# A VERY BASIC MATLAB PRIMER

JACEK POLEWCZAK

## Contents

1. A Few Elementary Calculations 2
2. Finding Help in Matlab 2
3. Working with Arrays of Numbers 3
4. Creating Script Files and Simple Plots 4
5. Creating and Executing a Function File 5
6. Saving/Loading Data, Recording a Session, Clearing Variables, and More on Plotting 7
7. Matrices and Vectors 8
7.1. Appending or Deleting a Row or Column 10
7.2. Matrix Operations 11
7.3. Array Operations 12
7.4. Matrix Functions 13
8. Character Strings 14
8.1. `eval` Function 15
9. `for` and `while` Loops and `if` Conditionals 16
10. Interactive Input 18
11. Input/Output Commands 19
12. Linear Algebra 23
12.1. Solving Linear Square Systems 23
12.2. Gaussian Elimination 25
12.3. Inverses, Determinants, and Cramer's Rule 26
12.4. Pseudoinverses or Generalized Inverses 28
12.5. Underdetermined and Overdetermined Linear Systems 29
12.6. Finding Eigenvalues and Eigenvectors 35
12.7. Generalized Eigenvectors 38
12.8. Matrix Factorizations, Matrix Powers and Exponentials 41
12.9. Characteristic Polynomial of a Matrix and Cayley-Hamilton Theorem 46
13. Curve Fitting and Interpolation 47
13.1. Hands-on Understanding of Curve Fitting Using Polynomial Functions 48
13.2. Interpolation 49
14. Solving Nonlinear Algebraic Equations 50
15. Numerical Integration - Quadrature 51
15.1. Quadrature For Double Integrals 52
16. Solving Ordinary Differential Equations 52

*Date*: Last updated November 2009.

Some of the examples in this tutorial are the modified versions of the items from Rudra Pratap's, *Getting started with Matlab*, Oxford University Press, 2002.

16.1. Solving Linear Systems of First Order Differential Equations 57
16.2. Linearization of a Double Pendulum 65
17. Use of Global Variables 68
18. Publishing Documents and Opening Web Pages in Matlab 69
19. How to Download the Matlab Files Used in this Tutorial 72

## 1. A Few Elementary Calculations

```
>> 2+4
ans =
     6
```

Enter $2+4$ and hit return/enter key.

```
>> x = 2+5
ans =
     7
```

Assigning the value of an expression to a variable.

```
>> z= 3^4+log(pi)*cos(x);
>> z

   81.8630
```

A semicolon suppresses screen output. You can recall the value of $z$ by typing $z$

```
>> format short e
>> z
z =
   8.1863e+01
```

The floating point output display is controlled by the `format` command.

```
>> format long
>> z
z =
  81.863014441556018
>> format short
>> z
z =
   81.8630
>> quit
```

Quit Matlab. Alternatively you can type `exit` command or select `Exit Matlab` from the File Menu.

*For additional help type in Matlab:* `help format`

## 2. Finding Help in Matlab

There are several ways to get help in Matlab:

- `help functionname` provides direct on-line help on the function whose eaxct name you know. For example `help help` provides help on the function `help`.
- `lookfor keyword` provides a list functions with the string `keyword` in their description. For example,

```
>> lookfor prime
isprime                       - True for prime numbers.
factor                        - Prime factors.
primes                        - Generate list of prime numbers.
```

By clicking on `isprime`, `factor`, or `primes` additional help on those functions will be listed.

- `helpwin` provides the click and navigate help. A new help windows opens. One can also select help choices from the *Help* menu.
- `helpdesk` is the web browser-based help, including printable (PDF) documentation on the Web.
- `demo` provides the demonstration programs on Matlab.

Additionally, one can find detailed online help on Matlab at:

`http://www.mathworks.com/access/helpdesk/help/techdoc/index.html`

## 3. Working with Arrays of Numbers

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

$x$ is a row vector with $3$ elements.

```
>> size(x)
ans =
     1     3
```

Size of array. Here, 1 row and 3 columns.

```
>> y = [4; 5; 6]
y =
     4
     5
     6
```

$y$ is a column vector with $3$ elements.

```
>> size(y)
ans =
     3     1
```

Size of `y`: 3 rows and 1 column.

```
>> x'
ans =
     1
     2
     3
```

x' creates a column vector from `x`.

```
>> b = [3 -1 0]
b =
     3    -1     0
>> a = x+b
a =
     4     1     3
```

Addition of two vectors of the same size. The same holds for subtraction.

```
>> x+y
??? Error using ==> plus
Matrix dimensions must agree.
```

Addition or subtraction of a row vector to a column vector is not allowed.

```
>> c = x.*a
c =
     4     2     9
>> d = x./a
d =
    0.2500    2.0000    1.0000
```

One can multiply (or divide) the elements of two same-sized vectors term by term with the *array operator*: .* (or ./). The array operator .^ produces term by term exponentiation.

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

However, `x*a` produces an error.

```
>> X = linspace(-2,7,4)
X =
    -2     1     4     7
>> X = -2:3:7
X =
    -2     1     4     7
```

linspace(a,b,n) creates a linearly spaced vector of length $n$ from $a$ to $b$. Note that u=linspace(a,b,n) is the same as u=a:(b-a)/(n-1):b, another way of creating vectors (see also Section 7).

```
>> y = cos(X);
>> z = sqrt(abs(X)).*y
z =
-0.5885    0.5403   -1.3073    1.9946
```

Elementary math functions operate on vectors term by term.

## 4. Creating Script Files and Simple Plots

A script file is an external file that contains a sequence of Matlab statements. By typing the filename, subsequent Matlab input is obtained from the file. Script files have a filename extension of .m and are often called *M-files*.

Select New from *File* menu (alternatively, use any text editor) and type the following lines into this file:

```
% ELLIPSE_PARABOLA - A script file to draw ellipse (x/2)^2+(y/3)^2=1 and parabola y=x^2.
alpha = linspace(0,2*pi,100);
x = 2*cos(alpha);
y = 3*sin(alpha);
t = linspace(-2,2,100);
z = t.^2;
plot(x,y,t,z);
axis('equal');
xlabel('x')
ylabel('y')
title('Ellipse (x/2)^2+(y/3)^2=1 and parabola y=x^2')
clear alpha x y t z;
```

Lines starting with % sign are ignored; they are interpreted as comment lines by Matlab.

Select Save As... from *File* menu and type the name of the document as, e.g., ellipse_parabola.m. Alternatively, if you are using an external editor, use a suitable command to save the document in the file with .m extension.

Once the file is placed in the current working directory of Matlab, type the following command in the command window to execute ellipse_parabola.m script file.

```
>> ellipse_parabola
```

Execute the file. You should see the plot of the ellipse $\left(\frac{x}{2}\right)^2 + \left(\frac{y}{3}\right)^2 = 1$ and parabola $y = x^2$ (see Figure 1, below).

```
>> print
```

Print the created graph on the default printer.

```
>> print my_graph.eps -depsc
```

Save the graph in (color) Encapsulated PostScript format in the file my_graph.eps. Use -djpeg, -dtiff, or -dpng options for JPEG, TIFF, or PNG graphic formats, respectively.

The above script file also illustrates how to generate plot $x$ vs $y$ from the generated $x$ and $y$ coordinates of 100 equidistant points between 0 and $2\pi$. For example, in the above script change the entry plot(x,y,t,z)

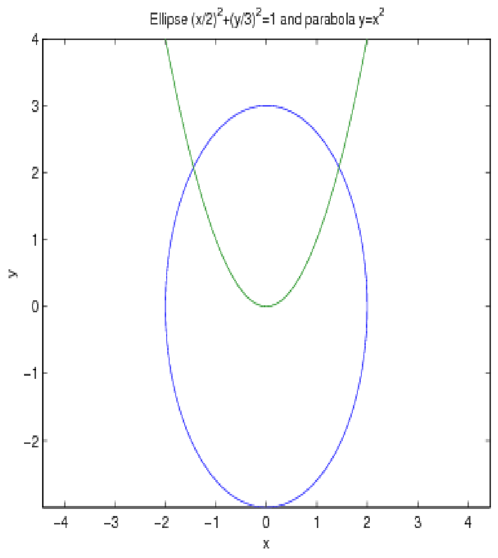


FIGURE 1.

to `plot(x,y,'--',t,z,'o')` and see the difference in graphs drawings. For more information, check Matlab help on `plot` command and its options by invoking `help plot` in the command window of Matlab. For `print` options enter `help print` in the command window of Matlab.

## 5. Creating and Executing a Function File

Function file is a script file (M-file) that adds a function definition to Matlab's list of functions. Edit `ellipse_parabola.m` script file (listed in Section 4 to look like the following:

```
function circle_f1(r);
% CIRCLE_F1 - Function to draw circle x^2+r^2=r^2.
alpha = linspace(0,2*pi,100);
x = r*cos(alpha);
y = r*sin(alpha);
plot(x,y);
axis('equal');
xlabel('x')
ylabel('y')
title(['Circle of radius r =',num2str(r)]) % put a title with the value r.
```

Save it in `circle_f1.m` file (remember to place it in the working directory of Matlab) and try it out.

```
>> radius=3;
>> circle_f1(radius)
```

Specify the input and execute the function. You should see the graph of the circle with radius 3. You can also just enter `circle_f1(2.4)` to draw the circle with radius $r = 2.4$.

And here is slightly different version of `circle_f1.m` function file, called `circle_f2.m` This function file, in addition to drawing a circle with radius $r$, computes the area inside this circle together with the outputs for x and y coordinates.

```
function [A,x,y] = circle_f2(r);
% CIRCLE_F2 - Function draws circle x^2+r^2=r^2
% and output its area together with x and y coordinates.
alpha = linspace(0,2*pi,100);
x = r*cos(alpha);
y = r*sin(alpha);
```

```
A = pi*r^2;
plot(x,y);
axis('equal');
xlabel('x')
ylabel('y')
title(['Circle of radius r =',num2str(r)]) %put a title with the value r.
```

Save it in `circle_f2.m` file (remember to place it in the working directory of Matlab) and execute it in one of the following ways:

```
>> circle_f2(5)
>> [Area] = circle_f2(5)
>> circle_f2(5);
>> [Area] = circle_f2(5);
```

The first two commands output the graph of a circle with radius $5$ and its area. The last two commands create the graph while suppressing the output for the area.

```
>> [AREA,X,Y] = circle_f2(5)
```

The complete output of the function is displayed: the graph of a circle with radius $5$ and its area, together with the outputs for x and y coordinates.

Finally, suppose you need to evaluate the following function many times at different values of $x$ during your Matlab session: $f(x) = \exp(-0.3x)\sin(30x) + \frac{x^{10}}{1+x^{10}}$. You can create an inline function and pass its name to other functions:

```
>> fun = inline('exp(-0.3*x).*sin(30*x)+x.^10./(1+x.^10)');
>> fun(2)
ans =
    0.8317
>> sin(fun(1))+fun(cos(1))
ans =
   -0.6363
>> x=linspace(0,10,100);
>> plot(x,fun(x))
>> x0=fzero(fun,10)
x0 =
    1.0193
```

Use Matlab help command `help fzero` to check what this last command computes.

In the case you want to use function $f(x) = \exp(-0.3x)\sin(30x) + \frac{x^{10}}{1+x^{10}}$ in future Matlab sessions, create the following function file:

```
function f = my_f(x);
% MY_F evaluates function f = exp(0.3x)sin(30x)+x^10/(1+x^10)
f = exp(-0.3*x).*sin(30*x)+x.^10./(1+x.^10);
```

and save it in `my_f.m` file (remember to place it in the working directory of Matlab). You can use it in many ways:

```
>> my_f(2)
ans =
    0.8317
>> sin(my_f(1))+my_f(cos(1))
ans =
   -0.6363
>> x=linspace(0,10,100);
>> plot(x,my_f(x))
>> x0=fzero(@my_f,10)
x0 =
    1.0193
```

Use the command `help function_handle` to check what is the meaning of `@` symbol in the last command.

And here is Matlab function that for a given $x \in \mathbb{R}$ and given $n \in \mathbb{N}$ computes

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!},$$

```
function s=expseries(x,n);
% EXPSERIES: function calculates the sum
% 1+x+(x^2)/2!+(x^3)/3!+...+(x^n)/n! (up to nth power).
% Call syntax:
%         s=expseries(x,n);
% where x is real and n nonnegative integer.
k = 0:n;                         % create a vector from 0 to n
series = (x.^k)./factorial(k);   % create a vector of terms in the series
s = sum(series);                 % sum all terms of the vector 'series'.
```

Save the script in `expseries.m` file (remember to place it in the working directory of Matlab) and try it out. Alternatively, you can also download it from:

http://www.csun.edu/~hcmth008/matlab_tutorial/expseries.m

## 6. Saving/Loading Data, Recording a Session, Clearing Variables, and More on Plotting

| | |
|---|---|
| `>> save` | Saves all workspace variables in the file `matlab.mat`. |
| `>> save FILENAME.dat` | Saves all workspace variables in the file `FILENAME.dat`. If `FILENAME` has no extension, `.mat` is assumed. |
| `>> save X` | Saves variable X in the file `X.mat`. |
| `>> save FILENAME.dat X` | Saves variable X in the file `FILENAME.dat`. If `FILENAME` has no extension, `.mat` is assumed. |
| `>> load` | Loads the variables saved in the default file `matlab.mat`. |
| `>> load FILENAME` | Loads the variables saved in the file `FILENAME.mat`. |
| `>> diary FILENAME` | Causes a copy of all subsequent command window input and most of the resulting command window output to be appended to the named file. If no file is specified, the file `diary` is used. |
| `>> diary off` | Suspends diary. |
| `>> diary on` | Turns diary back on. The command `diary`, by itself, toggles the diary state. |
| `>> clear X` | Removes variable `X` from memory. |
| `>> clear all` | `clear all` command removes all variables, functions and MEX links. Also `clear all` at the command prompt removes the Java packages import list. It is a good idea to start a Matlab session with this command ! |

In addition to `plot` command, introduced in Section 4, Matlab has many plotting utilities. Here are sample uses of two of them.

### `fplot` command

```
>> fplot('exp(-x).*sin(30.*x)',[-1,5])
>> xlabel('x'),ylabel('f(x)=e^{-x} sin(30x)')
>> title('How to use fplot')
```

Plots a function between $-1$ and $5$. The last two commands are optional and only annotate the graph with the title and other details.

```
>> F = 'exp(-x).*sin(30.*x)';
>> fplot(F,[0,10])
>> F = inline('exp(-0.3*x).*sin(30*x)');
>> fplot(F,[0,10])
>> fplot('[my_f(0.5*x),cos(10*x),sin(x)^2]',[-7,7])
```

Other uses of `fplot`, including multiple plots. Here, `my_f` is the function file defined in Section 5.

```
>> [X,Y] = fplot('exp(-x).*sin(30.*x)',[-1,1])
>> X(1)
ans =
    -1
>> Y(1)
ans =
    2.6857
```

Returns X and Y such that $Y = \exp(-X)\sin(30X)$ (output not provided here). No plot is drawn on the screen. One can check the length of vectors $X$ and $Y$ with `length(X)` and `length(Y)` commands. Also, `X(i)` or `Y(k)` provide $i$-th and $k$-th elements of vectors $X$ and $Y$, respectively.

### `ezplot` command

```
>> F = 'exp(-x).*sin(30.*x)';
>> ezplot(F)
>> F = inline('exp(-0.3*x).*sin(30*x)');
>> ezplot(F)
>> ezplot('my_f')
>> ezplot('my_f(3*x^2)')
```

Plots the function over the default domain $-2\pi < x < 2\pi$ and with the default title. Also, `ezplot(fun,[min,max])` plots `fun(x)` over the domain: `min < x < max`.

```
>> ezplot('x^2*y^2-sin(x^2+y^2)=1')
>> ezplot('t*sin(t)','1+cos(2*t)^2')
```

Additionally, `ezplot` can be used for implicitly defined functions and parametrically defined curves. For parametric curves $(x(t), y(t))$, the default domain is $0 < t < 2\pi$.

For other plotting utilities check Matlab help on the following commands:
`ezcontour, ezcontourf, ezmesh, ezmeshc, ezplot3, ezpolar, ezsurf, ezsurfc`.

## 7. Matrices and Vectors

```
>> A=[0 1 2; 3, 4, 5; 6 7 8]
A =
     0     1     2
     3     4     5
     6     7     8
```

While entering matrices, rows are separated by semicolons and columns are separated by spaces or commas.

```
>> A=[0 1 2
3 4 5
6 7 8]
A =
     0     1     2
     3     4     5
     6     7     8
```

A matrix can be also entered across multiple lines using carriage returns at the end of each row.

```
>> A=[0 1 ...
2; 3 4 5; 6 7 8]
A =
     0     1     2
     3     4     5
     6     7     8
```

Three consecutive periods (ellipsis) signal continuation of the input on the next line.

```
>> A(3,2)
ans =
     7
```

Element $A_{ij}$ of matrix $A$ is accessed by `A(i,j)`.

```
>> A(3,:)
ans =
     6     7     8

>> A(:,2)
ans =
     1
     4
     7
```

The colon by itself as a row or column index specifies all rows or columns of the matrix.

```
>> A(:,2:3)
ans =
     1     2
     4     5
     7     8

>> Z = [-2.4 1 5 2.6 0]
Z =
   -2.4000   1.0000   5.0000   2.6000   0
```

$Z$ is a row vector with $5$ elements.

```
>> D=[1;-3;sin(1);exp(1)]
D =
    1.0000
   -3.0000
    0.8415
    2.7183
```

$D$ is a column vector with $4$ elements.

```
>> v=1:10:111
v =
     1    11    21    31    41    51    61    71    81    91   101   111
```

The command `v=InitValue:Increment:FinalValue` creates a vector over a given range with a specified increment, while `x=InitValue:FinalValue` produces a vector over a given range with increment $1$.

```
>> x=-2:10

x =
    -2    -1     0     1     2     3     4     5     6     7     8     9    10
```

The command `linspace(a,b,n)` creates a linearly spaced (row) vector of length $n$ from $a$ to $b$. Note that `u=linspace(a,b,n)` is the same as the command `u=a:(b-a)/(n-1):b`.

Also, `logspace(a,b,n)` generates a logarithmically spaced (row) vector of length $n$ from $10^a$ to $10^b$. For example,

```
>> v=logspace(0,4,5)
v =
           1          10         100        1000       10000
```

Finally, `logspace(a,b,n)` is the same as `10.^(linspace(a,b,n)` where `.^` is term by term operator of exponentiation (see Section 7.3).

There are many ways to create special vectors, for example, vectors of zeros or ones of specific length:

```
>> u=zeros(1,5)
u =
     0     0     0     0     0
```

```
>> u=ones(2,1)
u =
     1
     1
```

7.1. **Appending or Deleting a Row or Column.** A row/column can be easily appended or deleted, provided the row/column has the same length as the length of the rows/columns of the existing matrix.

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

The command `eye(m,n)` produces $m$ by $n$ matrix with ones on the main diagonal. `eye(n)` produces $n \times n$ matrix with ones on the main diagonal. The commands `zeros`, `ones`, `rand` work in a similar way.

```
>> v=[5; 6; 7]; C=[C v]
C =
     1     0     0     5
     0     1     0     6
     0     0     1     7
```

The command `C=[C v]` appends the column vector `v` to the columns of $C$.

```
>> C=eye(3); v=[5 6 7]; C=[C; v]
C =
     1     0     0
     0     1     0
     0     0     1
     5     6     7
```

The command `C=[C; v]` appends the row vector `v` to the rows of $C$.

```
>> A=[0 1 2 3; 4 5 6 7; 8 9 -1 -2]
A =
     0     1     2     3
     4     5     6     7
     8     9    -1    -2
```

```
>> A(:,2)= []
A =
     0     2     3
     4     6     7
     8    -1    -2
```

The command `A(:,2)=[]` deletes the second column of $A$.

```
>> A=[0 1 2 3; 4 5 6 7; 8 9 -1 -2]
A =
     0     1     2     3
     4     5     6     7
     8     9    -1    -2
```

```
>> A(2:3,:)=[]
A =

     0     1     2     3
```

The command `A(2:3,:)=[]` deletes the second and third row of $A$.

7.2. **Matrix Operations.** In Matlab, as in Pascal, Fortran, or C, matrix product is just `C=A*B`, where $A$ is an $m \times n$ matrix and $B$ is $n \times k$ matrix, and the resulting matrix $C$ has the size $m \times k$.

- `A+B` and `A-B` are valid if the matrices $A$ and $B$ have the same size (i.e., the same number of rows and the same number of columns).
- `A*B` is valid if $A$'s number of columns equals $B$'s number of rows. The command `A^2`, which equals `A*A`, makes sense if $A$ is a square matrix.
- `X=A/B` (*right* division) is valid if $A$ and $B$ have the same number of columns and it denotes the solution to the matrix equation $XA = B$. In particular, if $A$ and $B$ are square matrices, $A/B \equiv AB^{-1}$, if $B^{-1}$ exists.
- `X=A\B` (*left* division) is valid if $A$ and $B$ have the same number of rows and it denotes the solution to the matrix equation $AX = B$. In particular, if $A$ and $B$ are square matrices, $A\backslash B \equiv A^{-1}B$, if $A^{-1}$ exists.

If $A$ is $m \times p$ and $B$ is $p \times n$, their product $C = AB$ is $m \times n$. Matlab defines their product by `C = A*B`, though this product can be also defined using Matlab `for` loops (see Section 9) and `colon` notation considered in the Section 7.1. However, the internally defined *vectorized* form of the product `A*B` is more efficient; in general, such *vectorizations* are strongly recommended, whenever possible.

```
>> A=pascal(3); B=magic(3);       >> A=pascal(3); B=magic(3);
                                  m=3; n=3;
                                  for i = 1:m
                                  for j = 1:n
                                  C(i,j) = A(i,:)*B(:,j);
                                  end
                                  end

>> C = A*B                        >> C
C =                               C =
    15    15    15                    15    15    15
    26    38    26                    26    38    26
    41    70    39                    41    70    39
```

The matrix `A=magic(3)` belongs to the family of so called *magic squares*. `M = magic(n)` returns an $n \times n$ matrix ($n \geq 3$) constructed from the integers 1 through $n^2$ such that the $n$ numbers in all rows, all columns, and both diagonals sum to the same constant. The constant sum in every row, column and diagonal is called the magic constant or magic sum $M$. The magic constants of a magic square depend only on $n$ and have the values

$$15,\ 34,\ 65,\ 111,\ 175,\ 260,\ \ldots,\ \frac{n(n^2+1)}{2},\ \ldots$$

(See, for example, `http://en.wikipedia.org/wiki/Magic_square`.)

```
>> magic(3)             >> magic(4)                     >> magic(5)
ans =                   ans =                           ans =
     8     1     6          16     2     3    13            17    24     1     8    15
     3     5     7           5    11    10     8            23     5     7    14    16
     4     9     2           9     7     6    12             4     6    13    20    22
                             4    14    15     1            10    12    19    21     3
                                                            11    18    25     2     9
```

The commands below check that the sums in all rows, columns, and in main diagonals of `A=magic(5)` are indeed the same and equal to 65.

```
>> A=magic(5);
>> for i=1:5
rows(i)=sum(A(i,:));
columns(i)=sum(A(:,i));
end

>> rows
rows =
    65    65    65    65    65

>> columns
columns =
    65    65    65    65    65
```

```
>> A=magic(5); dr=0;
for i=1:5
dr=dr+A(i,5-i+1);       % right diagonal
end

>> A=magic(5); dl=0;
for i=1:5
dl=dl+A(i,i);           % left diagonal
end

>> dr
dr =
    65

>> dl
dl =
    65
```

For information on `pascal(n)` matrices see Section 12.1.

The scalar (dot) product of two vectors or multiplication of a matrix with a vector can be done easily in Matlab.

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

>> dot(a,b)  % Scalar (dot) product of a and b
ans =
    70
```

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

>> a*b'  % dot product using matrix product
ans =
    70
```

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

```
>> X=[0;-1;-2]
X =
     0
    -1
    -2
```

```
>> A*X
ans =
    -8
   -17
   -26
```

7.3. **Array Operations.** Array operations are done on an element-by-element basis for matrices/vectors of the same size:

.* denotes element-by-element multiplication;
./ denotes element-by-element *left* division;
.\ denotes element-by-element *right* division;
.^ denotes element-by-element exponentiation.

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

>> x.*y
ans =
     3     8

>> x./y
ans =
    0.3333    0.5000
```

```
>> y=[3 4]
y =
     3     4

>> x.^y
ans =
     1    16

>> x.\y
ans =
     3     2
```

**Comparison of matrix and array operations**

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

```
>> B=[5 6; 7 8]
B =
     5     6
     7     8
```

```
>> A*B
ans =
    19    22
    43    50
```

```
>> A.*B
ans =
     5    12
    21    32
```

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

```
>> B=[5 6; 7 8]
B =
     5     6
     7     8
```

```
>> A/B
ans =
    3.0000   -2.0000
    2.0000   -1.0000
```

```
>> A./B
ans =
    0.2000    0.3333
    0.4286    0.5000
```

```
>> A\B
ans =
    -3    -4
     4     5
```

```
>> A.\B
ans =
    5.0000    3.0000
    2.3333    2.0000
```

```
>> A^2
ans =
     7    10
    15    22
```

```
>> A.^2
ans =
     1     4
     9    16
```

## 7.4. **Matrix Functions.**

`expm(A)` computes the exponential of a square matrix $A$, i.e., $e^A$. This operation should be distinguished from `exp(A)` which computes the element-by-element exponential of $A$;

`logm(A)` computes $\log(A)$ of square matrix $A$. $\log(A)$ is the matrix such that $A = e^{\log(A)}$;

`sqrtm(A)` computes $\sqrt{A}$ of square matrix. $\sqrt{A}$ is the matrix such that $A = (\sqrt{A}) * (\sqrt{A})$.

For more information on these functions see also Section 12.8.6.

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

>> expm(A)
ans =

    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679

>> exp(A)
ans =

    2.7183    2.7183    1.0000
    1.0000    1.0000    7.3891
    1.0000    1.0000    0.3679
```

Notice that the diagonal elements of the two results are equal. This would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

*For additional help type in Matlab:* `help matfun`
or *find* ***expm****,* ***logm****, or* ***sqrtm*** *in:*
http://www.mathworks.com/access/helpdesk/help/techdoc/index.html

## 8. Character Strings

*(This section can be skipped in the first reading of the tutorial.)*

`S = 'Any Characters'` creates a character array, or string. All character strings are entered using single right-quote characters. The string is actually a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The length of `S` is the number of characters. A quotation within the string is indicated by two single right-quotes.

`S = [S1 S2 ...]` concatenates character arrays `S1`, `S2`, etc. into a new character array, `S`.

`S = strcat(S1, S2, ...)` concatenates `S1`, `S2`, etc., which can be character arrays or *Cell Arrays of Strings.*

Trailing spaces in `strcat` character array inputs are ignored and do not appear in the output. This is not true for `strcat` inputs that are cell arrays of strings. Use the `S = [S1 S2 ...]` concatenation syntax, to preserve trailing spaces.

```
>> name = ['Thomas' ' R. ' 'Lee']
name =
Thomas R. Lee

>> S = ['toge' 'ther']
S =
together

>> S = ['toge ' 'ther']
S =
toge ther
```

```
>> name = strcat('Thomas',' R.',' Lee')
name =
Thomas R. Lee

>> S = strcat('toge', 'ther')
S =
together

S = strcat('toge ', 'ther')
S =
together
```

A column vector of text objects is created with `C=[S1; S2; ...]`, where each text string must have the same number of characters.

```
>> S1=['alpha ';'gamma ']
S1 =
alpha
gamma

>> S=[S1 S2]
S =
alpha ALPHA
gamma GAMMA

>> size(S)
ans =
     2    11
```

```
>> S2=['ALPHA'; 'GAMMA']
S2 =
ALPHA
GAMMA

>> S=strcat(S1,S2)
S =
alphaALPHA
gammaGAMMA

>> size(S)
ans =
     2    10
```

The command `char(S1,S2,...)` converts strings to a matrix. It puts each string argument `S1`, `S1`, etc., in a row and creates a string matrix by padding each row with the appropriate number of blanks; in other words, the command `char` does not require the strings `S1`, `S2`, . . . to have the same number of characters. One can also use `strvcat` and `str2mat` to create strings arrays from strings arguments, however, `char` does not ignore empty strings but `strvcat` does.

```
>> S=char('Hello','','World')
S =
Hello

World

>> size(S)
ans =
     3     5

>> S(1,:)
ans =
Hello

>> S(2,:)
ans =


>> S(3,:)
ans =
World
```

```
>> S=strvcat('Hello','','World')
S =
Hello
World


>> size(S)
ans =
     2     5

>> S(1,:)
ans =
Hello

>> S(2,:)
ans =
World

>> S(3,:)
??? Index exceeds matrix dimensions.
```

*For additional help type in Matlab:* `help strings` *or* `help strfun`

or *find* ***`strings`*** *in:*

http://www.mathworks.com/access/helpdesk/help/techdoc/index.html

8.1. `eval` **Function.** `eval` executes string containing Matlab expression. For example, the command

```
>> eval('y=exp(3)*cos(pi/4)')
y =
   14.2026
```

evaluates $y = \exp(3)\cos(\pi/4)$ and is equivalent to typing $y = \exp(3)\cos(\pi/4)$ on the command prompt. `eval` function is often used to load or create sequentially numbered data files. Here is a hypothetical example:

```
>> for d=1:10
   s = ['load August' int2str(d) '.mat']
   eval(s)
end
```

Loads MAT-files August1.mat to August10.mat into the Matlab workspace.

These are the strings being evaluated:

```
s =
   load August1.mat
s =
   load August2.mat
s =
   load August3.mat
   ⋮
```

*For additional help type in Matlab:* `help eval`
or *find* ***eval*** *in:*
http://www.mathworks.com/access/helpdesk/help/techdoc/index.html

## 9. for and while Loops and if Conditionals

`for x=initval:endval, statements end` command repeatedly executes one or more Matlab `statements` in a loop. Loop counter variable `x` is initialized to value `initval` at the start of the first pass through the loop, and automatically increments by 1 each time through the loop. The program makes repeated passes through `statements` until either `x` has incremented to the value `endval`, or Matlab encounters a `break`, or `return` instruction,

`while expression, statements, end` command repeatedly executes one or more Matlab `statements` in a loop, continuing until `expression` no longer holds true or until Matlab encounters a `break`, or `return` instruction.

`if` conditionally executes statements. The general form of the `if` command is

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

The `else` and `elseif` parts are optional.

**Example 1.**

The graph of function $f(x) = 3x^2 - e^x$

indicates that $f(x) = 3x^2 - e^x$ has two positive zeros.

The computation of these two positive roots can be achieved by the so called fixed-point iteration scheme. For example, the smaller positive root $p$ is the fixed point of $g_1(x) = \sqrt{\left(\frac{1}{3} \cdot e^x\right)}$. defined on $[0, 1]$, i.e., such that

$$g_1(p) = p, \qquad 0 \le p \le 1.$$

The larger positive root is the fixed point $c$ of $g_2(x) = \ln(3x^2)$ defined on $[3, 4]$, i.e., such that

$$g_2(c) = c, \qquad 3 \le c \le 4.$$

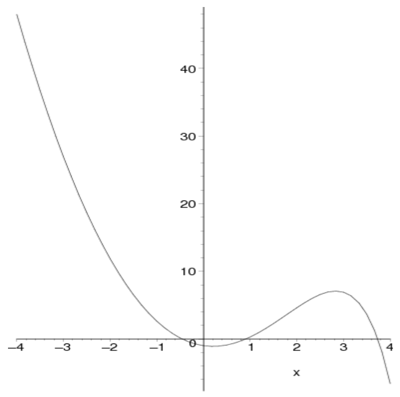


FIGURE 2. The graph of function $f(x) = 3x^2 - e^x$.

These fixed points (and ultimately the positive roots of $f(x) = 3x^2 - e^x$) can be computed accurately by running the so called fixed-point iteration scheme in the form:

$$x_{n+1} = g_1(x_n) \quad \text{or} \quad x_{n+1} = g_2(x_n), \quad n \geq 0, \quad \text{with } x_0 \text{ provided.}$$

And here is the actual Matlab session for the above fixed-point iteration scheme based on $g_1$:

```
>> g1=inline('((1/3)*exp(x))^(1/2)');      Define g_1;
>> p0=1.0; iter=1;                          Define initial p_0 and iteration counter;
>> for n=1:200                              Start for loop;
    p=g1(p0);                               Computation of the next approximation p_{i+1} = g_1(p_i);
    err=abs(p-p0);                          Compute the difference between the two consecutive iterations;
    if err>=0.00001                         If the difference is greater than (0.00001), advance iteration by
        iter=iter+1;                        one step;
        p0=p;                               change p_0 to p, the next approximation computed;
    else
        break                               Terminates the execution when the error is less than 0.00001.
    end
end
>> p
p =
    0.9100                                  The computed value of the root of f(x) = 3x^2 - e^x.
```

and here fixed-point iteration scheme based on $g_2$:

```
>> g2=inline('log(3*x^2)');                 Define g_2;
>> p0=4.0; iter=1;                          Define initial p_0 and iteration counter;
>> for n=1:200                              Start for loop;
    p=g2(p0);                               Computation if the next approximation p_{i+1} = g_1(p_i);
    err=abs(p-p0);                          Compute the difference between the two consecutive iterations;
    if err>=0.00001                         If the difference is greater than (0.00001), advance iteration by
        iter=iter+1;                        one step;
        p0=p;                               change p_0 to p, the next approximation computed;
    else
        break                               Terminates the execution when the error is less than 0.00001.
    end
end
>> p
p =
    3.7331                                  The computed value of the root of f(x) = 3x^2 - e^x.
```

The accurate value of the root of $f(x) = 3x^2 - e^x$ in $[0, 1]$ (to within $10^{-10}$) is $p = 0.9100075725$.

The accurate value of the root of $f(x) = 3x^2 - e^x$ in $[3, 4]$ (to within $10^{-10}$) is $p = 3.733079029$.

**Note**: *It is interesting to know that the fixed-point iteration based on $g_1$ does not converge to the larger positive root of $f(x) = 3x^2 - e^x$. Similarly, the fixed-point iteration based on $g_2$ does not converge to the smaller positive root of $f(x) = 3x^2 - e^x$.*

**Example 2.**

The Euler numerical method for approximating the solution to the differential equation,

$$y' = f(x, y), \quad y(x_0) = y_0,$$

has the form

$$y_{n+1} = y_n + hf(x_n, y_n), \quad \text{where } x_n = x_0 + nh, \text{ and } n = 0, 1, 2, \ldots.$$

In the above scheme, $y_n$, for $n = 1, 2, \ldots$, approximates true solution at $x_n$, i.e., $y(x_n)$.

The initial value problem

$$y' + 2y = 3\exp(-4x), \quad y(0) = 1,$$

has the exact solution

$$y(x) = 2.5\exp(-2x) - 1.5\exp(-4x).$$

We will use the Euler method to approximate this solution. We will also compute the differences between the exact and approximate solutions.

```
>> h=0.1;
>> x=0:h:2;
>> clear y;             % clear possibly existing variable y
>> y(1)=1;              % x1=0, thus y(1) corresponds to y(x1)=1
>> f=inline('3*exp(-4*x)-2*y')
f =
     Inline function:
     f(x,y) = 3*exp(-4*x)-2*y
>> size(x)              % size of x to be used in determining the size of vector i
ans =
     1    21
>> for i=1:20 y(i+1)=y(i)+h*f(x(i),y(i)); end
>> Y_exact=2.5*exp(-2*x)-1.5*exp(-4*x);
>> plot(x,y,'--',x,Y_exact)
```

## 10. Interactive Input

*(This section can be skipped in the first reading of the tutorial.)*

The commands `input`, `keyboard`, `menu`, and `pause` provide interactive user input and can be used inside a script or function file.

```
>> R = input('How many apples: ')
How many apples: sqrt(36)+factorial(4)
R =
     30
```

gives the user the prompt in the text string and then waits for input from the keyboard. The input can be any Matlab expression, which is evaluated, using the variables in the current workspace, and the result returned in R. If the user presses the return key without entering anything, `input` returns an empty matrix.

```
>> R = input('How many apples: ','s')
How many apples: sqrt(36)+factorial(4)
R =
sqrt(36)+factorial(4)
```

gives the prompt in the text string and waits for character string input. The typed input is not evaluated; the characters are simply returned as a Matlab string.

`keyboard` command, when placed in a script or function file, stops execution of the file and gives control to the user's keyboard. The special status is indicated by a K appearing before the prompt. Variables may be examined or changed - all Matlab commands are valid. The keyboard mode is terminated by executing the command `return` and pressing the return key. Control returns to the invoking script/function file.

```
>> error = abs(Y_exact-y)'
error =
         0
    0.0587
    0.0793
    0.0794
    0.0696
    0.0560
    0.0418
    0.0289
    0.0180
    0.0092
    0.0025
    0.0024
    0.0058
    0.0080
    0.0093
    0.0099
    0.0100
    0.0097
    0.0092
    0.0086
    0.0079
```

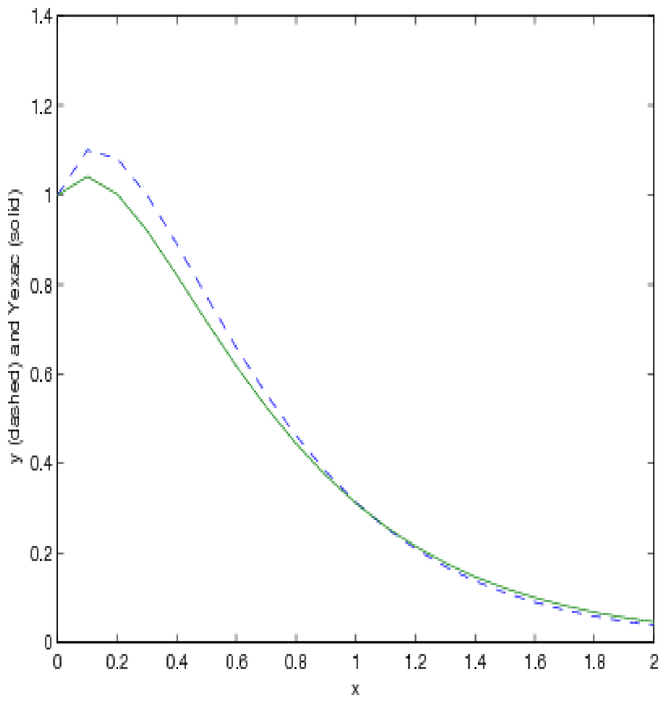


Figure 3. Plots of the exact solution (solid line) and the approximate solution (dashed line) on the interval $[0, 2]$.

Command `menu` generates a menu of choices for user input. On most graphics terminals `menu` will display the menu-items as push buttons in a figure window, otherwise they will be given as a numbered list in the command window. For example,

```
choice = menu('Choose a color','Red','Blue','Green')
```

creates a figure with buttons labeled `Red`, `Blue` and `Green`. The button clicked by the user is returned as `choice` (i.e. `choice = 2` implies that the user selected Blue).

After input is accepted, the dialog box closes, returning the output in `choice`. You can use `choice` to control the color of a graph:

```
t = 0:.1:60;
s = sin(t);
color = ['r','g','b']
plot(t,s,color(choice))
```

`pause` commands waits for user response. `pause(n)` pauses for n seconds before continuing.

`pause` causes a procedure to stop and wait for the user to strike any key before continuing.

`pause off` indicates that any subsequent `pause` or `pause(n)` commands should not actually pause. This allows normally interactive scripts to run unattended.

`pause on` (default setting) indicates that subsequent `pause` commands should pause.

`pause query` returns the current state of pause, either `on` or `off`.

## 11. Input/Output Commands

*(This section can be skipped in the first reading of the tutorial.)*

Matlab provides many C-language file I/O functions for reading and writing binary and text files. The following five I/O functions: `fopen`, `fclose`, `fgetl`, `fscanf`, and `fprintf` are frequently used.

`fid = fopen(filename)` opens or creates the `filename`. The filename argument is a string enclosed in single quotation marks.
Output value `fid` is a scalar Matlab integer that you use as a file identifier for all subsequent low-level file input/output routines.

`fclose(fid)` closes the specified file if it is open, returning 0 if successful and -1 if unsuccessful. `fid` is an integer file identifier associated with an open file.

`fclose('all')` closes all open files.

`tline = fgetl(fid)` returns the next line of the file associated with the file identifier `fid`. The returned string `tline` does not include the newline characters with the text line. To obtain the newline characters, use `fgets` function.

`A = fscanf(fid, format)` reads data from the text file specified by `fid`, converts it according to the specified `format` string, and returns it in matrix `A`. Argument `fid` is an integer file identifier obtained from `fopen`. `format` is a string specifying the format of the data to be read.

`count = fprintf(fid, format, A, ...)` formats the data in the real part of matrix A (and in any additional matrix arguments) under control of the specified `format` string, and writes it to the file associated with file identifier `fid`. `count` returns a count of the number of bytes written.
Argument `fid` is an integer file identifier obtained from `fopen`. It can also be 1 for standard output (the screen) or 2 for standard error. Omitting fid causes output to appear on the screen. For example, `fprintf('A unit circle has circumference %g radians.\n', 2*pi)` displays a line on the screen:
`A unit circle has circumference 6.283186 radians.`

The example below reads and displays the first three lines from the file `circle_f2.m` (see, Section 5) and available from: `http://www.csun.edu/~hcmth008/matlab_tutorial/circle_f2.m`
It also tests for end-of-file and then closes `circle_f2.m` file.

```
>> fid = fopen('circle_f2.m');
tline = fgetl(fid); i=1;
while feof(fid)==0 & i<=5     % Alternatively, you can use ischar(tline)
                              % instead of feof(fid)==0;
                              % == and <= are relational operators
                              % EQUAL and LESS THAN OR EQUAL,
                              % respectively;
                              % & denotes the logical operator AND.
disp(tline)                   % disp(X) displays an array, without
                              % printing the array name. If X contains
                              % a text string, the string is
                              % displayed.
tline = fgetl(fid);
i=i+1;
end
is_end_of_file = feof(fid)    % feof(fid) command checks for end-of-file.
fclose(fid);                  % Closes circle_f2.m file.
                              % The result is displayed below:

function [A,x,y] = circle_f2(r);
% CIRCLE_F2 - Function draws circle x^2+r^2=r^2
% and output its area together with x and y coordinates.

is_end_of_file =
     0
```

The value of variable `is_end_of_file = 0` indicates that the end of `circle_f2.m` file was not reached. Changing name of the file and replacing `i<=3` by `i<=n` will make the above script read and display the first `n` lines. The entire file will be displayed if the number of lines in the file is less or equal to `n`.

The next script reads and displays every line from the file `my_f.m` (created in Section 5) and available from:
http://www.csun.edu/~hcmth008/matlab_tutorial/my_f.m
It also tests for end-of-file and then closes the file `my_f.m`. No a priori information about the number of lines in the file is needed in the script.

```
>> fid = fopen('my_f.m');
tline = fgetl(fid);
while feof(fid)==0         % Alternatively, you can use ischar(tline) instead of feof(fid)==0.
disp(tline)
tline = fgetl(fid);
end
is_end_of_file = feof(fid) % Checks for end-of-file.
fclose(fid);               % Closes my_f.m file.
                           % The result is displayed below:

function f = my_f(x);
% MY_F evaluates function f = exp(0.3x)sin(30x)+x^10/(1+x^10)
f = exp(-0.3*x).*sin(30*x)+x.^10./(1+x.^10);

is_end_of_file =
     1
```

The value of variable `is_end_of_file = 1` indicates that the end of `my_f.m` file was reached.

The next example is more advanced. It shows how to read and display lines from given file in a non-sequential order. It uses `eval` and `int2str` commands for a customized display. The script below reads and displays the 4th, 5th, 8th, and 9th lines from `circle_f2.m` file. It also appends the numbers to the above lines. No a priori information about the number of lines in the file is needed in the script.

```
>> fid = fopen('circle_f2.m'); % output -->  <4> alpha = linspace(0,2*pi,100);
tline = fgetl(fid); i=1;       % output -->  <5> x = r*cos(alpha);
while feof(fid)==0             % output -->  <8> plot(x,y);
if i<=3                        % output -->  <9> axis('equal');
tline = fgetl(fid);
end
if 4<=i & i<=5
disp(['<' eval('int2str(i)') '>' ' ' tline])
tline = fgetl(fid);
end
if i>=6 & i<=7
tline = fgetl(fid);
end
if 8<=i & i<=9
disp(['<' eval('int2str(i)') '>' ' ' tline])
tline = fgetl(fid);
end
if i>=10
tline = fgetl(fid);
end
i=i+1;
end
fclose(fid);
```

The next examples show how to use `fprintf` and `fscanf` commands. First, create a text file called `exp.txt` containing a short table of the exponential function. (On Windows platforms, it is recommended that you use `fopen` with the mode set to `'wt'` to open a text file for writing.)

```
x = 0:.1:1;
y = [x; exp(x)];
fid = fopen('exp.txt', 'wt');
fprintf(fid, '%6.2f %12.8f\n', y);
fclose(fid)
```

%6.2f stands for fixed-point notation field of width 8 with two digits after the decimal point, while %12.8f stands for fixed-point notation field of width 12 with 8 digits after the decimal point. \n stands for new line.

Now, examine the contents of `exp.txt` with `type filename` command.

```
type exp.txt

  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183
```

One can now read this file back into a two-column Matlab matrix with the script:

```
fid = fopen('exp.txt', 'r');
a = fscanf(fid, '%f %f', [2 inf]);
a = a';
fclose(fid);
```

It has two rows now. `inf` indicates to read to the end of the file.

Comparing it with `exp.txt` and long format outputs we observe no loss of accuracy.

```
>> disp(a)              >> type exp.txt        >> format long; disp(a)
         0    1.0000      0.00   1.00000000                       0   1.000000000000000
    0.1000    1.1052      0.10   1.10517092     0.100000000000000   1.105170920000000
    0.2000    1.2214      0.20   1.22140276     0.200000000000000   1.221402760000000
    0.3000    1.3499      0.30   1.34985881     0.300000000000000   1.349858810000000
    0.4000    1.4918      0.40   1.49182470     0.400000000000000   1.491824700000000
    0.5000    1.6487      0.50   1.64872127     0.500000000000000   1.648721270000000
    0.6000    1.8221      0.60   1.82211880     0.600000000000000   1.822118800000000
    0.7000    2.0138      0.70   2.01375271     0.700000000000000   2.013752710000000
    0.8000    2.2255      0.80   2.22554093     0.800000000000000   2.225540930000000
    0.9000    2.4596      0.90   2.45960311     0.900000000000000   2.459603110000000
    1.0000    2.7183      1.00   2.71828183     1.000000000000000   2.718281830000000
```

As the final example, start with a file `temp.dat` that contains temperature readings

$$78^\circ\mathrm{F} \quad 72^\circ\mathrm{F} \quad 65^\circ\mathrm{F} \quad 105^\circ\mathrm{F} \quad -10^\circ\mathrm{F}$$

File `temp.dat` is available from: http://www.csun.edu/~hcmth008/matlab_tutorial/temp.dat

Open the file using `fopen` and read it with `fscanf`. If you include ordinary characters (such as the degrees character ° (ASCII 176) and Fahrenheit (F) symbols used here) in the conversion string, `fscanf` skips over those characters when reading the string. This allows for further processing (see below). The display on the right is given for a comparison.

```
>> fid = fopen('temp.dat', 'r');
degrees = char(176);
a=fscanf(fid, ['%d' degrees 'F']);
fclose(fid);
disp(a')
78    72    65   105   -10

>> sin(a)
ans =
    0.5140
    0.2538
    0.8268
   -0.9705
    0.5440
```

```
>> fid = fopen('temp.dat', 'r');
a=fscanf(fid, '%c');
%
fclose(fid);
disp(a)
78°F  72°F  65°F  105°F  – 10°F

>> sin(a)
??? Undefined function or method 'sin' for input
arguments of type 'char'.
```

## 12. Linear Algebra

### 12.1. **Solving Linear Square Systems.**

Write the system of linear algebraic equations

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= 1 \\ x_1 + 2x_2 + 3x_3 + 4x_4 &= -1 \\ x_1 + 3x_2 + 6x_3 + 10x_4 &= 2 \\ x_1 + 4x_2 + 10x_3 + 20x_4 &= 3 \end{aligned}$$

in matrix form

$$\mathbf{AX} = \mathbf{b},$$

with

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ -1 \\ 2 \\ 3 \end{bmatrix}.$$

Please note that the number of columns of **A** equals to the number of rows of **X**, and thus the matrix multiplication of **A** with **X** is defined.

Enter the matrix **A** and vector **b**, and solve for vector **X** with `A\b`.

```
>> A = [1 1 1 1; 1 2 3 4; 1 3 6 10; 1 4 10 20]
A =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20

>> X = A\b   % Solve for X
X =
    15
   -33
    26
    -7
```

```
>> b = [1; -1; 2; 3]
b =
     1
    -1
     2
     3

>> c = A*X   % Check the solution
c =
     1
    -1
     2
     3
```

The matrix **A** belongs to the family of so called Pascal matrices and can be entered by the command `A = pascal(4)`.

```
>> pascal(1)        >> pascal(2)        >> pascal(3)                   >> pascal(4)
ans =               ans =               ans =                          ans =
     1                   1     1             1     1     1                  1     1     1     1
                         1     2             1     2     3                  1     2     3     4
                                             1     3     6                  1     3     6    10
                                                                            1     4    10    20
```

In general, `A = pascal(n)` returns the Pascal matrix of order n: a symmetric positive definite matrix with integer entries taken from Pascal's triangle that determines the coefficients which arise in binomial expansions. See, for example, http://en.wikipedia.org/wiki/Pascal's_triangle.

12.1.1. *Singular Coefficient Matrix.* A square matrix **A** is *singular* if it does not have linearly independent columns. **A** is singular if and only if the determinant of **A** is zero (see Section 12.3). If **A** is singular, the equation $\mathbf{AX} = \mathbf{b}$ either has no solution or it has infinite many solutions. Actually, one can show that $\mathbf{AX} = \mathbf{b}$ system has a solution if and only if $\mathbf{b} \perp \mathbf{N}$ (i.e, the scalar product of **b** and **N** is zero) for all **N** such that $\mathbf{A}'\mathbf{N} = \mathbf{0}$, where $\mathbf{A}'$ is the *transpose* (the *complex conjugate transpose*, if **A** has complex entries) of **A**, i.e., the matrix where the original rows and columns are interchanged. The matrix

```
>> A=[1 3 7; -1 4 4; 1 10 18]
A =
     1     3     7
    -1  \label{sub:factors}\label{sub:factors}   4     4
     1    10    18
```

is singular, as easy verified by typing

```
>> det(A)       % det(A) calculates the determinant of A
ans =
     0
```

For `b = [5;2;12]`, the equation $\mathbf{AX} = \mathbf{b}$ has a solution given by

```
>> b = [5;2;12]; format rat; X=pinv(A)*b, format;          >> A*X

X =          % "format rat" requests a rational format,    ans =    % Solution's verification
    82/213   % while the last command "format" returns         5.0000
   -47/426   % to the default format.                          2.0000
    301/426                                                   12.0000
```

For more information on `pinv(A)`, see Section 12.4. In this case, command `A\b` returns an error condition.

One can also verify that `b = [5;2;12]` is perpendicular to any solution of the homogeneous system $\mathbf{A}'\mathbf{N} = \mathbf{0}$:

```
>> A'   % transpose of A          >> N = null(A','r')

ans =                             N =                 % Solves the homogeneous equation A'*N=0
     1    -1     1                   -2               % with an option 'r' requesting
     3     4    10                   -1               % a rational format
     7     4    18                    1
```

and

```
>> dot(N,b)
ans =
    0
```

This verifies that $\mathbf{b} \perp \mathbf{N}$.

The general (complete) solution to $\mathbf{AY} = \mathbf{b}$ is given by

$$\mathbf{Y} = c\mathbf{Z} + \mathbf{X}, \quad \text{with} \quad \mathbf{Z} = \begin{bmatrix} -16/7 \\ -11/7 \\ 1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{X} = \begin{bmatrix} 82/213 \\ -47/426 \\ 301/426 \end{bmatrix},$$

where **Z** solves the homogeneous system $\mathbf{AZ} = \mathbf{0}$ and $c$ is any constant (scalar), as verified by

```
>> format rat; Z = null(A,'r'), format;                          >> c = 1.5; A*(c*Z+X)

Z =                    % Solves the homogeneous equation         ans =
   16/7                % A*Z=0 with an option 'r'                    5.0000
  -11/7                % requesting a rational format.               2.0000
    1                                                               12.0000
```

On the other hand, for `b = [1;2;3]`, $\mathbf{b} \not\perp \mathbf{N}$. (indeed, `dot(N,b)=-0.4082`) and the system $\mathbf{AX} = \mathbf{b}$ does not have an exact solution.

In this case, `pinv(A)*b` returns a least squares solution $\mathbf{X}$, i.e.,

$$\mathbf{X} = \min_{\mathbf{Y}} \|\mathbf{AY} - \mathbf{b}\|,$$

where for $\mathbf{v} = (v_1, v_2, \dots, v_n)$, $\|\mathbf{v}\| = \left(\sum_{i=1}^{n} |v_i|^2\right)^{1/2}$ is the so called norm of vector $\mathbf{v}$. Matlab command for the norm of vector `v` is `norm(v)`. We have

```
>> b = [1;2;3]                        >> format rat; X=pinv(A)*b, format;

b =                                   X =                    % a least squares solution
     1                                   -395/1278
     2                                    1081/2556
     3                                   -107/2556
```

```
>> format rat; A*X, format;                                      >> norm(A*X-b)

ans =          % It does not get back the original vector        ans =
     2/3       % b=[1;2;3].                                          0.4082
    11/6
    19/6
```

12.2. **Gaussian Elimination.** One of the techniques learned in linear algebra courses for solving a system of linear equations is *Gaussian elimination.* One forms the augmented matrix that contains both the coefficient matrix $\mathbf{A}$ and vector $\mathbf{b}$, and then, using *Gauss-Jordan reduction* procedure, the augmented matrix is transformed into the so called *row reduced echelon form.* Matlab has built-in function that does precisely this reduction. For `A = pascal(4)` and `b = [1; -1; 2; 3]`, considered in Section 12.1, the commands `C=[A b]` and `rref(C)` calculate the augmented matrix and the row reduced echelon matrix, respectively.

```
>> A = pascal(4)                   >> b = [1; -1; 2; 3]          >> C = [A b]; rref(C)
A =                                b =                           ans =
     1     1     1     1                1                             1     0     0     0    15
     1     2     3     4               -1                             0     1     0     0   -33
     1     3     6    10                2                             0     0     1     0    26
     1     4    10    20                3                             0     0     0     1    -7
```

Since matrix `A=pascal(4)` is not singular, the last column of `rref(C)` (One could also use `rref([A b])` command here.) provides the solution `X` already obtained in Section 12.1 by using `X = A\b` command.

Here is another use of `rref` command. One can determine whether $\mathbf{AX} = \mathbf{b}$ has an exact solution by finding the row reduced echelon form of the augmented matrix `[A b]`. For example, enter

```
format rat; A = [1 3 7; -1 4 4; 1 10 18]; b = [1;2;3]; rref([A b]), format;

ans =                                      % For better visibility, the command
       1      0      16/7    0             % "format rat" displays the result in
                                           % rational format, while the last
       0      1      11/7    0             % command  "format" returns the
                                           % default format.
       0      0      0       1
```

Since the bottom row contains all zeros except for the last entry, the equation $\mathbf{AX} = \mathbf{b}$ does not have a solution. Indeed, the above row reduced echelon form is equivalent to the following system of equations:

$$\begin{aligned} 1\cdot x_1 + 0\cdot x_2 + (16/7)x_3 &= 0\\ 0\cdot x_1 + 1\cdot x_2 + (11/7)x_3 &= 0\\ 0\cdot x_1 + 0\cdot x_2 + 0\cdot x_3 &= 1 \end{aligned}$$

As indicated in Section 12.4, in this case, `pinv(A)*b` command returns a least squares solution.

12.3. **Inverses, Determinants, and Cramer's Rule.** For a square and nonsingular matrix $\mathbf{A}$, the equations $\mathbf{AX} = \mathbf{I}$ and $\mathbf{XA} = \mathbf{I}$ (here $\mathbf{I}$ is an identity matrix), have the same solution, $\mathbf{X}$. This solution is called the *inverse* of $\mathbf{A}$ and is denoted by $\mathbf{A}^{-1}$. Matlab function `inv(A)` computes the inverse of $\mathbf{A}$.

The inverse of `A=pascal(4)` is

```
>> A = pascal(4)                               >> format rat; inv(A)
A =                                            ans =
     1     1     1     1                             4    -6     4    -1
     1     2     3     4                            -6    14   -11     3
     1     3     6    10                             4    11    10    -3
     1     4    10    20                            -1     3    -3     1
```

and thus, the solution of the system $\mathbf{AX} = \mathbf{b}$ with `b=[1;0;2;0]` is

```
>> b = [1;0;2;0]; format rat; X = inv(A)*b    >> Y=A\b    % the backslash operator is another
                                                          % way to compute the same solution
                                                          % and is much more efficient for
                                                          % large size matrices
X =                                           Y =
    12                                            12
   -28                                           -28
    24                                            24
    -7                                            -7
```

The determinant is a special number associated with any square matrix $\mathbf{A}$ and is denoted by $|\mathbf{A}|$ or $\det(A)$. The meaning of a determinant is a scale factor for measure when the matrix is regarded as a linear transformation. Thus a $2\times 2$ matrix, with determinant 3, when applied to a set of points with finite area will transform those points into a set with 3 times the area.

The determinant of

$$\mathbf{A} = \begin{bmatrix} a & b & c\\ d & e & f\\ g & h & i\end{bmatrix} \quad \text{is} \quad \det(A) = aei - afh + bfg - bdi + cdh - ceg$$

and the *Rule of Sarrus* is a mnemonic for this formula:

The determinant of $n\times n$ matrix $\mathbf{A} = (a_{ij})_{i,j=1,2,\dots n}$ $(n \ge 1)$ can be defined by the Leibniz formula:

$$\det(A) = \sum_{\sigma\in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} a_{i\,\sigma(i)}\,.$$

The matrix $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ has the determinant $\det(\mathbf{A}) = ad - bc$ and the geometric interpretation presented in Figure 4.

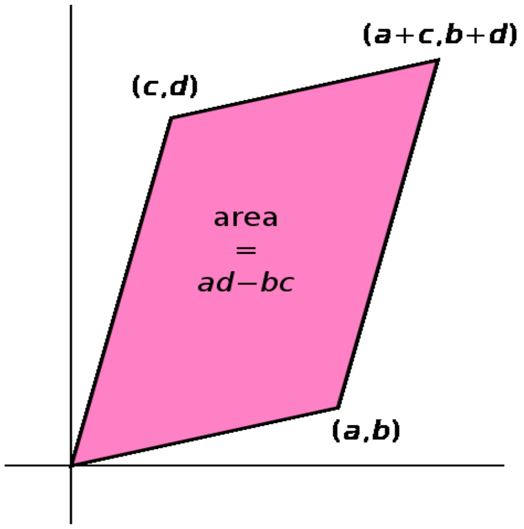


FIGURE 4. The area of the parallelogram is the absolute value of the determinant of the matrix formed by the vectors representing the parallelogram's sides.

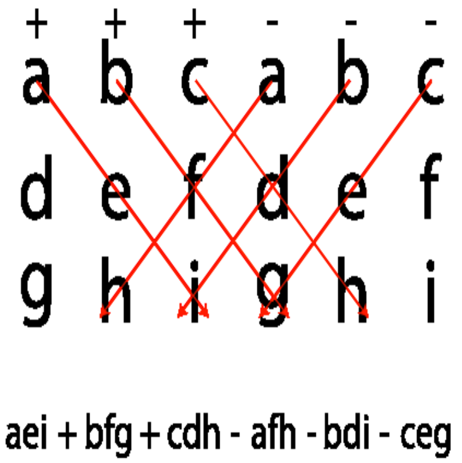


FIGURE 5. The determinant of a 3 × 3 matrix can be calculated by its diagonals.

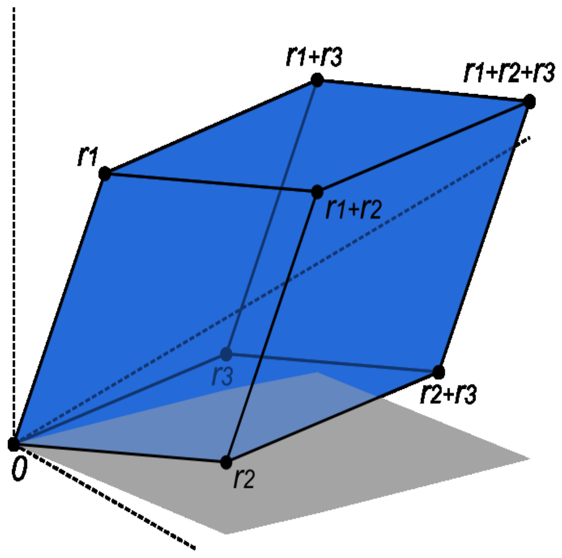


FIGURE 6. The volume of this parallelepiped is the absolute value of the determinant of the matrix formed by the rows r1, r2, and r3.

Here the sum is computed over all permutations $\sigma$ of the numbers $\{1, 2, \ldots, n\}$. A permutation is a function that reorders this set of integers. The set of such permutations is denoted by $S_n$. The sign function of permutations in the permutation group $S_n$ returns $+1$ and $-1$ for even and odd permutations, respectively.

For example, for $n = 4$ and $\sigma = (1, 4, 3, 2)$, $\operatorname{sign}(\sigma) = -1$ (one pair switch) and $\prod_{i=1}^{4} a_{i\,\sigma(i)} = a_{11}a_{24}a_{33}a_{42}$.

Furthermore, for $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $S_2 = \{\sigma_1, \sigma_2\}$, with $\sigma_1 = (1,2)$, $\sigma_2 = (2,1)$, $\text{sign}(\sigma_1) = +1$, $\text{sign}(\sigma_2) = -1$, and we have

$$\prod_{i=1}^{2} a_{i\,\sigma_1(i)} = a_{11}a_{22}, \quad \prod_{i=1}^{2} a_{i\,\sigma_2(i)} = a_{12}a_{21}, \quad \text{and thus,} \quad \det(A) = a_{11}a_{22} - a_{12}a_{21}.$$

Matlab function `det(A)` computes the determinant of a square matrix `A`. For `A = pascal(4)`, we have

```
>> A = pascal(4)                                  >> det(A)
A =                                               ans =
     1     1     1     1                                1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

Actually, `det(pascal(n))` is equal to 1 for any $n \geq 1$.

A square matrix $\mathbf{A}$ has the inverse $\mathbf{A}^{-1}$ if and only if $\det(\mathbf{A}) \neq 0$. In this case, $\det(\mathbf{A}^{-1}) = 1/\det(\mathbf{A})$.

12.3.1. *Cramer's rule.* For square matrices $\mathbf{A}$ that are invertible, the system of equations $\mathbf{AX} = \mathbf{b}$ with a given $n$-dimensional column vector $\mathbf{b}$ can be solved by using so called Cramer's rule. This rule states that the solution $\mathbf{X} = (x_1, x_2, \ldots, x_n)^T$ is obtained by the formula:

$$x_i = \frac{\det(\mathbf{A}_i)}{\det(\mathbf{A})}, \quad i = 1, 2, \ldots, n,$$

where $\mathbf{A}_i$ is the matrix formed by replacing the ith column of $\mathbf{A}$ by the column vector $\mathbf{b}$.

Here is Matlab code for Cramer's rule applied to `A=magic(5)` and `b=[1;2;3;4;5]`

```
>> A = magic(5);                                                  >> format rat; X
>> b = [1;2;3;4;5];                                               X =
>> format rat;                                                           1/78
>> for i=1:5                               % only these three            1/78
C=A; C(:,i)=b; X(i,:)=det(C)/det(A);       % lines represent             7/39
   end                                     % the actual code             1/78
                                                                         1/78
```

Check that the backslash operator `A\b` and the command `inv(A)*b` provide the same solution.

**Note:** *For large size matrices, the use of Cramer's rule and of the inverse is not computationally efficient as compared to the backslash operator.*

## 12.4. **Pseudoinverses or Generalized Inverses.**

*(This section can be skipped in the first reading of the tutorial.)*

The pseudoinverse $\mathbf{A}^+$ f an $m \times n$ matrix $\mathbf{A}$ is a generalization of the inverse matrix. The terms Moore-Penrose pseudoinverse or generalized inverse are also used as synonyms for pseudoinverse. Pseudoinverse is a unique $n \times m$ matrix $\mathbf{A}^+$ satisfying all of the following four conditions:

(1) $\mathbf{AA}^+\mathbf{A} = \mathbf{A}$;
(2) $\mathbf{A}^+\mathbf{AA}^+ = \mathbf{A}^+$;
(3) $(\mathbf{AA}^+)' = \mathbf{AA}^+$ (i.e.,$\mathbf{AA}^+$ is *Hermitian*);
(4) $(\mathbf{A}^+\mathbf{A})' = \mathbf{A}^+\mathbf{A}$ (i.e.,$\mathbf{A}^+\mathbf{A}$ is *Hermitian*).

Here $\mathbf{A}'$ is the Hermitian transpose (also called conjugate transpose) of a matrix $\mathbf{A}$. For matrices whose elements are real numbers instead of complex numbers, $\mathbf{A}' = \mathbf{A}^T$, In Matlab notation, Hermitian transpose of `A` is `A'`.

Pseudoinverses satisfy many properties. Here are some of them:

- The pseudoinverse exists and is unique for any matrix $\mathbf{A}$.
- If $\mathbf{A}$ has real entries, then so does $\mathbf{A}^+$; if $\mathbf{A}$ has rational entries, then so does $\mathbf{A}^+$.
- If the matrix $\mathbf{A}$ is invertible, the pseudoinverse and the inverse coincide, i.e., $\mathbf{A}^+ = \mathbf{A}^{-1}$.
- The pseudoinverse of a zero matrix is its transpose.
- The pseudoinverse of the pseudoinverse is the original matrix, i.e., $(\mathbf{A}^+)^+ = \mathbf{A}$.
- Pseudoinversion commutes with transposition, conjugation, and taking the conjugate transpose.
- The pseudoinverse of a scalar multiple of $\mathbf{A}$ is the reciprocal multiple of $\mathbf{A}^+$, i.e., $(\alpha\mathbf{A})^+ = \alpha^{-1}\mathbf{A}^+$ for $\alpha \neq 0$.

For more information go to the webpage: http://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse

A common use of the pseudoinverse is to compute a *best fit* (least squares) solution to a system of linear equations that lacks a unique solution. In general, a given linear system $\mathbf{AX} = \mathbf{b}$ may not have a solution. Even if there exists such a solution, it may not be unique. One can however ask for a vector $\mathbf{X}$ that brings $\mathbf{AX}$ *as close as possible* to vector $\mathbf{b}$, i.e., a vector that minimizes the Euclidean norm

$$\|\mathbf{AX} - \mathbf{b}\|,$$

where for $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, $\|\mathbf{v}\| = \left(\sum_{i=1}^{n} |v_i|^2\right)^{1/2}$. Matlab command for the norm of vector `v` is `norm(v)`.

If there are several such vectors $\mathbf{X}$, one could ask for the one among them with the smallest Euclidean norm. Thus formulated, the problem has a unique solution, given by the pseudoinverse:

$$\mathbf{X} = \mathbf{A}^+\mathbf{b}.$$

Matlab function that computes the pseudoinverse of a matrix `A` is `pinv(A)`. If for given `A` and `b`, the *least square* solution `X` is unique, it can be computed by entering either `X=pinv(A)*b` or `X=A\b`. If there exist many *least square* solutions (for the so called *rank-deficient* matrices), `X=pinv(A)*b` computes vector `X` with the smallest norm `norm(X)`, while `X=A\b` computes a vector `X` that has at most $r$ nonzero components. Here, $r$ is the rank of matrix `A` equal to `rank(A)`.

Sections 12.5.1 and 12.5.2 provide many examples for the use of `pinv(A)`.

## 12.5. **Underdetermined and Overdetermined Linear Systems.**

### 12.5.1. *Underdetermined linear systems.*

Underdetermined linear systems involve more unknowns than equations. Such systems are studied in *linear programming*, especially when accompanied by additional constraints. A solution, when exists, is never unique.

For `A=[0 1 1 0; 1 1 0 1; 1 2 1 1]` and `b=[1;2;1]`, the corresponding system has three equations and four unknowns:

$$\begin{aligned} x_2 + x_3 &= 1 \\ x_1 + x_2 + x_4 &= 2 \\ x_1 + 2x_2 + x_3 + x_4 &= 1 \end{aligned}$$

```
>> A=[0 1 1 0; 1 1 0 1; 1 2 1 1]; b=[1;2;1]; reduced_echelon_form =rref([A b])

reduced_echelon_form =
     1     0    -1     1     0
     0     1     1     0     0
     0     0     0     0     1
```

The row reduced echelon form shows that the system has no exact solutions. In this case, least square solutions $\mathbf{X}$ such that

$$\mathbf{X} = \min_{\mathbf{Z}} \|\mathbf{AZ} - \mathbf{b}\|, \quad \text{where for} \quad \mathbf{v} = (v_1, v_2, \ldots, v_n), \quad \|\mathbf{v}\| = \left(\sum_{i=1}^{n} |v_i|^2\right)^{1/2},$$

can be obtained by entering `A\b` or `pinv(A)*b` (see Section 12.4):

```
>> format rat; X=A\b

Warning: Rank deficient, rank=2,  tol=2.1756e-15.
X =
       1
       1/3
       0
       0
```

```
>> format rat; Y=pinv(A)*b

Y =
       7/15
       2/5
      -1/15
       7/15
```

We observe that there are many vectors `Z` (here, `X` and `Y`) that minimize `norm(A*Z-b)`. This is due to the fact that matrix $\mathbf{A}$ has only 2 linearly independent columns (so called *rank-deficient system*, while the full rank would be 3), the least square problem is not unique.

Consider now the following two equations with four unknowns:

$$6x_1 + 8x_2 + 7x_3 + 3x_4 = 1$$
$$3x_1 + 5x_2 + 4x_3 + x_4 = 2$$

Enter `A=[6 8 7 3;3 5 4 1]` and `b=[1;2]`, and calculate `rref([A b])`:

```
>> A=[6 8 7 3;3 5 4 1]; b=[1;2]; format rat; rref([A b])

ans =

    1    0   1/2   7/6  -11/6
    0    1   1/2  -1/2    3/2
```

Thus, the system has an exact solution that can be obtained by entering `X=A\b` or `W=pinv(A)*b`:

```
>> format rat; X=A\b

X =
       0
       5/7
       0
     -11/7
```

```
>> format rat; W=pinv(A)*b

W =
     -81/137
     119/137
      19/137
    -154/137
```

Both `X` and `W` are exact solutions:

```
>> A*X

ans =
       1
       2
```

```
>> A*W

ans =
       1
       2
```

The general (complete) solution to $\mathbf{AY} = \mathbf{b}$ is given by

$$\mathbf{Y} = c_1\mathbf{Z_1} + c_2\mathbf{Z_2} + \mathbf{X}, \quad \text{with} \quad \mathbf{Z_1} = \begin{bmatrix} -1/2 \\ -1/2 \\ 1 \\ 0 \end{bmatrix}, \qquad \mathbf{Z_2} = \begin{bmatrix} -7/6 \\ 1/2 \\ 0 \\ 1 \end{bmatrix}, \qquad \text{and} \qquad \mathbf{X} = \begin{bmatrix} 0 \\ 5/7 \\ 0 \\ -11/7 \end{bmatrix},$$

where $c_1$ and $c_2$ are any constants (scalars), and $\mathbf{Z_1}$ and $\mathbf{Z_2}$ are fundamental solutions of the system $\mathbf{AZ} = \mathbf{0}$, as verified by

```
>>D = null(A,'r')                                >> Z1 = D(:,1); A*Z1

D =                                              ans =
      -1/2           -7/6                               0
      -1/2            1/2                               0
       1              0
       0              1                          >> Z2 = D(:,2); A*Z2

                                                 ans =
                                                        0
                                                        0
```

The other solution, `W=pinv(A)*b`, is also of the form $c_1\mathbf{Z_1} + c_2\mathbf{Z_2} + \mathbf{X}$, for some constants $c_1$ and $c_2$. Indeed, the constants $c_1$ and $c_2$ are the solutions of the following linear system `D*C=W-X` with `C=[c1;c2]`, as verified by

```
>> D=null(A,'r')                                 >> format rat; C=D\(W-X)

D =                                              C =
      -1/2           -7/6                              19/137         %c1 = 19/137
      -1/2            1/2                             429/959         %c2 = 429/959
       1              0
       0              1
```

12.5.2. *Overdetermined linear systems.* Overdetermined linear systems involve more equations than unknowns. For example, they are encountered in various kinds of curve fitting to experimental data (see Section 13).

Enter the following data into Matlab:

```
t=[0;0.25;0.5;;0.75;1.00;1.25;1.50;1.75;2.00;2.25;2.50];
y=sin(t)+rand(size(t))/10;
```

We will model the above randomly corrupted sine function data with a quadratic polynomial function.

$$y(t) = c_1 + c_2 t + c_3 t^2$$

In other words, we want to approximate the vector `y` by a linear combination of three other vectors:

$$\begin{bmatrix} y(0.00) \\ y(0.25) \\ y(0.50) \\ y(0.75) \\ y(1.00) \\ y(1.25) \\ y(1.50) \\ y(1.75) \\ y(2.00) \\ y(2.25) \\ y(2.50) \end{bmatrix} = c_1 \begin{bmatrix} 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \end{bmatrix} + c_2 \begin{bmatrix} 0.00 \\ 0.25 \\ 0.50 \\ 0.75 \\ 1.00 \\ 1.25 \\ 1.50 \\ 1.75 \\ 2.00 \\ 2.25 \\ 2.50 \end{bmatrix} + c_3 \begin{bmatrix} (0.00)^2 \\ (0.25)^2 \\ (0.50)^2 \\ (0.75)^2 \\ (1.00)^2 \\ (1.25)^2 \\ (1.50)^2 \\ (1.75)^2 \\ (2.00)^2 \\ (2.25)^2 \\ (2.50)^2 \end{bmatrix}.$$

The above system has eleven equations and three unknowns: $c_1$, $c_2$, and $c_3$. It is represented 11-by-3 matrix `A` and `b=y`. The row reduced echelon form of `A` shows the above system does not have an exact solution:

```
>> A = [ones(size(t)) t  t.^2]       >> b=y            >> rref([A b])
A =                                  b =               ans =
    1.0000         0         0           0.0863             1     0     0     0
    1.0000    0.2500    0.0625           0.4295             0     1     0     0
    1.0000    0.5000    0.2500           0.5158             0     0     1     0
    1.0000    0.7500    0.5625           0.7344             0     0     0     1
    1.0000    1.0000    1.0000           0.8706             0     0     0     0
    1.0000    1.2500    1.5625           0.9762             0     0     0     0
    1.0000    1.5000    2.2500           1.1714             0     0     0     0
    1.0000    1.7500    3.0625           1.0999             0     0     0     0
    1.0000    2.0000    4.0000           1.0193             0     0     0     0
    1.0000    2.2500    5.0625           0.8071             0     0     0     0
    1.0000    2.5000    6.2500           0.7691             0     0     0     0
```

Since `rank(A)=3`, both the backslash operator (i.e., `A\b`) and `pinv(A)*b` provide the same (unique) least square solution:

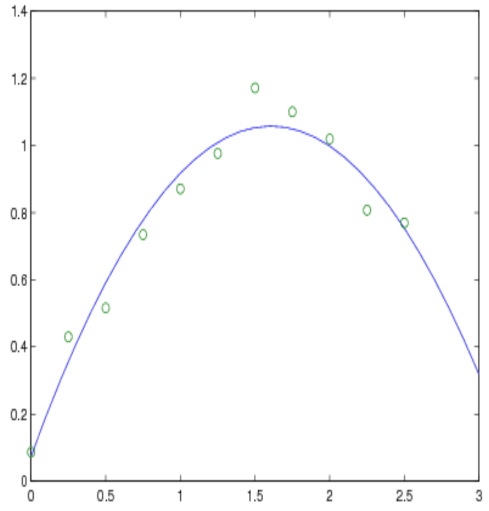
```
>> X=A\b                                      >> Y=pinv(A)*b
X =                                           Y =
    0.0207     % c1                               0.0207     % c1
    1.2570     % c2                               1.2570     % c2
   -0.4041     % c3                              -0.4041     % c3
```

In other words, the least squares fit to the data is

$$y(t) = 0.0207 + 1.2570t - 0.4041t^2$$

The following commands evaluate the model at regularly spaced increments in $t$ (finer than the original spacing) and then plot the result including the original data.

```
>> T=(0:0.1:3.0)';
>> Y=[ones(size(T)) T  T.^2]*X;
>> plot(T,Y,'-',t,y,'o')
```



Another way of obtaining the same least square solution is to enter the command

```
>> polyfit(t,y,2)
ans =
   -0.4041    1.2570    0.0207    % c3   c2   c1
```

For more information on `polyfit`, see Section 13.1 on curve fitting and interpolation.

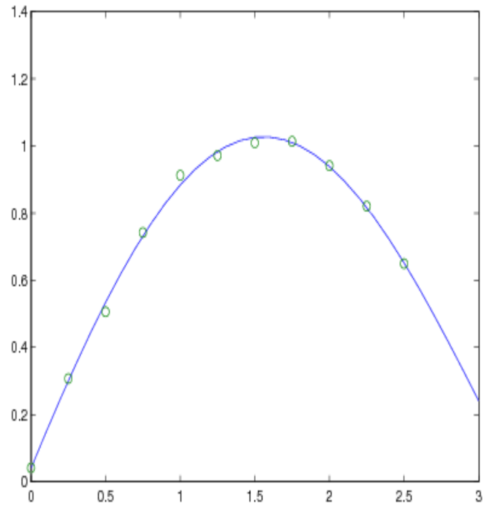
**Note**: *Due to randomness of* `y = sin(t) + rand(size(t))/10`, *your results will differ slightly from the above interpolating polynomial and the graph.*

And here is another attempt at fitting randomly corrupted sine function data into

$$y(t) = c_1 + c_2 \sin(t) + c_2 \cos(t) + c_4 \exp(-t).$$

```
A = [ones(size(t)) cos(t) sin(t) exp(-t)]          >> b=y            >> X=A\b
A =                                                b =               X =
    1.0000    1.0000         0    1.0000              0.0409            -0.0461    % c1
    1.0000    0.9689    0.2474    0.7788              0.3069            -0.0336    % c2
    1.0000    0.8776    0.4794    0.6065              0.5056             1.0582    % c3
    1.0000    0.7317    0.6816    0.4724              0.7419             0.1777    % c4
    1.0000    0.5403    0.8415    0.3679              0.9126
    1.0000    0.3153    0.9490    0.2865              0.9712
    1.0000    0.0707    0.9975    0.2231              1.0092
    1.0000   -0.1782    0.9840    0.1738              1.0137
    1.0000   -0.4161    0.9093    0.1353              0.9412
    1.0000   -0.6282    0.7781    0.1054              0.8205
    1.0000   -0.8011    0.5985    0.0821              0.6493
```

```
>> T=(0:0.1:3.0)';
>> Y=[ones(size(T)) cos(T) sin(T) exp(-T)]*X;
>> plot(T,Y,'-',t,y,'o')
```



And here is another attempt at fitting randomly corrupted sine function data into

$$y(t) = c_1 + c_2 \sin(t) + c_2 \cos(t) + c_4 \exp(-t).$$

The next two examples consider the systems with more equations than unknowns that have exact solutions.

Consider the system generated by 5 equations with 4 unknowns.

```
>> A=magic(5); A = A(:,1:4)                        b=[1;0;-43/648;0;0]
A =                                                b =
    17    24     1     8                              1.0000
    23     5     7    14                                   0
     4     6    13    20                             -0.0664
    10    12    19    21                                   0
    11    18    25     2                                   0
```

By computing the row reduced echelon form of `A`, we immediately get a solution.

```
>> rref([A b])                                             >> X=A\b          >> Y=pinv(A)*b
ans =                                                      X =               Y =
    1.0000         0         0         0   -0.0026          -0.0026           -0.0026
         0    1.0000         0         0    0.0434           0.0434            0.0434
         0         0    1.0000         0   -0.0305          -0.0305           -0.0305
         0         0         0    1.0000    0.0040           0.0040            0.0040
         0         0         0         0         0
```

This solution is also confirmed by using the backslash operator, `A\b`, and `pinv(A)*b`. The solution is unique since the homogeneous system $\mathbf{AZ} = \mathbf{0}$ has only the trivial solution, as verified by the command

```
>> null(A)
ans =
   Empty matrix: 4-by-0
```

Finally, consider the linear system with 8 equations and 6 unknowns as generated by

```
>> A = magic(8); A = A(:,1:6)                 >> b = 260*ones(8,1)
A =                                           b =
    64     2     3    61    60     6             260
     9    55    54    12    13    51             260
    17    47    46    20    21    43             260
    40    26    27    37    36    30             260
    32    34    35    29    28    38             260
    41    23    22    44    45    19             260
    49    15    14    52    53    11             260
     8    58    59     5     4    62             260
```

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to `A*X = b` would be a vector of all 1's. With only six columns, the equations are still consistent, as verified by looking at the row reduced echelon form of `A`.

```
rref([A b])
ans =
     1     0     0     1     1     0     4
     0     1     0     3     4    -3     8
     0     0     1    -3    -4     4    -4
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
```

So a solution exists, but it is not all 1's. Since the matrix is rank deficient (`rank(A)=3`), there are infinitely many solutions. Two of them are

```
>> format rat; X=A\b                                          >> format rat; W=pinv(A)*b
Warning: Rank deficient, rank=3, tol=1.8829e-13.

X =                                                           W =
       4                                                              15/13
       5                                                              19/13
       0                                                              18/13
       0                                                              18/13
       0                                                              19/13
      -1                                                              15/13
```

The general (complete) solution to $\mathbf{AY} = \mathbf{b}$ is given by $\mathbf{Y} = c_1\mathbf{Z_1} + c_2\mathbf{Z_2} + c_3\mathbf{Z_3} + \mathbf{X}$ with

$$\mathbf{Z_1} = \begin{bmatrix} 719/2961 \\ 373/2112 \\ -1161/6842 \\ -2694/3481 \\ 316/595 \\ -75/10834 \end{bmatrix}, \quad \mathbf{Z_2} = \begin{bmatrix} -542/2847 \\ -1262/1851 \\ 93/152 \\ -136/1045 \\ 642/2003 \\ 257/3674 \end{bmatrix}, \quad \mathbf{Z_2} = \begin{bmatrix} 1389/2738 \\ 475/4663 \\ 159/326 \\ -199/1255 \\ -2226/6383 \\ -102/173 \end{bmatrix}, \quad \text{and} \quad \mathbf{X} = \begin{bmatrix} 4 \\ 5 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix},$$

where $c_1$, $c_2$, and $c_3$ are any constants (scalars), and $\mathbf{Z_1}$, $\mathbf{Z_2}$, and $\mathbf{Z_3}$ are fundamental solutions of the system $\mathbf{AZ} = \mathbf{0}$, as verified by

```
>> format rat; null(A)

ans =
     719/2961      -542/2847      1389/2738
     373/2112     -1262/1851       475/4663
   -1161/6842        93/152        159/326
   -2694/3481      -136/1045      -199/1255
     316/595        642/2003     -2226/6383
     -75/10834      257/3674      -102/173
```

12.6. **Finding Eigenvalues and Eigenvectors.** An *eigenvalue* and *eigenvector* of a square $n \times n$ matrix $\mathbf{A}$ are a scalar $\lambda$ and a nonzero (n-dimensional) column vector $\mathbf{v}$ that satisfy

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{1}$$

Usually, one obtains the solution by solving for the $n$ eigenvalues from the characteristic equation

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0,$$

and then solving for the eigenvectors by substituting the corresponding eigenvalues in (1), one at a time. The eigenvectors are determined up to multiplicative scalars. In Matlab, $\lambda$ and $\mathbf{v}$ are computed using `[V,D]=eig(A)` command. For `A=magic(3)`, and requesting the rational format (with `fornat rat`) for better visibility, we have

```
>> A=magic(3)

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

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

V =
    -780/1351       -735/904        -452/1323
    -780/1351       1121/2378      -1121/2378
    -780/1351        452/1323        735/904

D =
      15              0               0
       0           4801/980           0
       0              0           -4801/980
```

Matrix $\mathbf{V}$ contains the eigenvectors of $\mathbf{A}$ as its columns, while matrix $\mathbf{D}$ contains the eigenvalues on its diagonal. By comparing

```
>> A*V(:,1)

ans =
   -1351/156
   -1351/156
   -1351/156
```

```
>> 15*V(:,1)

ans =
   -1351/156
   -1351/156
   -1351/156
```

```
>> A*V(:,2)

ans =
   -3541/889
    1351/585
    3319/1983
```

```
>> (4801/980)*V(:,2)

ans =
   -3541/889
    1351/585
    3319/1983
```

and similarly for the third column of $\mathbf{V}$, we easily recover the eigenvalues and the corresponding eigenvectors of the matrix $\mathbf{A}$.

Also by checking

```
>> format rat; det(V)
ans =
   -1121/1189
```

we observe that $\mathbf{V}^{-1}$ exists, with the property

```
>> format rat; V*D*inv(V)

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

```
>> A=magic(3)

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

Thus, we have $\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$. Furthermore, $\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{D}$. Because of this last property, we say that the matrix $\mathbf{A}$ is *diagonalizable.*

The eigenvalues and eigenvectors can be complex valued as the next example shows:

```
>> A=[0 -6 -1;6 2 -16;-5 20 -10]

A =
   0    -6     -1
   6     2    -16
  -5    20    -10

>> format rat; [V,D]=eig(A)

V =
   -383/460     899/4489 - 387/2777i   899/4489 + 387/2777i
   -1071/3014  -753/3568 - 519/805i   -753/3568 + 519/805i
    -489/1151  -1736/2505             -1736/2505

D =

 -2294/747              0                       0
    0       -1841/747 + 4277/243i               0
    0                   0              -1841/747 - 4277/243i
```

Not every $n \times n$ matrix $\mathbf{A}$ has $n$ linearly independent eigenvectors. For example, it is easy to show that matrix $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ has a double eigenvalue $\lambda = 0$ and only one eigenvector $\mathbf{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

One can prove that if $n \times n$ matrix $\mathbf{A}$ has $n$ distinct eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, then $\mathbf{A}$ has $n$ linearly independent eigenvectors: $\mathbf{K}_1, \ \mathbf{K}_2, \ldots, \mathbf{K}_n$. Furthermore, if $\mathbf{V}$ denotes $n \times n$ matrix whose columns are eigenvectors $\mathbf{K}_1, \ \mathbf{K}_2, \ldots, \mathbf{K}_n$, then

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1} \quad \text{and} \quad \mathbf{D} = \mathbf{V}^{-1}\mathbf{A}\mathbf{V}, \quad \text{where} \quad \mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

Because of the identity $\mathbf{V}^{-1}\mathbf{A}\mathbf{V} = \mathbf{D}$, we say that the matrix $\mathbf{A}$ is *diagonalizable.* An important class of diagonalizable matrices are square symmetric matrices whose entries are real. A square matrix $\mathbf{A}$ is symmetric if it is equal to its *transpose*, i.e., $\mathbf{A} = \mathbf{A}'$. In other words, if for $\mathbf{A} = (a_{ij})_{i,j=1,2,\ldots,n}$ , we have $a_{ij} = a_{ji}$, for $i, j = 1, 2, \ldots, n$.

A matrix with repeated eigenvalues may still have a full set of linearly independent vectors, as the next example shows:

```
>> A=[1 -2 2;-2 1 -2;2 -2 1]

A =
     1    -2     2
    -2     1    -2
     2    -2     1
```

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

V =
    -247/398       1145/2158       780/1351
     279/1870      1343/1673      -780/1351
    1040/1351      1013/3722       780/1351

D =
      -1              0              0
       0             -1              0
       0              0              5
```

Matrix **A** has eigenvalues $\lambda_1 = \lambda_2 = -1$ and $\lambda_3 = 5$.

A more readable expressions for the eigenvectors can be obtained by entering the command `[P,J]=jordan(A)` (see Section 12.8.5 for more information on Jordan decomposition):

```
>> format rat; [P,J]=jordan(A)

P =
   2/3       1/3      -1
   1/3      -1/3       0
  -1/3       1/3       1

J =
   -1        0        0
    0        5        0
    0        0       -1
```

The columns of **P** represent the eigenvectors of **A**. Furthermore, after taking into account that eigenvectors are determined up to multiplicative scalars, here are the eigenvalues and the corresponding eigenvectors:

$$\lambda_1 = -1, \quad \mathbf{K_1} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}; \qquad \lambda_2 = 5, \quad \mathbf{K_2} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}; \qquad \lambda_2 = -1, \quad \mathbf{K_3} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}.$$

As verified below, the first two columns of **V** (i.e., the eigenvectors corresponding to the repeated eigenvalue $-1$) can be expressed as the linear combinations of the eigenvectors $\mathbf{K}_1$ and $\mathbf{K}_3$:

```
>> format rat; [3*P(:,1) P(:,3)]\V(:,1)

ans =
     279/1870      % c1
     624/679       % c2
```

```
>> format rat; [3*P(:,1) P(:,3)]\V(:,2)

ans =
    1343/1673      % c1
     947/881       % c2
```

Consider the following non diagonalizable matrix:

```
>> A=[5 4 2 1;0 1 -1 -1;-1 -1 3 0;1 1 -1 2]

A =
     5     4     2     1
     0     1    -1    -1
    -1    -1     3     0
     1     1    -1     2
```

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

V =
    0.5774    0.5774    0.5774    0.7071
    0.0000    0.0000   -0.5774   -0.7071
   -0.5774   -0.5774    0.0000    0.0000
    0.5774    0.5774    0.5774   -0.0000

D =
    4.0000         0         0         0
         0    4.0000         0         0
         0         0    2.0000         0
         0         0         0    1.0000
```

We observe that the first two columns of matrix $\mathbf{V}$ are identical, thus matrix $\mathbf{A}$ does not have 4 linearly independent eigenvectors. Further checking shows that Matlab cannot compute $\mathbf{V}^{-1}$ within its default error of tolerance:

```
>> det(V)



ans =
   1.3084e-16
```

```
>> inv(V)
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.602469e-16.


ans =

   1.0e+15 *

    1.8014    1.8014    1.8014   -0.0000
   -1.8014   -1.8014   -1.8014    0.0000
    0.0000   -0.0000    0.0000    0.0000
   -0.0000   -0.0000   -0.0000   -0.0000
```

## 12.7. **Generalized Eigenvectors.**

*(This section can be skipped in the first reading of the tutorial.)*

A non diagonalizable $n\times n$ matrix does not have $n$ linearly independent eigenvectors. Generalized eigenvectors are introduced to form a complete basis of such matrix. A generalized eigenvector of matrix $\mathbf{A}$ is a nonzero vector $\mathbf{w}$, which has associated with it an eigenvalue $\lambda$ having algebraic multiplicity $m \geq 1$, satisfying

$$(\mathbf{A}-\lambda\mathbf{I})^m\mathbf{w}=\mathbf{0}, \tag{2}$$

where $\mathbf{I}$ denotes $n\times n$ identity matrix. For $m=1$, generalized eigenvectors become ordinary eigenvectors.

Assume $\lambda$ is an eigenvalue of multiplicity $m \geq 1$, and there is only one eigenvector $\mathbf{v}_1$. The generalized eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$ satisfy

$$(\mathbf{A}-\lambda\mathbf{I})\mathbf{v}_k=\mathbf{v}_{k-1}, \qquad k=1,2,\ldots,m, \quad \text{with} \quad \mathbf{v}_0=\mathbf{0},$$

and they are linearly independent. Furthermore, for any $k \geq 1$,

$$(\mathbf{A}-\lambda\mathbf{I})^k\mathbf{v}_k=\mathbf{0}. \tag{3}$$

In particular, (3) yields

$$(\mathbf{A}-\lambda\mathbf{I})^m\mathbf{v}_k=0, \qquad k=1,2,\ldots,m,$$

and thus implying (2).

Matlab `jordan(A)` function can be used to find generalized eigenvectors of matrix $\mathbf{A}$ (see also Section 12.8.5 for more information on Jordan decomposition). Consider the following example:

```
>> A=[5 4 2 1;0 1 -1 -1;-1 -1 3 0;1 1 -1 2]

A =
     5     4     2     1
     0     1    -1    -1
    -1    -1     3     0
     1     1    -1     2
```

```
>> [P,J]=jordan(A)

P =

     1     1    -1     1
     0     0     1    -1
    -1     0     0     0
     1     0     0     1

J =
     4     1     0     0
     0     4     0     0
     0     0     1     0
     0     0     0     2
```

The numbers 1, 2, and 4 are the eigenvalues of matrix **A**. Furthermore 4 is double eigenvalue. Let us denote the columns of **P** by $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$, and $\mathbf{p_4}$, then we have

$$(\mathbf{A}-2\mathbf{I})\mathbf{p_4}=\mathbf{0}, \quad (\mathbf{A}-1\mathbf{I})\mathbf{p_3}=\mathbf{0}, \qquad (\mathbf{A}-4\mathbf{I})\mathbf{p_1}=\mathbf{0}, \quad \text{and} \quad (\mathbf{A}-4\mathbf{I})\mathbf{p_2}=\mathbf{p_1}.$$

Indeed,

```
>> (A-2*eye(4))*P(:,4)
ans =

     0
     0
     0
     0
```

```
>> (A-1*eye(4))*P(:,3)
ans =

     0
     0
     0
     0
```

```
>> (A-4*eye(4))*P(:,1)

ans =
     0
     0
     0
     0
```

```
>> (A-4*eye(4))*P(:,2)

ans =
     1
     0
    -1
     1
```

Thus $\mathbf{p_1}$, $\mathbf{p_3}$, and $\mathbf{p_4}$ are ordinary eigenvectors corresponding to the eigenvalues 4, 1, and 2, respectively. Since $(\mathbf{A}-4\mathbf{I})\mathbf{p_2}=\mathbf{p_1}$ and $(\mathbf{A}-4\mathbf{I})\mathbf{p_1}=\mathbf{0}$, we also have $(\mathbf{A}-4\mathbf{I})^2\mathbf{p_2}=\mathbf{0}$ and $\mathbf{p_2}$ is a generalized eigenvector corresponding to the eigenvalue 4.

The matrix in the next example has the eigenvalues 1 (with multiplicity 3), 2 (with multiplicity 2), and 3 with multiplicity 1).

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

A =
     1     1     1     2    -1     2
     0     1     2    -1    -1     2
     0     0     1     2     1     2
     0     0     0     2     2    -1
     0     0     0     0     2     2
     0     0     0     0     0     3
```

```
>> format rat; [P,J]=jordan(A)
P =

29/4     -58       116       -29     -145/2        -493/4
 7/2    -174/7   2378/49      0        -29           -58
 5      -116/7   435/49       0          0          -29/2
 3      -58/7    -87/49       0          0             0
 2        0      -29/7        0          0             0
 1        0        0          0          0             0

J =

3         0         0         0          0             0
0         2         1         0          0             0
0         0         2         0          0             0
0         0         0         1          1             0
0         0         0         0          1             1
0         0         0         0          0             1
```

Let us denote the columns of $\mathbf{P}$ by $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$, $\mathbf{p_4}$, $\mathbf{p_5}$, $\mathbf{p_6}$. then we have,

$$(\mathbf{A}-3\mathbf{I})\mathbf{p_1}=\mathbf{0},\ (\mathbf{A}-2\mathbf{I})\mathbf{p_2}=\mathbf{0},\ (\mathbf{A}-2\mathbf{I})\mathbf{p_3}=\mathbf{p_2},\ (\mathbf{A}-\mathbf{I})\mathbf{p_4}=\mathbf{0},\ (\mathbf{A}-\mathbf{I})\mathbf{p_5}=\mathbf{p_4},\ \text{and } (\mathbf{A}-\mathbf{I})\mathbf{p_6}=\mathbf{p_5},$$

Since $(\mathbf{A}-2\mathbf{I})\mathbf{p_2}=\mathbf{0}$ and $(\mathbf{A}-2\mathbf{I})\mathbf{p_3}=\mathbf{p_2}$, we have $(\mathbf{A}-2\mathbf{I})^2\mathbf{p_3}=\mathbf{0}$ and $\mathbf{p_3}$ is a generalized eigenvector corresponding to the eigenvalue 2. Also, since $(\mathbf{A}-\mathbf{I})\mathbf{p_4}=\mathbf{0}$, $(\mathbf{A}-\mathbf{I})\mathbf{p_5}=\mathbf{p_4}$, $(\mathbf{A}-\mathbf{I})\mathbf{p_6}=\mathbf{p_5}$, we have $(\mathbf{A}-\mathbf{I})^3\mathbf{p_6}=\mathbf{0}$ and $(\mathbf{A}-\mathbf{I})^3\mathbf{p_5}=\mathbf{0}$, implying that $\mathbf{p_5}$ and $\mathbf{p_6}$ are generalized eigenvectors corresponding to the eigenvalue 1. On the other hand, $\mathbf{p_1}$, $\mathbf{p_2}$, and $\mathbf{p_4}$ are the ordinary eigenvectors corresponding to the eigenvalues, 3, 2, and 1, respectively.

In the last example, the matrix also has the eigenvalues 1 (with multiplicity 3), 2 (with multiplicity 2), and 3 with multiplicity 1), however here, there are two linearly independent eigenvectors corresponding to the eigenvalue 1.

```
>> A=[1 0 1 2 3 4;0 1 5 8 -1 2; 0 0 1 7 4 2;0 0 0 2 6 9; 0 0 0 0 2 5; 0 0 0 0 0 3]

A =
     1     0     1     2     3     4
     0     1     5     8    -1     2
     0     0     1     7     4     2
     0     0     0     2     6     9
     0     0     0     0     2     5
     0     0     0     0     0     3
```

```
>> format rat; [P,J]=jordan(A)

P =
 489/4      -2093/10  -29117/56    -18837/43   22271/56    991/1554
2093/4     -2093/2    -10465/4      -2093       2093         0
 295/2         0        -2093/10   -14651/43   22637/404     0
  39           0           0         -2093/43   -4216/117     0
   5           0           0            0        2093/258     0
   1           0           0            0          0          0

J =
   3           0           0            0          0          0
   0           1           1            0          0          0
   0           0           1            0          0          0
   0           0           0            2          1          0
   0           0           0            0          2          0
   0           0           0            0          0          1
```

Let us denote the columns of $\mathbf{P}$ by $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$, $\mathbf{p_4}$, $\mathbf{p_5}$, $\mathbf{p_6}$. We have,

$$(\mathbf{A}-3\mathbf{I})\mathbf{p_1}=\mathbf{0},\ (\mathbf{A}-\mathbf{I})\mathbf{p_2}=\mathbf{0},\ (\mathbf{A}-\mathbf{I})\mathbf{p_3}=\mathbf{p_2},\ (\mathbf{A}-2\mathbf{I})\mathbf{p_4}=\mathbf{0},\ (\mathbf{A}-2\mathbf{I})\mathbf{p_5}=\mathbf{p_4},\ \text{and } (\mathbf{A}-\mathbf{I})\mathbf{p_6}=\mathbf{0},$$

Since $(\mathbf{A}-\mathbf{I})\mathbf{p_2}=\mathbf{0}$ and $(\mathbf{A}-\mathbf{I})\mathbf{p_3}=\mathbf{p_2}$, we have $(\mathbf{A}-\mathbf{I})^2\mathbf{p_3}=\mathbf{0}$, thus $\mathbf{p_3}$ is a generalized eigenvector corresponding to the eigenvalue 1. Also, since $(\mathbf{A}-2\mathbf{I})\mathbf{p_4}=\mathbf{0}$ and $(\mathbf{A}-2\mathbf{I})\mathbf{p_5}=\mathbf{p_4}$, thus we have $(\mathbf{A}-2\mathbf{I})^2\mathbf{p_5}=\mathbf{0}$ and $\mathbf{p_5}$ is a generalized eigenvector corresponding to the eigenvalue 1. On the other hand, $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_4}$, and $\mathbf{p_6}$, are the ordinary eigenvectors corresponding to the eigenvalues, 3, 1, 2, and 1, respectively.

## 12.8. **Matrix Factorizations, Matrix Powers and Exponentials.**

*(This section can be skipped in the first reading of the tutorial.)*

12.8.1. *Cholesky factorization.* The Cholesky factorization expresses a symmetric (or more generally Hermitian) and positive definite matrix $\mathbf{A}$ as the product of an upper triangular matrix $\mathbf{R}$ and its transpose $\mathbf{R}'$, i.e., $\mathbf{A}=\mathbf{R}'\mathbf{R}$.

```
>> A=pascal(5)

A =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70

>> R=chol(A)

R =
     1     1     1     1     1
     0     1     2     3     4
     0     0     1     3     6
     0     0     0     1     4
     0     0     0     0     1
```

```
>> R'*R                  % verification

ans =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70
```

For a symmetric (or more generally Hermitian) and positive definite matrix the Cholesky factorization exists. In this case, it is also unique in the sense that there exists only one upper triangular matrix $\mathbf{R}$ with strictly positive diagonal entries such that $\mathbf{A}=\mathbf{R}'\mathbf{R}$.

12.8.2. *LU factorization.* The LU factorization is a matrix decomposition which writes a matrix $\mathbf{A}$ as the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, i.e., $\mathbf{A} = \mathbf{LU}$. The product sometimes includes a permutation matrix as well. The LU decomposition is a modified form of Gaussian elimination. Not all square matrices have LU factorization, For example, $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ cannot be expressed at the product of triangular matrices. Even when such a factorization exists it is not unique (since $\mathbf{A} = \mathbf{LU} = (-\mathbf{L})(-\mathbf{U})$), unless one puts some restrictions on $\mathbf{L}$ and $\mathbf{U}$ matrices. Often, one such conditions is to require that the diagonal of $\mathbf{L}$ (or $\mathbf{U}$) consists of ones. Though, the matrix

$$\mathbf{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 0 & -1/\epsilon \end{bmatrix} \tag{4}$$

can be expressed as the product of the lower and upper triangular matrices, however, when $\epsilon$ is small, the factors are large and magnify errors, so even the permutations matrix is not necessary, it is desirable. For a given matrix $\mathbf{A}$, Matlab command `[L,U,P]=lu(A)` returns an upper triangular matrix $\mathbf{U}$, a lower triangular matrix $\mathbf{L}$ with ones on its diagonal, and a permutation matrix $\mathbf{P}$, such that $\mathbf{LU} = \mathbf{PA}$. (A permutation matrix is matrix of zeros and ones that has exactly one entry 1 in each row and column.) Furthermore, such decomposition always exist (even $\mathbf{A}$ is not square matrix). On the other hand, the command `[L,U]=lu(A)` returns an upper triangular matrix $\mathbf{U}$, and a permuted lower triangular matrix $\mathbf{L}$ such that $\mathbf{LU} = \mathbf{A}$. LU decomposition is used in numerical analysis to solve systems of linear equations or calculate the determinant. Finally, when $\mathbf{A}$ is symmetric (Hermitian) and positive definite matrix, one can choose $\mathbf{L}$ in such a way that $\mathbf{L} = \mathbf{U}'$ and we get back to the Cholesky factorization.

For matrix in (4) with $\epsilon = 1/1000$, we have

```
>> format rat; A=[0.001 1;1 0]
A =
       1/1000         1
       1              0
```

```
>> [L,U,P]=lu(A)
L =
       1              0
       1/1000         1

U =
       1              0
       0              1

P =
       0              1
       1              0
```

```
>> format rat; A=[0.001 1;1 0]
A =
       1/1000         1
       1              0
```

```
>> [L,U]=lu(A)
L =
       1/1000         1
       1              0

U =
       1              0
       0              1
```

And here is another example.

```
>> A=magic(5)
A =
   17     24       1       8      15
   23      5       7      14      16
    4      6      13      20      22
   10     12      19      21       3
   11     18      25       2       9

>> format rat; [L,U,P]=lu(A)
L =

       1          0            0            0            0
     17/23        1            0            0            0
     11/23     359/467         1            0            0
      4/23     118/467     1199/2322        1            0
      10/23    226/467     1679/2322      12/13          1

U =
      23         5           7           14           16
       0      467/23       -96/23       -54/23        73/23
       0         0        6787/273    -1350/467     -510/467
       0         0           0         845/43      2752/145
       0         0           0            0          -200/9

P =
       0         1           0            0            0
       1         0           0            0            0
       0         0           0            0            1
       0         0           1            0            0
       0         0           0            1            0
```

12.8.3. *QR factorization.* An *orthogonal* matrix **Q** is a real matrix whose columns all have length one and are perpendicular, equivalently $\mathbf{Q}'\mathbf{Q} = \mathbf{I}$, where **I** is the identity matrix. This also implies that $\mathbf{Q}^{-1} = \mathbf{Q}'$. The following is the simplest $2 \times 2$ orthogonal matrix representing two-dimensional coordinate rotations:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

For complex matrices, the corresponding term is *unitary.* Orthogonal (unitary) matrices preserve length, preserve angles, and do not magnify errors, and thus they are often used in numerical computations.

The `qr` function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of an orthogonal (unitary) matrix and an upper triangular matrix.

Matlab command `[Q,R] = qr(A)` produces an upper triangular matrix **R** of the same dimension as **A** and a unitary matrix **Q** so that $\mathbf{A} = \mathbf{QR}$. If $m \times n$ is the size of **A**, then **Q** is $m \times m$ and **R** is $m \times n$. Since we also have $\mathbf{A} = (-\mathbf{Q})(-\mathbf{R})$, with $-\mathbf{Q}$ being orthogonal (unitary) and $-\mathbf{R}$ being upper triangular, this factorization is not unique. If **A** is square and invertible matrix, then this factorization is unique if, for example, we require that the diagonal elements of **R** are positive (negative).

```
>> A=[12 -51 4; 6 167 -68;-4 24 -41]
A =
      12            -51            4
       6            167          -68
      -4             24          -41

>> format rat; [Q,R]=qr(A)
Q =
      -6/7           69/175        58/175
      -3/7         -158/175        -6/175
       2/7           -6/35         33/35

R =
     -14            -21           14
       0           -175           70
       0              0          -35
```

```
>> Q*R            % verification
ans =
      12            -51            4
       6            167          -68
      -4             24          -41
```

12.8.4. *Schur decomposition.* For a matrix $\mathbf{A}$, there exist orthogonal (unitary) matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{U}$ such that

$$\mathbf{A} = \mathbf{Q}\mathbf{U}\mathbf{Q}^{-1}.$$

Note that $\mathbf{Q}^{-1} = \mathbf{Q}'$. Since $\mathbf{U}$ is similar to $\mathbf{A}$ and $\mathbf{U}$ is triangular, the eigenvalues of $\mathbf{A}$ are the diagonal entries of $\mathbf{U}$. The corresponding Matlab command is `[Q,U]=schur(A)`.

```
>> A=[6 12 19;-9 -20 -33;4 9 15]
A =
       6             12           19
      -9            -20          -33
       4              9           15
```

```
>> format rat; [Q,U]=schur(A)
Q =
    -540/1139        926/1393       780/1351
    1467/1805        286/3657       780/1351
    -489/1444      -2506/3373       780/1351

U =
        -1          1351/65       -9520/213
         0             1           -523/858
         0             0              1
```

12.8.5. *Jordan normal (canonical) form.*

*(This section requires Symbolic Math Toolbox.)*

For a given square matrix $\mathbf{A}$ there exists an invertible matrix $\mathbf{P}$ (similarity matrix) such that $\mathbf{P}^{-1}\mathbf{A}\mathbf{P} = \mathbf{J}$ and $\mathbf{J}$ is block diagonal matrix

$$\mathbf{J} = \begin{bmatrix} J_1 & & \\ & \ddots & \\ & & J_k \end{bmatrix}$$

where each block $J_i$ $(1 \leq i \leq k)$ is a $1 \times 1$ matrix or it is a square matrix of the form of

$$J_i = \begin{bmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{bmatrix}$$

$\mathbf{J}$ is such that the only non-zero entries of $\mathbf{J}$ are on the diagonal and the super-diagonal. $\mathbf{J}$ is called the *Jordan normal form* of $\mathbf{A}$ and each $J_i$ is called a *Jordan block* of $\mathbf{A}$. In a given Jordan block, every entry on the super-diagonal is 1. The Jordan decomposition is not unique.

We have the following properties:

- Counting multiplicity, the eigenvalues of $\mathbf{J}$ (and therefore $\mathbf{A}$) are the diagonal entries.
- Given an eigenvalue $\lambda_i$, its *geometric multiplicity* is the dimension of $\mathrm{Ker}(\mathbf{A}-\lambda\mathbf{I})$, and it is the number of Jordan blocks corresponding to $\lambda_i$.
- The sum of the sizes of all Jordan blocks corresponding to an eigenvalue $\lambda_i$ is its *algebraic multiplicity*.
- $\mathbf{A}$ is diagonalizable if and only if, for every eigenvalue $\lambda$ of $\mathbf{A}$, its geometric and algebraic multiplicities coincide.
- $\mathbf{A} = \mathbf{PJP}^{-1}$ and the columns of $\mathbf{P}$ are the generalized eigenvectors of $\mathbf{A}$ (see Section 12.7).

The following $11 \times 11$ Jordan block matrix

$$\begin{bmatrix}
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & i & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & i & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & i & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & i & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7
\end{bmatrix}$$

has

one $1 \times 1$ block corresponding to eigenvalue 2 (with algebraic multiplicity 1 and geometric multiplicity 1);
one $3 \times 3$ block corresponding to eigenvalue 0 (with algebraic multiplicity 3 and geometric multiplicity 1);
two $2 \times 2$ blocks corresponding to eigenvalue $i$ (with algebraic multiplicity 4 and geometric multiplicity 2);
one $3 \times 3$ block corresponding to eigenvalue 7 (with algebraic multiplicity 3 and geometric multiplicity 1).

Matlab `J=jordan(A)` computes the Jordan canonical (normal) form of $\mathbf{A}$, The matrix must be known exactly. Thus, its elements must be integers or ratios of small integers. Any errors in the input matrix may completely change the Jordan canonical form.

The command `[P,J]=jordan(A)` computes both $\mathbf{J}$, the Jordan canonical form, and the similarity transform, $\mathbf{P}$, whose columns are the generalized eigenvectors.

```
>> A=[5 4 2 1;0 1 -1 -1;-1 -1 3 0;1 1 -1 2]
A =
     5     4     2     1
     0     1    -1    -1
    -1    -1     3     0
     1     1    -1     2
```

```
>> [P,J]=jordan(A)
P =
     1     1    -1     1
     0     0     1    -1
    -1     0     0     0
     1     0     0     1

J =
     4     1     0     0
     0     4     0     0
     0     0     1     0
     0     0     0     2
```

12.8.6. *Matrix Powers and Exponentials.* If $\mathbf{A}$ is a square matrix and $k$ a positive integer, then `A^k` effectively multiplies $\mathbf{A}$ by itself $k-1$ times. If $\mathbf{A}$ is a square and invertible matrix and $k$ a positive integer, then `A^(-k)` effectively multiplies $\mathbf{A}^{-1}$ (i.e., `inv(A)`, in Matlab notation) by itself $k-1$ times. Fractional powers, like `A^(3/4)` are also permitted.

The operator `.^` produces element-by-element powers. For example, for `A=[1 2 3;3 4 5;6 7 8]`,

```
>> A^2                                          >> A.^2
ans =                                           ans =
      25          31          37                     1           4          9
      45          57          69                     9          16         25
      75          96         117                    36          49         64
```

The command `sqrtm(A)` computes `A^(1/2)` by a more accurate algorithm. The `m` in `sqrtm` distinguished this function from `sqrt(A)` which, like `A.^(1/2)` does its job element-by-element.

The command `expm(A)` computes the computes the exponential of a square matrix $\mathbf{A}$, i.e., $e^{\mathbf{A}}$. This operation should be distinguished from `exp(A)` which computes the element-by-element exponential of $\mathbf{A}$.

```
>> A=[1 2; 0 1]                                 >> expm(A)
A =                                             ans =
       1               2                            2.7183    5.4366
       0               1                                 0    2.7183
```

Using *Symbolic Math Toolbox* additional symbolics operations are possible:

```
>> syms t;                            % defines symbolic variable t
>> f=inline('expm(t*A)')              % defines symbolic inline
                                      % function in A and t
f =
     Inline function:
     f(A,t) = expm(t*A)

>> f(A,t)                                       >> f(A,1)

ans =                                           ans =

[ exp(t), 2*t*exp(t)]                               2.7183    5.4366
[      0,     exp(t)]                                    0    2.7183
```

12.9. **Characteristic Polynomial of a Matrix and Cayley-Hamilton Theorem.** Matlab command `p = poly(A)`, where $\mathbf{A}$ is $n\times n$ matrix returns an $n+1$ element row vector whose elements are the coefficients of the characteristic polynomial of $\mathbf{A}$:

$$\det(\mathbf{A}-\lambda\mathbf{I})=0.$$

The coefficients are ordered in descending powers: if a vector $\mathbf{c}=(c_1,c_2,\dots,c_{n+1})$ has $n+1$ components, the polynomial it represents is

$$c_1\lambda^n+\dots+c_n\lambda+c_{n+1}.$$

The command `p = poly(r)`, where $\mathbf{r}$ is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of $\mathbf{r}$.

```
>> A=[1 2 3;4 5 6;7 8 0]          >> format rat; p=poly(A)
A =                               p =
     1     2     3                    1    -6   -72   -27
     4     5     6
     7     8     0

>> A=[1 2 3;4 5 6;7 8 0]          >> p=poly(sym(A)) % requires Symbolic Math Toolbox
A =                                                 % computes the characteristic polynomial
     1     2     3                                  % of A in symbolic variable x
     4     5     6                p =
     7     8     0                x^3 - 6*x^2 - 72*x - 27
```

```
>> A=[1 2 3;4 5 6;7 8 0]          >> pA = polyvalm(p,A)
A =                               pA =
     1     2     3                      *       *       *
     4     5     6                      *       *       *
     7     8     0                      *       *       *

>> format rat; p=poly(A)
p =
    1    -6   -72   -27
```

The command `polyvalm(p,A)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `A` in the polynomial `p`. The zero matrix `pA` verifies Cayley-Hamilton theorem. This theorem states that if $p(\lambda) = p_0 + p_1\lambda + \ldots + (-1)^n\lambda^n$ is the characterstic polynomial of $\mathbf{A}$, then

$$p(\mathbf{A}) = p_0\mathbf{I} + p_1\mathbf{A} + \ldots + (-1)^n\mathbf{A}^n = \mathbf{0}.$$

Finally, the last script below computes the roots of the characteristic polynomial `p`.

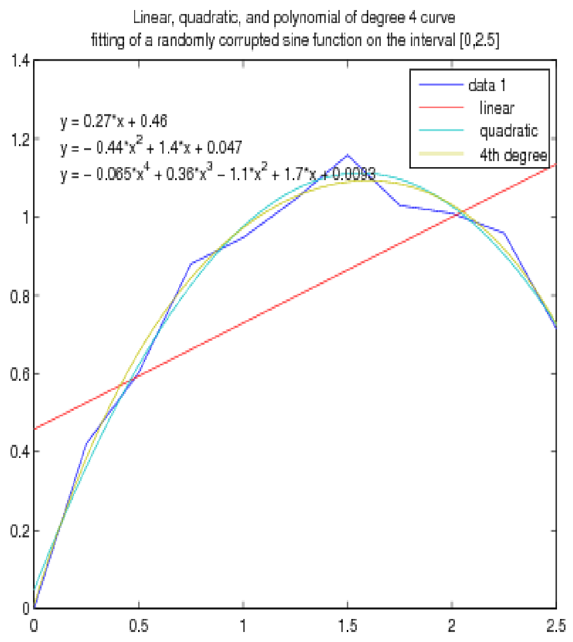
```
>> A=[1 2 3;4 5 6;7 8 0]
A =
     1     2     3
     4     5     6
     7     8     0

>> format rat; p=poly(A)          >> format rat; roots(p)    % computes the roots of
p =                               ans =                      % the polynomial p, i.e.,
    1    -6   -72   -27               2170/179               % the eigenvalues of A
                                      -648/113
                                      -769/1980
```

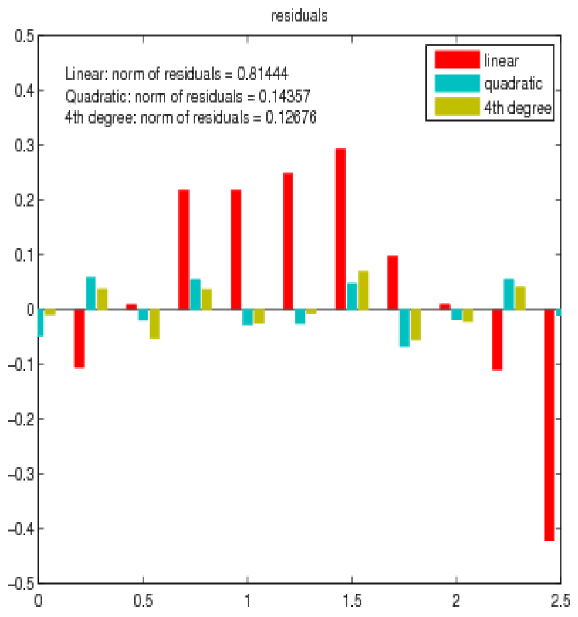
## 13. Curve Fitting and Interpolation

Curve fitting is the process of constructing a curve, or mathematical function, that has the best fit (in a *least square sense*) to a series of data points, possibly subject to additional constraints. Interpolation is the method of constructing new data points within the range of a discrete set of known data points. Extrapolation is the process of constructing new data points outside a discrete set of known data points.

Matlab has many utilities for curve fitting and interpolation. The example below involves fitting the randomly corrupted sine function using linear, quadratic, and polynomial of degree 4 curves fitting.

```
>> x = (0: 0.25: 2.5);
>> y = sin(x) + rand(size(x))/5;
>> plot(x,y)
```





Go to your figure window and click on `Tools`, and select `Basic Fitting`. A new window opens with `Basic Fitting` menu. Check `linear, quadratic`, and `4th degree polynomial`, together with the boxes for `Show equations, Plot residuals`, and `Show norm of residuals`. Select also `Separate figure`.

**Note**: *Due to randomness of* `y = sin(x) + rand(size(x))/5`, *your results will differ slightly from the above interpolating polynomials, graphs and norms.*

13.1. **Hands-on Understanding of Curve Fitting Using Polynomial Functions.** `p = polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length `n+1` containing the polynomial coefficients in descending powers of $x$, i.e, $p = [p_1, p_2 \ldots, p_n, p_{n+1}]$ corresponding to

$$p(x) = p_1 x^n + p_2 x^{n-1} + \cdots + p_n x + p_{n+1}$$

Next, the command `polyval(p,x)` evaluates the polynomial at the data points `x(i)` and generates the values `y(i)` such that

$$y_i = p_1 x_i^n + p_2 x_i^{n-1} + \cdots + p_n x_i + p_{n+1}.$$

The next example involves fitting the error function, `erf(x)`, by a polynomial in `x` of degree 6. In mathematics, the error function (also called the Gauss error function) is a special function (non-elementary) of sigmoid shape which occurs in probability, statistics, materials science, and partial differential equations. It is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

```
>> x = (0: 0.1: 2.5)';
>> y=erf(x);
```

First generate x, a (column) vector of equally spaced points in the interval; then evaluate erf(x) at those points.

```
>> p = polyfit(x,y,6)
p =
    0.0084   -0.0983    0.4217   -0.7435    0.1471    1.1064    0.0004
```

Compute the coefficients in the approximating polynomial of degree 6.

There are seven coefficients and the polynomial is

$$0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004$$

```
>> f = polyval(p,x);
```

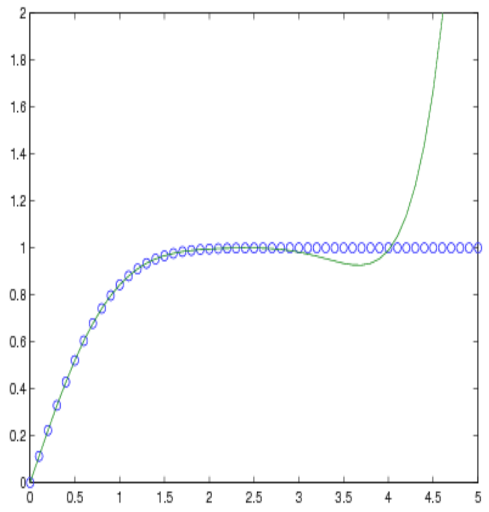
Evaluates the polynomial at the data points.

```
>> table = [x y f y-f]
table =
         0         0    0.0004   -0.0004
    0.1000    0.1125    0.1119    0.0006
    0.2000    0.2227    0.2223    0.0004
    0.3000    0.3286    0.3287   -0.0001
    0.4000    0.4284    0.4288   -0.0004
    ...

    2.1000    0.9970    0.9969    0.0001
    2.2000    0.9981    0.9982   -0.0001
    2.3000    0.9989    0.9991   -0.0003
    2.4000    0.9993    0.9995   -0.0002
    2.5000    0.9996    0.9994    0.0002
```

The (partial) table is showing the data, fit, and error.

So, on the interval $[0, 2.5]$, the fit is good to between three and four digits. Beyond this interval, the graph below shows that the polynomial behavior takes over and the approximation quickly deteriorates.

```
>> x = (0: 0.1: 5)';
>> y = erf(x);
>> f = polyval(p,x);
>> plot(x,y,'o',x,f,'-')
>> axis([0  5  0  2])
```



13.2. **Interpolation.** Given the data points $(x_i, y_i)$, find $y_j$ at desired $x_j \neq x_i$ from $y_j = f(x_j)$, where $f$ is a continuous function to be found from interpolation. The command is `ynew = interp1(x,y,xnew, method)`. The methods are:

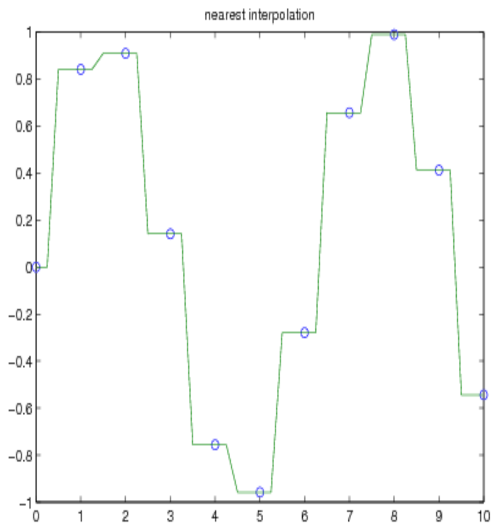
| | |
|---|---|
| 'nearest' | Nearest neighbor interpolation |
| 'linear' | Linear interpolation (default) |
| 'spline' | Cubic spline interpolation |
| 'pchip' | Piecewise cubic Hermite interpolation |
| 'cubic' | (Same as 'pchip') |

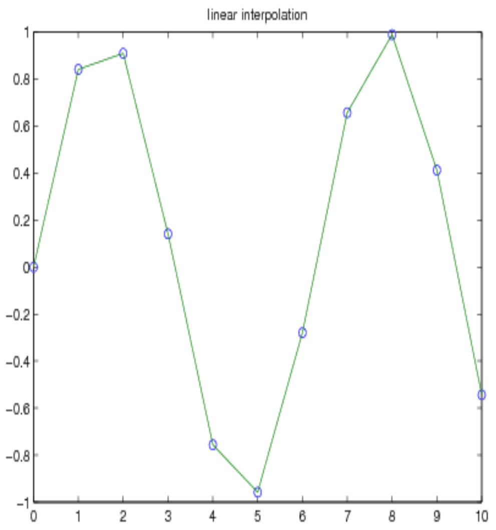
The default method is `linear`.

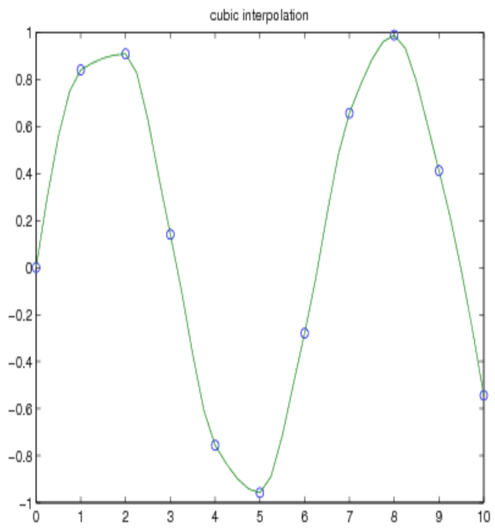
As an example, we generate a coarse sine curve and interpolate over a finer abscissa.

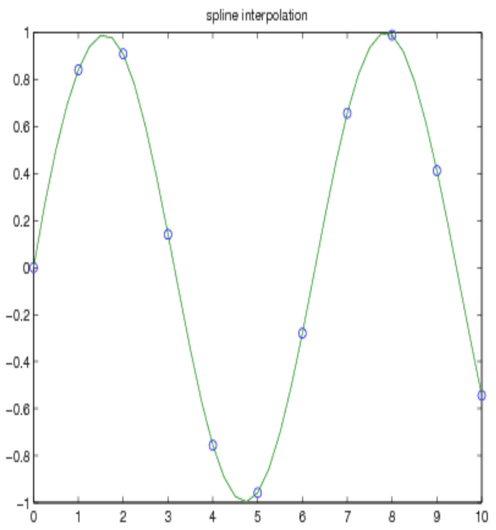
```
x = 0:10;
y = sin(x);
xi = 0:.25:10;
yi = interp1(x,y,xi,'nearest');      Uses nearest method.
plot(x,y,'o',xi,yi)
```

The figures below show the results of different interpolating methods.









For other interpolating functions/methods check Matlab help on `spline` or `interpft`. For two- and three-dimensional analogs of `interp1`, check Matlab help on `interp2` and `interp3` commands, respectively.

## 14. Solving Nonlinear Algebraic Equations

The command `fzero` solves nonlinear equations involving one variable of the form:

$$f(x) = 0.$$

In the first example, we calculate $\pi$ by finding the zero of the sine function near 3.

```
>> x0=fzero('sin',3)
x0 =
    3.1416

>> x0=fzero('my_f',10)
x0 =
    1.0193
```

`my_f` is from the function file defined in Section 5.

```
>> x = fzero(@cos,[1 2])
x =
    1.5708
```

Finds the zero of cosine between 1 and 2. Use the command `help function_handle` to check what is the meaning of `@` symbol in this command.

```
>> f = @(x)x.^3-2*x-5;
>> z = fzero(f,2)
z =
    2.0946
```

An alternative way to find the zeros of the function $f(x) = x^3 - 2x - 5$ is to use an anonymous function `@(x)x.^3-2*x-5` and find the zero near 2.

Because $f(x) = 1 \cdot x^3 + 0 \cdot x^2 + (-2) \cdot x^1 + (-5) \cdot x^0$ is a polynomial, the following command,

```
>> roots([1 0 -2 -5])
ans =
   2.0946
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

finds the same zero and the remaining roots of this polynomial.

## 15. Numerical Integration - Quadrature

Numerical integration (often abbreviated to *quadrature*) is an algorithm for calculating the numerical value of a definite integral. Matlab command `y = quad('your_function',a,b,tol)` evaluates the integral of `your_function` of one variable from $a$ to $b$, using adoptive Simpson's rule. The optional parameter `tol` specifies absolute tolerance. The default value is $10^{-6}$,

If one uses the statement `[y,evals] = quad('your_function',a,b,tol)`, the procedure also returns, `evals`, the number of function evaluations used to find the numerical value of the integral.

For example, in order to compute

$$\int_0^2 \frac{1}{x^3 - 2x - 5}\, dx$$

type the following:

```
>> y = quad('1./(x.^3-2*x-5)',0,2)
y =
   -0.4605

>> f = inline('1./(x.^3-2*x-5)');
>> [y,evals] = quad(f,0,2)
y =
   -0.4605
evals =
    41
```

An alternative way to find the integral of the same function with the number of function evaluations provided as an outcome.

```
>> [integral,evals] = quad('my_f',0,1)
integral =
    0.0917
evals =
   125
```

The definite integral from 0 to 1 of `my_f`, the function file defined in Section 5.

Another Matlab quadrature, `y = quadl('your_function',a,b,tol)`, numerically evaluates integral using adaptive Lobatto algorithm. Often this quadrature is more accurate than `quad` method. In order to compare these two quadratures, we compute the integral,

$$\int_{0.3}^{0.7} \exp(-x^2)\, dx = \frac{\sqrt{\pi}}{2}\left[\operatorname{erf}(0.7) - \operatorname{erf}(0.3)\right],$$

where $\operatorname{erf}(x)$ is the internal Matlab function (see also Section 13.1),

```
>> format long
>> value = (sqrt(pi)/2)*(erf(0.7)-erf(0.3));
>> [y,evals] = quad('exp(-x.^2)',0.3,0.7);
>> [yl,evalsl] = quadl('exp(-x.^2)',0.3,0.7);
>> table = [value y yl (value-y)/value (value-yl)/value evals evalsl]'
table =
   0.309447785425779
   0.309447785513074
   0.309447785425838
  -0.000000000282099
  -0.000000000000191
  13.000000000000000
  18.000000000000000
```

With the default accuracy quad gives only $(0.00028 \times 10^{-6})$% error, while quadl gives $(0.00000019 \times 10^{-6})$% error. The algorithm quad uses 13 and quadl uses 18 functions evaluations to compute the integral.

## 15.1. **Quadrature For Double Integrals.**

For double integrals,

$$\int_{y_{min}}^{y_{max}} \left( \int_{x_{min}}^{x_{max}} f(x,y)\,dx \right) dy,$$

Matlab provides `q = dblquad('your_function',xmin,xmax,ymin,ymax,tol,@method)`, where `@method` can be `@quad` (default) or `@quadl`. You need `[]` as a placeholder for the default value of `tol`, when the optional argument `@method` is used. For example, for the double integral,

$$\int_0^{\pi} \left[ \int_{\pi}^{2\pi} (y \sin x + x \cos y)\; dx \right] dy = -\pi^2,$$

we can compare quad and quadl methods:

```
>> format long
>> value =-pi^2;
>> q = dblquad('y*sin(x)+x*cos(y)',pi,2*pi,0,pi);
>> ql = dblquad('y*sin(x)+x*cos(y)',pi,2*pi,0,pi,[],@quadl);
>> table = [value q ql (value-q)/value (value-ql)/value]'
table =
  -9.869604401089358
  -9.869604377254573
  -9.869604289913852
   0.000000002414969
   0.000000011264434
```

Note that the lower order method (quad) performs better in this case than the higher order method (quadl).

## 16. Solving Ordinary Differential Equations

Matlab has many algorithms for solving systems of ordinary differential equations (ODE). Two procedures, ode23 and ode45, described below are implementations of 2nd/3rd-order and 4th/5th-order Runge-Kutta methods, respectively.

For the system of ODE in the vector form,

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t),$$

where

$$\mathbf{x} = [x_1\ x_2\ \dots\ x_n]^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \dot{\mathbf{x}} = \left[\frac{dx_1}{dt}\ \frac{dx_2}{dt}\ \dots\ \frac{dx_n}{dt}\right]^T = \begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \\ \vdots \\ \frac{dx_n}{dt} \end{bmatrix},$$

and

$$\mathbf{f} = [f_1\ f_2\ \dots\ f_n]^T = \begin{bmatrix} f_1(x_1, x_2, \dots, x_n, t) \\ f_2(x_1, x_2, \dots, x_n, t) \\ \vdots \\ f_n(x_1, x_2, \dots, x_n, t) \end{bmatrix},$$

one has to write a function that computes $f_1, f_2 \dots, f_n$, given the input $(\mathbf{x}, t)$, where $\mathbf{x}$ is the column vector $\mathbf{x} = [x_1\ x_2\ \dots\ x_n]^T$. Your function must return the state derivative $\dot{\mathbf{x}}$ as a column vector.

Here is the syntax for `ode23` (for `ode45`, just replace `ode23` with `ode45`):

```
(time, solution) = ode23('your_function', tspan, x0),
```

where `tspan`= $[t_0, t_{final}]$ and `x0` is the initial condition(s). For a system of $n$ equations, the output `solution` contains $n$ columns. Note which column corresponds to which variable in order to extract the correct column.

**Example 1.** For the first order differential equation,

$$\frac{dx}{dt} = x^2 - \frac{t^2}{t^4+1}, \quad \text{with the initial condition } x(-2) = 0.55, \tag{5}$$

create the following function file:

```
function xdot = ode1(t,x);
% ODE1: computes xdot.
xdot = x.^2-t.^2./(t.^4+1);
```

and save it as `ode1.m` in the current working directory of Matlab.

```
>> x0 = 0.55;
>> tspan = [-2,15];
>> [t,x] = ode23('ode1',tspan,x0);
>> plot(t,x)
```

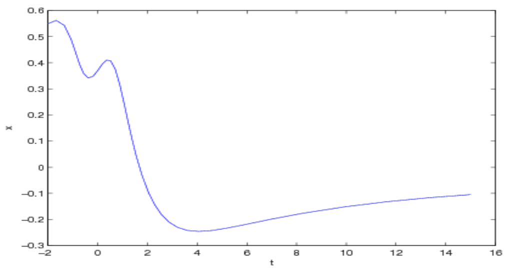


FIGURE 7. Graph of the solution to (5).

**Example 2a.** The one-dimensional motion of an object projected vertically downward (toward the surface of the Earth) in the media offering resistance proportional to the square root of velocity can be modeled by the following differential equation:

$$m\frac{dv}{dt} = mg - k\sqrt{v}, \quad v(0) = v_0 \geq 0, \tag{6}$$

where $v$ is velocity, $m$ is the mass of the object, $g = 9.81\,(m/sec^2)$ is the acceleration due to gravity, $k$ is the proportionality constant, and $v_0$ is the initial velocity. The positive direction of the velocity is in the downward direction.

In this example we are not interested in the positions of the moving object, though you can imagine that the object was thrown vertically downward from a very high building. Computation of the positions is done in Example 2b.

In the calculations below, we choose $m = 1$, the drag constant $k = 9$, and experiment with different (positive) initial velocities.

Create the following function file:

```
function dvdt = ode2(t,v);
% ODE2A: computes dvdt.
m =1; g = 9.81; k = 9;
dvdt = g-(k/m)*sqrt(v);
```

and save it as `ode2a.m` in the current working directory of Matlab.

```
>> v0 = 1;
>> tspan = [0,3];
>> [t,v] = ode23('ode2a',tspan,v0);
>> plot(t,v)
>> hold on
>> v0 =1.5;
>> [t,v] = ode23('ode2a',tspan,v0);
>> plot(t,v,'--')
```

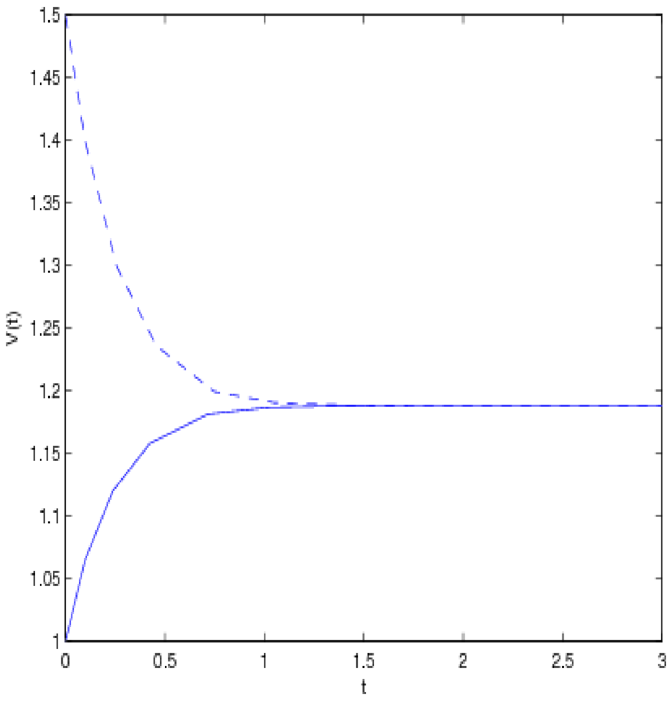


FIGURE 8. Graph of the solutions to (6).

The dashed curve in Figure 8 represents a function decreasing in time and thus, it does not describe the intended physical situation. Indeed, the terminal velocity for the problem described by (6) is given by

$$V_{terminal} = \left(\frac{mg}{k}\right)^2 = \left(\frac{9.81}{9}\right)^2 = 1.1881,$$

corresponding to one of the equilibrium solutions of (6): $v(t) = (mg/k)^2 = 1.1881$. Therefore, for initial velocities greater than 1.1881, as is the case of the initial velocity $v_0 = 1.5$, the typical behavior of such solutions is indicated by the dashed curve in Figure 8.

**Practice exercise.** Write you own function file to solve the problem of an object projected vertically upward in the media offering resistance proportional to the square root of velocity (with $k = 1$, $g = 9.81$, and $m = 1$). As before, choose the positive direction of the velocity in the downward direction and the initial velocity $v_0 = -10$. Additionally, find the time $t_0$ such that $v(t_0) = 0$.

**Note**: *The direction of the drag force always opposes the motion of the object, and thus it is different for upward/downward motions.*

**Example 2b.**

In this example we consider the same physical problem as in Example 2a and, in addition to the velocity, we also compute the positions of the object of mass $m = 1$ kg that is dropped from the height of 100 meters.

$$m\frac{d^2x}{dt^2} = mg - k\sqrt{\frac{dx}{dt}}, \quad x(0) = 0, \quad \frac{dx}{dt}(0) = 0, \tag{7}$$

We assume that $k = 1$ and the positive direction of the velocity and the position is in the downward direction.

Equation (7) can be written as the system of two differential equations for $z_1 = x$ and $z_2 = dx/dt = v$ as follows:

$$\frac{d\mathbf{z}}{dt} = \begin{bmatrix} \frac{dz_1}{dt} \\ \frac{dz_2}{dt} \end{bmatrix} = \begin{bmatrix} z_2 \\ g - \frac{k}{m}\sqrt{z_2} \end{bmatrix}, \qquad \begin{bmatrix} z_1(0) \\ z_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Create the following function file:

```
function dzdt = ode2b(t,z);
% ODE2B: computes dzdt.
g = 9.81; k = 1; m = 1;
dzdt = [z(2); g-(k/m)*sqrt(abs(z(2)))];
```

and save it as `ode2b.m` in the current working directory of Matlab.

```
>> z0=[0,0];
>> tspan =[0,10];
>> [t,z] = ode23('ode2b',tspan,z0);
>> x=z(:,1); v=z(:,2);
>> plot(t,x,t,v,'--')
```

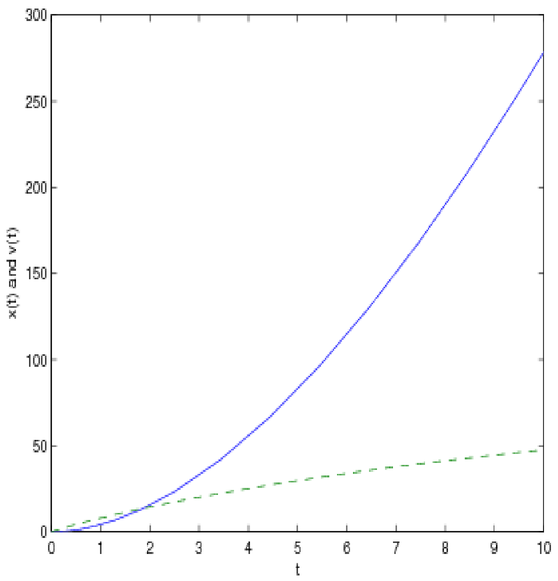


Figure 9. Graph of the solutions to (7): the solid curve represents positions and the dashed curve represents velocities.

Please note that in the commands `x=z(:,1)` and `v=z(:,2)`, $x$ denotes the 1st column (i.e., the position) and $v$ denotes the 2nd column (i.e, the velocity) of the vector $z = [z_1, z_2]$, respectively.

Next, we would like to compute the time needed for the object to reach the ground level. Let us look at the output:

```
>> [t,x]
ans =
         0         0
    0.0000    0.0000
    0.0000    0.0000
    0.0002    0.0000
    0.0007    0.0000
    0.0019    0.0000
    0.0048    0.0001
    0.0113    0.0006
    0.0225    0.0024
    0.0397    0.0075
    0.0676    0.0214
    0.1111    0.0572
    0.1772    0.1432
    0.2750    0.3391
    0.4168    0.7626
    0.6183    1.6368
    0.8994    3.3665
    1.2857    6.6575
    1.8092   12.6973
    2.5101   23.4179
    3.4389   41.8660
    4.4389   66.6306
    5.4389   96.0028
    6.4389  129.5746
    7.4389  166.9978
    8.4389  207.9697
    9.4389  252.2230
   10.0000  278.3993
```

We see that this event occurs between 5.4389 and 6.4389 seconds. In order to determine this time more precisely we use the interpolation from Section 13.2 to build a continuous function that represents the trajectory of the object (we use here $s$ variable to denote time since $t$ is already used as a variable). This can be done with the command `interp1(t,x,s)`. Now, the function $f(s) = 100 - x\_dist$, where `x_dist=interp1(t,x,s)`, is the height of the object as a function of time. Furthermore, with `fzero` command (nonlinear equation solver) from Section 14, we can compute the zero of $f(s)$, which will represent the time needed for the object to reach the ground level.

And here is the actual Matlab session to do the above calculations:

```
>> f = inline('100-interp1(t,x,s)')
f =
     Inline function:
     f(s,t,x) = 100-interp1(t,x,s)
>> fzero(@(s)f(s,t,x),5)
ans =
    5.5580
```

This time can be compared to the time needed by the same object to reach the ground if there is no drag force:

$$t = \sqrt{\frac{2h}{g}} = \sqrt{\frac{2\cdot 100}{9.81}} = 4.5152.$$

**Practice exercise.** Repeat Example 2b (the same mass and height, and with $k = 1$) when the drag force is proportional to $v$ and to $v^2$.

**Example 3.**

The motion of the nonlinear pendulum is

$$\ddot{\theta} + \omega^2 \sin\theta = 0. \tag{8}$$

We consider the initial conditions

$$\theta(0) = 0, \qquad \dot{\theta}(0) = 1.$$

Equation (8) can be written as the system of two differential equations for $\mathbf{z} = [z_1, z_2]$, with $z_1 = \theta$ and $z_2 = \dot{\theta}$, as follows:

$$\frac{d\mathbf{z}}{dt} = \begin{bmatrix} \dfrac{dz_1}{dt} \\ \dfrac{dz_2}{dt} \end{bmatrix} = \begin{bmatrix} z_2 \\ -\omega^2 \sin(z_1) \end{bmatrix}, \qquad \begin{bmatrix} z_1(0) \\ z_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Create the following function file:

```
function dzdt = ode3(t,z);
% ODE3: computes dzdt.
w=1;
dzdt = [z(2); -w^2*sin(z(1))];
```

and save it as `ode3.m` in the current working directory of Matlab.

```
>> z0=[0,1];
>> tspan =[0,20];
>> [t,z] = ode45('ode3',tspan,z0);
>> x=z(:,1); v=z(:,2);
>> plot(t,x,t,v,'--')
>> figure(2)
>> plot(x,v)
```

Please note that in the commands `x=z(:,1); v=z(:,2);` $x$ denotes the 1st column (i.e., the position $\theta$ ) and $v$ denotes the 2nd column (i.e, the velocity $\dot{\theta}$) of the vector $z = [z_1, z_2]$, respectively.

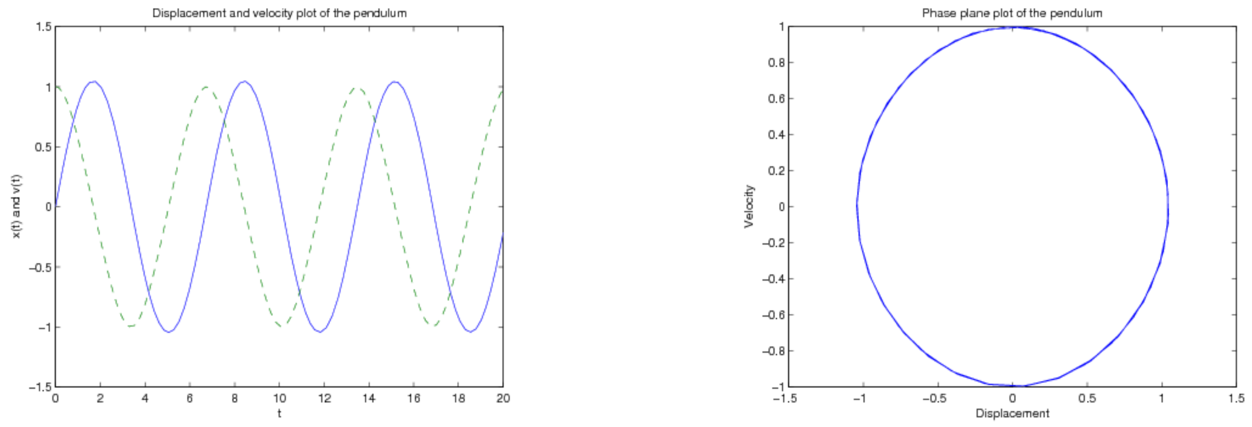


FIGURE 10. On the left: graphs of the solutions to (8). The solid curve represents positions $\theta(t)$ and the dashed curve represents velocities $\dot{\theta}(t)$. On the right: the phase portrait of the pendulum.

16.1. **Solving Linear Systems of First Order Differential Equations.** Consider a system of homogeneous linear first order differential equations

$$\dot{\mathbf{X}} = \mathbf{A}\mathbf{X}, \tag{9}$$

where

$$\mathbf{X} = [x_1\ x_2\ \dots\ x_n]^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \dot{\mathbf{X}} = \begin{bmatrix} \frac{dx_1}{dt} & \frac{dx_2}{dt} & \dots & \frac{dx_n}{dt} \end{bmatrix}^T = \begin{bmatrix} \dfrac{dx_1}{dt} \\ \dfrac{dx_2}{dt} \\ \vdots \\ \dfrac{dx_n}{dt} \end{bmatrix},$$

and $\mathbf{A}$ is $n \times n$ matrix with real entries.

16.1.1. *Distinct real eigenvalues.* In the case $\mathbf{A}$ has $n$ real distinct eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ and the corresponding (linearly independent) eigenvectors $\mathbf{K}_1, \mathbf{K}_2, \dots, \mathbf{K}_n$, the general solution of (9) can be written in the form

$$\mathbf{X}(t) = c_1\mathbf{X}_1(t) + c_2\mathbf{X}_2(t) + \cdots + c_n\mathbf{X}_n(t),$$

where

$$\mathbf{X}_1(t) = \exp(\lambda_1 t)\mathbf{K}_1, \quad \mathbf{X}_2(t) = \exp(\lambda_2 t)\mathbf{K}_2, \quad \dots, \quad \mathbf{X}_n(t) = \exp(\lambda_n t)\mathbf{K}_n\,,$$

and constants $c_1, c_2, \dots, c_n$ are determined from the initial condition at $t = t_0$ on $\mathbf{X}$, i.e., $\mathbf{X}(t_0) = \mathbf{X}_0$, where vector $\mathbf{X}_0$ is provided.

```
>> A=[-4 1 1;1 5 -1;0 1 -3]            >> format rat; [V,D]=eig(A)
                                       V =
A =
      -4          1          1              -129/1048        201/203          985/1393
       1          5         -1              -129/131       -201/2030            *
       0          1         -3              -129/1048        201/2030         985/1393

                                       D =

                                               5               0                 0
                                               0              -4                 0
                                               0               0                -3
```

The columns of matrix $\mathbf{V}$ provide the eigenvectors of $\mathbf{A}$, where $*$ in the last column of $\mathbf{V}$ signifies 0. Eigenvectors are determined up to multiplicative scalars and since $129/131 = 8 \cdot (129/1048)$ and $201/203 = 10 \cdot (201/2030)$, the general solution is

$$\mathbf{X}(t) = c_1 \exp(5t) \begin{bmatrix} 1 \\ 8 \\ 1 \end{bmatrix} + c_2 \exp(-4t) \begin{bmatrix} 10 \\ -1 \\ 1 \end{bmatrix} + c_3 \exp(-3t) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

For the initial condition $\mathbf{X}_0 = (1, 2, 3)^T$, the constants $c_1$, $c_2$, and $c_3$ are given by

```
>> B=[1 10 1;8 -1 0;1 1 1]             >> X0=[1;2;3]                    >> format rat; C=B\X0
B =                                    X0 =                             C =
   1      10       1                           1                             2/9      % c1
   8      -1       0                           2                            -2/9      % c2
   1       1       1                           3                             3        % c3
```

and the solution is

$$\mathbf{X}(t) = (2/9)\exp(5t) \begin{bmatrix} 1 \\ 8 \\ 1 \end{bmatrix} - (2/9)\exp(-4t) \begin{bmatrix} 10 \\ -1 \\ 1 \end{bmatrix} + 3\exp(-3t) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} (2/9)\exp(5t) - (20/9)\exp(-4t) + 3\exp(-3t) \\ (16/9)\exp(5t) + (2/9)\exp(-4t) \\ (2/9)\exp(5t) - (2/9)\exp(-4t) + 3\exp(-3t) \end{bmatrix} \tag{10}$$

The general solution to (9) has also the following form,

$$\mathbf{X}(t) = \exp(t\mathbf{A})\mathbf{C},$$

where $\mathbf{C} = (c_1, c_2, \dots, c_n)^T$ and $\exp(t\mathbf{A})$ is the exponential of the matrix $t\mathbf{A}$. In particular, for

$$\dot{\mathbf{X}} = \mathbf{A}\mathbf{X}, \qquad \mathbf{X}(0) = \mathbf{X}_0\,,$$

the solution is given by

$$\mathbf{X}(t) = \exp(t\mathbf{A})\mathbf{X}_0\,.$$

Using the above example, the following Matlab session (that uses *Symbolic Math Toolbox*)

```
 >> A=[-4 1 1;1 5 -1;0 1 -3]
A =
      -4         1         1
       1         5        -1
       0         1        -3

>> X0=[1;2;3]
X0 =
       1
       2
       3
```

```
>> syms t    % defines symbolic variable t

>> f=inline('expm(t*A)')

f =
     Inline function:
     f(A,t) = expm(t*A)
```

```
>> X=f(A,t)*X0
X =
 3/exp(3*t) - 20/(9*exp(4*t)) + (2*exp(5*t))/9
              2/(9*exp(4*t)) + (16*exp(5*t))/9
  3/exp(3*t) - 2/(9*exp(4*t)) + (2*exp(5*t))/9
```

provides the solution already obtained in (10).

16.1.2. *Repeated real eigenvalues.* If $n \times n$ matrix $\mathbf{A}$ in the system (9) has only $k < n$ linearly independent eigenvectors $\mathbf{K}_1$, $\mathbf{K}_2, \ldots \mathbf{K}_k$ then, we have only $k$ linearly independent solutions of the form $\exp(\lambda_i t)\mathbf{K}_i$. $i = 1, \ldots k$. To find additional solutions we pick an eigenvalue $\lambda$ of $\mathbf{A}$ and find all vectors $\mathbf{v}$ for which $(\mathbf{A} - \lambda\mathbf{I})^2\mathbf{v} = \mathbf{0}$, but $(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} \neq \mathbf{0}$. For such vectors $\mathbf{v}$,

$$\exp(\mathbf{A}t)\mathbf{v} = \exp(\lambda t)\exp\left(\mathbf{A} - \lambda\mathbf{I}\right)t\mathbf{v} = \exp(\lambda t)\left[\mathbf{v} + t(\mathbf{A} - \lambda\mathbf{I})\mathbf{v}\right]$$

is an additional solution of (9).

If we still don't have enough solutions, then we find all vectors $\mathbf{v}$ for which $(\mathbf{A}-\lambda\mathbf{I})^3\mathbf{v} = \mathbf{0}$, but $(\mathbf{A}-\lambda\mathbf{I})^2\mathbf{v} \neq \mathbf{0}$. For such vectors $\mathbf{v}$,

$$\exp(\mathbf{A}t)\mathbf{v} = \exp(\lambda t)\exp\left(\mathbf{A} - \lambda\mathbf{I}\right)t\mathbf{v} = \exp(\lambda t)\left[\mathbf{v} + t(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} + \frac{t^2}{2!}(\mathbf{A} - \lambda\mathbf{I})^2\mathbf{v}\right]$$

is an additional solution of (9). We keep proceeding in this manner until we obtain $n$ linearly independent solutions. The result from linear algebra guarantees that this algorithm works, Indeed, if $n \times n$ matrix $\mathbf{A}$ has $k$ distinct eigenvalues $\lambda_1, \lambda_2 \ldots, \lambda_k$ with multiplicity $n_1 \ldots, n_k$, then for each $j = 1, \ldots k$, there exists $d_j \leq n_j$ such that the equation $(\mathbf{A} - \lambda\mathbf{I})^{d_j}\mathbf{v} = \mathbf{0}$ has at least $n_j$ linearly independent solutions. Thus, for each eigenvalue $\lambda_j$ of $\mathbf{A}$, we can compute $n_j$ linearly solutions of (9). All these solutions have the form

$$\mathbf{X}(t) = \exp(\lambda_j t)\left[\mathbf{v} + t(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} + \ldots + \frac{t^{d_j-1}}{(d_j-1)!}(\mathbf{A} - \lambda\mathbf{I})^{d_j-1}\mathbf{v}\right].$$

Furthermore, $n_1 + n_2 + \ldots n_k = n$ and such obtained solutions must be linearly independent. We know from Section 12.7 that vectors $\mathbf{v}$ satisfying $(\mathbf{A} - \lambda\mathbf{I})^{d_j}\mathbf{v} = \mathbf{0}$ are the generalized eigenvectors of $\mathbf{A}$.

Consider the system (9) with $\mathbf{A}$ given by

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 & -1 \\ 1 & -5 & 7 & -2 \\ 2 & -9 & 10 & -2 \\ 2 & -8 & 6 & 1 \end{bmatrix}$$

The command `[P,J]=jordan(A)` computes both $\mathbf{J}$, the Jordan canonical form, and the similarity transform, $\mathbf{P}$, whose columns are the generalized eigenvectors.

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

```
>> format rat; [P,J]=jordan(A)
P =
     4   -1   3   -3
     4   -2   5   -4
     4   -3   6   -4
     4   -4   6   -4

J =
     1    0   0    0
     0    2   1    0
     0    0   2    1
     0    0   0    2
```

Let us denote the columns of $\mathbf{P}$ by $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$, $\mathbf{v_4}$. We have,

$$(\mathbf{A}-\mathbf{I})\mathbf{v_1}=\mathbf{0},\ (\mathbf{A}-2\mathbf{I})\mathbf{v_2}=\mathbf{0},\ (\mathbf{A}-2\mathbf{I})\mathbf{v_3}=\mathbf{v_2},\ (\mathbf{A}-2\mathbf{I})\mathbf{v_4}=\mathbf{v_3},\ (\mathbf{A}-2\mathbf{I})^2\mathbf{v_3}=\mathbf{0},\ (\mathbf{A}-2\mathbf{I})^3\mathbf{v_4}=\mathbf{0},$$

as also verified by the following Matlab commands:

```
>> (A-eye(4))*P(:,1)

ans =
       0
       0
       0
       0

>> (A-2*eye(4))*P(:,2)

ans =
       0
       0
       0
       0

>> (A-2*eye(4))^2*P(:,3)

ans =
       0
       0
       0
       0
```

```
>> (A-2*eye(4))*P(:,3)

ans =
      -1
      -2
      -3
      -4

>> (A-2*eye(4))*P(:,4)

ans =
       3
       5
       6
       6

>> (A-2*eye(4))^3*P(:,4)

ans =
       0
       0
       0
       0
```

The solution of (9) corresponding to $\lambda = 1$ (eigenvalue of multiplicity 1) is

$$\mathbf{X}_1(t) = \exp(t)\mathbf{v_1} = \exp(t)\begin{bmatrix}4\\4\\4\\4\end{bmatrix}.$$

Three linearly independent solutions of (9) corresponding to $\lambda = 2$ (eigenvalue of multiplicity 3) are

$$\mathbf{X}_2(t) = \exp(2t)\mathbf{v_1} = \exp(2t)\begin{bmatrix}-1\\-2\\-3\\-4\end{bmatrix},\ \mathbf{X}_3(t) = \exp(2t)\left[\mathbf{I}+t(\mathbf{A}-2\mathbf{I})\right]\mathbf{v_3} = \exp(2t)\left[\mathbf{v_3}+t\mathbf{v_2}\right] = \exp(2t)\begin{bmatrix}3-t\\5-2t\\6-3t\\6-4t\end{bmatrix},$$

and

$$\mathbf{X}_4(t) = \exp(2t)\left[\mathbf{I} + t(\mathbf{A}-2\mathbf{I}) + \frac{t^2}{2!}(\mathbf{A}-2\mathbf{I})^2\right]\mathbf{v_4} = \exp(2t)\left[\mathbf{v_4} + t\mathbf{v_3} + \frac{t^2}{2!}\mathbf{v_2}\right] = \exp(2t)\begin{bmatrix} 3t - t^2/2 - 3 \\ 5t - t^2 - 4 \\ 6t - (3/2)t^2 - 4 \\ 6t - 2t^2 - 4 \end{bmatrix}$$

and the general solution is

$$\mathbf{X}(t) = c_1\mathbf{X}_1(t) + c_2\mathbf{X}_2(t) + c_3\mathbf{X}_3(t) + c_4\mathbf{X}_4(t)$$

For the initial condition $\mathbf{X}_0 = (1,0,0,0)^T$, the constants $c_1$, $c_2$, $c_3$, and $c_4$ are solutions of the system:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{X}_0 = c_1\mathbf{v_1} + c_2\mathbf{v_2} + c_3\mathbf{v_3} + c_4\mathbf{v_4} = \begin{bmatrix} 4 & -1 & 3 & -3 \\ 4 & -2 & 5 & -4 \\ 4 & -3 & 6 & -4 \\ 4 & -4 & 6 & -4 \end{bmatrix}\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$

Using Matlab, we obtain

```
>> X0=[1;0;0;0]

X0 =
       1
       0
       0
       0
```

```
 >> format rat; C=P\X0

C =
      1       % c1
      0       % c2
      0       % c3
      1       % c4
```

and the solution of (9) with the initial condition $\mathbf{X}_0 = (1,0,0,0)^T$ is

$$\mathbf{X}(t) = \mathbf{X}_1(t) + \mathbf{X}_4(t) = \begin{bmatrix} 4\exp(t) + (3t - t^2/2 - 3)\exp(2t) \\ 4\exp(t) + (5t - t^2 - 4)\exp(2t) \\ 4\exp(t) + (6t - (3/2)t^2 - 4)\exp(2t) \\ 4\exp(t) + (6t - 2t^2 - 4)\exp(2t) \end{bmatrix}. \tag{11}$$

Finally, the following Matlab session (that uses *Symbolic Math Toolbox*)

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

>> X0=[1;0;0;0]
X0 =
       1
       0
       0
       0
```

```
>> syms t   % defines symbolic variable t

>> f=inline('expm(t*A)')

f =
     Inline function:
     f(A,t) = expm(t*A)
```

```
>> X=f(A,t)*X0

 X =
   4*exp(t) - 3*exp(2*t) + 3*t*exp(2*t) - (t^2*exp(2*t))/2
       4*exp(t) - 4*exp(2*t) + 5*t*exp(2*t) - t^2*exp(2*t)
 4*exp(t) - 4*exp(2*t) + 6*t*exp(2*t) - (3*t^2*exp(2*t))/2
     4*exp(t) - 4*exp(2*t) + 6*t*exp(2*t) - 2*t^2*exp(2*t)
```

verifies the solution already obtained in (11).

16.1.3. *Complex eigenvalues.* If $\lambda = \alpha + i\beta$ is a complex eigenvalue of multiplicity 1 of the coefficient matrix $\mathbf{A}$ (with real entries) and $\mathbf{K}$ is the corresponding eigenvector, then

$$\exp(\lambda t)\mathbf{K} \qquad \text{and} \qquad \exp(\bar{\lambda} t)\overline{\mathbf{K}},$$

are complex-valued solutions of (9), where $\bar{\lambda} = \alpha - i\beta$ and $\overline{\mathbf{K}}$ is the conjugate vector of $\mathbf{K}$. Furthermore, if

$$\mathbf{B}_1 = \frac{1}{2}(\mathbf{K} + \overline{\mathbf{K}}) \qquad \text{and} \qquad \mathbf{B}_2 = \frac{i}{2}(\overline{\mathbf{K}} - \mathbf{K}),$$

then

$$\mathbf{X}_1(t) = [\mathbf{B}_1 \cos(\beta t) - \mathbf{B}_2 \sin(\beta t)] \exp(\alpha t)$$
$$\mathbf{X}_2(t) = [\mathbf{B}_2 \cos(\beta t) + \mathbf{B}_1 \sin(\beta t)] \exp(\alpha t)$$

are linearly independent real-valued solutions of (9) corresponding to the eigenvalue $\lambda$ of multiplicity 1.

Consider (9) with $\mathbf{A} = \begin{bmatrix} 4 & -5 \\ 5 & -4 \end{bmatrix}$. After using `[P,J]=jordan(A)` command,

```
>> A=[4 -5;5 -4]

A =
       4     -5
       5     -4

>> format rat; [P,J]=jordan(A)

P =
       1/2 + 2/3i      1/2 - 2/3i
       0   + 5/6i       0  - 5/6i

J =
         0 - 3i             0
           0              0 + 3i
```

```
>> B1=(1/2)*(P(:,1)+conj(P(:,1)))

B1 =
       1/2
       0

>> B2=(i/2)*(conj(P(:,1))-P(:,1))

B2 =
       2/3
       5/6
```

we obtain

$$\mathbf{X}_1(t) = \mathbf{B}_1 \cos(3t) + \mathbf{B}_2 \sin(3t) = \begin{bmatrix} \frac{1}{2}\cos(3t) + \frac{2}{3}\sin(3t) \\ \frac{5}{6}\sin(3t) \end{bmatrix}$$

$$\mathbf{X}_2(t) = \mathbf{B}_2 \cos(3t) - \mathbf{B}_1 \sin(3t) = \begin{bmatrix} \frac{2}{3}\cos(3t) - \frac{1}{2}\sin(3t) \\ \frac{5}{6}\cos(3t) \end{bmatrix}$$

The general solution is a linear combination of $\mathbf{X}_1$ and $\mathbf{X}_2$ in the form

$$\mathbf{X}(t) = c_1 \begin{bmatrix} \frac{1}{2}\cos(3t) + \frac{2}{3}\sin(3t) \\ \frac{5}{6}\sin(3t) \end{bmatrix} + c_2 \begin{bmatrix} \frac{2}{3}\cos(3t) - \frac{1}{2}\sin(3t) \\ \frac{5}{6}\cos(3t) \end{bmatrix}$$

With $X_0 = (1, 0)^T$, $c_1 = 2$ and $c_2 = 0$ and the solution is

$$\mathbf{X}(t) = \begin{bmatrix} \cos(3t) + \frac{4}{3}\sin(3t) \\ \frac{5}{3}\sin(3t) \end{bmatrix}.$$

One can easily plot phase portrait of the solutions using Matlab `expm(A)` and `for` loop commands:

```
>> X=[]; Y=[]; Z=[];
>> for t=0:.01:10
X=[X expm(t*A)*[1;0]]; Y=[Y expm(t*A)*[1;3]]; Z=[Z expm(t*A)*[10;-2]];
end
>> plot(X(1,:),X(2,:),'r',Y(1,:),Y(2,:),'b',Z(1,:),Z(2,:),'g')
```

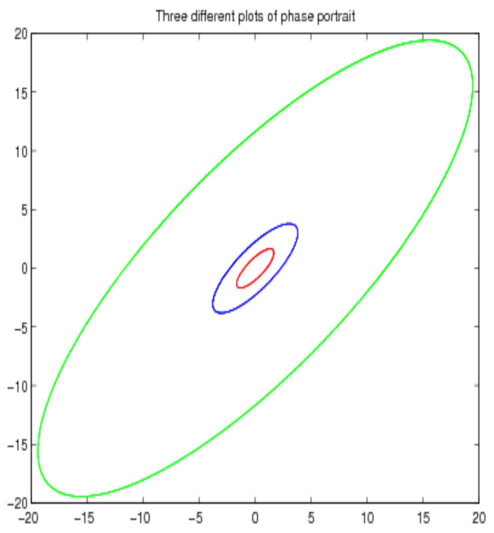


FIGURE 11. Three different plots of phase portrait. Red curve corresponds to the initial condition $X_0 = (1,0)^T$, blue curve corresponds to $X_0 = (1,3)^T$, and green curve corresponds to $X_0 = (10,-2)^T$.

And here is another example of use of `expm(A)` and three dimensional phase plot (`plot3`) commands:

```
>> A = [0 -6 -1;6 2 -16;-5 20 -10]
 A =
      0        -6      -1
      6         2     -16
     -5        20     -10

>> X=[];
for t=0:.01:1
X=[X expm(t*A)*[1;1;1]];
end

>> plot3(X(1,:),X(2,:),X(3,:),'o')
```

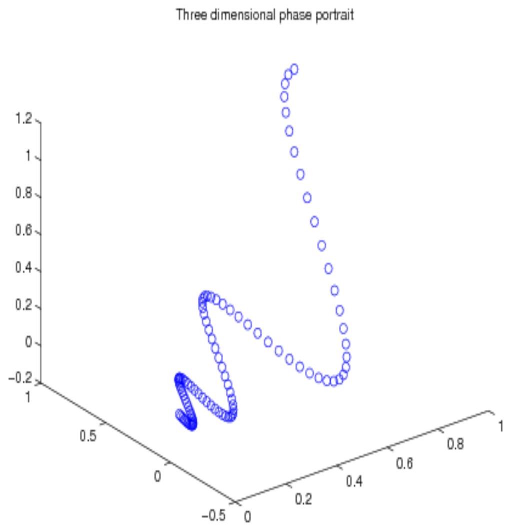


FIGURE 12. Three-dimensional phase portrait shows the solutions (with the initial condition $X_0 = (1,1,1)^T$) spiraling in toward the origin.

16.1.4. *Nonhomogeneous systems.* The solution of the nohomogeneous system

$$\dot{\mathbf{X}} = \mathbf{A}\mathbf{X} + \mathbf{f}, \qquad \mathbf{X}(t_0) = \mathbf{X}_0, \tag{12}$$

is given by the variation of parameters formula,

$$\mathbf{X}(t) = \exp\left[(t-t_0)\mathbf{A}\right]\mathbf{X}_0 + \int\limits_{t_0}^{t} \exp\left[(t-s)\mathbf{A}\right]\mathbf{f}(s)\,ds. \qquad t \geq t_0. \tag{13}$$

Here, $\mathbf{f}(t)$ is a given vector valued function of $t$.

Although the method of variation of parameters is often a time consuming process if done by hand, with Matlab, it can be easily performed.

Consider the nonhomogeneous system,

$$\dot{\mathbf{X}} = \begin{bmatrix} 2 & -2 & 2 & 1 \\ -1 & 3 & 0 & 3 \\ 0 & 0 & 4 & -2 \\ 0 & 0 & 2 & -1 \end{bmatrix}\mathbf{X} + \begin{bmatrix} t\exp(t) \\ \exp(-t) \\ \exp(2t) \\ 1 \end{bmatrix}, \qquad \mathbf{X}(0) = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}. \tag{14}$$

First, we find the general solution of (14).

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

A =
       2       -2        2        1
      -1        3        0        3
       0        0        4       -2
       0        0        2       -1
```

```
>> format rat; [P,J]=jordan(A)

P =
       1       -8/3      4        2/3
       2/3      4/3      4/3     -2/3
      -1/6      0        8/3      0
      -1/3      0        4/3      0

J =
       0        0        0        0
       0        1        0        0
       0        0        3        0
       0        0        0        4
```

The eigenvalues are 0, 1, 3, and 4, with the corresponding eigenvectors

$$\mathbf{K}_1 = \begin{bmatrix} 1 \\ 2/3 \\ -1/6 \\ -1/3 \end{bmatrix}, \qquad \mathbf{K}_2 = \begin{bmatrix} -8/3 \\ 4/3 \\ 0 \\ 0 \end{bmatrix}, \qquad \mathbf{K}_3 = \begin{bmatrix} 4 \\ 4/3 \\ 8/3 \\ 4/3 \end{bmatrix}, \qquad \text{and} \qquad \mathbf{K}_4 = \begin{bmatrix} 2/3 \\ -2/3 \\ 0 \\ 0 \end{bmatrix}.$$

A particular solution is given by

$$\mathbf{X}_p(t) = \exp(t\mathbf{A})\int \exp(-t\mathbf{A})\mathbf{f}(t)\,dt.$$

The corresponding Matlab script is,

```
>> syms t
>> Xp = expm(t*A)*int(expm(-t*A),t);     % The output is a very long
                                         % expression and thus initially
                                         % is suppressed.


>> Xp = simplify(Xp)                     % simplify command makes Xp
                                         % much simpler


Xp =
   (t^2*exp(t))/3 - 1/(5*exp(t)) - 5*exp(2*t) - exp(t)/27 - 4*t - (t*exp(t))/9 - 59/12
   exp(t)/27 - 3/(10*exp(t)) - 2*exp(2*t) - (8*t)/3 + (t^2*exp(t))/6 + (t*exp(t))/9 - 95/36
   (2*t)/3 - (3*exp(2*t))/2 + 2/9
   (4*t)/3 - exp(2*t) + 1/9
```

Thus, the general solution of (14) is

$$\mathbf{X}(t) = c_1 \begin{bmatrix} 1 \\ 2/3 \\ -1/6 \\ -1/3 \end{bmatrix} + c_2 \exp(t) \begin{bmatrix} -8/3 \\ 4/3 \\ 0 \\ 0 \end{bmatrix} + c_3 \exp(3t) \begin{bmatrix} 4 \\ 4/3 \\ 8/3 \\ 4/3 \end{bmatrix} + c_4 \exp(4t) \begin{bmatrix} 2/3 \\ -2/3 \\ 0 \\ 0 \end{bmatrix} + \mathbf{X}_p(t),$$

where

$$\mathbf{X}_p(t) = \begin{bmatrix} (t^2/3)\exp(t) - (1/5)\exp(-t) - 5\exp(2t) - (1/27)\exp(t) - 4t - (t/9)\exp(t) - 59/12 \\ (1/27)\exp(t) - (3/10)\exp(-t) - 2\exp(2t) - (8/3)t + (t^2/6)\exp(t) + (t/9)\exp(t) - 95/36 \\ (2/3)t - (3/2)\exp(2t) + 2/9 \\ (4/3)t - \exp(2t) + 1/9 \end{bmatrix}$$

Finally, the following Matlab script computes the solution to the initial value problem (14):

```
>> syms s t
>> X=expm(t*A)*[1;0;1;0]+expm(t*A)*int(expm(-s*A)*[s*exp(s);exp(-s);exp(2*s);1],0,t);

>> X=simplify(X)

X =

(8*exp(3*t))/3 - 1/(5*exp(t)) - 5*exp(2*t) - 4*t - (457*exp(4*t))/540
+ (332*exp(t))/27 + (t^2*exp(t))/3 - (t*exp(t))/9 - 95/12

(8*exp(3*t))/9 - 3/(10*exp(t)) - 2*exp(2*t) - (8*t)/3 + (457*exp(4*t))/540
+ (335*exp(t))/54 + (t^2*exp(t))/6 + (t*exp(t))/9 -167/36

(2*t)/3 - (3*exp(2*t))/2 + (16*exp(3*t))/9 + 13/18

(4*t)/3 - exp(2*t) + (8*exp(3*t))/9 + 10/9
```

The solution is

$$\mathbf{X}(t) = \begin{bmatrix} (8/3)e^{3t} - (1/5)e^{-t} - 5e^{2t} - 4t - (457/540)e^{4t} + (332/27)e^{t} + (t^2/3)e^{t} - (t/9)e^{t} - 95/12 \\ (8/3)e^{3t} - (3/10)e^{-t} - 2e^{2t} - (8/3)t + (457/540)e^{4t} + (335/54)e^{t} + (t^2/6)e^{t} + (t/9)e^{t} - 167/36 \\ (2/3)t - (3/2)e^{2t} + (16/9)e^{3t} + 13/18 \\ (4/3)t - e^{2t} + (8/9)e^{3t} + 10/9 \end{bmatrix}.$$

16.2. **Linearization of a Double Pendulum.** Consider the double-pendulum system consisting of a pendulum attached to a pendulum shown in Figure 13. The system oscillates in a vertical plane under influence of gravity, the mass of each rod is negligible, and there are no dumping forces acting on the system. The displacement $\theta_1$ is measured (in radians) from vertical line extending downward from the pivot of the system. The displacement $\theta_2$ is measured from a vertical line extending downward from the center of mass $m_1$. The positive direction is to the right, the negative direction is to the left. The system of nonlinear differential equations describing the motion is:

$$\begin{aligned} (m_1+m_2)L_1^2\ddot{\theta}_1 + m_2L_1L_2\ddot{\theta}_2\cos(\theta_1-\theta_2) + +m_2L_1L_2(\dot{\theta}_2)^2\sin(\theta_1-\theta_2) + (m_1+m_2)L_1g\sin\theta_1 = 0; \\ m_2L_2^2\ddot{\theta}_2 + m_2L_1L_2\ddot{\theta}_1\cos(\theta_1-\theta_2) - m_2L_1L_2(\dot{\theta}_1)^2\sin(\theta_1-\theta_2) + m_2L_2g\sin\theta_2 = 0. \end{aligned} \tag{15}$$

If the displacements $\theta_1(t)$ and $\theta_2(t)$ are assumed to be small, then $\cos(\theta_1-\theta_2) \approx 1$, $\sin(\theta_1-\theta_2) \approx 0$, $\sin(\theta_1) \approx \theta_1$, $\sin(\theta_2) \approx \theta_2$, and the following linearization of (15) is obtained:

$$\begin{aligned} (m_1+m_2)L_1^2\ddot{\theta}_1 + m_2L_1L_2\ddot{\theta}_2 + (m_1+m_2)L_1g\theta_1 = 0; \\ m_2L_2^2\ddot{\theta}_2 + m_2L_1L_2\ddot{\theta}_1 + m_2L_2g\sin\theta_2 = 0. \end{aligned} \tag{16}$$

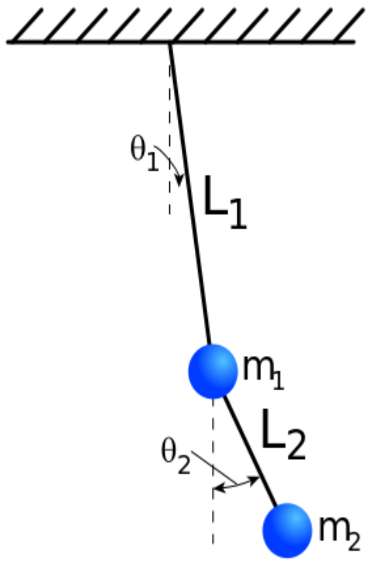


FIGURE 13. Double pendulum

Consider (16) with $m_1 = 3$, $m_2 = 1$, $L_1 = L_2 = 16$, $g = 32$, and the initial conditions: $\theta_1(0) = 1$, $\theta_2(0) = -1$, $\dot\theta_1(0) = 0$, and $\dot\theta_2(0) = 0$. We obtain the following system:

$$\begin{aligned} 4\ddot\theta_1 + \ddot\theta_2 + 8\theta_1 &= 0;\\ \ddot\theta_1 + \ddot\theta_2 + 2\theta_2 &= 0. \end{aligned} \tag{17}$$

Finally, with the $\mathbf{z} = (z_1, z_2, z_3, z_4)^T$ and $z_1 = \theta_1$, $z_2 = \dot\theta_1$, $z_3 = \theta_2$, and $z_4 = \dot\theta_2$, the system of second order differential equations becomes the system of first order differential equations:

$$\dot{\mathbf{z}} = \begin{bmatrix} \dot z_1 \\ \dot z_2 \\ \dot z_3 \\ \dot z_4 \end{bmatrix} = \mathbf{A}\mathbf{z} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -8/3 & 0 & 2/3 & 0 \\ 0 & 0 & 0 & 1 \\ 8/3 & 0 & -8/3 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}, \quad \text{and the initial conditions,} \quad \mathbf{z}(0) = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix}. \tag{18}$$

We compute the eigenvalues of $\mathbf{A}$ by computing the roots of the characteristic polynomial $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$.

```
>> A = [0 1 0 0;-8/3 0 2/3 0;0 0 0 1;8/3 0 -8/3 0]

A =
     0         1          0          0
   -8/3        0         2/3         0
     0         0          0          1
    8/3        0        -8/3         0

>> factor(poly(sym(A)))

ans =

((x^2 + 4)*(3*x^2 + 4))/3
```

The eigenvalues of matrix $\mathbf{A}$ are $\pm 2i$ and $\pm(2/\sqrt{3})i$. We can find the solution $\mathbf{z}(t)$ of (18) by computing $\mathbf{z}(t) = \exp(t\mathbf{A})z_0$. Here is the corresponding Matlab script:

```
>> z0=[1;0;-1;0]

z0 =
       1
       0
      -1
       0
```

```
>> syms t; f=inline('expm(t*A)')

f =

     Inline function:
     f(A,t) = expm(t*A)
```

```
>> z=f(A,t)*z0

z =
1/(8*exp((2*3^(1/2)*i*t)/3)) + exp((2*3^(1/2)*i*t)/3)/8 + 3/(8*exp(2*i*t)) + (3*exp(2*i*t))/8

(3*i*exp(2*i*t))/4 - (3*i)/(4*exp(2*i*t)) - (3^(1/2)*i)/(12*exp((2*3^(1/2)*i*t)/3)) +
(3^(1/2)*i*exp((2*3^(1/2)*i*t)/3))/12

1/(4*exp((2*3^(1/2)*i*t)/3)) + exp((2*3^(1/2)*i*t)/3)/4 - 3/(4*exp(2*i*t)) - (3*exp(2*i*t))/4

(3*i)/(2*exp(2*i*t)) - (3*i*exp(2*i*t))/2 - (3^(1/2)*i)/(6*exp((2*3^(1/2)*it)/3)) +
(3^(1/2)*i*exp((2*3^(1/2)*i*t)/3))/6
```

The first and third rows of `z` provide expressions for $\theta_1(t)$ and $\theta_2(t)$, respectively:

$$\begin{aligned}\theta_1(t) &= \frac{1}{4}\cos\left(\frac{2}{\sqrt{3}}t\right) + \frac{3}{4}\cos(2t)\\ \theta_2(t) &= \frac{1}{2}\cos\left(\frac{2}{\sqrt{3}}t\right) - \frac{3}{2}\cos(2t)\end{aligned} \tag{19}$$

From (19), we observe that the motion of the (linearized) pendulums is not periodic since $\cos(2t/\sqrt{3})$ has period $\sqrt{3}\pi$, $\cos(2t)$ has period $\pi$, and the ratio of these periods is $\sqrt{3}$, which is not a rational number.

The following Matlab script computes the plots of $\theta_1(t)$, $\theta_2(t)$, and the plot of $\theta_2$ versus $\theta_1$ (i.e., the Lissajous curve of compound pendulum).

```
>> A = [0 1 0 0;-8/3 0 2/3 0;0 0 0 1;8/3 0 -8/3 0]

A =
    0       1       0        0
   -8/3     0      2/3       0
    0       0       0        1
    8/3     0     -8/3       0
```

```
>> X=[];
for t=0:.1:30
X=[X expm(t*A)*[1;0;-1;0]];
end
```

```
>> t=0:.1:30;
plot(t,X(1,:))           % plot of theta1
figure(2)
>> plot(t,X(3,:))        % plot of theta2
figure(3)
>> plot(X(1,:),X(3,:))   % plot of X(3,:) versus X(1,:)
```

And finally, the link below provides animation of a double compound nonlinear pendulum showing its chaotic behaviour:

http://www.csun.edu/~hcmth008/matlab_tutorial/double_pendulum.html

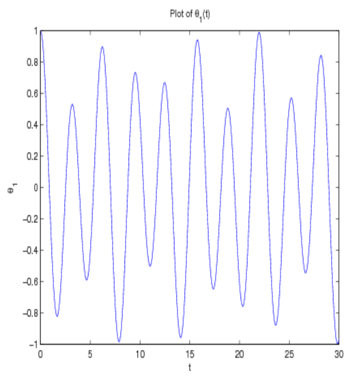


FIGURE 14. Plot of $\theta_1(t)$.

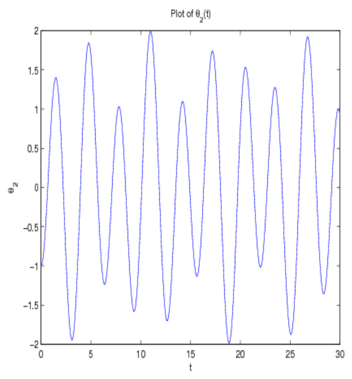


FIGURE 15. Plot of $\theta_2(t)$.

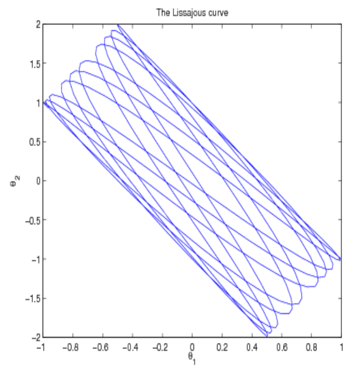


FIGURE 16. The Lissajous curve of compound pendulum.

## 17. Use of Global Variables

Occasionally, it is desirable to declare variables globally. Such variables will be accessible to all or some functions without passing these variables in the input list. The command `global u v w` declares variables `u`, `v`, and `w` globally. This statement must go before any executable in the functions and scripts that will use them. Global declarations of variables should be used with a caution.

Here is one example of a global declaration use. In Example 2a of Section 16, the one-dimensional motion of an object projected vertically downward (toward the surface of the Earth) in the media offering resistance proportional to the square root of velocity was modeled by the equation:

$$m\frac{dv}{dt} = mg - k\sqrt{v}, \quad v(0) = v_0 \geq 0,$$

where $v$ is velocity, $m$ is the mass of the object, $g = 9.81\,(m/sec^2)$ is the acceleration due to gravity, $k$ is the proportionality constant, and $v_0$ is the initial velocity. The positive direction of the velocity is in the downward direction.

The use of global variables in the following function (on the left) and script files

```
function dvdt = ode2a_global(t,v);
% ODE2A_GLOBAL: computes dvdt.
global m_value g_value k_value
dvdt = g_value-(k_value)/(m_value)*sqrt(v);
```

```
% script file to solve 1st order ode.
global m_value g_value k_value
m_value=3; g_value=9.81; k_value=6;
v0 = 2;
tspan = [0,30];
[t,v] = ode23('ode2a_global',tspan,v0);
plot(t,v)
```

allow for changing the values of `m_value`, `g_value`, and `k_value` in the script file without changing those variables in the function file. It's important to note that these global declarations will be available only to the above function and script files.

Save the above function and script file in `ode2a_global.m` and in `ode2a_global_script.m` files, respectively. Remember to place them in the working directory of Matlab and try them out. Alternatively, you can also download them from:

http://www.csun.edu/~hcmth008/matlab_tutorial/ode2a_global.m

and

http://www.csun.edu/~hcmth008/matlab_tutorial/ode2a_global_script.m

## 18. Publishing Documents and Opening Web Pages in Matlab

Matlab (version 7, or greater) includes easy to use automatic report generator. It publishes scripts in various formats: HTML/XML, LaTeX, (MS Word) DOC, and (MS PowerPoint) PPT formats. The last two formats are available only on PC Windows systems with MS Office installed.

Script `my_scrip.m`, placed in the working Matlab directory, can be published with the following command:

```
>> publish('my_script')
```

`publish('my_script')` command creates `my_script.html` file in `html` sub-directory of your current working directory. If the script creates plots/graphics, the plots and graphic images will be also included in `html` sub-directory and automatically linked to `my_script.html` file.

You can view this newly created `my_script.html` file in Matlab's internal browser with the command:

```
>> open html/my_script.html
```

or using the command:

```
>> web('html/ex1_script.html')
```

By the way, the command `web('url')` can be used to display any `url` in Matlab's internal browser, for example,

```
>> web('www.csun.edu')
```

displays the home page of CSUN.

Furthermore, any other browser can also be used to view this file. While viewing `my_script.html` file in Matlab's internal or in any external browser, one can also print the file using `Print` option from the `File` menu.

The command `publish('my_script')` is a default for `publish('my_script','html')` that generates HTML format. The other formats are created by the following commands:

| | |
|---|---|
| `publish('my_script','latex')` | – LaTeX source file `my_script.tex` is generated. If the script creates plots/graphics, the plots and graphic images will be included in `.eps` (encapsulated Postscript format) files. |
| `publish('my_script','xml')` | – XML file `my_script.xml` is generated and can be transformed with XSLT or other tools. |
| `publish('my_script','doc')` | – MS Word file `my_script.doc` is generated. Available only on PC Windows systems with MS Office installed. |
| `publish('my_script','ppt')` | – PowerPoint file `my_script.ppt` is generated. Available only on PC Windows systems with MS Office installed. |

Consider the script from Example 2 of Section 9, where the the Euler method is used to approximate the solution to a given differential equation. The differences between the exact and approximate solutions and the corresponding plots are also provided.

Here is the complete Matlab script:

```
h=0.1;                              % grid size
x=0:h:2;                            % specify the partition of the interval [0,2]
clear y;                            % clear possibly existing variable y
y(1)=1;                             % initial value: x1=0, thus y(1) corresponds to y(x1)=1
f=inline('3*exp(-4*x)-2*y');        % define your function
```

```
size(x);                          % size of x to be used in determining the size of vector i
for i=1:20 y(i+1)=y(i)+h*f(x(i),y(i));end   % compute the approximation
Y_exact=2.5*exp(-2*x)-1.5*exp(-4*x);        % the exact solution
error = abs(Y_exact-y)'                     % the differences between exact
                                            % and approximate solutions
plot(x,y,'--',x,Y_exact)                    % plot of the exact and approximate
                                            % solutions
```

Save the script in `ex1_script.m` file (remember to place it in the working directory of Matlab) and try it out. Alternatively, you can also download it from:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex1_script.m

Once the Matlab script is running and placed in the current working directory of Matlab, you can publish it using the following Matlab command:

```
>> publish('ex1_script')
```

`ex1_script.html` file will be created in `html` sub-directory of your current working directory. If the script creates plots/graphics (as this is the case here), the plots and graphic images will be also included in `html` sub-directory and automatically linked to `ex1_script.html` file.

You can view this newly created `ex1_script.html` file in Matlab's internal browser with the command:

```
>> open html/ex1_script.html
```

or using the command:

```
>> web('html/ex1_script.html')
```

And here is the link to `ex1_script.html` file:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex1_script.html

Any other browser can also be used to view this file. While viewing `ex1_script.html` file in Matlab's internal or in any external browser, you can also print the file using `Print` option from the `File` menu.

It is possible to add further formatting features to `ex1_script.m` for better readability. Furthermore, one can also use LaTeX markup commands to obtain nicely formatted mathematical expressions. And here is an example of such an improved script:

```
%% A simple example of publishing reports
%% The Euler approximation of the differential equation on [0,2]:
% $$y'+2y=3\exp(-4x),\quad y(0)=1.$$
%% Define the grid size, interval's partition, and the initial value
h=0.1;                          % grid size
x=0:h:2;                        % specify the partition of the interval [0,2]
clear y;                        % clear possibly existing variable y
y(1)=1;                         % initial value: x1=0, thus y(1) corresponds to y(x1)=1
%% Define the function
f=inline('3*exp(-4*x)-2*y');
%% Compute the Euler approximation
size(x);                        % size of x to be used in determining the size of vector i
for i=1:20 y(i+1)=y(i)+h*f(x(i),y(i));end
%% The exact solution
Y_exact=2.5*exp(-2*x)-1.5*exp(-4*x);
%% The differences between exact and approximate solutions
error = abs(Y_exact-y)'
%% Plot the exact and approximate solutions
plot(x,y,'--',x,Y_exact)
```

As before, save the script in `ex1_script_improved.m` file (remember to place it in the working directory of Matlab) and try it out. Alternatively, you can also download it from:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex1_script_improved.m

Once the Matlab script is running and placed in the current working directory of Matlab, you can publish it using the following Matlab command:

```
>> publish('ex1_script_improved')
```

`ex1_script_improved.html` file will be created in `html` sub-directory of your current working directory.

You can view this newly created `ex1_script_improved.html` file in Matlab's internal browser with the command:

```
>> open html/ex1_script_improved.html
```

And here is the link to `ex1_script_improved.html` file:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex1_script_improved.html

After typesetting the LaTeX file `publish('ex1_script_improved','latex')`, its `pdf` output can be found here:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex1_script_improved.pdf

The last example publishes the script in Example 3 of Section 16. The script uses an additional function file `ode3.m` that can be downloaded from

http://www.csun.edu/~hcmth008/matlab_tutorial/ode3.m

and placed in the working Matlab directory.

Here is the corresponding Matlab script:

```
%% Nonlinear pendulum
%% The differential equation:
% $$\ddot{\theta}+\omega^2\sin\theta=0,\qquad \theta(0)=0,\qquad \dot{\theta}(0)=1.$$
%% The corresponding system of the first order differential equations:
% $$\frac{dz_1}{dt}=z_2,\,\,\frac{dz_2}{dt}=-\omega^2\sin(z_1),\,\, z_1(0)=0,\,\, z_2(0)=1.$$
%% Input initial conditions:
z0=[0,1];
%% Define the interval on which solution is computed:
tspan =[0,20];
%% Solve the system using |ode45| procedure:
[t,z] = ode45('ode3',tspan,z0);
%% Extract the positions and velocities:
x=z(:,1); v=z(:,2);
%% Plots of the positions and velocities as functions of time:
% *Note:* _The dashed curve indicates velocities_
plot(t,x,t,v,'--')
%% Plot of the phase portrait (velocity as the function of position):
figure(2)
plot(x,v)
```

Save the script in `ex2_script.m` file (remember to place it in the working directory of Matlab) and try it out. Alternatively, you can also download it from:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex2_script.m

Publish it using the Matlab command:

```
>> publish('ex2_script')
```

and view it with the command:

```
>> open html/ex2_script.html
```

And here is the link to `ex2_script.html` file:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex2_script.html

Finally, the output generated by `publish('ex2_script','latex')` and converted to`pdf` file can be found here:

http://www.csun.edu/~hcmth008/matlab_tutorial/solved_problems/ex2_script.pdf

## 19. How to Download the Matlab Files Used in this Tutorial

The Matlab files created and used in this tutorial can be downloaded from:

http://www.csun.edu/~hcmth008/matlab_tutorial/

Department of Mathematics, CSUN

*E-mail address*: jacek.polewczak@csun.edu