Appendix A

MATLAB

MATLAB^{TM} is a powerful, general-purpose numerical program which combines routines for numerical analysis, advanced graphical and visualization capabilities, and a high-level programming language.[1] The name MATLAB stands for *MATrix LABoratory*, and was originally developed to allow access to numerical algorithms developed by the LINPACK and EISPACK projects, which provide state-of-the-art techniques for matrix manipulation and computation.

MATLAB has several internal commands which provide information about MATLAB commands as well as its capabilities:

`help`**:** lists all primary help topics;

`help` *topic***:** gives help on a specific topic;

`helpwin`**:** opens the on-line help within a separate window;

`helpdesk`**:** opens the on-line help within a Web browser;

`lookfor` *keyword***:** searches all MATLAB files for *keyword*;

`demo`**:** runs available demonstrations.

Input commands are terminated with a carriage return. When given an input, MATLAB usually returns with some response. However, this output is suppressed when the input command is followed by a semicolon. In what follows, we do not attempt to give an introduction to MATLAB. Rather we only present a few of the more common commands.

A.1 Basic Mathematics

MATLAB is a powerful calculator. Like any calculator it can add, subtract, multiply, and divide numbers, but on a much larger scale than any hand held calculator.

```
▷ 1 + 2
  ans =
     3
```

[1]The MATLAB commands used in this chapter are valid under version 6, release 12.

```
▷ 12*11
  ans =
     132
▷ x = 144/6;
```

Notice that the last command produced no output. The use of the semi-colon suppresses any MATLAB response. However, the result was calculated and stored in the variable `x`. Thus, we find:

```
▷ x + 6
  ans =
     30
```

The command `help elfun` will provide a listing of the elementary math functions which are known to MATLAB.

In addition to real numbers, MATLAB is equally comfortable with complex quantities. Within MATLAB, the imaginary number $\sqrt{-1}$ is represented as `i`. Thus a complex number may be represented as:

```
▷ y = 1 + 2*i
  y = 1.0000 + 2.0000i
```

and we have all the standard operations on complex numbers:

```
▷ z = 4 - 2*i;
▷ y*z
  ans =
  8.0000 + 6.0000i
```

MATLAB works with arrays, so that the above scalars are actually represented as 1×1 arrays. Likewise, vectors are naturally represented as $1 \times n$ arrays, which are distinct from $n \times 1$ arrays. Arrays are entered into MATLAB with square brackets. Commas (,) or spaces deliminate columns, while semicolons (;) deliminate rows:

```
▷ [1, 3, 5]
  ans =
  1  3  5

▷ [1; 3; 5]
  ans =
  1
  3
  5
```

With this, all the usual array operations are defined:

```
▷ [1, 3, 5]*[1; 3; 5]
  ans =
  35

▷ [1; 3; 5]*[1, 3, 5]
  ans =
   1  3   5
   3  9   15
   5  15  25
```

Notice that because their sizes differ $[1, 3, 5]$ and $[1; 3; 5]$ cannot be added together or subtracted from one another. The size of an array can be determined with `size`.

If two arrays are of identical size, mathematical operations may be performed element-by-element by preceeding the operation ($*$, or /) by a dot, such as:

```
▷ [1, 3, 5; 2, 4, 6].*[1, 3, 5; 1, 1, 1]
  ans =
   1   9   25
   2   4   6

▷ [1, 3, 5; 2, 4, 6]./[1, 3, 5; 1, 1, 1]
  ans =
   1   1   1
   2   4   6
```

Recall that addition and subtraction are normally defined as element-by-element operations on identically sized arrays.

Arrays can also be defined recursively, as:

```
▷ t = 0:0.1:5;
```

which defines `t` as a $1 \times n$ array whose elements differ by 0.1, from 0 to 5.

A.2 Graphics

MATLAB has substantial graphics capabilities in both two- and three-dimensions. For a simple plot, one can define an input array `t` and an output `y` = `sin`(`t`) as:

```
▷ t = 0:0.1:10;
▷ y = sin(t);
```

The result can be plotted with:

```
▷ plot(t,y);
```

There are several ways to plot two or more graphs on a single figure. In the `plot` command, two or more sets of arguments can be prescribed:

```
▷ t = 0:0.1:10;
▷ y = sin(t);
▷ z = cos(t);
▷ plot(t,y,'-r',t,z,'ob')
```

The fields $'-\mathtt{r}'$ and $'\mathtt{ob}'$ specify the line styles for each graph (see `help plot` for more details). Alternatively, the current plot can be overlayed with the `hold` command:

`hold`: holds the current graph. `hold on` holds the current plot so that subsequent graphs are drawn over the existing figure. `hold off` allows for the next plot to erase the current figure.

```
▷ t = 0:0.1:10;
▷ y = sin(t);
▷ plot(t,y);
▷ hold on;
▷ z = cos(t);
```

```
▷ plot(t,z)
```

Commonly used plot attributes include `xlabel`, `ylabel`, `title`, and `legend`. In addition the axis can be scaled with:

`axis`([$\texttt{x}_{\texttt{min}}$ $\texttt{x}_{\texttt{max}}$ $\texttt{y}_{\texttt{min}}$ $\texttt{y}_{\texttt{max}}$])**:** sets the scale for the x- and y-axes on the active plot. With `axis auto` MATLAB automatically scales the figure.

A.3 MATLAB Scripts

Every MATLAB command can be entered on the command line. However, often used commands and collections of commands can be saved and reevaluated using user-defined scripts called *M-files*. MATLAB M-files must end with a "`.m`" extension and are simply evaluated as if typed interactively.

For example, define the script `example01.m` to contain the following commands:

```
t = 0:0.01:10;
x = sin(t);
plot(t,x);
```

When we input the following command to MATLAB:

```
▷ example01
```

we produce a plot of $\sin t$ over the interval $0 \leq t \leq 10$. To evaluate a script, MATLAB must know where to search for the script. The search path is set with the command `path`, while `pathtool` provides a GUI to manipulate the current path.

A.4 MATLAB Functions

MATLAB also provides the ability to create user-defined functions. A function is anything that accepts a number of arguments (possibly zero), performs some user-defined operations, and returns some number of values. Such user-defined functions must be saved in MATLAB *M-files*, and must end with a "`.m`" extension. All user-defined functions must begin with the following line in the `.m` file:

```
function [f1, f2] = function_name(p1, p2, p3)

    function statements
```

where `f1` and `f2` are the results to be returned by the function with name `function_name`, and `p1`, `p2`, `p3` are the input arguments to the function. This then defines the function `function_name`, which takes three arguments and returns two values. The function must then be saved in the file "`function_name.m`". Notice that the name of the file and the name of the function *must be identical*.

For example, if we wish to define the expression:

$$g_1(x) = 5\ e^{-x/10}\ \sin\left(x + \frac{\pi}{3}\right) + 2\ x,$$

we can create the following function:

```
function [g] = gfunct(x)

% The function gfunct(x) takes the argument x and
% returns the user-defined function g_1(x)
```

```
g = 5*exp(-x/10)*sin(x + pi/3) + 2*x
```

This is then saved in the file `gfunct.m`. When this function is run from the MATLAB command window, we find the following:

```
▷ gfunct(0)
  ans =
     4.3301
▷ gfunct(1)
  ans =
     6.0204
▷ gfunct(10)
  ans =
     18.1631
```

MATLAB functions can contain any set of valid commands, including numerical commands (as above), but also graphics commands (e.g. plotting) and programming logic.

A.5 MATLAB Programming

MATLAB has the capability to interpret programming statements, including logic operators and flow control. These can easily be incorporated into `m`-files to dramatically extend the capabilities of MATLAB.

A.5.1 Flow Control

`if`: a set of statements is executed if the corresponding expression evaluates to a non-zero value (true):

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

Note that in this construction, only one set of statements is evaluated (and only once).

`for`: the statements are evaluated as the variable takes on each value in the expression:

```
for variable = expression
     statements
end
```

Note that in this construction, the statements are evaluated once for each element of the expression.

`while`: the statements are evaluated as long as the expression evaluates to a non-zero value (true):

```
while expression
     statements
end
```

Note that in this construction, the number of times that the statements are evaluated is not necessarily predetermined.

In addition, these constructions can be nested (e.g. `for` loops inside `if` statements).

A.6 Linear Algebra

One of MATLAB's most powerful features is its ability to manipulate arrays.

Matrices are entered with semicolons separating rows and commas separating entries within rows.

Example A.6.1 *The matrix A, defined as:*

$$\texttt{A} = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

is inputed by the MATLAB command:

```
▷ A = [1,0.5; 0.5,1];
```

If a matrix A is square $(n \times n)$ and nonsingular, we can determine a second matrix, known as the inverse and denoted as A^{-1}, so that:

$$AA^{-1} = I, \qquad A^{-1}A = I.$$

Further, a square matrix is nonsingular, and therefore has an inverse, if its determinant is nonzero. MATLAB provides functions to numerically calculate both the determinant and the inverse of a square matrix:

`det(A)`**:** calculates the determinant of `A`.

`inv(A)`**:** determines the inverse of `A`. The dimensions of $\texttt{A}^{-1}$ are identical to those of `A`.

Eigenvalues and eigenvectors (known together as eigenpairs) may be determined using the commands:

`eig(A)`**:** determines the eigenvalues of `A`, provided as a column vector.

$[\texttt{Q}, \texttt{d}] =$ `eig(A)`**:** determines a square matrix `Q` containing the eigenvectors of `A` as columns and a diagonal square matrix `d` containing the eigenvalues (λ) of `A`. `Q` is normalized so that $\texttt{Q} * \texttt{Q}'$ is the identity matrix.

Example A.6.2 *For the matrix* `A`*, given above, the eigenvalues and eigenvectors are found by the commands:*

```
▷ A = [1,0.5; 0.5,1];
▷ [Q,d] = eig(A);
```

which provide:

$$\texttt{Q} = \begin{bmatrix} 0.7071 & 0.7071 \\ -0.7071 & 0.7071 \end{bmatrix}, \qquad \texttt{d} = \begin{bmatrix} 0.50 & 0.0 \\ 0.0 & 1.50 \end{bmatrix}$$

A.7 Polynomials

MATLAB represents polynomials as vectors whose elements are precisely the coefficients of the polynomial. Therefore an n-th order polynomial is represented by a vector with $n + 1$ elements.

Example A.7.1 *The polynomial:*

$$f(x) = 4x^5 + 3x^4 + 2x^2 + x + 7,$$

is written in MATLAB as:

```
▷ f = [4  3  0  2  1  7];
```

Note that the cubic term, which is absent in the polynomial is represented by a 0 *in the vector of coefficients. If this term was missing from the MATLAB command,* $f(x)$ *would be interpreted as a* 4*-th order polynomial, rather that a* 5*-th order equation.*

Two polynomials can be added together or subtracted from one another using the usual + and − operations. However, multiplication of two polynomials is equivalent to a *convolution* of two vectors:

`p = conv(a, b)`**:** returns the vector `p` which contains the coefficients of the polynomial obtained when the two polynomials `a` and `b` are multiplied together.

Example A.7.2 *Consider two polynomials:*

$$f(x) = 4\ s^3 + 2\ s + 1, \qquad g(x) = s^2 + 3\ s + 12,$$

which are represented in MATLAB as:

```
▷ f = [4  0  2  1];
▷ g = [0  2  3  12];
```

Notice that `g` *is written with a leading zero so that the size of both* `f` *and* `g` *are the same. Adding* $f(x) + g(x)$ *gives:*

```
▷ f + g
 ans =
    [4  2  5  13]
```

while subtracting $g(x)$ *from* $f(x)$*:*

```
▷ f - g
 ans =
    [4  -2  -1  -11]
```

Finally, multiplying $f(x)$ *and* $g(x)$ *together should yield:*

$$f(x) \cdot g(x) = (4\ s^3 + 2\ s + 1)(2\ s^2 + 3\ s + 12) = 8\ s^5 + 12\ s^4 + 52\ s^3 + 8\ s^2 + 27\ s + 12$$

or in MATLAB:

```
▷ h = conv(f,g);
▷ h
 ans =
    [0  8  12  52  8  27  12]
```

The function `roots` can be used to find the roots of a polynomial:

`r = roots(p)`**:** computes a column vector `r` whose elements are the roots of the polynomial whose coefficients are the elements of the vector `p`.

Example A.7.3 *For the polynomial* $f(x)$ *defined above, its roots may be found to be:*

```
▷ roots(f)
 ans =
    0.1927 + 0.7819i
    0.1927 - 0.7819i
    -0.3855
```

Another convenient MATLAB function is `residue`, which can be used to perform a partial fraction expansion of the two ratio of two polynomials:

$[\mathtt{r}, \mathtt{p}, \mathtt{k}] = \mathtt{residue}(\mathtt{num}, \mathtt{den})$**:** returns the vectors `r`, `p`, and `k` containing the residues, poles and direct terms of a partial fraction expansion of the ratio of two polynomials, represented by `num` and `den`.

Example A.7.4 *For the following ratio of two polynomials:*

$$\frac{3\ s^5 - 20\ s^4 + 10\ s^3 + 157\ s^2 - 269\ s + 199}{s^4 - 6\ s^3 - 2\ s^2 + 58\ s - 51}$$
$$= 3\ s - 2 + \frac{2}{s-1} + \frac{1}{s+3} + \frac{1/2}{s-(4+i)} + \frac{1/2}{s-(4-i)}$$

Using MATLAB, this may be calculated as:

```
▷ num = [ 3  -20  10  157  -269  199];
▷ den = [ 1  -6  -2  58  -51];
▷ [r,p,k] = residue(num,den)
 r =
    0.5000 + 0.0000i
    0.5000 - 0.0000i
    1.0000
    2.0000

 p =
    4.0000 + 1.0000i
    4.0000 - 1.0000i
   -3.0000
    1.0000

 k =
    3  -2
```

A.8 Laplace Analysis

MATLAB contains a number of commands which deal directly with the response of systems represented in transfer function form. Transfer functions can be input with the command `tf`:

$\mathtt{G} = \mathtt{tf}(\mathtt{num}, \mathtt{den})$**:** creates a continuous-time transfer function `G` with numerator `num` and denominator `den`.

With this, the impulse and step response of the system with transfer function `G` can be found with the commands `impulse(G)` and `step(G)`.:

`impulse(G)`**:** plots the impulse response $x(t)$ of the transfer function `G`, that is:

$$X(s) = G(s) \cdot 1.$$

`step(G)`**:** plots the step response $x(t)$ of the transfer function `G`, that is:

$$X(s) = G(s) \cdot \frac{1}{s}.$$

In addition, transfer-function objects (defined using `tf`), can be manipulated directly.

Example A.8.1 *For the system described by the transfer function*

$$T(s) = \frac{D(s)\ G(s)}{1 + D(s)\ G(s)}, \qquad D(s) = 2\left(1 + 0.5\ s\right), \quad G(s) = \frac{2}{3\ s^2 + s + 4}$$

we can input the transfer function $T(s)$ with:

```
▷ D = tf([ 1  2 ], [1]);
▷ G = tf([2], [ 3  1  4 ]);
▷ T = D*G/(1 + D*G)
    Transfer function:
                6s^3 + 14s^2 + 12s + 16
          -------------------------------
           9s^4 + 12s^3 + 39s^2 + 20s + 32
```

Notice that MATLAB does not necessarily simplify the transfer function to its irreducible form. Instead it simply multiplies out all of the individual terms.

The response of the system described by this transfer function can then be generated with the commands:

```
▷ step(T);
▷ impulse(T);
```

to create two seperate plots of the step and impulse response. We can overlay these by using the `hold` *command:*

```
▷ step(T);
▷ hold on;
▷ impulse(T);
```

A.9 Numerical Integration

Numerical techniques for solving ordinary differential equations are based on Taylor series:

$$\begin{aligned} x(t) &= x(t_0) + \left.\frac{dx}{dt}\right|_{t=t_0} (t - t_0) + \left.\frac{d^2x}{dt^2}\right|_{t=t_0} \frac{(t - t_0)^2}{2} + \cdots, \\ &= x(t_0) + \dot{x}(t_0)\ (t - t_0) + \ddot{x}(t_0)\ \frac{(t - t_0)^2}{2} + \cdots, \end{aligned}$$

which, as $(t - t_0) \equiv \Delta t \to 0$, is approximately:

$$x(t) \sim x(t_0) + \dot{x}(t_0)\ \Delta t.$$

Consider the solution to a first-order equation of the form:

$$\dot{x} = -\alpha x.$$

One view of this differential equation is that it provides a rule telling us how the value of x changes, depending on the present value of x. Therefore given x we know $\dot{x}$. However, reflecting on the Taylor expansion above, we find that if we know $x(t_0) = x_0$, that is, x at some initial time, then:

$$\begin{aligned} x(t) = x(t_0 + (t - t_0)) = x(t_0 + \Delta t) &\sim x(t_0) + \dot{x}(t_0)\ \Delta t, \\ &\sim x_0 + (-\alpha x_0)\Delta t. \end{aligned}$$

Thus, given x at some initial time t_0, we can approximately calculate x at some new time $t_0 + \Delta t$. We define this new state as $(t_1, x_1) = (t_0 + \Delta t, x_0 + (-\alpha x_0)\Delta t)$. Repeating this procedure, we find that:

$$x(t_{i+1}) = x_{i+1} \sim x_i + (-\alpha x_i)\ \Delta t.$$

We have derived a simple algorithm for numerically solving a first-order ordinary differential equation, known as Euler's method.

Numerical methods for general ordinary differential equations are based on systems of first-order equations[2]. We can generalize Euler's method to equations of the form:

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

where $\boldsymbol{x}(t)$ is a vector of states, and $\boldsymbol{f}(\boldsymbol{x}, t)$ describes how these states change in time. Therefore, Euler's method can be written as:

$$\boldsymbol{x}(t_{i+1}) = \boldsymbol{x}_{i+1} \sim \boldsymbol{x}_i + \boldsymbol{f}(\boldsymbol{x}_i, t)\ \Delta t.$$

The above method can only consider first-order equations. However, for ODEs of order two and above, we introduce new variables and rewrite the original equations as a system of first-order equations.

Example A.9.1 *The single second-order ordinary differential equation:*

$$\ddot{x} + 4\dot{x} + 12x = \sin t,$$

can be rewritten as a system of two first-order equations through the transformation $y = \dot{x}$. Therefore, $\dot{y} = \ddot{x}$ and so:

$$\begin{aligned} \dot{x} &= y, \\ \dot{y} &= -12x - 4y + \sin t. \end{aligned}$$

If x represents a position, then y is the velocity. Recall, that to determine a specific solution to a second-order equation, we require two auxiliary conditions, usually the initial position and velocity for mechanical systems. Thus we say that the position and velocity are the two independent states of the system. By writing the single second-order equation as two first-order equations, we explicitly determine equations on each state x and y.

In general, systems of n second-order equations of the form:

$$\ddot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t), \qquad \boldsymbol{x} = (x_1, x_2, \ldots, x_n),$$

can be written as $2n$ first-order equations:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{y}, \\ \dot{\boldsymbol{y}} &= \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t). \end{aligned}$$

These first-order equations are in a form suitable for numerical integration with $\boldsymbol{X} = [\boldsymbol{x}, \boldsymbol{y}]^T$, and $\dot{\boldsymbol{X}} = [\boldsymbol{y}, \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}, t)]^T$.

Euler's method, although conceptually simple, is relatively inaccurate and unsuitable for most applications. However MATLAB provides two internal functions `ode23` and `ode45` which may be used to solve systems of first-order ordinary differential equations:

[2]The order of an ordinary differential equation is the order of its highest derivative.

$[\mathtt{t},\mathtt{x}] = \mathtt{ode23}(@\mathtt{deriv},\mathtt{tspan},\mathtt{x}_0)$**:** returns the set $[\mathtt{t},\mathtt{x}]$ that represents $\mathtt{x}(\mathtt{t})$, computed using second- and third-order Runge-Kutta methods. The function `deriv` defines the derivatives of `x` as functions of time and the states `x`. The value `tspan` is an array which contains the initial and final integration times over which we determine $\mathtt{x}(\mathtt{t})$, i.e. $\mathtt{tspan} = [\mathtt{t}_0\ \mathtt{t}_1]$, where $\mathtt{t}_0$ and $\mathtt{t}_1$ specify the initial and final times respectively. $\mathtt{x}_0$ specifies the initial state at $\mathtt{t}_0$.

$[\mathtt{t},\mathtt{x}] = \mathtt{ode45}(@\mathtt{deriv},\mathtt{tspan},\mathtt{x}_0)$**:** has an identical syntax as `ode23`; combines fourth- and fifth-order Runge-Kutta methods.

If we omit the left-hand side of these function calls, that is, input simply $\mathtt{ode45}(\cdot)$ or $\mathtt{ode23}(\cdot)$, MATLAB will provide a plot of the response versus time.

Both of these functions use Runge-Kutta algorithms (not discussed here) to numerically calculate the solution. These techniques are more accurate than Euler's method and are more suitable for general purpose applications. `ode45` is the recommended algorithm of the two. Although for fixed step size `ode23` is faster, it is also less accurate compared to `ode45`. In general, the higher-order methods are more accurate, but require more computational effort. `ode45` is a good compromise between accuracy and efficiency.

Example A.9.2 *Using MATLAB, numerically solve* $\dot{x} = -4x$*, over the time interval* $0 \geq t < 10$*, subject to the initial condition* $x(0) = 7$*.*

Solution:

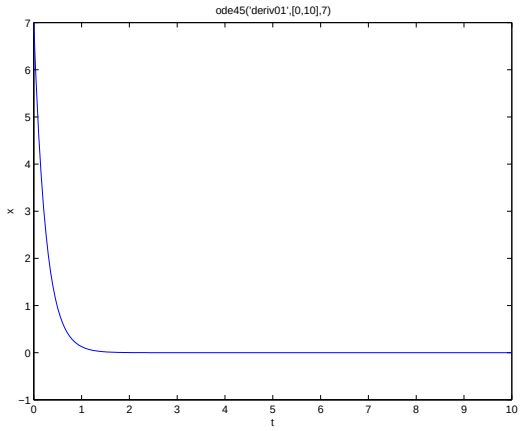
We first define a function `deriv01` *which contains the derivative of* x *with respect to time:*

```
function y = deriv01(t,x)
y = -4*x;
```

This user-defined function must be saved in the file "`deriv01.m`*". Then, we input the following command in MATLAB:*

```
▷ [t,x] = ode45(@deriv01,[0,10],7);
▷ plot(t,x)
▷ xlabel('t')
▷ ylabel('x')
▷ title('ode45(''deriv01'',[0,10],7)')
```

which produces the desired plot of the response versus time.



Alternatively, simply typing the command

```
▷ ode45(@deriv01,[0,10],7)
```

will automatically plot the response x vs. t without the text labels.

For higher-order systems, the above defined variables are one-dimensional arrays.

Example A.9.3 *Write a MATLAB code to solve the following equation:*

$$\ddot{x} + 4\ \dot{x} + 12\ x = \sin(t),$$

subject to the initial conditions:

$$x(0) = 0.10, \qquad \dot{x}(0) = 0.00.$$

Solution:

a) We must write this second-order equation as a system of first order equations. Thus we identify the position and velocity as separate states, e.g., $x = x_1$ and $\dot{x} = x_2$. Therefore $\ddot{x} = \dot{x}_2$ and the system of first order equations becomes:

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= -12x_1 - 4x_2 + \sin t. \end{aligned}$$

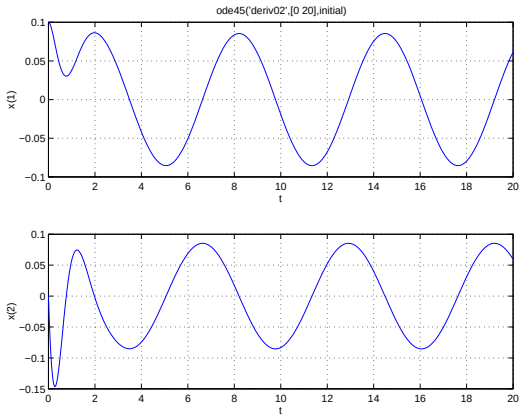
b) We define the function `deriv02`*, containing the derivative of x:*

```
function y = deriv02(t,x)
% we define the vector field below
y(1) = x(2);
y(2) = -12*x(1) - 4*x(2) + sin(t);
% this function must return a column vector
y = [y(1); y(2)];
```

This user-defined function must be saved in the file "`deriv02.m`*". Then, we input the following commands in MATLAB:*

```
▷ initial = [0.1 0];
▷ [t,x] = ode45(@deriv02,[0,20],initial);
▷ subplot(2,1,1), plot(t,x(:,1)),...
  xlabel('t'),ylabel('x'),grid,...
  subplot(2,1,2), plot(t,x(:,2)),...
  xlabel('t'),ylabel('y'),grid
```

where we have chosen $x_1(0) = 0.1$ and $x_2(0) = 0$ as the initial conditions. This produces the following output:



the `subplot` *command has been used to create two separate plots for* x_1 *and* x_2 *vs.* t*. We could however simply input* `plot(t,x)` *or* `plot(t,x(:,1),t,x(:,2))` *to plot both responses on the same graph.*

A.10 Control System Design

MATLAB, through its Control Systems Toolbox, has a number of functions designed for the analysis of control systems. Many of these functions have been included in the Student Edition of MATLAB.

For user-specified open loop transfer function (or equivalent), the root-locus plot may be automatically generated in MATLAB through the command `rlocus`:

$[$`r`$,$`k`$] =$ `rlocus(num,den,m)`**:** returns the set $[$`r`$,$`k`$]$ where `r` are the roots on the closed-loop characteristic equation for the corresponding values of the gains `k`. The open-loop transfer function is defined by the vectors `num` and `den`, which contains the coefficients of the numerator and denominator of the open-loop transfer function. The optional argument `m` allows the user to specify the gain vector.

A.11 Glossary

δ_{ij}**:** the Kronecker delta is defined as:

$$\delta_{ij} = \begin{cases} 0, & i \neq j, \\ 1, & i = j, \end{cases}$$