

A Primer for MATLAB<sup>®</sup> –Tutorial and Documentation  
for Auburn University  
Chemical Engineering Students

Profs. W. Robert Ashurst and T. Placek

Version 1.0

12/15/11

Auburn University  
Chemical Engineering Department  
212 Ross Hall

© 2011, Prof. W. R. Ashurst,  
All Rights Reserved, Worldwide.

# Contents

<b>Preface</b>	<b>i</b>
<b>Revision History</b>	<b>ii</b>
<b>Nomenclature</b>	<b>iii</b>
<b>1 Introduction and Scope</b>	<b>1</b>
<b>2 Solving Algebraic Equations</b>	<b>2</b>
2.1 Solving One Equation for One Real Variable . . . . .	2
2.2 Solving a System of Linear Equations . . . . .	9
2.3 Solving One Equation for One Complex Variable . . . . .	11
2.4 Solving a System of Non-Linear Equations . . . . .	11
2.5 Passing Additional Function Arguments Through <code>fsolve</code> . . . . .	15
2.6 Chapter Summary . . . . .	17
<b>3 Solving Ordinary Differential Equations</b>	<b>19</b>
3.1 Solving One First Order ODE . . . . .	19
3.2 Solving a System of First Order ODEs . . . . .	19
3.3 Solving One Second Order ODE . . . . .	19
3.4 Solving a System of ODEs with Parameters Passed Through <code>ode45</code> . . . . .	19
<b>4 Fitting Data to Nonlinear Model Functions</b>	<b>20</b>
4.1 Model Functions of One Independent Variable . . . . .	20
4.2 Model Functions of More than One Independent Variable . . . . .	20
4.3 Characterizing “Goodness of Fit” . . . . .	20
<b>5 Probability and Statistics</b>	<b>21</b>
5.1 Definitions and Basic Concepts . . . . .	21
5.2 Characteristics of Probability Distributions . . . . .	21
<b>6 Hypothesis Testing</b>	<b>22</b>
6.1 One Sample Tests . . . . .	22
6.2 Two Sample Tests . . . . .	22
<b>7 Miscellaneous Notes</b>	<b>23</b>
7.1 Functions and Function Handles . . . . .	23
7.2 Elements of Style . . . . .	23
<b>Index</b>	<b>24</b>

# Preface

Preface here.

# Revision History

Original document

W. R. Ashurst and T. D. Placek

Date: December 15, 2011.

# Nomenclature

Throughout this document, the following typesetting conventions will be used.

Description	Typesetting Example
A scalar variable (single number)	$x$ (lowercase)
A vector or matrix	$X$ (uppercase)
A MATLAB command	<code>roots</code>
A MATLAB Session	<pre>&gt;&gt; A = [6, 4; 7, 4]; &gt;&gt; B = [-2; 1]; &gt;&gt; A\B</pre>
A MATLAB Function (M-File)	<pre>function result = my_fun(x) result = x + x; end</pre>
MATLAB Output	<pre>ans =     3.0000    -5.0000</pre>

# Chapter 1

## Introduction and Scope

This document is intended to supplement the text “Introduction to MATLAB for Engineers” by William J. Palm, III, in the context of CHEN 3600, Computer Aided Chemical Engineering at Auburn University. This document covers several essential MATLAB functions such as `fsolve`, `ode45` and `nlinfit` which are either not covered or receive minimal coverage in the Palm text. It is assumed that you are using the most current release of MATLAB (Version 7.11.0.584 (R2010b)) as of the writing of this document and have access to the Optimization Toolbox as well as the Statistics Toolbox. These components are part of the Student Edition of MATLAB and are available on all College of Engineering computers.

MATLAB is a versatile and powerful, high level scripting language that has many mathematical operations of use to engineering available as “built-in” functions. Consequently, our department has elected to utilize this modern engineering tool throughout the curriculum. You will encounter the use of MATLAB in other courses, including heavy use in CHEN 3650 and use of the SIMULINK Package in CHEN 4170. It is in your best interest to fully utilize MATLAB.

## Chapter 2

# Solving Algebraic Equations

Throughout your career as an engineer, you will be faced with situations in which you have an equation (or set of equations) and need to determine the value of some quantity (or quantities) that satisfy the given equation(s). This activity is referred to as “solving” the equation(s). Depending on the type of variables, structure of the equation(s), number of equation(s) and sensitivity of the solution, this activity can be tedious, imperiled or even dubious.

Ultimately, you are responsible for the correctness of your solution. You must realize that MATLAB has no sense of scale and is incapable of applying any sort of engineering judgment. The philosophy is that you develop the solution strategy and use MATLAB as a *tool* to carry out your will.

### Professor Ashurst’s Special Note #1:

One should not attempt solving a problem using MATLAB without a well planned approach for a solution prior to beginning typing in MATLAB.

## 2.1 Solving One Equation for One Real Variable

Consider that you are working with the van der Waals equation of state and you need to determine the molar volume for air at a given temperature and pressure. The van der Waals equation is can expressed as shown in Eq. 2.1.

$$\left[ P - \frac{a}{V^2} \right] (V - b) = RT \quad (2.1)$$

If we take  $T$ ,  $P$ , and  $V$  as *variables* and recognize that  $R$  is the gas constant, then Eq. (2.1) is said to be *parameterized* in  $a$  and  $b$ . In our assumed problem scenario, we are operating with a given  $T$  and  $P$ . It is assumed that we would determine  $a$  and  $b$  from literature or reference information. We are therefore in the situation of solving Eq. 2.1 for  $V$ . Specifically, this means that we seek a certain value of  $V$  such that the right hand side of Eq. 2.1 is exactly equal to the left hand side of Eq. 2.1. Since Eq. 2.1 is actually *cubic* in  $V$ , there are (in principle) three values to  $V$  that may satisfy the equation. There are several approaches we may take with MATLAB to identify these solutions. They include (in no particular order):

1. Plotting the equation (first re-casting it into *function* form) and estimating where  $f(V) = 0$
2. Manually guessing  $V$  and checking the right hand side against the left
3. Utilize the `fzero` function
4. Utilize the `fsolve` function

5. Re-cast the Equation into function form, taking its absolute value and utilizing the `fminbnd` function
6. Re-cast the equation into cubic polynomial function form and use the `roots` function

Certainly there are advantages and drawbacks to each of the approaches enumerated. Let us begin our coverages of these approaches with the first one listed. For your parameters  $a$  and  $b$ , the gas  $\text{CO}_2$  is chosen at the conditions of 10 atm and 300 K. For  $\text{CO}_2$ ,  $a = 3.59$  and  $b = 0.0427$  in consistent units. The MATLAB session is as follows, and the plot generated is shown in Fig. 2.1

```
>> a = 3.59      % atm L^2/mol^2
>> b = 0.0427   % L/mol
>> R = 0.082;   % (L atm/K mol),
>> P = 10;      % atm
>> T = 300;     % K
>> vdW = @(V) (P-a./V.^2).*(V-b)-R*T;
>> v = linspace(-1, 3, 300);
>> plot(v, vdW(v), 'k', 'LineWidth', 3)
>> axis([-1, 3, -50, 50]); grid on
>> xlabel('Molar Volume, V');
>> ylabel('f(V)')
```

Note that in the MATLAB session the anonymous function handle `vdW` was created. Also note that this function is a *vectorized* MATLAB function which means that the supplied with a vector argument of length  $n$ , the function returns a vector of length  $n$ , each value being the value of the function at the corresponding value (by index) of the input vector. We will make use of this feature in subsequent approaches.

The three roots for our chosen case are approximately  $V = 2.6, 0.03$  and  $-0.2$  in L/mol. We know that we may discard the negative root since it is physically unrealistic and we treat the

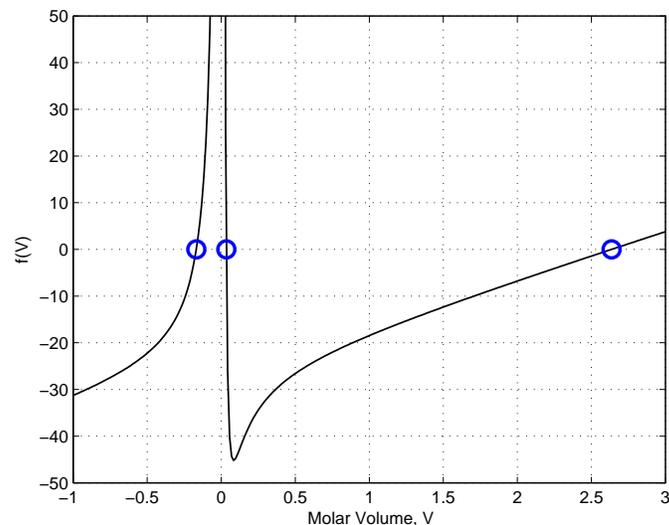


Figure 2.1: Plot of the van der Waals function,  $f(V)$  for  $\text{CO}_2$  at 10 atm and 300 K. The circles indicate the roots of the function, and represent the molar volume values we seek.

smaller magnitude as being associated with the liquid phase molar volume and the larger as that for the gas phase. However, MATLAB will not (and cannot) make this distinction for us and will only assist us in finding the numeric values for these roots.

Let us now utilize the `roots` approach. First we must algebraically re-arrange Eq. 2.1 into a polynomial in  $V$ . The result is shown in Eq. 2.2.

$$PV^3 + (-RT - bP)V^2 - aV + ab = 0 \quad (2.2)$$

Therefore, the session (as a continuation of the previous session) and output is as follows.

```
>> pcoefs = [P, -R*T-b*P, -a, a*b]
>> V_star = roots(pcoefs)

pcoefs =
 10.000000000000000 -25.027000000000001 -3.590000000000000  0.153293000000000
V_star =
 2.636652456997016
-0.168463864799292
 0.034511407802278
```

Note that the elements of the vector `pcoefs` are the coefficients (in polynomial order) from Eq. 2.2.

### Professor Ashurst's Special Note #2:

The `roots` command operates on a vector that holds the coefficients of a *polynomial*. Do not even contemplate the use of the `roots` function unless you are working with a polynomial.

Now let us investigate the use of the MATLAB function functions `fzero`, `fsolve` and `fminbnd`. At this point, the `doc` system information for these functions should be examined. Briefly, the function `fzero` locates a root by identification of a sign change in the value of the function. Therefore, `fzero` is used to find the root(s) of an equation of the form  $f(x) = 0$  where  $x$  represents a single variable.

On the other hand, `fsolve` utilizes more sophisticated numerical methods and can solve the same type of problems as `fzero` as well as solving multiple equations in multiple unknowns. The approach employed by `fminbnd` is similar to `fzero` in that only one variable will be adjusted, and the determination of a minimum in the absolute value of the function will almost always be co-located with a root.

Each of these three methods require an initial guess for the root that should be near the root. In the case of `fminbnd`, the initial guess must be a range that contains the root or else `fminbnd` will fail to identify the root.

In the session that follows (which is again a continuation of the previous session), the three methods will be invoked for each root.

```
>> % Look for root at about V = 2.6 with all three 'function' function methods
>> V1_star_fzero = fzero(vdw, 2.6)
>> V1_star_fsolve = fsolve(vdw, 2.6)
>> avdw = @(V) abs(vdw(V))
>> V1_star_fminbnd = fminbnd(avdw, 2.5, 2.7)

>> % Look for root at about V = 0.03 with all three 'function' function methods
```

```
>> V2_star_fzero = fzero(vdw, 0.03)
>> V2_star_fsolve = fsolve(vdw, 0.03)
>> V2_star_fminbnd = fminbnd(avdw, 0.02, 0.04)

>> % Look for root at about V = -0.2 with all three 'function' function methods
>> V3_star_fzero = fzero(vdw, -0.2)
>> V3_star_fsolve = fsolve(vdw, -0.2)
>> V3_star_fminbnd = fminbnd(avdw, -0.4, -0.1)
```

```
V1_star_fzero =
    2.636652456997014
```

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

```
V1_star_fsolve =
    2.636652456986320
avdw =
    @(V)abs(vdw(V))
V1_star_fminbnd =
    2.636665870993662
V2_star_fzero =
    0.034511407802278
```

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

```
V2_star_fsolve =
    0.034511407802278
V2_star_fminbnd =
    0.034495689933276
V3_star_fzero =
   -0.168463864799292
```

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

```
V3_star_fsolve =
   -0.168463864797615
V3_star_fminbnd =
   -0.168471663591385
```

Note that `fsolve` by default reports a healthy bit of text information to the display. One can (and often should) suppress this output by setting the option flag `Display` to `off` by using the `optimset` function. Further, since we have a vectorized function, we may use `fsolve` to find all three values simultaneously. The following session utilizes the options structure and locates all three roots with one call to `fsolve`.

```
>> opts = optimset('Display','off');
>> vguess = [-0.2, 0.03, 2.6];
>> V_star_fsolve = fsolve(vdw, vguess, opts)

V_star_fsolve =
    -0.168463864799292    0.034511407802278    2.636652456997014
```

This approach works because the initial guess passed into `fsolve` is a length 3 vector. This causes the anonymous function `vdw` to return a length 3 vector. `fsolve` then independently varies each *element* of the initial guess vector until the function returns a length 3 vector of zeros. This is the same, mathematically, as writing down the van der Waals equation of state three times with a different symbol for  $V$ , and then solving the *system* of equations (3 equations, 3 unknowns, uncoupled system). The functions `fzero` and `fminbnd` are not capable of solving systems of equations or for finding more than one root at a time.

It is of interest to note that the selection of the initial guess then using `fsolve` can have awkward consequences. For example, suppose that we were seeking the minimum root, and so we make an initial guess of  $V = -1$ . It is left to the reader to verify that `fsolve` produces the result  $V = 2.63665$ , the upper root. Figure 2.2 illustrates the effect of the initial guess on the returned result of `fsolve`. It is noteworthy that the most probable solution seems to be the upper root, while the root that has the narrowest initial guess window seems to be the root closest to zero, and this root is found with the selection of two ranges of initial guesses.

For another (simpler) example of the use of `fsolve` we will consider the function  $f(x) = x^2$ . Clearly, this function has one repeated root, namely  $x = 0$ , and the root is real. Let us apply the commands `fsolve`, `fzero`, `fminbnd` and `roots`. We will pretend that we do not know the root in advance, and make an initial guess around  $x = 1$ . The session and output follow.

```
>> clear all; clc;
>> opts = optimset('Display','off');
>> the_func = @(x) x.^2;
>> root_fsolve = fsolve(the_func, 1, opts)
>> root_fzero = fzero(the_func, 1, opts)
>> root_fminbnd = fminbnd(the_func, -1, 1, opts)
>> root_roots = roots([1, 0, 0])

root_fsolve =
    0.007812507392371
root_fzero =
    NaN
root_fminbnd =
    -2.775557561562891e-017
root_roots =
     0
     0
```

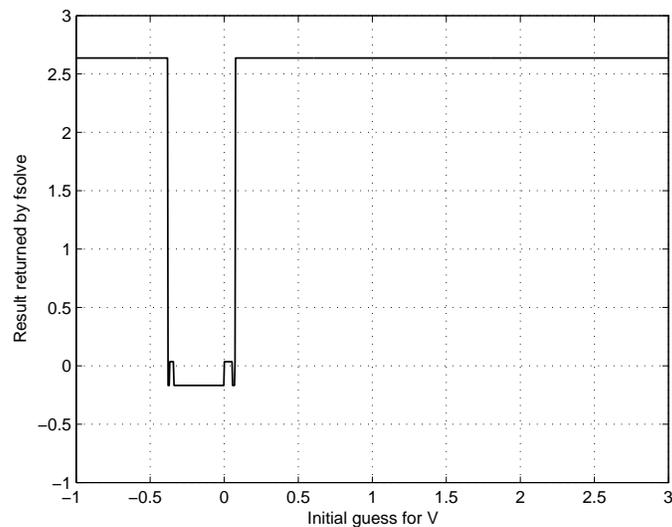


Figure 2.2: Mapping of the result returned by `fsolve` operating on Eq. (2.1) using initial guesses for the root between -1 and 3.

Note that, as expected, the `roots` function returns the exact result, correct with repeated roots. Also note that the function that returns the value closest to zero is `fminbnd`. Furthermore, `fsolve` returns a number that is only about eight thousandths away from zero, while `fzero` fails. It is not unexpected that `fzero` would fail, because this function detects roots by identification of a sign change in the function value. Since the base quadratic never changes sign, (always positive) `fzero` can never find the root except for the special case where the initial guess is close enough to the root that the convergence criteria are satisfied on the initial pass.

Now, one might say the `fsolve` has done a poor job of finding the root. This may be because the root is so obvious and that zero is a special number to people. The error of about eight thousandths would probably not stand out if the root were a number like 4.5382 (say versus 4.5301 or 4.5462). Regardless, we can control the accuracy of the function functions by the use of other parameters in `optimset`. Specifically `TolFun` and `TolX` are useful. Consider the session (and output) that follows.

```
>> clear all; clc;
>> opts = optimset('Display','iter', 'TolFun', 1e-10 , 'TolX', 1e-10);
>> the_func = @(x) x.^2;
>> root_fsolve = fsolve(the_func, 1, opts)
>> root_fminbnd = fminbnd(the_func, -1, 1, opts)
```

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	2	1		2	1
1	4	0.0625	0.5	0.25	1
2	6	0.00390625	0.25	0.0313	1.25
3	8	0.000244141	0.125	0.00391	1.25
4	10	1.52588e-005	0.0625	0.000488	1.25
5	12	9.53675e-007	0.03125	6.1e-005	1.25
6	14	5.96048e-008	0.015625	7.63e-006	1.25

7	16	3.7253e-009	0.0078125	9.54e-007	1.25
8	18	2.32832e-010	0.00390625	1.19e-007	1.25
9	20	1.45521e-011	0.00195312	1.49e-008	1.25
10	22	9.09522e-013	0.000976562	1.86e-009	1.25
11	24	5.68469e-014	0.000488281	2.33e-010	1.25
12	26	3.55315e-015	0.000244141	2.91e-011	1.25

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the selected value of the function tolerance, and the problem appears regular as measured by the gradient.

```
root_fsolve =
    2.441480736870023e-004
```

Func-count	x	f(x)	Procedure
1	-0.236068	0.0557281	initial
2	0.236068	0.0557281	golden
3	0.527864	0.27864	golden
4	-2.77556e-017	7.70372e-034	parabolic
5	3.33333e-011	1.11111e-021	parabolic
6	-3.33334e-011	1.11111e-021	parabolic

Optimization terminated:  
the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-010

```
root_fminbnd =
    -2.775557561562891e-017
```

```
>> clear all; clc;
>> opts = optimset('Display','iter', 'TolFun', 1e-10 , ...
    'TolX', 1e-10, 'MaxIter', 50000, 'MaxFunEvals', 200000, ...
    'Algorithm', 'Levenberg-Marquardt');
>> the_func = @(x) x.^2;
>> root_fsolve = fsolve(the_func, 1, opts)
>> root_fminbnd = fminbnd(the_func, -1, 1, opts)
```

Iteration	Func-count	Residual	First-Order optimality	Lambda	Norm of step
0	2	1	2	0.01	
1	4	0.0631258	0.252	0.001	0.498753
2	6	0.00396107	0.0316	0.0001	0.250374
3	8	0.00024796	0.00395	1e-005	0.125386
4	10	1.55074e-005	0.000494	1e-006	0.0627331
5	12	9.69458e-007	6.18e-005	1e-007	0.0313745
6	14	6.05973e-008	7.72e-006	1e-008	0.0156888
7	16	3.78749e-009	9.66e-007	1e-009	0.00784474
8	18	2.36723e-010	1.21e-007	1e-010	0.00392244
9	20	1.47954e-011	1.51e-008	1e-011	0.00196123
10	22	9.24729e-013	1.89e-009	1e-012	0.000980618
11	24	5.77974e-014	2.36e-010	1e-013	0.000490309
12	26	3.61256e-015	2.95e-011	1e-014	0.000245155
13	28	2.25812e-016	3.68e-012	1e-015	0.000122577

```

14          30    1.41167e-017    4.61e-013  2.22045e-016    6.12887e-005
15          32    8.82723e-019    5.76e-014  2.22045e-016    3.06444e-005
16          34    5.52238e-020    7.21e-015  2.22045e-016    1.53222e-005

```

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the selected value of the function tolerance, and the problem appears regular as measured by the gradient.

```

root_fsolve =
    1.532962917156759e-005

```

Func-count	x	f(x)	Procedure
1	-0.236068	0.0557281	initial
2	0.236068	0.0557281	golden
3	0.527864	0.27864	golden
4	-2.77556e-017	7.70372e-034	parabolic
5	3.33333e-011	1.11111e-021	parabolic
6	-3.33334e-011	1.11111e-021	parabolic

Optimization terminated:

the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-010

```

root_fminbnd =
    -2.775557561562891e-017

```

Note that the call to `optimset` in the latter input utilized several options. It is left to the reader to examine the doc system for `optimset` for details on each option.

Note that if you issue the command `>> doc optimset`, you will be shown the help for the basic options structure. Within this description is a note with a link to the reference page for the enhanced `optimset` function in the Optimization Toolbox. Since the Optimization Toolbox comes with the Student Edition and it is installed on the lab computers, the enhanced option structure is the appropriate reference to consult for `optimset`.

### Professor Ashurst's Special Note #3:

The Levenberg-Marquardt method is a hybrid conjugate gradient/steepest descent approach with generally good convergence and is also generally fast. I generally recommend it. For further reading see e.g., *Numerical Recipes 3rd Edition: The Art of Scientific Computing* by Press, Teukolsky, Vetterling and Flannery, Cambridge University Press,

## 2.2 Solving a System of Linear Equations

Suppose that you are in a position where you need to solve a system of linear equations. This would be a typical case where there were  $n$  equations and  $n$  unknowns, and each of the unknowns appeared *linearly* in the equations. An example of such equation set is given as Eq. 2.3

$$\begin{aligned}
 2x + 3y + 13z &= 1.618 \\
 5x + 7y + 17z &= 2.718 \\
 21x - y + 5z &= 3.141
 \end{aligned}
 \tag{2.3}$$

MATLAB is very capable of performing matrix operations. In fact, the name of the software is an abbreviation of its former name, MATrix LABoratory. As such, the solution to a simple linear system is as straightforward as the session that follows.

```
>> A = [2, 3, 13; 5, 7, 17; 21, -1, 5];
>> B = [1.618, 2.718, 3.141]';
>> A\B
```

```
ans =
    0.134255707762557
    0.091651826484018
    0.082656392694064
```

This solution is exact, (to numerical precision) and is attained through the use of matrix operations. We will now use `fsolve` to achieve the solution. First, the system of equations must be re-cast into a system of *functions*. These functions must be expressed in terms of a vector whose elements will be zero when a solution is found. The function must have one argument (at least) that is a vector of length equal to the number of unknowns in the system. This single vector will be adjusted by `fsolve` until the function returns a zero vector. Additionally, `fsolve` requires an initial guess, which in this case is taken to be the vector  $[1, 1, 1]$ . The reader may verify that the solution produced by `fsolve` is not sensitive to this initial guess. The session that follows contains a reiteration of the exact analytic solution and the solution produced by `fsolve`. Note that the solutions differ by less than `eps`. The initial guesses may be changed to outlandish values where this is not the case.

```
>> clear all; clc;
>> A = [2, 3, 13; 5, 7, 17; 21, -1, 5];
>> B = [1.618, 2.718, 3.141]';
>> linsol_leftdivide = A\B

>> opts = optimset('Display','off', 'TolFun', 1e-10 , ...
    'TolX', 1e-10, 'MaxIter', 50000, 'MaxFunEvals', 200000, ...
    'Algorithm', 'Levenberg-Marquardt');
>> eqsys = @(U) [2*U(1) + 3*U(2) + 13*U(3) - 1.618;
    5*U(1) + 7*U(2) + 17*U(3) - 2.718;
    21*U(1) - U(2) + 5*U(3) - 3.141];
>> linsol_fsolve = fsolve(eqsys, [1,1,1], opts)' % note use of transpose
```

```
linsol_leftdivide =
    0.134255707762557
    0.091651826484018
    0.082656392694064
linsol_fsolve =
    0.134255707762557
    0.091651826484018
    0.082656392694064
```

## 2.3 Solving One Equation for One Complex Variable

It is of note that none of the methods described thus far are natively capable of handling complex roots, except for the `roots` function, which only is used for finding the roots of polynomials. However, we may utilize a bit of mathematics to decompose an equation into its real and complex parts and solve for the complex root piecewise. This practice is almost always done in computation, although programs generally hide the details of the operation from the user.

For this illustration, consider the function  $f(x) = 2x^2 + 1$  which clearly has complex conjugate roots of  $x = 0 \pm \left(\sqrt{\frac{1}{2}}\right) i$ . We will prepare a function file and pass its handle into `fsolve`. The form of the solution returned by `fsolve` will be a two element vector where the first element is the real part and the second element is the magnitude of the imaginary part. The function file will “assemble” a complex number from the guessed solution, carry out the complex computation, and split the result back into its real and imaginary magnitude. The function file is as follows.

```
function make_me_zero = the_eq(X)
xx = complex(X(1), X(2));
make_me_zero(1) = real(cplxfun(xx));
make_me_zero(2) = imag(cplxfun(xx));

    function result = cplxfun(x)
    result = 2*x.^2+1;
    end
end
```

Note the use of a nested function for clarity. We now call `fsolve` as shown in the session below.

```
>> cplx_roots(1,:) = fsolve(@the_eq, [1, +5], opts);
>> cplx_roots(2,:) = fsolve(@the_eq, [1, -5], opts);
>> cplx_roots
```

```
cplx_roots =
    0.0000000000000001    0.707106781186548
    0.0000000000000001   -0.707106781186548
```

Note that there are two calls to `fsolve` since the function is designed to process one complex number at a time. Also note the use of the `@` character to create the function handle. The initial guesses of  $1 \pm 5i$  are capricious.

## 2.4 Solving a System of Non-Linear Equations

Frequently, engineers will be in a situation where the solution of a system of nonlinear equations is required. Fortunately, `fsolve` is an excellent tool for this application. As an example, suppose that one has a series of three tanks, where each tank drains into the next by gravity assuming potential fluid flow behavior. The last tank drains to the surroundings. The first tank has a constant inlet flow rate. The applicable steady state model for this system is given below. (Note that for an unsteady state model, the zeros on the left hand side of the first three equations would be replaced by terms of the form  $A_i \frac{dh_i}{dt}$ , and initial conditions would be required.)

$$\begin{aligned}
 0 &= q_0 - q_1 \\
 0 &= q_1 - q_2 \\
 0 &= q_2 - q_3 \\
 q_1 &= A_o \sqrt{2gh_1} \\
 q_2 &= A_o \sqrt{2gh_2} \\
 q_3 &= A_o \sqrt{2gh_3} \\
 q_0 &= 0.002 \text{ (m}^3/\text{s)} \\
 A_o &= 0.001 \text{ (m}^2\text{)} \\
 g &= 9.81 \text{ (m/s}^2\text{)}
 \end{aligned} \tag{2.4}$$

It is clearly possible (and straightforward) to combine these nine equations in such a way that there were only three. However, such simplification is generally not needed when using MATLAB, and so it is advisable to type in more “short” equations rather than fewer “complicated” equations.

**Professor Placek’s Special Note #1:**

The matching (or mis-matching) of parentheses has caused many problems in MATLAB code. Parentheses control the order of operation when a statement is evaluated and a high degree of care must be exercised when typing in equations with many levels of parenthesis. I recommend typing in first the *structure* of the equation and *then* filling in the terms with variables.

The general setup is that constants are declared and assigned first, followed by constitutive equations and finally the balance equations. **It must be kept in mind that the statements within a function are evaluated sequentially.** The function file is as follows.

```
function make_me_zero = tanksys( HH )
h1 = HH(1);  h2 = HH(2);  h3 = HH(3);
q_0 = 0.002;  A_o = 0.001;  g = 9.81;

q_1 = A_o*sqrt(2*g*h1);
q_2 = A_o*sqrt(2*g*h2);
q_3 = A_o*sqrt(2*g*h3);

make_me_zero = [q_0 - q_1;
                q_1 - q_2;
                q_2 - q_3];
end
```

Note that the return is a column vector of length 3. A row vector could also have been used because `fsolve` uses a process called “linear indexing” to order the returned values. The session to invoke several solutions is as follows.

```
>> opts = optimset('Display','iter');
>> ss_hs1 = fsolve(@tanksys, [1,1,1], opts);
>> ss_hs1

>> opts = optimset('Display','iter', 'TolFun', 1e-10 , ...
```

```

    'TolX', 1e-10, 'MaxIter', 50000, 'MaxFunEvals', 200000, ...
    'Algorithm', 'Levenberg-Marquardt');
>> ss_hs2 = fsolve(@tanksys, [1,1,1], opts);
>> ss_hs2

>> opts = optimset('Display','iter','Algorithm','Levenberg-Marquardt');
>> ss_hs3 = fsolve(@tanksys, [1,1,1], opts);
>> ss_hs3

```

Iteration	Func-count	f(x)	Norm of step	First-order optimality	Trust-region radius
0	4	5.90221e-006		5.38e-006	1
1	8	1.39916e-006	1	4.96e-006	1
2	12	2.18563e-007	0.611479	3e-006	2.5
3	16	3.15269e-009	0.103757	2.83e-007	2.5

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

```

ss_hs1 =
    0.203873111827341    0.192586544502948    0.192586544502948

```

Iteration	Func-count	Residual	First-Order optimality	Lambda	Norm of step
0	4	5.90221e-006	5.38e-006	0.01	
1	8	5.89643e-006	5.38e-006	0.001	0.000537528
2	12	5.83937e-006	5.34e-006	0.0001	0.00532468
3	16	5.33555e-006	4.99e-006	1e-005	0.0487638
4	20	3.07138e-006	2.89e-006	1e-006	0.282167
5	24	3.82345e-007	8.65e-007	1e-007	0.815761
6	28	2.27791e-008	6.28e-007	1e-008	0.388891
7	32	1.0154e-010	4.73e-008	1e-009	0.0457323
8	36	1.53896e-015	1.92e-010	1e-010	0.00268356
9	40	4.86124e-025	3.13e-015	1e-011	8.81639e-006
10	44	1.12847e-036	2.13e-021	1e-012	2.2975e-010

Equation solved.

fsolve completed because the vector of function values is near zero as measured by the selected value of the function tolerance, and the problem appears regular as measured by the gradient.

```

ss_hs2 =
    0.203873598369011    0.203873598369011    0.203873598369011

```

Iteration	Func-count	Residual	First-Order optimality	Lambda	Norm of step
0	4	5.90221e-006	5.38e-006	0.01	
1	8	5.89643e-006	5.38e-006	0.001	0.000537528
2	12	5.83937e-006	5.34e-006	0.0001	0.00532468

3	16	5.33555e-006	4.99e-006	1e-005	0.0487638
4	20	3.07138e-006	2.89e-006	1e-006	0.282167
5	24	3.82345e-007	8.65e-007	1e-007	0.815761
6	28	2.27791e-008	6.28e-007	1e-008	0.388891
7	32	1.0154e-010	4.73e-008	1e-009	0.0457323
8	36	1.53896e-015	1.92e-010	1e-010	0.00268356
9	40	4.86124e-025	3.13e-015	1e-011	8.81639e-006
10	44	1.12847e-036	2.13e-021	1e-012	2.2975e-010

Equation solved.

`fsolve` completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

```
ss_hs3 =
    0.203873598369011    0.203873598369011    0.203873598369011
```

The first solution, `ss_hs1` is the result obtained with the default options (except the display). It should be immediately apparent that there is something odd about the solution. Upon examination of the form of the model, it should be clear that for steady state to occur, the levels must all be the same. However, `fsolve` returned a level for tank 1 that is slightly greater than that for tanks 2 and 3 (which seem identical). Therefore, to investigate the effect of the convergence criteria, another call to `fsolve` is made with more stringent convergence criteria. The solution that results from that process, `ss_hs2` exhibits the requisite equality. As an aside, a third call to `fsolve` is shown where only the Levenberg-Marquardt algorithm is used. It is interesting to note that the solution is identical, to the second call, even though the options of “TolX” and “TolFun” were not set, and therefore the default values were used.

As a follow-on example, assume now that tanks 2 and 3 are also each supplied with an input stream (called  $q_{0,2}$  and  $q_{0,3}$ , respectively) whose magnitude is inversely proportional to the level in the tank previous to it. The model represented by Eq. (2.4) is modified accordingly to form a new model as shown below in Eq. (2.5).

$$\begin{aligned}
 0 &= q_0 - q_1 \\
 0 &= q_{0,2} + q_1 - q_2 \\
 0 &= q_{0,3} + q_2 - q_3 \\
 q_1 &= A_o \sqrt{2gh_1} \\
 q_2 &= A_o \sqrt{2gh_2} \\
 q_3 &= A_o \sqrt{2gh_3} \\
 q_{0,2} &= \frac{0.001}{h_1} \\
 q_{0,3} &= \frac{0.005}{h_2} \\
 q_0 &= 0.002 \text{ (m}^3/\text{s)} \\
 A_o &= 0.001 \text{ (m}^2\text{)} \\
 g &= 9.81 \text{ (m/s}^2\text{)}
 \end{aligned} \tag{2.5}$$

Modifying the function `tanksys` and re-issuing the `fsolve` command illustrates the effect of the additional input streams.

```

function make_me_zero = tanksys( HH )
h1 = HH(1);  h2 = HH(2);  h3 = HH(3);
q_0 = 0.002;  A_o = 0.001;  g = 9.81;

q_1 = A_o*sqrt(2*g*h1);
q_2 = A_o*sqrt(2*g*h2);
q_3 = A_o*sqrt(2*g*h3);
q_02 = 0.001/h1;
q_03 = 0.005/h2;

make_me_zero = [q_0 - q_1;
                q_02 + q_1 - q_2;
                q_03 + q_2 - q_3];
end

>> opts = optimset('Display','off','Algorithm','Levenberg-Marquardt');
>> ss_hs_mod = fsolve(@tanksys, [1,1,1], opts);
>> ss_hs_mod

ss_hs_mod =
    0.203873598395347    2.430123597986995    4.094116152161702

```

## 2.5 Passing Additional Function Arguments Though fsolve

It can often be advantageous to parameterize functions in order to make them more general. For example, consider the function  $f(x) = \sin(ax)$  where  $a$  may be any number such as 3 or 7.7. The session to plot the function  $f$  with  $a = 3$  making use of an anonymous function is:

```

>> f = @(x, a) sin(a*x);  % handle to f (anonymous)
>> x = linspace(0, 2*pi);
>> plot(x, f(x, 3))      % passes 3 into f as the value of a

```

As you can see from the plot, there are a number of roots for this function. One root is near the value  $x = 1$ . One might attempt a MATLAB session (with output) involving `fsolve` to locate a precise value for this root such as:

```

>> fsolve(f, 1, []) % note empty bracket placeholder for options

```

```

??? Input argument "a" is undefined.

```

```

Error in ==> @(x,a)sin(a*x)

```

```

Error in ==> fsolve at 254
    fuser = feval(funfcn{3},x,varargin{:});

```

```

Caused by:

```

```

    Failure in initial user-supplied objective function evaluation. FSOLVE cannot continue.

```



One could go through and call `fsolve` as was done in section 2.1, and change  $T$  and  $P$  to a value on the table, do the solving, and store away the result, representing 96 different solutions. However, this would be time consuming, prone to error and needless. A better alternative is to parameterize the van der Waals function and create vectors for  $T$ ,  $P$  values to pass in as additional arguments from within for loops. The session to accomplish this is shown below.

```
>> format short; clc; clear all;
>> a = 1.372E6; % bar*cm^6/(mol^2)
>> b = 37.24; % cm^3/(mol)
>> R = 83.14472; % bar*cm^3/(K*mol)
>> P = [1, 5, 10, 20:20:100, 200:100:500]; % bar
>> T = [180:10:250]; % K
>> opts = optimset('Display','off', 'TolFun', 1e-10, ...
    'TolX', 1e-10, 'MaxIter', 50000, 'MaxFunEvals', 200000, ...
    'Algorithm', 'Levenberg-Marquardt');
>> vdW_eos = @(V, TT, PP) R*TT/(V-b) - a/V^2 - PP;
>> Z = zeros(length(T), length(P));
>> V = Z;
>> for j = 1:length(T)
    for k = 1:length(P)
        V(j,k) = fsolve(vdW_eos, 2000, [], T(j), P(k));
        Z(j,k) = (P(k)*V(j,k))/(R*T(j));
    end
end
>> Z
```

```
Z =
0.9964 0.9816 0.9629 0.9244 0.8433 0.7612 0.6966 0.6764 0.8907 1.1758 1.4608 1.7418
0.9969 0.9842 0.9682 0.9355 0.8686 0.8030 0.7494 0.7234 0.8905 1.1539 1.4220 1.6875
0.9973 0.9863 0.9726 0.9447 0.8888 0.8353 0.7910 0.7654 0.8940 1.1364 1.3886 1.6399
0.9976 0.9881 0.9762 0.9524 0.9052 0.8610 0.8243 0.8011 0.9005 1.1227 1.3599 1.5980
0.9979 0.9897 0.9794 0.9588 0.9187 0.8818 0.8512 0.8310 0.9091 1.1121 1.3352 1.5610
0.9982 0.9910 0.9820 0.9643 0.9301 0.8990 0.8735 0.8562 0.9190 1.1042 1.3139 1.5283
0.9984 0.9921 0.9843 0.9689 0.9396 0.9134 0.8920 0.8775 0.9296 1.0985 1.2956 1.4992
0.9986 0.9931 0.9863 0.9730 0.9478 0.9255 0.9076 0.8955 0.9403 1.0945 1.2799 1.4734
```

The anonymous function handle, `vdW_eos` is formed to require three arguments,  $V$ ,  $TT$  and  $PP$ . `fsolve` will provide the first, as required by `fsolve`, since we seek a root on  $V$ . We must then pass values for  $T$  and  $P$ , one element at a time, into the function through `fsolve`. Consequently, we must instruct `fsolve` to pass these quantities through by adding them to the calling line after the options structure. In the alternative that the default options were sufficient, we would call `fsolve` as in:

```
V(j,k) = fsolve(vdW_eos, 2000, [], T(j), P(k));
```

The elements of the  $Z$  matrix are populated one at a time, and both  $Z$  and  $V$  matrices were pre-allocated.

## 2.6 Chapter Summary

- The preferred approach to solving algebraic equations involves the use of `fsolve`

- By default, the convergence criteria are “loose” and should generally be “tightened” via tailoring the options structure with the use of the command `optimset`
- For most applications, the following options set should suffice, and this (or similar) should be included in your MATLAB initialization script

```
opts = optimset('Display','off', 'TolFun', 1e-10 , ...  
              'TolX', 1e-10, 'MaxIter', 500, 'MaxFunEvals', 2000, ...  
              'Algorithm', 'Levenberg-Marquardt')
```

## Chapter 3

# Solving Ordinary Differential Equations

3.1 Solving One First Order ODE

3.2 Solving a System of First Order ODEs

3.3 Solving One Second Order ODE

3.4 Solving a System of ODEs with Parameters Passed Through  
ode45

## Chapter 4

# Fitting Data to Nonlinear Model Functions

- 4.1 Model Functions of One Independent Variable
- 4.2 Model Functions of More than One Independent Variable
- 4.3 Characterizing “Goodness of Fit”

## Chapter 5

# Probability and Statistics

5.1 Definitions and Basic Concepts

5.2 Characteristics of Probability Distributions

## Chapter 6

# Hypothesis Testing

6.1 One Sample Tests

6.2 Two Sample Tests

## Chapter 7

# Miscellaneous Notes

7.1 Functions and Function Handles

7.2 Elements of Style

# Index

## D

doc, 4

## F

feval, 16

fminbnd, 3, 4, 6, 7

for, 16

fsolve, 1, 3, 4, 6, 7, 10–17

fzero, 2, 4, 6, 7

## L

linear indexing, 13

## N

nlinfit, 1

## O

ode45, 1

optimset, 6, 7, 9, 17

## R

roots, iii, 3, 4, 6, 7, 11

## V

varargin, 16