School of Electrical, Electronic and Computer Engineering

Lecture Notes for Matlab Introduction

September 2011

Charalampos C. Tsimenidis

charalampos.tsimenidis@ncl.ac.uk

Module webpage

Contents

1 Introduction

Matlab (=Matrix laboratory) is an interactive matrix-based software package intended for complex scientific computations. Presently, Matlab is the *de facto* software used in simulation of communications systems and digital signal processing. It has several andvantages over high-level programming languages, such as C/C++, Fortran etc.

Advantages:

- A large set of toolboxes is available. A toolbox is a collection of matlab functions specific for a subject, e.g. the signal processing toolbox or the communications toolbox.
- At any time, variables (results of simulations) are stored in the workspace for debugging and inspection,
- Excellent visualisation (plots) capabilities,
- Easy programming, e.g. it is not required to define variable types (unless needed). All variables are usually of type `double` (64 bit representation).
- Quick code development.

Disadvantages:

- Very expensive for non students, although there are some free *clones*, such as Octave or Sscilab that are matlab compatible (but not 100%).
- Code execution can be slow if programmed *carelessly* without vectorisation.
- Algorithms developed in Matlab will need to be translated in C or assembly if to be used in DSPs.

1.1 Getting Started

1.1.1 Starting the Matlab Integrated Development Environment (IDE)

Currently, Matlab 2010a is available in the Merz Court PC cluster with licenses for both the communications and signal processing toolboxes. To start the Matlab IDE go to and select from the windows start menu:

Start/All Programs/MATLAB/R2011a/MATLAB 2011a

1.1.2 Input/Output

- To create an m-file select File ⟶ New ⟶ M-File from the main menu. Enter data and commands in the built-in editor. To run m-file select Debug ⟶ Run menu or press `F5`
- The command prompt (>>) can be also used to enter data, run built-in Matlab functions and other m-files, and display results.
  ```
  >>x=[0 1 2 3]
  ```
- To suppress output end a statement with a semicolon (;) and then press `Enter` or `Return`. Matlab runs the statement without displaying any output, which useful when you generating large matrices.
  Compare: `>>A=randn(100);` and `>>A=randn(100)`
- Entering multiple functions in a line: `clc,clear,close all,x=sqrt(4);y=cos(2*pi);`
- Line continuation (for long statements):
  ```
  >>x=cos(2*pi)+...
  >>0.1*sin(2*pi/2)+...
  >>0.01*sin(2*pi/2)
  ```

- Keeping session logs:
  ```
  >>diary('sim1.m')
  >>clear,clc,close all
  >>t=0:10;
  >>x=cos(2*pi*t);
  >>diary('off')
  ```
  To open the file: `>>edit sim1.m`
- Output format may controlled using the `format` command: `>>format long` or `>>format short`

1.1.3 Essential PDF Guides

Getting Started
Matlab Programming
Communications Toolbox User's Guide
Signal Processing Toolbox User's Guide

1.1.4 Online Help on the Command Prompt

Examples: (to type in matlab command prompt followed by `enter`)

- To check how the `help` command works: `>> help help`
- To check toolboxes available: `>> help`
- To check functions available in the communications toolbox: `>> help comm`
- To check how the function `randint` works: `>> help randint`
- To check functions available in the signal processing toolbox: `>> help signal`
- To check how the function `fft` works: `>> help fft`

Type `help helptools` for more details.

1.1.5 Online Help on the Help Window

Examples: (to type in matlab command prompt followed by `enter`)

- To check functions available in the communications toolbox: `>> helpwin comm`
- To check how the function `randint` works: `>> helpwin randint`
- To check functions available in the signal processing toolbox: `>> helpwin signal`
- To check how the function `fft` works: `>> helpwin fft`

`doc` command works similarly but uses `html` display: `>>doc fft`

Type `doc helptools` for more details.

1.1.6 Toolboxes

Table of the most relevant matlab toolboxes.

Basic Toolboxes	Description
general	General purpose commands.
ops	Operators and special characters.
lang	Programming language constructs.
elmat	Elementary matrices and matrix manipulation.
elfun	Elementary math functions.
specfun	Specialized math functions.
matfun	Matrix functions & numerical linear algebra.
datafun	Data analysis and Fourier transforms.
polyfun	Interpolation and polynomials.
funfun	Function functions and ODE solvers.
sparfun	Sparse matrices.
scribe	Annotation and Plot Editing.
graph2d	Two dimensional graphs.
graph3d	Three dimensional graphs.
specgraph	Specialized graphs.
graphics	Handle Graphics.
uitools	Graphical user interface tools.
strfun	Character strings.
imagesci	Image and scientific data input/output.
iofun	File input and output.
audiovideo	Audio and Video support.
timefun	Time and dates.
datatypes	Data types and structures.
verctrl	Version control.
codetools	Commands for creating and debugging code.
helptools	Help commands.
demos	Examples and demonstrations.
timeseries	Time series data visualization and exploration.
Specialised Toolboxes	
comm	Communications Toolbox.
commdemos	Communications Toolbox Demonstrations.
commdocdemos	Communications Toolbox Documentation Examples.
signal	Signal Processing Toolbox.
sigtools	Signal Processing Utilities.
sptoolgui	Signal Processing GUI-Based Utilities.
sigdemos	Signal Processing Demos.

Type `doc toolbox_name` to see more detail about a specific toolbox.

2 Guided Tutorial

2.1 Scalar Variables

2.1.1 Working with Scalar Variables

☐ `>>` means to be typed on command prompt and hit the return/enter key.

☐ All scalar variables in Matlab are stored in `double` precision (64 bits) unless explicitly specified.

☐ Defining scalar variables:

```
>> x=2
x =
     2
```

☐ Semicolon (;) can be used to suppress displaying of results in command prompt:

```
>> w=3;
```

☐ To check the value of the variable w after entering:

```
>> w
w =
     3
```

☐ Multiple variables can be typed simultaneously by using comma separation:

```
>> w=4, y=1, z1=3, z2=5.5
w =
     4
y =
     1
z1 =
     3
z2 =
    5.5000
```

2.1.2 The Matlab Workspace

☐ To list variables currently in workspace (short form):

```
>> who
Your variables are:
w   x   y   z1  z2
```

☐ To list variables currently in workspace (detailed form):

```
>> whos
Name      Size             Bytes  Class     Attributes

w         1x1                  8  double
x         1x1                  8  double
y         1x1                  8  double
z1        1x1                  8  double
z2        1x1                  8  double
```

☐ To clear specific variables in the work space, for example w and x:

```
>> clear w x
>> who
Your variables are:
y   z1  z2
```

☐ To clear variables starting with z:

```
>> clear z*
>> who
Your variables are:
y
```

☐ To clear all the variables currently in the work space:

```
>> clear all
```

Alternatively, variables can be inspected and deleted using the IDE.

☐ To clear the command-prompt window.

```
>> clc
```

This will only the command prompt, but not the workspace.

2.1.3 Special Variable Names and Constants: `ans, pi, eps, i, j, inf, nan`

☐ `ans`: Most recent answer.

```
>> 3
ans =
     3
```

☐ `pi`: $\pi = 3.141592653589793$

```
>> pi
ans =
   3.141592653589793
```

☐ `eps`: Smallest floating point number of the working computer.

```
>> eps
ans =
     2.220446049250313e-16
```

☐ `i, j`: Imaginary unit.

```
>> x=3+j*6;
```

☐ `Inf` or `inf`: Infinity (∞)

```
>> 1/0
ans =
   Inf

>> -1/0
ans =
  -Inf
```

☐ `NaN` or `nan`: *not a number*. This variable can be used to check for undefined numerical results such as $0 \cdot \infty$ or $\frac{\infty}{\infty}$

```
>> inf/inf
ans =
   NaN

>> 0*inf
ans =
   NaN
```

Tip: when programming overwriting/using this variables and constants should be avoided.

2.1.4 Built-in Functions

There is a vast amount of built-in functions in Matlab ranging from elementary mathematical functions to higher level. In contrast to other programming languages such as C most of the functions accept complex arguments and can be effortlessly applied to vectors and matrices.
☐ To inspect what is available:

```
>> doc elfun
>> doc specfun
```

and then click on specific functions (cos, sin, etc.). Alternatively, you can use `help` command:

```
>> help elfun
```

☐ Examples:

```
>> exp(1+j)
ans =
   1.4687 + 2.2874i

>> cos(pi/4)
ans =
    0.7071
```

```
>> atan(1)
ans =
    0.7854

>> erfc(2)
ans =
    0.0047

>> besselj(0,2)
ans =
    0.2239

>> sinc(pi)
ans =
   -0.0436
```

2.1.5 Operators and Expressions

☐ Commonly used operators for scalar variables:

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power

Addition has the lowest priority and Power the highest. Type `help +` to obtain more details.

☐ Examples of expressions:

```
>> mag=abs(1+j)
mag =
    1.4142

>> phi=angle(1+j)*180/pi
phi =
    45

>> w=sqrt(2)*exp(-0.2)
w =
    1.1579

>> a=3;
>> b=2;
>> c=sqrt(1+2*a+b/3+0.5*b^2)
c =
    3.1091
```

☐ Line continuation (for long statements):

```
>> x=cos(2*pi)+...
>> 0.1*sin(2*pi/3)+...
>> 0.01*sin(2*pi/3)
x =
    1.0953
```

Review Exercises 1

1. The discriminant of the quadratic polynomial $ax^2 + bx + c = 0$ is given as $\Delta = b^2 - 4ac$. Use matlab to find the discriminant of $2x^2 - 3x + 5 = 0$. Solution: `Delta = -31`

2. Compute the following values:

$$\cos(\frac{2\pi}{3}),\ \sin(\frac{\pi}{8}),\ \cos^{-1}(0.5),\ \tan^{-1}(\frac{\pi}{2})$$

   Solutions:
   `-0.5000, 0.3827, 1.0472, 1.0039`

3. Find the complex value, magnitude and phase (deg) of the folowing complex fraction:

$$\frac{(1+2j)(3+5j)}{8+j},\ \frac{(3+2j)(1+j)}{(1+3j)(2+j)^*}$$

   Solutions:
   `-0.6923 + 1.4615i, 1.6172, 115.3462` and
   `0.6000 + 0.4000i, 0.7211, 33.6901`

4. Find the complex value, magnitude and phase (deg) of the following 1st order lowpass transfer function at $f = 150$ Hz if $f_c = 100$ Hz:

$$H(f) = \frac{1}{1 + j\dfrac{f}{f_c}}$$

   Solutions:
   `H = 0.3077 - 0.4615i, Hmag = 0.5547, Hphase = -56.3099`

5. Convert the magnitude value of H in the previous example to dB (decibels).
   Solution:
   `-5.1188` (dB)

6. The s-parameters of a microwave transistor amplifier are given as:

$$s_{11} = 0.61e^{j165^o},\ s_{21} = 3.72e^{j59^o},\ s_{12} = 0.05e^{j42^o},\ s_{22} = 0.45e^{-j48^o}$$

   (a) Compute the stability factor:

$$K = \frac{1 - |s_{11}|^2 - |s_{22}|^2 + |D|^2}{2|s_{12}||s_{21}|}$$

   where $D = s_{11}s_{22} - s_{12}s_{21}$

   Solution: `K = 1.1752`

   (b) If $Z_0 = 50\Omega,\ Z_s = 10 + j20\Omega,\ Z_L = 30 - j40\Omega$ find the available power gain defined as:

$$G_A = \frac{|s_{21}|^2(1 - |\Gamma_S|^2)}{(1 - |s_{22}|^2) + |\Gamma_S|^2(|s_{11}|^2 - |D|^2) - 2\text{Re}\{\Gamma_S(s_{11} - Ds_{22}^*)\}}$$

   where Γ_S is output reflection coefficient defined as:

$$\Gamma_S = \frac{Z_S - Z_0}{Z_0 + Z_S}$$

   Solution: `GA = 11.4361`

2.2 Working with Vectors and Matrices

2.2.1 Entering Vector and Matrices

☐ Matrix and vector indexing starts in Matlab with 1.
☐ Matrices can be entered in Matlab via different ways:

- ■ Entering an explicit list of elements:

```
>> x=[1 2 3 4]
x =
     1     2     3     4

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

- ■ Generating matrices using built-in functions:

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10

>> A=randn(4)
A =
   -0.1567    1.4151   -0.9219    0.6145
   -1.6041   -0.8051   -2.1707    0.5077
    0.2573    0.5287   -0.0592    1.6924
   -1.0565    0.2193   -1.0106    0.5913
```

- ■ Create matrices with your own functions in m-files (see later).
- ■ Load matrices from external data files (see `load` function later).

2.2.2 The Colon Operator :

☐ Unit spacing:

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
```

☐ Non-unit spacing:

```
>> y=-0.3:0.1:0.3
y =
   -0.3000   -0.2000   -0.1000         0    0.1000    0.2000    0.3000
```

☐ Negative non-unit spacing:

```
>> z=0.5:-0.1:0
z =
    0.5000    0.4000    0.3000    0.2000    0.1000         0

>> w=0:pi/3:2*pi
w =
         0    1.0472    2.0944    3.1416    4.1888    5.2360    6.2832
```

2.2.3 Indexing Vectors and Matrices

☐ To access the 5th element of the vector `x`:

```
>> x(5)
ans =
     5
```

☐ To access the last element of the vector `x`:

```
>> x(10)
ans =
    10

>> x(end)
ans =
    10
```

☐ To access the elements 3 to 6 of the vector `x`:

```
>> x(3:6)
ans =
     3     4     5     6
```

☐ To access the elements 1 2 4 8 of the vector `x`:

```
>> x([1 2 4 8])
ans =
     1     2     4     8
```

☐ Indexes outside the valid range of vector/matrix elements generate typical error messages:

```
>> x(20)
??? Index exceeds matrix dimensions.
```

☐ Assign values on specific vector elements, for example 20 to the 5th element of `x`:

```
>> x(5)=20
x =
     1     2     3     4    20     6     7     8     9    10
```

☐ Syntax of other possible assignments:

```
>> x(3:7)=[-5 -4 -3 -2 -1]
x =
     1     2    -5    -4    -3    -2    -1     8     9    10

>> x(5:6)=randn(1,2)
x =
  Columns 1 through 8
    1.0000    2.0000   -5.0000   -4.0000   -0.6436    0.3803   -1.0000    8.0000

  Columns 9 through 10
    9.0000   10.0000
```

☐ To access the elements of the matrix `A`, for example `A(2,3)` corresponds to element of the 2nd row and 3rd column of `A`:

```
>> A(2,3)
ans =
   -2.1707
```

☐ Concatenation: matrices can be concatenated by using the `[]` operator. However, this can be inefficient and time consuming for large matrices.

```
>> x=1:5; y=x+5; z=y+5;
>> A=[x;y;z]
A =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15

>> B=[A A; A A]
B =
     1     2     3     4     5     1     2     3     4     5
     6     7     8     9    10     6     7     8     9    10
    11    12    13    14    15    11    12    13    14    15
     1     2     3     4     5     1     2     3     4     5
     6     7     8     9    10     6     7     8     9    10
    11    12    13    14    15    11    12    13    14    15
```

☐ Concatenation is useful for building tables:

```
>> t=[1:10].';
>> [t t.^2 t.^3]
ans =
           1           1           1
           2           4           8
           3           9          27
           4          16          64
           5          25         125
           6          36         216
           7          49         343
           8          64         512
           9          81         729
          10         100        1000
```

☐ Deleting rows:

```
>> A(1,:)=[]
A =
     6     7     8     9    10
    11    12    13    14    15
```

☐ Deleting columns:

```
>> A(:,1)=[]
A =
     7     8     9    10
    12    13    14    15
```

☐ Reshaping using the `reshape` command:

```
>> x=1:9
x =

     1     2     3     4     5     6     7     8     9
>> A=reshape(x,3,3)
A =
     1     4     7
     2     5     8
     3     6     9
>> y=reshape(A,9,1)
y =
     1
     2
     3
     4
     5
     6
     7
     8
     9

>> y=reshape(A,2,4) % example of wrong usage
??? Error using ==> reshape
To RESHAPE the number of elements must not change.
```

2.2.4 Vector and Matrix Sizes

☐ The length of a vector can be checked via the `length` command:

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10
>> length(x)
ans =
    10
```

☐ The size of a matrix can be checked via the `size` command:

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

>> size(A)
ans =
     2     5
```

☐ The `size` command can also be applied to vectors:

```
>> size(x)
ans =
     1    10
```

☐ The `length` command can also be applied to matrices. It returns the larger of the two dimensions of a matrix.

```
>> length(A)
ans =
     5
```

2.2.5 Vector and Matrix Generating Functions

Matlab provides several functions that generate basic matrices:
☐ `zeros()`

```
>> x=zeros(1,10)
x =
     0     0     0     0     0     0     0     0     0     0

>> x=zeros(3,1)
x =
     0
     0
     0

>> A=zeros(3)
A =
     0     0     0
     0     0     0
     0     0     0

>> B=zeros(2,6)
B =

     0     0     0     0     0     0
     0     0     0     0     0     0
```

☐ `ones()` works similarly, but generates all-one elements:

```
>> x=ones(2,10)
x =
     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1
```

☐ `rand()` generates random numbers uniformly distributed between [0, 1]:

```
>> A=rand(2,6)
A =
    0.7836    0.4490    0.1956    0.6162    0.4001    0.0702
    0.8887    0.9412    0.0481    0.4716    0.9274    0.7462
```

☐ `randn()` generates random numbers that are Gaussian distributed with mean 0 and standard deviation 1:

```
A =
   -0.4326    0.1253   -1.1465    1.1892    0.3273   -0.1867
   -1.6656    0.2877    1.1909   -0.0376    0.1746    0.7258
```

☐ `randsrc()` generates random numbers taken from a specific set:

```
>> A=randsrc(3,4,[-2 0 2])
A =
    -2     2    -2    -2
     2    -2     2    -2
    -2    -2     0    -2
```

There are many more matrix-generating functions that you will discover while working in Matlab but we can not cover here.

2.2.6 Vector and Matrix Operations

`'`	Transpose if matrix is real: $\mathbf{A}^T$ or Hermitian transpose if matrix is complex: $(\mathbf{A}^*)^T$,
`.'`	Transpose if matrix is real or complex: $\mathbf{A}^T$.
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`.*`	Element-by-element multiplication
`/`	Division
`./`	Element-by-element division
`^`	Power
`.^`	Element-by-element power

Type `doc datatypes` for more details.

☐ Transpose of vectors and matrices:

```
>> x=1:4
x =
     1     2     3     4

>> x'            % or x.' or transpose(x)
ans =
     1
     2
     3
     4

>> v=[1-j 2+j]
v =
   1.0000 - 1.0000i   2.0000 + 1.0000i

>> v.'
ans =
   1.0000 - 1.0000i
   2.0000 + 1.0000i

>> v'
ans =
   1.0000 + 1.0000i
   2.0000 - 1.0000i
```

Note the change of the sign of the imaginary part of `v(2,1)` for the Hermitian transpose.

```
>> A=[1:3; 4:6]
A =
     1     2     3
     4     5     6
>> size(A)
ans =
     2     3

>> B=A.'
B =
     1     4
     2     5
     3     6

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

☐ Addition/subtraction of vectors and matrices:

```
>> x=1:4
x =
     1     2     3     4

>> y=5:8
y =
     5     6     7     8

>> z1=x+y
z1 =
     6     8    10    12

>> z2=x-y
z2 =
    -4    -4    -4    -4

>> A=[x;y;z]
A =
     1     2     3     4
     5     6     7     8
     6     8    10    12

>> B=ones(3,4)
B =
     1     1     1     1
     1     1     1     1
     1     1     1     1

>> C=A+B
C =
     2     3     4     5
     6     7     8     9
     7     9    11    13

>> D=A-B
D =
     0     1     2     3
     4     5     6     7
     5     7     9    11
```

☐ Addition/subtraction of a constant to a vector or matrix:

```
>> w1=x+2
w1 =
     3     4     5     6

>> w=x-1
w =
     0     1     2     3

>> C=A+3
C =
     4     5     6     7
     8     9    10    11
     9    11    13    15

>> D=B-1
D =
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

Note: the constant is added/subtracted to all elements of `w` and `A`.

☐ Multiplication of vectors and matrices:

```
>> x*y
 ??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Inner dimensions of vectors must be correct for multiplication, e.g. (1xN)*(Nx1) or (Nx1)*(1xN):

```
>> transpose(x)*y       % or x'*y
ans =
     5     6     7     8
    10    12    14    16
    15    18    21    24
    20    24    28    32

>> x*transpose(y)       % or x*y'
ans =
    70
```

The later is equal to the `dot` product of two vectors. Compare with:

```
>> dot(x,y)
ans =
    70
```

Inner dimensions of matrices must be the same for multiplication, e.g. (MxN)*(NxM) or (NxM)*(MxN):

```
>> A*B
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> A*transpose(B)       % or A*B'
ans =
    10    10    10
    26    26    26
    36    36    36

>> transpose(A)*B       % or A'*B
ans =
    12    12    12    12
    16    16    16    16
    20    20    20    20
    24    24    24    24
```

☐ Element-by-element multiplication/division:

```
>> x=1:10
x =
     1     2     3     4     5     6     7     8     9    10

>> y=10:-1:1
y =
    10     9     8     7     6     5     4     3     2     1

>> x.*y
ans =
    10    18    24    28    30    30    28    24    18    10
```

```
>> x./y
ans =
    0.1000    0.2222    0.3750    0.5714    0.8333    1.2000    1.7500    2.6667
       4.5000   10.0000

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

>> B=A.*A
B =
     1     4     9
    16    25    36
    49    64    81

>> B=A./A
B =
     1     1     1
     1     1     1
     1     1     1
```

☐ Matrix division for square matrices:

```
>> A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

>> A/A
ans =

     1     0     0
     0     1     0
     0     0     1

>> inv(A)*A
ans =
    1.0000         0   -0.0000
         0    1.0000         0
         0    0.0000    1.0000
```

☐ Power of a matrix:

```
>> A^2          % same as A*A
ans =
    91    67    67
    67    91    67
    67    67    91
```

☐ Power aplied elementwise:

```
>> A.^2
ans =
    64     1    36
     9    25    49
    16    81     4

>> A.^0.5                % same as sqrt(A)
ans =
    2.8284    1.0000    2.4495
    1.7321    2.2361    2.6458
    2.0000    3.0000    1.4142
```

2.2.7 Polynomials

☐ Polynomials are represented in Matlab as one-dimensional arrays. For example, the following polynomials

$$\begin{aligned} p_1(x) &= x^2 + 5x + 2 \\ p_2(x) &= x^4 + 1 \\ p_3(x) &= x^3 + 5jx + 1 \end{aligned}$$

are represented in Matlab as:

```
>> p1=[1 5 2];
>> p2=[1 0 0 1];
>> p3=[1 0 5*j 2];
```

☐ Polynomial roots:

```
>> roots(p1)
ans =
   -4.5616
   -0.4384
>> roots(p2)
ans =
  -1.0000
   0.5000 + 0.8660i
   0.5000 - 0.8660i
```

☐ Polynomial multiplication: $p_4(x) = p_1(x)p_2(x) = x^6 + 5x^5 + 2x^4 + x^2 + 5x + 2$

```
>> p4=conv(p1,p2)
p4 =
     1     5     2     1     5     2
```

☐ Polynomial evaluation: $p_1(0.5)$ and $p_3(j)$

```
>> polyval(p1,0.5)
ans =
    4.7500
>> polyval(p3,j)
ans =
  -4.0000 - 1.0000i
```

☐ The second argument of `polyval()` can be also a vector

```
>> polyval([1 0 0],[0:4])
ans =
     0     1     4     9    16
```

☐ Fit polynomial to data using `polyfit()`.

```
clear, clc, close all
N=20;
t=0:N-1;
y=0.5*t+0.1*randn(1,N); % Noisy data model
p=polyfit(t,y,1); % Find 1st order polynomial to best fit data
y_f=polyval(p,t); % Compute fitted data
plot(t,y,'ro',t,y_f)
xlabel('t')
ylabel('y(t),  y_f (t)')
str2=strcat('y_f (t)  =',num2str(p(1),2),'+', num2str(p(2),1),' (fitted)');
legend('y(t)',str2,2)
```

$\boxed{\text{Review Exercises 2}}$

1. Using the colon operator `:`, create a vector that contains only the even integers between 11 and 31.

2. Let `x=0:9` and `y=0.1*[0:9]`

   (a) Add 5 to only the elements of `x` that have an odd index, e.g. 1, 3, 5, etc.
   (b) Add 10 only to the elements of `x` that have an even index, e.g. 2, 4, 6, etc.
   (c) Compute the square and square root of each element in x and y.
   (d) Raise each element of `x` to the power specified by the corresponding element in `y`.
   (e) Divide each element of `y` by the corresponding element in `x`.
   (f) Multiply `x` and `y` element-by-element, calling the result `z`.
   (g) Sum-up the elements in `z` and assign the result to a variable called `w`.
   (h) Compute `x*y'-w` and interpret the result.

3. Generate a 10-by-10 Gaussian random matrix.

   (a) Overwrite the 5th column with the value 3.
   (b) Delete the 5th column.
   (c) Overwrite the 6th row with the value 2.
   (d) Delete the 6th row.

4. Solve using Matlab the following system of equations, e.g. find x, y and z

$$\underbrace{\begin{pmatrix} 1.0 & 0.1 & 0.2 \\ 0.1 & 1.0 & 0.1 \\ 0.2 & 0.1 & 1.0 \end{pmatrix}}_{\mathbf{R}} \underbrace{\begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{\mathbf{a}} = \underbrace{\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}}_{\mathbf{v}}$$

   (a) Enter the matrix $\mathbf{R}$ and the vector $\mathbf{v}$,
   (b) Find the inverse $\mathbf{R}^{-1}$ of $\mathbf{R}$ using the `inv` function,
   (c) The solution is found by computing the product $\mathbf{a} = \mathbf{R}^{-1}\mathbf{v}$,
   (d) Find the determinant, eigenvalues and eigenvectors of $\mathbf{R}$. Use `lookfor` to find the corresponding built-in functions.

5. Generate using `randn()` a random Gaussian distributed 10^4-by-1 column vector `x` with mean 5 and standard deviation 2. Verify the result by computing the mean (`mean()`) and the standard deviation (`std()`) of `x`. Plot using the `stem` command the first 100 samples.

6. Generate 50 ms of the sinewave below that is sampled with a frequency of $f_s = 48$ kHz.

$$y(t) = A\sin(2\pi f_c t + \phi)$$

   where $A = \sqrt{2}$, $f_c = 100$ Hz, and ϕ is a random phase uniformly distributed between $[0, 2\pi]$.

   (a) Using the colon operator or the `linspace` command, generate the correct time samples based on duration and sampling frequency.
   (b) Generate the random phase using the `rand` function along with appropriate scaling.
   (c) Compute `y` using the formula above.
   (d) Display the sine wave using the correct time axis via the `plot` command.
   (e) Find the mean, standard deviation and variance of `y`.

7. Using the `randsrc` command generate 10000 samples of a 4-PAM signal {`-3, -1, 1, 3`}. The 4-PAM symbols are uniformly distributed. Compute the mean, standard deviation and variance of the 4-PAM signal. Repeat the exercise for a 4-QAM signal with constellation points {`1+j, -1+j, -1-j, 1-j`}.

2.3 Plotting

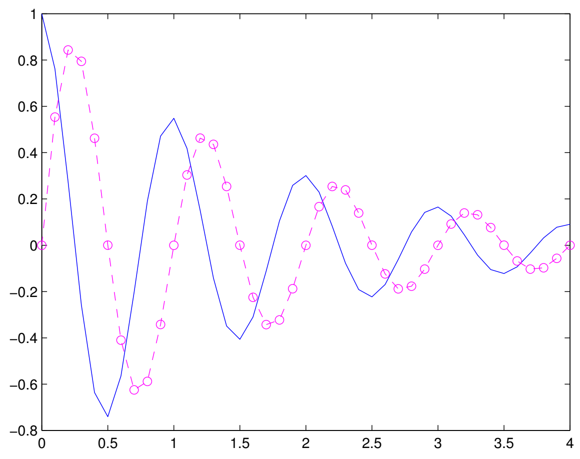
☐ Check how the plot command works via `help plot`

☐ Example: (To be typed and run from a file)

```
clear, clc, close all
t=0:0.1:4;
x=cos(2*pi*t).*exp(-0.6*t);
plot(t,x)
```

☐ The default line color is blue but it can be controlled via an additional argument to `plot()`

```
y=sin(2*pi*t).*exp(-0.6*t);
hold on, plot(t,y,'mo--')
```



☐ The 3rd argument is a string that can be composed as a combination of a line color, marker and line type.

☐ The following line colors are available:

Letter	Line Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

☐ The following line types are available:

Symbol	Line Type
-	solid
:	dotted
-.	dashdot
–	dashed
(none)	no line

☐ The following markers are available:

Symbol	Marker Type
`.`	point
`o`	circle
`x`	x-mark
`+`	plus
`*`	star
`s`	square
`d`	diamond
`v`	triangle (down)
`^`	triangle (up)
`<`	triangle (left)
`>`	triangle (right)
`p`	pentagram
`h`	hexagram

☐ The `hold` command holds the current plot so that subsequent plots are superimposed.

```
>> hold on      % sets the state of hold to on
>> hold off     % sets the state of hold to off
>> hold         % toggles the last state of hold
```

☐ Plot two curves at the same time:

```
clear, clc, close all
t=0:99;
noise1=randn(1,100);
noise2=5+0.2*randn(1,100);
plot(t,noise1,'ro-',t,noise2,'bx--')
```

☐ Add axes labels and title:

```
xlabel('time (s)')
ylabel('Amplitude (V)')
title('Gaussian random noise signals.')
```

☐ Add a legend:

```
legend('mu = 0, sigma = 1', 'mu = 5, sigma = 0.2',4)
```

The `legend` command requires as many string arguments as the curves used in the `plot` command. The last argument sets the position of the legend. Valid values are 1, 2, 3, 4, -1. Try the different values to reposition the legend.

☐ Control the axes ranges via `axis([x_min x_max y_min y_max])`:

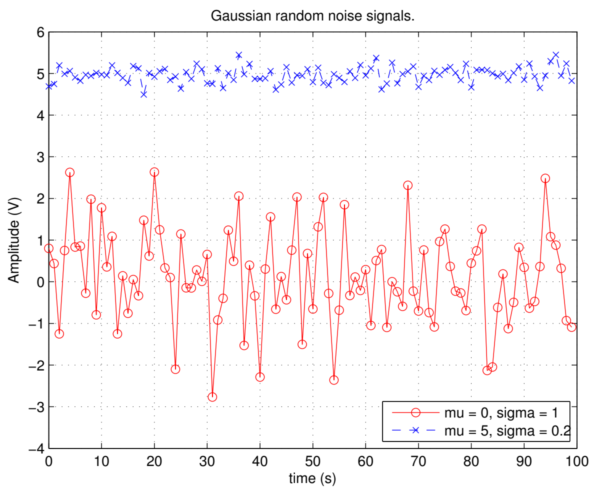
```
axis([0 100 -4 6])
```

☐ Available `axis()` options:

```
>> axis([0 200 -5 10])          % leave space so that legend does not cover graph
>> axis('square')               % reshape plot axes to square
>> axis('off')                  % turn axes off
>> axis('on')                   % turn axes on
>> axis([0 100 -4 6])           % return to normal
```

☐ To set the grid use the `grid` command. Available options:

```
>> grid on                      % turn grid on
>> grid off                     % turn grid off
>> grid                         % toggle last state of grid
```

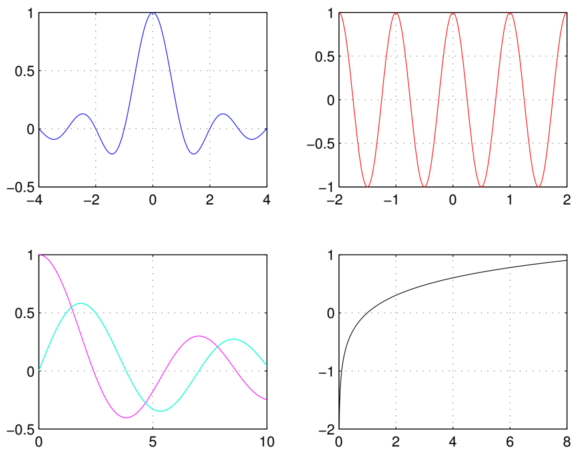


☐ Multiple plots can be created using the `figure()` command.

```
figure(1); plot(randn(1,100))
figure(2); plot(rand(1,100),'r')
```

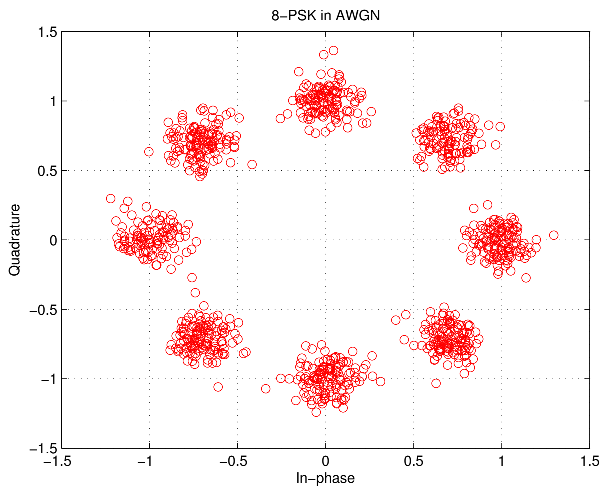
☐ Subplots in the same figure can be created via `subplot(mnk)`, where `m, n` and `k` are integers. The pair `m, n` gives the size of the array of plots to be generated, while `k` is used to index the plots. `k=1` corresponds to the upper-left plot. `k=m*n` corresponds to the lower-right plot. Plots are indexed row-wise for increasing `k`. If the index `k>n`, a new row of plots starts.

```
subplot(221), t=-4:0.01:4; plot(t,sinc(t)), grid
subplot(222), t=-2:0.01:2; plot(t,cos(2*pi*t),'r'), grid
subplot(223), t=0:0.01:10;
plot(t,besselj(0,t),'m',t,besselj(1,t),'g',t,besselj(1,t),'c'), grid
subplot(224), t=0:0.01:8; plot(t,log10(t),'k'), grid
```



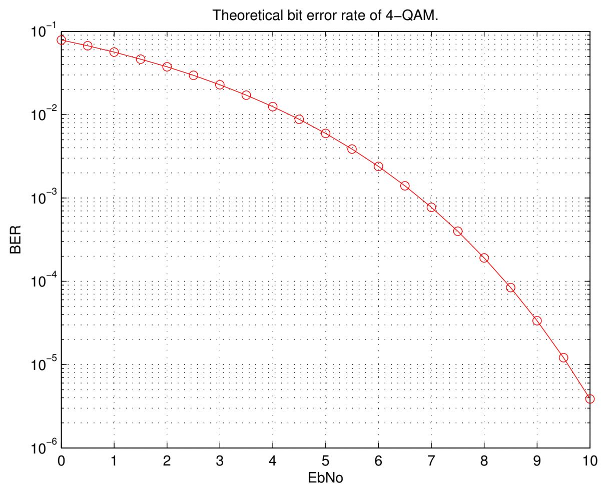
☐ Plotting complex signals. Example 8-PSK (Phase shift keying) in AWGN noise.

```
x=randsrc(1,1000,[exp(j*2*pi*[0:7]/8)])+0.1*randn(1,1000)+j*0.1*randn(1,1000);
plot(x,'ro'), grid
xlabel('In-phase')
ylabel('Quadrature')
title('8-PSK in AWGN')
```



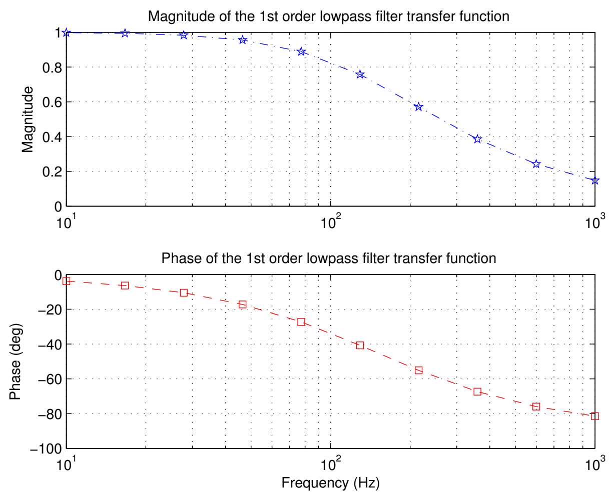
☐ Semilog plot (log y-axis):

```
EbNo=0:0.5:10;
BER=berawgn(EbNo,'QAM',4);
semilogy(EbNo,BER,'ro-'),grid
xlabel('EbNo')
ylabel('BER')
title('Theoretical bit error rate of 4-QAM.')
```



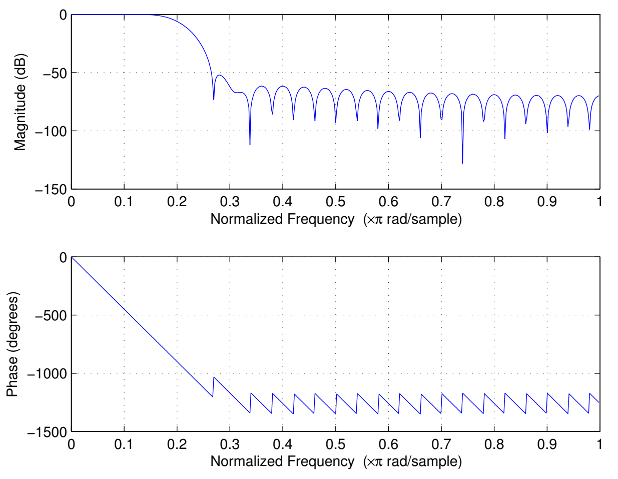
☐ Semilog plots (log x-axis):

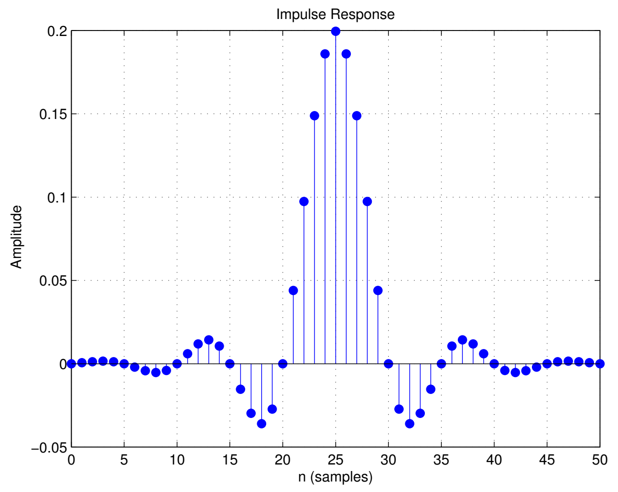
```
f=logspace(1,3,10);
H=1./(1+j*f/150);
Hmag=abs(H);
Hphase=angle(H)*180/pi;
subplot(211), semilogx(f,Hmag,'bp-.'), grid
title('Magnitude of the 1st order lowpass filter transfer function ')
ylabel('Magnitude')
subplot(212), semilogx(f,Hphase,'rs--'), grid
title('Phase of the 1st order lowpass filter transfer function ')
ylabel('Phase (deg)')
xlabel('Frequency (Hz)')
```

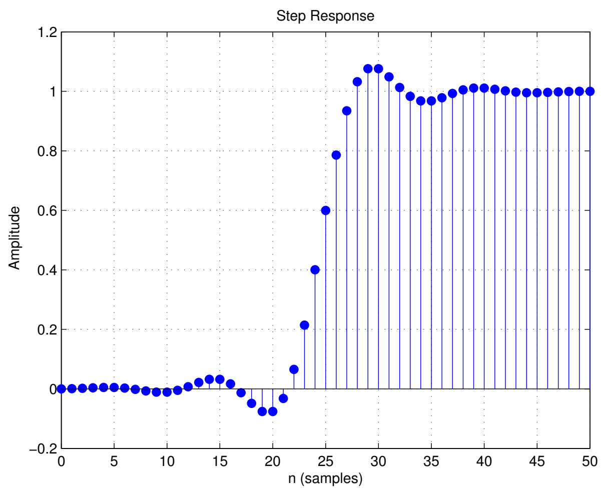


☐ Functions that generate graphical output: `freqz()`, `impz()` and `stepz()`

```
b=fir1(50,0.2);              % Digital FIR filter design: 51 coefficients
figure(1), freqz(b);         % Frequency response
figure(2), impz(b), grid;    % Impulse response
figure(3), stepz(b), grid    % Step response
```



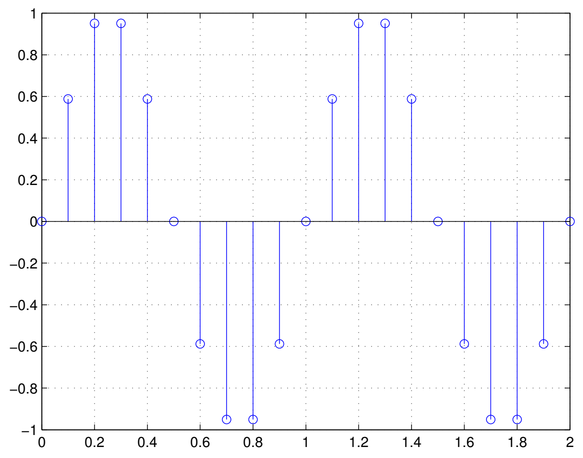




☐ Check also their analogue equivalents: `freqs()`, `imps()` and `step()`.

☐ Another useful plotting function is `stem`:

```
clear , clc,close all
n=0:0.1:2;
x=sin(2*pi*n);
stem(n,x), grid
```



2.4 Matlab Programming

2.4.1 Program Control Statements

☐ Relational operators:

Relational Operator	Description
==	Equal
∼=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

☐ Examples:

```
>> clear, clc
>> x=2;
>> y=3;
>> z=[x==y, x~=y, x<y, x>y, x<=y, x>=y]
ans =
     0     1     1     0     1     0

>> whos
  Name       Size              Bytes  Class      Attributes

  x          1x1                   8  double
  y          1x1                   8  double
  z          1x6                   6  logical
```

Note: `z` is of type `logical` and not `double`.

☐ Logical operators:

Logical Operator	Description
∼	Not
&	Element-wise logical AND
||	Element-wise Logical OR
`xor()`	XOR, Logical EXCLUSIVE OR

☐ Examples:

```
>> [~1, ~0 ~~1 ~~0]
ans =
     0     1     1     0

>> [0|0 0|1 1|0 1|1]
ans =
     0     1     1     1

>> [0&0 0&1 1&0 1&1]
ans =
     0     0     0     1

>> [1 0 1]|[0 1 1]
ans =
     1     1     1

>> [1 0 1]&[0 1 1]
ans =
     0     0     1

>> xor([1 0 1],[0 1 1])
ans =
     1     1     0
```

☐ Syntax of the `if` structure:

```
if  (condition statement)
        matlab commands
        ...
end
```

Reminder of the algorithm to find the roots of the quadratic polynomial: $ax^2 + bx + c = 0$, with $\Delta = b^2 - 4ac$ and $a \neq 0$
If $\Delta < 0$ then

$$x_{1,2} = \frac{-b \pm j\sqrt{-\Delta}}{2a}$$

If $\Delta >= 0$ then

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$$

☐ Simple `if` structure (To be typed in a file and run via F5 or via menu Debug/Run):

```
clear,clc
a=2; b=-3; c=5;
Delta=b^2-4*a*c;
if (Delta<0)
    disp('The polynomial has two complex-conjugate roots:')
    x1=(-b+j*sqrt(-Delta))/(2*a);
    x2=(-b-j*sqrt(-Delta))/(2*a);
end
[x1; x2]
```

☐ After running, the following is displayed in the command prompt:

```
The polynomial has two complex-conjugate roots:
ans =
   0.7500 + 1.3919i
   0.7500 - 1.3919i
```

☐ Generic `if-else` syntax:

```
if  (condition statement)
        matlab commands
        ...
else
        matlab commands
        ...
end
```

☐ Example:

```
clear,clc
a=1; b=3; c=2;
Delta=b^2-4*a*c;
if (Delta<0)
    disp('The polynomial has two complex-conjugate roots:')
    x1=(-b+j*sqrt(-Delta))/(2*a);
    x2=(-b-j*sqrt(-Delta))/(2*a);
else
    disp('The polynomial has two real-valued roots:')
    x1=(-b+sqrt(Delta))/(2*a);
    x2=(-b-sqrt(Delta))/(2*a);
end
[x1; x2]
```

☐ After running, the following is displayed in the command prompt:

```
The polynomial has two real-valued roots:
ans =
    -1
    -2
```

☐ Generic syntax of the `if-elseif-else` structure:

```
if  (condition statement)
        matlab commands
        ...
elseif  (condition statement)
        matlab commands
        ...
elseif (condition statement)
        matlab commands
        ...
else
        matlab commands
        ...
end
```

□ Example:

```
clear,clc
a=1; b=6; c=9;
Delta=b^2-4*a*c;
if (Delta<0)
    disp('The polynomial has two complex-conjugate roots:')
    x1=(-b+j*sqrt(-Delta))/(2*a);
    x2=(-b-j*sqrt(-Delta))/(2*a);
elseif (Delta>0)
    disp('The polynomial has two real-valued roots:')
    x1=(-b+sqrt(Delta))/(2*a);
    x2=(-b-sqrt(Delta))/(2*a);
else
    disp('The polynomial has a double real-valued root:')
    x1=-b/(2*a);
    x2=x1;
end
[x1; x2]
```

□ After running, the following is displayed in the command prompt:

```
The polynomial has a double real-valued root:
ans =
    -3
    -3
```

□ Extend the algorithm further to handle the case `a = 0`.

□ Roots of polynomials can be computed via the `roots()` command. Compare the results of your code with those of `roots()`. Use `doc roots` to see how the command works.

□ `switch`, `case`, and `otherwise`

```
switch switch_expr
 case case_expr
    statement, ..., statement
  case {case_expr1, case_expr2, case_expr3, ...}
    statement, ..., statement
  otherwise
    statement, ..., statement
end
```

□ Example:

```
clear, clc, close all
mod_sch='8PSK';
switch mod_sch
        case 'BPSK'
                Constel=[-1 1];
        case 'QPSK'
                Constel=[1+j -1+j -1-j 1-j];
        case '8PSK'
                Constel=exp(j*2*pi*[0:7]/8);
        otherwise
                error('Wrong modulation scheme !')
end
plot(Constel,'ro'), axis([-1.5 1.5 -1.5 1.5]), grid
```

2.4.2 Loops

□ The `for` loop, generic syntax:

```
for variable = initval:endval
    statement
    ...
    statement
end
```

☐ Example:

```
Data_4QAM=zeros(10);              % Pre-allocate memory for faster processing
for m=1:10
        for n=1:10
                Data_4QAM(m,n)=sign(randn)+j*sign(randn);
        end
end
```

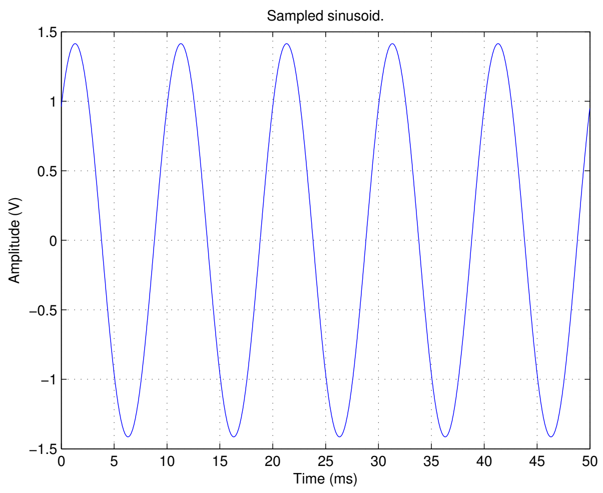
Although this could have been accomplished much easier (see below), sometimes `for` loops are necessary.

```
Data_4QAM=sign(randn(10))+j*sign(randn(10));
```

☐ Example: (Solution for the review exercise 6)

```
clear, clc, close all
A=sqrt(2);              % Amplitude
fc=0.1;                 % Carrier frequency normalised with kHz
Td=50;                  % Duration normalised with ms
fs=48;                  % normalised with kHz
Ts=1/fs;                % Sampling time 1/fs
phi=rand*2*pi;          % Random phase [0 2*pi]
t=0;                    % Starting time
L=Td*fs;                % Length of the resulting vector
y1=zeros(1,L);          % Pre-allocate memory
t(L)=0;                 % Same as t=zeros(1,L);
for n=1:L
        t(n)=(n-1)*Ts;
        y1(n)=A*sin(2*pi*fc*t(n)+phi);
end
plot(t,y1), grid
xlabel('Time (ms)')
ylabel('Amplitude (V)')
title('Sampled sinusoid.')
```

☐ The random value of `phi` for this graph was 0.7477.



☐ The `while` loop, generic syntax:

```
while (expression)
        statement
        ...
end
```

The while loop will run as long as the `(expression)` is `true`. *Bad* programming may cause the `(expression)` to be always `true`. In such case, in order to interrupt the `while` loop press the following keyboard combination: Ctrl+C

☐ The previous example using a `while` loop

```
y2=zeros(1,L);          % Pre-allocate memory
n=1;                    % Starting index
while (n<=L)
        t(n)=(n-1)*Ts;
        y2(n)=A*sin(2*pi*fc*t(n)+phi);
        n=n+1;
end
plot(t,y1-y2), grid     % Check the two results
```

☐ `for` and `while` can be terminated using `break`.

```
y2=zeros(1,L);          % Pre-allocate memory
n=1;                    % Starting index
while (n<=L)
        t(n)=(n-1)*Ts;
        y2(n)=A*sin(2*pi*fc*t(n)+phi);
        if (n>20)
                break;
        end
        n=n+1;
end
% while loop stops after n>20
n
stem(t(1:50),y2(1:50)) % Note the zeros in t and y2 for n>20
```

☐ `for` and `while` loops can be terminated using `break`.

2.4.3 Scripts and Functions

☐ Scripts and functions are files with `.m` extensions.

☐ Newly created scripts and functions names should not start with numbers or contain spaces.

☐ You should not use names for scripts and functions that are already in use. The `which` command can be used to check if a name is already allocated, e.g. `which fir1`

☐ Scripts execute the matlab commands found in the file.

☐ Scripts have no input or output variables.

☐ To create a script from the command prompt: `>> edit filename.m`

☐ Alternatively, you can use the corresponding toolbar icon or the new file menu entry.

☐ Variables created by a script are available in the workspace.

☐ Example: (type in a file test.m and run)

```
x=sign(randn(1,10));
whos
  Name      Size            Bytes  Class     Attributes
  x         1x10               80  double
```

☐ Functions may have input and output variables.

☐ m-file function syntax: `function [out1, out2, ...] = funname(in1, in2, ...)`

☐ Comments about the function usage can be put after the function declaration for clarity.

☐ The Matlab functionality can be extended by creating new functions to solve a specific problem.

☐ The following code creates a new Matlab function that generates binary phase shift keying symbols. Type it in a file `bpsk.m`:

```
function [d, C] = bpsk(N,M)
%BPSK - Generates binary phase shift keying (BPSK) symbols
% Usage: [d, constel] = bpsk(M, N)
%       M,N:    dimensions of the output BPSK matrix d
%       d:      output BPSK matrix d
%       C:      BPSK constellation

d=sign(randn(M,N));
C=[-1 1];
```

☐ Type `>> help bpsk` to see the output message.

☐ To run the function bpsk:

```
>> [x,constel]=bpsk(10,2)
x =
     1    -1    -1     1     1     1    -1    -1     1    -1
     1    -1    -1    -1     1    -1    -1    -1     1    -1
constel =
    -1     1
>> who
Your variables are:
constel  x
```

☐ Note how the variables `d` and `C` of the function are not available in the workspace after the function call.

☐ You may call a function without using all output arguments:

```
>> x=bpsk(10,1)
x =
     1    -1    -1    -1    -1    -1     1     1    -1    -1
```

☐ It is not possible to call a function that outputs its second, third, etc argument only.

☐ If no output argument is used the first output variable is stored in `ans`.

☐ You can create functions that have neither input or output arguments. Type in a file `plot_results.m` the code below:

```
function plot_reuslts()
% Plots BPSK signal
plot(bpsk(1,100),'ro')
axis([ 0 100 -2 2])
xlabel('Symbols')
```

☐ Run this file at the command prompt:

```
>> plot_reuslts
```

☐ Function definitions inside functions are possible. Type in a file `qpsk.m` the code below:

```
function [d, C] = qpsk(N,M)
%QPSK - Generates quadrature phase shift keying (QPSK) symbols
% Usage: [d, constel] = bpsk(M, N)
%       M,N:    dimensions of the output BPSK matrix d
%       d:      output BPSK matrix d
%       C:      QPSK constellation
d=iq(M,N)+j*iq(M,N);
C=[1+j -1+j -1-j 1-j];

function x=iq(M,N)
% Generate bpsk
x=sign(randn(M,N));
```

☐ The function `qpsk()` is called *primary*.

☐ The function `iq()` is called *subfunction*.

☐ Subfunctions such as `iq()` are only available to the primary function `qpsk.m`.

☐ Function definitions inside functions are useful to shorten repeated code segments.

☐ Anonymous functions can also be defined: `f = @(arglist)expression`

☐ Example: $f(x, y) = x^2 + y^2$ is defined as:

```
>> f=@(x,y)x.^2+y.^2
f =
    @(x,y)x.^2+y.^2

>> f(2,2)
ans =
     8
```

☐ Example: $g(x, y, z) = x^2 + y^2 + z^2$ is defined as:

```
>> g=@(x,y,z)x.^2+y.^2+z.^2
g =
    @(x,y,z)x.^2+y.^2+z.^2

>> g(1,2,1)
ans =
     6
```

☐ Anonymous functions are usually defined on the command prompt for quick calculations.

☐ The built-in function `nargin` can be used to avoid improper calls to functions. If called inside a function `nargin` returns the number of input arguments that were used by the function call. To illustrate its use, try to run the qpsk function with only one argument:

```
>> x=qpsk(10)
??? Input argument "M" is undefined.

Error in ==> qpsk at 8
d=iq(M,N)+j*iq(M,N);
```

☐ To avoid this situation the `qpsk` function can be extended as follows:

```
function [d, C] = qpsk(M,N)
%QPSK - Generates quadrature phase shift keying (QPSK) symbols
% Usage: [d, constel] = bpsk(M, N)
%       M,N:    dimensions of the output BPSK matrix d
%       d:      output BPSK matrix d
%       C:      QPSK constellation
if (nargin==0)
    error('At least one input argument is required.')
elseif (nargin==1)
    N=1;
end

d=iq(M,N)+j*iq(M,N);
C=[1+j -1+j -1-j 1-j];

function x=iq(M,N)
% Generate bpsk
x=sign(randn(M,N));
```

☐ To test the modification, try:

```
>> qpsk()
??? Error using ==> qpsk at 8
At least one input argument is required.

>> qpsk(1)
ans =
  -1.0000 + 1.0000i
```

☐ The built-in function `nargout` can be used to avoid improper calls to functions. If called inside a function `nargout` returns the number of otput arguments that were used by the function call. To illustrate its use, we may extend `qpsk.m` to abort if it is called without output arguments:

```
function [d, C] = qpsk(M,N)
%QPSK - Generates quadrature phase shift keying (QPSK) symbols
% Usage: [d, constel] = bpsk(M, N)
%        M,N:    dimensions of the output BPSK matrix d
%        d:      output BPSK matrix d
%        C:      QPSK constellation
if (nargin==0)
    error('At least one input argument is required.')
elseif (nargin==1)
    N=1;
end

if (nargout==0)
    error('At least one output argument is required.')
end

d=iq(M,N)+j*iq(M,N);
C=[1+j -1+j -1-j 1-j];

function x=iq(M,N)
% Generate bpsk
x=sign(randn(M,N));
```

☐ To test the modification, try:

```
>> qpsk(1,2)
??? Error using ==> qpsk at 14
At least one output argument is required.
```

☐ Functions may use `global` variables.

☐ Global variables can be defined inside a script or at command prompt as follows:

```
global x, y, z
```

☐ To use a global variable inside a function add the following to the function body:

```
global x, y, z
```

☐ Global variable are used to save workspace/memory. Thus, they are only useful for variables that are large in size, e.g. 1-by-1e7 or larger.

2.5 Debugging

☐ Read carefully the error messages at the command prompt, e.g. lines and column numbers of the error give a good indication of where the problem occurs and also the type of the problem encountered.

☐ Use the `whos` command to check if the dimensions of variables involved in the calculation generating the error are correct.

☐ Do not overwrite reserved names for variables and functions:

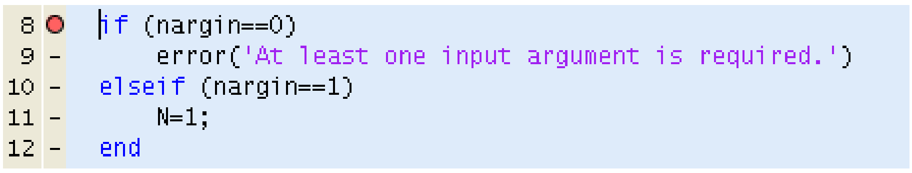
```
>> i=1; j=2; fft=10; % this is very bad practice
```

☐ Insert the `keyboard` command to stop the execution of a function/script to examine and modify variables. You will be presented with the following at the command prompt: `K>>`
☐ To continue the program execution, type `return`.

☐ To step through a script file when in debug mode, use the command `dbstep`.

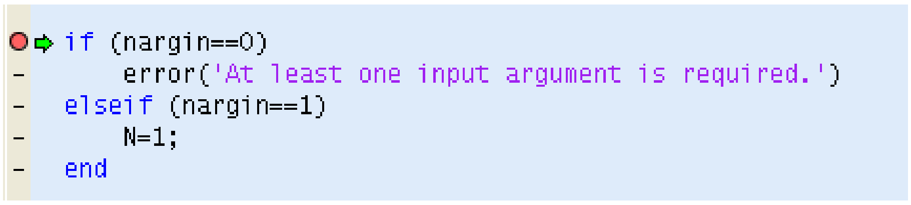
☐ See also `dbcont, dbstop, dbclear, dbtype, dbstack, dbup, dbdown, dbstatus, dbquit` for related debugging commands.

☐ Debugging can also be achieved via the editor. Set a break point in the editor (red dot).



```
if (nargin==0)
    error('At least one input argument is required.')
elseif (nargin==1)
    N=1;
end
```

☐ After running the script, execution stops at the break point (green arrow).



```
if (nargin==0)
    error('At least one input argument is required.')
elseif (nargin==1)
    N=1;
end
```

☐ You can step-in through the script via `F11` or via selecting Step In from the Debug menu.

☐ Add the word keyboard to the `qpsk.m` function after the last comments and try the following:

```
>> d=qpsk(1,2)
K>> nargin
ans =
     2
K>> nargout
ans =
     1
K>> dbstep
8   keyboard
K>> dbstep
10  if (nargin==0)
K>> dbstep
12  elseif (nargin==1)
K>> dbstep
16  if (nargout==0)
K>> dbstep
22  d=iq(M,N)+j*iq(M,N);
K>> dbstep
23  C=[1+j -1+j -1-j 1-j];
K>> dbstep
End of function qpsk.
K>> dbstep
d=qpsk(1,2)
K>> dbstep
d =
  -1.0000 + 1.0000i   1.0000 - 1.0000i
```

2.6 Data Files

☐ Use the `save` command to save the whole work space or specific variables.

☐ The default file name is `matlab.mat`

```
>> save
Saving to: matlab.mat
```

☐ Use the `load` command to load saved variables.

```
>> clear
>> whos
>> load
Loading from: matlab.mat

>> whos
  Name      Size             Bytes  Class     Attributes
  d         1x2                 32  double    complex
```

☐ You can also save only specific variables to a given file name:

```
>> clear, clc
>> SNR_dB=0:2:10;
>> ber_4qam=berawgn(SNR_dB,'qam',4);
>> save ber_4qam_result.mat SNR_dB ber_4qam
>> clear
>> who
>> load ber_4qam_result.mat
>> whos
  Name          Size             Bytes  Class     Attributes
  SNR_dB        1x6                 48  double
  ber_4qam      1x6                 48  double

>> semilogy(SNR_dB,ber_4qam)
```

☐ The save command can be also used to save in ascii format (`--ascii --tab`) or for mat-binary formats used in specific Matlab versions (`-v4.2, -v6, -v7, -v7.3`).

☐ `.mat` is the default Matlab file extension for saving files containing data.

☐ Data can also be saved/read in raw binary format using the `fopen`, `fwrite` and `fread` commands (similar to C/C++).

☐ To write data to a raw binary file using `double` precision:

```
>> clear, clc
>> SNR_dB=0:2:10;
>> ber_4qam=berawgn(SNR_dB,'qam',4);
>> fid=fopen('ber_4qam_result.bin','wb');       % Open the file for writing ('wb')
>> fwrite(fid, [SNR_dB; ber_4qam],'float64');   % Write matrix to file
>> fclose(fid);                                 % Close file
```

☐ To read the data from the previous binary file:

```
>> clear, clc
>> fid=fopen('ber_4qam_result.bin','rb');       % Open the file for reading ('rb')
>> [data,count]=fread(fid,[2 inf],'double');    % Read matrix from file
>> whos
  Name       Size             Bytes  Class     Attributes
  count      1x1                  8  double
  data       2x6                 96  double
  fid        1x1                  8  double
>> data
data =
         0    2.0000    4.0000    6.0000    8.0000   10.0000
    0.0786    0.0375    0.0125    0.0024    0.0002    0.0000
>> SNR_dB=data(1,:);
>> ber_4qam=data(2,:);
>> semilogy(SNR_dB,ber_4qam)
>> fclose(fid);                                 % Close file
```

Review Exercises 3

1. A digital filter is described by the following difference equation:

$$y(n) = 0.0201x(n) + 0.0402x(n-1) + 0.0201x(n-2) + 1.561y(n-1) - 0.6414y(n-2),\ n >= 0$$

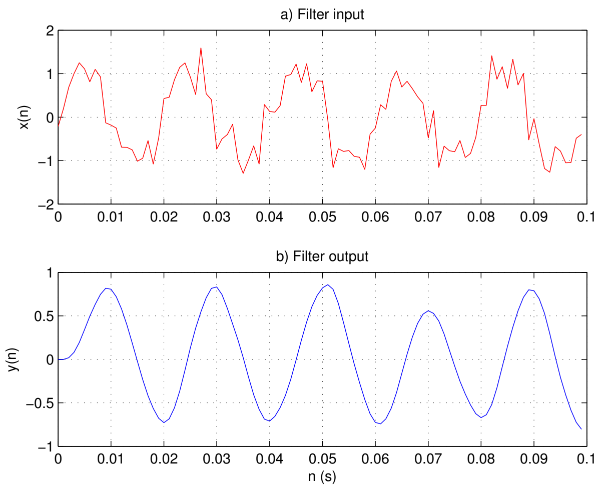
where the output of the filter is $y(n) = 0$ for $n < 0$, and $x(n)$ is the input of the filter.

   (a) Plot the frequency and impulse response of the filter using the `freqz()` and `impz()` commands.

   (b) Generate 100 ms of a signal $x(n)$ that is composed of a sinewave of amplitude 1, carrier frequency 50 Hz and sampling frequency 1000 Hz plus Gaussian noise of zero mean and standard deviation 0.2.

   (c) Using a `for` loop implement the difference equation and filter the composite signal $x(n)$.

   (d) Plot the signal $x(n)$ and $y(n)$ in the same figure using using `subplot`.

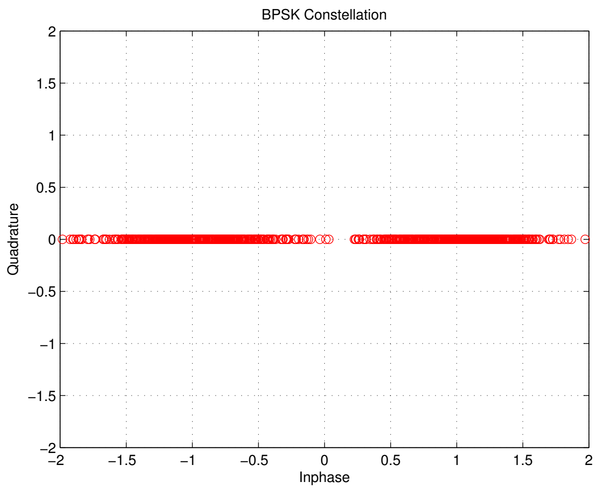


2. (a) Generate 1000 samples of a BPSK signal `d`.

   (b) Generate 1000 samples of a zero-mean Gaussian noise signal `w` with standard deviation 0.4.

   (c) Construct the composite signal `x=d+w`

   (d) Plot the IQ constellation of the signal.



   (e) A simple implementation of a BPSK detector can be achieved via the `dest=sign(x)` function. Find the number of errors after detection by comparing the estimated data `dest` with the actual transmitted data `d` and counting the errors.