Modeling of Wireless Communication Systems using MATLAB

Dr. B.-P. Paris
Dept. Electrical and Comp. Engineering
George Mason University

Approach

- This course aims to cover both
  - theory and practice of wireless commuication systems, and
  - the simulation of such systems using MATLAB.
- Both topics are given approximately equal treatment.
  - After a brief introduction to MATLAB, theory and MATLAB simulation are pursued in parallel.
  - This approach allows us to make concepts concrete and/or to visualize relevant signals.
- In the process, a toolbox of MATLAB functions is constructed.
  - Hopefully, the toolbox will be useful for your own projects.
  - Illustrates good MATLAB practices.

Outline - Prologue: Just Enough MATLAB to ...

Prologue: Learning Objectives

User Interface

Working with Vectors

Visualization

Outline - Part I: From Theory to Simulation

Part I: Learning Objectives

Elements of a Digital Communications System

Digital Modulation

Channel Model

Receiver

MATLAB Simulation

Outline - Part II: Digital Modulation and Spectrum

Part II: Learning Objectives

Linear Modulation Formats and their Spectra

Spectrum Estimation in MATLAB

Non-linear Modulation

Wide-Band Modulation

Outline - Part III: The Wireless Channel

Part III: Learning Objectives

Pathloss and Link Budget

From Physical Propagation to Multi-Path Fading

Statistical Characterization of Channels

Outline - Part IV: Mitigating the Wireless Channel

Part IV: Learning Objectives

The Importance of Diversity

Frequency Diversity: Wide-Band Signals

Part I

Prologue: Just Enough MATLAB to ...

Prologue: Just Enough MATLAB to ...

- MATLAB will be used throughout this course to illustrate theory and key concepts.
- MATLAB is very well suited to model communications systems:
  - Signals are naturally represented in MATLAB,
  - MATLAB has a very large library of functions for processing signals,
  - Visualization of signals is very well supported in MATLAB.
- MATLAB is used interactively.
  - Eliminates code, compile, run cycle.
  - Great for rapid prototyping and what-if analysis.

Outline

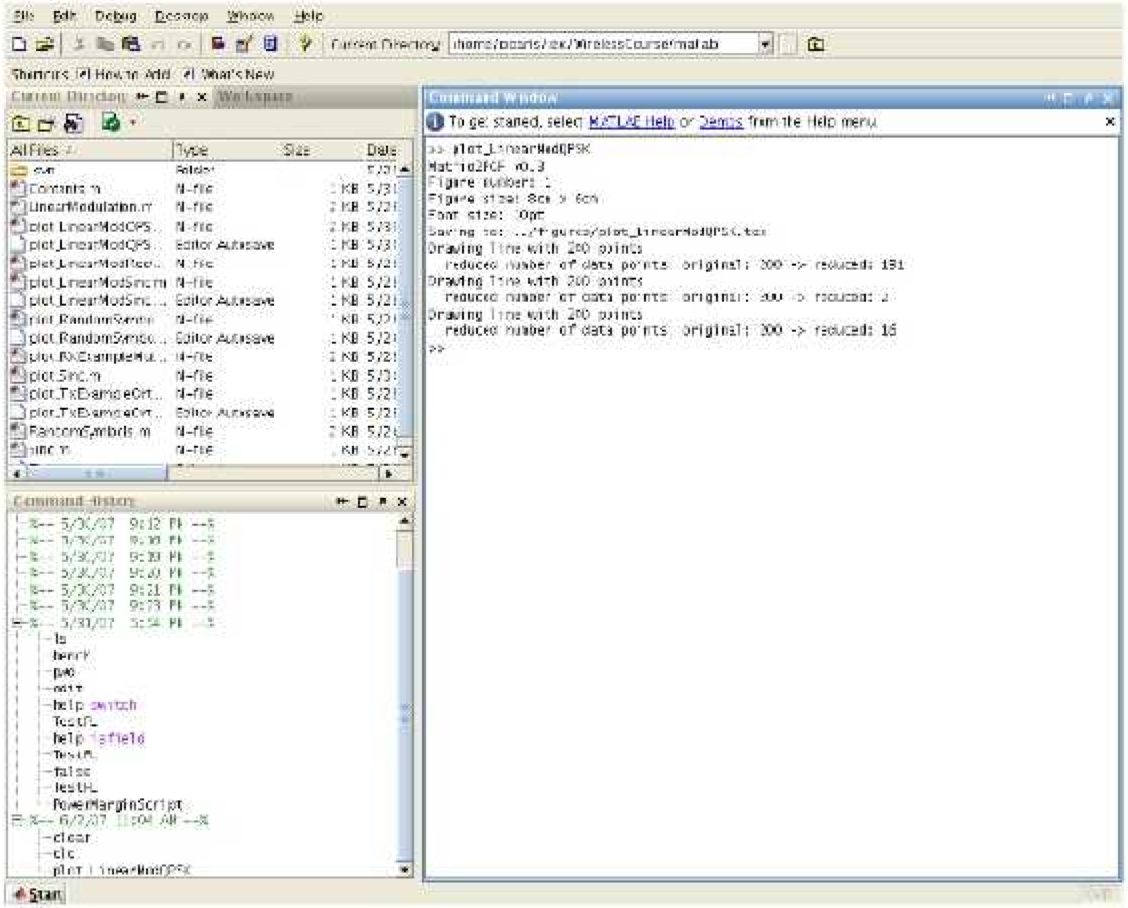
Learning Objectives

- Getting around in MATLAB
  - The user interface,
  - Getting help.
- Modeling signals in MATLAB
  - Using vectors to model signals,
  - Creating and manipulating vectors,
  - Visualizing vectors: plotting.

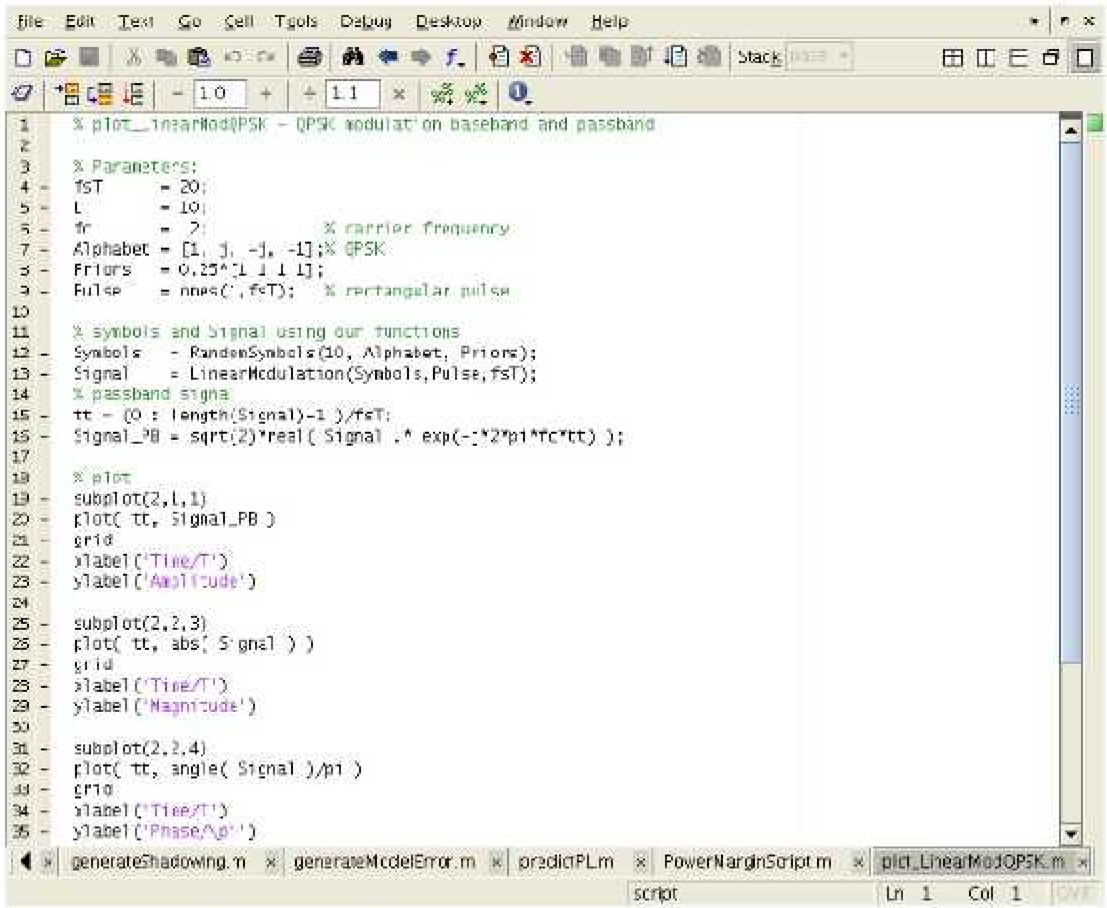
Outline

- Prologue: Learning Objectives
- User Interface
- Working with Vectors
- Visualization

MATLAB's Main Window



MATLAB's Built-in IDE



MATLAB's Built-in Help System

- MATLAB has an extensive built-in help system.
  - On-line documentation reader:
  - contains detailed documentation for entire MATLAB system,
  - is invoked by
    - typing `doc` at command line
    - clicking "Question Mark" in tool bar of main window,
    - via "Help" menu.
  - Command-line help provides access to documentation inside command window.
  - *Help*ful commands include:
    - **`help function`**`-name`, e.g., **`help fft.`**
    - **`lookfor`** `keyword`, e.g., **`lookfor`** `inverse.`
- We will learn how to tie into the built-in help system.

Interacting with MATLAB

- You interact with MATLAB by typing commands at the command prompt (») in the command window.
- MATLAB's response depends on whether a semicolon is appended after the command or not.
  - If a semicolon is **not** appended, then MATLAB displays the result of the command.
  - With a semicolon, the result is not displayed.
- **Examples:**
  - The command `xx = 1:3` produces

    ```
    xx =
    
         1     2     3
    ```
  - The command `xx = 1:3;` produces no output. The variable `xx` still stores the result.
  - Do use a semicolon with `xx = 1:30000000;`

Outline

Prologue: Learning Objectives

User Interface

Working with Vectors

Visualization

Signals and Vectors

- Our objective is to simulate communication systems in MATLAB.
  - This includes the signals that occur in such systems, and
  - processing applied to these signals.
- In MATLAB (and any other digital system) signals must be represented by samples.
  - Well-founded theory exists regarding sampling (Nyquist's sampling theorem).
  - Result: Signals are represented as a sequence of numbers.
- MATLAB is ideally suited to process sequences of numbers.
  - MATLAB's basic data types: vectors (and matrices).
  - Vectors are just sequence of numbers.

Illustration: Generating a Sinusoidal Signal

- The following, simple task illustrates key benefits from MATLAB's use of vectors.
- **Task:** Generate samples of the sinusoidal signal

$$x(t) = 3 \cdot \cos(2\pi 440t - \frac{\pi}{4})$$

  for t ranging from 0 to 10 ms. The sampling rate is 20 KHz.
- **Objective:** Compare how this task is accomplished
  - using MATLAB's vector function,
  - traditional (C-style) `for` or `while` loops.

Illustration: Generating a Sinusoidal Signal

- For both approaches, we begin by defining a few parameters.
  - This increases readability and makes it easier to change parameters.

```
%% Set Parameters
A   = 3;       % amplitude
f   = 440;     % frequency
phi = -pi/4;   % phase
7
fs = 20e3;     % sampling rate
Ts = 0;        % start time
Te = 10e-3;    % end time
```

Using Loops

- The MATLAB code below uses a `while` loop to generate the samples one by one.
  - The majority of the code is devoted to "book-keeping" tasks.

Listing : generateSinusoidLoop.m

```
%initialize loop variables
tcur = Ts;
kk   = 1;

while( tcur <= Te)
    % compute current sample and store in vector
    tt(kk) = tcur;
    xx(kk) = A*cos(2*pi*f*tcur + phi);

    %increment loop variables
    kk   = kk+1;
    tcur = tcur + 1/fs;
end
```

Vectorized Code

- Much more compact code is possible with MATLAB's vector functions.
  - There is no overhead for managing a program loop.
  - Notice how similar the instruction to generate the samples is to the equation for the signal.
- The vector-based approach is the key enabler for rapid prototyping.

Listing : generateSinusoid.m

```
%% generate sinusoid
tt = Ts : 1/fs : Te;    % define time-axis
xx = A * cos( 2*pi * f * tt + phi );
```

Commands for Creating Vectors

- The following commands all create useful vectors.
- `[ ]`: the sequence of samples is explicitly specified.
  - **Example:** `xx = [ 1 3 2 ]` produces `xx = 1 3 2`.
- `:` (colon operator): creates a vector of equally spaced samples.
  - **Example:** `tt = 0:2:9` produces `tt = 0 2 4 6 8`.
  - **Example:** `tt = 1:3` produces `tt = 1 2 3`.
  - **Idiom:** `tt = ts:1/fs:te` creates a vector of sampling times between `ts` and `te` with sampling period `1/fs` (i.e., the sampling rate is `fs`).

Creating Vectors of Constants

- `ones(n,m)`: creates an $n \times m$ matrix with all elements equal to 1.
  - **Example:** `xx = ones(1,5)` produces `xx = 1 1 1 1 1`.
  - Example: `xx = 4*ones(1,5)` produces `xx = 4 4 4 4 4`.
- **`zeros`**`(n,m)`: creates an $n \times m$ matrix with all elements equal to 0.
  - Often used for initializing a vector.
  - Usage identical to `ones`.
- Note: throughout we adopt the convention that signals are represented as row vectors.
  - The first (column) dimension equals 1.

Creating Random Vectors

- We will often need to create vectors of random numbers.
  - E.g., to simulate noise.
- The following two functions create random vectors.
  - `randn(n,m)`: creates an $n \times m$ matrix of independent Gaussian random numbers with mean zero and variance one.
    - **Example:** `xx = randn(1,5)` may produce `xx = -0.4326 -1.6656 0.1253 0.2877 -1.1465`.
  - `rand(n,m)`: creates an $n \times m$ matrix of independent uniformly distributed random numbers between zero and one.
    - **Example:** `xx = rand(1,5)` may produce `xx = 0.1576 0.9706 0.9572 0.4854 0.8003`.

Addition and Subtraction

- The standard + and − operators are used to add and subtract vectors.
- One of two conditions must hold for this operation to succeed.
  - Both vectors must have exactly the same size.
    - In this case, corresponding elements in the two vectors are added and the result is another vector of the same size.
    - **Example:** `[1 3 2] + 1:3` produces `2 5 5`.
    - A prominent error message indicates when this condition is violated.
  - One of the operands is a scalar, i.e., a 1×1 (degenerate) vector.
    - In this case, each element of the vector has the scalar added to it.
    - The result is a vector of the same size as the vector operand.
    - **Example:** `[1 3 2] + 2` produces `3 5 4`.

Element-wise Multiplication and Division

- The operators `.*` and `./` operators multiply or divide two vectors element by element.
- One of two conditions must hold for this operation to succeed.
  - Both vectors must have exactly the same size.
    - In this case, corresponding elements in the two vectors are multiplied and the result is another vector of the same size.
    - **Example:** `[1 3 2] .* 1:3` produces `1 6 6`.
    - An error message indicates when this condition is violated.
  - One of the operands is a scalar.
    - In this case, each element of the vector is multiplied by the scalar.
    - The result is a vector of the same size as the vector operand.
    - **Example:** `[1 3 2] .* 2` produces `2 6 4`.
    - If one operand is a scalar the '.' may be omitted, i.e., `[1 3 2] * 2` also produces `2 6 4`.

Inner Product

- The operator $\star$ with two vector arguments computes the inner product (dot product) of the vectors.
  - Recall the inner product of two vectors is defined as

$$\vec{x}' \cdot \vec{y} = \sum_{n=1}^{N} x(n) \cdot y(n).$$

  - This implies that the result of the operation is a scalar!
- The inner product is a useful and important signal processing operation.
  - It is very different from element-wise multiplication.
  - The second dimension of the first operand must equal the first dimension of the second operand.
    - MATLAB error message: `Inner matrix dimensions must agree.`
  - **Example:** `[1 3 2] * (1:3)' = 13`.
    - The single quote (`'`) transposes a vector.

Powers

- To raise a vector to some power use the `.^` operator.
  - **Example:** `[1 3 2].^2` yields `1 9 4`.
  - The operator `^` exists but is generally not what you need.
    - **Example:** `[1 3 2]^2` is equivalent to `[1 3 2] * [1 3 2]` which produces an error.
- Similarly, to use a vector as the exponent for a scalar base use the `.^` operator.
  - **Example:** `2.^[1 3 2]` yields `2 8 4`.
- Finally, to raise a vector of bases to a vector of exponents use the `.^` operator.
  - **Example:** `[1 3 2].^(1:3)` yields `1 9 8`.
  - The two vectors must have the same dimensions.
- The `.^` operator is (nearly) always the right operator.

Complex Arithmetic

- MATLAB support complex numbers fully and naturally.
  - The imaginary unit $i = \sqrt{-1}$ is a built-in constant named `i` and `j`.
  - Creating complex vectors:
    - **Example:** `xx = randn(1,5) + j*randn(1,5)` creates a vector of complex Gaussian random numbers.
- A couple of "gotchas" in connection with complex arithmetic:
  - Never use `i` and `j` as variables!
    - **Example:** After invoking `j=2`, the above command will produce very unexpected results.
  - Transposition operator (`'`) transposes and forms conjugate complex.
    - That is very often the right thing to do.
    - Transpose only is performed with `.'` operator.

Vector Functions

- MATLAB has literally hundreds of built-in functions for manipulating vectors and matrices.
- The following will come up repeatedly:
  - `yy=cos(xx)`, `yy=sin(xx)`, and `yy=exp(xx)`:
    - compute the cosine, sine, and exponential for each element of vector `xx`,
    - the result `yy` is a vector of the same size as `xx`.
  - `XX=fft(xx)`, `xx=ifft(XX)`:
    - Forward and inverse discrete Fourier transform (DFT),
    - computed via an efficient FFT algorithm.
  - Many algebraic functions are available, including **`log10`**, **`sqrt`**, **`abs`**, and **`round`**.
    - Try **`help`** `elfun` for a complete list.

Functions Returning a Scalar Result

- Many other functions accept a vector as its input and return a scalar value as the result.
- Examples include
  - **min** and **max**,
  - **mean** and `var` or **std**,
  - **sum** computes the sum of the elements of a vector,
  - **norm** provides the square root of the sum of the squares of the elements of a vector.
    - The norm of a vector is related to power and energy.
- Try **help** `datafun` for an extensive list.

Accessing Elements of a Vector

- Frequently it is necessary to modify or extract a subset of the elements of a vector.
  - Accessing a single element of a vector:
    - **Example:** Let `xx = [1 3 2]`, change the third element to 4.
    - **Solution:** `xx(3) = 4;` produces `xx = 1 3 4`.
  - Single elements are accessed by providing the index of the element of interest in parentheses.

Accessing Elements of a Vector

- Accessing a range of elements of a vector:
  - **Example:** Let `xx = ones(1,10);`, change the first five elements to -1.
  - **Solution:** `xx(1:5) = -1*ones(1,5);` Note, `xx(1:5) = -1` works as well.
  - **Example:** Change every other element of `xx` to 2.
  - **Solution:** `xx(2:2:end) = 2;`;
    - Note that **`end`** may be use to denote the index of a vector's last element.
    - This is handy if the length of the vector is not known.
  - **Example:** Change third and seventh element to 3.
  - **Solution:** `xx([3 7]) = 3;`;
- A set of elements of a vector is accessed by providing a vector of indices in parentheses.

Accessing Elements that Meet a Condition

- Frequently one needs to access all elements of a vector that meet a given condition.
  - Clearly, that could be accomplished by writing a loop that examines and processes one element at a time.
  - Such loops are easily avoided.
- **Example:** "Poor man's absolute value"
  - Assume vector `xx` contains both positive and negative numbers. (e.g., `xx = randn(1,10);`).
  - **Objective:** Multiply all negative elements of `xx` by -1; thus compute the absolute value of all elements of `xx`.
  - **Solution:** proceeds in two steps
    - `isNegative = (xx < 0);`
    - `xx(isNegative) = -xx(isNegative);`
  - The vector `isNegative` consists of logical (boolean) values; 1's appear wherever an element of `xx` is negative.

Outline

Prologue: Learning Objectives

User Interface

Working with Vectors

Visualization

Visualization and Graphics

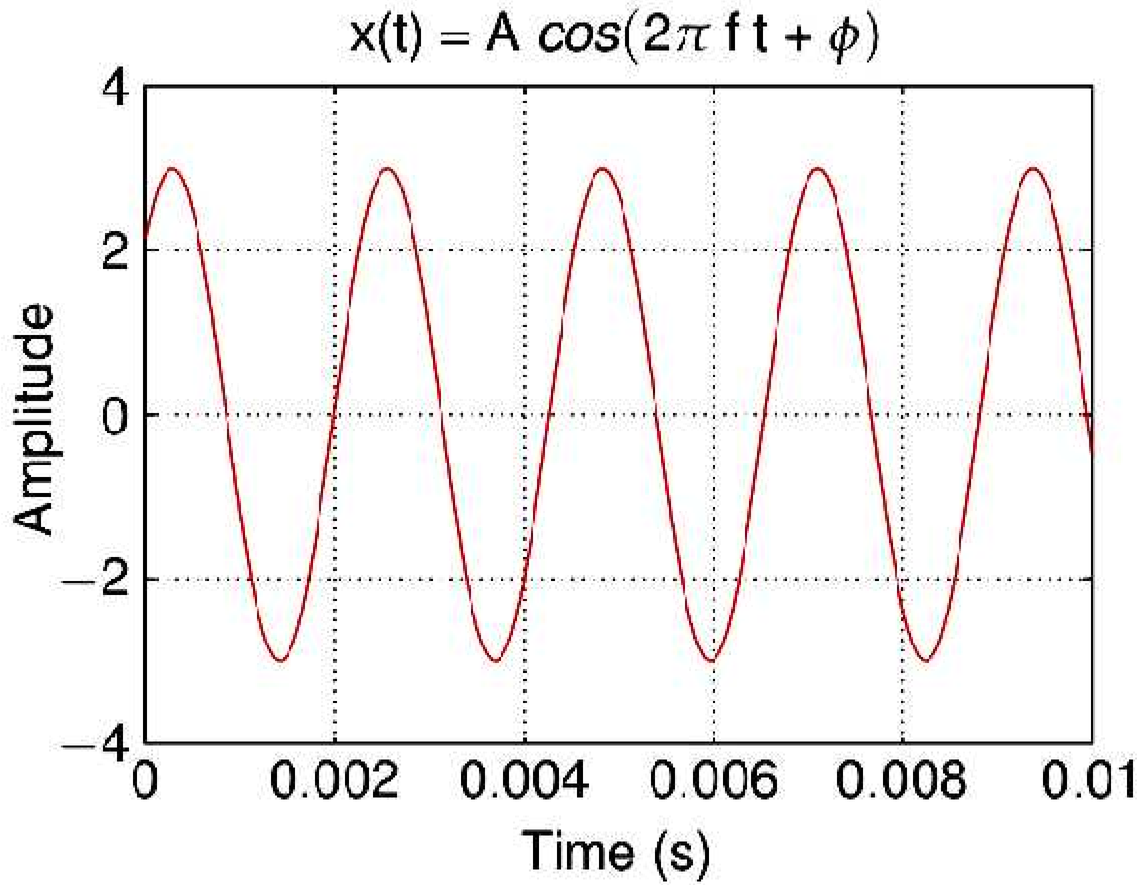
- MATLAB has powerful, built-in functions for plotting functions in two and three dimensions.
- Publication quality graphs are easily produced in a variety of standard graphics formats.
- MATLAB provides fine-grained control over all aspects of the final graph.

A Basic Plot

- The sinusoidal signal, we generated earlier is easily plotted via the following sequence of commands:
- Try **`help plot`** for more information about the capabilities of the **`plot`** command.

```
%% Plot
plot(tt, xx, 'r')          % xy-plot, specify red line
xlabel( 'Time (s)' )       % labels for x and y axis
ylabel( 'Amplitude' )
title( 'x(t) = A cos(2\pi f t + \phi)')
grid                       % create a grid
axis([0 10e-3 -4 4])       % tweak the range for the axes
```

Resulting Plot



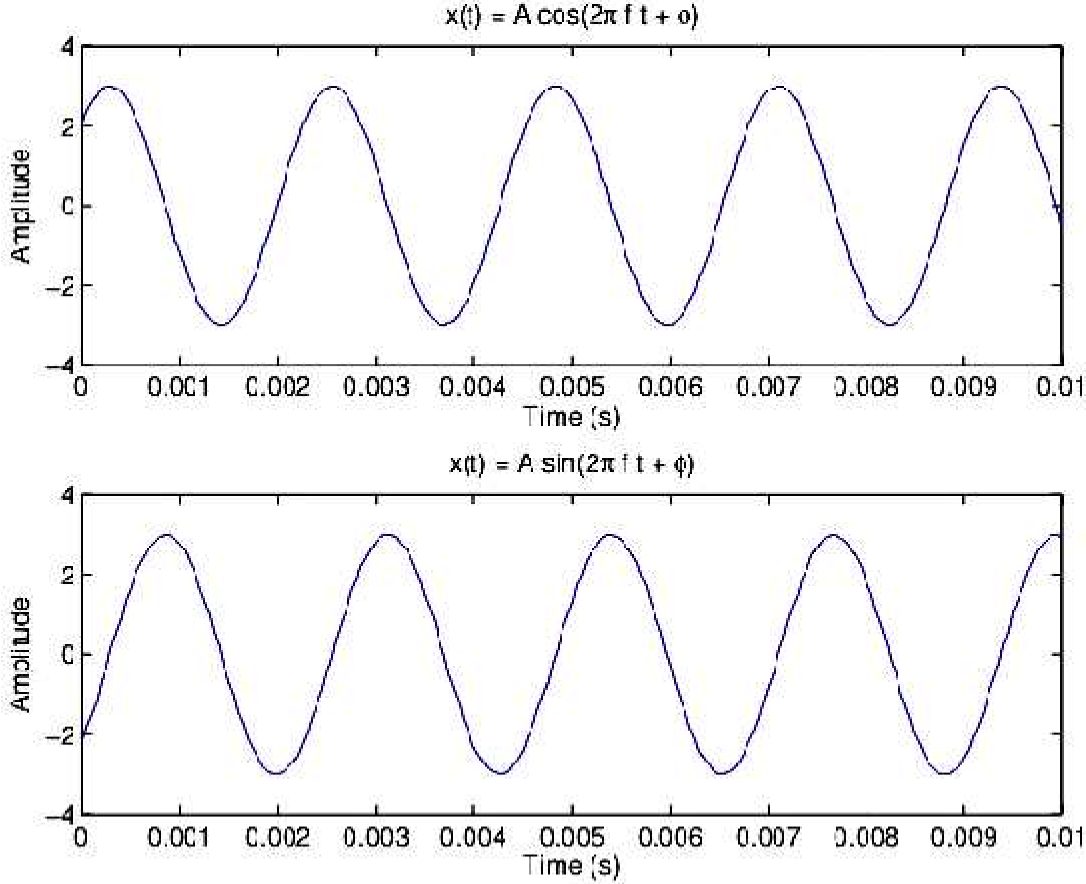
Multiple Plots in One Figure

- MATLAB can either put multiple graphs in the same plot or put multiple plots side by side.
- The latter is accomplished with the `subplot` command.

```
subplot(2,1,1)
plot(tt, xx )                % xy-plot
xlabel( 'Time␣(s)' )         % labels for x and y axis
ylabel( 'Amplitude' )
title( 'x(t)␣=␣A␣cos(2\pi␣f␣t␣+␣\phi)')

subplot(2,1,2)
plot(tt, yy )                % xy-plot
xlabel( 'Time␣(s)' )         % labels for x and y axis
ylabel( 'Amplitude' )
title( 'x(t)␣=␣A␣sin(2\pi␣f␣t␣+␣\phi)')
```

Resulting Plot



3-D Graphics

- MATLAB provides several functions that create high-quality three-dimensional graphics.
- The most important are:
  - **plot3**(x,y,z): plots a function of two variables.
  - **mesh**(x,y,Z): plots a mesh of the values stored in matrix Z over the plane spanned by vectors x and y.
  - **surf**(x,y,Z): plots a surface from the values stored in matrix Z over the plane spanned by vectors x and y.
- A relevant example is shown on the next slide.
  - The path loss in a two-ray propagation environment over a flat, reflecting surface is shown as a function of distance and frequency.

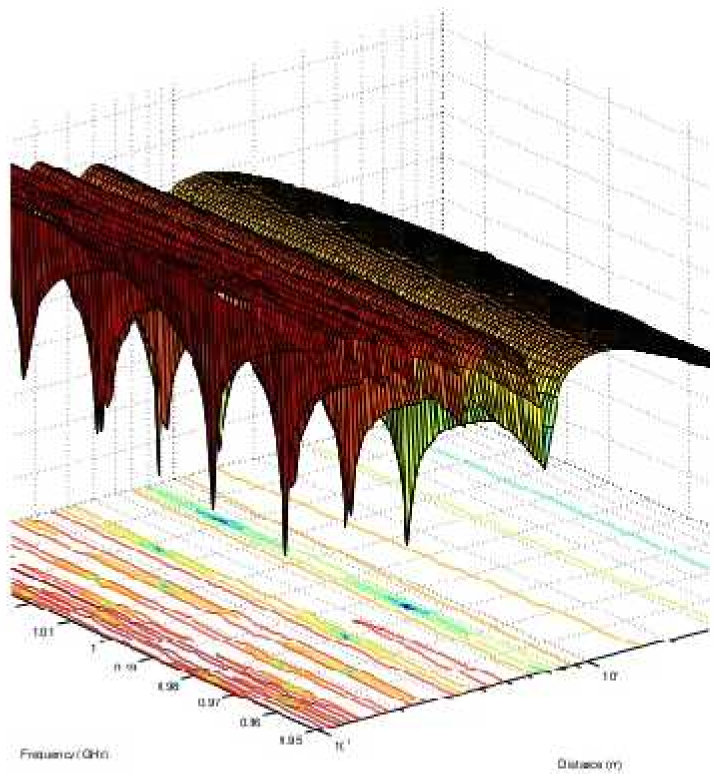


Figure: Path loss over a flat reflecting surface.

Summary

- We have taken a brief look at the capabilities of MATLAB.
- Specifically, we discussed
  - Vectors as the basic data unit used in MATLAB,
  - Arithmetic with vectors,
  - Prominent vector functions,
  - Visualization in MATLAB.
- We will build on this basis as we continue and apply MATLAB to the simulation of communication systems.
- To probe further:
  - Read the built-in documentation.
  - Recommended MATLAB book: D. Hanselman and B. Littlefield, *Mastering MATLAB*, Prentice-Hall.