Introduction to MATLAB

ACM11 Spring 2015,
California Institute of Technology

May 10, 2015

Preface

ACM11 (now in Spring 2015) in Caltech is a course where we introduce math softwares MATLAB and Mathematica. This course aims for a 1-month learning in MATLAB and another month in Mathematica. For the MATLAB portion of this course, students are expected to learn basic operations, basic graphics, basic tools for linear algebra and sparse matrices, basic statistics tools, basic numerical ODE, PDE, nonlinear equations and optimization, processing signals and data, struct and class objects and basic GUIs.

Over the years class materials are passed on to different instructors (graduate students) and have condensed essential concepts and gathered inspiring examples from various interest of applications. However many of these materials were written in script m-files instead of a formal lecture note. Here I collect notes from those script files and write this lecture note.

Hence, I especially thank the previous instructors Eldar Akhmetgaliyev, Patrick Sanan, and many earlier instructors for their contributions.

— Albert Chern 2015

Contents

Chapter 1

Introduction and Basic Operations

As the first chapter of the lecture note, we give a brief introduction to MATLAB and a few basic operations in MATLAB. After going through this chapter, you can at least use MATLAB as a calculator.

1.1 What is MATLAB?

"MATLAB" is the short for **matrix laboratory**. It is a numerical computing environment and a programming language which provides a suite of tools for computation, visualization, and more. MATLAB is widely used in academic and research institutions as well as industrial enterprises.

When to use MATLAB?

1. for rapid prototyping of numerical algorithms.
2. for quick data analysis and visualization.

When not to use MATLAB?

1. when you need to handle truly massive datasets.
2. when efficiency is paramount.
3. when you need production quality code.
4. when you need to produce high-quality graphics.
5. when you need symbolic capabilities.

Competitors/Alternatives

- Julia, a high-level dynamic programming language aiming high-performance scientific computating (appeared in 2012)
- GNU Octave, an open-source clone of MATLAB
- computational languages/environments like S and R

- Excel
- Mathematica, Maple, Axiom, other CASes (Computer Algebra Systems)
- systems languages (C/C++, ...)
- general purpose languages (Java, Lisp, ...)
- scripting languages (Python/NumPy/SciPy, Perl, ...)
- MATLAB clones (Scilab, ...)

In one way or another, most of MATLAB's competitors can do what MATLAB does. For instance, you can solve Poisson's equation in Excel, but that doesn't mean you should. Here are some of MATLAB's advantages:

1. the language is intuitive and mathematically expressive (vectorization!)
2. the documentation is excellent
3. many toolkits are available which extend the functionality
4. the debugger and profiler are integrated and easy to use
5. MATLAB is an industry standard.
6. MATLAB matrix manipulation algorithms (esp. for sparse matrices) are state of the art

and disadvantages:

1. the scripting system is somewhat primitive
2. for complex tasks (especially ones which require for loops), MATLAB can sometimes be slower than hand-coded C or Fortran
3. MATLAB is expensive

1.2 MATLAB Interface

The MATLAB desktop consists of the following panels.

Command Window Users are allowed to type MATLAB codes or basic unix commands into the command window. Press `<return>` to execute the command. Input command lines follow after a ">>" symbol, while printed results typically do not. For instance,

```
>> 1+3*2

ans =

     7

>>
```

As another example, type `pwd` (the unix command "print working directory") to see current folder. Type `clc` command to clear commands displayed in the window. Features such as `<tab>`-completion and `<up>`-key for command history are available in the command window.

Command History shows previous commands executed in the command window. Other history features are available at the command window (<up> key, drag-n-drop).

Current Folder shows the content of the current working directory.

Workspace shows the variables. After you type `a = 1+1` in the command window, you see `a` appear in the workspace.

Editor MATLAB built-in editor, which can be called out by clicking "New Script" or pressing <ctrl>+N (or <cmd>+N). (You can use your own editor!)

1.3 M-files

MATLAB codes can be written in files known as the "m-files" having extension "`.m`". An m-file can either be a *script* m-file, a *function* m-file, or a *class* m-file. It is the first non-comment line of the code in a m-file that determines which kind (script, function or class) the m-file is.

1.3.1 Script m-file

An m-file is by default a **script m-file**. A script m-file is literally a list of MATLAB commands. Suppose `my_script.m` is a script m-file saved in the current directory (that is, visible in the Current Folder window). Then one may type

```
>> my_script
```

in the command window to execute the list of commands in `my_script.m`.

Example 1.1. An example of a script m-file:

`my_script.m`

```
a = 3;
b = 4;
c = sqrt( a^2 + b^2 )  % Pythagorean formula
```

Run the script by typing the filename in the command window:

```
>> my_script

c =

     5

>>
```

From the above example, you may have noticed that variables `a`, `b` and `c` appear in the workspace window. You may have also noticed what semicolons "`;`" do: a semicolon postfix suppresses MATLAB from printing the result of the command.

In the above example, `sqrt( )` is a MATLAB built-in function (square-root function).

1.3.2 Function m-file

A **function m-file** is a file where one defines a function. The first (non-comment) line of a function m-file takes the form

```
function  [output_args] = my_function( input_args )
```

Importantly, the name of the function "`my_function`" need to be the same as the filename of the m-file "**`my_function.m`**".

Suppose **`my_function.m`** is a function m-file saved in the current folder, one may use this user-customized function "`my_function`" in the command window or script m-files in the same folder.

Example 1.2. An example of a function m-file:

my_function.m

```
% a few lines of comment
%

% or blank lines are ok
function c = my_function( a, b )
aSquare = a^2;
bSquare = b^2;
s = aSquare + bSquare;
c = sqrt( s );
```

Now in the Command Window

```
>> d = my_function( 3, 4)

d =

     5

>>
```

As one execute the command "`d = my_function( 3, 4)`", MATLAB "dives into" the file **`my_function.m`** and runs the commands with `a`= 3 and `b`= 4. As MATLAB reaches the end of the file, having the calculated output value `c`= 5, MATLAB "jumps out of" the file and assign the value 5 to `d`. Notice that none of the variables `a`, `b`, `aSquare`, `bSquare`, `s`, `c` in the function m-file are reflected in the workspace. Variables in a function m-file are *local variables*. When MATLAB "dives into" or "jumps out of" a function, it only brings the values of the input arguments and output arguments with it.

Example 1.3. MATLAB function can have multiple output arguments.

my_function.m

```
function [c,s] = my_function(a,b)
s = a^2 + b^2;
c = sqrt(s);
```

```
>> [d,s] = my_function(3,4);
>> d

d =

     5

>> s

s =

    25

>>
```

1.3.3 Class m-file

The idea of **class m-file** is usually covered in a more advanced topic in MATLAB. It is just for the purpose of completing seeing all 3 types of m-file that we show class m-files in this chapter. It does not make sense to dig into details of involved concepts of object oriented programming before knowing the basic operations in MATLAB. So, skip this section if you do not find yourself enjoying the following example.

Example 1.4. An example of a class m-file:

RightTriangle.m

```
classdef RightTriangle
% class of right triangle
    properties
        a = 0 % length of a leg, default = 0
        b = 0 % length of the other leg, default = 0
    end
    methods
        function m = area(obj)
        % returns the area of a RightTriangle obj
            m = obj.a * obj.b / 2;
        end
        function c = hypotenuse(obj)
        % returns the hypotenutes of a RightTriangle obj
            c = sqrt( obj.a ^ 2 + obj.b ^ 2 );
        end
    end
end
```

In the command window

```
>> tri = RightTriangle;
>> tri.a = 3;
>> tri.b = 4;
>> tri.area

ans =
```

```
     6

>> tri.hypotenuse

ans =

     5

>> hypotenuse(tri)

ans =

     5

>>
```

Similar to function m-file, the file name must agree with the *class name*, which is "`RightTriangle`" in this case. When one types `tri = RightTriangle`, one creates a variable (or an *object*, or an *instance*) `tri` of class "`RightTriangle`". The object `tri` has *properties* (or called *class members*) `a` and `b` as found in **`RightTriangle.m`**. As shown in this example, the property `a` of `tri` can be accessed by typing `tri.a`.

One can literally say "this variable `tri` is a right triangle, which is determined by the length of the two legs `a` and `b`".

Functions defined for objects of a class are called *class methods*. One may call the class method `area` of the right triangle `tri` by typing either `area(tri)` or `tri.area`.

The high-level concept for class method is that one can say "the value returned from `area` is a function of a right triangle `tri`".

User-customized classes can be very useful in general purpose. For example, one may define one's own number system with customized `+`,`*` operations. For another example, a class can be defined with *properties* being experimental data and parameters of a specific problem, and with *methods* being various data analysis tools (or anything returning values as a function of the data and parameters). A correct usage of class (or object-oriented programming) will in general provide a high-level and organized program.

1.3.4 Comments in Code

You may have noticed that "`%`" in MATLAB code denotes that the rest of that line is a comment. Another useful way of creating comments is the pair "`%{`, `%}`".

```
% This is a comment
%{
  These lines
  are
  all
  comments
%}
```

Use comments to document your code!

1.4 The Help System: your best friend

The command `help` shows the format(s) for using a command and directs you to related commands; without any arguments, it gives you a hyperlinked list of topics to find help on; with a topic as an argument, it gives you a list of subtopics.

```
help plot
help qr
help
```

If you want to see all the commands associated with *elementary matrix manipulation*

```
help matlab/elmat
```

The command `doc` is like `help`, except it comes up in a different window, and may include more details

```
help fft
doc fft
```

The command `lookfor` is used when you do not know what command you want; it does something like a keyword search through the documentation

```
lookfor wavelet
```

Similarly, you can use `docsearch`

```
docsearch fourier
```

The command `which` helps you tell which file a particular command refers to, or whether it is built in

```
which abs
which hadamard
```

`demo` gives video guides and example code

```
demo
```

1.5 Abort Command: Ctrl + C

If MATLAB is running a program and you want to terminate it, type Ctrl + C.

1.6 Basic Operations

In this section you will learn the basic operations in MATLAB and know how to use MATLAB as a calculator. You can try all examples in the command window.

Here are some standard commands to start a session:

```
clear all  % clears all variable definitions
close all  % closes all figures
clc        % clears the screen
```

1.6.1 Scalar Arithmetics

MATLAB has the basic arithmetics for scalars such as `+`, `-`, `*`, `/`, power `^`.

```
a = 3;
b = 5;
a + b
a - b
a * b
a / b
a ^ b
mod(b,a)
```

Fractional power and square root are also available

```
8^(1/3)
sqrt(9)
```

One may also create complex numbers by taking square root to negative number

```
>> z = 1+sqrt(-1)
z =
   1.0000 + 1.0000i
```

or

```
>> z = 1 + 1i
z =
   1.0000 + 1.0000i
```

Its complex conjugation and absolute value are

```
>> conj(z)
ans =
   1.0000 - 1.0000i
>> abs(z)
ans =
    1.4142
```

One may verify the famous *Euler formula*

$$e^{i\pi} = -1$$

```
>> exp(1i*pi)
ans =
  -1.0000 + 0.0000i
```

Note that MATLAB has the math constant π built-in, but not for the natural constant e. To obtain the natural constant e, use `exp(1)`.

```
>> pi
ans =
    3.1416
>> exp(1)
ans =
    2.7183
```

Although MATLAB stores numerical values of variables with double precision (16 decimal digits), the command window displays numerics result only up to 4 digits. To show more digits in display

```
>> format long
>> pi
ans =
   3.141592653589793
>> format short
>> pi
ans =
    3.1416
>>
```

Note that this only changes the number of digits that is displayed. It does not change computation accuracy.

Floating point types have special values "`inf`"(∞) and "`NaN`"(not-a-number). Try these out

```
0/0
1e999^2
isnan(NaN)  % tests if the argument is nan
isinf(Inf)
isfinite(NaN)
isfinite(-Inf)
isfinite(3)
```

1.6.2 Matrix Construction

The square brackets `[ ]` concatenate elements within and create a matrix

```
>> v = [1,2,3]
v =
     1     2     3
>> A = [1,2,3;4,5,6]
A =
     1     2     3
     4     5     6
```

The comma "`,`" is used to separate elements that belong in the same row, while the semicolon "`;`" creates a new row in the array (matrix). It is also common to use the following alternative expressions

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

To see the size of a matrix

```
>> size(A)
ans =
     2     3
```

or

```
>> [m,n] = size(A)
m =
     2
n =
     3
```

The function `zeros` and `ones` are handy to create all-zero and all-one matrices:

```
>> B = ones(3,3)
B =
     1     1     1
     1     1     1
     1     1     1
>> C = zeros(5,2)
C =
     0     0
     0     0
     0     0
     0     0
     0     0
```

where the integer arguments of `zeros` and `ones` are the desired size of the matrix.

Remember how the square brackets `[ ]` concatenate elements enclosed and construct a matrix?

```
>> [[A;B],C]
ans =
     1     2     3     0     0
     4     5     6     0     0
     1     1     1     0     0
     1     1     1     0     0
     1     1     1     0     0
```

The `repmat`, replicating and tiling matrices, is a useful function for generating matrices:

```
>> repmat([3,2],4,1)
ans =
     3     2
     3     2
     3     2
     3     2
```

The colon "`:`" is one of the most useful operators in MATLAB. One of its usage is to construct row vectors with regularly spaced values.

```
>> u = 2:5
u =
     2     3     4     5
```

Another example:

```
>> u2 = pi:6
u2 =
    3.1416    4.1416    5.1416
```

The spacing does not need to be 1

```
>> u3 = 2 : 0.5 : 4
u3 =
    2.0000    2.5000    3.0000    3.5000    4.0000
>> u4 = 5 : -1 : 2
u3 =
      5     4     3     2
```

The function "`linspace`", which generates linearly spaced row vector, has a similar functionality

```
>> x = linspace(2,4,5)
x =
    2.0000    2.5000    3.0000    3.5000    4.0000
```

(5 points linearly spaced between 2 and 4).

Other methods for generating matrices:

```
eye(7)      % the identity
ones(5)     % same as ones(5,5)
rand(2,2)   % random numbers distributed uniformly in [0,1]
randn(2,2) % random numbers with standard normal distribution
inf(3,3)
nan(4,1)
magic(5)
```

Summary The basic way to generate matrices is to use the square brackets "`[ ]`" operator, in which one uses symbols such as spaces, commas, semicolons or new lines. The colon operator "`:`" constructs row vectors with equispacing numbers. Functions such as `repmat`, `linspace`, `zeros`, `ones`, `eye`, `rand`, `randn`, etc. are useful to generate elementary matrices.

1.6.3 Matrix Indexing

Let

```
>> A = [1 2 3 4
        5 6 7 8
        9 10 11 12]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

One may get the value of a matrix entry by

```
>> a = A(3,2)
a =
    10
```

or set value

```
>> A(3,2) = -20
A =
     1     2     3     4
     5     6     7     8
     9   -20    11    12
```

One may get a block from a matrix:

```
>> A([2 3],[2 3 4])
ans =
     6     7     8
   -20    11    12
```

Equivalent expressions include

```
A( 2:3 , 2:4 )
A( 2:3 , 2:end )
```

In parentheses `( )` (array indexing), "end" indicates last array index. A single colon indicates selecting all indexes

```
>> A( : , 2:end)
ans =
     2     3     4
     6     7     8
   -20    11    12
```

so one may permute columns (or rows) simply by

```
>> B = A( : , [4,2,1,3] )
B =
     4     2     1     3
     8     6     5     7
    12   -20     9    11
```

One may assign values to an entire block of a matrix

```
>> B = zeros(5,6)
B =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
>> B(2:4,2:5) = A
B =
     0     0     0     0     0     0
     0     1     2     3     4     0
     0     5     6     7     8     0
     0     9   -20    11    12     0
     0     0     0     0     0     0
```

An important concept is that MATLAB uses *column major order*. That is, if we view the matrix

```
A =
     1     2     3     4
     5     6     7     8
     9   -20    11    12
```

as a 1D array (vector),

```
>> A(:)
ans =
     1
     5
```

```
     9
     2
     6
   -20
     3
     7
    11
     4
     8
    12
```

One may access `A` by a single index; for example `A(4)` is 2 in this case. The expression `A(4)` is known as the *linear indexing*. To convert linear indices to their corresponding rows and columns, use `ind2sub`:

```
>> [r,c] = ind2sub([3,4],4)
r =
     1
c =
     2
```

where `[3,4]` is the size of `A`. You may check that `A(4)` equals to `A(1,2)`. Conversely,

```
>> ind = sub2ind([3,4],1,2)
ind =
     4
```

Reshaping is frequently used as well:

```
>> reshape(A,4,3)
ans =
     1     6    11
     5   -20     4
     9     3     8
     2     7    12
```

which reshapes `A` to a matrix with size 4×3 so that after postfixed by `(:)` it recovers `A(:)`. Note that the argument 4 and 3 must multiply to 12, the total number of entries in `A`. One may replace one of them by "empty matrix" `[]`

```
>> reshape(A,[],2)
ans =
     1     3
     5     7
     9    11
     2     4
     6     8
   -20    12
```

and MATLAB will do the calculation the only matching numbers of columns or rows for you.

The operators "`.'`" and "`'`" are transpose.

```
>> A'
ans =
     1     5     9
     2     6   -20
     3     7    11
```

```
     4     8    12
>> A.'
ans =
     1     5     9
     2     6   -20
     3     7    11
     4     8    12
```

The "undotted" transpose is the *Hermitian transpose*, which also takes complex conjugation to each elements

```
>> [1+2i,3+4i]
ans =
   1.0000 + 2.0000i   3.0000 + 4.0000i
>> [1+2i,3+4i]'
ans =
   1.0000 - 2.0000i
   3.0000 - 4.0000i
>> [1+2i,3+4i].'
ans =
   1.0000 + 2.0000i
   3.0000 + 4.0000i
```

Summary With parentheses `()` postfixing a matrix `A` one may access the matrix elements using index. In the parentheses, one may use indices which need to be positive integers (1-based indexing); colon "`:`" and "end" notations are allowed. One may either use *subscript indices* `A(r,c)` or *linear indices* `A(ind)`, and one may convert the two using the functions `sub2ind` and `ind2sub`. Reshaping (`reshape`) and transposing (`.'` and `'`) are handy in indexing as well.

1.6.4 Logical Indexing

The comparison operators > (greater than), == (equal to), ~= (not equal to), < (less than), >= (greater or equal to), <= (less or equal to), returns logical matrices. A logical matrix contains entry of value *true* or *false*, displayed as 1 or 0:

```
>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> A > 9
ans =
     1     0     0     1
     0     1     1     0
     0     0     0     1
     0     1     1     0
>> A == 10
ans =
     0     0     0     0
     0     0     1     0
     0     0     0     0
     0     0     0     0
```

Common boolean operations "and", "or", and "not" for logical matrices are "`&`", "`|`" and "`~`" respectively:

```
>> (A>5) & (A<10)
ans =
     0     0     0     0
     0     0     0     1
     1     1     1     0
     0     0     0     0
>> (A<=5) | (A>=10)
ans =
     1     1     1     1
     1     1     1     0
     0     0     0     1
     1     1     1     1
>> ~((A==4) | (A==5))
ans =
     1     1     1     1
     0     1     1     1
     1     1     1     1
     0     1     1     1
```

When the matrix is a scalar (1-by-1), MATLAB will suggest you to replace `&` with `&&` and `|` with `||`. These *non-vectorized* "and" and "or" operators for logical scalars are the *short circuit*. For the case of `a||b`, it will return *true* if `a` is true, without looking at `b`; for the case of `a&&b`, it will return *false* if `a` is false without evaluating `b`.

Now let us look at logical indexing. One may access entries of a matrix by "plugging in" a logical matrix of the same size:

```
>> isSmall = A<=5
isSmall =
     0     1     1     0
     1     0     0     0
     0     0     0     0
     1     0     0     1
>> A(isSmall) = 0
A =
    16     0     0    13
     0    11    10     8
     9     7     6    12
     0    14    15     0
```

Logical indexing can be very handy

```
>> A(A>10) = 10
A =
    10     0     0    10
     0    10    10     8
     9     7     6    10
     0    10    10     0
```

Note Logical matrices can also be returned by following functions

```
>> true(3,2)
ans =
```

```
     1     1
     1     1
     1     1
>> false(2)
ans =
     0     0
     0     0
>> isnan([nan,0,1])
ans =
     1     0     0
```

`any` and `all` are useful

```
    any([1 0 0 0])  % true if any of the vector entries is true or nonzero
    all([1 1 1 1])  % true only if all vector entries are true or nonzero
    all([1 0 0 0])  % this case it retuns false
```

1.6.5 Matrix Arithmetics

There is no ambiguity of what $A + B$ means for A and B being matrices of the same size. It adds each counterpart components together. It is also clear that cA for a scalar c and a matrix A is the matrix with each component multiplied by c. In MATLAB these operations are intuitive

```
>> A = [-1,1,2;4,2,3]
A =
    -1     1     2
     4     2     3
>> B = [2,-4,3;1,0,0]
B =
     2    -4     3
     1     0     0

>> A + B
ans =
     1    -3     5
     5     2     3
>> A+1
ans =
     0     2     3
     5     3     4
>> A*10
ans =
   -10    10    20
    40    20    30
```

However, for matrix-matrix multiplications, there is a distinction between "componentwise multiplication" and "linear-algebraic multiplication". The componentwise multiplication denoted by "`.*`" views matrices as arrays and take products of numbers in the counterpart entries:

$$C = A.*B \quad \text{means} \quad C_{ij} = A_{ij}B_{ij} \quad \text{for each } i, j$$

```
>> A.*B
ans =
```

```
    -2    -4     6
     4     0     0
```

Linear-algebraic multiplication is denoted by "$\star$"

$$C = A \star B \quad \text{means} \quad C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where $n =$ (number of columns of A) = (number of rows of B).

```
>> A = [1,2;3,4]
A =
     1     2
     3     4
>> B = [-1,1;1,2]
B =
    -1     1
     1     2
>> A*B
ans =
     1     5
     1    11
```

There are other operations and functions that other than "multiplications" that take different notations in MATLAB for elementwise operations and linear algebraic operations.

Elementwise arithmetics

Elementwise arithmetics include

```
A + B   % plus
A - B   % minus
A .* B  % times
A ./ B  % (right) division (rdivide)
1 ./ B  % division by scalar
B .\ A  % right-to-left division (ldivide)
A .^ B  % power
```

Common functions listed below also operates elementwise (There are still many functions not listed here)

```
sqrt(A)  % square root
exp(A)   % natural exponential
log(A)   % natural logarithm
log10(A) % base 10 logarithm
abs(A)   % absolute values
sin(A), cos(A), tan(A), cot(A), sec(A), csc(A)  % trigonometric functions
asin(A), acos(A), atan(A), acot(A), asec(A), acsc(A) % inverse trigonometrics
sinh(A), cosh(A), tanh(A), coth(A), sech(A), csch(A) % hyperbolic functions
asinh(A), acosh(A), atanh(A), acoth(A), asech(A), acsch(A) % inverse hyperbolics
```

Linear algebraic arithmetics

In the following, `A` and `B` are matrices, `b` is a column vector, and `c` is a scalar.

```
A * B   % matrix times (mtimes)
A / c, c \ A   % divide by scalar (mrdivide, mldivide)
A \ b   % solves the linear system Ax = b (mldivide)
b'/A    % solves x'A = b', that is A'x = b
A^2     % this case is the same as A*A (mpower)
```

For square matrix A, the matrix exponential is defined to be

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} A^k.$$

To evaluate matrix exponential in MATLAB

```
expm(A)  % matrix exponential
```

A square root of a matrix A is a matrix B (may not be unique or even exist for general square matrix A) so that $B^2 = A$:

```
sqrtm(A) % matrix square root
```

Example 1.5. If I is the 3×3 identity matrix

```
>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
```

what are `exp(I)` and `expm(I)`?

Solution.

$$\texttt{exp(I)} = \begin{bmatrix} e & 1 & 1 \\ 1 & e & 1 \\ 1 & 1 & e \end{bmatrix}, \quad \texttt{expm(I)} = \begin{bmatrix} e & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & e \end{bmatrix}.$$

■

1.6.6 Strings

Strings are just arrays of character (`char`) values in MATLAB.

```
>> 'hello world'
ans =
hello world
>> ['h','e','llo w','orld']
ans =
hello world
```

Characters are essentially integers

```
>> char(77)
ans =
M
```

```
>> double('s')
ans =
   115
>> double('7')
ans =
    55
```

To convert a string that *represents a number* to a number

```
>> str2double('123')
ans =
   123
```

To convert a number to a string displaying that number

```
>> S = num2str(123)
S =
123
```

"`disp(S)`" displays a string `S`:

```
>> a = 123
a =
   123
>> disp(['My favorite number is ',num2str(a)])
My favorite number is 123
```

"`disp`" also displays numbers or other values

```
>> disp(3)
     3
```

A C-compatible expression

```
>> fprintf('integer: %d, double: %f, string: %s \n',1234,0.999, 'Hello World.')
integer: 1234, double: 0.999000, string: Hello World.
>> S = sprintf('integer: %d, double: %f, string: %s \n',1234,0.999, 'Hello World.')
S =
integer: 1234, double: 0.999000, string: Hello World.
```

To test equality of two strings, use `strcmp` instead of ==

```
strcmp('aaa','bbbb')
strcmp('aaa','aaa')
```

1.6.7 Loops and Controls

We learn four commands here: `for`, `while`, `if` and `switch`.

The basic form of "for loops" in MATLAB takes

```
for i=1:10
   disp(i);
end
```

Here `i` is the *index* or *iterator*, "`1:10`" is the *value array*, and everything between `for` and `end` are program statements that will be executed repeatedly for `i` iterating through each value in the value array.

"If control" in MATLAB

```
    if (1 > 0)
       disp(' 1 is greater than 0');
    else
        disp('this will never happen in this case')
    end
```

and while loops:

```
a = 1;
while a < 100
   disp(a);
   a = a*2; %MATLAB still doesnt't have *= and similar operators
end
```

The *expression* after `while` (here "`a<100`") or `if` is a logical scalar or a real numerics. The statements between `while` and `end` will be repeatedly executed until that *expression* is *false* or 0.

"`break`" and "`continue`" can terminate loops:

```
    % break and continue
    a = 0;
    while (1)
        a = a + 7;
        disp(a)
        if mod(a,5)==0
           break; %breaks the loop
        end
    end

    for i=1:100
       if i > 10
           continue; %skips the rest of the loop
       end
       disp(i)
    end
```

The "`switch`/`case`" switches among cases based on expression

```
a = 'str';
switch a % <-- a scalar or a string
    case 1,
        disp('one');
    case 2,
        disp('two');
    case 'str',
        disp(a);
    otherwise,
        disp('other');
end
```

Chapter 2

Functions and Graphics

In this chapter you will learn a few ways to define your own functions in MATLAB. You will also learn methods for plotting functions in MATLAB.

2.1 Functions

There are two ways to produce a MATLAB function:

1. write a function in a named file.
2. define at command line an *anonymous function*.

2.1.1 Function m-file (revisit)

We introduce how to write a customized function in MATLAB as an m-file by looking at an example. The style of using nested functions is also explained.

Example 2.1. Define the following function as a function m-file

$$S(x,n) = \frac{4}{\pi}\sum_{k=1}^{n} a_k \sin(kx)$$

where

$$a_k = \begin{cases} 0 & k \text{ is even} \\ \frac{1}{k} & k \text{ is odd.} \end{cases}$$

Side note. As $n \to \infty$, this trigonometric series with this particular choice of a_k "approaches" the *square-wave* function $f(x) = \begin{cases} 1 & x \in (2m\pi, (2m+1)\pi) \\ -1 & x \in ((2m-1)\pi, \pi) \end{cases}$.

Solution (straightforward).

SS.m

```
function y = SS(x,n)

s = 0; % initialize the sum
for k = 1:n
    if mod(k,2)==1 % if k is odd
        s = s + 1/k * sin(k*x);
    end
end
y = 4/pi * s;
```

Note that this function supports the case that `x` is an array (matrix), in which case the function effectively computes the result elementwise. In other words, when the input `x` is a matrix, the result `y` becomes a matrix of the same size, with $\mathtt{y}_{ij} = S(\mathtt{x}_{ij}, n)$.

■

Solution (using nested functions). One may write several functions in a single function m-file.

SS.m

```
function y = SS(x,n)

    s = 0; % initialize the sum
    for k = 1:n
        s = s + coef(k) * sin(k*x);
    end
    y = 4/pi * s;


function a = coef(k)
    if mod(k,2)==1 % if k is odd
        a = 1./k;
    else
        a = 0;
    end
```

The filename must agree the name of the *main function*, which is the first `function` appeared in the file (in this case `SS`). The *subfunctions* (in this case `coef`) are visible *only* in this m-file.

Subfunctions can be nested as well:

SS.m

```
    function y = SS(x,n)

    s = 0; % initialize the sum
    for k = 1:n
        s = s + coef(k) * sin(k*x);
    end
    y = 4/pi * s;
    end

    function a = coef(k)
    % coefficients of the sine series
```

```
if isOdd(k)
    a = 1./k;
else
    a = 0;
end

    function i = isOdd(k)
        % tests whether k is odd or not
        i = (mod(k,2)==1);
    end
end
```

In this case, `function`'s are paired with `end`'s. Note that in this case, the function "`isOdd`" is not visible to the main function "`SS`".

In nested functions, the *child function* recognizes the variables in its *parent function*. For example, in the above case `isOdd` is the child of `coef`; `coef` shares its data to `isOdd`. We may remove the input argument "`(k)`" from "`isOdd(k)`".

SS.m

```
function y = SS(x,n)

s = 0; % initialize the sum
for k = 1:n
    s = s + coef(k) * sin(k*x);
end
y = 4/pi * s;
end

function a = coef(k)
% coefficients of the sine series
if isOdd
    a = 1./k;
else
    a = 0;
end

    function i = isOdd
        % tests whether k is odd or not
        % the function recognizes the variable k in the parent function
        i = (mod(k,2)==1);
    end
end
```

■

2.1.2 Anonymous function

Suppose we want to define a function $Q(x, y) := \sin(x)\cos(y)$, we may once again write an m-file

MyFunctionNumber11

```
function z = MyFunctionNumber11(x,y)
% One of lots of functions... if every function is written in a file
    z = sin(x).*cos(y);
```

or define the function as an *anonymous function* with a single command using the "@" operator:

```
>> Q = @(x,y) sin(x).*cos(y);
```

To call this function,

```
>> Q(pi/2,pi)
ans =
    -1
```

The syntax of defining an anonymous function is

$$\textit{function\_handle} = \texttt{@(}\textit{argument\_1}, \textit{argument\_2}, \ldots\texttt{)}\ \textit{expression}.$$

The output object (in the above example) `Q` is a *function handle*, which you can view as another variable in the workspace, but representing a function instead of a numerics.

Speaking of function handles... One may create function handles from existing m-files:

```
>> f = @SS;
```

(Here we assume `SS.m` in Example 2.1 is in your current folder). Or one may create function handles from built-in functions

```
>> g = @sin;
```

Or anonymous function

```
>> h = @(a,b,c) a+b+c;
```

There are functions or commands that require you to feed in a function handle. For example, the built-in function

$$q = \texttt{quad(}\textit{fun}\texttt{,}\ a\texttt{,}\ b\texttt{)}$$

computes the numerical integral $q \approx \int_a^b fun(x)\,dx$. Here *fun* is a function handle.

```
>> quad(@sin,0,pi)
ans =
    2.0000

>> q = quad(@(x) x.^2,0,1);
>> q
q =
    0.3333

>> quad(@(x)SS(x,500),0,pi)
ans =
    3.1400
```

Example 2.2. Define the following function using anonymous function.

$$f(x) = \begin{cases} \sqrt{1-x^2}, & -1 < x < 1 \\ -\sqrt{x^2-1}, & x \geq 1 \text{ or } x \leq -1. \end{cases}$$

Solution. Logical expressions return array of true or false, which are 0's or 1's; *convex combining* expressions with coefficients 0's or 1's effectively switches between expressions.

```
f = @(x)   (x>-1 & x<1).*    sqrt(1-x.^2) + ...
          (x>=1 | x<=-1).* -sqrt(x.^2-1);
```

Note that this supports the input argument `x` being an array.

■

2.2 Figure and Axes

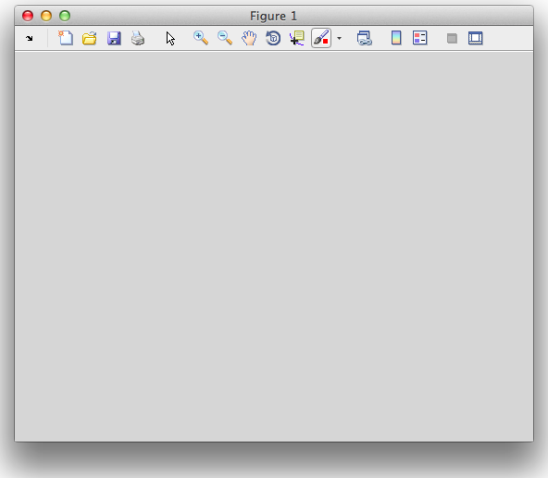
Figure is MATLAB's name for graphics window. A *figure* can contain *axes*, which is a frame equipped with a coordinate system (2D or 3D, linear or log-scale,...). An *axes* may contain plots.

2.2.1 Figure

Type

```
>> figure
```

to create a new figure.



MATLAB uses *figure handles* (represented as type `double`) to refer to these windows

```
hFig1 = figure
hFig333 = figure(333)
```

One may set properties of an existing figure

```
set(hFig1, 'Position', [0 0 400 500])
```

or a new figure

```
figure('Position', [0 0 400 500],'Name','An Empty Graphics Window')
```

An entire list of figure properties is available in MATLAB documentation

```
docsearch 'figure properties'
```

You can also adjust properties interactively

```
propertyeditor
```

We won't pursue MATLAB's extensive set of tools of this type (the `propertyeditor`), for two main reasons :

- These GUI-based tools are (in theory) largely self-documenting
- For most scientific applications, figures need to be regenerated frequently, so its worth the effort to do it with scripts

Do note all the menus and buttons MATLAB provides, though!

Useful commands:

```
gcf % creates a figure if none exists, and returns the current figure
clf %clears the current figure
close(hFig1)
close %same as close(gcf)
close all %close('all')
```

Handles (like figure handles or axes handles, handles for a plot, etc) are *references* to objects. One may pass the value of handles (just a number) to functions without copying the whole figure object.

Example 2.3. The following function creates a full screen window.

my_figure.m

```
function hFig = my_figure
hFig = figure;

fullScreen(hFig)
end

function fullScreen(hFig)
scrsz = get(0,'ScreenSize');
set(hFig,'Position',scrsz);
end
```

Note that this program only gives a copy of the figure handle `hFig` to the function `fullScreen`. With only this local variable, `fullScreen` has the full control of this existing figure `hFig` refers to.

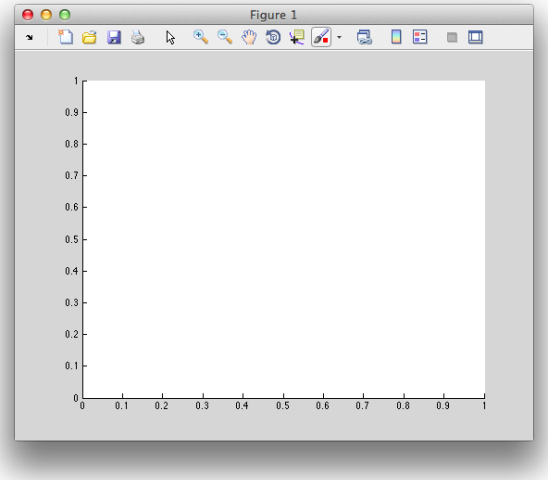
In the line `scrsz = get(0,'ScreenSize')`, "0" is the handle (reference) to the *root*. `'ScreenSize'` is one of the properties of the *root*.

2.2.2 Axes

Similar to *figure*, you may create a new *axes* by typing

```
hAxes = axes
```

which creates a new *axes* in `gcf` (the current figure).



Useful commands

```
gca % current axes
cla % clears the current axes
```

Find list of axes properties

```
docsearch 'axes properties'
```

`subplot` is a shortcut to create and work with grids of axes within a single figure

```
subplot(3,4,1)
subplot(3,4,4)
subplot(3,4,11);
```

2.3 2D Plotting

2.3.1 The `plot` function

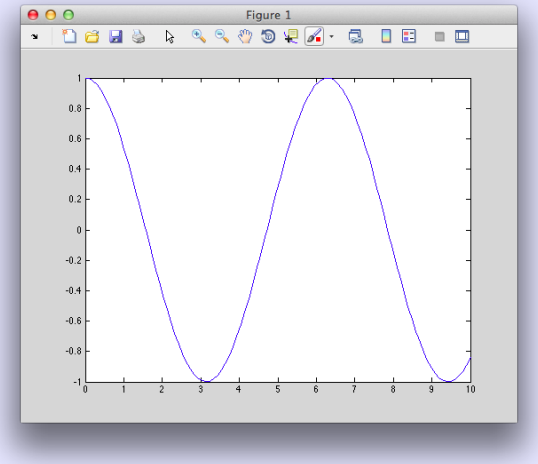
The `plot` function creates a sequence of lines in `gca`. Let `x` and `y` be two vectors of the same size. Then

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

plots the sequence of points $\Big((x_1, y_1), \ldots, (x_N, y_N)\Big)$. The style of how these points are displayed can be adjusted by setting properties.

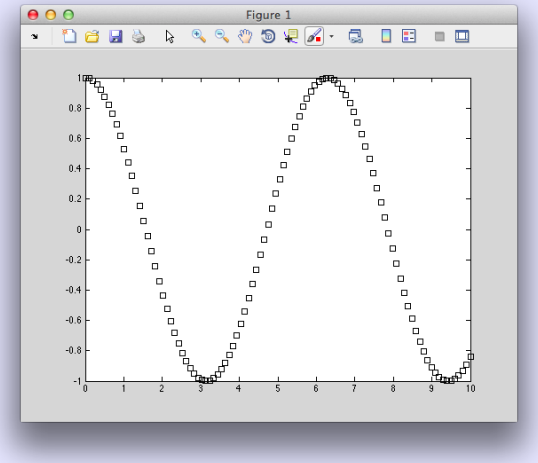
Example 2.4. Very basic plotting

```
close all;clear;clc;
x = linspace(0,10,100);
y = cos(x);
plot(x,y);
```



Search *LineSpec* (`docsearch LineSpec`), which is a basic specification of the line style using string syntax.

```
plot(x,y,'ks')
```

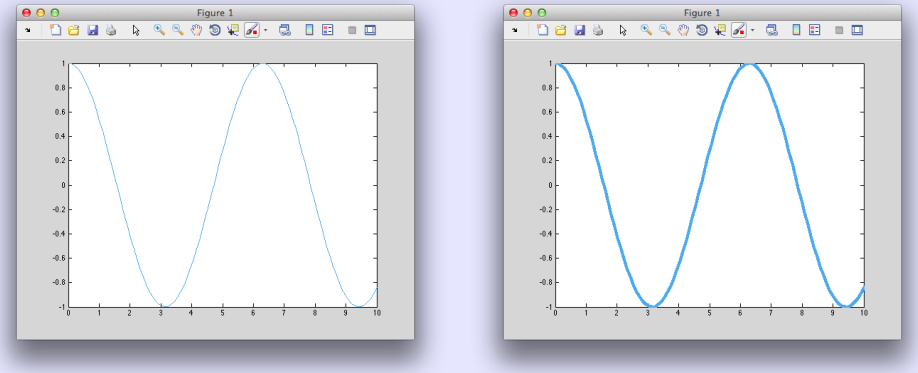


The object `plot` creates is *lineseries*. One may control the style of the plot by setting its property

```
docsearch 'Lineseries properties'
```

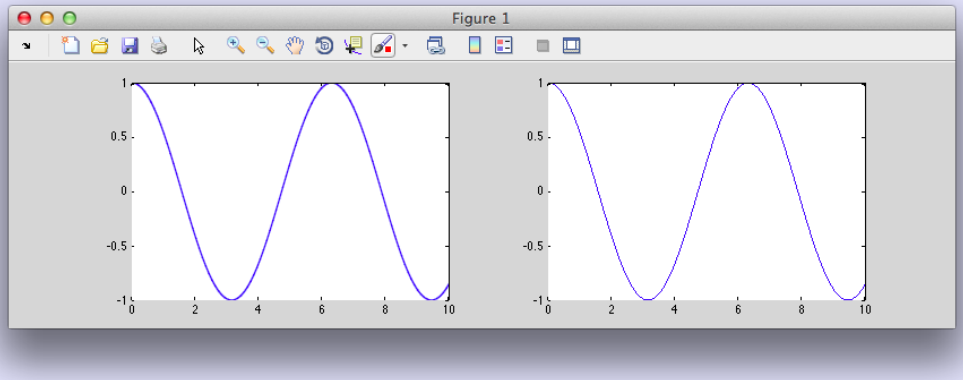
Example 2.5. Set lineseries properties.

```
close all;clear;
x = linspace(0,10,100);
y = cos(x);
hPlot = plot(x,y,'Color',[.3 .6 .9]); %hPlot is a lineseries handle
                        %  3-component color specifies RGB


set(hPlot,'LineWidth',3)
```



Example 2.6. This one is undocumented, and bug-riddled, but might be something to try if you want antialiased lines:

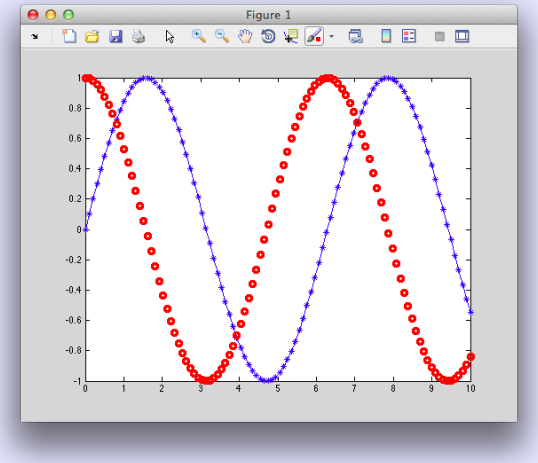
```
subplot(1,2,1)
plot(x,y,'LineSmoothing','on')
subplot(1,2,2)
plot(x,y)
```



One may plot multiple data sets in a single axes.

Example 2.7. Plot multiple data sets.

```
close all; clear; clc
x = linspace(0,10,100);
y1 = sin(x);
y2 = cos(x);
plot(x,y1,'*-')
hold on  % without this, new plot will clear the axes
plot(x,y2,'ro','LineWidth',2)
```

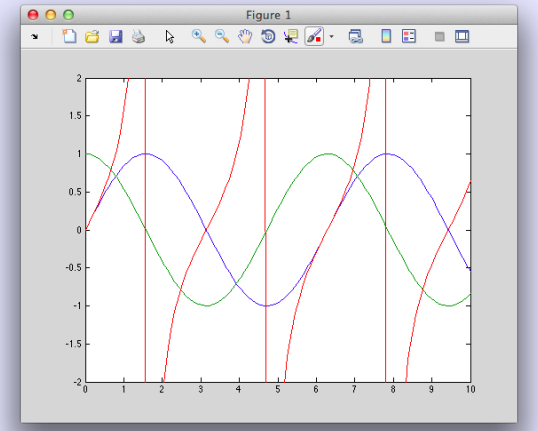


`hold` works with the `'NextPlot'` property of the current figure

```
set(gcf,'NextPlot','add'); % same as hold on
set(gcf,'NextPlot','replace'); % same as hold off
```

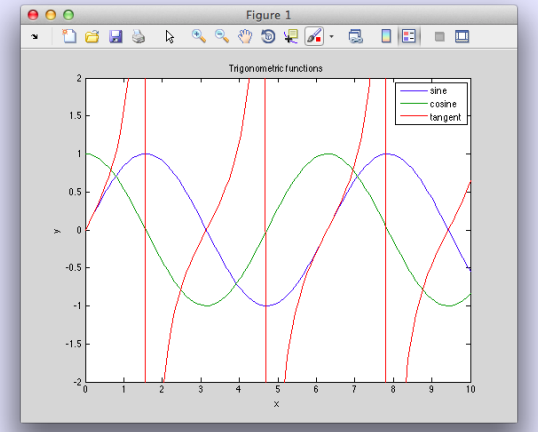
One may also plot multiple data sets using a single `plot` command. This time the `y` data is a matrix. Note that `plot` reads the data *columnwise*.

```
close all; clear; clc
x = linspace(0,10,100);
y1 = sin(x); y2 = cos(x); y3 = tan(x);
plot(x',[y1',y2',y3'])
ylim([-2,2])  % display only -2 < y < 2
```



Add labels:

```
xlabel('x')
ylabel('y')
title('Trigonometric functions')
legend('sine','cosine','tangent')
```



Example 2.8. `hold all` keeps the color and linestyle from resetting

```
figure
hold on
plot(cos(1:10)); %blue
plot(sin(1:10)); %also blue

figure
hold all
plot(cos(1:10)); %blue
```

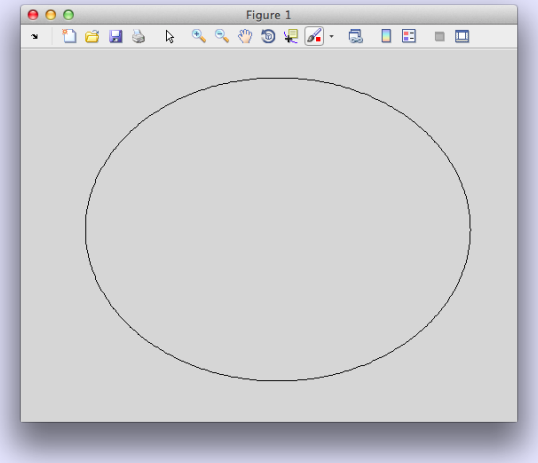
```
plot(sin(1:10)); %green
```

We saw `ylim` command that adjust the frame limit of the axes in Example 2.7. Related commands:

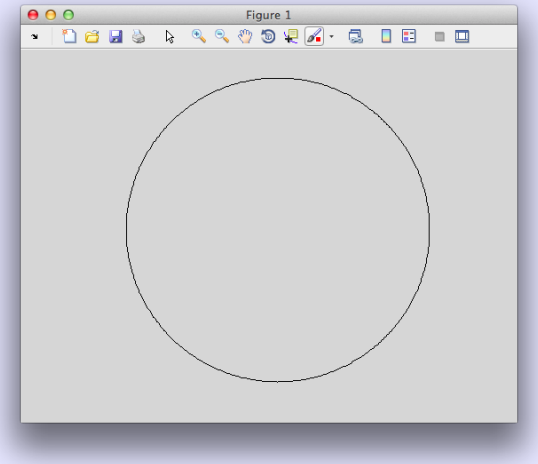
```
xlim([0 10])
% axis (not to be confused with axes)
axis([0 10 -2 2])
[xmin, xmax, ymin, ymax] = axis;
axis off % axis('off')
axis on
axis equal
axis square
help axis
axis xy
axis ij
axis image
axis normal
```

Example 2.9. Plot a circle

```
close all; clear; clc
t = linspace(0,2*pi,500);
x = cos(t);
y = sin(t);
plot(x,y,'k-')
axis off
```




```
axis equal
```

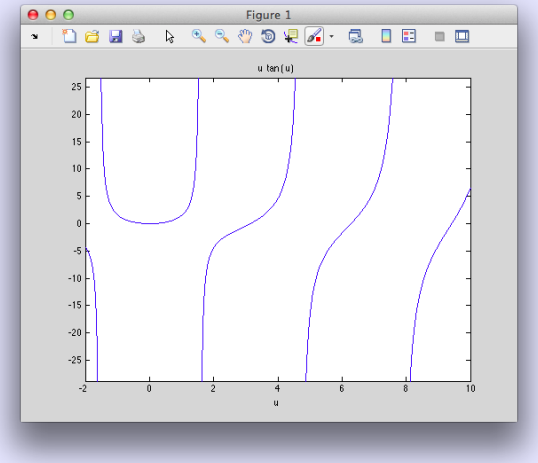


2.3.2 The `ezplot` family

The `ezplot` (easy-to-use function plotter) takes a *function handle* as input instead of data sets.

Example 2.10. The `ezplot` command plots a given function and labels your graph.

```
figure()
ezplot(@(u) u.*tan(u),[-2,10])
```



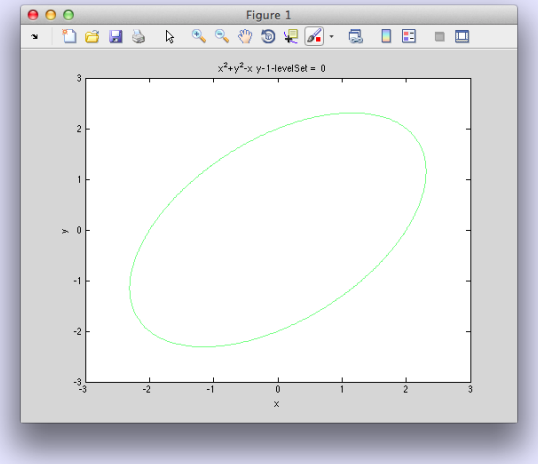
Note that `ezplot` usually takes care of the singularity (for example, it removes the vertical lines).

You can also plot implicitly defined functions using `ezplot` – if the argument is a function f of x and y, it will assume you want to plot $f(x, y) = 0$; the limits a, b apply to both x and y (you

can also specify `[xmin xmax ymin ymax]`)

Example 2.11. Plot a levelset of a function using `ezplot`.

```
figure()
levelSet = 3
ezplot(@(x,y) x.^2+y.^2 - x.*y-1 - levelSet, [-3,3])
```

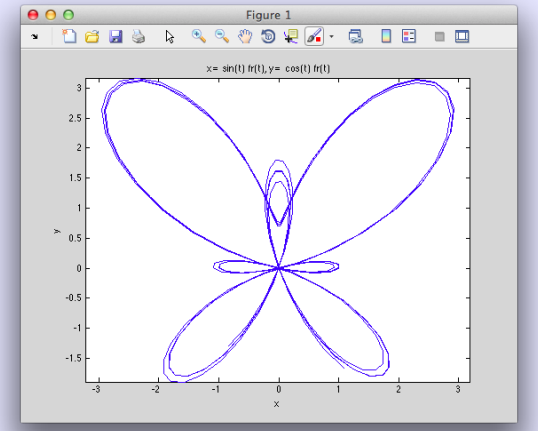


You can also plot a parametrized curve using `ezplot`.

Example 2.12. Plot

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} r(t)\sin(t) \\ r(t)\cos(t) \end{bmatrix}, \quad r(t) = e^{\cos(t)} - 2\cos(4t) - \sin^5(\tfrac{t}{12})$$

```
figure()
fr = @(t) exp(cos(t)) - 2*cos(4*t) - sin(t/12).^5;
fx = @(t) sin(t).*fr(t);
fy = @(t) cos(t).*fr(t);

ezplot(fx, fy, [-10,10])
```

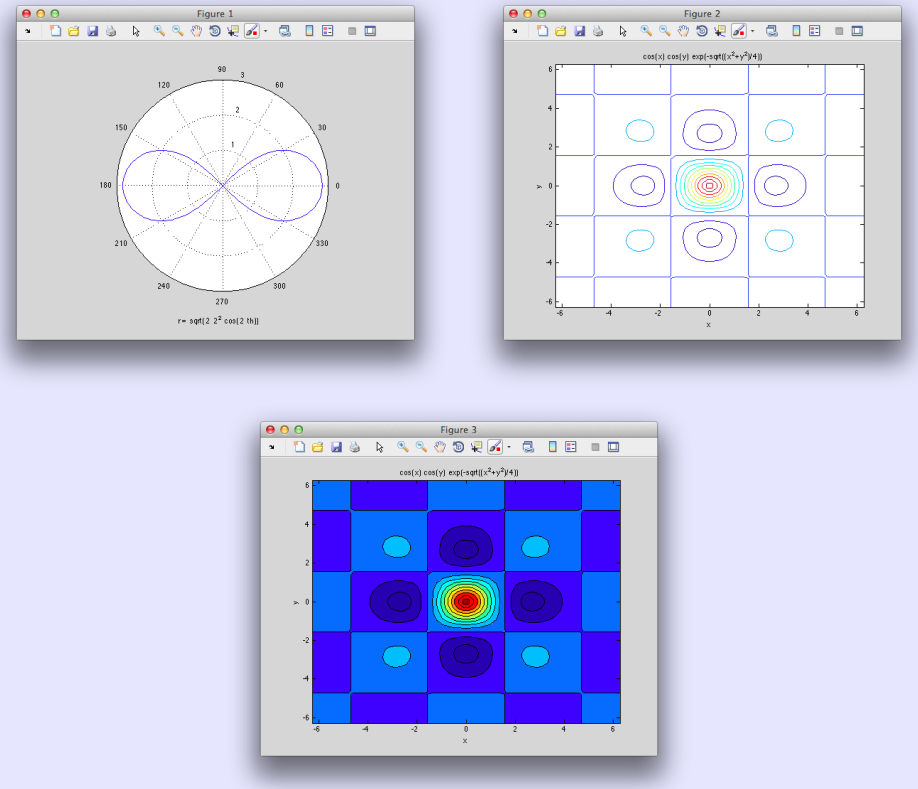


Note the low quality; we can't specify the number of points to use.

There are other `ez` plotters such as `ezpolar` and `ezcontour`

Example 2.13. Examples using `ezpolar` and `ezcontour`.

```
figure()
ezpolar(@(th) sqrt(2*2^2*cos(2*th)))

figure()
ezcontour(@(x,y) cos(x)*cos(y)*exp(-sqrt((x^2+y^2)/4)))
figure()
ezcontourf(@(x,y) cos(x)*cos(y)*exp(-sqrt((x^2+y^2)/4)))
```

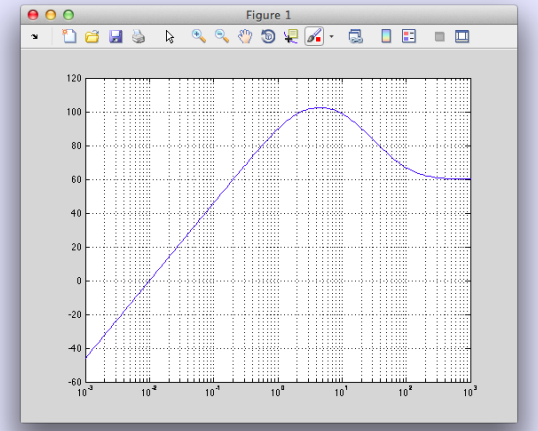


2.3.3 Other common plot commands

Log scale plots

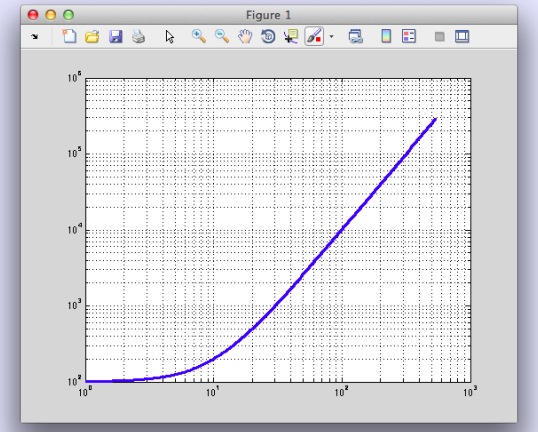
Example 2.14. The `semilogx` (resp. `semilogy`) creates plots with x- (y-)axis being log-scaled.

```
H = @(w) 20*(1i*w).*(1i*w + 100)./((1i*w+2).*(1i*w+10));
w = logspace(-3, 3, 200);
semilogx(w, 20 * log(abs(H(w)))); grid
```



The `loglog` command creates plot with both x and y axis log scaled.

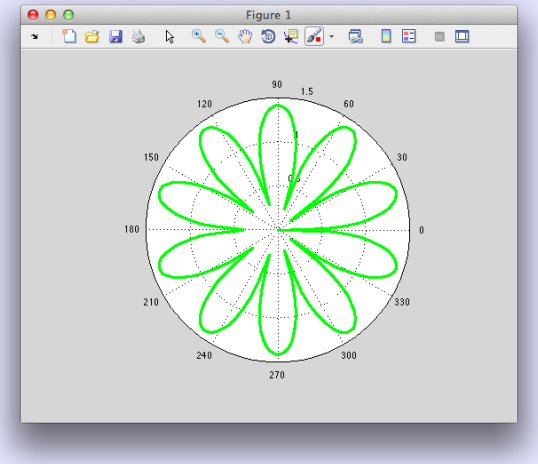
```
t = linspace(0, 2*pi, 200);
x = exp(t);
y = 100 + exp(2*t);
loglog(x,y, 'LineWidth', 2); grid
```



Polar plots

Example 2.15. Polar plot.

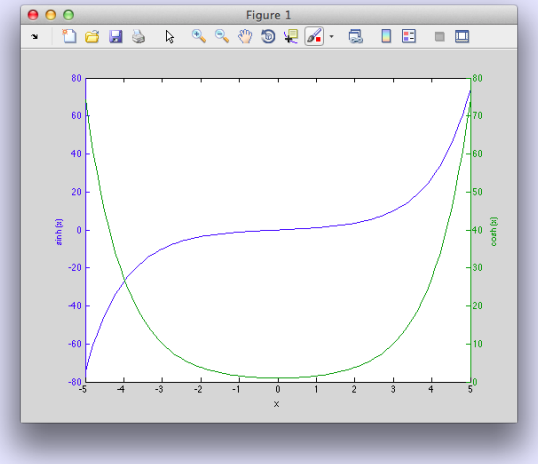
```
t = linspace(0, 2*pi, 200);
r = sqrt(abs(2*sin(5*t)));
p = polar(t,r, 'g');
set(p, 'LineWidth', 2);
```



Two-in-one axes

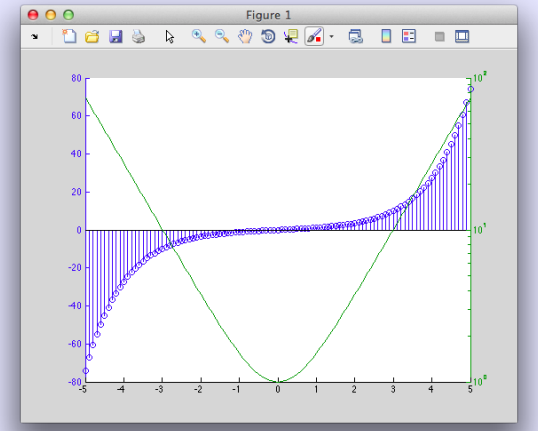
Example 2.16. `plotyy` manages a pair of axes for you.

```
x = -5:.1:5;
y1 = sinh(x);
y2 = cosh(x);
ax = plotyy(x, y1, x, y2);
xlabel('x');
hy1 = get(ax(1), 'ylabel');
hy2 = get(ax(2), 'ylabel');
set(hy1, 'string', 'sinh(x)');
set(hy2, 'string', 'cosh(x)');
```



`plotyy` with different scalings/plot commands:

```
plotyy(x,y1,x,y2,@stem,@semilogy)
```



See also

See also other 2D plot functions:

```
bar, barh, hist, histc  % bar charts, histograms
area,   % filled area plot
stairs, % stairstep graph
pie,    % pie chart
stem,   % discrete sequence data
compass, % arrows emanating from origin
comet,  % animated comet plot
contour, pcolor % contour plot, pseudocolor plot of a matrix (2D scalar data)
quiver, % arrow / vector field plot
```

2.4 Images

Another extensively used 2D visualization of data is *image*. Image could come from an actual image file, a processed image by some of your MATLAB program, or maybe a matrix of data. Here we introduce a few simple commands related to images.

2.4.1 Read images

Suppose `myImageFile.jpg` is an image file under your current working folder (other image extensions `.bmp`, `.cur`, `.gif`, `.hdf4`, `.ico`, `.jpg`, `.jp2`, `.pbm`, `.pcx`, `.pgm`, `.png`, `.ppm`, `.ras`, `.tiff`, `.xwd` work as well). Use the command `imread` to read the image:

```
A = imread('myImageFile.jpg');
```

Now, in the typical situation, `A` is a 3-dimensional array (like matrix, but matrix is 2-dimensional array) that stores the intensity of the image for each pixel and each color channel.

```
>> whos A
  Name         Size                    Bytes  Class    Attributes
```

```
  A         1001x1419x3            4261257  uint8
```

The third dimension of the array `A` indicates the RGB color channels. (Some image file may use CMYK color channels or RGBA channels supporting opacity/transparency). For example `A(:,:,1)` is the matrix of which the `(i,j)` element indicates the intensity of the color "red" at pixel `(i,j)`. Note that the `class` of `A` is `uint8`, an "8-bit unsigned integer", so you will observe that each entry of `A` only take integer value ranging from 0 to 255.

2.4.2 Show images

Suppose we have an image `A` (read from an image file using `imread` command mentioned above), one may view the image by

```
imshow(A)
```

or

```
image(A)
```

or

```
imagesc(A)
```

Both `imshow` and `image` display the color of the image using the standard scale: for `A` of class `uint8`, 0 is darkest and 255 is brightest; for `A` of class `double`, 0 is darkest and 1 is brightest. On the other hand `imagesc` displays the color that is scaled flexibly; it is usually used for visualizing a matrix of class `double` as an image.

In many image processing experiments, it is handier if `A` is just a matrix (instead of 3-channel array) of class `double`. To turn `A` into such form,

```
B = rgb2gray(A);
C = double(B)/255;
```

`B` will be a matrix (1-channel indicating grayscale). `C` will be of class `double` ranging from 0 to 1. It is notable that `rgb2gray` does a better job than just averaging the intensity of the 3 channel; it takes account of the human sensation of vision; for example yellow (RGB=(1,1,0)) looks brighter than magenta (RGB=(1,0,1)):

```
>> rgb2gray(reshape([1,1,0],1,1,3))
ans =
    0.8860
>> rgb2gray(reshape([1,0,1],1,1,3))
ans =
    0.4130
```

You may notice that `image` or `imagesc` show grayscale images in an unusual colormap: it maps numbers to colors that is not a list of gray colors ranging from black to white. This can be set by the `colormap` command.

```
imagesc(C)
colormap gray
```

`docsearch colormap` for more options of built-in color-maps, and for defining customized color-maps.

2.4.3 Write images

Use

```
imwrite(C,'myImage2.jpg')
```

to export a matrix `C` as an image file.

2.5 Create Animation

To create an animation, put your plot commands in a loop. Note that you will need the `drawnow` command in each iteration to update the figure window.

Example 2.17. Animate the time-dependent function

$$u(x,t) = \frac{1}{\sqrt{4\pi t}} e^{-x^2/(4t)}$$

(this is the *heat kernel*).

```
u = @(x,t) 1/sqrt(4*pi*t).*exp(-x.^2/(4*t));
t = 0.01;
x = linspace(-3,3,100);
while t<1
    hPlot = plot(x,u(x,t));
    set(hPlot,'LineWidth',2);
    ylim([-1,3]);

    drawnow
    t = t+0.01;
end
```

2.6 Enhance Performance

2.6.1 Vectorization v.s. For loops

You may have noticed that we commented "this function supports the input argument being an array" in both Example 2.1 and Example 2.2. This is the concept of "vectorization".

In the general sense, a function or a program is said to be *vectorized* if it operates on an entire array of data instead of processing a single value of the array N times.

Many CPUs can apply the same operation simultaneously to (say) four (or more) pieces of data. A program is truly *vectorized* if it processes a few pieces of data of an array *simultaneously*. Luckily, most MATLAB built-in functions are genuinely vectorized. It can make a big difference in performance between a vectorized program and a program full of loops.

To test performance, use the pair of commands `tic` and `toc`. The `toc` command returns the time the machine spent running commands between `tic` and `toc`.

Example 2.18. Here is an example evaluating a MATLAB built-in function on a large array.

```
x = rand(100000,1);
y = zeros(100000,1); % <- Preallocation. Even slower without this line
tic
for k = 1:100000
    y(k) = log10(x(k));
end
toc
```

```
Elapsed time is 0.351050 seconds.
```

A vectorized program has the loops replaced by a simultaneous (parallel) evaluation.

```
tic
y = log10(x);
toc
```

```
Elapsed time is 0.002357 seconds.
```

2.6.2 The `bsxfun` function

There are programs whose vectorization is not as obvious as Example 2.18. It is usually a fun challenge to come up with a clever code that is fully vectorized. The `bsxfun` (binary singleton expansion function) is a useful MATLAB built-in function in many situations.

To explain the particular type of situations, let us look at an example.

Example 2.19. Let V be an $N \times k$ matrix representing N number of $\mathbb{R}^k$-vectors

$$V = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1k} \\ v_{21} & v_{22} & \cdots & v_{2k} \\ & & \vdots & \\ v_{N1} & v_{N2} & \cdots & v_{Nk} \end{bmatrix}.$$

Normalize each row of V. That is, find the $N \times k$ matrix U such that each row of U

$$U\texttt{(j,:)} = \frac{\begin{bmatrix} v_{j1} & v_{j2} & \cdots & v_{jk} \end{bmatrix}}{\sqrt{v_{j1}^2 + v_{j2}^2 + \cdots + v_{jk}^2}}.$$

Assume that $v_{j1}^2 + v_{j2}^2 + \cdots + v_{jk}^2 \neq 0$ for all j.

The first attempt is to write a loop that visit each row of V.

Solution (using for loop).

normalize.m

```
function U = normalize(V)
% NORMALIZE normalizes an input vector V.
% The j-th row of U is the normalized j-th row of V.

[N,k] = size(V);
U = zeros(N,k);
for j = 1 : N
    U(j,:) = V(j,:)/sqrt(sum(V(j,:).^2));
end
```

```
>> V = rand(10000,5000);
>> tic; U = normalize(V); toc
Elapsed time is 5.217463 seconds.
```

■

Now we attempt to replace the for loop by a few built-in functions. We may vectorize the computation of the norms of rows of V.

Solution (attempt to vectorize).

normalize.m

```
function U = normalize(V)
norm_of_V = sqrt(sum(V.^2, 2));
U = V./norm_of_V;
```

This program does *not* work. The variable `V` is an N-by-k matrix while `norm_of_V` is an N-by-1 matrix, so we cannot perform the `./` (`rdivide`) operation in the last line.

One may fix this problem by creating k copies of `norm_of_V`:

normalize.m

```
function U = normalize(V)
[~,k] = size(V);
norm_of_V = sqrt(sum(V.^2, 2));
U = V./repmat(norm_of_V, 1,k);
```

```
>> tic; U = normalize(V); toc
Elapsed time is 1.534011 seconds.
```

■

A better way is to use the `bsxfun` function, which avoids the redundant data copying.

Solution (vectorization using `bsxfun`).

normalize.m

```
function U = normalize(V)
norm_of_V = sqrt(sum(V.^2, 2));
U = bsxfun(@rdivide,V,norm_of_V);
```

```
>> tic; U = normalize(V); toc
Elapsed time is 1.015402 seconds.
```

■

The syntax of `bsxfun` is

$$C = \texttt{bsxfun}(F, A, B)$$

where F is a function handle that takes two inputs. Suppose A, B are matrices of size m_A-by-n_A and m_B-by-n_B respectively. Then the output matrix C has its element

$$C_{(i,j)} = \begin{cases} F(A_{(i,j)}, B_{(i,j)}) & \text{when } m_A = m_B, \text{ and } n_A = n_B \\ F(A_{(i,j)}, B_{(i,1)}) & \text{when } m_A = m_B, \text{ and } n_B = 1 \\ F(A_{(i,j)}, B_{(1,j)}) & \text{when } m_A = m_B, \text{ and } n_A = 1 \\ F(A_{(i,1)}, B_{(i,j)}) & \text{when } m_B = 1, \text{ and } n_A = n_B \\ F(A_{(1,j)}, B_{(i,j)}) & \text{when } m_A = 1, \text{ and } n_A = n_B \\ F(A_{(i,1)}, B_{(1,j)}) & \text{when } m_B = 1, \text{ and } n_A = 1 \\ F(A_{(1,j)}, B_{(i,1)}) & \text{when } m_A = 1, \text{ and } n_B = 1 \end{cases}$$

In other words, when some dimensions of the input matrices A, B are *singleton* (size be 1), `bsxfun` effectively expands the singletons in evaluation of $F(A, B)$.

2.6.3 Preallocation

There are loops in a program that is unlikely to be replaced by a parallel evaluation (vectorized). In such cases be sure that you *preallocate* variables to enhance performance.

Example 2.20. Find the array $(x_1, \ldots, x_N)$ that satisfy

$$\begin{aligned} x_{n+1} &= 3.9\, x_n(1 - x_n) \text{ for } n = 1, \ldots, N-1 \\ x_1 &= 0.5. \end{aligned}$$

There is no analytic solution to this difference equation, hence you cannot plug `1:N` to a formula and get all $x_1, \ldots, x_N$ at once.

Side note. The coefficient "3.9" is picked close to and below 4. In this case the resulting sequence $x_1, \ldots, x_N$ is *chaotic*.

For such example, one has to use a for loop to evaluate each entry of the array

```
for k = 1:N
    x(k+1) = f(x(k));
end
```

where `f = @(x) 3.9*x*(1-x)`.

(Without preallocation).

```
clear
N = 10000;
f = @(x) 3.9*x*(1-x);

tic
x(1) = 0.5;
for k = 1:N-1
    x(k+1) = f(x(k));
end
toc
```

```
Elapsed time is 0.032758 seconds.
```

Before the iteration, MATLAB only sees `x(1) = 0.5` and has no idea how much space of memory should be reserved for the array `x`. When MATLAB needs to store number to `x(k+1)`, which is an unexpected entry, it creates another vector of a longer length and copy each value of `x(1:k)` to the new space.

Roughly speaking of number of operations the machine has to do, the original $O(N)$ complexity becomes $O(N^2)$.

■

(With preallocation).

```
clear
N = 10000;
f = @(x) 3.9*x*(1-x);

tic
x = zeros(1,N); % <- added this line
x(1) = 0.5;
for k = 1:N-1
    x(k+1) = f(x(k));
end
toc
```

```
Elapsed time is 0.007758 seconds.
```

■

2.6.4 Profiling

To spy the performance of your program, you may use the *profiling* tool MATLAB provides (instead of having lots of `tic`'s and `toc`'s). MATLAB profiling tool profiles execution time for functions, from the report of which you know what part of your program spent most of the running time.

Basic commands:

```
profile on
% run some program here

profile off
profile viewer
```

2.7 3D Plot – Curves

Generalizing from plane curve to space curve is direct. To create a figure of plane curves we use `plot`. To create a 3D space curve we use `plot3`.

$$\texttt{plot3(}x\texttt{,}y\texttt{,}z\texttt{,...)}$$

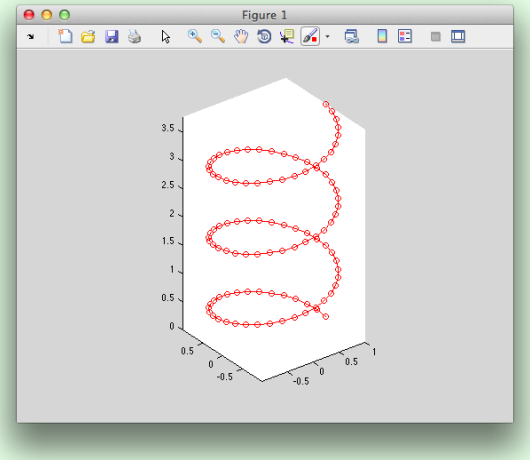
where x, y, z are vectors of same length which describe a list of 3D coordinates. When the adjacent points of the list are close to each other, the 3D plot resembles a continuous curve. You can also visualize list points that scatters without order; in that case you may set the `'LineStyle'` property to be `'none'`.

Example 2.21 (Helix)**.** Plot the following parametrized curve

$$\begin{cases} x(t) = \cos(t) \\ y(t) = \sin(t) \\ z(t) = 0.2\ t \end{cases} \qquad 0 \le t \le 6\pi.$$

Solution.

```
t = linspace(0,6*pi,100);
x = cos(t);
y = sin(t);
z = 0.2*t;

figure
plot3(x,y,z,'ro-')
axis equal
```



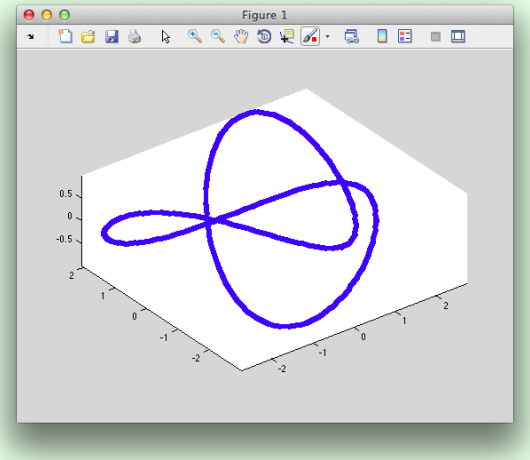
■

Example 2.22 (Trefoil knot)**.** Plot

$$\begin{cases} x(t) = \sin(t) + 2\sin(2t) \\ y(t) = \cos(t) - 2\cos(2t) \\ z(t) = -\sin(3t) \end{cases} \qquad 0 \leq t \leq 2\pi$$

Solution.

```
t = linspace(0,2*pi,200);
x = sin(t) + 2 * sin(2*t);
y = cos(t) - 2*cos(2*t);
z = -sin(3*t);

figure

plot3(x,y,z,'LineWidth',5)
axis equal
```



■

2.8 3D Plot – Surfaces

The basic command for plotting a surface is `surf`:

$$\texttt{surf}(X,Y,Z,C)$$

where X, Y, Z and C are matrices of same size, which specify a table of 3D coordinates and their color. The matrix specifying color C is a real-valued matrix (as oppose to RGB) so one needs to control the colormap to tune the displayed color. Here the argument C is optional.

To create a table of cartesian coordinate, you will find `meshgrid` function very useful.

The created surface object has properties that can be found with

```
docsearch surface properties
```

The properties often used are

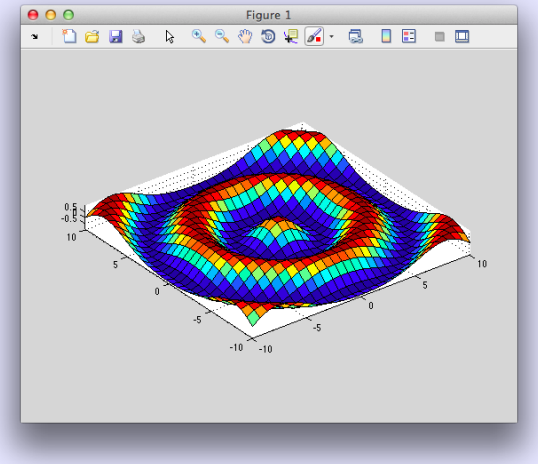
- `'XData'`, `'YData'`, `'ZData'`, `'CData'`. Their values are the X, Y, Z and C matrices. Setting the values of these properties updates the shape and the color of the surface object without creating a new surface.

- `'EdgeColor'`, `'FaceColor'`. The color of the edges and the faces. The value can be `'none'`, set to which the edge or face will be invisible. The value can be a `ColorSpec` code; for example `'k'` is black and `'b'` is blue. The value can also be a 3-vector specifying RGB. The value can also be `'flat'` or `'interp'`, which will display the edge color as the nearby value of C of `'CData'`. (`'flat'` gives piecewise constant color and `'interp'` gives piecewise linear or other interpolation of color depending on the `'Renderer'` property of the current figure `gcf`).

Example 2.23. In this example we plot the graph of the multivariate function

$$f(x,y) = \cos\left(\sqrt{x^2+y^2}\right)$$

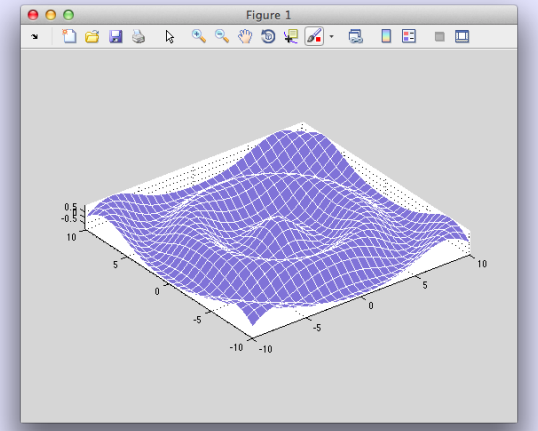
```
f = @(x,y) cos(sqrt(x.^2 + y.^2));

x = linspace(-10,10,30);
y = linspace(-10,10,30);
[X,Y] = meshgrid(x,y); % X,Y are matrices, X(i,j)=x(j), Y(i,j)=y(i).

hs = surf(X,Y,f(X,Y)); % Think of it, (X,Y,f(X,Y)) gives exactly the
                       % table of the desired X,Y,Z data
                    %  Here, hs is the handle to the surface object
axis equal
```



You can observe that the default `'FaceColor'` is `'flat'`, `'EdgeColor'` is `'k'` and `'CData'` is the same as `'ZData'`. Let us modify these properties.

The following code can modify the surface to have arbitrary plain color.

```
set(hs,'FaceColor',[0.4,0.4,0.8])
set(hs,'EdgeColor','w')
```



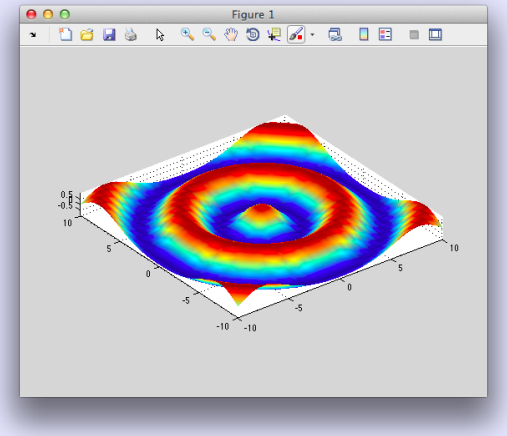
The following code makes the face color be continuously interpolated `'CData'`.

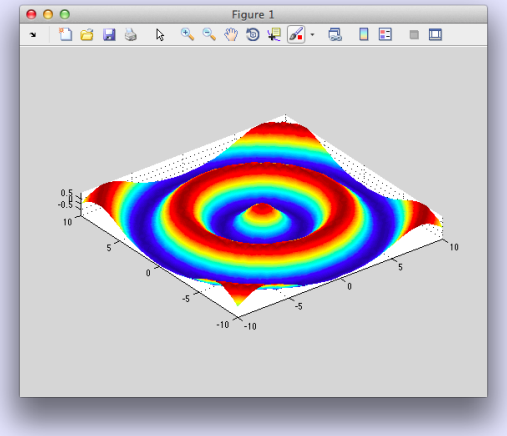
```
set(hs,'FaceColor','interp') % interpolate face color
set(hs,'EdgeColor','none')   % show no edge
```

which will produce the following (left) figure. The default renderer of a figure is `'opengl'` (see `get(gcf,'Renderer')`), which interpolates color piecewise linearly. This interpolation scheme creates artifact when the grid is not fine enough and the desired color gradient does not align with the direction of the grid. The `'zbuffer'` renderer does a better job in interpolating colors in this particular example:

```
set(gcf,'Renderer','zbuffer')
```

which produces the following (right) figure.





Let's play with a more creative `'CData'`. Let us define $C_{ij} = \Big(\lfloor i/m \rfloor + \lfloor j/m \rfloor\Big) \mod 2$, which is a checkerboard pattern with block size m.

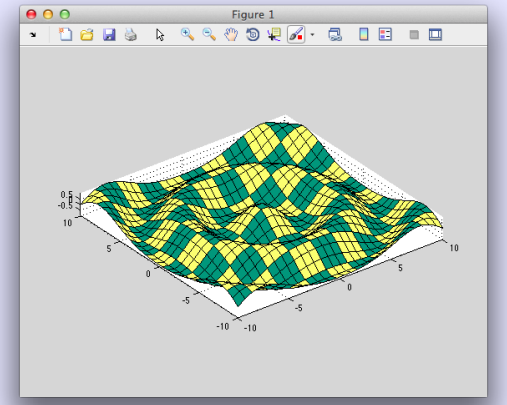
```
ckb_fun = @(i,j) mod(floor(i/3) + floor(j/3),2);
C = bsxfun(ckb_fun, 1:30, (1:30)');
set(hs,'CData',C)
set(hs,'FaceColor','flat','EdgeColor','k')
colormap summer
```

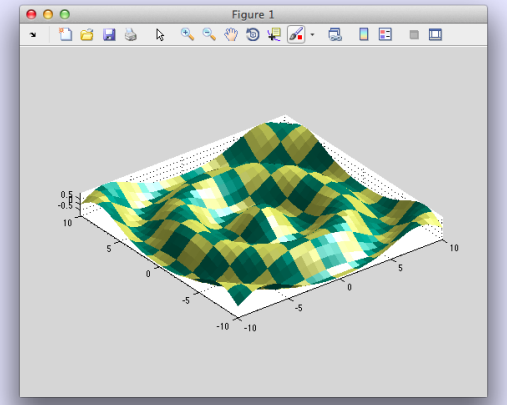
One may put light by the command

```
light
```

Let us turn off the edge color again

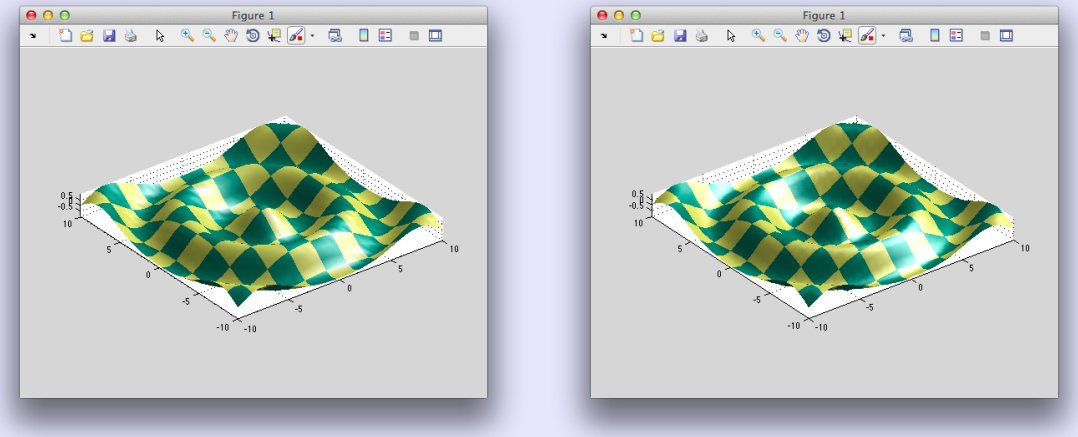
```
set(hs,'EdgeColor','none')
```





There is a `'FaceLighting'` property for each surface object. The value of it can be `'none'` (no lighting), `'flat'` (with effect shown above), `'gouraud'` (default for interpolated shading) and `'phong'` (giving the best lighting).

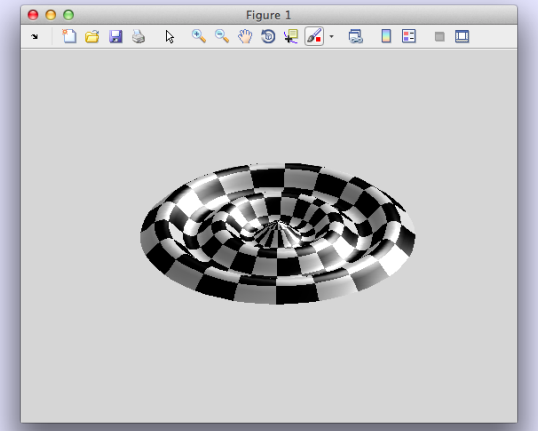
```
set(hs,'FaceLighting','phong') %  (left)
set(hs,'FaceLighting','gouraud') %  (right)
```



The `'XData'` and `'YData'` do not need to be a cartesian coordinate. You can parameterize the surface using polar coordinate as shown in the following example.

Example 2.24. We plot the same surface as the previous example, but using the polar coordinate. Now $f_{\text{polar}}(r,\theta) = \cos(r)$.

```
clear
close all
clc

f_polar = @(r,theta) cos(r);

N_r = 30; N_t = 60;  % resolution in radial and angular
r = linspace(0,15,N_r);    % radial
t = linspace(0,2*pi,N_t); % angular
[R,T] = meshgrid(r,t);    % polar coordinate

% create the surface object
hs = surf(R.*cos(T),R.*sin(T),f_polar(R,T));
axis equal

% put a checkerboard pattern
ckb_fun = @(i,j) mod(floor(i/3) + floor(j/3),2);
C = bsxfun(ckb_fun, 1:N_r, (1:N_t)');
set(hs,'CData',C)

% colormap and lighting
colormap gray

light
set(hs,'FaceColor','flat','EdgeColor','none')
set(hs,'FaceLighting','phong')

% remove the frame
axis off
```



2.9 Camera in a 3D Plot

You can set the camera angle by the command

```
view(10,40)
```

where the two arguments are the azimuth and elevation.

You can use the `rotate3d` tool by clicking the corresponding icon on the toolbar of the figure window or execute the command

```
rotate3d
```

However, the default `rotate3d` is not the best tool for 3D navigation. You will find

```
cameratoolbar
```

pretty useful.

Chapter 3

Linear Algebra with MATLAB

The two most fundamental problems in the study of numerical linear algebra are solving linear systems

$$Ax = b$$

and solving eigenvalue problems

$$Ax_j = \lambda_j x_j.$$

Linear algebra theory provides useful knowledge about general properties and criteria of existence of solutions to linear problems, as well as basic procedures if you were asked to solve these linear problems by hand. To name a few examples, *Gaussian elimination* can be used to solve a linear system $Ax = b$, *Cramer's formula* can invert a matrix, *Gram-Schmidt process* finds an orthonormal basis for a QR factorization, and finding the roots of the *characteristic polynomial* gives the eigenvalues of a matrix. These algorithms are often referred as the *direct methods*. They are often the proof of existence of the solutions themselves, and they provide the exact solution (if exist) in a finite number of procedural steps.

However, many of these direct method algorithms are not suitable for solving problems with large matrices using computer. For example, the Gaussian elimination procedure generally requires number of operations proportional to n^3 where a matrix of size $n \times n$. That is very inefficient comparing to $O(n)$ algorithms (usually the ultimate goal for linear problems). In addition, some direct methods (like the Gram-Schmidt process) give final results that are very sensitive to the numerical value of each step; this can yield undesirable result when computing on machines with finite precision.

In the study of numerical linear algebra people developed much more efficient algorithms for solving linear systems and eigenvalue problems. Many of them are *iterative methods*, as oppose to the direct methods, which computes the *approximated* solution as a sequence that robustly and efficiently converge to the true solution. The good news is that MATLAB has included algorithms of the state of the art.

3.1 Linear System

Let A be an m-by-n matrix and b be an m-by-1 vector. The problem is to find an n-by-1 vector such that $Ax = b$.

In most cases, this problem can be solved by MATLAB using the backslash operator (`mldivide`)

```
x = A\b;
```

If $m = n$ and A is invertible, then `x=A\b` computes roughly the same result as `x = inv(A)*b`, except that it is much more efficient.

If $m = n$ but A is singular or close to singular, a warning message will appear. In that case one either reformulates the problem (adding more auxiliary linear conditions to x to get a different linear system whose matrix is full-rank), or use `pinv` (pseudoinverse) instead:

```
x = pinv(A)*b;
```

which returns a *least squares solution*, which is a minimizer x of the Euclidean distance of Ax and b. Note that in this case, the direct call of `x = A\b` is not recommended.

If $m \neq n$ and you are looking for a least squares solution, go ahead and use

```
x = A\b;
```

When the rank k of `A` is n (that is `A` has full column-rank), then the least squares solution is unique, and the result of `x = A\b` agrees with `x = pinv(A)*b`. When the rank k of `A` is strictly smaller than n, there are infinitely (of dimension $(n-k)$) many `x` that are least squares solution. In this case, `x = pinv(A)*b` gives the one that `x` has the least norm, while `x = A\b` gives an `x` with only k nonzero elements.

See

```
doc mldivide
```

for excellent examples.

3.1.1 Related commands

The determinant and the trace of a matrix `A`:

```
det(A)
trace(A)
```

The operator norm of a matrix, which is defined as

$$\|A\|_p := \sup_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}, \quad \|x\|_p := \left(\sum_{j=1}^{n} |x_j|^p \right)^{1/p} \qquad (1 \leq p \leq \infty)$$

is computed by

```
norm(A,p)
```

or, when $p = 2$,

```
norm(A)
```

The condition number `cond(A,p)`, defined as $\|A\|_p \|A^{-1}\|_p$, can be computed with

```
cond(A,p)
```

(`cond(A)` is the same as `cond(A,2)`).

The Frobenius norm

$$\|A\|_F := \sqrt{\sum_{ij} |A_{ij}|^2}$$

can be computed with

```
norm(A,'fro')
```

If an LU-factorization $A = LU$ (or $PA = LU$, ...) is ever needed, `lu` is the command to search.

3.1.2 QR factorization

Let A be an m-by-n matrix whose column vectors are $\{a_j \in \mathbb{C}^m\}, j = 1, \ldots, n$. The QR factorization of A is the form

$$A = QR$$

where R is m-by-n upper triangular matrix ($R_{ij} = 0$ whenever $i > j$) and Q is m-by-m unitary matrix satisfying

$$\overline{Q}^T Q = Q\overline{Q}^T = I$$

meaning that columns $q_j \in \mathbb{C}^m$ of Q form an orthonormal (in the complex sense) basis of the vector space $\mathbb{C}^m$ (over the scalar element $\mathbb{C}$).

Here all "$\mathbb{C}$"'s can be replaced by "$\mathbb{R}$".

Note that for any $k \leq n$, the first k columns of A spans the same space as that spanned by the first k columns of Q. When $m > n$, the "extra" columns $q_{n+1}, \ldots, q_m$ are any orthonormal vectors that are orthogonal to the first n vectors $q_1, \ldots, q_n$ (or $a_1, \ldots, a_n$).

The QR factorization can be computed with

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

An alternative QR factorization is that Q is m-by-n (same size as A) and R is n-by-n square upper triangular matrix. To compute such *economic* QR factorization,

```
[Q,R] = qr(A,0);
```

Example 3.1 (Orthogonal complement)**.** Suppose $v = (1/\sqrt{2}, 1/\sqrt{2}, 0) \in \mathbb{R}^3$. Find the two vectors orthogonal to it.

Solution.

```
>> v = [1/sqrt(2), 1/sqrt(2) , 0]'

v =

    0.7071
    0.7071
         0

>> [Q,~] = qr(v)
```

```
Q =

   -0.7071   -0.7071         0
   -0.7071    0.7071         0
         0         0    1.0000

>> v2 = Q(:,2); v3 = Q(:,2); % the requested vectors
```

■

3.2 Eigenvalue Problems

Let A be an n-by-n square matrix. A complex number λ_j and a complex vector $x_j \in \mathbb{C}^n$ are a pair of *eigenvalue* and *eigenvector* if

$$Ax_j = \lambda_j x_j.$$

That is, the linear transformation $x \mapsto Ax$ is just a scaling in the eigenvector direction; eigenvectors are principal directions of the transformation. For a *diagonalizable* matrix, there are n eigenvalues $\lambda_1, \ldots, \lambda_n$ which correspond to n linearly independent eigenvectors $x_1, \ldots, x_n$. Concatenate $[x_1 x_2 \cdots x_n] =: V$ and $D := \texttt{diag}([\lambda_1, \ldots, \lambda_n])$, one has

$$AV = VD.$$

The matrix V of eigenvectors and D of eigenvalues can be computed by

```
[V,D] = eig(A);
```

If you need only the eigenvalues $d = (\lambda_1, \ldots, \lambda_n)^T$,

```
d = eig(A);
```

3.2.1 Symmetric Eigenvalue Problems

If A is symmetric (Hermitian, $\overline{A}^T = A$), A is always diagonalizable, eigenvalues are all real, and that V is unitary. Such eigenvalue problem is equivalent to finding the *critical points* $x_i \in \mathbb{C}^n$ of the quadratic function

$$q(x) = \overline{x}^T Ax \quad \text{with the constraint } \overline{x}^T x = 1.$$

You may check that such problem is formulated with method of Lagrange multiplier, which gives $\nabla q = 2Ax = \lambda \nabla(\overline{x}^T x) = 2\lambda x$, the eigenvalue problem.

If we replace the constraint $\overline{x}^T x = 1$ by $\overline{x}^T Bx = 1$ for some symmetric positive semi-definite matrix B, then the corresponding eigenvalue problem becomes

$$Ax = \lambda Bx$$

or

$$AV = BVD.$$

This is called the *generalized eigenvalue problem*, which can be computed by

```
[V,D] = eig(A,B);
```

Generalized eigenvalue problems usually arise as the above setup: B describes some (semi)norm that gives a notion of normalization ($\overline{x}^T Bx = 1$). But in general `[V,D] = eig(A,B)` takes arbitrary square `A` and `B` that do not need to be symmetric.

3.3 Singular Value Decomposition

The singular value decomposition (SVD) of an m-by-n matrix A is

$$A = U S \overline{V}^T$$

where U is an m-by-m unitary matrix (with columns $u_1, \dots, u_m$), V is an n-by-n unitary matrix (with columns $v_1, \dots, v_n$), and S is a diagonal m-by-n matrix taking the form

$$S = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_m \end{bmatrix}$$

(this is the case $m < n$) with *singular values* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 = \sigma_{r+1} = \cdots = \sigma_{m \text{ (or } n)}$. Note that r is the rank of A. SVD also takes the form

$$A = \sigma_1 u_1 \overline{v}_1^T + \sigma_2 u_2 \overline{v}_2^T + \cdots + \sigma_r u_r \overline{v}_r^T,$$

which means that A is a combination of rank-1 matrices $u_j \overline{v}_j^T$ written in a descendent order of "importance".

SVD becomes extremely important when r is small (low-rank) (or that σ_j drops rapidly many σ_j can be viewed zero) because $\{\sigma_1, \dots, \sigma_r, u_1 \dots, u_r, v_1, \dots, v_r\}$ is a much more efficient (less amount of data) way of expressing A. SVD is alternatively known as *principal component analysis* (PCA) in statistics. SVD are widely used in signal processing, pattern recognition, recommender system, quantum information, and many others.

In geometry and mechanics, SVD gives the polar decomposition

$$A = U S \overline{V}^T = (U \overline{V}^T)(V S \overline{V}^T) =: QY$$

a unitary Q and a positive semi-definite Y, which are the closest rotation to A (in the Frobenius norm) and the axial scaling.

In linear algebra SVD can be used to define the pseudo-inverse.

SVD can be viewed as a generalization of the eigenvalue decomposition for general asymmetric, non-square matrices. One may see that U, S, V in an SVD satisfy $(A\overline{A}^T)U = U(SS^T)$ and $(\overline{A}^T A)V = V(S^T S)$, which are standard symmetric eigenvalue problems.

Anyway, SVD can be computed in MATLAB with

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

3.4 Sparse Matrices

A matrix is regarded *sparse* if most of the entries are zero. Typically a sparse matrix only contain $O(n)$ (out of n^2) nonzero entries. For such matrices it is more efficient in storage and in basic operations that we store only the locations and values of the nonzero entries. MATLAB has such *sparse matrix* storage structure. As opposed to *sparse matrix*, an usual matrices is called a *full matrix*.

```
>> A = [2 0 0 0 ; 0 0 1 0 ; 0 pi 0 0 ; 0 0 0 0]

A =

    2.0000         0         0         0
         0         0    1.0000         0
         0    3.1416         0         0
         0         0         0         0

>> S = sparse(A)

S =

   (1,1)       2.0000
   (3,2)       3.1416
   (2,3)       1.0000

>> full(S)

ans =

    2.0000         0         0         0
         0         0    1.0000         0
         0    3.1416         0         0
         0         0         0         0

>> whos S
  Name      Size             Bytes  Class     Attributes

  S         4x4                 88  double    sparse

>> whos A
  Name      Size             Bytes  Class     Attributes

  A         4x4                128  double
```

From the above example you can see that `sparse(A)` turns a full matrix `A` to a sparse matrix, which only stores the values and the subscripted indices of the nonzero entries. The command `full(S)` turns a sparse matrix to a full matrix. Both `A` and `S` are of class `double`, which is what matrix elements take values in. Note that `S` has `sparse` in `Attributes`.

3.4.1 Operations

All MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices. Some more common functions for sparse matrices are listed below

```
spy(S)  % visualize the sparsity of S
full(S) % convert to a full matrix
issparse(S) % determine if S is sparse
nnz(S)  % number of nonzero elements
nonzeros(S) % returns the value of the nonzero elements
```

A few notes on the properties of sparse matrices:

- Sum (or difference) of two sparse matrices is sparse.
- Matrix product of two sparse matrices is sparse.
- Inverse of a sparse matrix is usually not sparse (but MATLAB still store the result as a sparse matrix)
- Matrix exponential (or other analytic function) of a sparse matrix is usually not sparse.

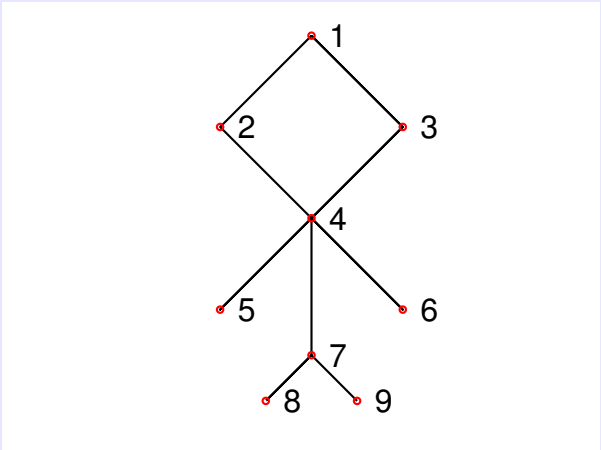
3.4.2 Creating a Sparse Matrix

Let $(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)$ be the location of the r nonzero entries of a sparse $m \times n$ matrix you want to construct, and let $v_1, \ldots, v_r$ be the values of those entries. Let $I = [i_1, i_2, \ldots, i_r]$, $J = [j_1, \ldots, j_r]$ and $V = [v_1, \ldots, v_r]$. Then the desired sparse matrix is constructed by

```
S = sparse(I,J,V,m,n);
```

Entry locations (i_k, j_k) may repeat. Elements of V that have duplicated values of I and J are added together. For example if $(i_2, j_2) = (i_5, j_5)$ then the resulting $S_{i_2,j_2} = v_2 + v_5$.

Example 3.2. Create the adjacency matrix of the following graph.



The adjacency A of graph is defined as

$$A_{ij} = \begin{cases} 1 & \text{node } i \text{ is connected with node } j \\ 0 & \text{otherwise} \end{cases}$$

Solution. Define a matrix `E` with two columns which denotes all edges. Each row of `E` contains the two node numbers that is connected by an edge:

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

The sparse adjacency matrix `A` is computed as

```
n = 9; % number of nodes
A = sparse(E(:,1),E(:,2),1,n,n);
A = A + A'; % A is symmetric (if A(i,j)==1 then A(j,i)=1)
```

The result of `A` is

```
>> full(A)
ans =

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

■

Other functions for constructing sparse matrices To create an all-zero, $m \times n$ sparse matrix,

```
A = sparse(m,n); % which is the same as A = sparse([],[],[],m,n);
```

To create the identity matrix

```
speye(m,n)  % 1's lie on the main diagonal
speye(n)    % same as speye(n,n)
```

The function `spdiags` can create band or diagonal matrices, which is useful for constructing, for example, tridiagonal sparse matrices.

3.5 Laplacian on a Regular Grid

The *Laplacian* or the *Laplace operator* $\mathcal{L}$ is a differential operator that widely appears in natural science and engineer, stochastic processes, networks, image processing, computer graphics,

mathematical study of differential equations, complex analysis, and many more. For a smooth (multi-variable) function $u(x, y)$,

$$(\mathcal{L}u)(x, y) = \frac{\partial^2}{\partial x^2}u(x, y) + \frac{\partial^2}{\partial y^2}u(x, y).$$

(Do not confuse with the Laplace transform, which is named after the same mathematician and often denoted by the letter $\mathcal{L}$ as well.) For a discrete function, an $m \times n$ matrix, $U : [1, \dots, m] \times [1, \dots, n] \to \mathbb{R}$, the result of it applied by the *standard discrete Laplacian* $\mathbf{L}$ is given by

$$(\mathbf{L}U)(i, j) = U(i + 1, j) + U(i - 1, j) + U(i, j + 1) + U(i, j - 1) - 4U(i, j).$$

One can see that $\mathbf{L}$ measures the difference between the "average of the immediate neighbors of (i, j)" and the value at (i, j).

For (i, j) on the boundary, $(i, j) = (1, j)$ for example, there are only three neighbors, hence we define

$$(\mathbf{L}U)(1, j) = U(2, j) + U(1, j + 1) + U(1, j - 1) - 3U(1, j).$$

Recall that in MATLAB $U(i)$ (with only one argument) uses the *linear indexing*

$$U(\texttt{sub2ind}([m, n], i, j)) = U(i, j).$$

Using linear indexing, we may view U as a vector, and $\mathbf{L}$ as a matrix multiplies on it. Suppose $i_1, \dots, i_4$ are the neighbors of the pixel i_0; then the i_0-th row of $\mathbf{L}U$ looks like

$$\begin{array}{c} \\ i_0 \to \end{array} \begin{array}{c} \begin{array}{ccccccccccc} & & i_1 & & & i_0 & & i_2 & i_3 & & i_4 \\ & & \downarrow & & & \downarrow & & \downarrow & \downarrow & & \downarrow \end{array} \\ \begin{bmatrix} \ddots & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & \ddots & & & & & & & & \\ & & 1 & & & -4 & & 1 & 1 & & 1 \\ & & & & & & \ddots & & & & \\ & & & & & & & \ddots & & & \\ & & & & & & & & \ddots & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & \ddots \end{bmatrix} \\ \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\mathbf{L}} \end{array} \begin{array}{c} \\ \begin{bmatrix} \vdots \\ U_{i_1} \\ \vdots \\ U_{i_0} \\ \vdots \\ U_{i_2} \\ \vdots \\ U_{i_3} \\ \vdots \\ U_{i_4} \end{bmatrix} \\ \underbrace{\qquad\qquad}_{U} \end{array}$$

In short, $\mathbf{L}$ is a matrix such that

$$\mathbf{L}_{i,j} = \begin{cases} 1 & \text{if pixel } i \text{ and } j \text{ (linear indexing) are neighbors} \\ 0 & \text{if pixel } i \text{ and } j \text{ are not neighbors, and } i \neq j \end{cases}$$

$$\mathbf{L}_{i,i} = -N, \text{ where } N \text{ is the number of 1's in the } i\text{-th row of } \mathbf{L}.$$

A code for generating $\mathbf{L}$ becomes straightforward.

Laplacian.m

```
function L = Laplacian(m,n)
% Discrete Laplacian for a regular m-by-n grid.
% The output L is an (m*n)-by-(m*n) sparse matrix.
%
% Using the linear indexing, if pixel i and pixel j are neighbors, then
% L(i,j) = 1.  The diagonal coefficients L(i,j) is chosen so that row sum
% of L is zero.  That is, L*ones(m*n,1) is an all-zero vector.

% Matrices I and J are defined so that I(i,j) = i and J(i,j) = j
% They will be pretty useful
[I,J] = ndgrid(1:m,1:n);

% indL and indR are matrices taking values of linear indices
% the corresponding term of indL and indR will refer to pixels that are
% horizontal neighbors
indL = sub2ind([m,n], I(:,1:end-1) , J(:,1:end-1) );
indR = sub2ind([m,n], I(:,2:end  ) , J(:,2:end  ) );

% indU and indD have corresponding term referring to vertically neighboring
% pixels
indU = sub2ind([m,n], I(1:end-1,:) , J(1:end-1,:) );
indD = sub2ind([m,n], I(2:end  ,:) , J(2:end  ,:) );

% Putting indL,..,indD together, corresponding entries of ind1 and ind2 are
% pixels that are neighbors.
ind1 = [ indL(:) ; indU(:) ];
ind2 = [ indR(:) ; indD(:) ];

% Construct L so that L(i,j)=1 for neighboring pixels i,j
L = sparse(ind1,ind2,1,m*n,m*n);
L = L + L';

% Now build the diagonal of L, which equals to minus the row sum.
row_sum = full(sum(L,2));
L = L - spdiags(row_sum,0,m*n,m*n);

% Done
```

3.5.1 Image contour detection

Example 3.3 (Image contour detection)**.** Let U be a matrix which take values ranging from 0 to 1, representing a gray-scale image. Now consider it applied by a Laplacian $\mathbf{L}U$. For pixels whose color changes rapidly comparing to the neighbors (like at the edge/contour of an object), the corresponding value in $\mathbf{L}U$ will be large; for pixels whose color changes relatively smoothly across neighboring pixels (like in the interior of an object), $\mathbf{L}U$ is small.

Suppose `'cameraman.gif'` is an image file in the current folder.

```
close all
clear
clc

M = imread('cameraman.gif');
% M = rgb2gray(M); <-- Need this if M is a color image
U = double(M)/255;  % U is an image with double value ranging from 0 to 1
U1 = U(:);  % column-vector-version of the image

% build Laplacian
[m,n] = size(U);
L = Laplacian(m,n);

% apply Laplacian to the image
LU1 = L * U1;
LU = reshape(LU1,m,n);

imshow([U,abs(LU)])
```

Laplacian effectively extracts the contour of the objects (right) of the original image (left).

3.5.2 Harmonic function

A function u is said to be *harmonic* if its Laplacian vanishes

$$\mathcal{L}u = 0.$$

Likewise, we say an image U is *harmonic* if its Laplacian is zero

$$\mathbf{L}U = 0.$$

That is, all pixels have value equal to the mean of the neighboring pixels. Roughly speaking a harmonic function/image is the "smoothest" function/image (with a given boundary condition).

3.5.3 Image inpainting/restoration

Let U be an image. Suppose the image is *broken* in some subset $\mathcal{N}$ of the pixels; for linear indices $i \in \mathcal{N} \subset \{1, \dots, mn\}$, the data $U(i)$ is lost. Examples of $\mathcal{N}$ are that it is the set occupied by strong noises (zero-or-one noises, or the *salt-and-pepper noises*), and that it is some scratches on a photo. We assume that we can identify what $\mathcal{N}$ is for the given broken image. Let us denote $\mathcal{C} = \{1, \dots, mn\} \setminus \mathcal{N}$ the *clean* part or the unbroken region of the image.

Our goal is to assign values in the set $\mathcal{N}$ so that the image looks good. An reasonable assumption is that the values assigned should be smooth across pixels, so that they do not look like noise.

In order to obtain a smooth inpainting in $\mathcal{N}$ we ask for U so that $\mathbf{L}U = 0$ on pixels in $\mathcal{N}$. For convenience of explanation, we permute rows and columns so that we write $\mathbf{L}$ and U in block matrix form

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{NN} & \mathbf{L}_{NC} \\ \mathbf{L}_{CN} & \mathbf{L}_{CC} \end{bmatrix}, \quad U = \begin{bmatrix} U_N \\ U_C \end{bmatrix}$$

where indices of the blocks correspond to the indices in $\mathcal{N}$ or in $\mathcal{C}$. In particular, U_C are the undamaged value of the image and U_N is to be solved. Now we formulate

$$\begin{bmatrix} \mathbf{L}_{NN} & \mathbf{L}_{NC} \\ \mathbf{L}_{CN} & \mathbf{L}_{CC} \end{bmatrix} \begin{bmatrix} U_N \\ U_C \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ | \end{bmatrix} \leftarrow (\mathbf{L}U = 0 \text{ on pixels in } \mathcal{N})$$

The vertical bar in the right-hand side are just some values which we do not really care. Block matrix multiplication gives

$$\mathbf{L}_{NN} U_N = -\mathbf{L}_{NC} U_C. \tag{3.1}$$

Note that the right-hand side is a column vector. This is a linear system with $\mathbf{L}_{NN}$ square and invertible. (The entire Laplacian $\mathbf{L}$ is not invertible because the row sums are all zero; the vanishing row sum property does not hold anymore for a submatrix $\mathbf{L}_{NN}$).

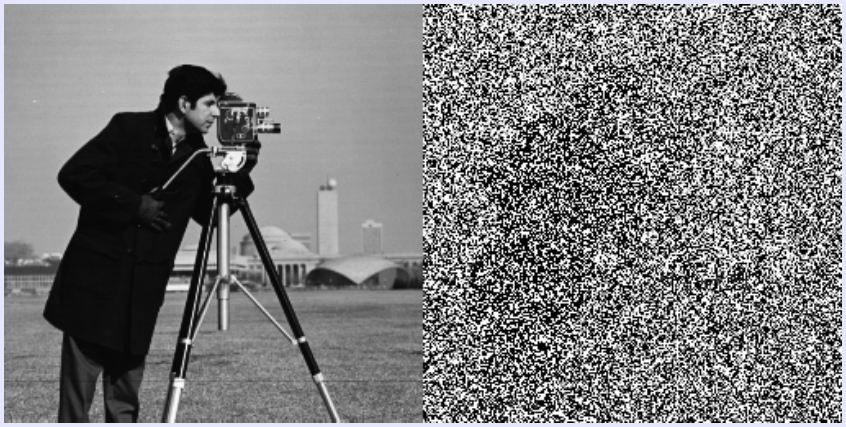
Example 3.4 (Destroying an image)**.** The following function adds salt-and-pepper noise on a given image with a given intensity.

add_noise.m

```
function B = add_noise(A,r)
% add_noise adds 0-or-1 noise to a given image A with intensity r.
% Every pixel will be replaced by a noise with probability r.  Each noisy
% pixel has 1/2 probability being black or white.

[m,n] = size(A);

random_numbers1 = rand(m,n);
random_numbers2 = rand(m,n);
isNoise = random_numbers1 < r;
isBlack = random_numbers2 < 0.5;

B = A;
B(isNoise) = 1;
B(isNoise & isBlack) = 0;
```

```
close all
clear
clc

M = imread('cameraman.gif');
U = double(M)/255;


U_noisy = add_noise(U,0.3);
imshow([U,U_noisy])
```

In the following figure we add white-and-pepper noise with intensity 0.8



Now, you are asked to restore the image from the noisy image.

(Restore from noisy image). Here we restore the image from `U_noisy`.

We will first identify the set $\mathcal{N}$ and $\mathcal{C}$, which will be `indNoise` and `indClean` in the code.

```
[m,n] = size( U_noisy );

isNoise = (U_noisy == 0) | (U_noisy == 1);
indNoise = find( isNoise );  % list of linear indices for noisy pixels
indClean = setdiff( 1 : m*n , indNoise );
```

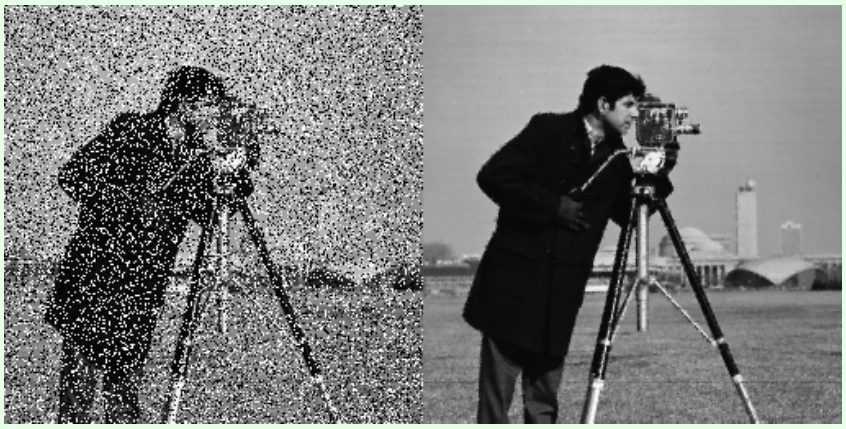
Then we construct the Laplacian, extract the submatrices $\mathbf{L}_{NN}, \mathbf{L}_{NC}$ and so on, and solve the linear system of Eq. (3.1)

```
% build Laplacian
L = Laplacian(m,n);

L_NN = L(indNoise,indNoise);
L_NC = L(indNoise,indClean);

U_noisy1 = U_noisy(:); % column-vector-vesion of U_noisy
rhs = -L_NC * U_noisy1(indClean);

U_denoise1 = U_noisy1;
U_denoise1(indNoise) = L_NN\rhs;
U_denoise = reshape( U_denoise1, m, n );
```

Then display the result

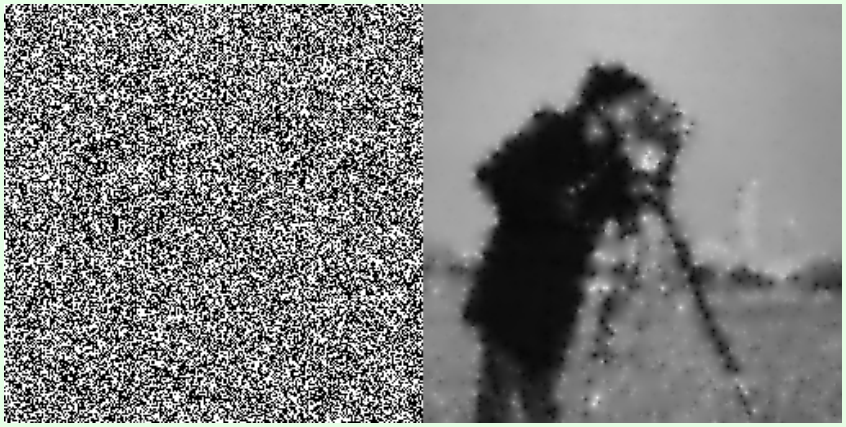
```
figure
imshow([U_noisy,U_denoise])
```

For noise density 0.3 we can recover the original image almost entirely:



With noise density 0.8 it is a lot that we can recover

For noise density 0.95,



■

3.6 Eigenvalue Problems for Sparse Matrices

Of a full matrix `eig` finds all eigenvalues and eigenvectors. For the eigenvalue problems for a sparse matrix, use `eigs` instead.

In many scenarios the sparse matrix we consider is large in size and we are only interested in a few of the eigenvalues and eigenvectors. The function `eigs` is equipped with efficient algorithms dealing with such cases. There are many options in `eigs` which you can find in

```
doc eigs
```

We present an example that gives a few eigenvectors corresponding to the few smallest (in magnitude) eigenvalues. For a sparse matrix `L`,

```
[V,D] = eigs( L, 5, 'sm');
```

returns the 5 smallest (in magnitude) eigenvalues on the diagonal of the 5-by-5 `D` and the corresponding eigenvectors stored in n-by-5 `V`.

3.6.1 Standing Waves of an L-shaped Drum

Suppose $u(x, y; t)$ is the height of a drum membrane at the planer location (x, y) at time t. Then the oscillating membrane satisfies the wave equation

$$\frac{\partial^2}{\partial t^2} u(x, y; t) = \mathcal{L}u(x, y; t)$$

where $\mathcal{L}$ is the Laplacian defined in the previous section. This is a partial differential equation (PDE). Now if we consider u an eigen*function* of $\mathcal{L}$, that is $\mathcal{L}u = \lambda u$ for some constant λ, the PDE becomes an ordinary differential equation (ODE) in time:

$$\frac{\partial^2}{\partial t^2} u(x, y; t) = \lambda u(x, y; t)$$

the solution to which is a simple harmonic oscillation with frequency $\sqrt{-\lambda}$. (Note that eigenvalues λ of $\mathcal{L}$ are ≤ 0). These eigenfunctions u are called standing waves, because every point of the membrane oscillates at a single coherent frequency.

Now let us solve these standing waves of the membrane of a drum.

If an image has pixel size h, then the discrete Laplacian with scaling

$$\frac{1}{h^2}\mathbf{L}$$

is an approximation to the differential operator $\mathcal{L}$. The eigenvalues of $\frac{1}{h^2}\mathbf{L}$ are approximations to those of $\mathcal{L}$ and the eigenvectors, after reshaping into an image, are approximations to the eigenfunctions of $\mathcal{L}$.

As what our experience tells us, the membranes are *fixed* on the boundary of the drum. So we will fix vectors U to zero at pixels that corresponds to the boundary of the drum. Let $\mathcal{C}$ be the set of linear indices that refers to the pixels in the boundary. Let $\mathcal{N}$ be the set of linear indices that are not in the boundary. Then we solve the eigenvalue problem in the non-boundary part

$$\frac{1}{h^2}\mathbf{L}_{NN}U_N = \lambda U_N \tag{3.2}$$

(think about it, this uses the assumption $U_C = 0$).

Example 3.5 (L-shaped drum)**.** Let the drum be $[0,1]\times[0,1]$ *with* $[0,1/2]\times[0,1/2]$ corner *subtracted*. The drum boundary is fixed. Plot the first five standing waves.

Solution. Let the image be $N \times N$ as a discretization of $[0,1]\times[0,1]$. Then the size of each pixel is $h = 1/N$.

```
clear
close all
clc

N = 50;   % number of grid on each side
h = 1/N;  % grid size
```

The boundary set is that i, j are on the boundaries of the image *union* the subtracted corner $\{(i,j)\,|\,i < N/2 \text{ and } j < N/2\}$.

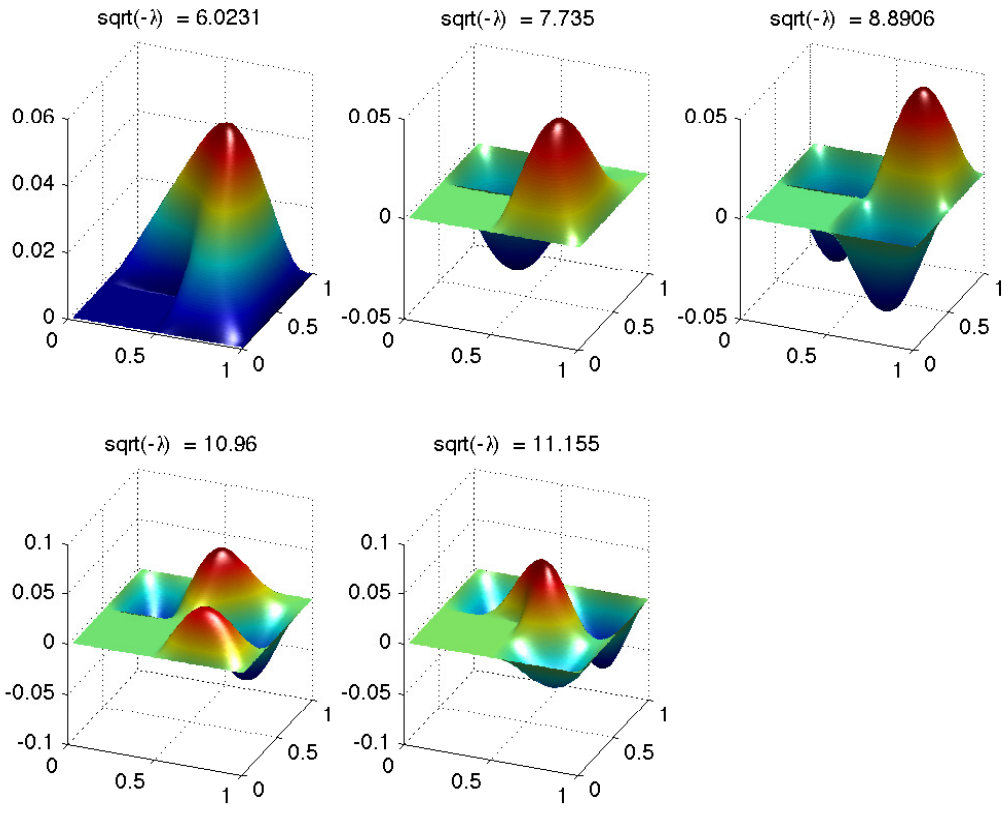
```
% Specify the boundary
[i,j] = ndgrid(1:N,1:N);
isBdy = ((i<N/2)&(j<N/2)) | (i==N) | (j==N) | (i==1) | (j==1);

% Linear indecies pointing to boundary and interior
indBdy = find(isBdy);
indInt = setdiff(1:N*N,find(isBdy));
```

Now solve the eigenvalue problem of Eq. (3.2)

```
% build Laplacian
L = Laplacian(N,N);
L_NN = L(indInt,indInt);

% solve the eigenvalue problem
[V,D] = eigs( L_NN / h^2 , 5, 'sm' );
```

We are pretty much done. Let us plot the solutions.

```
% plot the solutions
U = zeros( N, N );   % The image
for k = 1:5
    subplot( 2, 3, k)
    U(indInt) = -V(:,k); % Assign image value at interior point,
                         %  boundaries remain zero
    surf( h*i, h*j, U, ...
          'FaceLighting','phong',...
          'EdgeColor','none',...
          'FaceColor','interp')
    view(22,22)
    light
    title([ 'sqrt(-\lambda)  = ', num2str( sqrt(-D(k,k)) ) ])
end
```



■

Note that the first eigenfunction of the L-shaped drum is used as the MATLAB logo. (MATLAB logo uses a slightly different boundary).

Chapter 4

Ordinary Differential Equations

Every initial value problem (IVP) of an ordinary differential equation (ODE) has the standard form

$$\begin{aligned} &y' = F(t, y) \\ &y(t_0) = y_0 \\ &t \in [t_0, t_1] \end{aligned}$$

In general,

$$\begin{aligned} &t_0, t_1 \in \mathbb{R} \text{ with } t_0 < t_1 \\ &y(t) \in \mathbb{R}^n \\ &y_0 \in \mathbb{R}^n \\ &F : [t_0, t_1] \times \mathbb{R}^n \to \mathbb{R}^n. \end{aligned}$$

Here n is the dimension of the system ODE, which depends on your ODE problem. If your ODE problem can be written in the standard form, you can solve it numerically with MATLAB built-in ODE solvers.

4.1 MATLAB ODE Solvers

There are a few ODE solvers in MATLAB: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`. The syntaxes of each solver are basically the same. We take `ode23` as an example.

`[t,y] = ode23(`F`, [`t_0`,`t_1`] ,`y_0`);`

Here, F is a function handle that takes *two* arguments which defines $F(t, y)$ for each scalar input t and n-dimensional vector y. The returned value of $F(t, y)$ should be an n-dimensional *column* vector. The second slot `[`t_0`, `t_1`]` is any two dimensional vector with $t_1 > t_0$. In the third slot y_0 is an n-dimensional vector representing the initial condition.

The output **t** and **y** are the numerical solutions. **t** will be an N-by-1 column vector whose entries are a partition of the time interval $[t_0, t_1]$

$$t_0 = \mathbf{t}(1) < \mathbf{t}(2) < \cdots < \mathbf{t}(N) = t_1.$$

$\mathbf{y}$ will be an N-by-n matrix, so that

$$\mathbf{y}\texttt{(k,:)} \text{ is an approximation to the solution } y(\mathbf{t}\texttt{(k)}) \text{ at time } \mathbf{t}\texttt{(k)}.$$

In other words,

$$\texttt{plot(}\mathbf{t}\texttt{,}\mathbf{y}\texttt{)}$$

shows the plot of each component (of the n dimensions) of the solution with respect to time.

4.2 High-Order ODEs in the Standard Form

Example 4.1. Write the standard form of the initial value problem

$$\begin{aligned} &u''(t) + u'(t) + u(t) + u(t)^3 = \cos(10t) \\ &u(0) = 0, u'(0) = 1, \\ &0 \le t \le 20. \end{aligned}$$

Solution. For ODEs involving higher order derivatives, we introduce the variable

$$y(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} := \begin{bmatrix} u(t) \\ u'(t) \end{bmatrix} \in \mathbb{R}^2.$$

So

$$\begin{aligned} y'(t) = \begin{bmatrix} u'(t) \\ u''(t) \end{bmatrix} &= \begin{bmatrix} u'(t) \\ -u'(t) - u(t) - u(t)^3 + \cos(10t) \end{bmatrix} \\ &= \begin{bmatrix} y_2(t) \\ -y_2(t) - y_1(t) - y_1(t)^3 + \cos(10t) \end{bmatrix}. \end{aligned}$$

Hence we define $F : [0, 20] \times \mathbb{R}^2 \to \mathbb{R}^2$,

$$F(t, y) = \begin{bmatrix} y_2 \\ -y_2 - y_1 - y_1^3 + \cos(10t) \end{bmatrix}.$$

The initial condition for y is $y_0 = \begin{bmatrix} u(0) \\ u'(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. The problem in the standard form is

$$\begin{aligned} &y' = F(t, y) \\ &y(0) = y_0 \\ &t \in [0, 20]. \end{aligned}$$

To solve it with MATLAB,

```
F = @(t,y) [ y(2); -y(2)-y(1)-y(1)^3 + cos(10*t) ];
y0 = [ 0, 1 ];  % Can be row vector or column vector.
[ T, Y ] = ode23( F, [0,20], y0 );
```

The solution $(t, u(t))$ is approximated by

```
U = Y(:,1); % U(k) approximates the solution u(T(k)).
```

■

Chapter 5

Cell Arrays and Struct Arrays

In this chapter we introduce two more data types in MATLAB: cell arrays and struct arrays.

5.1 Cell Arrays

Let us first look at an example,

```
>> C = {'Some text', {1,2}; [1,2;3,4], @sin }

C =

    'Some text'     {1x2 cell}
    [2x2 double]    @sin

>> whos C
  Name      Size            Bytes  Class    Attributes

  C         2x2               770  cell
```

You can see that replacing `[ , ; , ]` by `{ , ; , }` you can create a new type of array, called *cell array*, which may contain data of different types and sizes, whereas matrices created by `[ , ; , ]` can only store values of the same type. Due to the flexibility of types and sizes in each cell element, cell arrays are useful in storing lists of text strings

```
>> NameList = {'J.S. Bach';'W.A. Mozart';'L.V. Beethoven'}
NameList =
    'J.S. Bach'
    'W.A. Mozart'
    'L.V. Beethoven'
```

and managing numerical arrays of different sizes

```
>> Samples = {randn(1,5); randn(1,100); randn(1,10000)}
Samples =
    [1x5     double]
    [1x100   double]
    [1x10000 double]
```

5.1.1 What is a Cell Array?

Cell array is a data type in MATLAB. A cell array, a value of class `'cell'`, is an indexed data container which can contain data of varying types and sizes. As a comparison, MATLAB matrices can only contain numbers of the same type, so matrices are also known as *homogeneous arrays*. Cell arrays, as general lists of data, do not have a clear meaning of vectors or matrices as homogeneous arrays do, so arithmetic operations and linear algebraic operations do not apply on cell arrays.

5.1.2 Creating Cell Arrays

You can create an empty cell array of size m-by-n with

```
>> C = cell(3,4)
C =
    []    []    []    []
    []    []    []    []
    []    []    []    []
```

You can also construct a cell array by the curly braces in which you put your desired values (similar to how you construct a matrix)

```
>> C = { @sin , @cos , @tan ; @csc , @sec , @cot }
C =
    @sin    @cos    @tan
    @csc    @sec    @cot
```

5.1.3 Concatenating Cell Arrays

The concatenation operator for cell arrays is still the *square brackets* (`[ , ; , ]`)

```
>> C = {@sin,@cos,@tan;@csc,@sec,@cot}
C =
    @sin    @cos    @tan
    @csc    @sec    @cot
>> D = {@(x) x.^2 ; @(x) 1./x.^2 }
D =
    @(x)x.^2
    @(x)1./x.^2

>> [C,D]
ans =
    @sin    @cos    @tan    @(x)x.^2
    @csc    @sec    @cot    @(x)1./x.^2
```

Note that if you use the curly braces { , ; , } what you get is a nested cell array

```
>> {C,D}
ans =
    {2x3 cell}    {2x1 cell}
```

5.1.4 Accessing Cell Arrays

There are two symbols you can use to access cells array: curly braces { } or smooth parentheses ( ).

Smooth parentheses access

Let

```
>> C = {@sin,@cos,@tan;@csc,@sec,@cot}
C =
    @sin    @cos    @tan
    @csc    @sec    @cot
```

Postfixed after a cell array variable, the smooth parentheses, in which (subscript/linear) indices are given, returns a *cell array*

```
>> C(:,2:3)
ans =
    @cos    @tan
    @sec    @cot
```

This is similar to how you extract submatrices from a matrix.

You can also set values to sub-cell-arrays by putting the expression `C(···)` on the left hand side of the assignment operator =. The values set to must be cell arrays of the same size

```
>> C([2;3;4]) = {'cosecant';'cosine';'secant'}
C =
    @sin          'cosine'    @tan
    'cosecant'    'secant'    @cot
>> C(:,3) = {'tangent','cotangent'}
C =
    @sin          'cosine'    'tangent'
    'cosecant'    'secant'    'cotangent'
>> C(1) = {'sine'}
C =
    'sine'        'cosine'    'tangent'
    'cosecant'    'secant'    'cotangent'
```

Curly braces access

While `()` accesses to a sub-cell-array of a cell array, the curly braces {} directly accesses to the values in the cells.

Let

```
>> C = {@sin,@cos,@tan;@csc,@sec,@cot}
C =
    @sin    @cos    @tan
    @csc    @sec    @cot
```

The expression `,2C2` accesses to the (2,2)-cell and returns the value within

```
>> C{2,2}
ans =
    @sec
```

The result is not a cell array but the actual value `@sec`, which is a function handle. Particularly you can do function handle evaluation by directly postfixing function-call parentheses

```
>> C{2,2}(pi)
ans =
```

```
    -1
```

The expression `,2:3C:` accesses to 4 entries of the cell array, and therefore it is effectively a function call which returns four outputs

```
>> C{:,2:3}
ans =
    @cos
ans =
    @sec
ans =
    @tan
ans =
    @cot
```

You may catch the multiple output values by the same way you catch multiple outputs from a function

```
>> [f1,f2,f3,f4] = C{:,2:3}
f1 =
    @cos
f2 =
    @sec
f3 =
    @tan
f4 =
    @cot
```

You may also create another cell array out of the multiple returned values

```
>> { C{:,2:3} }
ans =
    @cos    @sec    @tan    @cot
```

which is similar to

```
>> C(:,2:3)
ans =
    @cos    @tan
    @sec    @cot
```

You can set a value to a single cell

```
>> C{2,3} = 'cotangent'
C =
    @sin    @cos    @tan
    @csc    @sec    'cotangent'
```

To set values to multiple entries of a cell array,

```
>> D = {'cosine','secant'};
>> [C{:,2}] = D{:}
C =
    @sin    'cosine'    @tan
    @csc    'secant'    'cotangent'
```

where `D:` is an expression returning 2 values, and the expression `[C{:,2}] =` $\cdots$ catches the output (explained in the next section). In this case

```
>> C(:,2) = D(:)
C =
    @sin    'cosine'    @tan
    @csc    'secant'    'cotangent'
```

is more natural in syntax.

5.1.5 Comma-Separated Lists

A comma-separated list is a series of *values* (which can be evaluated from a variable or a function) separated by commas

```
@sin, 0, 'Hello World', sin(pi/2)
```

The evaluation of a comma-separated list looks like a function evaluation of multiple outputs

```
>> @sin, 0, 'Hello World', sin(pi/2)

ans =

    @sin


ans =

     0


ans =

Hello World


ans =

     1
```

In fact, we should view this another way around: An expression which returns multiple outputs *returns a comma-separated list.*

Typical examples of an expression returning multiple outputs are braces referencing of cell arrays

```
>> A = {pi,exp(1),1i,100}
A =
    [3.1416]    [2.7183]    [0 + 1.0000i]    [100]


>> A{[2,3,4]}
ans =
    2.7183
ans =
        0 + 1.0000i
ans =
   100
```

and we should understand `A{[2,3,4]}` as the comma-separated list

`A{2} , A{3} , A{4}`

Comma-separated list can be put in parentheses, brackets, braces:

- Array horizontal concatenation
  `x = [` *comma-separated list* `]` `%creates a homogeneous array`
  `x = {` *comma-separated list* `}` `%creates a cell array`
- Array subscripted-index referencing
  `x = M(` *comma-separated list* `)` `%M is a homogeneous array or a cell array`
  `x = C{` *comma-separated list* `}` `%C is a cell array`
- Function (handle) input arguments
  `x = f(` *comma-separated list* `)` `%f is a function or a function handle`
- Array assignment using subscripted-indexing
  `M(` *comma-separated list* `)` = *some value(s) or some cell array*
  `C{` *comma-separated list* `}` = *some value*
- Catching multiple outputs
  `[` *comma-separated list* `]` `= f(` $\cdots$ `)`
  `[` *comma-separated list* `]` `= C{` $\cdots$ `}`

5.1.6 Input/output argument with variable length

The expression `varargin` denotes an input argument list stored as a cell array. For example

myFunction.m

```
function [out1,out2, varargout] = myFunction( x, y, varargin )
%
% In this function varargin is a variable whose value is a cell array
```

When we call this function by `myFunction( 0, 1, 2, 3 )` then the *comma-separated list* `2, 3` is passed in to `myFunction`, with `varargin = {2,3}`. The cell array `varargin` is possibly empty. This is very useful for designing functions with optional inputs.

Similarly, `varargout` is a cell array to be assigned in `myFunction`. It can take different length or contain different types depending on what happens in the function.

The command `varargin` can also be used in anonymous functions.

Example 5.1. Define a function handle named `evaluate` which is able to perform the following task

`evaluate(`$f, a, b, \ldots$`)` returns $f(a, b, \ldots)$ for any function handle f

Solution.

```
evaluate = @(f, varargin) f( varargin{:} );
```

Any inputs start from the second argument of `evaluate` will be collected as a cell array, which is the value of the local variable `varargin`. Then `varargin{:}` turns the cell array into a

comma-separated list, which is located in the parentheses of the function call of `f`.

```
>> evaluate( @sin, pi/2 )
ans =
     1
>> evaluate( @plus, 1, 1 )
ans =
     2
>> evaluate( @quad, @sin, 0, pi )
ans =
    2.0000
```

■

5.1.7 The function `cellfun`

Let f be a function handle which takes 1 argument (could be one array) and returns a *scalar*. Then

`cellfun(`f`,` $\{a_1, a_2, \cdots\}$`)` returns `[`$f(a_1), f(a_2), \cdots$`]`.

Example 5.2. Suppose we want to compute the mean of a few random samples with different sample size.

```
>> Samples = {randn(1,5); randn(1,100); randn(1,10000)}
Samples =
    [1x5     double]
    [1x100   double]
    [1x10000 double]
```

Now to take their mean, call

```
>> cellfun( @mean, Samples )
ans =
    0.0556
    0.0477
   -0.0017
```

If f is a function handle that takes more input arguments and returns a scalar,

`cellfun(`f`,` $\{a_1, a_2, \cdots\}$`,`$\{b_1, b_2, \cdots\}$`)` returns `[`$f(a_1, b_1), f(a_2, b_2), \cdots$`]`.

If f returns non-scalar value, add an extra property-value pair

`cellfun(`f`,` $\{a_1, a_2, \cdots\}$`,`$\{b_1, b_2, \cdots\}$`, 'UniformOutput', 0)`
returns $\{f(a_1, b_1), f(a_2, b_2), \cdots\}$,

which is a cell array instead of a homogeneous array.

Example 5.3. Define a function handle `apply_list_of_functions` so that it is able to perform

$$\texttt{apply\_list\_of\_functions}(\{f_1, f_2, \cdots\}, x)$$
$$\text{returns } [f_1(x), f_2(x), \cdots]$$

Solution.

```
apply_list_of_functions = @(lf, x) cellfun(@(f) f(x), lf);
```

Now

```
>> apply_list_of_functions({@sin,@cos,@(x) x.^2}, pi/2 )
ans =
    1.0000    0.0000    2.4674
```

■

5.1.8 Summary

1. Cell arrays allow you to store data of different types and sizes as an array.
2. The smooth parentheses `C(I)` indexing represents a sub-cell-array.
3. The curly braces `C{I}` indexing represents a comma-separated list of cell elements.

5.2 Structure Array

Structure array is a MATLAB data type. Structure arrays, like cell arrays, are containers in which you can put values of different types and sizes.

5.2.1 What is a structure array?

Scalar structure

When you type

```
>> clear
>> card.suit = 'spade';
>> card.number = 5;
>> card

card =

      suit: 'spade'
    number: 5
```

you define a variable `card` whose value is of class `struct`. The value of `card` is a scalar (`size(card)` is 1-by-1). This scalar struct has two fields: `'suit'` and `'number'`, which was specified arbitrarily by us.

The set of fields (in this case {'suit','number'}) defines a *structure*. You can say that `card` is a scalar structure with fields `suit` and `number`.

Each field of a scalar structure is a container in which you can store a value of arbitrary type or size. To access a field of a scalar structure, use the dot "`.`"; for example, `card.number` represents the value we have set

```
>> card.number

ans =

     5
```

Structure array

A structure array is a *homogeneous array* (a matrix) of scalar structures *with the same set of fields*. For example,

```
>> anotherCard.suit = 'heart';
>> anotherCard.number = 'Q';
>> cards = [card , anotherCard];
```

Now `anotherCard` is also a scalar struct with the same set of fields {'suit','number'}. You can create a matrix `cards` by concatenating `card` and `anotherCard`. Note that it is okay if `card.number` is of class `double` and `anotherCard.number` is of class `char`. The value of `cards` is a structure array with fields `suit` and `number`.

5.2.2 Creating a structure array

To create a structure array, the simplest way is similar to the above example: define a field by setting some value in it. In the following example we preallocate `cards` as a structure array of size 52×1 with fields `suit` and `number`

```
>> clear
>> cards(52,1).suit = [];
>> cards(52,1).number = [];
>> cards

cards =

52x1 struct array with fields:
    suit
    number
```

Removing fields

To remove a field from a structure,

```
>> cards = rmfield(cards,'suit');
>> cards

cards =
```

```
52x1 struct array with fields:
    number
```

Note that you have to replace `cards` by the result `rmfield(cards,'suit')` because `rmfield` is a MATLAB function, which will not mutate the state of the variables in the input argument.

Adding a field to a structure

```
cards(1).suit = [];
>> cards

cards =

52x1 struct array with fields:
    number
    suit
```

Defining a new field `suit` (which had been removed) to an element of the structure array will effectively give the field to the entire array.

5.2.3 Accessing a structure array

Parentheses indexing

Structure arrays are homogeneous arrays (matrix), so the indexing in parentheses is exactly the same as how it works for matrices. In particular, `cards(I)` is another structure array with the same set of fields; and `cards(5)` is a scalar structure, which you can postfix "`.number`" to access the field value.

Field access

A structure array `S` postfixed with "`.`*field_name*" represents the comma-separated list:

`S.`*field_name* is the same as
`S(1).`*field_name*`, S(2).`*field_name*`,` $\cdots$ `S(end).`*field_name*

Example 5.4. Assigning values to a whole deck of playing cards:
First preallocate the struct array

```
>> clear
>> cards(52).number = [];
>> cards(52).suit = [];
```

Let

```
>> suitValues = repmat( {'club','diamond','heart','spade'}, 13, 1 );
```

so that `suitValues` is the cell array

```
suitValues =

    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
```

```
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
    'club'    'diamond'    'heart'    'spade'
```

Next, define the cell array

```
>> ranks = [{'A'};num2cell( (2:10)' );{'J'};{'Q'};{'K'}];
>> numberValues = repmat(ranks,1,4)

numberValues =

    'A'     'A'     'A'     'A'
    [ 2]    [ 2]    [ 2]    [ 2]
    [ 3]    [ 3]    [ 3]    [ 3]
    [ 4]    [ 4]    [ 4]    [ 4]
    [ 5]    [ 5]    [ 5]    [ 5]
    [ 6]    [ 6]    [ 6]    [ 6]
    [ 7]    [ 7]    [ 7]    [ 7]
    [ 8]    [ 8]    [ 8]    [ 8]
    [ 9]    [ 9]    [ 9]    [ 9]
    [10]    [10]    [10]    [10]
    'J'     'J'     'J'     'J'
    'Q'     'Q'     'Q'     'Q'
    'K'     'K'     'K'     'K'
```

Now,

```
>> [cards.number] = numberValues{:};
>> [cards.suit] = suitValues{:};
```

The first line is read as that `numberValue{:}` creates a comma-separated list as a multiple output, and those outputs are caught in the LHS, which is a comma-separated list in a bracket `[cards(1).number,cards(2).number,···,cards(52).number]`.

You can check that all cards have been set a value forming a deck of standard playing cards

```
>> cards(25)

ans =

    number: 'Q'
      suit: 'diamond'
```

Example 5.5. Shuffle the deck of cards

```
shuffledCards = cards( randperm(52) );
```

5.3 Places to use structure array in MATLAB

- When loading `.mat` files. You can save your workspace variables by `save('filename.mat')` as a `.mat` file. When you load the `.mat` file

  ```
  S = load('filename.mat')
  ```

  is a structure (scalar) whose fields are the variable names saved in `filename.mat`.

- Storing a list of data with common classes of information. For example, you can store a list of countries as a structure array with fields *country name*, *population*, *capital*.

- Variable naming and packing. In dealing with a complicated task, there would be lots of variables whose names are easily get messy. You can put variables under a structure so that it is easier to infer what the variables mean and prevent potential variable naming conflicts. For example,

  ```
  my_curve.x, my_curve.y
  my_surface.x, my_surface.y, my_surface.z
  ```

  uses `x` and `y` to indicate coordinate data. But under different structures they do not conflict each other. Another advantage of this is that you can pass a single structure variable `my_curve` into a function, otherwise you may have to let the function take a lot of input arguments.

Chapter 6

Class Objects

6.1 Why Object-Oriented Programming (OOP)?

Programming using *classes* is called an object-oriented programming (OOP). For non-OOP programmers handling larger codes

1. duplicating codes
2. changing codes

are often the issues to worry about. A copy-and-paste of a piece of code means that when you need to change the code, it requires you to change to multiple places in the program. The code *will always be changed*, either due to bug fixing or adding new feature to the program. Hence, an easy-to-read and easy-to-change properties of a code is very important.

OOP makes coding easier in the sense of making the program more organized and easy to change. Tasks the program take are written as functions (or called *methods*) which reduce duplication of the code – an update of the function changes everywhere your program calls that function.

6.2 What are Classes and Objects?

In an OOP you will define one or several *classes*. *Classes* are generalization to default classes or types such as `double`, `char` and so on. Imagine a class as a "species".

A value of the type being your customized class is called an *object* (also called an *instance*). An object is like "an individual" of the "species".

To define a class you define what the

- properties (class members)
- methods (class functions)

are in the class. Class *properties* to a class is similar to fields to a structure; *properties* are the variable names in which you can store data of arbitrary kind. Class *methods* are functions that belong to a class. *Properties* and *methods* of a class are like the "organs" in an individual of your "species"(class). As what you expect, each individual has the same set of organs in its body; likewise, each class object has those properties as slots where you can store data, and each class object is allowed to call the class methods.

6.2.1 Class M-file in MATLAB

In MATLAB, a class is defined in an m-file. Recall that a MATLAB m-file can either be a script, a function, or a class; now we are looking at a class m-file.

To declare that an m-file is a class m-file, it needs to have the following structure

MyClass.m

```
classdef MyClass
    properties
        ...
    end
    methods
        ...
    end
end
```

Note that the title needs to be the name of the class, which is `MyClass` in this case.

For example,

Example 6.1 (A triangle class)**.** The following is a class whose objects are triangles

Triangle.m

```
classdef Triangle
% Triangle is a class of triangle
%
    properties
        p1 = [0,0] % coordinate of the 1st vertex
        p2 = [0,0] % coordinate of the 2nd vertex
        p3 = [0,0] % coordinate of the 3rd vertex
    end
    methods
        function [e12,e13,e23] = edges( tri )
            % edges return the 3 edges of a triangle
            % the 3 returned values are in the order e12,e13,e23
            e12 = tri.p2 - tri.p1;
            e13 = tri.p3 - tri.p1;
            e23 = tri.p3 - tri.p2;
        end

        function A = area( tri )
             % area computes the area of a given triangle
            [e12, e13, ~] = tri.edges; % tri.edges is equivalent to edges(tri)
            A = sqrt( norm(e12)^2 * norm(e13)^2 - dot(e12,e13)^2 )/2;
        end

        function S = perimeter( tri )
            % perimeter computes the perimeter of a given triangle
            [e12, e13, e23] = tri.edges;
            S = norm(e12) + norm(e13) + norm(e23);
        end
    end
```

```
    end
```

One can see that there are three properties of the class `Triangle`: the coordinates of the three vertices. These values characterize a triangle, an object of this class.

Of each triangle, we can compute the edge vectors, the area, and the perimeters. The calculations of these geometric quantities are written as functions, which are the methods of `Triangle`.

To construct a triangle,

```
>> t = Triangle;
```

Now, `t` is a default triangle. You can access the properties of `t` similar to struct arrays

```
>> t.p1 = [1,0];
>> t.p2 = [0,2];
>> t.p3 = [0,0];
```

Now `t` is finally a triangle with three vertices $(1,0),(0,2),(0,0)$. You can call methods using "."

```
>> t.area
ans =
     1
```

which is equivalent to

```
>> area(t)
ans =
     1
```

In the above example, you have seen how to construct an object, access a property, and call a method. We will explain in detail with examples what the rules are later.

6.2.2 The @-Folder

You can write a class definition in a single m-file. But you will quickly discover that if the number of methods is a lot, and the codes in the methods are sophisticated, your class m-file would get very messy. The way to distribute methods as function m-files is to use an @-folder.

The @-folder is a folder with name `@MyClass`, where `MyClass` is the name of your class. The class m-file `MyClass.m` must be inside the folder `@MyClass`. Then the methods of `MyClass` can be written as separate function m-files in the folder `@MyClass`.

For a method being separated into an m-file, you still need to declare the function in `MyClass.m`. To declare a method, just write a line

```
[out1,out2] = myMethod1(input1,input2)
```

anywhere between `methods` and the corresponding `end`. (This is obviously just an example; you may have different number of inputs and outputs). Here `myMethod1` will need have a corresponding function m-file in `@MyClass`

myMethod1

```
function [out1,out2] = myMethod1(input1,input2)
```

```
    ...
```

When you use the class `@MyClass`, just use it as usual when `@MyClass` is in your current folder. In particular, do not go into the folder `@MyClass`.

Example 6.2 (Triangle class in @Triangle)**.** Example 6.1 can be redone by the following. Create a folder `@Triangle`. In the folder `@Triangle/` there are three files

- `Triangle.m`
- `area.m`
- `perimeter.m`

which are

Triangle.m

```
classdef Triangle
% Triangle is a class of triangles
%
    properties
        p1 = [0,0] % coordinate of the 1st vertex
        p2 = [0,0] % coordinate of the 2nd vertex
        p3 = [0,0] % coordinate of the 3rd vertex
    end
    methods
        function [e12,e13,e23] = edges( tri )
            % edges return the 3 edges of a triangle
            % the 3 returned values are in the order e12,e13,e23
            e12 = tri.p2 - tri.p1;
            e13 = tri.p3 - tri.p1;
            e23 = tri.p3 - tri.p2;
        end
        % area computes the area of a given triangle
        A = area( tri )

        % perimeter computes the perimeter of a given triangle
        S = perimeter( tri )
    end
end
```

area.m

```
    function A = area( tri )
    % area computes the area of a given triangle
    [e12, e13, ~] = tri.edges; % tri.edges is equivalent to edges(tri)
    A = sqrt( norm(e12)^2 * norm(e13)^2 - dot(e12,e13)^2 )/2;
```

perimeter.m

```
    function S = perimeter( tri )
```

```
    % perimeter computes the perimeter of a given triangle
    [e12, e13, e23] = tri.edges;
    S = norm(e12) + norm(e13) + norm(e23);
    end
```

This leaves the class m-file clean. Here `[e12,e13,e23] = edges( tri )` can be separated out as a function m-file.

Use the class `Triangle` outside the folder `@Triangle`.

With methods separated into function m-files, the class m-file is left with clean and easy-to-read definitions of the class, which are the properties and class methods decorated with comments.

6.3 Class Methods

A class method is a function belonging to a class. There are two different kinds of class methods

- Non-static methods
- Static methods

All methods we have seen in the previous examples are non-static methods. If you simply write functions under `methods`, they are non-static methods.

There is a special non-static method – the *class constructor*. In many applications, one should define a class constructor explicitly in the class m-file. If one does not define a class constructor, like in Example 6.1, MATLAB uses a default class constructor.

6.3.1 Class constructor

A class constructor is a method

- written between `methods` and `end`.
- whose name is the same as the class.
- which has exactly one output argument.
- whose output variable should be a class object.
- whose input arguments can be arbitrary.

When we create a class object

```
>> obj = MyClass(···)
```

it is the constructor function that is called.

Example 6.3 (Writing a constructor)**.** Let us write a constructor for `Triangle`

Triangle.m

```
classdef Triangle
% Triangle is a class of triangles
%
```

```
        properties
            p1 = [0,0] % coordinate of the 1st vertex
            p2 = [0,0] % coordinate of the 2nd vertex
            p3 = [0,0] % coordinate of the 3rd vertex
        end
        methods
            function tri = Triangle( varargin )
                % This is the class constructor
                if nargin==3
                    tri.p1 = varargin{1};
                    tri.p2 = varargin{2};
                    tri.p3 = varargin{3};
                end
            end


            % edges return the 3 edges of a triangle
            [e12,e13,e23] = edges( tri )

            % area computes the area of a given triangle
            A = area( tri )

            % perimeter computes the perimeter of a given triangle
            S = perimeter( tri )


        end
    end
```

To create a triangle,

```
>> tri = Triangle([1,0],[0,2],[0,0]);
```

Note There is no command `this` in MATLAB that refers to "this object" as in C++ or Java. So we have to write explicitly `tri.p1 = ...` instead of just writing `p1 = ...`.

6.3.2 Non-static methods

Every non-static method (except for the constructor) should be a function with

- at least one input argument
- one of the input arguments (usually the first argument) being an object of this class

A method

```
function [out1,out2] = myMethod( obj,in2,in3 )
```

can be called by the syntax

```
[out1,out2] = obj.myMethod( in2, in3 )
```

which would be translated back to `[out1,out2] = myMethod( obj, in2, in3)`. In particular, `obj.myMethod( in2, in3 )` is a function call with 3 input arguments.

For a typical class (*handle class* is the exception), methods, like all other functions, do not mutate the input arguments.

Int.m

```
classdef Int
% a class of integer
properties
    value = 0
end
methods
    function plus_one(obj)
        obj.value = obj.value + 1;
    end
end
end
```

You will see that `plus_one` will not change the property values in the object.

```
>> i = Int;
>> i.value = 1;
>> i.plus_one;
>> i.value
ans =
     1
```

6.3.3 Public/private methods

In Example 6.1 and Example 6.3, all methods (`area`, `edges`, `perimeter` and the constructor `Triangle`) are *public methods*. As the name suggests, these methods can be called in the command window, some script, or codes in some other class or functions. You can set a method to be *private*, which means that the function can be called only by methods in the same class; in particular, you cannot see the private methods when you use the object outside.

To set methods private, put them between another block of `methods`-`end` pair, with a specification on methods attribute as follows

```
classdef SomeClass
    properties
        % public properties
    end
    methods
        % public methods go here
    end
    methods (Access = private)
        % private methods go here
    end
end
```

Note that the constructor has to be a public method.

Example 6.4 (Private method)**.** Let us make the method `edges` private.

Triangle.m

```
classdef Triangle
% Triangle is a class of triangles
%
    properties
        p1 = [0,0] % coordinate of the 1st vertex
        p2 = [0,0] % coordinate of the 2nd vertex
        p3 = [0,0] % coordinate of the 3rd vertex
    end
    methods
        function tri = Triangle( varargin )
            % This is the class constructor
            if nargin==3
                tri.p1 = varargin{1};
                tri.p2 = varargin{2};
                tri.p3 = varargin{3};
            end
        end

        % area computes the area of a given triangle
        A = area( tri )

        % perimeter computes the perimeter of a given triangle
        S = perimeter( tri )
    end
    methods (Access = private)
        % edges return the 3 edges of a triangle
        [e12,e13,e23] = edges( tri )
    end
end
```

Now, `area` and `perimeter` can still use `edges`. But for users using this class `Triangle`, they can only call public methods `area`, `perimeter`, and the constructor.

For helper functions that are not going to be directly called by the user, make them private. It is always nice to pack your code as a class with a clean public interface and bunch of invisible private helper methods. This idea is called **encapsulation**, which is one of the key features of object-oriented programming.

6.3.4 Static methods

Static methods are methods that do not depend on the state of the objects. They are simply functions defined under the class. The input arguments of a static method should not be an object of the class.

To make a method static, put it under

```
methods (Static)
    % static methods go here
end
```

Example 6.5 (Static methods)**.** We continue the `Triangle` class. Let us add a method `A = side2area(l1,l2,l3)` which computes the area of a triangle with lengths of the sides `l1,l2,l3`. Such computation does not depend on any class object, so it should be written as a static method.

Triangle.m

```
classdef Triangle
% Triangle is a class of triangles
%
    properties
        p1 = [0,0] % coordinate of the 1st vertex
        p2 = [0,0] % coordinate of the 2nd vertex
        p3 = [0,0] % coordinate of the 3rd vertex
    end
    methods
        function tri = Triangle( varargin )
            % This is the class constructor
            if nargin==3
                tri.p1 = varargin{1};
                tri.p2 = varargin{2};
                tri.p3 = varargin{3};
            end
        end

        % area computes the area of a given triangle
        A = area( tri )

        % perimeter computes the perimeter of a given triangle
        S = perimeter( tri )
    end
    methods (Access = private)
        % edges return the 3 edges of a triangle
        [e12,e13,e23] = edges( tri )
    end
    methods (Static)
        function A = side2area(l1,l2,l3)
        % side2area computes the area of the triangle with given edge lengths
            s = (l1 + l2 + l3)/2;
            A = sqrt( s*(s-l1)*(s-l2)*(s-l3) );
        end
    end
end
```

To call the static method,

```
>> Triangle.side2area(1,2,sqrt(5))
ans =
     1
```

Static methods are just functions with function names prefixed "*ClassName*.".

When to use static methods?

The rule of thumb is, ask yourself whether the call of the function make sense if no object has been constructed. If the answer is positive, make is a static method; otherwise, make it non-static.

It does not make sense to call the function `area`, which computes an area of a triangle, if there is no triangle constructed. Thus `area` is non-static. When there is no triangle object created, it still makes sense to the convert 3 given edge lengths to an area. So `side2area` is static.

Named constructors

It makes a lot of sense to call a function which creates a class object when there were no class object created. These functions are called *named constructors*, which are static functions with output value being a class object. They are not *the* class constructor, which is the function that has the same name as the class.

Example 6.6 (Named constructors)**.** Let us define another static method

```
methods (Static)
   function A = side2area(l1,l2,l3)
   % side2area computes the area of the triangle with given edge lengths
       s = (l1 + l2 + l3)/2;
       A = sqrt( s*(s-l1)*(s-l2)*(s-l3) );
   end
   function tri = Equilateral(a)
   % Equilateral constructs an equilateral triangle with side length a
       tri = Triangle( [0,0], [a,0], [a/2,sqrt(3)*a/2] );
   end
end
```

```
>> tri = Triangle.Equilateral(2);
```

creates a triangle `tri` which is equilateral.

```
>> tri.area
ans =
    1.7321
```

6.3.5 Operator Overloads

When defining class methods, it is okay, and in fact recommended, to name a method the same as some MATLAB built-in function. For example, you can have a method under the class `MyClass` that has the name `plus`, which has two inputs

```
function obj3 = plus( arg1, arg2 )
```

Since methods belong to your class there is no worry of conflicting with the built-in ones. In this case, you can create `obj1` `obj2`, instances of `MyClass`, and perform

```
obj3 = obj1 + obj2;
```

MATLAB always translates `a + b` as `plus(a,b)` and thus call your implementation.

In this case we *overloads* the plus operator with new feature that works for our class.

Example 6.7. We add another method in `Triangle`

```
        methods
            function tri3 = plus(tri1,tri2)
            % plus computes the sum of two triangles
                tri3 = Triangle;
                tri3.p1 = tri1.p1 + tri2.p1;
                tri3.p2 = tri1.p2 + tri2.p2;
                tri3.p3 = tri1.p3 + tri2.p3;
            end
        end
```

```
        >> tri1 = Triangle.Equilateral(1);
        >> tri2 = Triangle.Equilateral(2);
        >> tri3 = tri1 + tri2;
```

6.4 Handle Classes

Recall that in Section 6.3.2 we said that methods are like functions which do not modify the state of the input variable, including the class object. But sometimes we do want the mutation of object state to happen. Making your class as a *handle* class, the class methods are allowed to change the state of the class object.

To make your class a handle class, simply write

```
classdef MyClass < handle
    ...
end
```

on the first line.

Example 6.8. The following class is defined so that the property values can be modified from the class methods.

Int.m

```
classdef Int < handle
    % a class of integer
    properties
        value = 0
    end
    methods
        function plus_one(obj)
            obj.value = obj.value + 1;
        end
    end
end
```

```
>> i = Int;
>> i.value = 1;
>> i.plus_one;
>> i.value
ans =
     2
```

What happens here is that `i` is just a *handle* which refers to an object somewhere back in the memory. From a handle, the syntax of accessing the property values of the object is exactly the same as non-handle object – use "*.propertyName*". When `plus_one(i)` is called, it is the handle that is copied to the local variable `obj`; when the function assign value to `obj.value`, it really modify the value somewhere back in the memory. Hence the value of the property changes even after we exit the function.

Since these objects are handles, not the real object itself, something funny may happen

Example 6.9. Continued from Example 6.8.

```
>> i1 = Int;
>> i1.value = 5; % now i1 is 5
>> i2 = i1;      % let i2 = i1,
>> i2.value      % i2 is also 5 as expected
ans =
     5
>> i1.plus_one;  % Add 1 to i1
>> i2.value      % both i1 and i2 become 6
ans =
     6
```

When we call `i2 = i1`, it is only the handle that is copied to `i2`. After that both `i1` and `i2` refers to the same class instance. So a change of state of the object can be seen from both `i1` and `i2`.

The following example is a more appropriate usage of handle class.

Example 6.10 (A class for solving ODEs)**.** The following class

ODEProblem.m

```
classdef ODEProblem < handle
% ODEProblem is a class for solving ODE problems
%

    properties
        odeFun % a function handle with input (t,y)
        tspan  % a two-vector
        y0     % initial condition

        T      % solution time grid
        Y      % solution values
    end
    methods
        function ode = ODEProblem(odeFun,tspan,y0)
```

```
            % class constructor
            ode.odeFun = odeFun;
            ode.tspan = tspan;
            ode.y0 = y0;
        end
        function reset(ode)
            % reset cleans up the solution
            ode.T = [];
            ode.Y = [];
        end
        function solve(ode,solver)
            % solve solves the ode problem with a given solver.
            % The solver is a function handle, e.g. @ode45
            [ode.T,ode.Y] = solver(ode.odeFun,ode.tspan,ode.y0);
        end
        function plot(ode)
            % plot plots the solution
            plot(ode.T,ode.Y)
        end
    end
end
```

The usage of such class is intuitive

```
>> ode = ODEProblem(@(t,y) -y, [0,2], 5 );
>> ode.solve(@ode23)
>> ode.plot
>> ode.reset
>> ode.solve(@ode45)
>> ode.plot
```

Note that when we call `solve` or `reset`, we really change the state of `ode`.

The great advantage of using handle classes is

- No copying of class property data when calling a method.
- Ability of modifying object state by function calls.

When to use handle classes

Use handle classes if

- Every object exists uniquely, and there is rarely copying an object going on.

For example,

- Characters in a role playing game – they exist uniquely in a game. And you can call

  ```
  monster1.go_to_sleep
  ```

  and change the state of the character. (If it is not a handle object, you will need to call `monster1 = monster1.go_to_sleep` and keep copying and replacing the character whenever it makes a move).

- The pack of a big program – there is only one universal class object in the program, and there are usually a lot of data that you do not want to copy. Like in the ODE example

```
ode.solve(@ode45)
```

and change some state in the object.

6.5 Subclasses

A subclass `MySubClass` of a superclass `MySuperClass` is a class whose class properties and methods are inherited from the superclass, and usually you put more class properties and methods in the subclass.

To declare a class being a subclass of a class

```
classdef MySubClass < MySuperClass
```

In particular, all handle classes are subclasses of the *abstract* handle class.

Subclasses can use the *public* and *protected* (`methods` with `Access = protected`) methods of the superclass. It also inherits the properties of the superclass. You can overwrite a superclass method or the default value of a superclass property.

6.5.1 When to make a class a subclass of another class?

Ask yourself whether it makes sense to say "`SubClass` *is a* `SuperClass`". For example, you can make a class `EquilateralTriangle` as a subclass of `Triangle`, with formula of perimeter overwritten to an easier one. It make sense to say "An equilateral triangle is a triangle".

Example 6.11 (Role playing game)**.** In the world of role playing game you will typically write a class

```
classdef LivingThing < handle
    properties
        health = 100
    end
end
```

Then

```
classdef Human < LivingThing
    properties
        name = 'anonymous'
        age
        ...
    end
end
```

```
classdef Warrior < Human
    properties
        weapon
        ...
    end
```

```
    end
```

whereas

```
classdef Sheep < LivingThing
    properties
        ...
    end
end
```

Now, a human is a living thing, which has class properties

- health
- name
- age.

Warriors in addition has a weapon.

Sheep are living things so it has health, but they probably do not have a name.

6.5.2 Call the methods in the superclass

Sometimes you will overwrite superclass methods, so the implementations of the method of the subclass is different from that of the superclass. To call the superclass method from the subclass,

```
methodName@MySuperClass( inputArguments )
```

6.5.3 Call the superclass constructor

Sometimes you want to write the constructor of the subclass which calls the superclass constructor with particular input arguments

```
obj = obj@MySuperClass( inputArguments )
```

Example 6.12 (Spring-mass system as a subclass of the ODEProblem)**.** First of all, a spring-mass system is an ODE problem. Hence it make sense to put it as a subclass of `ODEProblem`

MassSpringProblem.m

```
classdef MassSpringProblem < ODEProblem
    properties
        m = 1;
        k = 1;
    end
    methods
        function ode = MassSpringProblem
            ode = ode@ODEProblem( [], [0,1], [0;0] );
            ode.build
        end
        function build(ode)
```

```
            ode.odeFun = @(t,y) [y(2) ; -ode.k/ode.m * y(1)];
        end
    end
end
```

Here is a script which you can try out

```
%%
ode = MassSpringProblem;

ode.y0 = [1;0];
ode.tspan = [0,10];

ode.solve(@ode45)
ode.plot

%%
ode.reset
ode.k = 16;
ode.build;

ode.solve(@ode45)
ode.plot
```

6.6 Summary

The key features of OOP are

- Encapsulation – you can pack functions and variables into one or several organized class with a friendly and high-level interface
- Inheritance – you can create subclass from a superclass so that you don't need to repeat implementations of shared methods.
- Polymorphism – you can have an abstract class that supports different task when you subclass from it; for example `ODEProblem` can become a spring-mass problem or many other ODE problems.

These features reduce a lot of duplication of codes, and make the program much neater. They also make coding much easier when the code needed to be modified.

Chapter 7

Graphical User Interface

A *user interface* (UI), or a *graphical user interface* (GUI), is a graphical display in one or more windows which enable users to perform interactive tasks. Controls contained in a GUI window may include menus, sliders, push buttons and keyboard/mouse detections (just named a few).

7.1 How does a UI work?

UIs are made of UI *components*. These components can be objects under a figure such as push buttons, text boxes, axes, or they can be figures themselves. UI components wait for a user to perform certain manipulation, and typically, such events of manipulations will trigger an execution of a *callback function*. This kind of programming is known as *event-driven* programming.

7.2 UI Control objects

`uicontrol` is the function which creates a UI control object. UI control can be a text box, a pushbutton, a slider, or many more listed later. Each UI control occupies a region in the figure window. When creating a UI control object, you will typically specify the

1. style (what kind of control (pushbutton? slider?))
2. units (what unit is using when specifying position later)
3. position (`[x,y,width,hight]`)
4. string (optional, the text displayed on the control)
5. value (state of the control, such as for slider)
6. callback (callback function, a function handle)

of the UI control. The styles of UI controls include

- pushbutton
- togglebutton
- checkbox

- radiobutton
- slider
- edit
- text
- popupmenu
- listbox
- frame

Here is an example for creating a push button

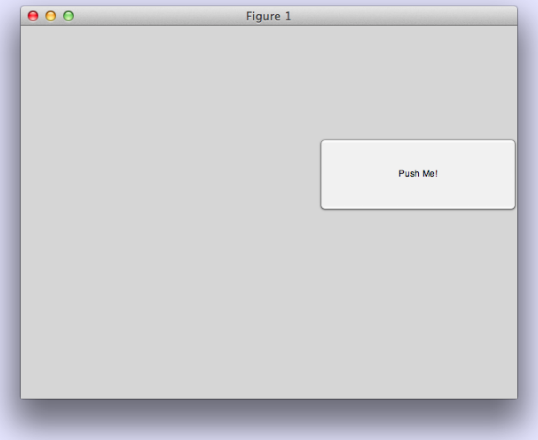
Example 7.1. The following function creates a figure with a push button in it

gui_example.m

```
function gui_example
hFig = figure;
hButton = uicontrol(hFig,... % this UI object lies under hFig
    'Style','pushbutton',... % it is a pushbutton
    'Units','normalized',... % the unit is relative position in the figure window
    'Position',[0.6,0.5,0.4,0.2],... % bottom-left corner at (0.6,0.5) w=0.4, h=0.2
    'String','Push Me!' ...
    );
end
```

Now,

```
>> gui_example
```



You can push the button but nothing would happen. There is no callback function set to this UI control.

7.3 Callback functions

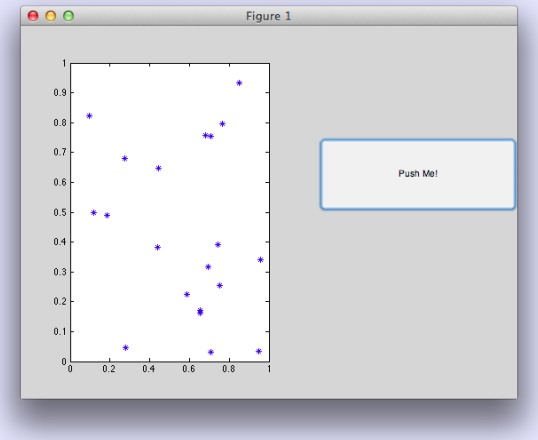
Example 7.2. Let us create an axes at the left half of the figure window. Now add a callback function to the pushbutton

gui_example.m

```
function gui_example
hFig = figure;
hButton = uicontrol(hFig,...
    'Style','pushbutton',...
    'Units','normalized',...
    'Position',[0.6,0.5,0.4,0.2],...
    'String','Push Me!', ...
    'CallBack',@buttonCallBack ...
    );
hAxes = axes(...
    'Parent',hFig,...
    'Units','normalized',...
    'Position',[0.1,0.1,0.4,0.8]);
end

function buttonCallBack( uiobj, eventdata )
plot(rand,rand,'*')
hold on
end
```

When you push the button, the function `buttonCallBack` is executed, with its two input argument `uiobj == hButton` (the object that calls the function) and a struct `eventdata` (in this case `eventdata = []`)

```
>> gui_example
```



A callback function for a UI control takes two arguments

1. An object handle – This is the handle of the UI control object that calls this callback function.

2. A structure – This is the event data which records what kind of events that triggers this function.

Example 7.3 (Keyboard detection)**.** The property `'WindowKeyPressFcn'` can be set to a callback function. In the following example, the callback function is an anonymous function.

```
figure('WindowKeyPressFcn',@(~,eventdata) display(eventdata))
```

It will create a figure which detects keyboard input. As you press a key on your keyboard, you see the value of eventdata.

In a more complicated callback function for keyboard detection, you may have a switch statement like

```
switch eventdata.Key
    case 'escape'
        ...
    case 'a'
        ...
end
```

Example 7.4 (Value of a slider)**.** The following example creates a slider. When you change the state of the slider, the `'Value'` property changes. You may want to use the value in the callback function, in which case you use `get(obj,'Value')` where `obj` is the handle to the slider, the first input argument in the callback function.

```
uicontrol('Style','slider',...
    'Units','normalized',...
    'Position',[0,0.5,1,0.1],...
    'CallBack', @(obj,~) display(get(obj,'Value')) )
```

7.4 Keyboard detection

First, see Example 7.3. The figure window property `'WindowKeyPressFcn'` is by default set to the empty callback string `''`. If you set that to a callback function, say `myKeyFcn(~,eventdata)`, such function will be executed when you press a key on the keyboard, with `eventdata.Key` being the key that is pressed.

Example 7.5. The following GUI shows an animation of propagating sine wave, and the user is allowed to control the wave and animation by

- left arrow – increase the frequency
- right arrow – decrease the frequency
- up arrow – increase the amplitude
- down arrow – decrease the amplitude
- spacebar – pause/play the animation

- escape – close the window

The main function will

1. create a figure and an axes.
2. declare all flags (for denoting whether the animation is paused) and variables (frequency, amplitude).
3. set the `'WindowKeyPressFcn'` callback.
4. jump into infinite loops, which include the animation and a small pause `pause(0.01)`, which is the small break that allows UIs to detect events.

wave_viewer

```
function wave_viewer
hFig = figure;                              % 1. Create figures and axes
hAxes = axes;

flag.play = true;                           % 2. Declare flags and variables
flag.quit = false;

wave.freq = 1;
wave.amp = 1;
t = 0;
x = linspace(-5,5);

set(hFig,'WindowKeyPressFcn',@myKeyFcn) % 3. Set the keyboard callback

while ~flag.quit                            % 4. Jump into infinite loops
    while flag.play
        plot(x, wave.amp * sin(wave.freq*(x-t)) );
        axis([-5,5,-2,2])
        drawnow
        t = t + 0.05;
        pause(0.01)
    end
    pause(0.01)
end
delete(hFig);


    function myKeyFcn( ~ , eventdata )
        switch eventdata.Key
            case 'leftarrow'
                wave.freq = wave.freq - 0.1;
            case 'rightarrow'
                wave.freq = wave.freq + 0.1;
            case 'uparrow'
                wave.amp = wave.amp + 0.1;
            case 'downarrow'
                wave.amp = wave.amp - 0.1;
            case 'space'
```

```
                flag.play = ~flag.play;
            case 'escape'
                flag.quit = true;
        end
    end
end
```

7.5 Mouse Detection

The following GUI serves as a great example for mouse detection.

Example 7.6 (Drawing board)**.** The following function allows a user to draw curves freely on an axes.

drawingGUI

```
function drawingGUI

% Creates all handles
hFIG = figure;
hAXES = axes;
hPLOT{1} = plot(nan);  % hPLOT will eventually be a cell array
                       % each cell element is a handle of a curve
hold on
axis equal
title('Draw a curve!')
set(hAXES,'DrawMode','fast')
set(hAXES,'Box','on','XLimMode','manual','YLimMode','manual')
set(hPLOT{1},'LineWidth',2)

% Declare variables
plot_count = 1;  % plotting the i-th curve
x = [];          % coordinates for the current curve
y = [];


% Default callbacks
set(hFIG,'WindowButtonDownFcn',@startDrawing);
set(hFIG,'WindowButtonUpFcn','');
set(hFIG,'WindowButtonMotionFcn','');


    function startDrawing(src,evnt)
        set(hFIG,'WindowButtonMotionFcn',@mouseMovingCallBack);
        set(hFIG,'WindowButtonUpFcn',@stopDrawing);
        drawAPoint
    end

    function stopDrawing(scr,evnt)
        set(hFIG,'WindowButtonMotionFcn','')

        % get ready for the next plot
```

```
        plot_count = plot_count+1;
        hPLOT{plot_count} = plot(nan);
        x = [];
        y = [];
        set(hPLOT{plot_count},'LineWidth',2,'Color',rand(1,3))
    end

    function mouseMovingCallBack(src,evnt)
        drawAPoint
    end

    function drawAPoint
        cp_full = get(hAXES,'CurrentPoint'); % the full 3D coordinates
        cp = cp_full(1,[1,2]);  % the 2D coordinate

        x = [x;cp(1)];
        y = [y;cp(2)];
        set(hPLOT{plot_count},'XData',x,'YData',y)
        drawnow
    end
end
```

To detect mouse button clicks and mouse motions, it is the properties

```
'WindowButtonDownFcn'
'WindowButtonUpFcn'
'WindowButtonMotionFcn'
```

of a figure to be set to some callback function handles. These properties are the items you will look up in `docsearch figure properties` if you want to learn more about various kind of mouse manipulations.

To detect the mouse coordinate in an axes, call

```
get(gca,'CurrentPoint')
```

which will return a 2×3 matrix $\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix}$, the two 3D coordinates on the bounding box of the 3D axes that coincides with the mouse cursor. 2D axes are 3D axes viewed from the top, so the 2D coordinate of the mouse cursor is simply the components (x_1, y_1).

7.6 GUIDE Tool

In every of above examples, one writes the GUI from scratch. An alternative of designing a GUI is to use the GUIDE tool.

Type

```
>> guide
```

and a GUI design wizard appears. You can design your own GUI with a MATLAB GUI interface. After having all your GUI components settled, save the file, which will generate a `.m` file and a `.fig` file. In the m-file, a halfway complete function for the GUI is written. All you need to do is to fill in the callback functions opened for you.

Using these GUI tools appropriately, you can be a productive MATLAB GUI designer .