are equivalent to typing commands directly into the MATLAB command window and being able to replay them.

An example of a script:

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

This script will define `length`, `width`, and `area` in the current workspace with the value `10`, `3`, and `30` respectively.

As stated before, the above script is functionally equivalent to typing the same commands directly into the command window.

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

Functions

Functions, when compared to scripts, are much more flexible and extensible. Unlike scripts, functions can accept input and return output to the caller. A function has its own workspace, this means that internal operations of the functions will not change the variables from the caller.

All functions are defined with the same header format:

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

The `function` keyword begins every function header. The list of outputs follows. The list of outputs can also be a comma separated list of variables to return.

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

Next is the name of the function that will be used for calling. This is generally the same name as the filename. For example, we would save this function as `myFunctionName.m`.

Following the function name is the list of inputs. Like the outputs, this can also be a comma separated list.

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

We can rewrite the example script from before as a reusable function like the following:

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

We can call functions from other functions, or even from script files. Here is an example of our above function being used in a script file.

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

As before, we create `l`, `w`, and `a` in the workspace with the values of `100`, `20`, and `2000` respectively.

Section 1.7: Helping yourself

MATLAB comes with many built-in scripts and functions which range from simple multiplication to image recognition toolboxes. In order to get information about a function you want to use type: `help functionname` in the command line. Let's take the `help` function as an example.

Information on how to use it can be obtained by typing:

```
>> help help
```

in the command window. This will return information of the usage of function `help`. If the information you are looking for is still unclear you can try the **documentation** page of the function. Simply type:

```
>> doc help
```

in the command window. This will open the browsable documentation on the page for function `help` providing all the information you need to understand how the 'help' works.

This procedure works for all built-in functions and symbols.

When developing your own functions you can let them have their own help section by adding comments at the top of the function file or just after the function declaration.

Example for a simple function `multiplyby2` saved in file `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
```

or

```
% 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
```

This is very useful when you pick up your code weeks/months/years after having written it.

The `help` and `doc` function provide a lot of information, learning how to use those features will help you progress rapidly and use MATLAB efficiently.

Section 1.8: Data Types

There are 16 fundamental data types, or classes, in MATLAB. Each of these classes is in the form of a matrix or array. With the exception of function handles, this matrix or array is a minimum of 0-by-0 in size and can grow to an

n-dimensional array of any size. A function handle is always scalar (1-by-1).

Important moment in MATLAB is that you don't need to use any type declaration or dimension statements by default. When you define new variable MATLAB creates it automatically and allocates appropriate memory space.

Example:

```
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
```

If the variable already exists, MATLAB replaces the original data with new one and allocates new storage space if necessary.

Fundamental data types

Fundamental data types are: numeric, `logical`, `char`, `cell`, `struct`, `table` and `function_handle`.

Numeric data types:

- Floating-Point numbers (*default*)

  MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function:

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

  >> whos
    Name      Size            Bytes  Class     Attributes

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

- Integers

  MATLAB has four signed and four unsigned integer classes. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

  MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you do not need a 32-bit integer to store the value 100.

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

  >> whos
  ```

```
  Name      Size            Bytes  Class    Attributes

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

To store data as an integer, you need to convert from double to the desired integer type. If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly `0.5`, then from the two equally nearby integers, MATLAB chooses the one for which the absolute value is larger in magnitude.

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

- char

  Character arrays provide storage for text data in MATLAB. In keeping with traditional programming terminology, an array (sequence) of characters is defined as a string. There is no explicit string type in retail releases of MATLAB.

- logical: logical values of 1 or 0, represent true and false respectively. Use for relational conditions and array indexing. Because it's just TRUE or FALSE it has size of 1 byte.

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

- structure. A structure array is a data type that groups variables of different data types using data containers called *fields*. Each field can contain any type of data. Access data in a structure using dot notation of the form structName.fieldName.

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

In order to access value1, each of the following syntax are equivalent

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

We can explicitly access a field we know will exist with the first method, or either pass a string or create a string to access the field in the second example. The third example is demonstrating that the dot parentheses notation takes a string, which is the same one stored in the field1 variable.

- table variables can be of different sizes and data types, but all variables must have the same number of rows.

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

- cell. It's very useful MATLAB data type: cell array is an array each element of it can be of different data type and size. It's very strong instrument for manipulating data as you wish.

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

or

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

- function handles stores a pointer to a function (for example, to anonymous function). It allows you to pass a function to another function, or call local functions from outside the main function.

There are a lot of instruments to work with each data type and also built-in data type conversion functions (`str2double`, `table2cell`).

Additional data types

There are several additional data types which are useful in some specific cases. They are:

- Date and time: arrays to represent dates, time, and duration. `datetime('now')` returns `21-Jul-2016 16:30:16`.
- Categorical arrays: it's data type for storing data with values from a set of discrete categories. Useful for storing nonnumeric data (memory effective). Can be used in a table to select groups of rows.

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

- Map containers is a data structure that has unique ability to indexing not only through the any scalar numeric values but character vector. Indices into the elements of a Map are called keys. These keys, along with the data values associated with them, are stored within the Map.
- Time series are data vectors sampled over time, in order, often at regular intervals. It's useful to store the data connected with timesteps and it has a lot of useful methods to work with.

Section 1.9: Reading Input & Writing Output

Just like all programming language, MATLAB is designed to read and write in a large variety of formats. The native library supports a large number of Text,Image,Video,Audio,Data formats with more formats included in each version update - check here to see the full list of supported file formats and what function to use to import them.

Before you attempt to load in your file, you must ask yourself what do you want the data to become and how you expect the computer to organize the data for you. Say you have a txt/csv file in the following format:

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

We can see that the first column is in the format of Strings, while the second, third are Numeric, the last column is in the form of Currency. Let's say we want to find how much revenue we made today using MATLAB and first we want to load in this txt/csv file. After checking the link, we can see that String and Numeric type of txt files are handled by `textscan`. So we could try:

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

where `%s` suggest that the element is a String type, `%f` suggest that the element is a Float type, and that the file is Delimited by ",". The HeaderLines option asks MATLAB to skip the First N lines while the 1 immediately after it

means to skip the first line (the header line).

Now C is the data we have loaded which is in the form of a Cell Array of 4 cells, each containing the column of data in the txt/csv file.

So first we want to calculate how many fruits we sold today by subtracting the third column from the second column, this can be done by:

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

Now we want to multiply this vector by the Price per unit, so first we need to convert that column of Strings into a column of Numbers, then convert it into a Numeric Matrix using MATLAB's `cell2mat` the first thing we need to do is to strip-off the "$" sign, there are many ways to do this. The most direct way is using a simple regex:

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

Or you can use a loop:

```
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
```

The `str2num` function turns the string which had "$" signs stripped into numeric types and `cell2mat` turns the cell of numeric elements into a matrix of numbers

Now we can multiply the units sold by the cost per unit:

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

totalrevenue = sum(revenue);
```

Chapter 2: Initializing Matrices or arrays

Parameter	Details
sz	n (for an n x n matrix)
sz	n, m (for an n x m matrix)
sz	m,n,...,k (for an m-by-n-by-...-by-k matrix)
datatype	'double' (default), 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'
arraytype	'distributed'
arraytype	'codistributed'
arraytype	'gpuArray'

MATLAB has three important functions to create matrices and set their elements to zeroes, ones, or the identity matrix. (The identity matrix has ones on the main diagonal and zeroes elsewhere.)

Section 2.1: Creating a matrix of 0s

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

Section 2.2: Creating a matrix of 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
```

Section 2.3: Creating an identity matrix

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

Chapter 3: Conditions

Parameter	Description
expression	an expression that has logical meaning

Section 3.1: IF condition

Conditions are a fundamental part of almost any part of code. They are used to execute some parts of the code only in some situations, but not other. Let's look at the basic syntax:

```
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
```

Output:

```
Not OK
```

In this example we see the `if` consists of 2 parts: the condition, and the code to run if the condition is true. The code is everything written after the condition and before the `end` of that `if`. The first condition was not fulfilled and hence the code within it was not executed.

Here is another example:

```
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
```

The condition above will always be true, and will display the output `It's true!`.

We can also write:

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

This time the condition is always false, so we will never get the output `Equal`.

There is not much use for conditions that are always true or false, though, because if they are always false we can simply delete this part of the code, and if they are always true then the condition is not needed.

Section 3.2: IF-ELSE condition

In some cases we want to run an alternative code if the condition is false, for this we use the optional `else` part:

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

```
else
    disp('a bigger than 10')
end
```

Here we see that because `a` is not smaller than `10` the second part of the code, after the `else` is executed and we get the output `a bigger than 10`. Now let's look at another try:

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

In this example shows that we did not checked if `a` is indeed smaller than 10, and we get a wrong message because the condition only check the expression as it is, and ANY case that does not equals true (`a = 10`) will cause the second part to be executed.

This type of error is a very common pitfall for both beginners and experienced programmers, especially when conditions become complex, and should be always kept in mind

Section 3.3: IF-ELSEIF condition

Using `else` we can perform some task when the condition is not satisfied. But what if we want to check a second condition in case that the first one was false. We can do it this way:

```
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
```

This is also called "nested condition", but here we have a special case that can improve code readability, and reduce the chance for an error - we can write:

```
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
```

using the `elseif` we are able to check another expression within the same block of condition, and this is not limited to one try:

```
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
```

Extra care should be taken when choosing to use `elseif` in a row, since **only one** of them will be executed from all the `if` to `end` block. So, in our example if we want to display all the divisors of `a` (from those we explicitly check) the example above won't be good:

```
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
```

not only 3, but also 5 is a divisor of 15, but the part that check the division by 5 is not reached if any of the expressions above it was true.

Finally, we can add one `else` (and only **one**) after all the `elseif` conditions to execute a code when none of the conditions above are met:

```
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
```

Section 3.4: Nested conditions

When we use a condition within another condition we say the conditions are "nested". One special case of nested conditions is given by the `elseif` option, but there are numerous other ways to use nested conditions. Let's examine the following code:

```
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
```

For `a=2`, the output will be `a is even`, which is correct. For `a=3`, the output will be `a is odd`, which is also correct, but misses the check if 3 is a divisor of `a`. This is because the conditions are nested, so **only** if the first is `true`, than we move to the inner one, and if `a` is odd, none of the inner conditions are even checked. This is somewhat opposite to the use of `elseif` where only if the first condition is `false` than we check the next one. What about checking the division by 5? only a number that has 6 as a divisor (both 2 and 3) will be checked for the division by 5, and we can test and see that for `a=30` the output is:

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

We should also notice two things:

1. The position of the `end` in the right place for each `if` is crucial for the set of conditions to work as expected, so indentation is more than a good recommendation here.
2. The position of the `else` statement is also crucial, because we need to know in which `if` (and there could be several of them) we want to do something in case the expression if `false`.

Let's look at another example:

```
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
```

And the output will be:

```
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
```

we see that we got only 6 lines for 6 numbers, because the conditions are nested in a way that ensure only one print per number, and also (although can't be seen directly from the output) no extra checks are preformed, so if a number is not even there is no point to check if 4 is one of it divisors.

Chapter 4: Functions

Section 4.1: nargin, nargout

In the body of a function `nargin` and `nargout` indicate respectively the actual number of input and output supplied in the call.

We can for example control the execution of a function based on the number of provided input.

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
```

In a similar way we can control the execution of a function based on the number of output parameters.

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
```

Chapter 5: Set operations

Parameter	Details
A,B	sets, possibly matrices or vectors
x	possible element of a set

Section 5.1: Elementary set operations

It's possible to perform elementary set operations with MATLAB. Let's assume we have given two vectors or arrays

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

and we want to find all elements which are in `A` and in `B`. For this we can use

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

`C` will include all numbers which are part of `A` and part of `B`. If we also want to find the position of these elements we call

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

`pos` is the position of these elements such that `C == A(pos)`.

Another basic operation is the union of two sets

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

Herby contains `D` all elements of `A` and `B`.

Note that `A` and `B` are hereby treated as sets which means that it does not matter how often an element is part of `A` or `B`. To clarify this one can check `D == union(D,C)`.

If we want to obtain the data that is in 'A' but not in 'B' we can use the following function

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

We want to note again that this are sets such that following statement holds `D == union(E,B)`.

Suppose we want to check if

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

is an element of either `A` or `B` we can execute the command

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

If `a==1` then `x` is element of `A` and `x` is no element is `a==0`. The same goes for `B`. If `a==1 && b==1` `x` is also an element of `C`. If `a == 1 || b == 1` `x` is element of `D` and if `a == 1 || b == 0` it's also element of `E`.

Chapter 6: Documenting functions

Section 6.1: Obtaining a function signature

It is often helpful to have MATLAB print the 1st line of a function, as this usually contains the function signature, including inputs and outputs:

```
dbtype <functionName> 1
```

Example:

```
>> dbtype fit 1

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

Section 6.2: Simple Function Documentation

```
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` then provides:

> **mymult** Multiply two numbers.
>
> output = **mymult**(a, b) multiplies a and b.
>
> See also fft, foo, sin.
>
> For more information, see Google.

`fft` and `sin` automatically link to their respective help text, and Google is a link to google.com. `foo` will not link to any documentation in this case, as long as there is not a documented function/class by the name of `foo` on the search path.

Section 6.3: Local Function Documentation

In this example, documentation for the local function `baz` (defined in `foo.m`) can be accessed either by the resulting link in `help foo`, or directly through `help foo>baz`.

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

% This won't be printed, because there is a line without % on it.
end
```

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

Section 6.4: Documenting a Function with an Example Script

To document a function, it is often helpful to have an example script which uses your function. The publish function in MATLAB can then be used to generate a help file with embedded pictures, code, links, etc. The syntax for documenting your code can be found here.

The Function This function uses a "corrected" FFT in MATLAB.

```
function out_sig = myfft(in_sig)

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

end
```

The Example Script This is a separate script which explains the inputs, outputs, and gives an example explaining why the correction is necessary. Thanks to Wu, Kan, the original author of this function.

```
%% 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-m
atlab-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 above.
%%
% 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]);
```

The Output The publish option can be found under the "Publish" tab, highlighted in the imageSimple Function Documentation below.

MATLAB will run the script, and save the images which are displayed, as well as the text generated by the command line. The output can be saved to many different types of formats, including HTML, Latex, and PDF.

The output of the example script given above can be seen in the image below.

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

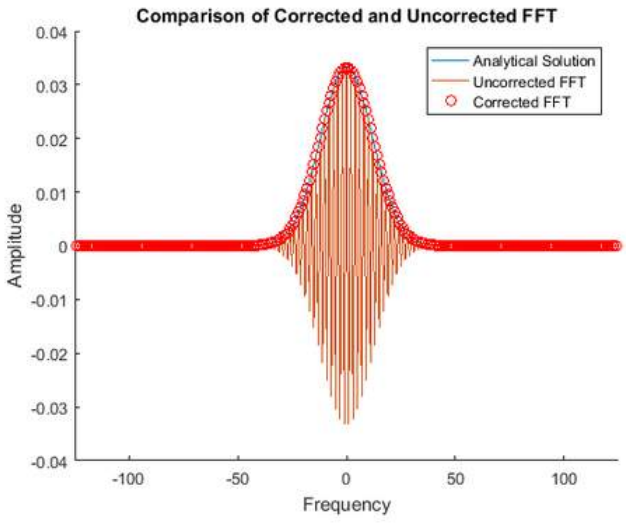
```
Bx = 10; A = sqrt(log(2))/(2*pi*Bx);    %Characteristics of Gaussian curve
x = exp(-t.^2/2/A^2);                   %Create Gaussian Curve
```

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]);
```



Published with MATLAB® R2016a

Chapter 7: Using functions with logical output

Section 7.1: All and Any with empty arrays

Special care needs to be taken when there is a possibility that an array become an empty array when it comes to logical operators. It is often expected that if `all(A)` is true then `any(A)` must be true and if `any(A)` is false, `all(A)` must also be false. That is not the case in MATLAB with empty arrays.

```
>> any([])
ans =
     0
>> all([])
ans =
     1
```

So if for example you are comparing all elements of an array with a certain threshold, you need to be aware of the case where the array is empty:

```
>> A=1:10;
>> all(A>5)
ans =
     0
>> A=1:0;
>> all(A>5)
ans =
     1
```

Use the built-in function `isempty` to check for empty arrays:

```
a = [];
isempty(a)
ans =
1
```

Chapter 8: For loops

Section 8.1: Iterate over columns of matrix

If the right-hand side of the assignment is a matrix, then in each iteration the variable is assigned subsequent columns of this matrix.

```
some_matrix = [1, 2, 3; 4, 5, 6]; % 2 by 3 matrix
for some_column = some_matrix
    display(some_column)
end
```

(The row vector version is a normal case of this, because in MATLAB a row vector is just a matrix whose columns are size 1.)

The output would display

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

i.e. each column of the iterated matrix displayed, each column printed on each call of `display`.

Section 8.2: Notice: Weird same counter nested loops

This is not something you will see in other programming environments. I came across it some years back and I couldn't understand why it was happening, but after working with MATLAB for some time I was able to figure it out. Look at the code snippet below:

```
for x = 1:10
    for x = 1:10
        fprintf('%d,', x);
    end
    fprintf('\n');
end
```

you wouldn't expect this to work properly but it does, producing the following output:

```
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,
```

The reason is that, as with everything else in MATLAB, the `x` counter is also a matrix—a vector to be precise. As such, `x` is only a reference to an 'array' (a coherent, consecutive memory structure) which is appropriately

referenced with every consequent loop (nested or not). The fact that the nested loop uses the same identifier makes no difference to how values from that array are referenced. The only problem is that within the nested loop the outer `x` is hidden by the nested (local) `x` and therefore cannot be referenced. However, the functionality of the nested loop structure remains intact.

Section 8.3: Iterate over elements of vector

The right-hand side of the assignment in a `for` loop can be any row vector. The left-hand side of the assignment can be any valid variable name. The `for` loop assigns a different element of this vector to the variable each run.

```
other_row_vector = [4, 3, 5, 1, 2];
for any_name = other_row_vector
    display(any_name)
end
```

The output would display

```
4
3
5
1
2
```

(The `1:n` version is a normal case of this, because in MATLAB `1:n` is just syntax for constructing a row vector of `[1, 2, ..., n]`.)

Hence, the two following blocks of code are identical:

```
A = [1 2 3 4 5];
for x = A
  disp(x);
end
```

and

```
for x = 1:5
  disp(x);
end
```

And the following are identical as well:

```
A = [1 3 5 7 9];
for x = A
  disp(x);
end
```

and

```
for x = 1:2:9
  disp(x);
end
```

Any row vector will do. They don't have to be numbers.

```
my_characters = 'abcde';
```

```
for my_char = my_characters
    disp(my_char)
end
```

will output

```
a
b
c
d
e
```

Section 8.4: Nested Loops

Loops can be nested, to preform iterated task within another iterated task. Consider the following loops:

```
ch = 'abc';
m = 3;
for c = ch
    for k = 1:m
        disp([c num2str(k)]) % NUM2STR converts the number stored in k to a character,
                             % so it can be concatenated with the letter in c
    end
end
```

we use 2 iterators to display all combinations of elements from `abc` and `1:m`, which yields:

```
a1
a2
a3
b1
b2
b3
c1
c2
c3
```

We can also use nested loops to combine between tasks to be done each time, and tasks to be done once in a several iterations:

```
N = 10;
n = 3;
a1 = 0; % the first element in Fibonacci series
a2 = 1; % the second 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
```

Here we want to compute all the Fibonacci series, but to display only the `n`th element each time, so we get

```
3
13
55
233
987
4181
17711
75025
317811
1346269
```

Another thing we can do is to use the first (outer) iterator within the inner loop. Here is another example:

```
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
```

This time we use the nested loop to format the output, and brake the line only when a new gap (`j`) between the elements was introduced. We loop through the gap width in the outer loop and use it within the inner loop to iterate through the 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
```

Section 8.5: Loop 1 to n

The simplest case is just preforming a task for a fixed known number of times. Say we want to display the numbers between 1 to n, we can write:

```
n = 5;
for k = 1:n
    display(k)
end
```

The loop will execute the inner statement(s), everything between the `for` and the `end`, for `n` times (5 in this example):

```
1
2
3
4
5
```

Here is another example:

```
n = 5;
for k = 1:n
```

```
    disp(n-k+1:-1:1) % DISP uses more "clean" way to print on the screen
end
```

this time we use both the `n` and `k` in the loop, to create a "nested" display:

```
5     4     3     2     1

4     3     2     1

3     2     1

2     1

1
```

Section 8.6: Loop over indexes

```
my_vector = [0, 2, 1, 3, 9];
for i = 1:numel(my_vector)
    my_vector(i) = my_vector(i) + 1;
end
```

Most simple things done with `for` loops can be done faster and easier by vectorized operations. For example, the above loop can be replaced by `my_vector = my_vector + 1`.

Chapter 9: Object-Oriented Programming

Section 9.1: Value vs Handle classes

Classes in MATLAB are divided into two major categories: value classes and handle classes. The major difference is that when copying an instance of a value class, the underlying data is copied to the new instance, while for handle classes the new instance points to the original data and changing values in new instance changes them in the original. A class can be defined as a handle by inheriting from the `handle` class.

```
classdef valueClass
    properties
        data
    end
end
```

and

```
classdef handleClass < handle
    properties
        data
    end
end
```

then

```
>> 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
```

Section 9.2: Constructors

A constructor is a special method in a class that is called when an instance of an object is created. It is a regular MATLAB function that accepts input parameters but it also must follow certain rules.

Constructors are not required as MATLAB creates a default one. In practice, however, this is a place to define a state of an object. For example, properties can be restricted by specifying attributes. Then, a constructor can initalize such properties by default or user defined values which in fact can sent by input parameters of a constructor.

Calling a constructor of a simple class

This is a simple class `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
```

The name of a constructor is the same the name of a class. Consequently, constructors are called by the name of its class. A class `Person` can be created as follows:

```
>> p = Person('John','Smith','London')
p =
  Person with properties:

       name: 'John'
    surname: 'Smith'
    address: 'London'
```

Calling a constructor of a child class

Classes can be inherited from parent classes if the share common properties or methods. When a class is inherited from another, it is likely that a constructor of a parent class has to be called.

A class `Member` inherits from a class `Person` because `Member` uses the same properties as the class Person but it also adds `payment` to its definition.

```
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
```

Similarly to the class `Person`, `Member` is created by calling its constructor:

```
>> m = Member('Adam','Woodcock','Manchester',20)
m =
  Member with properties:

    payment: 20
       name: 'Adam'
    surname: 'Woodcock'
    address: 'Manchester'
```

A constructor of `Person` requires three input parameters. `Member` must respect this fact and therefore call a constructor of the class `Person` with three parameters. It is fulfilled by the line:

```
obj = obj@Person(name,surname,address);
```

The example above shows the case when a child class needs information for its parent class. This is why a constructor of `Member` requires four parameters: three for its parent class and one for itself.

Section 9.3: Defining a class

A class can be defined using `classdef` in an `.m` file with the same name as the class. The file can contain the `classdef`...`end` block and local functions for use within class methods.

The most general MATLAB class definition has the following structure:

```
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
```

MATLAB Documentation: Class attributes, Property attributes, Method attributes, Event attributes, Enumeration class restrictions.

Example class:

A class called `Car` can be defined in file `Car.m` as

```
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
```

Note that the constructor is a method with the same name as the class. <A constructor is a special method of a class or structure in object-oriented programming that initializes an object of that type. A constructor is an instance method that usually has the same name as the class, and can be used to set the values of the members of an object, either to default or to user-defined values.>

An instance of this class can be created by calling the constructor;

```
>> myCar = Car('Ford', 'Mustang'); //creating an instance of car class
```

Calling the `drive` method will increment the mileage

```
>> myCar.mileage

    ans =
            0

>> myCar.drive(450);

>> myCar.mileage

   ans =
            450
```

Section 9.4: Inheriting from classes and abstract classes

Disclaimer: the examples presented here are only for the purpose of showing the use of abstract classes and inheritance and may not necessarily be of a practical use. Also, there is no such thing as polymorphic in MATLAB and therefore the use of abstract classes is limited. This example is to show who to create a class, inherit from another class and apply an abstract class to define a common interface.

The use of abstract classes is rather limited in MATLAB but it still can come useful on a couple of occasions.

Let's say we want a message logger. We might create a class similar to the one below:

```
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
```

Properties and methods

In short, properties hold a state of an object whilst methods are like interface and define actions on objects.

The property `scrh` is protected. This is why it must be initialized in a constructor. There are other methods (getters) to access this property but it is out of cope of this example. Properties and methods can be access via a variable that holds a reference to an object by using dot notation followed by a name of a method or a property:

```
mylogger = ScreenLogger(1);                          % OK
mylogger.LogMessage('My %s %d message', 'very', 1); % OK
mylogger.scrh = 2;                                   % ERROR!!! Access denied
```

Properties and methods can be public, private, or protected. In this case, protected means that I will be able to access to `scrh` from an inherited class but not from outside. By default all properties and methods are public. Therefore `LogMessage()` can freely be used outside the class definition. Also `LogMessage` defines an interface meaning this is what we must call when we want an object to log our custom messages.

Application

Let's say I have a script where I utilize my logger:

```
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');
```

If I have multiple places where I use the same logger and then want to change it to something more sophisticated, such as write a message in a file, I would have to create another object:

```
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
```

and just change one line of a code into this:

```
clc;
% ... a code
logger = DeepLogger('mymessages.log');
```

The above method will simply open a file, append a message at the end of the file and close it. At the moment, to be consistent with my interface, I need to remember that the name of a method is `LogMessage()` but it could equally be anything else. MATLAB can force developer to stick to the same name by using abstract classes. Let's say we define a common interface for any logger:

```
classdef MessageLogger
    methods(Abstract=true)
        LogMessage(obj, varargin);
    end
end
```

Now, if both `ScreenLogger` and `DeepLogger` inherit from this class, MATLAB will generate an error if `LogMessage()` is not defined. Abstract classes help to build similar classes which can use the same interface.

For the sake of this example, I will make slightly different change. I am going to assume that DeepLogger will do both logging message on a screen and in a file at the same time. Because `ScreenLogger` already log messages on screen, I am going to inherit `DeepLogger` from the `ScreenLoggger` to avoid repetition. `ScreenLogger` doesn't change at all apart from the first line:

```
classdef ScreenLogger < MessageLogger
// the rest of previous code
```

However, `DeepLogger` needs more changes in the `LogMessage` method:

```
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
```

Firstly, I simply initialize properties in the constructor. Secondly, because this class inherits from `ScreenLogger` I have to initialize this parent object as well. This line is even more important because `ScreenLogger` constructor requires one parameter to initalize its own object. This line:

```
obj = obj@ScreenLogger(screenhandler);
```

simply says "call the constructor of ScreenLogger and initalize it with a screen handler". It is worth noting here that I have defined `scrh` as protected. Therefore, I could equally access this property from `DeepLogger`. If the property was defined as private. The only way to intialize it would be using the constructor.

Another change is in section `methods`. Again to avoid repetition, I call `LogMessage()` from a parent class to log a message on a screen. If I had to change anything to make improvements in screen logging, now I have to do it in one place. The rest code is the same as it is a part of `DeepLogger`.

Because this class also inherits from an abstract class `MessageLogger` I had to make sure that `LogMessage()` inside `DeepLogger` is also defined. Inheriting from `MessageLogger` is a little bit tricky here. I think it cases redefinition of `LogMessage` mandatory--my guess.

In terms of the code where the a logger is applied, thanks to a common interface in classes, I can rest assured that changing this one line in the whole code would not make any issues. The same messages will be log on screen as before but additionally the code will write such messages to a file.

```
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');
```

I hope these examples explained the use of classes, the use of inheritance, and the use of abstract classes.

PS. The solution for the above problem is one of many. Another solution, less complex, would be to make `ScreenLoger` to be a component of another logger like `FileLogger` etc. `ScreenLogger` would be held in one of the properties. Its `LogMessage` would simply call `LogMessage` of the `ScreenLogger` and show text on a screen. I have chosen more complex approach to rather show how classes work in MATLAB. The example code below:

```
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{:}); % <-------- the change here
                fid = fopen(obj.fullfname, 'a+t');
                fprintf(fid, '%s\n', sprintf(varargin{:}));
                fclose(fid);
            end
        end
    end
end
```

Chapter 10: Vectorization

Section 10.1: Use of bsxfun

Quite often, the reason why code has been written in a `for` loop is to compute values from 'nearby' ones. The function `bsxfun` can often be used to do this in a more succinct fashion.

For example, assume that you wish to perform a columnwise operation on the matrix `B`, subtracting the mean of each column from it:

```
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
```

This method is inefficient if `B` is large, often due to MATLAB having to move the contents of variables around in memory. By using `bsxfun`, one can do the same job neatly and easily in just a single line:

```
A = bsxfun(@minus, B, mean(B));
```

Here, `@minus` is a function handle to the `minus` operator (`-`) and will be applied between elements of the two matrices `B` and `mean(B)`. Other function handles, even user-defined ones, are possible as well.

Next, suppose you want to add row vector `v` to each row in matrix `A`:

```
v = [1,  2,  3];

A = [8,  1,  6
     3,  5,  7
     4,  9,  2];
```

The naive approach is use a loop (*do **not** do this*):

```
B = zeros(3);
for row = 1:3
    B(row,:) = A(row,:) + v;
end
```

Another option would be to replicate `v` with `repmat` (*do **not** do this either*):

```
>> v = repmat(v,3,1)
v =
     1     2     3
     1     2     3
     1     2     3

>> B = A + v;
```

Instead use `bsxfun` for this task:

```
>> B = bsxfun(@plus, A, v);
B =
     9     3     9
     4     7    10
```

```
     5    11     5
```

Syntax

`bsxfun(@fun, A, B)`

where `@fun` is one of the supported functions and the two arrays `A` and `B` respect the two conditions below.

The name `bsxfun` helps to understand how the function works and it stands for **B**inary **FUN**ction with **S**ingleton e**X**pansion. In other words, if:

1. two arrays share the same dimensions except for one
2. and the discordant dimension is a singleton (i.e. has a size of `1`) in either of the two arrays

then the array with the singleton dimension will be expanded to match the dimension of the other array. After the expansion, a binary function is applied elementwise on the two arrays.

For example, let `A` be an `M`-by-`N`-by`K` array and `B` is an `M`-by-`N` array. Firstly, their first two dimensions have corresponding sizes. Secondly, `A` has `K` layers while `B` has implicitly only `1`, hence it is a singleton. All **conditions** are met and `B` will be replicated to match the 3rd dimension of `A`.

In other languages, this is commonly referred to as *broadcasting* and happens automatically in Python (numpy) and Octave.

The function, `@fun`, must be a binary function meaning it must take exactly two inputs.

Remarks

Internally, `bsxfun` does not replicate the array and executes an efficient loop.

Section 10.2: Implicit array expansion (broadcasting) [R2016b]

MATLAB R2016b featured a generalization of its scalar expansion1,2 mechanism, to also support certain element-wise operations between *arrays* of different sizes, as long as their dimension are compatible. The operators that support implicit expansion are1:

- **Element-wise arithmetic operators:** `+`, `-`, `.*`, `.^`, `./`, `.\`.
- **Relational operators:** `<`, `<=`, `>`, `>=`, `==`, `~=`.
- **Logical operators:** `&`, `|`, `xor`.
- **Bit-wise functions:** `bitand`, `bitor`, `bitxor`.
- **Elementary math functions:** `max`, `min`, `mod`, `rem`, `hypot`, `atan2`, `atan2d`.

The aforementioned binary operations are allowed between arrays, as long as they have "compatible sizes". Sizes are considered "compatible" when each dimension in one array is either exactly equal to the same dimension in the other array, or is equal to `1`. Note that trailing singleton (that is, of size 1) dimensions are omitted by MATLAB, even though there's theoretically an infinite amount of them. In other words - dimensions that appear in one array and do not appear in the other, are implicitly fit for automatic expansion.

For example, in MATLAB versions **before R2016b** this would happen:

```
>> magic(3) + (1:3)
Error using  +
Matrix dimensions must agree.
```

Whereas **starting from R2016b** the previous operation will succeed:

```
>> magic(3) + (1:3)
ans =

     9     3     9
     4     7    10
     5    11     5
```

Examples of compatible sizes:

Description	1st Array Size	2nd Array Size	Result Size
Vector and scalar	`[3x1]`	`[1x1]`	`[3x1]`
Row and column vectors	`[1x3]`	`[2x1]`	`[2x3]`
Vector and 2D matrix	`[1x3]`	`[5x3]`	`[5x3]`
N-D and K-D arrays	`[1x3x3]`	`[5x3x1x4x2]`	`[5x3x3x4x2]`

Examples of incompatible sizes:

Description	1st Array Size	2nd Array Size	Possible Workaround
Vectors where a dimension is a multiple of the same dimension in the other array.	`[1x2]`	`[1x8]`	`transpose`
Arrays with dimensions that are multiples of each other.	`[2x2]`	`[8x8]`	`repmat`, `reshape`
N-D arrays that have the right amount of singleton dimensions but they're in the wrong order (#1).	`[2x3x4]`	`[2x4x3]`	`permute`
N-D arrays that have the right amount of singleton dimensions but they're in the wrong order (#2).	`[2x3x4x5]`	`[5x2]`	`permute`

IMPORTANT:
Code relying on this convention is **NOT** backward-compatible with *any* older versions of MATLAB. Therefore, the explicit invocation of `bsxfun`[1,2] (which achieves the same effect) should be used if code needs to run on older MATLAB versions. If such a concern does not exist, MATLAB R2016 release notes encourage users to switch from `bsxfun`:

> *Compared to using `bsxfun`, implicit expansion offers faster speed of execution, better memory usage, and improved readability of code.*

Related reading:

- MATLAB documentation on "*Compatible Array Sizes for Basic Operations*".
- NumPy's Broadcasting[1,2].
- A comparison between the speed of computing using `bsxfun` vs. implicit array expansion.

Section 10.3: Element-wise operations

MATLAB supports (and encourages) vectorized operations on vectors and matrices.
For example, suppose we have `A` and `B`, two `n`-by-`m` matrices and we want `C` to be the element-wise product of the corresponding elements (i.e., `C(i,j) = A(i,j)*B(i,j)`).

The un-vectorized way, using nested loops is as follows:

```
C = zeros(n,m);
for ii=1:n
```

```
    for jj=1:m
        C(ii,jj) = A(ii,jj)*B(ii,jj);
    end
end
```

However, the vectorized way of doing this is by using the element-wise operator `.*`:

```
C = A.*B;
```

- For more information on the element-wise multiplication in MATLAB see the documentation of `times`.
- For more information about the difference between array and matrix operations see Array vs. Matrix Operations in the MATLAB documentation.

Section 10.4: Logical Masking

MATLAB supports the use of logical masking in order to perform selection on a matrix without the use of for loops or if statements.

A logical mask is defined as a matrix composed of only `1` and `0`.

For example:

```
mask = [1 0 0; 0 1 0; 0 0 1];
```

is a logical matrix representing the identity matrix.

We can generate a logical mask using a predicate to query a matrix.

```
A = [1 2 3; 4 5 6; 7 8 9];
B = A > 4;
```

We first create a 3x3 matrix, `A`, containing the numbers 1 through 9. We then query `A` for values that are greater than 4 and store the result in a new matrix called `B`.

`B` is a logical matrix of the form:

```
B = [0 0 0
     0 1 1
     1 1 1]
```

Or `1` when the predicate `A > 4` was true. And `0` when it was false.

We can use logical matrices to access elements of a matrix. If a logical matrix is used to select elements, indices where a `1` appear in the logical matrix will be selected in the matrix you are selecting from.

Using the same `B` from above, we could do the following:

```
C = [0 0 0; 0 0 0; 0 0 0];
C(B) = 5;
```

This would select all of the elements of `C` where `B` has a `1` in that index. Those indices in `C` are then set to `5`.

Our `C` now looks like:

```
C = [0 0 0
     0 5 5
     5 5 5]
```

We can reduce complicated code blocks containing `if` and `for` by using logical masks.

Take the non-vectorized code:

```
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
```

This can be shortened using logical masking to the following code:

```
A = [1 3 5; 7 9 11; 11 9 7];
B = A > 5;
A(B) = A(B) - 2;
```

Or even shorter:

```
A = [1 3 5; 7 9 11; 11 9 7];
A(A > 5) = A(A > 5) - 2;
```

Section 10.5: Sum, mean, prod & co

Given a random vector

```
v = rand(10,1);
```

if you want the sum of its elements, do **NOT** use a loop

```
s = 0;
for ii = 1:10
    s = s + v(ii);
end
```

but use the vectorized capability of the `sum()` function

```
s = sum(v);
```

Functions like `sum()`, `mean()`, `prod()` and others, have the ability to operate directly along rows, columns or other dimensions.

For instance, given a random matrix

```
A = rand(10,10);
```

the average for each **column** is

```
m = mean(A,1);
```

the average for each **row** is

```
m = mean(A,2)
```

All the functions above work only on one dimension, but what if you want to sum the whole matrix? You could use:

```
s = sum(sum(A))
```

But what if have an ND-array? applying `sum` on `sum` on `sum`... don't seem like the best option, instead use the `:` operator to vectorize your array:

```
s = sum(A(:))
```

and this will result in one number which is the sum of all your array, doesn't matter how many dimensions it have.

Section 10.6: Get the value of a function of two or more arguments

In many application it is necessary to compute the function of two or more arguments.

Traditionally, we use `for`-loops. For example, if we need to calculate the `f = exp(-x^2-y^2)` (do not use this if you need **fast simulations**):

```
% 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
```

But vectorized version is more elegant and faster:

```
% code2
[x,y] = ndgrid(-1.2:0.2:1.4, -2:0.25:3);
f = exp(-x.^2-y.^2);
```

than we can visualize it:

```
surf(x,y,f)
```

Note1 - Grids: Usually, the matrix storage is organized *row-by-row*. But in the MATLAB, it is the *column-by-column* storage as in FORTRAN. Thus, there are two similar functions `ndgrid` and `meshgrid` in MATLAB to implement the two aforementioned models. To visualise the function in the case of `meshgrid`, we can use:

```
surf(y,x,f)
```

Note2 - Memory consumption: Let size of `x` or `y` is 1000. Thus, we need to store `1000*1000+2*1000 ~ 1e6` elements for non-vectorized ***code1***. But we need `3*(1000*1000) = 3e6` elements in the case of vectorized ***code2***. In the 3D case (let `z` has the same size as`x` or `y`), memory consumption increases dramatically: `4*(1000*1000*1000)` (~32GB for doubles) in the case of the vectorized ***code2*** vs `~1000*1000*1000` (just ~8GB) in the case of ***code1***. Thus, we have to choose either the memory or speed.

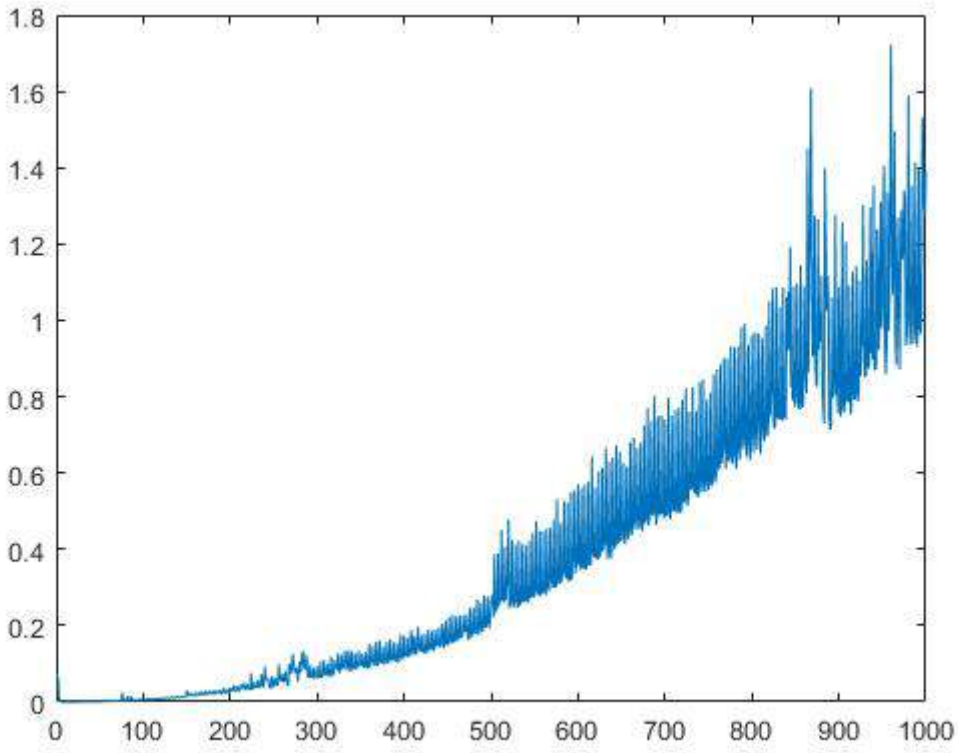
Chapter 11: Matrix decompositions

Section 11.1: Schur decomposition

If A is a complex and quadratic matrix there exists a unitary Q such that Q*AQ = T = D + N with D being the diagonal matrix consisting of the eigenvalues and N being strictly upper tridiagonal.

```
A = [3 6 1
    23 13 1
    0 3 4];
T = schur(A);
```

We also display the runtime of `schur` dependent on the square root of matrix elements:



Section 11.2: Cholesky decomposition

The Cholesky decomposition is a method to decompose an Hermitean, positive definite matrix into an upper triangular matrix and its transpose. It can be used to solve linear equations systems and is around twice as fast as LU-decomposition.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = chol(A);
```

This returns the upper triangular matrix. The lower one is obtained by transposition.

```
L = R';
```

We finally can check whether the decomposition was correct.

```
b = (A == L*R);
```

Section 11.3: QR decomposition

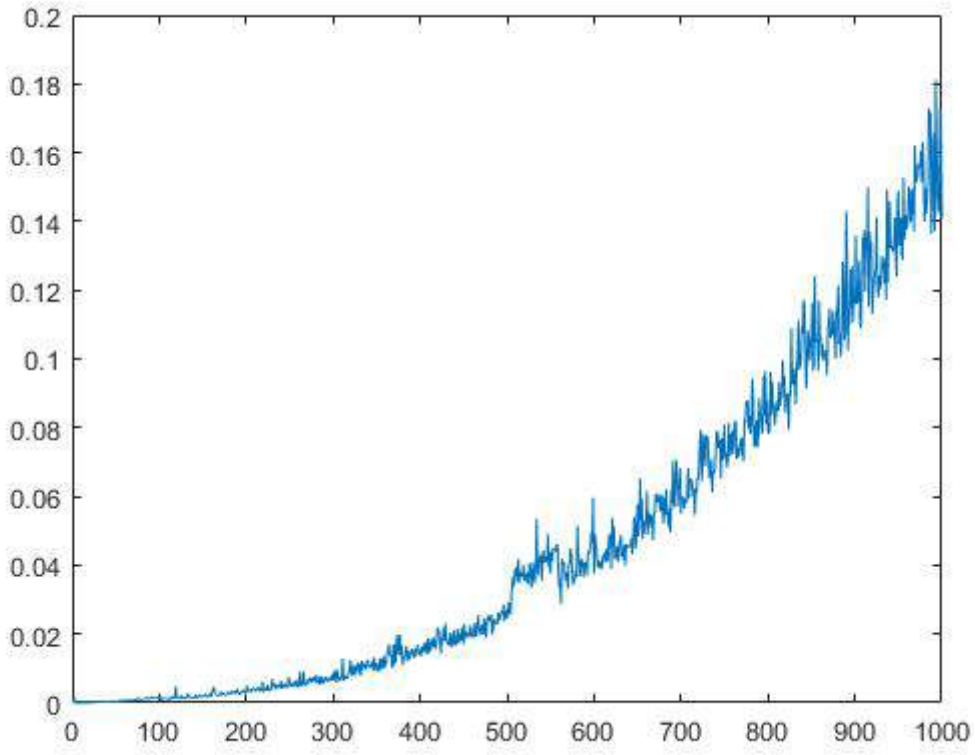
This method will decompose a matrix into an upper triangular and an orthogonal matrix.

```
A = [4 12 -16
    12 37 -43
    -16 -43 98];
R = qr(A);
```

This will return the upper triangular matrix while the following will return both matrices.

```
[Q,R] = qr(A);
```

The following plot will display the runtime of `qr` dependent of the square root of elements of the matrix.



Section 11.4: LU decomposition

Hereby a matrix will be decomposed into an upper triangular and an lower triangular matrix. Often it will be used to increase the performance and stability (if it's done with permutation) of Gauß elimination.

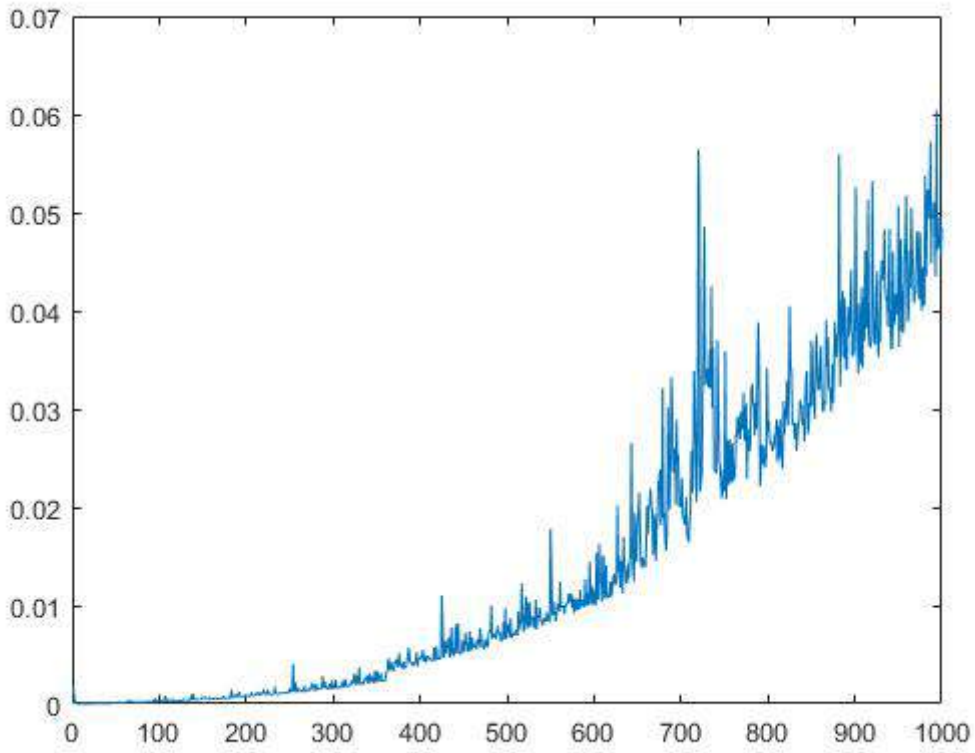
However, quite often does this method not or badly work as it is not stable. For example

```
A = [8 1 6
    3 5 7
    4 9 2];
[L,U] = lu(A);
```

It is sufficient to add an permutation matrix such that PA=LU:

```
[L,U,P]=lu(A);
```

In the following we will now plot the runtime of `lu' dependent of the square root of elements of the matrix.



Section 11.5: Singular value decomposition

Given an m times n matrix A with n larger than m. The singular value decomposition

```
[U,S,V] = svd(A);
```

computes the matrices U,S,V.

The matrix U consists of the left singular eigenvectors which are the eigenvectors of `A*A.'` while V consists of the right singular eigenvalues which are the eigenvectors of `A.'*A`. The matrix `S` has the square roots of the eigenvalues of `A*A.'` and `A.'*A` on its diagonal.

If m is larger than n one can use

```
[U,S,V] = svd(A,'econ');
```

to perform economy sized singular value decomposition.

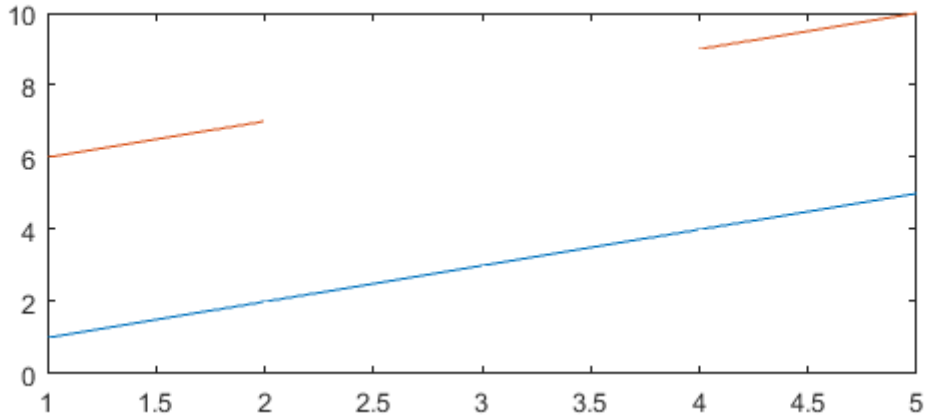
Chapter 12: Graphics: 2D Line Plots

Parameter	Details
X	x-values
Y	y-values
LineSpec	Line style, marker symbol, and color, specified as a string
Name,Value	Optional pairs of name-value arguments to customize line properties
h	handle to line graphics object

Section 12.1: Split line with NaNs

Interleave your `y` or `x` values with `NaNs`

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

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

plot(x)
```



Section 12.2: Multiple lines in a single plot

In this example we are going to plot multiple lines onto a single axis. Additionally, we choose a different appearance for the lines and create a legend.

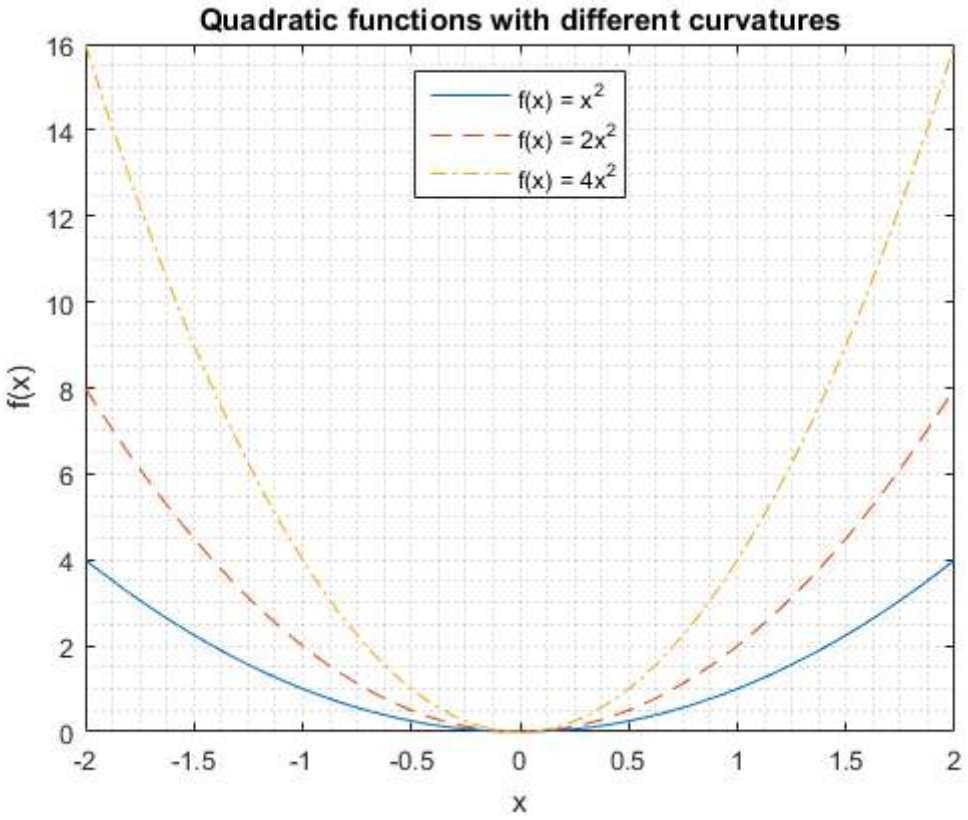
```
% create sample data
x = linspace(-2,2,100);                 % 100 linearly spaced points from -2 to 2
y1 = x.^2;
y2 = 2*x.^2;
y3 = 4*x.^2;

% 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');
```

In the above example, we plotted the lines with a single `plot`-command. An alternative is to use separate commands for each line. We need to *hold* the contents of a figure with `hold on` the latest before we add the second line. Otherwise the previously plotted lines will disappear from the figure. To create the same plot as above, we can use these following commands:

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

The resulting figure looks like this in both cases:

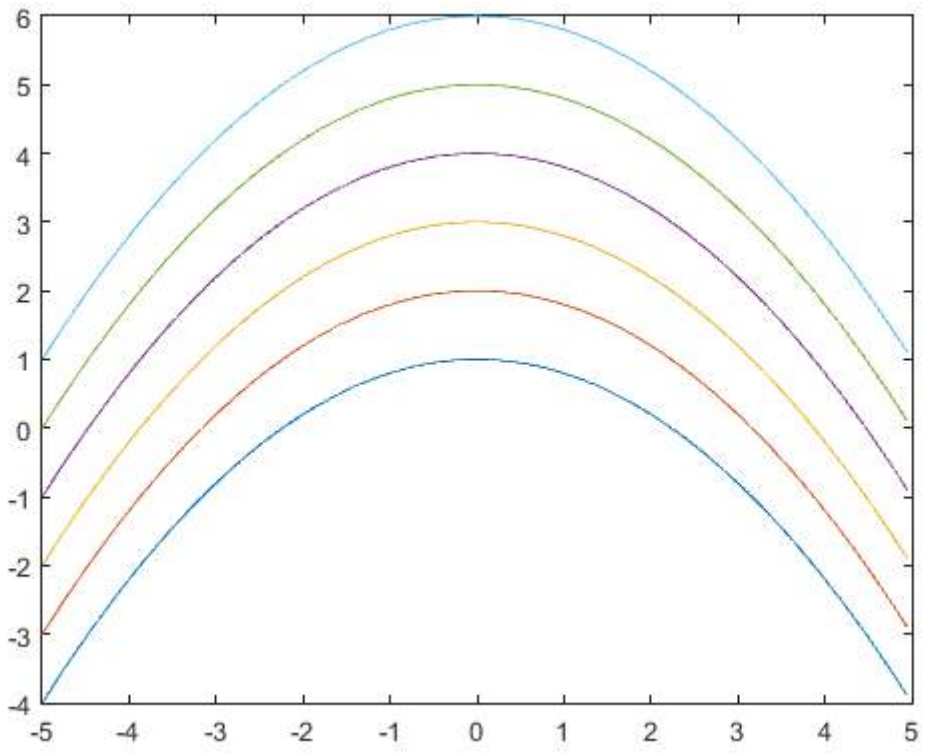


Section 12.3: Custom colour and line style orders

In MATLAB, we can set new *default* custom orders, such as a colour order and a line style order. That means new orders will be applied to any figure that is created after these settings have been applied. The new settings remains until MATLAB session is closed or new settings has been made.

Default colour and line style order

By default, MATLAB uses a couple of different colours and only a solid line style. Therefore, if `plot` is called to draw multiple lines, MATLAB alternates through a colour order to draw lines in different colours.



We can obtain the default colour order by calling `get` with a global handle `0` followed by this attribute `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
```

Custom colour and line style order

Once we have decided to set a custom colour order AND line style order, MATLAB must alternate through both. The first change MATLAB applies is a colour. When all colours are exhausted, MATLAB applies the next line style from a defined line style order and set a colour index to 1. That means MATLAB will begin to alternate through all colours again but using the next line style in its order. When all line styles and colours are exhausted, obviously MATLAB begins to cycle from the beginning using the first colour and the first line style.

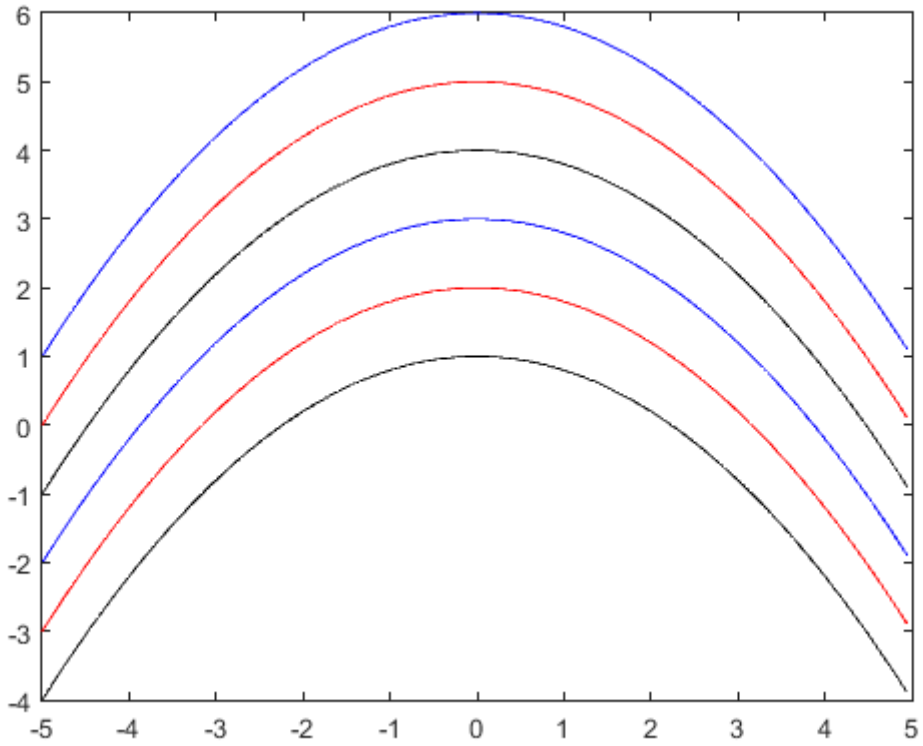
For this example, I have defined an input vector and an anonymous function to make plotting figures a little bit easier:

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

To set a new colour or a new line style orders, we call `set` function with a global handle `0` followed by an attribute `DefaultAxesXXXXXXX`; `XXXXXXX` can either be `ColorOrder` or `LineStyleOrder`. The following command sets a new colour order to black, red and blue, respectively:

```
set(0, 'DefaultAxesColorOrder', [0 0 0; 1 0 0; 0 0 1]);
```

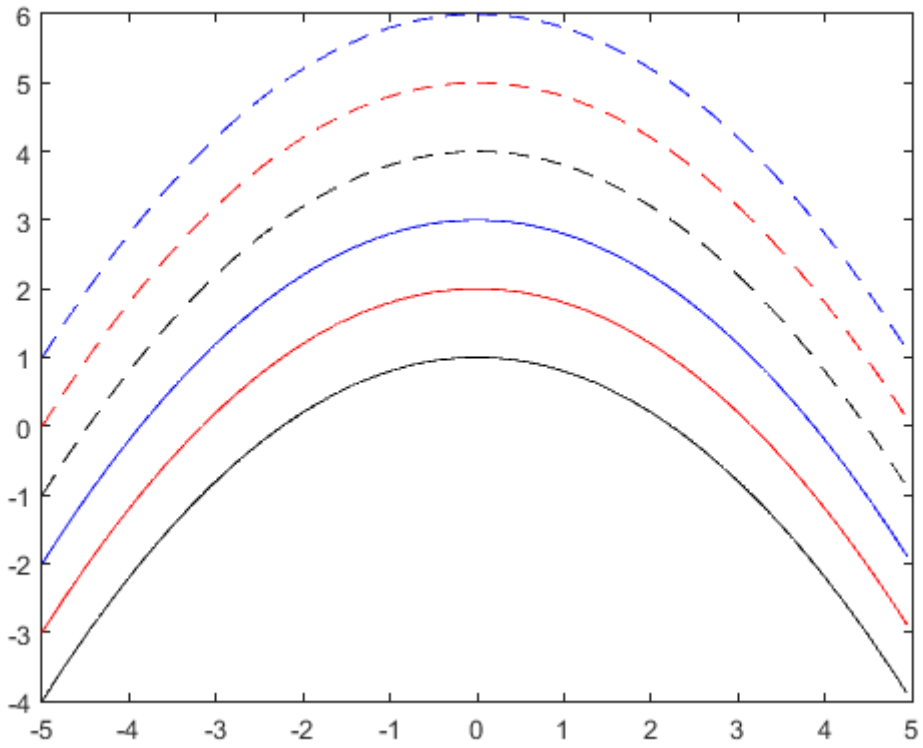
```
plot(x, F([1 2 3 4 5 6],x));
```



As you can see, MATLAB alternates only through colours because line style order is set to a solid line by default. When a set of colours is exhausted, MATLAB starts from the first colour in the colour order.

The following commands set both colour and line style orders:

```
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));
```



Now, MATLAB alternates through different colours and different line styles using colour as most frequent attribute.

Chapter 13: Graphics: 2D and 3D Transformations

Section 13.1: 2D Transformations

In this Example we are going to take a square shaped line plotted using `line` and perform transformations on it. Then we are going to use the same transformations but in different order and see how it influences the results.

First we open a figure and set some initial parameters (square point coordinates and transformation parameters)

```
%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;
```

Next we construct the transformation matrices (scale, rotation and translation):

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

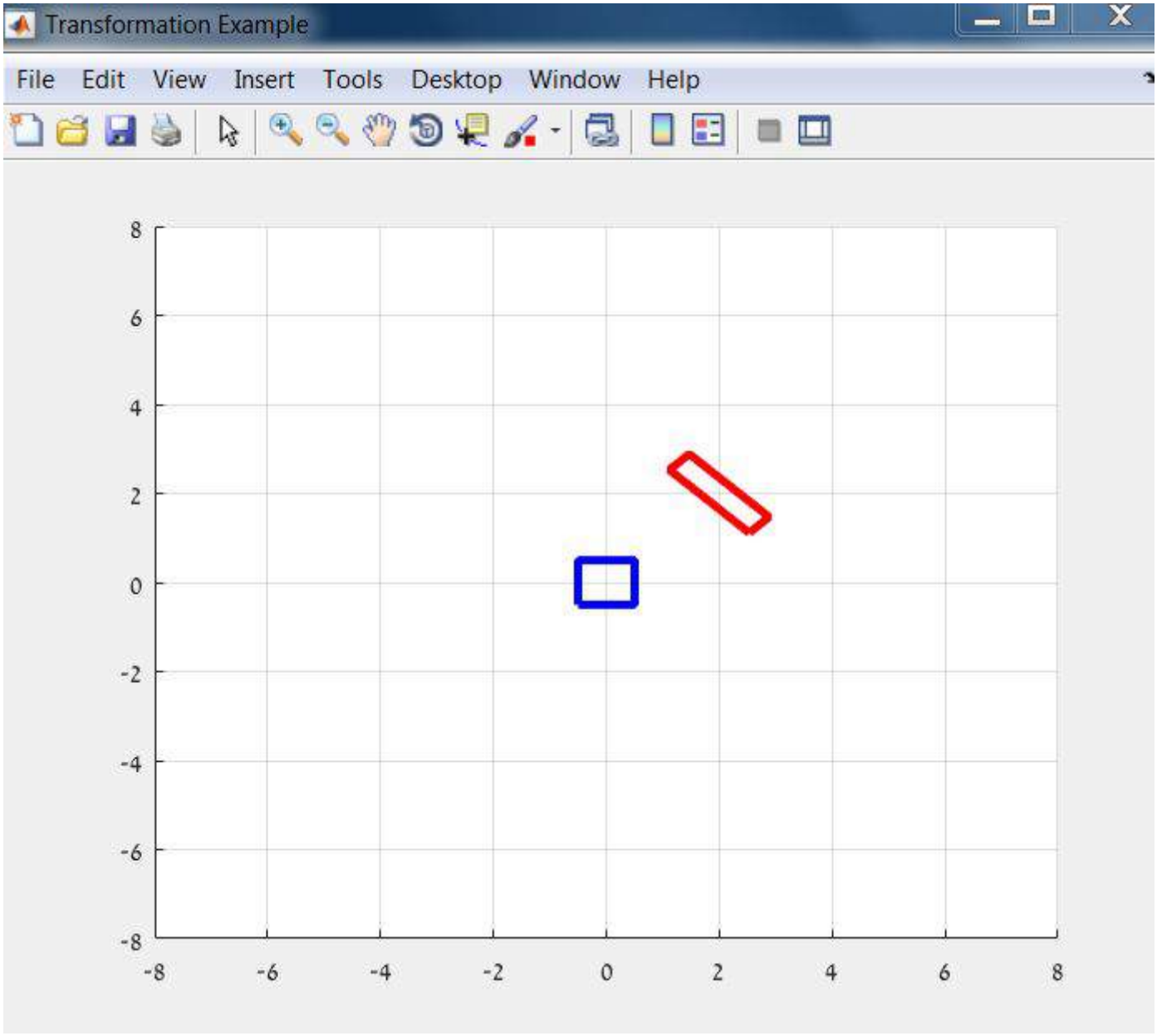
Next we plot the blue square:

```
%% 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
```

Next we will plot it again in a different color (red) and apply the transformations:

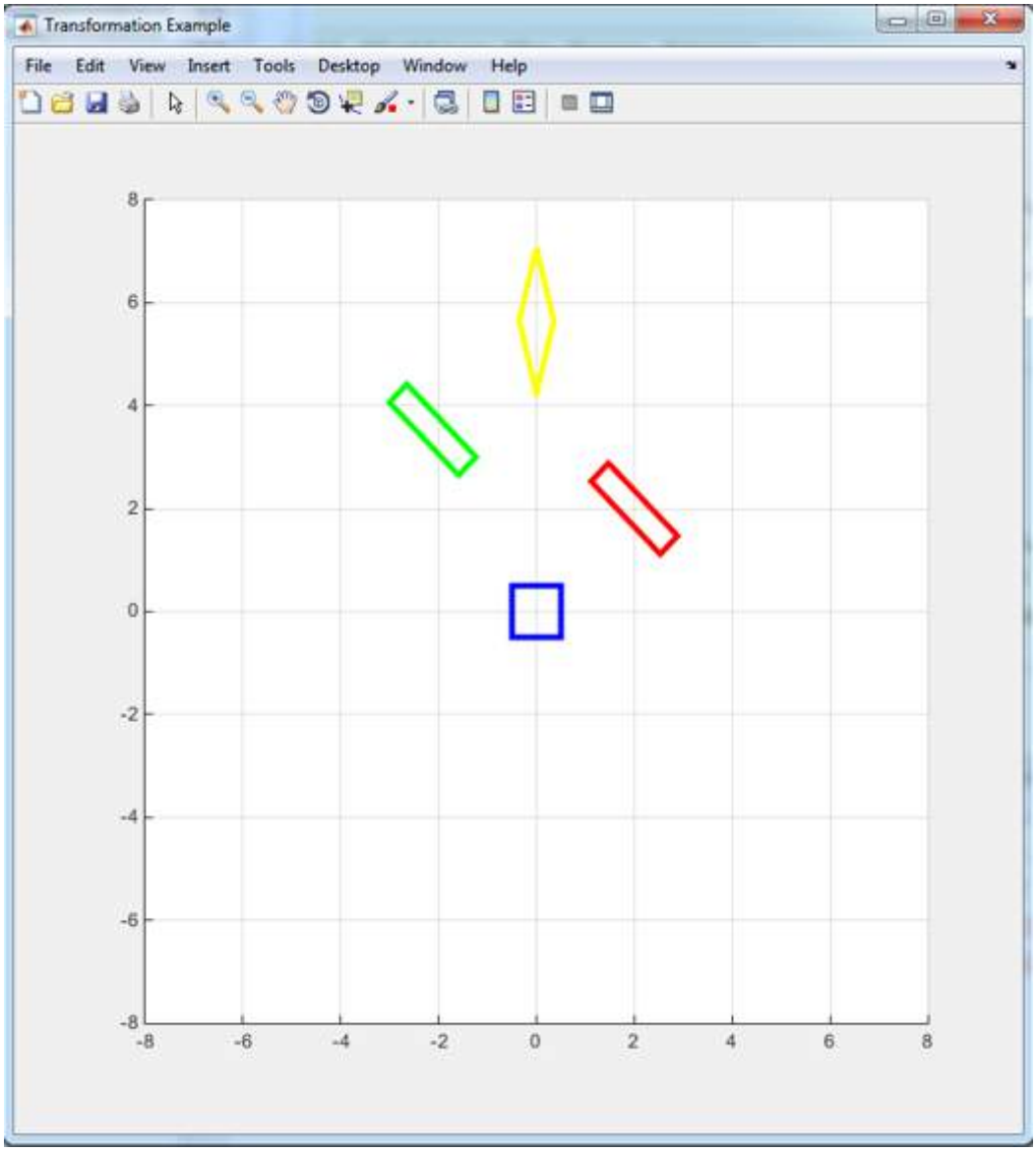
```
%% Plotting the Red Square
%Calculate rectangle vertices
HrectTRS=T*R*S;
RedSQ=line([square(:,1);square(1,1)],[square(:,2);square(1,2)],'Color','r','LineWidth',3);
%transformation of the axes
AxesTransformation=hgtransform('Parent',gca,'matrix',HrectTRS);
%setting the line to be a child of transformed axes
set(RedSQ,'Parent',AxesTransformation);
```

The result should look like this:



Now let's see what happens when we change the transformation order:

```
%% 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);
```


Transformation Example
File Edit View Insert Tools Desktop Window Help
8
6
4
2
0
-2
-4
-6
-8
-8 -6 -4 -2 0 2 4 6 8

Chapter 14: Controlling Subplot coloring in MATLAB

As I was struggling with this more than once, and the web isn't really clear on what to do, I decided to take what's out there, adding some of my own in order to explain how to create subplots which have one colorbar and they are scaled according to it.

I have tested this using latest MATLAB but I'm pretty sure it'll work in older versions.

Section 14.1: How it's done

This is a simple code creating 6 3d-subplots and in the end syncing the color displayed in each of them.

```
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
```

Chapter 15: Image processing

Section 15.1: Basic image I/O

```
>> img = imread('football.jpg');
```

Use `imread` to read image files into a matrix in MATLAB.
Once you `imread` an image, it is stored as an ND-array in memory:

```
>> size(img)
ans =
 256   320     3
```

The image `'football.jpg'` has 256 rows and 320 columns and it has 3 color channels: Red, Green and Blue.

You can now mirror it:

```
>> mirrored = img(:, end:-1:1, :);  %// like mirroring any ND-array in MATLAB
```

And finally, write it back as an image using `imwrite`:

```
>> imwrite(mirrored, 'mirrored_football.jpg');
```

Section 15.2: Retrieve Images from the Internet

As long as you have an internet connection, you can read images from an hyperlink

```
I=imread('https://cdn.sstatic.net/Sites/stackoverflow/company/img/logos/so/so-logo.png');
```

Section 15.3: Filtering Using a 2D FFT

Like for 1D signals, it's possible to filter images by applying a Fourier transformation, multiplying with a filter in the frequency domain, and transforming back into the space domain. Here is how you can apply high- or low-pass filters to an image with MATLAB:

Let `image` be the original, unfiltered image, here's how to compute its 2D FFT:

```
ft = fftshift(fft2(image));
```

Now to exclude a part of the spectrum, one need to set its pixel values to 0. The spatial frequency contained in the original image is mapped from the center to the edges (after using `fftshift`). To exclude the low frequencies, we will set the central circular area to 0.

Here's how to generate a disc-shaped binary mask with radius `D` using built-in function:

```
[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]);
```

Masking the frequency domain image can be done by multiplying the FFT point-wise with the binary mask obtained above:

```
masked_ft = ft .* mask;
```

Now, let's compute the inverse FFT:

```
filtered_image = ifft2(ifftshift(masked_ft), 'symmetric');
```

The high frequencies in an image are the sharp edges, so this high-pass filter can be used to sharpen images.

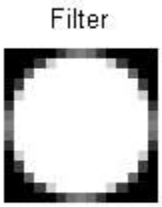
Section 15.4: Image Filtering

Let's say you have an image `rgbImg`, e.g., read in using `imread`.

```
>> rgbImg = imread('pears.png');
>> figure, imshow(rgbImg), title('Original Image')
```

Use `fspecial` to create a 2D filter:

```
>> h = fspecial('disk', 7);
>> figure, imshow(h, []), title('Filter')
```



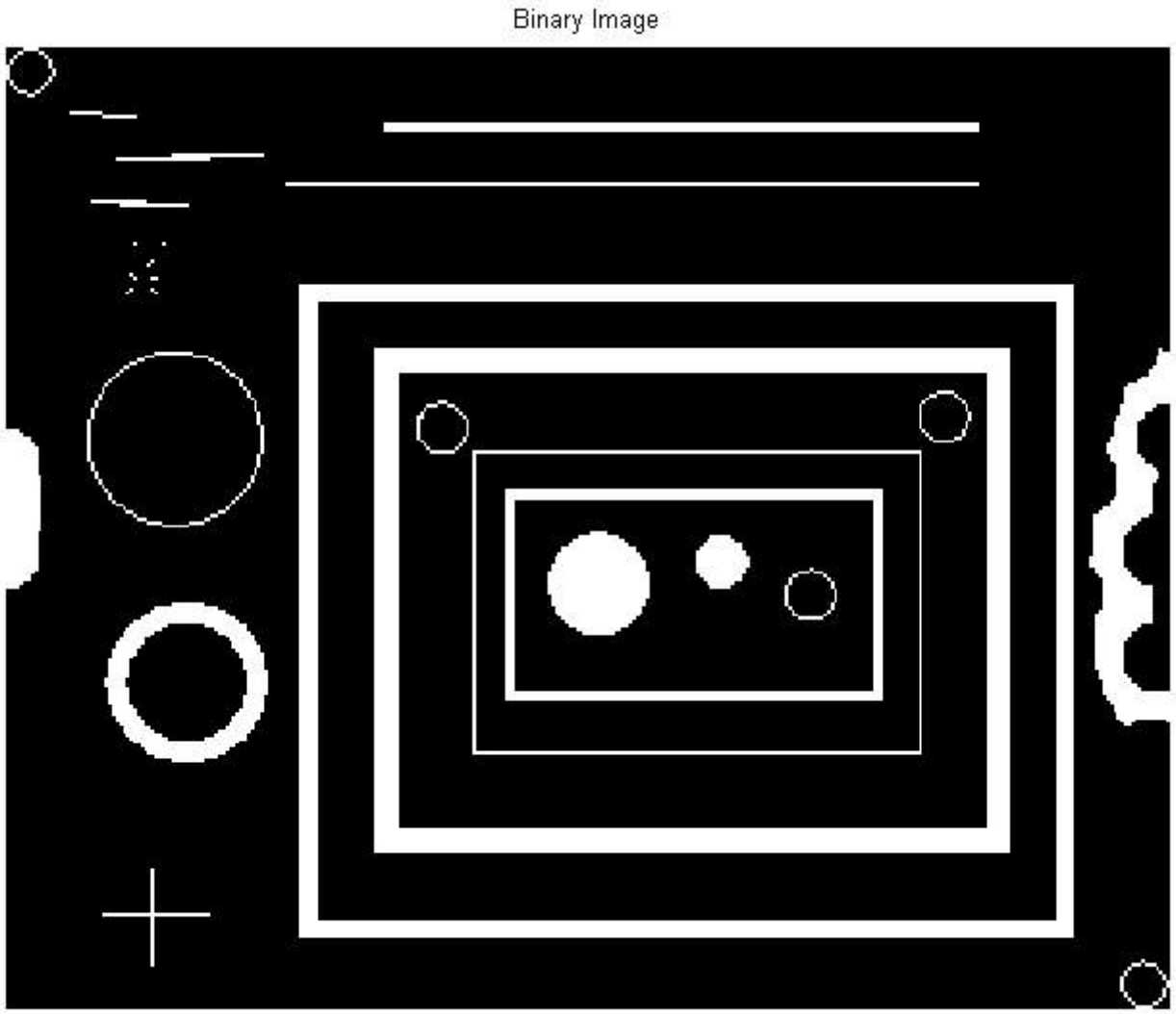
Use `imfilter` to apply the filter on the image:

```
>> filteredRgbImg = imfilter(rgbImg, h);
>> figure, imshow(filteredRgbImg), title('Filtered Image')
```

Section 15.5: Measuring Properties of Connected Regions

Starting with a binary image, `bwImg`, which contains a number of connected objects.

```
>> bwImg = imread('blobs.png');
>> figure, imshow(bwImg), title('Binary Image')
```



To measure properties (e.g., area, centroid, etc) of every object in the image, use `regionprops`:

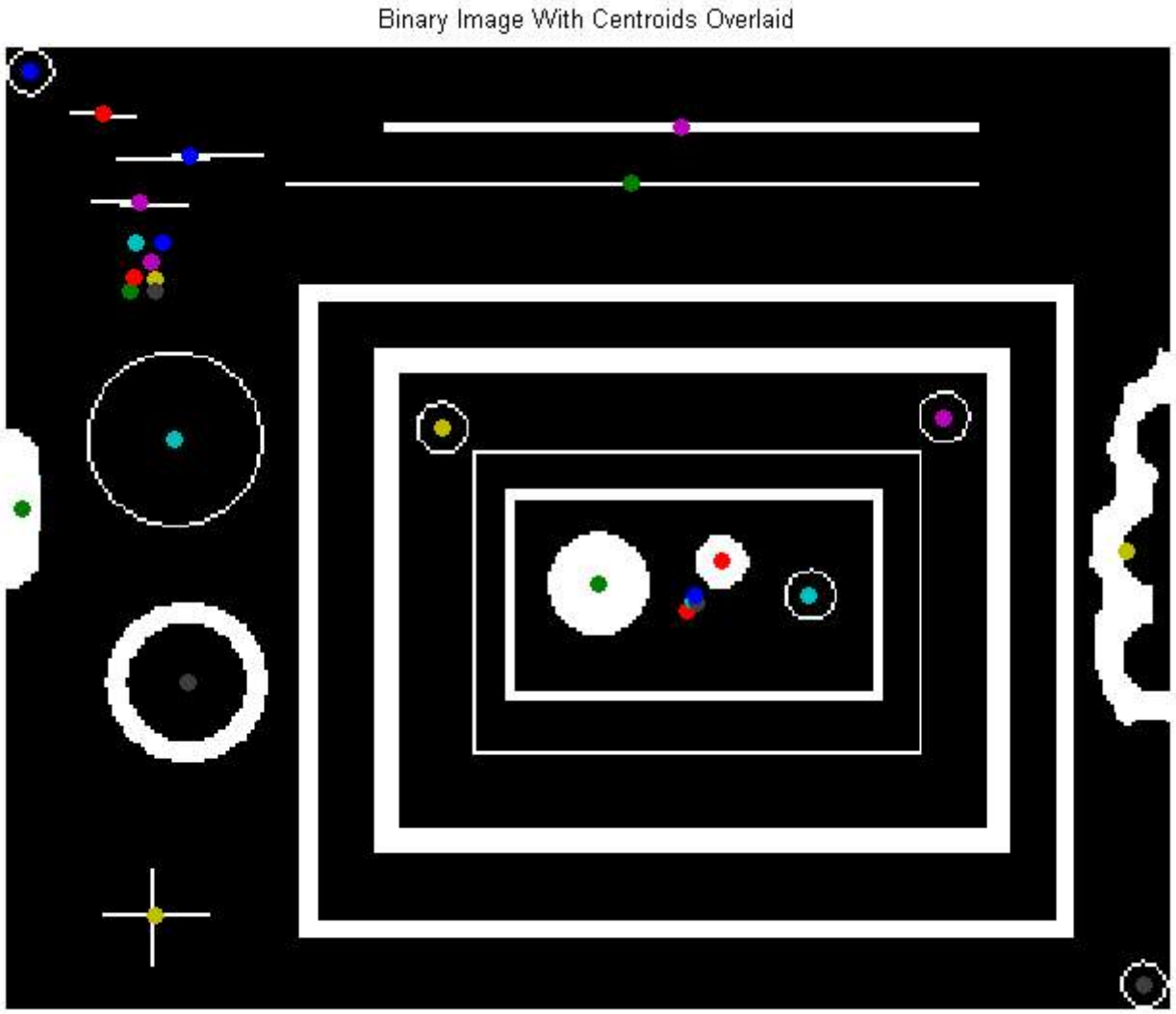
```
>> stats = regionprops(bwImg, 'Area', 'Centroid');
```

`stats` is a struct array which contains a struct for every object in the image. Accessing a measured property of an object is simple. For example, to display the area of the first object, simply,

```
>> stats(1).Area

ans =

    35
```

Visualize the object centroids by overlaying them on the original image.

```
>> 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
```


Binary Image With Centroids Overlaid

Chapter 16: Drawing

Section 16.1: Circles

The easiest option to draw a circle, is - obviously - the **`rectangle`** function.

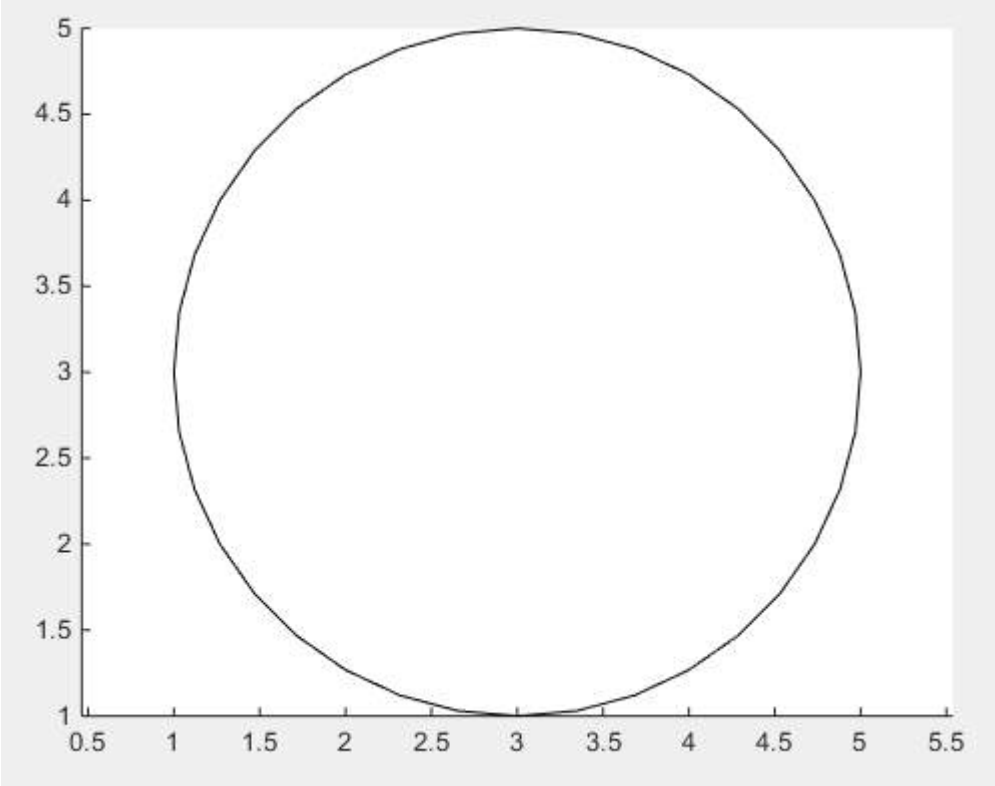
```
%// radius
r = 2;

%// center
c = [3 3];

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

but the curvature of the rectangle has to be set to **1**!

The `position` vector defines the rectangle, the first two values `x` and `y` are the lower left corner of the rectangle. The last two values define width and height of the rectangle.

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

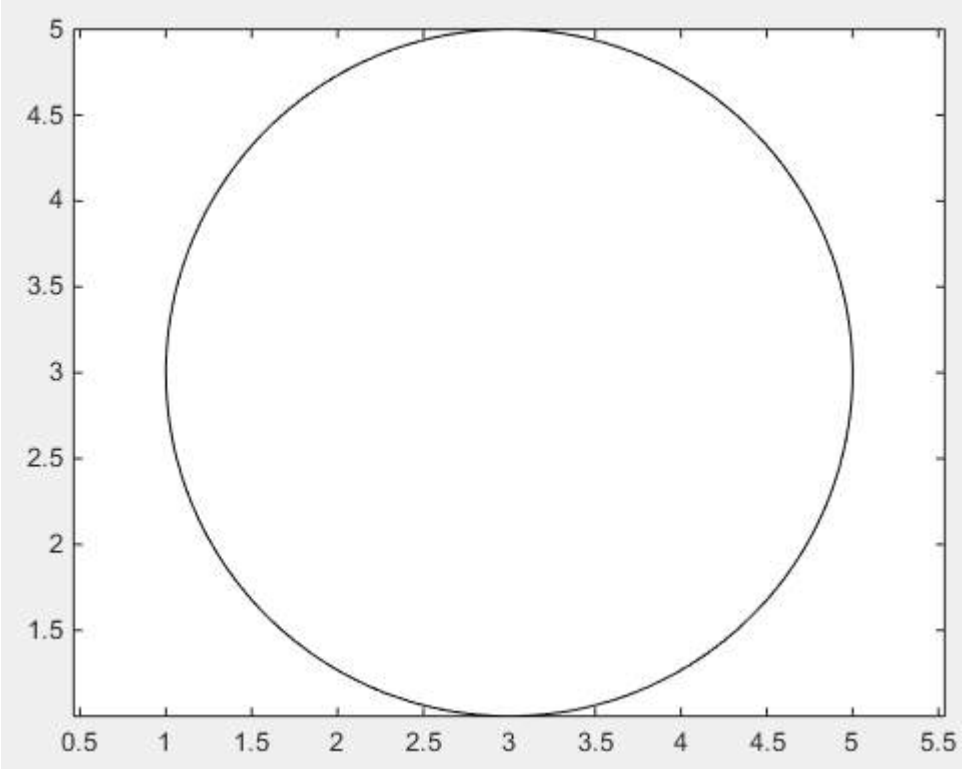
The lower left *corner* of the circle - yes, this circle has corners, imaginary ones though - is the **center** `c = [3 3]` **minus the radius** `r = 2` which is `[x y] = [1 1]`. **Width** and **height** are equal to the **diameter** of the circle, so `width = 2*r; height = width;`



In case the smoothness of the above solution is not sufficient, there is no way around using the obvious way of drawing an actual circle by use of **trigonometric functions**.

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

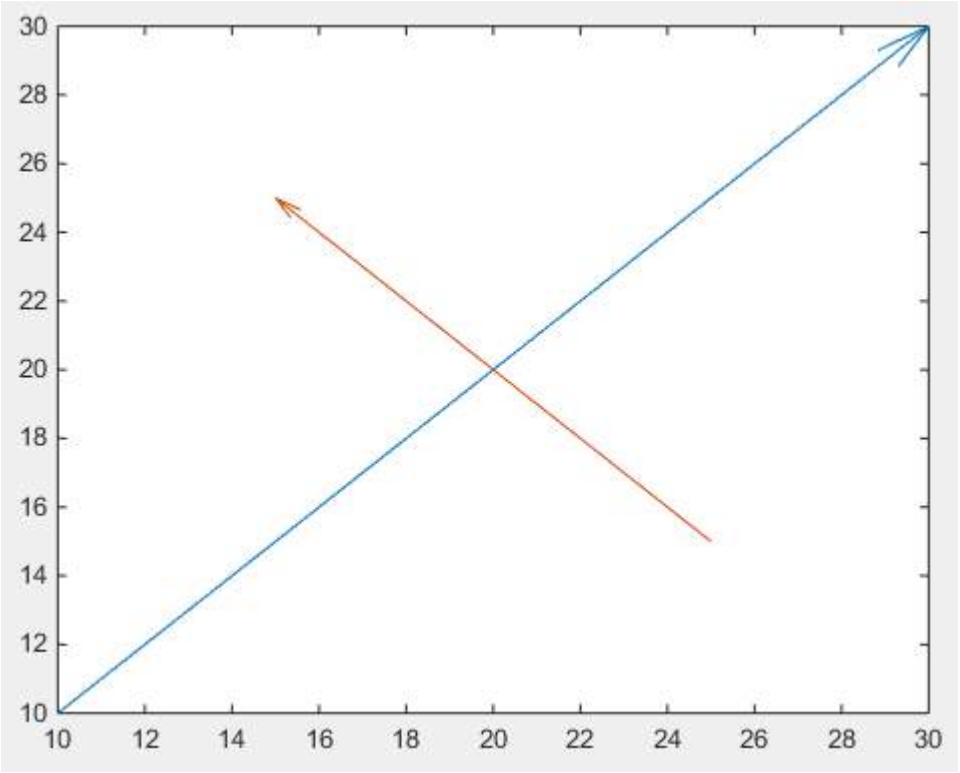
```
%// running variable
t = linspace(0,2*pi,n);

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
```



Section 16.2: Arrows

Firstly, one can use `quiver`, where one doesn't have to deal with unhandy normalized figure units by use of `annotation`

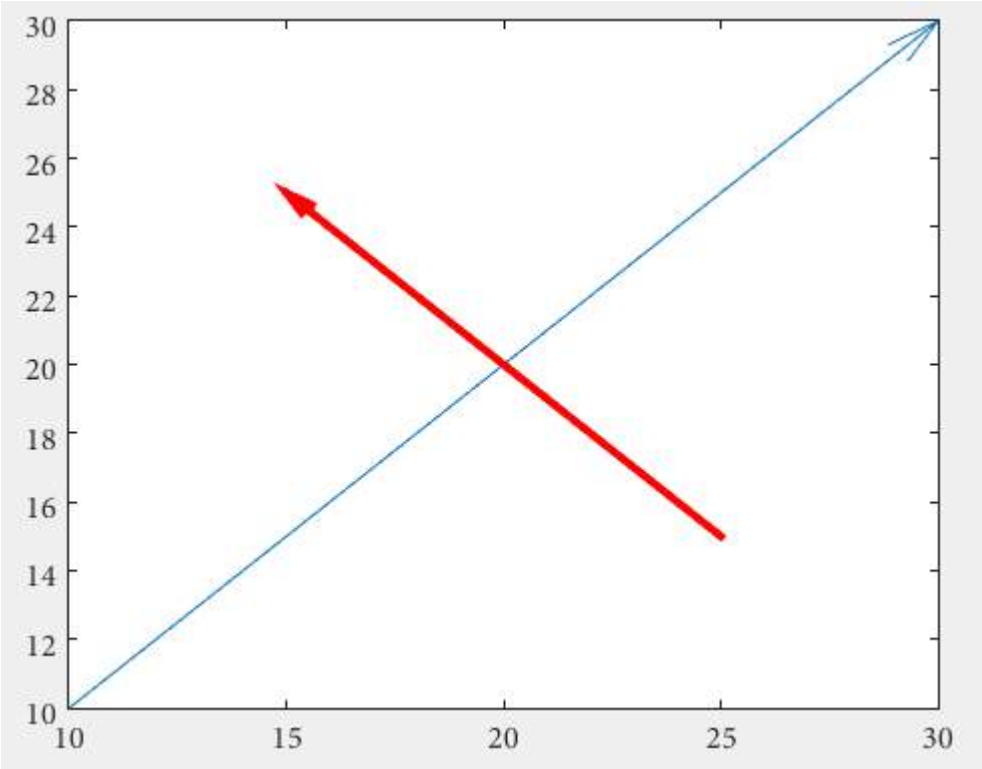
```
drawArrow = @(x,y) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0 )

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

drawArrow(x1,y1); hold on

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

drawArrow(x2,y2)
```



Important is the 5th argument of `quiver`: 0 which disables an otherwise default scaling, as this function is usually used to plot vector fields. (or use the property value pair `'AutoScale','off'`)

One can also add additional features:

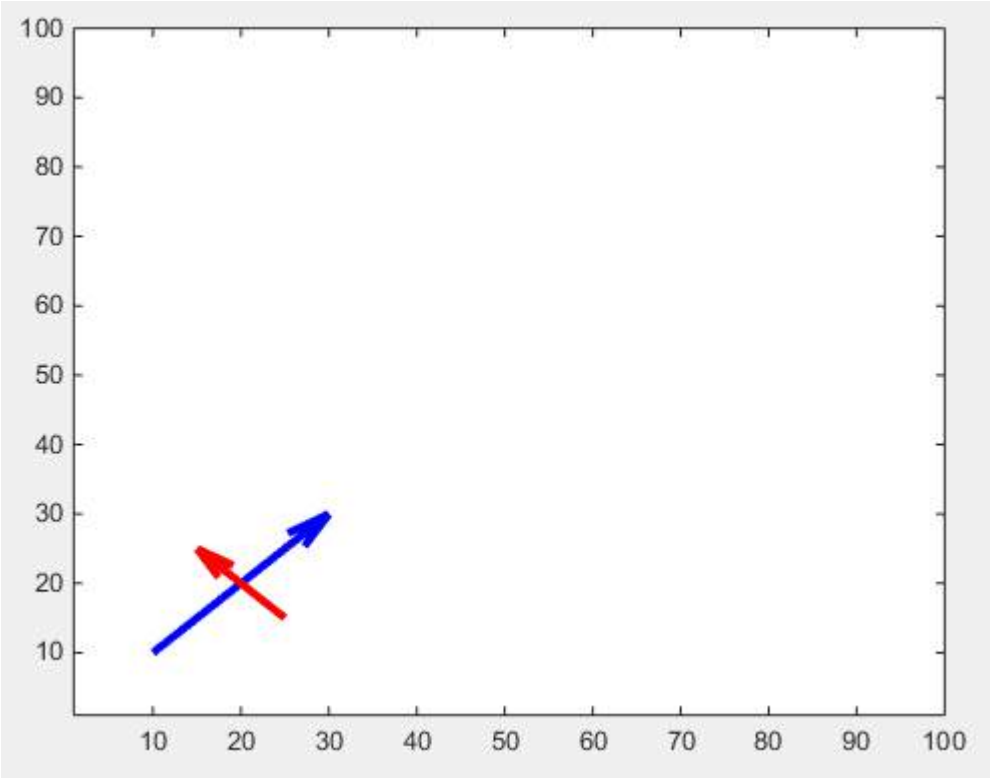
```
drawArrow = @(x,y,varargin) quiver( x(1),y(1),x(2)-x(1),y(2)-y(1),0, varargin{:} )
drawArrow(x1,y1); hold on
drawArrow(x2,y2,'linewidth',3,'color','r')
```



If different arrowheads are desired, one needs to use annotations (this answer is may helpful How do I change the arrow head style in quiver plot?).

The arrow head size can be adjust with the `'MaxHeadSize'` property. It's not consistent unfortunately. The axes limits need to be set afterwards.

```
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])
```

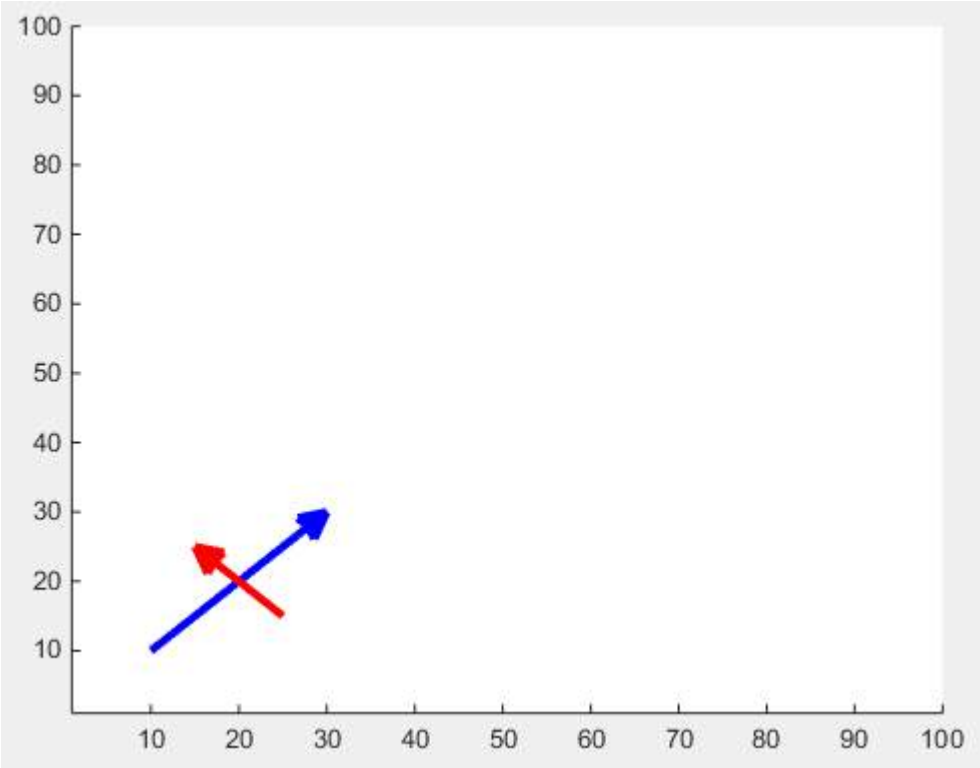


There is another tweak for adjustable arrow heads:

```
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
```

which you can call from your script as follows:

```
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
```



Section 16.3: Ellipse

To plot an ellipse you can use its equation. An ellipse has a major and a minor axis. Also we want to be able to plot the ellipse on different center points. Therefore we write a function whose inputs and outputs are:

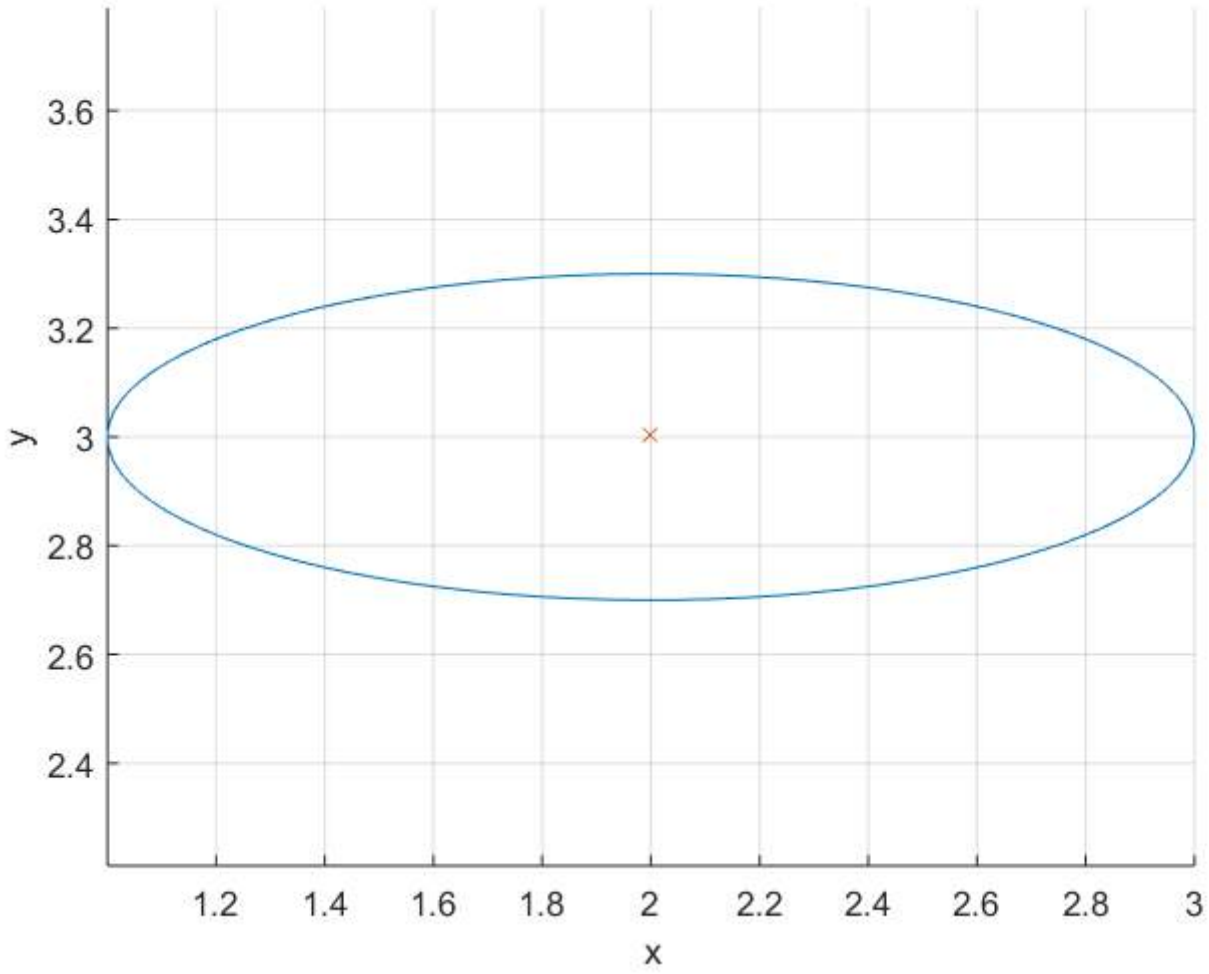
```
Inputs:
    r1,r2: major and minor axis respectively
    C: center of the ellipse (cx,cy)
Output:
    [x,y]: points on the circumference of the ellipse
```

You can use the following function to get the points on an ellipse and then plot those points.

```
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
```

Example:

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



Section 16.4: Pseudo 4D plot

A `(m x n)` matrix can be represented by a surface by using surf;

The color of the surface is automatically set as function of the values in the `(m x n)` matrix. If the colormap is not specified, the default one is applied.

A colorbar can be added to display the current colormap and indicate the mapping of data values into the colormap.

In the following example, the `z (m x n)` matrix is generated by the function:

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

over the interval `[-pi,pi]`. The `x` and `y` values can be generated using the meshgrid function and the surface is rendered as follows:

```
% 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
```

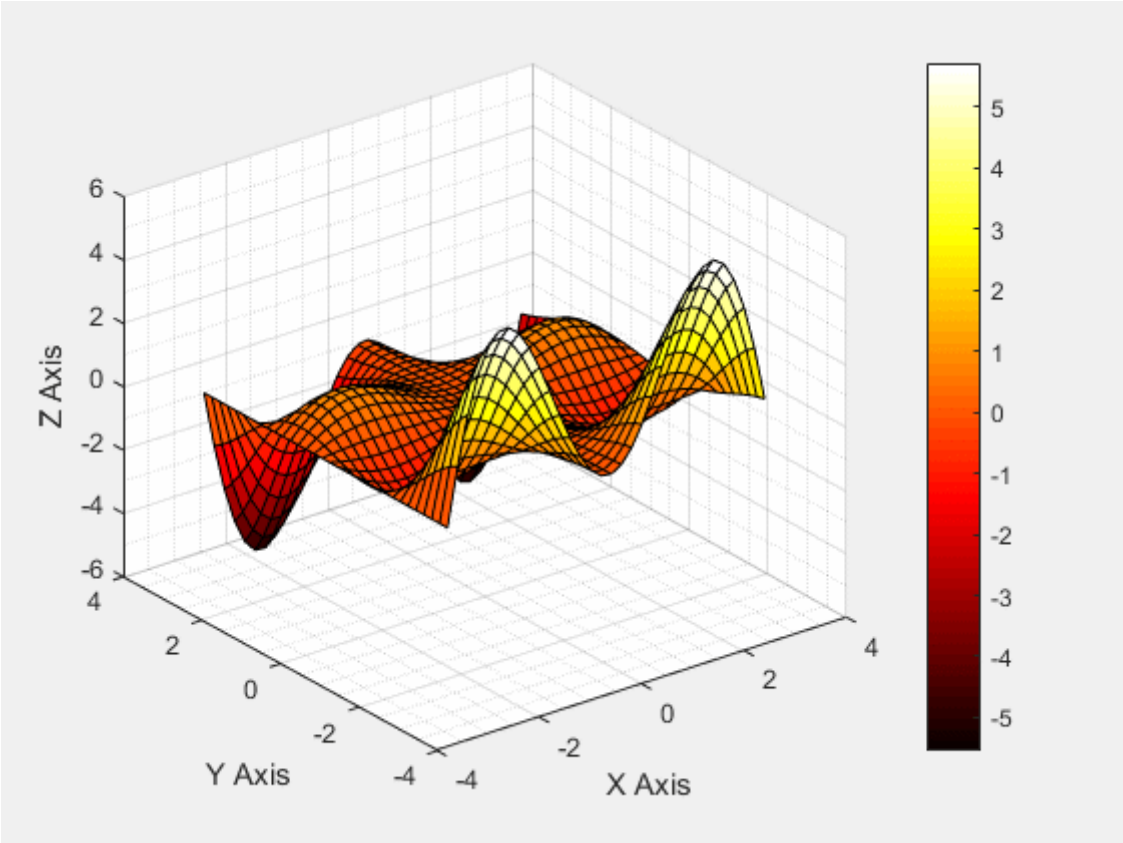


Figure 1

Now it could be the case that additional information are linked to the values of the `z` matrix and they are store in another `(m x n)` matrix

It is possible to add these additional information on the plot by modifying the way the surface is colored.

This will allows having kinda of 4D plot: to the 3D representation of the surface generated by the first `(m x n)` matrix, the fourth dimension will be represented by the data contained in the second `(m x n)` matrix.

It is possible to create such a plot by calling `surf` with 4 input:

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

where the `C` parameter is the second matrix (which has to be of the same size of `z`) and is used to define the color of the surface.

In the following example, the `C` matrix is generated by the function:

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

over the interval `[-pi,pi]`

The surface generated by `C` is

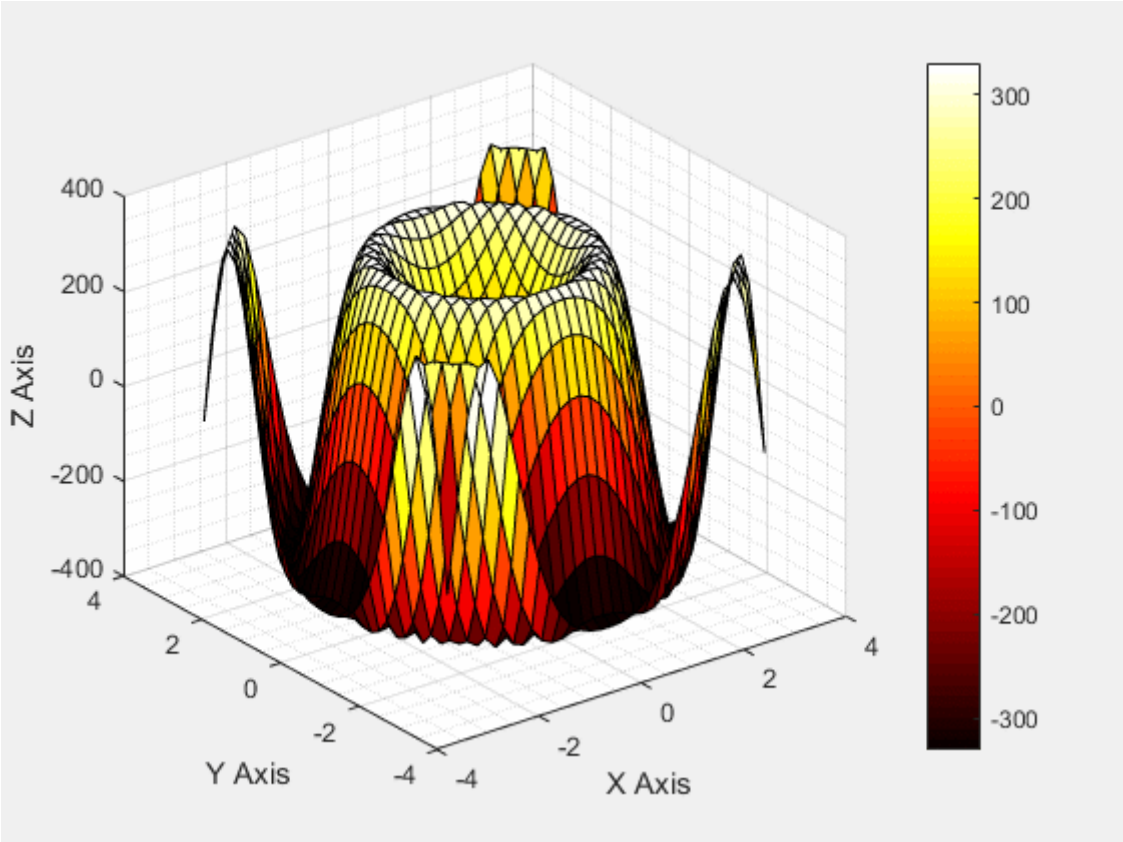


Figure 2

Now we can call `surf` with four input:

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

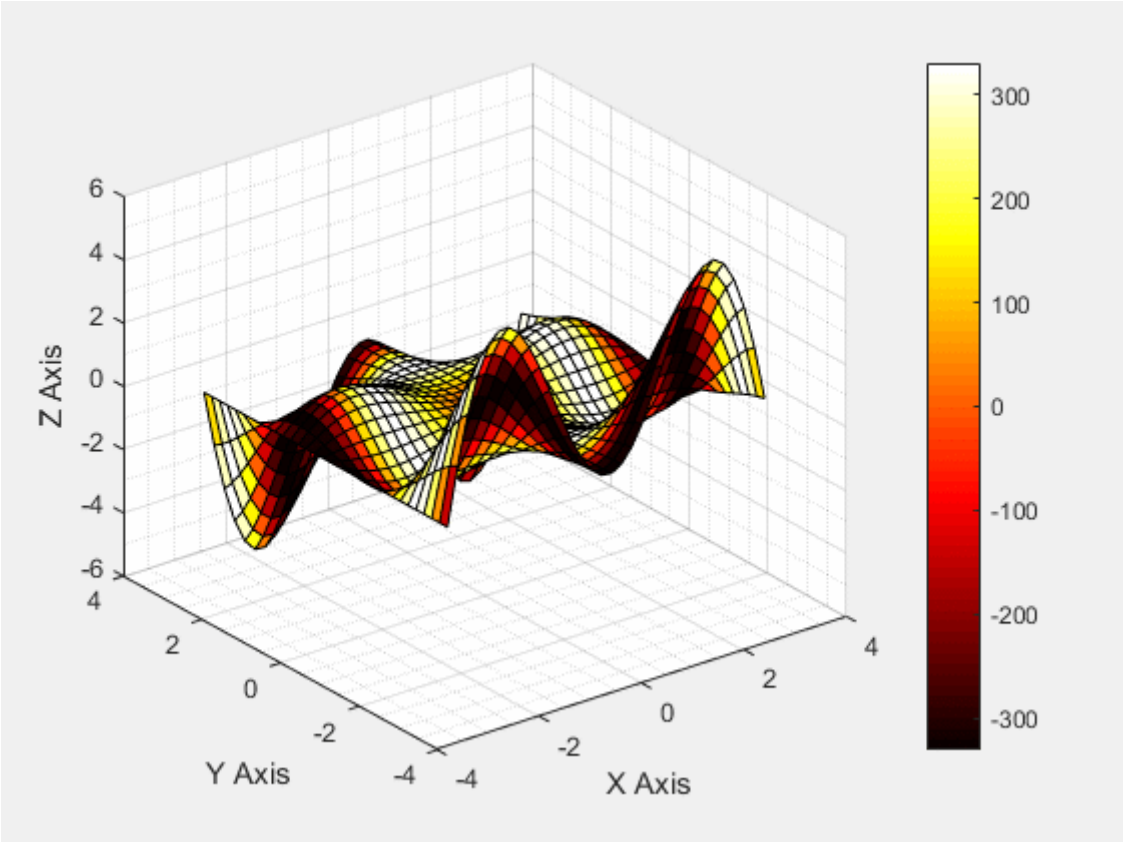


Figure 3

Comparing Figure 1 and Figure 3, we can notice that:

- the shape of the surface corresponds to the `z` values (the first `(m x n)` matrix)
- the colour of the surface (and its range, given by the colorbar) corresponds to the `C` values (the first `(m x n)` matrix)

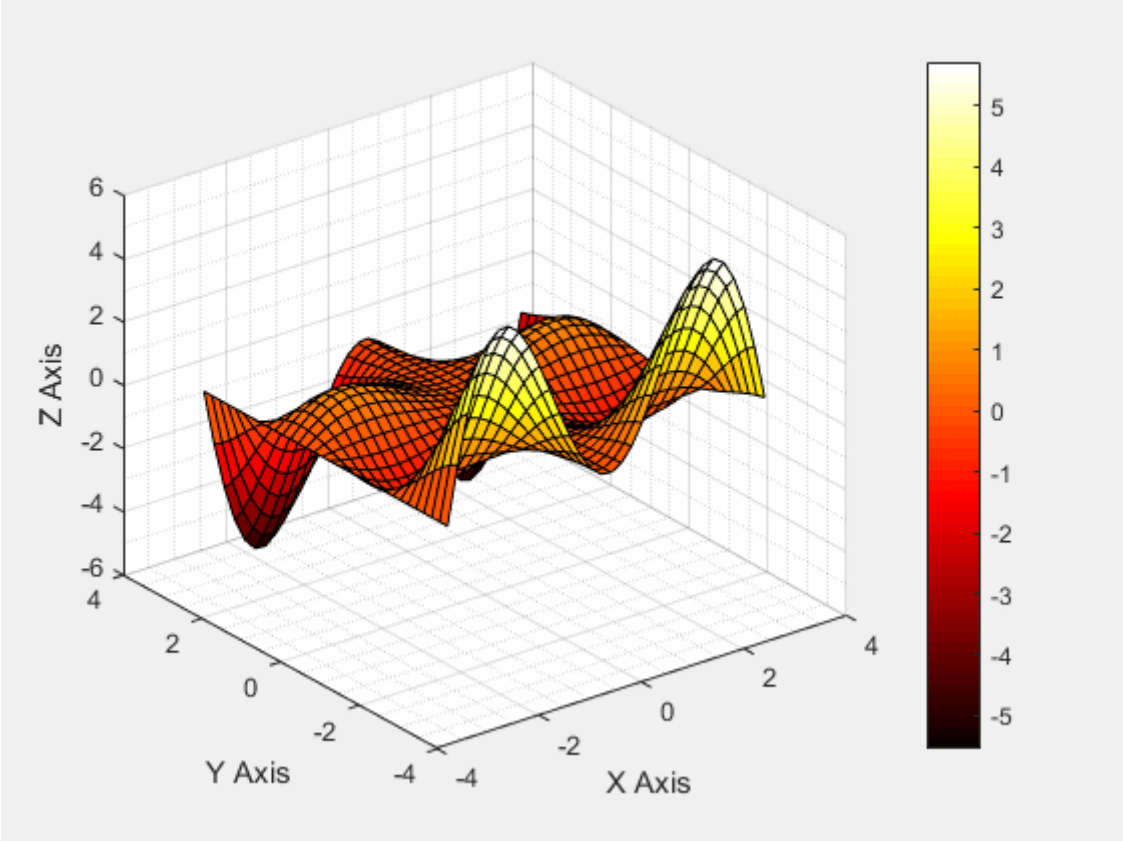
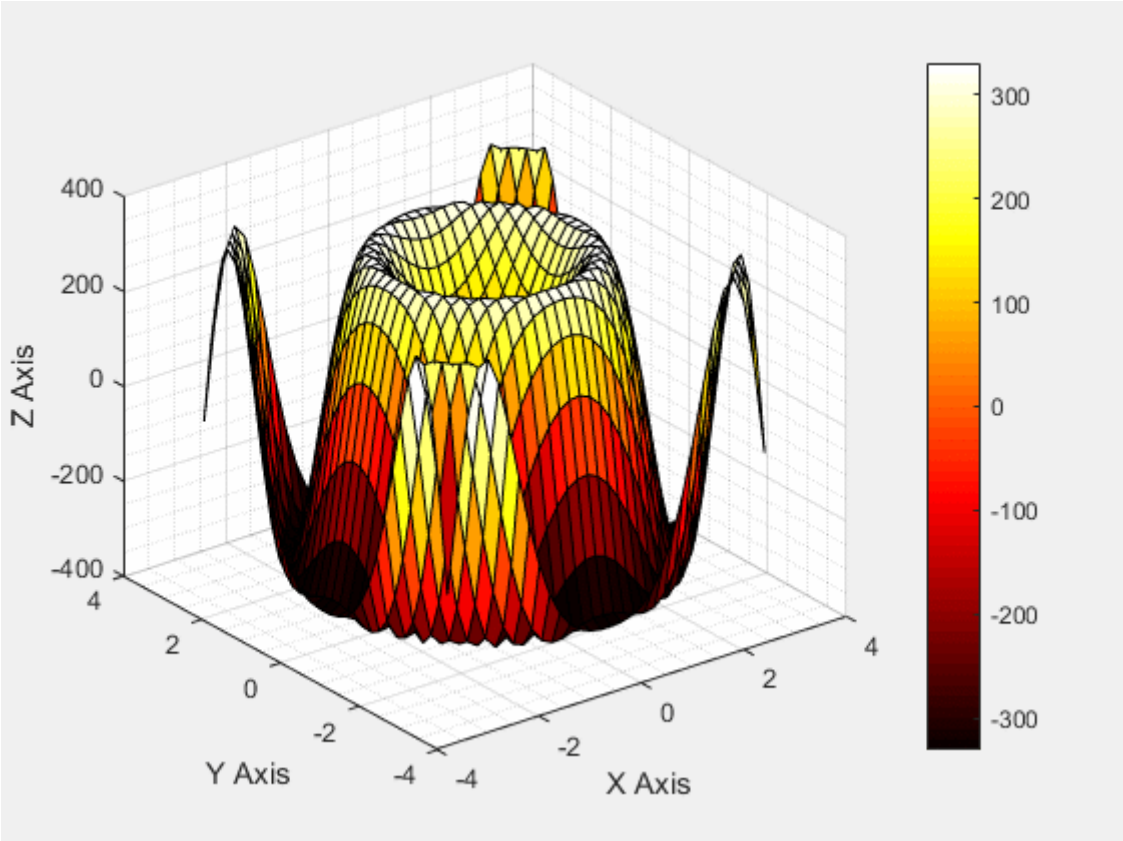


Figure 4

Of course, it is possible to swap `z` and `C` in the plot to have the shape of the surface given by the `C` matrix and the color given by the `z` matrix:

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

and to compare Figure 2 with Figure 4



Section 16.5: Fast drawing

There are three main ways to do sequential plot or animations: `plot(x,y)`, `set(h , 'XData' , y, 'YData' , y)` and `animatedline`. If you want your animation to be smooth, you need efficient drawing, and the three methods are not equivalent.

```
% 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
```

I get `5.278172 seconds`. The plot function basically deletes and recreates the line object each time. A more efficient way to update a plot is to use the `XData` and `YData` properties of the `Line` object.

```
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
```

Now I get `2.741996 seconds`, much better!

`animatedline` is a relatively new function, introduced in 2014b. Let's see how it fares:

```
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`, not as good as updating an existing plot, but still better than `plot(x,y)`.

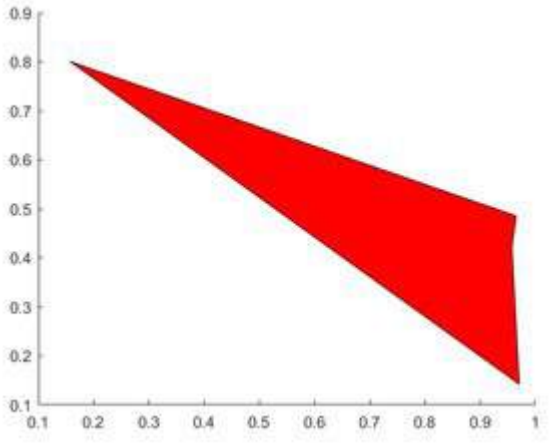
Of course, if you have to plot a single line, like in this example, the three methods are almost equivalent and give smooth animations. But if you have more complex plots, updating existing `Line` objects will make a difference.

Section 16.6: Polygon(s)

Create vectors to hold the x- and y-locations of vertices, feed these into `patch`.

Single Polygon

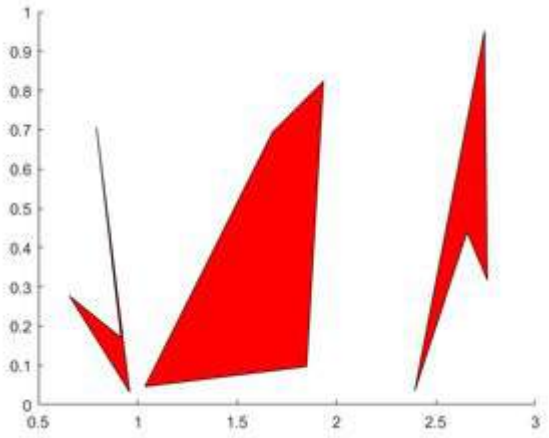
```
X=rand(1,4); Y=rand(1,4);
h=patch(X,Y,'red');
```



Multiple Polygons

Each polygon's vertices occupy one column of each of `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');
```



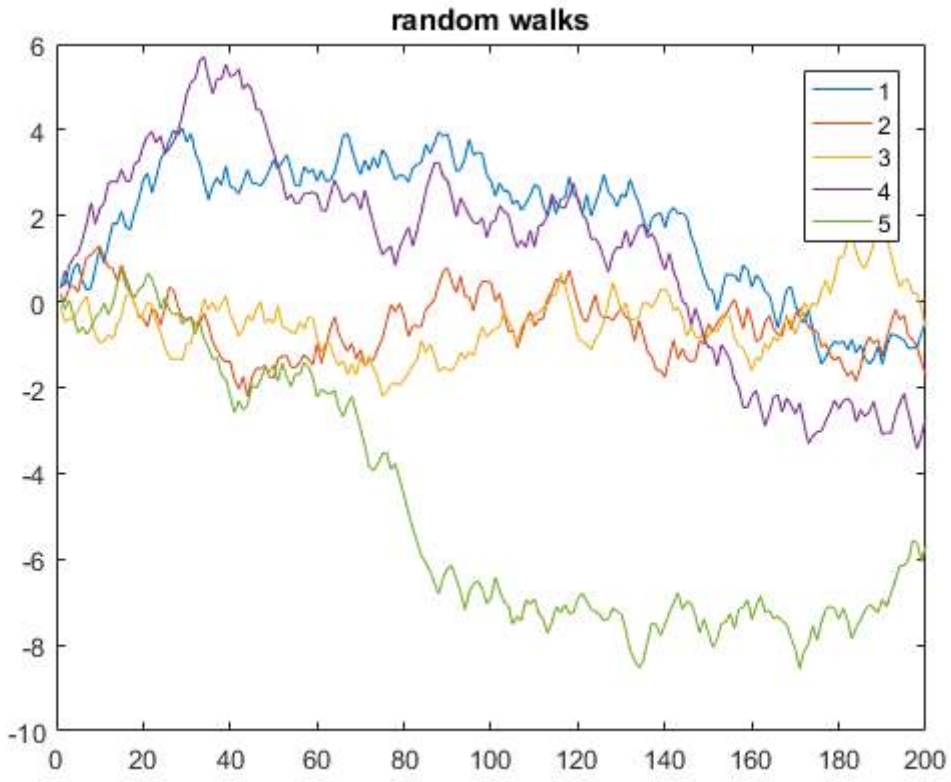
Chapter 17: Financial Applications

Section 17.1: Random Walk

The following is an example that displays 5 one-dimensional random walks of 200 steps:

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

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

In the above code, y is a matrix of 5 columns, each of length 200. Since x is omitted, it defaults to the row numbers of y (equivalent to using `x=1:200` as the x-axis). This way the `plot` function plots multiple y-vectors against the same x-vector, each using a different color automatically.



Section 17.2: Univariate Geometric Brownian Motion

The dynamics of the Geometric Brownian Motion (GBM) are described by the following stochastic differential equation (SDE):

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

I can use the **exact** solution to the SDE

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

to generate paths that follow a GBM.

Given daily parameters for a year-long simulation

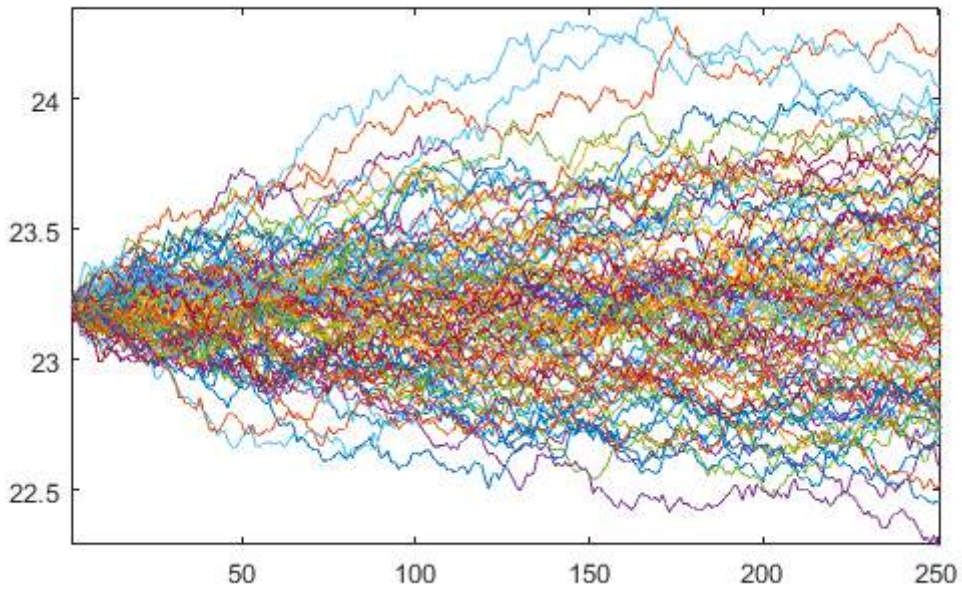
```
mu     = 0.08/250;
sigma  = 0.25/sqrt(250);
dt     = 1/250;
npaths = 100;
nsteps = 250;
S0     = 23.2;
```

we can get the Brownian Motion (BM) `W` starting at 0 and use it to obtain the GBM starting at `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);
```

Which produces the paths

```
plot(Y)
```



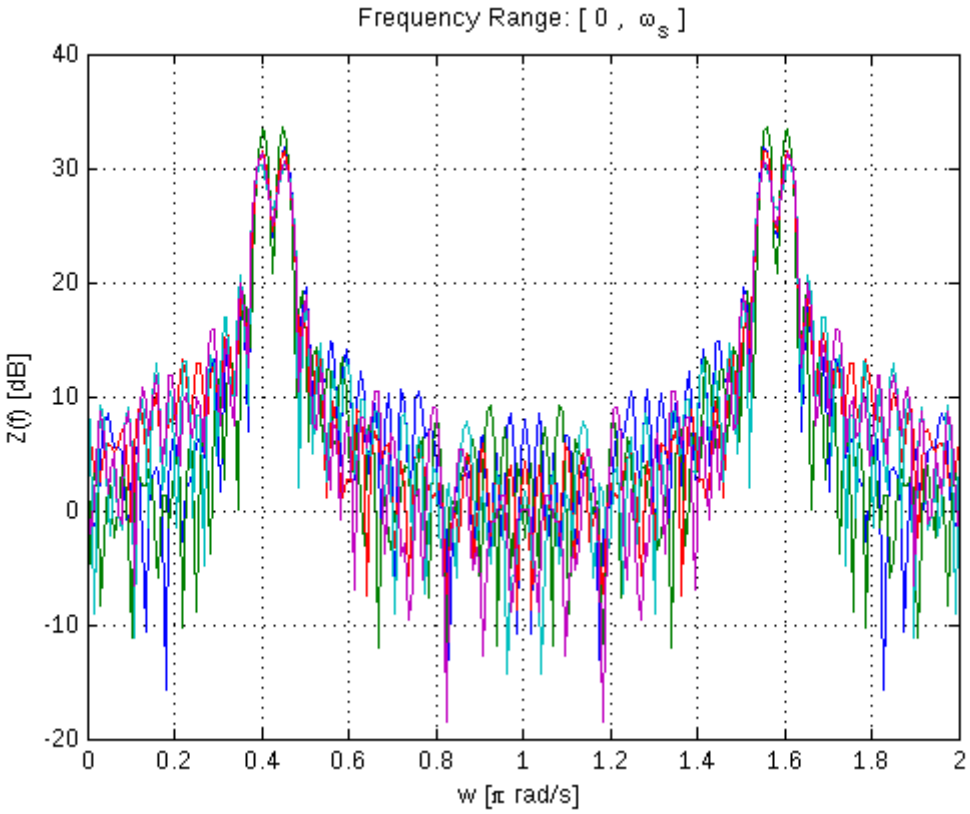
Chapter 18: Fourier Transforms and Inverse Fourier Transforms

Parameter	Description
X	this is your input Time-Domain signal, it should be a vector of numerics.
n	this is the NFFT parameter known as Transform Length, think of it as resolution of your FFT result, it MUST be a number that is a power of 2 (i.e. 64,128,256...2^N)
dim	this is the dimension you want to compute FFT on, use 1 if you want to compute your FFT in the horizontal direction and 2 if you want to compute your FFT in the vertical direction - Note this parameter is usually left blank, as the function is capable of detecting the direction of your vector.

Section 18.1: Implement a simple Fourier Transform in MATLAB

Fourier Transform is probably the first lesson in Digital Signal Processing, it's application is everywhere and it is a powerful tool when it comes to analyze data (in all sectors) or signals. MATLAB has a set of powerful toolboxes for Fourier Transform. In this example, we will use Fourier Transform to analyze a basic sine-wave signal and generate what is sometimes known as a Periodogram using 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
```



Note that the Discrete Fourier Transform is implemented by Fast Fourier Transform (fft) in MATLAB, both will yield the same result, but FFT is a fast implementation of 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 ]')
```

Section 18.2: Images and multidimensional FTs

In medical imaging, spectroscopy, image processing, cryptography and other areas of science and engineering it is often the case that one wishes to compute multidimensional Fourier transforms of images. This is quite straightforward in MATLAB: (multidimensional) images are just n-dimensional matrices, after all, and Fourier transforms are linear operators: one just iteratively Fourier transforms along other dimensions. MATLAB provides `fft2` and `ifft2` to do this in 2-d, or `fftn` in n-dimensions.

One potential pitfall is that the Fourier transform of images are usually shown "centric ordered", i.e. with the origin of k-space in the middle of the picture. MATLAB provides the `fftshift` command to swap the location of the DC components of the Fourier transform appropriately. This ordering notation makes it substantially easier to perform common image processing techniques, one of which is illustrated below.

Zero filling

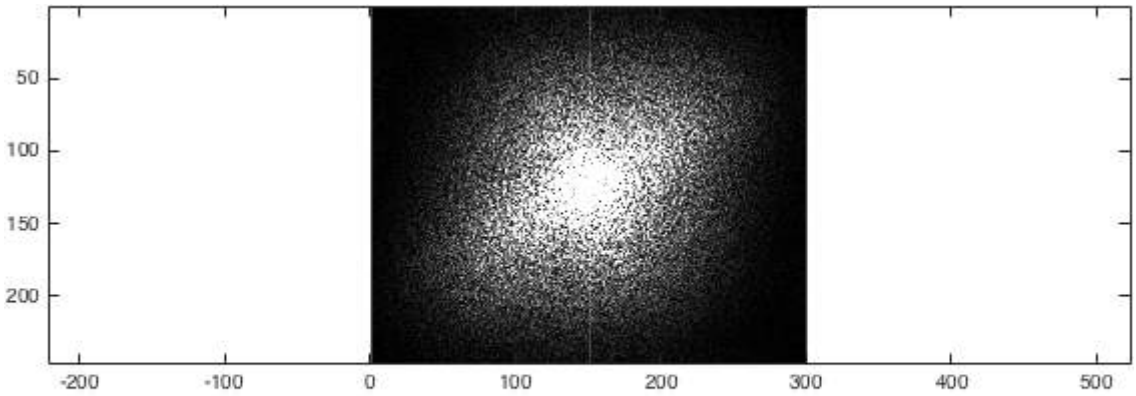
One "quick and dirty" way to interpolate a small image to a larger size is to Fourier transform it, pad the Fourier transform with zeros, and then take the inverse transform. This effectively interpolates between each pixel with a sinc shaped basis function, and is commonly used to up-scale low resolution medical imaging data. Let's start by loading a built-in image example

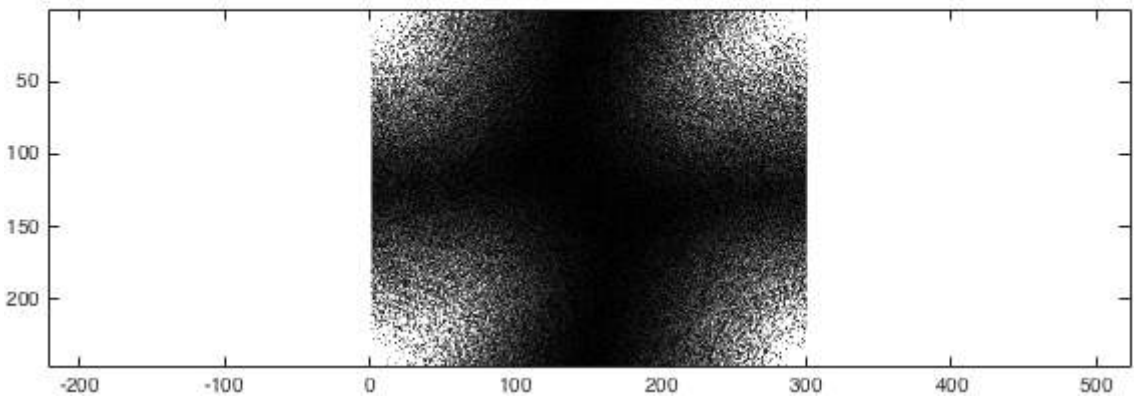
```
%Load example image
```

```
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
```

We can now obtain the Fourier transform of I. To illustrate what `fftshift` does, let's compare the two methods:

```
% 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).
```

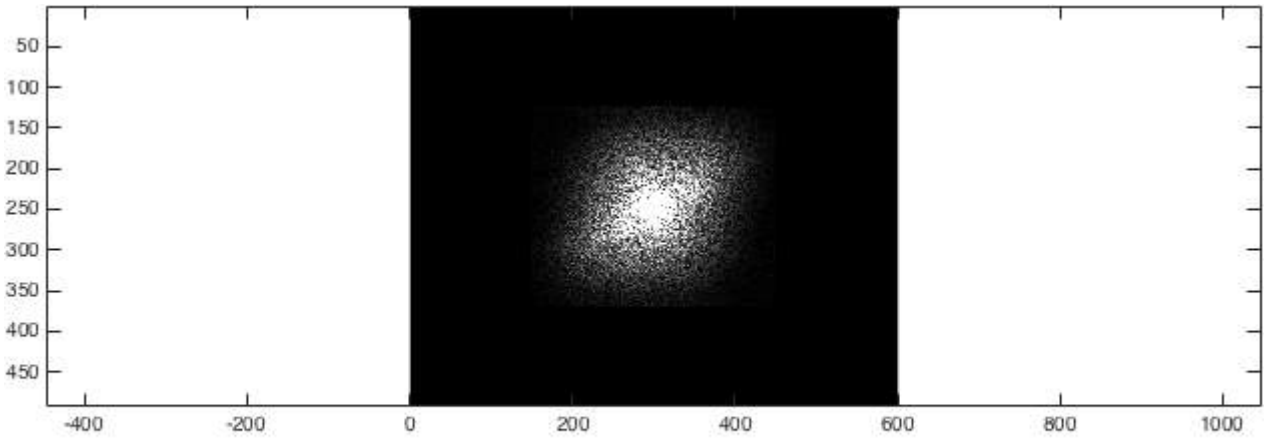


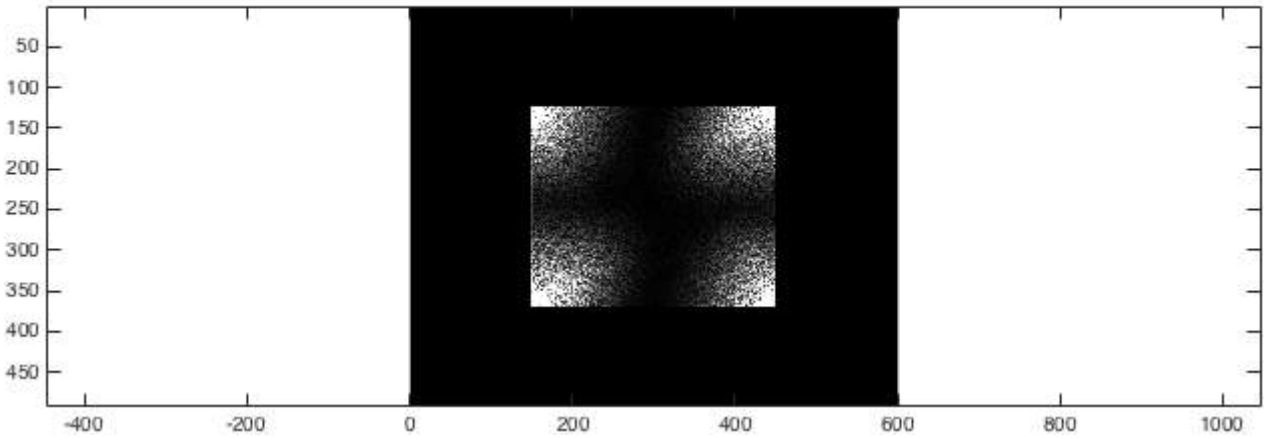


We now have obtained the 2D FT of an example image. To zero-fill it, we want to take each k-space, pad the edges with zeros, and then take the back transform:

```
%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).
```

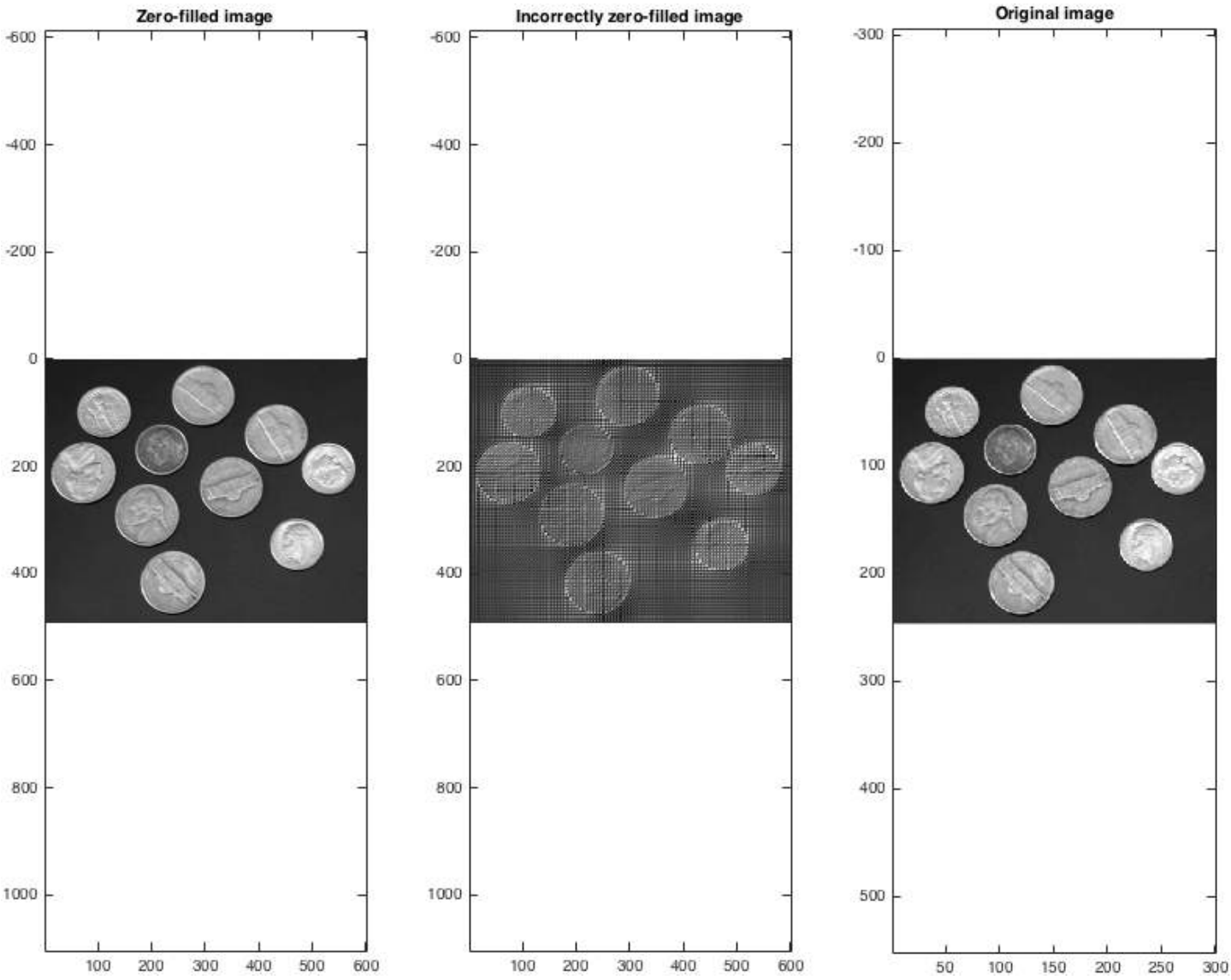
At this point, the result fairly unremarkable:





Once we then take the back-transforms, we can see that (correctly!) zero-filling data provides a sensible method for interpolation:

```
% 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');
```



Note that the zero-filled image size is double that of the original. One can zero fill by more than a factor of two in each dimension, although obviously doing so does not arbitrarily increase the size of an image.

Hints, tips, 3D and beyond

The above example holds for 3D images (as are often generated by medical imaging techniques or confocal microscopy, for example), but require `fft2` to be replaced by `fftn(I, 3)`, for example. Due to the somewhat cumbersome nature of writing `fftshift(fft(fftshift(...` several times, it is quite common to define functions such as `fft2c` locally to provide easier syntax locally -- such as:

```
function y = fft2c(x)

y = fftshift(fft2(fftshift(x)));
```

Note that the FFT is fast, but large, multidimensional Fourier transforms will still take time on a modern computer. It is additionally inherently complex: the magnitude of k-space was shown above, but the phase is absolutely vital; translations in the image domain are equivalent to a phase ramp in the Fourier domain. There are several far more complex operations that one may wish to do in the Fourier domain, such as filtering high or low spatial frequencies (by multiplying it with a filter), or masking out discrete points corresponding to noise. There is correspondingly a large quantity of community generated code for handling common Fourier operations available on MATLAB's main community repository site, the File Exchange.

Section 18.3: Inverse Fourier Transforms

One of the major benefit of Fourier Transform is its ability to inverse back in to the Time Domain without losing information. Let us consider the same Signal we used in the previous example:

```
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
```

If we open `F` in MATLAB, we will find that it is a matrix of complex numbers, a real part and an imaginary part. By definition, in order to recover the original Time Domain signal, we need both the Real (which represents Magnitude variation) and the Imaginary (which represents Phase variation), so to return to the Time Domain, one may simply want to:

```
TD = ifft(F,NFFT);   %Returns the Inverse of F in Time Domain
```

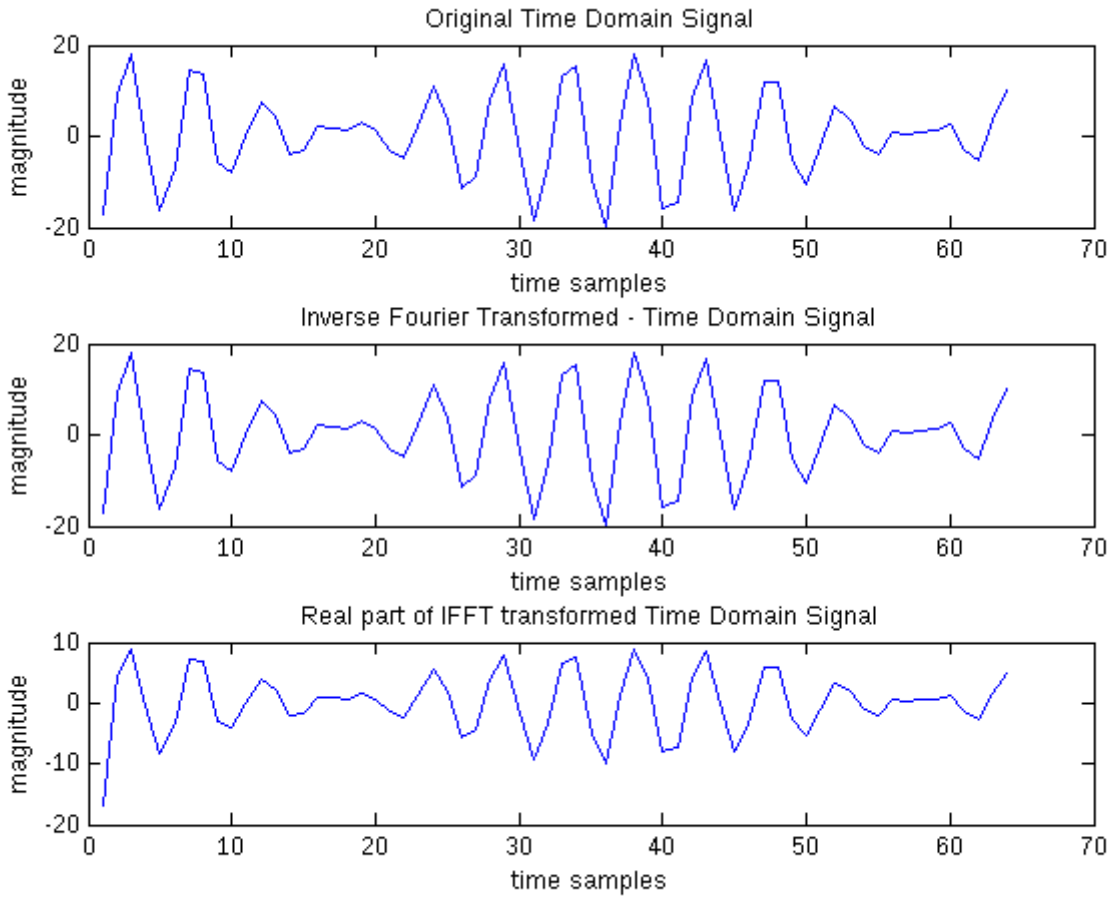
Note here that TD returned would be length 256 because we set NFFT to 256, however, the length of x is only 64, so MATLAB will pad zeros to the end of the TD transform. So for example, if NFFT was 1024 and the length was 64, then TD returned will be 64 + 960 zeros. Also note that due to floating point rounding, you might get something like 3.1 * 10e-20 but for general purposed: For any X, ifft(fft(X)) equals X to within roundoff error.

Let us say for a moment that after the transformation, we did something and are only left with the REAL part of the 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
```

This means that we are losing the imaginary part of our FFT, and therefore, we are losing information in this reverse process. To preserve the original without losing information, you should always keep the imaginary part of the FFT using `imag` and apply your functions to either both or the real part.

```
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')
```


Original Time Domain Signal
magnitude
time samples
Inverse Fourier Transformed - Time Domain Signal
magnitude
time samples
Real part of IFFT transformed Time Domain Signal
magnitude
time samples

Chapter 19: Ordinary Differential Equations (ODE) Solvers

Section 19.1: Example for odeset

First we initialize our initial value problem we want to solve.

```
odefun = @(t,y) cos(y).^2*sin(t);
tspan = [0 16*pi];
y0=1;
```

We then use the ode45 function without any specified options to solve this problem. To compare it later we plot the trajectory.

```
[t,y] = ode45(odefun, tspan, y0);
plot(t,y,'-o');
```

We now set a narrow relative and a narrow absolute limit of tolerance for our problem.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

We set tight relative and narrow absolute limit of tolerance.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-2);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

We set narrow relative and tight absolute limit of tolerance. As in the previous examples with narrow limits of tolerance one sees the trajectory being completely different from the first plot without any specific options.

```
options = odeset('RelTol',1e-2,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

We set tight relative and tight absolute limit of tolerance. Comparing the result with the other plot will underline the errors made calculating with narrow tolerance limits.

```
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
plot(t,y,'-o');
```

The following should demonstrate the trade-off between precision and run-time.

```
tic;
options = odeset('RelTol',1e-7,'AbsTol',1e-7);
[t,y] = ode45(odefun, tspan, y0, options);
time1 = toc;
plot(t,y,'-o');
```

For comparison we tighten the limit of tolerance for absolute and relative error. We now can see that without large gain in precision it will take considerably longer to solve our initial value problem.

```
tic;
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
[t,y] = ode45(odefun, tspan, y0, options);
time2 = toc;
plot(t,y,'-o');
```

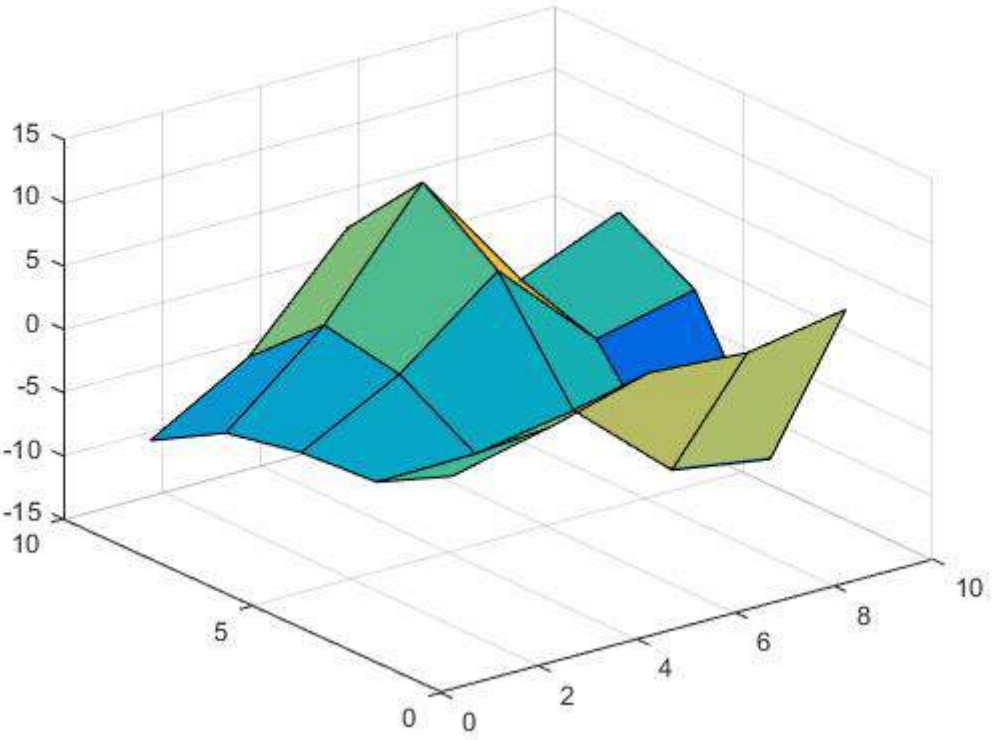
Chapter 20: Interpolation with MATLAB

Section 20.1: Piecewise interpolation 2 dimensional

We initialize the data:

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

The surface looks like the following.

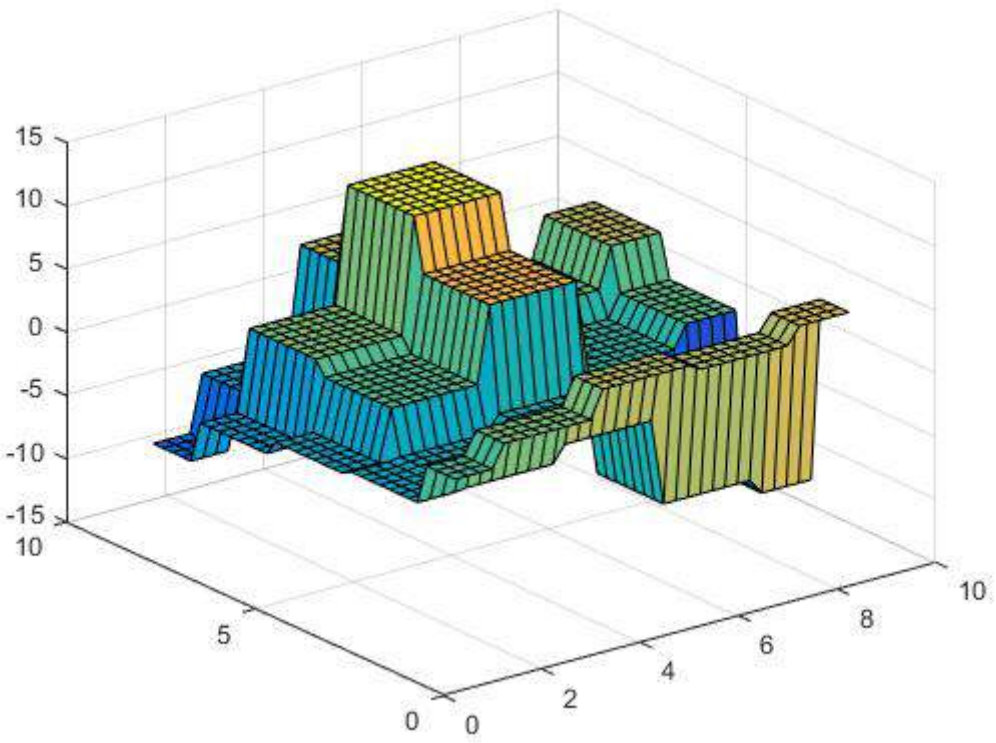


Now we set the points where we want to interpolate:

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

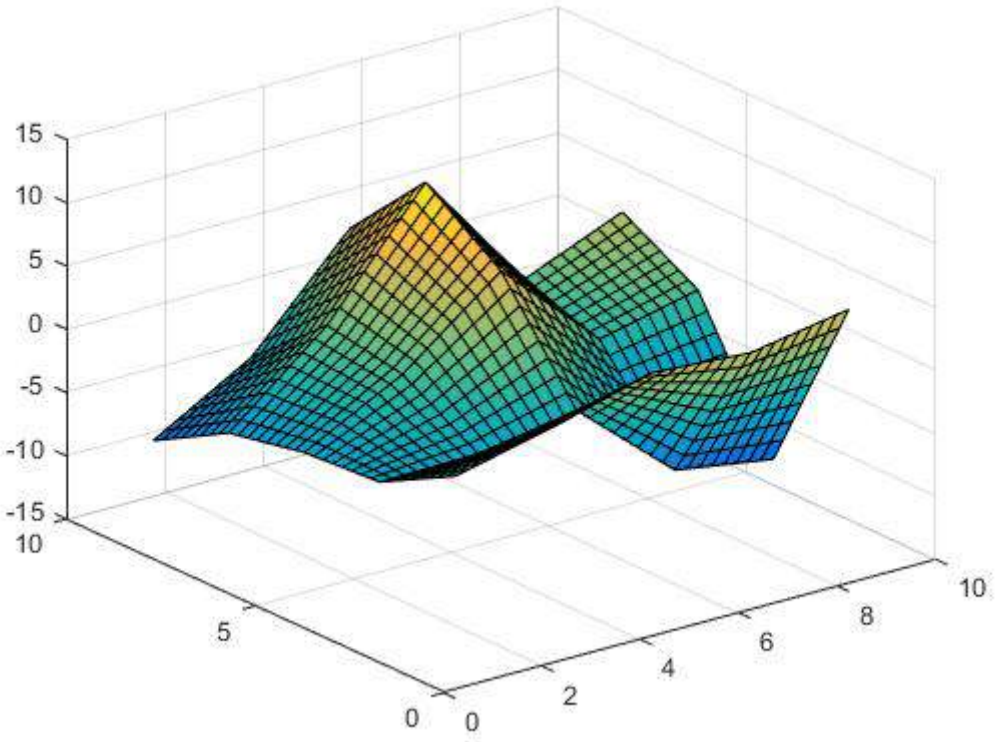
We now can perform nearest interpolation,

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



linear interpolation,

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

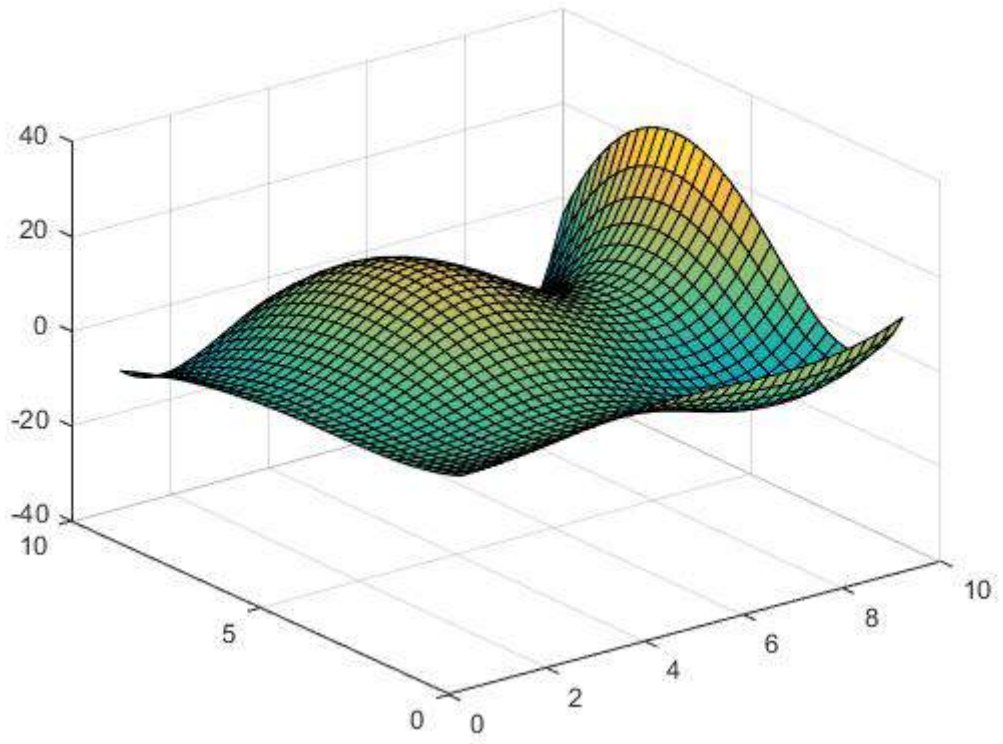


cubic interpolation

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

or spline interpolation:

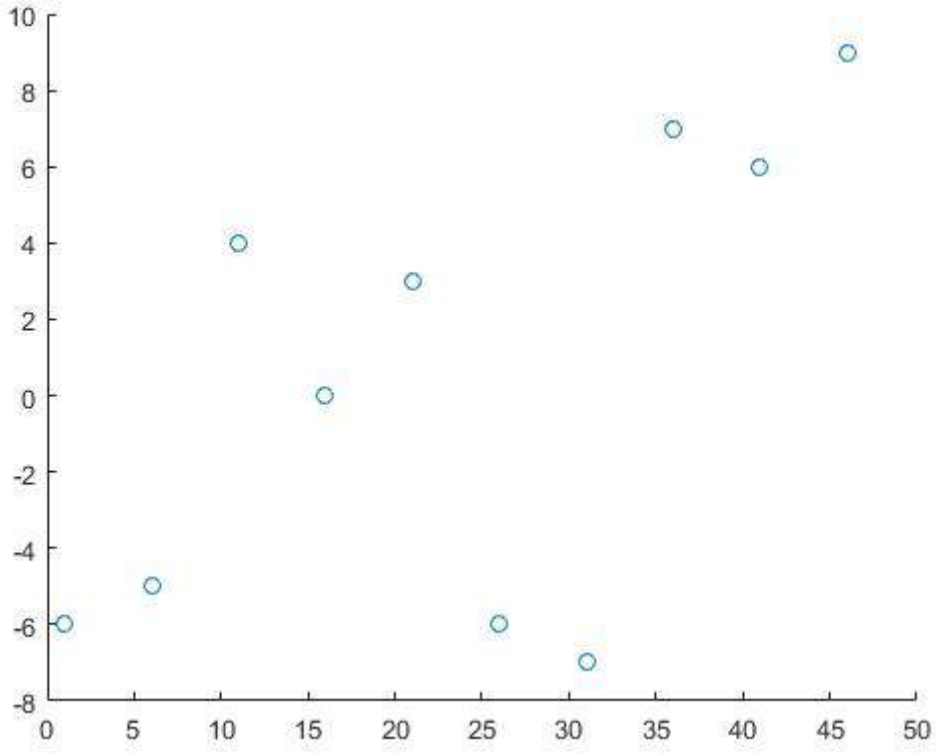
```
Vz = interp2(X,Y,Z,Vx,Vy,'spline');
```



Section 20.2: Piecewise interpolation 1 dimensional

We will use the following data:

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

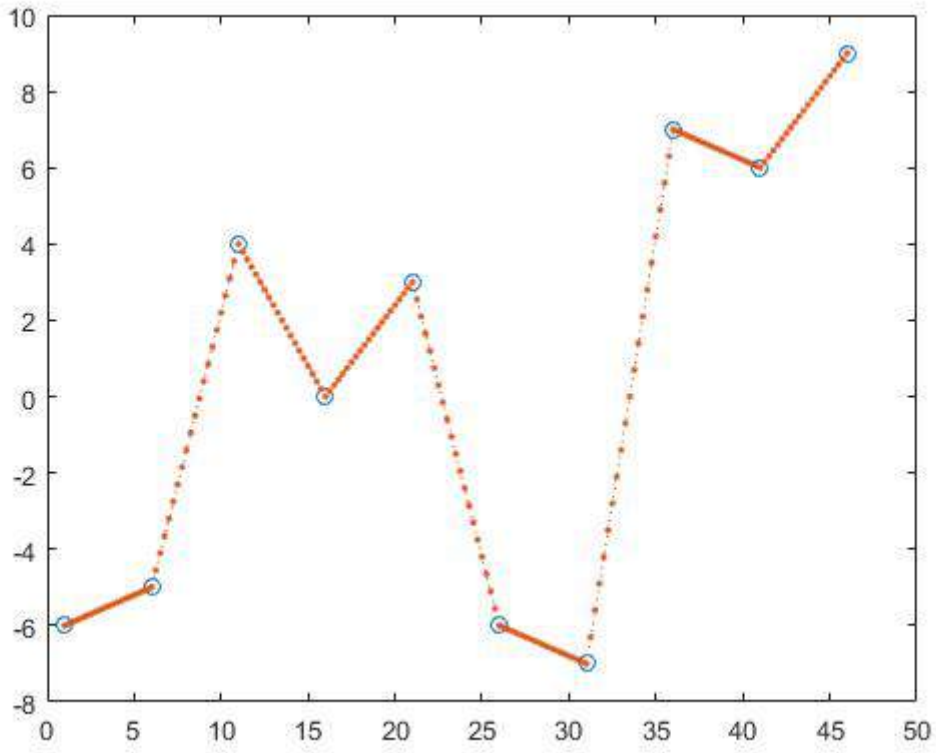


Hereby x and y are the coordinates of the data points and z are the points we need information about.

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

One way to find the y-values of z is piecewise linear interpolation.

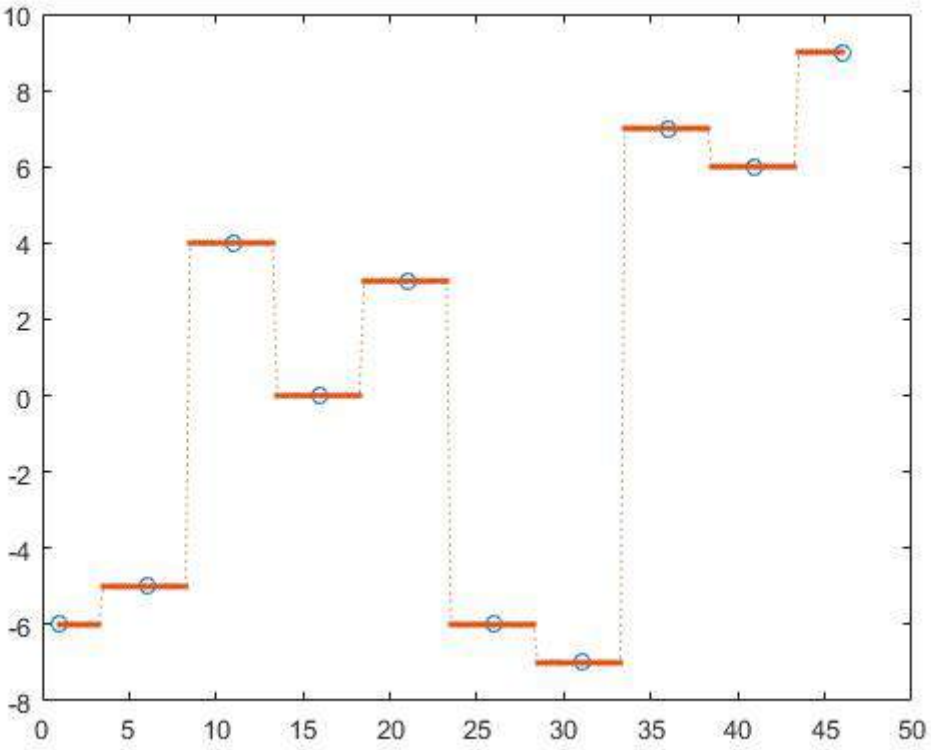
```
z_y = interp1(x,y,z,'linear');
```



Hereby one calculates the line between two adjacent points and gets `z_y` by assuming that the point would be an element of those lines.

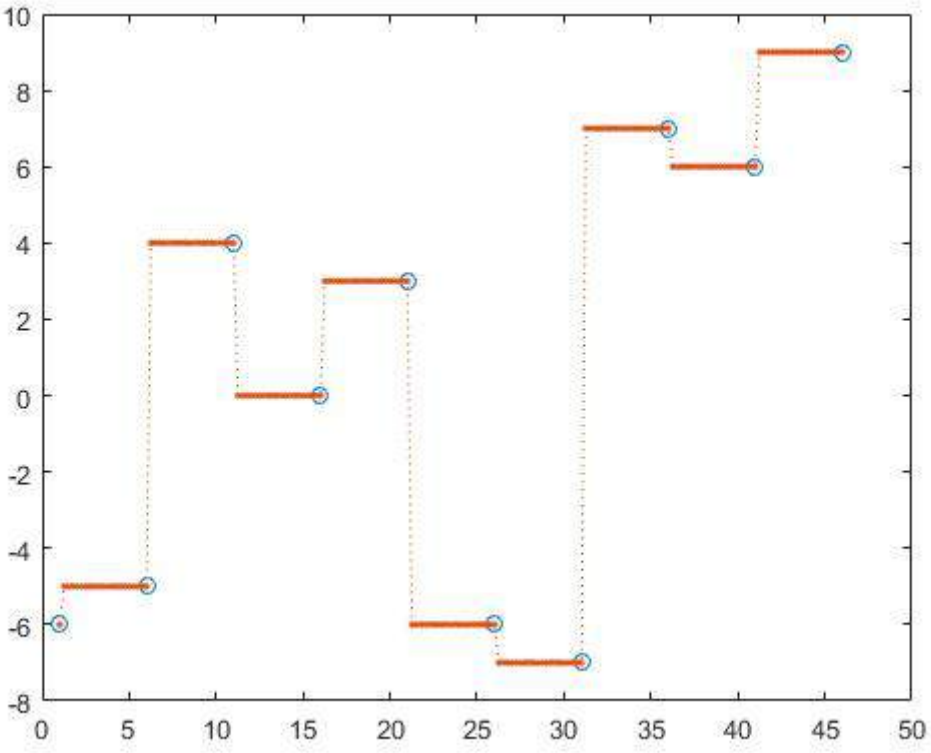
`interp1` provides other options too like nearest interpolation,

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



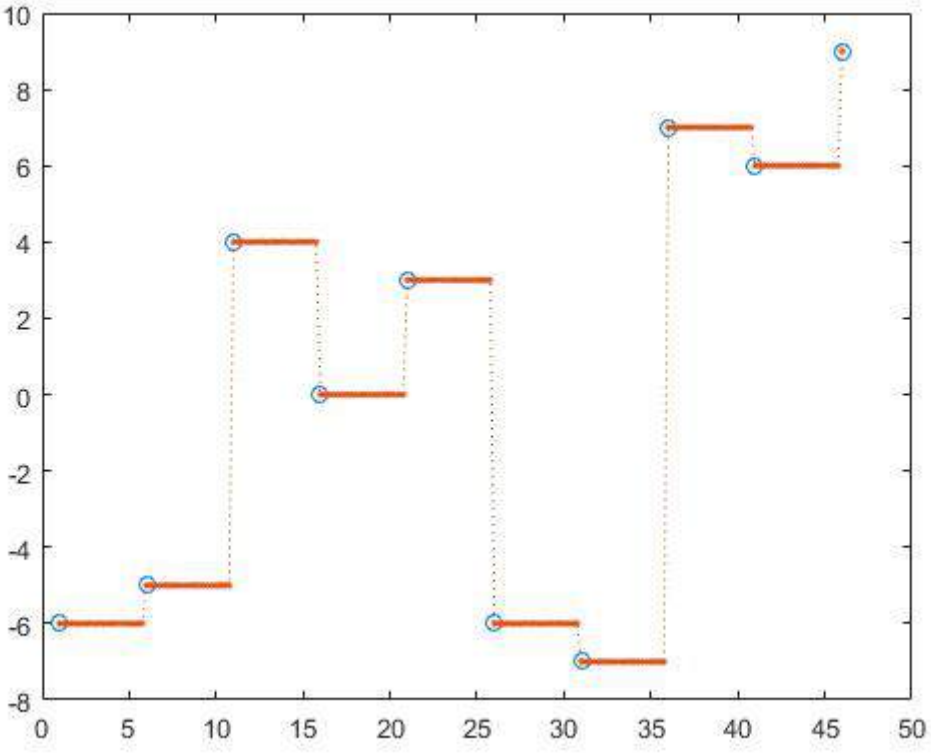
next interpolation,

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



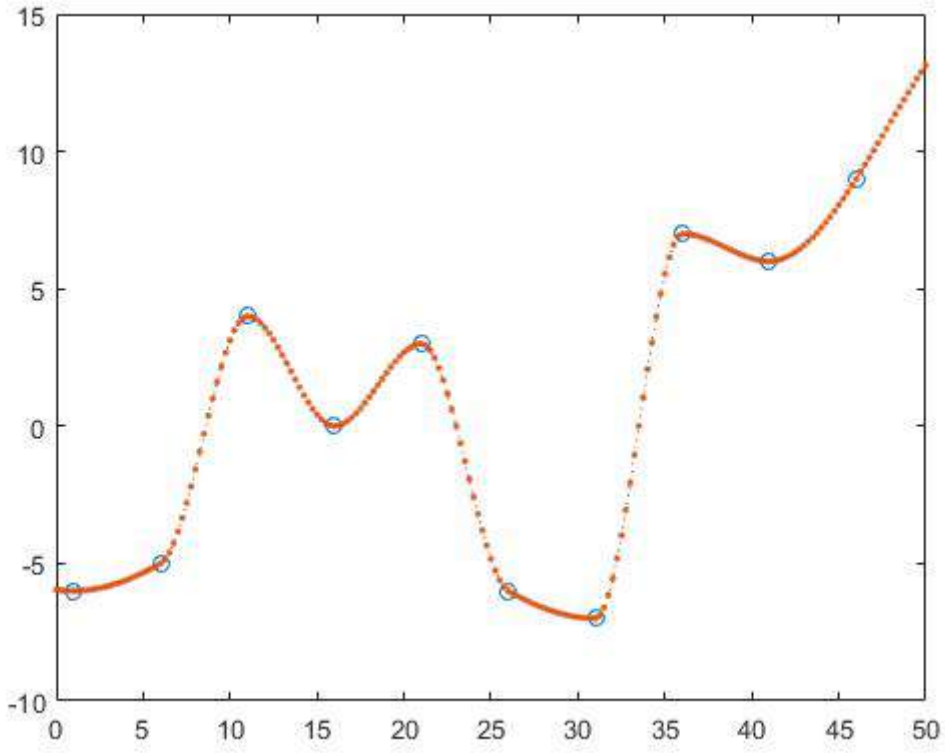
previous interpolation,

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

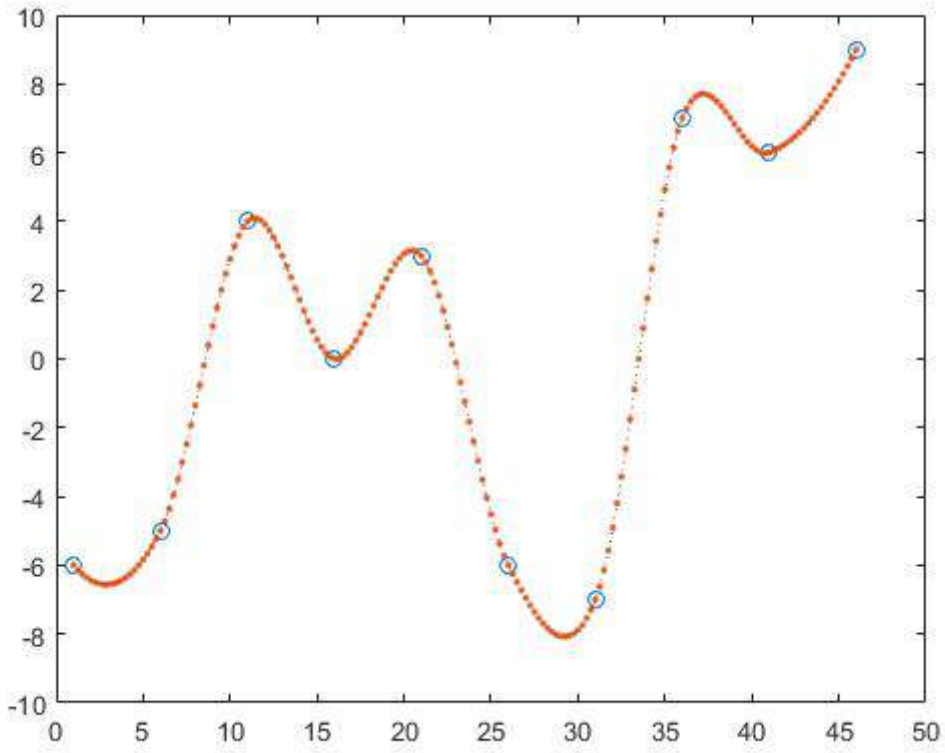


Shape-preserving piecewise cubic interpolation,

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

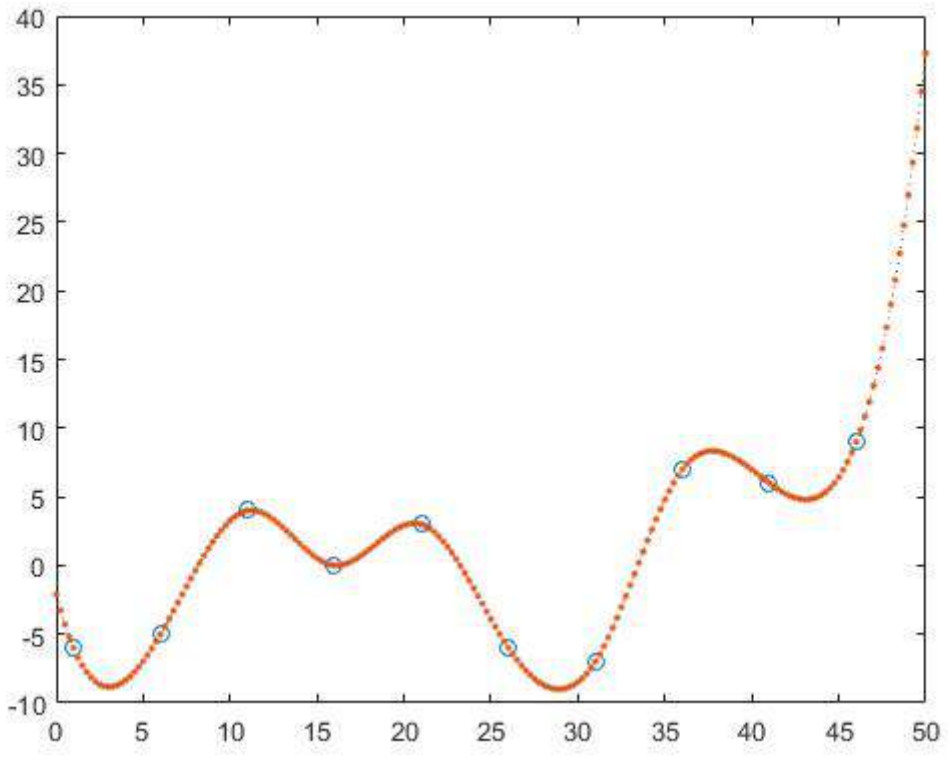


cubic convolution, z_y = interp1(x,y,z, 'v5cubic');



and spline interpolation

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



Hereby are nearst, next and previous interpolation piecewise constant interpolations.

Section 20.3: Polynomial interpolation

We initialize the data we want to interpolate:

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

This means the underlying function for the data in the interval [0,10] is sinusoidal. Now the coefficients of the approximating polynomials are being calculated:

```
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);
```

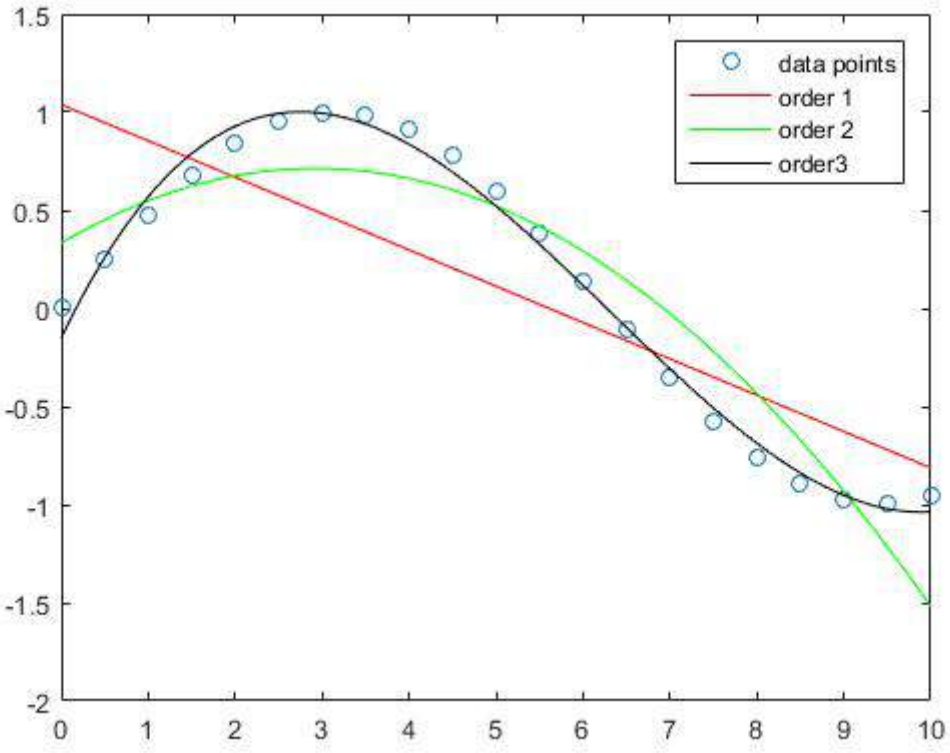
Hereby is `x` the x-value and `y` the y-value of our data points and the third number is the order/degree of the polynomial. We now set the grid we want to compute our interpolating function on:

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

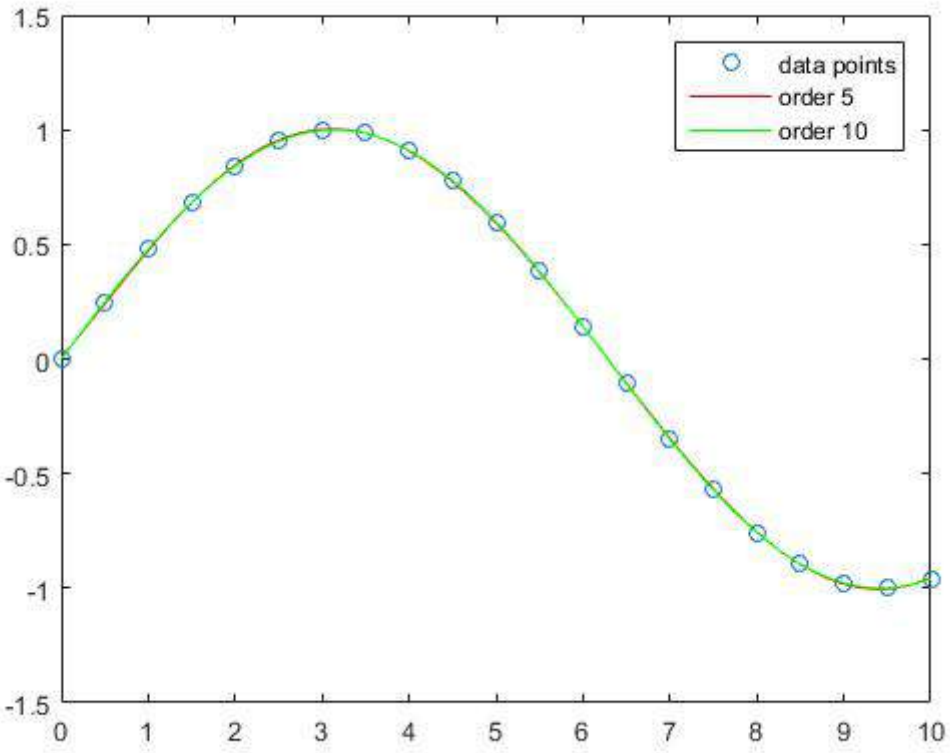
and calculate the y-values:

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

One can see that the approximation error for the sample gets smaller when the degree of the polynomial increases.



While the approximation of the straight line in this example has larger errors the order 3 polynomial approximates the sinus function in this interval relatively good.



The interpolation with order 5 and order 10 polynomials has almost no approximation error.

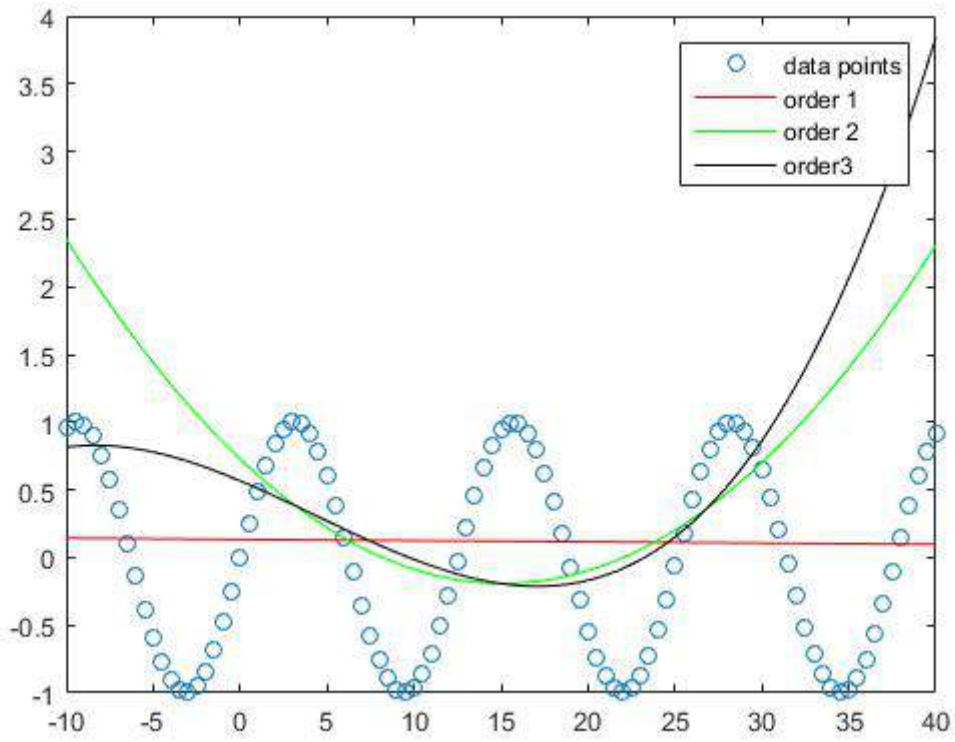
However if we consider the out of sample performance one sees that too high orders tend to overfit and therefore perform badly out of sample. We set

```
zx = -10:0.1:40;
p10 = polyfit(X,Y,10);
p20 = polyfit(X,Y,20);
```

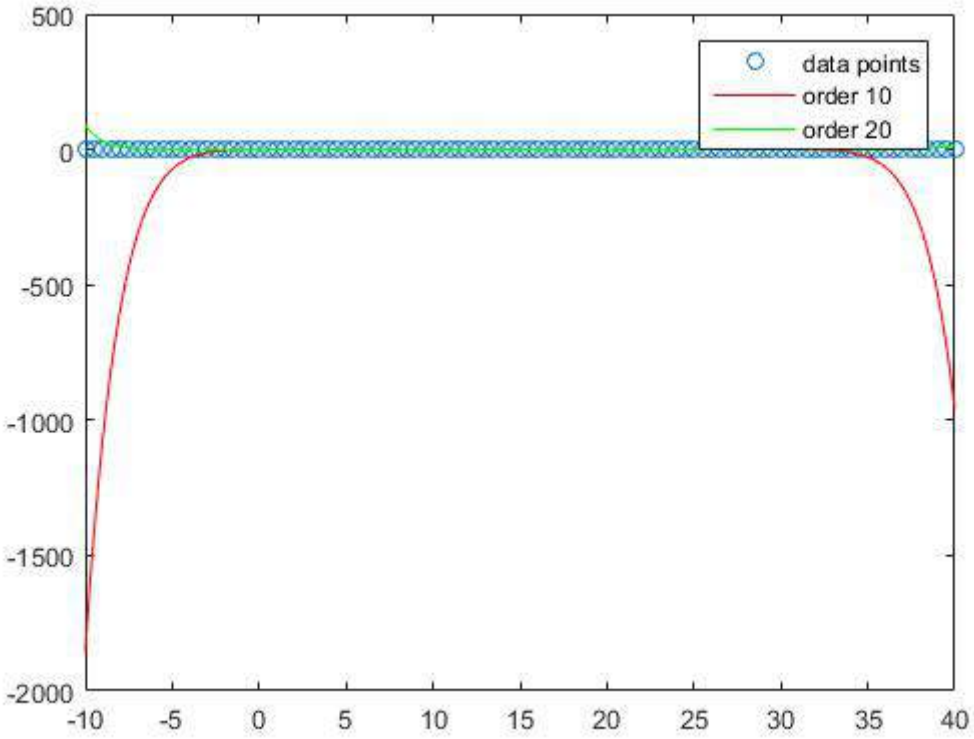
and

```
zy10 = polyval(p10,zx);
zy20 = polyval(p20,zx);
```

If we take a look at the plot we see that the out of sample performance is best for the order 1



and keeps getting worse with increasing degree.


500
0
-500
-1000
-1500
-2000
-10
-5
0
5
10
15
20
25
30
35
40
data points
order 10
order 20

Chapter 21: Integration

Section 21.1: Integral, integral2, integral3

1 dimensional

To integrate a one dimensional function

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

within the range

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

one can call the function

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

it's also possible to set boundaries for relative and absolute errors

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

2 dimensional

If one wants to integrate a two dimensional function

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

within the range

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

one calls the function

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

Like in the other case it's possible to limit the tolerances

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

3 dimensional

Integrating a three dimensional function

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

within the range

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

```
ymin = 1;
ymax = 4;
zmin = 6;
zmax = 13;
```

is performed by calling

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

Again it's possible to limit the tolerances

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

Chapter 22: Reading large files

Section 22.1: textscan

Assume you have formatted data in a large text file or string, e.g.

```
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
```

one may use `textscan` to read this quite fast. To do so, get a file identifier of the text file with `fopen`:

```
fid = fopen('path/to/myfile');
```

Assume for the data in this example, we want to ignore the first column "Data", read the date and time as strings, and read the rest of the columns as doubles, i.e.

```
 Data   ,   2015-09-16   , 15:41:52;801   , 800.000000   , 1.5123   ,   0.0043
ignore       string           string          double       double      double
```

To do this, call:

```
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

The asterisk in `%*s` means "ignore this column". `%s` means "interpret as a string". `%f` means "interpret as doubles (floats)". Finally, `'Delimiter',','` states that all commas should be interpreted as the delimiter between each column.

To sum up:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %s %s %f %f %f','Delimiter',',');
```

`data` now contains a cell array with each column in a cell.

Section 22.2: Date and time strings to numeric array fast

Converting date and time strings to numeric arrays can be done with `datenum`, though it may take as much as half the time of reading a large data file.

Consider the data in example **Textscan**. By, again, using textscan and interpret date and time as integers, they can rapidly be converted into a numeric array.

I.e. a line in the example data would be interpreted as:

```
 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
```

which will be read as:

```
fid = fopen('path/to/myfile');
data = textscan(fid,'%*s %f %f %f %f %f %f %f %f %f %f','Delimiter',',-:;');
fclose(fid);
```

Now:

```
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
```

Using `datenum` on 566,678 elements required 6.626570 seconds, whilst the method above required 0.048334 seconds, i.e. 0.73% of the time for `datenum` or ~137 times faster.

Chapter 23: Usage of `accumarray()` Function

Parameter	Details
`subscriptArray`	Subscript matrix, specified as a vector of indices, matrix of indices, or cell array of index vectors.
`valuesArray`	Data, specified as a vector or a scalar.
`sizeOfOutput`	Size of output array, specified as a vector of positive integers.
`funcHandle`	Function to be applied to each set of items during aggregation, specified as a function handle or `[]`.
`fillVal`	Fill value, for when `subs` does not reference each element in the output.
`isSparse`	Should the output be a sparse array?

`accumarray` allows to aggregate items of an array in various ways, potentially applying some function to the items in the process. `accumarray` can be thought of as a lightweight reducer (see also: Introduction to MapReduce).

This topic will contain common scenarios where `accumarray` is especially useful.

Section 23.1: Apply Filter to Image Patches and Set Each Pixel as the Mean of the Result of Each Patch

Many modern Image Processing algorithms use patches are their basic element to work on.
For instance one could denoise patches (See BM3D Algorithm).

Yet when building the image form the processed patches we have many results for the same pixel.
One way to deal with it is taking the average (Empirical Mean) of all values of the same pixel.

The following code shows how to break an image into patches and them reconstruct the image from patches using the average by using `[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
% boundaries, 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));
```

```
% Reshape the Vector into the Image
mO = reshape(mO, [numRows, numCols]);
```

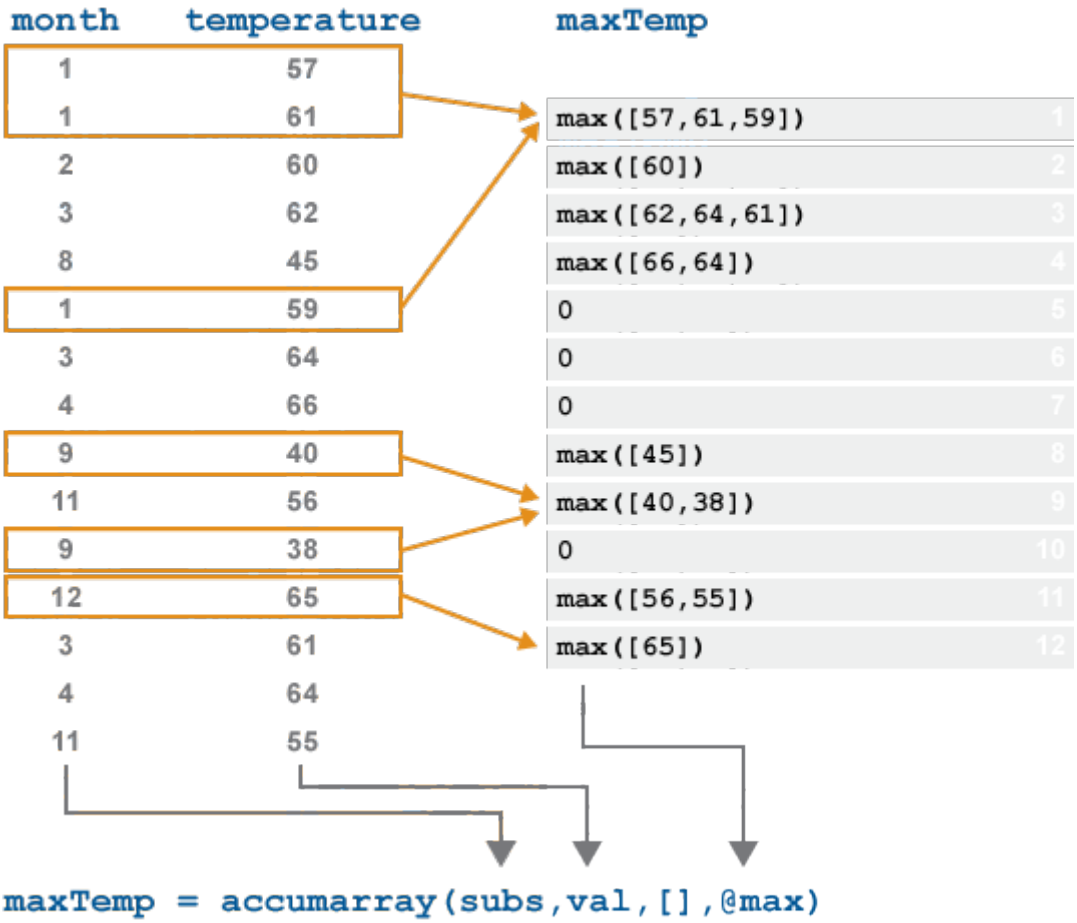
Section 23.2: Finding the maximum value among elements grouped by another vector

This is an official MATLAB example

Consider the following code:

```
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);
```

The image below demonstrates the computation process done by `accumarray` in this case:



In this example, all values that have the same `month` are first collected, and then the function specified by the 4th input to `accumarray` (in this case, `@max`) is applied to each such set.

Chapter 24: Introduction to MEX API

Section 24.1: Check number of inputs/outputs in a C++ MEX-file

In this example we will write a basic program that checks the number of inputs and outputs passed to a MEX-function.

As a starting point, we need to create a C++ file implementing the "MEX gateway". This is the function executed when the file is called from 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 it is 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.");
    }
}
```

First, we include the `mex.h` header which contains definitions of all the required functions and data types to work with the MEX API. Then we implement the function `mexFunction` as shown, where its signature must not change, independent of the inputs/outputs actually used. The function parameters are as follows:

- `nlhs`: Number of outputs requested.
- `*plhs[]`: Array containing all the outputs in MEX API format.
- `nrhs`: Number of inputs passed.
- `*prhs[]`: Array containing all the inputs in MEX API format.

Next, we check the number of inputs/outputs arguments, and if the validation fails, an error is thrown using `mexErrMsgIdAndTxt` function (it expects `somename:iD` format identifier, a simple "ID" won't work).

Once the file is compiled as `mex testinputs.cpp`, the function can be called in MATLAB as:

```
>> 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.
```

Section 24.2: Input a string, modify it in C, and output it

In this example, we illustrate string manipulation in MATLAB MEX. We will create a MEX-function that accepts a string as input from MATLAB, copy the data into C-string, modify it and convert it back to `mxArray` returned to the MATLAB side.

The main objective of this example is to show how strings can be converted to C/C++ from MATLAB and vice versa.

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);
}
```

The relevant functions in this example are:

- `mxIsChar` to test if an `mxArray` is of `mxCHAR` type.
- `mxArrayToString` to copy the data of a `mxArray` string to a `char *` buffer.
- `mxCreateString` to create an `mxArray` string from a `char*`.

As a side note, if you only want to read the string, and not modify it, remember to declare it as `const char*` for speed and robustness.

Finally, once compiled we can call it from MATLAB as:

```
>> mex stringIO.cpp

>> strOut = stringIO('theOneString')
strOut =
TheOneString

>> strOut = stringIO('somethingelse')
```

```
strOut=
omethingelse
```

Section 24.3: Passing a struct by field names

This example illustrates how to read various-type struct entries from MATLAB, and pass it to C equivalent type variables.

While it is possible and easy to figure out from the example how to load fields by numbers, it is here achieved via comparing the field names to strings. Thus the struct fields can be addressed by their field names and variables in it can be read by 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);
    }
  }
}
```

The loop over `i` runs over every field in the given struct, while the `if(0==strcmp)`-parts compare the MATLAB field's name to the given string. If it is a match, the corresponding value is extracted to a C variable.

Section 24.4: Pass a 3D matrix from MATLAB to C

In this example we illustrate how to take a double real-type 3D matrix from MATLAB, and pass it to a C `double*` array.

The main objectives of this example are showing how to obtain data from MATLAB MEX arrays and to highlight some small details in matrix storage and handling.

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;
}
```

The relevant concepts to be aware of:

- MATLAB matrices are all 1D in memory, no matter how many dimensions they have when used in MATLAB. This is also true for most (if not all) main matrix representation in C/C++ libraries, as allows optimization and faster execution.
- You need to explicitly copy matrices from MATLAB to C in a loop.
- MATLAB matrices are stored in column major order, as in Fortran, but C/C++ and most modern languages are row major. It is important to permute the input matrix , or else the data will look completely different.

The relevant function in this example are:

- `mxIsDouble` checks if input is `double` type.
- `mxIsComplex` checks if input is real or imaginary.
- `mxGetData` returns a pointer to the real data in the input array. `NULL` if there is no real data.
- `mxGetDimensions` returns an pointer to a `mwSize` array, with the size of the dimension in each index.

Chapter 25: Debugging

Parameter	Details
file	Name of `.m` file (without extension), e.g. `fit`. This parameter is *(Required)* unless setting special conditional breakpoint types such as `dbstop if error` or `dbstop if naninf`.
location	Line number where the breakpoint should be placed. If the specified line does not contain runnable code, the breakpoint will be placed on the first valid line **after** the specified one.
expression	Any expression or combination thereof that evaluates to a boolean value. Examples: `ind == 1`, `nargin < 4 && isdir('Q:\')`.

Section 25.1: Working with Breakpoints

Definition

> *In software development, a **breakpoint** is an intentional stopping or pausing place in a program, put in place for debugging purposes.*
>
> *More generally, a breakpoint is a means of acquiring knowledge about a program during its execution. During the interruption, the programmer inspects the test environment (general purpose registers, memory, logs, files, etc.) to find out whether the program is functioning as expected. In practice, a breakpoint consists of one or more conditions that determine when a program's execution should be interrupted.*
>
> -Wikipedia

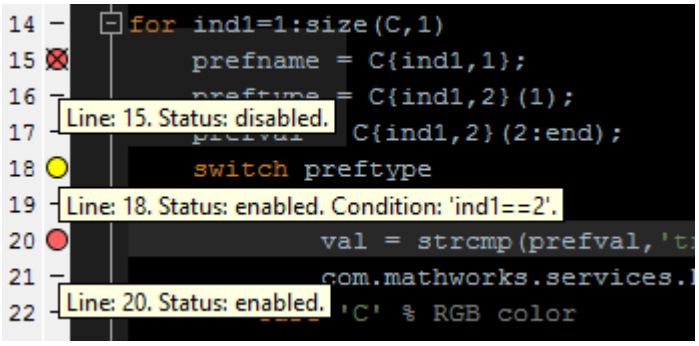
Breakpoints in MATLAB
Motivation

In MATLAB, when execution pauses at a breakpoint, variables existing in the current workspace (a.k.a. *scope*) or any of the calling workspaces, can be inspected (and usually also modified).

Types of Breakpoints

MATLAB allow users to place two types of breakpoints in `.m` files:

- Standard (or "*unrestricted*") breakpoints (shown in red) - pause execution whenever the marked line is reached.
- "Conditional" breakpoints (shown in yellow) - pause execution whenever the marked line is reached AND the condition defined in the breakpoint is evaluated as `true`.



Placing Breakpoints

Both types of breakpoints can be created in several ways:

- Using the MATLAB Editor GUI, by right clicking the horizontal line next to the line number.
- Using the `dbstop` command:

```
% 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
```

- Using keyboard shortcuts: the default key for creating a standard breakpoint on Windows is F12; the default key for conditional breakpoints is *unset*.

Disabling and Re-enabling Breakpoints

Disable a breakpoint to temporarily ignore it: disabled breakpoints do not pause execution. Disabling a breakpoint can be done in several ways:

- Right click on the red/yellow breakpoint circle > Disable Breakpoint.
- Left click on a conditional (yellow) breakpoint.
- In the Editor tab > Breakpoints > Enable\Disable.

Removing Breakpoints

All breakpoints remain in a file until removed, either manually or automatically. Breakpoints are cleared *automatically* when ending the MATLAB session (i.e. terminating the program). Clearing breakpoints manually is done in one of the following ways:

- Using the `dbclear` command:

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

- Left clicking a standard breakpoint icon, or a disabled conditional breakpoint icon.
- Right clicking on any breakpoint > Clear Breakpoint.
- In the Editor tab > Breakpoints > Clear All.
- In pre-R2015b versions of MATLAB, using the command `clear`.

Resuming Execution

When execution is paused at a breakpoint, there are two ways to continue executing the program:

- Execute the current line and pause again before the next line.

  F10 [1] in the Editor, `dbstep` in the Command Window, "Step" in Ribbon > Editor > DEBUG.

- Execute until the next breakpoint (if there are no more breakpoints, the execution proceeds until the end of the program).

  F12 [1] in the Editor, `dbcont` in the Command Window, "Continue" in Ribbon > Editor > DEBUG.

1 - default on Windows.

Section 25.2: Debugging Java code invoked by MATLAB

Overview

In order to debug Java classes that are called during MATLAB execution, it is necessary to perform two steps:

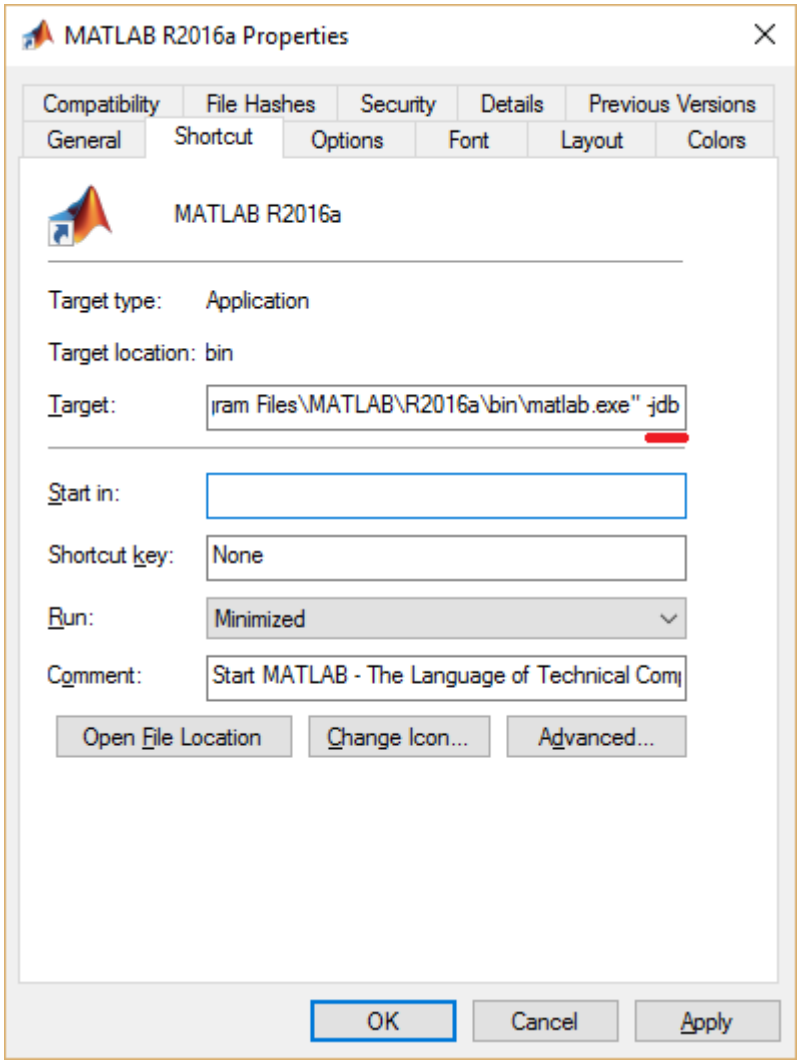
1. Run MATLAB in JVM debugging mode.
2. Attach a Java debugger to the MATLAB process.

When MATLAB is started in JVM debugging mode, the following message appears in the command window:

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

MATLAB end
Windows:

Create a shortcut to the MATLAB executable (`matlab.exe`) and add the `-jdb` flag at the end as shown below:



When running MATLAB using this shortcut JVM debugging will be enabled.

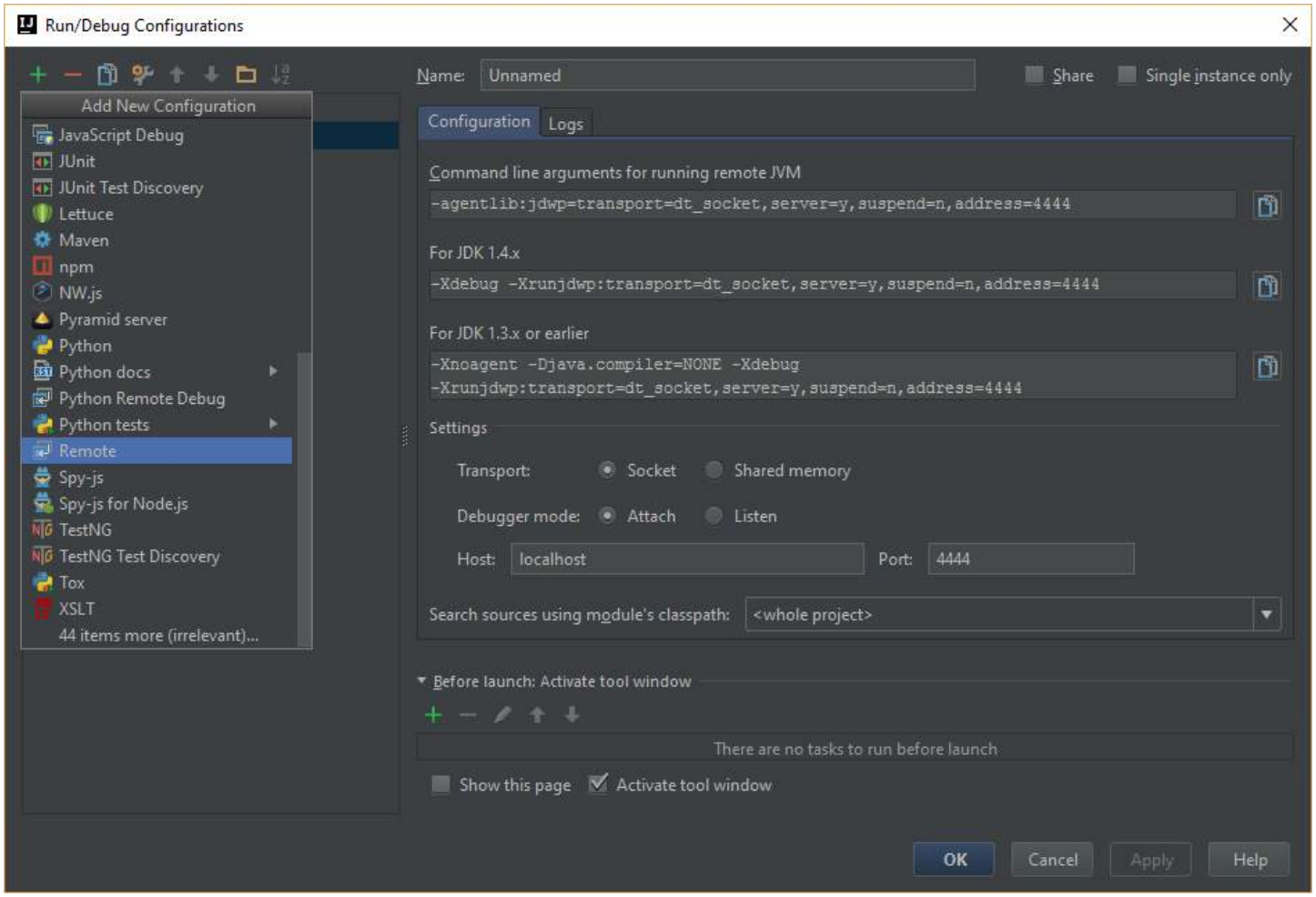
Alternatively the `java.opts` file can be created/updated. This file is stored in "matlab-root\bin\arch", where "matlab-root" is the MATLAB installation directory and "arch" is the architecture (e.g. "win32").

The following should be added in the file:

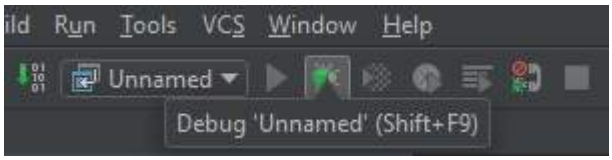
```
-Xdebug
-Xrunjdwp:transport=dt_socket,address=1044,server=y,suspend=n
```

Debugger end
IntelliJ IDEA

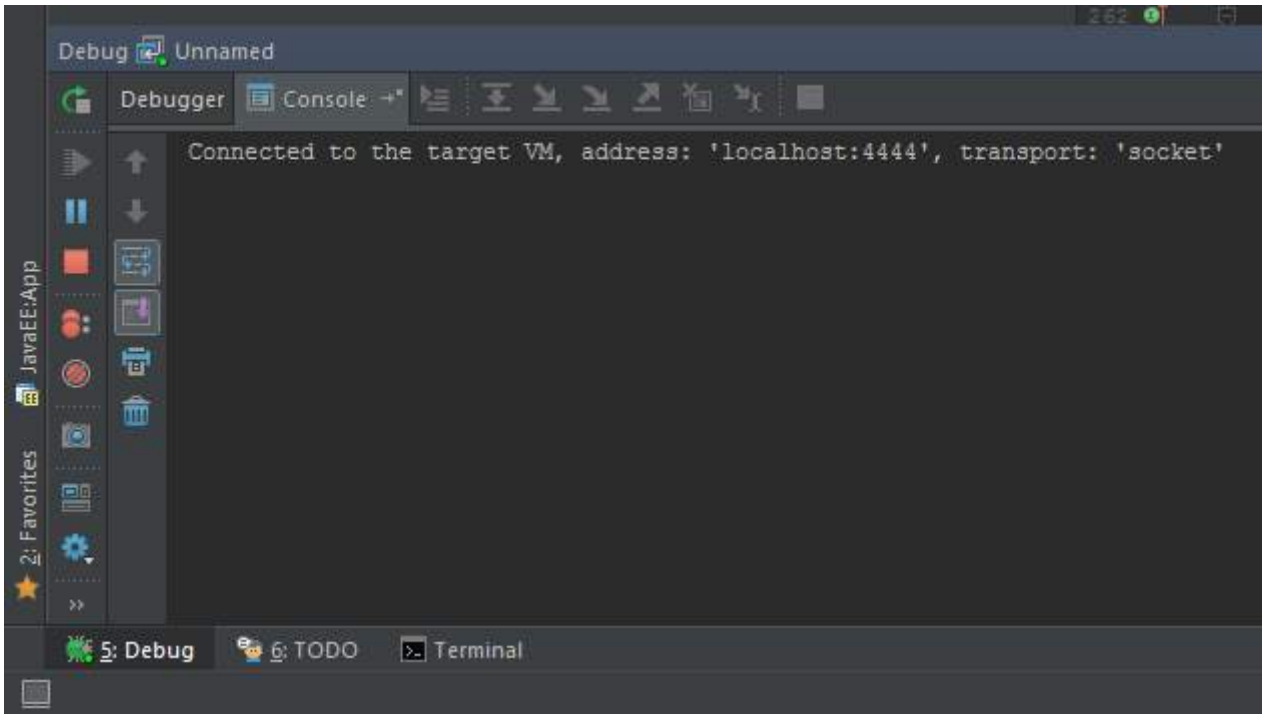
Attaching this debugger requires the creation of a "remote debugging" configuration with the port exposed by MATLAB:



Then the debugger is started:



If everything is working as expected, the following message will appear in the console:



Chapter 26: Performance and Benchmarking

Section 26.1: Identifying performance bottlenecks using the Profiler

The MATLAB Profiler is a tool for software profiling of MATLAB code. Using the Profiler, it is possible to obtain a visual representation of both execution time and memory consumption.

Running the Profiler can be done in two ways:

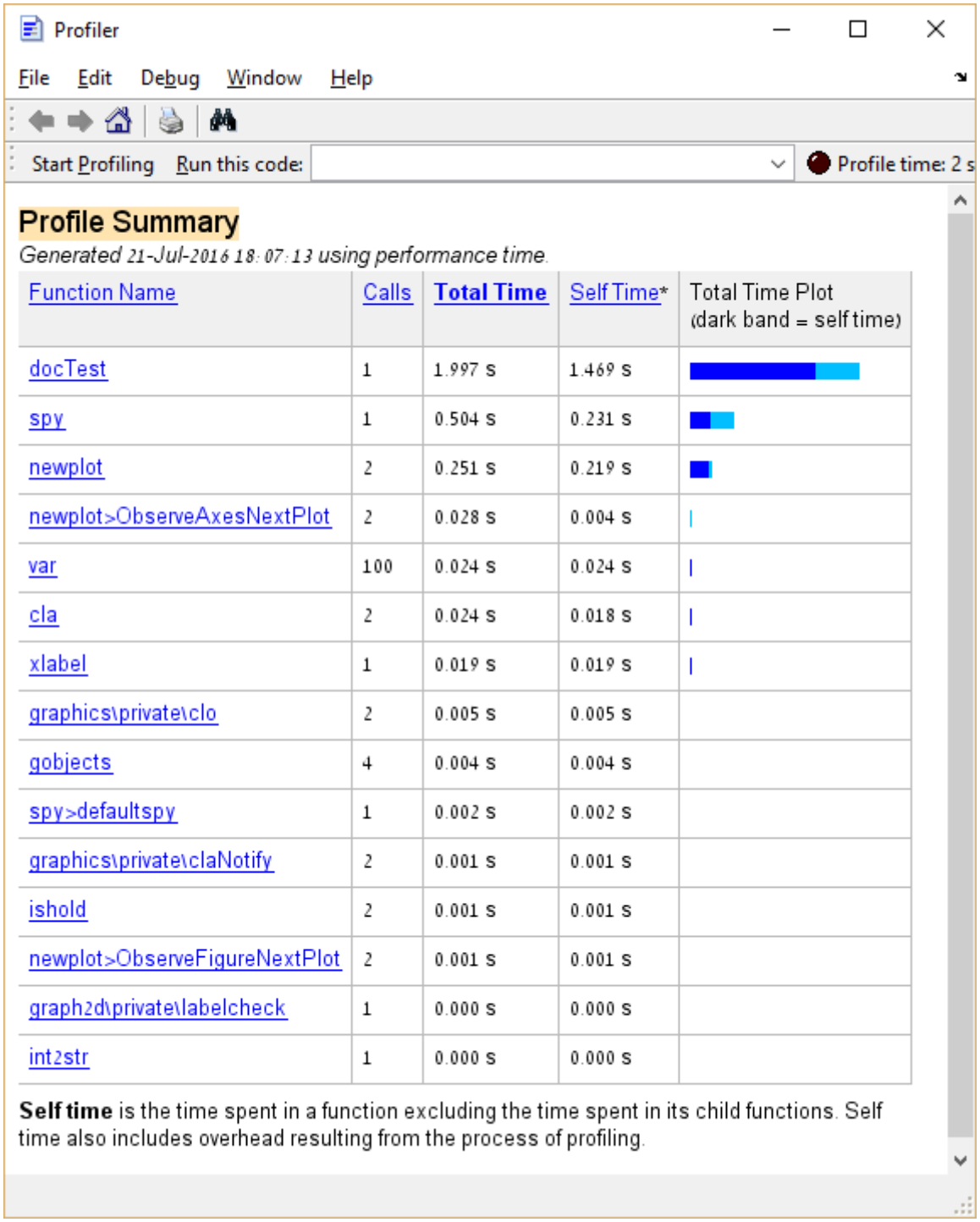
- Clicking the "Run and Time" button in the MATLAB GUI while having some `.m` file open in the editor (added in **R2012b**).

- Programmatically, using:

```
profile on
<some code we want to test>
profile off
```

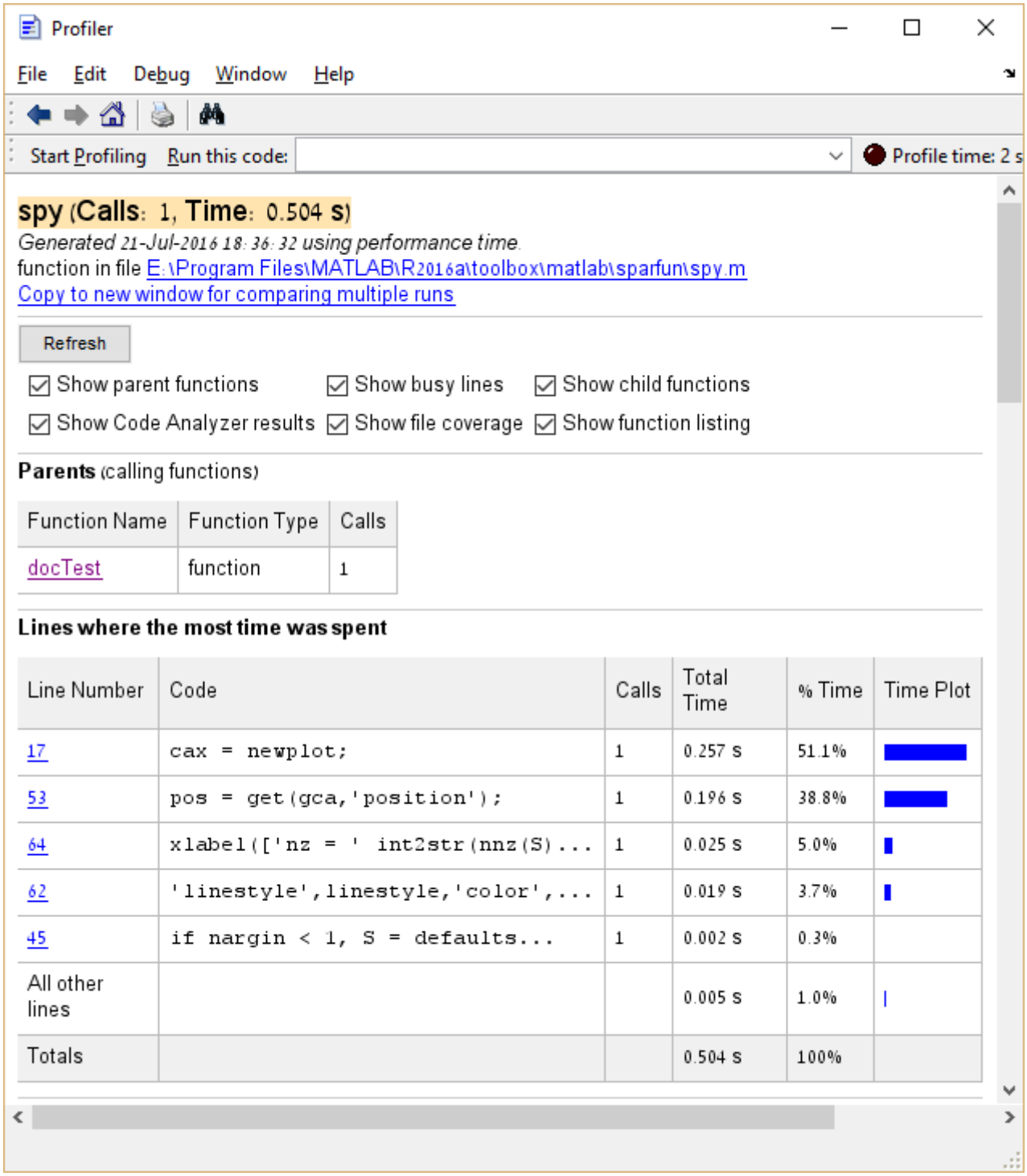
Below is some sample code and the result of its profiling:

```
function docTest

for ind1 = 1:100
  [~] = var(...
            sum(...
                randn(1000)));
end

spy
```

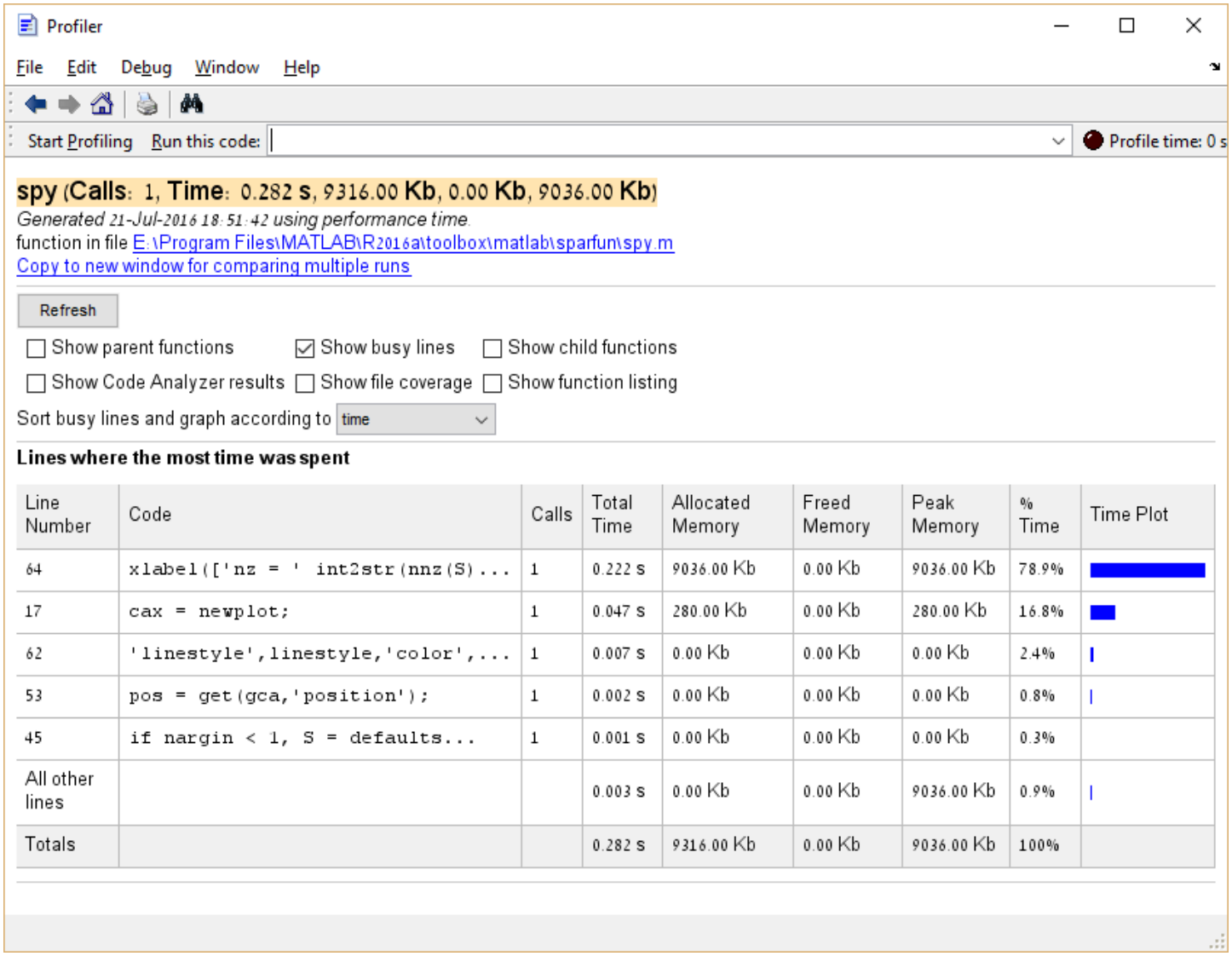


From the above we learn that the `spy` function takes about 25% of the total execution time. In the case of "real code", a function that takes such a large percentage of execution time would be a good candidate for optimization, as opposed to functions analogous to `var` and `cla` whose optimization should be avoided.

Moreover, it is possible to click on entries in the *Function Name* column to see a detailed breakdown of execution time for that entry. Here's the example of clicking `spy`:



It is also possible to profile memory consumption by executing `profile('-memory')` before running the Profiler.



Section 26.2: Comparing execution time of multiple functions

The widely used combination of **tic** and **toc** can provide a rough idea of the execution time of a function or code snippets.

For comparing several functions it shouldn't be used. Why? It is almost impossible to provide *equal conditions* for all code snippets to compare within a script using above solution. Maybe the functions share the same function space and common variables, so later called functions and code snippets already take advantage of previously initialized variables and functions. Also the there is no insight whether the JIT compiler would handle these subsequently called snippets equally.

The dedicated function for benchmarks is **timeit**. The following example illustrates its use.

There are the array `A` and the matrix `B`. It should be determined which row of `B` is the most similar to `A` by counting the number of different elements.

```
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
```

This way of benchmark was first seen in this answer.

Section 26.3: The importance of preallocation

Arrays in MATLAB are held as continuous blocks in memory, allocated and released automatically by MATLAB. MATLAB hides memory management operations such as resizing of an array behind easy to use syntax:

```
a = 1:4

a =

     1     2     3     4

a(5) = 10  % or alternatively a = [a, 10]

a =

     1     2     3     4    10
```

It is important to understand that the above is not a trivial operation, `a(5) = 10` will cause MATLAB to allocate a new block of memory of size 5, copy the first 4 numbers over, and set the 5'th to 10. That's a `O(numel(a))` operation, and not `O(1)`.

Consider the following:

```
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` is reallocated `n` times in this loop (excluding some optimizations undertaken by MATLAB)! Note that MATLAB gives us a warning:

> "The variable 'a' appears to change size on every loop iteration. Consider preallocating for speed."

What happens when we preallocate?

```
a=zeros(1,n);
tic
for i = 2:n
    a(i) = sqrt(a(i-1)) + i;
end
toc

Elapsed time is 0.410531 seconds.
```

We can see the runtime is reduced by an order of magnitude.

Methods for preallocation:

MATLAB provides various functions for allocation of vectors and matrices, depending on the specific requirements of the user. These include: `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
```

A data type can also be specified:

```
a = zeros(2, 1, 'uint8');  % allocates an array of type uint8
```

It is also easy to clone the size of an existing array:

```
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
```

And clone the type:

```
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
```

note that 'like' also clones *complexity* and *sparsity*.

Preallocation is implicitly achieved using any function that returns an array of the final required size, such as `rand`, `gallery`, `kron`, `bsxfun`, `colon` and many others. For example, a common way to allocate vectors with linearly varying elements is by using the colon operator (with either the 2- or 3-operand variant1):

```
a = 1:3
a =

     1     2     3

a = 2:-3:-4
a =

     2    -1    -4
```

Cell arrays can be allocated using the `cell()` function in much the same way as `zeros()`.

```
a = cell(2,3)
a =

    []    []    []
    []    []    []
```

Note that cell arrays work by holding pointers to the locations in memory of cell contents. So all preallocation tips apply to the individual cell array elements as well.

Further reading:

- Official MATLAB documentation on "**Preallocating Memory**".
- Official MATLAB documentation on "**How MATLAB Allocates Memory**".
- Preallocation performance on **Undocumented matlab**.
- Understanding Array Preallocation on **Loren on the Art of MATLAB**

Section 26.4: It's ok to be `single`!

Overview:

The default data type for numeric arrays in MATLAB is `double`. `double` is a floating point representation of numbers, and this format takes 8 bytes (or 64 bits) per value. In **some** cases, where e.g. dealing only with integers or when numerical instability is not an imminent issue, such high bit depth may not be required. For this reason, it is advised to consider the benefits of `single` precision (or other appropriate types):

- Faster execution time (especially noticeable on GPUs).
- Half the memory consumption: may succeed where `double` fails due to an out-of-memory error; more compact when storing as files.

Converting a variable from any supported data type to `single` is done using:

```
sing_var = single(var);
```

Some commonly used functions (such as: `zeros`, `eye`, `ones`, `etc.`) that output `double` values by default, allow specifying the type/class of the output.

Converting variables in a script to a non-default precision/type/class:

As of July 2016, there exists no documented way to change the *default* MATLAB data type from `double`.

In MATLAB, new variables usually mimic the data types of variables used when creating them. To illustrate this, consider the following example:

```
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
```

Thus, it may seem sufficient to cast/convert several initial variables to have the change permeate throughout the code - however this is **discouraged** (see *Caveats & Pitfalls* below).

Caveats & Pitfalls:

1. Repeated conversions are **discouraged** due to the introduction of numeric noise (when casting from `single` to `double`) or loss of information (when casting from `double` to `single`, or between certain integer types), e.g. :

   ```
   double(single(1.2)) == double(1.2)
   ans =
        0
   ```

   This can be mitigated somewhat using `typecast`. See also Be aware of floating point inaccuracy.

2. Relying solely on implicit data-typing (i.e. what MATLAB guesses the type of the output of a computation should be) is **discouraged** due to several undesired effects that might arise:

   - *Loss of information*: when a `double` result is expected, but a careless combination of `single` and `double` operands yields `single` precision.
   - *Unexpectedly high memory consumption*: when a `single` result is expected but a careless computation results in a `double` output.
   - *Unnecessary overhead when working with GPUs*: when mixing `gpuArray` types (i.e. variables stored in VRAM) with non-`gpuArray` variables (i.e. those *usually* stored in RAM) the data will have to be transferred one way or the other before the computation can be performed. This operation takes time, and can be very noticeable in repetitive computations.

- *Errors when mixing floating-point types with integer types*: functions like `mtimes` (`*`) are not defined for mixed inputs of integer and floating point types - and will error. Functions like `times` (`.*`) are not defined at all for integer-type inputs - and will again 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.
```

For better code readability and reduced risk of unwanted types, a defensive approach is **advised**, where variables are *explicitly* cast to the desired type.

See Also:

- MATLAB Documentation: Floating-Point Numbers.
- MathWorks' Technical Article: Best Practices for Converting MATLAB Code to Fixed Point.

Chapter 27: Multithreading

Section 27.1: Using parfor to parallelize a loop

You can use `parfor` to execute the iterations of a loop in parallel:

Example:

```
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
```

Note: `parfor` cannot be nested directly. For `parfor` nesting use a function in first `parfor` and add second `parfor` in that function.

Example:

```
parfor i = 1:n
[op] = fun_name(ip);
end

function [op] = fun_name(ip)
parfor j = 1:length(ip)
% Some Computation
end
```

Section 27.2: Executing commands in parallel using a "Single Program, Multiple Data" (SPMD) statement

Unlike a parallel for-loop (`parfor`), which takes the iterations of a loop and distributes them among multiple threads, a single program, multiple data (`spmd`) statement takes a series of commands and distributes them to **all** the threads, so that each thread performs the command and stores the results. Consider this:

```
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
```

It is important to note that each thread may be accessed during the `spmd` block by its thread index (also called lab index, or `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
```

In both examples, `q` is a composite object, which may be initialized with the command `q = Composite()`. It is important to note that composite objects are only accessible while the pool is running.

Section 27.3: Using the batch command to do various computations in parallel

To use multi-threading in MATLAB one can use the `batch` command. Note that you must have the Parallel Computing toolbox installed.

For a time-consuming script, for example,

```
for ii=1:1e8
   A(ii)=sin(ii*2*pi/1e8);
end
```

to run it in batch mode one would use the following:

```
job=batch("da")
```

which is enables MATLAB to run in batch mode and makes it possible to use MATLAB in the meantime to do other things, such as add more batch processes.

To retrieve the results after finishing the job and load the array `A` into the workspace:

```
load(job, 'A')
```

Finally, open the "monitor job gui" from *Home → Environment → Parallel → Monitor jobs* and delete the job through:

```
delete(job)
```

To load a function for batch processing, simply use this statement where `fcn` is the function name, `N` is number of output arrays and `x1`, `...`, `xn` are input arrays:

```
 j=batch(fcn, N, {x1, x2, ..., xn})
```

Section 27.4: When to use parfor

Basically, `parfor` is recommended in two cases: lots of iterations in your loop (i.e., like `1e10`), or if each iteration takes a very long time (e.g., `eig(magic(1e4))`). In the second case you might want to consider using `spmd` . The reason `parfor` is slower than a `for` loop for short ranges or fast iterations is the overhead needed to manage all workers correctly, as opposed to just doing the calculation.

Also a lot of functions have implicit multi-threading built-in, making a `parfor` loop not more efficient, when using these functions, than a serial `for` loop, since all cores are already being used. `parfor` will actually be a detriment in this case, since it has the allocation overhead, whilst being as parallel as the function you are trying to use.

Consider the following example to see the behaviour of `for` as opposed to that of `parfor`. First open the parallel pool if you've not already done so:

```
gcp; % Opens a parallel pool using your current settings
```

Then execute a couple of large loops:

```
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[-]';
```

Then do the same with `parfor` instead of `for`. You will notice that the average time per iteration goes up. Do realise however that the `parfor` used all available workers, thus the total time (`sum(Time)`) has to be divided by the number of cores in your computer.

So, whilst the time to do each separate iteration goes up using `parfor` with respect to using `for`, the total time goes down considerably.

Chapter 28: Using serial ports

Serial port parameter	what it does
`BaudRate`	Sets the baudrate. The most common today is 57600, but 4800, 9600, and 115200 are frequently seen as well
`InputBufferSize`	The number of bytes kept in memory. MATLAB has a FIFO, which means that new bytes will be discarded. The default is 512 bytes, but it can easily be set to 20MB without issue. There are only a few edge cases where the user would want this to be small
`BytesAvailable`	The number of bytes waiting to be read
`ValuesSent`	The number of bytes sent since the port was opened
`ValuesReceived`	The number of bytes read since the port was opened
`BytesAvailableFcn`	Specify the callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read
`BytesAvailableFcnCount`	Specify the number of bytes that must be available in the input buffer to generate a `bytes-available` event
`BytesAvailableFcnMode`	Specify if the `bytes-available` event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read

Serial ports are a common interface for communicating with external sensors or embedded systems such as Arduinos. Modern serial communications are often implemented over USB connections using USB-serial adapters. MATLAB provides built-in functions for serial communications, including RS-232 and RS-485 protocols. These functions can be used for hardware serial ports or "virtual" USB-serial connections. The examples here illustrate serial communications in MATLAB.

Section 28.1: Creating a serial port on 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);
```

Section 28.2: Choosing your communication mode

MATLAB supports *synchronous* and *asynchronous* communication with a serial port. It is important to choose the right communication mode. The choice will depend on:

- how the instrument you are communicating with behave.
- what other functions your main program (or GUI) will have to do aside from managing the serial port.

I'll define 3 different cases to illustrate, from the simplest to the most demanding. For the 3 examples, the

instrument I am connecting to is a circuit board with an inclinometer, which can work in the 3 modes I will be describing below.

Mode 1: Synchronous (Master/Slave)

This mode is the simplest one. It correspond to the case where the PC is the *Master* and the instrument is the *slave*. The instrument does not send anything to the serial port on its own, it only **replies** an answer after being asked a question/command by the Master (the PC, your program). For example:

- The PC sends a command: "Give me a measurement now"
- The instrument receive the command, take the measurement then send back the measurement value to the serial line: "The inclinometer value is XXX".

OR

- The PC sends a command: "Change from mode X to mode Y"
- The instrument receive the command, execute it, then send a confirmation message back to the serial line: "*Command executed*" (or "*Command NOT executed*"). This is commonly called an ACK/NACK reply (for "Acknowledge(d)" / "NOT Acknowledged").

Summary: in this mode, the instrument (the *Slave*) only send data to the serial line **immediately after** having been asked by the PC (the *Master*)

Mode 2: Asynchronous

Now suppose I started my instrument, but it is more than just a dumb sensor. It constantly monitor's its own inclination and as long as it is vertical (within a tolerance, let's say +/-15 degrees), it stays silent. If the device is tilted by more than 15 degrees and get close to horizontal, it sends an alarm message to the serial line, immediately followed by a reading of the inclination. As long as the inclination is above the threshold, it continues to send an inclination reading every 5s.

If your main program (or GUI) is constantly "waiting" for message arriving on the serial line, it can do that well ... but it cannot do anything else in the meantime. If the main program is a GUI, it is highly frustrating to have a GUI seemingly "frozen" because it won't accept any input from the user. Essentially, it became the *Slave* and the instrument is the *Master*. Unless you have a fancy way of controlling your GUI from the instrument, this is something to avoid. Fortunately, the *asynchronous* communication mode will let you:

- define a separate function which tells your program what to do when a message is received

- keep this function in a corner, it will only be called and executed **when a message arrives** on the serial line. The rest of the time the GUI can execute any other code it has to run.

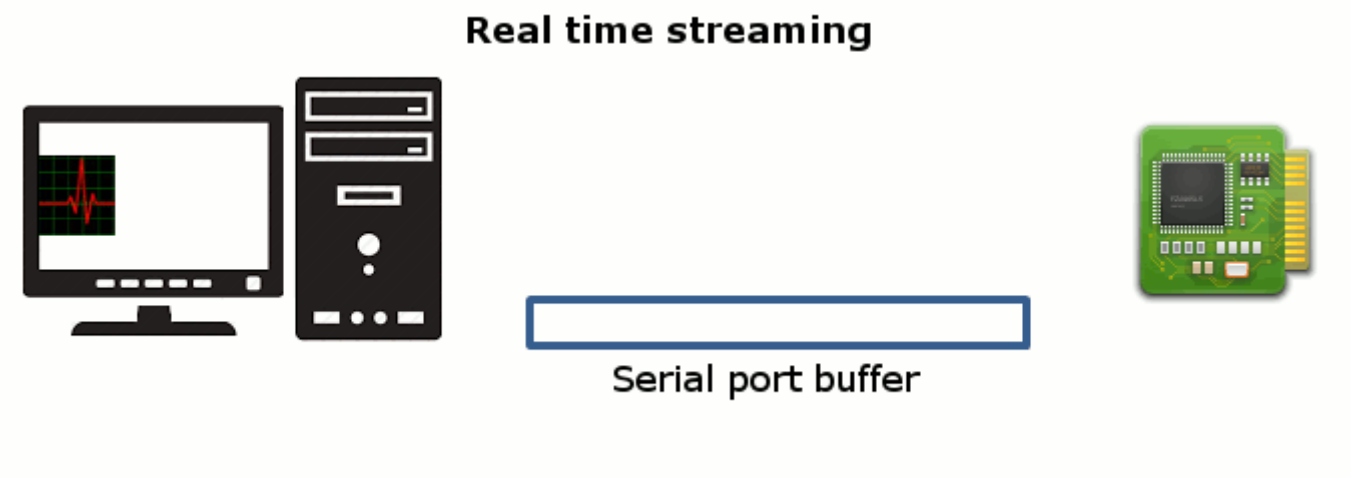
Summary: In this mode, the instrument may send message to the serial line at anytime (but not necessarily *all* the time). The PC does not *wait* permanently for a message to process. It is allowed to run any other code. Only when a message arrives, it activates a function which will then read and process this message.

Mode 3: Streaming (*Real time*)

Now let's unleash the full power of my instrument. I put it in a mode where it will constantly send measurements to the serial line. My program want to receive these packets and display that on a curve or a digital display. If it only send a value every 5s as above, no problem, keep the above mode. But my instrument at full whack sends a data point to the serial line at 1000Hz, i.e. it sends a new value every single millisecond. If I stay in the *asynchronous mode* described above, there is a high risk (actually a guaranteed certainty) that the special function we defined to process every new packet will take more than 1ms to execute (if you want to plot or display the value, graphic functions are quite slow, not even considering filtering or FFT'ing the signal). It means the function will start to execute, but before it finishes, a new packet will arrive and trigger the function again. The second function is placed in a queue for execution, and will only starts when the first one is done ... but by this time a few new packets arrived and each placed a function to execute in the queue. You can quickly foresee the result: By the time I am plotting the 5th points, I have already hundreds waiting to be plotted too ... the gui slows down, eventually freezes, the stack grows, the buffers fill up, until something gives. Eventually you are left with a completely frozen program or simply a crashed one.

To overcome this, we will disconnect even further the synchronisation link between the PC and the instrument. We will let the instrument send data at its own pace, without immediately triggering a function at each packet arrival. The serial port buffer will just accumulate the packets received. The PC will only collect data in the buffer at a pace it can manage (a regular interval, set up on the PC side), do something with it (while the buffer is getting refilled by the instrument), then collect a new batch of data from the buffer ... and so on.

Summary: In this mode, the instrument sends data continuously, which are collected by the serial port buffer. At regular interval, the PC collect data from the buffer and do something with it. There is no hard synchronisation link between the PC and the instrument. Both execute their tasks on their own timing.



Section 28.3: Automatically processing data received from a serial port

Some devices connected through a serial port send data to your program at a constant rate (streaming data) or send data at unpredictable intervals. You can configure the serial port to execute a function automatically to handle data whenever it arrives. This is called the "callback function" for the serial port object.

There are two properties of the serial port that must be set to use this feature: the name of the function you want for the callback (`BytesAvailableFcn`), and the condition which should trigger executing the callback function (`BytesAvailableFcnMode`).

There are two ways to trigger a callback function:

1. When a certain number of bytes have been received at the serial port (typically used for binary data)
2. When a certain character is received at the serial port (typically used for text or ASCII data)

Callback functions have two required input arguments, called `obj` and `event`. `obj` is the serial port. For example, if you want to print the data received from the serial port, define a function for printing the data called `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
```

For example, to execute the `newdata` function whenever 64 bytes of data are received, configure the serial port like this:

```
s = serial(port_name);
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcnCount = 64;
s.BytesAvailableFcn = @newdata;
```

With text or ASCII data, the data is typically divided into lines with a "terminator character", just like text on a page. To execute the `newdata` function whenever the carriage return character is received, configure the serial port like this:

```
s = serial(port_name);
```

```
s.BytesAvailableFcnMode = 'terminator';
s.Terminator = 'CR';  % the carriage return, ASCII code 13
s.BytesAvailableFcn = @newdata;
```

Section 28.4: Reading from the serial port

Assuming you created the serial port object `s` as in this example, then

```
% 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);
```

Section 28.5: Closing a serial port even if lost, deleted or overwritten

Assuming you created the serial port object `s` as in this example, then to close it

```
fclose(s)
```

However, sometimes you can accidentally lose the port (e.g. clear, overwrite, change scope, etc...), and `fclose(s)` will no longer work. The solution is easy

```
fclose(instrfindall)
```

More info at `instrfindall()`.

Section 28.6: Writing to the serial port

Assuming you created the serial port object `s` as in this example, then

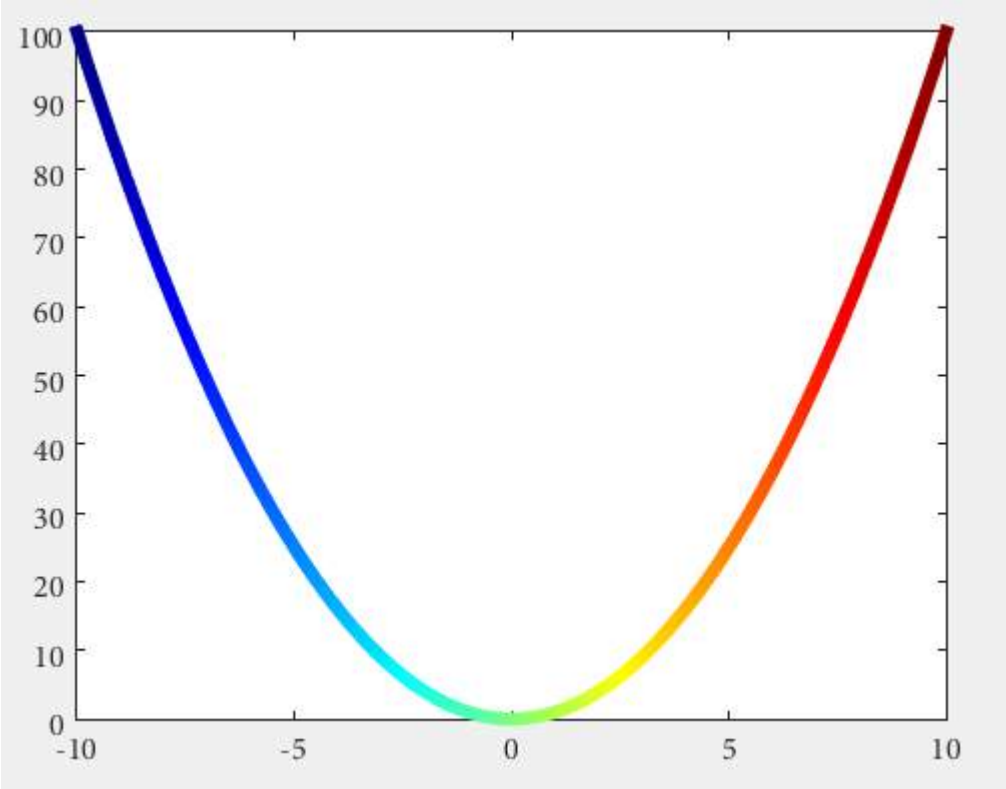
```
% Write one byte
fwrite(s, 255);

% 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);
```

Chapter 29: Undocumented Features

Section 29.1: Color-coded 2D line plots with color data in third dimension

In MATLAB versions prior to **R2014b**, using the old HG1 graphics engine, it was not obvious how to create color coded 2D line plots. With the release of the new HG2 graphics engine arose a new **undocumented feature introduced by 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)
```



Section 29.2: Semi-transparent markers in line and scatter plots

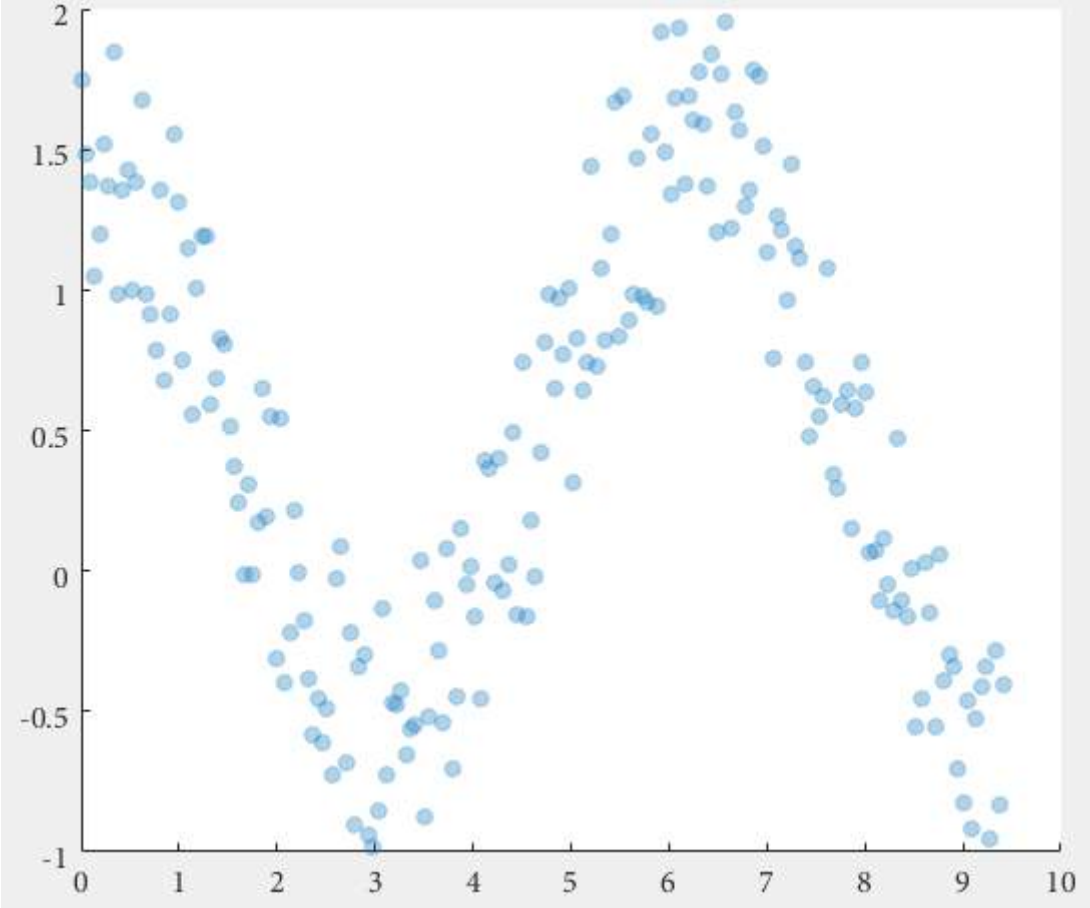
Since **MATLAB R2014b** it is easily possible to achieve semi-transparent markers for line and scatter plots using **undocumented features introduced by Yair Altman**.

The basic idea is to get the hidden handle of the markers and apply a value **< 1** for the last value in the `EdgeColorData` to achieve the desired transparency.

Here we go for `scatter`:

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

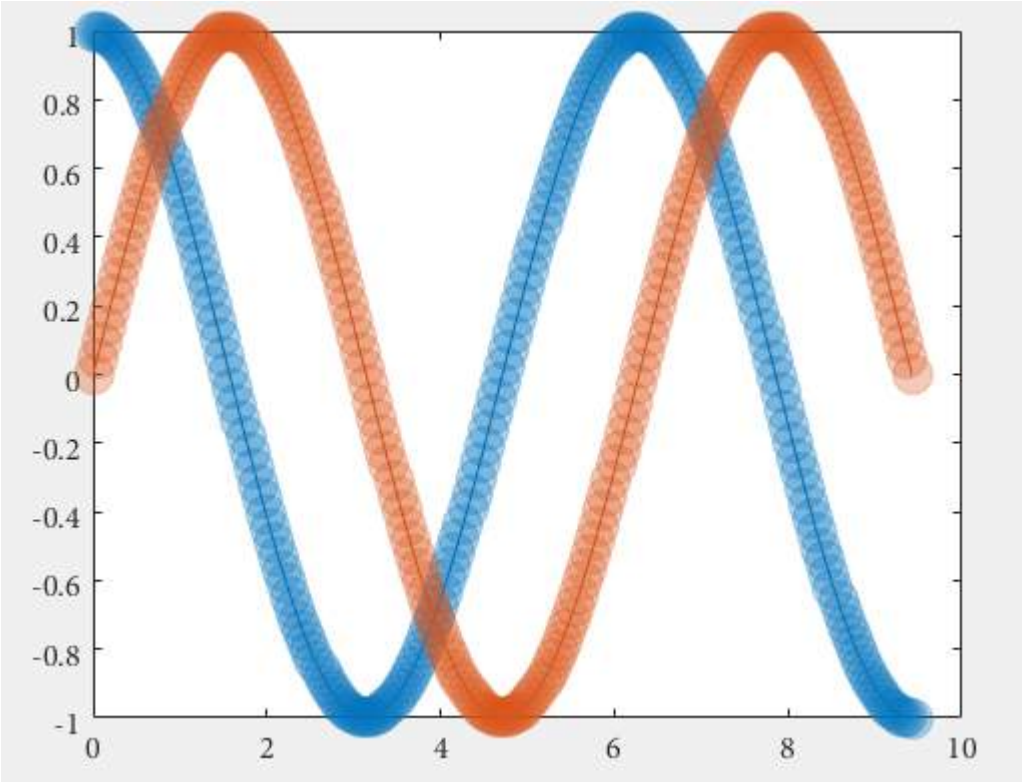
```
y = cos(x) + rand(1,200);

%// 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] );
```



and for a line `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] );
```



The marker handles, which are used for the manipulation, are created with the figure. The `drawnow` command is ensuring the creation of the figure before subsequent commands are called and avoids errors in case of delays.

Section 29.3: C++ compatible helper functions

The use of **MATLAB Coder** sometimes denies the use of some very common functions, if they are not compatible to C++. Relatively often there exist **undocumented helper functions**, which can be used as replacements.

Here is a comprehensive list of supported functions..

And following a collection of alternatives, for non-supported functions:

The functions `sprintf` and `sprintfc` are quite similar, the former returns a *character array*, the latter a *cell string*:

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

However, `sprintfc` is compatible with C++ supported by MATLAB Coder, and `sprintf` is not.

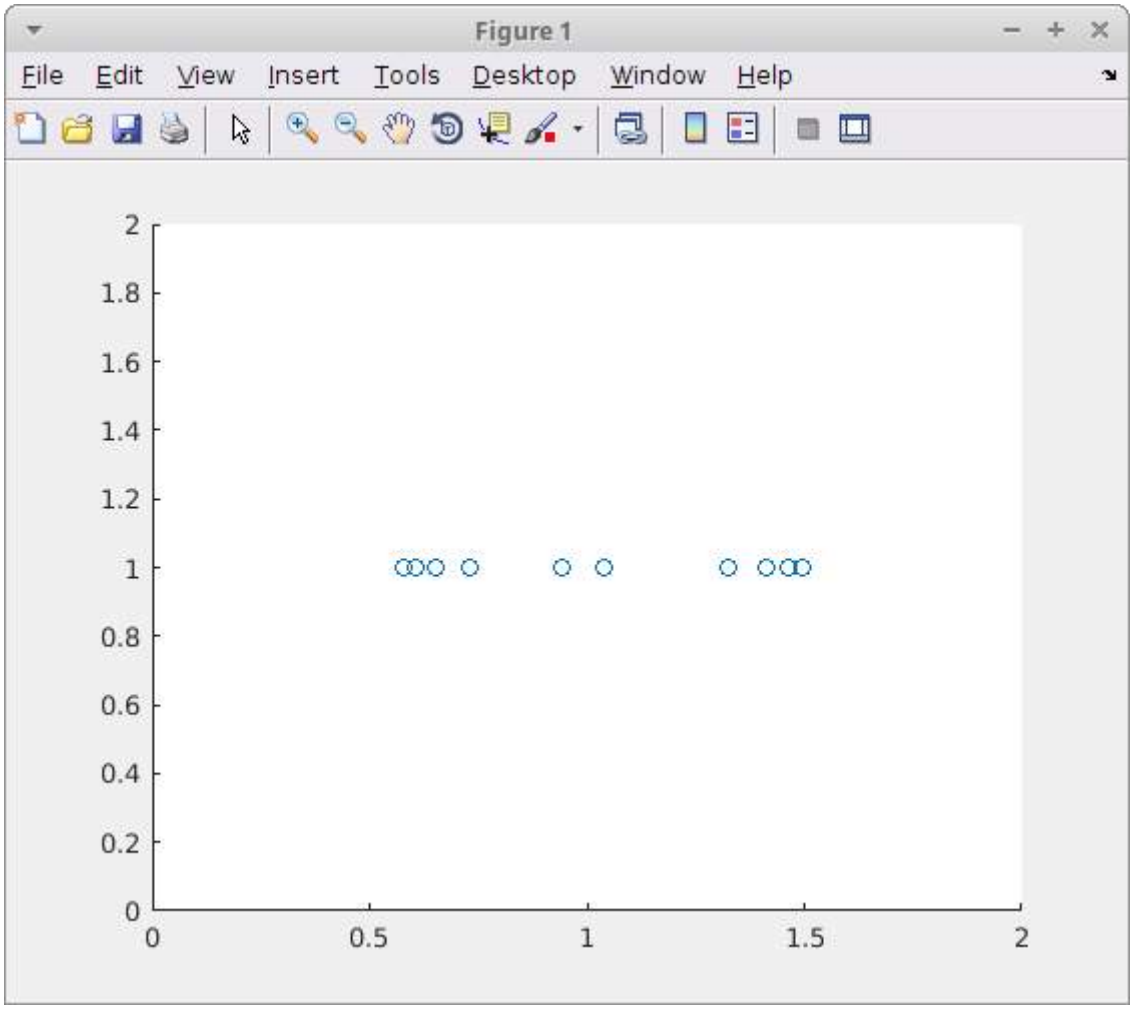
Section 29.4: Scatter plot jitter

The `scatter` function has two undocumented properties `'jitter'` and `'jitterAmount'` that allow to jitter the data on the x-axis only. This dates back to MATLAB 7.1 (2005), and possibly earlier.

To enable this feature set the `'jitter'` property to `'on'` and set the `'jitterAmount'` property to the desired absolute value (the default is `0.2`).

This is very useful when we want to visualize overlapping data, for example:

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



Read more on Undocumented Matlab

Section 29.5: Contour Plots - Customise the Text Labels

When displaying labels on contours MATLAB doesn't allow you to control the format of the numbers, for example to change to scientific notation.

The individual text objects are normal text objects but how you get them is undocumented. You access them from

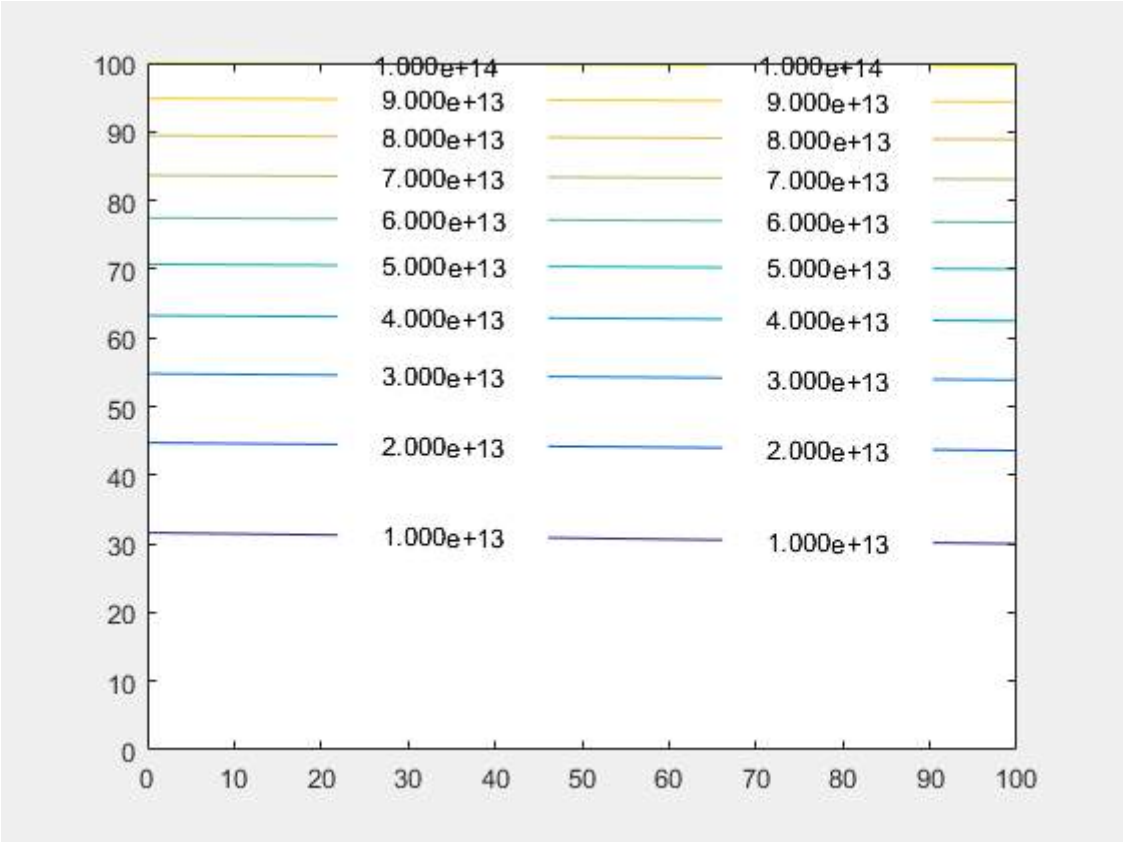
the `TextPrims` property of the contour handle.

```
  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
```

Note: that you must add a `drawnow` command to force MATLAB to draw the contours, the number and location of the txt objects are only determined when the contours are actually drawn so the text objects are only created then.

The fact the txt objects are created when the contours are drawn means that they are recalculated every time the plot is redrawn (for example figure resize). To manage this you need to listen to the `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
```



Example tested using MATLAB r2015b on Windows

Section 29.6: Appending / adding entries to an existing legend

Existing legends can be difficult to manage. For example, if your plot has two lines, but only one of them has a legend entry and that should stay this way, then adding a third line with a legend entry can be difficult. Example:

```
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
```

Now, to add a legend entry for `tan`, but not for `cos`, any of the following lines won't do the trick; they all fail in some way:

```
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)
```

Luckily, an undocumented legend property called `PlotChildren` keeps track of the children of the parent figure1. So, the way to go is to explicitly set the legend's children through its `PlotChildren` property as follows:

```
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
```

The legend updates automatically if an object is added or removed from its `PlotChildren` property.

1 Indeed: figure. You can add any figure's child with the `DisplayName` property to any legend in the figure, e.g. from a different subplot. This is because a legend in itself is basically an axes object.

Tested on MATLAB R2016b

Chapter 30: MATLAB Best Practices

Section 30.1: Indent code properly

Proper indentation gives not only the aesthetic look but also increases the readability of the code.

For example, consider the following code:

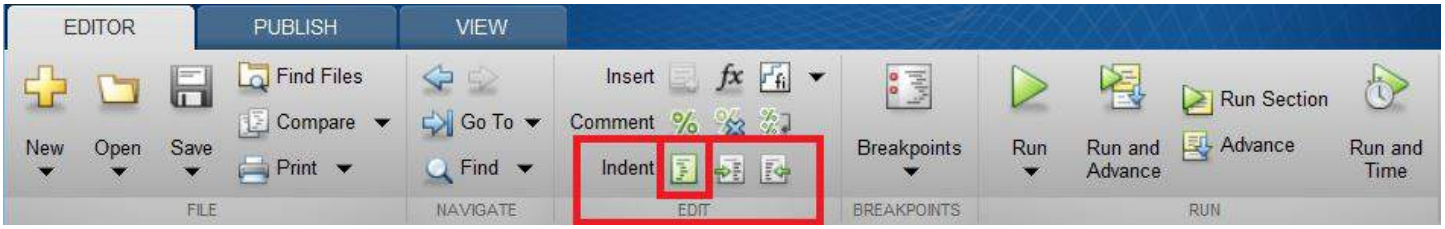
```
%no need to understand the code, just give it a look
n = 2;
bf = false;
while n>1
for ii = 1:n
for jj = 1:n
if ii+jj>30
bf = true;
break
end
end
if bf
break
end
end
if bf
break
end
n = n + 1;
end
```

As you can see, you need to give a careful look to see which loop and `if` statements are ending where.
With smart indentation, you'll get this 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
```

This clearly indicates the starting and ending of loops/`if` statement.

You can do smart indentation by:
• selecting all your code (Ctrl + A)
• and then pressing Ctrl + I or clicking  from edit bar.



Section 30.2: Avoid loops

Most of the time, loops are computationally expensive with MATLAB. Your code will be orders of magnitudes faster if you use vectorization. It also often makes your code more modular, easily modifiable, and easier to debug. The major downside is that you have to take time to plan the data structures, and dimension errors are easier to come by.

Examples

Don't write

```
for t=0:0.1:2*pi
    R(end+1)=cos(t);
end
```

but

```
t=0:0.1:2*pi;
R=cos(t)
```

Don't write

```
for i=1:n
    for j=1:m
        c(i,j)=a(i)+2*b(j);
    end
end
```

But something similar to

```
c=repmat(a.',1,m)+2*repmat(b,n,1)
```

For more details, see vectorization

Section 30.3: Keep lines short

Use the continuation character (ellipsis) `...` to continue long statement.

Example:

```
MyFunc( parameter1,parameter2,parameter3,parameter4, parameter5, parameter6,parameter7, parameter8,
parameter9)
```

can be replaced by:

```
MyFunc( parameter1, ...
        parameter2, ...
        parameter3, ...
```

```
        parameter4, ...
        parameter5, ...
        parameter6, ...
        parameter7, ...
        parameter8, ...
        parameter9)
```

Section 30.4: Use assert

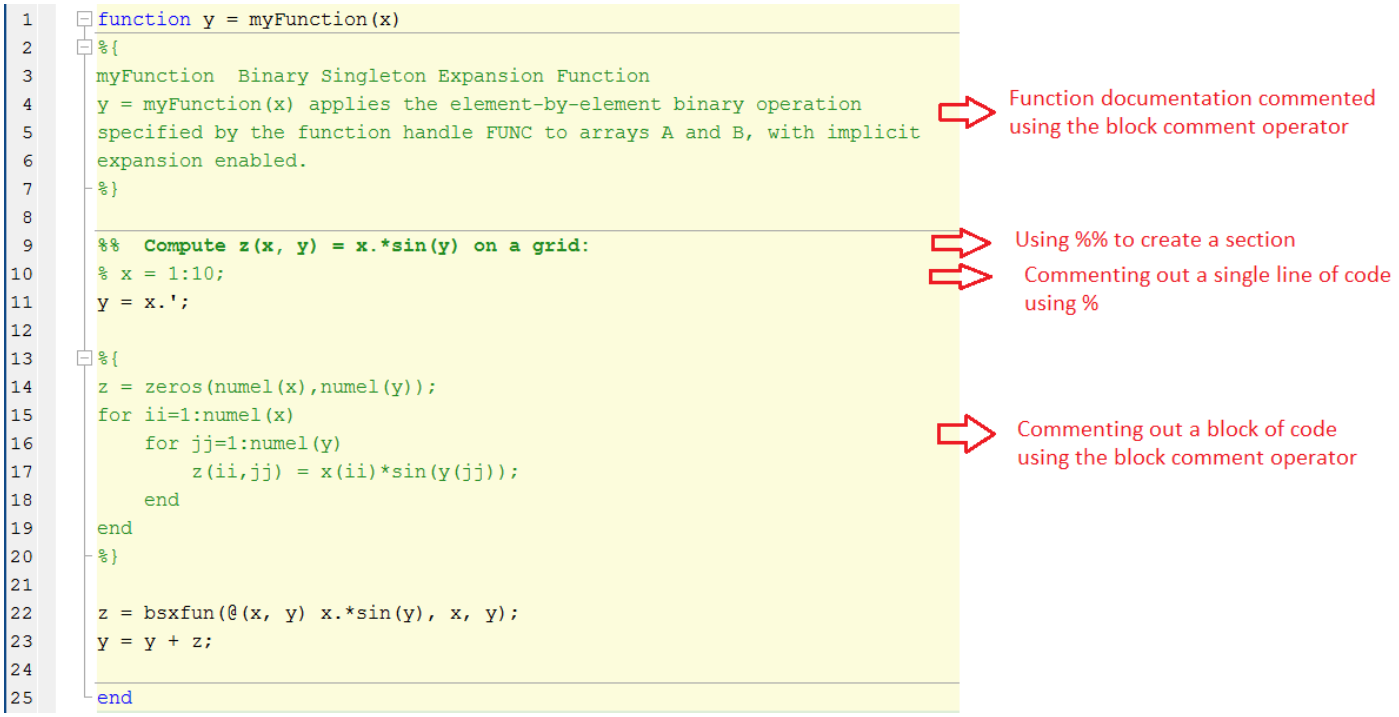
MATLAB allows some very trivial mistakes to go by silently, which might cause an error to be raised much later in the run - making debugging hard. If you **assume** something about your variables, **validate** it.

```
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};
```

Section 30.5: Block Comment Operator

It is a good practice to add comments that describe the code. It is helpful for others and even for the coder when returned later. A single line can be commented using the `%` symbol or using the shortkey `Ctrl` + `R`. To uncomment a previously commented line remove the `%` symbol or use shortkey `Crtl+T`.

While commenting a block of code can be done by adding a `%` symbol at the beginning of each line, newer versions of MATLAB (after 2015a) let you use the **Block Comment Operator** `%{ code %}`. This operator increases the readability of the code. It can be used for both code commenting and function help documentation. The Block can be **folded** and **unfolded** to increase the readability of the code.



As it can be seen the `%{` and `%}` operators must appear alone on the lines. Do not include any other text on these lines.

```
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
```

Section 30.6: Create Unique Name for Temporary File

While coding a script or a function, it can be the case that one or more than one temporary file be needed in order to, for example, store some data.

In order to avoid overwriting an existing file or to shadow a MATLAB function the tempname function can be used to generate a **unique name** for a temporary file in the system temporary folder.

```
my_temp_file=tempname
```

The filename is generated without the extension; it can be added by concatenating the desired extension to the name generated by `tempname`

```
my_temp_file_with_ext=[tempname '.txt']
```

The location of the system temporary folder can be retrieved by calling the tempdir function.

If, during the execution of the function / script, the temporary file is no longer needed, it can be deleted by using the function delete

Since `delete` does not ask for confirmation, it might be useful to set `on` the option to move the file to be deleted in the `recycle` folder.

This can be done by using the function recycle this way:

```
recycle('on')
```

In the following example, a possible usage of the functions `tempname`, `delete` and `recycle` is proposed.

```
%
% 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 variables in a temporary file
%
save(my_temp_file,'x','y','theta')
%
% Load the variables from the temporary file
%
load(my_temp_file)
%
% Set the recycle option on
%
recycle('on')
%
% Delete the temporary file
%
delete(my_temp_file)
```

Caveat

The temporary filename is generated by using the `java.util.UUID.randomUUID` method (randomUUID).

If MATLAB is run without JVM, the temporary filename is generated by using `matlab.internal.timing.timing` based on the CPU counter and time. In this case the temporary filename is not guaranteed to be unique.

Chapter 31: MATLAB User Interfaces

Section 31.1: Passing Data Around User Interface

Most advanced user interfaces require the user to be able to pass information between the various functions which make up a user interface. MATLAB has a number of different methods to do so.

`guidata`

MATLAB's own GUI Development Environment (GUIDE) prefers to use a `struct` named `handles` to pass data between callbacks. This `struct` contains all of the graphics handles to the various UI components as well as user-specified data. If you aren't using a GUIDE-created callback which automatically passes `handles`, you can retrieve the current value using `guidata`

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

If you want to modify a value stored in this data structure, you can modify but then you must store it back within the `hObject` for the changes to be visible by other callbacks. You can store it by specifying a second input argument to `guidata`.

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

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

The value of `hObject` doesn't matter as long as it is a UI component *within the same* `figure` because ultimately the data is stored within the figure containing `hObject`.

Best for:

- Storing the `handles` structure, in which you can store all the handles of your GUI components.
- Storing "small" other variables which need to be accessed by most callbacks.

Not recommended for:

- Storing large variables which do not have to be accessed by all callbacks and sub-functions (use `setappdata`/`getappdata` for these).

`setappdata`/`getappdata`

Similar to the `guidata` approach, you can use `setappdata` and `getappdata` to store and retrieve values from within a graphics handle. The advantage of using these methods is that you can retrieve *only the value you want* rather than an entire `struct` containing *all* stored data. It is similar to a key/value store.

To store data within a graphics object

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

% Store it using the key 'mykey'
```

```
setappdata(hObject, 'mykey', myvalue)
```

And to retrieve that same value from within a different callback

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

Note: If no value was stored prior to calling `getappdata`, it will return an empty array ([ ]).

Similar to `guidata`, the data is stored in the figure that contains `hObject`.

Best for:

- Storing large variables which do not have to be accessed by all callbacks and sub-functions.

UserData

Every graphics handle has a special property, `UserData` which can contain any data you wish. It could contain a cell array, a `struct`, or even a scalar. You can take advantage of this property and store any data you wish to be associated with a given graphics handle in this field. You can save and retrieve the value using the standard `get`/`set` methods for graphics objects or dot notation if you're using R2014b or newer.

```
% 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;
```

Then from within another callback, you can retrieve this data:

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

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

Best for:

- Storing variables with a limited scope (variables which are likely to be used only by the object in which they are stored, or objects having a direct relationship to it).

Nested Functions

In MATLAB, a nested function can read and modify any variable defined in the parent function. In this way, if you specify a callback to be a nested function, it can retrieve and modify any data stored in the main function.

```
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
```

Best for:

- Small, simple GUIs. (for quick prototyping, to not have to implement the `guidata` and/or `set`/`getappdata` methods).

Not recommended for:

- Medium, large or complex GUIs.
- GUI created with `GUIDE`.

Explicit input arguments

If you need to send data to a callback function and don't need to modify the data within the callback, you can always consider passing the data to the callback using a carefully crafted callback definition.

You could use an anonymous function which adds inputs

```
% 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))
```

Or you could use the cell array syntax to specify a callback, again specifying additional inputs.

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

Best for:

- When the callback needs `data` to perform some operations but the `data` variable does not need to be modified and saved in a new state.

Section 31.2: Making a button in your UI that pauses callback execution

Sometimes we'd like to pause code execution to inspect the state of the application (see Debugging). When running code through the MATLAB editor, this can be done using the "Pause" button in the UI or by pressing Ctrl + c (on Windows). However, when a computation was initiated from a GUI (via the callback of some `uicontrol`), this method does not work anymore, and the callback should be *interrupted* via another callback. Below is a demonstration of this principle:

```
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
```

To make sure you understand this example do the following:

1. Paste the above code into a new file called and save it as `interruptibleUI.m`, such that the code starts on the **very first line** of the file (this is important for the 1st method to work).
2. Run the script.
3. Click `Compute` and shortly afterwards click either `Pause #1` or on `Pause #2`.
4. Make sure you can find the value of `superSecretVar`.

Section 31.3: Passing data around using the "handles" structure

This is an example of a basic GUI with two buttons that change a value stored in the GUI's `handles` structure.

```
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 '-'
            % Subtract 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);
```

To test the example, save it in a file called `gui_passing_data.m` and launch it with F5. Please note that in such a simple case, you would not even need to store the value in the handles structure because you could directly access it from the edit box's `String` property.

Section 31.4: Performance Issues when Passing Data Around User Interface

Two main techniques allow passing data between GUI functions and Callbacks: setappdata/getappdata and guidata (read more about it). The former should be used for larger variables as it is more time efficient. The following example tests the two methods' efficiency.

A GUI with a simple button is created and a large variable (10000x10000 double) is stored both with guidata and with setappdata. The button reloads and stores back the variable using the two methods while timing their execution. The running time and percentage improvement using setappdata are displayed in the command window.

```
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);
```

On my Xeon W3530@2.80 GHz I get `Difference: 0.00018957 ms / 73 %`, thus using getappdata/setappdata I get a performance improvement of 73%! Note that the result does not change if a 10x10 double variable is used, however, result will change if `handles` contains many fields with large data.

Chapter 32: Useful tricks

Section 32.1: Extract figure data

On a few occasions, I have had an interesting figure I saved but I lost an access to its data. This example shows a trick how to achieve extract information from a figure.

The key functions are findobj and get. findobj returns a handler to an object given attributes or properties of the object, such as `Type` or `Color`, etc. Once a line object has been found, get can return any value held by properties. It turns out that the `Line` objects hold all data in following properties: `XData`, `YData`, and `ZData`; the last one is usually 0 unless a figure contains a 3D plot.

The following code creates an example figure that shows two lines a sin function and a threshold and a legend

```
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'});
```

The first use of findobj returns two handlers to both lines:

```
findobj(gcf, 'Type', 'Line')
ans =
  2x1 Line array:

  Line    (threshold)
  Line    (sin)
```

To narrow the result, findobj can also use combination of logical operators `-and`, `-or` and property names. For instance, I can find a line object whose `DiplayName` is `sin` and read its `XData` and `YData`.

```
lineh = findobj(gcf, 'Type', 'Line', '-and', 'DisplayName', 'sin');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

and check if the data are equal.

```
isequal(t(:),xdata(:))
ans =
     1
isequal(y(:),ydata(:))
ans =
     1
```

Similarly, I can narrow my results by excluding the black line (threshold):

```
lineh = findobj(gcf, 'Type', 'Line', '-not', 'Color', 'k');
xdata = get(lineh, 'XData');
ydata = get(lineh, 'YData');
```

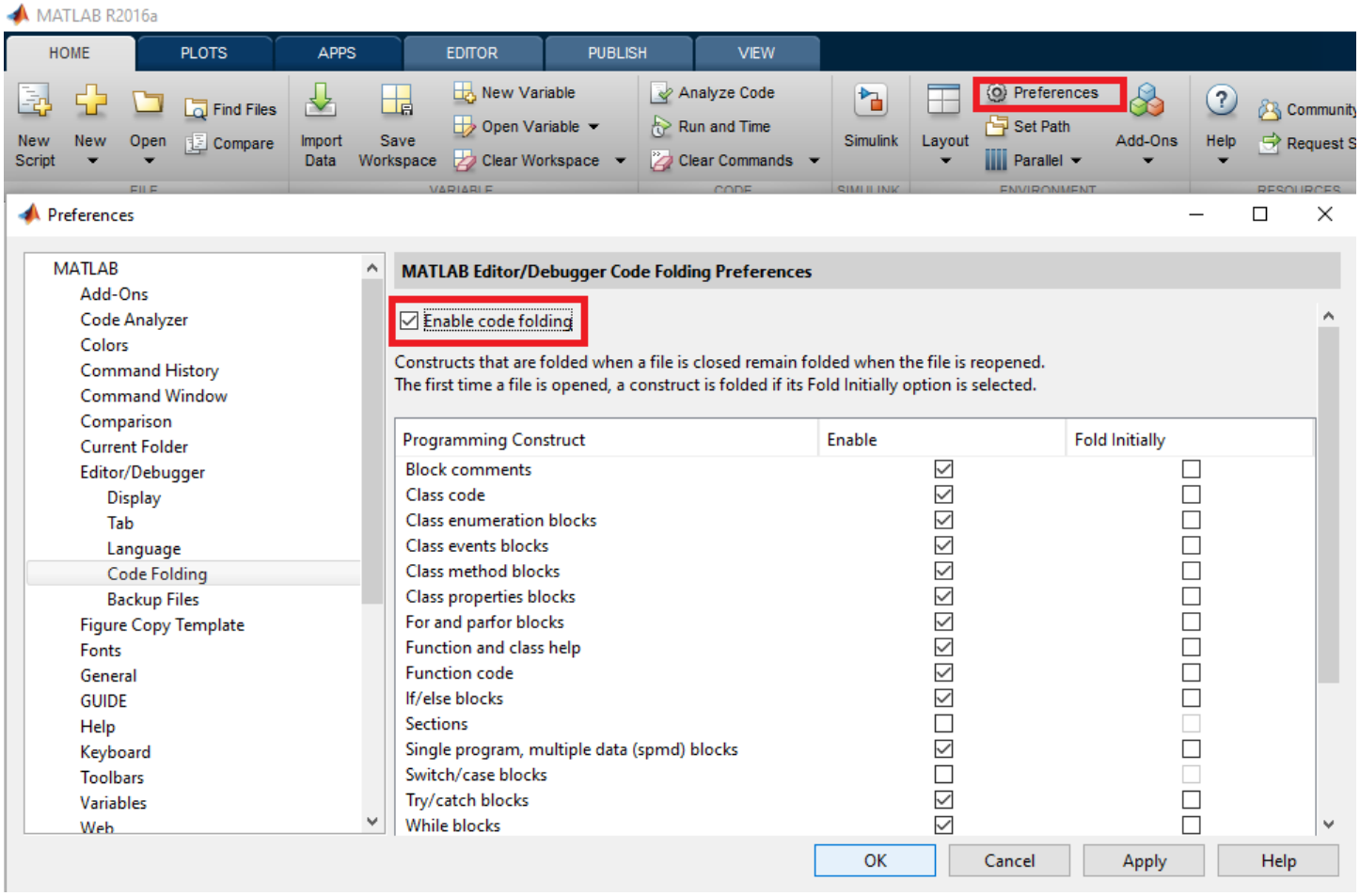
and last check confirms that data extracted from this figure are the same:

```
isequal(t(:),xdata(:))
ans =
     1
isequal(y(:),ydata(:))
ans =
     1
```

Section 32.2: Code Folding Preferences

It is possible to change Code Folding preference to suit your need. Thus code folding can be set enable/unable for specific constructs (ex: `if` block, `for` loop, `Sections` ...).

To change folding preferences, go to Preferences -> Code Folding:



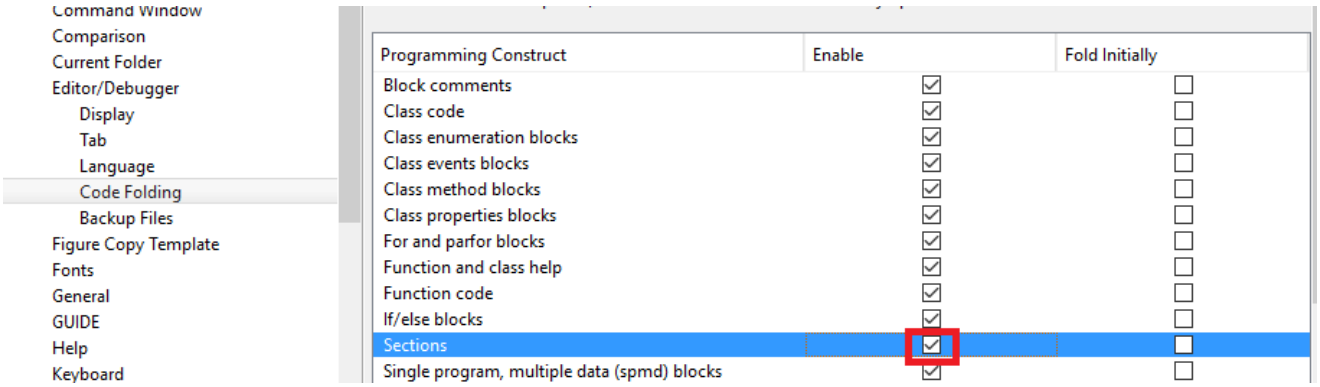
Then you can choose which part of the code can be folded.

Some information:

- Note that you can also expand or collapse all of the code in a file by placing your cursor anywhere within the file, right-click, and then select Code Folding > Expand All or Code Folding > Fold All from the context menu.
- Note that folding is persistent, in the sense that part of the code that has been expanded/collapsed will keep their status after MATLAB or the m-file has been closed and is re-open.

> **Example: To enable folding for sections:**
>
> An interesting option is to enable to fold Sections. Sections are delimited by two percent signs (`%%`).

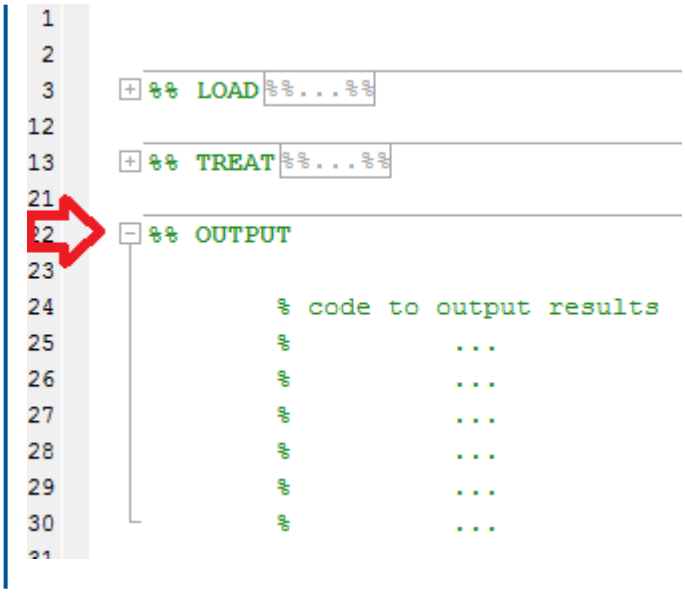
Example: To enable it check the "Sections" box:



Then instead of seeing a long source code similar to :

```
%% LOAD

        %code to load data
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...

%% TREAT
        % code to run the model
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...

%% OUTPUT

        % code to output results
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
        %           ...
```

You will be able to fold sections to have a general overview of your code :



Section 32.3: Functional Programming using Anonymous Functions

Anonymous functions can be used for functional programming. The main problem to solve is that there is no native way for anchoring a recursion, but this can still be implemented in a single line:

```
if_ = @(bool, tf) tf{2-bool}();
```

This function accepts a boolean value and a cell array of two functions. The first of those functions is evaluated if the boolean value evaluates as true, and the second one if the boolean value evaluates as false. We can easily write the factorial function now:

```
fac = @(n,f) if_(n>1, {@()n*f(n-1,f), @()1});
```

The problem here is that we cannot directly invoke a recursive call, as the function is not yet assigned to a variable when the right hand side is evaluated. We can however complete this step by writing

```
factorial_ = @(n)fac(n,fac);
```

Now `@(n)fac(n,fac)` evaluates the factorial function recursively. Another way to do this in functional programming using a y-combinator, which also can easily be implemented:

```
y_ = @(f)@(n)f(n,f);
```

With this tool, the factorial function is even shorter:

```
factorial_ = y_(fac);
```

Or directly:

```
factorial_ = y_(@(n,f) if_(n>1, {@()n*f(n-1,f), @()1}));
```

Section 32.4: Save multiple figures to the same .fig file

By putting multiple figure handles into a graphics array, multiple figures can be saved to the same .fig file

```
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);
```

This creates 3 scatterplots of random data, each part of graphic array h. Then the graphics array can be saved using savefig like with a normal figure, but with the handle to the graphics array as an additional argument.

An interesting side note is that the figures will tend to stay arranged in the same way that they were saved when you open them.

Section 32.5: Comment blocks

If you want to comment part of your code, then comment blocks may be useful. Comment block starts with a `%{` in a new line and ends with `%}` in another new line:

```
a = 10;
b = 3;
%{
c = a*b;
d = a-b;
%}
```

This allows you to fold the sections that you commented to make the code more clean and compact.

These blocks are also useful for toggling on/off parts of your code. All you have to do to uncomment the block is add another `%` before it starts:

```
a = 10;
b = 3;
%%{ <-- another % over here
c = a*b;
d = a-b;
%}
```

Sometimes you want to comment out a section of the code, but without affecting its indentation:

```
for k = 1:a
    b = b*k;
    c = c-b;
    d = d*c;
    disp(b)
end
```

Usually, when you mark a block of code and press Ctrl + r for commenting it out (by that adding the `%` automatically to all lines, then when you press later Ctrl + i for auto indentation, the block of code moves from its correct hierarchical place, and moved too much to the right:

```
for k = 1:a
    b = b*k;
```

```
    %     c = c-b;
    %     d = d*c;
    disp(b)
end
```

A way to solve this is to use comment blocks, so the inner part of the block stays correctly indented:

```
for k = 1:a
    b = b*k;
    %{
    c = c-b;
    d = d*c;
    %}
    disp(b)
end
```

Section 32.6: Useful functions that operate on cells and arrays

This simple example provides an explanation on some functions I found extremely useful since I have started using MATLAB: `cellfun`, `arrayfun`. The idea is to take an array or cell class variable, loop through all its elements and apply a dedicated function on each element. An applied function can either be anonymous, which is usually a case, or any regular function define in a *.m file.

Let's start with a simple problem and say we need to find a list of *.mat files given the folder. For this example, first let's create some *.mat files in a current folder:

```
for n=1:10; save(sprintf('mymatfile%d.mat',n)); end
```

After executing the code, there should be 10 new files with extension *.mat. If we run a command to list all *.mat files, such as:

```
mydir = dir('*.mat');
```

we should get an array of elements of a dir structure; MATLAB should give a similar output to this one:

```
10x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

As you can see each element of this array is a structure with couple of fields. All information are indeed important regarding each file but in 99% I am rather interested in file names and nothing else. To extract information from a structure array, I used to create a local function that would involve creating temporal variables of a correct size, for loops, extracting a name from each element, and save it to created variable. Much easier way to achieve exactly the same result is to use one of the aforementioned functions:

```
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'
```

How this function works? It usually takes two parameters: a function handle as the first parameter and an array. A function will then operate on each element of a given array. The third and fourth parameters are optional but important. If we know that an output will not be regular, it must be saved in cell. This must be point out setting `false` to `UniformOutput`. By default this function attempts to return a regular output such as a vector of numbers. For instance, let's extract information about how much of disc space is taken by each file in bytes:

```
mydirbytes = arrayfun(@(x) x.bytes, dir('*.mat'))
mydirbytes =
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
       34560
```

or 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
```

This time the output is a regular vector of double. `UniformOutput` was set to `true` by default.

`cellfun` is a similar function. The difference between this function and `arrayfun` is that `cellfun` operates on cell class variables. If we wish to extract only names given a list of file names in a cell 'mydirlist', we would just need to run this function as follows:

```
mydirnames = cellfun(@(x) x(1:end-4), mydirlist, 'UniformOutput', false)
mydirnames =
    'mymatfile1'
    'mymatfile10'
    'mymatfile2'
    'mymatfile3'
    'mymatfile4'
    'mymatfile5'
    'mymatfile6'
    'mymatfile7'
    'mymatfile8'
    'mymatfile9'
```

Again, as an output is not a regular vector of numbers, an output must be saved in a cell variable.

In the example below, I combine two functions in one and return only a list of file names without an extension:

```
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'
```

It is crazy but very possible because `arrayfun` returns a cell which is expected input of `cellfun`; a side note to this is that we can force any of those functions to return results in a cell variable by setting `UniformOutput` to false, explicitly. We can always get results in a cell. We may not be able to get results in a regular vector.

There is one more similar function that operates on fields a structure: `structfun`. I have not particularly found it as useful as the other two but it would shine in some situations. If for instance one would like to know which fields are numeric or non-numeric, the following code can give the answer:

```
structfun(@(x) ischar(x), mydir(1))
```

The first and the second field of a dir structure is of a char type. Therefore, the output is:

```
1
1
0
0
0
```

Also, the output is a logical vector of `true` / `false`. Consequently, it is regular and can be saved in a vector; no need to use a cell class.

Chapter 33: Common mistakes and errors

Section 33.1: The transpose operators

- `.'` is the correct way to **transpose** a vector or matrix in MATLAB.
- `'` is the correct way to take the **complex conjugate transpose** (a.k.a. Hermitian conjugate) of a vector or matrix in MATLAB.

Note that for the transpose `.'`, there is a **period** in front of the apostrophe. This is in keeping with the syntax for the other element-wise operations in MATLAB: `*` multiplies *matrices*, `.*` multiplies *elements of matrices* together. The two commands are very similar, but conceptually very distinct. Like other MATLAB commands, these operators are "syntactical sugar" that gets turned into a "proper" function call at runtime. Just as == becomes an evaluation of the eq function, think of `.'` as the shorthand for `transpose`. If you would only write `'` (without the point), you are in fact using the `ctranspose` command instead, which calculates the complex conjugate transpose, which is also known as the Hermitian conjugate, often used in physics. As long as the transposed vector or matrix is real-valued, the two operators produce the same result. But as soon as we deal with complex numbers, we will inevitably run into problems if we do not use the "correct" shorthand. What "correct" is depends on your application.

Consider the following example of a matrix `C` containing complex numbers:

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

Let's take the *transpose* using the shorthand `.'` (with the period). The output is as expected, the transposed form of `C`.

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

Now, let's use `'` (without the period). We see, that in addition to the transposition, the complex values have been transformed to their *complex conjugates* as well.

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

To sum up, if you intend to calculate the Hermitian conjugate, the complex conjugate transpose, then use `'` (without the period). If you just want to calculate the transpose without complex-conjugating the values, use `.'` (with the period).

Section 33.2: Do not name a variable with an existing function name

There is already a function `sum()`. As a result, if we name a variable with the same name

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

and if we try to use the function while the variable still exists in the workspace

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

we will get the cryptic **error**:

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

clear() the variable first and then use the function

```
clear sum

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

How can we check if a function already exists to avoid this conflict?

Use which() with the -all flag:

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

This output is telling us that sum is first a variable and that the following methods (functions) are shadowed by it, i.e. MATLAB will first try to apply our syntax to the variable, rather than using the method.

Section 33.3: Be aware of floating point inaccuracy

Floating-point numbers cannot represent all real numbers. This is known as floating point inaccuracy.

There are infinitely many floating points numbers and they can be infinitely long (e.g. π), thus being able to represent them perfectly would require infinitely amount of memory. Seeing this was a problem, a special representation for "real number" storage in computer was designed, the IEEE 754 standard. In short, it describes how computers store this type of numbers, with an exponent and mantissa, as,

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

With limited amount of bits for each of these, only a finite precision can be achieved. The smaller the number, smaller the gap between possible numbers (and vice versa!). You can try your real numbers in this online demo.

Be aware of this behavior and try to avoid all floating points comparison and their use as stopping conditions in loops. See below two examples:

Examples: Floating point comparison done WRONG:

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

ans =

  logical

   0
```

It is poor practice to use floating point comparison as shown by the precedent example. You can overcome it by taking the absolute value of their difference and comparing it to a (small) tolerance level.

Below is another example, where a floating point number is used as a stopping condition in a while loop:**

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

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

It misses the last expected loop (`0.3 <= 0.3`).

Example: Floating point comparison done RIGHT:

```
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
```

Several things to note:

- As expected, now `x` and `y` are treated as equivalent.
- In the example above, the choice of tolerance was done arbitrarily. Thus, the chosen value might not be suitable for all cases (especially when working with much smaller numbers). Choosing the bound *intelligently* can be done using the `eps` function, i.e. `N*eps(max(x,y))`, where `N` is some problem-specific number. A reasonable choice for `N`, which is also permissive enough, is `1E2` (even though, in the above problem `N=1` would also suffice).

Further reading:

See these questions for more information about floating point inaccuracy:

- Why is 24.0000 not equal to 24.0000 in MATLAB?
- Is floating point math broken?

Section 33.4: What you see is NOT what you get: char vs cellstring in the command window

This a basic example aimed at new users. It does not focus on explaining the difference between `char` and `cellstring`.

It might happen that you want to get rid of the `'` in your strings, although you never added them. In fact, those are *artifacts* that the **command window** uses to distinguish between some types.

A string will print

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

```
dsadasd
```

A cellstring will print

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

Note how the **single quotes** and the **indentation** are artifacts to notify us that `c` is a `cellstring` rather than a `char`. The string is in fact contained in the cell, i.e.

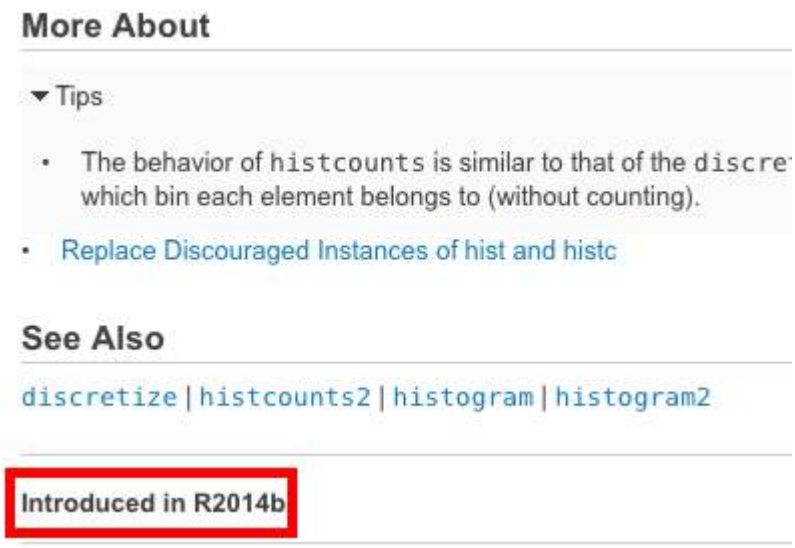
```
c{1}
ans =
dsadasd
```

Section 33.5: Undefined Function or Method X for Input Arguments of Type Y

This is MATLAB's long-winded way of saying that it cannot find the function that you're trying to call. There are a number of reasons you could get this error:

That function was introduced *after* your current version of MATLAB

The MATLAB online documentation provides a very nice feature which allows you to determine in what version a given function was introduced. It is located in the bottom left of every page of the documentation:

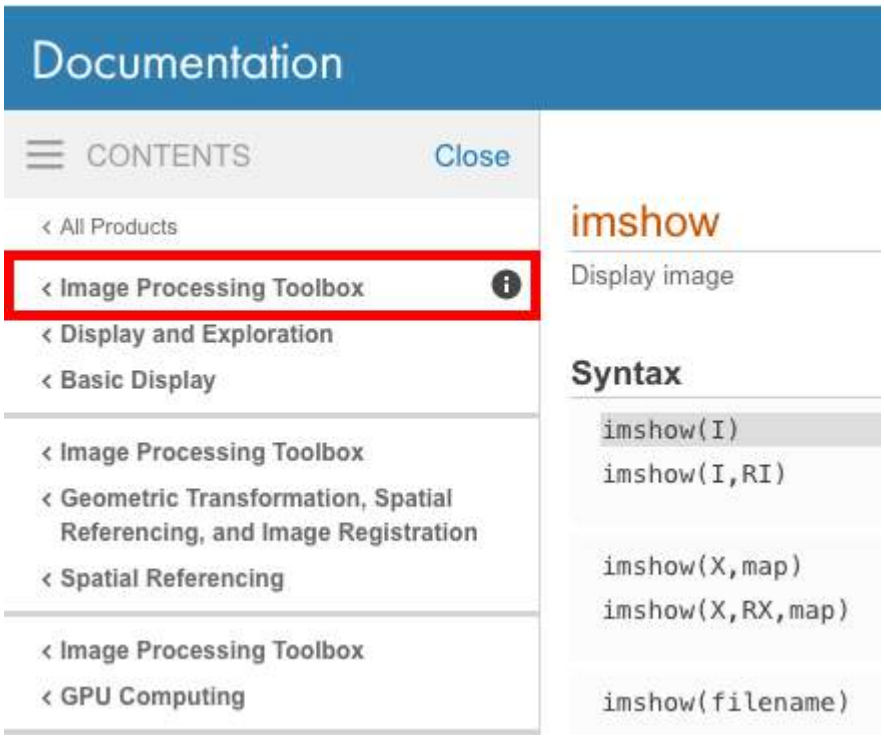


Compare this version with your own current version (`ver`) to determine if this function is available in your particular version. If it's not, try searching the archived versions of the documentation to find a suitable alternative in your version.

You don't have that toolbox!

The base MATLAB installation has a large number of functions; however, more specialized functionality is packaged within toolboxes and sold separately by MathWorks. The documentation for *all* toolboxes is visible whether you have the toolbox or not so be sure to check and see if you have the appropriate toolbox.

To check which toolbox a given function belongs to, look to the top left of the online documentation to see if a specific toolbox is mentioned.



You can then determine which toolboxes your version of MATLAB has installed by issuing the `ver` command which will print a list of all installed toolboxes.

If you do not have that toolbox installed and want to use the function, you will need to purchase a license for that particular toolbox from MathWorks.

MATLAB cannot locate the function

If MATLAB still can't find your function, then it must be a user-defined function. It is possible that it lives in another directory and that directory should be added to the search path for your code to run. You can check whether MATLAB can locate your function by using `which` which should return the path to the source file.

Section 33.6: The use of "i" or "j" as imaginary unit, loop indices or common variable

Recommendation

Because the symbols `i` and `j` can represent significantly different things in MATLAB, their use as loop indices has split the MATLAB user community since ages. While some historic performance reasons could help the balance lean to one side, this is no longer the case and now the choice rest entirely on you and the coding practices you choose to follow.

The current official recommendations from MathWorks are:

> - Since `i` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.
> - For speed and improved robustness in complex arithmetic, use `1i` and `1j` instead of `i` and `j`.

Default

In MATLAB, by default, the letters `i` and `j` are built-in `function` names, which both refer to the imaginary unit in the complex domain.

So by default `i = j = sqrt(-1)`.

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

and as you should expect:

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

Using them as a variable (for loop indices or other variable)

MATLAB allows using built-in function name as a standard variable. In this case the symbol used will not point to the built-in function any more but to your own user defined variable. This practice, however, is not generally recommended as it can lead to confusion, difficult debugging and maintenance (*see other example do-not-name-a-variable-with-an-existing-function-name*).

If you are ultra pedantic about respecting conventions and best practices, you will avoid using them as loop indices in this language. However, it is allowed by the compiler and perfectly functional so you may also choose to keep old habits and use them as loop iterators.

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

Note that loop indices do not go out of scope at the end of the loop, so they keep their new value.

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

In the case you use them as variable, make sure **they are initialised** before they are used. In the loop above MATLAB initialise them automatically when it prepare the loop, but if not initialised properly you can quickly see that you may inadvertently introduce `complex` numbers in your result.

If later on, you need to undo the shadowing of the built-in function (=e.g. you want `i` and `j` to represent the imaginary unit again), you can `clear` the variables:

```
>> clear i j
```

You understand now the MathWorks reservation about using them as loop indices *if you intend to use them in complex arithmetic*. Your code would be riddled with variable initialisations and `clear` commands, best way to confuse the most serious programmer (*yes you there!...*) and program accidents waiting to happen.

If no complex arithmetic is expected, the use of `i` and `j` is perfectly functional and there is no performance penalty.

Using them as imaginary unit:

If your code has to deal with `complex` numbers, then `i` and `j` will certainly come in handy. However, for the sake of disambiguation and even for performances, it is recommended to use the full form instead of the shorthand syntax. The full form is `1i` (or `1j`).

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

They do represent the same value `sqrt(-1)`, but the later form:

- is more explicit, in a semantic way.
- is more maintainable (someone looking at your code later will not have to read up the code to find whether `i` or `j` was a variable or the imaginary unit).
- is faster (source: MathWorks).

Note that the full syntax `1i` is valid with any number preceding the symbol:

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

This is the only function which you can stick with a number without an operator between them.

Pitfalls

While their use as *imaginary unit* **OR** *variable* is perfectly legal, here is just a small example of how confusing it could get if both usages get mixed:

Let's override `i` and make it a variable:

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

Now `i` is a *variable* (holding the value `3`), but we only override the *shorthand* notation of the imaginary unit, the full form is still interpreted correctly:

```
>> 3i
ans =
   0.0000 + 3.0000i
```

Which now lets us build the most obscure formulations. I let you assess the readability of all the following constructs:

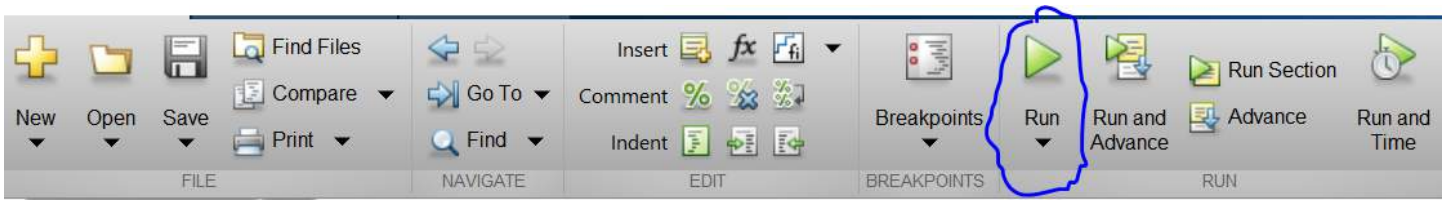
```
>> [ i ; 3i ; 3*i ; i+3i ; i+3*i ]
ans =
   3.0000 + 0.0000i
   0.0000 + 3.0000i
   9.0000 + 0.0000i
   3.0000 + 3.0000i
```

```
  12.0000 + 0.0000i
```

As you can see, each value in the array above return a different result. While each result is valid (provided that was the initial intent), most of you will admit that it would be a proper nightmare to read a code riddled with such constructs.

Section 33.7: Not enough input arguments

Often beginning MATLAB developers will use MATLAB's editor to write and edit code, in particular custom functions with inputs and outputs. There is a *Run* button at the top that is available in recent versions of MATLAB:



Once the developer finishes with the code, they are often tempted to push the *Run* button. For some functions this will work fine, but for others they will receive a `Not enough input arguments` error and be puzzled about why the error occurs.

The reason why this error may not happen is because you wrote a MATLAB script or a function that takes in no input arguments. Using the *Run* button will run a test script or run a function assuming no input arguments. If your function requires input arguments, the `Not enough input arguments` error will occur as you have written a functions that expects inputs to go inside the function. Therefore, you cannot expect the function to run by simply pushing the *Run* button.

To demonstrate this issue, suppose we have a function `mult` that simply multiplies two matrices together:

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

In recent versions of MATLAB, if you wrote this function and pushed the *Run* button, it will give you the error we expect:

```
>> mult
Not enough input arguments.

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

There are two ways to resolve this issue:

Method #1 - Through the Command Prompt

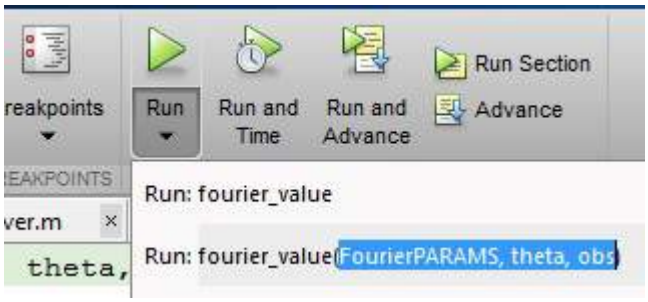
Simply create the inputs you need in the Command Prompt, then run the function using those inputs you have created:

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

Method #2 - Interactively through the Editor

Underneath the *Run* button, there is a dark black arrow. If you click on that arrow, you can specify the variables you

would like to get from the MATLAB workspace by typing the way you want to call the function exactly as how you have seen in method #1. Be sure that the variables you are specifying inside the function exist in the MATLAB workspace:



Section 33.8: Using `length` for multidimensional arrays

A common mistake MATLAB coders have, is using the `length` function for matrices (as opposed to **vectors**, for which it is intended). The `length` function, as mentioned in its documentation, "*returns the length of the largest array dimension*" of the input.

For vectors, the return value of `length` has two different meanings:

1. The total number of elements in the vector.
2. The largest dimension of the vector.

Unlike in vectors, the above values would not be equal for arrays of more than one non-singleton (i.e. whose size is larger than 1) dimension. This is why using `length` for matrices is ambiguous. Instead, using one of the following functions is encouraged, even when working with vectors, to make the intention of the code perfectly clear:

1. `size(A)` - returns a row vector whose elements contain the amount of elements along the corresponding dimension of `A`.
2. `numel(A)` - returns the number of elements in `A`. Equivalent to `prod(size(A))`.
3. `ndims(A)` - returns the number of dimensions in the array `A`. Equivalent to `numel(size(A))`.

This is especially important when writing "future-proof", vectorized library functions, whose inputs are not known in advance, and can have various sizes and shapes.

Section 33.9: Watch out for array size changes

Some common operations in MATLAB, like **differentiation** or **integration**, output results that have a different amount of elements than the input data has. This fact can easily be overlooked, which would usually cause errors like `Matrix dimensions must agree`. Consider the following example:

```
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)
```

Let's say we want to plot these results. We take a look at the array sizes and see:

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

But:

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

The array is one element shorter!

Now imagine you have measurement data of positions over time and want to calculate *jerk(t)*, you will get an array 3 elements less than the time array (because the jerk is the position differentiated 3 times).

```
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
```

And then operations like:

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

return errors, because the matrix dimensions do not agree.

To calculate operations like above you have to adjust the bigger array size to fit the smaller one. You could also run a regression (`polyfit`) with your data to get a polynomial for your data.

Dimension Mismatch Errors

Dimension mismatch errors typically appear when:

- Not paying attention to the shape of returned variables from function/method calls. In many inbuilt MATLAB functions, matrices are converted into vectors to speed up the calculations, and the returned variable might still be a vector rather than the matrix we expected. This is also a common scenario when logical masking is involved.
- Using incompatible array sizes while invoking implicit array expansion.

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

adjpayot	Chapter 1
Adriaan	Chapter 27
agent_C.Hdj	Chapter 8
Alexander Korovin	Chapter 10
alexforrence	Chapters 6 and 9
Amro	Chapters 1, 12 and 17
Ander Biguri	Chapters 6, 15, 24, 28 and 33
anyanwu	Chapter 3
Batsu	Chapter 4
Cape Code	Chapter 15
ceiltechbladhm	Chapter 6
Celdor	Chapters 9, 12 and 32
chrisb2244	Chapter 1
Christopher Creutzig	Chapter 1
codeaviator	Chapter 29
daleonpz	Chapter 26
Dan	Chapters 1 and 10
daren shan	Chapters 9 and 27
Dev	Chapters 6, 9, 10, 23, 25, 26, 29, 31 and 33
drhagen	Chapter 8
DVarga	Chapters 1 and 25
EBH	Chapters 1, 3, 8, 10 and 32
edwinksl	Chapter 33
Eric	Chapter 6
Erik	Chapters 1, 29 and 32
excaza	Chapter 1
flawr	Chapter 1
Franck Dernoncourt	Chapters 17 and 27
fyrepenguin	Chapters 1 and 32
GameOfThrows	Chapters 1 and 18
girish_m	Chapter 15
Hardik_Jain	Chapter 27
Hoki	Chapters 28, 31 and 33
il_raffa	Chapters 16, 26 and 30
itzik Ben Shabat	Chapter 13
jenszvs	Chapter 25
Jim	Chapter 27
jkazan	Chapter 22
Justin	Chapter 25
Kenn Sebesta	Chapter 28
Landak	Chapters 1, 7, 18 and 33
Lior	Chapter 1
Malick	Chapters 30, 32 and 33
matlabgui	Chapter 29
Matt	Chapters 1, 10, 12 and 33
MayeulC	Chapter 30
McLemon	Chapter 30

mhopeng	Chapter 28
mike	Chapter 24
Mikhail_Sam	Chapters 1 and 7
Mohsen Nosratinia	Chapters 7 and 9
nahomyaja	Chapter 33
nitsua60	Chapter 16
NKN	Chapters 16, 30 and 33
Noa Regev	Chapter 14
Oleg	Chapters 10, 12, 17, 28 and 33
pseudoDust	Chapter 26
R. Joiny	Chapter 33
rajah9	Chapter 2
rayryeng	Chapter 33
Royi	Chapter 23
S. Radev	Chapter 7
Sam Roberts	Chapter 1
Sardar Usama	Chapter 30
Shai	Chapters 1, 5, 10 and 15
StefanM	Chapters 5, 11, 19, 20 and 21
Suever	Chapters 31 and 33
thewaywewalk	Chapters 16, 26 and 29
Tim	Chapter 33
Trilarion	Chapter 15
Trogdor	Chapter 9
Tyler	Chapters 1, 8 and 10
Umar	Chapter 33
Zep	Chapters 16 and 31

You may also like