Numerical Methods in MATLAB

Center for Interdisciplinary Research and Consulting
Department of Mathematics and Statistics
University of Maryland, Baltimore County

www.umbc.edu/circ

Winter 2012
Mission and Goals: The Center for Interdisciplinary Research and Consulting (CIRC) is a consulting service on mathematics and statistics provided by the Department of Mathematics and Statistics at UMBC. Established in 2003, CIRC is dedicated to support interdisciplinary research for the UMBC campus community and the public at large. We provide a full range of consulting services from free initial consulting to long term support for research programs.

CIRC offers mathematical and statistical expertise in broad areas of applications, including biological sciences, engineering, and the social sciences. On the mathematics side, particular strengths include techniques of parallel computing and assistance with software packages such as MATLAB and COMSOL Multiphysics (formerly known as FEMLAB). On the statistics side, areas of particular strength include Toxicology, Industrial Hygiene, Bioequivalence, Biomechanical Engineering, Environmental Science, Finance, Information Theory, and packages such as SAS, SPSS, and S-Plus.

Copyright © 2003–2012 by the Center for Interdisciplinary Research and Consulting, Department of Mathematics and Statistics, University of Maryland, Baltimore County. All Rights Reserved.

This tutorial is provided as a service of CIRC to the community for personal uses only. Any use beyond this is only acceptable with prior permission from CIRC.

This document is under constant development and will periodically be updated. Standard disclaimers apply.

Acknowledgements: We gratefully acknowledge the efforts of the CIRC research assistants and students in Math/Stat 750 Introduction to Interdisciplinary Consulting in developing this tutorial series.

1 Introduction

In this tutorial, we will introduce some of the numerical methods available in Matlab. Our goal is to provide some snap-shots of the wide variety of computational tools that Matlab provides. We will look at some optimization routines, where we mainly focus on unconstrained optimization. Next, we discuss curve fitting and approximation of functions using Matlab. Our final topic will be numerical ODEs in Matlab.

Matlab provides a number of specialized toolboxes, which extend the capabilities of the software. We will have a brief overview of the various toolboxes in Matlab and will provide a list of some available toolboxes.

- Numerical Optimization
- Data Fitting / Approximation
- Numerical ODEs
- Matlab Toolboxes

2 Unconstrained Optimization

The commands we discuss in this section are two of the several optimization routines available in Matlab. First we discuss fminbnd, which is used to minimize functions of one variable. The command,

\[
[x \ fval] = \text{fminbnd}(f, \ a, \ b)
\]

finds a local minimizer of the function \( f \) in the interval \([a, b]\). Here \( x \) is the local minimizer found by the command and \( fval \) is the value of the function \( f \) at that point. For complete discussion of fminbnd readers can refer to the Matlab’s documentations. Here we illustrate the use of fminbnd in an example.

Consider the function \( f(x) = \cos(x) - 2\ln(x) \) on the interval, \([\frac{\pi}{2}, 4\pi]\). The plot of the function is depicted in Figure 1. We can first define the function \( f(x) \) in Matlab using

\[
f = @(x)(\cos(x) - 2*\log(x))
\]

Then we proceed by the following,

\[
>> \ [x \ fval] = \text{fminbnd}(f, 2, 4)
\]

\[
x = 3.7108
\]
UNCONSTRAINED OPTIMIZATION

Figure 1: Plot of $f(x) = \cos(x) - 2\ln(x)$.

\[
\begin{align*}
\text{fval} &= -3.4648 \\
\text{We see that fminbnd returns the (approximate) minimizer of } f(x) \text{ in the interval } [2, 4]
\end{align*}
\]

The next (unconstrained) optimization command we discuss is \texttt{fminsearch} which can be used to find a local minimizer of a function of several variables. The command,

\[
[x \ fval] = \text{fminsearch}(f, x0)
\]

finds a local minimizer of the function $f$ given the initial guess $x0$. For complete discussion of \texttt{fminsearch} readers can refer to the Matlab’s documentations. Here we illustrate the use of \texttt{fminsearch} in an example.

For a simple example, we minimize the function $f(x, y) = x^2 + y^2$ which clearly has its minimizer at $(0, 0)$. Let’s choose an initial guess of $x_0 = (0, 0)$. The following Matlab commands illustrate the usage of \texttt{fminsearch}.

\[
\begin{align*}
>> f &= @(x)(x(1)^2+x(2)^2); \\
>> x0 &= [1;1]; \\
>> [x \ fval] &= \text{fminsearch}(f, x0)
\end{align*}
\]

\[
\begin{align*}
x &= 1.0e-04 * \\
-0.2102
\end{align*}
\]
Of course, reader can try \texttt{fminsearch} on more complicated problems and see the results. For more information on optimization routines in Matlab, reader can investigate Matlab’s Optimization Toolbox which includes several powerful optimization tools which can be used to solve both unconstrained and constrained linear and non-linear optimization problems.

3 Curve-Fitting

Here we consider the problem of fitting a polynomial of degree (at most) $k$ into the data points $(x_i, y_i)$; to do this, we use the command,

\[
p = \text{polyfit}(x, y, k)
\]

which fits a polynomial of degree $k$ into the given data points. The output argument $p$ is a vector which contains the coefficients of the interpolating polynomial. The method used is least squares, in which we choose a polynomial $P$ of degree $k$ which minimizes the following:

\[
\sum_{i=1}^{m} [P(x_i) - y_i]^2.
\]  

(3.1)

For example, say we are given the data points

\begin{tabular}{ll}
  xi & yi \\
  1 & 1 \\
  2 & 0.5 \\
  3 & 1 \\
  4 & 2.5 \\
  5 & 3 \\
  6 & 4 \\
  7 & 5 \\
\end{tabular}

Suppose we would like to fit a polynomial of degree three into the given data points. We can use the following Matlab commands to get the interpolating polynomial.

\begin{verbatim}
>> xi = 1 : 7;
>> yi = [1 0.5 1 2.5 3 4 5];
>> p = polyfit(xi,yi,3);
\end{verbatim}
Once we have the vector $p$ which contains the coefficients of the interpolating polynomial we can use the Matlab function `polyval` to evaluate the approximating polynomial over a given range of $x$ values. We can proceed as follows:

$$
\begin{align*}
&\gg x = 1 : 0.1 : 7; \\
&\gg y = \text{polyval}(p, x); \\
&\gg \text{plot}(x, y, '*', x, y)
\end{align*}
$$

Which produces the Figure 2.

![Figure 2: Data fitting in Matlab](image)

Another useful idea is using `polyfit` to find an approximating polynomial for a given function. The idea is useful because polynomials are much simpler to work with. For example one can easily integrate or differential polynomials while it may not be so easy to do the same for a function which is not so well behaved. As an example, we consider the problem of approximating the function $\sin(\sqrt{x})$ on the interval $[0, 2\pi]$.

The following Matlab commands show how one selects a numerical grid to get data points which can be used to approximate the function using a polynomial of degree $k$ (in a least-squares sense).

$$
\begin{align*}
&\gg f = @(x)(\sin(\sqrt{x})); \\
&\gg xi = [0 : 0.1 : 2*\pi]; \\
&\gg yi = f(xi); \\
&\gg p = \text{polyfit}(xi, yi, 4);
\end{align*}
$$

To see how the function and its approximation compare, we can use the following commands,

$$
\begin{align*}
&\gg x = [0 : 0.01 : 2*\pi]; \\
&\gg y = \text{polyval}(p, x); \\
&\gg \text{plot}(x, y, x, f(x), '--');
\end{align*}
$$

The result can be seen in Figure 3.
4 Numerical ODEs

In this section we discuss numerical ordinary differential equations in Matlab. Matlab provides a number of ODE solvers; we will focus our attention to \texttt{ode45} which uses a four stage Runge-kutta method to solve a given ordinary differential equation. We will first see how one can solve a problem of the form

\[
\frac{dy}{dt} = f(t, y),
\]

\[y(t_0) = y_0.\]

We can solve such problems using

\[
[T \ Y] = \texttt{ode45}(f, \ tspan, \ y0).
\]

In the above syntax, the input argument \(f\) specifies the right hand side function of the differential equations, \(tspan\) is the time interval in which we want to solve the equation, and \(y0\) is the initial value. The output argument \(Y\) gives the numerical solution over the discretized time interval \(T\). Consider the following problem,

\[
\frac{dy}{dt} = t - y,
\]

\[y(0) = 1.\]

One can solve this problem analytically to get the solution, \(y(t) = 2e^{-t} + t - 1.\) To solve the problem numerically, we can use

\[
>> f = @(t, y) (t - y)
\]

\[
f =
\]

\[
@ (t, y) (t - y)
\]

\[
>> y0=1;
\]

\[
>> [T Y] = \texttt{ode45}(f, [0 \ 2], y0);
\]
The reader can see how good the numerical solution is by plotting both the numerical solution and the true solution in the same plot; better yet one can compute and plot the approximation error to get a better picture of how good the solution was. The following Matlab code can used to solve the above problem; it also plots the numerical solution, the true solution, and the approximation error (see Figure 4).

```matlab
function testode
y0 = 1;
[T Y] = ode45(@frhs, [0 2], y0);
y = ftrue(T);

subplot(311);
plot(T, Y);
title('Approximate Solution');

subplot(312);
plot(T, y);
title('True Solution');

subplot(313);
plot(T, abs(Y - y));
title('Approximation Error');

% subfunctions
function f=frhs(t, y);
f = t - y;

function f=ftrue(t);
f = 2*exp(-t)+t-1;
```

We can also solve systems of ODEs using `ode45`. To illustrate the idea we solve a classical predator-prey system. Let’s consider the interaction of foxes and rabbits, where foxes are predators and rabbits are the prey. Denote, the rabbit population at any time by \( y_1(t) \) and the fox population by \( y_2(t) \). The system of differential equations modeling the dynamics of this predator-prey system is given by the following

\[
\frac{dy_1}{dt} = gy_1 - d_1y_1y_2
\]
\[
\frac{dy_2}{dt} = -d_2y_2 + cy_1y_2
\]

In above equations the parameters \( g, d_1, d_2, \) and \( c \) denote:

1. \( g = \) natural growth rate of rabbit population in absence of foxes.

2. \( d_1 = \) the rate at which foxes die in the absence of rabbits.
3. \( d_2 \) = the death rate per each (deadly) encounter of rabbits due to foxes.

4. \( c \) = the contribution to fox population of each (food making) encounter of rabbits and foxes to fox population.

In addition to specifying the model parameters, we also need to specify the initial population of foxes and rabbits at \( t = 0 \). Let’s choose the model parameters as below:

- \( g = 0.04 \);
- \( d_1 = 0.001 \);
- \( d_2 = 0.1 \);
- \( c = 0.002 \);

Also, we assume the initial populations start at \( y_1(0) = 100 \) and \( y_2(0) = 100 \). To solve the resulting initial value problem, we can use \texttt{ode45}; the Matlab function \texttt{predatorprey} provided below solves the problem using \texttt{ode45} and plots the populations of foxes and rabbits on the same plot (Figure 5); moreover, the function creates a phase-plane diagram (Figure 5) which is a useful tool in analyzing such systems.
function predatorprey

[T,Y] = ode45(@yprime,[0 100],[100 100]);
subplot(2,1,1);
plot(T,Y(:,1),'-', T,Y(:,2), '--');
title('Population Dynamics of Foxes and Rabbits');
legend('Rabbit Population', 'Fox Population');
xlabel('t');
ylabel('Population');
grid on;

subplot(2,1,2);
plot(Y(:,1), Y(:,2));
title('Phase Plane Diagram for the fox-rabbit population');
xlabel('Rabbits');
ylabel('Foxes');
grid on;

%rhs function
function dy = yprime(t, y)
g = 0.4;
d1 = 0.001;
d2 = 0.1;
c = 0.002;
dy = [g*y(1)-d1*y(1)*y(2);
    c*y(1)*y(2) - d2*y(2)];
Figure 5: Dynamics of a predator prey (fox/rabbit) system

The reader can further experiment with the above Matlab code to see the outcome with different parameters and different initial populations.
Our discussion of Matlab’s ODE solvers here focused on the example of the function \texttt{ode45}, which is Matlab’s most popular ODE solver. Matlab has a suite of solvers, see \texttt{doc ode45} for full documentation and recommendations for when to use which method in table form. We complement this table here by discussing the methods and providing some additional information. See this documentation also for the list of options used to control the method, such as relative and absolute tolerances on the error estimator, as well as for a list of references on the subject of ODE solvers and the methods in particular.

All ODE solvers in Matlab use the same function interface, so it is easy to try several methods on the same problem and observe their behavior. Also, all methods compute an estimator for the error of the solution that is used to automatically select the size of the time step and also of the method order, if it is variable. ODE problems are roughly classified into \textit{stiff} and \textit{nonstiff} problems. General-purpose ODE solvers in Matlab that are appropriate for stiff problems are indicated by the letter “s” at the end of their names, namely \texttt{ode15s} and \texttt{ode23s}. The most important nonstiff solvers are \texttt{ode45} and \texttt{ode113}. The numbers in the names of the two methods \texttt{ode15s} and \texttt{ode113} that are variable-order methods indicate the method order ranging from 1 to 5 and from 1 to 13, respectively. All other methods are fixed-order methods with the first number indicating the order of the method, such as 4 in \texttt{ode45} and 2 in \texttt{ode23s}; the second number indicates the order of the second method used simultaneously in the error estimator.

The technical definition of the term \textit{stiff} is difficult, but their practical definition is readily stated: A problem is stiff, if the automatic step size control of the method chooses small steps even for large tolerances. This means that the step sizes are limited by the stability of the method and not the accuracy requested by the user. Note that all ODE solvers in Matlab use automatic step size control based on a sophisticated error estimator, hence the accuracy of their solution is ensured; but if it takes a longer time to compute the solution with nonstiff solvers such as \texttt{ode45} or \texttt{ode113} than with a stiff solver such as \texttt{ode15s}, the problem is considered stiff. In summary, for a particular problem, try \texttt{ode45} or \texttt{ode113} as potential nonstiff solvers and try also \texttt{ode15s} as potential stiff solvers. Then continue using whichever method performed most efficiently.
5 Matlab Toolboxes

In this section, we will discuss Matlab Toolboxes. In general, Matlab toolboxes extend the capabilities of Matlab by providing highly efficient routines which are specialized to handle specific situations. For example, if one is solving some problems in the area of neural networks, the Neural Network Toolbox provides powerful tools to handle problems of that type. As another example, Matlab’s Statistics Toolbox provides a wide range of statistical routines.

A good way to learn about a Matlab Toolbox is studying the associated Getting Started Guide; another good place to start is the user’s guide for the associated Toolbox. For example, in Figure 6, we have displayed the help screen for Matlab’s Statistics Toolbox (from Matlab’s Help facility). We can see the various documentations provided for a toolbox.

![Figure 6: Matlab’s Statistics Toolbox](image)

To give the readers an idea of the available Matlab toolboxes, a list of widely used Matlab Toolboxes is provided in Table 1.

To find out which Toolboxes are available in your version of Matlab you can type `ver` in Matlab’s command prompt. Issuing the `ver` command will provide something like the following:
<table>
<thead>
<tr>
<th>Math and Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization Toolbox</td>
<td></td>
</tr>
<tr>
<td>Symbolic Math Toolbox</td>
<td></td>
</tr>
<tr>
<td>Extended Symbolic Math Toolbox</td>
<td></td>
</tr>
<tr>
<td>Partial Differential Equation Toolbox</td>
<td></td>
</tr>
<tr>
<td>Genetic Algorithm and Direct Search Toolbox</td>
<td></td>
</tr>
<tr>
<td>Statistics and Data Analysis</td>
<td></td>
</tr>
<tr>
<td>Statistics Toolbox</td>
<td></td>
</tr>
<tr>
<td>Neural Network Toolbox</td>
<td></td>
</tr>
<tr>
<td>Curve Fitting Toolbox</td>
<td></td>
</tr>
<tr>
<td>Spline Toolbox</td>
<td></td>
</tr>
<tr>
<td>Model-Based Calibration Toolbox</td>
<td></td>
</tr>
<tr>
<td>Control System Design and Analysis</td>
<td></td>
</tr>
<tr>
<td>Control System Toolbox</td>
<td></td>
</tr>
<tr>
<td>System Identification Toolbox</td>
<td></td>
</tr>
<tr>
<td>Fuzzy Logic Toolbox</td>
<td></td>
</tr>
<tr>
<td>Robust Control Toolbox</td>
<td></td>
</tr>
<tr>
<td>Model Predictive Control Toolbox</td>
<td></td>
</tr>
<tr>
<td>Aerospace Toolbox</td>
<td></td>
</tr>
<tr>
<td>Signal Processing and Communications</td>
<td></td>
</tr>
<tr>
<td>Signal Processing Toolbox</td>
<td></td>
</tr>
<tr>
<td>Communications Toolbox</td>
<td></td>
</tr>
<tr>
<td>Filter Design Toolbox</td>
<td></td>
</tr>
<tr>
<td>Filter Design HDL Coder Wavelet Toolbox</td>
<td></td>
</tr>
<tr>
<td>Fixed-Point Toolbox RF Toolbox</td>
<td></td>
</tr>
<tr>
<td>Image Processing</td>
<td></td>
</tr>
<tr>
<td>Image Processing Toolbox</td>
<td></td>
</tr>
<tr>
<td>Image Acquisition Toolbox</td>
<td></td>
</tr>
<tr>
<td>Mapping Toolbox</td>
<td></td>
</tr>
<tr>
<td>Test and Measurement</td>
<td></td>
</tr>
<tr>
<td>Data Acquisition Toolbox</td>
<td></td>
</tr>
<tr>
<td>Instrument Control Toolbox</td>
<td></td>
</tr>
<tr>
<td>Image Acquisition Toolbox</td>
<td></td>
</tr>
<tr>
<td>SystemTest OPC Toolbox</td>
<td></td>
</tr>
<tr>
<td>Computational Biology</td>
<td></td>
</tr>
<tr>
<td>Bioinformatics Toolbox</td>
<td></td>
</tr>
<tr>
<td>Financial Modeling and Analysis</td>
<td></td>
</tr>
<tr>
<td>Financial Toolbox</td>
<td></td>
</tr>
<tr>
<td>Financial Derivatives Toolbox</td>
<td></td>
</tr>
<tr>
<td>GARCH Toolbox</td>
<td></td>
</tr>
<tr>
<td>Datafeed Toolbox</td>
<td></td>
</tr>
<tr>
<td>Fixed-Income Toolbox</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: A list of Matlab Toolboxes
Of course, the results may vary depending on the system on which the command `ver` is issued.