

Introductory Brief Guide for Matlab® Users

by Alejandra Mercado†

I. WHAT IS MATLAB?

First of all, Matlab is a registered trademark of The Mathworks Inc. For details about the product, see their webpage at <http://www.mathworks.com/products/matlab/>

Matlab combines numeric computation, advanced graphics and visualization, and a high level programming language. The name “**MAT**rix **LAB**oratory” summarizes that this program is especially designed for matrix operations. Therefore, when using Matlab, you should remember that the program works better, and much *much* faster if you design your program using vectors and matrices, as opposed to scalars.

Example 1: For Matlab, computing the sum

$$x = \sum_{i=1}^5 a_i b_i$$

is much faster if you define the vectors and then do an inner product:

$$\begin{aligned} a &= (a_1 \ a_2 \ a_3 \ a_4 \ a_5) \\ b &= (b_1 \ b_2 \ b_3 \ b_4 \ b_5) \\ x &= a^T b \end{aligned}$$

than if you write some form of iterative sum:

```
x = a1b1
for i = 2 : 5
    x = x + aibi
end
```

‡

Why do we use Matlab instead of, say C⁺⁺, or something? Matlab already has a number of time saving packaged functions, subroutines and commands that are already included in the software. So, say if you wish to get the eigenvalues of a matrix called A , you can just type `eig(A)` and you’re done. Also, the graphics package is very nicely developed and it allows you to edit axis, colors, line widths, legends, etc. interactively, as opposed to editing the program and running it again and again until you get what you want.

The debugging feature is quite similar to the one you might see in, say Visual C⁺⁺, but the actual code is much more compact, making debugging much, much faster. The downside is that the final program may not (indeed, often does not) run as quickly as a C⁺⁺ program, but for quick programs that doesn’t make too much of a difference and you’ve saved considerable amount of time in the development stage. Therefore, Matlab is quite useful for short and quick programs and for research and development. For actually implementing a marketable product, you may prefer to use something else.

†This document was created using a compilation of Matlab guides and sources. A list of these sources is detailed in Section XIII. The author is deeply grateful for the open sharing of all this information and examples.

II. WHERE TO GO

Matlab - 7 is available in these DotCIO PC Computer Labs

- LALLY 104
- LIBRARY PCS 1
- LIBRARY PCS 2
- PITTSBURGH 4114
- SAGE 3101
- SAGE 4510
- TROY 2012
- TROY 2015
- VCC LOBBY (SAGE 3101)
- VCC LOBBY (SAGE 4510)
- VCC LOBBY (TROY 2015)
- VCC NORTH
- VCC SOUTH

For more information about computer labs, and the software available, see the following web site:

<http://www.rpi.edu/dept/arc/pcinfo/pcinfo.html>

For on-line maps and directions to the different buildings on campus, try:

http://www.rpi.edu/virtual_tour/

III. HOW TO START THE PROGRAM

- 1) To start the program, one has to log into the machine with a valid RCS userid and password.
- 2) Once Windows loads, the program should be located as an icon labelled “Matlab 7.0” on the desktop.
- 3) If this is not the case, one can find it by clicking on the “Start” button in the lower left hand corner of the screen, clicking on “All Programs”, clicking on “Matlab 7.0” and then once again on “Matlab 7.0.”

For more information about your RCS account, see the web page:

<http://helpdesk.rpi.edu/update.do?catcenterkey=33>

IV. WHAT YOU WILL SEE (AND HOW TO QUIT)

Note: that Matlab typically takes several seconds to load. Eventually, you should see:

To get started, type one of these: helpwin, helpdesk, or demo.
For product information, type tour or visit www.mathworks.com.

>>

The line >> is the Matlab prompt.

To exit Matlab, type (at the Matlab prompt)

>> quit

or, with the mouse, choose the menu item **File** → **Exit MATLAB**.

V. ENTERING DATA FROM THE KEYBOARD

Once you've started the program and you're looking at the Matlab prompt

```
>>
```

You may enter variables like this (the part **without** the prompt is what Matlab shows you after your entry):

```
>> x = 5          (carriage return here)
```

```
x =
    5
```

Note that Matlab just considers scalars as 1×1 matrices.

TIP: Following your command with a semicolon (;) tells Matlab not to confirm your entry to you, so you would just see

```
>> x = 5;        (carriage return here)
```

with nothing following it (obviously, Matlab **does** keep the value in memory).

For a *row* vector, you may write:

```
>> y = [ 1  2  3 ]      (carriage return here)
```

```
y =
    1    2    3
```

Equivalently, for a *row* vector you may write

```
>> y = [ 1, 2 , 3 ]    (carriage return here)
```

```
y =
    1    2    3
```

Now, for a *column* vector, you would write (note the semicolon instead of comma):

```
>> z = [ 10; 20 ; 30 ] (carriage return here)
```

```
z =
    10
    20
    30
```

Equivalently, for a *column* vector, you may write

```
>> z = [ 10          (carriage return here)
```

```
20          (carriage return here)
```

```
30 ]       (carriage return here)
```

```
z =
    10
    20
    30
```

If you forgot what your data entry was for *y*, you may simply type

```
>> y          (carriage return here)
```

```
y =
    1    2    3
```

If you wish to make a matrix from *y* and *z*, you can transpose one of them and stick them together:

```
>> A = [ y; z' ]      (carriage return here)
```

```
A =
     1     2     3
    10    20    30
```

Note what we've done here: we opened a bracket indicating that we're about to define a matrix (or vector). Then we placed `y` followed by a semicolon, indicating that the vector `y` is the first row of our matrix. Now we couldn't just put `z` because its dimensions are wrong, so we put the transpose of `z`, written `z'`, as the second row of our matrix. Then we wrote another closing bracket to indicate that we're done.

Example 2:

```
>> B = [ z,y'; y' z ]           (carriage return here)
```

```
B =
    10     1
    20     2
    30     3
     1    10
     2    20
     3    30
```

#

TIP: If you've confused the names of your variables and simply looking at their dimensions would help, you may type

```
>> whos           (carriage return here)
```

Name	Size	Bytes	Class
A	2x3	48	double array
B	6x2	96	double array
x	1x1	8	double array
y	1x3	24	double array
z	3x1	24	double array

TIP: Using the command `who` simply gives the names of the variables so far, so you can make sure that you don't overwrite something that you may be using.

VI. PERFORMING OPERATIONS AND SIMPLE FUNCTIONS

A. Simple operations

Just as if you were working by hand, when performing operations with Matlab, you must make sure that the dimensions of the matrices you're working with are appropriate (so don't add a 2x3 matrix to a 3x2 matrix).

Basic Mathematical operations:

+	Addition
-	Subtraction
*	Matrix or scalar Multiplication
.*	Element-by-element multiplication
/	Division
./	Element-by-element division
^	Power (as in <code>x</code> squared is written <code>x^2</code>)
.^	For matrices, element-by-element power
'	Complex conjugate transpose
.'	Transpose without conjugating
()	Specifies evaluation order

Example 3: We compare * with .*

```
>> C = y * z           (carriage return here)
C =
    140
```

I'm going to stop mentioning that you need a carriage return here. I assume you get the idea by now.

```
>> C = y .* z'
C =
    10    40    90
```

Note that if we had tried to write $C = y * z'$, Matlab would have rebuked us for trying to multiply a 1x3 matrix with another 1x3 matrix (which is NOT allowed).

#

More operations for vectors and matrices:

max(y)	Returns the largest element of y, if y is a vector. If it's a matrix, returns a row vector containing the largest element of each column of the matrix.
min(y)	Returns the smallest element of y, if y is a vector. If it's a matrix, returns a row vector containing the smallest element of each column of the matrix.
abs(y)	Returns the absolute values of the elements of y. If y is complex, then it returns the magnitude of the elements of y.
size(A)	Returns the size of matrix A in a row vector (# of rows, # of columns).
length(y)	Returns the length of vector y.
sum(x)	Return the sum of the elements of x.

Example 4:

```
>> D = y * length(z)
D =
    3    6    9
```

#

Logical and relational operations are:

Operator	Meaning	Short Symbol
eq	Equal	==
ne	Not equal	~=
lt	Less than	<
gt	Greater than	>
le	Less than or equal	<=
ge	Greater than or equal	>=
and	Logical AND	&
or	Logical OR	
not	Logical NOT	~
xor	Logical EXCLUSIVE OR	
any	True if any element of vector is nonzero	
all	True if all elements of vector are nonzero	

Example 5:

```

>> q = [5 7];
>> q(1)== q(2)
ans =
    0
>> q(1) ~ = q(2)
ans =
    1
>> q(1) ~ = q(2) & q(1) < q(2)
ans =
    1
>> xor( q(1) ~ = q(2), q(1) < q(2) )
ans =
    0
>> xor( q(1) ~ = q(2), ~ (q(1) < q(2) ) )
ans =
    1

```

#

B. Constants

Matlab has some predefined values by default.

Some predefined constants:

pi	3.1415...
i	imaginary unit
j	equal to i
inf	infinity
NaN	Not-a-Number

It should be noted that these constants **may be overwritten**. Take extra care not to use the i and j as indices in loops, or dummy variables, lest you later forget and try to use them for their default values.

Note also that there is no predefined constant for $e^1 = 2.7183\dots$

TIP: If you wish to clear all of your variables and have these constants return to their default values, you need only enter the command `clear`.

Example 6:

```

>> i
ans =
    0 + 1.0000i
>> for i=1:2
a=3^i
end

```

```

a=
    3
a=
    9
>> i
i=
    2

```

#

Note: How to construct loops and how to avoid the bothersome habit of spilling each intermediate value will be seen in Section VI-D.

C. Simple functions

Trigonometric functions:

sin	Sine.
sinh	Hyperbolic sine.
asin	Inverse sine.
asinh	Inverse hyperbolic sine.
cos	Cosine.
cosh	Hyperbolic cosine.
acos	Inverse cosine.
acosh	Inverse hyperbolic cosine.
tan	Tangent.
tanh	Hyperbolic tangent.
atan	Inverse tangent.
atan2	Four quadrant inverse tangent.
atanh	Inverse hyperbolic tangent.
sec	Secant.
sech	Hyperbolic secant.
asec	Inverse secant.
asech	Inverse hyperbolic secant.
csc	Cosecant.
csch	Hyperbolic cosecant.
acsc	Inverse cosecant.
acsch	Inverse hyperbolic cosecant.
cot	Cotangent.
coth	Hyperbolic cotangent.
acot	Inverse cotangent.
acoth	Inverse hyperbolic cotangent.

Example 7:

```

>> cos(pi)
ans =
    -1

```

#

Note: A function that is quite useful for matrices is the `eig` function, which returns eigenvalues and eigenvectors for a Matrix. Similarly, `svd` performs singular value decomposition for a matrix. More details may be learned by typing `help eig` and `help svd`, respectively.

Exponential:	exp([x y z]) expm(X) log(x) log10(x) log2(x) pow2([x y z]) sqrt(x) nextpow2(x)	Exponential $[e^x e^y e^z]$. Matrix exponential e^X . Natural logarithm $\ln(x)$. Common (base 10) logarithm $\log_{10}(x)$. Base 2 logarithm $\log_2(x)$. Base 2 power $(2^x 2^y 2^z)$. Square root \sqrt{x} . Returns first P such that $2^P \geq abs(x)$.
Complex:	abs(x) angle(x) complex(A,B) conj(x) imag(x) real(x) unwrap(x) isreal(x)	Absolute value of elements of x. Phase angles, in radians, of the complex elements of x. Returns $A + Bi$ (take care, matrix dimensions must match) Complex conjugate x^* . Imaginary part of x. Real part of x. Unwrap radian phases x by changing absolute jumps greater than π to their 2π complement. Returns 1 if all elements in x have zero imaginary part and 0 otherwise.
Rounding and remainder:	fix(x) floor(x) ceil(x) round(x) mod(x,y) rem(x,y) sign(x)	Round elements of x to nearest integer towards zero. Round elements of x to nearest integer towards minus infinity. Round elements of x to nearest integer towards plus infinity. Round elements of x to the nearest integer. Returns $x - y \cdot floor(x/y)$ if $y \neq 0$ (x and y may be matrices). Returns $x - y \cdot fix(x/y)$ if $y \neq 0$ (x and y may be matrices). Signum of elements of x.

D. Loops

As Example 6 pointed out (in Section VI-B), you can create a very simple loop. Loops may have more than one command, and they may call more complicated functions and subroutines (some of your own making, which we'll see in Section X).

The best way to explain is by example, so...

Example 8:

```
>> a=100;
>> for l=1:-3:-6
a= [a    l]
end

a=
    100     1
a=
    100     1    -2
a=
    100     1    -2    -5
```

In this program, the variable `a` is initialized to be a scalar of value 100. The loop concatenates the index `l` to that scalar, forming a row vector. The index of the loop decreases by three each time, so we start with 1, then the second value is $1 - 3 = -2$, the third value is $-2 - 3 = -5$, and then Matlab computes the next index $-5 - 3 = -8$. However $-8 < -6$, which is the bound of the loop index, so the loop breaks at this point.

Note here that the first command ended in a semicolon (;), whereas the loop commands did not. **Adding a semicolon to the end of a command tells Matlab not to print the resulting value on the screen.**

#

Example 9:

```
>> p=3;
>> while p <= 10
p= p+2;
end

>> p
p=
    11
```

We placed semicolon (;) after the commands to avoid excess clutter in our screen. This loop works the same way that the previously mentioned `for p = 3 : 2 : 11` type loop does, but with `while` loops you don't need to define dummy variables and you can quit when a certain condition is met (that is not necessarily a counter).

#

Example 10:

```
>> p=[1:3:15]

p=
    1    4    7   10   13

>> while min(2.^p) < 32 & 5^p(3) < 8.1e62 & p(3)~= 52
p=1.9.^p;
end

>> log(p)

ans=
    1.2195    8.3647   57.3735   393.5249   Inf
```

Note that in the conditional argument of the `while` loops you can feel free to enter many logical operators. Note also that we may write 8.1×10^{62} as `8.1e62`.

TIP: You can create vectors with regularly spaced elements very quickly in the way we created `p` in this example.

#

E. Generating special matrices and noise

Matlab can quickly generate matrices for you to work with, such as these examples:

<code>zeros(x,y)</code>	An x by y array of zeros.
<code>ones(x,y)</code>	An x by y array of ones.
<code>eye(x)</code>	An x by x identity matrix.
<code>repmat(A,x,y)</code>	Replicates and tiles the matrix A to produce an x by y block matrix.
<code>rand(x,y)</code>	Produces an x by y matrix with elements that are uniformly distributed in the interval (0,1).
<code>randn(x,y)</code>	Produces an x by y matrix with elements that are Normally distributed with mean zero and variance one.
<code>linspace(x,y,N)</code>	Generates a row vector of N linearly equally spaced points between x and y.
<code>logspace(x,y,N)</code>	Generates a row vector of N logarithmically points between x and y.
<code>:</code>	Regularly spaced vector and index into matrix (so J:K is the same as [J,J+1,...,K]).

Example 11: We know that if a random variable X is Gaussian with mean μ and variance $\sigma^2 > 0$, then the random variable $Y = \frac{X-\mu}{\sigma}$ is Normally distributed (mean 0 and variance 1). Therefore, if I wish to create a vector of *realizations* of Gaussian random variables that had $\mu = 20$ and $\sigma^2 = 9$, I would do the following:

```
>> a=randn(1,10000);
```

```
>> b= 3*a + 20;
```

The vector b should be Gaussian distributed with mean 20 and variance $3^2 = 9$. To verify this, we use a simple histogram function on both a and b (the resulting plots are in Figure 1):

```
>> [N1,N2]= hist(a,50);
```

```
>> [G1,G2]= hist(b,50);
```

```
>> subplot(1,2,1),plot(N2,N1)
```

```
>> subplot(1,2,2),plot(G2,G1)
```

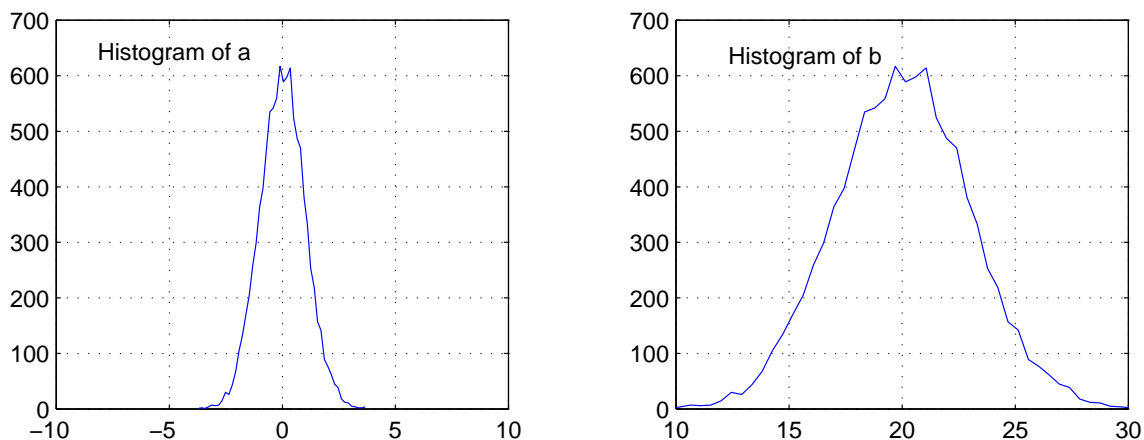


Fig. 1. The left-side plot shows a rough distribution of the elements of a . The right-side plot shows a rough distribution of the elements of b .

Though the plots indicate the means quite clearly, they don't give an exact idea of the variance, so we can check that explicitly:

```
>> var(b)
ans=
    8.9452
```

#

When you're dealing with noise in systems, it's useful to have statistical analysis tools. Matlab has a host of such functions, of which a short table follows. For a more complete list, you may type `help stats`.

<code>corrcoef(X)</code>	A matrix of correlation coefficients formed from array X whose each row is an observation and each column is a variable.
<code>cov(x)</code>	Returns variance of vector x.
<code>harmmean(x)</code>	Harmonic mean of elements of x (the inverse of the mean of the inverses of the elements).
<code>mad(x)</code>	Median Absolute Deviation of elements of x: $\frac{1}{N} \sum_{i=1}^N x_i - \bar{x} $.
<code>mean(x)</code>	Sample average of elements of x.
<code>median(x)</code>	50th percentile of elements of x (where its cumulative distribution equals 1/2).
<code>moment(x,N)</code>	Return the central moment of the elements of x, specified by the order N.
<code>nanmax(x)</code>	Maximum element of x ignoring NaNs.
<code>nanmean(x)</code>	Mean of elements of x ignoring NaNs.
<code>nanmedian(x)</code>	Median of elements of x ignoring NaNs.
<code>nanmin(x)</code>	Minimum of elements of x ignoring NaNs.
<code>nanstd(x)</code>	Standard deviation of elements of x ignoring NaNs.
<code>nansum(x)</code>	Sum of elements of x ignoring NaNs.
<code>prctile(x,P)</code>	Returns a value that is greater than P% of the values in x.
<code>range(x)</code>	Range: $\max x - \min x$.
<code>std(x)</code>	Returns the standard deviation of the elements of x.
<code>var(x)</code>	Return the variance of the elements in x.

TIP: If you wish to review some definitions of probabilities and statistics, a good source is <http://mathworld.wolfram.com/topics/ProbabilityandStatistics.html>

NOTE:

The Uniform distribution and Normal distribution are not the only kind of random variable that Matlab can produce. A shortened list of random number generators is (Again, for a more complete list, you may type `help stats`):

<code>binornd</code>	Binomial random numbers
<code>chi2rnd</code>	Chi square random numbers
<code>exprnd</code>	Exponential random numbers
<code>gamrnd</code>	Gamma random numbers
<code>geornd</code>	Geometric random numbers
<code>lognrnd</code>	Lognormal random numbers
<code>mvnrnd</code>	Multivariate normal random numbers
<code>ncx2rnd</code>	Noncentral Chi-square random numbers
<code>normrnd</code>	Normal (Gaussian) random numbers
<code>poissrnd</code>	Poisson random numbers
<code>random</code>	Random numbers from specified distribution
<code>raylrnd</code>	Rayleigh random numbers
<code>unidirnd</code>	Discrete uniform random numbers
<code>unifrnd</code>	Uniform random numbers

VII. PLOTS

One of the most pleasant aspects of Matlab is the ready-made package for plotting one's results. To plot the abscissa data (contained in vector x) against the ordinate data (contained in vector y) you need only type

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

Once you see the Figure pop up, you may edit the color of the line, the axis, the grid, the legend, the title, the labels for the abscissa and the ordinate, the width of the graph line, and mark any points of interest, as you like. Simply click on the top menu item "Tools" and on the sub-menu item "Enable Plot Editing". You may edit the axis, legend, and so on from the menu bar, or you may choose to do so from the Matlab prompt. For instructions on how to do the latter, simply type `help plot`

If you wish to plot two lines of data, x_1 and x_2 , in the same graph, you may do the following:

```
>> plot(x1,y)
```

```
>> hold
```

```
>> plot(x2,y)
```

If you wish to make one figure that has several graphs in an array (such as you saw in Figure 1), you may type

```
>> subplot(N,M,1),plot(x1,y1)
```

```
>> subplot(N,M,2),plot(x2,y2)
```

```
⋮
```

```
>> subplot(N,M, NM),plot(xNM,yNM)
```

This will produce an $N \times M$ array of graphs, each with its own data. Make sure you finish editing the appearance of graph (i, j) to your liking before you move on to plot the next graph.

The commands `semilogx`, `semilogy`, and `loglog` produce the types of graphs that the commands suggest.

To create 3-dimensional graphs, you may use the `mesh` command or the `surf` command. These types extend beyond the purpose of this tutorial, so we advise students to simply type `help mesh` and `help surf` to learn more about these.

VIII. QUICK HELP

The descriptions presented in this document are of the most rudimentary form. Students are strongly urged to learn more about any command by simply typing the help command for it, for example,

```
>> help cov
```

TIP: Almost always, Matlab provides a list of additional commands and functions at the end of a “help” description. They are at the bottom, where it says “See also ...”. It is *strenuously* recommended that the user browse through those additional functions, as more often than not, you may find a function or command that facilitates your job immensely.

If you seek a kind of function that you think Matlab may have, but you do not know the command for it, you may try to see if any Matlab help description carries that word. So, say for example that you want to see if Matlab will produce random numbers with a Poisson distribution, but when you type `help poisson` you don’t get anywhere. Then you may try

```
>> lookfor poisson
```

If you simply wish to browse through the many functions of Matlab, you may simply type `help`. If one of the subjects, such as the topic of “Sparse Matrices”, interests you, you can look at how Matlab labels that subject from this list (“sparfun”) and then type `help sparfun`.

IX. WRITING A PROGRAM FILE

As with any programming language, your first draft of any meaningful sequence of commands is almost certainly going to have a mistake. Therefore, it makes sense to write a text file with your commands and ask Matlab to reference that instead of your keyboard entries.

Matlab has a built-in editor, which has a very friendly debugging tool. However, you are not required to use Matlab’s editor to make the file. Any text editor will do just fine (Microsoft Notepad®, vi, Emacs, etc.¹), therefore, you are free to use whatever text editor you have at home to write a Matlab program and simply bring it in on a floppy (if they still exist), memory stick, or email it to your school account.

The only requirement is that the file must have the `m` extension, hence their name “*m-file*”.

Example 12:

Say you use Microsoft Notepad to create your program, called “`first_draft.m`”. You can open the editor, and write the following commands:

```
a=[1 5 10];
b=a.^-1
c=exp(b)
```

Then save the file as `first_draft.m`

Now when you initiate Matlab, simply type in the filename.

Note: If you’re having trouble, make sure the *m-file* `first_draft.m` is in the directory that you’re running Matlab from (you can check this by typing `path` at the Matlab prompt and you can move your current Matlab working directory by clicking on the menu item **File** → **Set Path...**)

```
>> first_draft
```

¹Microsoft Notepad and the Microsoft Notepad logo are the property of the Microsoft Corporation. vi is a screen editor created by Bill Joy. Emacs (**E**ditor **MAC**ro**S**): many versions of Emacs have appeared over the years, but nowadays there are two that are commonly used: GNU Emacs, written by Richard Stallman beginning in 1984, and XEmacs, a mostly-compatible fork of GNU Emacs that was started in 1991.

```

b=
    1.0000    0.2000    0.1000

c=
    2.7183    1.2214    1.1052

```

Note that the `a` variable did not appear, since its command had a semicolon after it. The other variables did appear. This tells you that there is no functional difference between using an *m-file* and typing the commands in by hand.

‡

If you wish to use the Matlab Editor, simply click with your mouse on the menu item at the top **File** → **New** → **M-file**. In a couple of seconds, you'll get a new window for editing your file; this window will have the heading "MATLAB Editor/Debugger - [Untitled]". Simply enter the commands, just as if you were entering the commands at the Matlab prompt and when you've finished, you can click with your mouse on the menu item at the top **File** → **Save As...** and choose the path and filename you wish.

TIP: When writing a program or entering commands with the keyboard, anything that appears after an % symbol is considered "commented out", which means that Matlab will ignore it. Therefore, feel free to use % liberally to add comments, remarks, constraints, and reminders into your program.

A. Debugging

For debugging in the MATLAB Editor/Debugger, simply type your commands into the editor and save the file with the desired filename (alternatively, open your existing file with the top menu items **File** → **Open...**).

Now that you have a named file, you can choose different lines of your program where you wish to pause and browse through the variables. Place your cursor on a line and with the mouse click on the sheet with the red dot on the top menu list. A red dot should appear to the left of the line you have chosen. Repeat this process for as many lines as you like.

Go back to the Matlab prompt and type the filename as if you wanted to run your program. The program will begin running and then pause at each line that has a red dot. You will know which line it is paused at because the red dot will be accompanied by a yellow arrow. Now you can browse through the values of your variables (just type the name of that variable at the Matlab prompt... or easier yet, glide the mouse over the variable on your Editor), or you can click on **View** → **Workspace Browser** to see the sizes of your matrices (good sanity check).

When you wish to advance the program beyond the current Breakpoint (red dot), simply click on the menu item **Debug** and choose if you want Matlab to **Continue** to the next Breakpoint, or **Single Step** to the next line, or **Step In** the function you are hovering over, etc.

When you're done debugging, you can click on the menu item showing a pencil eraser with a little red dots to remove all of the Breakpoints. Save the file one last time and run it to see if the problem is solved.

X. FUNCTIONS/SUBROUTINES

In addition to the functions already provided by Matlab (or one of the Matlab Toolboxes), we may generate our own functions in order to better organize our program and avoid clutter when a certain group of

commands should be repeated often in the overall program. Functions and subroutines also help in the debugging process (see Section IX-A) since it makes the program more modular.

To do so, make a new *m-file* altogether specifically for this function. *Don't forget that the extension must still be m!*

Again, the best way to explain it is by example, so...

Example 13:

Say you use vi to create a function called “lucky.m” for your program “second_draft.m”. You can open the editor, and write the following commands for your function:

```
function [out1,out2]=lucky(in1,in2,in3)
temp=in1+in2;
happy=temp/in3;
out1=exp(temp);
out2=happy+3;
return
```

Then save the file.

Now when you wish to incorporate your function into your program, `second_draft.m`, you simply edit the program and add the call to the function:

```
a=3;
b=4;
[s,t]=lucky(a,b,50)
```

When you run your program at the Matlab prompt, you'll get

```
>> second_draft

s=
    1.0966e+003

t=
    3.1400
```

Note: The variables `in1`, `in2`, `in3`, `temp`, `happy`, `out1`, and `out2` may be scalars, vectors, or matrices.

#

XI. SAVING YOUR VARIABLES AND DATA

Oftentimes, your program may take some time to run (usually when you handle many large variables). Matlab allows you to save the variables you have at any time so that you may load those variables in later and continue working with them. This is especially useful when a homework assignment includes giving you data which you must handle with a program.

For this, use the `save` command. This command will save some or all of your workspace variables to disk. The clearest description is provided by Matlab itself, so you may type `help save` to get a detailed idea.

TIP: It often happens that your program will run without error or bug until it reaches a certain point (such as when a variable approaches zero and is used as a divisor). If the program has been running for a long time and all the data you've created so far is useful and good, it can be quite frustrating to lose all of that data because of a bug that crops up late in the running time. Therefore, it may be useful to **save** certain variables every now and then to ensure that - should the program crash - you have at least salvaged some data.

In order to retrieve that data later, simply use the **load** command. Again, this is best described by Matlab, so simply type **help load**

XII. EXAMPLES OF A COMMUNICATIONS MATLAB FILES

Example 14:

[1] Define the periodic signal (over one period)

$$x(t) = \begin{cases} 1 & , |t| < 1 \\ \frac{1}{2} & , t = \pm 1 \\ 0 & \text{otherwise,} \end{cases}$$

which has period 4.

- 1) Determine the Fourier Series coefficients of $x(t)$ in exponential form.
- 2) Determine the Fourier Series coefficients of $x(t)$ in trigonometric form.
- 3) Plot the discrete spectrum of $x(t)$.

Answers:

- 1) From Section 2.2 of your text[4], we know that for this simple function the Dirichlet conditions are satisfied, so we either look up the Fourier Series in a table (for such a common function, you're bound to find it in most Fourier Series tables) or you can take the trouble of deriving it yourself:

$$x(t) = \sum_{n=-\infty}^{\infty} x_n e^{j2\pi \frac{n}{4} t} ,$$

where

$$x_n = \frac{1}{4} \int x(t) e^{-j2\pi \frac{n}{4} t} dt ,$$

where the integral is taken over one period. We can expand the expression for x_n as

$$\begin{aligned} x_n &= \frac{1}{4} \int_{-1}^1 1 \cdot e^{-j2\pi \frac{n}{4} t} dt , \\ &= \frac{1}{-2j\pi n} \left(e^{-j2\pi n/4} - e^{j2\pi n/4} \right) , \\ &= \frac{1}{\pi n} \left(\frac{e^{j\pi n/2} - e^{-j\pi n/2}}{2j} \right) , \\ &= \frac{1}{2} \frac{1}{\pi n} \sin \left(\frac{\pi n}{2} \right) , \\ &\triangleq \frac{1}{2} \text{sinc} \left(\frac{n}{2} \right) . \end{aligned}$$

- 2) The trigonometric form for a *real* function looks like this

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos(2\pi \frac{n}{T_0} t) + b_n \sin(2\pi \frac{n}{T_0} t) \right)$$

where $x_n = \frac{a_n - jb_n}{2}$, and for an *even* function, $b_n = 0$, so $a_n = \text{sinc}(\frac{n}{2})$.

- 3) We can write $x_n = |x_n|e^{j\angle x_n}$. Now, we know that the x_n are all real. The phase is either $\theta_n = 0$ when $x_n \geq 0$, or $\theta_n = \pi$ when $x_n < 0$. To plot this, we may write

```
>> n=[-20:20];    % This is the dummy variable for the abscissa.
>> xn=abs(.5 * sinc(n/2));    % This takes the abscissa values, applies the function for  $x_n$ 
    % and then takes the absolute value.
>> thetan=angle(.5 * sinc(n/2));    % This gets the phase angles for the  $x_n$ .
>> subplot(1,2,1),stem(n,xn)    % This creates a 1 x 2 graph and writes the first graph on it.
    % "stem" makes the point plot, as opposed to a line plot.
>> title('modulus')    % Writes the title for first graph.
>> xlabel('n')    % Labels the abscissa axis.
>> ylabel('modulus of x_n ')    % Labels the ordinate axis.
>> subplot(1,2,2),stem(n,thetan)    % This writes the second graph on it (see Figure 2).
>> title('phase')    % Writes the title for first graph.
>> xlabel('n')    % Labels the abscissa axis.
>> ylabel('phase of x_n')    % Labels the ordinate axis.
```

The resulting plot is shown in Figure 2.

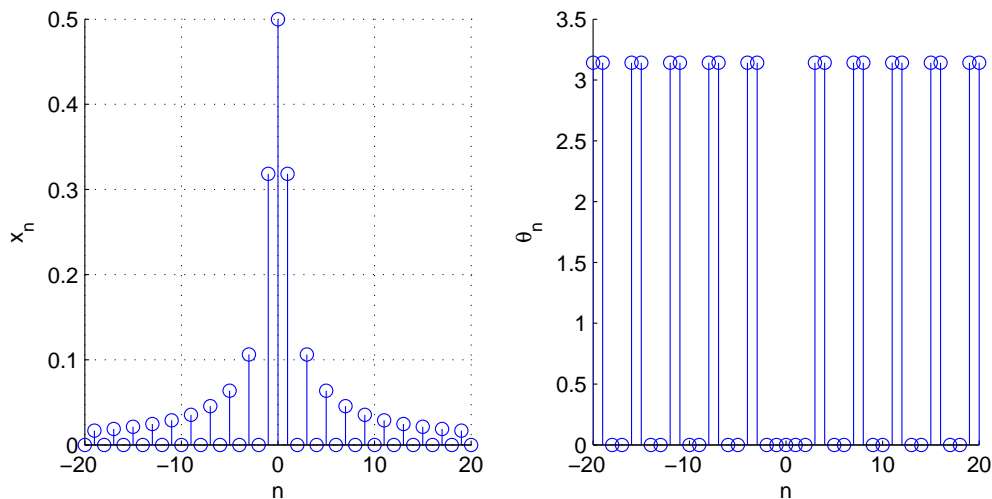


Fig. 2. Discrete spectrum of $x(t)$.

‡

Example 15:

[1] Define the periodic triangular pulse signal (over one period) as

$$x(t) = \begin{cases} t + 1 & , -1 \leq t \leq 0 \\ -t + 1 & , 0 < t \leq 1 \\ 0 & \text{otherwise,} \end{cases}$$

which has period $T_0 = 2$.

1) Determine the Fourier Series coefficients of $x(t)$.

2) Assume $x(t)$ passes through an L.T.I. system whose impulse response is

$$h(t) = \begin{cases} t & , 0 \leq t < 1 \\ 0 & \text{otherwise.} \end{cases}$$

Use Matlab to determine the transfer function of this system.

3) Plot the spectrum of output of the system, $y(t)$.

Answers:

1) Again, we know that for this simple function the Dirichlet conditions are satisfied, so we either look up the Fourier Series in a table (we might find it), or you may derive it yourself:

$$x(t) = \sum_{n=-\infty}^{\infty} x_n e^{j2\pi\frac{n}{2}t} ,$$

where

$$x_n = \frac{1}{2} \int x(t) e^{-j2\pi\frac{n}{2}t} dt ,$$

where the integral is taken over one period. We can expand the expression for x_n as

$$x_n = \frac{1}{2} \int_{-1}^1 x(t) \cdot e^{-j2\pi\frac{n}{2}t} dt ,$$

Here we replace the periodic $x(t)$ with a single triangular signal $\Lambda(t)$

$$= \frac{1}{2} \int_{-1}^1 \Lambda(t) \cdot e^{-j\pi nt} dt ,$$

But $\Lambda(t) = 0$ for $t \notin [-1, 1]$, by definition, so we may change the integral limits:

$$= \frac{1}{2} \int_{-\infty}^{\infty} \Lambda(t) \cdot e^{-j\pi nt} dt ,$$

$$= \frac{1}{2} \int_{-\infty}^{\infty} \Lambda(t) \cdot e^{-j2\pi\frac{n}{2}t} dt ,$$

Here we replace the $n/2$ with the label f , and we get the standard Fourier Transform:

$$= \frac{1}{2} \mathfrak{F}[\Lambda(t)] \quad \text{evaluated at } f = n/2 ,$$

Look that common transform up in a table:

$$= \frac{1}{2} \text{sinc}^2\left(\frac{n}{2}\right) .$$

2) Signals and Systems reminds us of the properties of passing a periodic signal, like $x(t)$, through an L.T.I. system. (You may wish to glance at Section 2.2.2 of [2] to review this.) From there, we know that the *transfer function* (sometimes called the *frequency response*) of the system, $h(t)$, is

$$H(f) = \int_{-\infty}^{\infty} h(t) \cdot e^{-j2\pi ft} dt .$$

To work with this in Matlab, we must revert to discrete-time analysis. We use Matlab's discrete Fourier transform function (type `help fft` for more details). In Matlab, `fft(X)` is the discrete Fourier transform (DFT) of vector `X`. In Matlab, we may write

```
>> ts=1/40; % This is the sampling interval (we'll review the need for this in Section XII-A).
```

```
>> t=[0:ts:1]; % This is the vector with the times for sampling.
```

```
>> h=[zeros(1,20),t,zeros(1,20)]; % This is the impulse response, h(t).
```

```
>> H=fft(h)*ts; % This is the transfer function.
```

```
>> Hcentered=fftshift(H); % Rearrange it so the center frequency is zero.
```

```
>> fs=1/ts; % This is the sampling frequency.
```

```

>> l=length(H) - 1;    % This is the number of points of the transform in the frequency domain.
>> df=fs/l;    % frequency resolution (so we can use it for the x-axis).
>> f=[0:df:fs]-fs/2;    % Frequencies for the plot.
>> plot(f,abs(Hcentered))    % Make the graph.
>> xlabel('frequency')    % Labels the abscissa axis.
>> ylabel('abs(H(f))')    % Labels the ordinate axis.

```

The plot of the transfer function is shown in Figure 3.

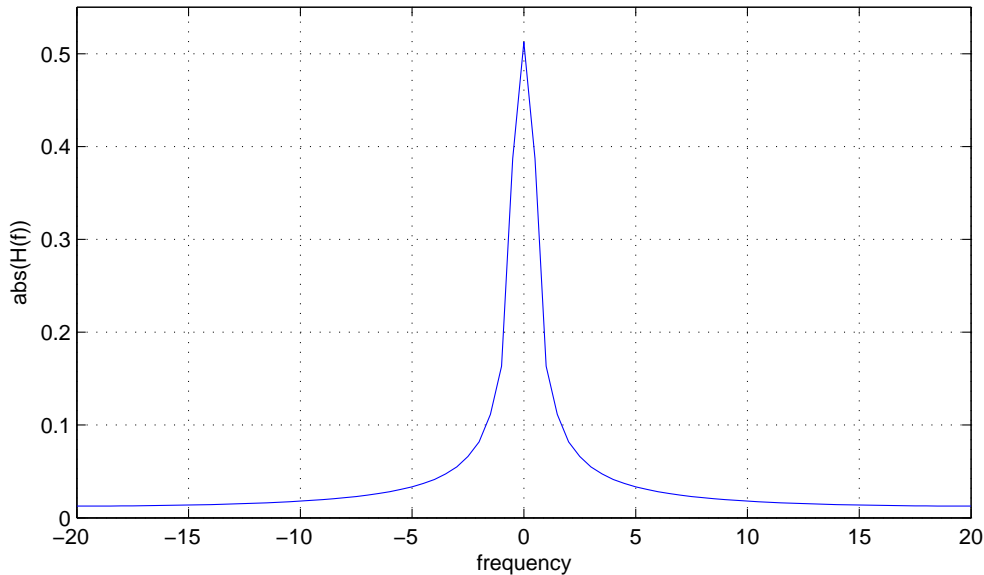


Fig. 3. Transfer function for $h(t)$.

- 3) From Signals and Systems (you may review [2]), we know that the relation between the input, $x(t)$, and the output, $y(t)$ is

$$\begin{aligned}
 y_n &= x_n \cdot H\left(\frac{n}{T_0}\right), \\
 &= \frac{1}{2} \text{sinc}^2\left(\frac{n}{2}\right) \cdot H\left(\frac{n}{2}\right),
 \end{aligned}$$

since $T_0 = 2$. So we can plot the discrete spectrum of $y(t)$ by noting that

$$\begin{aligned}
 |y_n| &= |\text{sinc}^2(\frac{n}{2})| \cdot |H(\frac{n}{2})| \\
 \text{and} \\
 \angle y_n &= \angle \text{sinc}^2(\frac{n}{2}) + \angle H(\frac{n}{2})
 \end{aligned}$$

```

>> n=[-20:20];    % This is the n index.
>> xn=.5*(sinc(n/2)).^ 2;    % This are the x_n we calculated in part (a).
>> length(xn);    % Here we see how many points x_n has in our Matlab vector.
ans =
    41
>> length(Hcentered);    % How long is the transfer function?
ans =
    81

```

```
>> length(Hcentered(21:61));    % We only need 41 central points to match  $x_n$ .
ans =
    41
>> subplot(1,2,1),stem(n,abs(xn).*abs(Hcentered(21:61)))
>> subplot(1,2,2),stem(n,angle(xn)+angle(Hcentered(21:61)))
```

After some simple plot editing to add the labels, we have the result shown in Figure 4.

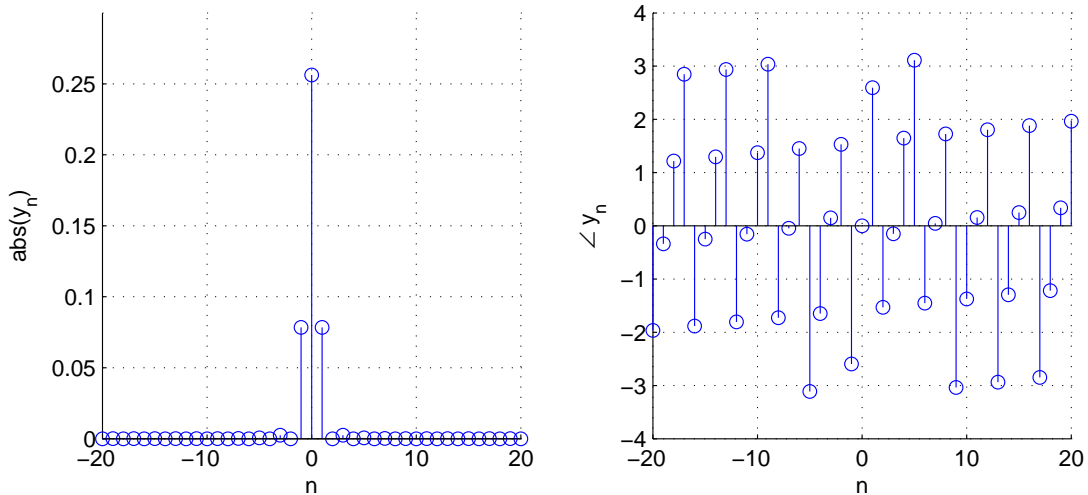


Fig. 4. Discrete spectrum of $y(t)$.

#

A. Spectral Analysis with Computers

In the analog world, we study the spectral properties of a (largely) arbitrary signal, $x(t)$, through its Fourier Transform, $\mathfrak{F}\{x(t)\} = X(j\Omega)$. Here, both the time dummy variable, t , and the frequency dummy variable, Ω , are continuous. Yet, when using a computer to analyze $x(t)$ it is impossible to do so in continuous domains. We are forced to sample $x(t)$ and enter its discrete-time samples, $x(k \cdot \Delta t) \triangleq x(t_k) \triangleq x[k]$.

The *sampling theorem* (review your Signals and Systems notes or browse over Section 7.1.1 of [2]) tells us that $x(t)$ can be recovered without any distortion from its samples, $x(t_k)$, as long as $x(t)$ is bandlimited (so its Fourier Transform vanishes for $|\Omega| > W$, for some W) and we have chosen $\Delta t \leq 1/2W$. If both of these requirements are met, then *no information is lost* in the process of sampling $x(t)$ and the entire signal can be recovered from its samples.

Now that we've been reminded that (under certain circumstances) we can represent a signal by its (discrete-time) sampled values, it remains to study the spectral characteristics of that signal in a discrete-frequency domain. After all, we expect to input a discrete number of samples and we expect to obtain a discrete number of outputs representing the spectrum. For such environments we have the Discrete Fourier Transform, defined by

$$F_x[n] = \sum_{k=0}^{N-1} x[k] e^{-j2\pi \frac{nk}{N}},$$

where N is the number of samples you have for $x(t)$, k is the index number for the time-domain $x[k] = x(t_k)$, and n is the index number for the frequency-domain transform. The inverse transform is

$$x[k] = \frac{1}{N} \sum_{n=0}^{N-1} F_x[n] e^{j2\pi \frac{kn}{N}} .$$

Like any Fourier-type transform, the D.F.T. reveals periodicities in input data as well as the relative strengths of any periodic components [3]. In general, the discrete Fourier transform of a *real* sequence of numbers will be a sequence of complex numbers of the same length. In particular, if $x[k]$ are real, then $F_x[N - n]$ and $F_x[n]$ are related by

$$F_x[N - n] = F_x[n]^* \quad \text{for } n = 0, 1, \dots, N - 1 .$$

Matlab yields the D.F.T. of a vector by applying the *Fast Fourier Transform*. This is nothing more than a clever, computationally efficient way of calculating the D.F.T.

The question that remains is: *What is the relation between the Fourier Transform of the continuous-time signal (which we seek) and the D.F.T. of the sampled signal (which is all that computers can provide)?*

B. Sampling Theorem Applied to Matlab Problems

In this section we'll look at a (near) arbitrary continuous-time signal, $x_c(t)$, and we'll derive a relation between it's Fourier Transform (what we seek) and the Discrete Fourier Transform of a set of it's sampled values (what Matlab can give us). We'll do this in three steps.

STEP 1:

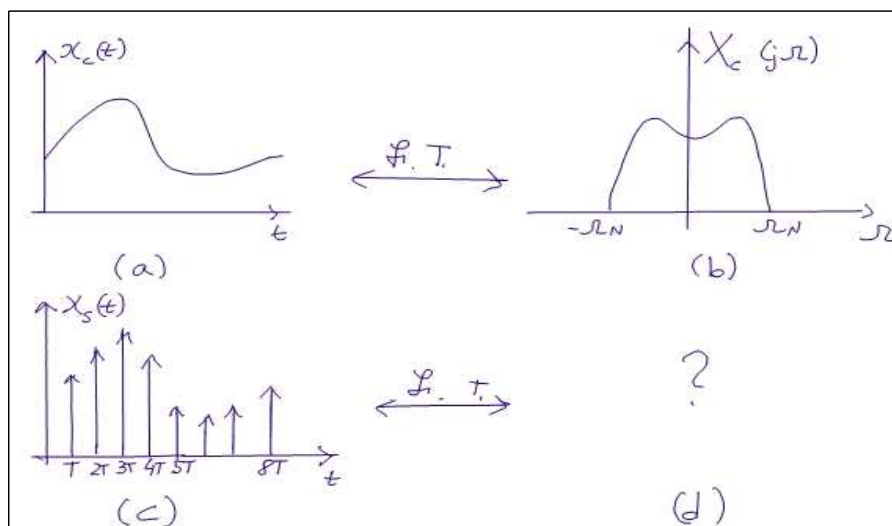


Fig. 5. (a) Continuous-time signal whose spectrum we wish to study with Matlab. (b) Actual (unknown to us) spectrum of $x_c(t)$. (c) Train of impulses weighted by the samples of the original signal, $x_c(t)$, where T is the sampling interval. We denote this $x_s(t)$ (d) Spectrum of $x_s(t)$, which we'll derive now.

In Figure 5(c) we see that the continuous-time signal (the object of our interest) was sampled and its samples are used to weight a train of impulses. This is the same as multiplying $x_c(t)$ with a train of impulses,

each separated by T seconds (we'll assume that we're sampling at exactly the Nyquist rate, for simplicity).

$$x_s(t) = x_c(t) \cdot \sum_{n=-\infty}^{\infty} \delta(t - nT) = \sum_{n=-\infty}^{\infty} x_c(nT)\delta(t - nT) . \tag{1}$$

The samples from the continuous-time signal, $x_c(nT)$, will be the entries we use into the computer, which we'll denote $x[n]$. The sequence of numbers $\{x[n]\}_{n=0, \dots, N-1}$ will be entered as a vector into Matlab ².

To get the Fourier Transform of $x_s(t)$, $\mathfrak{F}\{x_s(t)\}$, we use two facts:

$$\mathfrak{F}\{x_s(t)\} \triangleq X_s(j\Omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta\left(\Omega - \frac{k2\pi}{T}\right) ,$$

and

the F.T. of the product of two functions is $\mathfrak{F}\{f(t) \cdot g(t)\} = \frac{1}{2\pi} F(j\Omega) * G(j\Omega) ,$

where $*$ denotes the convolution operator. So, both of these facts allow us to write

$$\begin{aligned} X_s(j\Omega) &= \frac{1}{2\pi} X_c(j\Omega) * \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta\left(\Omega - \frac{k2\pi}{T}\right) \quad \text{here using both facts} \\ &= \frac{1}{2\pi} \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} X_c(j\Omega) * \delta\left(\Omega - \frac{k2\pi}{T}\right) \\ &= \frac{1}{T} \sum_{k=-\infty}^{\infty} X_c\left(j\Omega - \frac{kj2\pi}{T}\right) \quad \text{which the following figure depicts:} \end{aligned}$$

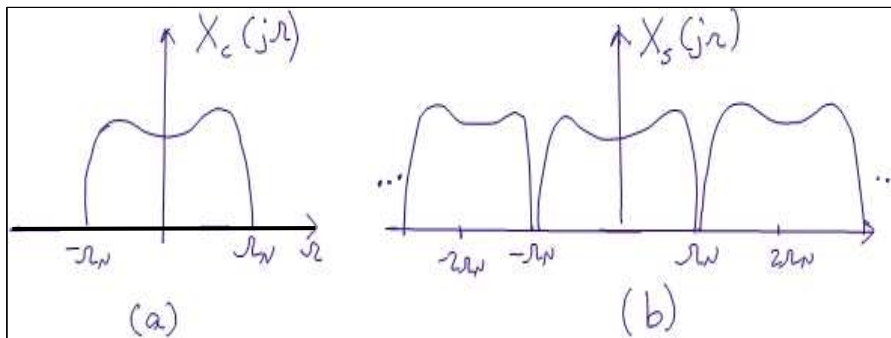


Fig. 6. (a) The Fourier Transform of $x_c(t)$. (b) The Fourier Transform of $x_s(t)$.

Observe Figure 6 and note that for frequencies between $-\Omega_N$ and Ω_N the only difference between $X_c(j\Omega)$ (that which we seek) and $X_s(j\Omega)$ is a constant $\frac{1}{T}$:

$$X_s(j\Omega) = \frac{1}{T} X_c(j\Omega) \quad \text{for } |\Omega| < \Omega_N . \tag{2}$$

Equation 2 is an important intermediate solution, which we will reference shortly. Now we go on to...

STEP 2:

Here, we'll make an association between the Fourier Transform of the train of weighted impulses, $X_s(j\Omega)$, and the collection of sample values from $x_c(t)$ that we would input into the computer, $x[1], x[2], \dots$

²Note that the entries into the computer do not carry the sampling interval, T , with them. We'll have to provide that information later.

First we write out the Fourier Transform for the weighted train of impulses:

$$\begin{aligned}
 X_s(j\Omega) &= \int_{-\infty}^{\infty} x_s(t) e^{-j\Omega t} dt \\
 &\text{replacing } x_s(t) \text{ from Equation (1), we get} \\
 &= \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x_c(nT) \delta(t - nT) e^{-j\Omega t} dt \\
 &\text{but } x_c(nT) \text{ are just the samples we use in discrete time, } x[n]. \\
 &= \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x[n] \delta(t - nT) e^{-j\Omega t} dt \\
 &= \sum_{n=-\infty}^{\infty} x[n] \int_{-\infty}^{\infty} \delta(t - nT) e^{-j\Omega t} dt \\
 &\text{by definition of the } \delta \text{ function} \\
 &= \sum_{n=-\infty}^{\infty} x[n] e^{-j\Omega nT} .
 \end{aligned}$$

So far, from Equation (2) we can conclude that the Fourier Transform of $x_c(t)$ (which is what we seek) can be found by

$$X_c(j\Omega) = T X_s(j\Omega) = T \sum_{n=-\infty}^{\infty} x[n] e^{-j\Omega nT} \quad \text{for } |\Omega| < \Omega_N . \quad (3)$$

STEP 3:

Finally, we consider the Discrete-Time Fourier Transform of the discrete-time samples of our signal:

$$X_{DTFT}(e^{j\omega}) \triangleq \sum_{n=-\infty}^{\infty} x[n] e^{-j\omega n} ,$$

and we note that this transform is nothing other than the same expression we obtained for $X_s(j\Omega)$ in Step 2, when we set $\omega = \Omega T$.

In fact, the Discrete-Time Fourier Transform is periodic in the frequency domain, and it has the same shape that $X_s(j\Omega)$ has over the frequencies, $|\Omega| < \Omega_N$, but since it has no sampling *period* information, it's spread over the normalized frequencies $|\omega| < \pi$.

The only remaining problem is that the Discrete-Time Fourier Transform is continuous in ω (the frequency domain) and Matlab can't handle continuous variables. Therefore, we revert to the Discrete Fourier Transform, which is discrete both in the time and in the frequency domains.

For the Discrete Fourier Transform we work with a *finite number of samples* (the computer doesn't have enough space to handle infinite samples, anyway), $x[0], x[1], \dots, x[N-1]$ in the time domain. This means that the infinite sum written above for the Discrete-Time Fourier Transform would only have n going from 0 to $N-1$, and not $-\infty$ to ∞ . But we also sample the Discrete-Time Fourier Transform for N points: so we divide the 2π interval by N and we pick off each sample:

$$X_{DFT}[k] = X_{DTFT}(e^{j\omega})|_{\omega=\frac{2\pi k}{N}} = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N} n k}$$

Putting all three steps together, we use this last equation along with Equation (3) to get a plot of the Fourier Transform of $x_c(t)$ through the expression $T \cdot X_{DFT}[k]$. An example of how to use this was presented in part (2) of Example 15 of this booklet. A good review source for these transforms and their properties is [2].

As mentioned before, Matlab yields the Discrete Fourier Transform by performing the Fast Fourier Transform. For more information on that, simply type `help fft`.

XIII. ON-LINE RESOURCES FOR MATLAB HELP

There are a number of sources for finding information about Matlab. I suggest you take the time to browse through some of them.

The Stat/Math Center (a subdivision of University Information Technology Services' Research and Academic Computing division, University of Indiana) has "Getting Started With MATLAB" at:

<http://www.indiana.edu/%7Estatmath/math/matlab/gettingstarted/index.html>

A well organized "Practical Introduction to Matlab" by Mark Gockenbach is at

<http://www.rpi.edu/%7Eroytbv/numcomp/gockenbach.pdf>

A very friendly "Matlab Primer" by Kermit Sigmon (Department of Mathematics, University of Florida) is at

<http://www.rpi.edu/%7Eroytbv/numcomp/sigmon.pdf>

There's a chapter on-line with nice graphics to give you a good idea of what Matlab looks like. "Matlab Primer", a chapter from Numerical Analysis and Graphic Visualization with Matlab by Shoichiro Nakamura:

<http://www.math.rpi.edu/Faculty/Holmes/Courses/NumDiffEqs/S03/Refs/matlabprimer.pdf>

The Mathworks (which owns Matlab) has a web page with tutorials and help for solving problems using Matlab:

<http://www.mathworks.com/academia/>

REFERENCES

- [1] J. Proakis, M. Salehi, and G. Bauch *Contemporary Communication Systems using Matlab* Thomson Brooks/Cole
- [2] J. Proakis and M. Salehi *Fundamentals of Communications Systems* Pearson Prentice Hall
- [3] Wolfram Research *MathWorld* <http://mathworld.wolfram.com/DiscreteFourierTransform.html>
- [4] J. Proakis and M. Salehi *Communication Systems Engineering, Second Edition* Prentice Hall